

目 录

第一篇 基础知识



第1章 嵌入式系统与嵌入式的 Linux	(3)
1.1 无所不在的嵌入式系统	(3)
1.1.1 身边的嵌入式系统	(3)
1.1.2 嵌入式系统的特点	(4)
1.1.3 RTOS 简介	(6)
1.1.4 RTOS 在中国	(7)
1.2 自由的企鹅——Linux	(8)
1.3 本章小结	(13)
第2章 Linux 概论	(14)
2.1 走进自由天地——初识 Linux	(14)
2.1.1 Linux 的成长	(14)
2.1.2 Linux 与 GNU	(16)
2.2 Linux 常用的版本	(17)
2.3 Linux 操作系统基本构成	(18)
2.3.1 系统概述	(18)
2.3.2 Linux 内核	(20)
2.3.3 系统数据结构	(23)
2.3.4 子系统的结构	(23)
2.4 Linux 的基本指令	(28)
2.4.1 Shell	(28)
2.4.2 Linux 命令的使用说明	(28)
2.5 五脏俱全的嵌入式 Linux	(39)
2.5.1 嵌入式 Linux 的其他版本	(39)
2.5.2 RT-Linux	(40)
2.5.3 uClinux	(42)
2.6 本章小结	(42)
第3章 Linux 下的 C 语言编程入门	(43)
3.1 C 语言和 Linux	(43)
3.1.1 C 语言的发展历史	(43)
3.1.2 C 语言的特点	(44)
3.1.3 C 语言和 Linux	(44)
3.1.4 C 语言和嵌入式系统的设计	(44)
3.2 GCC 编译器的使用	(45)
3.2.1 GNU C 编译器	(46)

3.2.2 使用 gdb	(48)
3.3 使用 make	(54)
3.3.1 makefile	(55)
3.3.2 make 命令	(57)
3.3.3 makefile 变量	(59)
3.3.4 在 makefile 中使用函数	(60)
3.4 实例分析	(61)
3.5 本章小结	(62)

第二篇 开发入门

第4章 嵌入式 Linux 的开发平台	(65)
4.1 华恒嵌入式 Linux 开发套件简介	(65)
4.2 软件系统配置	(67)
4.3 uClinux 操作系统	(71)
4.3.1 uClinux 简介	(71)
4.3.2 uClinux 的小型化	(73)
4.3.3 uClinux 的开发环境	(74)
4.3.4 uClinux 针对实时性的解决方案	(75)
4.3.5 uClinux 的内存管理	(76)
4.3.6 uClinux 系统对进程和线程的管理	(78)
4.4 uClinux 开发环境的建立	(81)
4.4.1 通过源代码建立开发环境	(81)
4.4.2 从所购买的正式发行的 CD-ROM 安装	(82)
4.4.3 使用 minicom	(85)
4.5 uCsimmm	(91)
4.5.1 uCsimmm 简介	(91)
4.5.2 加入 uCsimmm 的邮件列表	(93)
4.6 系统的核心——CPU	(93)
4.6.1 CPU 主要特性	(93)
4.6.2 CPU 各个部分的功能概述	(96)
4.7 其他的外围设备和接口	(99)
4.8 本章小结	(101)
第5章 嵌入式 Linux 的开发	(102)
5.1 如何构造一个嵌入式 Linux 系统	(102)
5.1.1 嵌入式 Linux 系统的概述	(102)
5.1.2 关于嵌入式 Linux 开发的一些问题和概念	(111)
5.1.3 构造一个嵌入式 Linux 的实例	(111)
5.2 嵌入式 Linux 的应用程序的编译和调试	(113)
5.2.1 嵌入式 Linux 的应用程序	(114)

5.2.2 gcc 在嵌入式 Linux 系统中的使用	(115)
5.2.3 GNU 的链接工具——ld	(119)
5.2.4 嵌入式 Linux 程序的调试——使用 gdb	(124)
5.3 应用软件的开发	(130)
5.3.1 建立开发环境	(131)
5.3.2 熟悉开发环境	(134)
5.3.3 在开发板上编写应用程序	(139)
5.4 本章小结	(153)

第三篇 应用与提高

第 6 章 嵌入式 Linux 网络功能的实现	(157)
6.1 接入互联网的嵌入式系统	(157)
6.1.1 嵌入式因特网技术的兴起与前景	(157)
6.1.2 嵌入式 Internet 的应用	(159)
6.1.3 嵌入式 Internet 的原理	(160)
6.2 使用 Linux 来构建嵌入式网络设备	(163)
6.2.1 低成本的嵌入式网络电器设备	(163)
6.2.2 使用 Linux 将 8/16 位的嵌入式设备接入互联网	(167)
6.3 Linux 下的网络编程	(171)
6.3.1 TCP/IP 协议概述	(171)
6.3.2 Linux 环境下的 socket 编程	(176)
6.3.3 应用实例:网口通信	(186)
6.4 连接上 Web	(190)
6.4.1 HTTP 协议	(190)
6.4.2 一个简单的 Web 服务器的样例	(196)
6.5 本章小结	(210)
第 7 章 嵌入式 Linux 下的串行通信	(211)
7.1 串行口的物理标准	(211)
7.1.1 关于总线	(211)
7.1.2 RS-232 串行口	(214)
7.2 Linux 下的串行通信编程	(215)
7.2.1 串行通信的基础	(215)
7.2.2 串行口的设置	(220)
7.2.3 MODEM 的通信	(226)
7.2.4 串行编程进阶	(228)
7.3 串行通信的实例	(231)
7.4 本章小结	(239)
第 8 章 嵌入式 Linux 系统的键盘和 LCD	(241)
8.1 嵌入式系统所用到的键盘和 LCD	(241)

8.2 为嵌入式系统接上小键盘实例	(242)
8.3 LCD 的显示和控制	(251)
8.3.1 LCD 的控制与 uClinux 对 LCD 的支持	(251)
8.3.2 应用程序的编制	(258)
8.4 本章小结	(282)

第四篇 专题讨论

第 9 章 嵌入式实时操作系统与实时 Linux	(287)
9.1 嵌入式实时操作系统简介	(287)
9.1.1 RTOS 的要求	(287)
9.1.2 各种流行的实时操作系统	(288)
9.1.3 实时系统的设计	(291)
9.2 实时 Linux——RT-Linux	(297)
9.2.1 RT-Linux 综述	(297)
9.2.2 RT-Linux 的实时内核	(303)
9.2.3 RT-Linux 的实现机理	(304)
9.3 RT-Linux 下的编程	(306)
9.3.1 RT-Linux 的 API	(306)
9.3.2 RT-Linux 的编程方法示例	(307)
9.3.3 程序原理	(308)
9.3.4 程序实现	(308)
9.3.5 例 9-5 执行结果	(314)
9.4 嵌入式 RT-Linux 的设计	(315)
9.4.1 将 RT-Linux 嵌入	(315)
9.4.2 设计嵌入式 RT-Linux	(317)
9.5 本章小结	(317)
第 10 章 嵌入式 Linux 图形用户界面	(318)
10.1 嵌入式系统的图形用户界面概述	(318)
10.1.1 图形用户界面	(318)
10.1.2 嵌入式系统下的图形用户界面	(321)
10.1.3 嵌入式 Linux 环境下的 GUI	(325)
10.2 MiniGUI	(326)
10.2.1 MiniGUI 的起源	(326)
10.2.2 MiniGUI 的重要特色	(327)
10.2.3 MiniGUI 的结构	(329)
10.2.4 面向对象技术的运用	(331)
10.2.5 MiniGUI 的算法	(332)
10.3 MiniGUI 下的 Native Engine	(333)
10.3.1 开发私有引擎的必要性	(333)

10.3.2	Native Engine 的结构	(334)
10.3.3	鼠标驱动程序	(335)
10.3.4	键盘驱动程序	(337)
10.3.5	图形驱动程序	(338)
10.3.6	Native Engine 的典型应用	(342)
10.4	嵌入式 Linux 下图形用户界面的展望	(344)
10.5	本章小结	(344)
第 11 章	uClinux 的移植	(345)
11.1	uClinux 的移植简介	(345)
11.2	交叉开发工具	(346)
11.3	设备驱动程序	(350)
11.4	本章小结	(355)
第 12 章	嵌入式 Linux 的存储设备	(356)
12.1	使用紧缩闪存卡进行系统设计	(356)
12.1.1	Compactflash 适配器	(356)
12.1.2	安装硬件	(357)
12.1.3	安装软件	(358)
12.1.4	将 Compactflash 分区并格式化	(358)
12.1.5	构建嵌入式内核	(358)
12.1.6	构建 root 文件系统	(358)
12.1.7	设置 Webserver	(361)
12.1.8	安装 Boot Loader	(362)
12.1.9	测试系统	(362)
12.1.10	结论	(363)
12.2	使用 EPROM 进行系统设计	(363)
12.2.1	概况	(363)
12.2.2	系统操作	(363)
12.2.3	开发过程	(371)
12.2.4	实验结果	(375)
12.3	嵌入式 Linux 的网络存储设备	(375)
12.4	本章小结	(377)
第 13 章	嵌入式 Linux 与 Java	(378)
13.1	Java 和嵌入式系统	(378)
13.2	嵌入式 Linux 和 Java	(380)
13.3	本章小结	(386)
结束语		(387)
附录 A	GNU GPL —— GNU 通用公共许可证	(389)
附录 B	GDB 远程串行通信协议	(394)
附录 C	嵌入式 Linux 开发的相关网络资源	(397)

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

第一篇

基础知识

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

第 1 章 嵌入式系统与嵌入式的 Linux

本章要点：

什么叫嵌入式系统？嵌入式 Linux 又是什么？为什么用 Linux 来做嵌入式系统的操作系统呢？嵌入式 Linux 有什么样的特点？本章主要回答这些问题。

本章主要内容：

- 无所不在的嵌入式系统
- 嵌入式系统的特点
- RTOS 简介
- 嵌入式系统与 Linux 的绝配
- 各种版本的嵌入式 Linux

1.1 无所不在的嵌入式系统

1.1.1 身边的嵌入式系统

如果有人告诉读者，现在每个人都生活在一个满是嵌入式的世界里，是否会感到惊奇呢？但事实如此，不得不信，在手表、电话、手机，甚至电饭锅里，都有嵌入式系统的身影。嵌入式系统小到一个芯片，大到一个标准的 PC 板，种类繁多，让人顿生目不暇接之感。传统上一般按照计算机的体系结构、运算速度、适用领域等方面将其分为大型计算机、中型机、小型机和微计算机。近年来随着微电子技术的迅速发展，实际应用领域产生了很大变化。微型计算机虽然占据了全球计算机工业的 90% 市场，但是各种各样的应用于工业设备、电子产品中专用的计算机却大量涌现。这些计算机隐藏在各种产品和系统中，嵌入式计算机由此而得名。嵌入式计算机系统的正式定义为：以应用为中心，以计算机技术为基础，软件硬件可裁剪，符合应用系统对功能、可靠性、成本、体积、功耗的严格要求的专用计算机系统。

事实上，嵌入式计算机在数量上远远超过了各种通用计算机，PC 的各种输入输出和外部设备均是由嵌入式处理器控制的。每台 PC 的外部设备中包含了 5~10 个嵌入式微处理器，在工业流水线控制、通讯、仪器仪表、汽车、船舶、航空航天、军事装备、消费类产品



等领域更是嵌入式计算机的天下。现在,嵌入式系统带来的工业年产值已超过了1万亿美元,1997年美国嵌入式系统大会的报告预测:未来5年仅基于嵌入式计算机系统的全数字电视产品,就将在美国产生一个每年1500亿美元的新市场。美国汽车大王福特公司的高级经理也曾宣称:“福特出售的‘计算能力’已超过了IBM”,由此可以推测嵌入式计算机的应用规模。美国著名未来学家尼葛洛庞帝1999年1月访华时预言:4~5年后,嵌入式智能产品将是继PC和因特网之后的最伟大的发明。

嵌入式产品可分为如下几类:

(1) 信息电器

后PC时代,计算机将无处不在,家用电器将向数字化和网络化发展。电视机、电冰箱、微波炉、电话等都将嵌入计算机并通过家庭控制中心与Internet连接,转变为智能网络家电。届时,人们在远程用手机等就可以控制家里的电器,还可以实现远程医疗、远程教育等。目前,智能小区的发展打开了机顶盒市场,机顶盒将成为网络应用的终端。它不仅可以使模拟电视接收数字电视节目,而且还可以上网。

(2) 移动计算设备

移动计算设备包括手机、PDA、掌上电脑等各种移动设备。中国拥有最大的手机用户。而掌上电脑或PDA由于易于使用、携带方便、价格便宜等特点,未来几年将在我国得到快速发展。PDA与手机已呈现融合趋势,用掌上电脑(或PDA)上网,人们可以随时随地获取信息。

(3) 网络设备

网络设备包括路由器交换机、Web server、网络接入盒等各种网络设备。基于Linux的网络设备,其价格低廉,将为企业提供更廉价的网络方案。

(4) 工控、仿真等

在工控领域,嵌入式设备早已得到了广泛应用。我国的工业生产需要完成智能化、数字化改造,智能控制设备、智能仪表自动控制等为嵌入式系统提供了很大的市场。而工控、仿真、数据采集等军用领域一般都要求操作系统支持。

1.1.2 嵌入式系统的特点

嵌入式系统是将先进的计算机技术、半导体技术和电子技术与各个行业的具体应用相结合后的产物。这一点就决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统。嵌入式系统工业的基础是以应用为中心的芯片设计和面向应用的软件产品开发。

1. 嵌入式系统的产品特征

嵌入式系统是面向用户、面向产品、面向应用的。如果独立于应用自行发展,则会失去市场。

与通用计算机不同,嵌入式系统是针对具体应用的专用系统。一般具有成本敏感性,它的硬件和软件都必须高效率地设计,量体裁衣去除冗余,力争在同样的硅片面积上实现更高的性能。好的嵌入式系统是完成目标功能的最小系统,这样的产品才更具有竞争力。

嵌入式处理器的功耗、体积、成本、可靠性、速度处理能力、电磁兼容性等方面均受到应用要求的制约。这些也是各个半导体厂商之间竞争的热点。嵌入式处理器针对用户的具体需求,对芯片配置进行裁剪和添加,才能达到理想的性能,但同时还会受用户订货量的制约。因此,不同的处理器面向的用户也不同,可能是一般用户、行业用户或单一用户。

嵌入式系统一般要求高可靠性。在恶劣的环境或突然断电的情况下,要求系统仍然能够正常工作。还有许多嵌入式应用要求实时功能,这就要求嵌入式操作系统(EOS)具有实时处理能力。

嵌入式系统和具体应用有机地结合在一起,它的升级换代也是和具体产品同步进行。因此,嵌入式系统产品一旦进入市场,便具有较长的生命周期。嵌入式系统中的软件一般都固化在只读存储器中或闪存中,而不是存储在磁盘等载体中。

2. 嵌入式系统软件的特征

嵌入式处理器的应用软件是实现嵌入式系统功能的关键。对嵌入式处理器系统软件和应用软件的要求也与通用计算机有所不同。

(1) 软件要求固化存储。为了提高执行速度和系统可靠性,嵌入式系统中的软件一般都固化在存储器芯片或单片机中,而不是存储于磁盘等载体中。

(2) 软件代码高质量和高可靠性。尽管半导体技术的发展,使处理器速度不断提高,芯片上存储器容量不断增加,但在大多数应用中,存储空间仍然是宝贵的,还存在实时性的要求。为此,要求程序编写和编译工具的质量要高,以减少程序二进制代码长度,提高执行速度。

(3) 许多应用要求系统软件(OS)具有实时处理能力。在多任务嵌入式系统中,对重要性各不相同的任务进行统筹兼顾的合理调度是保证每个任务及时执行的关键。单纯通过提高处理器速度是无法完成和没有效率的。这种任务调度只能由嵌入式操作系统来完成,因此要求操作系统具有实时处理能力。

(4) 多任务操作系统是知识集成的平台,也是走向工业标准化道路的基础。

3. 嵌入式系统开发需要的开发工具和环境

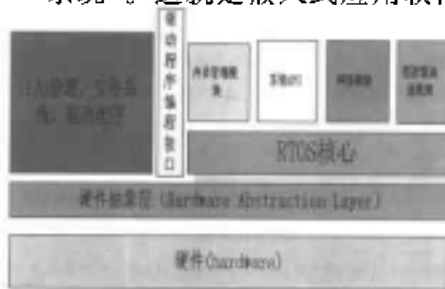
通用计算机具有完善的人机接口界面,增加一些开发应用程序和环境即可进行对自身的开发。而嵌入式系统本身不具备自行开发能力。即设计完成后,用户通常不能对其中的程序功能进行修改,必须有一套开发工具和环境才能进行开发。这些工具和环境一般是基于通用计算机上的软硬件设备、各种逻辑分析仪和混合信号示波器等。

4. 嵌入式系统软件需要 EOS 开发平台

通用计算机具有完善的操作系统和应用程序接口(API),是计算机基本组成不可分割的一部分。应用程序开发以后,应用软件都在 OS 平台上运行。嵌入式系统则不同,应用程序可以没有操作系统而直接在芯片上运行。但为了合理地调度多任务、利用系统资源、系统函数以及专家函数接口,用户必须自行选配嵌入式操作系统(EOS)开发平台。这样才能保证程序执行的实时性和可靠性,并减少开发时间,提高软件质量。一个优秀的 EOS 是嵌入式系统成功的关键。

1.1.3 RTOS 简介

上面谈到了嵌入式实时系统。嵌入式系统大多工作在对实时性要求很高的环境中，这样的操作系统往往称为实时多任务操作系统(RTOS)。从性能上讲，RTOS 和普通的 OS 存在的区别主要是在“实时”二字上。在实时计算中，系统的正确性不仅依赖于计算的逻辑结果，也依赖于结果产生的时间。从这个角度上看，可以把实时系统定义为“一个能够在指定或者确定的时间内，完成系统功能和对外部或内部、同步或异步时间做出响应的系统”。这就是嵌入式应用软件的基础和开发平台。目前，大多数嵌入式开发还是在单片



乃要有一个主程序负责调度各个任务。

代码中的程序，系统复位后首先执行，相当于用户的主程序立在 RTOS 之上的。不仅如此，RTOS 还是一个标准的内核，将寄存器等资源都包装起来，留给用户一个标准的 API 接口，并在不同任务之间分配 CPU 时间。RTOS 是针对不同处理器内核。RTOS 可以面对几十个系列的嵌入式处理器如 MPU、MCU、DSP、SOC 等提供类似的 API 接口。这是 RTOS 应用程序的开发基础。因此，基于 RTOS 上的 C 语言程序具有极大的可移植性。据专家测算，对于优秀的 RTOS 应用程序，其跨处理器平台的移植只需要修改 1%~4% 的代码。在 RTOS 基础上，可以编写出各种硬件驱动程序、专家库函数、行业库函数和产品库函数，与通用性的应用程序一起作为产品销售，促进行业的知识产权交流，因此 RTOS 又是一个软件开发平台。

RTOS 的结构如图 1-1 所示。其中，RTOS 的核心位于硬件抽象层和系统 API、网络模块、图形驱动函数库之间，占有重要的位置。

图 1-1 RTOS 体系结构图

从 1981 年 Ready System 发展了世界上第一个商业嵌入式实时内核 (VRTX32) 起，到今天已经有近 20 年的历史。在 20 世纪 80 年代，RTOS 的产品还只支持一些 16 位的微

处理器。这时候的 RTOS 还只有内核,以二进制代码为主。当时的产品除 VRTX 外,还有 IPI 公司的 MIOS 和 80 年代末 ISI 公司的 PSOS。其产品主要用于军事和电信设备。进入 20 世纪 90 年代,现代操作系统的设计思想,如微内核设计技术和模块化设计思想,开始渗入 RTOS 领域。老牌的 RTOS 厂家如 Ready System(在 1995 年与 Microtec Research 合并),也推出新一代的 VRTXsa 实时内核,新一代的 RTOS 厂家 Windriver 推出了 Vxwork。另外,在这个时期,各家公司都有力求摆脱完全依赖第三方工具的制约,而通过自己收购、授权或使用免费工具链的方式,组成一套完整的开发环境。例如,ISI 公司的 Prismt、著名的 Tornado(Windriver)和老牌的 Spectra(VRTX 开发系统)等等。

进入 20 世纪 90 年代中期,互联网在北美日渐流行。网络设备制造商和终端产品制造商都要求 RTOS 有网络和图形界面的功能。为了方便使用大量现存的软件代码,希望 RTOS 厂家都支持标准的 API,如 POSIX、Win32 等,以及要求 RTOS 的开发环境与 UNIX 和 Windows 一致。这个时期的代表性产品有 Vxwork、QNX、Lynx 和 Windows CE 等。

进入 20 世纪 90 年代后,RTOS 在嵌入式系统设计中的主导地位已经确定。越来越多的工程师使用 RTOS,更多的新用户愿意选择购买而不是自己开发。RTOS 的技术发展有以下一些变化:

(1) 因为新的处理器越来越多,RTOS 自身结构的设计更易于移植,以便在短时间内支持更多种的微处理器。

(2) 开放源码之风已波及 RTOS 厂家。数量相当多的 RTOS 厂家出售 RTOS 时,就附加了源程序代码并含生产版税。

(3) 后 PC 时代,更多的产品使用 RTOS,它们对实时性要求并不高,如手持设备等。微软公司的 Windows CE、Palm OS、Java OS 等 RTOS 产品就是顺应这些应用而开发出来的。

(4) 电信设备、控制系统要求的高可靠性,对 RTOS 提出了新的要求。

(5) 嵌入式 Linux 已经在消费电子设备中得到应用。韩国和日本的一些企业都推出了基于嵌入式 Linux 的手持设备。嵌入式 Linux 得到了许多半导体厂商的支持和投资,如 Intel 和 Motorola 等。

1.1.4 RTOS 在中国

中国的计算机基础工业落后于西方国家,在嵌入式处理器上也是如此。但是嵌入式系统面向应用的特点,决定了处理器应用开发的产值要占整个嵌入式工业的大部分。将嵌入式处理器与具体应用结合这种知识创新,只能由精通应用系统的用户来完成。因此在嵌入式系统方面,中国存在着相当大的发展机会。中国已经有 10 万余名单片机开发工程师。其中,很多人都是在资料和信息有限的条件下通过实践,精通单片机,并研制出了自己的产品。但与国外的开发相比,开发手段和水平还相对较低,标准化程度不够,重复劳动较多。这些问题主要是由于单片机开发中缺乏工程化、标准化管理和行业联合,在引入 RTOS 和嵌入式系统软件工程管理后可望得到很大的改变。

1.2 自由的企鹅——Linux

如果以人类年龄来算, Linux 还不过是个 10 岁的小孩子。它源于一位芬兰大学生——Linus Torvalds 的课余作品。当时, Linus Torvalds 正在学习计算机科学家 Andrew S. Tanenbaum 开发的 Minix 操作系统, 但发现 Minix 的功能很不完善, 于是就编写了一个保护模式下的操作系统(多酷!), 这就是 Linux 的原型。最开始, Linux 被定位于黑客用的操作系统, 并被放到 FTP 服务器上供人们自由下载。从此, 这个娃娃般的操作系统就在人们的关爱中, 迅速地以每两个星期修正一次版本的速度成长起来。现在, Linux 已经成为当前最流行的免费操作系统。Linux 的产生, 完全是无数计算机爱好者共同努力的结果。Linux 从诞生之日起就伴随着 Internet 一同成长。迄今为止, Linux 操作系统已经成为具有全部 UNIX 特性的 POSIX 兼容的操作系统, 而且只要遵守通用公共许可证(GPL)的条款, 任何人都可以自由使用 Linux 的源程序。Linux 操作系统有以下几大特征:

- 符合国际通用标准;
- 强大的兼容性;
- 先进的网络特征;
- 拥有真正的多用户、多任务能力;
- 具有动态链接能力;
- 系统性能十分稳定;
- 可移植非常灵活。

Linux 内核的开发是由 Linus Torvalds 领导的内核开发小组进行的。世界各地的高手们将自己对 Linux 内核需要做的改动交给 Linux 开发小组, 由这个小组进行统一控制, 随时对内核进行更新升级。整个开发的过程遵循同步版本系统(CVS)的版本控制, 保证开发的质量。目前在他们的公共站点(<http://www.kernel.org/>)上, 几乎每三天进行一次内核的升级, 目前最新的内核是 Linux2.4。

Linux 的系统界面和编程接口与传统的 UNIX 类似。在 UNIX 下的程序员可以很方便地从 UNIX 环境转移到 Linux 环境, 而不像从 UNIX 环境转移到 Windows 开发环境那样复杂。在 Linux 平台上的应用软件也不断地得到了扩充。许多著名的商业软件都有 Linux 下的版本: Applix 公司和 Star 公司提供了多种字处理、电子表格和图形处理的应用软件; Corel WordPerfect 8、Adabas D 和 Oracle 8 数据库、Netscape Navigator 6.0 网络浏览器、Apache 1.3.12 网络服务器、Adobe Acrobat Reader 4.0 等 Linux 下的应用程序都已纷纷推出。Linux 将来不再是高手的领域, 这种操作系统将来也必然走进千家万户, 成为 Windows 强大的竞争者。

近几年在网络服务器市场上, 商用 UNIX 系统在向大而复杂的方向发展, 使 UNIX 的复杂性不断增加, 管理整个 UNIX 系统也就变得越来越复杂。Linux 简单易用, 系统管理也比较容易上手, 从而成为在服务器高端的一个重要选择, 并且有不断上升的趋势, 大有取代昂贵、复杂的商用 UNIX 的趋势。几乎所有的商业用操作系统如 Microsoft 公司的 Windows98/NT Server/NT Workstation 系列, 都需要为每一个拷贝支付数量相当大的费

用。在其下的应用软件,获得每一个软件都需要支付大量的费用。在商用操作系统下建立一个开发工具链,除了要为操作系统本身付费外,还要为组成工具链的应用软件工具包支付大量的费用。但是 Linux 是免费软件,只要遵守 GPL(GNU General Public License)的规定,就可以免费获得拷贝。Linux 下有同样遵循 GPL 规定的 C、C++、Java 等一系列的软件工具开发包。从功能角度上看并不亚于商用开发包,同时可以极大地降低开发成本。这一优势是其他商用操作系统无法比拟的。

随着 Internet 的发展,各种智能信息产品层出不穷,机顶盒、数字电视等信息家电及个人 PDA、WAP 手机等产品都蕴藏着巨大的商机。IDG 发布的统计表明,未来的 4~5 年内,信息家电市场会增长五到十倍。各个硬件和软件厂商为此都摩拳擦掌,准备大干一场。智能数字产品的核心是其中的控制软件。在机顶盒、PDA、WAP 手机等产品的设计和功方面都很复杂,因此,需要有相应的操作系统支持。对这个市场觊觎已久的微软推出了 Windows CE,但是 Windows CE 并没有像定位为桌面系统的 Windows 那样横扫对手,所向披靡。其中一个重要原因就是现在厂商和用户多了一个选择——Linux。从 1997 年刮起来的 Linux 旋风,不仅使 Linux 在服务器市场上大出风头,而且正在蚕食着 Windows 桌面应用的市场。在智能数字终端操作系统这个才显端倪的市场上,Linux 更是大有先声夺人的气势。事实上,除了智能数字终端领域外,Linux 在移动计算平台、智能工控设备、金融业终端系统(POS/ATM 机等),甚至军事领域都有广泛的应用前景。这些 Linux 统称为“嵌入式 Linux”,下面就来揭开被称为“量体裁衣的 Linux 系统”的嵌入式 Linux 的面纱吧!

在早期,硬件设备很简单,软件的编程和调试工具也很原始,与硬件系统配套的软件都必须从头编写。程序大都采用宏汇编语言,调试是一件很麻烦的事。在 20 世纪 70 年代后期,出现了嵌入式系统的操作系统。它们采用汇编语言编写,而且只能运行在相应的专用处理器上,对新的处理器必须重新编写操作系统的所有代码。C 语言出现后,由于其稳定性和可移植性,采用 C 语言的操作系统开始流行,操作系统的编写方法有了很大进步。在 20 世纪 80 年代末,出现了几个著名的商业嵌入式操作系统。现在,它们已经成为主流嵌入式操作系统。比如,应用广泛的操作系统 Vxwork、pSOS、Nucleus 和 Windows CE。目前,国际上用于智能终端设备的嵌入式操作系统有 40 种左右(见第 9 章)。据最新资料,3Com 公司下属子公司的 PalmOS 全球占有份额达 50%,而 Windows CE 不过 29%。

由于嵌入式产品的体积、成本等方面有较严格的要求,所以处理器部分占用空间应尽可能的小。系统的可用内存和外存数量也要受限制,而嵌入式操作系统就运行在有限的内存(一般在 ROM 或快闪存储器)中,因此就对操作系统的规模、效率等提出了较高的要求。Windows CE 与 Windows 系列有较好的兼容性,无疑是 Windows CE 推广的一大优势。但从技术角度上讲,Windows CE 作为嵌入式操作系统有很多缺陷。Windows CE 没有开放源代码,使应用开发人员很难实现产品的定制。另外,Windows CE 在效率、功耗方面的表现并不出色,而且 Windows CE 像 Windows 一样占用过多的系统内存,应用程序庞大。Windows CE 的版权许可费用也是厂商不得不考虑的因素。

Linux 从 1992 年问世到现在,短短 8 年的时间内已发展成为一个功能强大设计完善的操作系统。目前 Linux 已可以与各种传统的商业操作系统分庭抗礼,占据了大部分市场。据 1999 年 IOS 统计,Linux 占有全球 Web 服务器总数的 28%,名列第一。IDC 统

计,2000 年 Linux 在服务器操作系统市场中占的份额达到 25%,Windows NT 占 38%,Novell 的市场份额为 19%,而各种版本 UNIX 的市场份额总和为 15%。

Linux 不仅在服务器领域取得了成功,在嵌入式领域也获得了飞速发展。目前,正在开发的嵌入式系统中,49% 的项目选择 Linux 作为嵌入式操作系统。Linux 之所以能在嵌入式系统市场上取得如此快的发展,与它自身的优良特性有着不可分割的关系:

1. 开放源码,丰富的软件资源

Linux 遵循 GPL(见附录 A),用法律保障了用户免费获得内核源代码的权利。由于嵌入式系统千差万别,往往需要针对应用修改和优化系统。这时能否获得源代码就至关重要了。

Linux 是自由的操作系统,它的开放源代码使用户获得了最大的自由度。Linux 上的软件资源十分丰富,每一种通用程序在 Linux 上都可以找到,并且每天都在增加。在 Linux 开发程序往往不需要从头做起,而是先选择一个类似的自由软件,进行二次开发。这就大大节省了开发工作量,缩短了开发时间。

2. 功能强大的内核,性能高效、稳定、多任务

Linux 的内核非常稳定。它的高效和稳定性已经在各个领域,尤其在网络服务器领域得到了事实的验证,而且 Linux 内核小巧灵活,易于裁剪。这使 Linux 能很适合嵌入式系统的应用。

3. 支持多种体系结构

Linux 能支持 X86、ARM、MIPS、ALPHA、SPARC 等多种体系结构。目前,Linux 已被移植到数十种硬件平台上,几乎所有流行的 CPU Linux 都支持。现在,Linux 已经可以在没有 MMU 的处理器上运行了(本书将要讨论的 uClinux 就是如此)。这就进一步促进了 Linux 在嵌入式系统中的应用。

4. 完善的网络通信、图形和文件管理机制

Linux 自产生之日起就与网络密不可分,网络是 Linux 的强项。另外,Linux 支持 ext2、fat16、fat32、romfs 等多种文件系统,有的文件系统具有防断电特性。在图形系统方面,Linux 上既有成熟的 X Window,也有 embedded QT、MiniGUI 等嵌入式 GUI,还有 svgalib、framebuffer 等优秀工具,可以适合不同的用途。

5. 支持大量的周边硬件设备,驱动丰富

Linux 上的驱动已经非常丰富了,支持各种主流硬件设备和最新硬件技术,而且随着 Linux 的广泛应用,许多芯片厂家也已经开始提供 Linux 上的驱动。这进一步促进了 Linux 各种硬件平台上的应用。

6. 大小功能都可定制

Linux 继承了 Unix 的优秀设计思想,内核与用户界面是完全独立的。它非常灵活,

各部分的可定制性都很强,能适合多种需求,是硬件资源有限的嵌入式系统。

正是嵌入式操作系统的特殊要求,为 Linux 在嵌入式系统中的发展提供了广阔的舞台。由于 Linux 的高度灵活性,程序员可以很容易地根据应用领域的特点对它进行定制开发,以满足自己的实际应用需要。Linux 固有的多任务、高效、稳定的系统特征,使 Linux 成为嵌入式操作系统中的新贵。嵌入式 Linux 一般是按照嵌入式目标系统的要求而设计,由一个体积很小的内核及一些可以根据需要进行裁剪的系统模块组成。一般来说,整个系统所占用的空间不会超过几兆大小。

目前,国外不少大学、研究机构和知名公司都加入了嵌入式 Linux 的开发工作,较成熟的嵌入式 Linux 不断涌现,如 RT-Linux、Embedix、uClinux 等。下面就来简要介绍一些嵌入式 Linux。



这是由美国新墨西哥理工学院开发的基于标准 Linux 的嵌入式操作系统(参见第 9 章)。截至目前为止,RT-Linux 已成功地应用于航天飞机的空间数据采集、科学仪器测控和电影特技图像处理等广泛的应用领域。RT-Linux 开发者并没有针对实时操作系统的特性重写 Linux 的内核,这样做工作量会非常大,而且要保证兼容性也非常困难。为此,RT-Linux 提供了一个精巧的实时内核,并把标准的 Linux 核心作为实时核心的一个进程同用户的实时进程一起调度。这样做的好处是对 Linux 的改动量最小,充分利用了 Linux 下现有的丰富软件资源。

NASA(美国国家宇航局)将装有 RT-Linux 的设备放在飞机上,以测量 George 飓风的风速,如图 1-2 所示。

图 1-2 该飞机上装有带 RT-Linux 操作系统的设备,以测量 George 飓风的风速

(2) Embedix

Embedix 是由嵌入式 Linux 行业主要厂商之一 Lineo 推出的,是根据嵌入式应用系统的特点重新设计的 Linux 发行版本。Embedix 提供了超过 25 种的 Linux 系统服务,包括 Web 服务器等。系统需要最小 8M 内存,3M ROM 或快闪内存。Embedix 基于 Linux2.2 核心,并已经成功地移植到了 Intel x86 和 PowerPC 处理器系列上。如其他 Linux 的发行版本一样,Embedix 可以免费获得。Lineo 公司还发布了另一个重要的软件产品,它可以让在 Windows CE 操作系统上运行的程序能够在 Embedix 系统上运行。Lineo 还将计划推出 Embedix 的开发调试工具包,基于图形界面的浏览器等。可以说,Embedix 是一种较完整的嵌入式 Linux 解决方案。

(3) XLinux

XLinux 是由美国网虎公司推出,主要开发者就是 CIH 病毒的作者——台湾的天才少年陈盈豪。这个让全世界电脑用户心惊胆战的少年,在加盟网虎几个月便开发出了基于 XLinux 的号称是世界上最小的嵌入式 Linux 系统,核心只有 143K 字节,而且还在不断减小。XLinux 核心采用了“超字元集”专利技术,让 Linux 核心不仅可能与标准字符集相容,还涵盖了 12 个国家地区的字符集。因此,XLinux 在推广 Linux 的国际应用方面有独特的优势。

(4) PocketIX

同时,致力于国产嵌入式 Linux 操作系统和应用软件开发的广州博利思软件公司,最近推出了嵌入式 Linux 中文操作系统——PocketIX 预览版。它基于标准的 Linux 内核,并包括一些可以根据需要进行定制的系统模块。PocketIX 预览版支持标准以太网和 TCP/IP 协议,支持标准的 X Window,带中文支持,提供桌面和窗口管理功能,带 Web 浏览器和文件管理器,并支持智能拼音和五笔字型输入,可适应个人 PDA、WAP 手机、机顶盒等广泛的智能信息产品。

(5) uClinux

uClinux 是专门用于微型控制领域的嵌入式 Linux 操作系统,它已经被成功地移植到了很多平台上。本书将详细探讨该嵌入式 Linux 操作系统的核心组成、关键技术和应用程序的开发等问题。

(6) 红旗嵌入式 Linux

这是由红旗公司推出,也是国内做得比较好的嵌入式 Linux。

目前,中科院计算所自行开发的开放源码的嵌入式操作系统——Easy Embedded OS (EEOS)也已开始进入实用阶段。该嵌入式操作系统重点支持 p-Java,系统目标一方面是小形化,一方面能重用 Linux 的驱动和其他模块。由于有中科院计算所的强大科研力量做后盾,EEOS 有望发展为功能完善、稳定、可靠的国产嵌入式操作系统平台。

除了上面介绍的几种以外,还有一些比较优秀的嵌入式 Linux 操作系统,在后续章节中会有详细的叙述。

曾被媒体炒得火热的维纳斯与女娲之争,使用户认识到了嵌入式操作系统领域的巨大商机。据预测,我国信息家电产品市场规模 2001 年将达到 500 亿~700 亿元规模,2003 年达到 2000 亿~2500 亿元规模。由于 Linux 开放源码的特点,全世界的开发厂商都站在同一起跑线上。国内的研究机构和企业也正在投入人力物力,力争在嵌入式操作系统市场上有所作为。

但应清醒地看到,绝大部分的嵌入式系统的硬件平台还掌握在外国公司的手中。国产的嵌入式操作系统在技术含量、兼容性、市场运作模式等方面还有很多工作要做,但是嵌入式操作系统的巨大商业价值和 Linux 的开放性,为民族软件产业的发展提供了难得的机会。应该在跟踪国外嵌入式操作系统最新技术的同时,坚持自主知识产权,力争找到自己的突破点,探索出一条适合中国国情的嵌入式操作系统的发展道路。

本书主要的着眼点就在于基于嵌入式 Linux 系统的开发和应用。全书的核心是紧紧地围绕着嵌入式 Linux 系统的构成和组建、应用程序的开发、实时系统的分析这条主线所展开的。建议读者按步骤循序渐进地学习。

1.3 本章小结

有人认为:嵌入式系统是信息产业走向 21 世纪知识经济时代的最重要的经济增长点之一。这是一个不可垄断的工业,对中国的信息产业来说充满了机遇和挑战。嵌入式工业的基础是以应用为中心的芯片设计和面向应用的软件开发。实时多任务操作系统(RTOS)进入嵌入式系统工业的意义,不亚于历史上机械工业采用三视图技术后的发展。对嵌入式软件的标准化和加速知识创新是一个里程碑。同时,嵌入式 Linux 的巨大优越性让我们将目光都投向它的身上。以嵌入式 Linux 为平台开发嵌入式应用,必将有光明的前途。

本章初步讲解了嵌入式系统的基本概念,以及 Linux 在这个领域强大的功能。准备好了,就要进入这个神奇的世界了。在起步之时,应先从基本的开始,下面的部分将会讲到一些 Linux 的基本知识和技巧。如果读者是一位 Linux 的高手,完全可以跳过不看。如果不是,那就从头开始吧!

第 2 章 Linux 概论



本章要点：

无论是经验丰富的开发人员还是初学者,读读本章都会大有好处。在本章,各位将了解到 Linux 的基本原理和体系构成,还有 Linux 的基本指令集以及一些有代表性的 Linux 的嵌入式版本,为后续章节的学习打下基础。

本章主要内容：

- 走进自由天地——初识 Linux
- Linux 常用的版本
- Linux 操作系统基本构成
- Linux 基本指令
- 五脏俱全的嵌入式 Linux
- RT-Linux 简介
- uClinux 简介

2.1 走进自由天地——初识 Linux

2.1.1 Linux 的成长

1. 成长

在 10 年前,一位芬兰赫尔辛基大学的年轻人 Linus Torvalds 为了实习 Minix(一套由计算机科学家 Tanenbaum 开发的 Unix 操作系统,可以支持 8086 和 80386,在很多 PC 机上较为流行),又不想天天排队等着上机,一咬牙自己掏钱买了台在当时还很不错的 486 微机。随后,他发觉 Minix 的功能十分不完善,便决心自己写一个保护模式下的操作系统,便诞生了 Linux 的原型。刚开始,工作是痛苦的,Linus 自己这样说道:

“最开始的确是一次痛苦的旅行,但是我终于可以拥有自己的一些设备驱动程序了,并且排除错误也变得更加容易,我开始用 C 语言来开发程序,这大大加快了开发速度……”

我梦想有一天我能在 Linux 下重新编写 GCC……

我花了两个月的时间来进行基本的配置工作,直到我拥有了一个磁盘驱动(有很多错

误,但很巧,能在我自己的机器上工作)和一个小小的文件系统,这就是我的 0.01 版,它并不完善,连软盘驱动程序都没有,什么事情也做不了,但是我已经被它吸引住了,除非我能够放弃使用 Minix,不然我不会停止使用它。”

Linus 在 1991 年 10 月 5 日发布了 Linux 的第一个正式版本,即 0.02 版,它能做的的确很少,简直有点可怜。它被设计成一个黑客型的操作系统,仅能运行 bash(GNU 的一个 UNIX shell 程序)和 GCC(GNU 的 C 编译器),再也干不了什么别的事情了。这时候的 Linux 还是个小娃娃,与今天常常用到的 Linux 差别很大。Linus 一开始就开放了 Linux 的源代码,并放到 FTP 服务器上供人们自由下载。管理员认为这是 Linus 本人的 Minix,所以就命名为 Linux,这个小娃娃也就有了它的名字。

星星之火,可以燎原。这个小小的娃娃以惊人的速度成长了起来,发展远远超过了 Linus 本人当初的估计。Linux 在 1994 年 3 月 14 日发布了其第一个正式版本 1.0 版,而 Linux 的讨论区也成为了 USENET 上最为热闹的讨论组之一,每天发表的文章数以万计。Linux 系统的内核也以惊人的速度成长,如今用户所用到的流行版本,如 Redhat 7.0,红旗 2.0 等的内核已经达到了 2.2.16。而内核版本号为 2.4 的 Linux 也问世了。短短几年,Linux 日趋成熟。

2. Linux 的辉煌

如今,Linux 已经成为微软的最强劲的对手。作为一个完全免费的操作系统,Linux 在全世界的用户已数以千万计。Linux 在网络服务器方面有着无比的优越性,Linux + Apache 已经成为网络上最为普遍的服务器构架模式。例如中国最有名的门户网站之一新浪网就采用了这种模式。Linux 作为数据库服务器也有着光明的前景:Oracle 公司的 Oracle 8I、Informix 公司的 InformixSE、Sybase 公司的 Sybase、Inprise 公司的 InterBase 5.0 和 IBM 公司的 DB2 等这些重量级的数据库都推出了它们的 Linux 版本。

Linux 不仅在服务器端有着惊人的表现,在家用桌面应用中也是异军突起。Linux 一改 UNIX 类操作系统拒人以千里之外的字符界面,而使用了友好的桌面环境如 KDE(K 桌面环境)、GNOME 等。现在,Linux 拥有大量的各类应用软件(从诸如 WordPerfect 8.0 之类的文字处理软件到 K Development 软件集成开发环境),再加上它强大的网络功能(Linux 本来就号称“网络之子”),对硬件的良好支持,使得越来越多的家庭用户也在自己的电脑里装上 Linux。Linux 已经成为同微软争夺低端服务器和桌面用户的最强有力的一个对手。它的中文化的工作早已经开始,目前成熟的中文 Linux 版本有如 TurboLinux 6.0、蓝点 2.0、红旗 2.0 等。如果厌倦了 Windows 的蓝天白云,如果对“Win 98”的频繁死机而感到头疼不已,如果对微软的种种恶劣行径感到气愤,那么请走进 Linux 这个奇妙的天地吧! Linux 的奇妙是远非这本书可以讲完的,很多都需要大家细心地去体会。

下面给出一些关于 Linux 的发展历史以及关键人物的资料:

Linux 的发展历史以及关键人物:

Ken Thompson、Dennis Ritchie:发明 UNIX,于 20 世纪 60 年代末。

Brian Kernighan, Dennis Ritchie: The C Programming Language,于 20 世纪 70 年代末。

Richard Stallman: FSF、GNU、GPL、emacs 和 gcc,于 20 世纪 80 年代中期。

Andrew S. Tanenbaum: Minix; Operating Systems: Design and Implementation, 于 20 世纪 80 年代末 90 年代初。

Linus Torvalds: 设计了 Linux, 于 20 世纪 90 年代。

Eric Raymond: 《黑客文化简史》、《如何成为一名黑客》、《大教堂和市集》、《开拓智域》和《魔法大锅炉》。

Linux 发展的重要里程碑:

1990 年, Linus Torvalds 首次接触 Minix。

1991 年中, Linus Torvalds 开始在 Minix 上编写各种驱动程序等操作系统内核组件。

1991 年底, Linus Torvalds 公开了 Linux 内核。

1993 年, Linux 1.0 版发行, Linux 转向 GPL 版权协议。

1994 年, Linux 的第一个商业发行版 Slackware 问世。

1996 年, 美国国家标准技术局的计算机系统实验室确认 Linux 版本 1.2.13 (由 OpenLinux 公司打包) 符合 POSIX 标准。

1999 年, Linux 的简体中文发行版问世。

2.1.2 Linux 与 GNU

谈及 Linux, 就不能不说说与 Linux 联系紧密的 GNU。GNU 即 GNU's Not Unix 的缩写。GNU 是由自由软件基金会的董事长 Richard M. Stallman 于 1984 年发起的, 距今已经有 17 年的历史了。Stallman 本来就职于美国麻省理工学院人工智能实验室, 是世界上屈指可数的顶尖程序员。他认为: UNIX 虽然不是最好的操作系统, 但也不会太差。他有能力将 UNIX 不足的地方加以改进, 使它成为一个优秀的操作系统。开发这个系统的目的就是让所有的计算机用户都可以自由地获得这个系统, 任何人都可以免费使用这个系统的源代码, 并可以相互地自由拷贝。所以在使用 GNU 软件时可以理直气壮地说, 我们使用的是正版货!

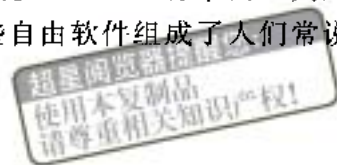
GNU 也有自己的版权声明 (GPL, 见附录 A), 即有名的 Copyleft (相对于英语的 Copyright, 版权)。本版权声明规定, 用户在获得自由软件之后, 可以自由使用和修改, 并重新传播。但是, 用户在散布 GNU 软件之时, 必须让下一个用户也有权利获得源代码, 并且必须告之他这一点。关于这一切在本书附录中都会有详细的介绍。总的来讲, GPL 就是用来保证 GNU 永远是免费和公开的。在本书附带的光盘中所包含的所有程序, 包括开发环境的软件系统和各种应用程序, 连同本书中的应用程序一道, 都严格遵循 GPL 2.0 版本的规定。在阅读本书的声明后, 如果赞同并愿意遵守, 就可以使用光盘中的所有程序并对其进行修改、拷贝和传播。

GNU 目前已经推出了许多优秀的软件, 如 Emacs (一个功能强大的编辑环境)、GCC (性能优越的多平台的 C、C++、Fortran 编译器) 和 K Development (一个优秀的集成开发环境)。其中, GCC 的成功给 GNU 带来了前所未有的影响。GCC 是一种可以在 11 种平台上运行的编译器, 效率惊人, 比一般编译器的平均效率高 20%~30%。例如, 各位电脑游戏迷们熟悉的 3D 游戏——Quake, 就是用 GCC 的 DOS 移植版本 digpp 编写的。其实, 目前的自由软件, 如 BIND、Perl、Apache、TCP/IP 等都是自由软件的经典之作。可以说,

如果没有它们,那么现在的互联网的真实面貌将惨不忍睹。

开发 Linux 使用了许多 GNU 工具。Linux 系统上用于实现 POSIX.2 标准的工具几乎都是 GNU 项目开发的, Linux 内核、GNU 工具以及其他一些自由软件组成了人们常说的 Linux。Linux 的软件构成如下:

- 符合 POSIX 标准的操作系统 Shell 和外围工具;
- C 语言编译器和其他开发工具及函数库;
- 窗口系统,如 X Window;
- 各种应用软件,包括字处理软件、图象处理软件等;
- 其他各种 Internet 软件,包括 FTP 服务器、WWW 服务器等;
- 关系数据库管理系统等。



2.2 Linux 常用的版本

Linux 的主要发行版如表 2-1 所示。除表中列出的发行版之外,还有大量的发行版存在,比如 Slackware、OpenLinux、Mandrake 等等。建议新手使用 RedHat Linux 或自己喜欢的某种本地化发行版(如红旗 2.0 版就很不错,界面很友好)。

表 2-1 Linux 的主要发行版

名称	特征	网络地址	备注
DebianGNU/Linux	系统初始化: Sys V init, 采用 dselect 和 dpkg 作为软件包管理程序	http://www.debian.org ftp://ftp.debian.org/debian	由 GNU 发行的 Linux 版本, 最符合 GNU 精神。提供了最大的灵活性, 适合 Linux 的高级用户
RedHat Linux	系统初始化: Sys V init, 采用 RPM 软件包管理工具大量图形化的管理工具	http://www.redhat.com ftp://ftp.redhat.com	CERNET 各大型的 FTP 网站均有最新的 RedHat Linux。采用 RPM 的软件包管理方式, 软件的安装、卸载和升级非常方便, 并提供了大量的图形化管理工具, 是初学者的最佳选择

简体中文 Linux 发行版有:

- TurboLinux: 国内最早的简体中文发行版之一, 目前流行的版本是 TurboLinux 7.0。
- BluePointLinux: 最新发布 Linux 中文版。利用 Linux 2.2 内核的 FrameBuffer, 可在控制台获得中文输入输出。具备多内码支持, 目前可以支持大陆国标码和港台大五码, 与 RedHat Linux 兼容。
- 红旗 Linux: 由中科红旗软件公司出品, 是以 Intel 和 Alpha 芯片为 CPU 的服务

器平台上第一个国产的操作系统版本。使用了 2.2.16 版核心,预装了炎黄中文平台,彻底支持 Informix-SE 等流行数据库和多种 PC 机与服务器。

注意:Linux 的版本号分为两个部分:内核和发行套件版本。对于初学者来说经常会搞混淆。实际上,内核版本指的是在 Linus 领导下的开发小组开发出的版本号,一般来讲,第二位小数为偶数的标本表明这是一个稳定的版本。为奇数的话就是个不太稳定的实验版本(beta 版)。而一些厂家或者组织将 Linux 系统内核和应用软件和文档包装起来,并提供一些安装界面和系统设定工具,这样就构成了一个发行套件,如常用的 Redhat Linux、红旗 Linux 等。

本书在介绍嵌入式 Linux 开发时,宿主机上采用的是 Redhat Linux 6.2 版本。在此简要介绍一下 Redhat Linux 系列。

Redhat Linux 是由 Redhat Software 公司发布的,想必“小红帽”的大名各位已经是不会陌生的了。Redhat 问世比 Slackware 和 Debian(Linux 的另外两个有名的版本)晚,但是后来居上,有凌驾于这两者之上的趋势。其特点有:

- 支持的硬件平台多。Redhat Linux 可以同时支持 Intel、Alpha 和 Sparc 三种硬件平台。
- 安装界面优秀。
- 独特的 RPM 升级方式。红帽的所有软件包都是以 RPM(Redhat Package Manager)方式包装的。这种方式可以让用户轻松地进行软件升级和卸载应用软件和系统,一点也不比 Windows95 上的 Install Shield 差。
- 丰富的软件包。红帽搜集了大量优秀的软件包,包括 GNU 软件和很多的 ShareWare 软件,并且所有的软件都经过了精心调试,为用户省掉了很多时间。
- 安全性能好。经过红帽配置后的系统,其安全性能非常优秀,并且提供 Pluggable Authentication Modules(PAM)以加强系统安全性能和系统管理的扩充性。
- 系统管理方便。在红帽中,为顾客提供了一套 X Window 下完整的系统管理软件,使原本非常复杂的系统管理在红帽中变得十分轻松。
- 帮助文档完整而丰富。在/usr/doc 下收录了完整的 HOWTO、LDP、FAQ 等系列说明文件,并且有详尽的 man 手册供在线查找。

2.3 Linux 操作系统基本构成

在本节先了解 Linux 操作系统的基本体系构成的知识。首先,给出系统的主要构成,然后再一一阐述其功能。

2.3.1 系统概述

整个 Linux 操作系统的结构图如图 2-1 所示。

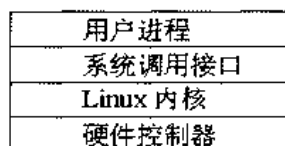


图 2-1 Linux 系统的构成示意图




从图中可以看到, Linux 系统由 4 个主要的部分组成:

(1) 用户进程

用户应用程序是运行在 Linux 操作系统最高层的一个庞大的软件集合。当一个用户程序在操作系统之上时,它就是操作系统的一个进程。计算机不同,程序的集合会有所变化,但是一般来讲,对于基本的系统,总会有一些如文字处理、Webserver 等程序。

(2) 系统调用接口

为了在应用程序中实现特定的任务,可以通过系统调用来调用系统内核中特定的过程,以实现特定的服务。一般认为,这些调用和服务也是操作系统内核的一部分,内核的编程接口也属于这一部分。系统调用本身也是由若干条指令组成的过程,但与一般过程不同的是:系统调用运行在内核模式,而一般的进程运行在用户模式。

 **注意:**进程和程序是有不同的。一般认为,程序是存储在磁盘上包含可执行机器指令和数据的静态实体。而进程是具有一定功能的程序关于一个数据集合的一次运行活动,是处于活动状态的计算机程序。进程是一个随执行过程不断变化的实体。和程序要包含指令和数据一样,进程也包含程序计数器和所有 CPU 寄存器的值。同时它的堆栈中存储着加子程序参数、返回地址以及变量之类的东西。

2.3.2 Linux 内核

从程序员的角度来讲,操作系统的内核提供了一个虚拟的机器接口。它抽象了许多硬件细节,程序可以以某种统一的方式来进行数据处理,而内核将所有的硬件抽象成统一的虚拟接口。

Linux 以统一的方式支持多任务,而这种方式对用户进程是透明的,每一个进程运行起来就好像只有它一个进程在计算机上运行一样,独占内存和其他的硬件资源。实际上内核在并发地运行几个进程,并且能够让几个进程公平合理地使用硬件资源,也能使各进程之间互不干扰安全地运行。

Linux 将系统的一些关键性程序分离出来构成所谓操作系统内核。像大部分 UNIX 操作系统的内核那样,Linux 内核必须完成下面的一些任务:

- 对文件系统的读写进行管理,把对文件系统的操作映射成对磁盘或其他块设备的操作。
- 管理程序的运行,为程序分配资源,并且处理程序之间的通讯。
- 管理存储器,为程序分配内存,并且管理虚拟内存。
- 管理输入输出,将设备映射成设备文件。
- 管理网络。

内核必须包含虚拟文件系统(VFS)管理程序以及将各种具体文件系统映射成 VFS 的程序。另外的几项功能几乎也像文件系统一样复杂。首先是对于内存的管理,Linux 使用虚拟存储管理方式,利用现代处理器的页面映射能力,在 x86 处理器上,Linux 使用 4GB 地址空间,但是,系统的物理存储器总是少于这个数字。操作系统除了使用物理存储器外,也支持将硬盘空间映射成为虚拟内存。所有存储器(物理内存和虚拟内存)被分成大小相等的页面,系统通过给出页号和页面内偏移量对某个内存地址进行访问。在物理内存紧张的时候,操作系统必须把某些没有使用的页面从内存移动到硬盘上以便腾出空闲的页面供程序使用,这个过程称为交换(SWAP)。显然交换需要虚拟存储空间。通常情况下,Linux 用交换分区(SWAP 分区)来处理这个问题,在硬盘上开设一个独立的分区专门用于映射虚拟内存,这种分区称为交换分区。交换分区可以不止一个,之所以这样,是由于早期的 Linux 核心要求每个交换分区不能超过 128MB。对于较重负荷的服务器,交换内存用到 256MB 甚至更多都是很正常的事情,因此那时的系统经常有多个交换分区。目前这个限制已经去除,但仍有人使用多于一个的交换分区。

核心的另外一个任务是执行用户程序,为此核心必须支持可执行文件格式。Linux 使用多种可执行文件的格式,诸如 elf、aout 等等(与 MS-DOS 不同,没有办法从名字上区分一个文件到底是什么格式,核心只关心二进制文件的具体形式)。

Linux 内核由 5 个主要的子系统组成,如图 2-2 所示。

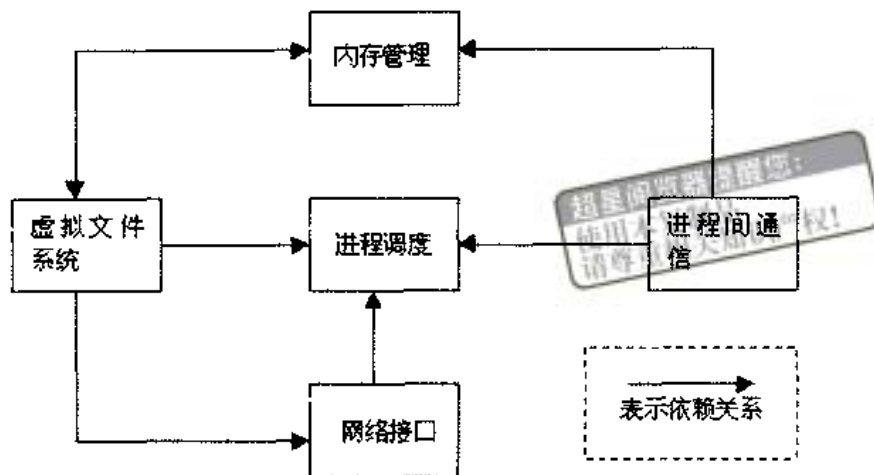


图 2 2 Linux 内核子系统的抽象结构

1. 进程调度(SCHED)

它控制着进程对 CPU 的访问。当需要选择下一个进程运行时,由调度程序选择最值得运行的进程。可运行进程实际上是仅等待 CPU 资源的进程,如果某个进程在等待其他资源,则该进程是不可运行进程。Linux 使用了比较简单的基于优先级的进程调度算法选择新的进程。

2. 内存管理(MM)

内存管理允许多个进程安全地共享主内存区域。Linux 的内存管理支持虚拟内存,即在计算机中运行的程序,其代码、数据和堆栈的总量可以超过实际内存的大小,操作系统只将当前使用的程序块保留在内存中,其余的程序块则保留在磁盘上。必要时,操作系统负责在磁盘和内存之间交换程序块。

内存管理从逻辑上可以分为硬件无关的部分和硬件相关的部分。硬件无关的部分提供了进程的映射和虚拟内存的对换;硬件相关的部分为内存管理硬件提供了虚拟接口。

3. 虚拟文件系统(VFS)

它可以隐藏各种不同硬件的具体细节,为所有设备提供统一的接口,VFS 还支持多达数十种不同的文件系统,这也是 Linux 较有特色的一部分。

虚拟文件系统可分为逻辑文件系统和设备驱动程序。逻辑文件系统指 Linux 所支持的文件系统,如 ext2、fat 等,设备驱动程序指为每一种硬件控制器所编写的设备驱动程序模块。Linux 支持许多文件系统,“支持”的意思是可以把载有这些文件系统的磁盘分区或者其他的设备直接连接到 Linux 文件系统中。一般情况下,如果要连接一个不是 ext2 的文件系统,需要在 mount 命令行中用 -t 指令明确地给出文件系统的类型。Linux 系统目前支持的文件系统类型主要有:

- minix:最早的 MINIX 系统的文件系统。
- ext2:Linux 的标准文件系统,它还有一个比较早的形式即 ext1,目前已经不用。

- msdos:这就是标准的 MSDOS 文件系统。
- umsdos:这是一个特殊的文件系统实现,可以用 MSDOS 来存储类似 UNIX 的长文件名文件。
- vfat:这是 Windows 95/98 使用的文件系统,支持 Windows 95 长文件名。
- iso9660:CD-ROM 的标准文件系统。
- hpfs:OS/2 用的文件系统。
- ntfs:Windows NT 4.0 用的文件系统。
- ufs:BSD 用的文件系统。
- sysv:System V 系列的一些 UNIX 使用的文件系统。

提示:Ext2 和 ISO9660 通常不需要使用类型说明,而 msdos/win95 文件系统则常常需要明确地说明其为 MS-DOS 或者 vfat。例如,在/dev/hdb2 上装有一个 Windows 95 文件系统,那么,将它连结到/mnt/win95 的命令是 `mount /dev/hdb2 /mnt/win95 -t vfat`。

4. 网络接口(NET)

提供对各种网络标准的存取和各种网络硬件的支持。网络接口可分为网络协议和网络驱动程序两部分。网络协议部分负责实现每一种可能的网络传输协议,网络设备驱动程序负责与硬件设备进行通信,每一种可能的硬件设备都有相应的设备驱动程序。

5. 进程间通信(IPC)

它的作用是支持进程间各种通信机制。常用的进程间通信机制有:管道、有名管道(FIFO)、SYS V 机制、网络 socket 方式和全双工管道。其中,SYS V 机制包括消息队列、信号量和共享内存。

这些知识在后面的嵌入式系统的编程中将会大量用到,在此先建立一个整体的概念。

从图 2-2 可以看出,处于中心位置的是进程调度,所有其他的子系统都依赖于它,因为每个子系统都需要挂起或恢复进程。一般情况下,当一个进程等待硬件操作完成时,它被挂起;当操作真正完成时,进程被恢复执行。例如,当一个进程通过网络发送一条消息时,网络接口需要挂起发送进程,直到硬件成功地完成消息的发送,当消息被发送出去以后,网络接口给进程返回一个代码,表示操作的成功或失败。其他子系统(内存管理、虚拟文件系统及进程间通信)以相似的理由依赖于进程调度。

各个子系统之间的依赖关系如下:

- (1)进程调度与内存管理之间的关系:这两个子系统互相依赖。在多道程序环境下,程序要运行必须为之创建进程,而创建进程的第一件事,就是要将程序和数据装入内存。
- (2)进程间通信与内存管理的关系:进程间通信子系统要依赖内存管理支持共享内存通信机制。这种机制允许两个进程除了拥有自己的私有内存外,还可存取共同的内存区域。
- (3)虚拟文件系统与网络接口之间的关系:虚拟文件系统利用网络接口支持网络文件系统(NFS),也利用内存管理支持 RAMDISK 设备。

(4) 内存管理与虚拟文件系统之间的关系: 内存管理利用虚拟文件系统支持交换, 交换进程定期地由调度程序调度, 这也是内存管理依赖于进程调度的唯一原因。当一个进程存取的内存映射被换出时, 内存管理向文件系统发出请求, 同时, 挂起当前正在运行的进程。

除了图 2-2 中所显示的依赖关系以外, 内核中所有子系统还要依赖一些共同的资源, 但在图中并没有显示出来。这些资源包括所有子系统都用到的过程, 例如, 分配和释放内存空间的过程、打印警告或错误信息的过程、系统的调试例程等等。

2.3.3 系统数据结构

在 Linux 内核的实现中, 有一些数据结构使用频度较高, 它们是:

1. task-struct

Linux 内核利用一个数据结构(task-struct)代表一个进程, 代表进程的数据结构指针形成了一个 task 数组(Linux 中, 任务和进程是两个相同的术语), 这种指针数组有时也称为指针向量。这个数组的大小由 NR-TASKS 决定(默认为 512), 表明 Linux 系统中最多能同时运行的进程数目。当建立新进程的时候, Linux 为新的进程分配一个 task-struct 结构, 然后将指针保存在 task 数组中。调度程序一直维护着一个 current 指针, 它指向当前正在运行的进程。

2. mm-struct

每个进程的虚拟内存由一个 mm-struct 结构代表。该结构中实际包含了当前执行映像的有关信息, 并且包含了一组指向 vm-area-struct 结构的指针, vm-area-struct 结构描述了虚拟内存的一个区域。

3. inode

虚拟文件系统(VFS)中的文件、目录等均由对应的索引节点(inode)代表。每个 VFS 索引节点中的内容由文件系统专属的例程提供。VFS 索引节点只存在于内核内存中, 实际保存于 VFS 的索引节点高速缓存中。如果两个进程用相同的文件打开, 则可以共享 inode 数据结构, 这种共享是通过两个进程中的数据块指向相同的 inode 而完成的。

2.3.4 子系统的结构

如前所述, Linux 是由五个主要的子系统所构成的。在下面阐述各个子系统的结构。

1. 进程调度

进程调度子系统的目的是控制对计算机 CPU 的访问。“对 CPU 的访问”包括了用户进程的访问和内核其他子系统的访问。进程调度是操作系统的核心, 它的功能是:

- 允许进程建立自己的新拷贝;



- 决定哪一个进程将占用 CPU；
- 发送信号给用户进程；
- 接受中断并把它们发送到合适的内核子进程；
- 管理定时器件；
- 支持动态装入模块,这些模块代表着内核启动后所增加的内核功能,这些可装入的模块将由虚拟文件系统和网络接口所使用。

调度程序可以分为三个模块：

- 调度策略模块：负责判定哪个进程对 CPU 有访问的权利。应该使各个进程有平等的权利来访问 CPU。
- 系统结构相关的调度模块：一般使用公共接口设计而成,从而可以抽象出特定的计算机细节。这些模块负责与计算机 CPU 的通信,来中断或者恢复计算机进程的执行。
- 独立于系统的模块：负责与策略模块通信,来确定接下来执行哪个进程,然后调用系统结构相关的模块来恢复适当的进程。

注意：在嵌入式系统的原理中,会看到在嵌入式系统中往往要保证某个或某组进程较其他的进程有更高的优先级。

进程调用提供了两级接口：第一,它提供用户进程可以调用的有限系统的接口。第二,它为内核的其他子系统提供丰富的接口。进程只能够通过拷贝已经存在的进程来创建其他的进程。当系统刚启动时,Linux 只有一个正在执行的进程——init,这个进程通过 fork()系统调用来产生其他的进程。系统提供了几个例程来处理可装入的模块。例如,create_module()系统调用将分配足够的内存装入一个模块,并初始化模块的数据结构。init_module()系统调用从磁盘装入模块并激活它,而 delete_module()卸载一个正在运行的模块。

进程调度程序通过维护一个叫做 task_struct 的数据结构来管理系统的进程。task_struct 的每一个项对应一个活跃的进程。它可以恢复或停止进程的执行。同时还包括了一些系统的信息,如状态记录等。在整个内核中都要用到这个结构。

2. 内存管理子系统

内存管理子系统负责控制进程对系统硬件内存资源的访问。这是通过硬件内存管理来实现的。该系统提供进程对内存的应用与计算机物理内存间的映射。内存管理子系统为每一个进程都维护这样一个映射关系。这样,两个进程就可以访问同一个虚拟的内存地址,而实际使用的是不同的物理内存地址。内存管理子系统还支持交换,它把暂时不用的内存页调出内存,使计算机获得比实际内存更多的虚拟内存。

内存管理提供的功能有：

- 扩大地址空间；
- 进程保护；
- 内存映射；

- 公平的物理内存分配;
- 共享虚拟内存。

内存管理提供了两级接口:用户使用系统调用的接口以及为其他内核子系统所用的接口。系统调用接口有 `malloc()/free()`、`mmap()/munmap()` 等,而内核的接口有 `kmalloc()/kfree()`、`verify_area()` 等,在此不一一叙述,各位可以参考相应的资料。

3. 虚拟文件系统

虚拟文件系统的目的是为了提供一个在硬件设备上统一的数据视图。它需要支持很多不同的逻辑文件系统和很多不同的硬件设备。计算机中所有的设备都是通过通用的设备驱动程序接口来表示的。由于存在虚拟文件系统,更使得计算机可以在任意的物理设备上挂载一个任意逻辑文件系统的集合。Linux 采用了设备驱动层来给所有的物理设备提供一个统一的接口,而虚拟文件系统(VFS)给所有的逻辑文件系统提供统一的接口。

Linux 的内核有三种类型的设备驱动程序:字符型、块型和网络型。与文件系统相关的两类设备是字符型(如磁带、鼠标)和块型设备(如磁盘)。从本质上讲,设备驱动程序实际上是处理或操作硬件控制器的软件,是在内核中有高级特权的、驻留内存的、可共享的底层硬件处理程序。

所有的逻辑文件系统都被装载在虚拟文件系统的—个装载点上。任何时候,一个逻辑设备仅仅支持一种逻辑文件系统。但磁盘在分区后,单个的物理磁盘就被划分为了多个逻辑分区,每个分区上可以存在一个文件系统。例如在自己的 PC 上装载了 Windows 98 和 Linux 时,就是这种情况。Linux 的这种特性给用户提供了很大的灵活性。Linux 使用索引节点来表示一个文件或块设备。每个索引节点都包含了位置信息,以指定文件块在物理设备上的位置。索引节点还存储了指向逻辑文件系统中进程的指针和将要执行所需读写操作驱动设备程序的指针。

虚拟文件系统的另外一个主要功能是动态的装载模块。这种“要时才装入”的功能使用户尽可能的把内核缩小,当真正使用时才装载所需设备的驱动程序和文件系统。

文件子系统的主要数据结构有:

- `super_block`:即超级块;
- `inode`:即磁盘文件在内存的数据结构;
- `file`:表示由特定的进程打开的文件。

4. 网络接口系统结构

网络接口系统允许 Linux 系统与其他系统相连接。网络接口系统将硬件设备和网络协议在实现上的细节抽象掉了。Linux 的网络系统支持机器间的网络连接和 sockets 通信模型。在 Linux 中实现了两种类型的 socket:BSD socket 和 INET sockets,见图 2-3。

Linux 的网络实现就是以 4.3 BSD 为模型的,它支持 BSD sockets(及一些扩展)和所有的 TCP/IP 网络。Linux 选用这个编程接口是因为它很流行,并且有助于应用程序从 Linux 平台移植到其他 UNIX 平台。

Linux 下的 TCP/IP 网络协议栈的各层之间是通过一系列互相连接层的软件来实现



超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

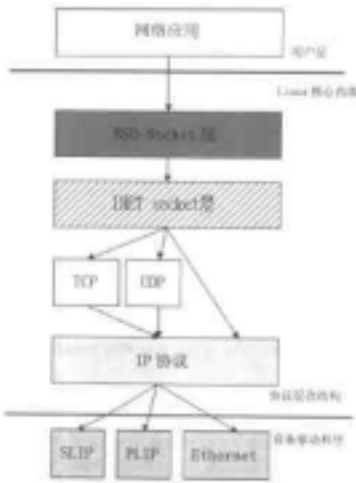


图 2-3 Linux 网络层的结构

Internet 地址族的。大家可能都知道,网络系统为不同的通信模型和服务质量提供了 TCP 和 UDP 两种传输协议。它们都是基于 IP 协议的。IP 协议则位于设备驱动程序之上。驱动程序则提供了三种不同类型的连接:串行线路连接(SLIP)、并行线路连接(PLIP)以及以太网连接。地址的解析协议则位于 IP 和以太网驱动程序之间,它的作用是把逻辑地址翻译成物理以太网地址。BSD socket 层由专门用来处理 BSD socket 的通用套接字管理软件来处理,它由 INET socket 层来支持。INET socket 为基于 IP 的协议 TCP 和 UDP 管理传输端点。用户数据报协议(UDP)是一个无连接协议,而传输控制协议(TCP)是一个可靠的端对端协议。传输 UDP 包的时候,Linux 不知道也不关心它们是否安全到达了目的地。TCP 则不同,在 TCP 连接的两端都需要加上一个编号以保证传输的数据被正确接收。在 IP 层,实现了 Internet 协议的代码。这些代码要给传输的数据加上一个 IP 头,并且知道如何把传入的 IP 包送给 TCP 或者 UDP 协议。在 IP 层以下,就是网络设备来支持所有的 Linux 网络工作,如 PLIP、SLIP 和以太网。不过要注意,这些网络设备不一定是物理设备,可以使用像 Loopback 一样的虚拟网络设备,纯粹是用软件来实现的。

网络子系统与其他子系统的关系如图 2-4 所示。

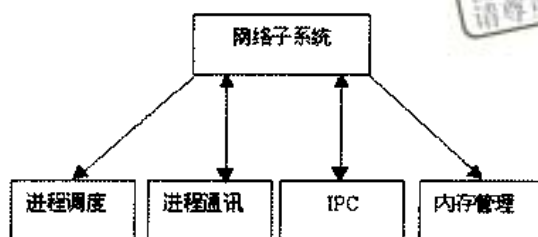


图 2-4 网络子系统与其他子系统的依赖关系

5. 进程通信子系统

在诸如网络程序设计中经常要用到进程之间的通信。为了使进程在同一项任务上协调工作,彼此间的通信是不可少的。Linux 支持很多种进程间通信的方式,概括起来有:

(1) 信号

系统用信号来通知一个或多个进程异步的发生,如键盘的按下、记时器记下某个特定事件的时间。也被系统用来处理某种严重的错误。

(2) 管道

简单的讲,管道就是连接一个程序输出和另外一个程序输入的单向通道,它在 UNIX/Linux 中用得极为广泛。

(3) 文件和记录锁定

它们是系统为共享资源而提供的相互排斥性的保障。

(4) 消息队列

消息队列就是在系统内核中保存的一个用来保存消息的队列。

(5) 信号量

信号量简单地讲就是用来控制多个进程对共享资源使用的计数器。它经常被用做一种锁定保护机制,当某个进程在对资源进行操作时阻止其他进程对该资源的访问。

(6) 共享内存

共享内存简单地讲就是被多个进程共享的内存。它在各种进程通信方法中是最快的。因为它将信息直接地映射入内存,省去了其他 IPC 方法的中间步骤。

进程通信在子系统间的关系如图 2-5 所示。

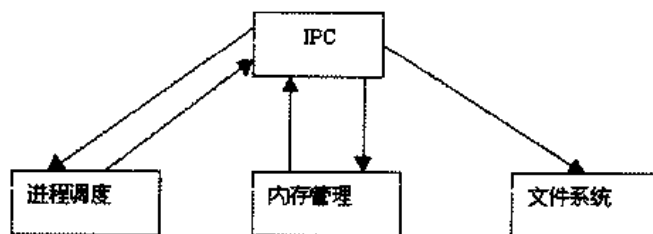


图 2-5 IPC 子系统与其他子系统的关系

Linux 的内核在概念上是由 5 个子系统所组成的,通过函数调用和共享数据结构,这些子系统间可以相互通信。由于 Linux 的内核是操作系统的灵魂,所以它将是本书讨论的重点。

2.4 Linux 的基本指令

鉴于嵌入式 Linux 的开发通常是采用宿主机——目标机的形式来工作的。所以对于刚刚接触 Linux 并有心了解嵌入式 Linux 的新手来讲,有必要对所要用到的 Linux 基本指令做一个大概的介绍。如果已经对 Linux 的基本指令体系有所掌握,那么这小节也可跳过不看。

2.4.1 Shell

Shell 既是 Linux 下面的命令解释器,用来接受并执行命令(包括运行命令文件和执进程),又是环绕在内核外层的操作系统和用户间的接口(shell 即“壳”之意)。

在 Linux 下,常用的 shell(shell 是 Linux 下的“命令解释器”)是:

- Bourne shell, /bin/sh
- Cshell, /bin/csh
- Kornshell, /bin/ksh

系统可以判断当前采用的是哪一个 shell。使用最多的是 Bourne shell,几乎每一个现存的 Linux 系统都提供 Bourne shell。

2.4.2 Linux 命令的使用说明

读者在使用没有列出的命令时,可以使用“-help”参数查询该命令所附的在线帮助,或者利用 man 命令来查询更详细的使用说明。

1. at

at, atq, atrm:安排、检查、删除队列中的工作。

at [-V] [-q 队列][-f 文件名] [-mldbv] 时间

at -c 作业 [作业...]

atq [-V][-q 队列][-v]

atrm [-V] 作业 [作业...]

at 在设定的时间执行作业。

atq 列出用户排在队列中的作业,如果是超级用户,则列出队列中的所有工作。atrm 删除队列中的作业。

范例

at -f work 4pm + 3 days

在三天后下午 4 点执行文件 work 中的作业。

at -f work 10am Jul 31

在七月 31 日上午 10 点执行文件 work 中的作业。



2. cat

功能:连接文件并打印到标准输出。cat 是 CATenate 的缩写,常常用来显示文件,类似于 DOS 下的 TYPE 命令。

范例

```
[root@zou /]# cat -b -E .lessrc 显示文件 .lessrc 的内容
```

```
1 back-line $ $
```

```
3 forw-scroll $
```

```
4 back-scroll $
```

```
5 goto-line $
```

```
6 goto-end $
```

```
[root@zou /]# cat myfile1 myfile2 >tmp 将文件 myfile1,
myfile2 连结起来输出到文件 tmp
```

注意:“>”是输出重定向符号,用来改变命令的输出。-b 参数表示从第一行开始记数,计算所有非空行的输出。-E 表示在每行结尾加上字符‘\$’。

3. cd

功能:改变当前目录。

格式:cd 目录名。

进入另外一个用户的目录只要 cd ~用户名即可。

注意:在此时要有根用户的权利。

范例

```
[root@zou /]# cd id1 进入 id1 目录
```

```
[root@zou /]# cd ~liyan 进入用户 liyan 的目录
```

```
[root@zou /]# cd / 回到根目录
```

4. chgrp

功能:改变文件的组。

其中,要改变的组可以是组号对应的数字,也可以是/etc/group 文件中的组名。

文件名:以空格分开的要改变组所有权的文件列表,支持通配符。如果用户不是该文件的属主或超级用户,则不能改变该文件的组。

范例

```
[root@zou /]# chgrp -R book /opt/local/book/*.*
```

改变/opt/local/book/及其子目录下的所有文件的组为 book。

5. chmod

功能:改变文件保护。文件保护是用来控制用户对文件的访问权的,有三个安全级别:所有者级别、组访问级别和其他用户访问级别。在这三个级别中,又有三种权限:读

(r)、写(w)和执行(x)。用户可用 `ls -lg` 来观看某一文件的所属 group。对于文件来说,读权限意味着可以看文件的内容,写文件权可以修改或删除文件,执行权限则可以执行它(类似于 DOS 下的 EXE、COM 和 BAT 文件)。对于目录来说,读权限意味着可以查看目录下的内容,写权限意味着能在目录下建立新文件,并可以从目录中删除文件,执行权限意味着可以从一个目录转变到另一个目录。

使用的格式为:[ugoa...][[+ - =][rwxXstugo...]...][, ...]

“ugoa”控制哪些用户对该文件的权限将被改变:“u”表示文件的所有者,“g”表示与文件所有者同组的用户,“o”表示其他用户,“a”表示所有用户。

操作符“+”使得用户选择的权限被追加到每个目标文件。操作符“-”使得这些权限被撤销。“=”使得目标文件只具有这些权限。

“rwxXstugo”选择新的属性。“r”表示读权限。“w”表示写权限。“x”表示执行权(或对目录的访问权)。“X”表示只有目标文件对某些用户是可执行的或该目标文件是目录时才追加 x 属性,同时设定用户或组 ID。“s”表示同时设定用户或组的 ID。“t”表示保存程序的文本到交换设备上。“u”表示目标文件属主。“g”表示目标文件属主所在的组。“o”表示其他用户。如果用数字来表示属性,则“0”没有权限,“1”为执行权,“2”读权,“4”写权,然后将其相加,所以数字属性的格式应为 3 个。从 0 到 7 的八进制数其顺序是“u”“g”“o”。文件名以空格分开的要改变权限的文件列表,支持通配。

范例

```
[root@zou /]# chmod a+x destfile 使所有用户对文件 destfile 有读写执行权。
```

```
[root@zou /]# chmod 644 destfile 使所有用户可以读文件 destfile,只有属主才能改变。
```

6. chown

功能:改变文件的属主和组。

```
Chown [Tcfv][recursive][--changes][--help][--version][--silent][--quiet][--verbose][用户][. .][组] 文件名
```

用户:可以是用户名或用户 id。

组:可以是组名或组的 id。

文件名:以空格分开的要改变权限的文件列表,支持通配符。

范例

```
[root@zou /]# chown tlc:book destfile
```

将文件 destfile 的属主改成 tlc,组改成 book。

7. clear

功能:清除屏幕(类似于 DOS 的 cls)。

范例

```
[root@zou /]# clear 清除屏幕,提示符被移动到左上角。
```

8. cp

功能:拷贝文件。

cp [options] 源文件 目标文件 cp [options] 源文件... 目标目录

-f(--force):删除已存在的目标文件。

-i(--interactive):在删除已存在的目标文件时给出提示。

-R(--recursive):整目录拷贝。

--help:在标准输出上输出帮助信息并退出。

--version:在标准输出上输出版本信息并退出。

范例

```
[root@zou /]# cp sourcefile destfile 拷贝文件 sourcefile 到文件 destfile。
```

```
[root@zou /]# cp * /tmp 拷贝当前目录下所有文件到/tmp 目录。
```

9. dd

功能:拷贝一个文件(并同时转化它)。

范例

将文件 sourcefile 拷贝到文件 destfile。

```
[root@zou/]# dd if= sourcefile of= destfile
```

```
0 + 1 records in
```

```
0 + 1 records out
```

10. df

功能:报告磁盘剩余空间。

```
Df [-aikPv][--t fstype][--x fstype][--all][--inodes][--type=fstype][--exclude-type=fstype][--kilobytes][--portability][--print-type][--help][--version][file-name...]
```

-a, --all:列出 block 为零的文件系统,默认为不列出。

-I, --inodes:用 inode 使用状况来代替 block 使用状况。

-k, --kilobytes:以 1K 为单位来输出 block。

-P, --portability:使用 Posix 格式输出。

-T, --print-type:输出每个文件系统的类型。

-t, --type=fstype:只输出不在 fstype 中的类型的文件。

--help:在标准输出上输出帮助信息并退出。

--version:在标准输出上输出版本信息并退出。

范例

```
[root@zou /]# df -a -T
```

```
Filesystem Type 1024-blocks Used Available Capacity Mounted on
/dev/hda1 ext2 14 136751 71% /
none proc 0 0 0 0% /proc
```

```
/dev/hda3 ext2 267 2222699 5% /home  
/dev/hda2 ext2 995147 49603 894136 5% /usr/local
```

11. du

功能:报告磁盘空间使用情况。

范例

```
[root@zou /]# du  
1 ./X11-unix  
15274 ./data  
4 ./id4  
15293 .
```

12. file

功能:探测文件类型。

范例

```
[root@zou/]# > file *  
destfile: ASCII text  
elm.rc.OLD: English text  
portnum: empty  
rc.inet1.OLD: Bourne shell script text
```

13. find

功能:用来在大量目录中搜寻特定文件。

find [路径...] [匹配表达式]

范例

```
[root@zou /]# find ./ -name "passwd" -print  
./usr/bin/passwd  
./home/ftp/etc/passwd  
./etc/passwd
```

14. grep、egrep 和 fgrep

功能:在文件中搜寻匹配的行并输出。

范例

在文件 services 中查找含有 ftp 的行。

```
[root@zou /]# grep ftp services  
ftp 21/tcp  
tftp 69/udp  
sftp 115/tcp
```



15. gzip、gunzip 和 zcat

功能:压缩或展开文件。

范例

```
[root@zou /] # gzip -v sourcefile
sourcefile: 15.2% -- replaced with sourcefile.gz
```



16. kill

功能:中止一个进程。

kill [-s 信号 | -p] [-a] 进程号 ...

kill -l [信号]

kill 向指定的进程发出特定的信号,如果没有指定信号则送出 TERM 信号,TERM 信号将杀死没有捕捉到这个信号的进程。对于某些进程可能要使用 Kill(9)信号强制杀死。例如:kill -9 11721,将强制杀死进程 11721。大多数 SHELL 内建 kill 命令。

范例

杀掉进程 11721

```
[root@zou /] # ps
PID    TTY STAT TIME COMMAND
11668  p1  S   0:00 -tcsh
11721  p1  T   0:00  cat
11737  p1  R   0:00  ps
[root@zou /] # kill 11721
[1] Terminated cat
```

17. last

功能:显示过去多少个用户或终端登录到本机器。

last [-数目] [-f 文件名] [-t tty] [-h 节点名] [-i IP 地址] [-l][-y] [用户名...]

范例

显示过去 3 次用户 fangh 登录的情况:

```
[root@zou /] # last -3 fangh
fangh tty1 csun01.ihep.ac.c Tue Aug 26 18:46 still logged in
fangh tty2 csun01.ihep.ac.c Mon Aug 25 22:32 - 23:14 (00:41)
fangh tty2 csun01.ihep.ac.c Mon Aug 25 19:58 - 21:59 (02:01)
```

18. less

功能:相对于 more,用来按页显示文件。

范例

显示 test 文件

```
[root@zou/] # less test
```



it is only a test1
<END>

19. ln

功能：在文件间建立连接。

ln [参数] 源文件 [目标文件] ln [参数] 源文件... directory

☞ 注意：对链接文件做改变属性的动作是没有意义的，因为只有它们链接到的文件属性才是文件的真正属性。

范例

将文件 sourcefile 连接到文件 test

```
[root@zou/]# ln -s sourcefile test
```

```
[root@zou/]# ls -la test
```

```
rw-rw-rw- 1 fangh users 10 Aug 26 20:36 test -> sourcefile
```

20. ls、dir 和 vdir

功能：列出目录下的文件(类似于 DOS 下的 DIR 命令)。

范例

列出当前目录下的所有文件

```
[root@zou/]# ls -la
```

total 6

```
drwxr-x-- 2 fangh users 1026 20:52 ./
```

```
drwxr-xr-x 19 root root 1026 21:09 ../
```

```
-rw-r--r- 1 fangh users 15 Aug 21 21:57 .bash_history
```

```
-rw-r--r- 1 fangh users 30 20:41 .less
```

```
-rw-r--r- 1 fangh users 115 19:58 .lessrc
```

```
rw-r--r- 1 fangh users 72 Aug 24 18:43 sourcefile
```

```
lrwxrwxrwx 1 fangh users 10 Aug 26 20:36 test -> sourcefile
```

21. man

功能：显示具有一定格式的在线手册。

man 对新手和老手来说都是非常有用的一个工具，用于快速查询命令和程序的使用方法和参数。编程人员也可以用来查询 C 函数的用法。对于 'rn(1)' 或 'ctime(3)' 这样的输出，其括弧中的数字是指 Linux 手册中该文件所在的章节。当打 man 3 ctime 时，表示是要查阅在第 3 节中的 ctime 的内容。下面是常见的 Linux 手册的分类：

- 用户命令；
- 系统调用；
- 库函数；
- 设备和设备驱动程序；

- 文件格式;
- 游戏;
- 有用的杂类,如宏命令包;
- 系统维护和管理命令。

范例

查询 ls 的用法

```
[root@zou/]# man ls
```

22. mkdir

功能:建立目录(同 DOS 下的 md)。

23. more

功能:在终端上按页观看文件的过滤器。more 的功能没有 less 那么强大,而且 less 还提供了对 more 的模拟。不过一般用户可能更习惯于使用同 DOS 环境下相似的 more。

范例

显示文件/etc/group 并搜寻字符串 bbs

```
[root@zou/]# more + /bbs /etc/group
```

```
... skipping
```

```
users::100:games
```

```
nogroup::-2:
```

```
bbs:x:99:bbs,bbsroot,bbsuser
```

24. mv

功能:将文件改名,也可以移动文件和目录。

```
mv [参数] 源文件 目标文件
```

```
mv [参数] 源文件列表(支持通配符) 目标目录
```

范例

```
[root@zou/]# mv -v sourcefile destfile
```

```
sourcefile -> destfile
```

```
[root@zou/]#
```

25. passwd

功能:设置用户的密码。

```
passwd [-f|-s] [用户名]
```

```
passwd [-g] [-r|R] 组名
```

```
passwd [-x max] [-n min] [-w warn] [-i inact] 用户名
```

```
passwd {-l|-u|-d|-S} 用户名
```

用户可以用 passwd 这个命令更改自己的登录密码。一般用户只能更改自己的密码,超级用户可以更改其他所有用户的密码。超级用户和组的管理者可以更改组的密码,还可以



用这个命令来更改用户的其他信息,如用户的全名、登录 shell、密码失效的时间间隔等等。

范例

更改密码

```
[root@zou/]# passwd
Changing password for fangh
Old password:oldpass (密码并不显示出来)
Enter the new password (minimum of 5, maximum of 8 characters)
Please use a combination of upper and lower case letters and numbers.
New password:newpass
Re-enter new password:newpass
Password changed.
```

[root@zou/]# passwd -S 显示用户情况

```
fangh P 09/05/97 0 99999 7 -1
```

26. ps

功能: 查看进程状态。Ps 结果往往有很多栏,如命令:ps -al。

```
[root@zou /home] $ ps -al
```

```
[root@zou /home] $ ps -al
```

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
100	S	0	563	551	0	60	0	-	440	wait4	tty1	00:00:00	bash
100	S	0	600	563	0	60	0	-	329	wait4	tty1	00:00:00	man
000	S	0	610	600	0	61	0	-	419	wait4	tty1	00:00:00	sh
000	S	0	612	610	0	61	0	-	595	read_c	tty1	00:00:00	less
000	S	501	614	613	0	67	0	-	439	wait4	pts/0	00:00:00	bash
000	R	501	620	614	0	71	0	-	634	-	pts/0	00:00:00	ps

下面解释各栏的意义:

- FLAGS:长格式的 F 域。
- UID:用户的 ID。
- PID:进程的 ID。
- PPID:父进程的 ID。
- PRI:进程优先级。
- NI:标准 UNIX 的优先级。
- SIZE:虚拟内存的大小。
- WCHAN:进程等待的内核事件。

范例

显示当前进程:

```
[root@zou/]# ps -c
```

```
PID TTY STAT TIME COMMAND
8724 p4 S 0:00 tcsh
```

```
8876 p6 R 0:00 ps
```

27. pwd

功能:显示正在工作或当前目录名。

范例

列出当前工作目录

```
[root@syzoul /home] $ pwd
/home
```

28. reset

功能:将终端复位。

在使用终端的过程中,有时会发现屏幕的字符“花”掉了,使用 reset 就可以恢复。reset 调用 tput 函数,向终端发出复位信号。如果屏幕被 Ctrl+S 锁住了,可以用 Ctrl+Q 来解除锁定。

29. rm

功能:删除文件或目录。

☞ 注意:在 Linux 下如果产生类似于“-f”这种文件名的文件,删除是先跟“-”选项。这表明后面所跟的“-f”不是选项而是文件名,例如“rm -f file”是删除文件“-file”,在删除文件名中包含特殊字符的文件时,可以用“\ + 特殊字符”,或用“”将文件名括起来,例如要删除文件“this is a test”可以用下列命令:rm this \ is \ a \ test 或者 rm "this is a test"。另外要特别注意,使用 rm -rf * 这个命令,如果用户是超级用户,并且在 root 下使用此命令,那么系统的所有文件将被删除。而在 UNIX/Linux 系统下恢复文件几乎是一件不可能的事情,因而要谨慎对待这个命令。另外建议用户将 rm 定义为“rm -i”,并加入到用户的登录文件中。

范例

删除文件 destfile

```
[root@zou/]# rm -v destfile
destfile
```

```
[root@zou/]# rmdir
rmdir - 删除空的目录。
```

30. su

功能:改变用户的 ID 或成为超级用户。

31. tail

功能:显示文件的最后一部分。tail 缺省显示文件名列表中的每个文件的最后十行。如果没有文件名或文件名为“-”,则它从标准输入中读取文件。如果有多个文件则它会



在文件前面加上“==> 文件名<==”以便区分。

范例

显示文件/etc/DIR_COLORS 的最后三行：

```
[root@zou/]# tail -v -n 3 /etc/DIR_COLORS
```

32. tar

功能：GNU 版的文件打包备份的工具。

范例

当前目录下所有.txt 文件打包并压缩到压缩文件 bak.tar.gz

```
[root@zou/]# tar czvf bak.tar.gz ./ * .txt
```

将目录./sec 打包到压缩文件 sec.tar.gz

```
[root@zou/]# tar czvf sec.tar.gz ./sec
```

33. top

功能：显示系统的最高进程。

这个命令可以即时显示当前系统最占 CPU 时间的进程，并提供一个交互的界面让用户可以观察系统进程情况。它可以按照 CPU 使用情况、占内存大小、运行的时间来对进程排序，这是系统管理的一项必不可少的工具。

34. touch

功能：改变文件的时间参数。

它将文件的访问时间、修改时间设置为系统的当前时间。如果该文件不存在则建立一个空的新文件。

范例

将当前目录下的文件的时间参数修改为当前时间：

```
[root@zou/]# touch *
```

35. mount 和 umount

mount 功能：装载一个文件系统。

umount 功能：卸下一个文件系统。

范例

列出系统目前 mount 的文件系统：

```
[root@zou/]# mount
```

```
/dev/hda1 on / type ext2 (rw)
```

```
none on /proc type proc (rw)
```

```
/dev/hda3 on /home type ext2 (rw)
```

```
/dev/hda2 on /usr/local type ext2 (rw)
```

加载光驱：



```
[root@zou/]# mount /dev/hdb /mnt
```

36. unarj、unzip 和 lha

功能:解开压缩文件。

在 DOS 下最常用的压缩软件有 arj、pkzip 和早期的 lha,它们产生的压缩文件如何在 Linux 下展开呢?这就要用到 unarj、unzip 和 lha 这三个工具软件了。其用法同其在 DOS 下的同类软件用法几乎一样。只是 unarj 和 lha 都是版权软件,是没有源码的,不属于 GNU。且 unarj 和 unzip 只能解压不能压缩。

37. vi

功能:一个功能强大的 UNIX 编辑器。

在介绍了前面 30 多条命令之后,再来看一个 UNIX 世界里最通用的全屏编辑器 vi。在所有的 UNIX/Linux 机器都提供该编辑器。vi 的原意是“visual”即可视编辑器,键入的会立即被显示出来。其强大的编辑功能可以同任何一个最新的编辑器相媲美,而且学会 vi 可以让用户在任何一台 UNIX 机器无论是 SUN、HP、AIX、SGI、Linux 或 FreeBSD 上,都可以得心应手地编辑文件。只要在命令行上键入 vi 就可以进入 vi 的编辑环境。vi 有两种状态,输入状态以及指令状态。用户在输入状态下可以输入文字资料,指令状态是用来执行打开文件、存档、离开 vi 等操作命令。执行 vi 后首先进入指令状态,此时输入的任何字符都作指令来处理。输入“vi 文件名”则 vi 自动装入文件或开始一个新文件,vi 屏幕的左方会出现波浪号“~”,代表本行为空行。

vi 有“编辑”和“命令”两种状态。在 vi 下,如果要从当前的编辑状态进入命令状态,只需要按“:”即可。进入输入状态,按“Esc”即可。vi 的退出是在命令状态下键入下列命令之一:q:结束编辑退出。如果想放弃编辑,用命令 q!。wq:保存当前的文件并退出。

vi 下的编辑状态的相应的指令有插入(a, i, o)、删除与恢复(x, dw, dd, d\$, dG)、修改(R, r, ~)、剪切与复制(yy, cc, p)等等诸多命令,在此不一一讲述了。

2.5 五脏俱全的嵌入式 Linux

2.5.1 嵌入式 Linux 的其他版本

在第 1 章,各位已经对嵌入式 Linux 有了一个感性的认识。嵌入式 Linux 虽然小,但是作为一个基本的计算机系统,它还是很齐全的。嵌入式 Linux 版本众多,除了第 1 章提到的那几种以外,还有如下的一些版本,现在简要介绍一下。

1. PocketLinux

由 Agenda 公司采用作为其新产品“VR3 PDA”的操作系统的嵌入式 Linux 版本。它可以提供跨操作系统构造的不同,构造统一、标准化和开放的信息通信基础结构的端到端

方案的完整平台。PocketLinux 资源框架开放,使普通的软件结构可以为所有用户提供一致的服务。PocketLinux 平台使用户的视线从设备、平台和网络上移开,由此引发了信息技术新时代的产生。在 PocketLinux 中,称之为用户化信息交换(CIE),即提供和访问为每个用户需求而定制的同步和“主题”信息的能力,而不管正在使用的设备是什么。

2. MidoriLinux

由 Transmeta 公司推出的 MobileLinux 操作系统。该系统代码开放,在 GUN 普通公共许可(GPL)下发布,可以在 midori.transmeta.com 上立即获得。该公司有个名为“MidoriLinux 计划”的计划。“MidoriLinux”这个名字来源于日文中的“绿色”这个词——midori,用来反映精力充沛的 Linux 操作系统的环保外观。

3. BlueCat

来自 LynuxWorks 的嵌入式 Linux。大名鼎鼎的“蓝猫”,不用多介绍了吧?

下面介绍 RT-Linux 和 uClinux。本书后面将有专门的章节来讲述如何在它们提供的平台下进行编程、开发的实例。

2.5.2 RT-Linux

1. NMT RT-Linux

NMT 是新墨西哥科技大学(New Mexico Technology)的英文缩写。这一套系统可以说是所有 real-time Linux 的鼻祖,目前已经发展到 3.0 版。这个系统是由 Victor Yodaiken 和他的学生 Michael Barabanov 所完成。这个系统的概念是“架空”Linux kernel,使它的实时进程得以尽快地被执行。

RT-Linux 是通过底层路线实现对 Linux 实时改造的产物,在 Linux 内核的下层实现了一个简单的实时内核。而 Linux 本身作为这个实时内核的优先级最低的任务,所有的实时任务的优先级都要高于 Linux 本身以及 Linux 下的一般任务。

目前,RT-Linux 已经得到了一些应用。NASA 用 RT-Linux 开发出来的数据采集系统采集 George 飓风中心的风速;好莱坞的电影制作人用 RT-Linux 作视频编辑工具;甚至在医学方面,还有用 RT-Linux 来控制心脏的跳动的例子。

RT-Linux 的性能是“硬”实时系统的性能。在一台 386 机器上,RT-Linux 从处理器检测到中断到中断处理程序开始工作不会超过 15 微秒。对一个周期性的任务,在 35 微秒内一定会执行。如果是普通的 Linux,一般是在 600 微秒内开始一个中断服务程序,对周期性的任务很可能会超过 20 毫秒(20 000 微秒)。这些实时性能参数与系统的负载无关,只取决于系统的硬件。比如在 PⅡ 350、64MB 内存的普通 PC 机上,最大的中断延迟不超过 1 微秒。这些数据已经接近了硬件的极限。RT-Linux 支持对称多处理器 SMP,当实时任务在多处理器系统中运行时,实时和非实时性能都会有较大的提高。

提示:POSIX 1003.1 对实时操作系统的定义是:实时操作系统必须有能在边界时间内提供所需级别服务的能力。按对延时的要求,可以将实时操作系统分为“软实时”和

“硬实时”两种。一般讲来,硬实时操作系统是指如果计算结果输出的时间超过了时限,将会引起灾难性后果的实时操作系统,如控制飞机发动机的实时操作系统。软实时操作系统是指如果计算时间超过了时限,虽然不会引起灾难性后果,但是应该尽可能满足时限的实时操作系统,如播放多媒体的操作系统。

RT-Linux 将 Linux 作为最低优先级别的执行线程运行。Linux 线程被做成完全可以抢先的,使实时线程和中断处理子进程永远不会被非实时操作所延迟。RT-Linux 中的实时线程可以通过共享内存或一个类似于文件的接口(FIFO 管道,即有名管道。它以一种特殊的设备文件形式存在于文件系统之中。它不仅有管道的通信功能,也具有普通文件的特点,解决了管道文件对外不可见性的问题)与 Linux 中的进程通信。这样,实时应用程序就能够利用 Linux 所有强大的、非实时的服务。这些服务包括:网络功能、图形功能、窗口系统、数据分析程序包、Linux 设备驱动程序以及标准的 POSIX API。例如,用 Perl 语言写一个在 X Window 中显示数据,对网络上传过来的命令作出响应,并从一个实时任务采集数据的程序是很容易的。又因为 RT-Linux 和作为其基础的 Linux 都是源代码开放的自由软件系统,功能强大、性能稳定、支持广泛,其应用的前景十分看好。

RT-Linux 中的实时任务并不是一个 Linux 的进程,而是一个 Linux 的可加载式核心模块。之所以要如此做的原因,是由于 Linux 是一个很大的系统,且在设计时并没有考虑实时需求。举个例子,仅一个 Linux 系统调用可能会花上超过 10ms 的时间。对有些像工业控制的应用而言,它们对时间的要求通常在 1ms 的等级上,Linux 根本无法满足这种需求。所以 NMT RT-Linux 采用一个比较简单的做法,不直接用 Linux 的任何功能,而把需要高度时间精确度的工作写成一个驱动程序的形式,然后直接用 PC 时序芯片所产生的中断调用这个驱动程序。如此一来,不管 Linux 系统调用的时间有多长都没有关系了。

从这个角度看,NMT RT-Linux 其实是一个实时驱动程序的构架,算不上是真正的 real-time Linux。但由于它出现得早,且其构架很符合自动控制的需求,所以使用者非常多,在自动控制领域应用十分广泛。

2. RTAI

RTAI 是 Real-Time Application Interface 的缩写。顾名思义知道它是一套可以用来写实时应用程序的界面。大致而言,RTAI 和 NMT RT-Linux 是相同的东西。它同样地架空了 Linux,而直接用可加载式核心模块来作为实时进程。

RTAI 和 NMT RT-Linux 最大的不同地方在于它非常小心地在 Linux 上定义了一组 RTHAL(Real-Time Hardware Abstraction Layer)。RTHAL 将 RTAI 需要在 Linux 中修改的部分定义成一组程序界面,RTAI 只使用这组界面和 Linux 沟通。这样做的好处在于可以直接修改 Linux 核心的程序代码直至最小,这使得将 RTHAL 移植到新版 Linux 的工作量减至最低。

RTAI 采取这种方法最大的原因在于,NMT RT-Linux 在由 2.0 版移植至 2.2 版的过程中遇到了问题,使基于 2.2 版核心的 NMT RT-Linux 一直无法完成。所以 Dipartimento di Ingegneria Aerospaziale Politecnico di Milano 的 Paolo Mantegazza 和他的同事们就决定自行做移植工作,但由于遇到 NMT RT-Linux 的困境,使他们体会到了必须采取上述的途径以解决将来可能再度面临的兼容性问题。于是 RTAI 便诞生了,它是一个比

NMT RT-Linux 更好的 NMT RT-Linux。

3. LXRT

由于 RTAI 无法直接使用 Linux 的系统调用,所以人们就设计了 LXRT,使用 RT-FIFO 将一个 RTAI 的实时内核模块和真正的 Linux 进程连接在一起,由这个进程做代理,为 Linux 系统调用,解决了这个问题。这就是 LXRT 产生的原因。

4. KURT

KURT 是由 Kansas 大学所创造的系统,它和 NMT RT-Linux 及 RTAI 有很大的不同,KURT 是第一个可以使用系统调用的 real-time Linux。但是它的实时进程的执行很容易受到其他非实时进程的影响。

2.5.3 uClinux

uClinux 是一种优秀的嵌入式 Linux 版本。在 uClinux 这个英文单词中,u 表示 Micro,小的意思,C 表示 Control,控制的意思,所以 uClinux 就是 Micro-Control-Linux,字面上的理解就是“针对微控制领域而设计的 Linux 系统”。由于本书讨论的应用程序设计不少是基于华恒公司(<http://www.hhcn.org>)的开发套件实现的,而开发套件的操作系统正是用的 uClinux,所以在此先提及一下,后面讲到开发时会进行详细阐述。

2.6 本章小结

在这章里,对 Linux 操作系统有了一个大概的了解。首先,学习了系统内核的构成、子系统的结构和系统重要的数据结构。然后,简要学习了系统操作的基本命令。最后,熟悉了一些重要的嵌入式 Linux。通过本章的学习,可以为后面的学习打下基础。



第3章 Linux 下的 C 语言编程入门

本章是迈向系统应用和开发的第一步。通过本章的学习,可以掌握 GCC 编译器的使用,并能通过 gdb 来调试程序,通过书写 makefile 文件来维护应用程序。在最后,通过分析一个 makefile 来巩固和结束本章的学习。

本章主要内容:

- C 语言和 Linux
- GCC 编译器
- 符号调试工具 gdb
- 使用 make 指令
- 书写简单的 makefile 文件
- 复杂的 makefile 文件的解读

3.1 C 语言和 Linux

3.1.1 C 语言的发展历史

C 语言是运用得极为广泛的一种现代计算机语言。它适合作为系统的描述性语言,即用来写系统软件,也适合写应用程序。在以前,操作系统等系统软件是采用汇编语言编写的(包括 UNIX 操作系统),由于汇编语言对硬件的严重依赖性,使程序的可读性、移植性很差。人们开始考虑采用高级语言。但是,一般的高级语言难以实现汇编语言的很多功能(例如,对内存地址的操作等),于是,C 语言出现了。C 语言是在 B 语言的基础上发展来的,它的根源可以追述到 ALGOL60。1972 年至 1973 年间,Bell 实验室的 D. M. Ritchie 在 B 语言的基础上设计出了 C 语言。以后,有人就用 C 语言对 UNIX 操作系统进行了改写,获得了成功。

随着 UNIX 的日益广泛应用,C 语言也迅速推广开来。从 1978 年起,C 语言先后移植到了各种大、中、小、微型计算机上。而以 1978 年发表的 UNIX 第 7 版中的 C 编译程序为基础,Brian W. Kernighan 和 Dennis M. Ritchie 发表了影响深远的名著《The C Programming》。此书中介绍的 C 语言后来成为广泛使用的 C 语言版本的基础,被称为标准 C。1983 年,美国国家标准化协会(ANSI)为 C 语言制定了新的标准,被称为 ANSI C。目前广泛流行的 C 编译系统基本上是以 ANSI C 87 为标准的,如 Microsoft C、Turbo C、

Quick C 等,它们之间还是存在着差异的,使用时要注意。

随着 C 语言的发展,为了满足对管理复杂程序的需要,1980 年,Bell 实验室的 Bjarne Stroustrup 开始对 C 进行改进和扩充。最初的成果被称为“带类的 C”,在经历了 3 次修订后,成为了目前的 C++ 语言。C++ 包含了整个 C,C 是建立 C++ 的基础。C++ 包括了 C 的全部特征、属性和优点,同时也添加了对面向对象编程(OOP)的完全支持。C++ 的出现,为 C 这门古老的语言带来了新的活力。

3.1.2 C 语言的特点

C 语言巨大的生命力来自于它不同于甚至优于其他语言的特点。C 语言的特点如下:

(1)语言简洁、紧凑,使用方便灵活

C 语言仅 32 个关键字,9 种控制语句,书写形式自由,压缩了一切不必要的成分。

(2)运算符和数据结构丰富

C 的运算符包含的范围很广,共 34 种,表达式类型多样化,可以实现在其他的高级语言中难以实现的运算。具有各种现代化的数据结构,可以实现复杂的运算。语法实现灵活,限制并不严格。

(3)C 语言允许直接访问物理地址,可以进行位操作,能实现汇编语言的大部分功能

因为 C 语言的这个特性,使它既是成功的系统描述语言,也是通用的程序设计语言。所以有人称之为“中级语言”。

(4)可移植性好

同汇编语言比,C 语言有很好的移植性,基本上用 C 语言编写的程序不用做什么修改就可以用于各种型号的计算机系统和操作系统中。

3.1.3 C 语言和 Linux

作为 UNIX 系统的克隆,Linux 和 C 语言有着天生的联系。作为一个操作系统,当然要支持用户在其上编写程序,而 C 语言在 Linux 上的成功应用就为用户和开发系统人员提供了强大的编程环境。其实,Linux 本身就是用 C 语言来写的。Linux 的创始人 Linus 盛赞 C 语言说:“……我开始用 C 语言编写程序,明显加快了速度……”。

同时,在 Linux 平台上,对 C/C++ 语言的支持也是十分完善的。例如 C 编译工具 gcc 和调试工具 gdb(还有 X Window 下的 xxgdb)等都是非常优秀和高效率的 C 编译与调试工具。在后面的小节中会详细讲述。在 Linux 的发行版中包含了很多软件开发工具,它们中的很多是用于 C 和 C++ 应用程序开发的。

3.1.4 C 语言和嵌入式系统的设计

在嵌入式系统的设计中,C(包括 C++)语言也是应用得最多的一种高级语言。C 语

言与硬件无关,并不包括输入输出语句,但是C语言需要通过输入输出语句同硬件打交道,而只有输入输出语句最底层的几个函数与硬件设计有关。换言之,只要程序设计人员提供给C编译器最基本的几个I/O函数,在程序设计中就没什么问题了。由于嵌入式系统本身存储器的原因不可能带很大的函数库,所以类似于scanf()、printf()等辅助函数就显得十分重要。

C语言在嵌入式系统中编写应用程序至少有以下一些好处:

(1)系统可以在其他的计算机上仿真

在硬件的设计完成后,软件开发的工作量则相当大。对于嵌入式系统的开发,比如在后面章节会讲到的基于uClinux系统的开发平台的应用程序开发,就是采用了在PC上用C语言编写应用程序并在PC上仿真的方法。这样,系统的软件开发就可以和硬件设计同时进行。

(2)应用程序有较好的可移植性

考虑到硬件产品的更新换代,用C语言写的应用程序往往可以直接移植,仅仅重新编译一下就可以了。

(3)便于程序的调试

应用程序的调试可以直接使用C语言,例如C语言中的printf()函数就是程序调试的有力工具。程序中的各个参数,包括一些中间变量都可以用这个函数在屏幕上显示。通过Linux下的gdb工具可以方便地进行调试。

(4)C语言库函数丰富。

C语言提供了很多有用的库函数,例如数学运算、码的转换、各种格式的输入和输出等,这为用户进行嵌入式开发提供了一个有力的手段。

了解了C在开发设计中的重要性后,下面就从一个实例入手,讲述GCC编译器的使用。随后将逐步深入,讲述gdb和makefile。最后讲述一个较为复杂的makefile。

3.2 GCC编译器的使用

先看下面的例子:

```
例 3-1 test3-1.c
#include <stdio.h>
main()
{char *str="I like embedded Linux!";
printf ("%s\n", str);
exit(0);
}
```

例3-1是一个极为简单的C程序。它的功能是将字符串“I like embedded linux!”打印出来,接下来用GCC编译。输入gcc -c test3-1.c,得到了目标文件test3-1.o。“-c”命令表示对文件进行编译和汇编,但并不连接。如果键入gcc -o test3-1 test3-1.o,那么将得到名为test3-1的可执行文件。其实,这两步可以一气呵成,直接键入命令:gcc -o test3-1

```
root@beagle ~# gcc -o test3-1.o
root@beagle ~#
root@beagle ~# gcc -o test3-1 test3-1.o
root@beagle ~#
root@beagle ~# gcc -o test3-1 test3-1.o
root@beagle ~#
root@beagle ~# gcc -o test3-1 test3-1.o
root@beagle ~#
root@beagle ~# gcc -o test3-1 test3-1.o
root@beagle ~#
root@beagle ~# gcc -o test3-1 test3-1.o
root@beagle ~#
```

以生成可执行文件了。各个步骤如图 3-1 所示,键入(当前的目录下执行)。



图 3-1 使用 GCC 编译器编译 test3-1.c

这仅仅是 GCC 的一个简单应用,也许会觉得基于命令行方式的编译器比不上如 VC 之类的建成开发环境。的确,GCC 的界面有待改进之处,但是一旦用得熟练之后就会感到,GCC 的效率是如此之高!

下面开始学习 GCC。

3.2.1 GNU C 编译器

GNU C 编译器(即 GCC)是一个全功能的 ANSI C 兼容编译器。如果熟悉其他操作系统或硬件平台上的一种 C 编译器,能很快掌握 GCC。在 shell 的提示符号下键入 `gcc -v`,屏幕上就会显示出正在使用的 GCC 的版本。同时这也是一个相当可靠的方法,可以确定现在所用的是 ELF 或是 a.out 格式。

本小节将介绍如何使用 GCC 和 GCC 编译器最常用的一些选项。

1. 使用 GCC

GCC 是基于命令行的,使用时通常后跟一些选项和文件名。GCC 命令的基本用法如下:

```
gcc [options] [filenames]
```

命令行选项(即编译选项 options)指定的操作将对命令行上每个给出的文件(filenames)执行。实际上,在 gcc 后面可以跟上很多的选项。下面列出了一些常用的选项。它们的使用频率相当高,希望大家能掌握。

2. GCC 选项

(1) 编译选项

GCC 有超过 100 个的编译选项可用。这些选项中的许多可能永远都不会用到,但一些主要的选项将会频繁用到。很多的 GCC 选项包括一个以上的字符,因此必须为每个选项指定各自的连字符,并且就像大多数 Linux 命令一样,不能在一个单独的连字符后跟一组选项。例如,下面的两个命令是不同的:

```
gcc -p -g test.c
```

```
gcc -pg test.c
```

同样“-dr”参数和“-d -r”也是截然不同的。


第一条命令告诉 GCC 编译 test.c 时使用 prof 命令,即为本文件建立剖析信息并且把调试信息加入到可执行的文件里。第二条命令只告诉 GCC 使用 gprof 命令建立剖析信息。

当不用任何选项编译一个程序时,GCC 将会建立(假定编译成功)一个名为 a.out 的可执行文件。例如,下面的命令将在当前目录下产生一个叫 a.out 的文件:

```
gcc test.c
```

用-o 编译选项来为将产生的可执行文件指定一个文件名,由此来代替 a.out(a.out 是 Linux 中使用的一种通用文件格式,现在 Linux 的标准二进制格式为 ELF 格式)。例如,将一个叫 count.c 的 C 程序编译为名叫 count 的可执行文件,输入下面的命令来实现:

```
gcc -o count count.c
```

 注意:当使用-o 选项时,-o 后面必须跟一个文件名。

GCC 同样有指定编译器处理多少的编译选项。-c 选项告诉 GCC 仅把源代码编译为目标代码而跳过汇编和连接的步骤(例如 test3-1.c 的第一步)。这个选项使用得非常频繁,因为它使编译多个 C 程序时速度更快,并且更易于管理。缺省时 GCC 建立的目标代码文件有一个.o 的扩展名。

-s 编译选项告诉 GCC 在为 C 代码产生了汇编语言文件后,就停止编译。GCC 产生的汇编语言文件的缺省扩展名是.s。-E 选项指示编译器仅对输入文件进行预处理。当这个选项被使用时,预处理器的输出被送到标准输出(如显示器)而不是储存在文件里。

(2) 优化选项

用 GCC 编译 C/C++ 代码时,它会试着用最少的的时间完成编译并且使编译后的代码易于调试。易于调试意味着编译后的代码与源代码有同样的执行顺序,编译后的代码没有经过优化。有很多选项可用于告诉 GCC 在耗费更多编译时间和牺牲易调试性的基础上产生更小更快的可执行文件。这些选项中最典型的是-O(即 optimize,优化之意)和-O2 选项。

-O 选项告诉 GCC 对源代码进行基本优化。这些优化在大多数情况下都会使程序执行得更快。-O2 选项告诉 GCC 产生尽可能小和尽可能快的代码。-O2 选项将使编译的速度比使用-O 时慢,但通常产生的代码执行速度会更快。

除了-O 和-O2 优化选项外,还有一些低级选项用于产生更快的代码。这些选项非常

特殊,而且只有完全理解这些选项将会对编译后的代码产生什么样的效果时再去使用。这些选项的详细描述,参考 GCC 的指南页,在命令行上键入以下命令即可:

```
man gcc
```

(3) 调试和剖析选项

GCC 支持数种调试和剖析选项。在这些选项里最常用的是-g 和-pg 选项。

-g 选项告诉 GCC 产生能被 GNU 调试器(如 gdb)使用的调试信息,以便调试用户的程序。GCC 提供了一个很多其他 C 编译器里没有的特性,即在 GCC 里能使-g 和-O(产生优化代码)两个选项联用。这一点非常有用,因为能在与最终结果尽可能相近的情况下调试代码。在同时使用这两个选项时,各位必须清楚的是,所写的某些代码已经在优化时被 GCC 作了改动。关于调试 C 程序的更多信息见下一节“gdb 的使用”。

-pg 选项告诉 GCC 在用户的程序里加入额外的代码,执行时,产生 gprof 用的剖析信息以显示程序的耗时情况。

如果想详细参考 GCC 编译器参数的说明,键入命令 gcc info page(在 Emacs 内,按下 C-h i,然后选“gcc”的选项)。

3.2.2 使用 gdb

1. 使用方法

Linux 包含了一个叫 gdb 的 GNU 调试程序。gdb 是一个用来调试 C 和 C++ 程序的强有力的调试器。在程序运行时,它使用户能观察程序的内部结构和内存的使用情况。以下是 gdb 所提供的一些功能:

- 使用户能监视程序中变量的值;
- 使用户能设置断点以使程序在指定的代码行上停止执行;
- 使用户能执行一行行的代码。

在命令行上键入 gdb 并按回车键就可以运行 gdb 了。如果一切正常的话,gdb 将被启动并在屏幕上显示出类似的内容:

```
[root@zou /]# gdb
GNU gdb 5.0
Copyright 2000 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux".
(gdb)
```

启动 gdb 后,能在命令行上指定很多的选项,也可以下面的方式来运行 gdb。

`gdb filename`

用这种方式运行 `gdb` 时,能直接指定想要调试的程序。这将告诉 `gdb` 装入名为 `filename` 的可执行文件(注意是可执行文件!)。也可以用 `gdb` 去检查一个因程序异常终止而产生的 `core` 文件(当程序异常终止时,就可以在应用目录下发现一个名为“`core`”的文件,在 Redhat 6.2 下的图标是个炸弹。它报告出错的情况并供调试用),或者与一个正在运行的程序相连。可以参考 `gdb` 指南页,或在命令行上键入 `gdb -h` 得到一个有关这些选项的说明的简单列表。

2. 编译代码以供调试

为了使 `gdb` 正常工作,必须使程序在编译时包含调试信息。调试信息包含程序里的每个变量的类型、在可执行文件里的地址映射以及源代码的行号。`gdb` 利用这些信息使源代码和机器码相关联。

如何在编译时包含调试信息呢?其实很简单,如上小节所述,在编译时用 `-g` 选项打开调试选项就可以了。

3. gdb 基本命令

`gdb` 支持很多命令,可实现不同的功能。这些命令不仅包括了像装入文件等简单命令,还包括了诸如允许用户检查所调用的堆栈内容的复杂命令,表 3-1 列出了用 `gdb` 调试时会用到的一些命令。想了解 `gdb` 的详细使用,请参考 `gdb` 的指南页。

表 3-1 gdb 命令一览

命令名称	功能
<code>file</code>	装入想要调试的可执行文件
<code>kill</code>	终止正在调试的程序
<code>list</code>	列出产生执行文件的源代码的一部分
<code>next</code>	执行一行源代码但不进入函数内部
<code>step</code>	执行一行源代码而且进入函数内部
<code>run</code>	执行当前被调试的程序
<code>quit</code>	终止 <code>gdb</code>
<code>watch</code>	能监视一个变量的值而不管它何时被改变
<code>print</code>	显示表达式的值
<code>break</code>	在代码里设置断点,这将使程序执行到这里时被挂起
<code>make</code>	不退出 <code>gdb</code> 就可以重新产生可执行文件
<code>shell</code>	不离开 <code>gdb</code> 就执行 UNIX shell 命令

`gdb` 支持很多与 UNIX shell 程序一样的命令编辑特征。像在 `bash` 或 `tcsh` 里那样,按 `Tab` 键可以让 `gdb` 帮用户补齐一个惟一的命令。如果不是惟一的命令,`gdb` 会列出所有匹配的命令,也能用光标键上下翻动历史命令。

4. gdb 应用举例

本节用一个实例教用户一步步地用 `gdb` 调试程序。被调试的程序相当简单,但它展示了 `gdb` 的典型应用。

下面列出了将被调试的程序(example3-1.c),它完成的功能很简单,就是输入一个字符串(例如,“I like embedded Linux!”),然后反向将它输出。

例 3-2 example3-1.c

```
#include <stdio.h>
static void my_print (char *);
static void my_print2 (char *);

main ()
{
char my_string[] = "I like embedded Linux!";
my_print (my_string);
my_print2 (my_string);
}

void my_print (char * string)
{
printf ("The string is %s", string);
}

void my_print2 (char * string)
{
char * string2;
int size, i;
size = strlen (string);
string2 = (char *) malloc (size + 1);
for (i = 0; i < size; i++)
string2[size - i] = string[i];      ←/* 在此设立一个断点 */
string2[size] = '\0';
printf ("The string printed backward is %s \n", string2);
}
```

键入命令:

```
gcc -o example3-1 example3-1.c
```

这个程序执行时显示结果如图 3-2 所示。

输出的第一行是正确的,但第二行打印出的东西并不是所期望的,所设想的输出应该是:

```
The string printed backward is ! xuniL deddebme ekil I
```

由于某些原因,my_print2 函数没有正常工作。用 gdb 看看问题究竟出在哪儿,先键入如下命令:

```
gdb example3 - 1
```





```
root@hsong /home# gcc -o example3-1 example3-1.c
root@hsong /home# ./example3-1
example3-1: ./example3-1: 4194: test3-1.c: test3-1.c
example3-1: ./example3-1: 4194: test3-1.c: test3-1.c
example3-1: ./example3-1: 4194: test3-1.c: test3-1.c
The string is I like embedded Linux!
The string printed backward is
root@hsong /home#
```

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

图 3-2 编译并执行 example3-1.c(改正前)

如果在输入命令时忘了把要调试的程序作为参数传给 gdb,可以在 gdb 提示符下用 file 命令来载入它:

```
(gdb) file hello
```

这个命令将载入 example3-1 可执行文件,就像在 gdb 命令行里装入它一样。这时能用 gdb 的 run 命令来运行 example3-1。当它在 gdb 里被运行后结果会像这样:

```
(gdb) run
```

```
Starting program: /home/example3-1
```

```
The string is I like embedded Linux!
```

```
The string printed backward is
```

```
Program exited with code 040
```

这个输出和在 gdb 外面运行的结果一样。问题是,为什么反序打印没有工作?为了找出症结所在,在 my_print2 函数的 for 语句后设一个断点,具体的做法是在 gdb 提示符下键入 list 命令三次,列出源代码:

```
(gdb) list
```

```
(gdb) list
```

```
(gdb) list
```

技巧:在 gdb 提示符下按回车键将重复上一个命令。



第一次键入 list 命令的输出如下：

```
1 #include
2
3 static void my _ print (char * );
4 static void my _ print2 (char * );
5
6 main ()
7 {
8 char my _ string[] = "hello world!";
9 my _ print (my _ string);
10 my _ print2 (my _ string);
```

如果按下回车键，gdb 将再执行一次 list 命令，给出下列输出：

```
11 {
12
13 void my _ print (char * string)
14 {
15 printf ("The string is %s", string);
16 }
17
18 void my _ print2 (char * string)
19 {
20 char * string2;
```

再按一次回车键，将列出 example3-1.c 程序的剩余部分：

```
21 int size, i;
22
23 size = strlen (string);
24 string2 = (char * ) malloc (size + 1);
25 for (i = 0; i < size; i + + )
26 string2[size - i] = string[i];
27 string2[size + 1] = "\0";
28
29 printf ("The string printed backward is %s", string2);
30 }
```

根据列出的源程序，能看到要设断点的地方在第 26 行，在 gdb 命令行提示符下键入如下命令设置断点：

```
(gdb) break 26
```

gdb 将作出如下的响应:

```
Breakpoint 1 at 0x804857c: file hello.c, line 26.
```

现在再键入 run 命令,将产生如下的输出:

```
Starting program: /home/example3-1
```

```
The string is I like embedded Linux!
```

```
Breakpoint 1, my_print2 (string=0xbffffab0 "hello world!") at hello.c:26
```

```
26 string2[size - i] = string[i];
```

通过设置一个观察 string2[size - i] 变量值的观察点,可以观察错误是怎样产生的,做法是键入:

```
(gdb) watch string2[size - i]
```

gdb 将作出如下回应:

```
Hardware watchpoint 2: string2[size - i]
```

现在可以用 next 命令来一步步地执行 for 循环了:

```
(gdb) next
```

经过第一次循环后,gdb 告诉用户 string2[size - i] 的值是“I”。gdb 用如下的显示告诉这个信息:

```
Hardware watchpoint 2: string2[size - i]
```

```
Old value = 0 '00'
```

```
New value = 073 'I'
```

```
my_print2 (string=0xbffffab0 "I like embedded Linux") at example3-1.c:25
```

```
25 for (i = 0; i < size; i++)
```

这个值正是期望的。后来的数次循环结果都是正确的。当 i=21 时,表达式 string2[size - i] 的值等于‘!’,size - i 的值等于 1,最后一个字符已经拷到新串里了。

如果再把循环执行下去,会看到已经没有值分配给 string2[0] 了,而它是新串的第一个字符。因为 malloc 函数在分配内存时把它们初始化为空 (null) 字符 (当然也包括 string2[0]),所以 string2 的第一个字符是空字符。这解释了为什么在打印 string2 时没有任何输出了。

现在找出了问题出在哪里,修正这个错误是很容易的。在把代码里写入 string2 的第一个字符的偏移量改为 size - 1 而不是 size,也就是讲,将字符“I”写到 string[size - 1],而不是 string[size]。

改正方法非常简单,将 string2[size - i] 改为 string2[size - 1 - i] 就可以了。这样字符“I”的起始偏移量为 20,这是这种解决办法的代码。

注意:在 size = strlen (string); 这个函数里,strlen 并不将 string 末尾的 ‘\0’ 记入。也就是讲,strlen = 22,而 string2 = (char *) malloc (size + 1); 就将 string2 的长度设为了 23,那么第一个字符“!”就只能写到 string2[1],string2[0] 就是空字符了。



改正后输出的结果,如图 3-3 所示。

```

[root@bwang /home]# gcc -o example3-1 example3-1.c
[root@bwang /home]# ./example3-1
The string is I like embedded Linux!
The string printed backward is !xunil deddehne ekil I
[root@bwang /home]# █

```

图 3-3 example3-1.c 改正后的输出结果

通过上面一个简单的例子,初步掌握了 gdb 的使用方法。更多的东西希望各位在实践中细心体会和发掘。

5. xxdgdb

xxgdb 是 gdb 的一个基于 X Window 系统的图形界面版本。xxgdb 包括了命令行版的 gdb 上的所有特性。xxgdb 使用户能通过按钮来执行常用的命令。设置了断点的地方也用图形来显示。在一个 Xterm 窗口里键入下面的命令来运行它：

xxgdb

可以用 gdb 里任何有效的命令行选项来初始化 xxgdb。此外,xxgdb 也有一些特有的命令行选项,表 3-2 列出了这些选项。

表 3-2 xxgdb 命令行选项

选项名称	选项描述
db name	指定所用调试器的名字,默认是 gdb
db prompt	指定调试器提示符,默认为 gdb
gdbinit	指定初始化 gdb 的命令文件的文件名,默认为 gdbinit
bigicon	使用大图标
nx	告诉 xxgdb 不执行 .gdbinit 文件

3.3 使用 make

如今的程序多向大型化、复杂化的趋势发展。如何维护和管理程序就是个突出和紧

迫的问题。在软件工程中,往往强调程序的所谓高聚合、低耦合的特性,使程序模块内部的完整性得到加强,而相互依赖性减弱。但是显而易见,模块间无论如何也存在着相互的联系和制约,而且其关系也是非常复杂的。假如当某个模块需要改进、更换或者删除的话,传统上是依靠手工进行的。这对于程序员来讲不下于痛苦的折磨,因为当程序的规模达到由成百上千个模块组成的话,那么对程序的编译、链接的工作将会有多么复杂。

在Linux平台下,通过make指令来完成生成和维护目标程序。make程序可以判断程序修改的情况,根据makefile文件对要维护的程序模块进行重新编译和链接,以此来保证程序的更新。

3.3.1 makefile

程序模块的内部关系决定了源程序编译和链接的顺序。通过建立makefile可以描述模块间的相互依赖关系。make命令从中读取这些信息,然后根据这些信息对程序进行管理和维护。在makefile里主要提供的是有关目的文件(即“target”)与依靠文件(即“dependencies”)之间的关系,还指明了用什么命令生成和更新目标文件。有了这些信息,make会检查磁盘上的文件,如果目的文件的时间标志(该文件生成或被改动时的时间)比它的任意一个依靠文件旧,make就执行相应的命令,以便更新目的文件(目的文件不一定是最后的可执行文件,它可以是任何一个文件)。

make判断要维护文件的时序关系。make命令首先判断各个文件是否“过时”,即判断这个文件依赖的源代码或模块是否已经更新而它本身却未被改动。由于Linux系统为每个文件记录了最后修改的时间,所以判断过时与否仅仅是个简单的比较问题。make命令在对目标程序进行维护时,会一次检查它所有的依赖文件,并检查出哪些已经被更新,然后再对目标进行更新操作。

来看个例子:

例3-3 makefile编制示例

某个应用程序的模块组成如图3-4所示。

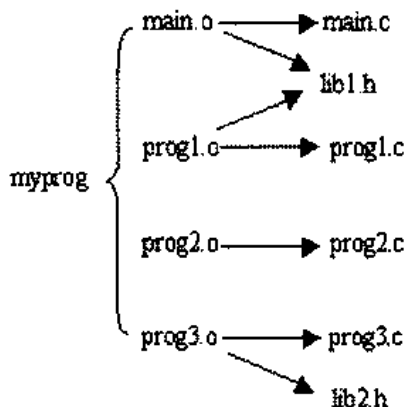


图3-4 模块关系

“→”表示依赖关系。

根据上小节的知识,可以写出生成myprog的命令:

```
gcc -o myprog main.o prog1.o prog2.o prog3.o
```

而生成.o文件(如 main.o)的命令如下:

```
gcc -c main.c lib1.h
```

同理可以写出生成 prog1.o、prog2.o 和 prog3.o 的命令。

可以用一个叫做“makefile”的文件来描述这些关系。make 命令通过读取 makefile 中间的信息来运作。makefile 的基本单位是“规则”,即描述一个目标所依赖的文件或模块,并给出其生成和更新目标文件所需要用到的命令。规则的格式如下:

目标[属性]

分隔符号 [依赖文件][;命令列]

{<tab>命令列}

与 Linux 下面的命令格式相同,[]中的内容表示为可选择项,{ }中的内容表示可出现多次。

各个条目的意义如下:

- 目标:目标文件列表,即要维护的文件列表。
- 属性:表示该文件的属性。
- 分隔符:用来分割目标文件和依赖文件的符号,如冒号“:”等。
- 依赖文件:目标文件所依赖的文件的列表。
- 命令列:重新生成目标文件的命令,可以有多条命令。

注意:在 makefile 中,除了第一条命令,每一个命令行的开头必须是一个<tab>符号,也就是制表符,而不能因为制表符相当于4个空格而不去键入 tab 符号。因为 make 命令是通过每一行的 tab 符号来识别命令行的。另外,对于第一条命令而言,不必用<tab>键,就可以直接跟在依赖文件的列表后面。对于注释的文字,起头应该用#符号,并用换行符号结束。如果要引用#符号,要用到“”。

根据以上的规则,写出其 makefile 如下:

```
# This is our first makefile
myprog: main.o prog1.o prog2.o prog3.o
gcc -o -O myprog main.o prog1.o prog2.o prog3.o
main.o: main.c lib1.h
gcc -c -O main.c
prog1.o: prog1.c lib1.h
gcc -c -O prog1.c
prog2.o: prog2.c
gcc -c -O prog2.c
prog3.o: prog3.c lib2.h
gcc -c -O prog3.c
```

makefile 需要维护的目标文件就是 myprog,“O”表示对输出进行优化。它需要依赖的文件是:main.o、prog1.o、prog2.o 和 prog3.o 这4个。改动这4个文件中的任意一个都会导致目标文件的改动和更新,从而达到了对目标程序的维护和管理。

通过一个简单的例子,了解了基本 makefile 的书写方法。对于 make 命令而言,其默认的文件名就是 makefile。如果喜欢自己命名,也可以命名为“*.mk”,如“myprog.mk”等。

3.3.2 make 命令

make 命令的使用格式为:

```
make [选项][宏定义][目标文件]
```

选项给出了 make 命令工作的方式方法,宏定义给出了执行 makefile 时所用的宏值,目标文件就是需要更新的文件列表。

一般从 makefile 的第一条规则指定的目标开始维护,而后面的命令菜单并不会立刻执行。为了确定 myprog 文件是否过时,先检查 main.o、prog1.o、prog2.o 和 prog3.o 这 4 个依赖文件是否过时。make 对依赖文件的列表从左到右逐个检查。对于那些非源文件的文件,make 把它当作新的目标继续往下查看,等到所有的依赖文件都是源文件为止。如图 3-4 所示。make 在维护 prog1.o 时,先将它看为目标文件,检察它的两个依赖文件 prog1.c 和 lib1.h 是否更新,即查看这两个文件建立的时间是否比 prog1.o 要晚,如果是,那么将使用以下命令来重新生成 prog1.o,否则对 prog1.o 不进行任何的操作。

```
gcc -c prog1.c
```

依照同样的方法,对 main.o、prog2.o 和 prog3.o 进行维护。最后再回到对 myprog 的检查。由此可见,make 对模块的维护,是由底往上,逆序遍历的过程。先检查节点,然后是父节点,在同一个层次里由左到右。这种方法即是所谓的“反向链接法”,在人工智能领域应用非常广泛。

到此,使用 make 工具来建立程序的好处就显现出来了——所有繁琐的检查步骤都由 make 做了。在源代码文件里,任何一个简单改变都会造成该文件被重新编译(因为.o

(即显示器,它是默认的标准输出)显示运行过程。而加号(+)则在该命令需要执行时,始终执行该命令。而减号(-)则忽略该命令行的执行错误,否则 make 将报告错误而停止执行。

(2) 使用伪目标

make 命令的目标可分为实目标和伪目标两种。前面的“myprog”就是要真正形成的实际存在的目标文件。而有时需要用 make 命令来做些辅助性的工作,或者对多个文件进行维护。可以通过设置伪目标来实现。

假设一个项目最后需要产生两个可执行文件,但这两个文件是相互独立的。如果一个文件需要重建,并不影响另一个,则可用“伪目标”来达到这种效果。一个伪目标跟一个正常的目标几乎是一样的,只是这个目的文件是不存在的。因此,make 总是会假设它需要被生成,当把它的依赖文件更新后,就会执行它规则里的命令行。如:

```
all : 目标1 目标2 . . . . .
```

在此,将所有需要维护的目标文件做为 all 的依赖文件。这条规则写在 makefile 的最开始,当 make 命令遇到这个规则时,由于 all 这个文件并不存在,所以它永远也不会被创建,all 后面所有的目标文件都比 all 要“新”。make 命令就按 3.3.1 小节所讲的“逆序”法对所有的目标文件进行更新。

(3) 指定需要维护的目标

一般 make 维护的是 makefile 中的第一个目标文件,但有时用户并不关心最终的目标文件如何,反而关心中间的目标文件。使用目标参数可以管理 make 的执行。如上面的例子,如果仅仅关心 prog1.o,使用以下命令即可:

```
make -f myprog prog1.o
```

还可以使用伪目标。如想把所有由 make 产生的文件删除,可以在 makefile 里设立这样一个规则:

```
veryclean :  
rm * .o  
rm myprog
```

当然,前提是没有其他规则依靠这个“veryclean”目标文件,而且 veryclean 这个目标不能在依赖文件的列表中出现。make 在执行时永远也不会执行到这条指令。如果想执行它,可以键入“make veryclean”命令。目标文件总是被 make 认为是过时的,所以此命令被执行,从而达到删除文件的目的。

(4) 使用命令选项参数

make 命令有多个选项参数,列举参数含义如下:

- -f:指定需要维护的目标。
- -i:忽略运行 makefile 中命令产生的错误,不退出 make。
- -r:忽略内部规则。
- -s:执行但是不显示所执行的命令。
- -x:将所有的宏定义都输出到 shell 环境。
- -V:列出 make 的版本号。

具体的使用方法可以参考有关书籍,在此不详述了。



3.3.3 makefile 变量

makefile 里主要包含了一些规则,除此之外就是变量定义,被称为宏定义。makefile 里的变量就像一个环境变量。事实上,环境变量在 make 过程中可以看成 make 的变量。这些宏定义是大小写敏感的,一般使用大写字母。它们几乎可以从任何地方被引用,也可以被用来做很多事情,比如:

- 储存一个文件名列表

生成可执行文件的规则包含一些目标文件名作为依赖文件。在这个规则的命令里,同样的那些文件被输送给 gcc 做为命令参数。如果在这里使用一个宏来储存所有目标文件名,那么就会很容易加入新的目标文件,而且不易出错。

- 储存可执行文件名

如果程序被用在一个非 gcc 的系统里,或者想使用一个不同的编译器,就必须将所有使用编译器的地方改成用新的编译器名。但是如果使用一个宏来代替编译器名,那么只需要改变一个地方,其他所有地方的命令名就都改变了。

- 储存编译器命令选项

假设想给所有的编译命令传递一组相同的选项(例如 -Wall -O -g),如果把这组选项存入一个宏,那么可以把这个宏放在所有调用编译器的地方。而当要改变选项的时候,只需在宏定义的地方改变这个变量的内容。

要定义一个宏,只要在一行的开始写下这个宏的名字,后面跟一个“=”号和要设定这个变量的值。以后引用这个变量时,写一个 \$ 符号,后面是括号里的变量名。格式如下:

```
$(宏名)
```

或者

```
$(宏名)
```

make 将 \$ 符号作为宏引用的开始。如果要表示 \$ 符号,那么就用 \$\$ 即可。

宏引用还支持多层引用,在处理时按照顺序依次展开。当宏名仅有单个字符时,可以省略括号,例如 \$ F = \$(F)。宏定义的地方可以在 makefile 文件中。

把前面的 makefile 利用变量重写一遍。

```
=== makefile 开始 ===
```

```
OBJS = main.o prog1.o prog2.o prog3.o
```

```
CC = gcc
```

```
CFLAGS = -O (-O 表示对编译出的代码进行优化)
```

```
myprog : $(OBJS)
```

```
$(CC) -o $(CFLAGS) myprog $(OBJS)
```

```
main.o: main.c lib1.h
```

```
$(CC) $(CFLAGS) -c main.c
```

```
prog1.o: prog1.c lib1.h
```

```
$(CC) $(CFLAGS) -c prog1.c
```



```

prog2.o:prog2.c
$(CC) $(CFLAGS) -c prog2.c
prog3.o:prog3.c
$(CC) $(CFLAGS) -c prog3.c
=== makefile 结束 ===

```

宏定义能大大简化 makefile 的书写,方便对程序的自动维护。在程序中遇到宏时,将宏出现的地方用它代表的字符串展开即可。这样,如要修改程序,仅对宏定义进行修改就可以了。

还有一些设定好的内部变量,称其为内定义宏。它们根据每一个规则内容定义,几个比较有用的变量是 \$*、\$?、\$@、\$< 和 \$- (这些变量不需要括号括住)。

- \$@:扩展成当前规则的目标文件名。
- \$<:扩展成依靠列表中的第一个依赖文件。
- \$-:扩展成整个依靠的文件列表(除掉了里面所有重复的文件名)。
- \$? :表示目标文件中新的依赖文件的列表。
- \$* :是表示依赖文件的文件名,不含扩展名。

利用这些变量,可以简化 makefile 的书写,当然也可以按照自己的需要来定义这些变量。这就是为什么用 GCC 的 -M 或 -MM 开关输出的代码可以直接用在一个 makefile 里的原因(关于 -M 选项的用法见 man 手册)。

3.3.4 在 makefile 中使用函数

makefile 里的函数跟它的变量很相似。在调用时,用一个 \$ 开始,然后是开括号、函数名,再空格,然后跟一系列由逗号分隔的参数,最后用关括号结束。例如,在 GNU Make 里有一个叫“wildcard”的函数,它有一个参数,功能是展开成一系列所有符合由其参数描述的文件名,文件之间用空格隔开。可以这样使用此命令:

```
SOURCES = $(wildcard *.c)
```

这行会产生一个所有以“.c”结尾的文件列表,然后存入变量 SOURCES 里。当然不一定要把结果存入一个变量。

另一个有用的函数是 patsubst(patten substitute, 匹配替换的缩写)函数。它需要 3 个参数。第一个是一个需要匹配的式样,第二个表示用什么来替换它,第三个是一个需要被处理的由空格分隔的字列。例如,处理那个经过上面定义后的变量:

```
OBJS = $(patsubst %.c, %.o, $(SOURCES))
```

这行将处理所有在 SOURCES 字列中的字(一系列文件名),如果它的结尾是“.c”,就用“.o”把“.c”取代。

- ⚠ 注意:这里的 % 符号将匹配一个或多个字符,而它每次所匹配的字串叫做一个“柄”。在第二个参数里,% 被认为是用第一参数所匹配的那个柄。

3.4 实例分析

下面给出一个在嵌入式 Linux 的开发中时常要用到的一个 makefile 模板,以此来作为这章的总结。通过学习这个 makefile,应用并巩固前面所学过的很多知识。

这个 makefile 是随后要讲到的以华恒公司(<http://www.hhcn.com>)开发套件为平台编写应用程序时常常要用到的一个模板。很多细节先撇开不论,将主要精力放在对 makefile 的分析上。

例 3-4 makefile 样例分析

```
CC = m68k-pic-coff-gcc
COFF2FLAT = coff2flt
```

```
CFLAGS = -I/uclinux/lib/include
LDFLAGS = /uclinux/lib/libc.a
```

```
ethernet:ethernet.o
$(CC) -o $@.coff ethernet.c $(CFLAGS) $(LDFLAGS)
$(COFF2FLAT) -o ethernet ethernet.coff
cp ethernet /ethernet
```

```
clean:
rm -f ethernet ethernet.o ethernet.coff
```

首先是 4 个宏定义。“CC = m68k-pic-coff-gcc”表示编译器是 gcc。“COFF2FLAT = coff2flt”表示的是一个命令,它将“coff”格式的文件转换成“flat”格式。“CFLAGS = -I/uclinux/lib/include”是 gcc 编译器的编译选项,“-I”表示包含其后面所列出的目录和文件。“LDFLAGS = /uclinux/lib/libc.a”表示一个文件及其路径。“ethernet”是要维护的目标文件,在随后的章节里会详细分析它。“ethernet.o”是目标文件的依赖文件。

随后就是宏的引用过程。“\$@”是动态宏引用,它扩展成当前规则的目标文件名,在此就是“ethernet.coff”。

最后,“clean”表示了一个伪目标,它永远也不会被创建,而且它又不出现在依赖文件的列表中,所以此命令总是被 make 忽略执行。如果要删除“ethernet ethernet.o ethernet.coff”这些文件,键入“make clean”就达到了删除文件的目的。

☞ 注意:“-f”参数表示“强制删除,不给出提示”。

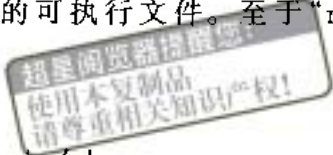
将宏展开,就可以清楚看到 makefile 的功能了。

```
ethernet:ethernet.o
m68k-pic-coff-gcc -o ethernet.coff ethernet.c -I/uclinux/lib/include uclinux/lib/libc.a
coff2flt -o ethernet ethernet.coff
```

```
clean:
```

```
rm -f ethernet ethernet.o ethernet.coff
```

首先,gcc 编译器将“ethernet.c”源文件转换成 ethernet.coff 文件,然后调用 coff2flt 命令,将 ethernet.coff 文件转换成名为“ethernet”的可执行文件。至于“m68k-pic-coff-”“coff2flt”之类的参数和命令,在后面会详细讲述。



3.5 本章小结

在这一章里,系统地学习了在 Linux 平台上编译、调试、维护应用程序的基本方法。首先学习了 gcc 编译器,掌握了它的基本命令和使用方法。然后通过实例,深入地了解了 gdb 编译器,接下来讲述了 make 命令的使用和 makefile 文件的编写。这些都是进行嵌入式 Linux 的开发和设计所必须用到的基本知识,希望引起大家注意。在阅读中,要多问一些“为什么?这一步是如何实现的?”之类的问题,并好好地进行实践,通过总结和归纳自己的所思所想,才能有所提高。

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

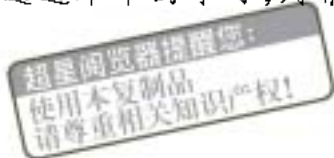
第二篇

开发入门

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

第 4 章 嵌入式 Linux 的开发平台

本章是第二篇的开始,也是走进嵌入式 Linux 开发的第一步。本章详细介绍了在开发中使用到的软件和硬件系统。通过本章的学习,对常用的开发平台会有所了解,为实际的开发做好准备。



本章主要内容:

- 华恒嵌入式 Linux 开发套件简介
- 软件系统基本配置
- uClinux 操作系统
- uClinux 开发环境的建立
- uCsim
- 系统的核心——CPU
- 其他的外围设备和接口

4.1 华恒嵌入式 Linux 开发套件简介

在读者准备研究嵌入式 Linux 并有心做开发和应用时,开发的平台也是一个不可小视的问题。如果是经验丰富的开发者,不妨自己动手组装一个,然后再挑选合适版本的嵌入式 Linux 系统,将其移植到开发平台上。当然也可以使用做好的开发平台,这样就可以将精力集中在应用的开发上。由于嵌入式系统自身的多样性,所以在挑选开发平台时必须有所考虑。应该考虑研究嵌入式 Linux 的目的是什么?是仅仅为了满足对嵌入式 Linux 技术的兴趣,还是准备以嵌入式 Linux 为软件平台,挑选一套好的硬件平台来开发自己的产品从而抢占先机占领市场呢?如果是后者,那么还要考虑更多的问题,诸如开发的周期、硬件的成本、OEM 的费用等等一系列问题才能作出决定。作者在开展对嵌入式 Linux 的研究和开发之时,经过比较和鉴别,选用了华恒公司(<http://www.hhcn.com>)新近推出的嵌入式 Linux 开发套件 II。华恒嵌入式 Linux 开发套件 II 是一套完整的基于 MC68EZ328 处理器的嵌入式 Linux 开发平台,定位于各种能够接入 Internet 的手持设备、工控设备以及网络设备的应用开发。作者在使用后觉得它非常适合大专院校师生和广大 Linux 爱好者,为深入研究应用 Linux 技术提供了一个非常理想的软硬件环境。本书的第二、三篇将围绕该开发平台展开讨论。

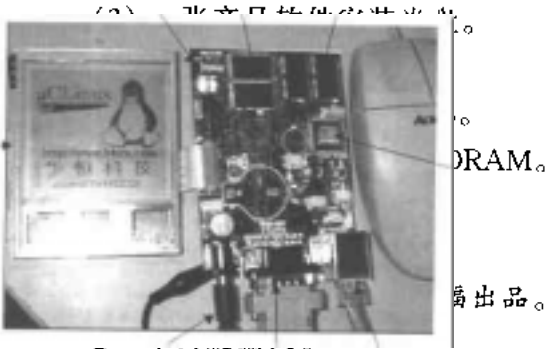
由于嵌入式系统同硬件级别的电路往往有着千丝万缕的联系,而硬件间往往有不小的区别,所以本书力图避开底层硬件琐碎的细节,对系统硬件基本特性进行讲述之后,将

精力放在编程和开发思路的阐述上。因为华恒开发套件之类的产品提供了一个较好的二次开发平台,所做的就是在该平台上进行研究和开发了。

华恒嵌入式 Linux 开发套件 II 包括:

(1) 68EZ328 开发套件。内含 68EZ328 开发原型板(即所谓的 ADS,应用开发系统)和 uClinux 软件系统。

(2) 嵌入式 uClinux 开发中文技术手册。包括了基于该套件的应用开发指南、应用开发实例以及应用开发常用接口(API)的中文说明文档。



- 预留出 I/O 管脚,可以做用户自己定义的接口。
- 直流 6V 外接电源。

开发平台的外观和各个部分的名称如图 4-1 所示。

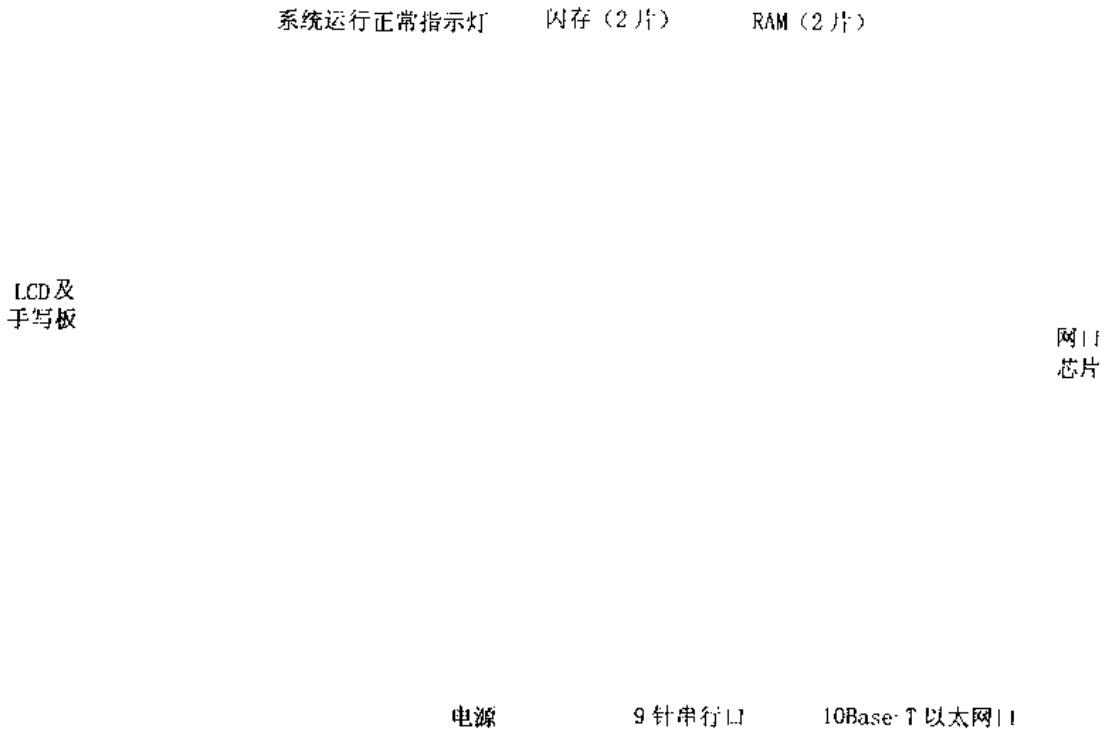


图 4-1 华恒开发套件外观图

系统还包括了由带状的电缆引出的 SPI 接口(即串行外设接口)和部分的 I/O 管脚接口,以供用户进行功能评估用。

系统的软件配置包括了 uClinux 操作系统程序及内核源码、TCP/IP 的支持程序及样例通信源码、Webserver 控制系统程序及源码、图形界面 GUI、汉字显示系统程序及源码、手写板驱动支持、软键盘系统程序及源码、成功移植的实用工具程序及其源码和串口通信样例源码。这些丰富的源代码都是了解系统、学习编写应用的绝好样例。通过分析、研究和模仿这些例子,很快就可以上手开发了。本书在随后的章节中会有选择地介绍、分析一些由作者和其他专业程序员或业余爱好者设计开发出来的样例程序。无论它们是长还是短,哪怕仅是打印一下串口,都有研习的价值。希望对这些程序进行细心研读,会使用户真正了解到开发和编写程序的流程,从而迅速地走入嵌入式 Linux 开发的美好天地。

再次需要声明的是,本书中(含光盘)的所有程序均为自由软件,遵循 GPL 原则发行。按照 GPL 2.0 的原则(见附录 A)可以对这些程序进行修改和重新发布。

4.2 软件系统配置

在本书附带的光盘中可以找到一个名为“/uclinux”的目录,该目录的结构如图 4-2 所示。

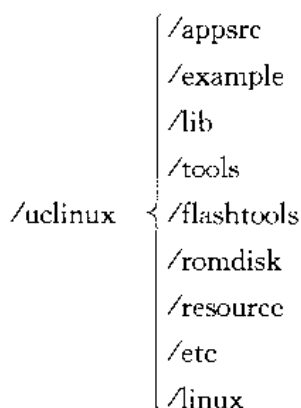


图 4-2 uClinux 目录结构

摘取几个重要的目录进行介绍。

1. appsrc 目录

该目录用于存放开发出的应用程序。这些程序在调试通过后,将通过开发板附带的工具烧写到开发板上面的 flash 中去。在该目录里还包括了如下的系统运行的重要文件和目录:

(1) makefile 文件

appsrc 目录下的 makefile 文件和与其相关的 make 文件如图 4-3 所示。其中,makefile 是主编译文件,它包含了各个 *.mk 文件,其中对各个小目录下 makefile 的调用被包含在 bins.mk 中。在 appsrc 目录下键入“make”进行编译时,就调用了 appsrc/romdisk/makefile。它调用了各个小目录(包括自己应用程序的目录)下的 makefile,编译连接各个文件,生成可执行文件,然后拷贝到/uClinux/romdisk/romdisk 目录中,在/uClinux/romdisk



目录里生成 romdisk.img 文件,并将它的格式修改成为 romdisk.s19 后,就可以烧写了(.s 文件是 Motorola 公司的十六进制的目标文件格式,在后面会专门讲述)。

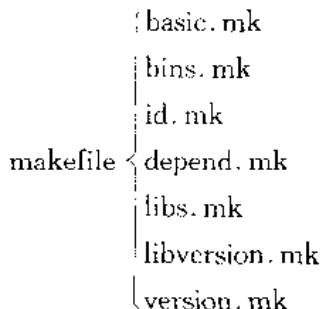


图 4-3 appsrc 目录下的 makefile 文件的组成

(2) 应用程序目录

这些目录是开发板自带的应用程序和自己开发的应用程序所存放的目录。

例如:

- /blight: 开发板打开/关闭背光源程序。
- /handpad: 手写板输入程序的源代码。
- /httpd: 开发板上的一个小型的 Webserver 程序的源代码。
- /gui: 图形用户界面的样例程序源代码。

(3) 系统和有关工具目录

这些目录存放了系统的各种工具和维持系统运转必要的程序代码。

- /get(还有/put): 从系统的内存中读取/写入数据的程序的源代码。
- /mount: 提供了诸如 nfsmount 之类的系统工具。
- /ping: ping 命令的源程序。
- /ramimage: ram 盘的镜像文件,其中还包括了 holes.c 文件,修改该文件就可以更改 ram 盘大小及地址。
- /appsrc/arch/embed: 包括了 crt0.S 和 user.ld 等重要的与硬件特定结构相关的文件。

对这些文件解释如下:

ld 文件

即 link descript 文件。该目录下的 user.ld 指定了程序在连接时,各个段所处的位置。

各个段的定位是根据所用的硬件板上 RAM 及 flash 的大小,以及系统软件的要求决定的。例如,开发板将内核放在 flash 上,启动后也仍然直接在 flash 上执行,这样可以节省内存。

crt0.S

即所谓的“Start up code”,是应用程序应该连接的启动代码,该文件包括了初始化程序栈和程序退出时的处理工作。为了链接并运行的 C/C++ 程序,需要定义一个小模块(通常用汇编语言写成),也就是 crt0.S,从而确保程序在调用主函数“main”之前,硬件可以根据 C 语言的常量得以初始化。要编写自己的 crt0.S 的话,就必须获取以下的信息:

- 内存映射。开发板上有何种内存? 地址是什么?

- 堆栈是如何构成的?
- 输出的模式是什么?

crt0.S 至少应该做下列的工作:

① 定义 start 符号(在汇编语言中写作 `start`),程序在 start 符号处开始执行。

② 设置堆栈指针 `sp`。

在开发板的内存映射表中,可以随意选择堆栈的存储位置。最简单的方法就是在未初始化的数据区(我们常常把这个区域叫做 `bss`)中的某处选择一个固定大小的区域来存储。选择放在内存的高位还是低位取决于堆栈生长的方向是向上还是向下。将未初始化的数据区(`bss`)中的所有内存数据初始化为零。利用链接脚本(即上面讲的 `ld` 文件),可以很轻松地完成这个工作。通过链接脚本,可以定义诸如 `bss_start` 和 `bss_end` 等符号,来记录该段的边界和大小。最后可以通过使用“for”循环来初始化所有的段间的内存。

③ 调用 main 主程序。

一个更加完整的 `crt0.S` 应该进行下列的工作:

- 定义一个 `_exit` 子程序。这是个 C 语言形式的名字。在汇编语言中,应该使用两倍长的下划线,写成 `__exit` 的形式。它具体的行为依据系统的具体情况和具体需求而定。如果程序中含有启动监控程序, `__exit` 就跳转到启动监控程序那儿去。如果没有启动监控程序, `__exit` 就跳转到启动程序处;或者干脆跳转到 `start` 符号处重新开始执行。

- 执行一些对硬件的初始化工作,例如初始化某个 MMU(内存管理单元,对于 `uClinux` 而言是没有 MMU 的,也自然不做这么一步了)或者是某个附属的浮点运算芯片等。

- 定义一个低层次的输入和输出子程序。例如,在 `crt0.S` 中就很适合定义一些汇编语言级别的子程序或者是例行的程序。

2. example 目录

该目录下面放置了一些样例程序。后面章节的相关部分会详细讲解这些程序。

3. lib 目录

该目录存放了 `uclibc` 中的重要各种头文件、定义等。在编写应用时,使用“`#include`”的编译预处理命令时所指定的默认目录。其中存放了系统基本的链接共享库文件。

📢 注意: `uClinux` 不支持动态链接。

4. tools 目录

`tools` 目录存放的是重要的系统工具。例如,在 `/uClinux/tools/bin` 目录中存放了 `m68k-pie-coff-gcc`,这是用户使用的编译器。在这里对一些名词做些解释。

`m68k`

即 `m68000`,是 Motorola 公司的一个芯片族。`Gcc` 编译器可以运行在不同的 CPU 之



上,针对不同的 CPU, gcc 使用的形式也略有不同。M68k 即是 gcc 针对 m68k 体系的 CPU 所加上的开关项。

pic

pic 即 position independence code, 与位置无关的二进制格式文件, 在程序段中必须包括 reloc 段, 从而使代码加载时可以进行重新定位。

coff

coff (common object file format) 是一种通用的对象文件格式, 同 Linux 采用的 elf (executable linked file) 文件格式一样, 都是定义目标文件的标准格式。

在 tools 目录中还有如下的工具:

m68k-pic-coff-ld

ld 即 GNU 的链接器。在嵌入式系统中, 一个程序要由源代码经过编译、链接和重新定位三步才能够变成可执行的程序。ld 工具通过运行上面讲述的 ld 文件来控制输出的目标文件。

m68k-pic-coff-ar

ar 是 GNU 的库管理命令。ar 命令可以建立并改善文档, 也可以从文档中获取摘要信息。

m68k-pic-coff-as

as 是 GNU 汇编语言的编译器。as 被设计用来在不同的平台上工作, 就如同该平台上的自带编译器一样。对 m68k 体系的芯片而言, 其汇编语言所用的符号较为特殊, 自然要在 as 命令前加上选项。

5. flashtools 目录

flashtools 目录包括了擦除和烧写闪存的工具 flash_erase 和 flash_romdisk。在本书附带光盘的 /flashtools 目录中, 可以找到一个 source.c 文件, 就是这两个工具的源代码。在编写完应用程序后, 先用 flash_erase 工具将闪存擦写干净, 然后用 flash_romdisk 将应用程序烧写进去。

6. romdisk 目录

该目录包含了 /romdisk 目录和 romdisk.img、romdisk.map 和 romdisk.s19 这三个文件。其中, romdisk.s19 是 Motorola 公司十六进制的目标文件格式, 烧写闪存时将它烧写进去。

romdisk/romdisk 目录和开发板上的目录结构是一样的, 其结构如图 4-4 所示。

其中, bin/ 目录用来存放烧写到开发板上的可执行文件, 如 lissa、gui 等; proc/ 目录是反映内核运行状况的虚的文件系统, 并不实际地存在于内存上。htdocs/ 目录包含了开发板上的 Webserver 所用到的文档资料。tmp/ 目录是个很重要的目录, 它提供装载逻辑文件系统在虚拟文件系统的装载点。通过 nfsmount 命令将宿主机的硬盘装载到开发板的虚拟文件系统上, 这样就可以用 flash_romdisk 命令向目标板下载文件。lib/ 目录提供了程序运行必要的库文件。而 etc/ 目录中包含了 init 文件。毫无疑问, 这是系统的初始

化程序。dev/目录包含了 uClinux 的所有的外部设备。系统提供了对外部设备的驱动程序,访问各种外部设备就如同访问文件一样轻松。例如文件/dev/ttyS0 就代表了串行通信口。



图 4-4 romdisk/romdisk 的目录结构

7. etc 目录

这个包含了系统管理的所需要的配置文件,是由 uClinux 系统所产生的 minirc.dfl 文件。使用“minicom -s”命令可以修改它的参数。

8. resource 目录

该目录包含了开发板所使用的所有的各种软件资源的压缩包和一些相关的文档,例如 keyboard.bmp 和 l_keyboard.bmp 等样例程序要用到的位图文件。

软件资源包括了 linux-2.0.38.tar.bz2(它是 2.0.38 版本的 Linux 的内核压缩包)和 uClinux-2.0.38.990904.diff.gz(uClinux 的补丁)等许多非常重要的文件。4.4“uClinux 开发环境的建立”一节中,将讲述这些文件的含义。

4.3 uClinux 操作系统

在本节中,讲述软件中最关心的部分——uClinux 操作系统。

4.3.1 uClinux 简介

uClinux 是 Linux 2.0 版本的一个分支,它被设计用来应用微控制领域。我们都知道,Linux 是一种很受欢迎的操作系统,它与 UNIX 系统兼容,开放源代码。它原本被设计为桌面系统,现在广泛应用于服务器领域。而更大的影响在于它正逐渐地应用于嵌入式设备。uClinux 正是在这种氛围下产生的。uClinux 的标志和吉祥物分别如图 4-5 和

4-6 所示。



图 4-5 uClinux 的标志



图 4-6 uClinux 的吉祥物

uClinux 念作“you see Linux”。在 uClinux 这个英文单词中，“u”来自希腊文，表示 Micro，小的意思，“C”表示 Control，控制的意思，所以 uClinux 就是 Micro-Control-Linux，字面上的理解就是“针对微控制领域而设计的 Linux 系统”。

uClinux 最大的特征就是没有 MMU(内存管理单元模块)。它很适合那些没有 MMU 的处理器，例如 m68ez328 等。这种没有 MMU 的处理器在嵌入式领域中应用得相当普遍。

同标准的 Linux 相比，由于 uClinux 自身不支持 MMU，多任务的实现就需要技巧了。但是，在 uClinux 上运行的绝大多数的用户程序并不需要多任务。另外，针对 uClinux 内核的二进制代码和源代码都经过了重新编写，以紧缩和裁剪基本的代码。这就使得 uClinux 的内核同标准的 Linux 2.0 的内核相比非常之小，但是它仍然保持了 Linux 操作系统的主要的优点，如稳定性、强大的网络功能和出色的文件系统支持等。uClinux 包含 Linux 常用的 API、小于 512K 的内核和相关的工具。操作系统所有的代码加起来 < 900KB。

uClinux 有一个完整的 TCP/IP 协议栈，同时对其他许多的网络协议都提供支持。这些网络协议都在 uClinux 上得到了很好的实现。uClinux 可以称做是一个针对嵌入式系统的优秀网络操作系统。

uClinux 所支持的文件系统有多种，其中包括了最常用的 NFS(网络文件系统)、ext2 (第二扩展文件系统，它是 Linux 文件系统的实际上的标准)、MS-DOS 及 FAT16/32 等。

在华恒公司的网站(<http://www.hhen.org>)上提供了 uClinux 的详细分析，经过允许，现将其整理如下。

4.3.2 uClinux 的小型化

1. 标准 Linux 可能采用的小型化方法

对于标准的 Linux,可能采用的小型化方法有:

(1) 重新编译内核

Linux 内核采用模块化的设计,即很多功能块可以独立地加上或卸下,开发人员在设计内核时把这些内核模块作为可选的选项,在编译系统内核时指定。因此一种较通用的做法是对 Linux 内核重新编译。在编译时,要仔细选择嵌入式设备所需要的功能支持模块,同时删除不需要的功能。通过对内核的重新配置,可以使系统运行所需要的内核显著减小,从而缩减资源使用量。

(2) 制作 root 文件系统映像

Linux 系统在启动时必须加载根文件系统,因此剪裁系统同时包括 root file system 的剪裁。在 x86 系统下, Linux 可以在 DOS 下,使用 Loadlin 文件加载启动。

2. uClinux 采用的小型化方法

(1) uClinux 的内核加载方式

uClinux 的内核有两种可选的运行方式:

- flash 运行方式: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)。同时这也是标准 Linux 系统采用的启动方式。

(2) uClinux 的根(root)文件系统

uClinux 系统采用 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:第一,内核支持 romfs 文件系统比支持 ext2 文件系统需要的代码更少。第二,romfs 文件系统相对简单,在建立文件系统超级块(用来描述文件系统整体信息的数据结构,是文件系统的核心所在)时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 盘的方法进行处理(ram 盘采用 ext2 文件系统)。

(3) 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,相对于越来越大且越来越全的 glibc 库,uClibc 对 libc 做了精简。uClinux 对用户程序采用静态连接的形式,这种做法会使应用程序变大,但是基于内存管理的问题,不得不这样做。

4.3.3 uClinux 的开发环境

1. GNU 开发套件

GNU 开发套件作为通用的 Linux 开放套件,包括一系列的开发调试工具。主要组件有以下三种:

(1)Gcc:编译器,可以做成交叉编译的形式,即在宿主机上开发编译目标上可运行的二进制文件。

(2)Binutils:一些辅助工具,包括可以反编译二进制文件(objdump)、汇编编译器(as)、连接器(ld)等。

(3)Gdb:调试器,可使用多种交叉调试方式,如背景调试工具(gdb-bdm)和使用以太网网络调试器(gdbserver)。

2. uClinux 的打印终端

通常情况下,uClinux 的默认终端是串口,内核在启动时所有的信息都打印到串口终端(使用 printk 函数打印),同时也可以通过串口终端与系统交互。

uClinux 在启动时启动了远程登录服务(telnetd),操作者可以远程登录到系统上,从而控制系统的运行。至于是否允许远程登录,可以在烧写 romfs 文件系统时由用户决定是否启动远程登录服务。

3. 交叉编译调试工具

为了支持一种新的处理器(例如 m68ez328),必须具备一些编译、汇编工具,使用这些工具可以形成可运行于这种处理器的二进制文件。对于内核使用的编译工具同应用程序使用的有所不同。

内核编译连接时,使用 ucsimm.ld 文件,形成可执行文件映像,所形成的代码段既可以使用间接寻址方式(即使用 reloc 段进行寻址),也可以使用绝对寻址方式。这样可以给编译器更多的优化空间。因为内核可能使用绝对寻址,所以内核加载到的内存地址空间必须与 ld 文件中给定的内存空间完全相同。

应用程序的连接与内核连接方式不同,应用程序由内核加载(可执行文件加载器将在后面讨论)。由于应用程序的 ld 文件给出的内存空间与应用程序实际被加载的内存位置可能不同,这样在应用程序加载的过程中需要一个重新定位的过程,即对 reloc 段进行修正,使程序进行间接寻址时不至于出错。这个问题与在 i386 等高级处理器上方法有所不同,本文将在后面进一步分析。

由上述讨论可知,至少需要两套编译连接工具。在讨论过 uClinux 的内存管理后,下面将给出整个系统的工作流程以及系统在 flash 和 ram 中的空间分布。

提示:所谓交叉编译器,就是运行在一个计算机平台上并对另一个平台进行编译操作的编译器。

4. 可执行文件格式

先对一些名词作一些说明:

- `coff`(common object file format):一种通用的对象文件格式。
- `elf`(excutive linked file):一种为 Linux 系统所采用的可执行连接文件的通用文件格式。
- `flat`:`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。

uClinux 系统使用 `flat` 可执行文件格式。`gcc` 的编译器不能直接形成这种文件格式,但可以形成 `coff` 或 `elf` 格式的可执行文件,这两种文件需要 `coff2flt` 或 `elf2flt` 工具进行格式转化,形成 `flat` 文件。

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(可执行文件加载器,详见 `fs/binfmt_flat.c`)。以下对 `reloc` 段进一步讨论。

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加载到的内存空间不同。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(设若实际地址为 `0x10000`)。这个实际地址是根据 `ld` 文件计算出来(因为连接器假定该程序将被加载到由 `ld` 文件指定的内存空间)。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将按 `ld` 文件加载。这时如果程序仍然跳转到绝对地址 `0x10000` 处执行,通常情况这是不正确的。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。若使用变量 `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,则减去一个偏移量)。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。

这种 `reloc` 方式从某种程度上讲是由 uClinux 的内存分配问题引起的,这一点将在 uClinux 内存管理分析时说明。

4.3.4 uClinux 针对实时性的解决方案

uClinux 本身并没有关注实时问题,它并不是为了 Linux 的实时性而提出的。另外一种 Linux—RT-Linux 关注实时问题(详见第 9 章)。RT-Linux 执行管理器把普通 Linux

的内核当成一个任务运行,同时还管理了实时进程。而非实时进程则交给普通 Linux 内核处理。这种方法已经应用于很多的操作系统以增强操作系统的实时性,包括一些商用版 UNIX 系统、Windows NT 等。这种方法优点之一是实现简单,且实时性能容易检验。优点之二是由于非实时进程运行于标准 Linux 系统,同其他 Linux 商用版本之间保持了很大的兼容性。优点之三是可以支持硬实时的应用。uClinux 可以使用 RT-Linux 的 patch,从而增强 uClinux 的实时性,使 uClinux 可以应用于工业控制、进程控制等一些实时要求较高的应用。

4.3.5 uClinux 的内存管理

应该说 uClinux 同标准 Linux 的最大区别就在于内存管理,同时也由于 uClinux 的内存管理引发了一些标准 Linux 所不会出现的问题。在此把 uClinux 内存管理同标准 Linux 的内存管理部分进行比较分析。

1. 标准 Linux 使用的虚拟存储器技术

标准 Linux 使用虚拟存储器技术,这种技术用于提供比计算机系统中实际使用的物理内存大得多的内存空间。使用者会感觉到程序好像可以使用非常大的内存空间,从而使编程人员在写程序时不用考虑计算机中物理内存的实际容量。

为了支持虚拟存储器的管理, Linux 系统采用分页(paging)的方式来载入进程。所谓分页即是把实际的存储器分割为相同大小的段,例如每个段 1024 个字节,这样 1024 个字节大小的段便称为一个页面(page)。

虚拟存储器由存储器管理机制及一个大容量的快速硬盘存储器支持,它的实现基于局部性原理。当一个程序在运行之前,没有必要全部装入内存,而是仅将那些当前要运行的那部分页面或段装入内存运行,其余暂时留在硬盘上。程序运行时,如果它所要访问的页(段)已存在,则程序继续运行。如果发现不存在的页(段),操作系统将产生一个页错误(page fault),这个页错误导致操作系统把需要运行的部分加载到内存中。必要时操作系统还可以把不需要的内存页(段)交换到磁盘上。利用这样的方式管理存储器,便可把一个进程所需要用到的存储器以化整为零的方式,视需求分批载入。而核心程序则凭借属于每个页面的页码来完成寻址各个存储器区段的工作。

标准 Linux 是针对有内存管理单元的处理器设计的。在这种处理器上,虚拟地址被送到内存管理单元(MMU),把虚拟地址映射为物理地址。通过赋予每个任务不同的虚拟物理地址转换映射,来支持不同任务之间的保护。地址转换函数在每一个任务中定义,在一个任务中的虚拟地址空间映射到物理内存的一个部分,而另一个任务的虚拟地址空间映射到物理存储器中的另外区域。计算机的内存管理单元(MMU)一般有一组寄存器来标识当前运行的进程转换表。在当前进程将 CPU 放弃给另一个进程时(一次上下文切换),内核通过指向新进程地址转换表的指针加载这些寄存器。MMU 寄存器是有特权的,只能在内核态才能访问。这就保证了一个进程只能访问自己空间内的地址,而不会访问和修改其他进程的空间。当可执行文件被加载时,加载器根据默认的 ld 文件,把程序加载到虚拟内存的一个空间。因为这个原因实际上很多程序的虚拟地址空间是相同的,

但是由于转换函数不同,所以实际所处的内存区域也不同。而对于多进程管理,当处理器进行进程切换并执行一个新任务时,一个重要部分就是为新任务切换任务转换表。Linux 系统的内存管理至少能实现以下功能:

- 运行比内存还要大的程序。理想情况下可以运行任意大小的程序。
- 运行只加载了部分的程序,缩短了程序启动的时间。
- 可以使多个程序同时驻留在内存中提高 CPU 的利用率。
- 可以运行重定位程序。即程序可以放于内存中的任何一处,而且可以在执行过程中移动。
- 可编写与机器无关的代码。程序不必事先约定机器的配置情况。
- 减轻程序员分配和管理内存资源的负担。
- 可以进行共享。如果两个进程运行同一个程序,它们应该可以共享程序代码的同一个副本。
- 提供内存保护,进程不能以非授权方式访问或修改页面,内核保护单个进程的数据和代码以防止其他进程修改它们。否则,用户程序可能会偶然(或恶意)地破坏内核或其他用户程序。

提示:多任务系统中,上下文切换是指 CPU 的控制权由运行任务转移到另外一个就绪任务时所发生的事件,当前运行任务转为就绪(或者挂起、删除)状态,另一个被选定的就绪任务成为当前任务。上下文切换包括保存当前任务的运行环境和恢复将要运行任务的运行环境。上下文的内容依赖于具体的 CPU。

当然,虚存系统并不是没有代价的。内存管理需要地址转换表和其他一些数据结构,留给程序的内存减少了。地址转换增加了每一条指令的执行时间,而对于有额外内存操作的指令会更严重。当进程访问不在内存的页面时,系统发生处理失效。系统处理失效,并将页面加载到内存中,就需要极耗时间的磁盘 I/O 操作。总之内存管理活动占用了相当一部分 CPU 时间(在较忙的系统的大约占 10%)。

2. uClinux 针对无 MMU 的特殊处理

对于 uClinux 来说,其设计针对没有 MMU 的处理器,即 uClinux 不能使用处理器的虚拟内存管理技术。uClinux 仍然采用存储器的分页管理,系统在启动时把实际存储器进行分页。在加载应用程序时程序分页加载,但是由于没有 MMU 管理,所以实际上 uClinux 采用实存储器管理策略(real memory management)。这一点影响了系统工作的很多方面。

uClinux 系统对于内存的访问是直接的(它对地址的访问不需要经过 MMU,而是直接送到地址线上输出),所有程序中访问的地址都是实际的物理地址。操作系统对内存空间没有保护(这实际上是很多嵌入式系统的特点),各个进程实际上共享一个运行空间(没有独立的地址转换表)。

在 uClinux 中,一个进程在执行前,系统必须为进程分配足够的连续地址空间,然后全部载入主存储器的连续空间中。与之相对应的是标准 Linux 系统在分配内存时没有必要保证实际物理存储空间是连续的,而只要保证虚存地址空间连续就可以了。程序加载地址与预期(ld 文件中指出的)通常都不相同,这样 relocation 过程就是必须的。此外磁盘

交换空间也是无法使用的,系统执行时如果缺少内存,将无法通过磁盘交换来得到改善。

uClinux 对内存的管理减少就给开发人员提出了更高的要求。如果从易用性这一点来说,uClinux 的内存管理是一种倒退,退到了 UNIX 早期或是 DOS 系统时代,开发人员不得不参与系统的内存管理。从编译内核开始,开发人员必须告诉系统这块开发板到底拥有多少内存,从而系统将在启动的初始化阶段对内存进行分页,并且标记已使用的和未使用的内存。系统将在运行应用时使用这些分页内存。

由于应用程序加载时必须分配连续的地址空间,不同的硬件平台对于可一次成块(连续地址)地分配内存大小限制是不同的(目前针对 ez328 处理器的 uClinux 是 128K,而针对 coldfire 处理器的系统内存则无此限制),所以开发人员在开发应用程序时必须考虑内存的分配情况,并关注应用程序需要运行空间的大小。另外,由于采用实存储器管理策略,用户程序同内核以及其他用户程序在一个地址空间,程序开发时要保证不侵犯其他程序的地址空间,以使程序不至于破坏系统的正常工作,或导致其他程序的运行异常。

从内存的访问角度来看,开发人员的权利增大了(开发人员在编程时可以访问任意的地址空间),但与此同时,系统的安全性也大为下降。此外,系统对多进程的管理将有很大的变化,这一点将在 uClinux 的多进程管理中说明。

虽然 uClinux 的内存管理与标准 Linux 系统相比功能相差很多,但这是嵌入式设备的选择。在嵌入式设备中,由于成本等敏感因素的影响,普遍采用不带有 MMU 的处理器,这决定了系统没有足够的硬件支持来实现虚拟存储管理技术。从嵌入式设备实现的功能来看,嵌入式设备通常在某一特定的环境下运行,只要实现特定的功能,内存管理的要求完全可以由开发人员考虑。

4.3.6 uClinux 系统对进程和线程的管理

1. 标准的 Linux 系统对进程和线程的管理

(1) 进程

进程是一个运行程序并为其提供执行环境的实体,包括一个地址空间和至少一个控制点,进程在这个地址空间上执行单一指令序列。进程地址空间包括可以访问或引用的内存单元的集合,进程控制点通过一个程序计数器的硬件寄存器控制和跟踪进程指令序列。

(2) fork

由于进程为执行程序的环境,因此在执行程序前必须先建立这个能“跑”程序的环境。Linux 系统提供系统调用拷贝现行进程的内容,以产生新的进程,调用 fork 的进程称为父进程;而产生的新进程则称为子进程。子进程会继承父进程的一切特性,但是它有自己的数据段。也就是说,尽管子进程改变了所属的变量,却不会影响到父进程的变量值。

父进程和子进程共享一个程序段,但是各自拥有自己的堆栈、数据段、用户空间以及进程控制块。换言之,两个进程执行的程序代码是一样的,但是各有各的程序计数器与私人数据。

当内核收到 fork 请求时,它会先查核三个条件:首先检查存储器是否足够;其次是进

程表是否仍有空缺;最后则是用户是否建立了太多的子进程。如果上述三个条件满足,那么操作系统会给予进程一个进程识别码,并且设定 CPU 时间。接着设定与父进程共享的段,同时将父进程的文件节点索引(inode)拷贝一份给予进程运用,最终子进程会返回数值 0 以表示它是子进程。至于父进程,它可能等待子进程的执行结束,或与子进程各做各的。

(3) exec 系统调用

该系统调用提供一个进程去执行另一个进程的能力。exec 系统调用是调用覆盖旧有



问题。

fork 系统调用必须给予进程提供一个与父进程地址空间逻辑上明显不同的副本。大多数情况下,当子进程在 fork 后调用 exec 或 exit,它会遗弃这个地址空间。因此,在老 UNIX 系统的实现中,为进程建立一个实际的地址空间副本是非常浪费的。

这个问题通过两种方式来解决。第一种是 copy-on-write(写中复制)方法,它首先由 System V 采用,现在为包括 Linux 在内的大多数 UNIX 系统使用。在这种方法中,父进程的数据和堆栈页暂时设置为只读,并被标记为 copy-on-write。子进程有一份自己的地址转换表(比如页表),与父进程共享内存页。而无论是父进程还是子进程试图修改页时,都会产生页面失效异常(由于页面标识为只读)并且会调用内核的失效处理程序(linux: do_wp_page)。该处理程序识别这是一个 copy-on-write 页,为其建立一个新的可写的页副本。由此只有那些被修改的页才需复制,而不是全部页面。若子进程调用 exec 或 exit,页面回到原来的保护方式,copy-on-write 标识被清除。

BSD 提供了另一种解决方案——新的 vfork 系统调用。若用户进程希望在调用 fork 后随即调用 exec,它可以调用 vfork 而不是 fork。vfork 不进行复制。相反,父进程将自己的地址空间租借给子进程,并将自己阻塞,直至子进程将地址空间归还给它。因此子进程会一直使用父进程的地址空间直到它调用 exec 或 exit。于是内核将地址空间返回给父进程并唤醒它。vfork 非常快,甚至连地址映射表(如页表)也无需复制。地址空间传给子进程只是简单地拷贝地址映射寄存器。然而,由于它允许一个进程使用并修改另一个进程的地址空间,这是一个非常危险的调用。某些程序,如 csh,就利用这个特性。

uClinux 显然无法采用第一种方案,只好通过 vfork + exec 并通过引入一个 vforkwait 信号量来解决上述的竞态条件。

实现代码如下:

```

kernel/exit.c
static inline void __exit_mm(struct task_struct * tsk)
{
.....
wake_up(&tsk->mm->vforkwait); //这里会唤醒父进程
.....
}
kernel/fork.c
int do_fork(unsigned long clone_flags, unsigned long usp, struct pt_regs
* regs)
{
.....
#ifdef NO_MM
if (clone_flags & CLONE_WAIT) {
sleep_on(&p->mm->vforkwait); //这里父进程进入睡眠
}

```

```

#ifndef /* NO_MM */
.....}
fs/exec.c
int do_execve(char * filename, char ** argv, char ** envp, struct pt_regs
* regs)
{
.....
#ifdef NO_MM
        wake_up(t->mm->vforkwait); //这里会唤醒父进程
#endif /* NO_MM */
.....}

```

当然,如果子进程不及时调用 exec,父子进程的并发也就是所谓的多任务是一个问题,应该承认,COW(copy-on-write)的支持更为优雅有效,但 vfork 的性能甚至可能更好一些,当然要求 uClinux 的用户开发时注意配合(vfork + exec),至于安全性,没有 MMU 也就无暇顾及了。

4.4 uClinux 开发环境的建立

建立 uClinux 开发环境的方法有很多,而且也十分方便。一般有两种方法:购买正式发行的 CD-ROM 和从网上直接下载 uClinux 的源代码自己组建开发环境。下面着重讲述后者。

4.4.1 通过源代码建立开发环境

可以从 <http://www.uclinux.org/pub/uClinux> 处下载 uClinux 的源代码。具体步骤如下:

(1) 下载一个未经过修改的 Linux 的 2.0.38 版本,名为“linux-2.0.38.tar.gz”的压缩文件。使用如下的命令:

```
tar -xzvf linux-2.0.38.tar.gz /opt/uClinux
```

将其解压缩到/opt/uClinux 目录下。当然也可将它解压到别的地方,并无硬性的规定。

(2) 下载最新版本的 uClinux 的补丁程序,例如 uClinux-2.0.38-990904.diff 文件。下载地点是 <http://www.uclinux.org/pub/uClinux/>,然后把它解压到/opt/uClinux/linux 目录下。解压后,键入如下命令给 uClinux 打补丁:

```
patch -p1 < uClinux-2.0.38.990904.diff
```

(3) 从网上下载并解压 uClinuxgcc-kit-160899.tar.gz 文件到/opt/uClinux/目录中。

注意:该文件包含了用来构建针对 m68k 体系的 CPU 编译器的补丁。

接着,下载(不要解压)这两个文件:binutils-2.9.1.tar.gz 和 gcc-2.7.2.3.tar.gz 到目录/opt/uClinux 中,执行以下步骤:

(1)cd /opt/uClinux/uclinuxgcc-kit-160899。

(2)编辑 makefile,将安装目录(即 INSTALLDIR 项)设置成要安装 uClinux 的编译器的目录。

(3)键入命令“make”来安装这一系列工具。

注意:必须检查机器上是否安装了 flex 程序。flex 是个快速的语法分析工具,它产生相应的程序来对文本进行文本匹配的检查。

如果要为 uCsimm(uCsimm 是由维护 uClinux 发展和推广的那个公司推出的 uClinux 开发硬件环境)建立一个 uClinux 的内核,则执行以下步骤:

(1)cd /opt/uClinux/linux。

(2)键入“make menuconfig”,选择想要的参数。

(3)保存。

(4)退出 menuconfig。

(5)键入“make dep”。

(6)键入“make linux.bin”来编译内核。

最后,执行如下步骤:

(1)下载 uC-libc-160899.tar.gz 和 uC-libm-060199.tar.bz2 这两个文件,并将它们解压到/opt/uClinux/目录中。

(2)cd uClib-c。

(3)键入命令“make”。

对于 uC-libm 目录,执行同样的步骤。

上述步骤完成后,将产生 libc.a 和 libmf.a 文件,就完成了 uClinux 软件环境的建立。

另外,读者还可以从华恒公司的网站(<http://www.hhcn.org>)上下载源代码并执行类似的步骤进行安装。详细的情况请参见华恒公司的网站(<http://www.hhcn.org>)上的“uClinux 开发环境建立步骤”一文。

4.4.2 从所购买的正式发行的 CD-ROM 安装

如果从 uClinux 的网站上订购了 uClinux 的正式发行版的 CD-ROM,那么安装起来就比较轻松了。

1. 安装二进制文件

安装 uClinux 最容易的方法莫过于使用 RPM 的二进制包来安装了。从/RPMS/目录中,可以找到所需要的二进制文件。

首先,要搞清楚所用的标准 Linux 的发行版本所使用的共享库是哪一个。办法是键入命令:

```
# ldd /bin/ls
```

注意: ldd 命令的功能是输出程序所需要的共享库。

然后,“cd”到适当的目录下(/RPMS/libc5 或者是/RPMS/libc6),键入“make”命令。

注意:对于 uClinux 的初始化安装而言,这些步骤都可能是必须的。

由于所使用的 Linux 的发行版本可能有区别,可能要编辑一下 makefile 文件,以去掉相关的文件依赖性的检查。例如:

```
install:
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/m68k-coff-binutils-2.9.1-1.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/m68k-coff-gcc-2.7.2.3-1.i386.rpm
```

“-nodeps”参数表示在安装或升级一个 RPM 包时不进行依赖性检查。“-force”表示强制安装,即使 RPM 包中的某些部分已经在系统上安装过。键入“make install”命令时,rpm 命令就不会进行依赖性检查,并将 RPM 包全部安装在用户机器上。

为了节省时间,可直接从 <http://www.uClinux.org> 网站上下载一个已经编辑好的 makefile。以下就是一个编辑好的 makefile。

```
#
```

```
# this just makes all the rpms in the right order
```

```
#
```

```
all: install
```

```
install:
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/m68k-coff-binutils-2.9.1-1.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/m68k-coff-gcc-2.7.2.3-1.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/m68k-pic-coff-binutils-2.9.1-1.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/m68k-pic-coff-gcc-2.7.2.3-1.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/uC-libc-0.9.1-1.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/uC-libm-0.9.1-1.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/uClinux-2.0.38.1pre5-2.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/uC-src-0.9.2-2.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/genromfs-0.3-1.i386.rpm
```

```
rpm -i --force --nodeps /cdrom/RPMS/libc6/romdisk-0.9.1-1.i386.rpm
```

```
echo Done installing RPMs
```

```
uninstall:
```

```
rpm -e uC libc
```

```
rpm -e uC-libm
```

```
rpm -e uClinux
```




```
rpm -e uC-src
rpm -e genromfs
rpm -e romdisk
rpm -e m68k-pic-coff-binutils
rpm -e m68k-coff-binutils
rpm -e m68k-coff-gcc
rpm -e m68k-pic-coff-gcc
echo Completely uninstalled
```

```
build:
rpm -ba -vv m68k-coff-binutils-2.9.1.spec
rpm -ba -vv m68k-coff-gcc-2.7.2.3.spec
rpm -ba -vv m68k-pic-coff-binutils-2.9.1.spec
rpm -ba -vv m68k-pic-coff-gcc-2.7.2.3.spec
rpm -ba -vv uClinux-2.0.38.1pre5.spec
rpm -ba -vv uC-libc-0.9.1.spec
rpm -ba -vv uC-libm-0.9.1.spec
rpm -ba -vv uC-src-0.9.2.spec
rpm -ba -vv genromfs-0.3.spec
rpm -ba -vv romdisk-0.9.1.spec
echo removing garbage
rm -rf /opt/uClinux/ *
```

是否需要进一步改编,取决于 CD-ROM 的装载点和用户是否有 libc5 或 libc6。

☞ 注意:如果使用的是 RedHat,应该在安装 uClinux 前将 genromfs 类型的包移走。

键入以下命令:

```
make -f [我们自己的 makefile]
```

把 uClinux 的系列工具、源代码和应用程序安装到 /opt/uClinux 中。

2. 设置工作环境

设置工作环境的步骤如下:

(1) 创建一个工作目录。

(2) 在该目录下键入“buildenv”。

(3) 键入“make”。这个命令将把 uClinux 的源代码拷贝过来,并将它们编译成默认的二进制镜像文件。

这样就得到了二进制镜像文件 image.bin,它是在 uCsimms 上预装的默认镜像文件。

3. 构建镜像文件

工作目录中包含了一个名为“deftemplate.sh”的文件,此脚本文件规定了哪些二进制

文件将被加入 image.bin 文件。它仅仅加入二进制文件。要删除不想要的工具程序,应该将它们从模板文件中删除,同时也应该手工将它们从/romdisk 目录中删除。在 simm 上的 ROMfs 镜像是/romdisk 目录的一个拷贝。所以,要保证/romdisk 目录仅仅包含了需要的文件。要检查或修改 romdisk/etc 中的文件(如 rc, resolv.conf 等)来满足用户需要。一旦编辑过 romdisk 和 template 文件,运行 make 就能在工作目录中构建一个新的 image.bin。

“image.bin”文件实际上是将内核的镜像“linux.bin”和文件系统的镜像“romdisk.img”串联起来的一个文件。如果要重新编译内核的话,执行下列的步骤:

- (1) cd /opt/uClinux/linux。
- (2) 键入 make menuconfig。
- (3) 键入 make dep。
- (4) 键入 make linux.bin。
- (5) 进入到工作目录,运行 make,将这个新的 linux.bin 包含到 image.bin 中。

4. 将镜像文件烧写入 flash ROM 中

确保 simm 已经通过串行口同 PC 相连,在 PC 上运行一个终端仿真程序(例如 minicom)。检查串行口,确保其波特率已经被设置成 9600bps,不要硬件或软件流控。设置调制解调器的 init 和 reset 的项(用 minicom -s 命令,使用其提供的菜单就可以设置)为空项。给 uCsimm 加电后,可以看到以下显示:

```
uCbootstrap v1.x...etc
```

随后出现提示符号。如果选择了 AUTOBOOT,在限定的时间内按 Esc 键,然后键入 bootloader 命令(bootloader 是系统的一端很小的引导程序,相当于 Linux 系统的 LILO)。

键入“fast”来改变端口的速率至 115200bps(不要忘记同时在 minicom 中也将其修改)。在提示符下键入“rx”,然后,终端仿真中就开始执行 XMODEM 的上载(在 minicom 中按 Ctrl-s 键),将用来编写的文件传送到模块中(通常来自工作目录的 image.bin 文件)。当上载完毕后,键入“program”将镜像写入 ROM。然后键入“go”来执行新写入的镜像(如果重新启动,则要将终端仿真器波特率设置为 9600bps)。

☞ 注意:如果从 RAM 中执行镜像,参考 4.5 节。

如果购买华恒公司的开发套件,那么开发环境的建立将更加简单。具体的过程将在第 5 章中详细介绍。

4.4.3 使用 minicom

在前文中讲到了 minicom 的使用。minicom 是一个友好易用的串口通信程序。通过它可以监视并控制串行口的信息。由于在后面的开发中经常要使用 minicom,它也是开发环境中十分重要的一个部分,所以本小节对 minicom 做详细介绍。

1. minicom 的启动

有两种方法可以启动 minicom,一种是在控制台方式下,键入以下命令即可:

```
minicom
```

以 Redhat 6.2 版本为例)下,按照以下方法启动 minicom:

。minicom 的界面如图 4-8 所示。

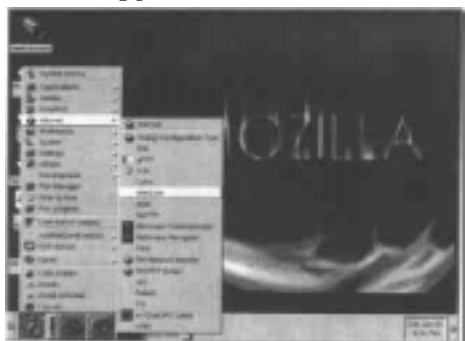


图 4-7 minicom 的启动方法

图 4-8 minicom 的界面

2. minicom 基本语法

```
minicom [-s|om|lz8] [-c on|off] [-S script] [-d entry]
```

```
[-a on|off] [-t term] [-p pty] [-C capturefile]
[configuration]
```

minicom 是一个通信程序,像共享软件 TELIX,但其源码可以自由获得,并能够运行于多数的 UNIX 系统。它包括以下特性:

- 自动重拨号的拨号目录;
- 对串行设备 UUCP 格式的 lock 文件的支持;
- 独立的脚本;
- 语言解释器;
- 文件捕获;
- 多用户单独配置。

关于命令行参数解释如下:

(1) -s: 设置

root 使用此选项在 /etc/minirc.dfl 中编辑系统范围的缺省值。使用此参数后, minicom 将不进行初始化,而是直接进入配置菜单。如果系统被改变,或者第一次运行 minicom 时, minicom 不能启动,这个参数就会很有用。对于多数系统,已经内定了比较合适的缺省值。

(2) -o: 不进行初始化

若选择此参数, minicom 将跳过初始化代码。如果未复位(reset)就退出了 minicom, 又想重启一次会话(session),那么用这个选项就比较好(不会再有错误提示: modem is locked),但是也有潜在的危险。由于未对 lock 文件等进行检查,因此一般用户可能会与 uucp 之类的东西发生冲突,也许以后这个参数会被去掉。

(3) -m: 用 Meta 或 Alt 键重载命令键

在 1.80 版中这是缺省值,也可以在 minicom 菜单中配置这个选项。若一直使用不同的终端,其中有些没有 Meta 或 Alt 键,那么方便的做法还是把缺省的命令键设置为 Ctrl + A。当有支持 Meta 或 Alt 键的键盘时,再使用此选项。

(4) -z: 使用终端状态行

使用该参数必须保证下列两个条件:终端支持使用该参数,并且在其 termcap 或 terminfo 数据库入口中有相关信息。

(5) -l: 逐字翻译高位被置位的字符

使用此标志, minicom 将不再尝试将 IBM 行字符翻译为 ASCII 码,而是将其直接传送。许多 PC-UNIX 克隆不经翻译也能正确显示它们(Linux 使用专门的模式: Coherent 和 Sco)。

(6) -a: 特性使用

有些终端,特别是 televideo 终端,有个很讨厌的特性处理(串行而非并行)。 minicom 缺省使用“-a on”,若在用这样的终端,就必须加上选项“-a off”。尾字“on”或“off”都必须加上。

(7) -t: 终端类型

使用此标志,可以重载环境变量 TERM,这在环境变量 MINICOM 中使用很方便;可以创建一个专门的 termcap 入口供 minicom 在控制台上使用,它将屏幕初始化为 raw 模



式。这样,连同“-f”标志一起就可以不经翻译而显示 IBM 行字符。

(8) -S:脚本

启动时执行给定名字的脚本。到目前为止,还不支持将用户名和密码传送给启动脚本。如果还使用了“-d”选项,已在启动时开始拨号,此脚本将在拨号之前运行。拨号项目入口由“-d”指明。

(9) -d:启动时拨打拨号目录中的一项

可以用索引号指明,也可以使用入口项的一个子串。所有其他程序初始化过程结束后,拨号将会开始。

(10) -p:要使用的伪终端

它超载配置文件中定义的终端端口,但仅当其为伪 tty 设备。提供的文件名必须采用这样的形式:(/dev/)tty[p-z][0-f]。

3. minicom 的使用

minicom 是基于窗口的。要弹出所需功能的窗口,可按 Ctrl+A 键,然后再按各功能键(a-z 或 A-Z)。先按 Ctrl+A 键,再按 z 键,将出现一个帮助窗口,它提供了所有命令的简述。配置 minicom(-s 选项,或者 Ctrl+A,O)时,可以改变这个转义键。

以下键在所有菜单中都可用:

- UP arrow-up 或“k”。
- DOWN arrow-down 或“j”。
- LEFT arrow-left 或“h”。
- RIGHT arrow-right 或“l”。
- CHOOSE Enter。
- CANCEL Escape。

屏幕分为两部分:上部 24 行为终端模拟器的屏幕。在此窗口中解释 ANSI 或 VT100 转义序列。若底部还剩有一行,那么状态行就放在这儿;否则,每次按 Ctrl+A 键时状态行出现。

下面按字母顺序列出可用的命令:

Ctrl+A:按两次 Ctrl+A 将发送一个 Ctrl+A 命令到远程系统。如果把“转义字符”换成了 Ctrl+A 以外的什么字符,则对该字符的工作方式也类似。

- A:切换“Add Linefeed”为 on/off。若为 on,则每次回车键在屏幕上显示之前,都要加上一个 linefeed。
- B:提供了一个回滚的缓冲区。

可以按 u 上滚,按 d 下滚,按 b 上翻一页,按 f 下翻一页。也可用箭头键和翻页键。可用 s 或 S 键(大小写敏感)在缓冲区中查找文字串,按 N 键查找该串的下一次出现。按 c 键进入引用模式,出现文字光标,就可以按 Enter 键指定起始行。回卷模式结束后,带有前缀“>”的内容将被发送。

- C:清屏。
- D:拨一个号,或转向拨号目录。
- E:切换本地回显为 on/off (若 minicom 版本支持)。

- F: 将 break 信号送 modem。
- G: 运行脚本(Go)。运行一个登录脚本。
- H: 挂断。
- I: 切换光标键在普通和应用模式间发送的转义序列的类型。
- J: 跳至 shell。返回时, 整个屏幕将被刷新。
- K: 清屏, 运行 kermi, 返回时刷新屏幕。
- L: 文件捕获开关。打开时, 所有到屏幕的输出也将被捕获到文件中。
- M: 发送 modem 初始化串。若 online, 且 DCD 线设为 on, 则 modem 被初始化前将要求用户进行确认。
- O: 配置 minicom, 转到配置菜单。
- P: 通信参数。允许改变 bps 速率, 校验奇偶和位数。
- Q: 不复位 modem 就退出 minicom。如果改变了 macros, 而且未存盘, 会提供一个 save 的机会。
- R: 接收文件。从各种协议(外部)中进行选择。若 filename 选择窗口和下载目录提示可用, 会出现一个要求选择下载目录的窗口。否则将使用 Filenames and Paths 菜单中定义的下目录。
- S: 发送文件。选择在接收命令中使用的协议。

如果未设置文件名选择窗口为可用(在 File Transfer Protocols 菜单中设置), 就只能在一个对话框窗口中写文件名。若将其设置为可用, 将弹出一个窗口, 显示上传目录中的文件名。可用空格键为文件名加上或取消标记, 用光标键或 j/k 键上下移动光标, 被选的文件名将高亮显示。目录名在方括号中显示, 按两次空格键可以在目录树中上下移动。最后, 按 Enter 发送文件, 或按 Esc 键退出。

- T: 选择终端模拟: 彩色(ANSI)或 VT100。此处还可改变退格键, 打开或关闭状态行。
- W: 切换 linewrap 为 on/off。
- X: 退出 minicom, 复位 modem。如果改变了 macros, 而且未存盘, 提供了一个 save 的机会。
- Z: 弹出 help 屏幕。

4. minicom 的配置

通常 minicom 从文件“minirc.dfl”中获取其缺省值。若给 minicom 一个参数, 它将尝试从文件“minirc.configuration”中获取缺省值。因此, 为不同端口、不同用户等创建多个配置文件是可能的。最好使用设备名, 如: tty1、tty64、sio2 等。如果用户创建了自己的配置文件, 那么该文件将以“.minirc.dfl”为名出现在 home 目录中。

按 Ctrl + A 键、O 键, 就进入了 setup 菜单。任何人都可以改变其中的多数设置, 但有些仅限于 root。在此, 那些特权设置用星号(*)标记。

(1) Filenames and paths

此菜单定义用户的缺省目录。

- A- download: 下载文件的存放位置。



- B- upload:从此处读取上传的文件。
- C- script:存放 login 脚本的位置。
- D- script program:作为脚本解释器的程序。缺省是“runscript”,也可用其他的東西(如:/bin/sh 或“expect”)。Stdin 和 Stdout 连接到 modem,Stderr 连接到屏幕。若用相对路径(即不以“/”开头),则是相对于 home 目录,除了脚本解释器以外。
- E- kermit program:为 kermit 寻找可执行程序 and 参数的位置。命令行上可用一些简单的宏:“%l”扩展为拨出设备的完整文件名,“%b”扩展为当前波特率。

(2) File Transfer Protocols

此处规定的协议将在按下 Ctrl + A、s/r 键时显示。行首的“Name”为将要显示在菜单中的名字。“Program”为协议路径,其后的“Name”则确定了程序是否需要参数,如要传送的文件。“U/D”确定了该项是否在“upload/download”菜单中出现。“Fullscr”确定要否全屏运行,否则 minicom 将仅在一个窗口中显示其标准输出。“IO-Red”确定 minicom 要否将程序的标准 IO 连接到 modem 端口。“Multi”告诉文件名选择窗口协议能否用一个命令发送多个上传文件,它对于下载协议无效。如果不用文件名选择窗口,那么上传协议也会忽略它。老版本的 sz 和 rz 非全屏,并且设置了 IO-Red。但是,有些基于 curses 的版本,至少是 rz,不希望其 stdin 和 stdout 被改向,以及全屏运行。所有文件传输协议都以用户的 UID 运行,但并不是总有 UID = root。对于 kermit,命令行上可用“%l”和“%b”。在此菜单内,还能规定当提示文件要上传时,是否选择窗口,以及每次自动下载开始时是否提示下载目录。如果禁止下载目录提示,将使用 file and directory 菜单中规定的下载目录。

5. 串口设置

(1) * A - 串行设备

一般用/dev/tty1。Linux 下用/dev/cua 或/dev/modem。如果有多个 modem 连接到两个或两个以上的串口,可以在这儿列表指定,用空格、逗号或者分号作为分隔符。minicom 启动时,检查此列表直至发现有可用的 modem,并使用它。

(2) * B - Lock 文件位置

在多数系统上,应该用/usr/spool/uucp。Linux 系统则使用 var/lock。若此目录不存在,minicom 将不会试图使用 lock 文件。

(3) * C - Callin program

若串口上有 uugetty 设备,可能就需要运行某个程序,把 modem 的 cq 端口切换到 dialin/dialout 模式。这就是进入 dialin 模式所需的程序。

(4) * D - Callout program

这是进入 dialout 模式所用的程序。

(5) * E - Bps/Par/Bits

启动时的缺省参数。

minicom 将其配置文件保存在一个目录中,通常是:

/var/lib/minicom, /usr/local/etc 或者/etc。

想知道 minicom 编译时内定的缺省目录,可用“minicom -h”命令,可能还会找到 runscript(1)的 demo 文件。

4.5 uCsimmm

4.5.1 简介



前面多次提到了“uCsimmm”这个名词。那么，uCsimmm 到底是什么呢？其实，uCsimmm 就是专门针对 uClinux 操作系统而构造的微控制模块（“simmm”即单列直插模块之意）。它的标准外观如图 4-9 所示。

图 4-9 uCsimmm 外观图示

它仅一英寸高，带标准 30 针输出端子。由于嵌入式系统需要良好的稳定性和以太网功能，对空间要求严格，而且所处理的进程比较单一明确，所以 uCsimmm 特别适合嵌入式系统。它的用途相当多，并已经运用在从网络服务器到可编程逻辑控制器的广泛领域中。

uCsimmm 由 Motorola 公司的龙珠 68EZ328 处理器驱动，自带 2 兆的闪存和 8 兆的 DRAM。同时还带有一个 10Base-T 以太网口和 RS-232 高速串行口，并带有内置的 LCD 驱动器，能以 QVGA 方式显示 320 x 240 的矩阵。

关于 uCsimmm 的参数，如表 4-1 所示。

表 4-1 uCsimmm 外形参数

	外形物理参数
宽	3.5 inch
高	1 inch
厚	25 inch
管脚	标准 30 针

所用到的元器件：

- CPU: Microprocessor MC68EZ328 DragonBall™
- RAM Memory: 8 Mb DRAM (4096k x 16 bits)
- flash ROM: (可选)
 - 1Mb (512k x 16 bits)
 - 2 Mb (1024k x 16 bits)
 - 4 Mb (2048k x 16 bits)
- Ethernet 网口: 10baseT

电气指标:

电压: $+3.3\text{ V} \pm 5\%$

管脚分配:

- 1 Ethernet RX -
- 2 Ethernet RX +
- 3 Ethernet TX -
- 4 Ethernet TX +
- 5 PWMO / CTS
- 6 TI/O / RTS
- 7 VCC + 3.3 V
- 8 /MR
- 9 GND
- 10 STXD
- 11 SRXD
- 12 RSRXD
- 13 RSTXD
- 14 SCLCK
- 15 PD7 / IRQ 6
- 16 PD6 / IRQ 3
- 17 PD5 / IRQ 2
- 18 PD4 / IRQ 1
- 19 PD3 / INT 3
- 20 PD2 / INT 2
- 21 PD1 / INT1
- 22 PD0 / INT0
- 23 LACD
- 24 LCLK
- 25 LLP
- 26 LFLM
- 27 LD3
- 28 LD2
- 29 LD1
- 30 LD0

一个标准的 uCsim 板的配置如下:

- 一个 16MHz 的 68EZ328 DragonBall 微控制器;
- 2 MB flash ROM;
- 8 MB DRAM;
- 21 个通用的 I/O 管脚(如果带图形 LCD 板,最多可以有 13 个);
- 10Base-T 以太网控制芯片;



- RS-232 串行支持；
- 高速的(1Mbit/sec) I2C 总线；
- 3.3V 电压,在空闲态时其电流相当低,仅毫安培级。

4.5.2 加入 uCsimmm 的邮件列表

如果对 uCsimmm 感兴趣,可以参加 uCsimmm 的邮件列表。方法是发电子邮件到:
ucsimmm-request@uclinux.com

并在消息的正文中写上“subscribe uCsimmm”即可。这样,就可以收到许多关于 uCsimmm 和 uclinux 开发的讨论与答复的邮件。

超星阅读器
使用本复制品
请尊重相关知识产权!

4.6 系统的核心——CPU

4.6.1 CPU 主要特性

对于一个嵌入式的系统来讲,没有什么器件比 CPU 更重要了。在开发板中使用的是 Motorola 公司 MC68328CPU 家族龙珠系列中的一个新款——MC68EZ328。MC68EZ328 继承了龙珠系列 CPU 良好的显示支持功能,同时对 LCD 的支持更加灵活。该处理器主要针对外部设备较少的手持设备。该 CPU 工作电压为 3.3V,能很好适应当今手持移动设备市场不断增长的需要。该 CPU 的结构图如图 4-10 所示。

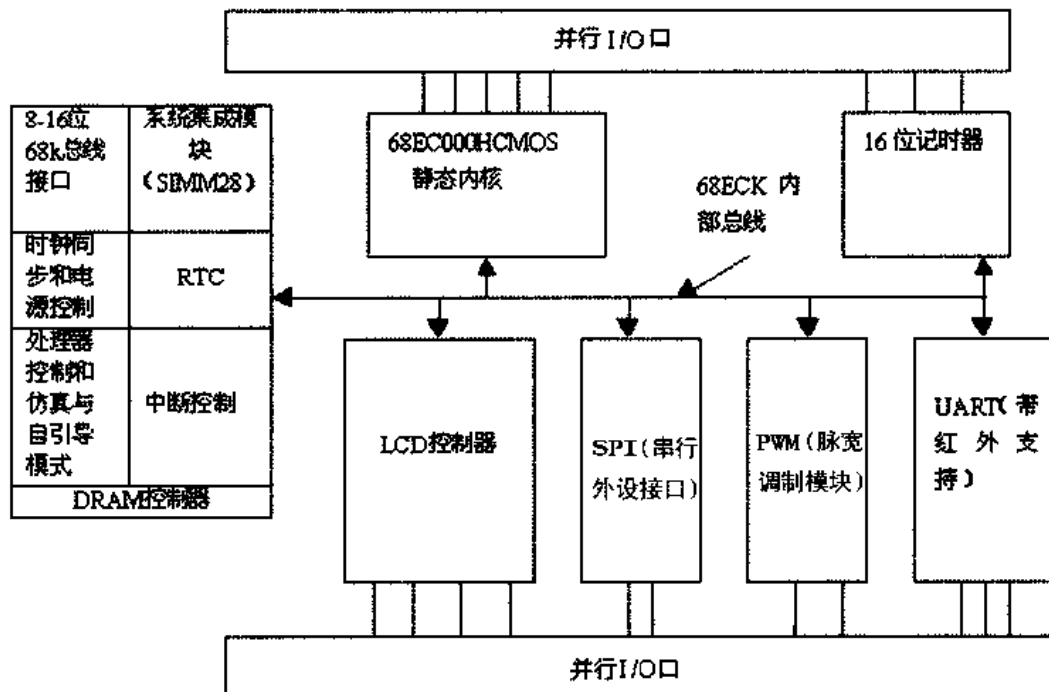


图 4-10 MC68EZ328 模块结构图

MC68EZ328 的主要特性如下所示:

1. 静态 68EC000 内核处理器

它有如下的特性:

- 与 MC68EC000 处理器功能等效;
- 完全兼容 MC68000 及 MC68EC000;
- 32-Bit 内部地址总线;
- 24-Bit 外部地址总线,带片选功能,如果使用片选信号 CSA 或 CSB,那么最大的寻址空间为 $4 \times 16\text{MB}$,如果使用 CSC 或 CSD,那么就为 $4 \times 4\text{MB}$;
- 16-Bit 片内数据总线;
- 采用静态设计,这使得可以停止处理器的时钟来节省能耗;
- 在 16.58 MHz 的处理器时钟下,处理器的指令执行速率为 2.7 MIPS(2.7 兆指令/每秒);在 20 MHz 的处理器时钟下,处理器的指令执行速率为 3.25 MIPS;
- 针对 8/16 位的数据端口,MC68EZ328 带有可选择总线宽度的外部总线接口。

2. 系统集成模块 (SIM28-EZ) 集成了许多外部阵列逻辑功能

其功能如下:

- 系统设置,可编程的地址映射;
- 同 SRAM、EPROM 和 FLASH 的无缝接口;
- 8 个可编程的片选信号,带等待发生逻辑;
- 4 个可编程的中断 I/O 口,带键盘中断功能;
- 5 个通用的可编程边缘/电平/极性中断请求;
- 47 个并行可编程 I/O,同其他外部设备功能相复用;
- 针对芯片内部外围设备的可编程中断向量响应;
- 低能耗模块控制。

3. DRAM 控制器

它具有以下的优点:

- 支持 CAS-before-RAS 刷新周期和自刷新模式的 DRAM;
- 支持 8/16 位端口的 DRAM;
- 为 LCDC 读取数据提供 EDO 或自动快速页面模式;
- 可编程的刷新率;
- 可编程列地址大小。

提示: CAS 表示列地址选通, RAS 表示行地址选通。

4. 通用异步收/发器 (UART)

其特点如下:

- 支持 IrDA(红外)物理层协议,最大速率可达 115.2kbps;
- Tx 上 FIFO 为 8 字节, Rx 线上是 12 字节。

5. 串行外设接口(SPI)

其特点如下:

- 16 位可编程 SPI 口,提供支持外设;
- 支持主控模式。

6. 16 位通用记数/计时器

其特点如下:

- 自动中断发生功能;
- 计时器 I/O 管脚。

7. 实时时钟/采样时钟

其特点如下:

- 可对实时时钟的独立电源供给;
- 一个可编程的闹钟功能;
- 可记数至 512 天;
- 具有可选频率的采样时钟 (4Hz、8Hz、16Hz、32Hz、64Hz、256Hz、512Hz 和 1kHz),可以为数据采集或键盘溢出提供中断。

8. LCD 控制器

其特点如下:

- 荧屏大小软件可编程(最大可达 640×512),支持单色显示。可支持 Motorola、Sharp、Hitachi 和 Toshiba 等多家公司的 LCD;
- 支持黑/白 4 灰度和 16 灰度等级的显示;
- 使用系统内存作为显存;
- 使用 8 位 PWM 为 LCD 对比控制器。

9. 脉宽调制器 (PWM) 模块

其特点如下:

- 5 字节的 FIFO 管道,提供了更加灵活的性能;
- 可产生声音和旋律。

10. 内置的仿真功能

其特点如下:

- 为了方便仿真调试功能而特地预留的地址空间,并可提供片选功能;
- 为 ICE 指定了级别为 7 的中断等级(最高级别);
- 带一个具有屏蔽功能的地址信号比较器和控制信号比较器,可提供单个或多个硬件中断执行点;
- 一个断点指令插入单元。



11. 自引导功能

其特点如下:

- 允许用户初始化系统,并由 UART 下载程序/数据至系统内存中;
- 允许使用执行指令来运行系统内存中的程序;
- 为 68k 指令的存储和执行,提供一个 8 字节长的指令缓冲。

12. 电源管理

其特点如下:

- 全静态 HCMOS 技术;
- 使用 32.768 kHz/38.4 kHz 的时钟;
- 低能耗停止功能;
- 单个模块可独立关闭;
- 最低能耗模式控制。

13. 操作电压为 3.0V~3.6V

其操作电压在 3.0V~3.6V 之间。

4.6.2 CPU 各个部分的功能概述

1. 系统集成模块

MC68EZ328 的系统集成模块(SIM28-EZ)包含了诸如控制系统启动、初始化、设置、控制外部总线的诸多功能。内存接口使用户可以方便地与现有的 SRAM、EPROM 和闪存交换数据。在片选逻辑的帮助下,等待状态可编程。中断控制器接收了内部产生的和外部发送的中断信号,并将它们按优先级排列。集成模块同时处理了电源控制的屏蔽和唤醒选择控制。通过改变 CPU 的频率或停止 CPU 的工作,低能耗控制逻辑模块可用来控制 CPU 的电能分配。另外,SIM28-EZ 通过设置其管脚,使用户可以选择是使用指定的 I/O 口还是并行的 I/O 口。这个特性使得在指定的功能未被使用时,该管脚可以用做 I/O 口,从而增加现有的 I/O 口数目。

2. 系统设置

MC68EZ328 的系统设置逻辑包括了系统控制寄存器(SCR),使用户可以设置下列主要功能:

- 系统状态和控制逻辑;
- 总线错误产生控制;
- 保护模块控制寄存器,使其不能被用户程序所存取。

3. VCO/PLL 时钟同步器

时钟同步器以外部晶振或外部的振荡器为参考而工作。它使用了内部的锁相环和压



控振荡器或外部的时钟。它可以以操作频率直接驱动时钟信号。

4. 片选逻辑

MC68EZ328 提供了 8 个可编程的通用片选信号。对某个给定的片选块来讲,无论 DTACK 信号是否自动产生,也无论过了多少个等待状态(由 0 到 6),用户都可以选择片选信号是只读还是读、写均可。

5. 外部总线接口

外部总线接口处理了内部的 68EC000 内核和内存、外设或其他在外部地址空间的功能部件的信息传送功能。它包括了一个 16 位内部 68000 总线接口和 8 位外部接口。

6. 中断处理器

中断处理器接受了内部和外部的中断请求,并对其按优先级排序,并在 CPU 中断确认周期时,产生一个向量号。同时还提供了中断嵌套,这样,一个低级别的中断服务例程可以被高级别的中断请求所打断。该片内中断控制器有如下的特征:

- 可按优先级别排列中断源(内部和外部的);
- 提供一个可完全屏蔽的中断环境;
- 可编程中断向量的产生;
- 中断的屏蔽;
- 唤醒中断屏蔽。

7. 并行通用 I/O 口

MC68EZ328 支持最大可达 54 位的通用 I/O 口。它们可以设置成通用的 I/O 口,也可以设置成为片内模块对指定的外部设备的接口管脚。即使当对某个外设的所有管脚都被设置为通用的 I/O 口,该外设仍然工作正常,虽然此时仅 RTC 模块和记时模块能发挥功能。

8. 低能耗停止逻辑

对节省电能消耗来讲,有不同的选择来达到该目的:关闭未使用的外部设备,降低处理器的时钟速率,关闭处理器或其中某些部分。

在中断控制逻辑中产生一个中断可以将 CPU 从低能耗模式中唤醒,该中断控制逻辑在处理器的低能耗状态中也一直工作。某些可选择的中断可产生一个 68EC000 内核的唤醒信号。片内的外设可以初始化一个唤醒行为。例如,可以设置记时器在一定的崩溃周期之后或一定的外部事件发生后苏醒过来。

9. DRAM 控制器

为了给系统设计人员提供更大灵活性,MC68EZ328 支持对 8 位或者 16 位的 DRAM 接口。该 DRAM 控制器支持最大 2 槽的 DRAM/EDO DRAM,每个槽可支持 $512k \times 8$ 、 $256k \times 16$ 、 $1M \times 16$ 或 $8M \times 8$ 的容量。同时,也支持 CAS-before-RAS 刷新周期和



Self-Refresh 刷新模式。

10. UART 和红外通信的支持

UART 支持标准的正常波特率的异步串行通信,并同 IrDA1.0 物理通信协议相兼容,最大的速率可达 Kbps 级别。

11. 串行外设接口(SPI)

串行外设接口是一种主控模式的,高速同步串行接口,连接诸如 A/D 转换器和非易失性的 RAM。SPIM(SPI 控制器)为数据传送提供了时钟信号,在主控模式工作。

12. 记时器

MC68EZ328 处理器包含了一个通用的 16 位记时器,带可编程的前置放大。该记时器还带有可编程的边缘触发和输出比较功能以及输入俘获功能。

13. 实时时钟 RTC

MC68EZ328 中的实时时钟由一个 32.768kHz/38.4kHz 的晶振驱动。它同时提供了闹钟中断功能。RTC 还提供采样功能。用户可从预定义的时钟频率中挑选想要的频率,并会产生一个中断。该采样时钟可用来进行数据采集等。

14. LCD 控制器

MC68EZ328 的 LCD 控制器同龙珠系列 LCD 控制器的功能相近,但是在支持灰度等级和更小的屏幕等方面有更大的灵活性。其特征如下:

- 提供对单彩 STN 和彩色 STN LCD 模块的接口;
- 最大可达 4 个等级的灰度,带帧控;
- 利用系统的 RAM 进行显示;
- 通过 DMA 进行屏幕刷新;
- 可设置的屏幕大小最大可达 640×512;
- 使用 8 位 PWM 作为 LCD 对比控制。

15. 脉宽调制模块(PWM)

为了从所存储的采样中产生高质量的声音,MC68EZ328 的脉宽调制模块经过了优化设计。当 FIFO 中有 1 个或 0 个字节时,就可以产生一个可屏蔽的中断。系统软件就可以向 FIFO 中写入 4 个字节或 2 个字节的采样,这样就可以加强音响效果。

16. 内置仿真模块(ICEM)

电路内仿真模块为 MC68EZ328 提供了片内仿真功能。它是针对仅使用 4 个接口信号的低成本仿真器设计的。更详细的信息由外部的仿真器提供。



4.7 其他的外围设备和接口

在这一节中将简要地介绍系统的一些重要的外围设备和接口。

1. 网卡芯片

开发板使用的网卡控制芯片是由 cirrus logic 公司推出的 CS8900A, 是专门针对嵌入式系统设计的嵌入式的网卡芯片。它是一种低价位的工业用以太网控制器, 专门针对工业和嵌入式领域中的应用。它采用了高度集成化的设计, 采用了片内 RAM、10Base-T 以太网发送接收滤波和一个直接的带 24mA 的驱动器的 ISA 总线接口(具有很好的通用性)。

CS8900A 产品的基本特性如下:

- 带直接 ISA 总线接口的单片 IEEE 802.3 以太网控制器;
- 最大电流消耗为 55mA(在 5 V 电压下);
- 操作电压为 3V;
- 运行在工业温度范围内;
- 全双工操作;
- 片内 RAM 缓冲, 可接收和发送帧;
- 带模拟滤波的 10BASE-T 以太网端口;
- 提供了自动极性校验和更正功能;
- 针对 10BASE2、10BASE5 和 10BASE-F 的 AUI 端口;
- 接收信息可程控;
- 对无跳变设置的 EEPROM 支持;
- 对无盘系统的 PROM 启动支持;
- 对显示连接状态和 LAN 活动状态的 LED 驱动支持, 并支持悬挂和停止模式。

2. RS-232 接口

RS-232 是美国电子工业协会(EIA)在 1969 年颁布的一种推荐标准。RS-232 是按串行位通信的总线, 可以在同步和异步两种方式下使用。所传送的数据类型和帧长均不受限制。RS-232 线是一种 DTE(数据终端设备)和 DCE(数据通信设备)间的信号传输线。DTE 是所传送数据的源或宿主, 可以是一台计算机或一个数据终端。DCE 可以是一个调制解调器或一种数据通信设备。

控制串行口通信的控制芯片是 Max232E 型芯片, 读者可以自行查阅相关的资料。

3. LCD 显示器

LCD 为英文 Liquid Crystal Display 的缩写, 即液晶显示器, 是一种数字显示技术。通过液晶和彩色过滤器过滤光源, 可以在平面面板上产生图像。

与传统的阴极射线管(CRT)相比, LCD 的优点在于占用空间小、低功耗、低辐射、无

闪烁和能降低视觉疲劳。与同大小的 CRT 相比,它的不足在于价格更加昂贵。

LCD 拥有许多传统的 CRT 显示技术所不具备的优点。它能够提供更加清晰的文本显示,而且屏幕无闪烁,从而能够有效降低长时间注视屏幕所产生的视觉疲劳。LCD 显示器的厚度一般不超过 10 英寸,对于华恒公司的 uClinux 开发板,LCD 是单显的,仅有黑白两种颜色。早在 1888 年,人们就发现液晶这一呈液体状的化学物质。像磁场中的金属一样,当受到外界电场影响时,其分子会产生精确的有序排列。如果对分子的排列加以适当的控制,液晶分子将会允许光线穿越。

无论是笔记本电脑还是桌面系统,采用的 LCD 显示屏都是由不同部分组成的分层结构。位于最后面的一层是由荧光物质组成的可以发射光线的背光层。背光层发出的光线在穿过第一层偏振过滤层之后,进入包含成千上万水晶液滴的液晶层。液晶层中的水晶液滴都被包含在细小的单元格结构中,一个或多个单元格构成屏幕上的一个像素。当 LCD 中的电极产生电场时,液晶分子就会产生扭曲,从而将穿越其中的光线进行有规则的折射,然后经过第二层过滤层的过滤在屏幕上显示出来。

对于简单的单色 LCD 显示器,如掌上电脑所使用的显示屏和开发板的显示屏,上述结构已经足够了。但对更加复杂的彩色显示器来说,还需要有专门处理彩色显示的色彩过滤层。通常,在彩色 LCD 面板中,每一个像素都是由三个液晶单元格构成,其中每一个单元格前面都分别有红色、绿色和蓝色的过滤器。这样,通过不同单元格的光线就可以在屏幕上显示出不同的颜色。

由于 LCD 是非主动发光器件,其速度低、效率差、对比度小。虽然能够显示清晰的文字,但是在快速显示图像时往往会产生阴影,影响视频的显示效果。因此,如今只被应用于需要黑白显示的掌上电脑、呼机或手机中。受 LCD 液晶层中实际单元格数量的影响,LCD 显示器一般只能提供固定的显示分辨率。如果用户要将 800×600 的分辨率提升到 1024×768 ,只能借助于特定软件的帮助才能实现模拟分辨率。

4. SPI 和 I2C 总线接口

任何一个微处理器都要与一定数量的部件和外围设备连接,但如果将各部件和每一种外围设备都分别用一组线路与 CPU 直接连接,那么连线将会错综复杂,甚至难以实现。为了简化硬件电路设计、简化系统结构,常用一组线路、配置适当的接口电路与各部件和外围设备连接,这组共用的连接线路被称为总线。采用总线结构便于部件和设备的扩充,尤其制定了统一的总线标准,则很容易使不同设备间实现互连。

微机中总线一般有内部总线、系统总线和外部总线。内部总线是微机内部各外围芯片与处理器之间的总线,用于芯片一级的互连。系统总线是微机中各插件板与系统板之间的总线,用于插件板一级的互连。外部总线则是微机 and 外部设备之间的总线,微机作为一种设备,通过该总线和其他设备进行信息与数据交换,它用于设备一级的互连。

另外,从广义上说,计算机通信方式可以分为并行通信和串行通信,相应的通信总线被称为并行总线和串行总线。并行通信速度快、实时性好,但由于占用的接口线多,不适用于小型化产品。而串行通信速度虽低,但在数据通信吞吐量不是很大的微处理电路中则显得更加简易、方便、灵活。串行通信一般可分为异步模式和同步模式。

随着微电子技术和计算机技术的发展,总线技术也在不断地发展和完善,而使计算机

总线技术种类繁多,各具特色。在上面所讲的 RS-232 就是一种外部总线的标准,而这里要介绍的是开发板的两个重要的内部总线。

(1) I2C 总线

I2C(Inter-IC)总线 10 多年前由 Philips 公司推出,是近年来在微电子通信控制领域广泛采用的一种新型总线标准。它是同步通信的一种特殊形式,具有接口线少,控制方式简化、器件封装形式小、通信速率较高等优点。在主从通信中,可以有多个 I2C 总线器件同时接到 I2C 总线上,通过地址来识别通信对象。

(2) SPI 总线

串行外围设备接口(SPI)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。

4.8 本章小结

在本章中,学习了常用的嵌入式 Linux 的开发平台的构成。首先,以华恒公司的开发套件为例,熟悉了一般的嵌入式的硬件平台。然后,介绍了一种典型的嵌入式 Linux 的版本——uClinux,对其运行机理进行了详细的分析。接着,介绍了如何自行构造一个嵌入式 Linux 的软件开发环境的步骤。最后,对开发板的中心器件——CPU 进行了介绍,并对开发板的外围设备进行了概述。通过本章的学习,对所要面对的软件和硬件系统有了基本的认识。由于嵌入式系统的多样性和复杂性,各个开发平台很难完全一致。但是,其开发的机理是有相通之处的。通过本章的学习,为学习下面的章节打好基础。

第5章 嵌入式 Linux 的开发

在熟悉了嵌入式 Linux 的开发平台后,就要学习嵌入式 Linux 的开发了。在这一章里,将学习如何构造和开发嵌入式 Linux 的系统,在嵌入式 Linux 环境下编写应用程序的详细方法和步骤,以及使用 GNU 工具对应用进行调试的方法。通过学习一个应用程序的编写、编译和调试的全过程,加深对理论学习的巩固。通过对第 1 章到第 4 章的学习,读者已经掌握如何在通用的 Linux 下编写程序,对嵌入式 Linux 软件平台也有一定了解。本章就开始学习如何构造嵌入式 Linux 系统和在现有的开发平台上编写自己的应用。

本章主要内容:

- 如何构造一个嵌入式 Linux 系统
- 嵌入式 Linux 的开发工具和开发模式
- 搭建开发平台
- 应用程序的编写和调试
- 开发中的注意事项

5.1 如何构造一个嵌入式 Linux 系统

大多数的 Linux 系统运行于 PC 平台,但是 Linux 也可在嵌入式系统中可靠地工作。下面先讲述嵌入式系统的概况,然后讨论 Linux 应用于嵌入式系统的问题。

5.1.1 嵌入式 Linux 系统的概述

1. 嵌入式系统的历史

那些用以控制设备的计算机,或叫嵌入式系统,很早就出现在人们周围。

在通信领域中,嵌入式系统早在 20 世纪 60 年代后期就被用来控制电话的电子式机械交换,并被称为“存储程控控制”系统。“计算机”一词在那时还不流行,所谓的存储程序是指那些放有程序和路由信息的内存。将这些控制逻辑存储起来而不是用硬件来实现是在观念上的一种真正突破。如今,这种工作机理是理所当然的了。

为适应每一个应用,这些计算机是被定做出来的(简言之,这些计算机是面向应用的),按现在的标准,它们有着奇怪的专用指令,以及与主要计算引擎集成在一起的 I/O 设

备,就像一批突变异种者。

微处理器通过一个小巧低价的并可以在大系统中像搭积木那样使用的 CPU“引擎”改变了这一情况。它利用一个基于被一条总线挂接在一起的不同外设所构建的严格的硬件体系结构来进行工作,并提供一个可以简化编程的通用目的编程模型。

此时,同硬件一起,软件也得到了发展。最初,只有一些简单的开发工具可供创建和调试软件。各工程项目的运行软件通常以信手涂鸦的方式编出来。由于编译器经常有很多错误而且也缺乏像样的调试器,这些软件差不多总是用汇编语言或宏语言来写。采用软件构建块和标准库的编程思想直到 20 世纪 70 年代中期才流行起来。

用于嵌入式系统的与构架无关的操作系统(OS)在 20 世纪 70 年代后期开始出现。它们中的许多是用汇编语言写成的,并且仅能用于为其编写的微处理器上。当这些微处理器变得过时的时候,它们使用的 OS 同时出现了。只能在新的处理器上重新写一遍才能运行。今天,许多这种早期的系统只不过成了人们模糊的记忆,例如 MTOS,现在就基本上没有人用了。当 C 语言出现后,OS 可以用一种高效的、稳定的和可移植的方式来编写。这种方式对使用和经营有直接的吸引力,因为它承载着人们当微处理器废弃不用时能保护软件投资的希望。直至今日,用 C 来编写 OS 仍然是一种标准。总之,软件的可复用性已经为人接受,而且正在很好地发挥作用。

许多用于嵌入式系统的商业操作系统在 20 世纪 80 年代获得了蓬勃发展。今天,已经有数以十计的商业性操作系统可供选择,出现了许多互相竞争的产品,如 VxWorks、pSOS、Neculeus 和 WindowsCE。

许多嵌入式系统根本就没有操作系统,只不过有一个控制环而已。对很简单的嵌入式系统来说,这可能已经足够了。不过,随着嵌入式系统在复杂性上的增长,一个操作系统显得重要起来。现实中确实有一些复杂得令人生畏的嵌入式系统,而且它们之所以变得复杂就因为它们的设计者坚持认为它们的系统不需要操作系统。

渐渐地,更多的嵌入式系统需要被连接到某些网络上,因而,在嵌入式系统中需要有网络协议栈的支持,甚至很多宾馆中的门把手都有一个连接到网络的微处理器。由于把网络协议栈添加到一个仅用控制环来实现的简单嵌入式系统带来了很大的复杂性,这就唤起了人们对一个操作系统的渴望。

除了各种商业性操作系统以外,还有多种私有操作系统。其中,有很多是涂鸦式写就的,像 Cisco 公司的 IOS 等。有些则源于对别的操作系统的改写,像很多网络产品都衍生于同一版本的伯克利 UNIX 操作系统,因为它有完整的网络支持能力;而还有一些则基于公共域 OS,比如 KA9Q 就来源于 PhilKarn。

作为候选的嵌入式操作系统,Linux 有一些吸引人的优势。它可以移植到多个有不同结构的 CPU 和硬件平台上,它有很好的稳定性、各种性能的升级能力,而且开发更容易。

如今,作为一种嵌入式的操作系统,Linux 已经越来越受到人们的注意,嵌入式 Linux 的产品数量也以惊人的速度增长,如图 5-1 和 5-2 所示,就是近年推出的基于嵌入式 Linux 的信息电器。

2. 嵌入式 Linux 的开发工具

在开发嵌入式系统中,有各种可用的工具是极为关键的一项。就像任何一个行业一

样，
不



而圆满地完成任务。在嵌入式系统开发的不同阶段，可能要用到



图 5-1 基于嵌入式 Linux 的 PDA,它使用 microwindows 作为图形界面

图 5-2 PocketLinux 公司推出的基于嵌入式 Linux 的掌上电脑

传统上,开发嵌入式系统的首选工具是仿真器。这是一块比较昂贵的设备,一般插于微处理器和它的总线之间的电路中,从而让开发者监视和控制所有输入和输出微处理器的各种活动和行为。在装配过程中,可能有一些困难,并且由于它们的侵入性,装上后可能造成不稳定的性能。尽管这样,它们却能在总线级上给出一个系统当前的运行状况的清晰描绘,而且使用户不必在硬件和软件接口最底层上的猜测。

以往一些工程项目在开发周期中的各个阶段中,依赖它作为主要的调试工具。不过,

一旦当编制的软件有能力支持一个串行口时,大量的调试可以不用 ICE 而使用别的方法来完成。同样,大部分新一代的嵌入式系统采用菜单式的微处理器设计。通信工作的启动代码使串行口尽快地工作,这意味着开发者能在没有 ICE 的情况下也能很好地进展。去掉了 ICE,从而降低了开发成本。一旦串行口工作起来,便能用于支持那些日渐复杂的开发工具的相关软件层。

Linux 基于 GNU C 编译器。GNU C 作为 GNU 工具集的一个组成部分,与源码级调试器 gdb 一起工作,提供了在开发一个嵌入式 Linux 系统中要用到的所有软件工具。下面是在为一个新的硬件开发一个新的嵌入式 Linux 系统时要用到的典型调试工具的序列和步骤:

- (1) 写出或移植一段启动代码。
- (2) 写一段代码。在串行口上输出像“Hello, World!”一类的简单字符串。
- (3) 移植 gdb 目标码使之能在串行口上工作。

这将允许同另一台正运行着 gdb 程序的 Linux 主机会话,只不过要告诉 gdb 是通过串行口调试该目标程序。gdb 通过串行口与被测试的计算机上的 gdb 目标码会话并给出全部 C 源码级的调试信息。也可以利用这一通信能力把附加的代码下载到 RAM 或闪存中。

(4) 借助 gdb,执行余下的所有硬件和软件的初始化代码,直到 Linux 内核开始接管为止。

(5) 一旦 Linux 内核启动后,上述的串行口就成为 Linux 的控制台端口并利用它的便利来进行后继开发过程。

(6) 如果目标平台运行的 Linux kernel 是具有完全功能的(即未经删减过功能),可以利用 gdb 或其图形化替代品(如 `xxgdb`)去调试应用进程。

关于使用 gcc 和 gdb 对嵌入式的目标系统进行编译和调试,在后面将专门开辟小节来论述。

3. 嵌入式系统和实时系统

嵌入式系统经常被错误地叫做实时系统,其实,大部分的系统并不是如此。它们中的大多数并不具备实时特性,实时性仅仅是相对的。实时严谨地定义应为硬实时,能在极短的时间(毫秒级)内响应,并以某种确定的方式处理事件。现在,许多硬实时功能正逐渐集中在 DSP 或 ASIC 的设计中,通过一些适当的硬件,如 FIFO、DMA 或其他专用硬件来实现。

对大多数系统来说,有 1 到 5 毫秒的实时响应时间应足够了。当然,另一种宽松的要求也是可以接受的,例如:Windows 98 的处理监视器崩溃画面的中断,要求必须在 4 微秒之内处理,占所有情况的 98%;而在 20 微秒之内处理的,占所有情况的 100%。

这些宽松的实时要求可以很容易达到。实现它们的过程中涉及到一些重要探讨的问

实时的严格性要求通常应由一个中断例程或内核中的现场驱动函数来处理以确保行为的一致性。当中断发生后,处理该中断所用的时间,即中断延迟,在很大程度上,由中断优先级与其他能临时屏蔽该中断的软件决定。实时系统中的中断必须被高效地设计和安排以确保满足时间上的要求,就像在其他 OS 中那样。在 Intel X86 处理器系列中,这项工作可以被扩充了实时性的 Linux 很好地处理(实时 Linux,即 RT-Linux,参看<http://www.rtLinux.org/>或本书第 9 章)。从本质上说,它提供了一个把 Linux 作为其后台任务而运行的中断处理调度器。一些关键的中断可以不为 Linux 其他部分所知而得到服务,因而,就有了对临界时间的控制权。这种做法提供了实时级别和时间限制性较为宽松的通用 Linux 级别之间的界面,并提供了一个与别的嵌入式操作系统类似的实时处理框架。从根本上讲,为满足实时性要求,采用了把实时性的关键代码段隔离开并进行高效的安排,然后对该段代码的处理结果再以更一般的方式(或许在进程级别上)做进一步处理。

4. 嵌入式系统特征

嵌入式系统是面向用户、面向产品、面向应用的。如果独立于应用自行发展,则会失去市场。嵌入式处理器的功耗、体积、成本、可靠性、速度、处理能力、电磁兼容性等方面均受到应用要求的制约,这些也是各半导体厂商竞争的热点。嵌入式处理器的应用软件是实现嵌入式系统功能的关键。软件要求被固化地存储在只读存储器中,代码要求高质量、高可靠性,系统软件(OS)的高实时性是基本要求。在制造工业、过程控制、通讯、仪器、仪表、汽车、船舶、航空、航天、军事装备、消费类产品等方面均是嵌入式计算机的应用领域。

过去有一种观点认为:如果某种应用没有用户界面,从而用户不能直接地和它交互,那么它就是嵌入式系统。这当然太简单化了。电梯控制系统是嵌入式系统,但却有一个用户界面:选择楼层的按钮和显示电梯正到达哪层的指示器。对于那些连入网络的嵌入式系统,如果该系统包含一个用于监视和控制的 Web 服务器,界面上的区别就更显模糊了。一个较好的定义应将重点放在描述该系统的重要功能或主要用途上。

由于 Linux 可以提供一个用以执行嵌入功能的基本内核以及各种用户界面元素,所以 Linux 有很强的通用特点。它能处理嵌入性任务和有关用户界面两方面的作业。可以把 Linux 看作如下的一个连续体:从一个只有内存管理,任务调度,定时器服务的缩简的微内核到一个支持各种文件系统和多种网络服务的完整服务器。

一个最小的嵌入式系统仅需如下基本组成部分:

- 一个用来引导系统的设施(或工具);
- 一个具备内存管理、进程管理和定时器服务的 Linux 微内核;
- 一个初始进程。

为让最小嵌入式系统变得有一定实用性,需加上一些东西:

- 硬件的驱动程序;
- 一个或几个应用进程以提供必要的应用功效。

随着对系统要求的增加,也许还要用到下面这些组件:

- 一个文件系统(或许放在 ROM 或 RAM 中);
- TCP/IP 网络协议栈;
- 一个磁盘用来存放半易失性数据和提供交换能力。

5. 选择硬件平台

挑选最好的硬件是一项很复杂的工作,充满着各种顾虑和干扰:公司的政策、各种成见、其他工程的影响以及缺乏完整、准确的信息。

当然,成本经常是一个关键性因素。当着眼成本时,一定要考虑产品的整体成本,而不要只看到 CPU。有时一个快速而廉价的 CPU 可能成了这个产品耗价的根源。如果用户只是一个听从命令的程序员,那么就只能对早已作好的决定撞运气了。如果是系统的设计者,就需要尽力制订一个合理的预算并且所选用的硬件要能很好地处理实时任务。

首先,要从实际中观察 CPU 到底需要多快才能把工作做好,然后把这个速度乘 3 后才是系统将要的 CPU 速度。因为 CPU 在理论上所能达到的能力到了现实中总是难以置信地大打折扣!而且也不要忽略缓存对系统的影响。

同时要计算出总线需要运行多快。如果有二级总线(像一条 PCI 总线那样),也要把它们包含进来。一条慢的或过多参与 DMA 传输的总线能够让一个快速的 CPU 慢得好像在爬行一样。

如果有集成外设的 CPU,那是很好的事,因为很少有硬件需要调试。为支持主流 CPU,它们的驱动程序经常是可用的。

6. 把 10 磅的 Linux 塞进 5 磅的口袋

对 Linux 的一个通常观点是:由于它太大,所以不宜用作嵌入式系统。这种观点不一定正确。面向 PC 机的 Linux 典型发布版有很多根本用不上的功能特征,甚至也超过了一个真正 PC 用户的需求。

首先,要把内核和各种任务分开来,标准的 Linux 内核总是长驻内存的。每一个要运行的应用程序需要从磁盘中装载到内存中,并在那里被执行。当该程序运行结束,它占用的内存被释放掉,也就是说,该程序被卸载掉。在一个嵌入式系统中,可能不存在一个磁盘。因系统的复杂程度和硬件的不同设计,有两种方法可以摆脱对一个磁盘的依赖性。在一个简单的系统中,当系统启动后,内核和各种应用进程均驻留在内存中,这是大多数传统嵌入式系统的工作方式。Linux 也支持这种方式。

使用 Linux 出现了另外一种方式。既然 Linux 有装载和卸载程序的能力,一个嵌入式系统可以利用这一点以节省 RAM。考虑一个比较典型的系统:有大约 8 兆到 16 兆的闪存和 8 兆 RAM,那么闪存可以被用作文件系统(如同我们使用的开发板)。用闪存驱动程序作为从闪存到文件系统的界面。作为一种选择,也可以用一个闪存磁盘,这是用闪存来摆脱系统对一个磁盘的依赖。使用这种方式的一个例子是 M-System (<http://www.m-systems.com/>) 中的 DiskOnChip 技术,该技术可以支持 160MB 的电子磁盘。所有应用程序以文件的形式被存放在闪存文件系统中,并在必要时被装载到内存中。这种“用到时才装载”的能力是一个很强大的特征。嵌入式 Linux 系统具有以下各种能力:

(1) 允许系统启动后抛弃那些初始化代码空间。

典型地,Linux 用到很多运行在内核外部的应用工具程序。它们通常仅在系统初始化期间运行一次,以后再也用不到,而且它们以互斥的方式,一个接一个地依序运行。这样,随着系统的启动,那些在启动过程中被使用的程序,随着系统的启动,被依次加载到内存中,并依次运行。

以很好地节省内存,特别是处理像网络栈那样的对象时,因为这些对象一经配置便永不更变。

提示:互斥是用来控制多任务对共享数据进行串行访问的同步机制。在多任务应用中,当两个或多个任务同时访问共享数据时,可能会造成数据破坏。互斥使它们串行地访问数据,从而达到保护数据的目的。

(2) 如果 Linux 支持可动态装卸模块的特征被包含在内核中,那么不仅各种应用程序,而且驱动程序也可被动态装卸。

这样,系统能够检查硬件环境并有选择性地仅仅装载那些适应当前硬件环境的软件。这可以降低让一个程序以占用更多闪存的代价处理众多硬件变数的复杂性。


(3) 系统的升级更加标准化(模块化),常常可以在系统运行过程中升级一些应用和驱动程序。

(4) 配置信息和运行时间参数可作为数据文件存放在闪存中。

7. 无虚拟内存交换

Linux 的另外一个特性就是虚拟内存交换。这一特性将应用程序的编写引入歧途,应用程序的内存需求量可以无限制地上升,因为操作系统在磁盘中提供了交换空间。而在一个无盘的嵌入式系统中,这种特性就用不上了。

如此强大的功能,在嵌入式系统中其实是无用武之地的。事实上,在一个严格的实时系统中,可能并不需要这种特性,因为它会导致定时功能的失控。与其他嵌入式系统一样,软件的设计必须很紧凑,以适应较小的物理内存。

 注意:这取决于用户的 CPU 体系。

比较明智的做法是保留 Linux 的这段代码,毕竟砍掉这些代码,还是要付出一些工作量的。但是,这些代码依然有理由被保留,因为它们能够支持代码共享,多个进程可以共享某一软件的同拷贝。如果无此功能,那么每个程序都必须拥有库程序的独立拷贝,例如:printf。

将交换空间大小简单地置为零,就可以关掉系统虚拟内存的分页交换机制了。当程序要求内存大于实际的内存时,系统的表现就如同交换空间溢出时一样,程序不会被加载,或者当要求过多内存时,malloc 调用就会失败。

在许多 CPU 体系中,虚存机制提供的内存管理可使进程的地址空间相互隔离。一般在嵌入式系统中不是这样,地址空间是简单的,平坦的。Linux 的虚存机制使出错的进程不致影响整个系统。在许多嵌入式系统中,因为效率原因而设置的全局数据,同时被几个进程所共享。这在 Linux 中也存在,那就是内存共享,它可以经过设置,使某一段内存成为共享内存。

8. 文件系统

许多嵌入式系统不存在一个磁盘或者一个文件系统。没有它们中的任何一个, Linux 也可以运行。正如前面提到的那样,应用任务可以随同内核一起被编译并在启动时作为一个映像被加载。对简单系统来说,这已经胜任。不过,它却缺少前面描述到的各种灵

活性。

事实上,如果观察过许多商业性嵌入式系统,会发现他们把文件系统作为可选项来提供。大部分要么是一个私人拥有专门的文件系统,要么是一个与 MS-DOS 兼容的文件系统。Linux 不但支持许多其他文件系统,也支持 MS-DOS 兼容的文件系统。通常推荐使用除 MS-DOS 兼容文件系统以外的其他文件系统,因为它们有较优的健壮性和容错性。

文件系统可以放在一个传统磁盘驱动器中、闪存中或任何可用的其他介质中。同样,一个小的 RAM 盘常常可以很好地存放易失性文件。闪存被分隔成很多小块,并被很好地组织起来。它们中可能有一个引导块,存放了 CPU 上电后运行的第一个软件,可能存放的就是 Linux 的引导程序。余下的闪存块可以被用作文件系统。Linux 的内核可以被引导程序从闪存中拷贝到 RAM 中;或者作为另一种选择,把内核放在闪存的一个独立区中并从那里直接运行。

对一些系统来说,另一可行的选择是包含一个廉价的 CD-ROM 驱动。它可能比闪存还要便宜,并且借助更换 CD-ROM 盘片就可很容易地得到升级。通过这种方法,Linux 只需从 CD-ROM 启动,就可以像对一个硬盘那样从该 CD-ROM 中获得所有用到的程序。

最后,对于网络上的嵌入式系统而言,Linux 支持 NFS(Network File System,网络文件系统),它打开了在一个网络支持系统中实现各种增值特征的通道。首先它允许通过网络加载各种应用程序。对于 Linux 而言,每一个嵌入式系统上的软件可以从一个公用的服务器加载,这个特性在控制软件的修订或升级中是很重要的。在系统运行的过程中,导入和导出数据、配置系统、备份状态信息也很有用。对用户监控而言,这是一个非常强大的特征。举例来说,一个嵌入式系统可能装配了一个 RAM 盘,它包含着与系统当前状态的更新维持一致的状态文件。那么别的嵌入式系统仅需通过网络把这个 RAM 盘作为远程磁盘挂载,便可以访问那些位于远端 RAM 盘中的状态文件。这也允许在另一台机器上的 Web 服务器借助简单的 CGI 脚本来访问那些状态信息。运行在其他机器上的应用程序包能够很容易地访问这些数。对更复杂的监控,像 MatLab (<http://www.mathworks.com/products/matlab/>)这样的应用程序包,能很容易地用图形化来显示在一个系统操作者的 PC 或工作站上的系统操作。

9. 引导内核

当一个微处理器最初启动时,它首先执行在一个预定地址处的指令。通常在这个位置是只读内存,其中存放着系统初始化或引导程序。在 PC 中,它就是 BIOS。这些程序要执行低级的 CPU 初始化并配置其他硬件。接着,BIOS 判断出哪一个磁盘包含有操作系统,再把 OS(操作系统)拷贝到 RAM 中,并把控制权交给 OS。实际上,整个过程远非这么简单,不过理解这么多已经足够。运行在 PC 上的 Linux 系统,依赖于该 PC 的 BIOS 来提供这些配置和 OS 加载功能。

在一个嵌入式系统里,常常没有上述的 BIOS,这样就需要去提供等价的启动代码。还好,一个嵌入式系统的 BIOS 并不需要像 PC BIOS 引导程序那样有那么多的灵活性,因为它通常仅需处理一种硬件配置方案。这些代码往往比较简单,是一些把特定的数写入指定硬件寄存器的指令序列。这是很关键的代码,因为这些数值必须要符合硬件并且按特定顺序来完成。在大多数情况下,这些代码中有一个最小化的加电自检模块用以检查

内存,让一些 LED 闪现一下,也可能探测一些其他让 Linux 操作系统启动和运行的必要硬件。这些启动代码是高度硬件专用性的,因而不具移植性。

有幸的是,大多数系统为核心微处理器和内存使用了菜单式的硬件设计。典型地,芯片制造商有一个可供设计参考的演示板,新设计多少可以从中直接拷贝一些。对这些菜单式的设计,经常有现成的启动代码可用,而且可以很容易地被修改以适应需要。很少会遇到有需要从头开始编写的启动代码。

为了测试启动代码,可以使用自带仿真内存的电路仿真器(ICE),这里的仿真内存用以替换目标内存。把待测代码加载到仿真器中并通过仿真器调试它。如果没有可用的仿真器,也可以跳过这一步,不过需要一个较长的调试周期。

这些代码最终要从非易失性存储器(ROM)中运行,通常是用闪存或 EPROM 芯片,要想办法把这些代码放进前述芯片中。放入的具体方法依赖于目标硬件和工具。一个常见的方法是把闪存或 EPROM 芯片插入到一个 EPROM 或闪存“烧结炉”中。这种方法将把程序“烧入”芯片中。把芯片插入到目标板上的一个插槽中,打开电源。这种方法要求在板子上具有插槽化部分。然而有些设备包结构不允许被插槽化。

另一种办法是通过一个 JTAG 接口。一些芯片包含一个 JTAG 接口,从而可以对芯片编程。这是一种最简便的办法。芯片可以被永久地焊接到板子上。一段电缆从板子上的 JTAG 连接器(通常是一个 PC 卡)连接着一个 JTAG 接口。接下来要求在操纵 JTAG 接口的 PC 上做一些用户定制性编程。在仅需较少运行量的产品中也可以使用这种方法。

10. 健壮性

嵌入式 Linux 的这个特性是显而易见的,作为一种选择,Linux 已被普遍地认为能够在 PC 平台上可靠地、稳定地运行。嵌入式内核自身有多稳定?对于大多数微处理器来说,Linux 是很好用的。将 Linux 移植到新的微处理器体系也是非常迅捷的。一般是将其移植到一种新型的目标板,这种新型的目标板包含有独特的外设,当然还有 CPU。

幸运的是,大部分的内核代码都是相同的。因为它们与微处理器无关,所以移植的工作都集中在那些不同的部分,通常是一些存储器管理及中断处理程序。一旦完成,它们往往是非常稳定的。如同前面谈到的,引导的过程随硬件的变化而变化,所以一定要周密地计划一番。

设备驱动程序虽然变化多端,但其中一些已相当稳定。同时选择也不算太多,一旦离开 PC 平台,就只有自己去写了。所幸的是,已有许多既有的设备驱动程序,总能找到一个近似的去修改它。驱动程序的接口是明确定义的。大多数驱动程序之间都是相似的,所以移植一个磁盘、网络和串行口驱动程序,从一个设备到另一种,也不是太难。

其实 Linux 与那些顶顶大名的商业操作系统一样稳定。总的来说,关于这些操作系统(包括 Linux)的问题都源于对系统工作策略的误解,而不是纯代码问题或基本设计错误。大量操作系统的问题故事,在这里没有必要再重提。而 Linux 的优点,就在于其源码是公开的,并有很好的注释和完整的文档说明。从而,也就拥有了控制与解决一切问题的能力基础。

11. 嵌入式 Linux 的不足

嵌入式 Linux 当然有它的不足。比如,它很占内存,尽管不比一些商业竞争者的情况更坏,但可以通过消减一些不必要的功能来改善,也有可能得不偿失,因为很可能产生比较严重的问题。

大多数 Linux 应用程序都会使用虚拟内存交换,这在很多嵌入式系统中是一种非确定因素,所以,不要假定任何一个无盘嵌入式系统能够运行什么 Linux 应用程序。低等级的、内核级的调试工具仍然不是很好用,而且调试仍然是以打印语句为主。

Linux 是一种极具适用性的操作系统,可是嵌入式系统在通常情况下是不具备这种性质的。它们是针对特定的用途,进行过仔细优化的。Linux 的这种适用性倾向,保持了系统的通用性和多变性。同时也是一个奢侈的目标,付出的代价很高,需要增添许多额外的工作,会有许多附加的程序产生,从而增加了软件包的体积,有时还会以降低性能为代价。来看一个常见例子:某个在网络接口上配置 IP 地址的配置程序,在 Linux 下一般是由启动脚本中的 ifconfig 完成的,这是一个大小约 28K 左右的程序。但是,在嵌入式系统中,其实只用几行代码就可以完成。因为它仅仅负责根据配置文件中的内容初始化一些相应的数据结构。

5.1.2 关于嵌入式 Linux 开发的一些问题和概念

前面已概括性地讲到了嵌入式 Linux 开发的步骤,下面来区分一些概念。

1. “模块”和“驱动程序”

在嵌入式 Linux 的开发中,“模块”和“驱动程序”两个词组有时会搞混淆。一个模块是一段代码,它可以通过使用“insmod”命令来插入内核中。装载一个模块和将其动态链接到内核是同样重要的。在此时,它是内核的一部分。驱动程序是一个使用特定接口的模块。通过该接口,应用程序可以执行读、写等面向文件的操作,然而,某些驱动程序并不是模块。

2. 驱动程序和模块的动态与静态

在 Linux 下,驱动程序是可以被直接编译进内核的。它们同内核一同编译,并同其一起链接,在内核启动时,被系统装载。对于经常要使用的设备来讲(如网卡),可以采用该方法。在此情况下,该驱动是长驻内核的,可以称其为静态驱动程序。

同时,Linux 也支持动态的驱动程序的转载(如使用 insmod 命令)。一旦装载,它们同静态的驱动程序没有什么区别。某些驱动程序可以采用两种方法。

5.1.3 构造一个嵌入式 Linux 的实例

下面从精简内核、系统启动、驱动程序和将 X Window 换成 MicroWindows 四个步骤,介绍嵌入式 Linux 的实际开发。参考该步骤就可以自己去构建一个初步的嵌入式 Linux

的应用系统。

1. 精简内核

构造内核的常用命令包括: `make config`、`dep`、`clean`、`mrproper`、`zImage`、`bzImage`、`modules` 和 `modules_install`。命令的具体使用方法参考 `man` 手册。

现在举个例子:

使用 Mandrake 版本 Linux 内附的 2.2.15 内核。对内核代码不做任何修改,完全只靠修改文件组合得到这些数据。

使用 `make config` 把所有可以拿掉的选项都拿走。

不要 `floppy`;不要 `SMP`、`MTRR`;不要 `Networking`、`SCSI`;把所有的块设备移除,只留下 `old IDE device`。把所有的字符设备都移除;把所有的 `filesystem` 移除,只留下 `minix`;不要 `sound` 支援。这样做之后,可以得到一个 188K 的核心。如果还想减小内核,可以把 `./Makefile`、`./arch/i386/kernel/` 和 `makefile` 文件中的 `-O3`、`-O2` 用 `-Os` 取代。

这样一来,整个核心变小了 9K,成为 179K。不过这个核心恐怕很难发挥 Linux 的功能,因此可以把网络加回去。把 `General` 中的 `network support` 加回去,重新编译,核心变成 189 K。10K 就加上一个 `TCP/IP` 栈。有 `stack` 没有 `driver` 也是枉然,所以可以把嵌入式开发板常用的 `RTL8139` 的驱动程序加回去,就有了 195K。如果需要 `DOS` 文件系统,那大小成为 213K。如果 `minix` 用 `ext2` 换代,则大小成长至 222K。

Linux 所需的内存大约在 600K~800K 之间。1MB 内存就可以启动了,但功能不完善,因为连载入 C 程序库都有困难。2MB 内存应该就可以做点事了,但要有 4MB 以上才可以执行一个比较完整的系统。

因为 Linux 的文件系统相当大,大约在 230K 左右,占了 1/3 的体积。内存管理占了 80K,和核心其他部分的总和差不多。`TCP/IP` 栈占了 65K,驱动程序占了 120K。`SysV IPC` 占了 21K,必要的话可以拿掉,核心文件可以再小个 10K 左右。如果要裁剪核心大小,应该动那里呢?显然是文件系统。Linux 的虚拟文件系统(VFS)简化了文件系统的设计,`buffer cache` 和 `directory cache` 增加了系统的效率,但这些对于嵌入式系统用处不大。如果可以把它们拿掉,核心马上缩小 20K 左右。如果跳过整个 VFS,直接将文件系统写成一个 `driver` 的型式,可以将 230K 缩减至 50K 左右,整个核心缩到 100K 左右。

2. 系统启动

系统的启动顺序及相关文件仍在核心源码目录下,看以下几个文件:

```
./arch/$ARCH/boot/bootsect.s
```

```
./arch/$ARCH/boot/setup.s
```

```
./init/main.c
```

`bootsect.S` 及 `setup.S`

这个程序是 Linux kernel 的第一个程序,包括了 Linux 自身的自启动程序,但是在说明这个程序前,必须先说明一般 IBM PC 开机时的过程(此处的开机是指“打开 PC 的电源”)。

打开电源时,PC 一般是由内存中地址 `FFFF:0000` (`FFFF` 为段地址, `0000` 为偏移量)开始执行,这个地址一定在 ROM BIOS 中,ROM BIOS 一般是在 `FE000h` 到 `FFFFFFh`

中。而此处的内容则是一个 jump 指令,跳转到另一个位于 ROM BIOS 中的位置,开始执行一系列的动作。

紧接着系统测试码之后,控制权会转移给 ROM 中的启动程序 ROM bootstrap routine。这个程序会将磁盘上的第零轨第零扇区读入内存中,至于读到内存的哪里呢?其实是绝对位置 07C0:0000 即 07C00h 处,这是 IBM 系列 PC 的特性。而位于 Linux 开机磁盘的 boot sector 上的,正是 Linux 的 bootsect 程序。

在这里,把大家所熟知的 MS DOS 与 Linux 的开机部分做个粗浅的比较。MS DOS 由位于磁盘上 boot sector 的 boot 程序负责把 IO.SYS 载入内存中,而 IO.SYS 负责把 DOS 的内核——MSDOS.SYS 载入内存。而 Linux 则是由位于 boot sector 的 bootsect 程序负责把 setup 及 Linux 的 kernel 载入内存中,再将控制权交给 setup。

3. 驱动程序

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。设备驱动程序所提供的入口点,在设备驱动程序初始化的时候向系统进行登记,以便系统在适当的时候调用。Linux 系统里,通过调用 register_chrdev 向系统注册字符型设备驱动程序。

在 Linux 中,除了直接修改系统核心的源代码,把设备驱动程序加入核心外,还可以把设备驱动程序作为可加载的模块,由系统管理员动态地加载它,使之成为核心的一部分。也可以由系统管理员把已加载的模块动态地卸载下来。Linux 中,模块可以用 C 语言编写,用 gcc 编译成目标文件(不进行链接,作为 *.o 文件存在)。为此,需要在 gcc 命令行里加上 -c 的参数(见第 3 章相关内容)。成功地向系统注册了设备驱动程序后(调用 register_chrdev 成功后),就可以用 mknod 命令把设备映射为一个特别文件。其他程序使用这个设备的时候,只要对此特别文件进行操作就行了。

4. 将 X Window 换成 MicroWindows

MicroWindows 是嵌入式 Linux 系统中使用得很广泛的一种图形用户界面(关于它的详细介绍见第 10 章)。MicroWindows 是使用分层结构的设计方法,允许改变不同的层来适应实际的应用。在最底一层,提供了屏幕、鼠标/触摸屏和键盘的驱动,使程序能访问实际的硬件设备和其他用户定制设备。在中间一层,有一个图形引擎,提供了绘制线条、区域填充、绘制多边形、裁剪和使用颜色模式的方法。在最上一层,提供了不同的 API 给图形应用程序使用。这些 API 可以提供或不提供桌面和窗口外形。目前,MicroWindows 支持 Windows Win32/WinCE GDI 和 Nano-X API。这些 API 提供了 Win32 和 X 窗口系统的紧密兼容性,使别的应用程序能很容易移植到 MicroWindows 上。

关于嵌入式 Linux 的图形界面,在后面会有专题论述。

5.2 嵌入式 Linux 的应用程序的编译和调试

在上面一节中了解了嵌入式 Linux 系统的构建方法。在接下来的本小节中,我们将

讨论嵌入式 Linux 系统的应用程序的编译、链接和重新定址的问题,然后再学习应用程序的调试。

5.2.1 嵌入式 Linux 的应用程序

在嵌入式系统中,由一个源文件变成最终可执行的二进制文件,一般要经过三个过程,即编译、链接和重新定位。通过编译或者汇编工具,将源代码变成目标文件。由于目标文件往往不只一个,所以需要链接工具将它们链接成另外一个目标文件,可以称其为“可重新定位程序”。经过定址工具,将“可重新定位程序”变成最终的可执行文件。整个过程的流程图,如图 5-3 所示。

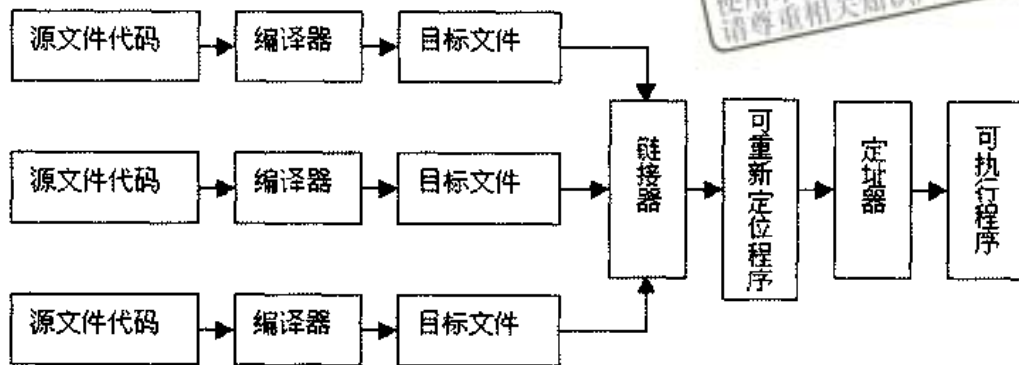


图 5-3 嵌入式系统应用软件编译、链接流程

稍微有点开发经验的人都知道,一般的嵌入式系统的开发,通常采用的是主—从模式。通过串行口(还可以有网口),使目标机和宿主机相连接。在开发中,每一个步骤都是在通用的计算机上执行软件转换的过程。必须清楚的是:图 5-3 中所标注的编译器、链接器和定址器都是在宿主机上(一般是 PC 上,对于嵌入式 Linux 的开发而言,还必须在 PC 上运行 Linux 操作系统)运行的,而最终经过编译—链接—重新定位所得到的二进制可执行文件却是在目标机上运行的,所以称其为“交叉调试”。

在嵌入式 Linux 的开发中,理所当然的是使用 GNU 的系列工具。在上一章对 uClinux 操作系统的简介中,简要介绍了 GNU 的开发套件。GNU 开发套件作为通用的 Linux 开放套件,包括一系列的开发调试工具。主要组件有:

- (1) gcc: 编译器,可以做成交叉编译的形式,即在宿主机上开发编译目标上可运行的二进制文件;
- (2) Binutils: 一些辅助工具,包括 objdump(可以反编译二进制文件)、as(汇编编译器)、ld(连接器)等等;
- (3) gdb: 调试器,可使用多种交叉调试方式,如 gdb-bdm(背景调试工具)和 gdbserve(使用以太网调试)。

对于嵌入式系统而言,GNU 工具套件是很好的选择。它们廉价稳定,有相当高的可移植性和优秀的跨平台特性。下面来学习在嵌入式 Linux 环境下,使用 GNU 的编译和链接工具的方法。

5.2.2 gcc 在嵌入式 Linux 系统中的使用

编译器的作用是将用高级语言或者汇编语言写就的源代码翻译成处理器上等效的一系列操作命令。针对嵌入式系统来说,其编译器数不胜数。其中,gcc 和汇编器 as 是非常优秀的编译工具,而且免费。当作为交叉编译工具使用时,gcc 支持很多种的平台和宿主计算机—目标机的组合,当然也包括所使用的 X86PC + 红帽 Linux—Motorola68328 + uClinux 的组合形式。

编译器的输出被称为目标文件。在开发平台中,如第 4 章所讲,它们的格式都是标准化的,如 coff(common object file format)、elf(executive linked file)、flat 格式等。

对于任何嵌入式系统而言,有一个高效的编译器、链接器和调试器是非常重要的。gcc 不仅在桌面领域中表现出色,还可以为嵌入式系统编译出高质量的代码。gcc 的优秀功能在此不多加阐述,在本节仅讲述 gcc 在嵌入式领域中的突出表现。

1. 语法和程序运行潜在错误的校验

gcc 提供了一些命令行选项,用来控制对程序的信任程度。例如,“-Wall”参数就让 gcc 用来警告用户任何可疑的东西。其他的一些有用的参数包括:

-W format(检验 printf()调用的参数)

-Wconversion(如果某个负整数的常数表达式被隐式的转换成为了一个无符号型的话,给出警告)

2. 内联汇编代码

针对用 C/C++ 语言编写的嵌入式系统的代码,gcc 提供了一个功能强大的语法结构,使汇编语言可以嵌入到其中去。在最基本的情况下,将汇编语言的指令插入到一块 C 语言的代码中去。如下所示:

```
void set _imask _to _6( void )
{
printf("switching to interrupt mask level 6. \n");
__asm __("andi #0xf8, sr");
__asm __("ori #6, sr");
printf("Interrupt mask level is now 6. \n");
}
```

在函数 set _imask _to _6(void)中,就插入了用汇编语言写成的语句:

```
__asm __("andi #0xf8, sr");
```

当然这仅仅是个很小的例子。gcc 编译器同时可以使在汇编语言的声明中很安全地应用 C 语言的目标,而不是让用户预测应该用哪个寄存器来存储所需的值。

在 gcc 的说明文件中,给出了一个例子:

如果想尽可能稳健地使用 68881 芯片的 fsinx 指令,写出的代码有最大的可移植性,可以这样做:


```
__asm__("fsinx %1, %0" : "=f"(result) : "f"(angle));
```

其中，“=f”和“f”为调用操作数约束项。它们用来告诉 gcc 要如何产生 %0 和 %1 的表达式，使操作码能正确地发挥作用，并告诉 gcc 操作数有何副作用。在该例子中，它们告诉 gcc 为了存储角度和结果变量，需用浮点寄存器，而且 fsinx 将返回值放在结果变量中。

操作数的约束项使 C/C++ 和汇编语言的混合语句能够正常地工作，即使在诸如优化等级的变化和其他编译器设置的改变可能使得这些语句功能有所改变时也是如此。在 gcc 中还有很多操作数和约束选项，在 gcc 手册的扩展 asm 部分中还提供了一些别的例子。

gcc 的内联汇编还有一个优点，就是它并不打断编译器正常的优化过程。为了说明这点，考虑下面的一个简单的函数：

```
int foo(int a)
{
    int b = 10;
    a = 20;

    __asm__("mov %1, %0" : "=r"(a) : "r"(b)); /* a = b */
    return a;
}
```

在此，汇编语言仅仅将 b 拷贝到 a，所以返回值总是 10。如果优化选项不打开，gcc 所产生的代码如下所示（本例中使用 Hitachi SH-2 的代码）：

```
_foo:

    mov.l r14,@-r15
    add #-8,r15
    mov r15,r14
    mov.l r4,@r14
    mov #10,r1
    mov.l r1,@(4,r14)
    mov #20,r1
    mov.l r1,@r14
    mov.l @(4,r14),r2
    mov r2,r2
    mov.l r2,@r14
    mov.l @r14,r1
    mov r1,r0
    bra L1
    nop
    .align 2
L1: add #8,r14
    mov r14,r15
```

```
mov.l @r15+, r14
rts
```

然而,加上优化等级(-O3 -fomit-frame-pointer)后,代码就变得大不一样了:

```
_foo:

mov #10, r1
mov r1, r0

rts
```



换言之,gcc 编译器得出了结论,即惟一可能的返回值是 10,这意味着它“懂”得所提供的汇编代码,并将它利用了起来。对于商业编译器而言,这样优秀的表现是不多的。

为什么 gcc 不首先就将 10 放入寄存器 r0 中间呢? 因为 gcc 围绕了用户提供的汇编语言代码进行了优化,但是并不会忽略它。也就是讲,mov r1, r0 命令的存在是由于内联汇编声明所造成的,而不是 gcc 把它放置在那里的。

3. 控制汇编代码中使用的名称

偶尔需要让 C 语言来获取某个汇编语言的目标文件,但是此时目标文件并不是同 C 兼容的格式。下列的代码使 C 语句可以使用一个叫 foo_in_C 的符号来代替一个被错误命名的汇编语言符号 foo_data(它少了一个起始的下划线,不能正常地被 C 语言获取)。

```
extern int foo_in_C asm ("foo_data");
在声明时也使用了类似的语法:
int bar asm ("bar_none");
extern int foo( void ) asm ("assembly_foo");
```

第一个声明使 gcc 创建了一个叫“bar”的 C 语言的符号,但是产生的汇编代码仍然称之为 bar_none(前面没有带下划线),而不是通常用的_bar 形式。第二个声明语句使 gcc 无论在 C 函数 foo()被调用或定义时,总是使用 assembly_foo(而不是foo)这个名字。

4. 段说明

许多商业交叉编译器允许通过使用命令行选项来指定目标的内存段。例如,为了将某个模块的所有全局声明常量放到一个叫“myconst”的段中,使用下列类似的命令:

```
my_commercial_c_compiler -C"myconst" main.c
```

该方法的毛病在于,它不可见地改变了常量关键字的作用,使将来在模块中添加常量时,有可能会在错误的内存空间中运行。它强迫用户将 myconst 和通用的常量声明分隔到不同的模块中,当项目成熟后,使维护系统的工作让人头疼。

另一方面,gcc 的方法是在预先声明的基础上指定分配段,实现的方法是使用它的扩展段属性语言:

```
const int put_this_in_rom __attribute__((section("myconst")));
const int put_this_in_flash __attribute__((section("myflash")));
```

同命令行方式相比,该方法允许将与某个功能相关的所有声明放置到同一个源模块中,而不论它们在目标系统的内存映射图中的位置如何。

段属性可以用来构造数据表。例如,对于 eCos 操作系统来讲(eCos 即开放源代码的嵌入式可设置的操作系统的简称),可以使用段属性将所有的设备信息结构(其中有一个放在每个设备驱动源代码模块中)放到一个叫 devtab 的段中,在操作系统运行时就对其进行分析。使用该方法,加入或删除某个驱动如同在应用程序中加入或删除一个模块那么简单,不需要去修正“主控设备驱动列表”。

5. 中断服务例程

对大多数的目标处理器来说,gcc 并不提供一个 interrupt_handler 的属性。这意味着在绝大多数情况下,并不需要用 C 语言来写整个的 ISR(中断服务例程),因为 gcc 并不知道用“异常返回”的操作码(例如 RTE)从函数中返回。

这个局限性并不像听起来那么严重。通常的做法是写一小段汇编语言的代码储存寄存器、调用 C 代码,然后以 RTE 为返回值并退出。如下所示:

```
void isr_C( void )
{
    /* 用 C 写成的程序 */
    ...
}

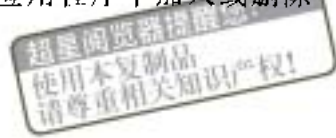
__asm__(

.global _isr
_isr:

    /* 压栈
    * 保留余下部分
    */
    push r0
    push r1
    ...

    /* 调用我们功能部分 isr_C */
    jsr _isr_C

    /* 清空堆栈,返回 */
    ...
    pop r1
    pop r0
    rte
```



");

为什么 gcc 不为 ISR 程序的编写者着想,把事情变得简单些呢?原因可能是由于 gcc 的主要目的(虽然该目的是经常很快的变化)在于为桌面的工作站产生代码,而工作站是不需要中断服务例程的,而且修改 gcc 的堆栈帧和对寄存器管理的实现不是件小事情,所以绝大多数 GNU 用户更乐意在这儿或那儿写一小段汇编语言的代码,而不愿意看到出现额外的问题。这样做是有其可取之处的。

6. 保留寄存器

有些时候,最好用户能告诉编译器绝不要使用某个特定的寄存器,因为有些汇编语言函数需要在 C 语言函数的调用之下保存该寄存器的值。gcc 是可以做到这一点的。仅仅将 `-ffixed-REG` 放在命令行之间即可,如下所示:

```
gcc -ffixed-a7 myprogram.c
```

如果实时操作系统(RTOS)保留了一到两个寄存器给自己使用时,这个保留寄存器的特性将十分有用。

7. 字符串形式的函数名

标准的 C 预编译宏为在运行时刻显示输出结果提供了所需的最少信息。例如,可以描述一个 `__FILE__` 的性质为 `__DATE__` 和 `__TIME__`,但是不可能有别的更多的信息了。

gcc 对这两者都支持,并增加了它自己的两种属性: `__FUNCTION__` 和 `__PRETTY_FUNCTION__`。这两个参数在 C 语言的环境下都产生同样的结果,但是后者在 C++ 模块中出现时,可以提供更多的信息。

看下面的例子:

```
printf("The function %s in file %s, was compiled on: %s. \n",  
__PRETTY_FUNCTION__,  
__FILE__, __DATE__);
```

产生的结果有两种情况:

```
The function foo, in file foo.c, was compiled on: Feb 10 1999.
```

或

```
The function int c::foo, in file foo.cpp, was compiled on: Feb 10 1999.
```

8. 调试信息

如要告诉 gcc 在输出的目标文件中包含调试信息,就可用“-g”标志来告诉 gcc。

```
gcc -g main.c
```

如果没有-g,应用程序可以运行,但是不能被调试。

5.2.3 GNU 的链接工具——ld

GNU 链接器 ld 是一个功能强大的应用程序,但是在大多数的情况下,没有必要调用

ld 的文件目录,因为 gcc 自动将其调用了,除非使用了 -c(仅编译)选项。

如同许多的商业链接器一样,ld 的绝大多数功能由链接命令文件控制(即第 4 章中讲的 .ld 文件)。这种文件都是文本文件,它们描述了诸如最终输出文件的内存组织之类的信息。下面给出一个链接器命令脚本文件的样例,它包含了一个链接命令脚本的示范。总而言之,该脚本定义了 4 个内存区域,被称为 vect、rom、ram 和 cache,紧接着还有输出段 vect、text、bss、init 和 stack。例 5-1 是一个链接器的命令行的脚本样例。

例 5-1 一个链接器命令行的脚本样例

```
/* 下列是需要链接的文件列(其他的文件在命令行中给出) */  
INPUT(libc.a libg.a libgcc.a libc.a libgcc.a)
```

```
/* 输出模式(在命令行中可以不予考虑) */  
OUTPUT_FORMAT("coff-sh")
```

```
/* 输出文件名(在命令行中可以不予考虑) */  
OUTPUT_FILENAME("main.out")
```

/* 我们程序的入口点。除了保证 S7 记录的正确性外,没什么太大的用处。这是因为在绝大多数嵌入式系统,重启向量实际上已经定义了“入口点” */

```
ENTRY(_start)
```

```
/* 我们程序的内存段分布 */
```

```
MEMORY
```

```
{
```

```
  vect : o = 0, l = 1k
```

```
  rom : o = 0x400, l = 127k
```

```
  ram : o = 0x400000, l = 128k
```

```
  cache : o = 0xffff000, l = 4k
```

```
}
```

```
/* 我们如何组织在每个模块中定义的内存段 */
```

```
SECTIONS
```

```
{
```

```
/* 中断向量表 */
```

```
  .vect :
```

```
{
```

```
  __vect_start = .;
```

```
* (.vect);
__vect_end = .;
} > vect

/* 代码和常量 */
.text :
{
__text_start = .;
* (.text)
* (.strings)
__text_end = .;
} > rom

/* 未初始化的数据 */
.bss :
{
__bss_start = .;
* (.bss)
* (COMMON)
__bss_end = .;
} > ram

/* 已经初始化的数据 */
.init : AT (__text_end)
{
__data_start = .;
* (.data)
__data_end = .;
} > ram

/* 应用堆栈 */
.stack :
{
__stack_start = .;
* (.stack)
__stack_end = .;
} > ram
```



ld 的相关特征和命令参数请见 ld 的联机参考。

除非指定让它进行别的操作,ld 总会使用一个默认的命令文件。为了让 ld 使用指定的命令文件而不是它默认的命令文件,在 gcc 编译时,加上一个这样的命令参数:

```
-Wl,T<filename>
```

接下来讲解例 5-1 中相关的内容。

1. OUTPUT _ FORMAT 命令

该命令控制了输出文件的格式。ld 支持很多文件格式,包括 S-格式、二进制格式、Intel 十六进制 Hex 和一些供调试用的格式,如 COFF(对 SH-2 的芯片是 coff-sh,对 CPU32 芯片是 coff-m68k 等)。

ld 工具生成文件格式的能力得源于一个名为 bfd 的目录。使用以下命令可以指出目标链接器版本所支持的文件格式。

```
Objdump -l
```

2. 内存命令

内存命令描述了目标系统的内存映射。这些内存空间就是在 SECTIONS 命令中所说明的对象。典型的语法如下:

```
MEMORY {
name : o = origin, l = length
name : o = origin, l = length
...
}
```

通常,在内存命令中,在内存命令声明和类型数目、目标硬件所支持的连续内存区域中存在一一对应的关系。然而,典型的例外是处理器的重启向量(某些情况下是整个中断向量表),它们常常被声明为一个独立的区域,其最终的位置是可以被严格控制的。

3. 段命令

段命令的声明描述了各个已命名的段所放置的位置,并且指定了要被加入到其中的输入段。每个命令文件仅能写一个段声明,但是该段声明可以有尽可能多的声明。

例如,声明:

```
/* 代码和常量 */
.text :
```

开始定义了一个叫 .text 的段(传统上叫它 .text 段,在 gcc 下, text 段实际上包括了应用程序的指令代码、常量声明和字符串,这依据各个段模块中提供的段属性而定)。随后花括号中的声明使链接器进行以下操作:

- (1) 创建了一个名为 _text_start 的符号,并将其放置在段的开始部分。
- (2) 将所有输入文件中的 .text 和 .string 段汇集到该段中。

(3) 创建了一个名为 `_text_end` 的符号,将其放置到段尾。

最后,声明:

```
\ > rom
```

告诉链接器将整个段放置在一个叫 `rom` 的内存区域,依据 `MEMORY` 命令,该区域起始地址为 `0x400`。

输入段也可以由文件定义。例如,可以加入这样的一行:

```
foo.o (.specialsection)
```

到 `.text` 的段定义中。链接器将文件 `foo.o` 中的 `.specialsection` 段合并加入 `.text` 段中去。

4. AT 指令

AT 指令告诉链接器将某个段数据装入某个别的地方,而不是其实际放置的地址。该特征主要是为了产生 ROM 的镜像,该镜像对嵌入式系统极其重要。

理解 AT 指令最好的方法是学习实例。来看一个程序,它仅有一个初始化的全局变量:

```
int a_global = 102;
```

在编译时, `gcc` 会在模块 `.data` 段中,声明一个目标整数,值为 102。在链接时,对 `.data` 段提供一个 AT 指令,链接器为 `a_global` 分配一个地址(通常是在 RAM 中),但将其初始值放在别的地方(`_text_end` 处,通常在 ROM 中)。

下列的代码初始化了 `a_global`(还包括了应用程序中的其他全局变量)。该代码使用了诸如 `_text_end`、`_data_start`、和 `_data_end` 的符号来查找初始值、决定其大小、并将其放置在 RAM 中的适当位置。

```
extern const char _text_start, _text_end;
```

```
extern char _data_start, _data_end;
```

```
memcpy( &_data_start, &_text_end, &_data_end - &_data_start );
```

现在,将它们放到一起。下面的命令行告诉 `gcc` 编译一个文件 `main.c`,然后用命令文件 `main.cmd` 将其链接。

```
gcc -g -Wl, -T main.cmd main.c
```

5. 使用 GNU 工具构建系统的相关问题

虽然 GNU 和其他开放源代码的工具随处可得而且免费,但是它们并不是无条件的免费。如果将应用程序同某个在 GPL 声明下发行的库相链接,那么按照 GPL 的规定,必须将应用程序源代码也公开。

6. 重新定址

在上一章中讲述了为什么需要重新定位的原因。把可重新定址的程序转变为可执行的二进制文件的工具称为定址器。通常需要自己编写代码,告诉定址器开发板上存储器的信息。定址器用该代码对可重新定址程序中的所有代码和数据段重新分配物理的地址。



对于 uClinux 而言,内核的编译使用了 ucsimm.ld 文件,形成了可执行文件的映像。所形成的代码段既可以使用间接寻址方式,也可以使用绝对寻址方式。内核一般是使用了绝对寻址,所以其内核加载到内存的地址空间和 ld 文件中给定的内存空间完全相同。

第 4 章讲到,应用程序的连接与内核连接方式不同,应用程序由内核加载。由于应用程序的 ld 文件给出的内存空间与应用程序实际被加载的内存位置可能不同,这样在应用程序加载的过程中需要一个重新定位的过程,即对 reloc 段进行修正,使程序进行间接寻址时不至于出错。

可执行文件加载器的内容,可以参考 uclinux/linux/fs/binfmt_flat.c。对于 GNU 工具来讲,没有专门的重定址器,而是由链接器来做这些工作。

5.2.4 嵌入式 Linux 程序的调试——使用 gdb

在这一小节将学习嵌入式 Linux 程序的调试。对于嵌入式 Linux 的开发,有两种方式可以进行调试程序。

- 对于复杂应用,应该先在 PC 机上调试通过。

因为这时可以用 gdb 调试,调试后再向目标板移植。移植的工作相对要简单多了。只要改一下 makefile,再补正一些缺失的例程即可。

- 对于简单应用,直接在目标板上调试。

这时的调试手段就太简单了,只有一些最原始和基本的手段。最常用的就是打印串口,在程序中设置一些 printf() 的语句来打印需要的值。

在本小结中先来学习第一种方式,即使用 gdb 来调试应用程序。第二种方法在后面的小节中讨论。

1. 概论

在第 3 章已经学习了 GNU 的调试工具 gdb 的使用。在这一章里将学习使用 gdb 通过一根串行线调试与 PC 相连的嵌入式 Linux 系统。

gdb 可以调试各种程序,包括 C、C++、JAVA、PASCAL、FORAN 和一些其他的语言,还包括 GNU 所支持的所有微处理器的汇编语言。

在 gdb 的所有可圈可点的特性中,有一点值得注意,就是当运行 gdb 的平台(宿主机)通过串行端口(或网络连接,或是其他别的方式)连接到目标板时(应用程序在板上运行),gdb 可以调试对应用程序进行调试。这个特性不光在将 GNU 工具移植到一个新的操作系统或微处理器时候很有用,如果目标系统使用了 GNU 支持的芯片的话,这个特性对于在这个目标系统上进行开发和设计也很有帮助。

当 gdb 被适当地集成到某个嵌入式系统中时,它的远程调试功能允许设计人员一步一步地调试程序代码、设置断点、检验内存,并且同目标交换信息。gdb 同目标板交换信息的能力相当强,胜过绝大多数的商业调试内核,其功能甚至相当于某些低端仿真器。

2. gdb 在嵌入式领域的功能实现

当调试一个远端目标设备时,gdb 依靠一个调试 stub 来完成其功能。调试 stub 是嵌

入式系统中一小段代码,它提供了运行 gdb 的宿主机和所调试的应用程序间的一个媒介。

gdb 和调试 stub 通过 gdb 串行协议(见附录 B)进行通信。gdb 串行协议是一种基于消息的 ASCII 码协议,包含了诸如读写内存、查询寄存器、运行程序等命令。由于绝大多数嵌入式系统设计人员为了最好地利用他们手中特定的硬件的特征,总是自己编写自己的 stub。

为了设置断点,gdb 使用内存读写命令,来无损害地将原指令用一个 TRAP 命令或其他类似的操作码(在此假定,被调试的应用程序是处在 RAM 中的,当然,如果 stub 有足够好的性能,硬件也不错的话,这也不一定)代替,在执行该命令时,可以使控制权转移到调试 stub 手中去。此时,调试 stub 的任务就是将当前场景传送给 gdb(通过远程串行通信协议),然后从 gdb 处接收命令,该命令告诉了 stub 下一步该做什么。

为了说明,见例 5-2,它是 Hitachi SH-2 处理器的一个 TRAP 异常处理程序。

例 5-2 一个 TRAP 异常处理程序

```
/* 将当前寄存器的值存储到堆栈中 */
/* 然后调用 gdb_exception. */
asm(
.global _gdb_exception_32
_gdb_exception_32:

/* 将堆栈指针和 r14 压入堆栈 */
mov.l r15, @-r15
mov.l r14, @-r15

/* 当执行一个陷阱异常时,sh2 自动地将 pc 和 sr 放入堆栈 */
/* 所以我们必须调整我们给 gdb 的堆栈指针值,以此来说明这个特别的数据 */
/* 换言之,在该陷阱被执行前,gdb 想看看堆栈指针的值, */
/* 而不是陷阱被执行当前时的值。 */
/* 所以,从我们刚压入堆栈的 sp 值中减去 8 */
/* (pc 和 sr 都是 4 个字节的) */

mov.l @(4,r15), r14
add #8, r14
mov.l r14, @(4,r15)

/* 将其他寄存器值压入堆栈 */
mov.l r13, @-r15
mov.l r12, @-r15
mov.l r11, @-r15
mov.l r10, @-r15
mov.l r9, @-r15
```



```

mov.l r8, @-r15
mov.l r7, @-r15
mov.l r6, @-r15
mov.l r5, @-r15
mov.l r4, @-r15
mov.l r3, @-r15
mov.l r2, @-r15
mov.l r1, @-r15
mov.l r0, @-r15
sts.l macl, @-r15
sts.l mach, @-r15
stc vbr, r7
stc gbr, r6
sts pr, r5

```

```

/* 调用 gdb_exception, 令其异常值 = 32 */
mov.l _gdb_exception_target, r1
jmp @r1
mov #32, r4

```

```

.align 2
_gdb_exception_target: .long _gdb_exception
");

```

```

/* 下面是一个从调试 stub 返回对某个应用程序的控制的样例(针对 Hitachi SH2) */
/* 如果用 C 语言写,那么该语句的原型为: */
/* void gdb_return_from_exception( gdb_sh2_registers_T registers ); */
/* 总而言之,我们可以用与 gdb_exception_nn 把寄存器压入堆栈同样的方式 */
/* 将其从堆栈中弹出。然而,通常返回指针同我们的返回堆栈指针不一样。 */
/* 所以如果我们在拷贝 pc 和 sr 到返回指针之前将 r15 弹出的话,我们就会 */丢失
掉 pc 和 sr。
*/
asm(
.global _gdb_return_from_exception
_gdb_return_from_exception:

/* 恢复某些寄存器 */

```

```
lds r4, pr
ldc r5, gbr
ldc r6, vbr
lds r7, mach
lds.l @r15+, macl
mov.l @r15+, r0
mov.l @r15+, r1
mov.l @r15+, r2
mov.l @r15+, r3
mov.l @r15+, r4
mov.l @r15+, r5
mov.l @r15+, r6
mov.l @r15+, r7
mov.l @r15+, r8
mov.l @r15+, r9
mov.l @r15+, r10
mov.l @r15+, r11
mov.l @r15+, r12

/* 将 pc 和 sr 弹出到应用程序的堆栈 */
mov.l @(8,r15), r14
mov.l @(16,r15), r13
mov.l r13, @-r14
mov.l @(12,r15), r13
mov.l r13, @-r14

/* 完成恢复寄存器的工作 */
mov.l @r15+, r13
mov.l @r15+, r14
mov.l @r15, r15

/* 调整应用程序的堆栈,来说明 pc, sr */
add #-8, r15

/* ... 返回到应用程序 */
rte
nop
");
```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

当处理器遇到了一个 TRAP 指令(该指令是由 gdb 设置的,做断点用)时,该指令使处理器的当前场景转向一个名为 `gdb_exception()` 的函数。最终,目标调用了 `gdb_return_from_exception()` 函数,该函数恢复了处理器的场景并将控制权交给应用程序。

远程串行协议的步进命令更有挑战性,当目标处理器不提供一个“跟踪位”或类似的功能时。在这些情况下,唯一的替代办法就是让 stub 把将要执行的指令反汇编。这样它就会知道程序下一步要执行到何处。

幸运的是,在 gdb 的源代码中也提供了一些如何实现这些步进命令的建议。对于 Hitachi SH-2 芯片而言,在 `gdb/sh-stub.c` 文件中说明了函数 `doSStep()` 的使用。对于其他种类的芯片,函数的名字也差不多,见文件 `gdb/i386-stub.c` 和 `gdb/m68k-stub.c`。

3. gdb 的其他功能

gdb 还可以求解在控制台中输入的任意 C 表达式的值,包括包含有对远端目标的函数功能调用的表达式。输入如下命令:

```
print foo( sh_sci[current_sci]->smr.brg )
```

gdb 就会将 `mr.brg` 的值传送给 `foo()`,并报告其返回值。

当然,gdb 也可以反汇编代码,只要有可能,还可以很好地为所需的数据提供等价的符号信息。例如,gdb 用下列输出:

```
jmp 0x401010 <main + 80>
```

告诉用户,所显示的地址与从函数 `main()` 的起始地址起偏移 80 个字节的地址相等。

gdb 可以显示其自身和所调试的目标间的远程串行调试信息,也可以将该信息记录到日志文件中去。这些特性对调试一个新的 stub,了解 stub 是如何使用远程串行协议来满足用户对数据、程序内存、系统调用等需求是十分有用的。

gdb 拥有脚本语言,允许对目标自动地设置和检测。该语言是对目标处理器独立的,所以应用程序从一个目标处理器移植到另外的处理器时,脚本可以重用。

最后,gdb 还提供了跟踪点的功能,该功能可以记录某个运行程序的信息,而尽可能不中断程序收集数据。跟踪点需要特别地调试 stub 来实现。

4. 一个典型的 gdb 会话过程

已经探讨了 gdb 的通用功能,现在来看 gdb 的执行。下面给出了一个典型的 gdb 调试会话过程。在该过程中,gdb 初始化了同一个运行调试 stub 的远端目标间的通信,然后下载程序,设置断点,并运行该程序。当遇到断点时,调试 stub 通知 gdb,然后 gdb 就将其源代码行显示给用户。接着,用户显示了一个变量,步进执行一个指令,然后退出 gdb。

下面并未显示用户在使用 gdb 时所见到的内容。用户所见到的是一个终端,显示的内容都是用英文写成的源代码、要显示的变量等等。但是,下面显示的脚本说明了当用户键入命令时在幕后发生的内容。

典型的 gdb 会话过程的描述

用户键入的内容

gdb 发送

串口发生的内容
目标响应

```

host> gdb myprogram
gdb> target remote /dev/ttyS0      + $ Hcl# 09          + $ OK# 9a
                                   + $ qOffsets# 4b     + $ Text = 0;Data = 0;Bss = 0# 04
                                   + $ ? # 3f          + $ S05# b8
                                   + $ g# 67          + $ 00001a00ffff81b200000020...
-----
gdb> load                          $ M401054,10;0040   + $ OK# 9a
                                   2024004020240040
                                   202400402024# 72
                                   [更多的 M 信息]
-----
gdb> breakpoint main               [什么都没有。gdb 在随后的命令发送之前物理地设置断点]
gdb> continue                      + $ M4015cc, 2:
                                   c320# 6d          $ OK
                                   + $ c# 63          # 9a
-----
[程序运行直到 main()函数]
                                   $ T050:00401400;1:00404850;2:000000
                                   01;3:00000030;4:ffffff;5:00000000;;
                                   00000010;7:00000010;8:0040161c;9:0
                                   0002070;a:00404068;b:004015bc;ffff
                                   fff;d:fffffef;e:00404840;;00404840;10
                                   :004015cc;11:004015cc;12:d04001e2;1
                                   3;00401000;14:00000000;
                                   15:00ffffff;16:000000f0;# d1
-----
                                   + $ m4015bc,2# 5a
-----
[目标在 main()处停止,地址为 0x4015cc]
-----
gdb> display foo                   + $ 2f86# 06
-----
[foo 的地址为 0x4015bc; 其值为 0x2f86]
-----
gdb> stepi                          +
-----
[目标执行一个指令]
-----
[PC 的值现在为 0x4015ce]
-----
gdb> quit

```

左边一栏显示了 gdb 控制台的一部分,在此用户键入命令并监视数据。右边一栏显示了一些使用 gdb 远程串行协议在宿主机和嵌入式设备之间的通信消息。在方括号中的是一些解释信息。如果了解这些信息的含义,见附录 B《gdb 远程串行协议》。

5. gdb 调试 stub 的源代码

虽然远程软件调试具有依赖于目标的特性,但还是可以创建一个有高度的可移植性的调试 stub。在不同的嵌入式处理器芯片之间可以被重用,而所需的修改最小。

有人已经尝试了这方面的工作。如果感兴趣,可到 <http://sourceforge.net/projects/gd-bstubs> 去查阅相关的资料。

处理器特定的代码包含在与处理器相关的文件名中,例如 `gdb_sh2*.c`。针对特定的处理器可以下载相关的文件(例如 `gdb_m68k*.c`),然后再用其替代用户机器上的相关内容。

6. 改造 gdb 来解决特定问题

gdb 使用了一个模块化的体系结构来实现,那么对某些不适合需要的特性就可以直接地处理。如果用户的产品仅仅有一个通信端口,而它使用的并不是 gdb 的通信协议,那么可以修改 gdb,使调试器同产品相匹配。

类似地,如果产品没有串行端口,而有别的通信接口(例如 CAN 端口),就可以加强 gdb 的远程通信功能以适应该端口。

也可以修改 gdb 的工作方式使其同嵌入式应用程序相容。如果正在使用 TRAPA #32 做同 gdb 无关的工作,就可以改变 gdb 为了设置断点而使用的操作码,也可以使用 gdb 来产生一个新的消息,告诉目标板完成诸如开启指令追踪的功能、断点产生功能等动作。

文件 `gdb/remote.c` 包含了 gdb 的远程串行协议的实现过程。对于研究 gdb 模块化的实现是如何快速地将它改造以适应特定的调试目标而言,该文件是个很好的起点。其他文件,例如 `gdb/remote-hms.c` 和 `gdb/remote-e7000.c`,使用了该模块化的结构来为诸如 Hitachi、Motorola 等公司的芯片调试器和仿真器提供支持。

7. 总结

gdb 对于调试目标(包括对其内存的使用、通信媒介等方面)的可适应性使它对于目标板的调试而言,常常是唯一的选择。考虑到单芯片高集成度、基于 IP 的嵌入式产品的普及,情况更是如此。在今天,嵌入式设备的复杂性与日俱增,在进行新的设计时,其供选择的技术也越来越多,要找到一个商业的开发产品是越来越困难了。

而使用 GNU 工具将是个很好的选择。GNU 工具对各种流行的嵌入式处理器的支持意味着,当正在使用的开发工具对将要在下一个设计中使用的处理器不支持时,可以减少寻找新的开发工具所带来的危险。

5.3 应用软件的开发

通过 5.1 和 5.2 节的学习,对嵌入式 Linux 系统的构建、应用程序的编译、链接和调

试都有了一个清楚的认识。应该说,这些知识是很概括的。如果没有经过实际操作,很难掌握其使用。下面讲述如何构建开发平台和开发环境,学习在开发平台上,如何编写、调试并将应用程序烧入 flash 中。

5.3.1 建立开发环境

在上一章中,介绍了嵌入式 Linux 软件和硬件环境的构建,并将开发板的开发环境的构建留到了这章来讲述。下面介绍开发环境的建立。

1. 搭建硬件环境

购买了华恒公司的开发套件后,按图 5-4 将开发板同 PC 相连接。开发套件的串行口是 DB-9 的 RS-232 接口。将 D 形口一端同 PC 相连(在 PC 的背后可以找到 DB-9 的接口),并将其固定,然后将其另外一头同开发板接好,然后找 2 根网线,将其如图 5-4 连接好。这样,硬件环境就搭建好了。如果一切正常,接上电源后,各个指示灯都会被点亮,并在 LCD 上显示出开机画面。参看图 4-1 的开发板外观图。

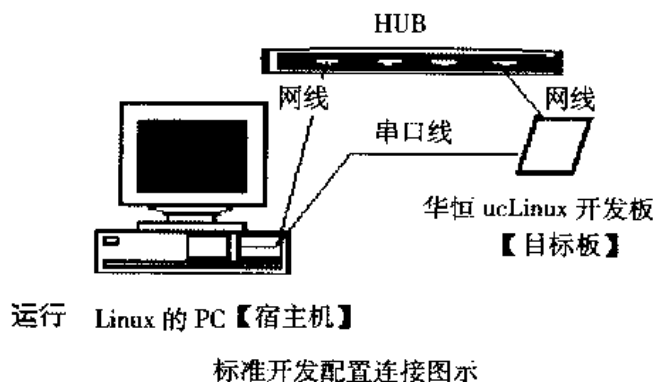


图 5-4 开发板和 PC 连接图

注意: 如果不需要集线器(HUB)而将 PC 和开发套件通过网线直接相连接,则需对网线改动,即将一端的 TX+ 和 RX+、TX- 和 RX- 分别交换。

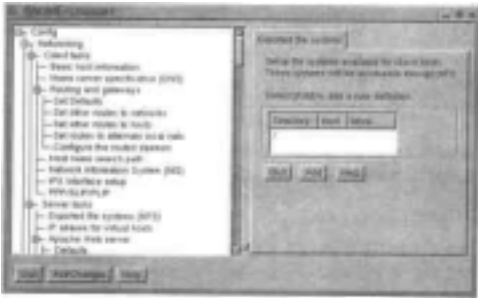
2. 软件配置

对于开发套件来讲,软件环境的配置往往更加重要。如果软件环境配置不周,往往会导致各种问题出现,最常见的是“nfsmount”命令不能正确执行。作者在刚刚接触开发套件时就遇到了一些问题,这些都是软件配置不当而造成的。

(1) 配置宿主机的环境

先确认自己的宿主机上安装的是 Redhat Linux 6.2/6.1 版本。一定要安装这两个版本的 Linux,其他版本的 Linux 可能会导致某些预料不到的错误。例如出现刚才讲的“nfsmount”命令不能正确执行的情况。然后,配置网络文件系统(NFS)。其步骤如下:

① 运行 Linuxconf,如图 5-5 所示。



超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

图 5-5 启动 Linuxconf

②在 config 项下,选择 Server tasks,选择 Exported file systems(NFS),然后选择 Add Directory,加入根目录/,然后选择 Accept,如图 5-6 所示。

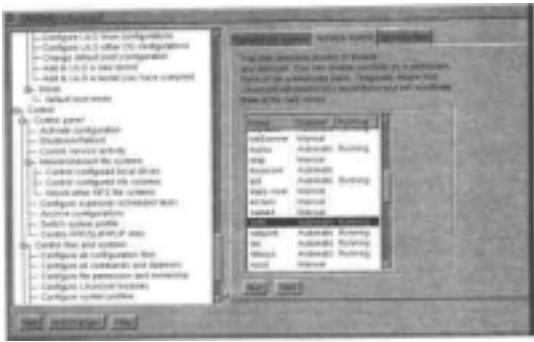
图 5-6 配置 NFS 第一步

③在 Control 项下,Control panel 的 Control service activity 处,选择 netfs enabled,然后选择 start,如图 5-7 所示。

④接受所有的设置,完成 NFS 的设置工作并退出。

(2) 安装开发套件附带的光盘

安装开发套件附带的光盘的方法如下:



超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

图 5-7 配置 NFS 第二步

①将开发套件附带的光盘放入光驱,并键入如下命令来挂载光驱。

```
mount /dev/cdrom /mnt/cdrom
```

对于红帽子 Linux 而言,光驱的加载是自动的,不必执行该命令。

华恒公司配套的光盘内容如图 5-8 所示。

图 5-8 华恒配套光盘内容

几个重要文件说明如下:

- cce.rpm: 汉化软件,提供一个中文化的环境。当然,是在控制台形式下使用;
- ucinst: 安装程序,执行它就会将 uClinux 安装到机器上;
- readme.txt: 说明文件。

②安装配套光盘。

安装配套光盘的过程非常简单。退出 X Window,在控制台方式下,键入以下命令启动中文平台:

```
[root@zou /]# cce
进入挂载光盘的那个目录:
[root@zou /]# cd /mnt/cdrom
然后,键入:
[root@zou /]# ./ucinst
会出现如图 5-9 的界面。
```

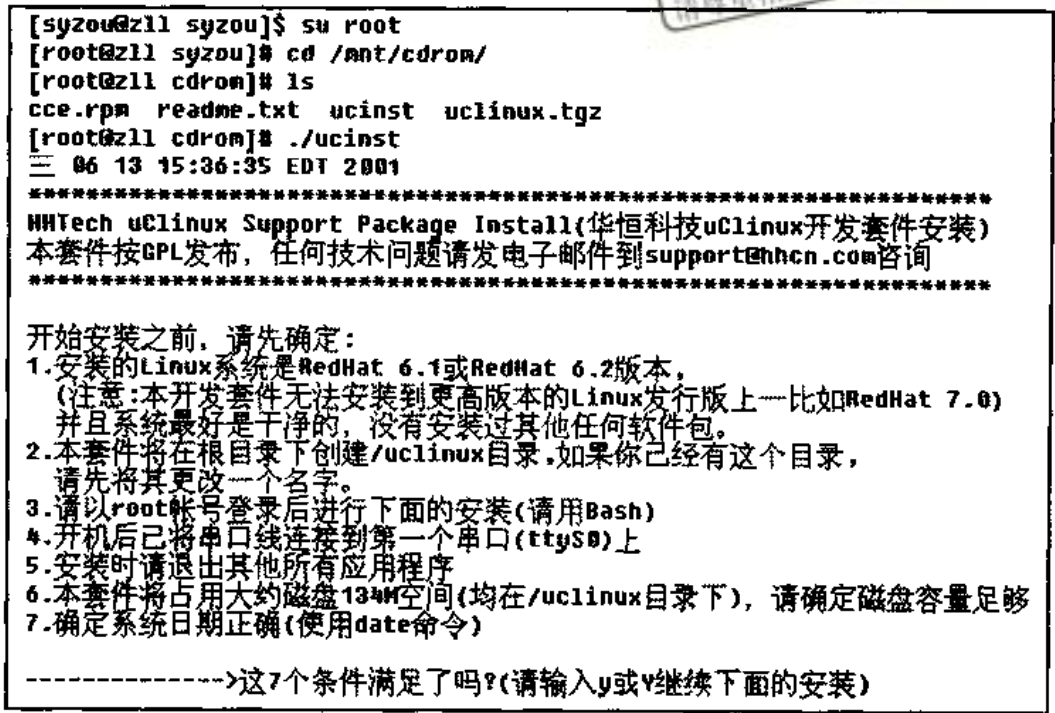


图 5-9 安装界面

如果所有条件都满足,就可以键入“y”,程序就会自动地安装好整个软件系统。

到此为止,所有的软件和硬件的配置已经齐备。现在就可以着手进行编写程序了。在动手前,还是先来熟悉一下如何使用开发环境。

5.3.2 熟悉开发环境

1. 启动并登录系统

启动开发板很简单,将系统通电,如果在 LCD 上显示了开机界面,所有指示灯都正常点亮,那么系统已经启动了。登录到开发板上有两种方式,一种是通过串口登录,另外一种是通过网口登录。

通过串口是很常用的一种方式。启动 minicom(方法见图 4-7,也可以在终端仿真下键入 minicom 命令),按下开发板上的重启键(一个小白色按钮),然后就可以看见



上。启动成功后,可以见到如图 5-10 所示的启动界面。



图 5-10 uClinux 的启动界面

从该启动界面上可以得到关于 uClinux 的基本信息。如 uClinux 的版本号是 2.0.38, RAM 盘被挂载到了 /var 目录下,开发板的以太网卡地址(MAC 地址)被设置成为了“aa.aa.aa.aa.aa.aa”等。另外,开发板的 IP 地址默认为 192.168.2.124,是可以修改的,具体方法在后面会详细讲述。

另外一种方法是通过网口,远程登录开发板。启动一个终端仿真程序,键入:

```
telnet 192.168.2.124
```

开发板的 telnet server 并不做身份验证,按“Enter”就可登录上去。如图 5-11 所示。

图 5-11 telnet 登录方式

登录成功后,就可以操纵系统了。根据作者的经验,采用串行口的方式较好,因为速

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

度要比后者快些。



基本命令和应用程序。各个应用程序和命令的可执行文件命令的方法也很简单。例如,为了检验网络的物理连接是否 minicom 下,bash 的提示符号后键入以下命令即可,如图



图 5-12 执行 ping 命令

```
ping 192.168.2.2
执行一个样例程序,例如:
>gui
```

就会在 LCD 上看见不断翻滚移动的“华恒公司”字样。

注意:要保证 PC 机同开发板在同一个网段里。如果开发板的 IP 为 192.168.2.124,那么 PC 机的 IP 必须为 192.168.2.xxx。当然,也可以修改开发板的 IP 地址。

到此,软硬件环境都已经搭建好了,就可以进行上层软件的开发了。在进入开发之前,关于 uClinux 的概念和开发中必须注意的事项再交代一下。

3. 关于系统的进一步说明

(1) 系统的内核

uClinux 系统的内核是基于 2.0.38 版本的 Linux 内核而构建的。在原有内核的基础上,uClinux 通过给其打补丁,从而构成了 uClinux 的内核。打补丁的过程包括加入对设备的支持和删除对 MMU 的依赖等。

(2) 工具链

对于 uClinux 而言,有两套工具链,前面曾提到过。一个用来编译内核和生成 32 位

M68K 固定位置可执行文件,另外一个用来编译用户这边的二进制代码并生成 32 位 M68K 的位置独立代码(PIC)。若开发应用程序,首先在 PC 下编译时,就要用到这个工具链。

- m68k-coff 工具:它是用来编译内核的,并不与标准的 uC-libc 库相连。
- genromfs 工具:ROM 文件系统生成工具。
- coff2flt 工具:其作用是用来将编译器产生的 coff 镜像转换成为可以在 uClinux 下运行的 flat 格式。

使用当前的 m68k-pic-coff 时,存在一个局限,即不能生成大于 32K 的可执行文件。可执行文件的大小是使用 16 位的带符号的偏移量来设置的。如果程序大于 32K,在调用编译器时使用 -fPIC 开关选项,就可以产生 32 位的偏移量。



质编译出来的文件较大,所以最好在需要时才使用。

的文件系统的基础。它的结构在第 4 章已经讲述了。

提供了一种在 UNIX 和类 UNIX 系统间输出和挂载文件用,它能很方便地将用户的工作目录输出到 uClinux 平台上。这个功能可以使用户在 PC 机上的 Linux 开发环境下,编译代码并在嵌入式系统已经挂载的网络驱动器上运行,但是并不需要将编译出的 flat 格式的文件拷贝到系统开发板的闪存里面去。这就大大节省了调试的时间。

在开发板上,挂载 NFS 的方法是键入命令:

```
nfsmount + 宿主机 IP/ + /tmp
```

出现 > 提示符号,表示挂载成功,如图 5-13 所示。如果出现诸如 mount failed 或其他提示,系统的软件和硬件的配置没有问题,那么可以多试几次。如果实在不行,一般是 NFS 的配置出了毛病,参看前面的内容。

图 5-13 挂载 NFS



置好开发环境来输出所需要的目录。这样,在 uClinux 下,到 uClinux 的/tmp 目录下,就可以通过访问/tmp 下面的文件特性对调试应用程序是大有帮助的,如图 5-14 所示。

图 5-14 通过挂载 NFS 来实现对应用程序的调试

从图 5-14 可见,用户所处的目录为/var/tmp/uclinux/romdisk/romdisk/bin(/tmp 目录其实是指向/var/tmp 目录的一个符号链接),该目录是宿主机上存放应用程序的可执行二进制文件的目录。无须将可执行文件烧写到开发板的 flash 中间去,而是通过网络挂载,直接执行或调试该程序,从而省却了很多的时间和精力。如键入:

```
> ./handpad
```

就执行了该程序。

☞ 注意:该程序是存在于宿主机的硬盘上,同开发板闪存内附带的 handpad 程序并不在一处。随后打印出来的信息是对该程序的一些调试信息,请见第 8 章相关内容。

(5) RAM 文件系统

由于 uClinux 操作系统没有硬盘,所以它使用了一个 RAM 文件系统,也可称之为 RAM 盘。在根目录下的/tmp 目录其实是一个符号链接,它指向了/var/tmp 目录,而/var 目录是 RAM 文件系统的挂载点。RAM 盘的物理设备名在/dev 目录下,是/dev/ram0。它是一种块设备,而不是字符型的设备。RAM 盘包括了一个 ext2 类型的文件系统,它是通过使用一个所谓的“零运行长度解码”算法(ZRLE)来进行压缩的。在系统驱动时,解压工具将 RAM 盘的镜像解压到 RAM 块设备/dev/ram0 处。当该步骤完成后,就形成了 ext2 类型的文件系统,再将该设备挂载到/var 目录下。

(6) 关于系统的能耗

系统的能耗取决于 CPU 是在运行还是在空闲态,以及以太网是否启动等因素。



5.3.3 在开发板上编写应用程序

通过前面的学习,对整个系统应相当熟悉了。下面,先来学习如何编写应用程序,并学习如何将在宿主机上编写的程序通过改写适应开发板。然后,学习如何对编写好的应用程序进行编译、调试。最后学习如何将调试运行通过后的应用程序烧写到开发板的 flash 中间去。

1. 应用程序的开发模式

应用程序的开发有两种模式。

(1) 模式 1

先在宿主机(X86 CPU)上调试通过,然后移植到开发板上去(CPU 为龙珠 68EZ328)。在这种方式下,可以使用宿主机上的 gdb 进行调试,对于大型的复杂的应用程序,这是必不可少的。关于移植的工作,主要是在程序移植时,由于开发板的 uClibc 库同 PC 标准的 libc 库是有所不同的,所以往往会出现函数没有定义的问题。这样,对某些标准 libc 库提供的而 uClibc 未提供的函数,就需要自己定义了。在后面将用一个实例来详细讲解实现的方法和步骤。同时,还需要改变 makefile 以适应开发板。

(2) 模式 2

本模式适合开发比较简单的应用程序。可直接在开发板上进行开发,该开发模式的步骤如图 5-15 所示。

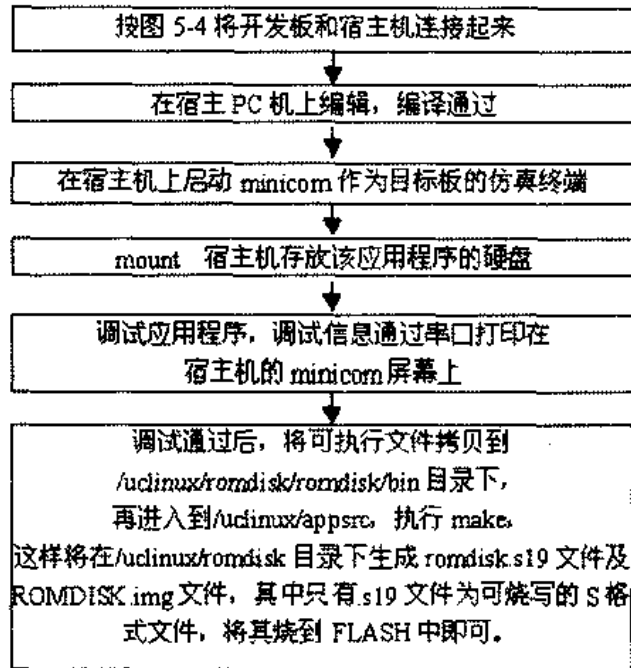


图 5-15 应用程序的开发模式二

图 5-15 给出了应用程序的开发模式。很遗憾,由于开发板上没有 gdb server,所以无法通过 gdb 的串行通信协议实现对开发板上的应用程序进行调试。对于这种打印串行

口的方法,仅适合开发小型的应用程序。至于将 gdb server 移植到开发板上的工作,有兴趣的不妨一试。

(3) 关于 makefile

对于应用程序而言,其 makefile 可以从华恒公司提供的样例中找到相应的模板,然后加以改造便是。

(4) 板子的内存空间、I/O 空间和中断向量表

对于嵌入式系统开发而言,其内存空间和 I/O 空间的分配是相当重要的。对于开发板而言,其 CPU 决定了内存空间和 I/O 空间的分配。

MC68EZ328 处理器的 RAM、flash 及 I/O 是统一编址的。华恒开发板有:

2 片 RAM,共 4M,从 000000-3FFFFFF;

2 片 1M flash,第一片:1000000-10FFFFFF,第二片:1100000-11FFFFFF;

至于 I/O 则要查看 MC68EZ328 CPU 手册了。

华恒 328 开发板将所有中断向量映射到同一个中断处理函数处,然后在该处理函数处再进行二次分发。这一点是由 MC68EZ328 处理器所采用的核决定的。它的核不支持中断的分发,因此 uClinux 的 328 版本都采用这种二次分发的机制。具体见 MC68EZ328 CPU 用户手册。

☞ 注意:就 uClinux 开发板来说,flash 就相当于 PC 机的硬盘,而 RAM 就相当于内存。

2. 应用实例

来看一个简单的客户/服务器模式的程序,该程序实现了一个简单的客户/服务器间的通信功能。先来看它的源代码,见例 5-3。

例 5-3 客户/服务器通信程序

(1) 客户端

```
/* client.c: 一个简单的客户端接收程序样例演示。  
* Copyright (C) 2001 <syzou@263.net.cn, liyan1011@sina.com>  
* 该程序是自由软件,您可以依据自由基金联合会  
* (FSF)的 GNU 通用公共许可证的条款  
* (GPL2.0 版本,或更高的版本)  
* 对该软件进行重新发布或修改。*/  
#include <stdio.h>  
#include <stdlib.h>  
#include <errno.h>  
#include <string.h>  
#include <netdb.h>  
#include <sys/types.h>  
#include <netinet/in.h>  
#include <sys/socket.h>
```

```
#include <stdlib.h>
#include <netinet/in.h>

#define MAXDATASIZE 100 /* MAXDATASIZE 定义了客户机一次可以接受
                        * 的字符数目 */

#define PORT 2000 /* 定义连接到服务器的端口号。 */
int main(int argc, char * argv[ ])
{
    int sockfd, numbytes; /* sockfd 是套接字描述符 */
    char buf[ MAXDATASIZE];
    struct hostent he; /* he 为 hostent 类型的数据结构,用来存放所获取的主机
                       * 信息 */
    struct sockaddr_in their_addr; /* 存放主机信息 */
    if(argc != 2)
    {
        fprintf(stderr, "usage: client hostname \n");
        exit(1);
    }
    if((he = gethostbyname(argv[1])) == NULL) /* 调用 gethostbyname 函数,
                                             * 获取主机信息 */
    {
        perror("gethostbyname");
        exit(1);
    }
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) /* 建立流式套接
                                                         * 字描述符 */
    {
        perror("socket");
        exit(1);
    }
    /* 给定远端主机信息 */
    their_addr.sin_family = AF_INET;
    their_addr.sin_port = htons(PORT);
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(their_addr.sin_zero), 8);

    /* 建立连接 */
    if(connect(sockfd, (struct sockaddr *) &their_addr, sizeof(struct sockaddr)) == -1)
    {
        perror("connect"); /* 错误校验 */
    }
}
```






```

exit(1);
}
if((numbytes != recv(sockfd, buf, MAXDATASIZE, 0)) == -1) /* 接收主机传送过
                                                    * 来的字符串 */
|perror("recv");
exit(1);
}
buf[numbytes] = '\0';
printf("Received: %s", buf);           /* 打印信息到标准输出 */
close(sockfd);
return 0;
}

```

以上是一个简单的客户端的信息接收程序。如果对基本的 Linux 网络编程不太了解,参看本书的第 6 章“嵌入式 Linux 网络功能的实现”,那里有关于 Linux 网络编程的介绍。

 **注意:**为了遵循 GPL 条例(参看附录 A),在编制的软件前,都加上了类似的一段说明文字。为了便于理解,在此将其翻译成中文。在以后的程序中,就不这么做,而直接使用英文了。

该程序实现的功能是,通过使用函数 connect(),连接到服务器的 2000 端口,然后获取服务器发送的字符串,并打印显示。

(2) 服务器端

```

/* servise.c: 一个简单的服务器端程序样例演示。
 * Copyright (C) 2001 <syzou@263.net.cn>
 * 该程序是自由软件,您可以依据自由基金联合会
 * (FSF)的 GNU 通用公共许可证的条款
 * (GPL2.0 版本,或更高的版本)
 * 对该软件进行重新发布或修改。 */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/wait.h>
#include <sys/socket.h>

#define MYPORT 2000      /* 服务器的监听端口 */
#define BACKLOG 10     /* BACKLOG 为可同时被接受的没有被 accept 的连

```

* 接个数,即等待队列的大小 */

```

main()
{
int sock _ fd,new _ fd;          /* sock _ fd 为监听用套接字描述符 */
struct sockaddr _ in my _ addr;
struct sockaddr _ in their _ addr;
int sin _ size;
if((sock _ fd=socket(AF _ INET,SOCK _ STREAM,0)) == -1)
|
perror("socket");
exit(1);
|
my _ addr.sin _ family= AF _ INET;
my _ addr.sin _ port = htons(MYPORT);
my _ addr.sin _ addr.s _ addr= INADDR _ ANY;
bzero(&(my _ addr.sin _ zero),8);
if(bind(sock _ fd,(struct sockaddr * )&my _ addr, sizeof(struct sockaddr)) == -1)
|
perror("bindB");
exit(1);
|
if(listen(sock _ fd,BACKLOG) == -1)
|
perror("listen");
exit(1);
|
while(1)
|
sin _ size = sizeof(struct sockaddr _ in);
if((new _ fd=accept(sock _ fd,(struct sockaddr * ) &their _ addr, &sin _ size)) == -1)
|
perror("accept");
continue;
|
printf("server:got connection from %s \n",inet _ ntoa(their _ addr.sin _ addr));

if(! fork())          /* 创建一个子进程,来处理与刚刚建立的套件字的通信 */
|

```



```

if(send(new_fd,"Hello,World! \n",14,0) == -1) /* 发送字符串 */
{
perror("send");
close(new_fd);
exit(1);
}
close(new_fd);
}
}
while(waitpid(-1,NULL,WNOHANG)>0)
;|

```

服务器端程序的功能是监听其 2000 端口。如果发现有客户机的请求到来,就调用 fork() 函数产生一个子进程,让子进程与客户机进行通信。显然,同时可以有多个客户机对服务器提出请求,而服务器可以应答多个客户机的请求。

(3) makefile 的编写

关于该程序的具体实现将在下一章讨论。在 PC 上,使用 gcc 对这两个程序进行编译链接,并运行通过。但是,如果要将某个程序(如客户机端的程序)烧写到开发板的闪存内去,(在 uClinux 操作系统下,仅支持 flat 格式的文件)那么,就需要编写一个 makefile 完成对应用程序的编译和格式转换的工作。

一般而言,编写 makefile 有两种方式。对应用程序而言,可以直接使用华恒公司附带的应用程序中的 makefile 作为模板,手工编写需要的 makefile,所做的仅是将其所维护的文件名修改一下即可。例如,有这么一个 makefile:

```

CC = m68k-pic-coff-gcc
COFF2FLT = coff2flt

CFLAGS = -I/uclinux/lib/include
LDFLAGS = /uclinux/lib/libc.a

ethernet:ethernet.o
$(CC) -o $@.coff ethernet.c $(CFLAGS) $(LDFLAGS)
$(COFF2FLT) -o ethernet ethernet.coff
cp ethernet /ethernet
clean:
rm -f ethernet ethernet.o ethernet.coff

```

将该 makefile 中的“ethernet”改为用户自己的应用程序名称,在这里就是“client”(所有的都要替换),如下所示:

```

CC = m68k-pic-coff-gcc
COFF2FLT = coff2flt

```

```

CFLAGS = -I/uclinux/lib/include
LDLAGS = /uclinux/lib/libc.a

client:client.o
$(CC) -o $@.coff client.c $(CFLAGS) $(LDLAGS)
$(COFF2FLT) -o client client.coff
cp client /client

clean:
rm -f client client.o client.coff

```



在/uclinux/appsrc 下建立一个自己的应用程序目录,如在该例中就可以建立一个/outhernethernet 目录。先将该 makefile 文件和客户端源程序拷贝到该目录下。然后,在自己应用的目录下执行 make,生成的可执行文件直接被拷贝到/uclinux/romdisk/romdisk/bin 中。再退到/uclinux/appsrc/目录下执行 make,则在/uclinux/romdisk/生成可烧写文件 romdisk.s19。最后,烧写 flash,则用户应用程序的可执行文件就被烧到板子的/bin/目录下。

🔊 注意:client:client.o 下一行的开头是一个制表符,即 Tab 键,而不是若干空格! 请见第3章关于 makefile 的编写规则。

先来解释一下该 makefile。

- CC = m68k-pic-coff-gcc:该句就是一个宏定义,定义符号 CC 代表了命令 m68k-pic-coff-gcc。根据前面的介绍,该命令就是用户工具链中的一个工具,将源代码编译为 coff 格式的文件。
- COFF2FLT = coff2flt:该宏定义在此定义了一个 COFF2FLT 宏。显然,它代表了前面讲述的将 coff 文件格式转化成为 flat 格式文件的工具。
- CFLAGS = -I/uclinux/lib/include 和 LDLAGS = /uclinux/lib/libc.a:这两个宏定义代表了编译时需要用到的库文件。
- client:client.o:指出目标文件所依赖的文件为 client.o。
- \$@.coff:表示动态宏,即说明它代表了当前正在运行的文件。
- Clean:这是一个伪目标。如果要重新编译从而想删除原有的文件,则键入“make clean”命令,就删除了 client、client.o 和 client.coff 这三个文件。

还可以使用/uclinux/appsrc/下提供的一个应用例子的 makefile 为模板进行修改。例如将/uclinux/appsrc/lissa 下的 makefile 拷贝到自己的应用目录下,假设自己开发应用为 myapp,在目录/uclinux/appsrc/myapp/下,有文件 myapp.c。则要作如下修改:

- 在最开始处添加一句:EMBED=1。
- 将 lissa 换为自己应用的名字,包括 .c 文件、.o 文件及最后生成的可执行文件名。

🔊 注意:makefile 默认的可执行文件的放置目录为:/uclinux/appsrc/build/embed(EXEC = \$(BUILD)/myapp),可将之修改为当前目录(修改为 EXEC = myapp)。

- 在自己应用的目录 myapp 下执行 make,生成的可执行文件会自动被移到/uclinux/appsrc/build/embed/目录下,将其拷贝到/uclinux/romdisk/romdisk/bin 中即可。

(4) 修改源代码以实现移植

现在就得到了三个文件: client.c、sevice.c 和 makefile(可以在本书附带光盘的/uclinux/appsrc/ourethernet/目录下找到)。如果需要将开发板当作客户机,PC 当作服务器,那么就在目录/ourethernet 下放入两个文件:client.c 和 makefile。然后键入 make,系统就会按照 makefile 定义的步骤对源程序进行操作。

在对 client.c 程序进行编译时,会发现函数“gethostbyname”没有定义的错误。这是由于开发平台提供的 libc 库同标准的 libc 库不一致,从而出现了函数未定义的问题。这样就需要自己编制这些函数。幸好华恒公司在其一个样例中提供了一个解决的办法:通过编写一个函数,将命令行输入的字符串转换成机器可识别的无符号整型 IP,从而完成地址转换的功能。将该函数做成一个头文件,包含在修改后的 client.c 的预编译命令中。该头文件如下所示:

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int aiptoi(char * pszip,unsigned int * piip)
/* 将命令行输入的字符串 IP 转换成 connect 函数可识别的整数 uiip */
{
char psziphere[17], * psztmp1, * psztmp2, * pchar; /* 定义指针 */
int i;
bzero(psziphere,17); /* 清空将进行操作的数组 */
strcpy(psziphere,pszip); /* 将要转换的 IP 地址存入该数组中 */
strcat(psziphere,"."); /* 在 IP 地址串末尾加 . */

for(i=0,psztmp1 = psziphere,pchar = (char *)piip;i<4;i++) /* 循环 4 次将 . 转
变成 0 并将字符串型转换成整型 */
{
if((psztmp2 = strstr(psztmp1,".")) == NULL) /* psztmp2 返回指向字符 "." 的位
置的指针 */
return 0;
psztmp2[0] = 0;
(pchar + i) = atoi(psztmp1); /* 调用 atoi() 函数,将该段进行转换 */
psztmp1 = psztmp2 + 1; /* 指针 psatmp1 下移到下一段的开始 */
}
```

```
return 1;
```

```
};
```

本函数处理的是字符串型的 IP 地址,例如“192.168.2.2”,将其转换成整型 IP 地址。首先,将该字符串放入 psziphere[]数组中,“psztmpl = psziphere”一句将字符串的首地址赋给了 psztmpl。其次,调用 strstr(psztmpl,“.”),将符号“.”添加到坐标字符串尾。这样做是为了便于使用循环。然后循环开始,先找到第一个“.”的位置,psztmp2 指向该处,并将该处赋零(psztmp2 为字符型的指针,所以该零为字符零,相当于一个结束标志符号)。接着调用 atoi(psztmpl),此时,psztmpl 指向的是 psziphere[]数组的首地址。再接着调用 atoi(psztmpl),将 psztmpl 指针所指处到其后面第一个结束符号标志处(即 psztmp2 处)的字符串转换为整型数,存放到 pchar 数组中。在第一个循环中,psztmpl 指向了“192.”的“1”处,而 psztmp2 指向了“192.168”间的那个分隔点。“psztmpl = psztmp2 + 1”这句将 psztmpl 指针指向了 psztmp2 的下一处,也就是下段的开始,然后开始下一个循环操作。这样,字符串型的 IP 就转换成为了机器可识别的 IP。

该头文件与 makefile、client.c 程序一起被放在 /uclinux/appsrc/ourethernethernet/ 目录下。

经过改造后的程序所实现的功能是:在客户机一端键入以下命令:

```
client + 宿主机的 IP
```

就可以接受来自宿主机的信息。

针对原来的 client.c 源程序,并不需要调用与 DNS 相关的函数 gethostbyname,而是通过自定义的函数来获取必要的信息,从而达到编译、调试通过的目的。那么,对原来的例 5-3 client.c 改造结果如例 5-4 所示。

例 5-4 改造后的 client.c 源程序

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>

#include </uclinux/appsrc/ourethernethernet/aipatoi.h> /* 自定义的头文件 */
#define MAXDATASIZE 100
#define PORT 2000
int
main(int argc, char * argv[])
{
int sockfd, numbytes, hostip;
```



```
char buf[MAXDATASIZE];

struct sockaddr_in their_addr;
if(argc != 2)
{
printf("usage: client hostip \n");    /* 给出使用方法 */
exit(1);
}

if(! aiptoi(argv[1], &hostip) || argc < 1)
{printf("the ip is wrong! \n");
return(0);
}

if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{printf("socked wrong! \n");
exit(1);
}

their_addr.sin_family = AF_INET;
their_addr.sin_port = PORT;
their_addr.sin_addr.s_addr = hostip;
bzero(&(their_addr.sin_zero), 8);
if(connect(sockfd, (struct sockaddr *)&their_addr, sizeof(struct sockaddr)) == -1)
{printf("can't connet to the host! \n");
return 0;}

if((numbytes = recv(sockfd, buf, MAXDATASIZE, 0)) == -1)
{printf("can't connet to the host! \n");
return 0;
}
buf[numbytes] = '\0';
printf("Received: %s", buf);

close(sockfd);
return 0;
}
```

在/uclinux/appsrc/ourethernet/目录下,键入“make”,就会自动对该程序进行编译、格式转换等工作。生成的可执行文件直接被拷贝到/uclinux/romdisk/romdisk/bin中。然后



再退到/uclinux/appsrc/目录下执行 make,那么就执行该目录下的 makefile。前面已经讲过,makefile 文件包含了各个 *.mk 文件,其中对各个小目录下 makefile 的调用都被包含在 bins.mk 中。在编译时,它调用 appsrc/romdisk/makefile 编译连接各个文件,生成可执行文件,然后拷贝到/uclinux/romdisk/romdisk 目录中,将 romdisk 目录生成 romdisk.img 的镜像文件,修改其格式成为 romdisk.s19 型文件。Romdisk.s19 文件就是可烧写入 flash 中的文件。用户自己的应用烧到 flash 中的位置为/bin,这是由 ROM FS 文件系统的组织



进行编译通过后,就可以运行该程序了。首先,启动主机的硬盘:

```
/ /tmp
```

nux/romdisk/romdisk/bin 和/tmp/uclinux/appsrc/ourether-ient”。然后,在 PC 一端的 bash 提示符号下键入以下命

启动服务器程序):

```
[root@syzou forpc] ./servise (将我们的服务器程序放在/forpc 目录下)
```

最后,在 minicom 中 uClinux 的 sh 提示符下,键入 ./client,就会打印出从服务器传送来的字符串,如图 5-16 和 5-17 所示。

图 5-16 启动客户机程序

客户机接收并打印了字符串,而服务器则显示了与客户机(即开发板)相连接的信号。显然,这两个程序成功地运行了。

还没有讲到调试的问题。很显然,只要 client.c 出现了错误,就可以在源代码中加入调试语句,用打印串行口的方式来调试。



图 5-17 启动服务器程序

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

下的就是将应用程序烧写固化到开发板上去了。烧写的过程很简单。

在 uClinux 的提示符 `>` 下,键入以下命令,就可以完成对 flash 的擦除:

```
>flash_erase
```

进入目录 `/tmp/uclinux/romdisk`,键入以下命令完成烧写工作:

```
>flash_romdisk -f romdisk.s19 -m MAC 地址
```

其中,MAC 地址是 48 位的网卡物理地址。即需要定义的开发板的以太网卡的物理地址,其形式必须为: `xx.xx.xx.xx.xx.xx`。xx 必须为 16 进制的字符串。烧写一个应用程序的示例如图 5-18 所示。

图 5-18 烧写应用程序

前面讲过,对于 MC68EZ328 处理器,其内存、flash 及 I/O 是统一编址的。华恒开发板共 4M RAM,从 000000-3FFFFFF,2 片 1M FLASH,第一片:1000000-10FFFFFF。第二片:1100000-11FFFFFF。那么,如图 5-18 所示的:

```
addr = 01104000
```

```
.....
```

所处的位置在哪里呢?显然,是处在第 2 片闪存上!开发板上其实有 2 个 ROM FS,第一个在第一片闪存上,地址在 1000000-10FFFFFF,第二个在 1100000-11FFFFFF 处。这样可保证系统不至于因为用户的误操作而崩溃。

烧写完毕后,将开发板重新启动。在/bin 目录下,就会发现增加了应用程序 client。这样,第一个应用程序从编写、改编、运行调试直到烧写入闪存的全过程,就介绍完成了。

(7) 关于 .s 型文件的说明

S 记录格式是 Motorola 公司的十六进制目标文件格式。它将程序和数据用可打印的 ASCII 格式表示,允许用标准的软件工具来检查目标文件,也可以在传送过程中显示其内容。S 记录格式还包括出错检验功能,可保证数据传送的正确性。此外,S 记录很容易编辑。

① S 记录内容

S 记录实际上是由五个部分组成的字符串集合。它包括记录类型、记录长度、存储器地址、程序/数据及校验和。每个二进制数据字节编为两个十六进制数字字符,第一个字符为字节的高四位,第二个字符为低四位。组成一个 S 记录的 5 个部分如下表所示。

字 段	字 符 数	内 容
类 型	2	记录类型(S0、S1 等)
记 录 长 度	2	记录中除类型和记录长度外的字符对的数目
地 址	4、6 或 8	数据将装入的存储器地址,地址可为 2、3 或 4 个字节(取决于记录类型)
程 序 / 数 据	0~2n	为 0~n 个字节可执行程序(目标程序)或数据或描述信息
校 验 和	2	它为组成记录长度、地址和程序/数据的所有字符之和值的反码的低字节

② S 记录类型

S 记录共定义了八种类型。它们提供了编码、传送和译码的功能。Motorola 的装入程序、记录传送控制程序、交叉汇编程序、文件生成和调试程序等均要使用 S 记录格式。以 MC68HC0 为例,简化了的 8 交叉汇编只使用两种类型:S1 和 S9。

S1:包含程序/数据和两字节地址,表示程序/数据的存放首地址。

S9:用于 S1 记录的结束记录。地址部分应包含程序的执行起始地址的两字节地址。如不定义,则采用输入时第一次遇到的入口地址,或默认此地址为 0。这部分没有编码/数据字段。

③ S 记录示例

下面是一个典型的 S 记录模块:

```
S12301009754545454DE0113BF50A40F97D60113B7518130313233343536373839A1424363
S106012044454609
```

S9030000FC

这个模块包含两个 S1 程序/数据记录和一个 S9 结束记录。

- S1:S 记录类型 S1,表示程序/数据记录,地址为两个字节。
- 23:十六进制 23(十进制 35)表示后面有 35 个字符对,即后面有 35 个字节二进制数,包括两个字节的地址、32 个字节程序/数据和一字节检验和。
- 0100:两字节地址(\$0100),表示后面程序/数据要装入的存储地址。
- 97...43:32 个字符对,是实际的程序/数据的 ASCII 字节。
- 63:第一个 S1 记录的校验和。
- S9:S 记录类型 S9,表示为一个结束记录。
- 03:十六进制 03,表示后面有 3 个字符对(3 个字节)。
- 0000:表示两字节地址,为 0。
- FC:S9 记录的校验和。

Motorola S 格式记录类型的完整定义如下:

- S0:S 格式文件的第一个记录,以 16 进制 ASCII 码值的形式记录本文件的文件名,首尾包括记录长度和校验码。
- S1:地址为 2 字节的代码/数据记录。
- S9:S1 类型文件的结束行记录。
- S2:地址为 3 字节的代码/数据记录。
- S8:S2 类型文件的结束行记录。
- S3:地址为 4 字节的代码/数据记录。
- S7:S3 类型文件的结束行记录。
- S5:如果有,标记本文件总共有多少个 S1、S2 或 S3 记录。

(8) 补充说明

在本节最后,对开发中的一些问题再作几点补充说明。

① 如何自动启动用户应用程序?

在宿主机/uclinux/appsrc/init/init.c 中加入一行 fork 即可。这属于用户应用程序部分,不属于内核。因此用户改完后,在/appsrc 下执行 make,即可生成 romdisk.S19,将其烧入 flash 即可实现在开发板启动时自行启动用户应用程序。

② 对 IP 的修改。

开发板出厂默认的 IP 为 192.168.2.124。这是由/appsrc/loattach/loattach.c 设置的。它的可执行文件在烧写 ROMDISK 时烧入板子,并在启动时执行。用户若要改板子的 IP,则只需改写 loattach.c 中的 IP 值,并对应修改 addroute 函数中的 dest net 值。

例如:要将板子的 IP 改为 10.0.0.5,则 loattach.c 中对应的改动有两处,分别为:

```
1.setifaddr(deve, "10.0.0.5");
2.addroute(deve, RTF_UP/* | RTF_HOST */ ,
"10.0.0.0" /* dest net */ ,
"255.0.0.0" /* netmask */ ,
0 /* gateway */);
```

编译后重新烧写 ROMDISK 即可。

③ 烧 flash 时死机怎么办？

在 minicom 下键入如下命令烧写 ROMDISK：

```
flash_romdisk -f romdisk.s19 -m MAC 地址
```

第一次执行 flash_romdisk 进行烧写实际上只是完成擦除 flash，而这可能会导致板子不响应，这时只要重启一次板子并重新执行上一命令即可。其实，烧写 flash 一次成功可能性很小，一般是两到三次才能成功。凡是遇到烧写 flash 时板子不响应的问题，一律只需重启重烧即可。

检查用户第二片 flash 是否烧写成功可以用以下命令：

```
cd /bin
```

查看里面是否增加了应用程序的可执行文件，如：handpad、sea 和 gui。华恒开发板的两套 ROM FS，一个在第一片 flash 中，另一个在第二片 flash 中，是用户烧写应用程序 ROMDISK 时烧入的。内核读取 ROM FS 时，正常情况以第二片中的用户 ROM FS 为准，若第二片 ROM FS 读取失败，则恢复读取第一片 flash 中的 ROM FS。因此即使用户烧写 ROMDISK 失败，板子也仍然能够正常启动。

● 板子经常重启。如在 flash_erase 执行时重启，这是怎么回事？

这是由于供电（如市电）电源电压过低，若出现此现象，建议采用 6V 稳压电源供电。

● 为何无法对板子上的文件进行写操作？

板子上的 ROM FS 为只读文件系统，是无法进行文件写的。只有 /tmp 是 RAM 盘，它是可写的。

● 板子 PING 不通怎么办？

先检查板子与 PC 机是否在同一网段；再检查 PC 机网络配置；最后检查网线、HUB 等。

● 使用 nfsmount 时无法挂载宿主机怎么办？

这一般都是由于宿主 PC 机的 NFS 没有配置好，或者是板子与 PC 机之间的网络根本没通。建议安装 Redhat6.x 时要选择定制，选择 everything，即完全安装。

5.4 本章小结

本章详细介绍了嵌入式 Linux 平台的构建方法和应用程序的编译、链接和调试的过程。首先，学习了如何构建一个嵌入式 Linux 操作系统，并通过一个实例来说明其具体实现的过程。然后，阐述了嵌入式 Linux 应用程序的编译和调试的方法。最后，以开发套件为平台，介绍了开发的常用模式，还以一个客户/服务器模式的通信程序为实例，详细地介绍了诸如改编程程序、编译、调试并烧写固化应用程序的方法和步骤。在本章中，对许多重要的问题和概念进行了阐述，希望大家好好研究体会。

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

第三篇

应用与提高

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

第6章 嵌入式Linux网络功能的实现

本章首先阐述了在嵌入式Linux系统下,接入互联网的基本方法。然后,讲述了Linux环境下网络编程的基本知识。同时,还介绍了与网络协议相关的概念。接着,给出了在开发套件下的一个网口通信的例子。在本章的后半部分详细讨论了嵌入式Linux系统的Web服务器,并讲述一个简单的Web服务器的样例来结束本章的学习。

本章主要内容:

- 连接入互联网的嵌入式系统
- 嵌入式Linux系统如何接入互联网
- Linux下网络编程基本概念和编程实例——网口通信
- HTTP协议和嵌入式Linux的Web服务器
- 一个简单的Web服务器的实现

6.1 连接入互联网的嵌入式系统

6.1.1 嵌入式因特网技术的兴起与前景

回顾Internet的发展和应用历史,不由得预测,Internet的明天对IT企业的战略设计和普通的家庭生活而言,都是意义重大的。

Internet的发展可分为以下几个阶段:

- 第一阶段:1980年—1990年,Internet发展的萌芽阶段。

Internet主要用来进行异种计算机的联网和信息交换。采用TCP/IP协议交换文件和信息,主要面向专业领域如银行、军用系统采用不同操作系统的大、中、小型计算机的联网,这时并不称之为Internet。TCP/IP网络仅是少数计算机专家的概念,这是Internet发展的萌芽阶段。

- 第二阶段:1990年—2000年,PC作为客户机,Internet席卷全球。

在TCP/IP协议网络之上发明的E-mail和WWW普遍应用,Internet国际互联网被大众接受。这时一个重要的条件是PC普及得非常广泛,即形成了Client/Server体系结构(客户机/服务器),进而发展为Browser/Server结构(浏览器/服务器),这时的客户机是已广泛普及的PC,而服务器是相对复杂的、价格昂贵的超级计算机,即所谓的“胖服务器”。Internet的发展使IT界的老牌公司也没有料想到会产生路由器的巨大需求,Cisco公司的

成功就在于此。另一方面,WWW 的应用又造就了一大批以 yahoo 为代表的 .com 公司,以及电子商务公司,这不得不承认 Internet 像一场革命改变了世界。

● 第三阶段:2000 年—2010 年,嵌入式 Internet 时代。

连入互联网的嵌入式系统的出现,将给地球披上“电子皮肤”,嵌入式片上系统(System On a Chip)称为“瘦服务器”。预测未来 Internet 将向何处去,这是全世界科学家关心的问题。贝尔实验室总裁 Arun Netravali 在内的一批科学家对此做出了预测:在这阶段,将比 PC 时代多成百上千倍的瘦服务器和超级嵌入式瘦服务器,这些瘦服务器将把世界你能想到的各种物理信息、生物信息相联接,通过 Internet 网自动地、实时地、方便地、简单地提供给需要这些信息的对象”。

由此可见,如何设计和制造嵌入式瘦服务器、嵌入式网关和嵌入式因特网路由器已成为嵌入式 Internet 时代的关键和核心技术。嵌入式 Internet 广泛应用将这个世界变得更加自动化、智能化和人性化。如今,嵌入式瘦服务器已经在市场上流行起来,图 6-1 所示,就是某日本公司推出的基于嵌入式 Linux 的瘦服务器。



图 6-1 某日本公司推出的基于嵌入式 Linux 的瘦服务器

Internet 的伟大之处就在于可以无限地膨胀。《数字化生存》的作者尼古拉·庞帝来中国讲演时,曾预言未来 PC 的市场份额将减少,可能会出现许多非常便宜的因特网接入设备,他建议的价格只有 1 美元。而单片机或微控制器(MCU),通称为嵌入式系统,已经在家庭和工业的各个领域得到了应用。目前大多数嵌入式系统还处于单独应用的阶段,以 MCU 为核心,与一些监测、伺服、指示设备配合实现一定的功能。Internet 现已成为社会重要的基础信息设施之一,是信息流通的重要渠道。如果嵌入式系统能够连接到 Internet 上面,则可以方便、低廉地将信息传送到几乎世界上的任何一个地方。

将嵌入式系统与 Internet 结合起来的想法其实很早以前就有了。主要的困难在于,Internet 上面的各种通信协议对于计算机存储器、运算速度等的要求比较高,而嵌入式系统中除部分 32 位处理器以外,大量存在的是 8 位和 16 位 MCU,支持 TCP/IP 等 Internet 协议将占用大量系统资源(后面将详细介绍 8 位和 16 位嵌入式系统接入互联网的方法)。以嵌入式微型因特网互联技术为基础的解决方案,可以把这种假设变成现实,实现嵌入式设备的 Internet 网络化。

在我国,前些年比尔·盖茨“维纳斯”信息家电计划一度闹得纷纷扬扬,而中科院也推出了自己的“女娲计划”与之对抗。2000 年 8 月 18 日,国内首个基于 Linux 操作系统的嵌入式上网设备方案“蓝点嵌入!(BluePoint Embedded!)”推出,其解决方案重点用于

WebTV 和 Set-Top Box 设备,可以使普通用户通过电视机就可以轻松上网,宽带用户还可以点播交互式节目。蓝点公司在 Linux 内核上进行了深度研发,突破了内存占用大和实时性弱等限制,BluePoint Embedded System 对内存的占用降至 300K 以下,使之适合内存用量小的互联网设备。为信息家电度身订做了 TV-Friendly 图形用户界面、flash 和 Java,以及著名的浏览器软件 Netscape Navigator 4.7 等。

计算机发展的目标是专用电脑,实现“普遍化计算”,因此可以称嵌入式智能芯片是构成未来世界的“数字基因”。沈绪榜院士预言:“未来十年将会产生针头大小、具有超过 1 亿次运算能力的嵌入式智能芯片”,这将为用户提供无限的创造空间。

随着市场对超微型嵌入式应用技术的不断增长,以及半导体技术和系统设计方法的进步,在一个硅片上实现一个(过去以为)复杂的系统的时代已经来临,这种芯片称之为“片上系统”(SOC)。SOC 的出现将改变并深刻地影响传统的集成电路产业的现状,使设计和功能定义以及决策者的远见变得更为重要,这一点也证实了“在后工业时代的经济中,发明创造的价值超过批量生产,所有垂直的东西突然变成水平”的经济学现象。

6.1.2 嵌入式 Internet 的应用

嵌入式 Internet 技术具有广阔的应用前景,其应用领域可以包括:

(1) 智能公路

包括交通管理、车辆导航、流量控制、信息监测与汽车服务。

(2) 植物工厂

包括特种植物工场,如实现野生名贵药材的远程监控培养和种植、无土栽培技术应用、智能种子工程等。

(3) 虚拟现实(VR)机器人

包括交通警察、门卫、家用机器人等。

(4) 信息家电

包括冰箱、空调等的网络化。

(5) 工业制冷

包括冷库、中央空调与超级市场冰柜。

(6) VR 库房

包括粮库、油库、食品库等。

(7) VR 精品店

指客户可以在 Internet 上实时地看到存货状况。

(8) VR 家政系统

包括水、电、煤气表的自动抄表和安全防火、防盗系统。

(9) 工业自动化

目前已经有大量的 8、16 和 32 位嵌入式微控制器在应用中。网络化是提高生产效率和产品质量、减少人力资源的主要途径,如制药工业过程控制、电力系统、电网安全、电网设备监测和石油化工系统。

(10) POS 网络及电子商务

包括公共交通无接触智能卡(CSC)发行系统、公共电话卡发行系统和自动售货机。

6.1.3 嵌入式 Internet 的原理

如前所述,在 8 位和 16 位 MCU 上实现 Internet 通信协议是比较困难的。如果将现有嵌入式系统中的 MCU 都更换成 32 位或 64 位的高性能处理器,从经济性和现实性上来说都不太可能。EMIT (嵌入式微控制器 Internet 技术)从另一个角度出发,对这个问题进行了很好的处理。

EMIT 采用桌面计算机或高性能的嵌入式处理器作为网关,称为 emGateway。它支持 TCP/IP 协议并运行 HTTP 服务程序,形成一个用户可以通过网络浏览器进行远程访问的服务器。emGateway 通过 RS-232、RS-485、CAN、红外、射频等轻量级总线与多个嵌入式设备联系起来。每个嵌入式设备的应用程序中包含一个独立的通信任务,称为 emMicro。监测嵌入式设备中预先定义各个变量,并将结果反馈到 emGateway 中;同时 emMicro 还可以解释 emGateway 的命令、修改设备中的变量或进行某种控制。

添加到嵌入式系统中的 emMicro 代码长度一般在 1~8 K Bytes 左右,不会影响 MCU 的正常运作。这样仅增加了一个 emGateway 网关,就解决了嵌入式设备上 Internet 的问题。网关还可以同时管理多个嵌入式设备,从而优化嵌入式网络的结构。

除桌面计算机和嵌入式处理器以外,emGateway 还可以作为 ISP 服务器中的一个任务运行,仅用软件实现。

1. 嵌入式 Internet 的开发

EMIT 技术包括一套嵌入式 Internet 的开发工具,其中包括多个部件,均以 Embedded Microcontroller 的字头缩写——emXxxx,命名各个部件,如 emGateway 称为嵌入式微控制器网关。EMIT 开发平台包括:

(1) emMicro

emMicro 是适合于小型电子设备的微型网络服务器。emMicro 驻留在嵌入式设备中,是 emGateway 和嵌入式设备系统软件之间的通信服务模块。emMicro 占用的字节可以小到 1K Bytes,和 emGateway 一起,为 8 位和 16 位嵌入式设备提供网络服务器功能。

(2) emGateway

emGateway 是 EMIT 分布式网络平台的关键。它是轻量级设备网络(如 RS-232、RS-485、CAN、RF 等)和大型高性能网络(如 Intranet 和 Internet)之间的桥梁。emGateway 提供 emMicro 中没有包括的网络服务功能,并且可以与多种用户界面相连接,如网络浏览器、数据库、应用程序等。EmGateway 能驻留在 PC、单板机、ISP 服务器或 32 位以上的嵌入式处理器上。

(3) EMIT Access Library

EMIT Access Library 是一个可以在通用高级语言(C、C++、Java、Visual Basic 等)下调用的库函数,实现从一个通用程序(如网络浏览器)或用户程序中访问和监测的设备。EMIT Access Library 能够将嵌入式设备中的数据输出到一个大的数据库或客户应用程

序中。EMIT Access Library 中包含按钮、表头等各种指示、控制控件,供开发人员选用。

(4) emLink

在 emGateway 中,支持最常见的物理层协议(RS-232、RS485、RF 和 etc.)的数据链路功能,为每个外部嵌入式设备提供通信管理功能,以保持网络连接。emLink Toolkit 允许开发者修改通信链路以适应特殊的网络物理层连接。

(5) emObjects

emObjects 是预先建立的 Java 对象,能实现从标准网络浏览器中访问和控制嵌入式设备。EMIT 的图形化开发界面非常简单直接,并具有多种预先制作好的控件,供用户在 emGateway 主页中选用。在嵌入式中增加 emMicro 通信任务时,并不需要大量改变原有代码。原来的代码是用 C 语言还是汇编语言编写并不会影响嵌入式设备的网络化开发。这样不但是新设备,而且现有的很多嵌入式设备也可以通过简单的改造实现网络化信息交流与控制。

嵌入式 Internet 与 MCU 技术密切相关,需要多方面的协作,因此包括 Motorola、Siemens/Infineon 和 Philips 在内的数十家公司联合成立了“嵌入式 Internet 联盟(ETI)”,共同推动这一市场。可以预言,嵌入式设备与 Internet 的结合代表着嵌入式系统和网络技术的真正未来。

2. 用于家庭网络的嵌入式实时 Linux

在家庭网络的嵌入式系统中,需要一个实时的操作系统来连接整个网络。它的功能在于:对内,管理家庭内部网里面的智能家电的运作、协调。对外,可以连接 Internet,实现家庭内部网和广阔 Internet 的连接,并且支持远端对家庭内部设备的控制与监测。

(1) 选用 Linux 作为家庭网络操作系统的理由

选用 Linux 作为家庭网络操作系统,是基于对网络实时操作系统的要求和 Linux 本身的特点的。

从性能上讲,实时操作系统(RTOS)和普通的操作系统(OS)存在的区别主要是在“实时”二字上。“在实时计算中,系统的正确性不仅仅依赖于计算的逻辑结果,也依赖于结果产生的时间。”从这个角度上看,可以把实时系统定义成“一个能够在实现指定或者确定的时间内完成系统功能和对外部或内部、同步或异步时间做出响应的系统”。这个定义要求:

- 系统应该有事先定义的时间范围内识别和处理离散事件的能力。
- 系统能够处理和存储控制系统所需要的大量数据。

而家庭网络实时操作系统(HOMENETWORKING-RTOS)在这个基础上又增添了以下几种要求:

- 系统应有稳定、高速、安全的网络支持。
- 可以实现多用户多任务的支持。
- 有一个易于控制的 GUI 接口。
- 系统应对网络设备有较好的支持。

Linux 内核对网络协议栈的设计是从简洁实用的角度出发,实现了一整套的网络协议模块。Linux 不仅可以支持一般用户需求的 ftp(file transfer protocol)、telnet 和 rlogin

协议,还能提供对网络上其他机器内文件的访问(NFS,网络文件系统)。Linux 还能支持 SLIP(Serial Line Interface Protocol)和 PLIP(Parallel Line Interface Protocol)协议,使通过串口和并口线进行连接成为可能。通过 AX.25 协议,Linux 可以提供通过无线电进行连接的方式。通过在 Linux 上开发 Novell 标准的 IPX 协议,Linux 可以访问 Netware 网络。如果在 Apple 机里,可以通过 AppleTalk 协议访问 Apple 的网络。在 Windows9x/NT 局域网里,可以通过 Samba 协议进行 Linux 和 Windows 之间的文件共享。通过 Apache 公司开发的免费网络服务器,可以利用 Linux 系统作为强大的网络服务器,提供电子商务和互联网数据服务。

除此之外,Linux 对网络中最常用的 TCP/IP 协议有着最完备的支持。这些在其他技术文档上都有详细的介绍。在 GUI 方面 Linux 直指业界的 X 窗口标准,并且在 X Window 上有完善的开发工具与开发环境。缺点在于它的体积过大,不利于嵌入式系统的要求。

当然 Linux 最大的优点在于它对多任务多用户下的高效性、安全性和稳定性。目前,大量的硬件厂商开始提供驱动,同时大量的硬件人员也开始为 Linux 书写一些驱动。这样在驱动方面可以省下大量的时间与人力物力。

作为家庭网络的实时操作系统,并不要求深内嵌,所以 Linux 可以在它的可缩减范围内达到用户的要求。而它在多用户多任务方面的优秀表现以及它所提供的大量驱动程序足可以弥补这个损失。最后还有最吸引人的一点,Linux 下的任何软件都是免费的。

综上所述,只要改造 Linux 的内核,为其添加处理实时任务的处理机制,它就是家庭网络嵌入实时系统的最好选择。

(2) Linux 内核的改造

● 内核体积的改造

由于家庭网络操作系统是装在中央控制器上的,所以第一步要做的就是剪裁 Linux 内核,将原先比较庞大的内核(相对嵌入式系统来说)改造成一个小巧的可配置的内核。要保持 Linux 的基本功能以及对 Linux 应用程序、驱动程序的完全支持。

本书已经提到过,Linux 目前可以运行在 Intel x86、Motorola/IBM PowerPC、Compaq (DEC)、Alpha、IA 64、S/390、SuperH 等处理器上。这样,内核中为这些处理器写了大量的代码,可以根据自己的处理器缩减有关内容。作为嵌入式系统,对于内核中大量的驱动和对 VFS 的支持可能都用不到。在这些部分可以做大量的剪裁工作。因为 Linux 本身就是在很小的一个内核上发展起来的,所以在剪裁上不存在多大问题。

● 内核可配置性的改造

Linux 内核采用的是整体式结构,因此对内核进行配置不是很容易。对内核功能的配置主要运用模块编程。对于常用组件,提供具体的任务模块。将用户编写的一个配置文件进行动态的编译加载,以减少通常意义上的模块编程带来的系统性能损失。

● 内核实时性的改造

对于 Linux 的实时性改造,将引入一个双内核结构。引入一个实时内核,处理一切实时进程。利用 Linux 的内核,可以实现一个建立在这个非实时内核基础上的实时内核,这两个内核共同工作,形成前面所描述的双内核实时系统。这样的实时内核可以满足短小精悍的要求,非实时内核又已经如前面所描述的那样强大,两者结合起来,可以充分发挥

出实时系统在嵌入式系统中的作用,也可以充分让嵌入式系统满足信息电器时代的要求,开发出强大合适的系统。当系统有实时任务时,将任务直接交由实时内核处理。而一般的进程通过 Linux 的调度器以后交给实时内核,然后才能运行。

关于实时进程和普通进程(用户/Linux 核心)的通信,可采用 FIFO 设置管道件、共享内存和自定义 API 接口三种方式。

● 编程接口

对于实时 Linux 系统,因为它完全支持 Linux 程序,所以可以运用一切在 Linux 上惯用的技术和手法来编制程序。运用 GNU 的系列编程工具,可以为自己的系统添加功能。

对于实时程序的编制,将提供一套组件工具和 API。所有的实时程序将以模块的形式运行。

(3) 网络支持

实时 Linux 支持所有 Linux 的网络协议。它把 TCP/IP 作为默认的网络协议,其他应用协议也可选。网络支持的主要任务是通过 Web 方式控制各个智能家电设备,可以通过一个内置的 Web 服务器,建立和各个设备的连接。同时,外部的连接通过这个 Web 服务器,可以进行远程控制并与家庭设备进行数据通信。

6.2 使用 Linux 来构建嵌入式网络设备

在上节中,概述了嵌入式设备接入互联网的情况和 Linux 在嵌入式设备中的使用。本小节,将论述如下问题:

- 如何评价 Linux 应用于小型网络装置中的优势?
- 如何构建嵌入式的网络设备?
- 如何将 8/16 位的嵌入式设备接入互联网?

6.2.1 低成本的嵌入式网络电器设备

低成本、嵌入式、网络应用设备这几种性质形成了一个有趣的组合。这个独特的组合使嵌入式系统的功能达到新的水平,从前不能做的事情,如今却成为了可能。

人们对于嵌入式设备的造价很敏感,尤其是那些用于大众消费的设备,而且这种价格压力会随着智能电器设备的逐渐普及而继续存在。低成本需求意味着硬件集成化,所有硬件数量减少而个体功能增强。随着市场上大众消费的多元化,硬件价格在逐步降低,制造更高级的计算装置也变得更加容易。这种计算装置只需要几十美元,但应用广泛,无所不能。

虽然可以用相对低廉的成本建立功能强大的电器设备,但不可否认,个人电脑及网站更具威力。比较起来,网络电器设备稍显逊色。它们只能是一些执行特定任务的小型系统,而不能成为通用的计算机平台。

那些嵌入式的低成本网络设备需要的硬件数量是很少的。它们的 RAM、ROM 存储容量较小,特别是其中没有块存储装置(硬盘、CD-ROM 等);其 CPU 比起 PC 和用做服

务器的电脑来说运行速度慢、专业化强;其外围设备受到的限制更多。这些特性使建造低成本、嵌入式电器设备变得容易。

可与外界进行通信是网络化电器设备的重要方面。因此,需要某种网络界面。究竟哪一种界面合适,要看设备的既定用途。现有通用的接口有异步串行(将广泛通过模拟调制解调器使用)和以太网口两种形式。以太网已成为各种类型 LAN 网络的最佳选择,其中还包括要求较高的工厂管理应用软件。以太网同样也是进入新的 WAN 路由像 ADSL 和 Cable 调制解调器的明智选择。当然,还有很多途径可供选择接入互联网,包括并行端口、同步串行口、光纤和无线等。

关于各种可编程设备的另一个重要问题是界面的配置,就是说,用户如何配置和设置嵌入式装置。许多嵌入式装置没有用户界面。设备网络化使这一问题简单化了。最简单的办法就是在嵌入式网络装置中使用一个小的网络服务器。这样做有许多优点,网络浏览器大都无处不在,并且使用起来很方便。

嵌入式装置中还存在许多限制,装置的大小及安装是两个主要因素。一些嵌入式装置需要安装在母设备之中,但随之带来的问题是它们要共用电源并产生其他消耗。这些限制使元件的选择受到很大局限并提高了嵌入式装置的造价。

影响嵌入式装置批量生产造价的主要因素是硬件的建设。其前期工程造价可以在产品的寿命中分期支付,但问题是产品推向市场的时间——即从最初产品形成到批量生产的时间。一个明显的趋势是现代化、复杂的网络化电器设备的发展进步主要依靠软件工程,嵌入式网络化电器设备的核心是开发功能强大的软件系统。

正如用户所看到的,嵌入式网络电器设备所具有的特点和局限性使其独具特色。很难找到基于最佳性能价格比的切合点,因为嵌入式网络系统家族的每个成员都与众不同。

嵌入式网络电器设备可用于许多地方。现在应用的实例有:个人计数助理(PDA,例如掌上电脑)、导航系统、网络印刷机、机顶盒、手提音频播放器、远程数据采集器和智能工厂自动化装置。

1. 使用 Linux 的理由

现在世面上有许多嵌入式操作系统可供选择。有可免费获得源代码的操作系统,像 RTEMS 和 uC/OS、VxWORKS 和 QNX 系统。现在这个竞技舞台上又有了新成员——Linux。

嵌入式网络电器设备采用 Linux 操作系统是个明智的选择,单从造价考虑就能说明这一点。采用免费核心软件有利于建立低成本系统。

另一个主要的优势是 Linux 支持许多微处理器、计算机平台和一系列外围设备。有这样强大的硬件支持,为设备投放市场缩短的时间是不可低估的。设计嵌入式网络电器设备时,其硬件开发者选择元件的余地也更大。而那些只适用于单一用户使用的操作系统就不存在这种优势。

网络堆栈和相应的工具对于网络化电器设备设计的成败至关重要。Linux 在这两方面同样具有优势,其网络堆栈成熟、开发工具独一无二。这给嵌入式装置提供了强大的网络支持。除此之外,还有大量的免费软件可以在 Linux 系统下运行。这为设计嵌入式网络装置的软件工程师提供了广阔的选择余地。从简单的命令行设置工具到网页脚本语言

等等无所不包。

以 Linux 作为嵌入式装置核心软件驱动的另一优点是 Linux 下编译、链接及调试工具也是免费的。在第 5 章已经介绍过,建立 GNU 工具链的成本极低,获得源代码显然可以使嵌入式软件工程师免于受编译器的干扰。

当然,并不是每一个选择都是十全十美的。在嵌入式网络装置中使用 Linux 也有一定的负面影响。Linux 比其他嵌入式操作系统占用的内存要大,这意味着装置需要的 RAM 空间更多。但是由于硬件,特别是 RAM 价格在不断下降,所以这并不是一个大问题。

Linux 作为通用交互式操作系统,其设计主要围绕文件系统。通常微型嵌入式系统没有内置文件系统,所以 Linux 需要为嵌入式装置提供一个文件存储器,还要在内核中设置文件系统代码以便可以运行文件,提高内存利用率。随着网络服务器作为网络通信的主要手段的广泛应用,嵌入式装置中也设置网页。最简单的范例就是存储文件,像普通网站中所做的一样。

2. 硬件平台范例

下面详细介绍低成本、嵌入式网络电器设备的实例。以 Moreton Bay NETtel 路由器为例。Moreton Bay NETtel 是一个低成本 Internet 路由器家族。它被设计成专为小型以太网(如家庭或办公室等环境)提供服务,带有简单的安全校验功能。不同的 NETtel 路由器可提供不同的连接方式选择,从模拟拨号调制解调器到高速 WAN 接口连接(Cable 和 ADSL 解调器)。另外,一些 NETtel 家族成员提供了标准外设接口(如 USB 主机接口),同样也适用于常规嵌入式装置。

NETtel 硬件平台专为低造价产品设计,充分考虑其 CPU、RAM/ROM/flash 资源及外围设备的选择。

(1) 硬件核心

作为核心部分,NETtel 硬件平台由摩托罗拉 5307 型 CPU(90MHz,4KB 内部高速缓冲存储器)、4MB SDRAM、1MB flash ROM、10 MBPS 以太网和 2 个 RS-232 串行接口元件组成。NETtel 家族产品中,还有些别的配置供选择,包括额外的 10BASE-T 以太网口、1 PCI slot 和 2 USB 主机接口。

利用高集成装置后,元件数量大大减少,最终生产成本也降低了,同时占用 RAM 和 flash 存储空间也小了。低造价、嵌入式装置的典型特征就是占用极小的内存空间,该产品也不例外。NETtel 没有大规模的存储结构,其操作代码和调试信息都存放在安全可靠的 ROM 中。

(2) 选择 CPU

NETtel 的核心部分是摩托罗拉 ColdFire CPU。它是一种专业化的嵌入式 CPU,运行速度快且造价低。每个芯片由许多标准逻辑元件组成,包括高速缓冲内存、中断控制器、直接存储器、计数器和存储器控制回路与片选装置,这同 MC68EZ328 很相似。而在传统的工作站及个人电脑中,这些部分所执行的功能都由 CPU 外部的芯片完成。

ColdFire 是畅销的摩托罗拉 68000 处理器系列中的衍生品。ColdFire 的指令和寻址方式与 68000 系列兼容,整个 CPU 内核的结构也相似。ColdFire 处理器是 32 位的,但它

没有虚拟内存,这给选择操作系统带来了问题。值得庆幸的是有一种 Linux 嵌入式版本,它是专门为那些没有内存管理单元的 CPU 设计的,它就是本书中进行了详细分析的 uClinux。

(3) 随机存储器及闪存的存储空间

NETtel 用 1MB 内存存放操作系统的内核、超级用户文件和系统配置文件。闪存是一个比较稳定的内存(当电源切断后保存在里面的内容不会丢失),可以重新编程设计其内容。通常来讲,内存容量越小,造价越低,所以怎样减小内核和应用软件占用的空间成为工程师最关注的问题。一些应用软件可以通过删除部分不必要的功能以减少内存占用量。

NETtel 中有 4MB 的 SDRAM 空间,主要用来存放正在运行的程序与相关信息,同时还包括了 RAM 文件系统。RAM 文件系统为一些应用程序的运行提供了临时内存,设置状态信息也记录在 RAM 中。

(4) 外围设备

外围设备为嵌入式装置连接网络提供了便利。串行接口在 ColdFire CPU 中属于集成设备。在 NETtel 中,可以有选择地添加两个 USB 主机接口,使其具有更好的扩展性。

3. 构建嵌入式网络设备的步骤

构建任何一种嵌入式装置与编写特定的应用软件是完全不同的。建设嵌入式装置是一项高技术的、耗时的任务。传统上新系统的建设需要分工合作,融合许多已有的源代码。

以 Linux 作为嵌入式装置操作平台在很大程度上解决了运行嵌入式系统出现的一些问题。现在,嵌入式装置中的 Linux 系统内核元件与普通工作站的基本相同,同样其软件应用环境也相同。这意味着许多程序运行界面和工具都可以应用到嵌入式装置的建设中。

uClinux 内核提供了同标准 Linux 兼容的 Linux 接口。需要的主要工具与工作站用的 Linux 相同,只不过要接入 ColdFire 处理器体系。而华恒公司的开发平台是基于 M68EZ328 的 CPU,两者是很类似的。

下面是一些关于开发 NETtel 方法的叙述。可以看到,应用标准操作系统和工具是如何大大简化嵌入式操作系统的建设进程的。

(1) 开发工具

GNU 工具箱提供了一系列编辑、汇编、链接及调试功能。实际上,它们的性能正如前面所讲的那样,与商业销售软件工具不相上下。在第 5 章中已经详细讲解了 GNU 工具是如何对嵌入式技术进行开发的。对于网络设备,当然也类似了。

对于 NETtel 的产品而言,使用的编译器不是 gcc 而是 GNU egcs,版本为 1.1.2。选择后者是因为它对 ColdFire 处理器的支持更好。另外,GNU 工具链被用来设置产生 ELF 格式的输出,而不是前面提到的 flat 格式。

(2) 将 uClinux 移植到 ColdFire 处理器上

现在来看看将 uClinux 移植到 ColdFire 处理器上的方法(关于 uClinux 的移植方法,本书在第 11 章将详细论述)。将现有的软件系统接入新的处理器体系中还有许多要解决

的问题。例如移植内核时,首先要将内核编译成针对该处理器的目标代码。Linux 内核包含了很多的内联汇编。即使对于运行在 M68K 体系的 CPU 上的 uClinux 而言,ColdFire 也有很多不同于其他同体系 CPU 的地方。这表明许多内核程序需要改写,特别是堆栈指针的不同会导致不少的问题。接着,要为系统选择合适的内存分配。所有的外部设备都映射到了处理器的物理内存空间上。类似地,所有的运行程序进程和内核都映射到了同一个物理地址空间,所以,必须对系统的内存非常清楚。下一步就是要考虑修改代码来支持 ColdFire CPU 的内置外设。这包括了对中断控制器、计时器等重要部件的支持。同样,必须对与处理器相关的重要外部设备进行很好的支持,例如 flash 等。flash 很重要,它可以充当系统的启动设备,还可以存储重要的非易失性的设置信息。

通过编写设备驱动程序,可以让硬件平台支持更多的外部设备。为了支持 ColdFire 内置的异步串行端口,需要编写新的驱动程序。Linux 对通用设备的支持是相当好的,支持的设备种类也很多,例如对以太网控制器的支持就内建于 Linux 中。用户所要做的就是针对特定的硬件平台的地址差异,修改驱动程序的代码。

最后就是对二进制应用程序的支持。由于缺乏虚拟内存,使用 uClinux 的 Coldfire 处理器在使用常规的二进制文件(例如,ELF 文件)时显得非常的不方便。所以在使用 uClinux 平台时,使用了一种新的文件模式,即先前提到的 flat 格式的文件。该文件包含了通常应用程序要用到的代码和数据段,但同时还包含了重新定位的信息。当某个程序被载入执行时,它需要重新定址。对于应用程序而言,该过程是透明的。

(3) 硬件与软件的集成

大多数嵌入式系统的设计都是与常用的硬件平台兼容的。这就意味着在硬件平台上,应不断地对系统及相关应用软件进行调试与完善。

使用 Linux 这样的操作软件大大简化了这一工作。所有的源代码都是公开的,系统工程师可以在此基础上对内核的任何部分加以改动使其更好地支持嵌入式硬件平台。

(4) 调试

嵌入式装置的调试工作是一项难度高而且费时的工作。请参看第 5 章中的相关内容。

未来的产品具有更强的生命力。所有的系统都必然要不断维护、升级以适应新的环境,完成新的任务。使用 Linux 和可免费获得的 GNU 工具链来建设网络设备,用户可以在拓展产品功能方面取得领先地位。

6.2.2 使用 Linux 将 8/16 位的嵌入式设备接入互联网

前面探讨了使用 Linux 将低端的嵌入式设备接入互联网的问题。现在,具体地讨论 8/16 位嵌入式设备的接入的实现问题。

在 6.1.3 节讲过,8/16 位的小型嵌入式微型控制器的网络通信是通过大型 32 位平台上的网关系统实现的。有了嵌入式 Linux,这些网关可以很容易地从 PC 宿主机上移植到诸如 PC104 等嵌入式平台上。

1. 网关法

网关系统提供了一个嵌入式的构成体系,使即便是最小的嵌入式微控制器都有网络功能。建立一个网关,让它作为轻型设备网络(RS-232、RS485、Modem、IR 和 RF)和重型设备网络(包括 Intranet 和 Internet)间的媒介。这样,就可以使嵌入式设备与外界进行通信。网关可以放置在 PC 机、单板机或者某个带有充足资源的设备上(要 32 位处理器)。网关系统有较多的资源,可以加强嵌入式设备的功能,从而实现更多信息和数据的交换,如图 6-2 所示。

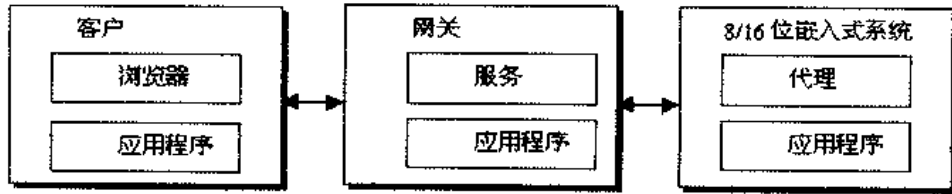


图 6-2 使用网关将嵌入式设备接入网络的实现方法

网关系统将嵌入式设备接入 Internet 或 Intranet,并管理其服务。针对嵌入式设备而言,用网关法实现网络功能有如下特点:

- 可充当嵌入式设备的防火墙

在必要时,嵌入式设备通过网关可提供认证和加密等功能。同时,网关还可以通过执行当前最流行的安全协议加强嵌入式设备的安全性能。

- 协议传输

网关提供很多方法适应不同的硬件和软件连接方案,并提供一种基于 TCP/IP 方案的接入服务。

- 设备监视

网关的一个可选功能是监视设备的状态。无论嵌入式设备是否一直连接在网络上,也无论该设备是否需要周期性地同网关保持联系,当某个设备不见了,网关都会向指定的客户报告。

- 提供事件处理

从一个设备来的特殊事件会需要执行一个应用程序,或者发送一个呼叫或 E-mail 信息。在此时,与正在等待消息的“客户端”间不存在连接。

- 提供设备服务

网络可以为嵌入式应用提供一系列的网络服务,而不能将其视为嵌入式应用的客户端。此时网关的作用是向嵌入式设备的应用程序提供服务。

- 提供对轻型网络的支持

如同嵌入式处理器一样,对于单个的物理网络传输而言,它不可能适用于所有的网络环境。无论是基于成本、通信的可靠性、安全性或其他原因,嵌入式设备的网络解决方案都必须支持各种网络功能。

- 多样的接口选择

有很多种方法可实现与嵌入式设备的接口界面,包括使用 Windows、Web 浏览器和

电话等。对于嵌入式的电子设备而言,没有诸如“万能接口”之类的东西。对于某些特定的应用,一个简单的基于文本的 HTML 接口界面就足够了。然而对于其他某些应用而言,就显得过于简单。为了提供丰富的网络设备化的解决方案,需要提供人一机界面。

对于网关系统,还需要考虑以下几个方面的特性:

- 可使用任意芯片

在市场上有 1000 多种微控制器供选择做网关。每款微控制器都被设计用来降低成本,并提供一组特定的服务。没有哪款处理器可以胜任所有嵌入式系统的服务请求,必须挑选适用的芯片。

- 微型网络功能

对于 8/16 位的网络嵌入式设备而言,要提供诸如 RS-232、拨号、RS485、无线、红外及 CAN 端口的轻型和微型接入互联网的方式。没有任何一个物理网络传输可以适用于所有的应用环境。无论是基于成本、通信可靠性、安全性或其他别的原因,嵌入式系统的网络解决方案都必须支持多种多样的网络功能。另外,基于 8/16 位处理器的网络需要在低能耗、连续工作的环境下运行。此时,嵌入式设备并不一定可以连续地与网络保持连通。这些轻型的网络嵌入式设备与大型广域网络(包括互联网)的连接方法必须是高效及时的。与那些行之有效的通信手段的方法,如以太网和 TCP/IP 协议相比,没有任何一个单一的轻型网络适合于所有小型的嵌入式设备。

- 灵活的网关系统

网关系统必须有灵活性、可裁剪性,并可设置实现多种应用。用户界面可以在远端的 Web 浏览器、膝上型电脑或甚至一个掌上 PDA 上运行。在不需要用户界面的地方,嵌入式设备就不能直接地连接上应用程序或数据库。如果使用网关,在网关上就能对接口信息进行完全充分的描述,而不需要在嵌入式设备上存储接口信息,这样就可以避免占用资源。

- 面向对象模型

用户需要一个针对轻型设备的面向对象模型,以适应嵌入式设备的多样性和通信系统的不同之处,并满足其协同工作的需求,并允许用户构建集成多对象环境(如 CORBA、JINI 和 DCOM)。这样,就可以在更大的程度上分享信息,并可以协同控制。嵌入式设备的通信功能应该是该设备能力的一个证明。在许多应用中,最好把嵌入式设备当作抽象的数据类型。封装特定类型的功能,并围绕该封装进行标准化,就可以建立一个环境来加强网络/客户应用的发展。网关通过对象服务,为外部提供了一条同嵌入式设备通信的通道。例如,在 PDA 或者浏览器上运行的家务管理程序,可以管理基于 HVAC 对象定义的热水控制单元。网关提供了与 HVAC 间的必要对象接口,并将客户从特定的网络和设备特性中独立出来。

2. 操作系统和网关

同微处理器和微控制器一样,不同的操作系统也在不断地改变以迎合特定的工业需求。以前,工程师们必须从头开始设计操作系统,而今天,这个工作依然存在,但已不是工程师们的首要工作了。设计人员可以购买商业操作系统,并将注意力放在开发应用程序上,而不是操作系统的编写上。如果拥有一个预先构建好了的操作系统,就使工程师们可

从众多该操作系统支持的 CPU 中挑选自己需要的品种。最近,嵌入式系统的开发从以 Linux 为中心的开放源代码运动中获得了好处。采用该操作系统,使嵌入式系统的设计增加了对实时功能的支持。Linux 操作系统的灵活性使设计者可以实现其需要的功能。

选择 Linux 而不选择 Windows NT 的理由有如下几点:

- 需要在更多的 MCU 和 CPU 上运行用户的程序;
- 需要大约 450KB 大小的空间来存储 OS,还要 4MB~8MB 的 RAM;
- 需要一个稳定的系统;
- 使用根用户的特权来对机器进行远端管理;
- 防范病毒和进行安全隔离;
- 可以除去操作系统的不必要的部分功能。

如表 6-1 所示。

表 6-1 各种系统所需存储器空间比较

要求	微型系统	小型系统	中等系统	高端嵌入式系统	嵌入式 PC	桌面系统	服务器	高端服务器
RAM 大小 (MB)	0~0.1	0.1~4	2~8	8~32	16~64	32~128	128+	更多
ROM/flash/DIS	0.1~0.5 MB	0.5~2 MB	2~4 flash	4~16 flash MB	MB 级别	GB 级别	GB 级别	GB~TB 级别

在嵌入式系统中,不可能按“reset”来重新启动系统。系统高度的可靠性必须通过对嵌入式系统各个方面的完全控制才能实现。Linux 是基于 GPL 原则发行的,可以获得其源代码和其他重要的系统组成部分。Linux 的内核可以经过调整以满足系统的特点和性能的需求,不需要的内核部分可以从内核中剥离。

人们做了很多努力,将互联网的开放标准放到嵌入式设备上。但是,广域网络的连接协议,例如 TCP/IP、UDP/IP 或基于 IP 的 SNMP 协议等等,对于嵌入式应用而言功能有点显得多余了,因为它们只要使用轻型的网络协议就可以很好运行。

为了同大型的网络相连,更好的办法是使用一个服务器来处理那些繁重的协议。有些黑客已经设计了一些方法将 TCP/IP 包头同一个轻型的网络协议捆绑起来,但是这却加重了系统的负担。

为什么不直接将嵌入式 Linux 装在嵌入式设备上并运行 SNMP 呢?以常规的服务器标准来看,嵌入式 Linux 非常小,而且也很适合装备在 8/16 位的嵌入式设备中。例如在汽车工业中,使用了大量的 8/16 位微控制器。一个简单的现代汽车使用了 80 多个 8 位的 MCU,这些 MCU 在诸如安全气囊、防抱死装置、引擎控制等方面发挥了重大的作用。这类复杂的嵌入式系统的设计者,面临的一个具有挑战性的问题是如何控制这些重要的设备。人们常常挑选一个操作系统来管理这些外部设备。来看一个例子:安全气囊控制器要通过 UART 同其他外部设备通信。设计者可能挑选一个硬件 UART,这对 MCU 的要求就很高。或者设计者编写一个软件的 UART 管理通信事务,这对 MCU 的要求也很高。但是,使用一个操作系统就会大有好处。于是,使用 Linux 来完成所需要的



任务。

6.3 Linux 下的网络编程

在前面两个小节,学习了嵌入式 Linux 设备接入互联网的问题。从本小节起,学习在 uClinux 开发平台下,编写网络通信应用程序的方法。首先,学习 TCP/IP 协议的基本知识。然后,介绍如何在 Linux 环境下进行 socket 编程和其常用函数的用法及 socket 编程的一般规则,还有客户/服务器模型的编程应注意的事项和常遇问题的解决方法。通过几个实例,最后学习网络通信功能是如何实现的。

6.3.1 TCP/IP 协议概述

1. TCP/IP 的参考模型

TCP/IP 参考模型是计算机网络的始祖 ARPANET 和其后继的因特网使用的参考模型。ARPANET 是由美国国防部 DoD(U.S.Department of Defense)赞助的研究网络,它通过租用的电话线连结了数百所大学和政府部门。当无线网络和卫星出现以后,现有的协议在和它们相连的时候出现了问题,所以需要一种新的参考体系结构。这个体系结构在它的两个主要协议出现以后,被称为 TCP/IP 参考模型。模型结构如图 6-3 所示。

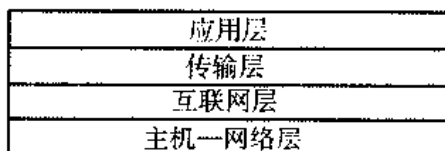


图 6-3 TCP/IP 的参考模型

由于国防部担心一些珍贵的主机、路由器和互联网关可能会突然崩溃,所以网络必须实现的另一目标是网络不受子网硬件损失的影响,已经建立的会话不会被取消,而且整个体系结构必须相当灵活。

TCP/IP 参考模型共有四层:应用层、传输层、互联网层和主机至网络层。

与 OSI 参考模型相比,TCP/IP 参考模型没有表示层和会话层。互联网层相当于 OSI 模型的网络层,主机至网络层相当于 OSI 模型中的物理层和数据链路层。

下面通过一个 WWW 浏览器来看看这四层的概况:

- 应用层:WWW、SMTP、DNS 和 FTP。
- 传输层:解释数据。
- 互联网层:定位 IP 地址与确定连接路径。
- 主机至网络层:与硬件驱动程序对话。

(1) 应用层

首先 WWW 浏览器必须调用 HTTP 协议,这个协议规定用什么样的命令来得到 WWW 文本,这种协议构成了 TCP/IP 的最高层——应用层。应用层包含所有的高层的

协议。这些高层协议有:虚拟终端协议 TELNET、文件传输协议 FTP、电子邮件传输协议 SMTP、域名系统服务 DNS 和超文本传输协议 HTTP 等。

- 虚拟终端协议 TELNET:允许一台机器上的用户登录到远程机器上并进行工作。
- 文件传输协议 FTP:提供有效地将数据从一台机器上移动到另一台机器上的方法。
- 电子邮件协议 SMTP:最初仅是一种文件传输,但是后来为它提出了专门的协议。
- 域名系统服务 DNS:用于把主机名映射到网络地址。
- 超文本传输协议 HTTP:用于在万维网(WWW)上获取主页等。

(2) 传输层

为了使用 HTTP 协议,浏览器要把命令发送到服务器上去,并且从服务器得出回答。但必须记住网络上传输的总是些字节,哪些字节是命令,哪些是回送数据,又有哪些是用于表示“就绪”、“传输中”或者“停止”的验证码。这些解释工作需要一串复杂的协议进行控制,这构成了 TCP/IP 分层结构的第三层——传输层。它的功能是使源端和目标主机上的对等实体可以进行会话。在这一层定义了两个端到端的协议。一个是传输控制协议 TCP,它是一个面向连接的协议,允许从一台机器发出的字节流无差错地发往另一台机器。它将输入的字节流分成报文段并传给互联网层。TCP 还要处理流量控制,以避免快速发送方向低速接收方发送过多的报文而使接收方无法处理。另一个协议是用户数据报协议 UDP,它是一个不可靠的、无连结的协议,用于不需要 TCP 排序和流量控制而是自行完成这些功能的应用程序中。

(3) 互联网层

所有上述的需求导致了基于无连结互联网络层的分组交换网络。这一层被称作互联网层,它是整个体系结构的关键部分。它的功能是使主机把分组发往任何网络并使分组独立地传向目标(可能经由不同的网络)。这些分组到达的顺序和发送的顺序可能不同,因此如果需要按顺序发送和接收时,高层必须对分组进行排序。

互联网层定义了正式的分组格式和协议,即 IP 协议。互联网层的功能就是把 IP 分组发送到应该去的地方。分组路由和避免阻塞是这里主要的设计问题。TCP/IP 互联网层和 OSI 网络层在功能上非常相似。

(4) 主机至网络层

该层处在互联网层下面, TCP/IP 参考模型没有真正描述这一部分,只是指出主机必须使用某种协议与网络相连。它只是完成最终的工作,即将地址、数据等变换成真正的电气信号,并且在网络上送出,让网络设备彼此对话。

2. IP 地址和子网

互联网用 IP 地址来标识主机地址,而数据传输则是通过把数据分成一系列小“包”来完成。TCP/IP 世界里的每一台主机都有唯一的 IP 地址,这个 IP 地址是一个 32 位无符号整数,不过通常使用点分十进制表示。例如,00010000110000011111111000000001 是二进制的 281148929,但是实际应用中把它按照 8 位一段的方法分成四段,第一段是 00010000,也就是 16,第二段是 11000000,十进制是 192,同样,剩下来的两段是 254 和 1,



因此这个地址是 16.192.254.1。

这听起来很简单,但是在实现中有些复杂,主要问题是 TCP/IP 必须兼顾许多困难的问题,其中之一是网络联接。全世界有太多的网络主机,通过一个主机号很难知道它在哪里。最大的难题是,也许主机 13.2.0.5 在纽约,而主机 64.0.7.6 却在北京。如果搜索一台主机地址就要查阅全世界的主机列表,那么互联网将立即崩溃。

解决这一问题的第一个要点是网络分类。分类把世界上的 IP 地址分段,各段之间独立管理,通常每一段用同样的方式连接到一起,本段的机器之间可以直接互相访问,这样的一个个段称为一个系列网络地址。

为了管理方便,分段用一种特殊的方式。例如,对一些公司(如 IBM),一个上万台机器的网络很正常,而小公司也许只有十几台机器,TCP/IP 实现中用 A、B、C 类地址来处理这个问题。

A 类地址用于超过 65534 个主机的网络,例如,一个前缀为 18 的网络使用 18.0.0.1~18.255.255.254 的网络地址,这些地址之间是直通的,主机之间可以彼此访问。为了说明这一点,TCP/IP 使用网络掩码和网络地址的概念。

在上面的例子中,网络地址是 18.0.0.0,这表示网络的包含地址段是 18.0.0.1~18.255.255.254,即地址部分除去前缀后,余下的部分由全零变成全 1,不过全 1 的地址将保留为广播使用。与网络地址相应,对在此网络中的主机,TCP/IP 使用网络掩码。在上面的例子中,掩码是 255.0.0.0,其含义是:假设 18.0.0.3 想与 18.110.75 对话,那么,将这两个地址相互异或(XOR),再与掩码取“与”(&),结果是零,说明这两个地址可以直接对话。相反,如果要与 19.1.1.1 对话,那么运算之后不为零,说明必须使用间接方式才能到达。

可以用直接方式理解掩码,掩码中的“1”用来描述网络地址(前缀)，“0”用来描述子网的主机,如 255.0.0.0 意味着将主机地址的前 8 位解释为网络号,而后 24 位解释成网内的主机地址。也就是说,与某个 A 类主机地址前 8 位相同的主机地址被认为是在同一子网内。

A、B、C 类用下面的方式规定:

- A 类地址使用 255.0.0.0 的掩码。为了管理方便,规定 A 类地址总是使用头一位为 0 的地址值,这意味着 A 类地址是从 1.0.0.0 到 126.0.0.0。另外还有一个特殊的 A 类地址 127.0.0.0,它用来表示“本机”,即自身。
- B 类地址使用 255.255.0.0 的掩码。B 类网从 128~0.0.1 到 191.255.0.0。
- C 类地址的掩码是 255.255.255.0。网络地址从 192.0.0.0 到 233.255.255.0。

首字节在 224 以上的地址用于实验和开发,部分用于组播,一般用的不多,可不必太关心。

地址 255 为特殊的含义,它用于广播,即“向本网上所有人通话”。这个概念对任何人来说,都是很容易理解的。使用广播有两种方法,并且几乎是等价的,即:

- 使用全 1 的地址:即使用 255.255.255.255。
- 使用本址广播,即用网络号加上全 1。

A 类网 180.0.0.0 可以使用 18.255.255.255 广播,B 类网 190.4.0.0 的广播地址则是 190.4.255.255。无论哪一种,对于广播地址的访问都将引发将数据发送给本网络上

的所有机器。

计算网络掩码和标志子网地址是人们经常需要做的工作。第一种计算是根据某个确定的主机地址和它的网络掩码求出所有可以和它直接通信(在同一子网之内)的主机地址。例如,主机地址是 202.112.50.3,掩码是 255.255.255.0,与它可以直接通话的主机地址可以这样计算:掩码是 255.255.255.0,因此 32 位网络地址的前 24 位被解释为网络地址。凡是与 202.112.50.3 的前 24 位内容相同的主机地址就和它处在同一子网之内,后 8 位是子网里面的机器地址,它可以全 0 变到全 1,所以所有和这个主机在同一子网之内的主机地址是 202.112.50.0~202.112.50.255。

网络地址/掩码一般用斜线分开,如网络地址是 202.199.248.0,掩码 255.255.255.0 在 UNIX 中一般写成 202.199.248.0/255.255.255.0。但也有另外一种写法,就是用掩码中 1 的个数来代替掩码的实际形式。例如,255.255.255.0 的前 24 位是 1,其他位是 0,因此可以写成 202.199.248.0/24。同样,下面的两栏地址形式是彼此等效的:

202.199.248.0/255.255.255.0 202.199.248.0/24

122.24.0.0/255.255.255.0 122.24.0.0/16

13.4.5.7/255.0.0.0 13.4.5.7/8

在前面一直设置掩码的 0/1 分界在字节边界处,但是这其实并不是必须的,完全可以有其他形式的掩码。例如,240 等于二进制的 11110000,因此一个 255.255.255.240 的掩码意味着前面 28 位是网络地址,而后面 4 位是子网内的 IP。同样,网络地址也可以不由 0 结尾。举个例子来说,202.112.50.16/255.255.255.240 是什么意思? 202.112.50.16 是二进制的 11001010011100000011001000010000,套上一个有 28 位为 1 的网络掩码,意味着最后四位由全 0 变成全 1,就是最小是 11001010011100000011001000010000,最大是 110010100 1110000001100100001111,这两个数是 202.112.50.16 和 202.112.50.31,所以这个网络地址代表 202.112.50.16 到 202.112.50.31 的所有主机。这个网络地址也可以写成 202.112.50.16/26。

3. Linux 中的 TCP/IP 网络层次结构

图 6-4 描述了 Linux 对 IP 协议族的实现机制,如同网络协议自身一样, Linux 也是通过视其为一组相连的软件层来实现的。其中 BSD 套接字由通用的套接字管理软件所支持,该软件是 INET 套接字层,用来管理基于 IP 的 TCP 与 UDP 的端到端互联。如前所述, TCP 是一个面向连接协议,而 UDP 则是一个非面向连接协议。当一个 UDP 报文发送出去后, Linux 并不知道也不去关心它是否成功地到达了目的主机。对于 TCP 传输,传输节点间先要建立连接,然后通过该连接传输已排好序的报文,以保证传输的正确性。IP 层中代码用以实现网际协议,这些代码将 IP 头增加到传输数据中,同时也把收到的 IP 报文正确地转送到 TCP 层或 UDP 层。在 IP 层之下,是支持所有 Linux 网络应用的网络设备层,例如点到点协议(PPP)和以太网层。网络设备并非总代表物理设备,其中有一些(例如回送设备)则是纯粹的软件设备。网络设备与标准的 Linux 设备不同,它们不是通过 mknod 命令创建的,必须是底层软件找到并进行了初始化之后,这些设备才被

创建并可用。因此只有当启动了正确设置了以太网设备驱动程序的内核后,才会有/dev/eth0 文件。ARP 协议位于 IP 层和支持地址解析的协议层之间。

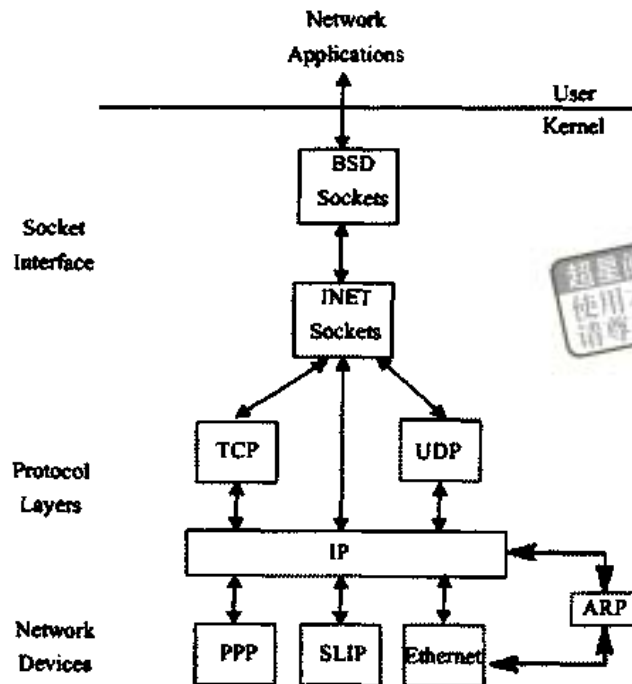


图 6-4 Linux 网络层次结构

4. BSD 套接字接口

它是一个通用接口,它不仅支持不同的网络结构,同时也是一个内部进程间通信机制。一个套接字描述了一个连结的一个端点,因此两个互联的进程都要有一个描述它们之间连接的套接字。可以把套接字看作是一种特殊的管道,只是这种管道对于所包含的数据量没有限制。Linux 支持套接字地址族中的多个,如下所列:

UNIX	UNIX 域套接字
INET	使用 TCP/IP 的因特网地址族
AX25	业余无线 X25
IPX	IPX
APPLETALK	APPLETALK
X25	X25

Linux 支持多种套接字类型。套接字类型是指创建套接字的应用程序所希望的通信服务类型。同一协议族可能提供多种服务类型,比如 TCP/IP 协议族提供的虚电路与数据报就是两种不同的通信服务类型。Linux BSD 支持如下几种套接字类型:

- Stream: 即流式套接字,它提供可靠的面向连接传输的数据流,保证数据传输过程中无丢失、无损坏和无冗余。INET 地址族中的 TCP 协议支持该套接字。
- Datagram: 即数据报套接字。它提供数据的双向传输,但不保证消息报文的准确到达,即使消息能够到达,也无法保证其顺序正确,并可能有冗余或损坏。INET 地址族中的 UDP 协议支持该套接字。

- Raw:它是低于传输层的低级协议或物理网络提供的套接字类型,比如通过分析为以太网设备所创建的 RAW 套接字,可看到裸 IP 数据流。
- Reliable Delivered Messages:它类似于数据报套接字,但能保证数据的正确到达。
- Sequenced Packets:类似于 Stream 套接字,但它的报文大小是可变的。
- Packet:这是 Linux 对标准 BSD 套接字类型的扩展,它允许应用程序在设备层直接访问报文数据。

6.3.2 Linux 环境下的 socket 编程



1.socket 定义

前面已经讲过一些关于套接字(socket)的基本概念。网络的 socket 数据传输是一种特殊的 I/O,socket 也是一种文件描述符。socket 也具有一个类似于打开文件的函数调用 socket(),该函数返回一个整型的 socket 描述符,随后的连接建立、数据传输等操作都是通过该 socket 实现的。常用的 socket 类型有两种:流式 socket,SOCK_STREAM 和数据报式 socket,SOCK_DGRAM。

流式 socket 是一种针对面向连接的 socket,对应于有连接的 TCP 服务应用。数据报式 socket 是一种无连接的 socket,对应于无连接的 UDP 服务应用。

2.socket 编程相关数据类型定义

在计算机中,数据存储有两种字节优先顺序:高位字节优先和低位字节优先。Internet 上数据以高位字节优先顺序在网络上传输,所以对于在内部以低位字节优先方式存储数据的机器,在 Internet 上传输数据时就需要进行转换。

(1) struct sockaddr

要讨论的第一个结构类型是:struct sockaddr,该类型是用来保存 socket 信息的。

```
struct sockaddr {
```

```
    unsigned short sa_family; /* 地址族, AF_ xxx */
```

```
    char sa_data[14]; /* 14 字节的协议地址 */
};
```

sa_family 一般为 AF_INET。

sa_data 则包含该 socket 的 IP 地址和端口号。

(2) struct sockaddr_in

该结构的定义为:

```
struct sockaddr_in {
```

```
    short int sin_family; /* 地址族 */
```

```
    unsigned short int sin_port; /* 端口号 */
```

```

struct in_addr sin_addr; /* IP 地址 */
unsigned char sin_zero[8]; /* 填充 0 以保持与 struct sockaddr 同样大小 */
};

```

这个结构使用更为方便。sin_zero(它用来将 sockaddr_in 结构填充到与 struct sockaddr 同样的长度)应该用 bzero()或 memset()函数将其置为零。指向 sockaddr_in 的指针和指向 sockaddr 的指针可以相互转换。这意味着如果一个函数所需参数类型是 sockaddr 时,就可以在函数调用时将一个指向 sockaddr_in 的指针转换为指向 sockaddr 的指针;或者相反。

下面讨论几个字节顺序转换函数:

- htons():htons 表示“Host to Network Short”,主机地址字节顺序转向网络字节顺序(对短型数据操作)。
- htonl():htonl 表示“Host to Network Long”,主机地址字节顺序转向网络字节顺序(对长型数据操作)。
- ntohs():ntohs 表示“Network to Host Short”,网络字节顺序转向主机地址顺序(对短型数据操作)。
- ntohl():ntohl 表示“Network to Host Long”,网络字节顺序转向主机地址顺序(对长型数据操作)。

在这里,h 表示“host”;n 表示“network”;s 表示“short”,短整型数据;l 表示“long”,长整型数据。

3. 打开 socket 描述符、建立绑定并建立连接

socket 函数原型为:

```
int socket(int domain,int type,int protocol);
```

domain 参数指定 socket 的类型:SOCK_STREAM 或 SOCK_DGRAM。

protocol 通常赋值“0”。socket()调用返回一个整型 socket 描述符,可以在后面的调用使用它。一旦通过 socket 调用返回一个 socket 描述符,应该将该 socket 与本机上的一个端口相关联,即“绑定”(往往在设计服务器端程序时,需要调用该函数。随后就可以在该端口监听服务请求;而客户端一般无须调用该函数)。

bind 函数原型为:

```
int bind(int sockfd,struct sockaddr * my_addr, int addrlen);
```

sockfd 是一个 socket 描述符,my_addr 是一个指向包含有本机 IP 地址及端口号等信息的 sockaddr 类型的指针,addrlen 常被设置为 sizeof(struct sockaddr)。

最后,对于 bind 函数要说明的一点是,可以用下面的赋值实现自动获得本机 IP 地址和随机获取一个没有被占用的端口号。

```
my_addr.sin_port = 0; /* 系统随机选择一个未被使用的端口号 */
```

```
my_addr.sin_addr.s_addr = INADDR_ANY; /* 填入本机 IP 地址 */
```

通过将 my_addr.sin_port 置为 0,函数会自动选择一个未占用的端口使用。同样,通过将 my_addr.sin_addr.s_addr 置为 INADDR_ANY,系统会自动填入本机 IP 地址。

bind()函数在成功被调用时返回 0;遇到错误时返回“-1”并将 errno 置为相应的错误号。当调用函数时,一般不要将端口号置为小于 1024 的值,因为 1~1024 是保留端口号,可以使用大于 1024 的任何一个没有被占用的端口号。

connect()函数用来与远端服务器建立一个 TCP 连接,其函数原型为:

```
int connect(int sockfd, struct sockaddr * serv_addr, int addrlen);
```

sockfd 是目的服务器的 socket 描述符;serv_addr 是包含目的机 IP 地址和端口号的指针。遇到错误时返回 -1,并且 errno 中包含相应的错误码。进行客户端程序设计无须调用 bind(),因为这种情况下只需知道目的机器的 IP 地址,而客户通过哪个端口与服务器建立连接并不需要关心,内核会自动选择一个未被占用的端口供客户端使用。

listen()函数:用来监听是否有服务请求。

在服务器端程序中,当 socket 与某一端口捆绑以后,就需要监听该端口,以便对到达的服务请求加以处理。

```
int listen(int sockfd, int backlog);
```

sockfd 是 socket 系统调用返回的 socket 描述符;backlog 指定在请求队列中允许的最大请求数,进入的连接请求将在队列中等待 accept()函数接受它们(关于 accept()函数,请见下文)。backlog 对队列中等待服务请求的数目进行了限制,大多数系统缺省值为 20。当 listen 遇到错误时返回 -1,错误码 errno 被置为相应的值。

一般服务器端程序通常按下列顺序进行函数调用:

```
socket()→bind()→listen(); /* 接下来就是 accept()函数 */
accept():连接端口的服务请求。
```

当某个客户端试图与服务器监听的端口连接时,该连接请求将排队等待服务器调用 accept()函数接受它。程序调用 accept()函数为该请求建立一个连接,accept()函数就返回一个新的 socket 描述符,供这个新连接使用。而服务器可以继续以前的那个 socket 上监听,同时可以在新的 socket 描述符上进行数据 send()(发送)和 recv()(接收)操作。

```
int accept(int sockfd, void * addr, int * addrlen);
```

sockfd 是被监听的 socket 描述符。addr 通常是一个指向 sockaddr_in 变量的指针,该变量用来存放提出连接请求服务的主机信息(某台主机从某个端口发出该请求)。addrlen 一般是一个整型指针变量,它指向的整型数值为 sizeof(struct sockaddr_in)。发生错误时返回一个 -1,并且设置相应的 errno 值。

send()和 recv():数据的发送和接收。

这两个函数使用面向连接的 socket 来进行数据传输。

send()函数原型为:

```
int send(int sockfd, const void * msg, int len, int flags);
```

sockfd 是用来传输数据的 socket 描述符,msg 是一个指向要发送数据的指针。len 是以字节为单位的数据长度,flags 一般情况下置为 0(关于该参数的用法可参照 man 手册)。

send()函数返回实际上发送出的字节数,可能会少于希望发送的数据,所以需对 send()的返回值进行测量。当 send()返回值与 len 不匹配时,应该对这种情况进行处理。

recv()函数原型为:

```
int recv(int sockfd, void * buf, int len, unsigned int flags);
```

sockfd 是接受数据的 socket 描述符; buf 是存放接收数据的缓冲区; len 是缓冲的长度。flags 也被置为 0。函数 recv() 返回实际上接收的字节数, 或当出现错误时, 返回 -1 并置相应的 errno 值。

sendto() 和 recvfrom(): 利用数据报方式进行数据传输。

在无连接的数据报 socket 方式下, 由于本地 socket 并没有与远端机器建立连接, 所以在发送数据时应指明目的地址。sendto() 函数原型为:

```
int sendto(int sockfd, const void * msg, int len, unsigned int flags,  
const struct sockaddr * to, int tolen);
```

该函数比 send() 函数多了两个参数, to 表示目的机的 IP 地址和端口号信息, 而 tolen 常常被赋值为 sizeof (struct sockaddr)。sendto 函数也返回实际发送的数据字节长度或在出现发送错误时返回 -1。

recvfrom() 函数原型为:

```
int recvfrom(int sockfd, void * buf, int len, unsigned int flags, struct sockaddr * from,  
int * fromlen);
```

from 是一个 struct sockaddr 类型的指针变量, 该变量保存源机的 IP 地址及端口号。fromlen 常置为 sizeof (struct sockaddr)。当 recvfrom() 返回时, fromlen 包含实际存入 from 中的数据字节数。

recvfrom() 函数返回接收到的字节数, 当出现错误时则返回 -1, 并设置相应的 errno 值。当数据报 socket 调用 connect() 函数时, 可以利用 send() 和 recv() 进行数据传输, 但该 socket 仍然是数据报 socket, 并且利用传输层的 UDP 服务。在发送或接收数据报文时, 内核会自动为之加上目的地址和源地址信息。

close() 和 shutdown(): 结束数据传输。

当所有的数据操作结束后, 可以调用 close() 函数来释放该 socket, 从而停止在该 socket 上的任何数据操作, 如:

```
close(sockfd);
```

也可以调用 shutdown() 函数来关闭该 socket。该函数允许用户只停止在某个方向上的数据传输, 而另一个方向上的数据传输继续进行。可以关闭某 socket 的写操作而允许继续在该 socket 上接受数据, 直至读入所有数据, 如:

```
int shutdown(int sockfd, int how);
```

sockfd 的含义是显而易见的, 而参数 how 可以设为下列值:

0——不允许继续接收数据。

1——不允许继续发送数据。

2——不允许继续发送和接收数据, 均为允许则调用。

close() 和 shutdown 在操作成功时返回 0, 在出现错误时返回 -1 (并置相应 errno)。

DNS: 域名服务相关函数。

由于 IP 地址难以记忆和读写, 为了读写记忆方便, 所以人们常用域名来表示主机, 这就需要进行域名和 IP 地址的转换。函数 gethostbyname() 就是完成这种转换的, 函数原

型为:

```

struct hostent * gethostbyname(const char * name);
函数返回一种名为 hostent 的结构类型,它的定义如下:
struct hostent {
char * h_name;          /* 主机的官方域名 */
char ** h_aliases;     /* 一个以 NULL 结尾的主机别名数组 */
int h_addrtype;       /* 返回的地址类型,在 Internet 环境下为 AF_INET */
int h_length;         /* 地址的字节长度 */
char ** h_addr_list;  /* 一个以 0 结尾的数组,包含该主机的所有地址 */
};
#define h_addr h_addr_list[0] /* 在 h_addr_list 中的第一个地址 */

```



当 gethostname()调用成功时,返回指向 struct hostent 的指针,当调用失败时返回 -1。当调用 gethostbyname 时,不能使用 perror()函数来输出错误信息,而应使用 perror()函数来输出。

4. 面向连接的客户/服务器代码实例

一个建立分布式应用时最常用的范例便是客户机/服务器模型。在这种方案中客户应用程序向服务器程序请求服务,这种方式隐含了在建立客户机/服务器间通信的非对称性。客户机/服务器模型工作时要求有一套为客户机和服务器所共识的惯例,以保证服务能够被提供(或被接收)。这一套惯例包含一套协议,它必须在通信的两端都被实现。在非对称协议中,一方不可改变的认为是主机,另一方则是从机。当服务被提供时必然存在“客户进程”和“服务进程”。一个服务器通常在一个众所周知的地址监听对服务的请求。也就是说,服务器一直处于休眠状态,直到一个客户对这个服务的地址提出连接请求。在这个时刻,服务程序被唤醒并且为客户提供服务,对客户请求做出了适当的反应。

下面来看上一章最后留下来的那个例子。这个服务器程序通过一个连接向客户发送字符串“Hello, world! \n”。只要在服务器上运行该服务器软件,在客户端运行客户软件,客户端就会收到该字符串。在开发板上已经将其实现了,现在来详细学习这个程序。

例 6-1 一个简单的 C/S 模式通信样例演示

(1) 客户端

```

* Copyright (C) 2001 <syzou@263.net.cn, liyan1011@sina.com>
* 该程序是自由软件,您可以依据自由基金联合会
* (FSF)的 GNU 通用公共许可证的条款
* (GPL2.0 版本,或更高的版本)
* 对该软件进行重新发布或修改。*/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

```

```

#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>

#define MAXDATASIZE 100 /* MAXDATASIZE 定义了客户机一次可以接受
                        * 的字符数目 */

#define PORT 2000 /* 定义连接到服务器的端口号。 */

int main(int argc, char * argv[])
{
    int sockfd, numbytes; /* sockfd 是套接字描述符 */
    char buf[MAXDATASIZE];
    struct hostent he; /* he 为 hostent 类型的数据结构,用来存放所获取的主机信
                       * 息 */
    struct sockaddr_in their_addr; /* 存放主机信息 */
    if(argc != 2)
    {
        fprintf(stderr, "usage: client hostname \n");
        exit(1);
    }
    if((he = gethostbyname(argv[1])) == NULL) /* 调用 gethostbyname 函数,获取主机信
                                             * 息 */
    {
        perror("gethostbyname");
        exit(1);
    }
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) /* 建立流式套接
                                                         * 字描述符 */
    {
        perror("socket");
        exit(1);
    }
    /* 给定远端主机信息 */
    their_addr.sin_family = AF_INET;
    their_addr.sin_port = htons(PORT);
    their_addr.sin_addr = *((struct in_addr *)he->h_addr);
    bzero(&(their_addr.sin_zero), 8);

```



```

/* 建立连接 */
if(connect(sockfd,(struct sockaddr *) &their_addr,sizeof(struct sockaddr)) == -1)
    {perror("connect");          /* 错误校验 */
      exit(1);
    }

if((numbytes = recv(sockfd,buf,MAXDATASIZE,0)) == -1) /* 接收主机传送过
                                                    * 来的字符串 */

    {perror("recv");
      exit(1);
    }
buf[numbytes] = '\0';
printf("Received: %s",buf);          /* 打印信息到标准输出 */
close(sockfd);
return 0;
}

```

以上是一个简单的客户端的信息接收程序。该程序实现的功能是通过使用函数 connect(), 连接到服务器的 2000 端口, 然后获取服务器发送的字符串, 并打印显示。首先通过服务器域名获得其 IP 地址, 然后创建一个 socket, 调用 connect 函数与服务器建立连接, 连接成功之后接收从服务器发送过来的数据, 最后关闭 socket, 结束程序。

无连接的客户/服务器程序在原理上和面向连接的客户/服务器程序是一样的。两者的区别在于无连接的客户/服务器中的客户一般不需要建立连接, 而且在发送接收数据时, 要指定远端机的地址。

(2) 服务器端

/* servise.c: 一个简单的服务器端程序样例演示。

* Copyright (C) 2001 <syzhou@263.net.cn>

* 该程序是自由软件, 您可以依据自由基金联合会

* (FSF) 的 GNU 通用公共许可证的条款

* (GPL2.0 版本, 或更高的版本)

* 对该软件进行重新发布或修改。 */

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/wait.h>

```

```
#include <sys/socket.h>
#define MYPORT 2000      /* 服务器的监听端口 */
#define BACKLOG 10      /* BACKLOG 指定在请求队列中允许的最大请求
                          * 数, 进入的连接请求将在队列中等待 accept() 函数
                          * 接受它们 */

main()
{
    int sock_fd, new_fd;      /* sock_fd 为监听用描述符 */
    struct sockaddr_in my_addr;
    struct sockaddr_in their_addr;
    int sin_size;

    if((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1) /* 错误检测 */
    {
        perror("socket");
        exit(1);
    }

    my_addr.sin_family = AF_INET;
    my_addr.sin_port = htons(MYPORT);
    my_addr.sin_addr.s_addr = INADDR_ANY;
    bzero(&(my_addr.sin_zero), 8);

    /* 绑定 */
    if(bind(sock_fd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1)
    {
        perror("bindB");
        exit(1);
    }

    if(listen(sock_fd, BACKLOG) == -1) /* 监听端口是否有请求 */
    {
        perror("listen");
        exit(1);
    }

    while(1)
    {
        sin_size = sizeof(struct sockaddr_in);
        if((new_fd = accept(sock_fd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
        {
            perror("accept");
        }
    }
}
```

```
        continue;
    }
    printf("server:got connection from %s\n",inet_ntoa(their_addr.sin_addr));

    if(! fork()) /* 子进程代码段。它创建一个子进程,来处理与刚刚建立的
                * 套接字的通信 */
    {
        if(send(new_fd,"Hello,World! \n",14,0) == -1) /* 发送字符串 */
        {
            perror("send");
            close(new_fd);
            exit(1);
        }
        close(new_fd); /* 父进程不再需要该 socket */
    }
}

while(waitpid(-1,NULL,WNOHANG)>0) /* 等待子进程结束,清除子进程所占用资源
                                   */
;}
```

服务器端程序的功能是监听其 2000 端口。如果发现有客户机的请求到来,就产生一个子进程(调用 `fork()`)与客户机进行通信。显然,同时可以有多个客户机对服务器提出请求,而服务器可以生成多个子进程来应答多个客户机的请求。

服务器首先创建一个 socket,然后将该 socket 与本地地址/端口号捆绑,成功之后就在相应的 socket 上监听。当 `accept` 捕捉到一个连接服务请求时,就生成一个新的 socket,并通过这个新的 socket 向客户端发送字符串“Hello, world! \n”,然后关闭该 socket。

`fork()` 函数用来生成一个子进程,以处理数据传输。`fork()` 函数是一个单调用双返回的函数。如果调用成功,那么在父进程中返回子进程的进程标识号,在子进程则返回零。如果调用不成功,则返回 -1。包含 `fork` 函数的 `if` 语句是子进程代码部分,它与 `if` 语句后面的父进程代码部分是并发执行的。

关于本程序的流程图如图 6-5 所示。

5. 关于阻塞的概念和 `select()` 函数

如上例,当服务器运行到 `accept` 语句时,如果没有客户连接服务请求到来,那么会发生什么情况? 这时服务器就会停止在 `accept` 语句上等待连接服务请求的到来。同样,当程序运行到接收数据语句时,如果没有数据可以读取,则程序同样会停止在接收语句上。这种情况称为 `blocking`,即“阻塞”。如果希望服务器仅仅注意检查是否有客户在等待连接,有就接受连接,否则就继续做其他事情,则可以通过将 socket 设置为非阻塞方式来实现非阻塞 socket。没有客户等待就使 `accept` 调用立即返回。

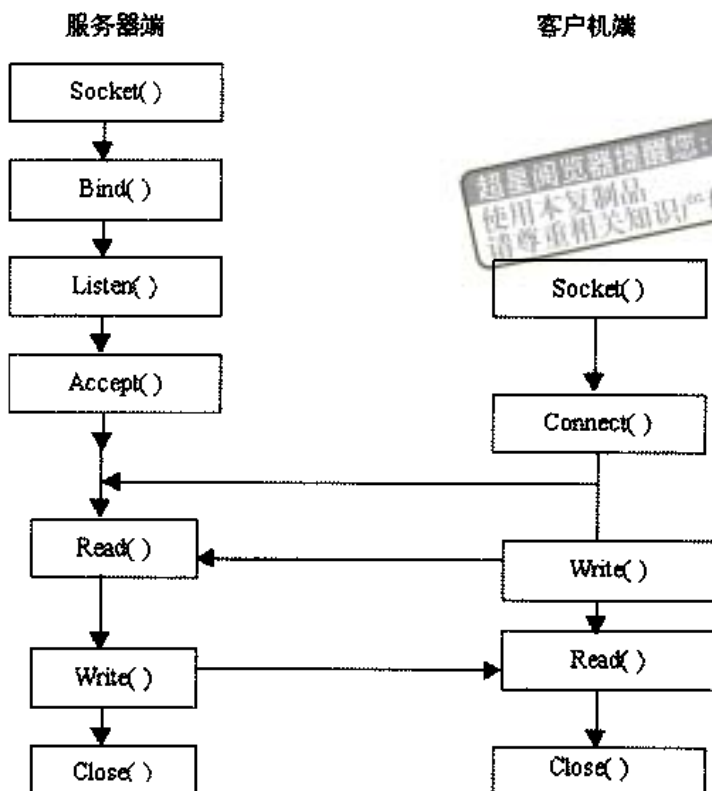


图 6-5 简单的服务器/客户机通信流程图

例如,可以使用如下的方式:

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```
...;
```

```
sockfd = socket(AF_INET,SOCK_STREAM,0);
```

```
fcntl(sockfd,F_SETFL,O_NONBLOCK);.....
```

fcntl()函数用来控制文件描述符,对应于上个命令的原型为:

```
int fcntl(int fd, int cmd, long arg);
```

cmd 为命令参数,它决定了 fcntl()函数将要进行的工作。F_SETFL 参数的含义是将描述符的标志设置为由 arg 参数所定义的属性。在此,可以设置 O_APPEND、O_NONBLOCK 和 O_ASYNC 这三种属性。

通过设置 socket 为非阻塞方式,可以实现“轮询”若干 socket。当企图从一个没有数据等待处理的非阻塞 socket 读入数据时,函数将立即返回,返回值置为 -1,并且 errno 置

件描述符已准备好的信息。这样就实现了为进程选出随机的变化,而不必由进程本身对输入进行测试而浪费 CPU 开销。select 函数原型为:

```
int select(int numfds,fd_set * readfds,fd_set * writefds,fd_set * exceptfds,struct
timeval * timeout);
```

其中 readfds、writefds、exceptfds 分别是被 select() 监视的读、写和异常处理的文件描述符集合。如果希望确定是否可以从标准输入和某个 socket 描述符读取数据,只需要将标准输入的文件描述符 0 和相应的 sockdtd 加入到 readfds 集合中。当 select 返回时,readfds 将被修改,指示某个文件描述符已经准备被读取,可以通过 FD_ISSET() 来测试。为了实现 fd_set 中对应的文件描述符的设置、复位和测试,它提供了一组宏:

- FD_ZERO(fd_set * set):清除一个文件描述符集。
- FD_SET(int fd,fd_set * set):将一个文件描述符加入文件描述符集中。
- FD_CLR(int fd,fd_set * set):将一个文件描述符从文件描述符集中清除。
- FD_ISSET(int fd,fd_set * set):判断文件描述符是否被置位。

timeout 参数是一个指向 struct timeval 类型的指针,它可以使 select() 在等待长度为 timeout 的一段时间后,如果没有文件描述符准备好,即返回。struct timeval 数据结构为:

```
struct timeval {
int tv_sec; /* seconds */
int tv_usec; /* microseconds */
};
```

使用 select() 函数,就可以有效地节省系统资源,提高系统效率。

6.3.3 应用实例:网口通信

在开发平台上,也可以开发出有一定功能的通信程序。例如,有这么一个样例程序:在开发平台上运行一个发送程序,将内存中一个文件(如图片)通过 10BASE-T 的网口传送到 PC 上,PC 同时运行接收程序将其接收并显示。有了本小节的知识,不难编写出该发送程序。接收方的应用程序是在 Windows 环境下运行的,需要一定的 Windows 环境下的网络编程知识。在本书所附光盘中有接收方应用程序的源代码,可自行研习。

来看发送方的程序,见例 6-2。

例 6-2 网口通信样例(发送方)

```
/* This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */
#include <sys/socket.h>
#include <sys/types.h>
```

```
#include <netinet/in.h>
```

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include <stdlib.h>
```

```
#include <string.h>
```

```
int aiptoip(char * pszip, unsigned int * piip)
```

```
/* 将命令行输入的字符串 IP 转换成 connect 函数可识别的整数 uiip */
```

```
{
```

```
char psziphere[17], * psztmp1, * psztmp2, * pchar; /* 定义指针 */
```

```
...
```




```

FILE * fp;

sockfd = socket(AF_INET,SOCK_STREAM,0);/* 创建 SOCKET */
bzero(&servaddr,sizeof(struct sockaddr_in));
servaddr.sin_family = AF_INET;
servaddr.sin_port = 8888;          /* 填入端口号 */
if(! aiptoi(argv[1],&uiip)||argc<=1)/* 如果转换不成功或参数输入个数<=1,
                                     * 显示错误 */
|
printf("the ip is not correct or lose the ip input! \n");
return 0;
|

servaddr.sin_addr.s_addr = uiip; /* 指定连接的对端 IP */
if(connect(sockfd,(struct sockaddr *)&servaddr,sizeof(struct sockaddr)))/ * 连接对端接
                                     * 收代码 */
|
printf("Can't connect to the server! \n");
return 0;
|
if((fp = fopen("test.bmp","r")) == NULL) /* 打开位图文件 */
|
printf("Can't find the test.bmp file");
return 0;
|
sprintf(head,"hhcn"); /* 在 head BUFFER 区头部填入 hhcn */
while(nsize == 1024) /* 每次从文件中读取 1024 个字节发送出去,若读出少于 1024
                       * 个字节则认为是到达了文件结尾 */
|
bzero(szsendbuf,1024); /* 清空缓冲区 */
nsize = * phead = fread(szsendbuf,1,1024,fp);/* 从文件中读取并发送至缓冲区中,填写
                                               * 通信头的数据长度 */

write(sockfd,head,8); /* 发送协议头 */
nsize = write(sockfd,szsendbuf,nsize); /* 发送数据 */

```

```

allsize + = nsize;                /* 统计发送字节数 */

if(! (allsize&0x3fff)) /* 若发送的总字节数大于 16K,则打印一次当前发送信息 */
printf("now size : %d this time %d times : %d \n",allsize,nsize,allsize/1024);
if(nsize <= 0)                /* 若发送完毕,退出 */
{
printf("Can't send data! \n");
return 0;
}

fclose(fp);                    /* 关闭位图文件 */
return 0;
}

```

按照在第5章中所讲述的方法,通过设置文件/uclinux/appsrc/init/init.c中相应的代码,使在开发板启动之时就运行程序。当然也可以在 Windows 98 下运行 telnet 登录到开发板上,在 uClinux 的 shell 提示符下,键入:

ethernet + 目标 PC 的 IP 地址(如,192.168.2.1)

并在接收方运行接收程序,就会显示出所传送过来的文件。

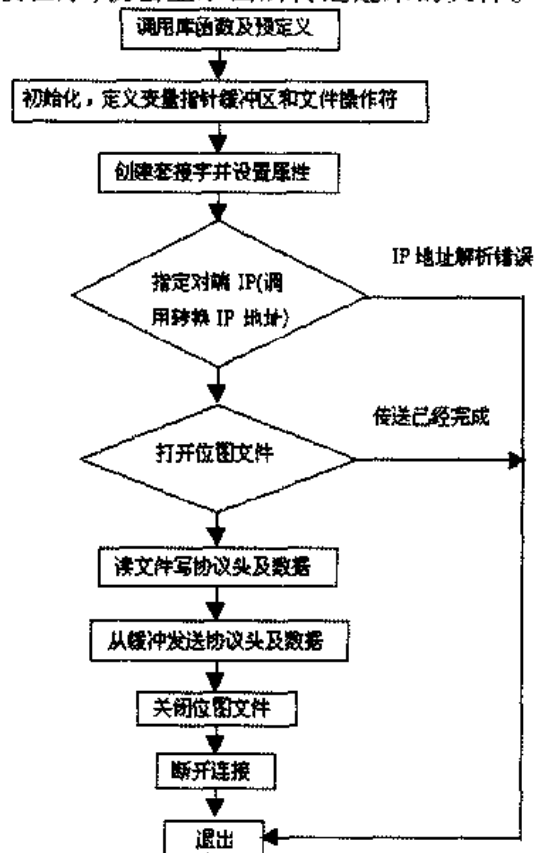
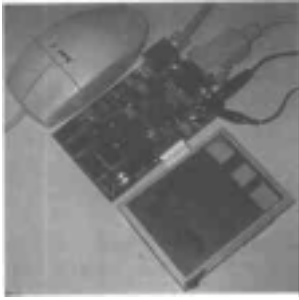


图 6-6 网口通信程序流程图

程。
址。



6-6 所示。从流程图中可以清楚地看到该应用程序的执行过
 送数据的缓冲区 `szsendbuf[1024]`，而 `uiip` 是经过转换后的 IP 地
 流式文件的操作函数，如 `fopen()`、`fread()` 等，参照 `man` 手册使用。
 据是一个位图文件 `test.bmp`，接收的结果如图 6-7 所示。



图 6-7 网口通信接收结果

注意：本样例采用了华恒公司提供的 BMP 文件 `test.bmp`，该文件大约有 600 多 K，传输时间要 5 到 6 分钟，需耐心等待。

6.4 连接上 Web

WWW(World Wide Web)的飞速发展和广泛应用得益于其提供的大量服务，这些服务为人们的信息交流带来了极大的便利。环球信息网是一个基于超文本方式的信息查询方式。它提供了一个友好的界面，大大方便了人们对信息的浏览。而嵌入式系统接入互联网也是大势所趋。在学习完编写网络通信程序后，再来看看如何将嵌入式 Linux 系统连接上 Web。首先简要介绍 HTTP 协议，然后以开发平台为例，学习一个 Webserver 的编写。

应该承认，这个 Webserver 的功能是非常弱小的。在此引用它只是为了阐述知识。如果有兴趣，在学习了它的实现机理后，可以在它的基础上给它增添更多的功能。

6.4.1 HTTP 协议

Internet 的基本协议是 TCP/IP 协议，已经在前面介绍过了。目前广泛采用的 FTP、Archie、Gopher 等是建立在 TCP/IP 协议之上的应用层协议，不同的协议对应不同的应用。

WWW 服务器使用的主要协议是 HTTP 协议,即超文本传输协议。由于 HTTP 协议支持的服务不限于 WWW,还可以是其他服务,因而 HTTP 协议允许用户在统一的界面下,采用不同的协议访问不同的服务,如 FTP、Archie、SMTP、NNTP 等。另外,HTTP 协议还可用于名字服务器和分布式对象管理。HTTP 为应用层网络协议,是一个可理解 URL 地址格式的小型化的、适合超文本的、多媒体环境的通信协议。使用 MIME(多用途互联网邮件扩充)处理数据,使它能适应多媒体技术。

1. HTTP 协议简介

HTTP 是一个属于应用层的面向对象的协议,由于其简捷、快速的方式,适用于分布式超媒体信息系统。它于 1990 年提出,经过十多年的使用与发展,得到不断的完善和扩展。

HTTP 协议的主要特点可概括如下:

- 支持客户/服务器模式。
- 简单快速:客户向服务器请求服务时,只需传送请求方法和路径。常用的请求方法有 GET、HEAD 和 POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单,HTTP 服务器的程序规模小,因而通信速度很快。
- 灵活:HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。
- 无连接:无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求,并收到客户的应答后,即断开连接。采用这种方式可以节省传输时间。
- 无状态:HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息,则必须重传,这样可能导致每次连接传送的数据量增大。在服务器不需要先前信息时,它的应答就较快。

2. HTTP 协议的几个重要概念

- 连接(Connection):一个传输层的实际环流,它建立在两个相互通信的应用程序之间。
- 消息(Message):HTTP 通信的基本单位,包括一个结构化的八元组序列并通过连接传输。
- 请求(Request):一个从客户端到服务器的请求信息包括应用于资源的方法、资源的标识符和协议的版本号。



“超文本”。

- 服务器(Server):一个接受连接并对请求返回信息的应用程序。服务器端的每一个 Web 服务器上运行着一个倾听 TCP 的 80 端口的进程,等待来自客户端的 HTTP 请求。当一个链接发生的时候,客户端发送一个请求,当服务器收到这个请求后,服务器会将客户端请求的数据返回到客户端。

3.HTTP 协议的运作方式

HTTP 协议是基于请求/响应模式的。一个客户机与服务器建立连接后,发送一个请求给服务器。服务器接到请求后,给予相应的响应信息。其格式为一个状态行,包括信息的协议版本号、一个成功或错误的代码,再跟上 MIME 信息,包括服务器信息、实体信息和可能的内容。

许多 HTTP 通信是由一个用户代理初始化的,并且包括一个申请在源服务器上资源的请求。最简单的情况可能是在用户代理(UA)和源服务器(O)之间通过一个单独的连接来完成。当一个或多个中介出现在请求/响应链中时,情况就变得复杂一些。中介有三种:代理、网关和通道。一个代理根据 URI 的绝对格式接受请求,重写全部或部分消息,通过 URI 的标识把已格式化的请求发送到服务器。网关是一个接收代理,作为一些其他服务器的上层,可以把请求翻译给下层的服务器协议。一个通道作为不改变消息的两个连接之间的中继点。当通信需要通过一个中介(如防火墙等)或者是中介不能识别消息的内容时,通道经常被使用。

在 Internet 上,HTTP 通信通常发生在 TCP/IP 连接之上。缺省端口是 TCP 80,其他端口也是可用的,但这并不预示着 HTTP 协议在 Internet 或其他网络的其他协议之上才能完成。HTTP 只预示着一个可靠的传输。

以上简要介绍了 HTTP 协议的宏观运作方式,下面介绍 HTTP 协议的内部操作过程。

现在,简单介绍基于 HTTP 协议的客户/服务器模式的信息交换过程。它分四个步骤:建立连接、发送请求信息、发送响应信息和关闭连接。

在 WWW 中,“客户”与“服务器”是一个相对的概念,只存在于一个特定的连接期间,即在某个连接中的客户在另一个连接中可能作为服务器。WWW 服务器运行时,一直在 TCP80 端口(WWW 的默认端口)监听,等待连接的出现。

下面,讨论 HTTP 协议下客户/服务器模式中信息交换的实现。

(1) 建立连接

连接的建立是通过申请套接字实现的。客户打开一个套接字并把它约束在一个端口上,如果成功,就相当于建立了一个虚拟文件。以后,就可以在该虚拟文件上写数据并通过网络向外传送。

(2) 发送请求

打开一个连接后,客户机把请求消息送到服务器的停留端口上,完成提出请求动作。HTTP/1.0 的请求消息的格式为:

请求消息 = 请求行(通用信息|请求头|实体头) CRLF[实体内容]

其中:

请求行 = 方法 请求 URL HTTP 版本号 CRLF

方法 = GET|HEAD|POST|扩展方法

URL = 协议名称 + 宿主名 + 目录与文件名

请求行中的方法描述指定资源中应该执行的动作,常用的方法有 GET、HEAD 和 POST。不同的请求对象对应 GET 的结果是不同的,对应关系如表 6-2 所示。

表 6-2 GET 命令所取回对象的内容

对象	GET 的结果
文件	文件的内容
程序	该程序的执行结果
数据库查询	查询结果

HEAD:要求服务器查找某对象的元信息,而不是对象本身。

POST:从客户机向服务器传送数据,在要求服务器和 CGI(通用网关接口)做进一步处理时会用到 POST 方法。POST 主要用于发送 HTML 文本中 FORM 的内容,让 CGI 程序处理。

一个请求的例子为:

GET http://WWW.swjtu.edu.cn/home.html HTTP/1.0

头信息又称为元信息,即信息的信息,利用元信息可以实现有条件的请求或应答。

- 请求头:告诉服务器怎样解释本次请求,主要包括用户可以接受的数据类型、压缩方法和语言等。
- 实体头:实体信息类型、长度、压缩方法、最后一次修改时间、数据有效期等。
- 实体:请求或应答对象本身。

(3) 发送响应

服务器在处理完客户的请求后,要向客户机发送响应消息。HTTP/1.0 的响应消息格式如下:

响应消息 = 状态行(通用信息头|响应头|实体头) CRLF [实体内容]

状态行 = HTTP 版本号 状态码 原因叙述

状态码表示响应类型。

- 1×× 保留
- 2×× 表示请求成功地接收,例如 200
- 3×× 为完成请求客户需进一步细化请求
- 4×× 客户错误
- 5×× 服务器错误

响应头功能包括给出服务程序名、告诉客户所请求的 URL 需要认证以及请求的资源何时能使用等信息。

(4) 关闭连接

客户和服务双方都可以通过关闭套接字结束 TCP/IP 对话。

当使用者单击有超文本链接的文字或图像时,浏览器就将有超文本链接所代表的 URL(Uniform Resource Locator)显示出来。一般来说,一个浏览器被使用者选中了一个超文本链接后,进行如下几步操作:

- ① 浏览器中的某个 URL 被使用者选中。
- ② 浏览器使用 DNS 查询 URL 中的域名所代表的 IP。
- ③ DNS 发出一个回应,通知浏览器查询域名所代表的 IP。
- ④ 浏览器通过 TCP 协议连接到 Web 服务器的 80 端口。
- ⑤ 浏览器向 Web 服务器发送一个 GET HTTP://... 请求。
- ⑥ Web 服务器将 index.html 发送过来。
- ⑦ 浏览器结束这个 TCP 连接。
- ⑧ 浏览器将 index.html 这个页面显示出来。
- ⑨ 浏览器将 index.html 中包含的图像、声音等下载、显示和播放。

所有新版本的 HTTP 协议支持两种不同的请求格式。

● 简单请求方式

它发送的请求数据的字符串只包含一个 GET 行,指明要取的页面地址,但没有注明 HTTP 协议的版本号。它所获得的服务器的回应只包含它所请求页面的数据,没有任何 HTTP 信息头,也没有 MIME 信息。

● 完全请求方式

它在发送请求页面的 HTTP 信息的时候,在 GET 命令行的末尾处要添加上 HTTP 协议的版本号。请求信息可以包含很多行的数据,最后跟着是一个空行,代表 HTTP 请求数据头的结束。

通常使用的都是完全请求方式。

下面简述一下 request/response 模式。

- request: 指出要访问的资源地址和输入数据过程需要的数据。
- response: 返回答复状态及描述数据内容的信息和数据内容本身。

这种请求/回应模式使用 MIME 来编码所请求数据,这种模式是无状态的,即每一个请求/回应过程之间是无联系的。在 HTTP/1.0 版本中,运行一次操作即建立一次 TCP 连接,完成这次交互操作即断开连接。HTML 页面包含多个图像内容,获得它要建立的多个 TCP 连接。这种通信方式效率低,因为在连接中要进行多个 HTML 操作。在 HTTP/1.1 中,浏览器和服务器可建立持久的 TCP 连接。这条连接可实现多个 HTTP 交互操作,但即使如此,每一个 HTTP 操作之间也没有逻辑关系。一个 HTTP 操作叫做一个“transaction”。HTTP 协议采用这种通信模式是有道理的,因为 HTML 为超链接,用户很可能从一个页面跳转到其他服务器中的页面,在这期间保持 HTTP 连接是无意义的。

HTTP 采用头信息数据为传递的数据提供说明信息。请求信息包括:访问资源地址、请求头标(说明请求)和请求消息正文。格式为:

请求行:请求方法(GET/Head/POST) 资源地址 版本号

请求头信息:列出浏览器能处理的数据的 MIME 类型



(空一行)

请求消息正文

响应信息包括:响应状态行(报告 HTTP 操作状态)、响应头标、响应消息正文及被访问资源内容。格式是:

状态行:版本号 状态码 原因状语

响应头信息:用来说明响应内容及服务器类型

(在此空一行)

消息正文

实现一个 HTTP 交互操作的过程如下:

① 在浏览器中键入地址。

例如:`http://WWW.nowhere.edu/index.html HTTP/1.0`。

② 浏览器产生一个 HTTP 请求。

例如:`GET /index.html HTTP/1.0`

`Accept: *.*`

③ 浏览器与服务器上的 HTTP 服务器建立连接,发送 HTTP 请求。

④ 服务器检查“/index.html”这个文件是否存在,若不存在则返回错误。

`HTTP/1.0 404 Not Found`

`Date: Sat, 8 Jul 2001 15:08 GMT`

`Server: ipserver - httpd 0.0.1`

⑤ 若文件存在,服务器所做的工作如下:

服务器确定文件类型(HTML、GIF 和 CGI),服务器向浏览器返回成功状态,文件内容即相关信息和文件数据内容本身。浏览器请求一个目录而不是确定文件时,标准应答从那个目录返回一个 index.html 文件。如果找到 index.html 文件,则将该文件返回给浏览器,找不到则产生目录内容列表将它作为 Html 文档返回给浏览器。

⑥ 浏览器从响应内容中提取内容,例如 HTML 文档等。

下面介绍一些关于 MIME 协议的知识。MIME 为多用途的网络邮件扩充协议。它的特点有:

- 传输一份消息中可以含有多个信件正文;
- 可同时表示 ASC II 码文本与非 ASC II 码文本文件;
- 可显示图像、音频及二进制等非文本信息;
- 多种语言、多种字体均可使用。

MIME 定义的 5 种头标:

- MIME-version:版本号;
- MIME-type:数据类型和子类型如表 6-3 所示;
- Content-Transfer-Encoding:内容传输编码方式;
- Content-id:内容标识;
- Content-Description:简单描述实体内容。

在 HTTP 请求/回应模式中用到了前两种头标。

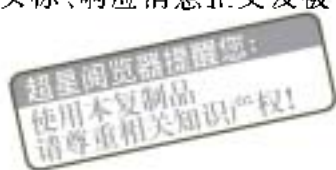


表 6-3 常用类型/子类型一览表

类型/子类型	描述	文件后缀
Text/plain	纯 ASCII 码文档	.txt
Text/html	HTML 文本	.html .htm
Image/gif	GIF 图像	.gif
Image/jpeg	JPEG 图像	.jpeg .speg
Application/msword	Microsoft word	
Video/mpeg	MEPG 视频	
Audio/wave	Wave 音频	
Application-tar	压缩文件	.tar

6.4.2 一个简单的 Web 服务器的样例

学习了 6.4.1 小节,对 HTTP 协议有了个基本的认识,对客户机/服务器间的通信方式有了基本的了解。下面来学习一个基于嵌入式 Linux 开发平台的 Web 服务器的程序。该程序的源代码可以从本书所附的光盘中找到。它被烧写在开发板的闪存内,在系统启动时其守护进程在后台运行。当打开一个浏览器并键入开发板的 IP 地址 192.168.2.124 时,开发板上的 Web 服务器会在收到浏览器的请求后,将目录/htdocs 下的文件 index.html、logo.gif 等传送给浏览器并在浏览器段显示出来。当然也可以改动 index.html、logo.gif 等文件编制个性化的网页。

一个运行于开发板的 Web 服务器的输出结果如图 6-8 所示。

1. 程序主体分析

这个小型服务器程序可响应浏览器对超文本文件以及其他格式文件的请求,并发送数据。如前所述:浏览器向服务器发送请求信息,服务器由等待状态进入接收状态,接收请求内容数据,进行判断、分析并回应。一般而言,请求行中的资源地址路径部分可能有以下几种形式:

- 路径字符串以空格与版本号隔开;
- 路径字符串以 '/' 为结束标志,版本号省略;
- 路径字符串以 '?' 结束,后接请求用户输入的数据(如:/cgi-bin/mail.pl? name = zhb3274 HTTP/1.0)。

程序截取路径字符串并进行分析,若是一个确定的目录而不是文件,则在相应的目录路径下查找是否有 index.html 文件。若找到了,则将该文件返回给浏览器;如果没有找到,则产生目录内容列表并将它作为 html 文档返回给浏览器。若字符串以后缀 .gif 结束,返回 GIF 文本;若以 .html 结尾,返回 HTML 超本文档;以 .txt 结束,返回 TEXT 文档。若不存在路径字符串,可查找相应网页下的 index.html 文件是否存在。若存在,则返回 HTML 文档。若服务器中不存在浏览器请求文件,则返回回应状态行,并告之文



图 6-8 Web 服务器的输出结果

基于以上的思想,来看运行在开发板上的样例程序。该程序源代码仅有几百行,所以都把它写出来了。该程序包括了一个必不可少的 `main()` 函数和一些诸如 `printhtmlheader()`、`Dohtml` 等的功能函数。其结构如图 6-9 所示。

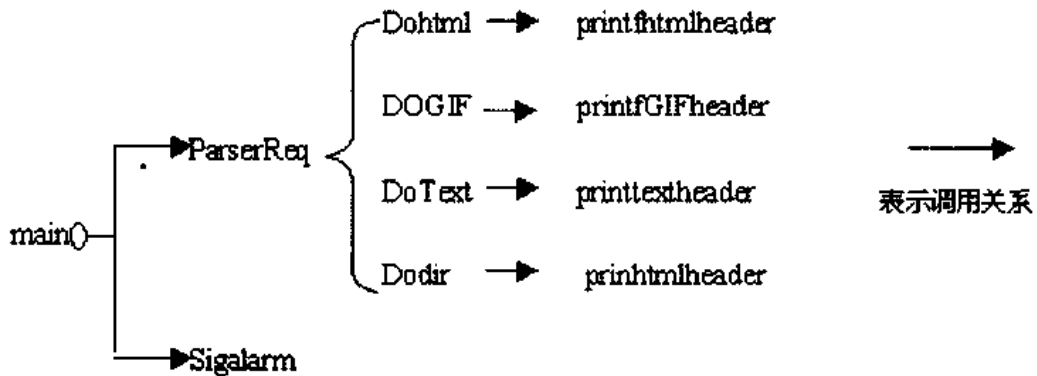


图 6-9 Webservice 程序调用结构图

各个函数的功能见下面的程序分行注释。简而言之,这个 Web 服务器的实现核心是:`main()` 函数使用 `ParserReq()` 函数对浏览器发送来的命令请求进行解析,然后根据不同的命令请求调用不同的功能函数来传送浏览器所需的文件和信息。

2. 程序及分行注释

例 6-3 简单的嵌入式 Web 服务器样例

```
/* httpd.c: A very simple http server
 * Copyright (C) 1998 Kenneth Albanowski <kjahds@kjahds.com>,
 * 1997, 1998 D. Jeff Dionne <jeff@ryebam.ee.ryerson.ca>,
 * The Silver Hammer Group, Ltd.
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <errno.h>
#include <sys/stat.h>
#include <dirent.h>
#include <signal.h>
#include <unistd.h>
#include <ctype.h>

#include "stamp.h"

int TIMEOUT = 30; /* 延时设置 */

/* 条件编译指令 */
#ifndef O_BINARY
#define O_BINARY 0
#endif

char referrer[128]; /* 该数组存放的信息指出是从哪个页面连接到当前资源的 */
```

```
int content_length; /* 存放文件长度 */

#define SERVER_PORT 80 /* 指定服务器端口号,这是通用的 http 协议端口号 */

int
PrintTextHeader(FILE *f) /* 打印文本文件信息头函数 */
{
    alarm(TIMEOUT); /* 设置报警时钟记时 30 秒 */
    fprintf(f,"HTTP/1.0 200 OK \n"); /* 响应状态行 版本号 状态码(被接受,可理
        * 解、访问) 原因状语 */
    fprintf(f,"Content-type: text/plain \n"); /* 文件类型:纯 ASCII 码文本 */
    /* fprintf(f,"Date: Sun, 26 Apr 1998 12:00:00 GMT \n"); /* 响应创建时间 */
    fprintf(f,"Server: ipserver-httpd 0.0.1 \n"); /* 描述服务器 */
    fprintf(f,"Expires: 0 \n"); /* 正文内容过期时间 */
    fprintf(f," \n"); /* 空白行 */
    alarm(0);
    return 0; /* 将报警时钟失效 */
}

int
PrintHTMLHeader(FILE *f) /* 打印 HTML 文本文件信息头函数 */
{
    alarm(TIMEOUT);
    fprintf(f,"HTTP/1.0 200 OK \n");
    fprintf(f,"Content-type: text/html \n"); /* HTML 文本 */
    /* fprintf(f,"Date: Sun, 26 Apr 1998 12:00:00 GMT \n"); */
    fprintf(f,"Server: ipserver-httpd 0.0.1 \n");
    fprintf(f,"Expires: 0 \n");
    fprintf(f," \n");
    alarm(0);
    return 0;
}

int
PrintGIFHeader(FILE *f) /* 打印 GIF 图像信息头文件信息 */
{

```

```
alarm(TIMEOUT);
fprintf(f,"HTTP/1.0 200 OK \n");
fprintf(f,"Content-type: image/gif \n");
/* fprintf(f,"Date: Sun, 26 Apr 1998 12:00:00 GMT \n"); */
fprintf(f,"Server: ipserver-httpd 0.0.1 \n");
fprintf(f,"Expires: 0 \n");
/* fprintf(f,"Expires: Fri, 31 Dec 1999 12:00:00 GMT \n"); */
fprintf(f," \n");
alarm(0);
return 0;
}

int
DoGif(FILE * f, char * name) /* 发送 GIF 图像类型信息函数 */
{
    char * buf;
    FILE * infile;
    int count;

    if (! (infile = fopen(name, "r"))) { /* 以只读方式打开 name 指针指向的文件 */
        alarm(TIMEOUT);
        fprintf(f, "Unable to open GIF file %s, %d \n", name, errno); /* 打开文件 */
        fflush(f);
        alarm(0);
        return -1;
    }

    PrintGIFHeader(f); /* 打印 GIF 图像信息头文件信息 */

    copy(infile,f); /* 将该 GIF 文件发送到浏览器端 */

    alarm(TIMEOUT);
    fclose(infile);
    alarm(0);

    return 0;
}
```



```
int
DoDir(FILE * f, char * name) /* 产生目录文档列表作为 HTML 文档发送给浏览器 */
{
    char * buf;
    DIR * dir;
    struct dirent * dirent; /* 目录结构体 */
    /* 打开目录列表 */
    if ((dir = opendir(name)) == 0) {
        fprintf(f, "Unable to open directory %s, %d\n", name, errno);
        fflush(f);
        return -1;
    }

    PrintHTMLHeader(f); /* 打印 html 文件头信息 */

    alarm(TIMEOUT);
    fprintf(f, "<H1>Index of %s</H1> \n\n", name); /* 向浏览器发送 Index of 信息, 浏
        * 览器将依照 html 文件格式将其显示出来 */
    alarm(0);
    /* 发送目录信息 */
    while(dirent = readdir(dir)) {
        alarm(TIMEOUT);
        /* 条件编译指令, */
        #if 0
            if (dirent->d_name[0] == '.')
                continue;
        #endif
        if (name)
            fprintf(f, "<p><a href = \"/... %s/ %s\">%s</a></p> \n", name/
* + 2 */ , dirent->d_name, dirent->d_name);
        else
            fprintf(f, "<p><a href = \"/.../ %s\">%s</a></p> \n",
dirent->d_name, dirent->d_name);
        alarm(0);
    }

    closedir(dir);
}
```



```
    return 0;
}

int
DoHTML(FILE * f, char * name) /* 返回 HTML 文本文件信息 */
{
    char * buf;
    FILE * infile;
    int count;
    char * dir = 0;
    /* 打开 HTML 文本文件 */
    if (! (infile = fopen(name, "r"))) {
        alarm(TIMEOUT);
        fprintf(f, "Unable to open HTML file %s, %d\n", name, errno);
        fflush(f);
        alarm(0);
        return -1;
    }

    PrintHTMLHeader(f);

    if (dir = strrchr(name, '/')) /* dir 指向请求信息中路径名的第一个 '/' 符号 */
    {
        * dir = '\0'; /* 填入结束符 */
        chdir(name); /* 使第一层路径为子进程当前目录 */
    }

    copy(infile, f); /* 打印当前页 */

    if (dir) {
        chdir ("/htdocs"); /* 恢复原来子进程路径 */
    }

    alarm(TIMEOUT);
    fclose(infile);
    alarm(0);
}
```



```
    return 0;
}

int
DoText(FILE * f, char * name) /* 返回 ASCII 码文本文件信息 */
{
    char * buf;
    FILE infile;          /* FILE 为文件类型的指针。它指向所打开的文本文件 */
    int count;

    if (! (infile = fopen(name, "r"))) /* 打开文件 */
    {
        alarm(TIMEOUT);
        fprintf(f, "Unable to open HTML file %s, %d \n", name, errno);
        fflush(f);
        alarm(0);
        return -1;
    }

    PrintTextHeader(f);          /* 打印文本文件头 */
    copy(infile, f);            /* 打印文本文件 */

    alarm(TIMEOUT);
    fclose(infile);
    alarm(0);

    return 0;
}

int
ParseReq(FILE * f, char * r) /* 处理请求信息函数 */
{
    char * bp;
    struct stat stbuf;
    char * arg;
    char * c;
    int e;
    int raw;
```




```

#ifdef DEBUG
    printf("req is '%s' \n", r);
#endif

while( *(++r) != '\0');
while(isspace( * r))/ * 跳过空格 */
    r++;

while( * r == '/')/* 跳过 '/' */
    r++;

bp = r;/* bp 指向资源地址字符串头部 */
/** r = '.'; */

while( * r && ( * (r) != '\0') && ( * (r) != '?'))/* 指向地址字符串尾部 */
    r++;

#ifdef DEBUG
    printf("bp = '%s', r = '%s' \n", bp, r);
#endif

if( * r == '?')/* 如果尾部字符为 '?' */{
    char * e;
    * r = 0;/* 将其内容设为结束符 */
    arg = r + 1;/* arg 指向下一个位置 */
    if( e = strchr(arg, '\0'))/* e 指向下一个空格的位置 */{
        * e = '\0';/* 设为结束符 */
    }
} else {
    arg = 0;/* 没有用户的输入数据 */
    * r = 0;/* 将其内容设为结束符 */
}

c = bp;/* c 指向资源路径字符串头部 */

if( c && ! stat(c, &stbuf))/* 如果请求资源路径字符串存在且服务器中存在相

```



```

应文件 */
    if (S_ISDIR(stbuf.st_mode)) /* 如果请求资源是一个目录 */
    {
        char * end = c + strlen(c); /* end 指向字符串尾 */
        strcat(c, "/index.html"); /* 将/index.html 接在路径字符串后 */
        if (! stat(c, &stbuf)) /* 测试能否在此目录下找到 index.html 文件 */
            DoHTML(f, c); /* 能够找到 index.html 文件,调用 dohtml 函数,将其发送 */
        else
            *end = '\0'; /* 若不能,调用 dkdir 函数,只将目录内容列表返回给浏览器 */
        DoDir(f, c);
    }
    else if (! strcmp(r + 4, ".gif")) /* 若请求资源以 .gif 结束,调用 dogif 函数 */
        DoGif(f, c);
    else if (! strcmp(r + 5, ".html")) /* 若请求资源以 .html 结束,调用 dohtml 函数 */
        DoHTML(f, c);
    else
        DoText(f, c); /* 非以上三种情况,调用 dotext 函数 */
} else
if( *c == 0) /* 若请求资源中不存在路径 */
{
    strcat(c, "index.html"); /* 若能找到相应 Index.html 文件,则将该文件返回给浏览器 */
    if (! stat(c, &stbuf))
        DoHTML(f, c);
}
else /* 若服务器中不存在请求路径下的文件,只返回响应头标,并告之浏览器请求文
    * 件不存在 */
{
    int e = errno;
    PrintHTMLHeader(f);
    alarm(TIMEOUT);
    fprintf(f, "<p>File '%s' not found (error %d)</p> \n",
/* 浏览器上显示 bp 所指路径字符串下的文件不存在 */
        bp, e);
    alarm(0);
}
return 0;

```

```

}

void sigalrm(int signo)
{
/* 获取警告信息,退出并重新循环 */
exit(0);
}

int
main(int argc, char * argv[])
{
    int fd, s;
    int len;
    volatile int true = 1;
    struct sockaddr_in ec;
    struct sockaddr_in server_sockaddr;
    FILE * f;
char buf[160];
    char buf1[160];

    check_version_argument;

    signal(SIGCHLD, SIG_IGN);/* 将 SIGCHLD 设为屏蔽信号 */
    signal(SIGPIPE, SIG_IGN);/* 同上 */
    signal(SIGALRM, sigalrm);/* SIGALRM 设为报警信号 */
chdir("/htdocs");/* 设置子进程当前目录为/htdocs,该目录在 romfs 中 */
restart:
if((s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) == -1)/* 创
                                                    * 建套接字 */ |
    perror("Unable to obtain network");
    exit(1);
}
if((setsockopt(s, SOL_SOCKET, SO_REUSEADDR, (void *)&true,
sizeof(true))) == -1)/* 为套接字设定选项 */|
    perror("setsockopt failed");
    exit(1);
}

```



```

/* 绑定套接字 */
server_sockaddr.sin_family = AF_INET;
server_sockaddr.sin_port = htons(SERVER_PORT);
server_sockaddr.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(s, (struct sockaddr *)&server_sockaddr, sizeof(server_sockaddr)) == -1) {
    if((errno == EADDRINUSE) && (getppid() != 1)) {
        /* 在命令行下启动 httpd 的话,报告错误 */
        printf("Please do not use this command \n");
    } else {
        perror("Unable to bind socket");
        exit(1);
    }
}

/* 等待 */
if(listen(s, 8 * 3) == -1) { /* 8 * 3 为请求排队的大小, 8 个文件/页, 最多 3
                             * 个客户 */
    perror("Unable to listen");
    exit(4);
    goto restart; /* 等待失败,重新创建套接字 */
}

while (1) /* 接收浏览器发送的信息 */
{
    len = sizeof(ec);
    if((fd = accept(s, (void *)&ec, &len)) == -1) {
        exit(5);
        close(s);
        goto restart; /* 接收失败,重新创建套接字 */
    }

    f = fdopen(fd, "a+"); /* 打开超文本信息文件 */
    if (!f) { /* 错误校验 */
        fprintf(stderr, "httpd: Unable to open httpd input fd, error %d \n", errno);
        alarm(TIMEOUT);
        close(fd);
        alarm(0);
        continue;
    }

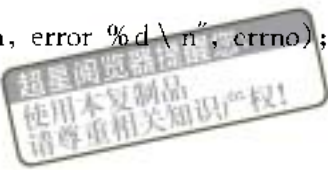
    setbuf(f, 0); /* 清空缓冲区 */
    alarm(TIMEOUT);
}

```

```

if (! fgets(buf, 150, f)) /* 读信息 */ {
    fprintf(stderr, "httpd: Error reading connection, error %d\n", errno);
    fclose(f);
    alarm(0);
    continue;
}
#ifdef DEBUG
    printf("buf = '%s'\n", buf);
#endif
alarm(0);
referrer[0] = '\0'; /* 连接超时,自动断开 */
content_length = -1;
alarm(TIMEOUT);
while (fgets(buf1, 150, f) && (strlen(buf1) > 2)) /* 套接字循环接收数据 */
{
    alarm(TIMEOUT);
#ifdef DEBUG
        printf("Got buf1 '%s'\n", buf1);
#endif
    if (! strncasecmp(buf1, "Referer:", 8)) /* 若数据前 7 个字符是 Referer,则 c 指向第
九个字符 */ {
        char * c = buf1 + 8;
        while (isspace(*c)) /* 跳过' ' */
            c++;
        strcpy(referrer, c); /* 将 c 所指字符串与 referrer 连接 */
    } else if (! strncasecmp(buf1, "Referer:", 9)) ? /* 若前 8 个字符是 Referer,则 c 指
向第九个字符 */ {
        char * c = buf1 + 9;
        while (isspace(*c))
            c++; /* 跳过' ' */
        strcpy(referrer, c); /* 接在 referrer 后面 */
    } else if (! strncasecmp(buf1, "Content-length:", 15)) /* 如果前面字符是
Content-length,则 c 指向第 16 个字符 */ {
        content_length = atoi(buf1 + 15); /* 返回请求文件整型长度 */
    }
}
}

```



```

alarm(0);
if (ferror(f)) { /* 读信息,若失败,错误显示 */
    fprintf(stderr, "http: Error continuing reading connection, error %d \n", errno);
    fclose(f);
    continue;
}

ParseReq(f, buf); /* 解析浏览器发送的请求命令参数,并做对应操作 */
alarm(TIMEOUT);

fflush(f); /* 强制将所有缓冲区的数据内容写到文件流 f 中去 */
fclose(f);
alarm(0);
}
}

```

本程序的流程图如图 6-10 所示。

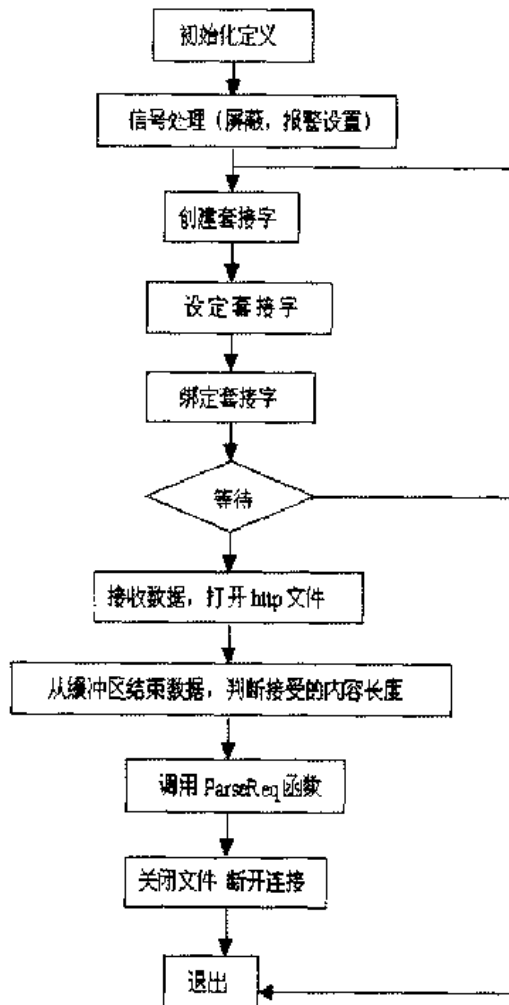


图 6-10 嵌入式 Webserver 流程图

在程序中用到了一些进程间通信函数,现解释如下:

```
alarm(TIMEOUT);
```

alarm()函数的功能是建立一个进程的报警时钟。当时钟定时器到达所定的时间时,向系统发送一个警报 SIGALRM 信号。在本程序中,使用 alarm()函数是保证在 TIMEOUT 时间内,完成所需功能。如果未完成,则向系统发出 SIGALRM 中断信号。在 main()函数中,有这么一句:

```
signal(SIGALRM, sigalrm);/* 将 SIGALRM 设为报警信号 */
```

它调用函数 signal(),将 SIGALARM 信号同函数 sigalrm()相关联。sigalarm()函数的功能是仅捕获警告信号 SIGALARM,然后退出。即是说,当某个子程序,如 PrintText-Header(FILE *f)在设定的 TIMEOUT 内未完成使命则会产生中断,退出并重新循环。这起到了控制传输时间的作用。

关于 signal 函数的使用,请参考 Linux 系统的进程间通信(IPC)部分的章节。

6.5 本章小结

本章主要讲述了嵌入式 Linux 设备接入互联网和其网络功能的实现等内容。首先,探讨了嵌入式 Internet 技术,讲述了设备接入互联网的方法,重点讨论了使用 Linux 将 8/16 位的嵌入式设备接入互联网的方法。然后,学习了 Linux 下网络编程的基本知识,并通过学习在嵌入式 Linux 开发平台上开发的一个网口通信程序,巩固了所学的知识。最后,了解了 HTTP 协议,并通过一个简单紧凑的 Webserver 的编写,系统地学习和实现了 HTTP 协议的基本功能。

在开发平台上所实现的网络功能还是比较单薄的,应在该开发平台上添加更多的功能。例如,可以在开发板上烧写 CGI 程序以完成对系统的控制。如果各位感兴趣,可以在该开发平台上一试身手。希望各位能够好好地研习本章所给出的范例程序,认真研读,从而编写出自己的应用程序。

第7章 嵌入式 Linux 下的串行通信

超星网资源
使用本复制品
请尊重相关知识产权!

嵌入式设备与外围设备通信的方式方法一直是人们关心的问题。本章讨论使用串行通信口与外部设备相通信的问题。首先,讲述串行口的物理规范,并以 RS-232 口为例,简述其物理特性。然后,学习在 Linux 下的串行通信规范及如何进行串行通信编程的基本知识和范例。最后,以开发平台为例,着重讲述嵌入式 Linux 环境下使用 RS-232 接口传递数据的编程方法和实现过程,并给出详尽的分析。

本章主要内容:

- 串行口物理特性
- RS-232 串行总线
- UNIX/Linux 下串行通信的规范简介
- 串行口编程的基本方法
- 串行口通信示例

7.1 串行口的物理标准

7.1.1 关于总线

在介绍串行口的物理标准前,先了解总线的含义。

任何一个微处理器都要与一定数量的部件和外围设备连接,但如果将各部件和每一种外围设备都分别用一组线路与 CPU 直接连接,那么连线将会错综复杂,甚至难以实现。为了简化硬件电路设计和系统结构,常用一组线路,配置适当的接口电路,与各部件和外围设备连接,这组共用的连接线路被称为总线。采用总线结构便于部件和设备的扩充,尤其制定了统一的总线标准后,更容易使不同设备间实现互连。

微机中总线一般有内部总线、系统总线和外部总线。

- 内部总线:是微机内部各外围芯片与处理器之间的总线,用于芯片一级的互连。
- 系统总线:是微机中各插件板与系统板之间的总线,用于插件板一级的互连。
- 外部总线:是微机和外部设备之间的总线,微机作为一种设备,通过该总线和其他设备进行信息与数据交换,用于设备一级的互连。

另外,从广义上说,计算机通信方式可以分为并行通信和串行通信,相应的通信总线被称为并行总线和串行总线。并行通信速度快、实时性好,但由于占用的端口线路多,不

适于小型化产品。而串行通信速率虽低,但在数据通信吞吐量不是很大的微处理电路中,则显得更加简单、方便、灵活。串行通信一般可分为异步模式和同步模式。

随着微电子技术和计算机技术的发展,总线技术也在不断地发展和完善,使计算机总线技术种类繁多,各具特色。下面仅介绍微机各类总线中比较流行的总线技术。

1. 内部总线

(1) I2C 总线

I2C 总线于 10 多年前由 Philips 公司推出,是近年来在微电子通信控制领域广泛采用的一种新型总线标准。它是同步通信的一种特殊形式,具有接口线少、控制方式简化、器件封装形式小、通信速率较高等优点。在主从通信中,可以有多个 I2C 总线器件同时接到 I2C 总线上,通过地址来识别通信对象。

(2) SPI 总线

串行外围设备接口 SPI 总线技术是由 Motorola 公司推出的一种同步串行接口。Motorola 公司生产的绝大多数微控制器(MCU)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。

(3) SCI 总线

串行通信接口 SCI 也是由 Motorola 公司推出的。它是一种通用异步通信接口 UART,与 MCS-51 的异步通信功能基本相同。

2. 系统总线

(1) ISA 总线

ISA 总线标准是 IBM 公司在 1984 年为推出 PC/AT 机而建立的系统总线标准,所以也叫 AT 总线。它是对 XT 总线的扩展,以适应 8/16 位数据总线要求。它在 80286 至 80486 时代应用得非常广泛,以至于现在奔腾机中还保留有 ISA 总线插槽。ISA 总线有 98 只引脚。

(2) EISA 总线

EISA 总线是 1988 年由 Compaq 等 9 家公司联合推出的总线标准。它是在 ISA 总线的基础上使用双层插座,在原来 ISA 总线的 98 条信号线上又增加了 98 条信号线,也就是在两条 ISA 信号线之间添加一条 EISA 信号线。在实用中,EISA 总线完全兼容 ISA 总线信号。

(3) VESA 总线

VESA 总线是 1992 年由 60 家附件卡制造商联合推出的一种局部总线,简称为 VL 总线。它的推出为微机系统总线体系结构的革新奠定了基础。该总线系统考虑到 CPU 和主存与 Cache 的直接相连,通常把这部分总线称为 CPU 总线或主总线,其他设备通过 VL 总线与 CPU 总线相连,所以 VL 总线被称为局部总线。它定义了 32 位数据线,且可通过扩展槽扩展到 64 位,使用 33MHz 时钟频率,最大传输率达 132MB/s,可与 CPU 同步工作。它是一种高速、高效的局部总线,可支持 386SX、386DX、486SX、486DX 及奔腾微处理器。

(4) PCI 总线

PCI 总线是当前最流行的总线之一,它是由 Intel 公司推出的一种局部总线。它定义了 32 位数据总线,且可扩展为 64 位。PCI 总线主板插槽的体积比原 ISA 总线插槽还小,其功能比 VESA、ISA 有极大的改善,支持突发读写操作,最大传输速率可达 132MB/s,可同时支持多组外围设备。PCI 局部总线不能兼容现有的 ISA、EISA 和 MCA 总线,但它不受制于处理器,是基于奔腾等新一代微处理器而发展的总线。

(5) Compact PCI

以上所列举的几种系统总线一般都用于商用 PC 机中。在计算机系统总线中,还有另一大类为适应工业现场环境而设计的系统总线,比如 STD 总线、VME 总线、PC/104 总线等。这里仅介绍当前工业计算机的热门总线之一——Compact PCI。

Compact PCI 的含义是“坚实的 PCI”,是当今第一个采用无源总线底板结构的 PCI 系统,是 PCI 总线的电气和软件标准加欧式卡的工业组装标准,是当今最新的一种工业计算机标准。Compact PCI 是在原来 PCI 总线基础上改造而来的。它利用 PCI 的优点,提供满足工业环境应用要求的高性能核心系统,同时还充分考虑了利用传统的总线产品,如 ISA、STD、VME 或 PC/104 来扩充系统的 I/O 和其他功能。

3. 外部总线

(1) RS-232-C 总线

RS-232-C 是美国电子工业协会 EIA 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。RS-232-C 标准规定:驱动器允许有 2500pF 的电容负载,通信距离将受此电容限制。例如,采用 150pF/m 的通信电缆时,最大通信距离为 15m。若每米电缆的电容量减小,通信距离就可以增加。传输距离短的另一原因是 RS-232 属单端信号传送,存在共地噪声和不能抑制共模干扰等问题,因此一般用于 20m 以内的通信。

(2) RS-485 总线

在要求通信距离为几十米到上千米时,广泛采用 RS-485 串行总线标准。RS-485 采用平衡发送和差分接收,因此具有抑制共模干扰的能力。加上总线收发器具有高灵敏度,能检测低至 200mV 的电压,故传输信号能在千米以外得到恢复。RS-485 采用半双工工作方式,任何时候只能有一点处于发送状态,因此,数据发送电路须由使能信号加以控制。RS-485 用于多点互连时非常方便,可以省掉许多信号线。应用 RS-485 可以联网构成分布式系统,其允许最多并联 32 台驱动器和 32 台接收器。

(3) IEEE-488 总线

上述两种外部总线是串行总线,而 IEEE-488 总线是并行总线接口标准。IEEE-488 总线用来连接系统,如微计算机、数字电压表、数码显示器等设备及其他仪器仪表均可用 IEEE-488 总线装配起来。它按照位并行、字节串行双向异步方式传输信号,连接方式为总线方式,仪器设备直接并联于总线上而不需中介单元。总线上最多可连接 15 台

设备,最大传输距离为 20 米,信号传输速度一般为 500KB/s,最大传输速度为 1MB/s。

(4) USB 总线

通用串行总线 USB 是由 Intel、Compaq、Digital、IBM、Microsoft、NEC 和 Northern Telecom 七家世界著名的计算机和通信公司共同推出的一种新型接口标准。它基于通用连接技术,实现外设的简单快速连接,达到方便用户、降低成本、扩展 PC 连接外设范围的目的。它可以为外设提供电源,而不像普通的使用串、并口的设备需要单独的供电系统。另外,快速是 USB 技术的突出特点之一,USB 的最高传输率可达 12Mbps,比串口快 100 倍,比并口快近 10 倍,而且 USB 还能支持多媒体。

7.1.2 RS-232 串行口

现在详细介绍 RS-232 总线及其接口。RS-232-C 是电子工业协会(EIA)的推荐标准,是一种很流行的异步传送标准串行总线。最初,RS-232-C 是为公共电话网络的数据通信而设计的。在 20 世纪 60 年代中,它被用于计算机和终端通过电话线和调制解调器进行的远距离通信。在 20 世纪 80 年代,随着微型计算机的发展,计算机和终端的通信不仅是远距离,而且近距离也采用了 RS-232-C 连接的方式。近距离通信(15m 内),计算机的接口和终端直接采用 RS-232-C 连接,而不再使用电话线和调制解调器。计算机和终端设备使用 RS-232 通信的原理,如图 7-1 所示。

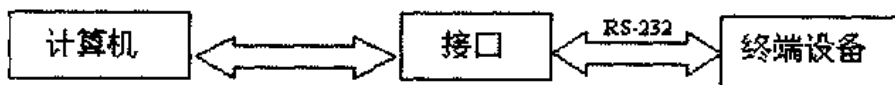


图 7-1 计算机和终端使用 RS-232 口的通信

RS-232 线是数据终端设备(DTE)和数据通信设备(DCE)间的信号传输线。DTE 是所传送数据的源或宿主,可以是一台计算机或一个数据终端。DEC 可以是一个调制解调器或一种数据通信设备。

开发板和 PC 之间的距离不过短短的 0.5m,当然使用的是计算机连接。使用的是 DB-9 型的串行接口,它仅有 9 针,而不是通用的 RS-232-C 的 25 针。各个针脚的定义如下:

DB9

- 1 空
- 2 TX
- 3 RX
- 4 空
- 5 GND
- 6 空
- 7 CTS
- 8 RTS
- 9 空

由此可见,同通用的 RS-232C 相比,DB-9 裁减了很多针脚和功能。

各个针脚的功能如下:

- 2 TX—发送数据线
- 3 RX—接收数据线
- 5 GND—信号地
- 7 CTS—允许发送线
- 8 RTS—请求发送线

RS-232 只能实现 DTE 和 DCE 间的直接连接。如果要实现 DTE 和 DTE 间(或 DCE 同 DCE)的直接连接,就需通过零调制解调器,进行收一发的转换。开发板已经做好了收一发的转换工作,就可以直接同 PC 相连接了。开发平台的串行口线是按照 DCE 的方式来做的。一般而言,DCE 可以是一个调制解调器或一种数据通信的设备和外围设备。

控制串行口通信的控制芯片是 Max232E 型芯片。该芯片是专门针对在恶劣环境下的 RS-232 和 V.28 通信而设计的。各个输出和输入针脚都经过了精心的设置,可以防止 $\pm 15\text{kV}$ 的静电震荡。其管脚分配如图 7-2 所示。各个管脚的使用方法参看相关的技术资料,在此不详细叙述了。

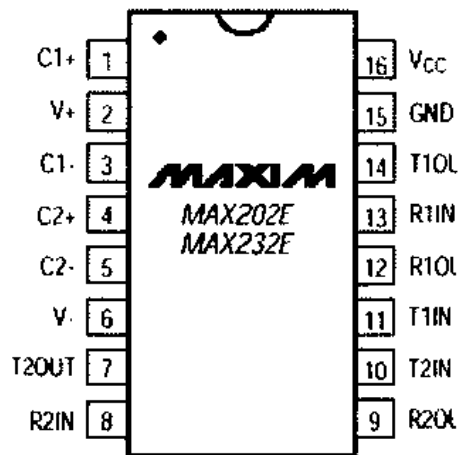


图 7-2 串行口控制芯片管脚分配图

7.2 Linux 下的串行通信编程

下面学习在 Linux 下如何进行串行通信编程。对于嵌入式 Linux 系统而言,其基本的思想是类似的。

7.2.1 串行通信的基础

1. 串行通信的含义

串行通信表示计算机一次传送一个位的数据。串行通信包括的范围很广,从绝大多

数的网络设备到键盘、鼠标、MODEMS 和终端等都使用了串行通信。当使用串行通信时,每个字的数据是一个位一个位地传输或者接收的。每个位不是高电平,就是低电平。

串行通信的速率通常是使用“位/每秒”的方式来表示的,即波特率。在计算机开始起步时,300 波特的速度就是很快的了。如今的计算机,使用 RS-232 通信的速率可以达到 430 800 波特的速率。

上节已讲过,对于串行的设备或接口,它们分为数据通信装置(DCE)和数据终端设备(DTE)两种。作为串行通信的接口标准,RS-232 通常有 3 种模式。各个模式定义了高/低电平的不同的等级,最常用的是 RS-232C 模式。

下面来看各个信号的定义。

RS-232 定义了用来进行串行通信的 18 个不同的信号。在 UNIX/Linux 系统环境下,通常只能使用如下 7 个信号。

(1) GND-逻辑地信号

通常逻辑地不是个信号,但是没有它,其他的信号是无法产生的。一般而言,逻辑地信号的作用是作为一个参考电压,电子设备便能检测出信号是正的还是负的。

(2) TXD-数据传输信号

TXD 信号负责将数据从工作站发送到另外一端的计算机或者终端。令高电平为 1,低电平令为 0。

(3) RXD-接收数据信号

RXD 信号负责将从远端的计算机或设备发送过来的数据传送到用户的工作站。

(4) DCD-数据载波检测信号

DCD 信号是从另外一端的计算机或设备处接收到的。它指出了计算机或设备是否被连接到线上。DCD 信号并不常用。

(5) DTR-数据终端准备好信号

DTR 信号由工作站产生,它告诉另外一端的计算机或设备用户已经准备好了。打开串行接口时,DTR 已经被自动地使能。

(6) CTS-允许发送信号

CTS 信号接收从串行通信线的另外一端发送过来的信号。一个低电平表示可以从工作站发送更多的串行数据。CTS 通常用来进行由工作站到另外一端的数据流控。

(7) RTS-请求发送信号

RTS 信号被用户的工作站设置成低电平,指示有更多的数据已准备好可供传输。同 CTS 一样,RTS 也通常用来进行由工作站到另外一端的数据流控。大多数的工作站将其设置为低电平。

2. 全双工和半双工

全双工意味着计算机可以同时收发数据。它有两个独立的数据通道,一个输入,一个输出。半双工意味着计算机不能同时收发信息,只能有一个通道进行通信。

3. 流控

通常,当数据在两个串行接口之间进行传输时需要对其进行控制。这常常依赖于串

行通信连接的各种规定。对异步数据传输的控制有两种方法。

一种是叫做“软件”流控。该方法使用特别的符号来启动(XON 或 DC1,代码为八进制的 021) 或停止(XOFF 或 DC3,代码为八进制的 023)数据流。这些字符在 ASCII 码中都有定义。虽然这些代码在传输文本信息时有效,但是如果没有使用特别的编程手段,在传输其他类型的信息时,这些代码就无法使用。

第二种方法叫做“硬件”流控。该方法使用 RS-232 的 CTS 和 RTS 信号,而不使用特殊的符号。当接收器准备接收更多的数据时,就将 CTS 设置为低电平,当它还没有准备好时,则设置为高电平。类似地,发送方在它将要发送更多的数据时,将 RTS 设置为低电平。由于硬件流控使用了一套独立的信号,比软件流控要快得多。软件流控在发送相同数量的数据时,需要发送或接收很多位的控制信息,才能达到同样的效果。CTS/RTS 流控并不为所有的硬件和操作系统所支持。

4. 中断

通常,发送或者接收数据的指示信号一般维持高电平,直到传递新的字符为止。如果该信号长期维持在低电平(通常 1/4 到 1/2 秒),即处在一个中断的状态。

有时,中断用来设置通信线路或改变诸如 MODEM 等通信硬件的操作模式。

5. 同步通信

同步与异步数据是不同的。同步数据像是一连贯的数据流,为了在线读取数据,计算机必须提供或者接收一个发送方和接收方可共同识别的数据位,使它们保持收发同步。

即使使用了该同步方法,计算机还是需要标记数据的起始位置。最通常的做法是使用数据包协议,如串行数据连接协议(SDLC)和高速数据连接协议(HDLC)。

各个协议定义了特定的位顺序以表示数据包的起始和结束位置。这些位顺序允许计算机明白数据包在什么地方开始。

由于同步协议没有使用与字符同步处理位,所以在性能上比异步通信至少要高出 25%,非常适合使用两个串行接口来进行远端的连网工作和设置。

尽管同步通信拥有上述的优势,基于其他硬件和软件方面的原因,绝大多数的 RS-232 硬件并不支持同步通信。

6. 访问串行口

同所有的设备一样,UNIX/Linux 通过设备文件访问串行口。为了访问串行口,仅需打开相应的设备文件即可。

在 UNIX 系统里,每个串行端口有一个或多个设备文件同其相关联(在/dev 目录中)。

下面给出了 UNIX 类系统串行端口 Port 1 和 Port 2 的端口设备文件系统。

```
IRIX(R) :/dev/ttyf1    /dev/ttyf2
```

```
HP-UX :/dev/tty1p0    /dev/tty2p0
```

Solaris® 及/SunOS®: /dev/ttya /dev/ttyb

Linux®:/dev/ttyS0 /dev/ttyS1

Digital UNIX®: /dev/tty01 /dev/tty02

(1) 打开一个串行口

由于串行口是一个文件,可以使用 open()函数来访问它。但是,对于一般用户而言,该文件是不能被访问的。如果非要访问不可,那么只能采用其他变通的办法,如改变所访问文件的访问允许属性、用户的身份运行程序等。

假定所有的用户都可以访问该文件,下面的例 7-1 给出在一个运行 IRIX 的工作站上如何打开串行口 1 的代码(Linux 下也非常类似,只是设备的文件名不同而已)。

例 7-1 打开一个串行口

```
#include <stdio.h> /* 标准输入/出定义 */
#include <string.h> /* 字符串功能定义 */
#include <unistd.h> /* UNIX 标准函数定义 */
#include <fcntl.h> /* 文件控制定义 */
#include <errno.h> /* 错误码定义 */
#include <termios.h> /* POSIX 终端控制定义 */

/*
 * 'open_port()' - 打开串行口 1.
 *
 * 成功的话,返回文件描述符,错误则返回 -1.
 */

int
open_port(void)
{
    int fd; /* 端口文件描述符 */

    fd = open("/dev/ttyf1", O_RDWR | O_NOCTTY | O_NDELAY);
    if (fd == -1)
    {
        /*
         * 无法打开串行口.
         */
    }
}
```

```

    perror("open _port: Unable to open /dev/ttyf1 ");
    |
else
    fcntl(fd, F_SETFL, 0);

    return (fd);
}

```

当打开设备文件时,除了使用 read + write 模式外,还可以使用两个其他的标志符号。
fd = open("/dev/ttyf1", O_RDWR | O_NOCTTY | O_NDELAY);

O_NOCTTY 标志告诉 Linux,该程序不想成为此端口的“控制终端”。如果没有强调这一点,那么任何输入(例如键盘的中断信号等)都会影响程序的执行。诸如 getty (1M/8)等程序使用了该特性,但是,通常的用户程序并不想要该特性。

O_NDELAY 标志告诉 Linux,该程序并不关注 DCD 信号线所处的状态,即不管另外一端的设备是在运行还是被挂起。如果没有指定该标志,那么程序就会被设置为睡眠状态,直到 DCD 信号线为低电平为止。

(2) 向端口写数据

向端口写数据是很容易的,只要使用 write()系统调用就可以了。例如:

```

n = write(fd, "ATZ\r", 4);
if (n < 0)
    fputs("write() of 4 bytes failed! \n", stderr);

```

write 函数返回发送数据的个数。如果出现错误,则返回 -1。

(3) 读端口数据

从端口读数据则需要些技巧。如果在原始数据的模式下对端口进行操作,read()系统调用将返回串行口输入缓冲区中所有的字符数据,不管有多少。如果没有数据,那么该调用将被阻塞,处于等待状态,直到有字符输入,或者到了规定的时限和出现错误为止。通过以下方法,能使 read 函数立即返回:

```
fcntl(fd, F_SETFL, FNDELAY);
```

FNDELAY 参数使 read 函数在端口没有字符存在的情况下,立刻返回 0。如果要恢复正常(阻塞)状态,可以调用 fcntl()函数,不要 FNDELAY 参数,如下所示:

```
fcntl(fd, F_SETFL, 0);
```

在使用 O_NDELAY 参数打开串行口后,同样也使用了该函数调用。

(4) 关闭串行口

要关闭串行口,使用如下的系统调用即可。

```
close(fd);
```

关闭一个串行口通常会将 DTR 信号设置为低电平,这就使绝大多数的 MODEM 被挂起。

7.2.2 串行口的设置

下面来讨论在 C 语言下,使用 POSIX 终端接口设置串行口的方法。

1. POSIX 终端接口

绝大多数系统支持 POSIX 终端(串行)接口交换诸如波特率、字符大小等参数。首先要做的就是包含文件 `<termios.h>`,它定义了终端控制结构和 POSIX 的控制函数。

两个最重要的 POSIX 函数是 `tcgetattr()`和 `tcsetattr()`。它们分别用于获取和设置终端的属性。可以提供一个指向 structure `termios` 的指针,该 structure 包含了所有串行参数。

所有的 `termios` 结构的成员及其描述如表 7-1 所示。

表 7-1 `termios` 结构的成员及其描述

参数	功能
<code>C_cflag</code>	控制参数
<code>C_lflag</code> 标志	线控制
<code>C_iflag</code> 标志	输入参数
<code>C_oflag</code> 标志	输出参数
<code>c_cc</code>	控制字符
<code>c_ispeed</code>	输入波特率(新接口)
<code>c_ospeed</code>	输出波特率(老接口)

(1) 控制参数

`c_cflag` 控制波特率、数据位数、硬件的流控等。下面是一些常用的波特率的常数。

`B0 0 baud (drop DTR)`

`B50 50 baud`

`B75 75 baud`

`B110 110 baud`

`B134 134.5 baud`

`B150 150 baud`

`B200 200 baud`

`B300 300 baud`

`B600 600 baud`

`B1200 1200 baud`

```

B1800 1800 baud
B2400 2400 baud
B4800 4800 baud
B9600 9600 baud
B19200 19200 baud
B38400 38400 baud
B57600 57,600 baud
B76800 76,800 baud
B115200 115,200 baud

```

`c_cflag` 成员包含了两个必须时刻保持使能状态的参数: `CLOCAL` 和 `CREAD`。这确保了程序在突发的作业控制和挂起信号到来时,不会成为端口的占有者,同时串行口的接口驱动会读取输入的数据。

波特率常数(`CBAUD`、`B9600` 等)是供那些缺乏 `c_ispeed` 和 `c_ospeed` 成员的老式接口使用的。不要直接地初始化 `c_cflag`(或其他标志)。应该使用诸如 `AND`、`OR` 及 `NOT` 等位操作符,在成员中设置或清除特定的位。这是由于不同的操作系统间对位操作是不同的,这样做可以避免错误的产生。

(2) 设置波特率

波特率存放的地点随操作系统的不同而不同。老式接口将波特率存放在 `c_cflag` 成员中,使用上述的波特率常数来表示。而新的办法是使用了 `c_ispeed` 和 `c_ospeed` 成员,它们包含了实际的波特率的值。

无论是何种操作系统,都可以使用 `cfsetospeed()` 和 `cfsetispeed()` 函数在 `termios` 结构中设置波特率。见下面的例 7-2。

例 7-2 设置波特率

```

struct termios options;
tcgetattr(fd, &options); /* 获取当前端口的参数信息... */

/* 将波特率设置为 19200... */
cfsetispeed(&options, B19200);
cfsetospeed(&options, B19200);

options.c_cflag |= (CLOCAL | CREAD); /* 使能接收,并设置本地状态... */

tcsetattr(fd, TCSANOW, &options); /* 为端口设置新参数.. */

```

`tcgetattr()` 函数把对串行口的当前设置赋予 `termios` 这个数据结构,在设置好波特率,并使能本地状态和串行数据接收后,使用 `tcsetattr()` 函数选择新的设置。`TCSANOW` 常数表示,无须等待数据发送或接收的结束,所有的改变必须立即生效。当然,还有别的常数可完成等待数据发送/接收结束的功能。绝大多数系统并不支持不同的输入和输出速



率,要确保将输入和输出设置为一致。

对 `tcsetattr` 函数所用到的常数的描述,如表 7-2 所示。

表 7-2 `tcsetattr` 函数所用到的常数的描述

常数名称	功能
TCSANOW	不等待数据收发完毕,就改变立即生效
TCSADRAIN	一直等待到所有数据传送完毕
TCSAFLUSH	将输入/出缓冲区清空,改变才生效

(3) 设置字符大小

同波特率的设置不同,不能很方便地用函数来设置字符大小。只能使用一些掩码方法来实现设置。字符大小是按位指定的,如下所示:

```
options.c_cflag &= ~CSIZE; /* 屏蔽字符大小位 */
```

```
options.c_cflag |= CS8; /* 选择 8 数据位 */
```

(4) 设置奇偶校验位

与字符大小设置相同,必须手动设置奇偶校验位。UNIX/Linux 的串行驱动程序支持生成奇、偶校验位,或者无校验位的生成。

实现方法如下:

无校验位 (8N1):

```
options.c_cflag &= ~PARENB
```

```
options.c_cflag &= ~CSTOPB
```

```
options.c_cflag &= ~CSIZE;
```

```
options.c_cflag |= CS8;
```

偶校验位 (7E1):

```
options.c_cflag |= PARENB
```

```
options.c_cflag &= ~PARODD
```

```
options.c_cflag &= ~CSTOPB
```

```
options.c_cflag &= ~CSIZE;
```

```
options.c_cflag |= CS7;
```

奇校验位 (7O1):

```
options.c_cflag |= PARENB
```

```
options.c_cflag |= PARODD
```

```
options.c_cflag &= ~CSTOPB
```

```
options.c_cflag &= ~CSIZE;
```

```
options.c_cflag |= CS7;
```

(5) 设置硬件流控

某些版本的 UNIX/Linux 支持使用 CTS 和 RTS 信号进行硬件流控。如果在系统中定义了 CNEW_RTSCCTS 或 CRTSCCTS 常数,那么系统就支持硬件流控。如下所示就可以使能硬件流控:

```
options.c_cflag |= CNEW_RTSCCTS; /* 同时也调用 CRTSCCTS */
```

类似的,取消硬件流控,如下所示:

```
options.c_cflag &= ~CNEW_RTSCCTS;
```

(6) 本地参数

在本地模式中,c_lflag 控制了输入字符如何被串口驱动程序管理的方式。一般而言,可以设置 c_lflag 标志,如下所示:

ISIG—使能 SIGINTR、SIGSUSP、SIGDSUSP 及 SIGQUIT 信号

ICANON—使能规范输入

XCASE—映射大写/小写

ECHO—使能输入字符的显示

ECHOE—显示对 BS-SP-BS 的擦除

ECHOK—清除字符后显示 NL

ECHONL—显示 NL

NOFLSH—使输入或关闭字符后对输入缓冲区的清除功能失效

IEXTEN—使能扩展功能

ECHOCTL—显示控制字符

ECHOPRT—在字符被清除的同时,将其显示出来

(7) 选择常规输入

在此方式下,输入字符被放入一个缓冲区,用户可以交互式地对该缓冲区进行编辑,直至接收到一个 CR 或 LF 为止。

如果要挑选该模式,则使用 ICANON、ECHO 及 ECHOE 参数,如下所示:

```
options.c_lflag |= (ICANON | ECHO | ECHOE);
```

(8) 选择原始输入法

在原始输入法中,对于接收到的数据不做任何处理,就输入缓冲区。通常,可以使用 ICANON、ECHO、ECHOE 和 SIG 参数来实现原始输入法,如下所示:

```
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
```

(9) 输入参数

输入模式成员 c_lflag 控制了端口接收到的所有字符的输入处理过程。与 c_cflag 域一样,在 c_lflag 中最终存储的值是所选参数的按位。

(10) 设置输入奇偶参数

在 c_cflag 的成员(PARENB)中使能了奇偶校验位后,应该使能输入奇偶校验。输

使用本文档品
请尊重相关知识产权!

入奇偶校验的相关常数是:INPCK、IGNPAR、PARMRK 和 ISTRIP。通常仅需选择 INPCK 和 STRIP 来使能校验奇偶位。

```
options.c_iflag |= (INPCK | ISTRIP);
```

IGNPAR 是一个危险的参数,它告诉串行口驱动程序忽略奇偶错误,并不对输入数据进行任何校验,好像此时没有错误发生。在测试通信连接的质量时该参数有用,但在实用上并无用处。

PARMRK 参数使奇偶校验错误在输入流中被标记出来。如果 IGNPAR 被使能,在每个带有奇偶错误的字符前都会加上一个 NUL 符号。否则,一个 DEL 就会与 NULL 一起发送。

(11) 设置软件流控

通过使用 IXON、IXOFF 和 IXANY 常数可实现软件流控,如下所示:

```
options.c_iflag |= (IXON | IXOFF | IXANY);
```

要禁止软件流控则可以屏蔽掉以下这些位:

```
options.c_iflag &= ~(IXON | IXOFF | IXANY);
```

在 `c_cc` 成员中,定义了起始数据位(XON)和停止数据位(XOFF)。

(12) 输出参数

`c_oflag` 成员包括输出过滤参数。与输入模式一样,可以选择已经处理过的数据输出,或者选择原始数据的输出。

其参数摘要如下:

OPOST—处理数据后再输出

OLCUC—将小写映射为大写

ONLCR—将 NL 映射为 CR-NL

OCRNL—将 CR 映射为 NL

(13) 选择处理后的数据输出

在 `c_oflag` 成员中设置 OPOST 参数,可以选择将数据处理后输出。

```
options.c_oflag |= OPOST;
```

在所有参数中,可能会用到 ONLCR 参数,该参数将新输出行(NL)映射成形式为 CR-LF 的输出。

(14) 选择原始数据输出

在 `c_oflag` 成员中将 OPOST 参数置位,可以选择原始数据的输出。

```
options.c_oflag &= ~OPOST;
```

当禁用 OPOST 位时,在 `c_oflag` 中的所有的其他参数位都被忽略。

(15) 控制字符

`c_cc` 字符数组包含了控制字符的定义和超时参数。所有定义的常数如表 7-3 所示。

表 7-3 c_cc 控制字符

c_cc 域中的控制字符	功能描述	对应按键
VINTR	中断	Ctrl-C
VQUIT	关闭	Ctrl-Z
VERASE	清除回退空格	BS
VKILL	清除行	Ctrl-U
VEOF	文件结束符号	Ctrl-D
VEOL	行结束返回	CR
VEOL2	二行结束返回	LF
VMIN	将要读字符的最小数目	
VTIME	等待数据的时间	


(16) 设置软件流控字符

c_cc 数组的 VSTART 和 VSTOP 元素包含了用来实现软件流控的字符。通常它们被设置为 DC1(八进制的 021)和 DC3(八进制的 023),即表示 ASCII 码中标准的 XON 和 XOFF 字符。

(17) 设置读超时时间

UNIX/Linux 串行接口驱动程序提供了指定字符和包的超时功能。在 c_cc 数组中,有两个元素用来设置超时时间:VMIN 和 VTIME。在常规的输入模式下,或使用 open 和 fcntl 函数打开文件并设置了 NDELAY 时,超时操作会被忽略。

VMIN 指定了所要读取字符的最小数量。如果它被设置成 0,那么 VTIME 参数就指定了对于每个将要读取字符的等待时间。超时操作适用于第一个字符,read 调用会返回现存字符的数目。

 注意:这并不意味着一个读取 N 个字节数据的 read 调用要等待 N 个字节的输入。

如果 VMIN 为非零,则 VTIME 指定了读取第一个字符的等待时间。如果某字符在给定的时间内读取了,所有的 read 操作都会阻塞,直到所有的 VMIN 字符都被读取为止。也就是讲,一旦首字符被读取后,串行口驱动程序就希望接收一整包的字符(总数为 VMIN)。如果在允许的时间内没有读取数据,那么 read 调用返回 0。该方法允许用户告诉串行口驱动程序只需要 N 字节的数据。任何 read 调用都返回 0 或者 1。然而,超时操作仅仅适用于所读取的第一个字符,所以,如果因为某种原因驱动程序将 N 字节数据包的首字符搞丢了,那么 read 调用就会永远阻塞,等待其他输入字符的到来。

VTIME 指定了等待输入字符的时间。如果 VTIME 被设置为 0,那么 read 操作会无限期地阻塞,直到使用 open 或 fcntl 函数将端口的 NDELAY 参数置位为止。



7.2.3 MODEM 的通信

本小节讨论 MODEM 通信的基本问题及 AT 指令集。如果想使用开发板接 MODEM,就应该认真阅读此小节。

1. MODEM 的功能

MODEM 是将串行数据调制为频率,使数据可在诸如电话线、有线电视电缆等模拟数据线路上传输的一种设备。标准的电话 MODEM 将串行数据转换为模拟信号(如声音),使其能在电话线路上传递。

电话用 MODEM 今天到处可见,它能以 53 000 bit/秒的速率传输数据。由于许多的 MODEM 还使用了数据压缩技术,所以能以近 100K 的速率传送数据。

2. 使用 MODEM 通信

使用 MODEM 通信的第一步是打开并设置原始数据输入,如例 7-3 所示。

例 7-3 打开 MODEM

```
int fd;
struct termios options;

/* 打开端口 */
fd = open("/dev/ttyf1", O_RDWR | O_NOCTTY | O_NDELAY);
fcntl(fd, F_SETFL, 0);

/* 获取当前参数 */
tcgetattr(fd, &options);

/* 设置原始输入,超时时间为 1 秒 */
options.c_cflag |= (CLOCAL | CREAD);
options.c_lflag &= ~(ICANON | ECHO | ECHOE | ISIG);
options.c_oflag &= ~OPOST;
options.c_cc[VMIN] = 0;
options.c_cc[VTIME] = 10;

/* 设置参数 */
tcsetattr(fd, TCSANOW, &options);
```

下面就需要同 MODEM 建立通信连接了。最好的方法是向 MODEM 发送“AT”命

令。这也使 MODEM 智能化地检测到了用户使用的波特率。此时,MODEM 会回应 OK。见例 7-4。

例 7-4 初始化 MODEM

```
int          /* 0 = MODEM ok, -1 = MODEM bad */
init_modem(int fd) /* I - 串行口文件 */
{
    char buffer[255]; /* 输入缓冲区 */
    char * bufptr;    /* 在缓冲区的当前字符 */
    int nbytes;      /* 所读字节数 */
    int tries;       /* 尝试最大次数 */

    for (tries = 0; tries < 3; tries++)
    {
        /* 发送一个 AT 命令,后面跟随一个 CR 字符 */
        if (write(fd, "AT\r", 3) < 3)
            continue;

        /* 将字符读入字符串缓冲区,直到接收到 CR 或 NL 为止 */
        bufptr = buffer;
        while ((nbytes = read(fd, bufptr, buffer + sizeof(buffer) - bufptr - 1)) > 0)
        {
            bufptr += nbytes;
            if (bufptr[-1] == '\n' || bufptr[-1] == '\r')
                break;
        }

        *bufptr = '\0';

        if (strncmp(buffer, "OK", 2) == 0)
            return (0);
    }

    return (-1);
}
```

3. 标准的 MODEM 命令

绝大多数 MODEM 使用 AT 命令集,之所以这样称呼,是因为每个命令都用 AT 字符

开头。在处理完信息后,MODEM 会返回对应的文本信息。具体的内容见相关书籍。



7.2.4 串行编程进阶

下面深入探讨使用 `ioctl()` 和 `select()` 函数进行串行编程的方法。

1. 串行口的 IOCTL

在上面的部分中,使用了 `tegetattr` 和 `tcsetattr` 函数来设置串行口。这些函数都是使用 `ioctl()` 函数来实现其功能的。`ioctl()` 函数使用了如下三种参数:

```
int ioctl(int fd, int request, ...);
```

`fd` 参数指定了串行口文件描述符。`request` 参数是一个在 `<termios.h>` 头文件中定义的常数,典型的有这些:

TCGETS—获取当前串行口设置,对应的 POSIX 函数为 `tegetattr`

TCSETS—立即设置串行口,对应的 POSIX 函数为 `tcsetattr(fd, TCSANOW, &options)`

TCSETSF—清空输入输出缓冲区后,设置串行口,对应的 POSIX 函数为 `tcsetattr(fd, TCSANOW, &options)`

TCSETSW—允许输入/输出缓冲区为空后,设置串行口。对应的 POSIX 函数为 `tcsetattr(fd, TCSANOW, &options)`

TCSBRK—在给定的时间之后,发送中断信号。对应的 POSIX 函数为 `csendbreak, tcdrain`

TCXONC—控制软件流控。对应的 POSIX 函数为 `tcflow`

TCFLSH—清空输入/输出队列。对应的 POSIX 函数为 `tcflush`

TIOCMGET—返回“MODEM”状态位

TIOCMSET—设置 MODEM 状态位

FIONREAD—返回输入缓冲区中的字节数目。

2. 获取控制信号

`ioctl` 函数从当前 MODEM 状态位获取了 `TIOCMGET` 的值,该值包括除 RXD 和 TXD 以外的所有 RS-232 信号线。

为了获取状态位,用一个指向整数的指针来调用 `ioctl`,以此保持位的值,见例 7-5。

例 7-5 获取 MODEM 状态位

```
#include <unistd.h>
```

```
#include <termios.h>
```

```
int fd;
```

```
int status;
```

```
ioctl(fd, TIOCMGET, &status); /* 设置 MODEM 状态位 */
```

设置控制信号

ioctl 在上面设置了 MODEM 状态位。为了将 DTR 信号复位,可以采用这样的方法:

```
#include <unistd.h>
#include <termios.h>

int fd;
int status;

ioctl(fd, TIOCMGET, &status);

status &= ~TIOCM_DTR; /* 将 DTR 复位 */
```

```
ioctl(fd, TIOCMSET, status);
```

操作系统、驱动程序和程序所使用的模块指明了哪些位可以进行设置。

3. 获取字节数目

ioctl 使用 FIONREAD 参数获取串行口输入缓冲区中的字节数目。与 TIOCMGET 一样,可以用一个指向整数的指针调用 ioctl,以返回数据的字节数。见例 7-6。

例 7-6 获取输入缓冲区的字节个数

```
#include <unistd.h>
#include <termios.h>

int fd;
int bytes;

ioctl(fd, FIONREAD, &bytes); /* 获取串行口输入缓冲区中字节数目 */
```

当轮询串行口获取数据时,在程序读取数据前,程序可以确定输入缓冲区内的数据个数。

4. 从串行口中挑选输入

虽然简单的应用程序可以轮询串行口来等待数据的输入,但绝大多数应用程序是比较复杂的,需要从多个来源中处理输入。

UNIX 使用了 select() 调用来提供该功能。该系统调用允许程序对一个或多个文件描述符进行输入、输出和错误情况的校验。文件描述符可以指向串行口、常规文件、其他设备、管道或者套接字。程序可以轮询数据,也可以无限制地等待输入。这就使 select()



系统调用非常灵活。

(1) select() 系统调用

select() 调用使用了 5 个参数：

```
int select(int max_fd, fd_set *input, fd_set *output, fd_set *error, struct timeval *timeout);
```

max_fd 参数指定了所有对应于输入文件描述符、输出文件描述符和错误代码文件描述符的最大值。这三个集合都使用了下面三个宏来初始化：

```
FD_ZERO(fd_set);
```

```
FD_SET(fd, fd_set);
```

```
FD_CLR(fd, fd_set);
```

FD_ZERO 宏清除了整个指令集。FD_SET 和 FD_CLR 分别在指令集中增加或删除一个文件描述符。

(2) 使用 select() 系统调用

假定正在从串行口和套接字中读取数据，对两个文件描述符都想进行数据输入校验。但是还想通知用户，如果在 10 秒钟内还没有读取到数据，就发出警告。为了达到该目的，应使用 select 系统调用，如例 7-7 所示。

例 7-7 使用 select() 系统调用处理多个数据源的数据输入

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/select.h>

int n;
int socket;
int fd;
int max_fd;

fd_set input;

struct timeval timeout;

/* 初始化输入集 */
FD_ZERO(input);
FD_SET(fd, input);
FD_SET(socket, input);

max_fd = (socket > fd ? socket : fd) + 1;
```

```
/* 初始化超时数据结构 */
timeout.tv_sec = 10;
timeout.tv_usec = 0;

/* 进行选择 */
n = select(max_fd, NULL, NULL);

/* 看看是否有错误 */
if (n < 0)
    perror("select failed");
else if (n == 0)
    puts("TIMEOUT");
else
{
    /* 获取输入 */
    if (FD_ISSET(fd, input))
        process_fd();
    if (FD_ISSET(socket, input))
        process_socket();
}
```



用户注意到,首先检验了 select 系统调用的返回值。值 0 和 -1 生成了合适的警告和错误信息。如果返回值大于 0,那么必定有一个或多个描述符对应的设备在等待数据的输入。

要发现是哪一个文件描述符在等待输入,可使用 FD_ISSET 宏测试每一个文件描述符对应的输入集。如果文件描述符标准位已经被置位,那么该文件描述符就在等待输入数据。

7.3 串行通信的实例

学习完 UNIX/Linux 串行通信编程的实例后,再学习在开发板上通过使用串行口,实现通信的过程。

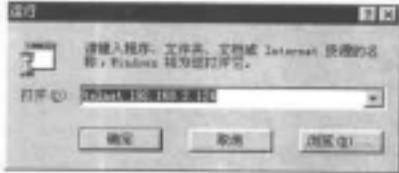
1. 程序的执行

该样例程序可以从本书附带的光盘中找到。程序可以分为两个部分:开发板上的发送程序和 PC 机上的接收程序。在此仅讲述开发板上的发送部分。

如在第 5 章中讲述的,可以先在 /uclinux/appsrc 目录下建立一个名为 /serial 的目录。在

该目录下,放置样例程序,包括源程序 serial.c,makfile 和要发送的文件 test.bmp。再键入:
make

在目录/uclinux/appsrc/serial 下就会生成可执行文件 serial,将它和文件 test.bmp 拷贝到目录/uclinux/romdisk/romdisk/bin 下,然后退回到/uclinux/appsrc 目录下,键入:



中的闪存。

反,进入到/bin 目录,就会发现其中多了两个文件 serial 和 test.bmp,然后退出 Linux,到 Windows 下,编译接收程序。成功后,则按如下步骤执行应用程序:

(1)进入 ms-dos 环境,键入:
TransBmpSerial 28800



接收程序的名称,28800 是 PC 机的接收代码的波特率。开发板,如图 7-3 所示。

图 7-3 启动 telnet,登录开发板

(3)登录后,进入目录/.bin,运行发送程序:

如图 7-4 所示键入:

serial 57600

即可运行发送程序。57600 为发送波特率。



注意:开发板的波特率仅能达到其设定的一半,所以要保持收发的平衡。在发送方必须使用 57600 的波特率,而且波特率的值都为标准的波特率值,如 2400、1800、

1200 等。

然后就可以看到发送的过程,如图 7-5 所示。在发送数据时,开发板不断地向串行口打印各种信息。在下面的小节中,将讲述它们代表的意义。

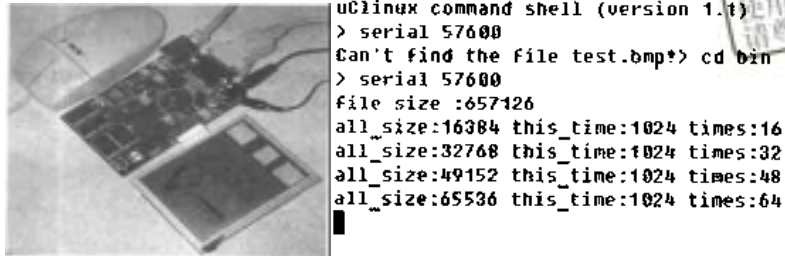


图 7-5 发送数据过程

(4)接收完毕,在 PC 机上就会弹出一个窗口,显示接收结果,如图 7-6 所示。

图 7-6 接收结果

由于接收结果文件较大,在实验时要耐心等待。

2. 程序分析

发送方程序如例 7-8 所示。其功能是由目标板传送一个 BMP(test.bmp)文件到一台执行了接收程序的 Windows PC 机上。

例 7-8 串行口通信程序(发送方)

```
/* This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */

#include <stdio.h> /* 标准输入/出定义 */
```

```
#include <string.h> /* 字符串函数定义 */
#include <unistd.h> /* UNIX 标准函数定义 */
#include <fcntl.h> /* 文件控制定义 */
#include <errno.h> /* 错误代码定义 */
#include <termios.h> /* POSIX 终端控制定义 */
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>

// #define BAUDRATE B300
// #define BAUDRATE B9600
// #define BAUDRATE B57600

#define MODEMDEVICE "/dev/ttyS0" /* 定义串行口设备,为/dev/ttyS0 */
#define _POSIX_SOURCE 1

#define FALSE 0
#define TRUE 1

volatile int STOP = FALSE;

int cnvbaudrate(char * pszbaudrate) /* 转换波特率函数,结果供函数 cfsetispeed 调用 */
{
    switch(atoi(pszbaudrate))
    {
        case 50:
            return B50;
        case 75:
            return B75;
        case 110:
            return B110;
        case 134:
            return B134;
        case 150:
            return B150;
        case 200:
            return B200;
    }
}
```

请尊重浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```
        return B200;
    case 300:
        return B300;
    case 600:
        return B600;
    case 1200:
        return B1200;
    case 1800:
        return B1800;
    case 2400:
        return B2400;
    case 4800:
        return B4800;
    case 9600:
        return B9600;
    case 19200:
        return B19200;
    case 38400:
        return B38400;
    case 57600:
        return B57600;
    default:
        return B57600;
}
}
```

```
int main(int argc, char ** argv)
{
    /* 初始化定义指针、端口及文件描述符 */
    int fd, c, res, nsize = 1024, allsize = 0, BAUDRATE;
    /* 定义终端数据结构 */
    struct termios oldtio, newtio;
    char szsendbuf[1024]; /* szsendbuf[]为发送数据缓冲区,1K大小 */
    int filclength;
    FILE * pfile; /* pfile 指向发送的文件 */
    struct stat st;

    fd = open(MODEMDEVICE, O_RDWR | O_NOCTTY); /* 打开串口 */
```




```

/* O NOCTTY 表示,该程序不想成为该端口的“控制终端” */
if (fd < 0)
{
    perror(MODEMDEVICE);
    printf("Error in open COM1 \n");
    exit(-1);

    argentr(fd, &oldtio); /* 在 oldtio 中保存原串口属性 */
    bzero(&newtio, sizeof(newtio)); /* 为新的串口属性设置做准备 */
    /* 串口新属性设置 */
    /*newtio.c_cflag = BAUDRATE | /* CRTSCTS | /* CS8; // | CLOCAL |
CREAD; /* 设置控制模式 */
    BAUDRATE = B57600; /* 设置波特率 */
    //printf("argc = %d, argv[1] = %s \n", argc, argv[1]);
    if(argc >= 2)
        BAUDRATE = cnvbaudrate(argv[1]);

    cfsetispeed(&newtio, BAUDRATE); /* 将波特率填入串口输入端。该波特率由
        * BAUDRATE 定义 */

    cfsetospeed(&newtio, BAUDRATE); /* 将波特率填入串口输出端 */

    newtio.c_cflag |= CRTSCTS | CS8 | CLOCAL | CREAD; /* 为端口设置控制模式 */
    newtio.c_iflag = IGNPAR | ICRNL; /* 为新端口设置输入模式, c_iflag 控制了对于在端口接收到的所有字符的输入处理过程。IGNPAR 表示忽略帧错误、奇偶错误。将返回的包翻译成新行在屏幕上显示 */。

    //newtio.c_iflag = ICANON; // | ECHO; /* 为端口设置输出模式 */

    /* 为串口通信设置控制键,见表 7-3 */
    newtio.c_cc[VINTR] = 0; /* Ctrl - c */
    newtio.c_cc[VQUIT] = 0; /* Ctrl - \ */
    newtio.c_cc[VERASE] = 0; /* del */
    newtio.c_cc[VKILL] = 0; /* @ */

```



```

newtio.c_cc[VEOF] = 4; /* Ctrl-d */
newtio.c_cc[VTIME] = 0;
newtio.c_cc[VMIN] = 1; /* 读取数据阻塞,直到有一个字符到来 */
newtio.c_cc[VSWTC] = 0; /* '\0' */
newtio.c_cc[VSTART] = 0; /* Ctrl-q */
newtio.c_cc[VSTOP] = 0; /* Ctrl-s */
newtio.c_cc[VSUSP] = 0; /* Ctrl-z */
newtio.c_cc[VEOL] = 0; /* '\0' */
newtio.c_cc[VREPRINT] = 0; /* Ctrl-r */
newtio.c_cc[VDISCARD] = 0; /* Ctrl-u */
newtio.c_cc[VWERASE] = 0; /* Ctrl-w */
newtio.c_cc[VLNEXT] = 0; /* Ctrl-v */
newtio.c_cc[VEOL2] = 0; /* '\0' */

tcflush(fd, TCIFLUSH); /* TCIFLUSH 表示溢出的数据可以接收,但不读 */
tcsetattr(fd, TCSANOW, &newtio); /* 立即将新属性赋予串口 */
stat("test.bmp", &st); /* 返回文件的状态属性 */
if((pfile = fopen("test.bmp", "r")) == NULL) /* 打开位图文件 */
{
    printf("Can't find the file test.bmp!");
    return 0;
}
filelength = st.st_size; /* 获得文件长度信息 */
printf("file size : %d \n", filelength); /* 打印文件长度 */
//write(fd, (char *)&filelength, sizeof(int));
while (nsize == 1024) /* 向串口循环写入文件 */
{
    bzero(szsendbuf, 1024);
    nsize = fread(szsendbuf, 1, 1024, pfile);
    nsize = write(fd, szsendbuf, nsize);
    allsize += nsize; /* 统计向串口发送文件长度 */
    if(! (allsize & 0x3fff)) /* 当发送文件长度 > 16k, 打印统计结果 */

```



```

        printf("all _ size: %d this _ time: %d times: %d \n", allsize, nsize, allsize/1024);
    }
    fclose(pfile); /* 关闭位图文件 */
    /* 立即恢复原串口属性设置 */
    tcsetattr(fd, TCSANOW, &oldtio);
    close(fd); /* 关闭串口 */
}

```

在上例中使用了 struct stat 型数据结构。如：

```
struct stat st
```

在 Linux 操作系统中,所有文件和设备都有一个索引节点 inode(即 index node)与之相对应,通过索引节点可以来访问它们。索引节点包括了关于文件的很多的信息,如文件类型、文件容量等。为了访问它们,可供进程使用的这些属性都被拷贝到 stat 结构中。

在该结构中有很多域,本程序中用到了:

st.st_size——表示以字节为单位的文件容量

该值被赋予 filelength 变量,在发送数据时被打印出来。

再来看看函数调用 stat。它的原型为:

```
int stat(char * pathname, stat * sbuf)
```

它得到了 inode 结构中的信息。pathname 表示的是文件名,sbuf 表示的是 inode 信息所存放的缓冲区。在本程序中的调用是:

```
stat("test.bmp", &st);
```

这就把 test.bmp 文件的属性存放在 st 中。

该程序的思路是很清楚的。首先,打开串行口,fd 为其文件描述符。对串行口的所有操作就变成了对文件描述符 fd 的操作。然后,调用 tcgetattr()函数,将串行口的属性取出,并存放在 oldtio 结构中。再将 newtio 结构内内容清除干净,并将新的串行口属性赋予它。最后,调用函数 stat(),打开要发送的 test.bmp 文件,用 pfile 指针指向它。

本程序的流程图如图 7-7 所示。

在例 7-8 中,

```

        if(! (allsize&0x3fff))
        printf("all _ size: %d this _ time: %d times: %d \n", allsize, nsize, allsize/1024);|

```

一句是用来在输出数据过程中打印发送的结果的。1K 的数据在二进制中的表示是 10000000000,在十六进制中是 400,那么,0x3fff 就相当于 $400 * 10 - 1$ (这个 10 是十六进制的 10,相当于十进制的 16)。由于 allsize 是一 K 一 K 往上面加的,当加到 16K 的倍数时,例如,16K(十六进制的 4000)时,有:

$!(4000 \& 0x3fff) = 1$.

此时,就可以打印发送信息了。

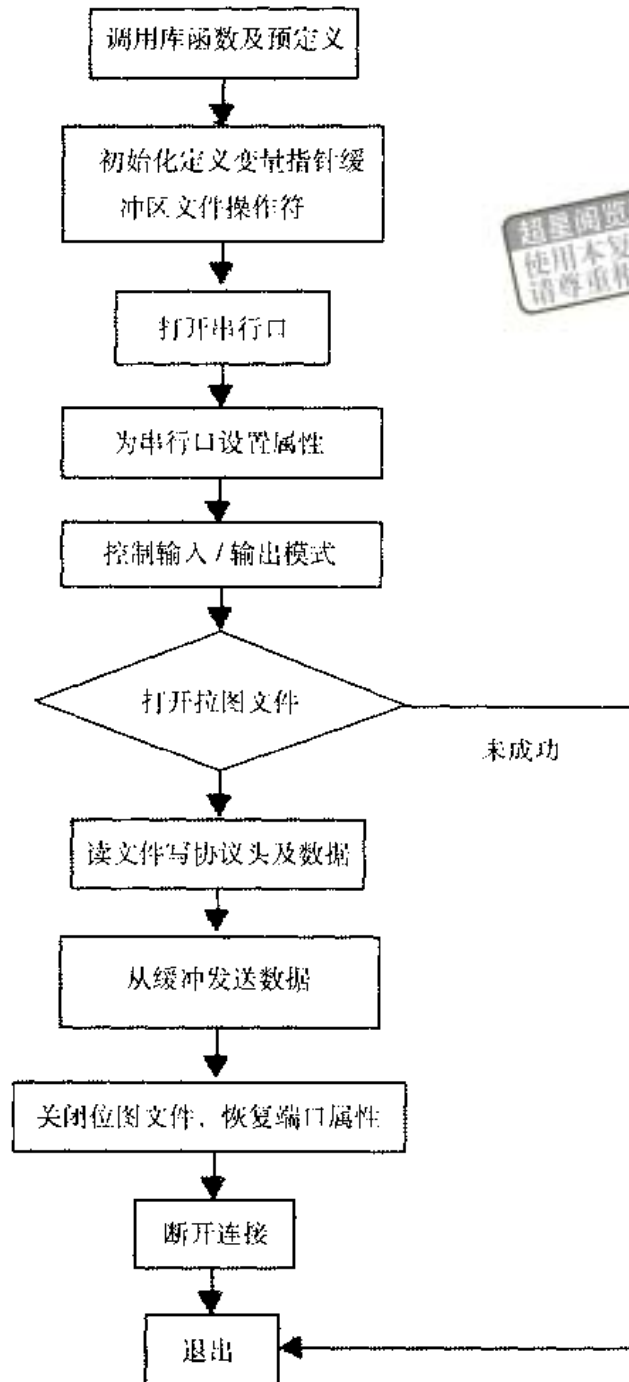


图 7-7 串行口通信流程图

7.4 本章小结

本章详细讲述了嵌入式 Linux 环境下,如何进行串行口通信的问题。从本章可以看出,在嵌入式 Linux 环境下,其串行口编程的实现同 UNIX/Linux 系统下的方法是基本相同的,但是还要注意其在细节上的不同之处。

本章先讲述了总线的种类,并以 RS-232 为标准,重点讲述了串行口的物理知识。然后学习了在 UNIX/Linux 环境下串行编程的基本知识。通过一个在开发板的 uClinux 环境下的数据发送/接收的样例程序,巩固了所学到的知识。认真研读该程序,就能对串行口编程有一个基本的认识,并可以以此为起点,开发出自己的应用程序来。

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

第 8 章 嵌入式 Linux 系统的 键盘和 LCD

在嵌入式系统的应用中,为系统扩充外部设备一直是用户所关注的问题。键盘、LCD 屏幕以及手写板等都是相当重要的外围设备。通过本章的学习,将学到如何为嵌入式 Linux 系统添加小键盘,如何利用 LCD 显示数据和使用手写板,以及如何用手写板控制系统。

本章主要内容:

- 嵌入式系统所用到的键盘和 LCD
- 为 MC68EZ328 接上小键盘
- MC68EZ328 对 LCD 的支持
- 在 LCD 上显示数据
- 样例程序——手写板
- 用手写板来控制系统

8.1 嵌入式系统所用到的键盘和 LCD

伴随着信息家电、手持设备、无线设备等的迅速发展,相应的硬件和软件也得到迅速发展。许多设备都配有 Intel、MIPS、Motorola 等公司生产的 32 位微处理器,甚至还使用了液晶显示器。在以 MCU 为核心的许多应用程序中,都需要键盘的输入功能以实现人机接口。一般而言,硬件上的实现方法有两种:第一种较为简单,直接使用一路 I/O 口,每个引脚引一个按键,在程序中使用轮询的方式判断是否有键按下,然后就具体的按键执行对应的功能。还有一种方法是采用阵列式按键键盘和 MCU 的一个带有键盘中断的并行 I/O 口组成,其原理如图 8-1 所示。

对于嵌入式系统而言,还可以使用 RS-232 串行口加上串行口控制芯片 8250 来扩充键盘。如图 8-2 所示。电路工作时,TXD、RXD 和 RTS 三条线由软件轮流设置为 +12V。任何时候,RTS 和 DTR 至少有一条线为低电平,通过 R1~R4 这 4 个下拉电阻,使在没有键被按下的情况下,CTS、DSR、RI 和 CD 均为低电平。如果有键按下,则 8250 中的 MODEM 状态寄存器的相应位将被置 1。在键盘扫描的程序中,CPU 查询 8250 的 MSR 寄存器,就可以判断是否有键按下以及是哪一个键被按下。

除了键盘以外,LCD 也是一个非常重要的人—机接口。而带有手写输入的 LCD 则更是由于其易用性和良好交互性,深得用户的喜爱和研发人员的重视。在本章,会重点讨

论基于 LCD 的显示和手写输入的编程方法。

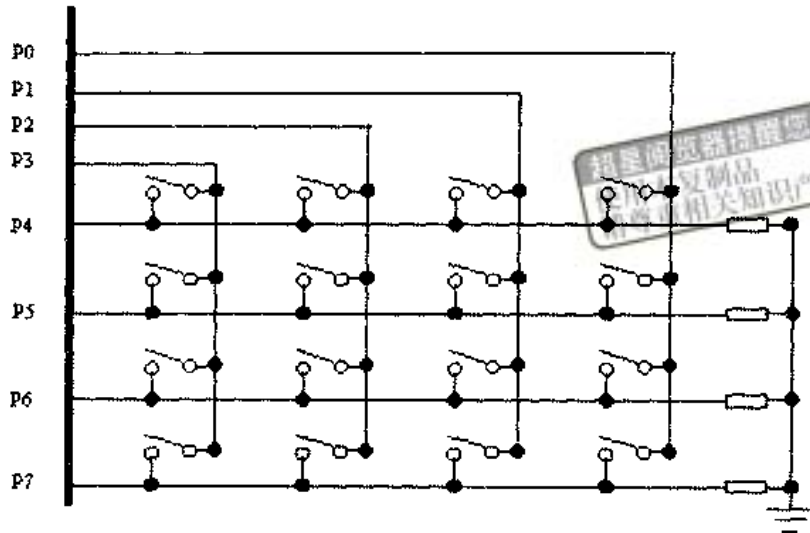


图 8-1 阵列式键盘组成

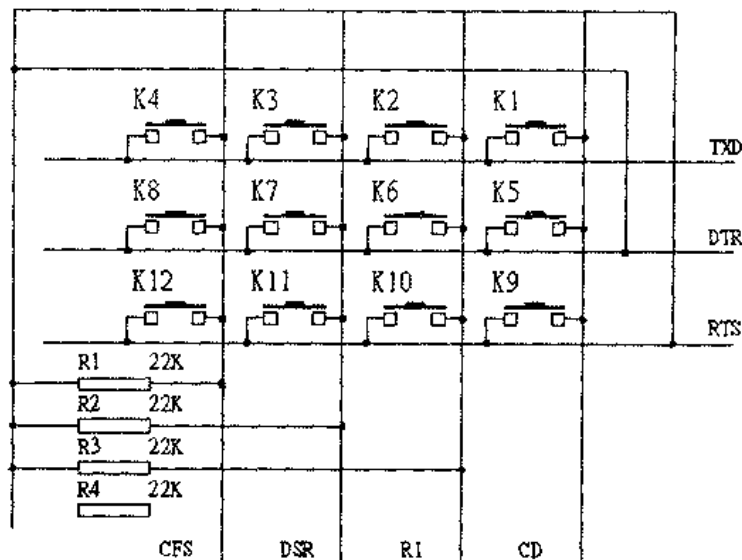


图 8-2 使用 RS-232 口接小键盘

8.2 为嵌入式系统接上小键盘实例

为嵌入式系统扩展外部设备是很重要的。本小节以开发板的 MC68EZ328 型 CPU 为例,讲述如何为嵌入式系统添加小键盘的方法和编程的步骤。

1. 简介

开发板使用的 CPU 是 MC68EZ328,那么,为系统添加小键盘的操作就和该 CPU 密

切相关。下面就来看看如何最少地利用 EZ328 的 I/O 端口,为矩形小键盘提供接口的方法和步骤。为此看一个实例:将一个 4×4 的矩阵式键盘同 CPU 相连。该键盘仅仅需要 5 个 I/O 口(一般而言,对于 $N \times N$ 的键盘仅仅需要 $N+1$ 个 I/O 口)。这是一个廉价的解决方案,并没有使用 TTL 的逻辑集成电路。在接口电路中主要使用了电阻和二极管等,可以极大地减小系统的成本和体积。

2. 硬件组成

键盘接口的功能模块图如图 8-3 所示,它由两个主要部分组成。

- (1) 中断和接口电路:当有键被按下时,产生中断并提供同 EZ328 的 I/O 口的连接。
- (2) 键盘阵列:一个 4×4 的矩阵键盘。

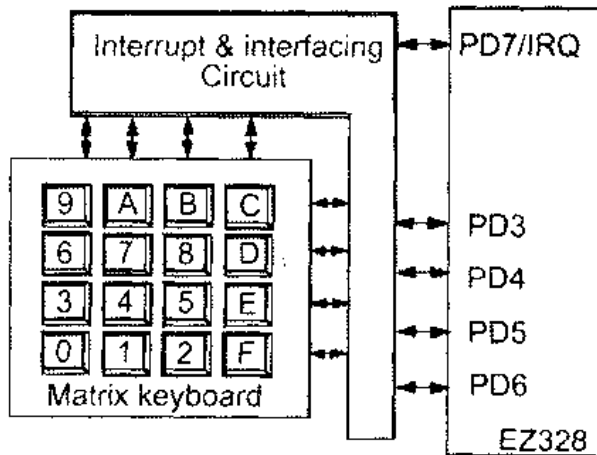


图 8-3 MC68328 连接小键盘图解

3. 中断和接口电路

中断和接口电路包括了一些二极管、电阻、上拉寄存器和一个 NPN 型晶体管。该晶体管被设计成一个转换器,其作用是当某个按键被按下时,产生中断。二极管主要有两组,其作用是控制单向信号流,以使该电路可以唯一地鉴别出被下压的按键。该两组二极管中的一组经过布线设计,可以提供 OR 功能,同时使在键盘的每个列中的任意键被按下时,可以给晶体管部分发送信号,让其生成一个中断信号。

4. 并行端口

本例的目的是使用最少的 I/O 端口来构建一个简单键盘,所采用的键盘为一个 4×4 的矩阵键盘,它仅需要 5 个 I/O 口作为其接口。其中的某一个端口在系统初始化时被用来产生中断,其后,用做键盘扫描操作的 I/O 端口。

☞ 注意:在该设计中,EZ328 使用了 5 个端口作为接口,但是仅仅其中一个端口需要中断功能。

5. 键盘扫描操作

该系统中使用了 5 个端口作为键盘扫描用。当某个键被按下时,其中一个端口

(PD7)用做中断脚。在键盘扫描过程开始前,除了带有中断功能的那个端口以外,所有的 I/O 端口都被设置成为输出高电平。当有键被按下时,键盘对应列的状态由低电平转向了高电平。由于所有的列都被连到一起构成了一个 OR 逻辑(对于 NPN 型晶体管而言),就会产生一个低电平有效中断信号给 EZ328,并初始化中断处理函数来扫描这个被按下的按键。当按键扫描程序开始时,5 个 I/O 端口中的一个端口将被设置为输出高电平,而其他的端口则被设置为输入。然后,读入所有端口的状态,并与先前存储在系统中的一个表中的模式相比较,以便定位按下的键。如果该键未被找到,那么,就将另外一个端口设置为输出高电平,并再读入端口状态。这样循环下去,直到所有端口都曾经被设置成输出高电平至少一次,或者发现该按下的键为止。

由于电路提供了反馈路径,所以,输出高电平的端口就会使某个输入端口由低电平状态转向高电平状态,而且系统获得的状态就可以唯一地判断出是哪个按键被按下。一个扫描过程的例子如下表所示。

PD3	PD4	PD5	PD6	PD7(IRQ)	描 述
OH	OH	OH	OH	IRQ	所有的端口都被设置为输出高电平。PD7 口被设置为中断端口,当有一个低电平有效的中断到来时被中断,
H	H	H	H	L	假定某个键被按下,如“8”。晶体管的基极电压会改变状态,由默认的低电压转换为高电压
OH	I	I	I	I	第一个端口(PD3)设置为输出高电平,而其他端口都变为输入
H	L	L	L	H	当读取数据寄存器时,得到 10001,但是没有任何键和该模式相符合,所以没有找到该按键
I	OH	I	I	I	将第二个端口(PD4)设置为输出高电平,而其他端口被设置为输入
L	H	H	L	L	由于端口 PD4、PD5 连接到了按键“8”,这两个按键都为高电平状态。此时,可以从数据寄存器中读取到模式“00110”,它就唯一地表示按键“8”

O—输出 H—高电平 I—输入 L—低电平

注意:有些按键的两端都同时接到了同一个端口上。该键仍然可以被检测出来,在这种情况下,当其他引脚为低电平时,中断引脚也为低电平。

6. 软件部分

软件部分是用 C 语言编写的,并使用了标准的 SDS 库。该部分使用了 Fputc.c 来实现在用户使用调试工具时交互地输入和输出数据。

初始化部分初始化了 D 端口的方向寄存器、数据寄存器、选择寄存器、中断等待寄存器和中断屏蔽寄存器,这样,它就可以从键盘接收中断了。在中断服务例程中,端口的一个引脚被设置为输出,其余的被设置为输入,这样就可以检测出所按下的键处在哪一行。然后,读入 D 端口寄存器的值,并将其同系统中保留的查询表中的各个字符的期望值进行比较,以找出哪个键被按下。键盘接口设置流程图和键盘中断服务例程的流程图分别如图 8-4 和 8-5 所示。

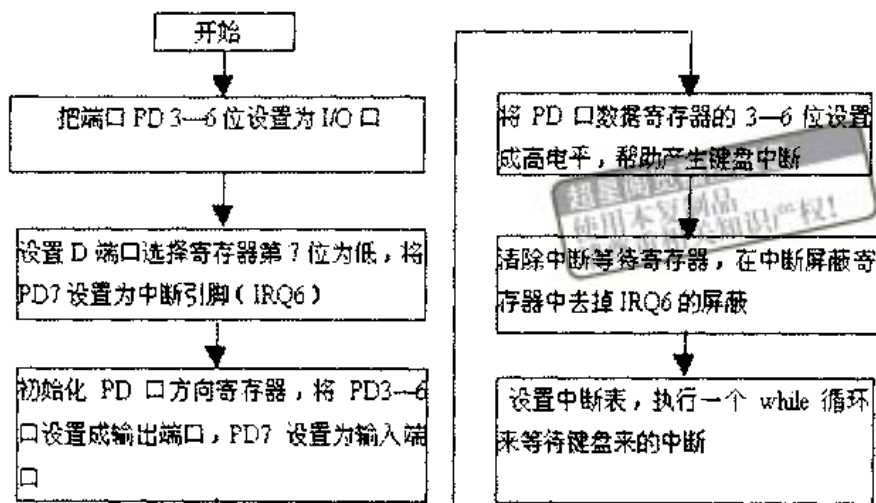


图 8-4 键盘接口设置流程图

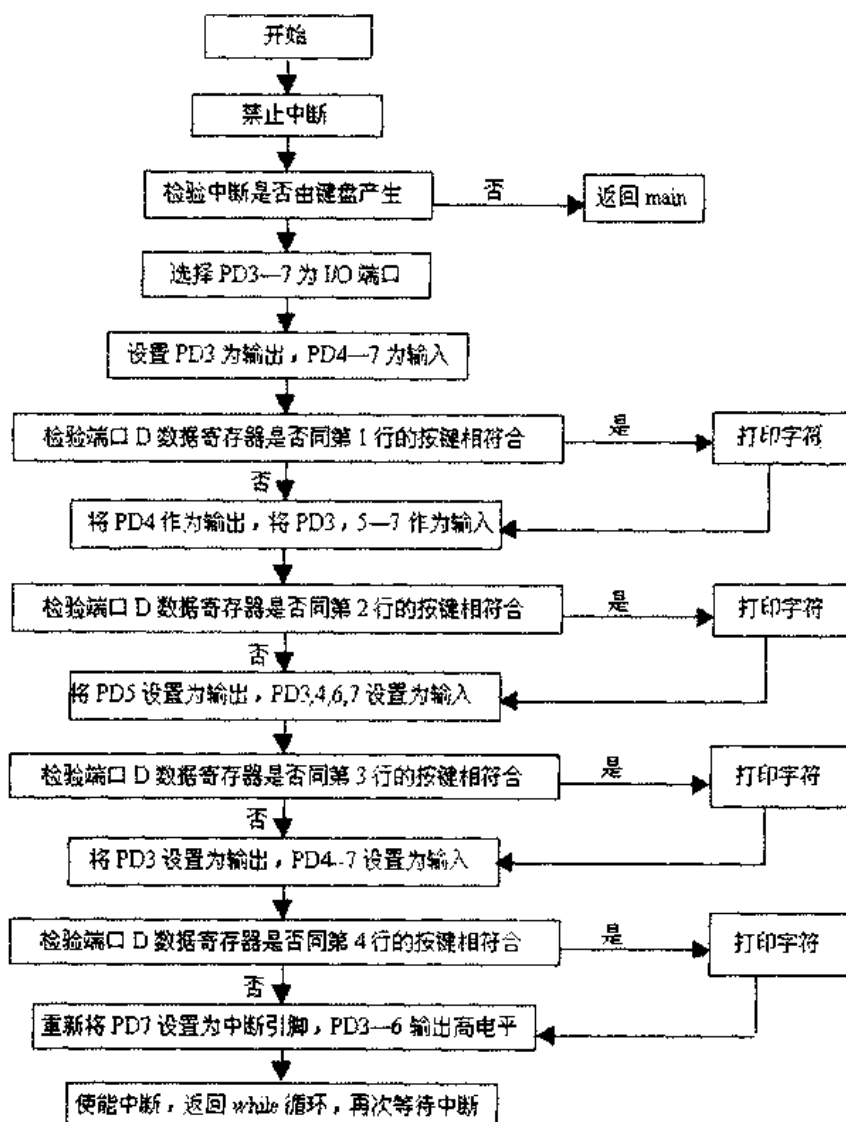


图 8-5 中断服务例程流程图

7. 源代码部分

例 8-1 小键盘中断服务例程

```

#include <stdio.h>

typedef unsigned char U8; /* 无符号 8 位数 */
typedef unsigned int U32; /* 无符号 32 位数 */
typedef U8 * P_U8; /* 无符号 8 位数 */
typedef U32 * P_U32; /* 无符号 32 位数 */

#define M328_IVR 0xffff300 /* 中断向量寄存器 */
#define M328_IMR 0xffff304 /* 中断屏蔽寄存器 */
#define M328_ISR 0xffff30C /* 中断状态寄存器 */
#define M328_IPR 0xffff310 /* 中断等待寄存器 */
#define M328_PDDIR 0xffff418 /* Port D 方向寄存器 */
#define M328_PDDATA 0xffff419 /* Port D 数据寄存器 */
#define M328_PDPHEN 0xffff41A /* Port D 上拉使能寄存器 */
#define M328_PDSEL 0xffff41B /* Port D 选择寄存器 */
#define M328_PDIRQEN 0xffff41D /* Port D IRQ 使能寄存器 */
#define LEVEL6V ((volatile long *) 0x118) /* 第 6 等级中断的向量偏移位置 */

U8 data;
U32 tmp, pendReg, count = 12000;

void delay(U32 time) /* 延时循环 */
{
    while(time--);
}

void KeyIRQ(void)
{
    asm(" ORI.W # $0700,SR"); /* 禁止所有的中断。该句使用了内联汇编 */
                               /* ORI 表示立即数或,SR 为状态寄存器。 */
    pendReg = *(P_U32)M328_IPR;
    pendReg &= ~(0x00f7ffff); /* 屏蔽其他位,除了 IRQ6 */

    if(pendReg &= 0x80000) /* 检验该中断是否来自键盘,即看地址 ffff310 的 IRQ6
    是否为 1 */
    {
        *(P_U8)M328_PDDIR = 0x08; /* 选择 PD3 为输出,PD4-7 为输入 */
    }
}

```

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

```
delay(count);                /* 延时 */
*(P_U8)M328_PDSSEL |= 0xf8; /* 将 PD3-7 口作为 I/O 口 */
*(P_U8)M328_PDDATA |= 0x08; /* 将 PD3 设置为高电平 */
switch( *(P_U8)M328_PDDATA) /* 检验是否同第一行的按键相吻合 */
{
case 0x0d:
printf("9 \n");
break;
case 0x1d:
printf("A \n");
break;
case 0x2d:
printf("B \n");
break;
case 0x4d:
printf("C \n");
break;
}
*(P_U8)M328_PDDIR = 0x10; /* 将 PD4 设置为输出,PD3,5-7 为输入 */
delay(count); /* 延时 */
*(P_U8)M328_PDDATA |= 0x10; /* 将 PD4 设置为高电平 */
switch( *(P_U8)M328_PDDATA) /* 检验是否同第二行的按键相吻合 */
{
case 0x1d:
printf("6 \n");
break;
case 0x15:
printf("7 \n");
break;
case 0x35:
printf("8 \n");
break;
case 0x55:
printf("D \n");
break;
}
*(P_U8)M328_PDDIR = 0x20; /* 将 PD5 设置为输出,PD3,4,6,7 为输入 */
```

```
delay(count); /* 延时 */
*(P_U8)M328_PDDATA |= 0x20; /* 将 PD5 设置为高电平 */
switch( *(P_U8)M328_PDDATA) /* 检验是否同第三行的按键相吻合 */
{
case 0x2d:
printf("3 \n");
break;
case 0x35:
printf("4 \n");
break;
case 0x25:
printf("5 \n");
break;
case 0x65:
printf("E \n");
break;
}

*(P_U8)M328_PDDIR = 0x40; /* 将 PD6 设置为输出,PD3-5,7 设置为输入 */
delay(count); /* delay */
*(P_U8)M328_PDDATA |= 0x40; /* 将 PD6 置为高电平 */

switch( *(P_U8)M328_PDDATA) /* 检验是否同第四行的按键相吻合 */
{
case 0x4d:
printf("0 \n");
break;
case 0x55:
printf("1 \n");
break;
case 0x65:
printf("2 \n");
break;
case 0x45:
printf("F \n");
break;
}

*(P_U8)M328_PDSEL &= ~(0x80); /* 将 PD7 设置为中断引脚 */
*(P_U8)M328_PDDIR |= 0x78; /* 将 PD3-6 设置为输出 */
```

```

* (P_U8)M328_PDDIR &= ~(0x80); /* 将 PD7 设置为输入 */
* (P_U8)M328_PDDATA |= 0x78; /* 将 PD3-6 设置为高电平 */
|
asm(" ANDI.W # $F0FF,SR"); /* 使能中断 */
|
void KeyIRQInit(void)
|
asm(" ORI.W # $0700,SR"); /* 禁止所有中断 */
* (P_U8)M328_IVR = 0x40; /* 设置 328 的中断向量 */
* (P_U32)M328_ISR &= ~(0x00080000); /* 重新设置 ISR */
* (P_U8)M328_PDSEL |= 0x78; /* 选择 PD3-6 as 为 I/O 端口 */
* (P_U8)M328_PDSEL &= ~(0x80); /* 选择 PD7 为中断引脚 */
* (P_U8)M328_PDPUEN = 0xff; /* 使能上拉电阻 */
* (P_U8)M328_PDIRQEN |= 0x80; /* 禁止 INT3 */
* (P_U8)M328_PDDIR |= 0x78; /* 将 PD3-6 设置为输出 */
* (P_U8)M328_PDDIR &= ~(0x80); /* 将 PD7 设置为输入 */
* (P_U8)M328_PDDATA |= 0x78; /* 将 PD3-6 设置为高电平 */
* (P_U32)M328_IPR &= ~(0x00080000); /* 清除等待寄存器 */
* (P_U32)M328_IMR &= ~(0x00080000); /* 去掉 IRQ6 的屏蔽 */
* LEVEL6V = (U32)KeyIRQ; /* 设置中断表 */
asm(" ANDI.W # $F0FF,SR"); /* 使能中断 */
|
U32 KeyTest(void)
|
printf("\nKeypad Test\n");
printf("===== \n");
printf("\n.....Pls press the Keypad. \n");
KeyIRQInit();
while(1);
asm(" ORI.W # $0700,SR"); /* 禁止所有中断 */
|
void main(void)
|
KeyTest();
|

```

超星数字图书馆
 使用本复制品
 请尊重相关知识产权!

8. 补充—EZ328 的中断机理

学习完了上面的例子后,对如何为嵌入式系统添加键盘就有了详细的了解和认识。由于上例论述到了 EZ328 的相关中断机制,在此简要补充一下,便于理解。

(1) 概述

EZ328 的中断控制器支持 18 个边缘和电平中断,其中断分为 7 个级别。第 7 级是最高级别,第 1 级最低。各级中断源情况如下:

- 第 7 级:EMUIRQ。
- 第 6 级:外部中断,如上计时器中断,以及脉冲宽度调制器中断。
- 第 5 级:外部中断。
- 第 4 级:串行外设接口(SPI 口)中断;UART 请求服务中断;软件看门狗计时器中断;实时时钟中断;键盘中断;通用中断引脚 INT[3:0]中断(这些引脚可以作为键盘中断,本小节所讲的例子即是这么用的)。
- 第 3 级:外部中断。
- 第 2 级:外部中断。
- 第 1 级:外部中断。

(2) 中断处理过程

在 EZ328 中,中断处理过程的顺序如下:

①中断控制器从片内和片外设备中收集中断场景,将其按优先级别排好顺序。如果没有其他更高级别的中断到来,那么将中断级别最高的请求发送到处理器。否则,先执行更高级的中断。

②内核处理器响应中断请求,在当前指令执行完成以后,执行一个中断确认总线周期。

③中断处理器发现中断确认信号周期(IACK),并将该中断信号对应的中断向量发送到内核处理器的总线上。

④内核处理器在异常中断向量表中读取到中断向量和中断处理句柄的地址,开始从该地址执行。

要进行中断操作,就必然要设置各种中断控制器、选择器等寄存器部件和 PIO 口的各个设置寄存器,如图 8-6 所示。其详细情况,可以对照程序参看 EZ328 的用户手册。

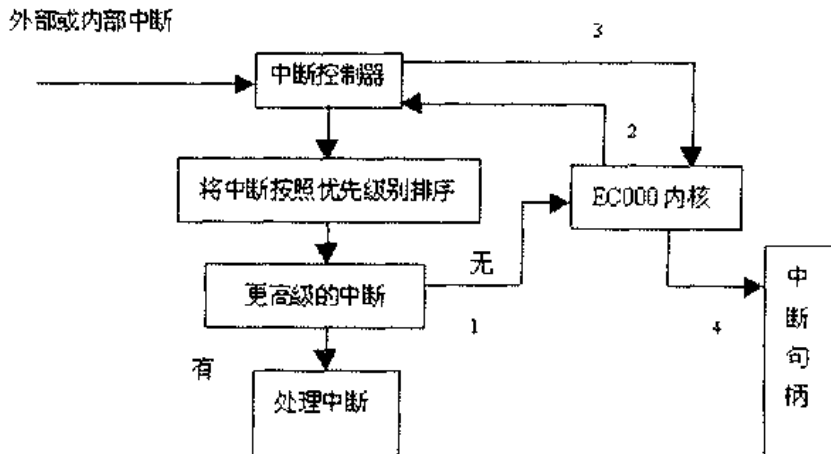


图 8-6 中断处理过程

8.3 LCD 的显示和控制

在第 4 章已经介绍过开发板使用的是黑白两色 LCD,那么如何在 LCD 上显示数据就是用户所关心的问题。从本小节起,学习如何在 LCD 上显示数据及控制手写板的方法。

8.3.1 LCD 的控制与 uClinux 对 LCD 的支持

1. LCD 简介

LCD 也就是液晶显示屏(Liquid Crystal Display)的英文缩写。液晶技术应用到显示方面已有三十多年的历史了。由于液晶具有各向异性和低弹性常数等特性能,这使得它具有丰富多彩的电光效应。目前已经发现具有使用价值的电光效应,如图 8-7 所示。其中以扭曲向列(TN)效应应用得最广。

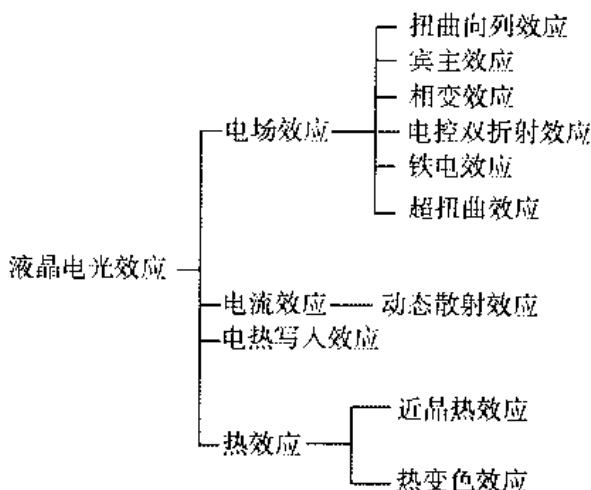


图 8-7 液晶电光效应分类

所谓电光效应,实际上就是指在电的作用下,液晶分子的初始排列改变为其他的排列形式,从而使液晶盒的光学性质发生变化。也就是说,以“电”通过液晶对“光”进行了调制。不同的电光效应可制成不同类型的显示器件。几种主要电光效应制成的液晶显示器件分别介绍如下:

(1) 扭曲向列(TN)效应液晶显示器件

将涂有透明导电层的两片玻璃基板间夹上一层正介电异向性液晶(Np),液晶层厚约 10μ ,液晶分子沿玻璃表面平行排列,不过排列方向在上下玻璃之间连续扭转了 90° 。这就是 TN 型液晶盒的构成。

由于液晶盒中液晶扭曲的螺距与可见光波长相比是相当大的,所以,当偏振光垂直射入液晶层后,其偏振方向会被液晶扭转 90° 。即这个液晶盒具有在平行偏振片间可以遮光、在正交偏振片间可以透光的功能。

对这种液晶盒施加电压,当达到一定电压值时,液晶分子长轴便开始沿电场方向倾斜。当达到 2 倍阈值时,除电极表面外的所有液晶分子长轴一律都沿电场方向进行再排列。这时 90 度旋光消失。即将其放在平行偏振片间透光、放在正交偏振片间遮光。

这就是 TN 型电光效应液晶显示器件的工作原理,它是应用最广的一类液晶显示器件。用户所看见的时隐时现的黑字,不是液晶变色,而是光被遮挡或被透过的结果。

(2) 宾主(GH)效应液晶显示器件

将沿长轴方向和短轴方向对可见光吸收不同的二色性染料作为客体,溶于定向排列的液晶作为主体,二色性染料将会“客随主变”地与液晶分子同向排列。当作为主体的液晶分子排列在电场作用下发生变化时,二色性染料分子的排列方向也将随之变化。即二色性染料对入射光的吸收也将发生变化。这就是所谓的宾主效应。

这种 GH 液晶显示是一种彩色显示,不用偏振片即可以获得足够对比度的显示效果,其视角范围比 TN 型的大得多。

(3) 电控双折射效应显示器件

这种电光效应的器件是将负介电单向性向列液晶垂直于玻璃表面,或一侧垂直于玻璃表面,一侧平行于玻璃表面的正介电单向性液晶制成的液晶盒。在施加电场时,由于在不同的电场强度下,液晶分子长轴与电极将产生一个不同的倾角。它随施加电压变化而变化,从而使液晶盒产生双折射效应。入射偏振光由于双折射而变成椭圆偏振光,它将被选择透过检偏振片。透过光由于干涉而着色,这就是所谓的电控变色效应。其色相随施加电压强度而变化。

ECB 是一种多色液晶显示的好方法,但由于其双折射率受温度影响较大,所以使用起来不太方便。

(4) 相变效应液晶显示器件

所用液晶为 N_p 时,需掺入 Ch 液晶,使混合液晶成为一长螺距液晶。此时,螺旋轴与玻璃平面呈自由杂散状,对外界产生散射,呈白浊状。

当施加一定电压后液晶长轴沿电场方向排列,螺旋解体,液晶盒呈透明状。如果将二色性染料掺入 Ch 液晶中,还可以制成彩色液晶显示器件。这种显示器件由于不需用偏振片,所以显示时明亮、视角大,可以实现负像显示,也可以实现正像显示。

(5) 动态散射效应液晶显示器件

如果在液晶盒中向列液晶中掺入一定比例的有机电介质,当通以一定频率的交流电时,随着电压的提升,液晶就会产生所谓的威廉畴。如果电压继续升高,最终会形成对光有强烈散射作用的湍流或搅动。这种现象称为动态散射效应。动态散射效应是最早应用于显示技术的效应,但由于它属于电流型器件,功耗较大,所以现在已很少使用。

用户的开发板所使用的 LCD 就是扭曲向列(TN)效应液晶显示器件。应该说,这种器件是比较落后的,它不能显示彩色图形。

2. EZ328 对 LCD 的控制

EZ328 为 LCD 的控制提供了很好的支持,其实现主要是通过 LCD 控制器完成的。LCD 控制器负责为外部的 LCD 驱动器提供数据。它通过周期性的 DMA 传送,从系统的内存中读取显示数据。它使用了很窄的总线带宽,以给予内核充分的处理时间。LCD 控

制器由中断寄存器组、控制逻辑模块、队列缓冲以及光标逻辑模块等部分构成,如图 8-8 所示。

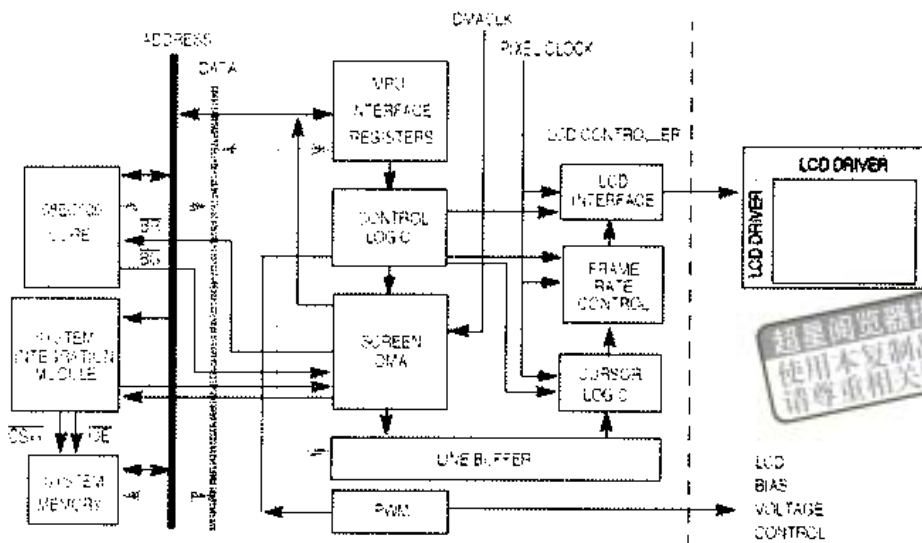


图 8-8 LCD 功能结构图

MPU 接口寄存器组首先使能 LCD 控制器的各种特征。它们与 68K 总线相连,控制逻辑模块为其他模块提供内部控制和计算信号。DMA 向核心发出总线请求,当总线被其占用后,它将执行一些内存突发操作以添满队列缓冲。每一个脉冲所需的 DMA 时钟周期的数目是可修改的,这样,就很容易使系统具有不同的存储速度。

在 DMA 时钟周期内,队列缓冲将显示数据从内存中读出来,并且将其送入光标逻辑模块。输入与较快的 DMA 时钟同步,而输出与较慢的 LCD 像素时钟同步。光标控制逻辑用来设置显示屏上的块状光标。可以调节光标的高度和宽度,光标的亮度和刷新率可以通过设置 LCD 闪烁控制寄存器来调节。

帧速率主要用来控制灰度等级并最大能从 16 个级别中设置 16 个灰度标度。灰度等级对应于当画面刷新时一像素打开的次数。由于晶体的形成方式和驱动电压有所不同,通过对 LCD 灰度调节寄存器(LGPMR)的编程可以很好地调节灰度。

LCD 接口逻辑将要显示数据打包成合适的大小,并传送到 LCD 控制面板的数据总线。通过编程可以很好地控制诸如 LFLM、LP、LCLK 等信号和像素数据的极性,以迎合不同 LCD 面板的需求。

以上简要介绍了 LCD 控制模块的基本工作原理。由于主要工作不是设计显示电路,关心的是如何通过编程序对 LCD 控制器进行操作,以达到所希望的显示结果。所以把主要精力都放在其编程模式上。

3. 编程模式

先来看看 LCD 控制寄存器的情况。

(1) LCD 起始寄存器

这一寄存器被用来描述显示数据的起始地址。其结构如图 8-9 所示。31-29 位是被保留的,它们必须被设置为零。

LXMA																
BIT	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
FIELD	RESERVED		0	1	2	3	4	5	6	7	8	9	10	11	12	13
RESET	0x00000000															
ADDR	0x00000000															

图 8-9 LCD 启动寄存器的结构

BIT	7	6	5	4	3	2	1	0
FIELD	VM	VM	VM	VM	VM	VM	VM	VM
RESET	0							
ADDR	0x00000000							

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	-															
RESET	0															
ADDR	0x00000000															

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

这一寄存器被用来描述 LCD 虚拟页面的宽度。其结构如图 8-10 所示。此寄存器定义了虚拟显示图形的宽度，这与后面要介绍的屏幕宽度寄存器是不同的。

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	-															
RESET	0															
ADDR	0x00000000															

图 8-10 LCD 虚拟页面宽度寄存器的结构

(3) LCD 屏幕宽度寄存器

这一寄存器被用来描述 LCD 屏幕宽度。其结构如图 8-11 所示。这一寄存器以像素的数目定义了 LCD 的宽度，而所用的 LCD 的像素宽度为 160，所以应将其设置为 0x00a0。

LXMAX

图 8-11 LCD 屏幕宽度寄存器的结构

(4) LCD 屏幕高度寄存器

这一寄存器被用来描述 LCD 屏幕高度。其结构如图 8-12 所示。这一寄存器以像素的数目定义了 LCD 的高度。开发板使用的 LCD 的像素高度为 160，所以应将其设置为 0x00a0。

图 8-12 LCD 屏幕高度寄存器的结构

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	CYP		CYP		CYP		CYP		CYP		CYP		CYP		CYP	
RESET	00000000															
ADDR	0x00000000															

器

这一寄存器被用来描述 LCD 光标的 X 坐标。其结构如图 8-13 所示。

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	CC1		CC0		CXC9		CXC8		CXC7		CXC6		CXC5		CXC4	
RESET	00000000															
ADDR	0x00000000															

CD 光标位置 X 坐标寄存器的结构

CC1 位与 CC0 位用来控制光标显示属性。

00 = 不显示光标

BIT	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FIELD	CXC9		CXC8		CXC7		CXC6		CXC5		CXC4		CXC3		CXC2	
RESET	00000000															
ADDR	0x00000000															

11 = 全白色光标

cyp9—0 用来表示光标的起始 X 坐标。

(6) LCD 光标位置 Y 坐标寄存器

这一寄存器被用来描述 LCD 光标的 Y 坐标。其结构如图 8-14 所示。cyp8—0 用来表示光标的起始 Y 坐标。



图 8-14 LCD 光标位置 Y 坐标寄存器的结构

(7) LCD 光标宽高寄存器

这一寄存器被用来描述 LCD 光标的高度与宽度。其结构如图 8-15 所示。

图 8-15 LCD 光标宽高寄存器的结构

CW4—0 位用来设置光标的宽度(从 1 到 31)。

CH4—0 位用来设置光标的高度(从 1 到 31)。

(8) LCD 闪烁控制寄存器

这一寄存器被用来描述 LCD 光标是如何闪烁的。其结构如图 8-16 所示。

BKEN 位是用来表示光标是否闪烁。

1 = 闪烁

0 = 不闪烁

BD6—0 用来表示闪烁的间隔时间。

BIT	7	6	5	4	3	2	1	0
FIELD	SD3h	SD2h	SD1h	SD4h	SD0	SD2	SD1	SD0
RESET	0x1F							
ADDR	0x0700001F							

BIT	7	6	5	4	3	2	1	0
FIELD					PBSIZX	PBSIZX	GSX	GSX
RESET	0x0							
ADDR	0x0700001F							

图 8-16 LCD 闪烁控制寄存器的结构

(9) LCD 接口设置寄存器

这一寄存器被用来定义显示 LCD 面板的数据总线宽度和 LCD 的颜色。其结构如图 8-17 所示。

BIT	7	6	5	4	3	2	1	0
FIELD	LCDON	DMCTx	-	-	DM0	DM1	DM0	DM0
RESET	0x0							
ADDR	0x0700001F							

图 8-17 LCD 接口设置寄存器的结构

PBSIZX 位用来定义数据总线宽度。

00 = 1 位

01 = 2 位

10 = 4 位 /* 开发板已经被定义为 4 位宽度 */

11 = 不使用

GSX 位用来定义灰度等级选择。

00 = 白与黑模式

/* 开发板使用该模式。请见文件/arch/m68knommu/platform/68EZ328/ucsimm - head.s */

01 = 四灰度等级模式

10 = 十六灰度等级模式

11 = 保留

(10) LCD 时钟控制寄存器

这一寄存器被用来开启 LCD 控制器并控制 LCD 的时钟周期。其结构如图 8-18 所示。

图 8-18 LCD 时钟控制寄存器的结构

LCDON 位表示 LCD 控制器的开关。

0 = 开启

1 = 关闭

DWIDTH 位表示显存的字长。

0 = 16 位字长

1 = 8 位字长

DWSX 位定义了显存的等待状态,它表示每次 DMA 的时钟周期数。

BIT	7	6	5	4	3	2	1	0
FIELD	DATA	DATA	DATA	DATA	DATA	DATA	DATA	DATA
RESET	0xFF							
ADDR	0xFFFFA2							

...

...

BIT	7	6	5	4	3	2	1	0
FIELD	G23	G22	G21	G20	G19	G18	G17	G16
RESET	0x0							
ADDR	0xFFFFA2							

(11) LCD 刷新率调整寄存器

这一寄存器用来调节 LCD 的刷新率。其结构如图 8-19 所示。这些位定义了画面保存的时间。

图 8-19 LCD 刷新率调整寄存器的结构

(12) LCD 灰度寄存器

这一寄存器用来调节 LCD 的灰度。其结构如图 8-20 所示。

图 8-20 LCD 灰度寄存器的结构

G23 - G20 位定义了二灰度等级的色调度。

G13 - G10 位定义了其他灰度等级的色调度。

4. uClinux 对 LCD 的支持

接着来看看 uClinux 对 LCD 的支持情况。uClinux 已经提供了对 LCD 显示屏的支持,但默认的,该编译开关(INIT_LCD)是关闭的。在文件/linux/arch/m68knommu/platform/68EZ328/ucsimm-flash-head.S 中可以打开 INIT_LCD 开关。参照 MC68EZ328 DragonBall-EZ(TM) Integrated processor User's Manual P7-7 7.2.4 的 Port D Registers 和 Section12 LCD Controller,来设置 LCD 控制寄存器(160×160 点阵)。

其设置方法如下:

先对寄存器进行设置:

```
#ifdef INIT_LCD /* 如果 INIT_LCD 开关已经打开 */
```

```

movel #splash_bits, 0xffffA00 /* LSSA,即 LCD 显示屏幕起始地址寄存器 */
moveb #0x28, 0xffffA05 /* LVPW 虚拟页宽寄存器 */
movew #0xa0, 0xFFFFFa08 /* LXMAX ,LCD 宽度寄存器 */
movew #0xa0, 0xFFFFFa0a /* LYMAX,LCD 高度寄存器 */
moveb #0, 0xfffffa29 /* LRRA,LCD 刷新速率调整寄存器 */
moveb #0, 0xfffffa25 /* LPXCD ,LCD 像素时钟除数寄存器 */
moveb #0x08, 0xFFFFFa20 /* LPICF,LCD 面板接口设置寄存器 */
moveb #0, 0xFFFFFA21 /* 设置去往 LCD 面板的控制信号的极性 */
moveb #0x81, 0xffffA27 /* LCKCON。这些位表示了静态显示内存的等待状态控制。其量值为每个 DMA 访问周期的时钟数目,这里定义的是 1 个时钟周期 */
movew #0xff00, 0xffff412 /* LCD pins,ffff412 对应的是 PC 口 */
#endif

```

然后进行以下操作：

→ cd /linux

→ make menuconfig

 General setup - - - >

 [*] Console support 打开

 [*] Frame buffer 打开

在启动开发板时,LCD 显示屏上将显示一个封面——一只可爱的小企鹅。它实现的方法就是直接写的 LCD 的显存。文件/uclinux/linux/arch/m68knommu/platform/68E7328/ucsimm head.S 中有：

```

...
splash_bits:
#include "penguin.rh"
.text
...

```

同时,系统在/dev 下建立 fb0(fb 即 frame buffer)设备,可以对该设备进行读写,即是对 LCD 显示屏上像素值的操作。

8.3.2 应用程序的编制

开发板提供了一系列的图形界面接口函数 API。它们仿照 Win 32 API 的接口,使用户能够以最短的时间熟悉并使用它们,迅速地成为 Linux 平台的开发者。API 的原型及详细的定义,可以参看华恒公司的开发套件应用手册,也可以参考本书携带的光盘中的/uclinux/appsrg/gui/graphic.c 文件。在该文件中,对各个 API 给出了详细的定义和使用方法。有了这些 API 函数,就减少了很多同底层硬件打交道的麻烦,从而使用户的注意力

集中到应用程序的编制上来。

下面,学习几个富有代表性的程序。在学习的同时,会讲解各个 API 的用法。

1. 在 LCD 上显示字符串

例 8-2 在 LCD 上显示一个字符串

```
/* This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include "graphic.c"
#include <unistd.h>
```

```
void main(void)
{ char * str = "hello,it is a test";
  initgragh();
  clearsreen();
  textout(20,100,str);
  closegragh();
  return 0;
}
```

编译通过后,执行该程序:

```
> ./test8_1
```

再看看 LCD,发生了什么结果? 在 LCD 的显示屏幕上,打印出了以下的字符串。

```
hello,it is a test
```

要注意该字符串在 LCD 上的显示位置,LCD 的布局如图 8-21 所示。

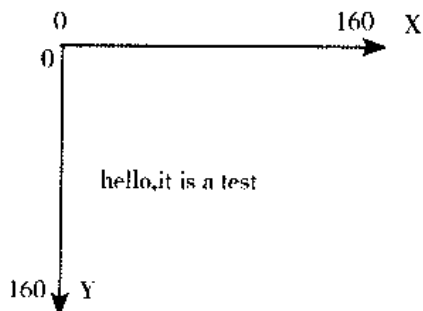


图 8-21 LCD 的布局和显示

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

X, Y 方向均为 160 个像素点。使用 `textout(20, 100, str)` 函数, 将字符串打印到 (20, 100) 起始处。使用这些图形函数能很容易实现显示功能, 这些函数的调用如下:

`initgraph(void)`

`short initgraph(void)`

功能: 初始化显示环境, 返回结果表示初始化是否成功。

`closegraph(void)`

`Void closegraph(void)`

功能: 关闭显示环境。

`clearscreen()`

`void clearscreen(void)`

功能: 清屏。

`textout()`

`void textout(short x, short y, unsigned char * s)`

功能: 在 (x, y) 坐标处输出字符串。

那么, `textout(20 ,100, str)` 就可以在 (20, 100) 处显示字符串 `str`。

这些函数, 包括在后面要用到的函数, 在 `graphic.c` 中都有定义。参看该文件, 可以了解各个函数的功能具体是如何实现的。

2. 图形用户界面功能样例——gui

例 8-3 图形用户界面样例程序: `gui.c`

功能: 显示功能系列演示。包括清屏、显示字符、翻滚移动地显示“华恒科技”四个字等。

(1) 第一部分: 头文件 `gui.h`

```
/* 该头文件是基本的 gui 程序的头文件, 在以后的程序中还会用到。 */
```

```
/*
```

```
* $ Id: gui.h, v 1.0 2001/06/12 19:03:26 till Exp $
```

```
*
```

```
* basic gui header file
```

```
*
```

```
* Copyright (C) 2001 Chen Yang <chyang@hhcn.org>
```

```
*
```

```
* This program is free software; you can redistribute it and/or modify
```

```
* it under the terms of the GNU General Public License as published by
```

```
* the Free Software Foundation; either version 2 of the License, or
```

```
* (at your option) any later version.
```

```
*
```

```
*/
```

```
#ifndef GUI_H /* 条件编译。如果没有定义 GUI_H, 则定义该头文件。在此,
```



GUI_H 可以看做一个开关项 */

* 可以看做一个开关项 */

```
# define GUI_H
```

```
typedef unsigned int UINT;
```

/* 定义 BMP 图形的头文件。函数 ShowBMP 中就调用了该结构。请看文件 graphic.c/

```
typedef struct {
```

```
    unsigned char id[2];
```

```
    long filesize;
```

```
    short reserved[2];
```

```
    long headsize;
```

```
    long infosize;
```

```
    long width;
```

```
    long height;
```

```
    short planes;
```

```
    short bits;
```

```
    long biCompression;
```

```
    long sizeimage;
```

```
    long biXpp;
```

```
    long biYpp;
```

```
    long biclruled;
```

```
    long biclrimportant;
```

```
};BMPHEAD;
```

/* 该联合定义了显示的模式。在 setmode() 中用来设置显示模式, 其意义很好理解。MODE 表示模式, SRC 表示源, NOT 表示取反, OR 表示或, XOR 表示异或, DST 表示目的。例如, MODE_SRC_OR_NOT_DST 就表示该模式为: 目的为与源的反相或之后的结果。*/

```
typedef enum {
```

```
    MODE_SRC,
```

```
    MODE_NOT_SRC,
```

```
    MODE_SRC_OR_DST,
```

```
    MODE_SRC_AND_DST,
```

```
    MODE_SRC_XOR_DST,
```

```
    MODE_SRC_OR_NOT_DST,
```

```
    MODE_SRC_AND_NOT_DST,
```

```
    MODE_SRC_XOR_NOT_DST,
```

```
    MODE_NOT_SRC_OR_DST,
```



```

MODE_NOT_SRC_AND_DST,
MODE_NOT_SRC_XOR_DST,
MODE_NOT_SRC_OR_NOT_DST,
MODE_NOT_SRC_AND_NOT_DST,
MODE_NOT_SRC_XOR_NOT_DST,
InvalidMode
} CopyMode;

```

```

typedef enum {
    BlackPattern = 0,
    WhitePattern,
    DarkGreyPattern,
    LightGreyPattern,
    MicroPengPattern,
    InvalidPattern

```

| PatternIndex; /* 该联合定义了填充的模式。各个模式对应了一个整数。BlackPattern 为 0, 其后的各个模式依次加 1 递增。 */

```

extern unsigned char * screen_ptr;
extern short initgraph(void);
extern void closegraph(void);

```

```

extern void clearsreen(void);

```

```

extern void setpixel(short x, short y, short color); /* 设置该像素((x,y)处)的颜色 */
extern short getpixel(short x, short y); /* 获取该点的颜色 */

```

```

extern void setmode(CopyMode mode); /* 设置显示模式 */
extern CopyMode getmode(void); /* 获取当前显示模式 */

```

```

extern void setcolor(short color); /* 设置显示前颜色 */
extern UINT getcolor(void); /* 获取当前的显示前颜色 */

```

```

extern void setfillpattern(PatternIndex index); /* 设置填充模式 */
extern PatternIndex getfillpattern(void); /* 获取当前填充模式 */

```

```

extern void bar(short x1, short y1, short x2, short y2); /* 以实填充模式在点(x1,

```



y1),(x2,y2)处做矩形 */

```
extern void ellipse(short x1,short y1,short x2,short y2);/* 在矩形(x1,y1),(x2,y2)
中做椭圆 */
```

```
extern void line(short x1, short y1, short x2,short y2);/* 从点(x1,y1) 到点(x2,y2)
做条直线 */
```

```
extern void lineto(short x1, short y1);/* 从当前所在点到点(x,y)间做一条直线 */
```

```
extern void moveto(short x,short y);/* 设置当前所在点为(x,y) */
```

```
extern void rectangle(short x1,short y1,short x2,short y2);/* 在(x1,y1,x2,y2)中做
矩形 */
```

```
extern void textout(short x,short y,unsigned char s);/* 在坐标(x,y)处输出字符串 s */
```

```
extern void V_scroll_screen(short height);/* 垂直方向滚动屏幕 */
```

```
extern void H_scroll_screen(short width);/* 水平方向滚动屏幕 */
```

```
#endif
```

(2) 第二部分:函数主体部分 gui.c

```
/* gui.c: Graphics demos
```

```
*
```

```
* Programmed By Chen Yang<chyang@hhcn.org>
```

```
*
```

```
*
```

```
* This program is free software; you can redistribute it and/or modify
```

```
* it under the terms of the GNU General Public License as published by
```

```
* the Free Software Foundation; either version 2 of the License, or
```

```
* (at your option) any later version.
```

```
*/
```

```
#include "gui.h" /* 要包含 gui.h 这个头文件,它定义了图形用户界面的各个函数 */
```

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

```
{
```

```
short i,j,w,h;
```

```
PatternIndex p=BlackPattern; /* 填充模式为黑色模式 */
```

```
char buf[512]; /* 做显存的数据缓冲区 */
```

```
if(initgraph()) /* 见 graphic.c,如果 initgraph()调用成功,则返回 1 */
```

```
{
```

```
if(argc>1)
```

```

    }
    ShowBMP(argv[1]);
    for(i=1;i<160;i++)
    {
        V_scroll_screen(1); /* 以 1 为单位往上滚动屏幕 */
    }
    ShowBMP(argv[1]);
    /* 如果键入:gui XX.bmp,而且该 bmp 文件是黑白二色的话,就可以在 LCD 上显示
    出来。如果不是黑白二色,则显示:Unsupported color bitmap! 的错误信息 */
    for(i=1;i<160;i++)
    {
        V_scroll_screen(-1); /* 输入的参数为 -1,表示以 1 为单位向下滚动屏幕 */
    }

    textout(0,0,"Press Enter To Show File."); /* 在(0,0)处显示"Press Enter To Show
                                                * File" */
    ShowBMP(argv[1]);

    for(i=1;i<160;i++)
    {
        H_scroll_screen(1); /* 水平方向向左滚动 1 个单位 */
    }
    ShowBMP(argv[1]);
    for(i=1;i<160;i++)
    {
        H_scroll_screen(-1); /* 水平方向向右滚动 1 个单位 */
    }
    }
    clearscreen(); /* 清屏 */
    for(;p<InvalidPattern;p++)
    {
        setfillpattern(p); /* 设置各种填充的模式 */
        sprintf(buf,"%d",p); /* 将填充模式输入到数组 buf 中 */
        textout(120,0,buf); /* 在(120,0)处打印输出模式 */
        fillrect(0,0,120,120); /* 以该模式填充(0,0,120,120)的矩阵 */
    }
    clearscreen(); /* 清屏 */

```



```

textout(0,0,"华恒科技");

for(i=0;i<16;i++)
    memcpy(buf+i*8,0x400+i*20,8); /* 拷贝显存,"华恒科技"4个汉字要用8
个字节来表示。显存起始地址由 LCD 起始地址寄存器 LSSA 定义,此处为 0x400 */

for(i=0;i<16;i++)
    memcpy(0x400+320+i*20,buf+i*8,8);

srand(0); /* srand()产生一个随机数,其均值为 0 */

for(;;)
{
    i=rand()%160; /* %表示取余数,i,j均为 0~160 之间的整数 */
    j=rand()%160;
    if(i<96)
        if(j<144)
/* bitblt()的调用方式为:
/* void bitblt(
/*     short src_x,
/*     short src_y, /* src_x,src_y 为像素在 LCD 显示屏上的源地址 */
/*     short w, /* 数据块的宽度 */
/*     short h, /* 数据块的高度 */
/*     short dest_x,
/*     short dest_y, /* dest_x,dest_y 为位块将要被传输到的目标地址 */
/*     unsigned char *src, /* src 表示源数据块的数据指针 */
/*     short src_units_per_line, /* 数据源每行的宽度 */
/*     unsigned char *dest, /* 位块数据将要传送到目标地址的指针 */
/*     short dest_units_per_line /* 目标方每行的宽度 */
/* )
/* 它的功能是在 LCD 上的显示数据按规定的方式传输 */

bitblt(0,0,64,16,i,j,buf,8,0x400,20);
    else
        if(j>=144)
            bitblt(0,0,64,159-j,i,j,buf,8,0x400,20);

```





```
else
    if(j < 144)
        bitblt(0,0,159 - i,j,i,j,buf,8,0x400,20);
    else
        if(j >= 144)
            bitblt(0,0,159 - i,159 - j,i,j,buf,8,0x400,20);
        }
    closegraph();
}
```

该程序流程如图 8-22 所示。

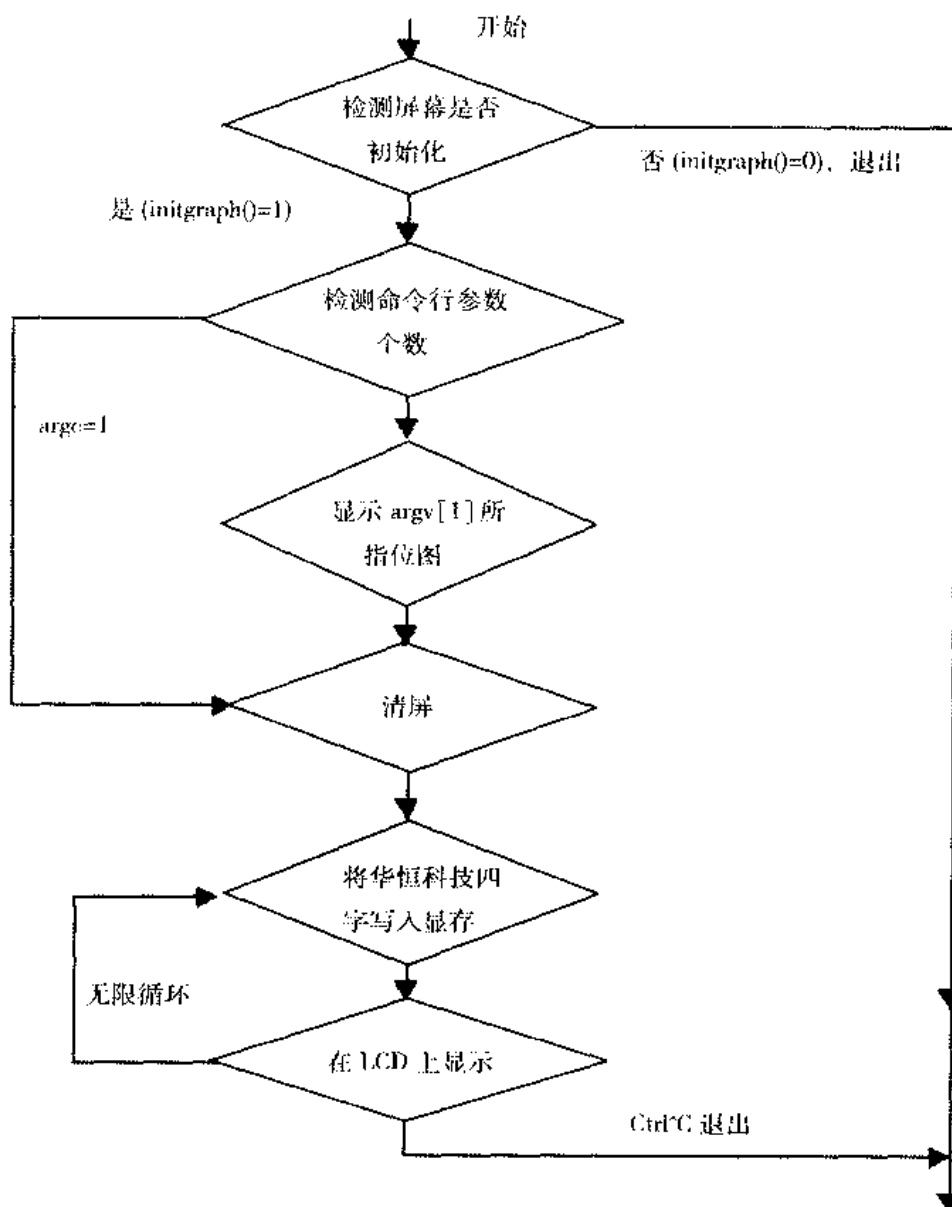


图 8-22 gui.c 样例程序流程图

(3) 输出结果

本程序的输出结果是:当键入命令为 gui 时,在整个 LCD 上将会出现“华恒科技”四个字,而且其出现的位置也是随机的。当键入命令为 gui a 时,将会在 LCD 上首先出现该



Handpad

Handpad 控制系统等功能是嵌入式系统,特别是诸如 PDA 等以开发平台为例,来看一个手写输入的过程是如何实现可以很快地在自己的平台上开发出自己的应用程序来。

启动手写板程序——handpad

启动手写板程序如图 8-23 所示。

图 8-23 启动手写板

其应用功能如下:

(1) 笔形输入

在 LCD 上的笔形输入区内,触摸 LCD 上的手写板即可在 LCD 上显示输入的图形,如图 8-24 所示。

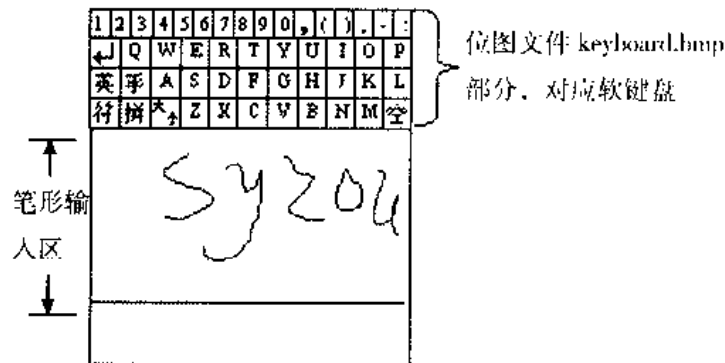


图 8-24 手写板功能演示 1



(2) 手写输入大写字母

触摸字符输入区,对应的字符区就会变暗,在 LCD 下部的显示区中显示该字符,如图 8-25 所示。

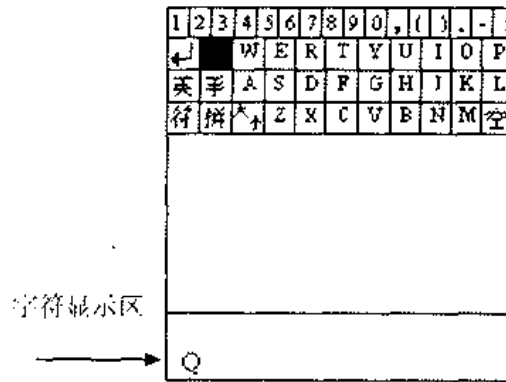


图 8-25 手写板功能演示 2

(3) 手写输入小写字母

先单击“大/小”软按键,软键盘输入字母就变成了小写,可以输入小写字母,如图 8-26 所示。

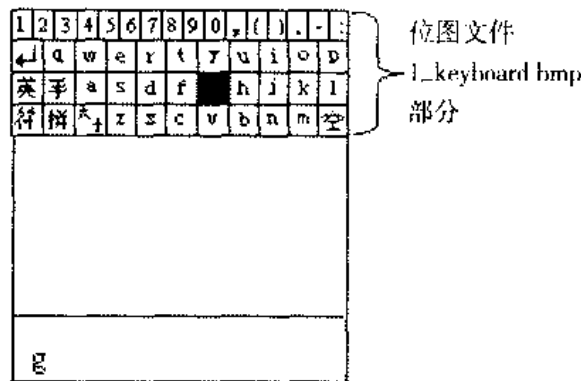


图 8-26 手写板功能演示 3

这个手写板的应用程序已经实现了一些基本的,但又是很重要的 PDA 功能。依照惯例,先了解其源代码,然后按其功能将程序分成各个功能模块。

(4) 源代码及分析

```

/* hand.c: HandPAD Demos
*
* Programmed By Chen Yang (chyang@sina.com)
* Programmed By YunPeng Chen (yunpeng_chen@sina.com)
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation; either version 2 of the License, or

```



```

* (at your option) any later version.
*
*/
#include <stdio.h>
#include <stdlib.h>
#include "../gui/gui.h"    /* 要包含 gui.h 头文件 */
#define HXMAX 216
#define HYMAX    206
#define HXMIN    15
#define HYMIN    15
#define WIDTH    192
#define HEIGHT   160
#define STARTX   0
#define STARTY   0

/* STARTX、STARTY 是计算 LCD 上像素的位置的起始参考点。在这里取为(0,
0)点 */
#define KEYBOARD "keyboard.bmp"
#define L_KEYBOARD "l_keyboard.bmp"
/* keyboard.bmp、l_keyboard.bmp 是系统所带的两个位图文件,用来显示软控制面
板。请见图 8-24 至图 8-26。 */
struct _keyboard{
char ch;
short startx,starty,endx,endy;
}kbd[] = {
{'1',1+STARTX,1+STARTY,9+STARTX,13+STARTY},
{'2',11+STARTX,1+STARTY,19+STARTX,13+STARTY},
{'3',21+STARTX,1+STARTY,30+STARTX,13+STARTY},
{'4',32+STARTX,1+STARTY,40+STARTX,13+STARTY},
{'5',42+STARTX,1+STARTY,50+STARTX,13+STARTY},
{'6',52+STARTX,1+STARTY,60+STARTX,13+STARTY},
{'7',62+STARTX,1+STARTY,70+STARTX,13+STARTY},
{'8',72+STARTX,1+STARTY,80+STARTX,13+STARTY},
{'9',82+STARTX,1+STARTY,90+STARTX,13+STARTY},
{'0',92+STARTX,1+STARTY,100+STARTX,13+STARTY},
{'.',102+STARTX,1+STARTY,110+STARTX,13+STARTY},
{'(',112+STARTX,1+STARTY,120+STARTX,13+STARTY},
{'}',122+STARTX,1+STARTY,129+STARTX,13+STARTY},
{'_',131+STARTX,1+STARTY,139+STARTX,13+STARTY},

```

```

{| '-', 141 + STARTX, 1 + STARTY, 149 + STARTX, 13 + STARTY},
{| ':', 151 + STARTX, 1 + STARTY, 158 + STARTX, 13 + STARTY},
{| '?', 1 + STARTX, 15 + STARTY, 14 + STARTX, 28 + STARTY}, //16
{| 'q', 16 + STARTX, 15 + STARTY, 29 + STARTX, 28 + STARTY},
{| 'w', 31 + STARTX, 15 + STARTY, 44 + STARTX, 28 + STARTY},
{| 'e', 46 + STARTX, 15 + STARTY, 58 + STARTX, 28 + STARTY},
{| 'r', 60 + STARTX, 15 + STARTY, 72 + STARTX, 28 + STARTY},
{| 't', 74 + STARTX, 15 + STARTY, 87 + STARTX, 28 + STARTY},
{| 'y', 89 + STARTX, 15 + STARTY, 102 + STARTX, 28 + STARTY},
{| 'u', 104 + STARTX, 15 + STARTY, 116 + STARTX, 28 + STARTY},
{| 'i', 118 + STARTX, 15 + STARTY, 130 + STARTX, 28 + STARTY},
{| 'o', 132 + STARTX, 15 + STARTY, 144 + STARTX, 28 + STARTY},
{| 'p', 146 + STARTX, 15 + STARTY, 158 + STARTX, 28 + STARTY},
{| '*', 1 + STARTX, 30 + STARTY, 14 + STARTX, 43 + STARTY}, //27
{| '* ', 16 + STARTX, 30 + STARTY, 29 + STARTX, 43 + STARTY},
{| 'a', 31 + STARTX, 30 + STARTY, 44 + STARTX, 43 + STARTY},
{| 's', 46 + STARTX, 30 + STARTY, 58 + STARTX, 43 + STARTY},
{| 'd', 60 + STARTX, 30 + STARTY, 72 + STARTX, 43 + STARTY},
{| 'f', 74 + STARTX, 30 + STARTY, 87 + STARTX, 43 + STARTY},
{| 'g', 89 + STARTX, 30 + STARTY, 102 + STARTX, 43 + STARTY},
{| 'h', 104 + STARTX, 30 + STARTY, 116 + STARTX, 43 + STARTY},
{| 'j', 118 + STARTX, 30 + STARTY, 130 + STARTX, 43 + STARTY},
{| 'k', 132 + STARTX, 30 + STARTY, 144 + STARTX, 43 + STARTY},
{| 'l', 146 + STARTX, 30 + STARTY, 158 + STARTX, 43 + STARTY},
{| '符', 1 + STARTX, 45 + STARTY, 14 + STARTX, 58 + STARTY}, //38
{| '拼', 16 + STARTX, 45 + STARTY, 29 + STARTX, 58 + STARTY},
{| '大/小', 31 + STARTX, 45 + STARTY, 44 + STARTX, 58 + STARTY},
{| 'z', 46 + STARTX, 45 + STARTY, 58 + STARTX, 58 + STARTY},
{| 'x', 60 + STARTX, 45 + STARTY, 72 + STARTX, 58 + STARTY},
{| 'c', 74 + STARTX, 45 + STARTY, 87 + STARTX, 58 + STARTY},
{| 'v', 89 + STARTX, 45 + STARTY, 102 + STARTX, 58 + STARTY},
{| 'b', 104 + STARTX, 45 + STARTY, 116 + STARTX, 58 + STARTY},
{| 'n', 118 + STARTX, 45 + STARTY, 130 + STARTX, 58 + STARTY},
{| 'm', 132 + STARTX, 45 + STARTY, 144 + STARTX, 58 + STARTY},
{| '空', 146 + STARTX, 45 + STARTY, 158 + STARTX, 58 + STARTY}
};

```

/* 定义 keyboard 结构体数组, 给出位图文件上各个字符在手写板上的对应的起始和终点位置 */

超星阅读器扫描
使用本复制品
请尊重相关知识产权!

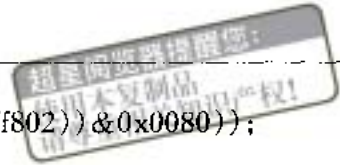
```
void init_handpad() /* 初始化手写板 */
{
    *(volatile unsigned short *) (0xffff304) |= 0x0004; /* 屏蔽 IRQ3 的中断请求, 不
接受来自手写板的中断请求。即下面的所有的初始化是必须完成的, 不可被打断。*/
    /* 对各个并行端口进行设置 */
    *(volatile unsigned short *) (0xffff42a) = 0x0464;
    *(volatile unsigned short *) (0xffff422) = 0xff00;
    *(volatile unsigned short *) (0xffff428) = 0x659c;
    *(volatile unsigned short *) (0xffff432) = 0x211c;
    *(volatile unsigned short *) (0xffff430) = 0x1c3a;
    *(volatile unsigned char *) (0xffff41b) |= 0x40;
    *(volatile unsigned char *) (0xffff418) &= 0xbf; /* 将 PD 口第 6 管脚作为输入,
准许接收 IRQ3 的中断。为 get_handpad() 中查询中断做准备 */
}

/* get_handpad(x,y) 的功能是获取触摸在手写面板上的点的位置。*/
int get_handpad(unsigned short * x, unsigned short * y)
{
    static unsigned char buff[4]; /* 存放两次落点采样的结果 */
    char flag = 0; /* 程序查看 flag 标志, 来判断是否采样成功 */
    if (! (*(volatile unsigned short *) (0xffff418) & 0x0040)) /* 判断手写板是否被释放 */

        *(volatile unsigned short *) (0xffff802) &= 0xfd7f; /* 清除 SPIM 中的 IRQ 位, 准备接受
        * 数据 */

        *(volatile unsigned short *) (0xffff430) = 0x1c2a; /* 设置各个端口状态 */
        *(volatile unsigned short *) (0xffff428) = 0x659e;
        *(volatile unsigned short *) (0xffff802) = 0x023c;
        *(volatile unsigned short *) (0xffff428) = 0x659a;

        *(volatile unsigned short *) (0xffff800) = 0x1800;
    /* 0x1800/0x1c00 都是手写板的控制字。控制手写板, 接收 y 坐标 */
        *(volatile unsigned short *) (0xffff802) = 0x0100; /* XCH 置位, 触发数据
        * 交换 */
}
```



```

while (! (( * (volatile unsigned short *) (0xfffff802)) & 0x0080));
    /* 轮讯 IRQ 位, 等待数据到来 */
buff[0] = ( * (volatile unsigned short *) (0xfffff800)); /* 读取数据, 为 Y 坐标 */

* (volatile unsigned short *) (0xfffff802) &= 0xfd7f; /* 又清除中断 */
* (volatile unsigned short *) (0xfffff428) = 0x659e; /* 再次设置各端口状态。 */
* (volatile unsigned short *) (0xfffff802) = 0x023c;
* (volatile unsigned short *) (0xfffff430) = 0x1c36;
* (volatile unsigned short *) (0xfffff428) = 0x659a;

* (volatile unsigned short *) (0xfffff800) = 0x1C00; /* 手写板控制字。控制手写
    * 板检测落点的 x 坐标。 */

* (volatile unsigned short *) (0xfffff802) |= 0x0100; /* 置位 XCH, 准备数据交换 */

while (! (( * (volatile unsigned short *) (0xfffff802)) & 0x0080));
    /* 再次轮讯 IRQ 位 */
buff[1] = * (volatile unsigned short *) (0xfffff800); /* buff[1] 保存了 X 坐标。 */

* (volatile unsigned short *) (0xfffff802) &= 0xfd7f; /* 清除中断, 防止有
额外的中断到来将下面的步骤打断 */
* (volatile unsigned short *) (0xfffff430) = 0x1c3a;
if (buff[0] == buff[2] && buff[1] == buff[3])
    /* 如果本次的和上一次所测的相同, 则确定为正确的坐标, 将其传送。 */

    /* 调试信息 */
    #ifdef DEBUGTRACE
        printf("buff[0] = %d buff[1] = %d \n", buff[0], buff[1]);
    #endif
    #ifdef LCDTHREADSORTLINE
        * x = buff[0] - HXMIN;
        * y = HYMAX - buff[1];
    #else
        * x = (HXMAX - buff[1]);
    #endif

```

超星阅读器出品
使用本复制品
请尊重相关知识产权!

```

        * y = buff[0] - HYMIN;
    # endif

    // * x = (IXMAX - buff[1]);
    /* 调整原始数据 */
        if( * x > 8) * x -= 8; else * x = 0;
        * x = (( * x) * 160) / 208;
    // * y = buff[0] - HYMIN;
    if( * y > 8) * y -= 8; else * y = 0;
        * y = (( * y) * 160) / 146;
    flag = 1;      /* 采样成功,将标志位置位 */
}

buff[2] = buff[0]; /* 保存这一次的采样值,供下次判别使用 */
buff[3] = buff[1];
}

if(flag) return 1;      /* 如果成功地获取了落点,返回 1。否则返回 0 */
return 0;
}

void main(void)
{
    unsigned short i, x, y, index = 0xffff; /* index 对应了手写软面板上的字符表的各个
    字符在 kdb[] 结构中的目录项 */
    char buf[10];
    short Upper = 1;      /* Upper 用来控制软面板的大/小写 */

    init_handpad();      /* 初始化手写板 */
    initgraph();         /* 初始化显示环境 */
    clearscreen();       /* 清屏 */
    ShowBMP(KEYBOARD, 0, 0); /* 在(0,0)处显示大写软面板 */
    setmode(MODE_SRC_XOR_DST); /* 设置显示模式 */
    rectangle(0, 60, 159, 127); /* 在(0,60,159,127)中做矩形 */
    while(1) {

        if(get_handpad(&x, &y)) /* 获取落点 */
        {
            # ifdef DEBUGTRACE

```

```

printf("x = %d y = %d \n", x, y);
#endif

if(index != 0xffff) /* 如果落点被成功地检测到,且落在软面板键盘上 */
{
fillrect(kbd[index].startx, kbd[index].starty,
        kbd[index].endx + 1, kbd[index].endy + 1);
/* 将对应落点处的小键盘按键填充为黑色 */

if(y < 58) /* y < 58, 在软键盘字符区 */
{
if((y > kbd[0].starty) && (y < kbd[0].endy)) /* 如果落点在第一行 */
for(i = 0; i < 16; i++)
if((x > kbd[i].startx) \
    && (x < kbd[i].endx))
break;
if(i == 16) index = 0xffff; else index = i; /* 如果落点不在 kbd[16]处,就将 kbd
[]数组的索引项赋给 index, 否则将 0xffff 赋给 index, 跳到下面的笔形输入区代码段 */
} else
if((y > kbd[16].starty) && (y < kbd[16].endy)) /* 如果落点在第二行 */
for(i = 16; i < 27; i++)
if((x > kbd[i].startx) \
    && (x < kbd[i].endx))
break;
if(i == 27) index = 0xffff; else index = i; /* 如果落点不在 kbd[16]处,就将 kbd[]
数组的索引项赋予 index, 否则将 0xffff 赋予 index, 跳到下面的笔形输入区代码段 */
} else
if((y > kbd[27].starty) && (y < kbd[27].endy)) /* 如果落点在第三行 */
for(i = 27; i < 38; i++)
if((x > kbd[i].startx) \
    && (x < kbd[i].endx))
break;
if(i == 38) index = 0xffff; else index = i; /* 如果落点不在 kbd[16]处,就将 kbd[]
数组的索引项赋予 index, 否则将 0xffff 赋予 index, 跳到下面的笔形输入区代码段 */
} else
if((y > kbd[38].starty) && (y < kbd[38].endy)) /* 如果落点在第四行 */
for(i = 38; i < 49; i++)

```



```

        if((x>kbd[i].startx) \
            &&(x<kbd[i].endx))
            break;
if(i == 49) index = 0xffff; /* kdb[49]处为“空” */
else
if(i == 40) | /* kdb[40]处为大/小,然后进行大/小写转换 */
    if(Upper) /* 如果原来为大写状态,则转换为小写状态 */
        ShowBMP(L_KEYBOARD,0,0); /* 显示小写键盘 */
    else
        ShowBMP(KEYBOARD,0,0);
    Upper = ! Upper;
    index = 0xffff;
    }
else
    index = i; /* 将目录项赋予 index,以输出 Z、X...M 等字符 */
}

/* 控制在(0,144)处显示字符 */
if(index<49){
fillrect(kbd[index].startx,kbd[index].starty,
        kbd[index].endx + 1,kbd[index].endy + 1);
if(Upper) buf[0] = toupper(kbd[index].ch); /* 如果为大写状态,就将要显示的字符
        * 转换为大写字符 */
else buf[0] = kbd[index].ch;
buf[1] = 0;
textout(0,144,buf); /* 在(0,144)处显示字符 */
}
}

else /* 如果 index = 0xffff,那么就进入笔形输入区 */
{
static unsigned short oldx,oldy,err; /* err 为静态存储变量。通过判断 err 的数值来
        * 决定是否连笔输入 */
if(y>60&& y<127) /* 如果落点在笔形输入区 */
{
    err = (oldx>x)? oldx - x : x - oldx;
    err + = (oldy>y)? oldy - y : y - oldy;
    /* err 为两个落点间的 x,y 坐标的差之和 */
}
}

```




```

if(err>0&&err<20){ /* 判断两个落点的间距大小,如果过大,就不连笔输入,
                    * 直接位移到该点显示。
                    * 否则从起始点到落点间做条直线 */
    lineto(olddx = x,oldy = y);
}
else{
    moveto(olddx = x,oldy = y);
}

}

index = 0xffff;
}

}

}

```



手写板和 LCD 并不是同一个物理设备。手写板附着在 LCD 上,是透明的一层薄膜。它负责将压力转换为模拟电信号,模拟信号再经过 A/D 转换,才能被采样,并送入 CP 进行处理,在 LCD 上显示。但就肉眼看来,好像是直接在 LCD 上“写”一样。手写板的大小也并不与 LCD 大小严格相等。

(5) 程序分析

还是先来看 kdb[]结构数组。该数组是同软键盘的分布密切相关的,如图 8-27 所示。

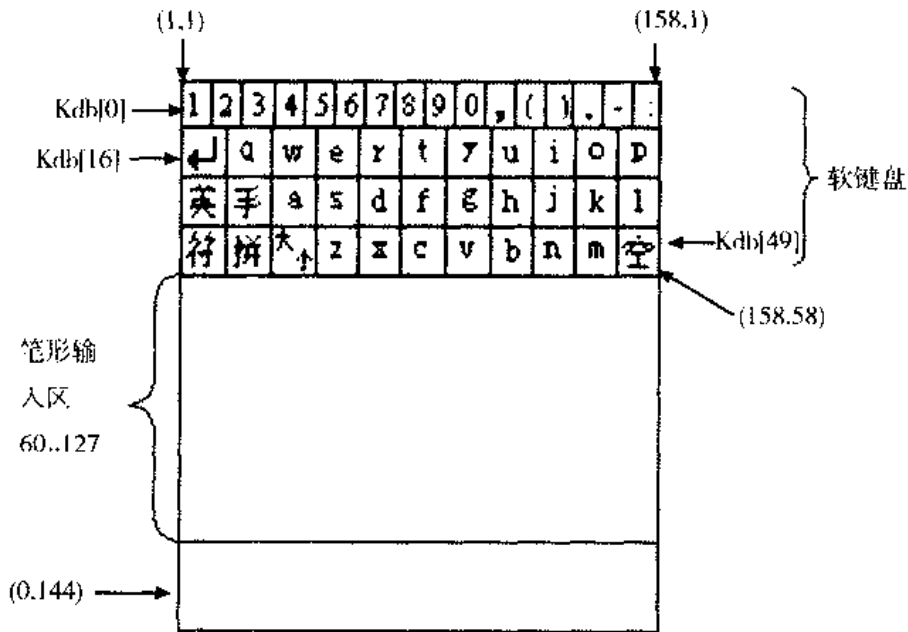


图 8-27 软键盘、笔形输入区和数据输出区在 LCD 上的分布

该数组的每一个元素都是一个数据结构,对应着软键盘上的一个按键。如图 8-27, kdb[0]对应着“1”这个按键。这个元素的信息如下:

```
{'1', 1 + STARTX, 1 + STARTY, 9 + STARTX, 13 + STARTY}
```

“1”表示在软键盘对应的字符是“1”, 1 + STARTX, 1 + STARTY, 9 + STARTX, 13 + STARTY 表示其所处的小矩形的各个位置坐标。已经定义了 STARTX = STARTY = 0, 所以在软键盘上包围“1”的矩形坐标位置为(1, 1, 9, 13), 如图 8-27 所示。

笔形输入区在如图 8-27 的 60 < y < 127 部分。余下的 130 < y < 160 区域为数据输出区。

对于本程序而言,重点是完成获取落点的功能函数 get_handpad(), 再来分析其实现过程,其流程图如图 8-28 所示。

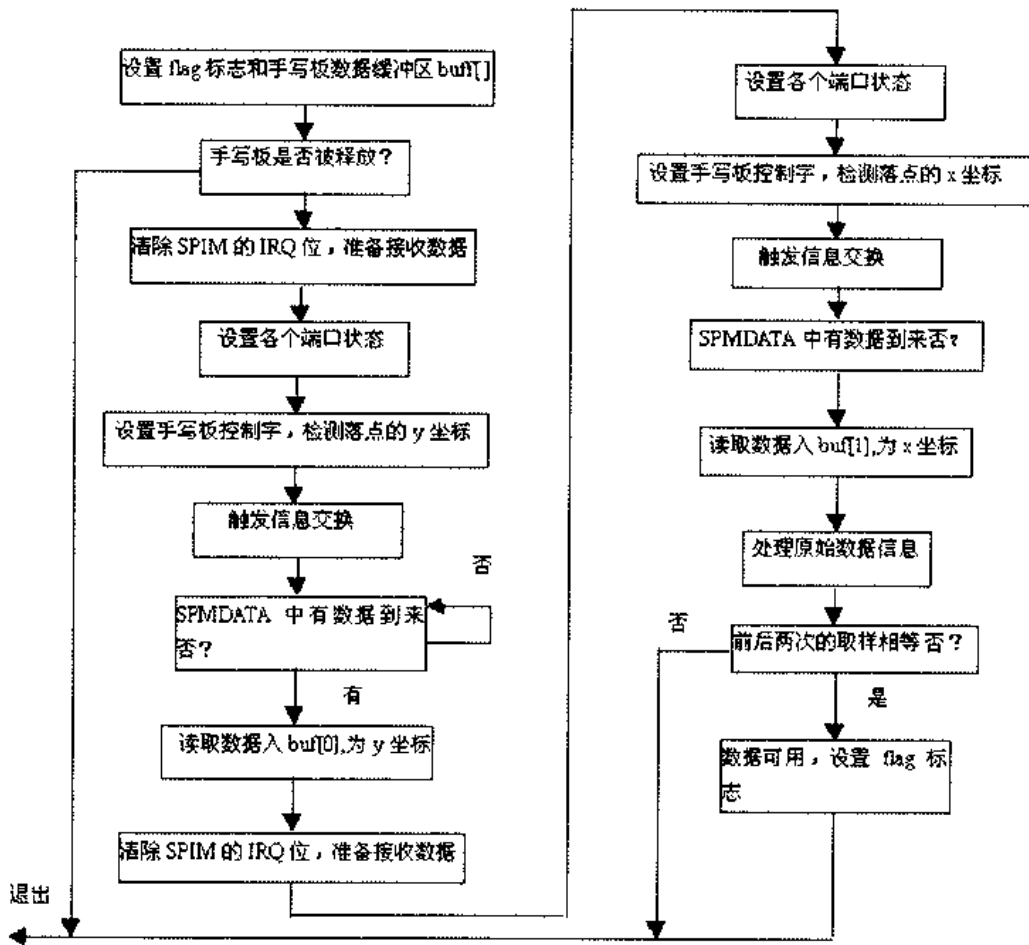


图 8-28 get_handpad() 的流程图

get_handpad(x, y) 的功能是获取触摸在手写面板上的点的位置。如果成功地获取了落点位置, 则返回 1, 否则返回 0。(x, y) 为所检测到的落点。在这里作参数的是指向 x 和 y 的指针。请注意关键字 static 的存在, 即 buff 中的字符都是静态存储变量。在函数调用结束后其占用的存储单元并不释放, 下一次调用时该变量的初始值就是上一次调用结束后的值。从后面可以发现, 这是为了保证采样的有效性。如果两次采样一致, 表明该采样数值有效, 可以发送到 LCD 上去, 否则必须重新采样。因此, 采用一个标志 flag 表示

采样是否有效。

注意：字符数组 buff[4] 是静态的。

```
if(! (* (volatile unsigned short *) (0xffff418) & 0x0040)) {
```

这一句负责查看 PD 口 D6 位的数据是否为 0。如果是，那么没有中断到来，可以执
表示手写板被释放的动作到来，即不在手写板上写东西



```
* (volatile unsigned short *) (0xffff802) &= 0xfd7f;
```

0xffff802 是 SPI 口的控制器 SPIM，它标志了 SPIM 的数率、使能、数据交换、中断和传输长度等。其各个位的功能如图 8-29 所示。上句程序中的 &= 0xfd7f 就是将 ENABLE 和 IRQ 位设为零，清除中断，停止 SPIM 工作，以设置各种状态。手写板是同 SPI 口相连的，A/D 采样的数据通过 SPI 口送入 CPU。这一句表示将 SPIM 中的 IRQ 位清除，并禁止中断产生。根据 EZ328 的手册，当有数据通过 SPI 口输入/出完成后，该位置位。如果清除该位，就为接收数据做好了准备。

图 8-29 SPI 口控制/状态寄存器

其各个位的功能如下：

- DATA RATE: 在这个域可选择 SPMCLK 信号的位速率。

该速率是系统时钟的等分，等分的除数如下：

(SPI 口控制器的主控时钟为系统时钟 SYSCLK，关于 SYSCLK 请参看 EZ328 用户手册)

000 = 主频率除以 4

001 = 主频率除以 8

...

111 = 主频率除以 512

- RESERVED: 12-10 位。这些位作为保留位，值为零。

- ENABLE 位：该位使能了 SPIM。

在启动数据交换前，必须将该位置位，在交换完成后，必须将该位复位。

0 = 禁止 SPIM

1 = 使能 SPIM

- XCH: 数据交换触发位。

该位触发了一次数据交换，并在数据交换进行时保持置位状态。

0 = 空闲

1 = 启动一次数据交换(写)或者忙(读)

- IRQ: 中断请求位。

当某次数据交换完成后,该位置位。如果设置了 IRQEN 位,就可产生中断。中断屏蔽寄存器的 MSPI 位必须清除,以便中断可以施加到内核上去。

● IRQEN: 中断请求使能位。

当 SPI 完成一次数据交换后,该位允许产生一个中断信号。它并不影响 IRQ 位的操作,而只影响施加到中断控制器的中断信号。

0 = 禁止产生中断

1 = 允许产生中断

● PHA: 相位。

该位控制时钟/数据的相位关系。

0 = 工作在 0 相

1 = 工作在 1 相

● POL: 极性。

该位控制 SPMCLK 信号的极性。

0 = 高电平有效

1 = 低电平有效

● BIT COUNT。

这个域用来选择传输数据的位长度。最大的可传输数据长度为 16 位。

0000 = 传输数据长 1 位

0001 = 传输数据长 2 位

...

1110 = 传输数据长 15 位

1111 = 传输数据长 16 位

看这两句:

```
* (volatile unsigned short *) (0xffff800) = 0x1800;
```

```
* (volatile unsigned short *) (0xffff800) = 0x1C00;
```

0x1800/0x1c00 都是手写板的控制字。手写板通过鉴别输入的是 1800 还是 1c00 来判断用户需要的输入是 x 坐标还是 y 坐标,然后采样并进行 ad 转换。0x1800 控制手写板检测落点的 y 坐标,而 0x1C00 控制检测 x 坐标。

```
* (volatile unsigned short *) (0xffff802) |= 0x0100;
```

这句将 XCH 置位。该位触发了一次数据交换,并当数据交换还在进行中时,保持置位的状态。数据交换包括对 SPI 口的读和写操作,所以手写板控制字被发送到手写板上,而落点信息将被接收到 SPIMDATA 中去。

```
while (! (( * (volatile unsigned short *) (0xffff802)) & 0x0080));
```

SPIM 第七位为 IRQ,表示中断到来,即表示此时用户要读取的数据已经读到 SPIM 的数据寄存器 SPIMDATA(在 ffff800)中,用中断来通知用户。所以,该句是轮询 IRQ 位,以判断数据是否到来。

在将采样数据读取到了 buf[0]、buf[1] 中间后,有这么一句:

```
if(buf[0] == buf[2] && buf[1] == buf[3])
```

它表示,如果本次和上一次所测的相同,则确定为正确的坐标,将其传送。已将 buff[] 设置成为了静态数组,上一次的数据读取结果就放置在 buff[3]和 buff[4]中。通过与本次测量的结果相比较,如果一致,就认为在这两次中落点无抖动,采样数据有效,否则就将采样数据抛弃重新测量。

相比之下,main()主函数就明了些。主要是通过一个标志项 index 来判断手写板所落的区域。首先,调用 init_handpad()初始化手写板,然后,使用 initgraph()初始化显示环境。clearscreen()的功能是清屏。

```
rectangle(0,60,159,127);          /* 在(0,60,159,127)中做矩形 */
```

其作用是绘制如图 8-27 所示的笔形输入区。

在获取了落点后,main()判断其落点。如果 index != 0xffff,那么该点就落在软面板键盘上的可显示部分。例如,“1”就是一个可显示的字符,将 index 置为对应的 i 值。该过程的代码如下:

```
if((y>kbd[0].starty)&&(y<kbd[0].endy))    /* 如果落点在第一行 */
    for(i=0;i<16;i++)
        if((x>kbd[i].startx)\
            &&(x<kbd[i].endx))
            break;
if(i==16) index=0xffff;else index=i;
```

当 i = 16 时,kbd[16]对应的是一个回车符号,无法在 LCD 上显示。

```
if((y>kbd[0].starty)&&(y<kbd[0].endy))
```

上面这句判断 y 方向的落点在哪一行。判断确定后,再进行定位 x 的工作。

如果 index 等于 0xffff,而且 y > 58(LCD 上显示的分割线也要占 2 个像素的宽度)时,那么落点就处在笔形输入区,执行跟踪笔落点并输入笔形对应数值的操作。其实现的思想是设定一个阈值,它表示本次落点和上一次落点坐标差的绝对值之和。如果该和超过了该阈值,那么就认为笔形输入是不连续的,将重新从该点开始显示。如果在该阈值内,那么就认为输入连续,在原先的落点与现在的点之间画一根直线。见图 8-24 的笔形输入演示。当然,该阈值可以由用户自己定义,其大小反映了期望输入的精确程度。

⚠ 注意:手写板的内核驱动是存放在 uclinux \ linux \ drivers \ char 目录下的 Handpad.c 这个文件中的。在此讲述的是手写板的应用驱动程序。

4. 用手写板来控制系统

根据上面的例子,可很容易地实现使用手写板对应用程序的控制。来看一个样例程序 syshand.c。

例 8-5 手写板控制系统样例 syshand.c

启动 syshand,在 LCD 上将显示如图 8-30 所示的使用界面。

触摸“lissa”区域,将运行样例程序 lissa,在 LCD 上显示一个不断运动的李萨如曲线。触摸“exit lissa”区域,则退出 lissa。



图 8-30 syshand 的使用界面

该程序与 handpad 样例程序的实现思想是基本一致的。它的头文件和 handpad.c 一模一样, init_handpad 和 get_handpad 也是如此。其不同点如下:

```
...
    # define SYSCTL "sysctl.bmp"
struct _keyboard{
char ch;
short startx, starty, endx, endy;
{kbd[] = {
{'1', 1 + STARTX, 1 + STARTY, 80 + STARTX, 13 + STARTY},
{'2', 81 + STARTX, 1 + STARTY, 158 + STARTX, 13 + STARTY},
};
```

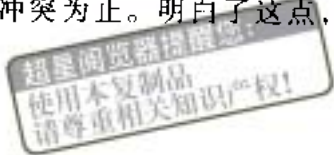
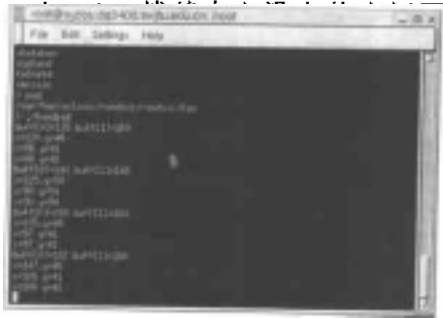
“1”表示了(1,1,80,13)的矩形区域,如图 8-30 所示,为“lissa”字样。“2”表示“exit lissa”字样。这个软键盘比 handpad 的软键盘简单了很多。

syshand 同 handpad 的不同之处在于:在获取了落点后,如果判断该点落在“lissa”字样区,则执行以下代码:

```
if(index == 0 && pid < 0)
{
    sprintf(processbuf, "./lissa"); /* 将"/lissa"输入 processbuf 中 */
    if((pid = vfork()) == 0) /* 如果 vfork()成功 */
    {
        execv(processbuf, NULL); /* 执行 lissa */
        return;
    }
}
```

在前面的章节中,已经讲述了 uClinux 多任务的实现方法。fork 调用并没有在 uClinux 上实现,这是由 uClinux 的内存分配方法所决定的。拷贝页面的问题在于:如果

和堆栈空间。父进程被阻塞,直到 exec 调用开始防止堆栈冲突为止。明白了这点,sys-



式法进行调试,那是根本不可想像的。利用 Linux 系统特有实现对应用程序的调试。调试 handpad.c 程序如图 8-31

图 8-31 对 handpad.c 的调试

在 get_hand()函数中,在想要了解某些变量数值的地方,插入相应的打印语句。重新编译通过后,不要立即烧写闪存,通过调用 nfs mount 功能,将宿主机的/uclinux 目录挂载到 uclinux 的/tmp 目录下。进入存放可执行文件的目录,在本例中即进入到目录/tmp/uclinux/romdisk/romdisk/bin 下,然后执行待调试的程序,即键入:

```
./handpad
```

再用手触摸手写板,调试信息就会不停地被打印出来。此时打印的是 buf[]数组中的原始采样数据和修正后的(x,y)坐标。对于用户自行开发的应用程序,也可以遵循该思路进行调试。

8.4 本章小结

Linux 的一个优秀的特点就在于它能支持多种外部设备,嵌入式 Linux 也不例外。本章重点探讨了两种重要的外部设备——键盘和 LCD。首先,学习了往嵌入式系统中添加键盘的常用方法,并以开发板为例,详细地阐述了添加一个小型的矩阵式键盘的软件和硬件的实现过程。LCD 是嵌入式系统中极为重要的人—机界面的硬件支持部分,因此,本章的后部分将重点放在基于开发平台的显示程序的开发上,详细地分析、学习了两个重

要的手写输入和手写控制的样例程序——handpad.c 和 syshand.c, 阐述了其应用程序的编制思想。最后, 还论述了应用程序的调试方法。

通过研习本章, 可以很快地实现对开发平台功能的扩充, 例如自行添加小键盘、鼠标或开发自己的应用显示程序。希望在学习的同时, 多问自己一些“为什么”、“怎么实现”的问题, 开拓自身的视野和思路, 尽早实现设计和编程水平的进步和飞跃。

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

第四篇

专题讨论

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

第9章 嵌入式实时操作系统 与实时 Linux



本篇各个章节独成体系,分别向各位讲述了嵌入式 Linux 这门技术的几个重要方面。读者朋友可以把它视为前三篇的延续和继承,也可以用独立的眼光来看待和学习本篇的各个章节。本章将学习嵌入式系统的一个主要应用——实时嵌入式操作系统,重点当然是放在学习实时 Linux 之上。

本章主要内容:

- 嵌入式实时操作系统
- 实时系统的设计
- RT-Linux 简介
- 实时内核的模块化实现
- RT-Linux 的 API
- RT-Linux 下编程示例
- RT-Linux 的模拟实现

9.1 嵌入式实时操作系统简介

实时 Linux 是 Linux 嵌入式应用的一个重要分支。本章将重点介绍实时 Linux 的概念、实现机理和在实时 Linux 下的编程方法。首先,还是来了解一下嵌入式实时操作系统的基本概念。

9.1.1 RTOS 的要求

实时系统与其他普通的系统之间最大的不同之处就是要满足处理与时间的关系。在实时计算中,系统的正确性不仅仅依赖于计算的逻辑结果,而且依赖于结果产生的时间。

对于实时系统来说最重要的要求就是实时操作系统必须有满足在一个事先定义好的时间限制中对外部或内部的事件进行响应和处理的能力。这样,实时系统可以定义为“一个能够在事先指定或确定的时间内完成系统功能和对外部或内部、同步或异步时间作出响应的系统”。此外,作为实时操作系统还需要有效的中断处理能力和高效的 I/O 能力,来处理异步事件和有严格时间限制的数据收发应用。其具体能力是:

- 系统应该有事先定义的时间范围内识别和处理离散事件的能力。
- 系统能够处理和存储控制系统所需要的大量数据。

开发嵌入式实时系统,就需要一个支持实时多任务的操作系统(RTOS)内核。因为

嵌入式的应用不仅仅局限于原来的只是面对系统级的需求,而是需要面对用户级的应用,满足在各个层次上尤其是消费电子产品的需求。在这个方面,嵌入式的应用系统的开发和定制变得越来越重要。传统的使用循环控制的嵌入式系统不能满足需求,目前,在中国大多数嵌入式软件开发还是处在基于处理器直接编写的阶段,而没有采用商品化的 RTOS。那么在开发嵌入式系统时无法将系统软件和应用软件分开处理,每次开发的时候都要特别定制系统软件和应用软件,开发的代价太大,成本过高。

使用 RTOS 内核,可以针对使用的处理器进行优化设计,做成一个高效率的实时多任务内核。并且上面可以根据不同处理器体系结构设计出不同的 API 接口,这些是 RTOS 基于设备独立的应用程序开发基础。

在 RTOS 基础上可以编写出各种硬件驱动程序、专家库函数、行业库函数、产品库函数,和通用性的应用程序一起,可以作为产品销售。从这个角度说,RTOS 又是一个软件开发平台。

在 RTOS 里面最关键的部分是实时多任务内核,需要实现任务管理、定时器管理、存储器管理、资源管理、事件管理、系统管理、消息管理、队列管理、旗语管理等等。实现效率高、体积小、移植功能强大、易于定制的 RTOS 是开发嵌入式系统的关键问题。

9.1.2 各种流行的实时操作系统

实时系统的实现多为微内核体系结构,这使核心小巧而可靠,易于 ROM 固化,并可模块化扩展。微内核结构系统中,OS 服务模块在独立的地址空间运行。所以,不同模块的内存错误便被隔离开来。但它也有弱点,进程间通信和上下文切换的开销大大增加。相对于传统集成化内核系统来说,它必须花费更多的系统调用来完成相同的任务。

- 可选调度策略: FIFO、轮转策略、适应性策略。

(2) QNX 提供的系统服务

QNX 提供的系统服务有:

- 多种资源管理器,包括各种文件系统和设备管理,支持多个文件系统同时运行,包括提供完全 POSIX.1 及 UNIX 语法的 POSIX 文件系统,支持多种闪存设备的嵌入式文件系统,支持对多种文件服务器(如 Windows NT/95、LAN Manager 等)透明访问的 SMB 文件系统、DOS 文件系统、CD-ROM 文件系统等。
- 设备管理。在进程和终端设备间提供大吞吐量、低开销接口服务。
- 图形/窗口支持。包括 QNX Window、X Window System for QNX、对 MSWindows NT/95 和 X Window 系统的远程图形连接。
- TCP/IP for QNX。
- 高性能、容错型 QNX 网络——FLEET,使所有连入网络的计算机变成一个逻辑上的超级计算机。
- 透明的分布式处理。FLEET 网络处理与消息传递和进程管理原语的集成,将本地和网络 IPC 统一起来,使网络对 IPC 而言是透明的。

QNX 是一个更加符合传统“分布式”概念的操作系统,目标是把整个局域网变成一个大的超级计算机,使网络的存在对用户透明,文件系统提供的服务也很丰富。但是,分布式的程度越高,意味着系统开销的增大。

(3) QNX 的开放性

QNX 具有很好的开放性,包括:

- QNX 的 POSIX 兼容性和其提供的 UNIX 特色的编译器、调试器、X Window 和 TCP/IP 都是 UNIX 程序员所熟悉的。
- 支持多种 CPU: AMD ElanSC300/310/400/410、Am386 DE/SE、Cyrix MediaGX、x86 处理器(386 以上)、Pentium 系列、STMicroelectronics 的 STPC。
- 多种总线: CompactPCI、EISA、ISA、MPE (RadiSys)、STD、STD 32、PC/104、PC/104-Plus、PCI、PCMCIA、VESA、VME。
- 各种外设:多种 SCSI 设备、IDE/EIDE 驱动器、10M/100M 以太网卡、TokenRing 网卡、FDDI 接口卡、多种 PCMCIA 设备、闪存、声卡等。

2. LynxOS

它是一个分布式、嵌入式、可规模扩展的实时操作系统,它遵循 POSIX.1a、POSIX.1b 和 POSIX.1c 标准,最早开发于 1988 年。LynxOS 目前还不是一个微内核结构的操作系统,但它计划使用所谓的“Galaxy”技术将其从大型集成化内核改造成微内核,这一技术将在 LynxOS 3.0 中引入。新的 28KB 微内核提供以下服务:核心启动和停止、底层内存管理、出错处理、中断处理、多任务、底层同步和互斥支持。

(1) LynxOS 调度策略

LynxOS 的调度策略为:

- LynxOS 支持线程概念,提供 256 个全局用户线程优先级。
- 硬实时优先级调度:在每个优先级上实现轮转调度、定量调度和 FIFO 调度策略。

- 快速正文切换和阻塞时间短。

- 抢占式的 RTOS 核心。

(2) LynxOS 提供的系统服务

LynxOS 提供的系统服务有：

- 网络和通信。由于使用 UNIX/POSIX API, Lynx 很适合于数据通信和 Internet 应用。又由于系统的开放性,网络软件很容易移植到 Lynx 上。同样, Lynx 亦提供关键的电话通信协议,使之适用于电信系统的基础架构、操作和多媒体应用。
- TCP/IP 协议栈。Lynx 自带优化的 TCP/IP 协议栈,提供高性能服务,如 TCP 头预测、高级路由算法、IP 级多址广播和链路级高速缓冲。
- Internet 工具。包括 Telnet、Ftp、tftp、PPP、SLIP、实时调度的嵌入式 Java 虚拟机、嵌入式 HTTP server、bootp、ARP/RARP、DNS 域名服务、电子邮件、Perl、电话通信协议等。
- SVR3 流。LynxOS 流机制为开发和移植基于流的驱动程序和应用提供核心支持。
- 文件系统。实时的类 UNIX 层次结构文件系统,连续结构文件、带缓冲/不带缓冲、原始分区和原始设备访问。
- 基于 Motif 的图形用户接口。
- 分布式计算资源。SCMP 与 VME 总线上的多处理结合,PCI 桥服务、Compact-PCIHot - swap Services、Lynx/HA - DDS 分布式数据系统。

3. VxWorks

VxWorks 是美国 Wind River System 公司推出的一个实时操作系统。VxWorks 是一个运行在目标机上的高性能、可裁减的嵌入式实时操作系统。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中,如卫星通讯、军事演习、弹道制导、飞机导航等。

(1) 特点

VxWorks 的开放式结构和对工业标准的支持,使开发者只需做最少的工作即可设计有效的、适合于不同用户要求的实时操作系统。VxWorks 主要有以下特点：

- 微内核结构(最小结构<8KB)；
- 微秒中断处理；
- 高效的^①任务管理。有多任务,具有 256 优先级；优先抢占和轮转调度；快速、确定的上下文转换；
- 多处理器支持；
- 灵活的任务间通信；
- 符合 POSIX 1003.1b 实时扩展标准；
- 满足 TCP/IP 网络标准；
- 可以灵活地从 ROM、磁盘或网络引导系统；
- 快速、灵活的 I/O 系统；
- 支持 MS-DOS 和 RT-11 文件系统；
- 完全符合 ANSI C 标准。

(2) 开发环境

VxWorks 的开发环境是 Tornado。Tornado 集成环境提供了高效明晰的图形化实时应用开发平台,它包括一套完整的面向嵌入式系统的开发和调测工具。

4. RT-Linux

RT-Linux 是一个嵌入式硬实时操作系统,它部分支持 POSIX.1b 标准。RT Linux 实现了一个小的实时核心,仅支持底层任务创建、中断服务例程的装入、底层任务通信队列、中断服务例程(ISR)和 Linux 进程。原来的非实时 Linux 核心作为一个可抢先的任务运行于这个小核心之上,所有的任务都在核心地址空间运行。它不同于微内核和大型内核,属于实时 EXE 体系结构。RT-Linux 是本章讨论的重点。

5. KURT-Linux

KURT-Linux 并不是为嵌入式应用设计的,不同于硬实时/软实时应用。它提出了“严格”实时应用的概念,如一些多媒体应用和 ATM 网络应用。KURT 是为这样一些应用设计的“严格的”实时系统。KURT-Linux 核心包括两个部分:内核和实时模块。内核负责实时事件的调度,实时模块为用户进程提供特定的实时服务。它不属于微内核结构。

KURT-Linux 可运行在两种状态之下:通常状态和实时状态。在通常状态下,所有进程都可以运行,但某些核心服务将带来中断屏蔽的不可预期性。实时模式只允许实时进程运行。它的特点是:

- 支持 FIFO 调度策略、轮转调度策略和 UNIX 分时调度策略;
- 增加了 SCHED-KURT 调度策略,这是一种静态调度策略,使用一个特殊的调度文件记录预先定义好的待调度进程的参数。

9.1.3 实时系统的设计

在此简要探讨实时系统的设计。由于实时系统在设计时与应用的关系非常密切,所以有许多分类的方法。各种分类的侧重点不同。一是分为周期性地和非周期性的。周期性的就是系统通过传感器或其他设备周期性地探测外部环境的变化,在周期内对探测到的变化作出反应。比如化工厂中反应炉的控制。非周期性的就是外部事件是循环发生的,但不是有规律的或者是突发事件。比如一架客机飞入空中交通管制范围所产生的事件。二是分为硬实时和软实时。硬实时系统就是系统必须及时地对事件作出反应,绝对不能发生错过事件处理 deadline 的情况。在硬实时系统中一旦发生了这种情况就意味着巨大的损失和灾难。比如控制核电站的系统,如果没有对堆芯过热作出及时的处理,后果将不堪设想。而在软实时系统中,当系统在重负载的情况下允许发生错过 deadline 的情况,而不会造成非常大的危害。比如在通信系统中允许 105 个电话中有一个接不通。对于软实时系统基于优先级调度的算法可以满足要求,提供高速的响应和大的系统吞吐率;而对于硬实时系统则完成 timely response 是必须的。这两类系统的区别在于调度算法。另外,还可以分为专用系统和开放系统,以及集中式系统和分布式系统。

由于上述和实时系统的应用有关的特点,导致在实时系统设计时面临着与原来的通

用系统不同的考虑因素。首先在实时系统中最基本的是系统应提供对时间正确性进行指定的方法,也就是在实时系统中不管是用户还是开发人员都需要系统提供一种指定时间尺度的方法。比如在有的实时系统中指定每隔一段时间就运行一段程序,或者是提供指定程序必须在某个时间点之前完成的方法等等。在实时系统中这是最基本的要求。这时通用系统中的功能就完全不适用了。例如 UNIX 和许多类似的通用系统中都提供一种延时的手段,但这种方法在实时系统中就无法达到要求,因为这种延时手段无法保证应用能够在 deadline 之前完成计算。第二是实时操作系统的设计或选用。在现代的实时系统当中一般都有实时操作系统存在。因为操作系统使系统的设计更加简便,保证系统的质量以及能够提供其他通用操作系统所提供的服务。这样,实时操作系统就面临着更高的设计要求。第三是实时系统的体系结构设计。实时系统的体系结构必须满足以下条件:

- 高运算速度;
- 高速的中断处理;
- I/O 吞吐率高;
- 合理的处理器和 I/O 设备的拓扑连接;
- 高速可靠的和有时间约束的通信;
- 体系结构支持的出错处理;
- 体系结构支持的调度;
- 体系结构支持的操作系统;
- 体系结构支持的实时语言特性。

另外,由于实时系统很多应用于一些关键性的场合,系统的稳定性和容错也非常重要。还有实时系统很多在自然形态上有分布式的特点,所以还要考虑到实时的分布式应用。

此外实时的通信也是实时系统的一大要求,很显然,在分布式实时系统中实时通信是最基本的。实时通信必须满足逻辑正确和要有确定的延迟时间(或通信延时时间的上限),其中包括建立通信的延迟和消息传递的延时。

作为实时系统,传统性能的衡量标准对它是不适用的。对传统通用系统的要求是系统吞吐量大,有合理的响应速度,对每个系统用户相对公平地进行计算资源的分配。然而在实时系统中,以上这些要求都不再适用或者是不再占重要位置。在实时系统中,系统的一切动作都以实时任务为中心。实时的数据吞吐取代了以吞吐量为目标的标准。对硬实时应用的优先响应取代了对每个用户的恰当的反应速度。系统的计算资源和其他外设资源必须优先满足实时应用的要求。针对实时系统新的要求,必须以实时的进程调度在实时操作系统中是一个关键性的问题。

实时操作系统的实时进程调度的根本要求是保证实时任务的时间正确性。此外实时操作系统的进程调度算法必须保证系统是可以事先定义的和易维护的。实时任务的时间正确性以实时任务是否能够总是满足 deadline 为标准。上面提到过实时系统的类型可以分为周期性和非周期性,实时操作系统的调度理论就是按照这个分类来考虑的。

1. 实时进程调度算法

在控制系统的应用中,实时任务一般来说是各自独立的、周期性的,也有一些是由外部事件来触发的。这样,用户可以设计一种最简单的调度方法:静态的周期性调度。这种调度算法的基本思想是将处理器的时间分为“帧”。一帧就是当一个系统内部时钟触发时钟中断时使用一系列不同的标准来衡量系统的性能、间隔。每个实时任务都在帧中占一段时间。仔细的设计帧的大小可以保证系统中所有的实时任务都能在 deadline 之前完成。而事件触发的实时任务则在每个帧中最后一个周期性实时任务完成之后到帧结束这段时间内得以运行。这种算法在系统相对简单,任务数少,又可以事先定义任务执行顺序的情况下很有效。还有一个更新的版本,就是将一个大帧分为若干个级别较低的小帧。这个算法又称为同步任务执行算法。

另外一种简单的调度算法就是 FIFO。也就是将系统中所有的任务组织成一个队列。先到先服务。在简单的和静态的实时系统中可以事先安排好任务的执行顺序,预先组成队列。这里要保证的是处理器执行的速度。当任务计算量一定,而处理器的执行速度合适时,FIFO 算法完全可以满足实时的需要,而且自然地保证时间正确性。在设计队列和应用时要极力避免关键任务长时间的阻塞。这种算法无疑是不精确的和适应性差的。

优先级队列算法。这种算法从 FIFO 发展而来。给每个任务设定优先级,然后在 FIFO 中按照优先级排列。这种算法保证了高优先级任务的完成,但是对于低优先级的任务很可能无法满足时间的正确性。而且对低优先级的任务来说等待的时间是无法预知的。用户无法知道在什么情况下会发生时间正确性上的错误,而且必须在设计时仔细地检查任务优先级的设定,要保证不会因为低优先级的任务无法按时完成而破坏整个系统的完整性。这个算法也是与特定应用有关的。当环境或情况变化时系统无法随之变化。

2. 多任务算法

以上的调度算法都是独占的,即任务运行时不允许别的任务抢先。完成一个任务后才能完成下一个。下面是多任务的算法:

(1) Rate Monatomic/Pacing 算法

此算法是基于静态优先级调度协议的方法。此算法给系统中每个任务设置一个静态的优先级。通过计算任务的周期时间和任务需要满足的截止时间的长短,就可以设定这个优先级。周期越短,截止时间越紧迫,优先级越高。按周期或计算要求将任务细分成程序块,整个任务的执行开销为 $P/(C/CP)$ (P 是周期, C 是计算开销, CP 是块的大小)。这样就将整个任务的执行按紧迫性要求分散在整个周期内。这种算法允许系统以多任务方式执行。

(2) Deadline Driven 算法

Deadline Driven 算法提供动态的优先级。因为此算法根据任务满足 deadline 的紧迫性来修改任务的优先级,以保证最紧迫的任务能够及时完成。当系统负载相对较低时,这种算法非常有效。但是当系统负载极端沉重时会引起大量的任务发生时间错误。甚至可能导致 CPU 的时间大量花费在调度上,这时系统的性能还不如 FIFO 方法。根据计算当系统负载超过 50% 时系统性能急剧下降,所以实际上没有哪个实时操作系统使用这种

方法。

(3) Priority Ceiling 算法

这种算法用于抢先式多任务的实时操作系统。该算法的基本思想是在系统中使用优先级驱动的可抢先的调度算法。也就是系统首先调度高优先级的任务运行。低优先级的任务在高优先级的任务运行时不能抢先。CPU 由高优先级进程独占。当中断发生时,正在运行的任务被中断,进入中断处理。如果中断引起的操作是属于一个较低优先级的任务,那么为了保证中断被及时处理,此低优先级进程暂时继承原来当中断发生时正在运行高优先级任务的优先级。当处理完关键区域后,此低优先级任务恢复原来的优先级并被挂起,然后恢复原来高优先级任务的运行。这种算法保证了高优先级的任务不会被低优先级的任务所挂起,即避免“优先级倒挂”现象的出现。

实际上现代实时操作系统中往往将上面提到的调度方式结合起来,以适应各种应用的不同需求。比如操作系统提供基于优先级的抢先式多任务,也同时允许时间片机制。

3. 实时系统的内存和外围设备管理

在计算机系统中,内存管理主要有两个基本的任务。第一是为系统中运行的许多实时或非实时任务提供共享的内存(也是最基本的)。第二是提供虚拟内存。虚拟内存就是将系统的一部分后备磁盘空间作为实际物理内存的扩充,由操作系统管理,对应用程序完全透明,应用程序所能感觉的只是系统的物理内存变大了。这对于程序员来说是很方便的。但是享受虚拟内存好处的同时也是有代价的,那就是虚拟内存将一部分最近未用的程序代码和数据换到硬盘上,当要使用时从硬盘读出的速度要比内存慢很多。这就引起了问题,应用程序不知道什么时候会被调换上物理内存,也不知道从硬盘中换回物理内存所花的时间。这样系统响应的时间就变成不确定的了,这在硬实时系统中是致命的。实际上有很多实时系统是没有硬盘的。它们就是本章所讲的嵌入式实时操作系统。一般来说这样的系统组成规模都尽可能小,所以一般没有后备存储器。因此对于实时操作系统来说,内存管理必须高效率 and 开销必须是可预见的。一种解决方法是预先分配内存,就是在系统构造或编译时为每个任务指定其使用的内存空间。这种方法对于硬实时系统来说是很合适的。而且嵌入式实时操作系统很多都是在 ROM 中运行,仅仅只有需要变化的数据才放在 RAM 中,这种系统在组成上无疑是静态的。另一种方法是系统可以有虚拟内存,但必须给实时任务提供方法,以便将实时任务“锁”进内存。也就是系统在管理虚拟内存时,不将“锁”住的内存块换出物理内存。这样可以提供比较好的响应速度和操作可预见性。但是,这不适合硬实时系统,因为虚拟内存引起的问题并没有解决。

其实实时操作系统的内核仅仅需要负责为程序分配内存,动态地分配内存和回收内存。为程序分配内存指当嵌入式系统从主机上下载程序或带硬盘的系统从硬盘上获得程序代码时,操作系统内核必须为程序代码和数据分配内存。这种分配规模比较大,但是在系统运行时较少发生。一般只有在系统启动或复位时发生。第二种内存管理要求就是在应用运行时动态地分配和回收数据。一般的实时应用总是尽量避免动态地分配内存和释放内存,因为这样会增加系统的不确定性,使系统的稳定性下降。第二种要求还用于实时任务之间的通信。系统分配信号量、消息队列等等。第二种要求必须简单和高效率地实现。

实时系统的外围设备管理相对比较简单。因为对于嵌入式实时系统来说,系统中很

少有外设要管理。即使有也是与某个特定的任务相联系,不需要操作系统来干预。但是对于实时的事务处理来说,外围设备的管理是不可回避的问题,主要是防止实时任务的死锁。解决死锁的经典算法是银行家算法,但是在实际应用中这种方法却因为开销太大而很少用。现在有很多针对某一类应用的死锁解决方法,但主要思想还是类似的——就是破坏死锁的循环条件。大多数系统对于死锁都是采用了这种所谓的“鸵鸟思想”,因此很多商用实时操作系统中死锁的避免是由应用设计人员完成的。

4. 实时系统的通信

在实时操作系统中实时的进程间通信也十分重要。特别是多处理器的情况,通过系统总线或局域网进行通信。这里主要讨论在分布式系统中的实时通信。

在实时分布式系统中的通信有着特殊的要求。与普通的分布式系统不同,也与基于总线的多处理器或单处理器不同。由于系统中没有可以用来通信的共享内存,进程之间的通信完全依靠网络协议来实现,而现有的网络协议没有一种是为实时应用设计的。对于实时通信来说当然是网络速度越快越好了,但光有高速的网络通道还是不够的,必须有可预见的和有确定开销时间的网络协议。在分布式系统或网络上实现实时应用的关键点在于实现可预见的和有确定开销的网络协议。

首先看一下局域网。最常用的 Ethernet 从本质上来说就是一个随机的协议。载波侦听/冲突检测本身是一个随机的协议。想发送数据的机器可能与网络上的其他机器发生冲突。一旦发生冲突,就放弃发送,随机等待一段时间后再次发送。不可能给出最坏情况的上下限,这样就造成了发送数据所需时间的不确定性,而这恰恰是实时操作系统中不允许的。另外一种局域网的协议是令牌环网。令牌环网的机制决定了它可以得到发送数据的开销的上限。每个机器当要发送数据时,首先要得到令牌,得到令牌的机器可以发送数据。局域网的长度有限,令牌在网上传输的速度是一定的,令牌环网上所能容纳的字节数也是确定的,所以可以确定发送一个包最多所需要的时间。这样的协议是可以被实时操作系统接受的,且令牌环网还可以允许带有优先级,允许优先服务高优先级的任务的通信请求。要实现这一点,只需要在包头加上一个代表优先级的字段即可。当发送数据时,只有优先级高于当前传输的包的机器才能在当前包发送完后立即发送。这时发送一定大小的包所用的最大开销就变成仅仅指定为最高优先级的包才能满足这个上限。

用于局域网中的通信手段还有 TDMA(时分多路复用)。这种机制中每个机器都在网络上有自己的时间片。通信被组织成固定大小的帧。每个帧含有 n 个时隙。时隙与网络上的机器对应。这样就避免了机器发送时的冲突。每个机器都有确定的带宽。现代的通信技术提供了高带宽的传输能力,局域网内的实时通信基本成熟。在广域网上的实时通信也走向应用。通常这种通信是面向连接的。当建立连接后,实时通信的质量是得到保证的。通信质量包括最大延迟时间、包发送速度的最大变化率、最小带宽等等。为了保证通信质量,应事先分配好内存缓冲、表空间、CPU 时间、通信信道等。一般用户要支付占用这些资源的费用,无论他是否使用这些资源。广域网上实时通信的最大问题是在广域网上丢包率非常高。通常解决这个问题的方法是发送者使用一个时钟,当超时,重发数据。但这样会在系统中引起不可预见性。发送一个包的开销不可预见,这在实时操作系统中是不允许存在的。一个简单的解决方法是发送者每次都将一个包发送两次或更多

次,这样通信的质量得到了保证,无非浪费一些带宽而已。

实时操作系统分为嵌入式和普通系统两种。虽然这两种系统的规模、应用、性能及可靠性要求都不同,但是这两种实时操作系统一般是基于微内核的和模块化的。系统可以在最小规模下工作,操作系统仅仅提供一些最基本的服务。而大量的服务则由开发商提供的作为应用运行的系统级任务完成。所以商业化的实时操作系统遵循前面提到的开放的系统标准,系统高度模块化。实时操作系统的内核必须满足以下一系列特殊的要求。

- 实时操作系统的内核必须非常小。操作系统的内核小,可以为用户应用留出空间。另外操作系统所占用的内存小也代表了系统效率的提高。
- 操作系统内核必须是可重入的。这一点对于实时系统来说特别重要。因为不可重入的内核必然带来慢速的中断响应和不可预料的操作时间。
- 系统能够快速地进行任务切换。快速的任務切换在满足高优先级的任务抢先时,提供了完成 deadline 的保证。
- 能够快速响应外部中断。正如前面所提到的操作系统的内核必须可重入。将临界区缩到最小可以提供最大的中断响应效率。
- 尽量减少禁止中断的时间。
- 要提供固定或可变的内存管理机制。内存管理机制在内核中的实现要求尽可能的简单。高级的功能可以由系统级进程来完成。
- 在系统级任务中提供可高速访问的文件系统。
- 系统维护一个满足应用要求的实时时钟。因为对实时时钟的要求是随应用程序的要求而变化的。而且实时时钟是与硬件密切相关的,所以一般是操作系统提供一个接口,真正的管理是要求硬件提供者或用户自己提供的。
- 实时操作系统必须提供一种合适的进程调度方法。一般商业化的实时操作系统像 pSOS 和 QNX,都是提供的基于优先级的抢先式的进程调度算法。
- 实时操作系统必须通过使用实时时钟的接口来提供 time_outs 和 time_alarm。
- 实时操作系统还应该提供允许应用任务来自己修改系统。

这样的实时操作系统应该使用高级语言编写,而且不能带有任何与硬件有关的特性。操作系统可以向下定义一组接口,由用户或硬件的提供者来提供实际操纵硬件的程序。包括系统中设备的初始化和使用。实时时钟也必须这样管理,这样才能保证不用为每一种嵌入式系统设计不同的内核。这样嵌入式实时操作系统内核中需要完成的仅仅是内存管理,进程调度和定义一系列的接口,以及使用接口的基本功能。更高级一些,可以提供一种进程间通信的方法。这样的实时操作系统内核已经是完善的了,操作系统内核没有必要将所有设备都自己管理,况且在设计内核时谁都想不到会出现什么样的设备。因此,在嵌入式系统中一般允许应用直接访问底层的硬件设备。

5. 用 Linux 实现实时系统

如何通过 Linux 实现实时系统呢?一般说来,有两种途径来实现,第一种方法是通过 POSIX 方法,另一种方法是通过底层编程的方法实现。

POSIX 是标准化类 UNIX 操作系统必须具有的特征和接口的运动方式。使用 POSIX 的思想就是为了促进为 UNIX 编写软件的可移植性,从而使 UNIX 程序员的工作更为

容易。并且在 POSIX 里面加进了对实时性的扩展,如 POSIX.1b 或者 IEEE 1003.1b 已经加入到 POSIX 标准中。在这些实时性扩展里面定义了一些工具,如信号、内存锁定、时钟和计数器、消息队列以及优先级抢先调度等等。不过通过 POSIX 基础来标准化实时操作系统并不被开发者所看中,因为 POSIX 系统调用反映了 UNIX 系统调用的复杂和笨重。

现在已经有很多程序员在 Linux 下做实现 POSIX.1b 的工作,并且已经实现了 POSIX 的内存锁定工具和决定调度算法。另外,计数器函数和 POSIX.1b 信号还没有完成,而且信号和消息队列也还没有实现。在 Linux 中实现一个进程内可以有多个线程,共享相同的地址空间。但是可以这样想像,即使把 POSIX.1b 函数完全移植到 Linux 上来,而 Linux 内核却不能被实时任务所抢占。那么这样的实现恐怕还是没有意义的,无法实现真正意义上的“硬”实时。

另外一条实现实时的途径就是使用底层编程的方法。在 Linux 内核中作出最少的改动,在 Linux 内核中影响实时性能的地方增加控制,将控制权交给 RT-Linux 内核。因此,RT-Linux 只需提供一些必要的功能,如底层任务创建、中断服务程序,并且为底层任务、ISR 和 Linux 进程之间进行通信排队。这就是下面要讲述的 RT-Linux 的实现机理。

9.2 实时 Linux——RT-Linux

9.2.1 RT-Linux 综述

RT-Linux 是美国新墨西哥州大学计算机科学系 Victor Yodaiken 和 Michael Branano 开发的。它在 Linux 内核的下层实现了一个简单的实时内核,而 Linux 本身作为这个实时内核的优先级最低的任务,所有的实时任务的优先级都要高于 Linux 本身以及 Linux 下的一般任务。

RT-Linux 的性能是“硬”实时的。在一台 386 机器上,RT-Linux 从处理器检测到中断处理程序开始工作不会超过 15 微秒。对一个周期性的任务,在 35 微秒内一定会执行。如果是普通的 Linux,一般是在 600 微秒内开始一个中断服务程序,对周期性的任务很可能会超过 20 毫秒(20 000 微秒)。

1. RT-Linux 介绍

如果想用一台个人电脑去控制照相机、机器人或者一台科学设备,考虑使用 Linux 是很自然的,它可使用户从这一开发环境中得到益处。但是 Linux 不能可靠地运行那些硬实时设备。用一个小实验来说明这个问题。将一个扬声器挂装在并口的一个针脚上,然后运行一程序,此程序被绑定在这一端口上。如果只运行这一个程序,扬声器将奏出美妙而平稳的声音。当 Linux 每两秒更新文件系统时,用户会发现声音会有些小变化。如果移动鼠标点两个以上的视窗,声音会变得不规则。如在一个视窗中运行 netscape,声音会变得时断时续。

像大多数操作系统一样, Linux 优化每一功能, 并试图给每一进程公平分配时间片。这对一般目的的操作是很重要的, 但对实时操作, 计时与预测功能比一般功能要重要得多。例如, 一个照相机每微秒要填充一缓冲区, 这样负责读取缓冲区的进程只要有瞬时的延迟就会造成数据丢失。如果用精确的中断控制平板印刷机的步进电机开关来减小颤动并将打印片置于正确的位置, 电机的一个瞬时延迟会造成不可挽回的失败。如果一个控制使化学实验立即停止的进程非要等到 netscape 刷新视窗之后才能运行, 那么将会发生什么后果?

这表明重新设计的具有实时功能的 Linux 系统将能承担很多工作, 并能依照用户的本意来运作。这里需要一种定制的包括 Linux 主要部分的专用操作系统, 而不是去掉支架的通用操作系统。现在所做的就是塞入一个小的、简单的、实时的操作系统——嵌入式 Linux。Linux 变成了当无实时任务时运行的一个任务, 并且无论何时, 只要实时任务需要 CPU 时, Linux 都要被先清空。Linux 一般并不明白实时操作系统怎样运行程序、发出中断及控制硬件设备。但实时任务能以高水准的精度运行。例如在 P120 测试系统中, 可以用 20 微秒的误差使一系列任务顺序运行。

2. 使用 RT-Linux 2.0

这里看一个例子。假设用户想编写一个应用软件来控制一设备采集数据并将采集来的数据存储到一个文件中。隐藏在 RT-Linux 之后的设计思想是: 应用硬实时约束, 将实时程序分割成短小简单的部分, 较大的部分承担较复杂的任务。

根据这一原则, 将应用程序分为两部分。硬实时部分被作为实时任务来执行, 并从外部设备拷贝数据到一个叫做实时有名管道(FIFO)的特殊 I/O 端口。程序的主要部分将被作为标准的 Linux 进程来执行。它将从实时栈中读取数据, 然后显示并存储到文件中, 实时部分将被写入内核。Linux 容许不重新启动而编译并启动内核。一个模块的代码是从模块定义和包含它的头文件开始执行的。由此, 先包含实时头文件 `rt_sched.h` 和 `rt_fifo.h`, 再声明一个 `RT_TASK` 结构:

```
#define MODULE
#include<Linux/module.h>
/* 对实时任务总是需要下面的头文件 */
#include<Linux/rt_sched.h>
#include<Linux/rt_fifo.h>
```

```
RT_TASK mytask;
```

`RT_TASK` 结构包括指令与数据指针和任务的进度信息。此结构定义在第一个头文件中。目前, RT-Linux 只有一个简单的进度表, 将来这些进度表会变成可执行的模块。Linux 通过叫做实时有名管道(FIFO)的特殊队列来处理实时任务的先后顺序。设计实时有名管道是为了使实时任务在读和写数据时不被阻塞。图 9-1 就说明了实时有名管道。

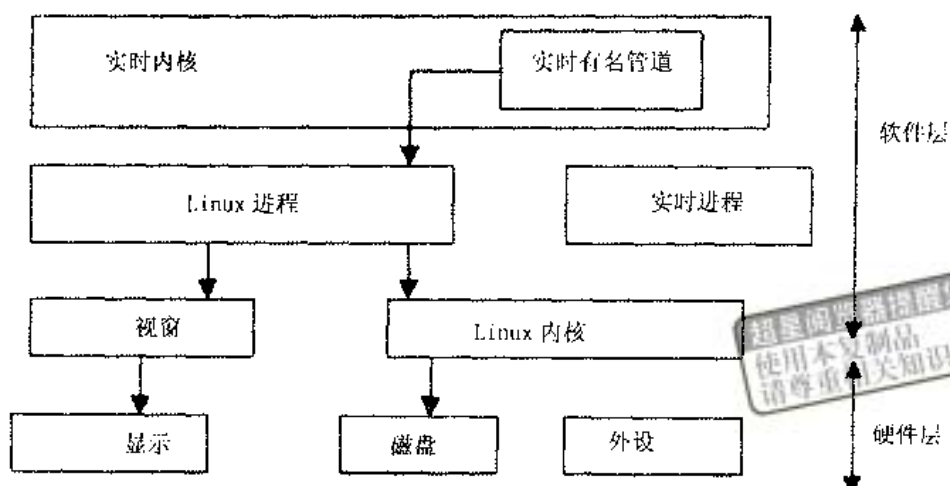


图 9-1 实时有名管道

例 9-1 的作用是循环地从外设读取数据,向实时有名管道写数据,并等待一定的时间。

例 9-1 读取外部设备数据

```

/* 主程序部分 */
void mainloop(int fifodesc)
{ int data;
/* 该循环从外设取数据并发送 */
/* 进入管道 */
while(1){
data = get_data();

rt_fifo_put(fifodesc, (char *)&data, sizeof(data));

/* 放弃 CPU 直到下一个时间片 */
rt_task_wait();
}
}

```

所有的模块都包含一个初始化函数。这个初始化函数将会记录当前时间,初始实时任务结构将任务加入进度表。`rt_task_init` 函数初始化任务结构并为其分配、传递参数。在这儿,此参数是实时有名管道的描述符。`rt_make_periodic` 函数将新任务放在循环调度队列中。循环调度的意思就是说,任务每隔一定时间就要被调度一次。但只有当中断激活它时,任务才会执行,见例 9-2。

例 9-2 初始化模块

```

/* 任何模块都需要这一函数。当模块加载时调用它 */
int init_module(void) /* 初始化函数 */
{

```



```

#define RTFIFODESC 1
RTIME now = rt _ get _ time(); /* 取当前时间 */

/* 'rt _ task _ init'用 RT _ TASK 结构关联 */

/* 并设定几个实参:priority level=4,stack size=3000 bytes */
/* 设定按 1 键回 mainloop */
rt _ task _ init(&mytask,mainloop,RTFIFODESC,3000,4);

/* 将'mytask'标定为周期性的 */
/* mytask 将会占用 25000 个时间片 */
/* 第一部分从现在开始执行 1000 个时间片 */
rt _ task _ make _ periodic(&mytask,now+1000,25000);

return 0;
}

```

Linux 也要求每个模块有挂起例程。对实时任务而言,须保证死任务不再被调用。清除模块的方法见例 9-3。

例 9-3 清除模块

```

/* 当模块被卸载时,被挂起的例程将继续执行 */
void cleanup _ module(void)

{ /* 杀死实时任务 */
rt _ task _ delete(&mytask);
}

```

下例是模块的最后一部分,用户需要一个能像普通的 Linux 进程一样运行的程序。在本例中,这一程序将从有名管道中读数据并将数据拷贝到标准输出文件。

例 9-4 非实时任务程序

```

#include<rt _ fifo . h>

#include<stdio . h>

#define RTFIFODESC 1
#define BUFSIZE 10
int buf[BUFSIZE];
int main()
{ int i;
int n;

/* 用 1000 字节建立有名管道 1 */
rt _ fifo _ create(1,1000);

for(n=0;n<100;n++)

/* 从有名管道读数据并打印 */

```



```

rt_fifo_read(1,(char *)buf,BUFSIZE * sizeof(int));
for(I=0;I<BUFSIZE;i++)
{
    printf("%d",buf[i]);
}
printf("\n");
}
/* 撤消有名管道 1 */
rt_fifo_destroy(1);

return 0;
}

```



主程序完成显示数据,传送数据到 Internet 等工作。所有这些都是非实时的。有名管道必须足够大以免数据溢出。当发生某个管道出现数据溢出时,该管道可以被检测出来,此时,应该使用另一有名管道来通知主程序关于数据溢出的情况。

3. Linux 或经简单改进的 Linux 都不能运行实时任务的原因

虽然 Linux 系统每隔一定的时间挂起进程,但不能保证间隔一过进程就恢复。用户进程会在不可预测的时刻被占用,从而不得不等待 CPU 分配时间。即使让关键任务具有最高特权也没有用,部分原因是 Linux“公平”时间分配的调度算法。这一算法试图保证公平分配给每一用户程序占用 CPU 时间。但是,设计的目的是保证实时任务只需要 CPU 时就能得到它,而不论这有多么不公平。Linux 的虚拟内存方式也增加了不可预测性。任何用户进程的页面在任何时刻都能被交换到硬盘上。在 Linux 中将需要的页面返回到 RAM 中都要花掉一段不确定的时间。

在这些问题中,部分问题是简单的或者是较简单的。创造新一类有较好实时性能的 Linux 进程是可能的。用户可以改变调度算法使实时进程循环地或定时地被调用,并能锁定内存中的实时进程,从而使其页面不被交换出去。其实,这两种方法是 POSIX.1b-1993 标准的一部分。这个标准定义了实时进程,而它也被 Linux 所采用。在新版的 Linux 中,系统调用提供了如下操作:锁住内存中的用户页面;修改调度规则使基本进程具有特权;对信号更可靠的处理。但该标准不能解决所有的问题,它并不是真的试图解决所关心的“硬实时”的实现问题,其目标是实现所谓的软实时程序。在视窗下播放视频文件的程序就是一个软实时任务的好例子。

对硬实时问题来讲,POSIX 标准有几个缺陷:

- 重量级的 Linux 进程通过进程开关与有效的内务操作联系在一起。虽然 Linux 在进程开关转换方面相对较快,但它也要花几百微秒。这使调度不可能完成诸如每隔 200 微秒从传感器取样的实时任务。
- Linux 继承了标准 Unix 内核进程最优先的技术。也就是说,当进程进行系统调用时,无论其他进程有多高的特权它都不会被迫放弃 CPU。对于编写操作系统的人来讲,这将大大减少复杂的同步问题的出现。然而对于想运行实时程序的

人来讲,当内核支持不重要的进程而重要的进程却不能被调用是不妙的。例如,如果 Netscape 调用 `fork()`,在 `fork()`调用完成之前任何其他进程都不能运行。

- Linux 不能中断代码的关键部分。这就意味着实时中断将被延迟直到当前进程完成它的关键部分,而不论它的权限有多低。

考虑下面这条代码:

```
temp = qhead;
qhead = temp->next;
```

假设内核到达第一行前,`qhead` 包含了一个数据结构的地址,这一数据结构在队列中是唯一的。`qhead->next` 为 0。现在假设内核完成了第一行并计算了 `temp->next` 的值(这里为 0),然后被中断,这一中断导致一新成员加入队列。当中断完成后,`qhead->next` 不再为 0,但当内核例程继续时,它将其赋 0 值,并丢失新成员。为了预防这类错误,当执行这些关键部分时,Linux 内核扩展使用“cli”命令来清除中断请求。本例中在内核例程修改队列前不能被中断,而只有当操作完成后中断才能恢复。这就意味着有时中断会被延迟。由于关键部分而造成的延迟是很难计算的,用户将不得不仔细检查每个驱动程序,对操作系统其他部分进行正确的估算。修改 Linux 内核成为实时内核以减少中断的延迟需要重写内核代码。实时 Linux 使用了简单而有效的方法来达到实时的效果。

4. RT-Linux 工作的原理

基本的方法是使 Linux 在实时内核的控制下运行。当有实时任务时,实时操作系统运行其中的一个。当没有实时任务时,实时内核调度 Linux 运行。因此,Linux 是实时内核最低权限的任务。

消减掉实时内核中与中断相关的 Linux 例程,可以解决 Linux 不能被中断的问题。例如,无论何时 Linux 调用 `cli()`例程,软件中断标志位都将被置 0。实时内核通过标志位的状态及中断标志来捕捉中断,传送中断到 Linux 内核。

当然,即使中断对 Linux 无效的话,它们对实时内核也是有效的。Linux 内核例程调用 `cli()`清除软中断标志位。当中断发生时,实时执行程序将捕捉它并决定如何做。如果中断导致实时任务,实时执行程序将保存 Linux 状态并立即执行实时任务。如果中断仅仅需要被传送到 Linux,实时执行程序将设置一标志位表明这是一个等待中断,然后恢复 Linux 而不执行中断处理程序。当 Linux 恢复中断时,实时执行程序将执行所有的等待中断并导致相应的中断处理程序执行。

实时内核自身并无优先权,但由于其例程非常快而且简短,并不会导致大的延迟。在奔腾 120 上测试,最大的延迟小于 20 微秒。

实时任务运行在内核特权级是为了与计算机硬件更接近。它们为代码和数据调整内存分配。否则,当任务要求新的内存或写代码页时,用户将面临不可预测的延迟。实时任务不能使用 Linux 系统调用或直接调用例程或在内核中存取一般的数据结构,因为这将引起系统的不兼容。在图 9-2 中,内核调用 `cli()`改变队列,但这并不影响实时任务的运行。因此不能允许实时任务直接进入队列。然而,又确实需要一种方法使实时任务能与内核和用户任务交换数据。例如,在一个数据采集器中,用户可能需要传送实时任务从网络上采集的数据,或当在屏幕上演示它时将它写入一文件中。

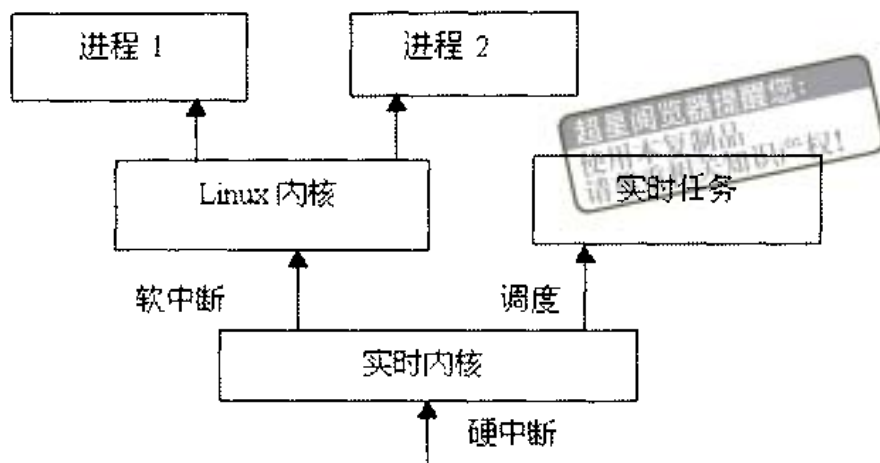


图 9-2 实时 Linux 工作原理图解

实时进程和一般 Linux 进程之间通过实时有名管道(FIFO)传送信息。实时有名管道像实时任务一样从不换页。这将减少由于翻页而造成的不确定的延迟。实时有名管道用来疏导实时任务。

再来看看实时内核怎样跟踪实时任务。当实时系统进行调度时,会在时钟中断的速率和稳定性之间进行折中。典型的,当时钟中断处理程序运行时,处于睡眠状态的任务就会恢复。越低的时钟中断速率占用越少的系统资源。实时 Linux 通过对标准时钟中断补充一个高间隔的一次性定时程序来解决这一问题。当需要时,定时中断处理程序会准时唤醒任务。

9.2.2 RT-Linux 的实时内核

RT-Linux 实现了一个在 Linux 本身内核之下的一个小而精巧的内核,这个内核只需要完成底层任务创建、中断服务程序,并为底层任务、ISR 和 Linux 进程之间进行通信排队的工作,将原来的 Linux 作为实时内核下的一个随时可被实时任务抢占的优先级最低的任务。因为 RT-Linux 工作是和 Linux 相结合进行工作,因此它可以设计得很简单,大部分的应用都不需要考虑,可以由 Linux 来完成。因为简单,使它不容易出现错误,即使出现错误也容易得到更正。同时因为简单,并且完全抢占 Linux 内核任务,使它的响应速度特别快。

1. 模块化的设计方案

RT-Linux 对 Linux 内核做了修改,做成一个小而简单的实时内核,在现有的 Linux 内核下运行。对 RT-Linux 的调度方法、用户实时任务的工作是通过 Linux 的可导入模块的方式进行的。可以自己实现一个实时调度算法,作为 Linux 的模块插入到内核运行空间,作为实时任务的调度策略。用户实时任务的编程也是通过模块编程来实现的。

2. 和 Linux 内核的结合

图 9-2 的 RT-Linux 工作方式的图解中采用的双内核系统有很多好处。RT-Linux 应用中存在有两个域,一个是实时域,一个是非实时域。放在实时域的函数能满足其实时

的要求,不过这些实时任务必须简单,因为可用的资源受到了限制;非实时域的函数可以利用整个 Linux 的资源,不过不能提供任何实时性能。在两个域之间可以通过多种途径进行通信,如 FIFO、共享内存等。

3. 利用 Linux 内核的好处

通过 RT-Linux 内核和 Linux 内核的结合,RT-Linux 可以利用存在 Linux 的非实时域函数的长处:

- Linux 提供了现代操作系统、环境的方便和强大功能,如网络服务、GUI 系统、C/C++/Java 开发工具和标准的编程接口。
- 利用 Linux 及 Linux 下开发环境快速发展的优点。因为 RT-Linux 对 Linux 所做的改动比较少,RT-Linux 可以很方便地移植到 Linux 较新的版本中,升级速度较快。

4. 与 Linux 进行通信的方法

在 RT-Linux 的实时任务并不能直接调用系统调用,它必须通过特定的方法和 Linux 进程进行通信。RT-Linux 提供了三种通信的方法:

(1) 共享内存

在 RT-Linux 启动的时候,通过指定给内核一个 mem 参数决定内核可以使用的内存大小,空出来的内存空间用于实时任务和 Linux 进程进行通信的共享内存。在 RT-Linux 任务中通过/dev/mem 设备在这段内存中寻址,Linux 进程也通过读取这段内存的数据获得实时任务提供的信息,这样完成实时任务和 Linux 进程之间的通信。

(2) FIFO 设备

该方法利用一种特殊属性的管道进行通信。通过在/dev/下面创建 FIFO 的字符设备。实时任务可以往这个字符设备写数据,非实时任务可以从这个设备中读取数据。对 FIFO 设备的读写是不同步的,非实时任务对设备的读取并不会影响实时任务对它的写。

这种机制是 RT-Linux 应用中使用最广泛的通信方法,比如在一个数据采集系统上,通过实时任务将数据采集并处理,然后写到 FIFO 设备中,通过一个在 X Window 环境中的图形界面程序读取 FIFO 的数据,然后显示在 X Window 窗口中。这样,既保证了对数据的实时处理,又提供了友好的监控界面。

(3) mbuf 驱动程序

它是由 Tomasz Motylewski 提供的一个使用共享内存的驱动程序,用来实现核心内存空间和用户内存空间之间的共享。通过使用 mbuf 提供的 mbuf_alloc() 函数给申请的内存取一个名字,mbuff 驱动程序使用一个链表通过这个名字来管理这些申请的内存。通过这个驱动程序可在包括 RT-Linux 任务的 Linux 内核内存空间和用户内存空间之间共享内存。

9.2.3 RT-Linux 的实现机理

在 9.2.1 中概述了 RT-Linux 的基本原理,在本节将展开讨论。

RT-Linux 对 Linux 内核进行改造,将 Linux 内核工作环境作了一些变化,如图 9-3 所示。在 Linux 进程和硬件中断之间,本来由 Linux 内核完全控制,现在在 Linux 内核和硬件中断的地方,加上了一个 RT-Linux 内核的控制。Linux 的控制信号都要先交给 RT-Linux 内核进行处理。在 RT-Linux 内核中实现了一个虚拟中断机制, Linux 本身永远不能屏蔽中断,它发出的中断屏蔽信号和打开中断信号都修改成向 RT-Linux 发送一个信号。如在 Linux 里面使用“sti”和“cli”宏指令来屏蔽和中断。这是通过向 x86 处理器发送一个指令,而 RT-Linux 修改了这些宏指令,让 RT-Linux 里面的某些标记做了修改。也就是说将所有的中断,分成 Linux 中断和实时中断两类。如果 RT-Linux 内核收到的中断信号是普通 Linux 中断,那就设置一个标志位;如果是实时中断,就继续向硬件发出中断。在 RT-Linux 中执行 sti 将中断打开之后,那些设置了标志位表示的 Linux 中断就继续执行。因此,cli 并不能禁止 RT-Linux 内核的运行,却可以用来中断 Linux。Linux 不能中断自己,而 RT-Linux 可以。

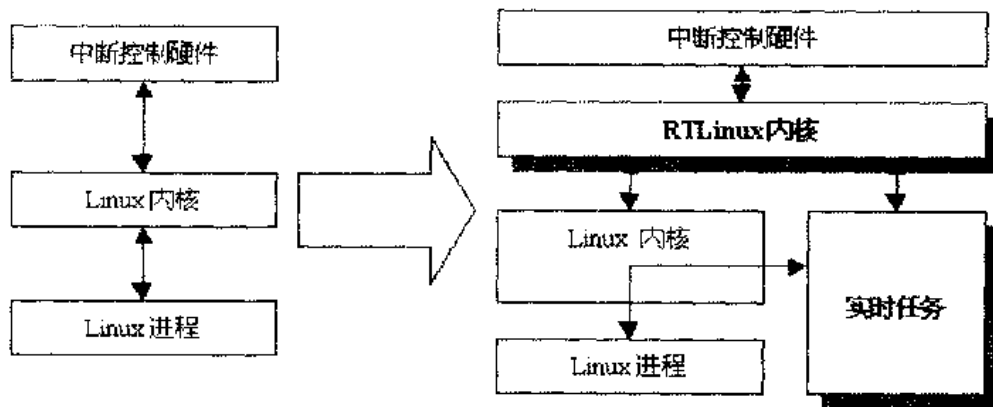


图 9-3 RT-Linux 对 Linux 内核工作环境的改变

这里体现了 RT-Linux 设计过程中的原则:在实时内核模块中的工作尽量少,如果能在 Linux 中完成而不影响实时性能的话,就尽量在 Linux 中完成。因此,RT-Linux 内核尽量做得简单。在 RT-Linux 内核中,不应该等待资源,也不需要使用共享旋转锁。实时任务和 Linux 进程间的通信也是非阻塞的,从来不用等待进队列和出队列的数据。

RT-Linux 将系统和设备的初始化交给了 Linux 完成,对动态资源的申请和分配也交给了 Linux。RT-Linux 使用静态分配的内存来完成硬实时任务,因为在没有内存资源的时候,被阻塞的线程不可能具有硬实时能力。

9.3 RT-Linux 下的编程

9.3.1 RT-Linux 的 API



在 2.0 版本的 RT-Linux 里面包括了两个部分的内容：

1. 对 Linux 内核的修改

该部分是对 Linux 内核的修改,使 Linux 内核中形成一个 RT-Linux 实时小内核。在这个内核中提供了小延时,并且不能被 Linux 延迟和抢占的中断处理过程,还提供了一些底层的同步和中断的控制过程。现在可以支持 SMP(对称多处理器系统),同时为了简化 RT-Linux 结构,把一些没有必要的功能从其中移走了。

2. Linux 标准模块的出现

这一部分是作为 Linux 标准模块出现的,提供了 RT-Linux 的编程和控制接口(API)。通过使用这些 API 可以提供对实时任务的创建和删除、任务的调度和控制等功能。这些模块如下:

(1)rtl_sched 提供了基于优先级的调度方法,支持 POSIX 接口和 1.0 版本的 RT-Linux API 函数。

(2)rtl_time 提供了控制处理器时钟,并且提供了一个时钟处理过程的抽象接口。

(3)rtl_posixio 提供了使用 POSIX 方式的驱动程序的读/写/打开调用。

(4)rtl_fifo 提供了实时任务和 Linux 进程的通信接口,通过一个 Linux 进程可以读写的 FIFO 设备进行通信。

(5)semaphore 是由 Jerry Epplin 提供的给实时任务提供信号量的模块。

(6)mbuff 是由 Tomasz Motylewski 提供的内核进程和 Linux 用户进程进行通信的共享内存驱动程序。通过 mbuff 驱动程序可以让实时任务和用户线程共享内存。

在这些模块里面,提供的 API 函数主要有如下几类:

- 中断控制 API 函数;
- 时钟控制和获取;
- 线程的创建和删除(在 2.0 版本的 RT-Linux 里面,不再通过 rt_task_init 和 rt_task_delete 创建和删除实时任务,而是通过 pthread 线程库进行操作)以及线程优先级和调度策略的控制 API 函数;
- POSIX 方式的驱动接口;
- FIFO 设备驱动程序;
- 串口驱动程序的 API 函数;
- mbuff 驱动 API 函数;
- 浮点运算 API 函数。

具体的 API 介绍和使用方法可以参考《RT-Linux Version Two White Pages》。

9.3.2 RT-Linux 的编程方法示例

在这里,学习一个 RT-Linux 编程示例,说明在 RT-Linux 下面编程的方法。看一个验证 RT-Linux 的任务调度性能测试的例子。在介绍例子之前,先介绍需要用到的 RT-Linux 标准 API 函数:

1. 任务生成和调度函数

```
int pthread_create(pthread_t * p, pthread_attr_t * a, void * (* f)(void *), void * x);
```

创建一个线程,这个线程运行函数指针 f 指向的过程, x 是这个函数指针的入口参数。 a 是这个线程的属性值,可以为这个属性设置 CPU 号、堆栈大小等等属性。返回 0 值表示成功创建一个线程,线程号存放在 p 所指向的空间;返回非 0 表示创建线程失败。

```
int pthread_delete_np(pthread_t thread);
```

删除一个线程,并且释放该线程的所有资源。返回 0 表示成功删除,非 0 表示删除失败。

```
int pthread_setschedparam(pthread_t thread, int policy, const struct sched_param *);
```

设置一个线程的调度参数,用 policy 和 sched_param 两个参数设置 thread 的调度参数属性。 policy = SCHED_RR 时使用 Round-Robin 方法进行调度; policy = SCHED_FIFO 时使用先进先出的方法进行调度。返回值为 0 表示调度成功,否则就是失败。

```
int pthread_getschedparam(pthread_t thread, int * policy, const struct sched_param *);
```

获得一个线程的调度参数。将获得的 policy 和 sched_param 结构放在入口参数所指向的地址里面。

```
int pthread_attr_init(pthread_attr_t * attr);
```

初始化线程运行的属性值。

```
pthread_t pthread_self(void);
```

获得当前正在运行的线程号。

```
int pthread_attr_getcpu_np(pthread_attr_t * attr, int * cpu);
```

设置当前的线程属性的 CPU 号。

```
clockid_t rti_getschedclock(void);
```

获得当前调度方法的时钟。

mode_param 参数是周期的长度。

```
int pthread_wait_np(void);
```

当前周期的线程运行结束,总是返回 0 值。

2. 时间控制函数

```
void clock_gettime(clockid_t clock, struct timespec * resolution);
```

获得 clock 时钟的解析度,并且将解析度存放在 resolution 所指向的地址里面。

3. FIFO 控制函数

```
int rtf_create(unsigned int fifo, int size);
```

创建一个 FIFO 设备,该 FIFO 设备为第 fifo 个,缓冲区大小为 size 字节。

```
int rtf_destroy(unsigned int fifo);
```

销毁第 fifo 个 FIFO 设备。



9.3.3 程序原理

该程序的原理是测出在 RT-Linux 中进行实时任务调度过程中调度需要花费时间的多少。算法描述如下:

```
/* 实时任务端 */
```

对于每 500 个周期;

 等待上一个周期的任务完成;

 获得当前时间和上次周期任务完成时间的差,就是调度的时间;

 循环;

 向 FIFO 输出 500 个周期中完成的最大值和最小值。

```
/* 应用程序端 */
```

 读取 FIFO 设备,获取最大值和最小值;

 在屏幕上打印出来。

这种编程方法是进行 RT-Linux 编程的通用方法,将一个任务分为实时部分和非实时部分。在实时部分完成的是实时任务;在非实时部分主要完成显示等不需要实时的功能。程序的体系结构如图 9-4 所示。



图 9-4 实时程序结构图

9.3.4 程序实现

正如上面所说的,程序分为两个部分,实时部分和非实时部分。实时部分通过使用一


```

        return -1;
    }
    else { /* 成功创建 */
        rtl_printf("created RT-thread \n");
    }
    return 0;
}

```



(2) cleanup_module() /* 清除模块 */

```

_____ cleanup_module()
void cleanup_module(void)
{
    pthread_delete_np(task1); /* 删除该线程 */
    close(fifo); /* 关闭/dev/rtf0 的文件描述符 */
    rtf_destroy(0); /* 销毁该 FIFO 设备 */
}

```

(3) 实时任务过程 thread_code()

```

_____ thread_code()
void * thread_code(void * param) {
    hrtime_t expected, diff, now, min, max; /* 纳秒为单位 */
    struct data samp; /* 写到 FIFO 设备的形式 */

    int i;
    int count = 0;
    DECLARE_CPUID(cpu_id); /* 获得 CPU 号,前面指定为第一个 CPU */

    rtl_printf("Measurement task starts on CPU %d \n", cpu_id);

    if (mode) { /* 周期模式 */
        int ret = rtl_setclockmode (rtl_getschedclock(), RTL_CLOCK_MODE_PERIODIC, period);

        if (ret != 0) {
            rtl_printf("Setting periodic mode failed \n");
            mode = 0; /* 仍然是 oneshot 模式 */
        }
    }
}

```

```

|
if (mode) ///周期模式
    struct timespec resolution;
    clock_getres (rtl_getschedclock(), &resolution);

    period = timespec_to_ns (&resolution);/从时钟里获得解析度*/
|
else ///oneshot 模式
rtl_setclockmode (rtl_getschedclock(), RTL_CLOCK_MODE_ONESHOT, 0);
|
fifo = open("/dev/rtf0", O_NONBLOCK);/非阻塞模式打开 rtf0*/
/使用 rtf0 来和 Linux 进程进行通信*/
if (fifo < 0) ///打开失败*/
    rtl_printf("Error in opening /dev/rtf0; returned %d\n", fifo);
    return (void *) -1;
|
expected = clock_gettime(rtl_getschedclock()) + 5 * period;/开始时间*/
pthread_make_periodic_np(pthread_self(), expected, period);

/设置调度的开始时间和周期*/
/死循环*/
do {
    min = 2000000000;///2 seconds.
    max = -2000000000;///-2 seconds.

    for (i = 0; i < ntests; i++) ///*/
        ++count;
        pthread_wait_np();
        now = clock_gettime(CLOCK_REALTIME);
        if (!mode) {
            if (now < expected) {
                rtl_delay (expected - now);
            }
            now = clock_gettime(CLOCK_REALTIME);
        }
        ///if*/
        diff = now - expected;

```



```

        if (diff < min) {
            min = diff;
        }
        if (diff > max) {
            max = diff;
        }

        expected += period; /* 下一个周期的开始时间 */
    } /* for */ /* 一次循环结束,得到最大值最小值 */
    samp.min = min;
    samp.max = max;
    write (fifo, &samp, sizeof(samp)); /* 向 FIFO 设备写 */
} while (1); /* 无穷的循环 */
return 0;

```



(4) 程序头

```

_____ mymeasurement.c
#include <rtl.h>
#include <rtl_fifo.h>
#include <rtl_sched.h>
#include <rtl_sync.h>
#include <rtl_debug.h>

#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <errno.h>
#include "myheader.h"

pthread_t task1;

int ntests = 100;
int period = 100000;
int mode = 0;
/* 可以向该模块传入下面三个整型参数 */
MODULE_PARM(ntests, "i"); /* 每次循环的遍数 */
MODULE_PARM(period, "i"); /* 周期的长度 */

```



```
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <rtl_fifo.h>

#include "myheader.h"
```

3. 公用头文件

该头文件定义了向 FIFO 写数据的格式。文件名为 myheader.h。

```
_____ myheader.h

struct sample { /* 调度的最长时间和最短时间 */
    hrtime_t min;

    hrtime_t max;

}; /* myheader.h */
```

9.3.5 例 9-5 执行结果

在一台 Pentium 120,40MB 内存, RT-Linux2.2 的机器上, 进行如下编译:

```
# gcc -I/usr/src/Rt-Linux-2.2/Linux/include -I/usr/src/Rt-Linux-2.2/include
-I/usr/src/Rt-Linux-2.2 -I/usr/src/Rt-Linux-2.2/include/posix -Wall -Wstrict-prototypes -O2
-fomit-frame-pointer -D__RTL__ -D__KERNEL__ -DMODULE -pipe -fno-strength-reduce
-m386 -DCPU=386 -c -o mymeasurement.o mymeasurement.c

# gcc -I/usr/src/Rt-Linux-2.2/Linux/include -I/usr/src/Rt-Linux-2.2/in-
clude -I/usr/src/Rt-Linux-2.2 -Wall -O2 -o monitor monitor.c
```

生成两个文件 mymeasurement.o 和 monitor, 前者是模块方式的实时任务, 后者是非实时部分的应用程序。执行如下程序:

```
# insmod mymeasurement.o ntests = 500 period = 1000000 mode = 0
```

说明是采用每次进行 500 个循环寻找最大值和最小值, 每个循环的周期为 1 百万纳秒, 即 1 毫秒, 使用的是 oneshot 的方式。

```
# ./monitor
```

```
In every hundreds of loops, the schedule accuracy test.
```

```
MIN          M AX
12 288,      12 224
12 224,      12 320
12 256,      12 320
12 192,      12 320
```



……(单位为纳秒)

如果执行如下程序:

```
# insmod mymeasurement.o ntests = 500 period = 1000000 mode = 1
```

说明是采用每次进行 500 个循环寻找最大值和最小值,每个循环的周期为 1 百万纳秒,即 1 毫秒,使用的是 periodic 模式。

```
# ./monitor
```

```
In every hundreds of loops, the schedule accuracy test.
```

```
MIN                MAX
```

```
5 978,             5 974
```

```
5 976,             5 978
```

```
5 970,             5 972
```

```
5 968,             5 972
```

```
5 978,             5 972
```

```
5 970,             5 966
```

……(单位为纳秒)

9.4 嵌入式 RT-Linux 的设计

Linux 因为本身的特性,使用户可以利用 Linux 内核做出很多有意义的事情。比如,把 Linux 内核放在一张软盘上,并且利用 ramdisk 技术在内核导入到内存之后,在内存中申请出一片空间,存放 root 文件系统。也可以使用 Linux 做一个网络启动的无盘系统。本节来看看两种利用 RT-Linux 创建“模拟嵌入式系统”的方法。

9.4.1 将 RT-Linux 嵌入

人们已经可以把 Linux 内核嵌入在 EPROM 中,从而可以在嵌入式设备中利用 Linux 内核启动系统(见本书第 12 章的专门论述)。在此介绍一种模拟实现方法:用一张 1.44M 软盘作为在 EPROM 中的固态电子盘 SSD 的模拟,让这张软盘可以启动一个系统。如果这样能成功,就可以把软盘的数据写到 EPROM 的 SSD 中去,并且把对 1.44M 3.5 寸软盘的驱动改成对 EPROM 的驱动程序,就可以完成设计任务。

可以发现,在软盘里面提供必要的工具,就可以把 RT-Linux 提供的那几个实时模块加进去,从而形成一个实时任务运行环境。这样的工作很有意义,因为这样做出来的系统经过少许改动就可以利用于嵌入式环境的系统。

实现这样一个系统的技术支持是使用 Linux 的 ramdisk 技术,在系统启动之后,通过在内存里面申请出一块空间,作为虚拟的硬盘空间,然后把软盘里面的数据装载上来。

这张软盘使用了 Linux Router Project 项目的成果。Linux Router Project 项目专门开发了一个基于嵌入式 Linux 的系统。该系统将 Linux 嵌入在一张软盘上,并且能用该软盘启动,系统包含有路由功能、瘦服务器、瘦客户端功能、网络应用功能。在这个工作的基

础之上,可以自己定制一个 Linux 内核,并且将实时功能嵌入。这样,这张软盘启动的一个系统不仅仅是一个提供了 Web 服务器和 telnet 服务器的系统,而且还是一个运行实时任务的环境。

实现这样的环境有个最大的难点,就在于如何将内核的体积变小。因为在嵌入式系统中,并不需要 Linux 内核这种现代操作系统的所有强大功能,而应该根据需要的功能,自行定制合适的内核,这才是用户的目的。在嵌入到 EPROM 中或者 flash RAM 中时,也能节省很大的一笔费用。

实现内核定制,从一定程度上说,就是在嵌入式系统开发过程中“个性化定制内核”,需要将内核的所占体积尽量缩小,而不应该在内核中留有不需要的功能。进行这样的工作需要有内核在线调试环境。

下面介绍如何自己做一个支持 RT-Linux 内核的软盘系统。

1. 做一个压缩文件作为根文件系统

做一个压缩文件,它在软盘系统启动之后作为根文件系统。在 Linux 内核中,对 ramdisk 的支持可以直接支持 gzip 压缩的文件作为一个根文件系统装载,具体做法是先将这个文件解压缩,然后装载到 /dev/ram0 中。/dev/ram0 并不是一个硬盘,而是从内存中划分出去的一块,以作为根文件系统。在系统启动时,可以给内核传递参数指定 ramdisk 的大小,也可以利用 rdev 命令直接进行操作。在根文件系统中需要提供一些简单的应用程序和自己需要的一些函数库、RT-Linux 需要的模块等等。

ramdisk 的制作步骤如下:

```
# dd if = /dev/zero of = /dev/ram bs = 1k count = 4096
```

先将 /dev/ram 设备前的 4096 字节清零,并且该 ramdisk 大小为 4MB。

```
# mkfs. minix -vm0 /dev/ram 4096
```

创建文件系统,因为 minix 文件系统比 ext2 文件系统需要的内核体积小,因此采用 minix 文件系统而不使用 ext2 文件系统。

然后将 /dev/ram 装载上来,把自己需要的东西都拷贝到这个文件系统下面,然后卸载该文件系统,再进行如下操作:

```
# dd if = /dev/ram bs = 1k count = 4096 | gzip -v9 > /tmp/ram _ image. gz
```

ram _ image. gz 就是这张软盘的操作系统在启动时需要装载的根文件系统。

2. 定制自己的内核

假设这个内核大小为 420KB,那么将内核使用 dd 命令直接写到软盘上:

```
# dd if = zImage of = /dev/fd0 bs = 1k
```

然后将 ramdisk 写到内核之后的位置,现在这种情况可以写在 420KB 之后,1440KB-ramdisk 大小的任意一处,这里写到 430KB 的位置:

```
# dd if = /tmp/ram _ image. gz of = /dev/fd0 bs = 1k seek = 430
```

修改 /dev/fd0 的启动参数,利用 rdev 命令完成,使用 rdev 可以将内核启动参数直接写到 /dev/fd0 上面。关于内核的 ramdisk 参数,是用两个字节表示的,从 0 到 10 位表示的是 ramdisk 的位置(2¹¹ 表示最大的偏移量为 2096Bytes),第 15 位表示是否需要提示插入 ramdisk 的软盘,第 14 位表示需要导入 ramdisk,其他位目前还没有意义。那么,现在

的软盘参数应该是 $2 \cdot 15 + 2 \cdot 14 + 430 = 49\ 582$ 。操作如下：

```
# rdev /dev/fd0 /dev/fd0  
# rdev -r /dev/fd0 48952
```

然后使用这张软盘,就可以启动一台无盘、8MB 内存的机器。在根文件系统中存放 RT-Linux 的启动模块和程序后,就可以正常运行 RT-Linux 了。



9.4.2 设计嵌入式 RT-Linux

在这里看看如何通过网络启动的方法设计嵌入式 RT-Linux。Linux 提供了一种很方便的方式来创建无盘工作站,这样的无盘工作站可以从一台网络服务器启动。网络启动的时候,通过网络获得 Linux 内核、驱动程序模块和应用程序。这样的启动过程如下:

- (1)从 bootp 或 dhcp 服务器获得这台机器的 IP 地址。
 - (2)从一台 tftp 服务器获得 Linux 的内核。
 - (3)从一台 NFS 服务器上装载文件系统。
- 必要的话,可以装载进 X-Server 软件并且运行 X Window。
- (4)运行应用程序。

9.5 本章小结

本章的着眼点是嵌入式的实时操作系统和实时 Linux。首先了解了当前比较有代表性的几种嵌入式实时操作系统。然后,了解了它们的设计原理,如调度算法、内存管理、通信等基本方面。

本章详细介绍了 RT-Linux 的实现原理、编程原理和示例。RT-Linux 是一种通过 Linux 实现的“硬实时系统”,系统的实时性能很好。在此分析了实时 Linux 内核的构造特征,讲述了它在普通 Linux 的内核上所进行的改造和所实现的结构关系。然后,学习了 RT-Linux 的常用 API,并通过编制程序巩固了所学知识。最后介绍了两种利用 Linux 创建“模拟嵌入式系统”的方法:使用一张软盘模拟 EPROM 环境和使用网络自举的系统设想。实时 Linux 是嵌入式 Linux 的一个重要分支和应用。读者在了解嵌入式 Linux 技术时,应该对本章内容给予足够的重视。

第 10 章 嵌入式 Linux 图形用户界面



图形用户界面一直是计算机的重要组成部分,对于嵌入式系统而言,也是如此。本章将介绍嵌入式系统常用的一些 GUI,重点讲解一个嵌入式 Linux 平台上使用得非常广泛的 GUI-MiniGUI。通过本章的学习,读者将对嵌入式 Linux 环境下的 GUI 开发有深入的领会。

本章主要内容:

- 各种嵌入式图形用户界面简介
- MicroWindows
- MiniGUI 简介
- 多线程与多窗口
- 消息和消息循环
- 图形和输入抽象
- MiniGUI 的结构和算法
- MiniGUI 的典型应用

10.1 嵌入式系统的图形用户界面概述

本节首先对嵌入式系统的图形用户界面进行一个概述,介绍其现状和几个典型的例子。当然,重点是放在基于嵌入式 Linux 的图形用户界面之上。

10.1.1 图形用户界面

1. 图形用户界面的历史

计算机用户界面是指计算机与其使用者之间的对话接口,是计算机系统的重要组成部分。计算机的发展史不仅是计算机本身处理速度和存储容量飞速提高的历史,而且是计算机用户界面不断改进的历史。早期的计算机是通过面板上的指示灯来显示二进制数据和指令,人们则通过面板上的开关、按键及穿孔纸带送入各种数据和命令。20 世纪 50 年代中、后期,由于采用了作业控制语言(JCL)及控制台打字机等,使计算机可以批处理多个计算任务,从而代替了原来笨拙的手工按键方式,提高了计算机的使用效率。

1963 年,美国麻省理工学院在 709/7090 计算机上成功地开发出第一个分时系统 CTSS,该系统连接了多个分时终端,并最早使用了文本编辑程序。从此,以命令行形式对话的多用户分时终端成为 20 世纪 70 年代乃至 80 年代用户界面的主流。

20 世纪 80 年代初,由美国 Xerox 公司 Alto 计算机首先使用的 Smalltalk-80 程序设计开发环境,以及后来的 Lisa、Macintosh 等计算机,将用户界面推向图形用户界面的新阶段。随之而来的用户界面管理系统和智能界面的研究均推动了用户界面的发展。用户界面已经从过去的由人去适应笨拙的计算机,发展到今天计算机不断地适应人的需求。

用户界面的重要性在于它极大地影响了最终用户的使用,影响了计算机的推广应用,甚至影响了人们的工作和生活。由于开发用户界面的工作量极大,加上不同用户对界面的要求也不尽相同,因此,用户界面已成为计算机软件研制中最困难的部分之一。当前,Internet 的发展异常迅猛,虚拟现实、科学计算可视化及多媒体技术等对用户界面提出了更高的要求。

2. 图形用户界面系统的结构模型

一个图形用户界面系统通常由三个基本层次组成,即显示模型、窗口模型和用户模型。用户模型包含了显示和交互的主要特征,因此图形用户界面这一术语有时也仅指用户模型。下表给出了图形用户界面系统的层次结构。

桌面管理系统
用户模型
窗口模型
显示模型
操作系统
硬件平台

表中的最底层是计算机硬件平台,如 Macintosh、Sun、SPARC 等。硬件平台的上层是计算机的操作系统。大多数图形用户界面系统都只能在一两种操作系统上运行,只有少数的产品例外。

操作系统之上是图形用户界面的显示模型。它决定了图形在屏幕上的基本显示方式。不同的图形用户界面,系统所采用的显示模型各不相同。例如大多数在 UNIX 之上运行的图形用户界面系统都采用 X 窗口作显示模型;MS Windows 则采用 Microsoft 公司自己设计的图形设备接口(GDI)作显示模型。

显示模型之上是图形用户界面系统的窗口模型。窗口模型确定窗口如何在屏幕上显示,如何改变大小,如何移动及窗口的层次关系等。它通常包括两个部分,即编程工具及对如何移动、输出和读取屏幕显示信息的说明。因为 X 窗口不但规定了如何显示基本图形对象,也规定了如何显示窗口。所以它不但可以充当图形用户界面的显示模型,也可以充当它的窗口模型。

窗口模型之上是用户模型,图形用户界面的用户模型又称为图形用户界面的视感。它也包括两个部分:一是构造用户界面的工具;二是对于如何在屏幕上组织各种图形对象,以及这些对象之间如何交互的说明。比如,每个图形用户界面模型都会说明它支持什

么样的菜单和什么样的显示方式。

图形用户界面系统的应用程序接口由其显示模型、窗口模型和用户模型的应用程序接口共同组成。例如 OSF/Motif 的应用程序接口就是由它的显示模型、窗口模型的应用程序接口 Xlib、用户模型的应用程序接口 Xt Intrinsics 及 Motif Toolkit 共同组成的。

3. 典型的图形用户界面系统 X Window

X 窗口系统(以下简称 X 窗口)是 UNIX/Linux 上标准的图形界面。虽然各自在 UNIX 上所看到的图形界面都不太一样,但是其图形界面都是以 X 协定为基础,使得各厂商之间的 UNIX 图形界面可以互相沟通。

早期 X 窗口只是架在 UNIX 工作站上,没有 PC 版。后来成立了 XFree86 计划,其目的是提供一个 PC 版的 X 窗口,主要是移植到 Intel 的 x86 系列处理器上,所以才称作 XFree86 计划,后来也被移植到其他的处理器上。

XFree86 虽然不是以 GPL 授权,但是它也可以自由拷贝、散播,也可以使用在商业用途上,所以大部份的 PC 版 UNIX 如 Linux、BSD 等等,都将 XFree86 加入在操作系统的套件内。

X 窗口采用的是客户/服务器式的结构模型。它由 X 协议、X 服务器、客户和 Xlib 函数库等几部分组成。客户也称客户程序,是指在本地或者网络上运行的利用 X 进行显示输出的应用程序。X 服务器是一个在图形工作站上运行的服务进程,负责对显示器的输出、键盘和鼠标的输入进行管理。客户与服务器通过网络相连接,并通过 X 协议进行通信。

在 X 窗口中,客户程序和应用程序是两个等价的概念。客户程序可根据需要调用 Xlib 中的函数,对 Xlib 函数的调用被转换为 X 的协议请求,并通过网络发送给 X 服务器。当服务器收到协议请求后,就按照协议的要求去完成制定的任务,并通过 X 协议对用户做出回答。多个客户可同时连到同一服务器上,一个客户亦可与多个服务器相连接。一个运行 X 系统的网络环境如图 10-1 所示。

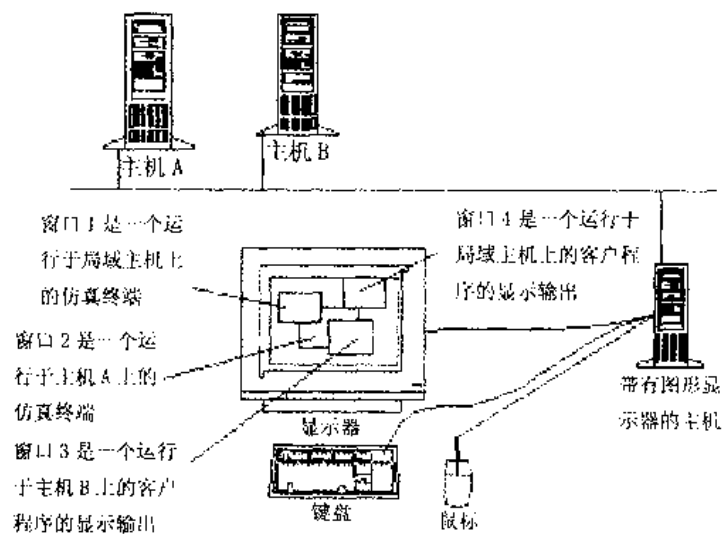



图 10-1 运行 X 系统的网络环境图示

10.1.2 嵌入式系统下的图形用户界面

1. GUI 在嵌入式或实时系统中的地位



在常规 PC 机等小型手持式设备上,由于硬件条件等的限制,所看到的用户界面都非常简单。与常规 PC 机上华丽美观的 GUI 支持。但最近出现的 Palm 等手持式电脑或者在 Windows CE 等面向嵌入式系统的操作系统上,已经看到了完整的图形用户界面支持(见图 10-2,这是一款 Agenda 计算机公司的便携式 LinuxPC-VR3)。随着手持式设备的硬件条件的提高,估计嵌入式系统对轻量级 GUI 的需求会越来越迫切。

图 10-2 Agenda 计算机公司的便携式 LinuxPC-VR3

近来的市场需求显示,越来越多的嵌入式系统,包括 PDA、机顶盒、DVD/VCD 播放机、WAP 手机等等系统均要求提供全功能的 Web 浏览器。这包括对 HTML 4.0 的支持、JavaScript 的支持,甚至包括对 Java 虚拟机的支持。而这一切均要求有一个高性能、高可靠的 GUI 的支持。

另外一个迫切需要轻量级 GUI 的系统是工业实时控制系统。这些系统一般建立在标准 PC 平台上,硬件条件相对嵌入式系统要好,但对实时性的要求非常高,并且比起嵌入式系统来说,对 GUI 的要求也更高。这些系统一般不希望建立在庞大累赘的、非常消耗系统资源的操作系统和 GUI 之上,比如 Windows 或 X Window。目前许多这类系统都建立在 DOS 等系统上,并且采用比较简单的手法实现 GUI。但是,在出现 Linux 系统之后,尤其在 RT-Linux 系统出现之后,许多工业控制系统开始采用 RT-Linux 作为操作系统,但 GUI 仍然是一个问题。X Window 太过庞大和臃肿。这样,这些系统对轻型 GUI 的需求更加突出。

但是,必须清楚的是,嵌入式系统往往是一种定制设备,它们对 GUI 的需求也各不相同。有的系统只要求一些图形功能,而有些系统要求完备的 GUI 支持。因此,GUI 也必须是可定制的。

综上所述,GUI 在嵌入式系统或者实时系统中的地位将越来越重要,这些系统对 GUI 的基本要求包括:(1)轻型、占用资源少;(2)高性能;(3)高可靠性;(4)可配置。

2. 各种嵌入式图形用户界面简介

目前的图形界面编程接口 API 的主流是 Win32、X Window、Java 等,各制造商一般都力求能使自己的 GUI 系统能提供至少是应用一级的与上述图形窗口系统兼容。很显然,如何实现具体嵌入式系统的图形窗口界面,与用户所选择的操作系统平台和硬件平台是紧密相关的。下面是如今流行的一些专门针对嵌入式系统的图形用户界面。

(1) Win CE

WinCE 是 Microsoft 针对嵌入式产品的一套模块化设计的操作系统,并且它已经为用户准备了很友好的 GUI,用户所要考虑的就是要什么或不要什么。WinCE 模块化的设计为:基本组合(核心模块+文件系统 250KB~350KB)、基本网络组合(上述模块+通信模块 350KB~500KB)、基本图形界面(上述模块+输入模块+GDI 绘图模块 450KB~750KB)、基本视窗组合(上述模块+视窗管理模块+COM750KB~1MB)、基本 Shell 组合(上述模块+工作管理器、视窗控制组件、主控台控件等 1MB~2MB)和全功能多媒体组合(上述模块+多媒体组件、数据库控件、浏览器等 2MB~6MB)。

WinCE 支持 Win32API 和 MFC 的子集。OEM 厂商可以针对特定的需要选取操作系统模块,定制出自己的 WinCE。甚至 OEM 厂商也可以自行替换微软所提供的操作系统核心模块,代替自己设计的模块。对于 GDI(图形设备接口),它的多组件的构成方式也使 OEM 厂商便于建立自己的 GDI。也就是说,选择了 WinCE,GUI 需要用户做的工作会很少。

(2) Sun 的 Embedded System 解决方案

JavaOS 是一个完整的操作系统平台,它以 Sun 的 Chorus OS 为核心。Chorus OS 是一个高度可扩展的可靠的嵌入式操作系统,它提供了实时功能和很强的网络通信能力。其很自然地成为 switch、router 和 hub 网络设备的理想操作系统。JavaOS 是 Webphone、机顶盒、手持电脑等设备的理想的解决方案。Sun 将它的 Personal Java 和 Embedded Java 都应用于 JavaOS 产品系列中。Personal Java 和 Embedded Java 是为支持用户设备中运行的实时操作系统而设计的,使得在这些设备上可以运行 Java 应用程序并连接到网络。Personal Java 软件支持的是那些需要拥有复杂的显示界面,比如窗口系统的应用场合。而 Embedded Java 软件支持的是那些没有界面要求或是只要求简单的基于字符的显示界面的应用场合。

(3) 紧缩的 X Window

X Window 系统是一种免费的图形系统,在大多数的 UNIX 系统、DEC 的 VAX/VMS 操作系统以及许多 Linux-based 系统中都配置了它。X Window 系统的特点是网络透明。它的体系结构是建立在 Client/Server 模型基础之上的,由 X-server(X 服务器)来创建并操纵屏幕上的图形和字符的显示以及输入事件。X Window 将系统依赖硬件的大部分细节都隐藏在 X-server 中,这意味着两个特点:一方面,使 X Window 程序具有设备独立性;另一方面降低了实现效率,增加了本身的大小,这是其在嵌入式系统中的最大障碍。X-server 的实现通常需要多于 1M 的 ROM,2M~4M 的 RAM,Client 需要与 M 大小的 X Window 库相连。实际上有些公司提供了 X Window 的嵌入式版本,如 Viscom 公司的 Rt-X Window 就是一个具有很好的扩展性的图形软件包,它具有 X API 界面并提供实时性

能。它还可以支持无 Net/File 的配置。

(4) ZIN&UGL

UGL(universal graphic library)由 Zinc 公司提供,是 VxWorks 的一种 GUI 构造方案。遵循公开发布策略,源码可以下载(嵌入式系统的源码不公开),使用时需要付费。UGL 没有提供综合的图形界面工具箱,如 widget 等,但是可与 ZincApplication FrameWork、Personal Java 配套使用。ZAF(zincapplication framework)是一个与 MFC 相类似的库,ZAF 同时包含一个界面构造工具 Zinc Designer,基于 ZAF 的程序可在多个平台如 Win32、X Window、VxWorks 下运行。

(5) PEG

PEG(portable embedded graphic)由 Swell software 公司提供,是专门为嵌入式系统设计的图形库,相对其他嵌入式的 GUI,PEG 体积小、速度快,易于移植到各种硬件上,并能可视化地配置视屏输出。PEG 并不是一个操作系统,但它的设计使其非常易于集成到各种商业的实时操作系统中。这就使嵌入式系统制造商可以很自由地选择适合它们的实时平台。设计上考虑了嵌入式系统的显示分辨率、系统资源限制。它使用 MSDEV 开发环境。PEG 库完全用 C++ 编写,实现了事件驱动机制,提供了强大的 API。PEG 直接对视屏和输入硬件作用,使其可以达到最快的速度。

(6) SDL

SDL(standard drawing library)是由 Ruster Graphics 公司提供的 C 语言图形库,它是为实时和非实时操作系统设计的。其特点是体积小、具有可扩展性,其目标代码的大小可根据具体的嵌入式应用需要的 SDL 库函数的多少来定。SDL 的设计目标是可运行于任何 CPU,任何使用线性编址和被 ANSIC 编译器和链接器支持的操作系统上。SDL 要比 X Window 小得多,小于 30KB。它包括了画点、线、圆、矩形、文本、像素处理、Clip、逻辑区域和字库等 61 个图形函数。

(7) DinX

DinX 非常适合于在很小的系统上运行多窗口程序,它简单、轻巧,并且快速。DinX 并不是 X,它使用 Linux 核心的 framebuffer 视频驱动,采用 Client/Server 模式。为此,系统提供了两个界面:/dev/dinxsvr 和/dev/dinxwin。

一个服务器程序连接到/dev/dinxsvr,并决定来自各程序窗口的 request 各占有视屏的各个部分。它也负责给各窗口发送像鼠标移动这样的事件消息。Clinet 程序连接到/dev/dinxwin,与 Server 进行消息通信等。Server 进程还负责处理事件、窗口管理、调色板配置等功能。DinX 是一个实验性的窗口系统,它处在发展阶段中,还存在一些缺陷和问题。DinX 的 license 属于 MPL(mozilla public license),也可以转化为 GPL(见附录 A)。这样,DinX 核心模块可以集成到 Linux 中,DinX 库可以链接到其他的 GPL 程序中。

(8) MiniGUI

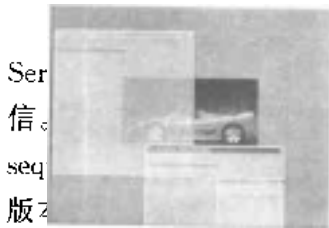
MiniGUI 是中国人做得较好的自由软件之一。它是在 Linux 控制台上运行的多窗口图形用户界面支持系统,可以在以 Linux 为基础的应用平台上提供一个简单可行的 MiniGUI 支持系统。“小”是 MiniGUI 的特色,MiniGUI 可以应用在电视机顶盒、实时控制系统、掌上电脑等诸多场合。它是基于 SVGA 库和 LinuxThread 库的。

MiniGUI 使用类似于 Win32API 来获得简单的具有 Win98 风格的图形用户界面。

MiniGUI 的组件是四种类型的窗口:主窗口、对话框、控件和子窗口。MiniGUI 中的主窗口与 Microsoft Windows 应用的主窗口类似。每个主窗口属于一个唯一的线程。通过调用相应的函数可以在一个线程中创建一个主窗口。每个线程有一个唯一的消息队列。该线程的主窗口接收来自于这个队列的消息,并进行处理。MiniGUI 的目标是保持其小巧精致的特点,以提供一个很小的窗口系统支持库。它尽量保持与 Win32 的兼容,这样在 WinCE 应用的任何场合,也可以使用 MiniGUI。

鉴于此,在本章中将重点放在对 MiniGUI 的学习上。

(9) MGR



图形窗口系统。它由一个具有内置窗口管理器和终端仿真器的 Client 运行在终端仿真器上,并用它来与 Server 进行通信。这个 Client 运行在终端仿真器上,并用它来与 Server 进行通信。选择菜单或是键盘来交互,也可以由写在 Client 伪终端上的 escape 序列来交互。当前可以运行在 Linux、FreeBSD、Sun 和 Conherent 上。各种老版本 MGR 运行在 Macintosh、AtariSTMiNT、Xenix、386Minix 和 DEC3100 上。在欧洲,OS9 和 Lynx 的很多小型的工业实时系统都是用 MGR 作为系统的用户界面。

(10) MicroWindows

MicroWindows 是一个开放源代码项目,它使那些只在具有相当的硬盘和 RAM 配置的高端 Windows 系统中才能实现的窗口系统,如 Microsoft Windows 和 X Window,可以在嵌入式设备上运行。正是以 Linux 为平台的便携式和手持 PC/Device(LinuxCE)市场的驱动,使 MicroWindows 大有用武之地。读者可以看看图 5-1,其展示的即是一款基于 MicroWindows 的 PDA。图 10-3 则展示了 MicroWindows 的 Alpha 混合的高级功能。

图 10-3 MicroWindows 的 Alpha 混合功能演示

MicroWindows 能够在没有任何操作系统或其他图形系统的支持下运行,它能对裸显示设备进行直接操作。这样,MicroWindows 就显得十分小巧,便于移植到各种硬件和软件系统上。对于 Linux 平台的嵌入式系统,MicroWindows 无疑是合理的选择之一。MicroWindows 现在除了可以运行在拥有 Framebuffer 驱动的 32 位的 Linux 系统上,还可以运行在流行的 SVGAlib 库上,也能运行在 16 位的 LinuxELKS 和实模式的 MSDOS 上。MicroWindows 已经有了 1、2、4、8、16 和 32 位/像素的 screendriver。MicroWindows 的图形引擎使其能够运行在任何支持 readpixel、writepixel、drawhozline、drawvertline 和 setpal-

ette 的系统。所有的 bitmap、font、cursor 和 color 支持的函数都是建立在这些基本函数之上的。可以说, MicroWindows 对 8、15、16、32 位的色彩系统, 以及 1、2、4 和 8 位的调色板色彩系统都提供了支持。同时, MicroWindows 的 X11 driver 使 MicroWindows 应用可以运行在 X Windowdesktop 上。

尽管为了速度, 少部分的 MicroWindows 程序用汇编语言改写了, 但还是可以认为 MicroWindows 是完全用 C 语言写的。它便于移植, 可以运行在 Intel 的 16 和 32 位的 CPU、MIPS R4000 和 ARM 芯片上。这些芯片都是在手持和便携式 PC 中常用的。在 16 位的系统中, 整个 MicroWindows (包括 screen、mouse 和 keyboard 驱动) 不到 64KB。在 32 位的系统中, 加上对 proportional fonts 的支持也不过 100KB。

MicroWindows 的设计是分层的, 这样的设计便于用户按自己的需要来修改、删减和增加。它分 3 层: 最底层是 screen、mouse/touchpad 和 keyboard 驱动程序, 它们直接与显示和输入硬件打交道; 中间层是一个可移植的图形引擎层, 它使用最底层提供的服务完成对话线、区域填充、文本、多边形、裁剪区域、色彩等的支持; 最上层是 API, 提供给图形化应用程序调用。目前, 这些 API 支持 Win32 和 Nano X 接口。这样一来, 它们就与 Win 32 和 X Window 窗口系统保持了兼容, 在这些系统间移植应用软件就要容易得多。因为 WinCE API 是 Win 32 API 的子集, 所以 MicroWindows 也与 Win CE 在应用接口一级兼容, 可在 MicroWindows 的嵌入式系统上运行 Win CE 应用。

10.1.3 嵌入式 Linux 环境下的 GUI

在上面的小节中已经介绍了嵌入式 Linux 环境下常用的几种 GUI, 这里再补充几个。

1. OpenGUI

OpenGUI 在 Linux 系统上存在已经很长时间了。最初的名字叫 FastGL, 只支持 256 线性显存模式, 但目前也支持其他显示模式。是用 C++ 编写的, 只提供 C++ 接口。

OpenGUI 基于一个用汇编实现的 x86 图形内核, 提供了一个高层的 C/C++ 图形/窗口接口。它和 MiniGUI 一样, 也是使用 LGPL 许可证。OpenGUI 提供了 2 维绘图原语, 消息驱动的 API、BMP 文件格式支持。OpenGUI 功能强大, 使用方便。用户甚至可以实现 Borland BGI 风格的应用程序, 或者是 QT 风格的窗口。OpenGUI 支持鼠标和键盘的事件, 在 Linux 上基于 Framebuffer 或者 SVGALib 实现绘图。Linux 下 OpenGUI 也支持 Mesa3D。颜色模型方面, OpenGUI 已经支持 8、15、16 和 32 位模型。

由于其基于汇编实现的内核并利用 MMX 指令进行了优化, OpenGUI 运行速度非常快, 可以用 UltraFast 形容, 它支持 32 位的机器, 能够在 MS-DOS、QNX 和 Linux 下运行。主要用来在这些系统中开发图形应用程序和游戏。由于历史悠久, OpenGUI 非常稳定。当然, 也可以看出, 由于其内核用汇编语言实现, 可移植性受到了影响。通常驱动程序一级性能和可移植性是矛盾的, 必须为其找到一个折中。

2. Qt/Embedded

Qt/Embedded 是著名的 QT 库开发商 Trolltech 正在进行的面向嵌入式系统的 QT

版本。这个版本的主要特点是可移植性较好,许多基于 QT 的 X Window 程序可以非常方便地移植到嵌入式系统。但是该系统不是开放源码的,如果要使用这个库,可能需要支付昂贵的授权费用。

3. MiniGUI 和 MicroWindows 的比较

MiniGUI 和 MicroWindows 均为自由软件,只是前者遵循 LGPL 条款,后者遵循 MPL 条款。这两个系统的技术路线也有所不同。MiniGUI 的策略是建立在比较成熟的图形引擎之上的。比如 SvcLib 和 LibGGI,开发的重点在于窗口系统、图形接口之上;MicroWindows 目前的开发重点则在底层的图形引擎之上,窗口系统和图形接口方面的功能还比较欠缺。举个例子来说,MiniGUI 有一套用来支持多字符集和多编码的函数接口,可以支持各种常见的字符集,包括 GB、BIG5、UNICODE 等,而 MicroWindows 在多字符集的支持上尚没有统一接口。

10.2 MiniGUI

从上面可以知道,MiniGUI 是嵌入式 Linux 平台上的一种比较成熟的图形用户界面系统,也是国内做得很好的一套自由软件。所以,下面将重点讨论对它的研究。

对 MiniGUI 的研究已经有几年的历史了,不少人为它的发展已经作出了富有创新性的贡献。朗讯公司的宋立新先生在他的研究学位论文中就对 MiniGUI 有了深入的研究。经过他的同意,笔者在此摘录并适当改编了文中的部分关键内容,以飨各位。非常感谢宋先生的协助,正是这种互帮互助的精神才让 Linux 充满了活力和希望。

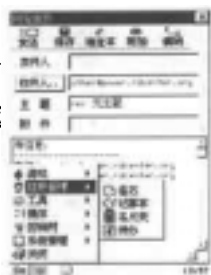
10.2.1 MiniGUI 的起源

MiniGUI 最初是为了满足一个工业控制系统(计算机数控系统)的需求而设计和开发的。这个工业控制系统是清华大学为一台数控机床设计的计算机数控系统(CNC)。在比较 DOS、Windows 98、Windows NT、Linux 等系统之后,该项目组决定选择 RT-Linux 作为实时操作系统,以便满足 2ms 甚至更高的实时性。但是图形用户界面是一个问题,因为 X Window 不适合于实时控制系统,并且当时 X Window 系统的本地化也不尽人意。因此,决定自己开发一套图形用户界面支持系统。这就是 MiniGUI 产生的背景。显然,MiniGUI 一开始就针对实时系统而设计,因此,在设计之初就考虑到了小巧、高性能和高效率。目前,这个数控系统的开发已近尾声,MiniGUI 在其中担当了非常重要的角色。

在国内,对 MiniGUI 的研究已经广泛地开展了起来。用户可以登录网站 <http://www.minigui.org> 参看最新的进展。图 10-4 展示的是一个国内某公司开发的基于 MiniGUI 的 PDA 的界面。

在考虑到其他不同于数控系统的嵌入式系统时,为了满足千变万化的需求,必须要求 GUI 系统是可配置的。在 CNC 系统中得到成功应用之后,立即着手于 MiniGUI 可配置的设计。通过 Linux 下的 automake 和 autoconf 接口,实现了大量的编译配置选项,通过

这些选



UI 库中包括哪些功能而同时不包括哪些功能。



图 10-4 某公司开发的基于 MiniGUI 的 LinuxPDA

因此,MiniGUI 是一个非常适合于工业控制实时系统以及嵌入式系统的可定制的、小巧的图形用户界面支持系统。

10.2.2 MiniGUI 的重要特色

1. 多线程和多窗口

在 MiniGUI 中,图形用户界面包括如图 10-5 所示的基本元素。MiniGUI 中的窗口基本分四类,分别为主窗口、对话框、控件和主窗口中的子窗口。

MiniGUI 中的主窗口和 Windows 应用程序的主窗口概念类似,但有一些重要的不同之处,MiniGUI 中的每个主窗口及其附属主窗口对应于一个单独的线程,通过函数调用可建立主窗口以及对应的线程。每个线程有一个消息队列,属于同一线程的所有主窗口从这一消息队列中获取消息并由窗口过程(回调函数)进行处理。

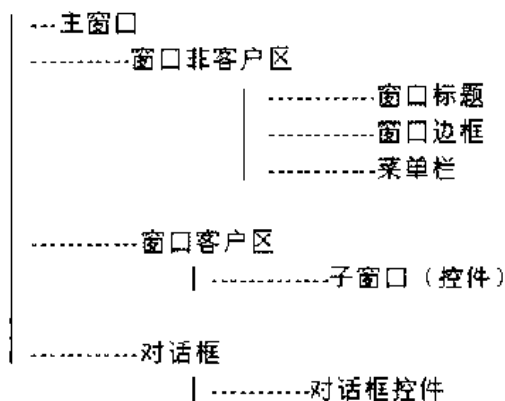


图 10-5 图形用户界面的基本元素

2. 对话框和标准控件

MiniGUI 中的对话框是一种特殊的窗口,对话框一般和控件一起使用,这两个概念和 Windows 或 X Window 中的相关概念是类似的。MiniGUI 支持的控件类型有:

- 静态框:文本、图标或矩形框等;
- 文本框:单行或多行的文本编辑框;
- 按钮:单选钮、复选框和一般按钮等;
- 列表框;
- 进度条。



除此之外,MiniGUI 还支持级联式菜单、插入符、定时器、光标、快捷键等常见的 GUI 元素。

3. 消息和消息循环

在任何 GUI 系统中,均有事件或消息驱动的概念。在 MiniGUI 中,使用消息驱动作为应用程序的创建构架。

在消息驱动的应用程序中,计算机外设发生的事件(例如敲击键盘、鼠标等),都由支持系统收集,将其按事先的约定格式翻译为特定的消息。应用程序一般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循环,在这个循环中读取消息并处理消息,直到特定的消息传来为止。这样的循环称为消息循环。一般地,消息由代表消息的一个整型数和消息的附加参数组成。

应用程序一般要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,应用程序在此函数中处理相应的消息。图 10-6 就是一个消息驱动的应用程序的简单构架示意图。

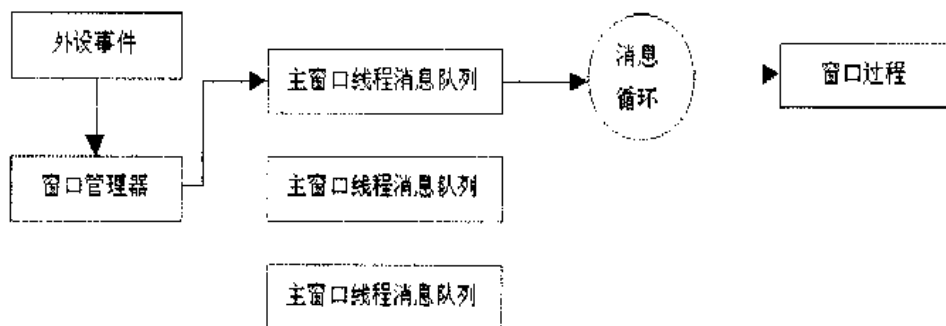


图 10-6 消息驱动的应用程序的简单构架

MiniGUI 支持如下几种消息的传递机制。这些机制为多线程环境下的窗口间通信提供了基本途径:

- 通过 SendMessage 发送。该函数可以向任意一个窗口发送消息,消息处理完成之后,该函数返回。如果目标窗口所在线程和调用线程是同一个线程,该函数直接调用窗口过程,如果处于不同的线程,则用 PostSyncMessage 函数发送同步消息。
- 通过 SendNotifyMessage 发送。该函数向指定的窗口发送通知消息,将消息放入消息队列后立即返回。由于这种消息和邮寄消息不同,是不允许丢失的,因此,系统以链表的形式处理这种消息。
- 通过 SendAsyncMessage 发送。利用该函数发送的消息称为“异步消息”,系统直接调用目标窗口的窗口过程。

4. 图形抽象层和输入抽象层

在 MiniGUI0.3.xx 的开发中,引入了图形抽象层和输入抽象层(GAL 和 IAL)的概念。抽象层的概念类似 Linux 内核虚拟文件系统的概念。它定义了一组不依赖于任何特殊硬件的抽象接口,所有顶层的图形操作和输入处理都建立在抽象接口之上。而用于实现这一抽象接口的底层代码称为“图形引擎”或“输入引擎”,类似操作系统中的驱动程序,这实际是一种面向对象的程序结构。利用 GAL 和 IAL,MiniGUI 可以在许多图形引擎上运行,比如 SVGALib 和 LibGGI,并且可以非常方便地将 MiniGUI 移植到其他 POSIX 系统上,只需要根据抽象层接口实现新的图形引擎即可。目前,MiniGUI 已经编写了基于 SVGALib 和 LibGGI 的图形引擎。利用 LibGGI,MiniGUI 应用程序可以运行在 X Window 上,将大大方便应用程序的调试。目前,有关的公司正在进行 MiniGUI 私有图形引擎的设计开发。通过 MiniGUI 的私有图形引擎,可以最大程度地针对窗口系统对图形引擎进行优化,最终提高系统的图形性能和效率。

10.2.3 MiniGUI 的结构

1. 多线程的分层设计

从整体结构上看,MiniGUI 是分层设计的,层次结构如图 10-7 所示。在最底层,GAL 和 IAL 提供底层图形接口以及鼠标和键盘的驱动;中间层是 MiniGUI 的核心层,其中包括了窗口系统必不可少的各个模块;最顶层是 API,即编程接口。

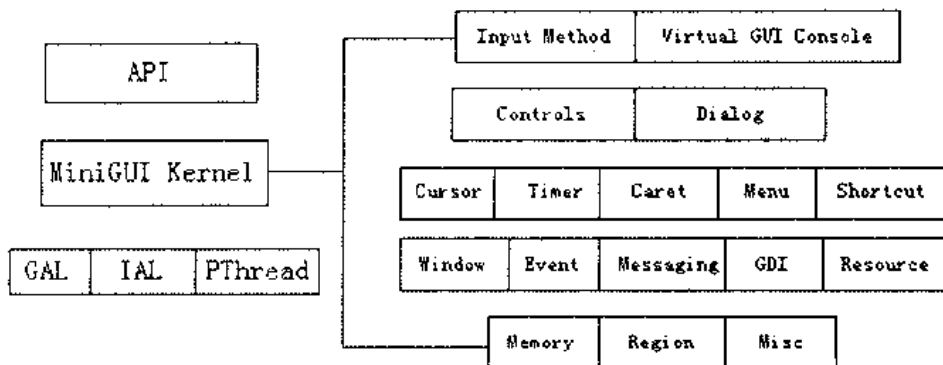


图 10-7 MiniGUI 的层次结构

GAL 和 IAL 为 MiniGUI 提供了底层的 Linux 控制台或者 X Window 上的图形接口以及输入接口,而 Pthread 是用于提供内核级线程支持的 C 函数库。

MiniGUI 本身运行在多线程模式下,它的许多模块都以单独的线程运行。同时,MiniGUI 还利用线程来支持多窗口。从本质上讲,每个线程有一个消息队列,消息队列是实现线程数据交换和同步的关键数据结构。一个线程向消息队列中发送消息,而另一个线程从这个消息队列中获取消息,同一个线程中创建的窗口可共享同一个消息队列。利用消息队列和多线程之间的同步机制,可以实现下面要讲到的微客户/服务器机制。

多线程有其一定的好处,但不方便的是不同的线程共享了同一个地址空间。因此,客户线程可能会破坏系统服务器线程的数据,但有一个重要的优势是,由于共享地址空间,线程之间就没有额外的数据复制开销。由于 MiniGUI 是面向嵌入式或实时控制系统的,因此,这种应用环境下的应用程序往往具有单一的功能。从而使得采用多线程而非多进程模式实现图形界面有了一定的实际意义,也更加符合 MiniGUI 的“mini”特色。目前,MiniGUI 开发组准备开发出基于多进程客户/服务器机制的类 X Window 的 API,以便于用于不同的应用环境。

2. 微客户/服务器结构

在多线程环境中,与多进程间的通信机制类似,线程之间也有交互和同步的需求。比如,用来管理窗口的线程维持全局的窗口列表,而其他线程不能直接修改这些全局的数据结构,而必须依据“先来先服务”的原则,依次处理每个线程的请求,这就是一般性的客户/服务器模式。MiniGUI 利用线程之间的同步操作实现了客户线程和服务器线程之间的微客户/服务器机制,之所以这样命名,是因为客户和服务器是同一进程中的不同线程。

微客户/服务器机制的核心实现主要集中在消息队列数据结构上。比如,MiniGUI 中的 desktop 微服务器管理窗口的创建和销毁。当一个线程要求 desktop 微服务器建立一个窗口时,该线程首先在 desktop 的消息队列中放置一条消息,然后进入休眠状态而等待 desktop 处理这一请求。当 desktop 处理完成当前任务之后,或正处于休眠状态时,它可以立即处理这一请求。请求处理完成时,desktop 将唤醒等待的线程,并返回一个处理结果。

当 MiniGUI 在初始化全局数据结构以及各个模块之后,MiniGUI 要启动几个重要的微服务器,它们分别完成不同的系统任务:

- desktop 用于管理 MiniGUI 窗口中的所有主窗口,包括建立、销毁、显示、隐藏、修改 Z-order,获得输入焦点等等。
- parser 线程用来从 IAL 中收集鼠标和键盘事件,并将收集到的事件转换为消息而邮寄给 desktop 服务器。
- timer 线程用来触发定时器事件。该线程启动时首先设置 Linux 定时器,然后等待 desktop 线程的结束,即处于休眠状态。当接收到 SIGALRM 信号时,该线程处理该信号并向 desktop 服务器发送定时器消息。当 desktop 接收到定时器消息时,desktop 会查看当前窗口的定时器列表,如果某个定时器过期,则会向该定时器所属的窗口发送定时器消息。

10.2.4 面向对象技术的运用

1. 控件类和控件

MiniGUI 中的每个控件都属于某种子窗口类,是对应子窗口类的实例。这类似于面向对象技术中类和对象的关系。

每个控件的消息实际都是由该控件所属控件类的回调函数处理的,从而可以让每个属于统一控件类的控件均保持有相同的用户界面和处理行为。

但是,如果在调用某个控件类的回调函数之前,首先调用自己定义的某个回调函数的话,就可以让该控件重载控件类的某些处理行为,从而让该控件一方面继承控件类的大部分处理行为,另一方面又具有自己的特殊行为。这实际就是面向对象中的继承和派生。比如,一般的编辑框会接收所有的键盘输入,当希望自己的编辑框只接收数字时,就可以用这种办法屏蔽非数字的字符输入。

2. GAL 和 IAL

GAL 和 IAL 的结构是一样的,这里只拿 GAL 作为实例说明面向对象技术的运用,如图 10-8 所示。

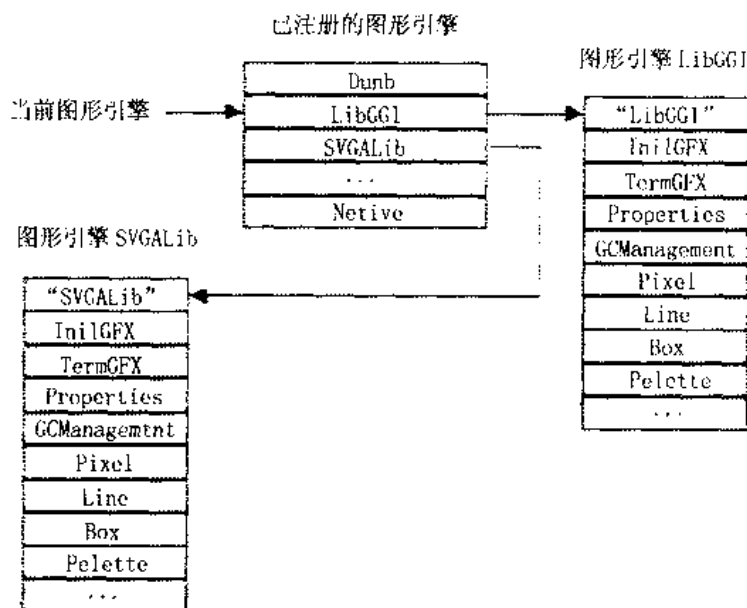


图 10-8 GAL 的构成

系统维护一个已注册图形引擎数组,保存每个图形引擎数据结构指针。系统利用一个指针保存当前使用的图形引擎。一般而言,系统中至少有两个图形引擎,一个是“哑”图形引擎,不进行任何实际的图形输出;一个是实际要使用的图形引擎,比如 LibGGI 或 SVGALib。每个图形引擎的数据结构定义了该图形引擎的一些信息,比如标识符、属性等;更重要的是,它实现了 GAL 所定义各个接口,包括初始化和终止、图形上下文管理、两点处理函数、画线处理函数、矩形框填充函数、调色板函数等等。

如果在某个实际项目中所使用的图形硬件比较特殊,现有的图形引擎均不支持时,就可以按照 GAL 所定义的接口实现自己的图形引擎,并指定 MiniGUI 使用这种私有的图形引擎即可。这种软件技术实际就是面向对象多态性的具体体现。

利用 GAL 和 IAL,大大提高了 MiniGUI 的可移植性,并且使程序的开发和调试变得更加容易。可以在 X Window 上开发和调试自己的 MiniGUI 程序,通过重新编译就可以让 MiniGUI 应用程序运行在特殊的嵌入式硬件平台上。

10.2.5 MiniGUI 的算法

1. 消息传递

现代 GUI 系统一般均使用消息或事件驱动的编程模型。因此,消息或事件的传递功能是 GUI 系统首先面对的一个重要功能。消息传递中需要注意一些问题:

- 消息一般附带有相关的数据,这些数据对各种消息具有不同的含义。在多窗口环境中,尤其是多进程环境下,消息数据的有效传递非常重要。
- 消息作为窗口间进行数据交换的一种方式,要提供多种传递机制。某些情况下,发送消息的窗口要等到这个消息处理完成且知道处理的结果之后才能继续执行;而在有些情况下,发送消息的窗口只是简单地向接收消息的窗口通知某些事件的发生,一般发送出消息之后就返回。后一种情况类似于邮寄信件,所以通常称为邮寄消息。更有一种较为复杂的情况,就是等待一个可能长时间无法被处理的消息时,发送消息的窗口设置一个超时值,以便能够在消息得不到及时处理的情况下能够恢复执行。
- 某些特殊消息的处理也需要注意,比如定时器。当某个定时器的频率很高,而处理这个定时器的窗口的反应速度又很慢,这时如果采用邮寄消息或者发送消息的方式,窗口的消息队列最终就会塞满。
- 最后一个问题是消息优先级的问題。一般情况下,要考虑优先处理鼠标或键盘的输入消息,其次才是 PAINT、定时器等消息。

2. 图形上下文和坐标映射

(1) 图形设备

在 MiniGUI 中,采用了在 Windows 和 X Window 中普遍采用的图形设备概念。每个图形设备都定义了计算机显示屏幕上的一个矩形输出区域。

在调用图形输出函数时,均要求指定经初始化,或经建立的图形设备上下文,或设备环境(DC)。每个图形输出均局限在图形设备指定的矩形区域内。在多窗口系统中,各个图形设备之间的输出互相剪切,以避免图形输出之间互相影响。

(2) 剪切域

剪切域就是在图形设备上定义的一个区域,所有在该图形设备上进行的图形输出,超过剪切域的部分,均被裁剪。只有在剪切域上的图形输出才是可见的输出。MiniGUI 中的剪切域,定义为矩形剪切域的集合。

(3) 映射模式

映射模式指定了特定图形输出的坐标值如何映射到图形设备的坐标值。

图形设备的坐标系原点定义为图形设备矩形区域的左上角。向右为正 X 坐标轴方向；向下为正 Y 坐标轴方向。这一坐标系称为设备坐标系。

通过 GDI 模块的映射模式操作函数，可定义自己的逻辑坐标系。逻辑坐标系可以是设备坐标系的水平或垂直反转、缩放或者偏移。

多数 GDI 输出函数指定的是逻辑坐标系。默认情况下，逻辑坐标系和设备坐标系是重合的。

3. 窗口管理

窗口管理和剪切的关系非常密切。当窗口 A 的一部分被另外一个窗口 B 覆盖时，在窗口 A 中任何一个绘图操作都不应该影响窗口 B。为了达到这个目标，窗口 A 中的绘图就要被剪切。而在多窗口环境中，哪个窗口的哪一部分应该被剪切就由窗口的 Z 序决定。窗口的建立、销毁、显示、隐藏等操作，均要修改窗口 Z 序。MiniGUI 实际维护着两个 Z 序，一个是普通窗口形成的 Z 序，另外一个顶层窗口（即永远处于普通窗口之上的窗口）形成的 Z 序。

4. 剪切算法

当因为窗口的互相覆盖产生剪切时，首先要有一个高效的剪切域维护算法。通常，窗口的剪切域定义为互不相交的矩形集合。GUI 系统的底层图形引擎在进行输出时，要根据当前输出的剪切域进行输出的剪切操作。MiniGUI 目前采用的图形引擎尚不支持剪切域，只支持剪切矩形，所以有一些性能损失。

10.3 MiniGUI 下的 Native Engine

MiniGUI 下的 Native Engine，即私有引擎，是为了解决 MiniGUI 使用通用图形引擎的不足，由宋立新先生开发出来的专用的图形引擎。本节将简要地介绍一下该私有引擎。

10.3.1 开发私有引擎的必要性

目前，MiniGUI 已经实现了比较成熟的窗口系统，最新的 0.9.6 已经支持子控件，使控件、窗口可以嵌套，同时工具栏也得到了增强。MiniGUI 也得到很多的应用（见应用实例），特别是随着 MiniGUI 被越来越多地应用到嵌入式系统上，使用别人的图形引擎越来越多地暴露出缺点。

在 10.2 小节提到，MiniGUI 目前采用的图形引擎尚不支持剪切域，只支持剪切矩形。事实上，在窗口系统中，剪切是一个非常关键的算法，可以说，剪切无处不在。剪切的效率大大地影响了整个窗口系统的响应时间。为什么 MiniGUI 不能直接利用剪切域进行剪切呢？因为 MiniGUI 使用的是别人的图形引擎。比如 LibGGI，它只能基于矩形剪

切。这样,上层做剪切时必须对一个矩形队列进行循环,例如,画一条水平线的伪代码如下:

```

pClipRect = pdc->ecrqn.head;
while(pClipRect)
{
    SetClipping (current _ gc, pClipRect->rc.left, pClipRect->rc.top,
                pClipRect->rc.right , pClipRect->rc.bottom );
    DrawHLine (current _ gc, x, y, startx - x, pdc->pencolor);
    pClipRect = pClipRect->next;
}

```



参数 `current _ gc` 里的 `gc` 表示 `graphics context`(图形上下文)。

而如果底层支持剪切域,代码就非常简单,下面是画同样一条水平线的伪代码:

```

SetClipping (current _ gc, pdc->ecrqn);
DrawHLine (current _ gc, x, y, startx - x, pdc->pencolor);

```

这样代码少多了。事实上,效率也提高了很多。如果考虑在同一剪切域下画多个 GDI 对象,那效率相差就更多了:剪切域实现只要调用一次 `SetClipping` 函数,而剪切矩形实现每次都要循环调用 `SetClipping`,每个循环里面都要调用 GDI 函数。这种效率的差别是惊人的。

此外,还有诸多原因导致要开发私有引擎,如效率因素、冗余代码、通用引擎存在的 bug 等。所以,有必要设计一个 `Native Engine` 既能够直接支持各种硬件,也能够支持各种通用引擎来更高效地实现各种图形引擎的通用功能,如剪切、底层 GDI 函数、异常处理、内存 GDI 等等。

10.3.2 Native Engine 的结构

从体系机构的角度看,`Native Engine` 接口 `HAL` 很简单。各驱动程序的结构如图 10-9 所示。

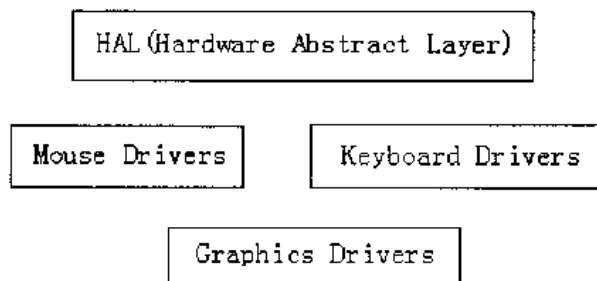


图 10-9 Native Engine 的结构

驱动程序包括鼠标驱动程序、键盘驱动程序和图形驱动程序。对上层来说,它只是调用 `HAL` 提供的功能。这些功能包括:

- 基本的初始化和终结函数: Init 和 Term。
- 鼠标和键盘的功能函数: 这在 Mouse 和 Keyboard Drivers 里介绍。HAL 主要是提供它们各函数的一个 Wrap。
- 图形输入/输出函数: 主要利用下层的图形驱动程序来完成各种功能。
- 基本 GDI 函数: 比如画一条斜线, 或者一个椭圆, 三次样条, 它们最终调用底层图形驱动程序完成任务。它主要是实现一些图形学算法, 之所以不放到更上层实现, 是为了提高效率。

剪切是一个窗口系统非常重要的功能, 其效率关系到图形系统的响应时间。某些剪切过程, 由于需要底层的一些信息, 必须在更底层实现才能做到高效。

相应于这些功能, HAL 实现了下列接口函数。

HAL _ Init	HAL _ Term		
HAL _ UpdateMouse	HAL _ GetMouseX	HAL _ GetMouseY	HAL _ WaitEvent
HAL _ SetMouseXY	HAL _ GetMouseButton	HAL _ SetMouseRange	
HAL _ UpdateKeyboard	HAL _ GetKeyboardState	HAL _ SetLeds	
HAL _ BitsPerPixel	HAL _ Width	HAL _ Height	HAL _ Colors
HAL _ AllocateGC	HAL _ FreeGC	HAL _ SetGC	
HAL _ EnableClipping	HAL _ DisableClipping	HAL _ SetClipping	HAL _ GetClipping
HAL _ GetBgColor	HAL _ GetFgColor	HAL _ SetBgColor	HAL _ SetFgColor
HAL _ MapColor	HAL _ UnmapPixel	HAL _ PackColors	HAL _ UnpackPixels
HAL _ GetPalette	HAL _ SetPalette	HAL _ SetColorfulPalette	
HAL _ FillBox	HAL _ PutBox	HAL _ GetBox	
HAL _ PutBoxMask	HAL _ PutBoxPart	HAL _ PubBoxWithOp	HAL _ ScaleBox
HAL _ CopyBox	HAL _ CrossBlit	HAL _ DrawHLine	HAL _ PutHLine
HAL _ GetHLine	HAL _ DrawVLine	HAL _ PutVLine	HAL _ GetVLine
HAL _ DrawPixel	HAL _ PutPixel	HAL _ GetPixel	HAL _ Ellipse
HAL _ Line	HAL _ Rectangle		

除了上述的这些接口函数, 还实现了一些内部功能函数, 一个重要的功能是虚屏切换。该功能的实现主要是为了 MiniGUI 开发的方便。比如, 因为某些 bug 导致程序不能正常响应, 就可以切换到其他屏幕, 将其杀死, 而不是重新启动机器。

此外, 还在 HAL 层实现一些剪切的功能函数, 比如剪切域的和差运算, 对点、线、矩形进行剪切。其他模块通常只要调用这些功能函数就能够完成剪切功能。剪切算法主要是利用了图形学里的区域编码裁剪算法 (Cohen-Sutherland 裁剪算法)。

硬件的自动配置功能也是在 HAL 上面实现。目前, 可直接通过配置文件决定使用什么硬件以及一些模式的选择。初始化时, HAL 根据配置文件, 选择相应的鼠标驱动程序、键盘驱动程序和图形驱动程序。下面将分别讨论这些驱动程序的结构。

10.3.3 鼠标驱动程序

鼠标驱动程序非常简单, 抽象意义上讲, 初始化鼠标后, 每次用户移动鼠标, 就可以得到一个 X 和 Y 方向上的位移值, 驱动程序内部维护鼠标的当前位置, 用户移动了鼠标后,

当前位置被加上位移值,并通过上层 Cursor 支持,反映到屏幕上,用户就会认为鼠标被正确地“移动”了。

事实上,鼠标驱动程序的实现是利用内核或其他驱动程序提供的接口来完成任务的。Linux 内核驱动程序使用设备文件对大多数硬件进行了抽象,比如,Ps/2 鼠标就是/dev/psaux,鼠标驱动程序接口如下:

```
typedef struct _mousedevice {
    int (* Open)(void);
    void (* Close)(void);
    int (* GetButtonInfo)(void);
    void (* GetDefaultAccel)(int * pscale,int * pthresh);
    int (* Read)(int * dx,int * dy,int * dz,int * bp);
    void (* Suspend)(void);
    void (* Resume)(void);
} MOUSEDEVICE;
```

现在有各种各样的鼠标,例如 MS 鼠标、PS/2 鼠标、总线鼠标和 GPM 鼠标,它们的主要差别在于初始化和数据包格式。

例如,打开一个 GPM 鼠标非常简单,只要将设备文件打开就可以了,当前终端被切换到图形模式时,GPM 服务程序就会把鼠标所有的位移信息放到设备文件中去。

```
static int GPM_Open(void)
{
    mouse_fd = open(GPM_DEV_FILE, O_NONBLOCK);
    if (mouse_fd < 0)
        return -1;
    return mouse_fd;
}
```

对于 PS/2 鼠标,不但要打开它的设备文件,还要往该设备文件写入控制字符,以使鼠标能够开始工作。

```
static int PS2_Open(void)
{
    uint8 initdata_ps2 = { PS2_DEFAULT, PS2_SCALE11, PS2_ENABLE };
    mouse_fd = open(PS2_DEV_FILE, O_RDWR | O_NOCTTY | O_NON-
BLOCK);
    if (mouse_fd < 0)
        return -1;
```

```

write(mouse_fd, initdata_ps2, sizeof(initdata_ps2));
return mouse_fd;
|

```



各鼠标的数据包格式是不一样的。而且在读这些数据时,首先要根据内核驱动程序提供的格式读数据,还要注意同步:每次扫描到一个头,才能读后面相应的数据。由于 MicroWindows 没有同步,在某些情况下,鼠标就会不听“指挥”。

鼠标驱动程序中,还有一个“加速”的概念。程序内部用两个变量:scale 和 thresh 来表示。当鼠标的位移超过 thresh 时,就会被放大 scale 倍。这样,最后的位移就是:

```

dx = thresh + (dx - thresh) * scale;
dy = thresh + (dy - thresh) * scale;

```

至此,鼠标驱动程序基本上很清楚了,上面的接口函数中 GetButtonInfo 用来告诉调用者该鼠标支持那些 button 的,suspend 和 resume 函数是用来支持虚屏切换的,下面的键盘驱动程序也一样。

10.3.4 键盘驱动程序

在实现键盘驱动程序中遇到的第一个问题就是使用设备文件/dev/tty 还是/dev/tty0。例如:

```

# echo 1 > /dev/tty0
# echo 1 > /dev/tty

```

结果都将把 1 输入到当前终端上。另外,如果从伪终端上运行它们,则第一条指令会将 1 输出到控制台的当前终端,而第二条指令会把 1 输出到当前伪终端上。tty0 表示当前控制台终端,tty 表示当前终端(包括伪终端)。

tty0 的设备号是 4,0(4 为主设备号,0 为从设备号);tty1 的设备号是 5,0。

☞ 注意:/dev/tty 是和进程的每一个终端联系起来的,/dev/tty 的驱动程序所作的只是把所有请求送到合适的终端。

默认情况下,/dev/tty 是普通用户可读写的,而/dev/tty0 则只有超级用户能够读写。基于这个原因,目前使用/dev/tty 作为设备文件。后面所有有关终端处理的程序都采用它作为当前终端文件,这样也可以和传统的 UNIX 相兼容。

键盘驱动程序接口如下:

```

typedef struct _kbddevice {

int (* Open)(void);

void (* Close)(void);

void (* GetModifierInfo)(int * modifiers);

int (* Read)(unsigned char * buf,int * modifiers);

void (* Suspend)(void);

```

```
void (* Resume)(void);
} KBDDEVICE;
```

基本原理非常简单,初始化时打开/dev/tty,以后就从该文件读出所有的数据。由于 MiniGUI 需要捕获 KEY_DOWN 和 KEY_UP 消息,键盘被置于原始(RAW)模式。这样,程序从/dev/tty 中直接读出键盘的扫描码,比如用户按下 A 键,程序就读到了 A 键对应的扫描码,松开则又读到取消码(30)。原始模式下,程序必须自己记下各键的状态,特别是 Shift、Ctrl、Alt、Caps lock 等,所以程序维护一个数组,记录了所有键盘的状态。

这里说明一下鼠标移动、按键等事件是如何被传送到上层消息队列的。MiniGUI 工作在用户态,所以它不可能利用中断这种高效的机制。没有内核驱动程序的支持,它也很难利用信号等 UNIX 系统的 IPC 机制。MiniGUI 可以做到的就是看/dev/tty、/dev/mouse 等文件是否有数据可读。上层通过不断调用 HAL_WaitEvent 尝试读取这些文件。这也是线程 Parser 的主要任务。HAL_WaitEvent 主要利用了 select 这一类系统调用完成该功能,select 调用在 UNIX 系统中地位仅次于 ioctl 调用。然后,将等待到的事件作为返回值返回。

至此介绍了键盘和鼠标的驱动程序。作为简单的输入设备,它们的驱动是非常简单的。事实上,它们的实现代码也比较少,就是在嵌入式系统中要使用的触摸屏,如果操作系统内核支持,其驱动程序也是非常简单的。它只不过是一种特殊的鼠标,就如同在上一章中讲到的华恒公司开发平台的触摸屏驱动程序一样。相比较而言,下面要学习的图形驱动就要复杂多了,也是 Native Engine 实现的关键。

10.3.5 图形驱动程序

图形驱动程序目前已经提供了基于 Linux 内核提供 Framebuffer 之上的驱动。在下一步的工作中,将编写基于 X Window 的图形驱动程序,以便于应用程序的调试。

前面已经看到,HAL 提供的接口函数大多数与图形相关,它们主要就是通过调用图形驱动程序来完成任务的。图形驱动程序屏蔽了底层驱动的细节,完成底层驱动相关的功能,而不是与硬件相关的一些功能,如一些画圆、画线的 GDI 函数、普通的剪切,可直接在 HAL 接口层实现。

下面在已经实现的基于 Framebuffer 的驱动程序的基础上,讲一些实现上的细节。首先列出核心数据结构 SCREENDEVICE。为了讲解方便,这里删除了一些次要的变量或函数。

```
typedef struct _screendevic {
    int xres;          /* X screen res (real) */
    int yres;          /* Y screen res (real) */
    int planes;        /* # planes */
    int bpp;           /* # bits per pixel */
    int linelen;       /* line length in bytes for bpp 1,2,4,8, line length in pixels for
```

bpp 16,

```

        * 24, 32 */
int size;      /* size of memory allocated */
gfx_pixel gr_foreground; /* current foreground color */
gfx_pixel gr_background; /* current background color */
int gr_mode;

int flags;    /* device flags */
void * addr; /* address of memory allocated (memdc or fb) */

PSD (* Open)(PSD psd);
void (* Close)(PSD psd);
void (* SetPalette)(PSD psd, int first, int count, gfx_color * cmap);
void (* GetPalette)(PSD psd, int first, int count, gfx_color * cmap);
PSD (* AllocateMemGC)(PSD psd);
BOOL (* MapMemGC)(PSD mempsd, int w, int h, int planes, int bpp, int linelen, int
size, void * addr);
void (* FreeMemGC)(PSD mempsd);
void (* FillRect)(PSD psd, int x, int y, int w, int h, gfx_pixel c);
void (* DrawPixel)(PSD psd, int x, int y, gfx_pixel c);
gfx_pixel (* ReadPixel)(PSD psd, int x, int y);
void (* DrawHLine)(PSD psd, int x, int y, int w, gfx_pixel c);
void (* PutHLine)(HAL hal, int x, int y, int w, void * buf);
void (* GetHLine)(HAL hal, int x, int y, int w, void * buf);
void (* DrawVLine)(PSD psd, int x, int y, int w, gfx_pixel c);
void (* PutVLine)(HAL hal, int x, int y, int w, void * buf);
void (* GetVLine)(HAL hal, int x, int y, int w, void * buf);
void (* Blit)(PSD dstpsd, int dstx, int dsty, int w, int h, PSD srcpsd, int srcx, int
srcy);
void (* PutBox)(HAL hal, int x, int y, int w, int h, void * buf);
void (* GetBox)(HAL hal, int x, int y, int w, int h, void * buf);
void (* PutBoxMask)(HAL hal, int x, int y, int w, int h, void * buf);
void (* CopyBox)(PSD psd, int x1, int y1, int w, int h, int x2, int y2);

```



```
}; SCREENDEVICE;
```

上面 PSD 是 SCREENDEVICE 的指针, HAL 是 HAL 接口的数据结构。图形显示有个显示模式的概念, 一个像素可以用一位比特表示, 也可以用 2、4、8、15、16、24、32 个比特表示。另外, VGA16 标准模式使用平面图形模式, 而 VESA2.0 使用的是线性图形模式。所以即使是同样基于 Framebuffer 的驱动, 不同的模式也要使用不同的驱动函数: 画一个 1 比特的单色点和画一个 24 位的真彩点显然是不一样的。

基于这些原因, 图形驱动程序使用了子驱动程序的概念来支持各种不同的显示模式, 事实上, 它们才是最终的功能函数。为了保持数据结构在层次上不至于很复杂, 可通过图形驱动程序的初始函数 Open, 直接将子驱动程序的各功能函数赋到图形驱动程序的接口函数指针上, 从而初始化结束就使用一个简单的图形驱动接口。下面是子图形驱动程序接口:

```
typedef struct {
    int (* Init)(PSD psd);
    void (* DrawPixel)(PSD psd, int x, int y, gfx_pixel c);
    gfx_pixel (* ReadPixel)(PSD psd, int x, int y);
    void (* DrawHLine)(PSD psd, int x, int y, int w, gfx_pixel c);
    void (* PutHLine)(HAL hal, int x, int y, int w, void * buf);
    void (* GetHLine)(HAL hal, int x, int y, int w, void * buf);
    void (* DrawVLine)(PSD psd, int x, int y, int w, gfx_pixel c);
    void (* PutVLine)(HAL hal, int x, int y, int w, void * buf);
    void (* GetVLine)(HAL hal, int x, int y, int w, void * buf);
    void (* Blit)(PSD dstpsd, int dstx, int dsty, int w, int h, PSD srcpsd, int srcx, int srcy);
    void (* PutBox)(HAL hal, int x, int y, int w, int h, void * buf);
    void (* GetBox)(HAL hal, int x, int y, int w, int h, void * buf);
    void (* PutBoxMask)(HAL hal, int x, int y, int w, int h, void * buf);
    void (* CopyBox)(PSD psd, int x1, int y1, int w, int h, int x2, int y2);
}; SUBDRIVER, * PSUBDRIVER;
```

可以看到, 该接口中除了 Init 函数指针外, 其他函数指针都与图形驱动程序接口中的函数指针一样。这里, Init 函数主要用来完成图形驱动部分与显示模式相关的初始化任务。

下面介绍 SCREENDEVICE 数据结构, 这样基本上就可以清楚图形引擎了。

一个 SCREENDEVICE 代表一个屏幕设备, 它既可以对应物理屏幕设备, 也可以对应一个内存屏幕设备。内存屏幕设备的存在主要是为了提高 GDI 质量, 比如先在内存中生成一幅位图, 再画到屏幕上, 这样给用户的视觉效果就比较好。首先介绍几个变量。

- xres 表示屏幕的宽(以像素为单位)。

- `yres` 表示屏幕的高(以像素为单位)。
- `planes`: 当处于平面显示模式时, `planes` 用于记录所使用的平面数。如平面模式相对时线性模式, 此时该变量没有意义。通常将其置为 0。
- `bpp`: 表示每个像素所使用的比特数, 可以为 1、2、4、8、15、16、24、32。
- `linelen`: 对于 1、2、4、8 比特每像素模式, 它表示一行像素使用的字节数, 对于大于 8 比特每像素模式, 它表示一行的像素总数。
- `size`: 表示该显示模式下该设备使用的内存数。 `linelen` 和 `size` 的存在主要是为了方便为内存屏幕设备分配内存。
- `gr_foreground` 和 `gr_background`: 表示该内存屏幕的前景颜色和背景颜色, 主要被一些 GDI 函数使用。
- `gr_mode`: 说明如何将像素画到屏幕上, 可选值为: `MODE_SET`、`MODE_XOR`、`MODE_OR`、`MODE_AND`、`MODE_MAX`, 比较常用的是 `MODE_SET` 和 `MODE_XOR`。
- `flags`: 该屏幕设备的一些选项, 比较重要的是 `PSF_MEMORY` 标志, 表示该屏幕设备代表物理屏幕设备还是一个内存屏幕设备。
- `addr`: 每个屏幕设备都有一块内存空间用来作为存储像素。 `addr` 变量记录了这个空间的起始地址。

下面介绍各接口函数:

(1) `Open` 和 `Close`

它是基本的初始化和终结函数。前面已经提到, 在 `Open` 函数里要选择子图形驱动程序, 将其实现的函数赋予本 PSD 结构的函数指针。这里讲基于 Framebuffer 图形引擎的初始化。

`fb_open` 首先打开 Framebuffer 的设备文件 `/dev/fb0`, 然后利用 `ioctl` 读出当前 Framebuffer 的各种信息, 填充到 PSD 结构中。并且根据这些信息选出子驱动程序。程序当前支持 `fbvga16`、`fblin16` 和 `fblin8`, 即 VGA16 标准模式、VESA 线性 16 位模式、VESA 线性 8 位模式。然后将当前终端模式置于图形模式, 并保存当前的一些系统信息如调色板信息。最后, 系统利用 `mmap` 将 `/dev/fb0` 映射到内存地址。以后程序访问 `/dev/fb0` 就像访问一个数组一样简单。当然, 这是对线性模式而言的, 如果是平面模式, 问题要复杂得多。光从代码来看, 平面模式的代码长度是线性模式的一倍左右。

(2) `SetPalette` 和 `GetPalette`

当使用 8 位或 8 位以下的图形模式时, 要使用系统调色板。这里是调色板处理函数, 它们和 Windows API 中的概念类似, Linux 系统利用 `ioctl` 提供了处理调色板的接口。

(3) `AllocateMemGC`、`MapMemGC` 和 `FreeMemGC`

前面屡次提到内存屏幕的概念, 内存屏幕是一个伪屏幕, 在对屏幕图形操作过程中, 比如移动窗口, 可先生成一个内存屏幕, 将物理屏幕的一个区域拷贝到内存屏幕, 再拷贝到物理屏幕的新位置, 这样就减少了屏幕直接拷贝的延时。 `AllocateMemGC` 用于给内存屏幕分配空间, `MapMemGC` 做一些初始化工作, 而 `FreeMemGC` 则释放内存屏幕。

(4) DrawPixel、ReadPixel、DrawHLine、DrawVLine 和 FillRect

这些是底层图形函数,分别是画点、读点、画水平线、画竖直线和画一个实心矩形。所以在底层实现这么多函数,是为了提高效率。图形函数支持多种画图模式,常用的有直接设置或 Alpha 混合模式,从而可以支持各种图形效果。

(5) PutHLine、GetHLine、PutVLine、GetVLine、PutBox、GetBox 和 PutBoxMask

Get * 函数用于从屏幕拷贝像素到一块内存区,而 Put * 函数用于将存放于内存区的像素画到屏幕上。PutBoxMask 与 PutBox 的唯一区别是要画的像素如果是白色,就不会被画到屏幕上,从而达到一种透明的效果。

从上面可以看到,这些函数的第一个参数是 HAL 类型而不是 PSD 类型,这是因为它们需要 HAL 层的信息以便在函数内部实现剪切功能。之所以不和其他函数一样在上层实现剪切,是因为这里的剪切比较特殊。比如 PutBox,在剪切输出域时,要同时剪切在缓冲中待输出的像素,超出剪切域的像素不应该被输出。所以,剪切已经不单纯是对线,矩形等 GDI 对象的剪切,对像素的剪切当然需要知道像素的格式。这些只是为底层所有,所以为了实现高效的剪切,则可选择在底层实现它们。这里所有的函数都有两个部分:先是剪切,再是读或写像素。

(6) Blit 和 CopyBox

Blit 用于在不同的屏幕设备(物理的或内存的)之间拷贝一块像素点, CopyBox 则用于在同一屏幕上实现区域像素的拷贝。如果使用的是线性模式, Blit 的实现非常简单,直接拷贝内存就可以了,而 CopyBox 为了防止覆盖问题,必须根据不同的情况,采用不同的拷贝方式。比如从底到顶的拷贝,当新老位置在同一水平位置并且重复时,则需要利用缓冲间接拷贝。如果使用平面显示模式,这里就比较复杂了。因为内存设备总是采用线性模式的,所以就要判断是物理设备还是内存设备,再分别处理。这也大大地增加了 fbvga16 实现的代码。

10.3.6 Native Engine 的典型应用

使用 Native Engine,系统的响应平滑了很多。这里介绍一些典型的应用。

1. 应用一 —— HappyLinux 安装程序

HappyLinux 是由联想公司开发的 Linux 简体中文发行版。安装程序基于 MiniGUI 开发。安装程序的典型屏幕如图 10-10 所示。

2. 应用二 —— 游戏

Bomb 游戏,是 Windows 扫雷游戏的克隆,如图 10-11 所示。

3. 应用三 —— ehome

ehome 是一个采用 Web 技术进行小区网上购物、物业管理的智能家居设备。其客户端实际是一个 Web 浏览器,图 10-12 为正在进行网上购物的 ehome。

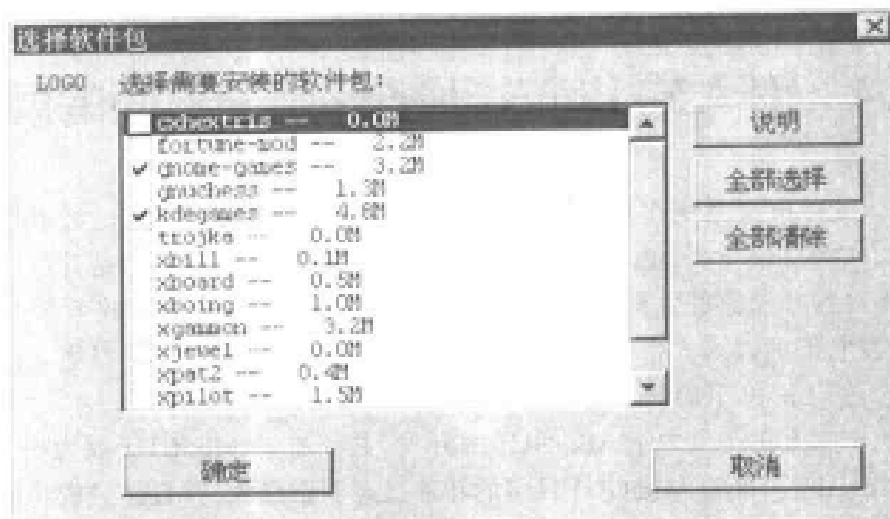


图 10-10 HappyLinux 的软件包选择界面

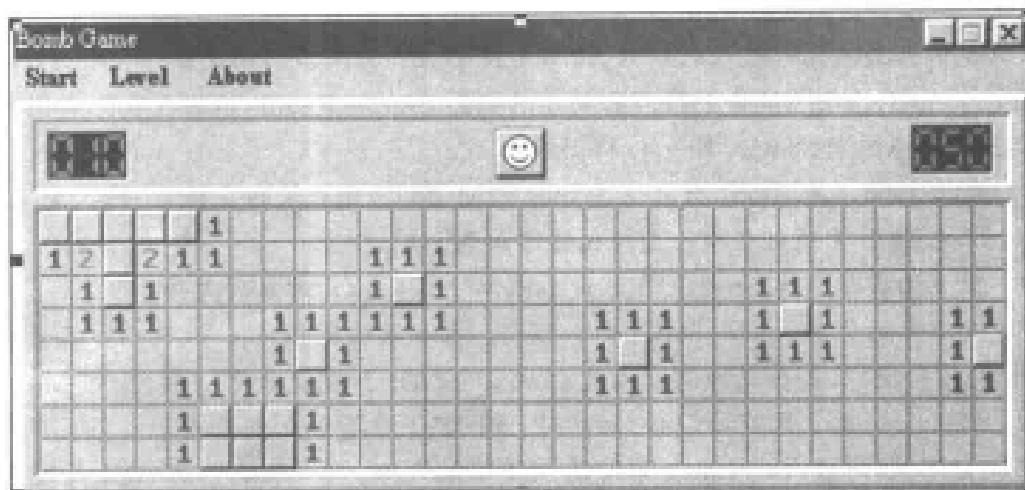


图 10-11 扫雷游戏的克隆

深圳蓝点公司承办智能家居系统北京研发中心开发

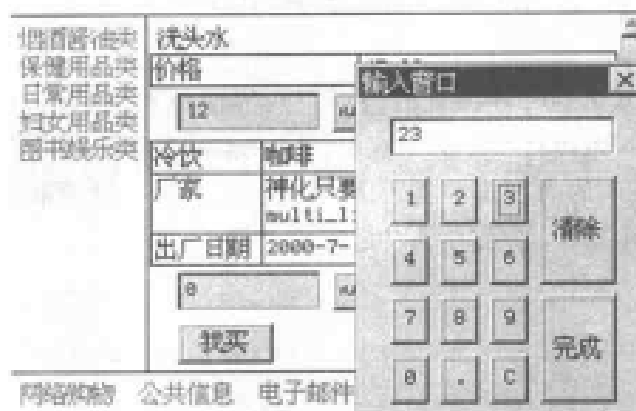


图 10-12 正在进行网上购物的 chrome 客户端

10.4 嵌入式 Linux 下图形用户界面的展望

实时嵌入式 Linux 系统在自由软件社区和众多开发者的推动下,已经取得了长足的发展。同样,作为底层支持的 GUI 系统来讲,也应该像操作系统一样是开放源码的自由软件,并应该得到开发者的共同推动。到目前为止,已经有多家嵌入式系统开发商采用 MiniGUI 开发它们的嵌入式系统,并且已经开发出了许多重要的应用程序。很多人都已经加入到嵌入式 Linux 下的图形用户界面的开发中来。

在我国,不少人正在致力于 MiniGUI 的开发,共同推动 MiniGUI 成为嵌入式 Linux 系统上的标准 GUI。尽管 MiniGUI 目前的功能已经非常强大,并且已经成功应用于许多嵌入式系统,但还需要在如下几个领域进行开发:

- 建立一个 C++ 类库来封装 MiniGUI 的 API。
- 建立基于传统进程级客户/服务器体系结构的 MiniGUI,以便能够让 MiniGUI 适合于一些较大的嵌入式系统,比如支持全功能浏览器的机顶盒。

随着 MiniGUI 不断地得到推广和功能的加强,Native Engine 也将在实际应用中不断走向成熟。具体说来,Native Engine 将提供:

- 更多的驱动程序,包括对 X Window、Mode X 显示等的支持。同时,对 Libggi、SVGALib 等通用引擎的支持也将添加到 HAL 下。
- 更多的图形功能,如对多边形的支持和对更多 GDI 对象的支持。
- 更成熟的代码,使 MiniGUI 更加稳定。主要是处理各种信号、中断以及多线程。

同时,在全世界,各种其他的嵌入式 Linux 的 GUI 开发也非常活跃,例如 MicroWindows、qt/embedded 等。用户可以登录到它们的网站上参看最新的进展。

总而言之,实时嵌入式 Linux 系统的前景非常广阔,这一系统上的 GUI 开发正处于开始阶段,在这个领域,有许多技术难题等待人们来解决。

10.5 本章小结

本章着眼于嵌入式 Linux 环境下的图形用户界面技术,以 MiniGUI 为例,详细地讲解了 MiniGUI 的结构、面向对象方法的应用、算法、驱动程序以及私有引擎的开发等关键问题。由于嵌入式 GUI 的内容非常复杂,在此只能择其最关键的部分讲述。本章内容很有分量,加之 GUI 的重要性,请读者予以足够的重视。

第 11 章 uClinux 的移植



本章专门讨论了将 uClinux 移植到新的硬件平台上的思想和实现方法。本章从编译工具和设备驱动两个方面着手,详细地讨论了诸如内核裁剪、设备驱动、对 flat 格式内存的支持等在移植时需要重点考虑的问题。

本章主要内容:

- 交叉开发工具
- C 编译器
- 对内核的裁剪
- 调度任务
- 内核函数库
- 串行口的驱动
- 运行时库
- 堆栈分配

11.1 uClinux 的移植简介

Linux 操作系统在桌面和服务器领域中引起了一场革命。由于开放源代码内核的稳定性和其强大的网络功能,这场运动的影响是很深远的。这些原因,也是采用 Linux 来作为支持嵌入式网络应用程序操作系统的驱动力所在。虽然 Linux 的标准内核支持大部分的应用,但是对于内存管理单元(MMU)的需求限制了它在某些嵌入式领域中的应用。由于 Microcontroller Linux(即 uClinux),实现了对 flat 型内存模式的支持,那么它就不再需要 MMU 了。因此就可以使许多新的结构体系来利用内核的各种功能和现有的各种开放源代码的应用程序。以前,支持这些微控制器的操作系统都是一些实时操作系统(RTOS)。构建绝大多数的 RTOS 环境的方法,是先建立一个基本的任务内核,然后再往里面添加诸如 TCP/IP 协议栈或建立 POSIX 环境等要素。软件的模块化不可避免地导致了系统性能的降低和模块依存性的复杂化,从而使得走 RTOS 这条路有不少的困难且行不通。

由于软件模块化是 Linux 内核的固有特征,而且 Linux 的内核已经经受了跨平台的检测,所以软件对象模块间是无缝集成的。例如,采用 NET+ARM 构架的 NETsilicon 公司的嵌入式网络处理器(<http://www.netsilicon.com/>),就利用了 Linux 的这个优势。

NET+ARM 处理器基于 ARM7TDMI 内核,它缺乏内存管理功能,但是它的特点是集成了许多的外部设备。如 10/100Base-T 以太网口、异步串行外设、SPI 口、HDLC 模块、

IEEE1284 兼容并行口和支持数据在外设和内核间传输的多 DMA 引擎。用户程序可以在内存中任何地址读取和写入数据。内存段给予了 Linux 的应用程序基本上是无限制的堆栈空间。对于可执行程序, uClinux 在数据段的末端分配堆栈空间。如果 uClinux 的堆栈过大的话, 将会覆盖掉静态的数据和代码区域。对于物理内存的使用是有限的, 这是因为不存在交换空间的缘故。

这里 Linux(uClinux 也一样)将“体系”和“处理器”这两个概念分开对待。“处理器”的概念是指一个家族的处理器, 如 Intel 386 的 i386 处理器和其后的处理器, 或者例如 Motorola 68000 的 m68k 处理器和其衍生的产品。“体系”指的是处理器所处的系统。例如, 同样属于 m68k 型处理器家族, Motorola MVME147 VME 处理器和 Macintosh II 就是属于不同的体系。本章将假定, 新体系的 CPU 也包括新型的处理器的。

在本章讨论的是如何将 uClinux 操作系统移植到新硬件结构的实现方法。这里将从以下的各个方面来探讨这个问题。

11.2 交叉开发工具

要让 uClinux 支持一个新硬件体系, 要做的第一件事情就是收集和构建代码的工具。

开发工具是开发应用程序的基础。GNU 工具是一个极好的选择, 人们使用它已经编译出了许多代码。另外一个优势是 GNU 工具是以源代码的方式发行的, 因此可以通过打补丁来不断地增加工具的各种功能。但是, 即使 GNU 工具支持开发所选定的处理器, 但是为了生成所需格式的目标文件 and 应用程序的启动代码(已经讲过, 启动代码是用来为高级语言写的软件做好运行前准备的一小段汇编语言), 还必须对 GNU 工具做些小小的变动。

某些情况下, 选择别的开发工具可能会更加有效。重要的是, 该工具集必须有足够的灵活性, 以迎合系统的需要。

对于生成内核代码和用户代码而言, 其要求是不一样的。内核代码是静态地链接到内存的一个特别的地址, 通常是物理内存的基地址处。所有的数据和程序的地址可以是绝对的, 也可以是相对的。这给予编译器更大的自由来优化代码。对于那些将要从文件系统中载入的用户程序而言, 必须使用相对地址, 并且必须支持重新定址。

对于开发工具的讨论重点将放在 GNU 工具的设置上。GNU 的文档中对处理器的低级支持已经有了很详细充分的说明。这里将重点讨论 uClinux 环境下特定的问题, 如对 binutils (包括汇编器和链接器)、编译器 GCC、调试器 GDB 和其他工具的设置。GNU 的所有工具都使用了由 autoconf 生成的设置脚本文件。有一些通用的参数来指定诸如安装文件目录、目标机体系结构、主机体系结构(如例 11-1 中的参数分别对应着/usr/local/armtools _glibc、arm-uClinux 和 i386-pc-linux)等内容。

例 11-1 使用 Bourne shell 脚本来为 GNU 工具设置参数

```
#! /bin/sh
export CC="gcc"
```

```
/bin/sh ../arm_uClinux_gcc/configure < \ \ >
--prefix = "/usr/local/armtools_glibc" --build = i386-pc-linux < \ \ >
--host = i386-pc-linux --target = arm-uClinux --enable-languages =
```

1. 二进制工具 (Binutils)

GNU binutils 包包括了汇编工具、链接器及基本的目标文件处理工具。对 binutils 包的设置定义了所需的目标文件的格式和字节顺序。Binutils 包中的工具都使用了二进制文件描述符 (BFD) 库来交换数据。通过设置文件 config.bfd, 可以指定默认的二进制文件格式 (例如 elf little endian) 和任何工具可用的格式, 见例 11-2。

例 11-2 在 config.bfd 中添加的用来指定目标二进制格式的代码

```
arm- * -uClinux * | armel- * -uClinux * )
targ _ defvec = bfd _ elf32 _ littlearm _ vec
targ _ selvecs = "bfd _ elf32 _ bigarm _ vec armcoff _ little _ vec armcoff _ big _ vec"
```

2. C 编译器

GNU 编译器集 GCC 是通过使用一种叫做“寄存器转换语言”(RTL)的方式实现的。假定现在有一个基本的机器描述性文件, 它已经能满足大家的需要。现在要做的仅仅是设置默认情况下使用的参数和如何将文件组合成可执行文件的方式。GNU 的文档提供了所有必需的资料, 使得用户可以为新型的处理器的指令集合提供支持。如果要针对体系的机器建立一个新的目标机器, 那么就必须指定默认编译参数和定制系统的特定参数, 见例 11-3。对于特定的目标系统, 可以使用 TARGET_DEFAULT 宏来在 target.h 文件中定义编译器的开关。目标 t-makefile 段指定了应该构建哪一个额外的例程和其编译的方式。

例 11-3 使用 uClinux-arm.h 来指定默认的编译参数 (摘录)

```
# undef TARGET_DEFAULT
# define TARGET_DEFAULT (ARM_FLAG_APCS_32 | ARM_FLAG_NO
GOT)
```

例 11-4 为了生成 crtbegin.o 和 crtend.o 而往 makefile 片段 t-arm-uClinux 中添加的内容

```
MULTILIB_OPTIONS = mno-got
MULTILIB_DIRNAMES = pic
EXTRA_MULTILIB_PARTS = crtbegin.o crtend.o
```

3. 调试器

对于开发系统软件而言, 对调试功能的支持是必不可少的。GNU 调试器 GDB 提供了符号化的源代码调试功能。通过某种处理器模块, 例如 Motorola BDM and ARM Em-

beddedICE 接口,就可以实现对这些处理器的本地调试。如果不提供硬件支持的话,使用 kGDB 的内核模块补丁就可以实现远端的 GDB 调试。

4. 目标格式工具

由于 GNU 工具本身并不支持用户程序的 flat 二进制格式文件,于是人们开发出了二进制应用工具,来生成 flat 格式的二进制文件。例如,uClinux 就使用了 Cofl2flt 和 elf2flt 来将二进制文件转换成 uClinux 内核装载工具 fs/binfmt ...flat.c 使用的 flat 格式的文件。

5. 内核的裁剪

虽然 Linux/uClinux 的内核代码的大部分是独立于处理器和其体系结构的,但是其最低级的代码是特定于各个系统的。虽然各个系统中存在有相同之处,但是它们的中断处理上下文、内存映射的维护、任务上下文和初始化过程都是独特的。这些例行程序在代码树(所有的源代码都是按目录方式树状放置的,称为代码树)中被单独放置在 arch/目录下。由于 Linux 支持的体系结构的种类是很多的,所以对于一个新型的体系,其低级例程可以模仿其相似的体系例程编写。同时,初始化串行端口传输调试信息对于这些低级例程的运行也很有用。

对于许多将要移植 uClinux 的目标系统而言,它们一般都有串行终端设备。当装载串行设备驱动程序时,可以有选择的向系统注册该串行终端。尽可能早地提供一个 printk 调试信息的单向输出通道,可以确保有一个足够的可视化的环境,这样可以加快开发。

相对处理器的体系而言,在本小节所论述的问题同处理器所处的家族更加联系紧密。某些例子中,也涉及到了体系结构的问题。如果某个系统使用一个外部设备来在不同的中断源间共享一个中断行,就会出现这类问题。这些级连的中断源需要在中断处理向量中进行处理(好像它们是额外的中断行一样),而不仅仅处理处理器特定的中断。

6. 初始化

初始化有两个阶段:bootloader 的低级初始化阶段,负责处理重新启动系统,为系统执行 start _ kernel 程序作好准备;第二个阶段是初始环境设置。虽然某些任务可以在两个阶段中的任意一个中执行,但是用户的目的是使在 bootloader 中的工作尽可能少。bootloader 通常是同系统的体系密切相关的,然而用户可以确保设置代码具有最大的通用性。启动初始化包括了所有的操作,从重启系统一直到调用语句的入口点。初始化代码(head.S)负责设置系统的基本操作,并将系统的所有外部设备设置为确定(静止状态为好)的状态。这包括了设置内存的读取和映射、拷贝任何需要重新定位的数据段、为启动例程建立堆栈等过程。

初始化堆栈可以非常小,而且可以映射到内存空间的一个未保留段中,在内核内存分配逻辑初始化完成后,还可以将其收回。一旦系统处于该状态时,就调用 start _ kernel (在 init/main.c 中),它是 C 语言的内核入口点。环境设置代码(setup _ arch,在 arch/ kernel/setup.c 中)建立了基本的系统参数。这些参数包括了现有的内存空间的起始和终止

地址,对从 loader 传递过来的任何命令行参数进行解析,并有选择地为内核指定了一个在它启动时要执行的基本任务。setup_arch 例程也是一个调用 register_console 来建立早期的终端调试信息的好地方。

7. 调度任务

uClinux 下的多任务比 Linux 要容易,这是由于在 uClinux 下不处理预处理页表和保护模式。内核的调度没有做任何的改变,所做的实际工作仅是对上下文的保存和恢复,在功能上同系统的调用是共享的。该上下文信息通常包括了在某个进程被中断时要保存的所有寄存器的值。上下文寄存器常常保存在 struct pt_regs 结构中(文件 asm/ptrace.h 中定义了该结构,这个结构是系统体系所特定的)。该结构表示的顺序必须同寄存器在中断时被压入的顺序一致。另外,如果本体系的计算机使用了寄存器来传递返回值,可能需要存储从那个寄存器(在样例代码中是 ARM_ORIG_r0)返回的初始值。当系统从某个系统调用返回时,会将返回值放在参数寄存器中,然而当系统从某个中断返回时,必须恢复初始值。

例 11-5 定义寄存器来支持上下文的保存和恢复

```
struct pt_regs {
long uregs[18];
};
#define ARM_cpsr uregs[16]
#define ARM_pc uregs[15]
#define ARM_lr uregs[14]
#define ARM_sp uregs[13]
#define ARM_ip uregs[12]
#define ARM_fp uregs[11]
#define ARM_r10 uregs[10]
#define ARM_r9 uregs[9]
#define ARM_r8 uregs[8]
#define ARM_r7 uregs[7]
#define ARM_r6 uregs[6]
#define ARM_r5 uregs[5]
#define ARM_r4 uregs[4]
#define ARM_r3 uregs[3]
#define ARM_r2 uregs[2]
#define ARM_r1 uregs[1]
```

```
#define ARM_r0 uregs[0]
#define ARM_ORIG_r0 uregs[17] /* -1 */
```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

8. 内存记数

由于没有虚拟内存管理的任务,内存管理的任务就是负责为系统资源进行记数。虽然该进程是独立于系统体系的,paging_init 例程产生了空闲的内存区域的基本集合。如果系统是用一个稀疏分布的内存映射所设置的话,该例程可以将各个内存段独立地加入内存映射图。

9. 中断

中断管理给系统提供了一个最大的挑战。虽然硬件中断处理子程序必须是处理器特定的,但是,必须提供一些系统的入口点,它们包括:enable_irq、disable_irq、get_irq_list、request_irq、free_irq、probe_irq_on、probe_irq_off 及 init_IRQ。内核的其他部分和驱动程序调用了这些例程来控制系统的行为。

10. 记时器

记时处理程序包括了日期时间和系统周期中断。对于系统功能的惟一实际需求是系统的周期性中断。系统的“心跳”对于触发系统的许多行为是相当关键的。内核调用了 time_init 例程(它与系统硬件体系相关),来初始化系统的时间和中断,然后等待中断以大约 100Hz 的频率来触发一个对 do_timer 的调用。

11. 内核函数库

Linux 使用了一个基本的库函数来支持 C 函数库(C 函数库是可以被分别替换的)。一个较好的方法是通过提供一个处理器特定的子例程库来代替原有的函数库以优化内核性能,虽然这并不是必须执行的。对诸如 memset 之类的函数的调用贯穿于整个内核中。显然,这些优化是级别较低的,但是却很有效。

11.3 设备驱动程序

一旦内核支持基本的处理器功能,它就可以运作了,但是如果缺乏驱动程序的支持,那么就不会发生什么有趣的事情。典型的驱动程序集合中包括了诸如控制台终端、基本串行设备和一个可能包含了根文件系统的块设备的驱动等。很有趣的是,这些驱动程序都是被严格指定的,这是由于它们并不是直接由某个用户进程使用的,而是已经由另外一个内核模块(例如 tty 的驱动程序或某个文件系统)来使用。

1. 控制台终端

要提供一个完整的串行驱动程序,第一步就是提供一个仅供调试用的终端,该终端仅

供输出。通过 `register_console` 调用, 调试用终端可以在启动的任何时候调用, 最好使用 `console_init` 函数来调用, 该函数可以为其串口设置确定位的波特率。要做的仅仅是初始化终端, 并让它响应从它的唯一的入口点处开始执行后遇到的打印请求就可以了。内核中内置的 `printk` 例程提供了格式化输出的功能, 所以该函数调用要做的所有事情仅仅是将输入到端口的字符流排好队。

2. 串口驱动

用串口驱动程序来处理一个 `tty` 驱动程序的输入和输出。由于用户的进程是同 `tty` 驱动程序相连的, 实际上并没有直接同串口驱动程序相连, 串口驱动程序只要考虑从 `tty` 设备来的调用即可。这些驱动程序严格地遵循了标准的 `serial.c` 驱动程序模型。

3. 文件系统块设备

也许启动一个系统最好的方法就是将块设备驱动程序(通过 `blkmem.c`)和 `Ramdisk` 驱动程序集成。`Ramdisk` 驱动程序用来分配空间并从某个块设备中读取其内容。数据可以按其本来的格式保存, 或者以压缩的方式保存。

`Ramdisk` 读取第一个块, 来决定数据应该以何种方式来保存。除了包含下面将要谈到的同建立库文件和应用程序(库文件和应用程序构成了系统的可执行程序)问题相关的内容外, 根文件系统还必须包含设备文件和挂载其他文件系统(例如, `/proc` 文件系统或 `NFS` 分区)的装载点。


4. 运行时库——Runtime Library

`Runtime Library` 提供了用户程序到内核例程间的一个接口。虽然很多人认为他们的系统是 `Linux` 操作系统(指运行的内核), 但是, 系统的特性是由 `C` 运行时库决定的, 而不是内核决定的。例如, `Runtime Library` 将一个对 `printf` 的调用转换成一系列上下文。先缓冲数据, 然后执行一个“写”的系统调用来将输出发送到文件描述符 1(`stdout`, 标准输出)中。系统调用库必须依据系统的功能而裁剪。`uClibc` 的程序包实现了一个最小化的库, 它拥有最常用的 `C` 库函数。也可以使用功能更为强大的 `Cygnus newlib` 库或 `GNU glibc`, 但是需要很多的修正。如果要让 `uClinux` 支持新体系的机器, 就需要建立程序的入口点、定义系统调用的 `API`、裁剪 `I/O` 例程、建立支持 `flat` 格式的内存的体系等。

5. 程序入口代码

虽然用户程序的基本功能始于 `main` 函数, 实际的入口点却是 `C` 库函数的一部分。入口代码初始化了静态数据, 例如错误代码号(`errno`)和环境指针(`environ`), 并将参数 `argc` 和 `argv` 传递给 `main`。典型的情况是, 依据调用的常规方法, 程序使用宏例程 `start_thread`(在 `proc/processor.h`)将参数和环境变量放置在寄存器或堆栈内。`execve` 系统调用使用了 `start_thread` 来设立一个虚假存储的上下文。因此, 程序的入口代码必须完成 `start_thread` 的功能。

例 11-6 程序入口点(注意对 environ 的处理,它由内核调用 sys_execve 传递)



```

@ #include <sysdep.h>
@ r0 = argc
@ r1 = argv
@ r2 = envp
@ s1 = data segment
.data
.align 2
.global environ,errno
environ: .long 0 @ allocate static space for environ pointer
errno: .long 0
@ allocate static space for errno
.text
.align 2
.global start, _start, _syscall _error
.type start, %function
.type _syscall _error, %function
start:
_start:
ldr r3, = _data _start @ adjust the data segment base pointer
sub s1,s1,r3
ldr r3, .L3 @ load the address of environ
str r2,[s1,r3] @ store it to the static data
bl main @ call the function main
ldr r0, =0
bl exit @ normal exit... return 0
.L3: .word environ @ address of label 'environ'
_syscall _error: @ handle errors during system calls

ldr r3, .L4 @ load the address of errno
rsb r2, r0, $0
str r2, [s1,r3]@ errno = -result
mvn r0, $0 @ return -1
mov pc, lr
.L4: .word errno @ address of label 'errno'

```

6. 系统调用

用户程序是通过系统调用来使用系统资源的。由于该过程包括了从用户上下文到系

统上下文的切换,所以系统调用代码是同处理器有相当紧密联系的。这个转换是通过软件中断或自陷实现的。

系统进程保存了用户上下文,获取调用参数,并执行所规定好的步骤。此时,用户进程被挂起,直到内核完成该传送并将线程恢复到前台。在样例的系统调用中,参数作为变量保存,执行线程由软件中断(SWI)停止。内核执行了一个上下文保存,并决定了代码(0x900004 表示一个 sys_write 调用)要执行哪个调用。当内核执行完了该调用后,它就恢复用户上下文,用户上下文通过查询在 r0 中返回的状态位来继续执行。例 11-7 就是 write 系统调用反汇编后的代码。

例 11-7 反汇编 write 系统调用

```
00002550 <write>;
2550: ef900004 swi 0x00900004
2554: e3700a01 cmn r0, #4096 ; 0x1000
2558: 2aff6b0 bcs 20 < \> _syscall_error < \> >
255c: c1a0f00e mov pc, lr
```

7. 基本 I/O 例程

由于 uClinux 支持的体系资源有限,而且当前并不支持共享库,所以库例程必须非常的高效。文件的 I/O,无论对于文件系统还是设备而言,在标准的桌面 Linux 系统上都是资源密集型的,它们使用了很多的缓冲区来实现其性能。

8. 堆栈分配

对于一个用户程序而言,动态内存分配是通过调用一个叫作 malloc 的函数来分配程序堆栈存储的。在一个虚拟内存的系统中,是通过使用 sbrk 函数来完成动态内存的分配。该函数扩展了程序的数据段,然后将该分配好的内存区分隔成许多的小包。由于虚拟内存也允许分段,所以很可能在堆栈的顶和底之间存在有巨大的间隙。在 flat 模式下,没有使用间隙来填充虚拟页。所以,堆栈的存储是使用的 mmap 调用来实现的。由于 uc-libc 的内存分配非常简单,所以处理页面的任务都交给了内核来完成。诸如 glibc 之类的更加复杂的库给用户提供了缓冲区的线程管理功能(在系统分页管中分配),但是仅仅当基于 sbrk 的上下文分配被基于 mmap 的进程所替换时才可使用,如图 11-1 所示。

9. 堆栈大小

从图 11-1 中还可以看到堆栈同静态数据是物理上相连的,必须分配给堆栈足够的空间,以保证不会覆盖掉静态数据和代码空间。在常规的 Linux 程序中,虚拟间隔使得堆栈可以无限制地增长。系统可以监视页面的使用情况,来防止堆栈的扩张超过它的界限。在 uClinux 下,实现这样的间隔将需要系统分配物理内存而不是虚拟内存。由于缺乏内存保护来确保堆栈不超过其区域限制,所以在程序联结时必须仔细地选择堆栈的大小。一旦其大小确定,GDB 就可以在运行程序中监控堆栈的使用,来确保所选值是充分的。

11.4 本章小结



本章以专题的方式论述了 uClinux 的移植方法。首先概述了 uClinux 的移植方法,然后学习了 uClinux 的交叉编译工具的组成。还讨论了 uClinux 内核裁剪的方法,还有实现初始化和任务调度、内存计数、中断的思想,并论述了针对不同体系的移植方法。

要移植 uClinux,必然牵涉到对目标的硬件支持问题。本章探讨了诸如串口驱动、文件系统、Runtime Library、系统调用等在移植时必须注意的问题,并讲述了对 flat 模式的内存支持方法。在深入了解理论后,希望读者能够自己动手,将 uClinux 使用到自己的应用平台上。

第 12 章 嵌入式 Linux 的存储设备

由于嵌入式系统一般是没有硬盘的,所以诸如 Compactflash、EPROM 等存储设备在嵌入式系统中就有相当重要的地位。在本章,首先学习如何将 Linux 裁剪入 Compactflash 和 EPROM 中。然后探讨如何在 Compactflash 和 EPROM 中启动 Linux 的方法。通过两个实例,探讨其实现的基本思想。

本章主要内容:

- 使用 Compactflash 紧缩闪存卡进行系统设计
- 使用 Compactflash 闪存卡进行系统设计和硬件
- 使用 Compactflash 闪存卡进行分区并格式化
- 使用 Compactflash 闪存卡进行系统启动
- 从 EPROM 中读取数据
- EPROM 盘的压缩方法
- 嵌入式 Linux 的网络存储设备

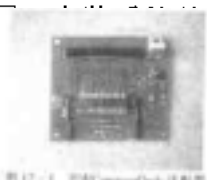


图 12-1 使用 CompactFlash 适配器

12.1 使用紧缩闪存卡进行系统设计

当设计嵌入式系统时,有很多存储设备供选择。本节通过一个实例来讲解如何在一个设计中用紧缩闪存卡替代硬盘作为嵌入式 Linux 系统的存储设备。

12.1.1 Compactflash 适配器

有一对 IDE 接口的 Compactflash 适配器,如图 12-1 所示。该电路板允许用户使用标准的 Compactflash 内存卡,以起到 PC 中标准的 IDE 设备的功能(如硬盘)。

如数码相机或者手持 MP3 播放机,便会看到一个 Compactflash。例如, Nikon 数码相机就使用了 Compactflash 以储藏图像信息。这些卡拥有 2MB 到 192MB 的存储空间,是一种可读写的存储器,当掉电时,其中的信息不会丢失。图 12-2 显示的就是一个 Compactflash,可以看到,它比硬币大不了多少。

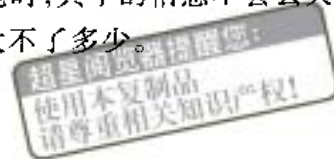


图 12-2 Compactflash 图示(左边的是一枚硬币)

当设计一个基于 Linux 的嵌入式系统时,第一个问题就是如何启动系统。一种可行的方法是购买一个 ISA 或者 PCI 卡,卡上带有 ROM,然后将代码放入到这些 ROM 中。当然,EPROM 也可以,具体方法在本章中也有所论述。还可以在嵌入式系统中使用一个软盘驱动器或硬盘驱动器。但应该明白的是,只要减少可移动部分的数目,系统的可靠性就会提高。这些驱动器还耗废了很多的电能,发热严重,所以使用它们会造成一些别的麻烦。

使用 Compactflash 适配器可以让用户从 Compactflash 上启动系统。这些卡并不贵,而且非常小。它们没有活动部件,所以对于嵌入式系统而言,它们是个很好的选择。

在本节,会讲述如何构建一个基于 Compactflash 的 Linux 系统。当启动系统时,同时也启动了一个 Web 服务器。

12.1.2 安装硬件

Compactflash 卡有一个 IDE 连接器、一个 Compactflash 连接器和一个电源连接器,还有一个跳线,使用跳线可以将该设备设置成主 IDE 设备或从 IDE 设备。

Compactflash 很容易安装到系统中间。可以使用常见的开发系统开发软件,并用一个目标系统来测试和运行该软件。将 Compactflash 适配器安装到开发平台的从 IDE 连接器上,它被设置为该控制器的主控设备,所以 Linux 将其视为 `/dev/hdc`。开发平台有一个硬盘驱动器 `/dev/hda`,带有 Mandrake Linux 7.2。当然,它上面可以运行其他任何 Linux 的发行版本。这个开发系统上还带有 IDE 接口的 CD-ROM,在主 IDE 控制器上视为设备 `/dev/hdb`。该系统使用了一个 32MB 的 Compactflash。当目标系统启动时,会使用 Compactflash,所以它在目标系统中应该是设备 `/dev/had`。它应该连接到主 IDE 控制器上,并设置为一个主设备。在该应用中,它是目标系统的唯一一个 IDE 设备。根据需要,也可以扩充一些其他的设备。

如果 BIOS 支持驱动参数的自动检测,那么它应该检测到 Compactflash。如果不支持,就需要手动输入各个参数,例如柱面、头、分段等。如果对这些不清楚,可以查找关于

该 Compactflash 的相关资料,以保证在当系统通电时不要插上或拔出 Compactflash。因为这会导致数据的崩溃,至少这会使系统紊乱。

本设计中,目标系统使用了一个 NE2000 兼容的以太网卡。应保证拥有目标系统的以太网卡驱动程序。如果以模块的方式安装了该驱动程序,还要修改/etc/init.d/rcS 文件。



12.1.3 安装软件

对于构建嵌入式系统而言,它不可能在小小的一张 Compactflash 上装下诸如 Red Hat 或 Mandrake 等标准的 Linux 发行版的所有内容。有些小的发行版本却是可用的,如 LEM、BlueCat 或 Hard Hat Linux。如何选择发行版本需要看用户系统的要求而定。

12.1.4 将 Compactflash 分区并格式化

使用 fdisk 命令,在 Compactflash 上建立两个分区。为此,键入命令:

```
fdisk /dev/hdc
```

将一个分区作为根目录,另外一个作为/var 目录。根目录以只读方式挂载,所以即使当某个程序错误运行时,任何一个文件都不会改变。/var 目录是可写的,所以任何参数或可变数据都可放置到/var 目录下。即使给每个分区都分 16MB,根分区也仅仅占 5MB,/var 分区占 3MB。

12.1.5 构建嵌入式内核

下面要做的就是构建一个内核,选用 2.2.18 的内核。虽然现在已经发行了 2.4 版本的内核,但是它的很多额外功能用户并不需要。如果目标系统只有很少的内存,挑选一个 2.0 版本的内核(假定它包含有目标系统需要的所有驱动程序)是个不错的选择。选一个 Pentium 作为目标系统,如果选用一台 386 或 486,必须确保所构建的内核可以在该处理器上运行。例如,如果使用一台 386,就要使能数学仿真功能,这是因为该处理器没有一个内置的数学协处理器。

在内核中只包含所需的驱动程序是个很好的做法。如果需要,可以使用模块,但当需要这些模块时,必须将其装入用户的代码中。绝大多数的 Linux 发行版本自带了诸如 depmod 等工具,使用这些工具可以将所有需要的模块都装载到内核中间去。如果要使用这些工具,需确认它们是否已经安装到了嵌入式系统中。

能支持 Compactflash 的唯一的驱动程序是 IDE 磁盘驱动程序。Linux 认为用户的 Compactflash 就是一个 IDE 硬盘驱动器,不需要别的驱动程序读取该设备。

12.1.6 构建 root 文件系统

构建好内核后,就需要建立一个根文件系统。下面列出了使用的所有软件包:

linux-2.2.18.tar.gz -- the Linux kernel
 busybox-0.48.tar.gz -- many Linux utilities
 net-tools-1.57.tar.gz -- network support applications
 lilo-21.6.1.tar.gz -- a boot loader for the kernel
 tftpd-2.20b.tar.gz -- a small web server
 ash-0.2 -- a small /bin/sh compatible shell

还从 Mandrake 系统中拷贝了下列库文件,它们都被上面所列出的包所使用。

ld-linux.so.2, libcrypt.so.1, libnss_files.so.2, libc.so.6, libnsl.so.1 and libpwdb.so.0.

有两个工具可帮助用户确定其嵌入式系统需要什么样的库。ldd 指令告诉用户哪个库同哪个可执行文件动态连接。如果试图在一个 2MB 的 Compactflash 卡上或一个软盘上运行系统,那么就要很小心地编译代码,确保它们尽可能的小。本例的设计中使用的是 32MB 的卡,空间还不算小。如果磁盘空间有问题,那么就要考虑更换一个库。对于在绝大多数标准的 Linux 发行版本中使用的标准 libc 库而言,有很多的库比它们要小得多。

另外一个工具是 strace 命令。在嵌入式系统中,软件 Webserver 应该是一个叫“no-body”的用户。Webserver 代码需要一些其他的库来读/etc/passwd 文件。由此,ldd 并没有告诉用户 Webserver 需要这些特别的库。strace 命令将在目标系统上运行,并告诉用户它试图打开的每个文件。这样,就可以发现所需要的额外库文件。

在这个嵌入式 Linux 的文件系统中,大概有 200 个文件。在本章中不可能一一介绍,仅对主要文件进行简要的阐述。

- /bin: 该目录包含了应用程序。
- /boot: 该目录包含了内核和 LILO bootloader 文件。
- /dev: 该目录包含了系统所需要的设备文件 MAKEDEV 命令可以帮助用户建立这些文件。这些文件要占用磁盘空间,所以在该目录下最好只放置所需要的文件。
- /etc: 该目录包含了系统所需要的设置文件。对于这个系统,需要 8 个设置文件。稍后将详细讲述这些文件。
- /lib: 共享库。如果在内核中使用了模块,那么这些模块就会被放置到/lib 的模块目录下。
- /mnt: 通常该目录为空。但如果想要挂载另外的文件系统时,可以使用该目录。
- /proc: 该目录为空。内核将系统的状态文件放置到该目录下。
- /sbin: 系统工具常常放置在该目录下。这些工具影响着整个系统。
- /tmp: 该目录是一个到/var/tmp 的软链接。/tmp 是一个可写的目录。当程序需要写一个临时文件时,它会在/tmp 目录中进行操作。根文件系统是只读的,所以程序不能写/tmp 目录。以读/写方式挂载/var 文件系统,就可以在/var 下建立一个临时目录。这与华恒公司的开发板是一样的。
- /usr: 该目录包含了用户的应用程序,如 Webserver 程序等。
- /var: 该目录通常包含了日志文件和其他的可写文件。在用户的系统中,/var 目录以可读/写方式挂载。在/var 目录中唯一需要的是一个临时文件夹,其中包含

了用户的网络设置文件。把它放在/var 目录下的原因在于这样可以改变网络设置。

在启动后,内核运行的第一个程序是/sbin/init。该程序初始化系统,并处理控制台终端和系统关机任务。init 程序从/etc/inittab 文件中获取其参数:

```
::sysinit:/etc/init.d/rcS
::askfirst:/bin/sh
::ctrlaltdel:/bin/umount -a -r
```

第一行告诉 init,立即运行 rcS 脚本程序。第二行告诉 init,当初始化完成后,在控制台上启动一个 shell 环境。第三行告诉 init,当系统关机时,卸载所有的文件系统。

例 12-1 列出了 rcS 脚本的全部内容。它包括了挂载所有的文件系统、启动网络和 Web 服务器等工作。在脚本中规定,要查找/var/etc/network 文件是否存在。如果存在,就指出该文件的来源。这就允许用户改变网络参数,如 IP 地址和网关地址等。如果该文件不存在,那么在脚本中就有一个默认的参数。使用 DHCP 也是个好主意。下面就是/var/etc/network 文件的内容:

```
NETWORK_IP=192.168.1.2
NETWORK_MASK=255.255.255.0
NETWORK_BROADCAST=192.168.1.255
NETWORK_GATEWAY=192.168.1.254
```

应修改 IP 地址以保持同网络一致。

例 12-1 挂载文件系统、网络和 Web 服务器的 rcS 脚本代码

```
#!/bin/ash
/bin/mount -a
# Do the networking stuff
echo "Configuring Network"
/sbin/insmod 8390
/sbin/insmod ne io=0x300
if [-f /var/etc/network ]
then
    . /var/etc/network
else
    NETWORK_IP=192.168.1.2
    NETWORK_MASK=255.255.255.0
    NETWORK_BROADCAST=192.168.1.255
    NETWORK_GATEWAY=192.168.1.254
fi
```

```

/sbin/ifconfig eth0 $ NETWORK_IP netmask $ NETWORK_MASK
/sbin/ifconfig lo 127.0.0.1 netmask 255.0.0.0
/sbin/route add -net 127.0.0.0 netmask 255.0.0.0 dev lo
/sbin/route add default gw $ NETWORK_GATEWAY

# Do the web server stuff
echo "Starting web server"

/sbin/thttpd -C /etc/thttpd.conf -c "/cgi-bin/*"

```

在系统上还需要一些别的设置文件。例如, mount 命令需要文件/etc/fstab 来确认各个文件系统挂载的位置。

```

proc /proc proc defaults 0 0
/dev/hda2 /var ext2 defaults 1 1

```

Webserver 需要/etc/thttpd.conf、/etc/nsswitch.conf、/etc/passwd 和 etc/group 这四个文件。当 Webserver 启动时,它就成为一个没有根用户权力的普通用户。nsswitch.conf 文件告诉了 Webserver 程序到何处去寻找用户和组的信息。用户信息在/etc/passwd 文件中,组信息在/etc/group 中,可以从系统中拷贝这两个文件。例 12-2 列出了 nsswitch.conf 的内容。它告诉了应用程序去文件中找寻用户信息和口令。

例 12-2 nsswitch 文件的一个推荐设置

```

#
# /etc/nsswitch.conf
#
passwd:      files nisplus nis
shadow:     files nisplus nis
group:      files nisplus nis
hosts:      files nisplus nis dns
bootparams: nisplus [NOTFOUND=return] files
ethers:     files
netmasks:  files
networks:   files
protocols:  files
rpc:        files
services:   files
netgroup:   nisplus
publickey:  nisplus
automount:  files nisplus
aliases:    files nisplus

```

12.1.7 设置 Webserver

在这个嵌入式应用中,选择了一个 tiny-turbo Webserver。该 Webserver 很小,只占用

了很少的内存。它同时也支持 CGI 应用程序,所以可以编写 Web 程序。这就允许用户编写一个 Web 应用程序作为同设备的用户界面。Webserver 安装在/usr/httpd 目录下,设置文件是/etc/httpd.conf,其内容为:

```
dir = /usr/httpd/html
chroot
cgipat = /cgi-bin/ *
```



12.1.8 安装 Boot Loader

将 flash 卡分好区,并建立文件系统和内核后,就可以安装一个 boot loader 了。当系统启动时,PC 硬件中的 BIOS 会装载第一个硬盘驱动器的第一个扇区。系统需要软件从硬盘驱动器的这个扇区中启动内核。LILO boot loader 就是实现这个功能的软件,它功能很多,但在这个应用中,只使用了最根本的功能。

在开发系统(注意,不是目标系统)中,假定将 Compactflash 挂载为/mnt/hda1,那么在 Compactflash 上启动 boot loader 的命令是:

```
/mnt/hda1/sbin/lilo -r /mnt/hda1 -C etc/lilo.conf
```

要小心执行该命令,否则很可能错误地将系统的 boot loader 替代掉,这会导致开发系统不能正确地启动。为此,最好有一张启动软盘,预防万一。

使用 LILO 设置,Compactflash 就可以启动了。

例 12-3 设置 LILO 启动 Compactflash

```
boot = /dev/hdc
disk = /dev/hdc
bios = 0x80
map = /boot/map
install = /boot/boot.b
vga = normal
default = linux
lba32
prompt
timeout = 10
menu-scheme = wb:bw:wb:bw
image = /boot/vmlinuz
    label = linux
    root = /dev/hda1
    read-only
```

12.1.9 测试系统

对于这个用 Compactflash 启动的系统,还要测试其性能。关掉宿主机,拿走 Com-

compactflash,并将该 Compactflash 安装到目标系统上,将目标机通电。为了测试,在目标系统中最好要有一个显示适配器。如果没有,可以设置内核,使其使用一个串行端口,把它当作控制台终端设备。如果目标系统有一个显示器,就会看到当内核启动时打印出来的内核信息。当内核启动完成后,会提示用户按 Enter 来启动一个 shell。

12.1.10 结论

到此为止,就由零开始建立了一个嵌入式的 Linux 操作系统。在这个过程中,学到了在嵌入式 Linux 的环境下,如何使用 Compactflash 作为系统的主要存储设备的方法。通过该实例,应领会到 Compactflash 卡在嵌入式 Linux 环境中的重要作用。

12.2 使用 EPROM 进行系统设计

本节学习如何使用 EPROM 设计基于嵌入式 Linux 的嵌入式系统。同样,也要通过一个实例来说明。

本节要介绍的应用实例是一个运行于显示器上的操作界面显示系统,它是由波音公司的飞行检测中心开发的。飞行环境中需要一种可以防止突然性断电的机制。为了满足这样的要求,可在无硬盘的系统上实现操作界面。

12.2.1 概况

需要解决的基本的问题包括:从一个可擦写的只读存储器(EPROM)的固态电子盘(SSD)引导 Linux;将 root 文件系统从 EPROM 拷贝至一个 RAM 盘;从客户机加载操作界面软件并执行。本文主要讨论系统工作机制的细节以及它所使用的开发技巧。

选用的硬件为一台基于 VME 的,拥有 16MB RAM、一个能够容纳 4M EPROM 的 PC104 SSD 和其他一些 PC104 板的 80486 单板机(SBC)。该单板机的 BIOS 支持使用 SSD。本系统还使用了一个可编程的键盘和一块标准的 VGA 显示卡。

12.2.2 系统操作

引导系统时,需要考虑以下两个选项:

- 在 DOS 下使用 loadlin(可以加在 Autoexec.bat 文件中)引导 Linux。
- 安装 LILO 直接引导 Linux。

第二种选项的好处是引导所用的时间较小。但本书采用第一种方案,因为可使用一个可编程的键盘——在 DOS 运行为键盘编程的软件。

为了使系统工作,需要对内核进行一点裁剪。ramdisk.c 代码被改成可以从任意块设备加载而不仅仅是从软盘(见例 12-4)。另外,还编写了一个新的块设备驱动程序,从 EPROM 设备读取数据,见例 12-5。



例 12-4 ramdisk.c(加载 ramdisk)

```

/* ramdisk.c 代码片段 */
/* 旧代码 - 如果从软盘启动,则仅加载 ramdisk */
    if (MAJOR(ROOT_DEV) != FLOPPY_MAJOR) {
        return 0;
        printk(KERN_NOTICE "VFS: Insert ramdisk floppy and press
ENTER \n");
        wait_for_keypress();
    }
... 从块设备加载 ramdisk 的代码段
/* 新代码 - 总尝试加载 ramdisk */
    if (MAJOR(ROOT_DEV) == FLOPPY_MAJOR) {
        /* 从硬盘加载或从 EPROM 加载 */
        printk(KERN_NOTICE "VFS: Insert ramdisk floppy and press
ENTER \n");
        wait_for_keypress();
    }
... 从块设备加载 ramdisk 的代码段

```

例 12-5 epromdisk.c(读取 EPROM 设备数据)

```

/*
 * linux/kernel/blk_drv/epromdisk.c
 *
 * from code by Theodore Ts'o, 12/2/91
 * Dave Bennett 11/95
 *
 */
#include <linux/sched.h>
#include <linux/minix_fs.h>
#include <linux/ext2_fs.h>
#include <linux/fs.h>
#include <linux/kernel.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <asm/system.h>
#include <asm/segment.h>

```



```
#include <asm/io.h>
#define MAJOR _NR EPROM _MAJOR

#include "blk.h"
#define EPROMDISK _MINOR 1
#define EPROMIMAGE _MINOR 2

static int ed_length;
static int sector_map;
static int sector_offset;
static int ed_blocksizes[2] = {0, 0};

int get_edisk(unsigned char *buf, int sect, int num_sect);
int get_image(unsigned char *buf, int ofs, int len);

#define EPROM_WINDOW 0x0D0000
#define EPROM_START 0x080000
#define EPROM_START2 0x100000
#define EPROM_SIZE 0x100000
#define EPAGE_SIZE 0x010000
#define CONTROL_REG1 0x0274
#define CONTROL_REG2 0x0674

static void do_ed_request(void)
{
    int len,
        ofs;
repeat:
    INIT_REQUEST;
    ofs = CURRENT->sector << 9;
    len = CURRENT->current_nr_sectors << 9;
    / if ( ! ( (MINOR(CURRENT->dev) == EPROMDISK _MINOR) ||
        (MINOR(CURRENT->dev) == EPROMIMAGE _MINOR) ) ||
        (ofs + len > ed_length) ) {
printk("EPROMDISK: minor = %d ofs = %d len = %d
ed_length = 0x%x \n", MINOR(CURRENT->dev), ofs, len, ed_length);
    end_request(0);
```



```

        goto repeat;
    }
    if (CURRENT->cmd == READ) {
        if (MINOR(CURRENT->dev) == EPROMDISK _ MINOR) {
            get _ edisk(CURRENT->buffer ,CURRENT->sector ,CURRENT->
current _ nr _ sectors);
        }
        if (MINOR(CURRENT->dev) == EPROMIMAGE _ MINOR) {
            get _ image(CURRENT->buffer,ofs,len);
        }
    } else {
        panic("EPROMDISK: unknown RAM disk command ! \n");
    }
    end _ request(1);
    goto repeat;
}

static struct file _ operations ed _ fops = {
    NULL,          /* lseek -默认值 */
    block _ read, /* read - 通用块设备读操作 */
    NULL,         /* write - 通用块设备写操作 */
    NULL,         /* rcaddr - bad */
    NULL,         /* select */
    NULL,         /* ioctl */
    NULL,         /* mmap */
    NULL,         /* 无特殊的 open 代码 */
    NULL,         /* 无特殊的释放代码 */
    NULL          /* fsync */
};
/*
 * 返回需要保存的内存大小
 */
long ed _ init(long mem _ start, int mem _ end)
{
    int i, ep;

```

```

short version, length, s_ofs;

if (register_blkdev(EPROM_MAJOR, "ed", &ed_ops)) {
    printk("EPROMDISK: Unable to get major %d. \n", EPROM_MA-
JOR);
    return 0;
}
blk_dev[EPROM_MAJOR].request_fn = DEVICE_REQUEST;
for(i=0;i<2;i++) ed_blocksizes[i] = 1024;
blksize_size[MAJOR_NR] = ed_blocksizes;

/* 搜寻有效的 eprom 盘 */
ep = EPROM_START;

get_image((unsigned char *)&version, ep, sizeof(version));

if (version != 2) { /* 没有找到 */
    ep = EPROM_START2;

    get_image((unsigned char *)&version, ep, sizeof(version));

    if (version != 2) { /* 未找到 */
        printk("EPROMDISK: Unable to find EPROM \n");
        return 0;
    }
}

get_image((unsigned char *)&length, ep + 2, sizeof(length));
get_image((unsigned char *)&s_ofs, ep + 4, sizeof(s_ofs));

if (length < 4) {
    printk("EPROMDISK: Length (%d) Too short. \n", length);
    return 0;
}

ed_length = length * 512;
sector_map = ep + 6;
sector_offset = ep + s_ofs;

printk("EPROMDISK: Version %d installed, %d bytes \n", (int)version,
ed_length);

return 0;

```

```

}

int get_edisk(unsigned char * buf, int sect, int num_sect)
{
    short    ss;    /* 段起始地址 */
    int      s;    /* 段偏移量 */
    for(s=0;s<num_sect;s++) {
        get_image((unsigned char *)&ss,sector_map + (s+sect) * sizeof(short), 2);
        get_image(buf + s * 512,sector_offset + (int)ss * 512,512);
    }
    return 0;
}

int get_image(unsigned char * buf, int ofs, int len)
{
    static int socket[4] = {0x00,0x01,0x04,0x05};
    int nb, bp, bofs, sock, page, offset, cr1, cr2;
    bp = ofs;
    bofs = 0;
    for(;len>0;) {
        sock = bp / EPROM_SIZE;
        page = (bp % EPROM_SIZE) / EPAGE_SIZE;
        offset = bp % EPAGE_SIZE;
        nb = (len + offset) > EPAGE_SIZE ? EPAGE_SIZE - (offset % EPAGE_SIZE) : len;
        cr1 = socket[sock] | ((page << 4) & 0x30) | 0x40;
        /* 当前并没有选择开发板 */
        cr2 = (page >> 2) & 0x03;
        outb((char)cr1,CONTROL_REG1);
        outb((char)cr2,CONTROL_REG2);

        memcpy(buf + bofs,(char *) (EPROM_WINDOW + offset),nb);

        len    - = nb;
        bp     + = nb;
        bofs   + = nb;
    }
    return 0;
}

```

实现 EPROM 设备驱动程序的第一种思想,是在 EPROM 中生成一个磁盘映像。这会为用户提供一个与 EPROM 相同大小的 RAM 盘,在这种情况下,RAM 盘大小为 3.5MB(SSD 的 DOS 分区占 1/2MB)。为了得到一个大的 RAM 盘,可使用一个压缩的磁盘映像。压缩的思想很简单,相同的扇区只存储一次。这样的好处是磁盘映像的空白区域不占用 EPROM 空间。所用的 SSD 的压缩方法如图 12-3 所示。

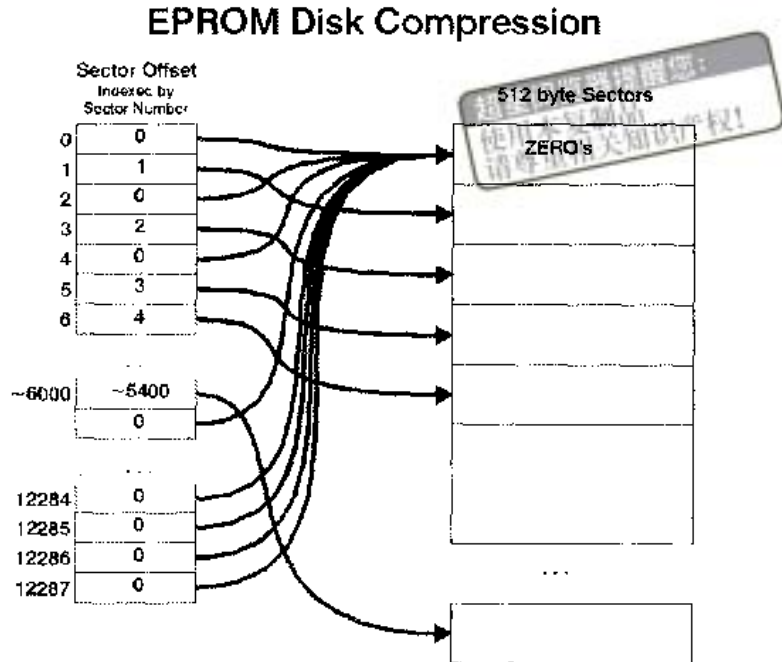


图 12-3 EPROM 盘压缩方法的图示

为了能自动运行操作界面程序,需要一个程序替代 `getty` 函数。这个程序(`dboot.c`, 见例 12-6)应该运行登录程序,可以为指定的虚拟终端设置 `stdin`、`stdout` 和 `stderr`。引导的过程如下:

- (1)加电后进行内存检测。
- (2)加载运行 `AUTOEXEC.BAT` 的 DOS。
- (3)运行键盘应用程序。
- (4)运行 `LOADLIN`——从 DOS 分区中读取 Linux 内核执行。
- (5)由 Linux 内核接管系统。
- (6)从 EPROM 中加载 RAM disk。
- (7)转换 `root` 文件系统到 RAM disk。
- (8)程序 `init` 读取文件 `inittab`,注意在 `inittab` 中设置执行 `dboot` 而不是 `getty`。
- (9)启动操作界面。

例 12-6 `dboot.c` —— 自动运行操作界面

```
/* dboot.c
```

例:

```
使用命令 c1:respawn:/usr/local/dboot tty1 login -f adams
```

将在控制台 1 启动用户 'adams'

```
*/
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
int main(int ac, char ** av)
{
    char cdev[50];
    char buf[50];
    int fd;
    pid_t pid;

    strcpy(cdev, "/dev/");
    strcat(cdev, av[1]); /* 控制台设备 */
    if (-1 == (fd = open(cdev, O_RDWR))) {
        perror("open");
        exit(0);
    }
    close(0);
    if (-1 == (dup(fd))) {
        perror("dup 0");
        exit(0);
    }
    close(1);
    if (-1 == (dup(fd))) {
        perror("dup 1");
        exit(0);
    }
    close(2);
    if (-1 == (dup(fd))) {
        perror("dup 2");
        exit(0);
    }

    if (-1 == execvp(av[2], &av[2])) {
```



```
perror("execvp");
exit(0);
```



12.2.3 开发过程

前面讲述了如何构建一个 EPROM 的驱动程序。下面学习如何组织 EPROM 的磁盘内容。可以使用如下分区的开发辅助盘来完成这项任务：

- /dev/hda1 - 80M Linux 系统；
- /dev/hda2 - 6M EPROM；
- /dev/hda3 - 20M DOS 分区；
- 使用 Lilo 引导。

直接对 EPROM 进行编程是很费时的事，所以本设计中使用辅助盘进行大部分开发。

开发磁盘映像必须确定开发系统需要些什么东西。首先应该完成一个最小的系统，然后再将操作界面所需的项目加到系统中。这个最小系统的完成是一个试验、纠错、再实验的循环过程，必须不断地试验。由于缺乏某个子程序或库文件而导致出现错误时，必须加入所需文件，直到系统能正常运行。

下一步是将 Linux 分区的内容拷贝到 6M 的 EPROM 分区中，然后在 DOS 下进行如下操作：

```
loadlin zimage root=/dev/hda2 ro
```

如果系统稳定，则将 6M 分区载入到 RAM disk 中。这与从 EPROM 中加载是相似的，但是由于没有对 EPROM 进行编程，所以系统的开发将加快。为了避免对 EPROM 编程而测试系统，在 DOS 下进行如下操作：

```
loadlin zimage root=/dev/hda2 ramdisk=6144 ro
```

因为对 ramdisk.c 进行了修改，所以 /dev/hda2 磁盘映像被加载到 RAM 中，root 文件系统转换为 RAM disk。用户需要不断地修改磁盘镜像，直到它能正常工作为止。

1. 对 EPROM 编程

对 EPROM 进行编程(即“烧片”)时，首先使用 tar 命令将小的盘镜像打包，然后将文件包解压缩到一个“干净”的文件系统。将文件系统放入一个干净的盘中，清除所有的未使用的分区，并执行盘压缩，见例 12-4。

为了打包盘镜像，先引导 Linux 的所有分区，然后加载 6M EPROM 分区。不要对 proc 文件系统执行 tar 命令。使用以下命令：

```
mount -t ext2 /dev/hda2 /mnt
```

```
cd /mnt
```

```
tar -cpf /tmp/eprom.tar *
```

为了生成(未经压缩的)磁盘镜像，使用另一台拥有 6M RAM 盘的计算机，执行如下的命令：



```
dd if=/dev/zero of=/dev/ram count=12288
mke2fs /dev/ram 6144
mount -t ext2 /dev/ram /mnt
cd /mnt
tar -xpf /eprom.tar
dd if=/dev/ram of=/eprom.dsk count=12288
```

生成一个 `eprom.dsk` 文件，它是 RAM 盘的分段镜像。而被编程并烧入 EPROM 的数据是压缩的镜像。烧写 EPROM 是使用 `med.c` 程序(见例 12-7)实现的，它读取磁盘镜像(即文件 `eprom.dsk`)、运行 RAM 盘的压缩程序，并输出一个二进制文件 `eprom.img`，该文件就是要被编程烧写入 EPROM 的文件。程序执行如下：

```
med ~/eprom.dsk ~/eprom.img
```

然后 EPROM 编程器将 `eprom.img` 烧入 EPROM 中。

例 12-7 从 ramdisk 镜像生成 eprom 盘镜像

```
/* med.c - 从 ramdisk 镜像中产生 eprom 盘镜像 */
```

```
#include <stdio.h>
#include <string.h>
#define DISK_SIZE (6291456)
#define NUM_SECT (DISK_SIZE/512)

void write_eprom_image(FILE *fi, FILE *fo);
int main(int ac, char **av)
{
    FILE *fi, *fo;
    char fin[44], fon[44];
    if (ac > 1) {
        strcpy(fin, av[1]);
    } else {
        strcpy(fin, "hda3.ram");
    }
    if (ac > 2) {
        strcpy(fon, av[2]);
    } else {
        strcpy(fon, "hda3.eprom");
    }

    fi = fopen(fin, "r");
    fo = fopen(fon, "w");
```

```
if (fi == 0 || fo == 0) {
    printf("Can't open files \n");
    exit(0);
}
write _ eprom _ image(fi, fo);
fclose(fi);
fclose(fo);
}

void write _ eprom _ image(FILE * fi, FILE * fo)
{
    char * ini;
    char * outi; /* 输入/出的镜像 */
    short * smap; /* 段映射 */
    char * sp;
    char c = 0;

    struct {
        unsigned short version;
        unsigned short blocks;
        unsigned short sect _ ofs;
    } hdr;
    int ns, s, i, fs;
    ini = (char *)malloc(DISK _ SIZE); /* 当前大小最大为 6M */
    outi = (char *)malloc(DISK _ SIZE); /* 当前大小最大为 6M */
    smap = (short *)malloc(NUM _ SECT * sizeof(short));

    if (ini == NULL || outi == NULL || smap == NULL) {
        printf("Can't allocate memory :(\n");
        exit(0);
    }

    if (DISK _ SIZE != fread(ini, 1, DISK _ SIZE, fi)) {
        printf("Can't read input file :(\n");
        exit(0);
    }
}
```

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

```

memcpy(outi,ini,512); /* 拷贝第一个段 */
smap[0] = 0;
ns = 1; /* outi 中的段数 */

for(i=1;i<NUM_SECT;i++) { /* 将每个段映射入 outi */
    sp = ini+i*512;
    for(s=0;s<ns;s++) { /* Found = sector */
        if(0 == memcmp(sp,outi+s*512,512)) {
            break;
        }
    }
    if(s==ns) {
        memcpy(outi+ns*512,sp,512); /* 拷贝新段 */
        ns += 1;
    }
    smap[i] = s;
}

/* Filler - 使段起始于 512 字节的边界处 */

fs = 512 - ((sizeof(hdr) + sizeof(short) * NUM_SECT) % 512);
hdr.version = 2;
hdr.blocks = NUM_SECT;
hdr.sect_ofs = sizeof(hdr) + sizeof(short) * NUM_SECT + fs;

fwrite(&hdr,sizeof(hdr),1,fo); /* 头信息 */
fwrite(smap,sizeof(short),NUM_SECT,fo); /* 段映射 */
for(i=0;i<fs;i++) {
    fwrite(&c,1,1,fo); /* Filler - 段起始于 */
} /* 512 字节的边界处 */
fwrite(outi,512,ns,fo); /* 输出段 */
}

```



2. DOS 的启动用固态电子盘 SSD

SBC 的 SSD 功能可以帮助用户构建磁盘镜像。DOS 的 SSD 盘最少需要如下的文件: DOS 引导文件、command.com、autoexec.bat、键盘加载程序、loadlin 与 zImage。

12.2.4 实验结果

由于直接在辅助盘上做的开发工作量很大,所以需要采取措施减少工作量。在本小节中,成功地实现了将 Linux 嵌入到 EPROM 中,将 EPROM 当做了系统的启动设备。

12.3 嵌入式 Linux 的网络存储设备

下面,来了解嵌入式 Linux 家庭中新兴的存储设备——嵌入式 Linux 的网络存储设备。

1. NetDrive 设备的特点

网络存储设备是一项用于网络内容存储的新兴存储技术,是互联网技术和万维网(Web)发展的产物。这项技术最早于 1997 年出现在美国,用于把存储设备直接连接到用户网络上,就像普通的文件服务器那样。

网络存储设备 NetDrive 实际上是一个瘦网络终端设备(也称网络家电—Network Appliance),其作用类似于一个专用文件服务器。这种设备的微内核进行过特别的优化,非常适合于处理来自网络的 I/O 请求。它不仅响应快,而且数据传输速率高。NetDrive 可以方便地接入到用户网络上,是一种即插即用的网络设备。为用户提供了易于安装、易于使用和管理、可靠性高和可扩展性好的网络存储解决方案。

NetDrive 设备的特点如下:

- 易于安装、使用和管理。

NetDrive 设备接入用户网络就能使用,无需安装任何服务器软件。初级用户无需配置就可以使用,而高级用户可以使用 NetDrive 内部已提供的工具,能方便地进行远程配置和管理。安装、卸装均无需关掉网络,可随意移动。

- 高性能优化。

NetDrive 设备具有一般文件服务器具有的所有功能,包括用户管理、安全管理、磁盘使用管理等。同时,NetDrive 的嵌入式软件经过了专门优化,可同时处理来自网络上多个用户及来自多种不同操作系统的请求,以实现最短的响应时间和提供最高的数据处理能力。普通的文件服务器软件都没有做到这些。

- 高可靠性和可扩充性。

NetDrive 设备的硬件和软件都是专用的,经过优化配置,可靠性高。此外,系统软件是固化的,可抗病毒侵袭。由于 NetDrive 设备是即插即用的,用户可以根据需要在网络中安装任意台。

- 跨平台和网络。

NetDrive 设备支持多种客户平台,包括 Windows、Windows 95/98、Windows NT、UNIX/Linux、Macintosh、Netware 等。还支持多种流行的网络协议,包括 Internet 上的许多协议。

- 价格低。

NetDrive 设备在提供给用户高性能的同时,更给用户省钱。NetDrive 的硬件是一个经过特别优化的单板机,具有最佳的性能价格比。另一方面,用户无需为不需要的功能付出,如显示器、键盘及其他的控制元器件。用户也无需购买和安装专用的文件服务器软件,如 Novell Netware、Window NT 等。

- 安全性好。

支持 C2 级安全标准。

2. NetDrive 设备的性价比高

NetDrive 设备具有很好的性能价格比。一个与通用的 PC 服务器具有同等性能的 NetDrive 设备,其价格只有 PC 服务器的 1/3 到 1/2。

3. NetDrive 设备的规格

其规格如下:

- 主板:优化的 PC 主板。
- CPU:Pentuum 级别。
- OS:自主开发的嵌入式 Linux 操作系统,支持实时多任务/多线程。
- 内存:32MB,可扩充至 128MB。
- 网络:内置 10M/100M 自适应以太网卡。
- 磁盘:内置高性能 IDE Ultra 66 控制器,可接 2 块 ATA 66 高速硬盘,可达 66MB/s 传输速率。
- 纠错:完全的 UDMA CRC 错误检查。
- 其他:可支持 APC 兼容的 UPS 设备。
- 网络协议支持:TCP/IP、IPX、NetBEUI、NFS 2.0 和 HTTP 1.1;支持 DHCP、BOOTP 和 RARP 实现自动获取 IP 地址。
- 网络类型支持:Microsoft NT 网络与 NFS (UNIX、Solaris、SCO UNIX、Red Hat Linux、HP-UX 和 PC-NFS)。
- 网络客户类型支持:Windows NT、Windows 95/98、Windows for Workgroups、NetWare、Macintosh 和 UNIX。
- 文件服务仿真:Windows NT 4.0 和 NFS 2.0。

4. NetDrive 与其他扩充存储容量的解决方案的比较

通常扩展网络存储容量的方式是在用户计算机或服务器上直接增加额外的磁盘,或者在用户网络上增加通用的文件服务器。

(1) 增加磁盘方式

要求专业人士安装,同时要安装磁盘的计算机或服务器时必须关机,否则会影响用户

或整个部门的工作。此外,新增加的磁盘是否能有效地工作,仍需专业人士进行配置和管理。通常增加一个新磁盘并使它能够有效地工作需要几个小时,而且新增的磁盘容量只能由特定人员使用。

(2) 增加通用文件服务器方式

增加一个新的通用文件服务器通常需要比较大的投资,包括购置服务器硬件、服务器软件、服务器安装以及服务器的维护。仅服务器软硬件的投资就是同等配置条件下的 NetDrive 设备投资的 2 至 3 倍。每一台通用文件服务器的安装通常需要专业人士花费一到两天以上的时间,后期运行维护也需要专业人员。

(3) 设备方式

使用 NetDrive 设备,无需专业人士安装和维护,普通用户只需花费十分钟就可以使用。更为重要的是,它无需维护,没有使用过程中的后顾之忧。购买 NetDrive 设备的投资仅为同等容量和性能的通用文件服务器投资的三分之一到二分之一。NetDrive 设备没有用户数量限制,而 NT、Netware 等通用服务器都有用户数的限制。

5. NetDrive 设备的几个应用示例

- 远程办公室服务器(Remote office server):易于安装、易于管理、使用方便、易于远程管理,从而易于实现远程文件共享。
- 软件派送(Software distribution):把要安装和更新的软件放在 NetDrive 设备中,每一台终端设备都可以共享、安装,而无需用 CD 逐台复制。
- 工作组文件共享:把大型文件、常用的文件存储在 NetDrive 设备中,可提高网络中应用服务器的性能,减小网络传输瓶颈。
- 图形库(Graphics library):可把大的图形文件、CAD 文件等存储在 NetDrive 设备中,以提高用户的访问速度,无需从存档中调取这些文件。

12.4 本章小结

由于嵌入式系统的特殊需求,有必要将 Linux 经过裁剪放到小型的存储设备中。在本章中,探讨了在嵌入式 Linux 系统中使用 Compactflash 和 EPROM 中的方法。

在第一个实例中,主要讲述了如何构建一个嵌入式 Linux 设备的步骤。其中,使用了 Compactflash 作为存储设备。学习了如何将 Compactflash 分区、如何设置系统、如何构建内核等关键问题。

通过第二个应用实例,学习了 EPROM 的压缩原理和方法。并用几个很有代表性的程序讲述了如何从 EPROM 加载 ramdisk、如何读取 EPROM、如何自动生成操作界面、如何从 ramdisk 镜像生成 eprom 盘镜像等重要方面。最后,还就如何对 EPROM 编程进行了阐述。

在第三小节,对当前流行的嵌入式 Linux 的网络存储设备(NetDrive)进行了简要的介绍。

通过对本章实例的学习,势必对如何在基于 Linux 的嵌入式系统中使用存储设备有一个清醒的认识。从而可以在设计中应用其设计思想,并作出自己的选择。

第 13 章 嵌入式 Linux 与 Java

Java 是当今最为有生命力的一种语言。由于它本来就是为嵌入式系统的设计所开发的,所以,同嵌入式 Linux 的结合也就顺理成章了。本章专门讨论了将嵌入式 Linux 和 Java 语言相结合这一问题,探讨了诸如 IDE、内存回收、代码重用、选择开发平台、通信接口等关键方面。通过本章的学习,相信读者能对在嵌入式 Linux 环境下使用 Java 开发应用程序的优势、方法有个较为全面的了解。本章讨论了嵌入式 Linux 系统中应用 Java 的方法。先来看看在嵌入式系统中使用 Java 的方法。然后,再看看如何在嵌入式 Linux 系统中使用 Java。

本章主要内容:

- Java 和嵌入式系统
- Java 的优势
- 使用 IDE
- 嵌入式 Linux 与 Java
- 选择 Java 的理由
- Java 和 Linux 的协作
- 通信机制

13.1 Java 和嵌入式系统

Java 程序语言在其产生之初,就是为机顶盒设备设计的。后来,由于它在互联网上的出色表现,使它赢得了巨大的声誉和财富。现在它又回到自己原来的领地——嵌入式系统。

不过,现在用 Java 来开发嵌入式系统,产生了许多复杂的新问题:由于新设备、新技术的出现,使其在速度、内存、大小和时间定义等方面面临着一些以前从没有遇到过的问题。当然,由于继承了 Internet 的完整性,再加上开发者现有的知识,Java 完全有能力解决这些困难。Java API 的一个子集 Java2 Micro Edition(J2ME),是专为只有很小内存的嵌入式设备设计的。在其客户端,只包括了 Java 的一些非常重要的功能,J2ME 用“配置文件”对两类设备进行设计:128KB~512KB 内存和 512KB 以上的内存的系统。基于不同的设备配置文件,将会有不同的用户接口和其他可用的程序包。

需要引起注意的是,那些在 Internet 和桌面程序上优秀的特性在嵌入式系统里往往

会导致一些错误。还有,由于运行在 JVM(Java 虚拟机)上和缺乏指针,所以 Java 要去控制硬件将非常困难。自动的碎片收集功能虽然使得编程变得很容易,但同时,也使得实时控制变得非常困难。此外,Java 运行非常慢并且还携带着一个很大的脚本。目前这其中大部分问题都可以得到解决。所以编制嵌入式系统程序时,有时 Java 不失为一个很好的选择。

1. 什么情况下选择 Java

对于要处理中断信号并指导火星登陆者这样的微处理器来说,Java 可能不是最好的选择。如果所要完成的任务是轻量级的,并且要求是高效的,有明确时间限制(如火箭制动、传感器通讯、宇宙飞船定位等),则应该用 C 或汇编语言。而对于在运输行业中使用的定位设备,应用 Java 将是一个非常好的选择。这种设备需要和其他的设备进行交流,或者需要继承很多 Internet 方面的功能。

2. Java 的优势

Java 流行的原因之一是其平台无关性(WORA)。因为它是一种解释型语言,同样的 Java 代码可以在 Mac、PC、Solaris,甚至是大型机上运行,但这同时也使其运行效率大大降低。虽然目前由于实时编译器的出现,使其耗费的时间大大缩短,但是用户仍然很难想像在通信时,人们要等 Java 分配内存、完成加载所需的类后再打电话(这一过程最少需要 8 秒时间)。AOT(ahead-of-time)编译器可以先解释 Java 代码,将其优化并转换成相应平台的二进制代码。但是使用 AOT,开发者就不能在一个中心服务器上管理并维护一个被编译过的代码。当然,这对于那些基于静态代码的使用环境来说(比如嵌入式系统),这应该不算一个很大的损失。

3. 大小至关重要

对于一个服务器来说,有几个 G 的内存应该不是新鲜事了,但对于一个可运行 Java 的设备的来说,其内存可能小于 512KB。所以在使用 Java 时最好使用核心类,去掉所有没用的代码、方法和类(在编译时 AOT 将优化这些问题)。有的商家有时也自己写核心 Java 类的专用平台。它们更简练、更高效,并且仍旧支持已公布的 Java API。

4. 使用 IDE

据统计表明,C 程序里 80% 的漏洞是由于指针引起的,所以 Java 自然就避免了这些错误的发生。Java 的安全模式禁止直接寻址系统内存和硬件。当然,想要直接、快捷地访问内存时,指针的确是一种很方便的方法。所以用 Java 写的嵌入式系统必须使用一个接口,通过该接口从 Java 里调用 C 代码,去实现一些对硬件的操作。这就意味着程序员必须具有处理多种语言代码的能力,也就是说一个程序员需要掌握一种额外的技能。对于一个公司来说,也就意味着需要额外的程序员和额外的投入。所以,现在的问题是如何把这些代码连接成一个模块,如果出错的话如何进行排错。

集成开发环境(IDE)是一个功能强大、丰富的多语言开发环境。它使程序员可以在同一个 IDE 和 IBM 的基于 J2ME 的 VisualAge 里用不同语言来编写程序。当然,由于

IDE 本身在解决问题时就非常复杂,所以排除其错误也不是一件容易的事。

5. 内存回收

在嵌入式系统里,通常不使用 Java 的理由是它实时性太差,也就是说它很难在预定时间内执行完相应的程序。此外,Java 使用自动垃圾回收技术来回收没有使用的内存,开发者对内存的回收几乎没有什么控制权。为了解决这个问题,商家通常使用不同的途径或算法来回收内存。NewMonic 公司的 RTE(Real Time Executive)和 Windriver 公司的 FastJ 都可以使编程者获得绝对的时间控制权。Sun 公司也有一种不同的解决方法,其目的仍然是保证其实时性。事实上,很多技术(比如目标重用)也可以减少这种内存回收方式带来的影响。

6. 重用性

虽然使用 AOT 编译器对于编程人员来说,付出的代价是其程序将不再放之四海而皆准,但由于 Java 代码便于携带,所以这应该不是什么大问题。当推出一款新型硬件时,除了接口可以保留外,与接口交互的代码将不得不更改。这将大大增加工作量,所以使用面向对象的开发语言非常重要。也就是说,使用 Java 使程序各模块的重用性大大增强。

7. 在项目里使用 Java

在什么时候使用 Java 来开发,需要进行成本分析。众所周知,内存和处理器速度都比较便宜,并且会变得越来越便宜。虽然从感觉上来说,用户可能在服务器的内存上花了上百甚至上千美金,但在生产电话时的成本就会减少。如果在每部电话上因为少使用内存而节约了 1 美元,那么 2 千万部电话就将节约 2000 万美元。

在嵌入式系统开发中使用 Java 技术应该说不是一个重要的问题。如果用户恰好有一家公司的,而公司里又有很多用 Java 从事 Web 开发的程序员,那么肯定不愿意花时间和财力去把他们培养成嵌入式系统(ES)程序员。所以一个很好的办法就是用 Java 来开发,当然,最好不要整个项目都用 Java 来开发。

应该引起足够注意的是,开发环境同普通的运行 J2ME 字节码环境的差别越大,所编的应用程序就越难于管理。所以除非有特殊的问题需要解决,一般不要这样做。不管怎么说,在嵌入式系统中使用 Java 开发是一个很有吸引力的选择。

13.2 嵌入式 Linux 和 Java

下面将探讨一下在嵌入式系统应用中使用 Linux 和 Java 的方法。首先将分别讨论它们各自的优点,然后来看看将其联合起来后会带来什么好处。在此还讲述一下在繁多的软件产品、开发平台和处理器、开发工具、执行引擎、优化工具中,Java 所处的独特地位。

1. 使用 Linux 来进行嵌入式系统开发

在现今快速发展的市场条件下,要开发嵌入式的应用产品,需要做许多的决策。这些

决策对于项目的成功有着重要的决定性意义,它们包括挑选处理器和平台、开发工具等。当然,还包括对诸如产品的特性、功能、生命周期等的考虑。一旦该产品已经开始处于研发阶段时,任何策略的改变都会导致金钱和时间上的浪费。所以,在项目的开始做好正确决定,可以最大程度地保证项目的成功。

嵌入式应用的设计目的决定了对于嵌入式应用如何挑选处理器和平台,这是不能怠慢的。幸运的是,有许多种类的处理器和开发平台供挑选,包括许多免费的选择方案。虽然挑选处理器非常的重要,但是,挑选平台常常更加重要。

一旦选定了开发平台,就必须选择一个有效的开发软件的集合。第一步要做的是选择一个合适和稳健的操作系统来支持该平台,它可以为项目提供支持和所需工具,并有足够的灵活性来支持任何特殊项目的要求,并为最终的项目结果提供一个良好的应用环境。然而,由于现有的处理器的种类相当多,这就使得挑选一个合适的 OS 来适应用户的处理器变得很困难。大多数 OS 供货商提供的解决方案并不能赶上处理器结构种类和开发板快速增长的脚步。

现在,那些开放源代码的工具已经或多或少地解决了该问题。但是,绝大多数的 OS 供应商仅仅将其视为一种当他们的 OS 不能满足需要时的候补方案,从而限制了它们功能的发挥。许多情况下,开发板是没有软件支持包的。可能的话,需要新建一个。在处理诸如对特别的 I/O 设备的支持和对下一代产品相兼容平台的支持等问题时,就会显得更加麻烦。当然,使用开发源代码的软件来构建用户自己的 OS 也是可行的,它也包括了一些商业实时操作系统(RTOS)的特性。对于项目的开发和管理人员而言,采用一套基于嵌入式 Linux 的软件是很有吸引力的。一旦经过适当的改造, Linux 就可以为项目研发人员提供一个构建嵌入式解决方案的更加优秀的选择。

这些年中, Linux 开发环境变得非常流行,使得 Linux 下的开发活动变得相当活跃。 Linux 的不断成长,对新型设备、技术、协议和服务提供的支持也越来越好。将 Linux 集成到不同平台上的工作主要是完成嵌入式 Linux 系统的移植。这表明要完全支持开发板的特性和功能,开发驱动程序和系统集成是很重要的。虽然绝大多数的嵌入式 Linux 供货商是遵循开发源代码规范的,但是,很少是完全将其源代码公开的。 Hard Hat Linux (在第 9 章曾经提到)就是个例子。嵌入式 Linux 继承了桌面和服务器的 Linux 的许多基本特性,例如可靠性、开放性和良好的性能。 MontaVista 公司增强了 Linux 的很多性能,使得它更加适合嵌入式系统的开发。虽然是专门由 MontaVista 开发的,但是这些改进部分可以包括在相关的开发源代码系统的代码树中,从而也促进了 Linux 的发展。其特性包括:

- 无需控制台即可启动系统、执行应用程序;
- 闪存启动,无盘操作;
- 可裁剪的 Linux 内核不过 500K;
- CompactPCI 系统和 I/O 板支持;
- CompactPCI back-plane 网络功能;
- 支持定制设计和支持压缩闪存启动功能;
- 支持 PCIMG 基本热交换。

Hard Hat Linux 支持许多处理器/开发板的组合,并支持很多处理器体系,包括: Intel

Pentium、Pentium II/III 及兼容产品;386/486 及兼容产品;StrongARM 110、1100、1110 及 Xscale;Motorola PowerPC 603/604、740/750/7400、PowerQUICC 823/860/8260;IBM PowerPC 405;MIPS 4K、5K 和 7K;Super Hitachi (SH) 3 和 4。

当然,为项目挑选软件是比 OS 更重要的。在今天充满竞争的环境下,研发人员需要性能优秀的开发工具,来增加产量,支持跨开发队伍的联合开发。许多应用要求拥有用 C、C++ 或汇编语言写成的具有低级硬件访问功能的函数。如果要优化程序结果,就必须加强对目标的设置、测试、调试等功能。用户可以获得相应的工具来支持把应用程序移植到嵌入式 Linux 系统上。

另外一个必须考虑的问题是中间环境,它可以简化并加速新应用软件的开发。但同时也可以继承原有软件代码,或特定的高效模块(例如对声音和图像的控制模块、图形用户接口模块等等)。鉴于此,嵌入式 Linux 和 Java 有着相互支持的地方。

2. 选择 Java 的理由

Java 较 C 或者汇编语言而言,在嵌入式系统中有着很明显的优势。其最明显的优点在于它容易开发和维护、可重用代码、原有代码和 Java 代码容易集成等。

(1) 易于开发和维护

在整个开发周期中,Java 环境都使开发和维护变得容易。如果目标系统是基于虚拟机的话,则代码就更加容易执行、调试、分析、热替换和维护。同以前的各种嵌入式系统相比较,相连的设备可以复杂得多。在项目的生命周期中,手动的升级系统再也不会是有效的了。相反,设备的连通性提供了对各组成部分进行远程管理的能力,允许开发团队增加产品的特色,解决各种问题,并且在产品安装之后对设备上的软件进行维护和升级。

Java 使真正的交互性开发成为可能。开发者们可以通过网络交换信息,建立起一个能供公司内或团队内成员访问的虚拟实验室,来共享这些进展。

运行 Java 程序的环境还减少了与内存管理相关的问题,它允许使用自动的“垃圾清理”技术来对释放出的内存碎片进行清除。

(2) 易于代码的重用

由于不同的嵌入式系统,在软件和硬件上的要求各不相同,其软件开发者使用的开发方法也就相当原始。有时候,每做一个新的项目都要从头开始。现在,随着嵌入式技术的成熟以及系统自身日趋庞大和复杂,人们对于可重用部分,或是由一个产品到另一个产品的充分应用越来越感兴趣。这种重新使用使开发中的一次投资有可能通过多个项目获得扩充性的回报。

Java 环境能够让一个部分的所有者将其改编并应用到许多项目和平台上,并且通常价格较低。即使顾客们要求的目标不同,或是技术需要引进了新的硬件(CPU、其他设备)和软件(包括不同版本的 Linux)。

(3) 易于 Java 代码与原始代码的结合

Java 应用程序、虚拟机器和基本硬件之间的接口为嵌入式系统提供了最好的兼容特性。原始代码虽然用起来并不方便,却是解决许多功能和设备界面问题的最佳途径。

在 C、C++ 或者汇编语言中加入标准的通信方法、新用户接口和安全措施,将会非常昂贵和耗时。Java 基本库除了提供了这些要素外,还提供了很多别的优点,从而可以加快

开发的速度。图 13-1 显示了 Java 应用程序和其他语言的应用程序的关系以及整个应用系统的构成。

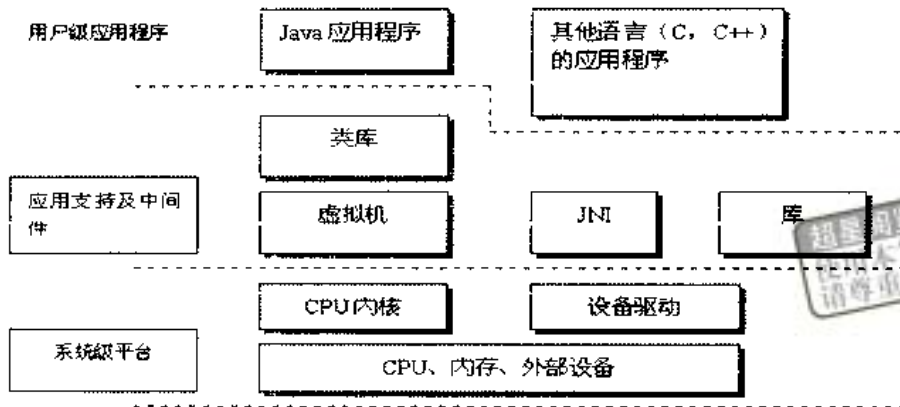


图 13-1 各个部分的联系

(4) 易于掌握开发技巧

现在许多开发者们正在接受关于 Java 应用软件开发方面的教育。例如,IBM 就有着数以几千计的编写 Java 软件的程序员。随着各个学校纷纷开设对这种流行的语言和管理环境的教育,Java 开发者的数目也正在增加。

下一代嵌入式项目的开发者需要有能力来有效地处理相关的网络设备的复杂性。由于 Linux 和 Java 都执行对 TCP/IP 通信组件堆栈的访问,所以就省却了很多为支持通信协议而要做的工作,并且是通过一种广泛认可的标准方式完成的。

(5) 易于中间设备的选择

中间设备有很多选择,其中大部分是作为个体组件使用的。例如协议堆栈、图形开发工具和管理工具,以及专用 OS 设备如高效实时扩展设备。使用它们中的任何一种都不算太难,但是,随着总体数目和多样性的增加,堆栈内增大的复杂性不可避免地会影响开发者的效率,并延迟软件的集成、检测和调试的过程。

Java 最初是设计用于解决桌上型电脑和服务器的市场上问题的。但是,由于可以提供嵌入式软件开发环境,它已经开始得到愈加广泛的接受。Java 技术的提供者已经完成了对一些标准工具的集成,其中包括通信、安全和组件管理。Java 软件设计所固有的面向对象的特性,使用户能够以一种有效的的方式来处理完全不同的中间设备。其他中间设备环境也能实现对这些基本程序的细节提取,但它们都不像现在的 Java 这样拥有广泛的产业支持。

最后,可以将 Linux 延伸,开发出实时性的程序。有许多途径可以实现这种控制。MontaVista 公司提供了对标准 Linux 平台特征的延伸,而不是修正它们。例如,为了支持实时程序的执行,Hard Hat Linux 使用了新的 API 来控制内存管理、安排进程和程序的运行。

(6) 易于高效地执行程序

由于建立了标准的 Java 系统程序库,使得控制程序的大小完全成为可能。实际上,研究已经表明,如果以 Java 字节代码进行配置的话,一个占用 500 多 KB 的程序可以变得

更加紧凑。这是因为字节代码比许多硬件设备的指令设置要更加紧凑。

IBM 使用了一个独特的方法来创建它的嵌入式 Java 程序,实际上就是从同一个源代码创建所有的 J9 目标版本和所有的 Java 系统程序库结构。IBM 对于 J9 和 Java 系统程序库的源代码描述了如何构造组件的形式。一个单独的端口层被用来描述目标处理器和 ROTS 界面的独特性。通过使用专门的技术编写出这些组件的单独端口,然后使用嵌入式 Linux ROTS 销售者提供的工具对它们进行编译。内存的大小和应用复杂度的关系如图 13-2 所示。

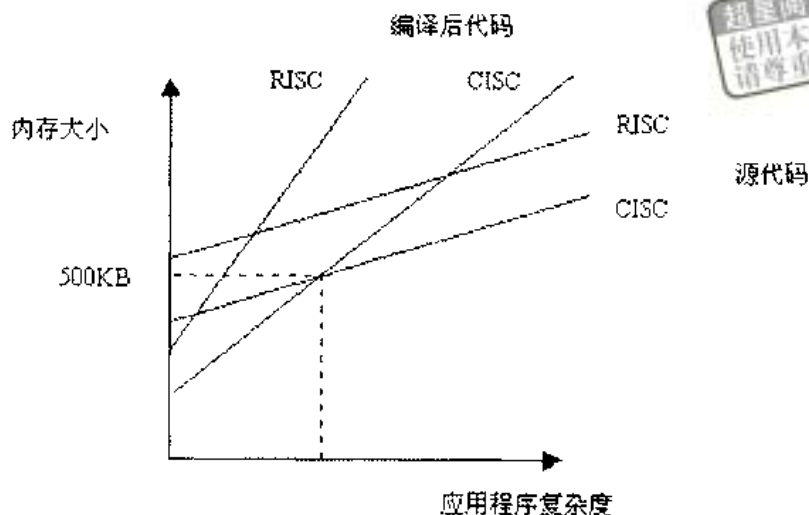


图 13-2 内存大小和应用复杂度的关系(横轴为应用的复杂度,纵轴为内存空间的大小)

3. Java 和 Linux 的协作

一个可正确执行的嵌入式 ROTS 必定能形成嵌入式 Linux/Java 平台的内核。嵌入式设备在开放系统、设备驱动器和其他基本组件上设置了一些限制和约束。研究那些使用了嵌入式目标平台执行程序的 Linux 版本是非常重要的,并不是每一种版本的 Linux 都可以这样在嵌入式目标平台上执行程序。这里必须特别注意如 RAM、ROM 和 Flash 这样并不丰富的嵌入式设备资源。在缺乏处理器特征的地方(如浮点数学加速器),必须提供有效而无漏洞的程序来运行的硬件。

嵌入式系统必须适应目标设备上独特的可用资源,因此,能够快速有效地重新构建 Linux 操作系统模式是很重要的。它允许开发者和工程师们根据虚拟机器和 Java 程序的需要来增加、删除和改装 Linux 的特色。在嵌入式系统的设计中,对于设备和应用程序的每个方面都应考虑,权衡它们的大小、速度和资源限制。

同标准的个人电脑或面向企业的服务器相比,嵌入式平台有着极大的不同。用户尤其应注意能够访问嵌入式目标的通路。对通信端口进行访问的一致路径、闪存的序列、显示设备和声音界面都必须被激活。由于嵌入式 RTOS 开发者们对于 Linux 部署已经拥有广泛的经验,许多面向嵌入式标准已经取得进展,能够用于这些设备。其标准包括:

(1) 显示设备管理

嵌入式 LinuxOS 和 Java 技术的结合使得构建图形用户界面的效率达到了一个新的

水平,这种图形用户界面一般都是基于装备有触摸屏的 LCD 显示设备的。这些显示设备低能耗、功能紧凑,而且通常是彩色显示,对消费者和产品设计者都很有吸引力。嵌入式平台一般都采用了图形用户界面,该界面可以是基于位图的图标框结构或基于窗口控制的框架结构。和基于 Linux 的服务器或工作站不同(它们使用基于服务器/客户模式的 X Window 图形用户界面的框架,或以其他方式来管理窗口),嵌入式 Linux 提供了一种有效和直接的方法进行管理。例如, MontaVista 公司的 Hard Hat Linux 就将 MicroWindows 移植到了许多处理器/设备平台上。MicroWindows 直接运行在 Linux 的帧缓冲(frame buffer)的顶端,提供了一个低端的硬件接口和底层硬件打交道。其设计非常的高效,可生成具有良好响应能力的接口。

MontaVista 公司就使用了 IBM 的 J9 和 Java 类库来实现 MicroWindows。因此,很多嵌入式设备都拥有统一的图形支持。IBM 提供了两个基本的图形类库:简单窗口工具集(SWT)和“MicroView”。SWT 的功能是通过复杂的窗口控制来建立分层的用户界面。而 MicroView 则是用来在简单的嵌入式设备上建立比较粗糙的基于图标的用户界面的。

(2) 通信接口

几乎每个开发平台和嵌入式计算机都包括了通信接口。Linux 移植的基本责任就是为 RS-232 串行口和以太网口提供驱动程序。

通信的实现有两种方式:基于 Linux 提供的 TCP/IP 通信协议栈的实现方式,或对简单串行口、总线接口简单控制的实现方式。TCP/IP 协议栈提供了访问因特网的能力和 socket 接口,实现程序到程序的段通信。

直接连接到串行设备上的特定的通信总线和设备可以直接通过 Java 的扩展类库来管理。这些设备包括了可同自动总线(如 CAN、MOST、IEEE J-1850 等)相连接的设备。

在开发中,通过串行连接可以直接地访问某些设备。例如蜂窝电话、汽车电台和 GPS 单元等。这些设备通过在串行连接上传送特定的协议来控制。在生产中,这些设备实际是可以连接到一个自动通信总线上。由此,为了控制设备,就需要一个分层的体系结构。

4. 灵活性

Java 技术和嵌入式 Linux 操作系统的结合已经在许多的机器上得以实现。开发人员和工程师可以依据项目的需求设置并裁剪虚拟机、Java 类库成员和嵌入式操作系统。如果需要,可以加入设备驱动程序。同时,针对许多相关的设备的应用程序的开发和配置可以保持统一的方法。该方法允许重用改良后的程序代码。无论对于最小还是最大的嵌入式设备而言, Linux 操作系统和 Java 技术的结合都是很有可裁剪性的。

5. 结论

嵌入式 Linux 和 Java 的结合为工程师和开发人员提供了一个新的选择。由于嵌入式系统开发人员的努力,现在已经有很多设备平台支持 Java 应用了。

在服务器上运行 Linux 的历史已经很长了,这使 Linux 的内核和相关应用程序的稳定性相当好。嵌入式 Linux 的开发也享用了这些优点。今天,随着针对嵌入式 Linux 的 Java 应用程序的增长(例如 IBM 公司的 VisualAge Micro Edition),在嵌入式 Linux 开发平

台上所开发的 Java 应用程序,已经完全得到了种类丰富的交叉开发工具的支持。

在嵌入式 Linux 的开发领域中,当前的潮流是倾向于设计有更好的可设置性的、更小的操作系统组件,这样就可以加快设备初始化的速度和提高资源的利用率。当嵌入式 Linux 同 Java 应用环境的强大功能和灵活性结合起来时,就可以使开发人员设计出下一代具有高度可靠性的嵌入式系统。

使用本文档知识
请尊重相关知识版权

13.3 本章小结

本章着重介绍了嵌入式 Linux 技术同 Java 结合的优势和实现方法。首先,介绍了在嵌入式系统中使用 Java 的优势。然后,针对嵌入式 Linux 系统,专门探讨了 Java 同 Linux 的原代码的结合、代码的重用、选择硬件平台等方面的问题。最后,探讨了 Java 对嵌入式设备的硬件的支持问题和通信标准。

通过对本章的学习,读者可以对在嵌入式 Linux 平台下使用 Java 有个定性的了解,为以后的学习打下基础。

结束语

——展望嵌入式 Linux 在未来的发展

到此,本书也该结束了,但是嵌入式 Linux 技术——应该说是一门艺术,正在飞速发展着,向世人展现着它的生命力和魅力。本书向读者介绍的,应该还只是这门艺术的一个概况,一个全貌。其间无穷的精妙之处,也是难以在这方寸间可以展现得了的。

新生事物的出现给人们带来的惊喜,很大部分是源自它带给人们的美好希望。那么,我们就来展望一下嵌入式 Linux 在未来的发展,权做本书的结束语吧!

从本书可以看到,嵌入式 Linux 技术已经有了很大的进步。但是,如果基于 Linux 的嵌入式技术要想走向成熟,还需要围绕下面三个方面作进一步的发展。

1. 实时系统的可扩展性

将 Linux 开发改造成为实时系统,最初可能是出于一些技术工程师的爱好而兴起的。但是 Linux 作为一种通用操作系统,它本身的发展不可能考虑基于它的实时系统及其兼容性问题。那么,如何让这种实时系统能够具有向上和向外的扩展性显得尤为重要。所谓的向外扩展,就是让实时性的支持面向范围更为广泛,支持的设备更多。目前开发面向的设备限于较简单的有实时需求的串/并口数据采集、浮点数据计算等等,而实时网络等实时系统的高级应用还需要进一步发展。向上扩展是这种通过 Linux 内核改造成的实时系统方式尤其需要注意的地方。在 Linux 内核中作出一些改动可以获得系统对实时任务运行的支持,但是在 Linux 内核有比较大的变动时,实时系统的升级就很难做到同步了。比如说,Fsmlabs 公司的 RT-Linux 在 Linux 从 2.0 版本升级到 2.2 版本之后,花了半年多时间才将新版本的 RT-Linux V2.2 系统推出。

2. 面向嵌入目的的 Linux 内核改造

Linux 的内核体系采用的是 Monolithic 体系结构,和 MicroKernel 结构相比,前者更适合于实时系统,而后者更适合于嵌入式系统。因为在 Monolithic 的体系结构中,内核的所有部分都集中在一起,可以有效地获得系统资源。系统的各部分可以直接互相沟通,紧急任务的切换时间短,系统的响应速度也快。但是在系统比较大的时候,维护起来比较困难。而且所有的部件在一起编译连接,体积也比较大,这样不太满足嵌入式系统资源有限的要求。对于 MicroKernel 的体系结构,在内核中只包括了一些基本的内核功能如创建和删除任务、任务调度、内存管理和中断处理等部分。而文件系统、网络协议栈等部分都是在用户内存空间运行,这样可以减小内核的体积,同时方便整个系统的升级、维护和移植,但是执行效率就不如 Monolithic 类型的内核的系统了。

使用 Linux 内核,在实时性和嵌入需求中间采取一种折中的方案就是将 Monolithic 的内核改造成部分 MicroKernel 体系结构的内核。如何改动,是需要解决的一个技术难点。

3. 集成开发环境

提供完整的集成开发环境是广大嵌入式系统开发人员的最大需求。人们知道,一个完整的嵌入式系统的集成开发环境需要提供如下工具。

(1) 编译/连接器

Linux 实际上应该称为 GNU/Linux,这是因为在 Linux 下的开发几乎都是基于 GNU 提供的一系列工具,尤其是使用了 GNU 的强大编译器工具链——gcc 工具链;基于 GNU 工具链的开发和改造,尤其是面向嵌入式体系结构,如微控制器、SuperH 处理器、Strong-ARM 处理器等等,是工具链方面的巨大需求。

(2) 内核调试/跟踪器

一般开发嵌入式操作系统的程序调试和跟踪都是使用仿真器(ICE)来实现的,但是使用 Linux 系统做原型时,可以绕过这个障碍,直接使用内核调试器来做操作系统的内核调试和查错。一般使用的方式是基于 GNU 的调试器 gdb 的远程调试功能,由一台客户机运行调试程序,来调试在宿主机运行的操作系统内核。其中,使用远程开发时还可以使用交叉平台的方式,如在 Windows 平台下的调试跟踪器对 Linux 的宿主系统做调试。

(3) 集成图形界面开发平台

编辑器、调试器、软件仿真器、监视器都应该提供良好的界面,提高开发效率。同时,还需要提供基于图形界面的特定系统定制平台,提供良好的扩展性。

国外的开发如火如荼,国内的开发也不甘示弱。863 重点支持项目中就有使用 Linux 作为嵌入式系统开发的方向。嵌入式 Linux 具有强大的生命力和利用价值,很多公司和大学都不同程度地表现出对这个方面的兴趣。相信嵌入式 Linux 的发展将带领人们进入嵌入式系统的新时代!

附录 A GNU GPL —— GNU 通用公共许可证

1991.6 第二版

版权所有 (C) 1989, 1991 Free Software foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA 允许每个人复制和发布这一许可证原始文档的副本, 但绝对不允许对它进行任何修改。

1. 序言

大多数软件许可证决意剥夺你的共享和修改软件的自由。对比之下, GNU 通用公共许可证力图保证你的共享和修改自由软件的自由——保证自由软件对所有用户是自由的。GPL 适用于大多数自由软件基金会的软件, 以及由使用这些软件而承担义务的作者所开发的软件(自由软件基金会的其他一些软件受 GNU 库通用许可证的保护)。你也可以将它用到你的程序中。

当我们谈到自由软件(free software)时, 我们指的是自由而不是价格。我们的 GNU 通用公共许可证决意保证你有发布自由软件的自由(如果你愿意, 你可以对此项服务收取一定的费用); 保证你能收到源程序或者在你需要时能得到它; 保证你能修改软件或将它的一部分用于新的自由软件; 而且还保证你知道你能做这些事情。为了保护你的权利, 我们需要作出规定: 禁止任何人不承认你的权利, 或者要求你放弃这些权利。如果你修改了自由软件或者发布了软件的副本, 这些规定就转化为你的责任。例如, 如果你发布这样一个程序的副本, 不管是收费的还是免费的, 你必须将你具有的一切权利给予你的接受者; 你必须保证他们能收到或得到源程序; 并且将这些条款给他们看, 使他们知道他们有这样的权利。

我们采取两项措施来保护你的权利。

(1) 给软件以版权保护。

(2) 给你提供许可证、它给你复制、发布和修改这些软件的法律许可。同样, 为了保护每个作者和我们自己, 我们需要清楚地让每个人明白, 自由软件没有担保(no warranty)。如果由于其他某个人修改了软件, 并继续加以传播, 我们需要它的接受者明白: 他们所得到的并不是原来的自由软件。由其他人引入的任何问题, 不应损害原作者的声誉。

最后, 任何自由软件不断受到软件专利的威胁。我们希望避免这样的风险: 自由软件的再发布者以个人名义获得专利许可证, 事实上, 将软件变为私有。为防止这一点, 我们必须明确: 任何专利必须以允许每个人自由使用为前提, 否则就不准许有专利。

2. 有关复制、发布和修改的条款和条件

0. 此许可证适用于任何包含版权所有声明的程序和其他作品, 版权所有者在声明中明确说明程序和作品可以在 GPL 条款的约束下发布。下面提到的“程序”指的是任何这样的程序或作品。而“基于程序的作品”指的是程序或者任何受版权法约束的衍生作品。也就是说包含程序或程序的一部分的作品, 可以是原封不动的, 或经过修改的和/

或翻译成其他语言的(程序)。在下文中,翻译包含在修改的条款中,每个许可证接受人用你来称呼。

许可证条款不适用于复制、发布和修改以外的活动。这些活动超出这些条款的范围。运行程序的活动不受条款的限制。仅当程序的输出构成基于程序作品的内容时,这一条款才适用(如果只运行程序就无关)。是否普遍适用取决于程序具体用来做什么。

1. 只要你在每一副本上明显和恰当地发表版权声明和不承担担保的声明,保持此许可证的声明和没有担保的声明完整无损,并和程序一起给每个其他的程序接受者一份许可证的副本,你就可以用任何媒体复制和发布你收到的原始的程序的源代码。

你可以为转让副本的实际行动收取一定费用。你也有权选择提供担保以换取一定费用。

2. 你可以修改程序的一个或几个副本或程序的任何部分,以此形成基于程序的作品。只要你同时满足下面的所有条件,你就可以按前面第一款的要求复制和发布这一经过修改的程序或作品。

a) 你必须在修改的文件中附有明确的说明:你修改了这一文件及具体的修改日期。

b) 你必须使你发布或出版的作品(它包含程序的全部或一部分,或包含由程序的全部或部分衍生的作品)允许第三方作为整体按许可证条款免费使用。

c) 如果修改的程序在运行时以交互方式读取命令,你必须使它在开始进入常规的交互使用方式时打印或显示声明:包括适当的版权声明和没有担保的声明(或者你提供担保的声明)。用户可以按此许可证条款重新发布程序的说明,并告诉用户如何看到这一许可证的副本(例外的情况:如果原始程序以交互方式工作,它并不打印这样的声明,你的基于程序的作品也就不需要打印声明)。

这些要求适用于修改了的作品的整体。如果能够确定作品的一部分并非程序的衍生产品,可以合理地认为这部分是独立的,是不同的作品。当你将它作为独立作品发布时,它不受此许可证和它的条款的约束。但是当你将这部分作为基于程序的作品的一部分发布时,作为整体它将受到许可证条款约束。准许其他许可证持有人的使用范围扩大到整个产品。也就是每个部分,不管它是谁写的。

因此,本条款的意图不在于索取权利,或剥夺全部由你写成的作品的权利。而是履行权利来控制基于程序的集体作品或衍生作品的发布。

此外,将与程序无关的作品和该程序或基于程序的作品一起放在存储体或发布媒体的同一卷上,并不导致将其他作品置于此许可证的约束范围之内。

3. 你可以以目标码或可执行形式复制或发布程序(或符合第2款的基于程序的作品),只要你遵守前面的第1,2款,并同时满足下列3条中的1条。

a) 在通常用作软件交换的媒体上,和目标码一起附有机可读的完整的源码。这些源码的发布应符合上面第1,2款的要求。或者

b) 在通常用作软件交换的媒体上,和目标码一起,附有给第三方提供相应的机器可读的源码的书面报价。有效期不少于3年,费用不超过实际完成源程序发布的实际成本。源码的发布应符合上面的第1,2款的要求。或者

c) 和目标码一起,附有你收到的发布源码的报价信息(这一条款只适用于非商业性发布,而且你只收到程序的目标码或可执行代码和按(b)款要求提供的报价)。

作品的源码指的是对作品进行修改最优先选取的形式。对可执行的作品讲,完整的源码包括:所有模块的所有源程序,加上有关的接口的定义,加上控制可执行作品的安装和编译的 script。作为特殊例外,发布的源码不必包含任何常规发布的供可执行代码在上面运行的操作系统的主要组成部分(如编译程序、内核等),除非这些组成部分和可执行作品结合在一起。

如果采用提供对指定地点的访问和复制的方式发布可执行码或目标码,那么,提供对同一地点的访问和复制源码可以算作源码的发布,即使第三方不强求与目标码一起复制源码。

4. 除非你明确按许可证提出的要求去做,否则你不能复制、修改、转发许可证和发布程序。任何用其他方式复制、修改、转发许可证和发布程序是无效的,而且将自动结束许可证赋予你的权利。然而,对那些从你那里按许可证条款得到副本和权利的人们,只要他们继续全面履行条款,许可证赋予他们的权利仍然有效。

5. 你没有在许可证上签字,因而你没有必要一定接受这一许可证。然而,没有任何其他东西赋予你修改和发布程序及其衍生作品的权利。如果你不接受许可证,这些行为是法律禁止的。因此,如果你修改或发布程序(或任何基于程序的作品),就表明你接受这一许可证,以及它的所有有关复制、发布和修改程序或基于程序的作品条款和条件。

6. 每当你重新发布程序(或任何基于程序的作品)时,接受者自动从原始许可证颁发者那里接受受这些条款和条件支配的复制、发布或修改程序的许可证。你不可以对接受者履行这里赋予他们的权利强加其他限制。你也没有强求第三方履行许可证条款的义务。

7. 如果由于法院判决或违反专利的指控或任何其他原因(不限于专利问题)的结果,强加于你的条件(不管是法院判决,协议或其他)和许可证的条件有冲突,他们也不能用许可证条款为你开脱。

在你不能同时满足本许可证规定的义务及其他相关的义务时,作为结果,你可以根本不发布程序。例如,如果某一专利许可证不允许所有那些直接或间接从你那里接受副本的人们在不付专利费的情况下重新发布程序,唯一能同时满足两方面要求的办法是停止发布程序。

如果本条款的任何部分在特定的环境下无效或无法实施,就使用条款的其余部分。并将条款作为整体用于其他环境。

本条款的目的不在于引诱你侵犯专利或其他财产权的要求,或争论这种要求的有效性。本条款的主要目的在于保护自由软件发布系统的完整性。它是通过通用公共许可证的应用来实现的。许多人坚持应用这一系统,已经为通过这一系统发布大量自由软件作出慷慨的奉献。作者/捐献者有权决定他/她是否通过任何其他系统发布软件。许可证持有人不能强制这种选择。

本节的目的在于明确说明许可证其余部分可能产生的结果。

8. 如果由于专利或者由于有版权的接口问题使程序在某些国家的发布和使用受到限制,将此程序置于许可证约束下的原始版权拥有者可以增加限制发布地区的条款,将这些国家明确排除在外,并在这些国家以外的地区发布程序。在这种情况下,许可证包含的限制条款和许可证正文一样有效。

9. 自由软件基金会可能随时出版通用公共许可证的修改版或新版。新版和当前的版本在原则上保持一致,但在提到新问题或有关事项时,在细节上可能出现差别。

每一版本都有不同的版本号。如果程序指定适用于它的许可证版本号以及“任何更新的版本”,你有权选择遵循指定的版本或自由软件基金会以后出版的新版本,如果程序未指定许可证版本,你可选择自由软件基金会已经出版的任何版本。

10. 如果你愿意将程序的一部分结合到其他自由程序中,而它们的发布条件不同。写信给作者,要求准予使用。如果是自由软件基金会加以版权保护的软件,可写信给自由软件基金会。我们有时会作为例外的情况处理。我们的决定受两个主要目标的指导。这两个主要目标是:我们的自由软件的衍生作品继续保持自由状态,以及从整体上促进软件的共享和重复利用。

没有担保

11. 由于程序准予免费使用,在适用法准许的范围内,对程序没有担保。除非另有书面说明,版权所有者和/或其他提供程序的人们一样不提供任何类型的担保,不论是明确的,还是隐含的,包括但不限于隐含的适销和适合特定用途的保证。全部的风险,如程序的质量和性能问题都由你来承担。如果程序出现缺陷,你承担所有必要的服务、修复和改正的费用。

12. 除非适用法或书面协议有要求,在任何情况下,任何版权所有者或任何按许可证条款修改和发布程序的人们都不对你的损失负有任何责任,包括由于使用或不能使用程序引起的任何一般的、特殊的、偶然发生的或重大的损失(包括但不限于数据的损失,或者数据变得不精确,或者你或第三方的持续的损失,或者程序不能和其他程序协调运行等)。即使版权所有和其他人提到这种损失的可能性也不例外。

3. 最后的条款和条件

如何将这些条款用到你的新程序

如果你开发了新程序,而且你需要它得到公众最大限度的利用,要做到这一点的最好办法是将它变为自由软件,使得每个人都能在遵守条款的基础上对它进行修改和重新发布。为了做到这一点,给程序附上下列声明。最安全的方式是将它放在每个源程序的开头,以便最有效地传递拒绝担保的信息。每个文件至少应有“版权所有”行以及在什么地方能看到声明全文的说明。

<用一行空间给出程序的名称和它用来做什么的简单说明>

版权所有(C)19xx(<作者姓名>

这一程序是自由软件,你可以遵照自由软件基金会出版的 GNU 通用公共许可证条款来修改和重新发布这一程序。或者用许可证的第二版,或者(根据你的选择)用任何更新的版本。

发布这一程序的目的是希望它有用,但没有任何担保,甚至没有适合特定目的的隐含的担保。更详细的情况请参阅 GNU 通用公共许可证。

你应该已经和程序一起收到一份 GNU 通用公共许可证的副本。

如果还没有,写信给:

The Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA02139, USA

还应加上如何和你保持联系的信息。

如果程序以交互方式进行工作,当它开始进入交互方式工作时,使它输出类似下面的简短声明:

Gnomovision 第 69 版,版权所有(C)19XX,作者姓名,

Gnomovision 绝对没有担保。要知道详细情况,请输入 'show w'。

这是自由软件,欢迎你遵守一定的条件重新发布它,要知道详细情况,请输入 'show c'。

假设的命令 'show w' 和 'show c' 应显示通用公共许可证的相应条款。当然,你使用的命令名称可以不同于 'show w' 和 'show c'。根据你的程序的具体情况,也可以用菜单或鼠标选项来显示这些条款。

如果需要,你应该取得你的上司(如果你是程序员)或你的学校签署放弃程序版权的声明。下面只是一个例子,你应该改变相应的名称:

Ynyodyne 公司以此方式放弃 James Harker 所写的 Gnomovision 程序的全部版权利益。

<Ty coon 签名>, 1989.4.1

Ty coon 副总裁

这一许可证不允许你将程序并入专用程序。如果你的程序是一个子程序库。你可能会认为用库的方式和专用应用程序连接更有用。如果这是你想做的事,请使用 GNU 库通用公共许可证代替本许可证。

附录 B GDB 远程串行通信协议

GDB 远程串行通信协议(RSP)是 gdb 和远端目标所进行交流的通用语言。它定义了读写数据的信息,控制被调试的应用程序,并报告应用程序的状态。这些功能在主机一方是通过 gdb 的 remote.c 源文件实现的。下列是 GDB 的最重要的命令的简短描述:

使用 GDB 远程串行通信协议时,在 gdb 和远端目标间交换的数据是使用简单的 ASCII 码字符实现的。一个美元符号(\$)表示信息的起始,一个#和一个八位的校验和表示信息的结束。换言之,各个信息如下所示:

```
$ <data> # CKSUM_MSN CKSUM_LSN
```

在此,<data>是一个典型的 ASCII 码十六进制字符串[0-9,a-f,A-F]。CKSUM_MSN 和 CKSUM_LSN 都是十六进制的 ASCII,表示 <data>的一个八位的校验和。当发送信息时,接受方响应如下:

A + (如果接收校验和正确而且此时接收方已经准备好接收下一个包)

A - (如果接收校验和不正确,此时,信息必须重新发送)

目标响应从 gdb 来的信息的方式有两种:一个表示 OK 的数据符号(如果消息发送正确)或由目标自己定义的错误码,如附表 B-1 所示。当 gdb 接收到错误码时,就通过 gdb 控制台向用户报告该错误码代号。对于<data>的定义如附表 B-2、附表 B-3、附表 B-4 和附表 B-5 所示。

附表 B-1 目标状态信息(来自目标的响应)

消息名称	<data>	定义描述
last signal response	Snn	对于 last signal 指令的最小化回应
expected response	Tnnr...v...r...v...	最近一次的 signal 内容,加上 key 寄存器的值
console output	Ovvvvvvvv...	从目标发送文本信息至 gdb 的控制台

附表 B-2 寄存器相关指令

命令名称	<data>定义	描述
read registers	G	返回所有寄存器的值
write registers	GXX..XX	将寄存器的值设置为 XX..XX
write register nn	Pnn = XX..XX	设置寄存器 NN 的值

附表 B-3 内存相关指令

命令名称	<data>定义	描述
read memory	mAA..AA,LL..LL	从内存读值
write memory	MAA..AA,LL..LL: XX..XX	向内存写值

附表 B-4 目标信息指令

命令名称	<data>定义	描述
query section offsets	qOffsets	返回段偏移信息

附表 B-5 程序控制指令

命令名称	<data>定义	描述
set thread	Hc	设置当前程序
step	sAA..AA	执行某个汇编指令
continue	cAA..AA	继续执行应用程序
last signal	?	报告最后的信号
kill	k	终止应用程序的执行

set thread 指令是设计在仅可以单线程执行时来单步调试和设置断点的,同时不中断其他的线程的运行。在运行多线程应用程序的嵌入式系统中(例如,在包括 RTOS 的设计中),调试用 stub 可使用该指令提供的信息来决定当遇到断点命令时如何重新运作。如果指定的线程遇到了断点指令的话,就让指令的执行挂起;如果没有遇到,那么就继续执行。在非多任务的嵌入式系统中,或在单步调试时停止所有线程的嵌入式系统中,对于该信息较好的办法是把它忽略掉,或回应以 OK。前者较好,但是后者可提高 gdb 的性能。

continue 命令告诉目标继续执行程序。如果有参数 AA...AA 存在,那么控制命令就从此处执行。

step 命令告诉目标执行应用程序的一个指令。如果提供了参数地址 AA...AA,目标就应该从该处的指令开始执行,否则,目标就只执行下一条指令。当目标执行完了指令后,它以 continue 指令一样的方式发出回应信息。

对于 continue 或 step 指令而言,并没有直接的响应信息。当目标遇到了下一个断点或者出现未经处理的异常,或应用程序退出时,目标响应以下面列出来的“目标状态”信息。

gdb 使用 last signal 指令来请求目标的当前状态,包括由 stub 接收的从应用程序发送过来的最近期信号。Stub 也使用了下面给出的目标状态信息来响应。

当 gdb 想终止调试应用程序时,就发送 kill 指令。

例 附 B-1 set thread

gdb 发送: \$Hc-1#09

目标响应: + \$OK#9a

例 附 B-2 kill

gdb 发送: \$k#6b

目标响应: + \$OK#9a

last signal response 信息是对 gdb 发送的 last signal 指令的最小化的回应。

Expected response 是 last signal 响应信息的加强,它包含了 last signal 和寄存器的值。如果该信息中包含了 gdb 所需要的所有的存储器的值(例如,PC 和堆栈指针),那么,gdb 就不会接着产生一个读寄存器的指令。这样就增强了 gdb 的性能,特别是对于速度缓慢的串行连接而言。

对于 last signal response 和目标状态响应信息而言,nn 域是一个由 GDB 信号表获取的一个信号数。在目标状态响应信息中,r 是一个寄存器数,v 是寄存器的值(都以十六进

制的 ASCII 码形式出现)。

控制台信息可以用来由目标向 gdb 控制台发送文本信息。同其他的信息一样,这个要显示的文本信息必须也同样是十六进制的 ASCII 码格式,必须用一个“\n”结尾。

例 附 B-3 step

gdb 发送: S s#73

目标响应以: + \$S05#b8

该响应指出,目标执行了一个指令,并正在等待新的指令。

例 附 B-4 continue(expedited response)

gdb 发送: \$c#63

目标响应以: + \$T0510:1238;F:FFE0..#xx

该目标响应指出,它遇到一个断点(signal 5),在地址 PC = 0x1238 处(目标的 PC 是寄存器 16),应用程序的堆栈指针是 SP = 0xffe0(目标的 SP 是寄存器 15)。

例 附 B-5 向 gdb 控制台发送“Hello, world!”

目标发送: \$O48656c6c6f2c20776f726c64210a#55

gdb 响应以: +

最后,附表 B-6 给出了所有的 GDB 信号值,并给出了为 GDB 的各种响应信号的定义。

附表 B-6 GDB 信号

信号值	信号名称
0	Hangup(挂起)
1	Interrupt(中断)
2	Quit(关闭)
3	Illegal instruction(非法指令)
4	Trace/breakpoint trap(跟踪/断点陷阱)
5	Aborted(异常终止)
6	Emulation trap(仿真陷阱)
7	Arithmetic exception(代数异常)
8	Killed(杀死)
9	Bus error(总线错误)
10	Terminated(终止)
15	User defined(用户定义)
33...63	User-defined(由用户定义)





Think different.

Powered by xiaoguo's publishing studio
QQ:8204136