高等院校嵌入式系统通用教材 · ARM嵌入式技术系列教程

ARM嵌入式VxWorks实践教程

李忠民　杨　刚　顾亦然　刘尚军　编著

北京航空航天大学出版社

ARM嵌入式技术系列教程

★ ARM嵌入式VxWorks实践教程

◇ 配套 JXARM9-2410 ARM嵌入式教学实验系统

★ ARM嵌入式技术原理与应用

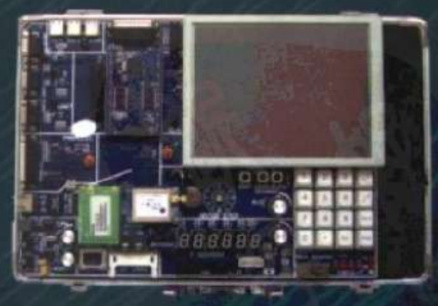
◇ 配套开放式多媒体教学课件

★ ARM嵌入式技术实践教程

◇ 配套开放式多媒体实验教学课件
◇ 配套 JX44B0 ARM嵌入式教学实验系统

★ ARM9嵌入式技术及Linux高级实践教程

◇ 配套开放式多媒体实验教学课件
◇ 配套 JXARM9-2410 ARM嵌入式教学实验系统



本书配套 JXARM9-2410-3 嵌入式教学实验系统

ISBN 7-81077-747-5
定价：28.00元

高等院校嵌入式系统通用教材·ARM嵌入式技术系列教程

ARM嵌入式VxWorks实践教程

TP332
62

李忠民　杨　刚
顾亦然　刘尚军　编著

北京航空航天大学出版社

内容简介

本书是《ARM嵌入式技术系列教程》之一。采用JX2410实验系统作为硬件平台，详细讲解了风河公司开发平台Tornado II的建立和使用，内容覆盖典型应用系统开发的各个阶段。以S3C2410处理器为例，结合ARM嵌入式处理器的结构特点，描述如何在Tornado II上进行BSP板级支持包的移植，从最基本的中断处理、定时器处理到一些常用的设备驱动（如串口、键盘驱动和网卡驱动等）；还涉及VxWorks应用程序的设计与开发；另外，还从实际应用出发，介绍如何使用Tornado II来调试VxWorks应用程序。

本书可作为VxWorks初学者的实践教程，对于VxWorks开发人员也有一定参考价值。

图书在版编目(CIP)数据

ARM嵌入式VxWorks实践教程/李忠民等编著. —北京:北京航空航天大学出版社,2006.3

ISBN 7-81077-747-5

Ⅰ. A… Ⅱ. 李… Ⅲ. ①微处理器,ARM—教材 ②实时操作系统,VxWorks—程序设计—教材

Ⅳ. TP332②TP316.2

中国版本图书馆CIP数据核字(2006)第016847号

©2006,北京航空航天大学出版社,版权所有。

未经本书出版者书面许可,任何单位和个人不得以任何形式或手段复制或传播本书内容。侵权必究。

ARM嵌入式VxWorks实践教程

李忠民 杨 刚 顾亦然 刘尚军 编著

责任编辑 芦潇静

*

北京航空航天大学出版社出版发行

北京市海淀区学院路37号(100083) 发行部电话:010-82317024 传真:010-82328026

http://www.buaapress.com.cn E-mail:bhpress@263.net

涿州市新华印刷有限公司印装 各地书店经销

*

开本:787 mm×960 mm 1/16 印张:20.5 字数:459千字

2006年3月第1版 2006年3月第1次印刷 印数:5 000册

ISBN 7-81077-747-5 定价:28.00元

序

VxWorks是嵌入式操作系统中的一颗奇葩，几年前在中国开始推广。它凭借其优秀的实时性和稳定性，得到了广大用户的认可。在该操作系统上进行应用开发的队伍不断壮大，领域也在逐步拓展，随之而来希望学习该操作系统的人员也就越来越多；但目前的状况是缺少资料，特别是一些与实际相结合的资料。本书的出版将改变这一状况，它是一本学习VxWorks操作系统难得的好书，具有以下几个特点：

(1) 理论联系实际

本书以JX2410教学系统为硬件平台，该平台采用Samsung公司的S3C2410微处理器，这是目前很有代表性也非常流行的一款ARM处理器。有了这个硬件平台，书中的内容就不再是空中楼阁。读者一方面可在S3C2410系列的平台上，参考本书内容学习VxWorks操作系统；另一方面也可按照本书提供的思路，将VxWorks移植到其他平台，比如x86、PPC和MIPS等。

(2) 内容丰富

本书涉及使用Tornado开发工具进行系统开发的各阶段，从最基本的Tornado开发环境的安装和设置，到工程的建立、BSP移植、驱动程序和应用程序的开发，以及Tornado调试工具的使用等。详细分析了BSP在JX2410平台上的移植，并且讲述了一些典型驱动程序的编写，包括字符设备、块设备和网络设备等。

(3) 重点突出

学习VxWorks操作系统的人员大多感觉Tornado调试工具很好，但难以使用；目标系统的BSP移植很困难。本书针对这一情况，对调试工具的使用和BSP的移植进行了重点介绍，

并且通过大量的图表和程序，结合实例讲述了调试工具的作用和使用方法，分析了 BSP 的构成以及 BSP 的移植过程。

(4) 适用范围广

本书最初的目标对象是针对一些初学者，比如学习这门课程的大专院校学生以及进行 VxWrosk 应用开发的初学者等；但是由于本书的内容覆盖面较广，因此一些从事过 VxWorks 应用开发的工程技术人员，也会在本书中发现有很多值得一读的内容。

总之，本书无论是对初学者还是对工程技术人员来说，都是一本很有价值的书，相信读者通过阅读本书可以学到不少东西。

美国风河公司北京代表处首席代表

韩　青

2006 年 2 月 16 日

前 言

嵌入式系统是近几年流行起来的一个新方向，涉及众多应用领域，包括工业控制、汽车电子、消费电子和军事国防等。当然早期的单片机和工控机系统等都可归入这个范畴，但与它们相比，现在的嵌入式系统有了一些新的特点：

- 功能强：单片机的运算能力一般为几个～几十个 MIPS，远远不能满足现在的一些嵌入式应用的要求；
- 功耗低：现在的嵌入式系统十分强调功耗，这对工控机来说是个巨大的挑战；
- 使用嵌入式操作系统：现在的嵌入式应用中，由于系统的复杂性，比如需要一些通信协议和图形系统等，这决定采用操作系统的必要性。

正是上述特点，使得现在的嵌入式系统从硬件平台到软件上都有了一些变化，广泛采用 SOC 体系的嵌入式处理器和专用的嵌入式操作系统。

ARM 是嵌入式处理器中事实上的标准，而 VxWorks 则是美国风河公司经过 20 多年的经验积累，开发出的一个非常优秀的嵌入式操作系统。本书以 JX2410 实验系统为硬件平台讲述 VxWorks 操作系统的应用开发。

JX2410 采用 S3C2410 这款非常有代表性的 ARM 处理器。该芯片集成了大量的外设，如串口、主/从 USB 口、LCD 控制器、SD/MMC 控制器、内存控制器和中断控制器等，而且具有 MMU 管理单元，运行频率高达 203 MHz。JX2410 实验系统还扩展了网卡以及 GPS、GPRS 等模块，接口比较全面。本书针对这些接口模块，讲述如何为 VxWorks 编写字符设备、块设备以及网络设备的驱动等。

在 VxWorks 操作系统的应用中，BSP 开发是一个难点。本书通过循序渐进的方式讲述在 S3C2410 上进行 BSP 开发的过程，从 BSP 代码的构成到 BSP 各模块代码的编写，以及 BSP 的发布和使用。与其他嵌入式开发工具相比，风河公司的开发工具——Tornado，在仿真调试方面具有很大优势。该开发工具集成了多种调试手段（如目标机 Shell 工具、目标机浏览器和 WindView 等），可以十分方便地用于软件调试和系统性能的分析。本书对这些调试工具也进行了详细的讲解。

本书的编写过程中，得到了美国风河公司北京代表处韩青、王祖强、戴宇文等的大力支持，他们参与了本书的编写和前期校对等工作，还得到了赵敏、张芬和刘铁刚等的协助，在此表示衷心的感谢；同时还要感谢美国风河公司大学计划所提供的大力支持。

由于作者水平有限，书中难免存在错误和不当之处，恳请各位同仁和读者批评指正。

作　者

2006 年 2 月 13 日

目　录

第 1 章　嵌入式系统概述 …… 1
1.1　嵌入式系统——后 PC 时代的主宰 …… 1
1.2　嵌入式系统的发展历程 …… 2
1.3　嵌入式系统的一些基本概念 …… 4
1.4　嵌入式操作系统 …… 5
1.5　嵌入式系统的应用领域 …… 8
1.6　嵌入式系统的未来发展趋势 …… 11
第 2 章　ARM 体系结构 …… 13
2.1　ARM 微处理器的应用领域及特点 …… 13
2.1.1　ARM 微处理器的应用领域 …… 13
2.1.2　ARM 微处理器的特点 …… 14
2.2　处理器模式 …… 16
2.3　寄存器组织 …… 16
2.3.1　ARM 状态下的寄存器组织 …… 17
2.3.2　Thumb 状态下的寄存器组织 …… 19
2.3.3　程序状态寄存器 …… 20
2.4　异　常 …… 22
2.4.1　ARM 体系结构所支持的异常类型 …… 23
2.4.2　对异常的响应 …… 23

2.4.3 从异常返回 …… 24
2.4.4 各类异常的具体描述 …… 24
2.4.5 异常向量 …… 25
第 3 章 Tornado II 集成开发环境的建立与使用 …… 27
3.1 Tornado II 集成开发环境简介 …… 27
3.2 Tornado II 集成开发环境安装 …… 30
3.3 使用 Tornado II 创建新的工程 …… 36
3.3.1 新建工程 …… 36
3.3.2 工程管理 …… 38
3.4 Tornado II 的调试工具 …… 46
3.4.1 集成仿真工具 …… 46
3.4.2 目标机服务器 …… 51
3.4.3 调试命令行解释器 …… 54
3.4.4 调试器 …… 57
3.4.5 目标机浏览器 …… 69
3.4.6 软件逻辑分析器 …… 74
第 4 章 VxWorks BSP 的移植 …… 76
4.1 VxWorks 内核的特点及 BSP 简介 …… 76
4.1.1 VxWorks 内核的特点 …… 76
4.1.2 VxWorks 的主要功能和结构 …… 78
4.1.3 VxWorks BSP 的简介 …… 79
4.1.4 VxWorks BSP 的文件组织 …… 81
4.2 VxWorks 的引导过程 …… 82
4.3 VxWorks BSP 的移植 …… 86
4.3.1 Makefile …… 86
4.3.2 内核配置 …… 95
4.3.3 带 ROM 启动功能内核前期初始化 …… 100
4.3.4 定时器处理 …… 112
4.3.5 中断处理 …… 120
4.4 组件管理 …… 123
4.5 BSP 的调试 …… 142
第 5 章 VxWorks 驱动程序的编写 …… 149
5.1 设备驱动分类及特点 …… 149
5.2 字符设备驱动 …… 155

5.2.1 字符设备驱动程序 …… 155
5.2.2 键盘驱动程序编写 …… 160
5.3 块设备驱动 …… 169
5.3.1 块设备驱动程序 …… 169
5.3.2 RAMDISK 驱动程序编写 …… 172
5.4 串口设备驱动 …… 178
5.4.1 串口设备驱动程序 …… 178
5.4.2 ttyDrv 的层次结构 …… 179
5.4.3 S3C2410 串口驱动的编写 …… 181
5.5 网络设备驱动 …… 195
5.5.1 MUX 网络设备驱动程序 …… 195
5.5.2 RTL8019 网络芯片简介 …… 198
5.5.3 网络驱动程序编写 …… 202
5.6 文件系统 …… 231
5.6.1 TSFS …… 231
5.6.2 dosFS …… 233
5.6.3 TrueFFS …… 236
5.7 驱动程序中的数据一致性 …… 245
第6章 VxWorks 应用程序的编写 …… 247
6.1 VxWorks 应用程序调试环境的建立 …… 247
6.2 任务管理 …… 252
6.2.1 任 务 …… 252
6.2.2 任务调度 …… 254
6.2.3 任务操纵 …… 255
6.2.4 共享代码和可重入代码 …… 265
6.2.5 系统任务 …… 267
6.2.6 注意事项 …… 267
6.3 任务通信 …… 268
6.3.1 共享存储区 …… 269
6.3.2 互 斥 …… 269
6.3.3 信号量 …… 270
6.3.4 消息队列 …… 277
6.3.5 管 道 …… 280
6.4 看门狗定时器管理 …… 280

6.5 中断管理 …… 282
6.6 网络通信 …… 285
6.6.1 网络协议 …… 285
6.6.2 套接字的使用 …… 287
6.6.3 网络通信程序及说明 …… 288
6.7 异常捕捉和错误处理 …… 291
附录 ARM 微处理器的指令系统 …… 295
参考文献 …… 315

第1章 嵌入式系统概述

1.1 嵌入式系统——后PC时代的主宰

几年前，有位IT界的资深人士坦言："PC时代即将结束"。全世界都为之震惊，在随后的几年中，整个IT产业在大家的见证下开始走入另一个时代，有人将这个新的时代称为"Internet时代"，有人称之为"3C时代"，也有人称之为"后PC时代"。计算机、通信和消费类产品的技术结合起来，以3C产品的形式迅速渗透到民用消费品、工业控制以及军事国防等领域。

现在的社会是一个高度信息化、网络化的社会，计算机和网络已经全面渗透到日常生活的每个角落。我们需要的已经不仅是那种放在桌上处理文档，进行工作管理和生产控制的计算机"机器"。各种各样的新型嵌入式系统设备在应用数量上已经远远超过通用计算机。任何一个普通人都可能拥有从小到大各种使用嵌入式技术的电子产品，小到MP3播放器、PDA、手机和数码相机等微型数字化产品，大到车载电子以及正在兴起的网络家电和智能家电等；在工业控制领域，使用嵌入式技术的数字机床、智能工具和工业机器人等也层出不穷；在军事国防等领域都可以见到嵌入式系统的广泛应用。也正是嵌入式技术的广泛应用，使得嵌入式系统技术成为最热门的技术之一，吸引了大批的优秀人才。

广义上讲，凡是带有微处理器的专用软硬件系统都可称为"嵌入式系统"。作为系统核心的微处理器又包括3类：微控制器（MCU）、数字信号处理器（DSP）和嵌入式微处理器（MPU）。简单地说：嵌入式系统就是指操作系统和功能软件集成于计算机硬件系统之中的设备。也有人认为嵌入式系统就是以应用为中心，以计算机技术为基础，软硬件可裁剪，适用于应用系统对功能、可靠性、成本、体积和功耗要求严格的专用计算机系统。嵌入式系统是一个非常广泛的概念，但还是可以为它归纳出一些特点：

1. 专用性

系统的功能非常明确，仅包含一些必需的功能，这点与传统 PC 有很大区别。在传统的 PC 机或者小型机、大型机上，安装不同的软件即可构成不同的系统；而嵌入式系统则要求功能非常明确。按照这个特点，一个嵌入式系统的资源，无论是软硬件，还是整个系统的体积和功耗等，都应该是高度精简和严格控制的。嵌入式系统面向用户、面向产品、面向应用，它必须与具体应用相结合才会具有生命力，才更具有优势。也正是这个原因，必须结合实际系统需求对嵌入式系统进行合理的裁剪。

2. 嵌入性

系统和被控制的对象是紧密连接的，一般无需人为干预。从这点上讲，也就对嵌入式系统的环境适应性、稳定性和可靠性等提出了一些要求。

3. 智能性

嵌入式系统需要一个中央处理器单元来实现对对象的智能控制，而中央处理器单元的种类很多，如 x86 体系结构的处理器、8/16 位单片机、32 位处理器和 DSP 等。从这点上看，嵌入式系统与计算机技术是密不可分的。

根据 IEEE（国际电气及电子工程师协会）的定义，嵌入式系统是"控制、监视或者辅助控制机器和设备运行的系统"（devices used to control, monitor or assist the operation of equipment, machinery or plants）。该定义主要从应用的角度出发，从中可以看出嵌入式系统是软件和硬件的结合体，同时还涵盖机械等附属装置。嵌入式系统是将计算机技术、半导体技术和电子技术与各行业的具体应用相结合的产物，是一门综合技术学科。由于空间和各种资源相对不足，因此必须对嵌入式系统的软硬件进行高效的设计，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，这样才能在具体应用中对处理器的选择更具竞争力。

1.2 嵌入式系统的发展历程

虽然嵌入式系统这一名词在最近几年才流行起来，但早在 20 世纪 80 年代，国际上就有一些 IT 组织、公司，开始进行商用嵌入式系统和专用操作系统的研发。从硬件方面讲，32 位和 64 位微处理器是目前嵌入式系统的核心，它们的使用同样也是未来发展的一大趋势。为了抢占这个无限广阔的市场，各大硬件厂商竞相推出产品，包括 Intel、Motorola、Philips 和 AMD 等公司都不甘示弱，几乎每个月都有新产品出现。市场之争日益激烈，同时也给嵌入式技术的发展带来了无限活力。从 20 世纪 70 年代单片机的出现，到今天各式各样的嵌入式微处理器、微控制器的大规模应用，嵌入式系统已有近 30 年的发展历史。纵观嵌入式技术的发展过程，

大致经历了 4 个阶段。

1. 无操作系统的嵌入式算法阶段

这一阶段的嵌入式系统是以单芯片为核心的系统，具有与一些监测、伺服和指示设备相配合的功能。一般无明显的操作系统支持，而是通过汇编语言编程对系统进行直接控制。主要特点是：系统结构和功能相对单一，针对性强，无操作系统支持，几乎没有用户接口。

嵌入式系统的出现最初是基于单片机的。20 世纪 70 年代单片机的出现，使得汽车、家电、工业机器、通信装置，以及成千上万种产品可以通过内嵌电子装置来获得更佳的使用性能：更易使用，更快，更便宜。这些装置已初步具备了嵌入式的应用特点，但这时的应用只是使用 8 位芯片，执行一些单线程的程序，还谈不上“系统”的概念。

最早的单片机是 Intel 公司的 8048，出现在 1976 年。在随后的时间里，其他公司也相继推出了自己的单片机处理器，例如：Motorola 公司推出了 68HC05，Zilog 公司推出了 Z80 系列等。这些早期的单片机一般包含一些必备的资源，如：约 256 字节的 RAM、4 KB 的 ROM、4 个 8位并口、1 个全双工串行口和 2 个 16 位定时器。之后在 20 世纪 80 年代初，Intel 公司又进一步完善了 8048，在此基础上研制成功了 8051，这在单片机史上是值得纪念的一页。迄今为止，51 系列单片机仍然是最为成功的单片机芯片，在各种产品中有着非常广泛的应用，特别是在一些价格敏感，而对性能又无严格要求的场合。这些曾经风光一时的处理器都可以算作是嵌入式处理器的鼻祖。

2. 简单监控式的实时操作系统阶段

这一阶段的嵌入式系统主要以嵌入式处理器为基础，以简单监控式操作系统为核心。系统的特点是：处理器种类繁多，通用性较差；开销小，效率高；一般配备系统仿真器，具有一定的兼容性和扩展性；用户界面不够友好，主要用来控制系统负载以及监控应用程序运行。

从 20 世纪 80 年代早期开始，嵌入式系统的程序员开始用商业级的“操作系统”编写嵌入式应用软件，从而获取更短的开发周期、更少的开发资金和更高的开发效率，真正的“嵌入式系统”出现了。确切地说，这时的操作系统是一个实时核，这个实时核具有许多传统操作系统的特征，包括任务管理、任务间通信、同步与互斥、中断支持和内存管理等功能。其中比较著名的有 Ready System 公司的 VRTX、ISI(Integrated System Incorporation)公司的 pSOS、风河公司的 VxWorks 和 QNX 公司的 QNX 等。这些嵌入式操作系统都具有嵌入式的典型特征：抢占采用抢占式调度，响应时间很短，任务执行的时间可以确定；系统内核很小，可裁剪，可扩充，可移植到各种处理器上；实时性和可靠性较强，适合嵌入式应用。这些嵌入式实时多任务操作系统的出现，使得应用开发人员从小范围的开发中解放出来，同时也促使嵌入式有了更为广阔的应用空间。

3. 通用的嵌入式实时操作系统阶段

以通用型嵌入式实时操作系统为标志的嵌入式系统(如 VxWorks、pSOS 和 Win CE)是

这一阶段的典型代表。这一阶段嵌入式系统的特点是:能运行在各种不同的微处理器上;具有强大的通用型操作系统功能,例如具备了文件和目录管理、多任务、设备驱动支持、网络支持、图形窗口以及用户界面等功能;具有丰富的 API 和嵌入式应用软件。

20 世纪 90 年代以后,随着对实时性要求的提高,软件规模不断扩大,实时核逐渐发展为实时多任务操作系统(RTOS),并作为一种软件平台逐步成为目前国际嵌入式系统的主流。这时更多的公司看到了嵌入式系统的广阔发展前景,开始大力发展自己的嵌入式操作系统。除了上述几家老牌公司以外,还出现 Palm OS、Win CE、嵌入式 Linux、Lynx、Nucleus、以及国内的 Hopen 和 Delta Os 等嵌入式操作系统。随着嵌入式技术的发展前景日益广阔,相信会有更多的嵌入式操作系统软件出现。

4. 以 Internet 为标志的嵌入式系统阶段

随着通用型嵌入式实时操作系统的发展,面向 Internet 网络和特定应用的嵌入式操作系统日益引起人们的重视,成为重要的发展方向。嵌入式系统与 Internet 的真正结合,嵌入式操作系统与应用设备的无缝结合代表着嵌入式操作系统发展的未来。

1.3 嵌入式系统的一些基本概念

嵌入式系统中有许多非常重要的概念,它们从不同的侧面体现了嵌入式系统的一些特征。

1. 嵌入式处理器

嵌入式系统的核心,是控制、辅助系统运行的硬件单元。其范围极其广阔,从最初的 4 位处理器,目前仍在大规模应用的 8 位单片机,到最新的受到广泛青睐的 32 位和 64 位嵌入式微处理器。

2. 实时操作系统

操作系统是嵌入式系统目前最主要的组成部分之一。根据操作系统的工作特性,“实时性”是指进程完成操作的时间确定性。实时操作系统(Real Time Operating System)具有实时性,一般都会采取一些特殊的手段来保证整个系统的实时性。其中:实时性是第一要求,操作系统会调度一切可利用的资源完成实时控制任务;其次才着眼于提高计算机系统的使用效率,重要特点是须满足对时间的限制和要求。

3. 分时操作系统

对于分时操作系统,软件的执行在时间上的要求并不严格;时间上的错误一般不会导致灾难性的后果。目前,分时操作系统的强项在于多任务的管理;而实时操作系统的重要特点则是具有系统的确定性,即系统能对运行的最好和最坏等情况做出精确的估计。

4. 任 务

在实时操作系统中，“任务”是参与资源（如CPU、Memory和I/O设备等）竞争的基本单位。在概念上，一个任务与另一个任务之间可以并行、相互独立地执行。

任务包括系统任务和用户任务两类。关于用户任务的划分并没有一个固定的法则，但很明显，划分太多将导致任务间的切换过于频繁，系统开销太大；划分太少又会导致实时性和并行性下降，从而影响系统的效率。一般来说，功能模块A与功能模块B是分为两个任务还是合为一个任务，可从是否具有时间相关性、优先性、逻辑特性和功能耦合等几方面考虑。

5. 多任务操作系统

系统支持多任务管理以及任务间的同步与通信，传统的单片机系统和DOS系统等对多任务支持的功能很弱，而目前的Windows是典型的多任务操作系统。在嵌入式应用领域中，多任务是一个普遍的要求。

6. 实时操作系统中的重要概念

系统响应时间(System Response Time)：系统发出处理要求到系统给出应答信号的时间。

任务切换时间(Context-Switching Time)：任务之间切换而使用的时间。

中断延迟(Interrupt Latency)：计算机接收到中断信号到操作系统作出响应，并完成任务切换转入中断服务程序的时间。

7. 实时操作系统的工作状态

实时系统中的任务有4种状态：运行(Executing)、就绪(Ready)、挂起(Suspended)和休眠(Dormant)。

运行：获得CPU控制权。

就绪：进入任务等待队列，通过调度转为运行状态。

挂起：任务发生阻塞，移出任务等待队列，等待系统实时事件的发生而唤醒，从而转为就绪或运行。

休眠：已完成或因错误而被清除的任务，也可认为是系统中不存在的任务。

任何时刻系统中只能有一个任务处在运行状态，各任务按级别通过时间片或其他策略分别获得对CPU的控制权。

1.4 嵌入式操作系统

嵌入式操作系统是一种支持嵌入式系统应用的操作系统软件。它是嵌入式系统（包括软硬件系统）极为重要的组成部分，通常包括与硬件相关的底层驱动软件、系统内核、设备驱动接

口、通信协议、图形界面和标准化浏览器等。嵌入式操作系统具有通用操作系统的基本特点，例如：能够有效地管理越来越复杂的系统资源；能够把硬件虚拟化，使得开发人员从繁忙的驱动程序移植和维护中解脱出来；能够提供库函数、标准设备驱动程序以及工具集等。与通用操作系统相比，嵌入式操作系统在系统实时高效性、硬件的相关依赖性、软件集成化以及应用的专用性等方面具有较为突出的特点。

早在20世纪70年代，大部分嵌入式系统的软件都是由汇编语言编写而成的，只能应用于某种特定的微处理器和应用。当然，这对于某些简单的应用足够了。自1981年发展了世界上第一个商业嵌入式实时操作系统（VRTX）至今，嵌入式实时操作系统已有20多年的发展历程。20世纪80年代的产品还仅支持16位微处理器，只有内核，以销售二进制代码为主；当时的产品（如VRTX和pSOS等）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的设计思想已开始渗入其中。特别是因特网日渐风行，用户都要求有网络（即支持浏览器）和图形功能，能方便地使用大量现有的软件代码，能支持标准的API（如POSIX和Win32等），并希望其开发环境与大家熟悉的Unix和Windows一致。在这个时期出现了几十种产品，比较有代表性的包括VxWorks、QNX、Nucleus和Win CE等。

随着计算机技术的迅速发展和芯片制造工艺的不断进步，嵌入式系统的应用日益广泛：从民用的电视、手机等电子设备到军用的飞机、坦克等武器，到处都有嵌入式系统的身影。在嵌入式系统的应用开发中，采用嵌入式实时操作系统（简称RTOS）能够支持多任务，使得程序开发更加容易，便于维护，同时能够提高系统的稳定性和可靠性。这已逐渐成为嵌入式系统开发的一个发展方向。嵌入式实时操作系统典型产品如下：

1. VxWorks

VxWorks是美国Wind River System公司（简称风河公司，即WRS公司）推出的一个实时操作系统。风河公司组建于1981年，是一个专门从事实时操作系统开发与生产的软件公司，该公司在实时操作系统领域被世界公认为最具领导性。

VxWorks是专门为实时嵌入式系统设计开发的操作系统内核，为程序员提供了高效的实时多任务调度、中断管理，实时的系统资源以及实时的任务间通信。在各种CPU平台上提供了统一的编程接口和一致的运行特性，尽可能屏蔽不同CPU之间的底层差异。程序员可将更多的精力放在应用程序本身，而不必再去关心系统资源的管理。基于VxWorks操作系统的应用程序可以方便地移植到不同的CPU平台上。

VxWorks是一个运行在目标机上的高性能、可裁剪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，如卫星通信、军事武器、弹道制导和飞机导航等。在美国的F-16、FA-18战斗机，B-2隐形轰炸机，爱国者导弹，以及1997年4月在火星表面登陆的火星探路者上均使

用了 VxWorks。

VxWorks 是一种功能非常强大的操作系统，包括进程管理、存储管理、设备管理、文件系统管理、网络协议以及系统应用等几部分。VxWorks 只占用很小的存储空间，并可高度裁剪，从而保证了系统能以较高的效率运行。

2. pSOS

pSOS 是 ISI 公司研发的产品。该产品推出时间较早，因此比较成熟，可以支持多种处理器。它曾是国际上应用得最广泛的产品，主要应用领域是远程通信、航天、信息家电和工业控制。但 ISI 公司已被风河公司兼并，从 VxWorks 5.5 开始，已将 pSOS 的主要特点融入 VxWorks 中。

pSOS 是一个由标准软组件组成的、可裁剪的实时操作系统。其系统结构可分为内核层、系统服务层和用户层。

内核层负责任务的管理与调度、任务间通信、内存管理、实时时钟管理和中断服务；可以动态生成或删除任务、内存区、消息队列和信号灯等系统对象；实现了基于优先级的、选择可抢占的任务调度算法，并提供了可选的时间片轮转调度。pSOS 内核还提供了任务间通信机制及同步、互斥手段，如消息、信号灯、事件和异步信号等。pSOS 操作系统在内核层中将与具体硬件有关的操作放在一个模块中，对系统服务层以上屏蔽了具体的硬件特性，从而使得 pSOS 很方便地从支持 Intel 80x86 系列转换到支持 MC68xxx 系列，并且在系统服务层上为不同应用系统、不同用户提供标准的软组件（如 PNA＋和 PHILE＋等）。

pSOS 系统服务层包括 PNA＋、PRPC＋和 PHILE＋等组件。PNA＋实现了完整的基于“流”的 TCP/IP 协议集，并具有良好的实时性能，网络组件内中断屏蔽时间不长于内核模块中断屏蔽时间。PRPC＋提供了远程调用库，支持用户建立一个分布式应用系统；PHILE＋提供了文件系统管理和对块存储设备的管理；PREPC＋提供了标准的 C、C＋＋库，支持用户使用 C、C＋＋语言编写应用程序。由于 pSOS 内核屏蔽了具体的硬件特性，因此 pSOS 系统服务层的软组件是标准的，与硬件无关。这意味着 pSOS 各种版本，无论是对 80x86 系列还是 MC68xxx 系列，其系统服务层各组件是标准的、统一的，这就减少了软件维护工作，增强了软件可移植性。每个软组件都包含一系列的系统调用。就用户而言，这些系统调用就像一个个可重入的 C 函数，然而它们却是用户进入 pSOS 内核的惟一手段。

用户编写的应用程序是以任务的形式出现的。任务通过系统调用而进入 pSOS 内核，并为 pSOS 内核所管理和调度。pSOS 还为用户提供了一个集成开发环境（IDE）。pSOS_IDE 可驻留于 Unix 或 DOS 环境下，该集成开发环境具有 C 和 C＋＋优化编译器、CPU 和 pSOS 模拟仿真以及 DEBUG 功能。

3. VRTX

VRTX 是国际上最早推出的实时系统之一，比较成熟。其特点是内核紧凑，在模块化方

面与其他早期的实时操作系统相比有重大改善。

4. Nucleus

其特点是约95%的代码使用C语言编写，方便移植，同时，可提供 Web 支持、网络、图形和文件系统等模块。可全部提供源代码，用户只须通过动态链接便可进行任务级调试。

5. Lynx

Lynx 与 Unix 兼容，符合 POSIX 标准，是专为要求快速响应、复杂的实时应用设计的。它能为任何一个 Unix 平台上的应用，提供源代码级水平的兼容性。

6. Win CE

Microsoft 公司的嵌入式操作系统支持众多的硬件平台，其最主要特点是拥有与桌面型 Windows 系列一致的程序开发界面。因此，Windows 上开发的程序都能在 Win CE 上运行。但嵌入式操作系统追求高效、节省，而 Win CE 在这方面是笨拙的，它占用内存过大，应用程序也很庞大。

7. Linux

Linux 是一种提供源代码、开放式自由软件，具有嵌入式操作系统的很多特色，突出的优势是适用于多种 CPU 和多种硬件平台，性能稳定，裁剪性好，开发和使用都很容易。它是发展未来嵌入式设备的绝佳资源，具有广泛的应用前景。国内也有不少单位在 Linux 方面做了大量卓有成效的工作。

此外，后 PC 时代的众多产品（如手持设备等），并不要求强实时性。PalmOS 和 JavaOS 等应运而生，而 QNX 公司拥有的 QNX 则是一种限于 x86 平台，可提供集成化开发环境的实时操作系统。

1.5 嵌入式系统的应用领域

嵌入式技术具有非常广阔的应用前景，其应用包括以下诸多领域：

1. 工业控制

基于嵌入式芯片的工业自动化设备获得了迅速的发展，目前已有大量 8 位、16 位和 32 位嵌入式微控制器应用于工业过程控制、数字机床、电力系统、电网安全、电网设备监测和石油化工等领域中。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。就传统的工业控制产品而言，低端型采用的往往是 8 位单片机。但随着技术的发展，32 位和 64 位处理器逐渐成为工业控制设备的核心。相对于其他领域，机电产品可以说是嵌入式系统应用中最典型、

最广泛的领域之一。从最初的单片机到现在的工控机、SOC等，在各种机电产品中均占有巨大的市场。

工业设备是机电产品中最大的一类，在目前工业控制设备中，工控机的使用非常广泛。这些工控机一般采用工业级的处理器和各种设备，其中以x86的MPU最多。工业控制的要求往往较高，需要各种各样的设备接口；除了进行实时控制之外，还需将设备状态和传感器的信息等在显示屏上实时显示。这些要求8位单片机是无法满足的，以前多数使用16位处理器，随着处理器的迅速发展，32位和64位处理器逐渐替代了16位处理器，进一步提升了系统性能。采用PC104总线的系统，体积小，稳定可靠，受到了很多用户的青睐。不过这些工控机采用的往往是DOS或者Windows系统，虽然具有嵌入式的特点，却不能称为纯粹的嵌入式系统。另外，在工业控制器和设备控制器方面，则是各种嵌入式处理器的天下。这些控制器往往采用16位以上的微处理器，ARM、MIPS、68K系列处理器在控制器中占据核心地位。这些处理器提供了丰富的接口总线资源，可以通过它们实现数据采集、数据处理、通信以及显示（显示一般是连接LED或LCD）。

2. 交通管理

在车辆导航、流量控制、信息监测和汽车服务方面，嵌入式系统技术已获得广泛应用，内嵌GPS模块、GPRS模块的移动定位终端已成功应用于各种运输行业。目前GPS设备已从尖端产品进入了普及应用阶段，只需几千元，即可随时随地找到你的位置。

3. 信息家电

这是嵌入式系统最大的应用领域，冰箱、空调等的网络化和智能化将引领人们的生活步入一个崭新的空间。即使不在家里，也可通过电话线和网络进行远程控制。在这些设备中，嵌入式系统将大有用武之地。

传统的电视、电冰箱中也嵌有处理器，但这些处理器只是用于控制方面。而现在只有按钮、开关的电器显然已不能满足人们的日常需求，具有用户界面，能远程控制、智能管理的电器才是未来的发展趋势。IDG发布的统计数据表明，未来信息家电将会增长5～10倍。中国的传统家电厂商向信息家电过渡时，首先面临的挑战是核心操作系统软件开发工作。硬件方面，进行智能信息控制并不是很高的要求，目前绝大多数嵌入式处理器都可以满足硬件要求；真正的难点是如何使软件操作系统容量小，稳定性高且易于开发。Linux核心可以起到很好的桥梁作用，作为一个跨平台的操作系统，它可支持二三十种CPU，而目前已有众多家电业都开始做Linux的平台移植工作。1999年就登陆中国的微软“维纳斯”计划给出了数字家庭的概念，引导各大家电厂商纷纷投入到这场革命中来。虽然最终未能获得成功，却使信息家电深入人心。如今各大厂商仍在努力推出适用于新一代家电应用的芯片，Intel公司已专为信息家电业研发了名为xScale的ARM CPU系列。这一系列CPU本身不像x86 CPU那样须整合不同的芯片组，它在一颗芯片中可包括所需的各项功能，即硬件系统实现了SOC的概念。美国

网虎公司已将全球最小的嵌入式操作系统——QUARK 成功移植到 StrongARM 系列芯片上。这是首次把 Linux、图形界面和一些程序进行完整移植(QUARK 的内核只有 143 KB),它将为信息家电提供功能强大的核心操作系统。相信在不久的将来,数字智能家庭必将实现。

4. 家庭智能管理系统

水、电、煤气表的远程自动抄表,安全防火、防盗系统,其中嵌入的专用控制芯片将代替传统的人工检查,并且更准确,更安全,性能更高。目前在服务领域,如远程点菜器等已经体现了嵌入式系统的优势。

5. POS 网络及电子商务

公共交通无接触智能卡(CSC, Contactless Smart Card)发行系统、公共电话卡发行系统、自动售货机,以及各种智能 ATM 终端将全面走入人们的生活,到时手持一卡就可以行遍天下。

6. 环境工程与自然

对于水文资料实时监测、防洪体系及水土质量监测、堤坝安全监测、地震监测、实时气象信息监测以及水源和空气污染监测,在很多环境恶劣、地况复杂的地区,嵌入式系统将实现无人监测。

7. 智能玩具和机器人

嵌入式芯片的发展将使机器人在微型化、高智能方面的优势更加明显;同时会大幅度降低机器人的价格,使其在工业领域和服务领域获得更广泛的应用。

机器人技术的发展从来都是与嵌入式系统的发展紧密联系在一起的。最早的机器人技术是 20 世纪 50 年代 MIT 提出的数控技术,当时使用的还远未达到芯片水平,只是简单的"与非"门逻辑电路。之后由于处理器和智能控制理论发展缓慢,从 20 世纪 50 年代到 70 年代初期,机器人技术一直未能获得充分发展。20 世纪 70 年代中期之后,由于智能理论的发展和 MCU 的出现,机器人逐渐成为研究热点,并且获得了长足的发展。近来由于嵌入式处理器的高速发展,机器人从硬件到软件也呈现出新的发展趋势。火星车就是一个典型例子,这个价值 10 亿美元、技术高密集的移动机器人,采用的是美国风河公司的 VxWorks 嵌入式操作系统,可在不与地球联系的情况下自主工作。1997 年美国发射的"索杰纳"火星车带有机械手,可以采集火星上的各种地况,并且通过摄像头把火星上的图像发回地面指挥中心。这台火星车在火星上自主工作了 3 个月,充分体现了 VxWorks 系统的高可靠性。以 Sony 公司的机器狗为代表的智能机器宠物,可仅使用 8 位 AVR、51 单片机或者 16 位 DSP 来控制舵机,进行图像处理,就能制造出那些人见人爱的玩具,让人不得不惊叹嵌入式处理器功能的强大。32 位处理器和 Win CE 等 32 位嵌入式操作系统的盛行,使得操控一个机器人只需在手持 PDA 上获取远程机器人的信息,并通过无线通信控制机器人的运行。与传统的采用工控机相比,要轻巧、

便捷得多。随着嵌入式控制器越来越微型化、智能化,微型机器人和特种机器人等也将获得更多的发展机遇。

8. 军事国防领域

嵌入式计算机系统,最早出现在 20 世纪 60 年代武器控制中,后来用于军事指挥控制和通信系统,所以军事国防历来就是嵌入式系统的一个重要应用领域。现在各种武器控制(如火炮控制、导弹控制和智能炸弹制导引爆控制),以及坦克、舰艇、轰炸机,陆海空各种军用电子装备,雷达、电子对抗军事通信装备和野战指挥作战等各种专用设备上,都可以看到嵌入式系统的影子。应用嵌入式技术的武器在伊拉克战争中就曾被广泛使用。

1.6　嵌入式系统的未来发展趋势

信息时代、数字时代使得嵌入式产品获得了巨大的发展契机,为嵌入式市场展现了美好的前景,同时也对嵌入式生产厂商提出了新的要求。从中可以看出嵌入式系统的几大发展趋势:

① 嵌入式开发是一项系统工程,因此要求嵌入式系统厂商不仅提供嵌入式软硬件系统本身,而且还须提供强大的硬件开发工具和软件包支持。目前很多厂商已经充分考虑到这一点,在主推系统的同时,将开发环境也作为重点推广。例如:Samsung 公司在推广 ARM7、ARM9 芯片的同时,还提供开发板和板级支持包(BSP);而 Win CE 在主推系统时也提供 Embedded VC++作为开发工具;还有 VxWorks 的 Tornado 开发环境、DeltaOS 的 Lamda 编译环境等,都是这一趋势的典型体现。当然,这也是市场竞争的结果。

② 网络化、信息化的要求随着因特网技术的成熟、带宽的日益提高,使得电话、手机、冰箱和微波炉等设备功能不再单一,结构更加复杂。这就要求芯片设计厂商在芯片上集成更多功能。为了满足应用功能的升级,设计师们在硬件方面采用更强大的嵌入式处理器(如 32 位、64 位 RISC 芯片或 DSP)以增强处理能力;同时增加功能接口(如 USB),扩展总线类型(如 CAN BUS),以加强对多媒体和图形等的处理,从而形成了一系列功能强大,能满足不同需求的片上系统。软件方面采用实时多任务编程技术和交叉开发工具技术,来控制功能复杂性,简化应用程序设计,保障软件质量和缩短开发周期。

③ 网络互联成为必然趋势。未来的嵌入式设备为了适应网络发展的需要,必然要求硬件上提供各种网络通信接口。传统的单片机对网络支持不足,而新一代嵌入式处理器已开始内嵌网络接口。除了支持 TCP/IP 协议,有的还支持 IEEE1394、USB、CAN、BlueTooth 或 IrDA 通信接口中的一种或几种;同时也须提供相应的通信组网协议软件和物理层驱动软件。软件方面系统内核支持网络模块,甚至可在设备上嵌入网页浏览器,真正实现随时随地使用各种设备上网。

④ 精简系统内核、算法，降低功耗和软硬件成本。未来的嵌入式产品是软硬件紧密结合的设备。为了降低功耗和成本，需要设计者尽量精简系统内核，只保留与系统功能紧密相关的软硬件，利用最低的资源实现最适当的功能。这就要求设计者选用最佳的编程模型，并不断改进算法，优化编译器性能。因此，既需要软件人员有丰富的硬件知识，又需要发展先进的嵌入式软件技术，如 Java、Web 和 WAP 等。

⑤ 提供友好的多媒体人机界面。嵌入式设备能与用户亲密接触，最重要的因素就是它能提供非常友好的用户界面。图像界面和灵活的控制方式，使得人们感觉嵌入式设备就像一个老朋友。这就要求嵌入式软件设计者在图形界面和多媒体技术上痛下苦功。手写文字输入，语音拨号上网，电子邮件收发以及彩色图形、图像都会使使用者获得自由的感受。目前一些先进的 PDA 在显示屏幕上已实现汉字写入和短消息语音发布，但一般的嵌入式设备距离这个要求还有很长的路要走。

第2章 ARM 体系结构

ARM(Advanced RISC Machines),既可认为是一个公司的名字,也可认为是对一类微处理器的通称,还可认为是一种技术的名字。1991 年 ARM 公司成立于英国剑桥,主要出售芯片设计技术的授权。目前,采用 ARM 技术知识产权(IP)核的微处理器,即通常所说的 ARM 微处理器,已遍及工业控制、消费类电子、通信系统、网络系统、无线系统和军用系统等各类产品市场,基于 ARM 技术的微处理器应用约占据 32 位 RISC 微处理器 70%以上的市场份额,ARM 技术正逐步渗入我们生活的各个方面。ARM 公司是专门从事基于 RISC 技术芯片设计开发的公司,作为知识产权供应商,本身不直接从事芯片生产,通过转让设计许可,由合作公司生产各具特色的芯片。世界各大半导体生产商从 ARM 公司购买其设计的 ARM 微处理器核,根据各自不同的应用领域,加入适当的外围电路,从而形成自己的 ARM 微处理器芯片进入市场。目前,全世界有上百家大的半导体公司都使用 ARM 公司的授权,既使得 ARM 技术获得更多的第三方工具、制造和软件的支持,又使整个系统成本降低,从而使产品更容易进入市场并被消费者所接受,更具有竞争力。

2.1 ARM 微处理器的应用领域及特点

2.1.1 ARM 微处理器的应用领域

到目前为止,ARM 微处理器及技术的应用几乎已深入各个领域:

(1) 工业控制领域

作为 32 位的 RISC 架构,基于 ARM 核的微控制器芯片不但占据了高端微控制器市场的大部分市场份额,同时也逐渐向低端微控制器应用领域扩展。ARM 微控制器的低功耗、高性

价比，向传统的8位/16位微控制器提出了挑战。

(2) 无线通信领域

目前已有超过85%的无线通信设备采用了ARM技术，ARM以其高性能和低成本，在该领域的地位日益巩固。

(3) 网络应用

随着宽带技术的推广，采用ARM技术的ADSL芯片正逐步获得竞争优势。此外，ARM在语音及视频处理上进行了优化，并获得广泛支持，也对DSP的应用领域提出了挑战。

(4) 消费类电子产品

ARM技术在目前流行的数字音频播放器、数字机顶盒和游戏机中得到了广泛应用。

(5) 成像和安全产品

现在流行的数码相机和打印机中绝大部分采用了ARM技术；手机中的32位SIM智能卡也采用了ARM技术。

除此以外，ARM微处理器及技术还应用到许多不同的领域，并会在将来获得更加广泛的应用。

2.1.2 ARM微处理器的特点

采用RISC架构的ARM微处理器一般具有如下特点：

1. 采用RISC指令集，使得处理器体积小、功耗低、成本低、性能高

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

基于以上不合理性，1979年美国加州大学伯克利分校提出了RISC(Reduced Instruction Set Computer，精简指令集计算机)的概念。RISC并非只是简单地减少指令，而是把着眼点放在了如何使计算机的结构更加简单以及合理地提高运算速度上。RISC结构优先选取使用频率最高的简单指令，避免复杂指令；将指令长度固定，指令格式和寻址方式种类减少；以控制逻辑为主。到目前为止，RISC体系结构还没有严格的定义。一般认为，RISC体系结构应具有如下特点：

- 采用固定长度的指令格式，指令规整、简单，基本寻址方式有2～3种；
- 使用单周期指令，便于流水线操作执行；
- 大量使用寄存器，数据处理指令只对寄存器进行操作，只有加载/存储指令可以

访问存储器，以提高指令的执行效率。

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

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

由于 ARM 处理器采用了 RISC 指令集以及一些特殊的管理方式，使得 ARM 处理器只需较小的体积和较低的功耗，即可获得较高的执行效率。

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

ARM 微处理器支持两种指令集：ARM 指令集和 Thumb 指令集。其中：ARM 指令为 32 位长度；Thumb 指令为 16 位长度。Thumb 指令集为 ARM 指令集的功能子集；但与等价的 ARM 代码相比，可节省 30%～40%以上的存储空间，同时具备 32 位代码的所有优点。

从编程的角度看，ARM 微处理器的工作状态有两种：第一种为 ARM 状态，此时处理器执行 32 位的、字对齐的 ARM 指令；第二种为 Thumb 状态，此时处理器执行 16 位的、半字对齐的 Thumb 指令。

当 ARM 微处理器执行 32 位 ARM 指令时，工作在 ARM 状态；执行 16 位 Thumb 指令时，工作在 Thumb 状态。在程序的执行过程中，微处理器可随时在两种工作状态之间切换；并且，处理器工作状态的转变并不影响处理器的工作模式和相应寄存器中的内容。

状态切换方法：ARM 指令集和 Thumb 指令集均有切换处理器状态的指令，并可在两种工作状态之间切换；但 ARM 微处理器在开始执行代码时，应处于 ARM 状态，同时中断程序的入口代码也必须是 ARM 状态。

进入 Thumb 状态：当操作数寄存器的状态位(位 0)为 1 时，可采用执行 BX 指令的方法，使微处理器从 ARM 状态切换到 Thumb 状态。

进入 ARM 状态：当操作数寄存器的状态位为 0 时，执行 BX 指令时可使微处理器从 Thumb 状态切换到 ARM 状态。此外，在处理器进行异常处理时，把 PC 指针放入异常模式连接寄存器中，并从异常向量地址开始执行程序，也可使处理器切换到 ARM 状态。

3. 大量使用寄存器，多数数据操作都在寄存器中完成，指令执行速度更快

ARM 处理器共有 37 个寄存器，被分为若干个组。这些寄存器包括：

- 31 个通用寄存器，包括程序计数器(PC 指针)，均为 32 位寄存器；
- 6 个状态寄存器，用以标识 CPU 的工作状态及程序的运行状态，均为 32 位，目前只使

用了其中一部分。

同时，ARM 处理器又有 7 种不同的处理器模式，在每一种处理器模式下均有一组寄存器与之对应。即在任意一种处理器模式下，可访问的寄存器包括 15 个通用寄存器（R0～R14）、1～2个状态寄存器和程序计数器。在所有寄存器中，有些是在 7 种处理器模式下共用同一个物理寄存器，而有些则是在不同的处理器模式下有不同的物理寄存器。

4. 寻址方式灵活简单，执行效率高

寻址方式有常见的立即数寻址、寄存器寻址和基变址寻址等，同时还引入了多寄存器寻址的方式，使得在一条指令中可以同时加载多达 15 个寄存器的内容。ARM 指令系统的基变址寻址也很有特色，该指令可采用非常灵活的方式来修改变址寄存器的内容，以满足一些循环操作和堆栈操作等的需要。有关寻址方式的详细内容，请参见附录。

2.2 处理器模式

ARM 微处理器支持 7 种运行模式，分别为：

- 用户模式（usr）：ARM 处理器正常的程序执行状态；
- 快速中断模式（fiq）：用于高速数据传输或通道处理；
- 中断模式（irq）：用于通用的中断处理；
- 管理模式（svc）：操作系统使用的保护模式；
- 中止模式（abt）：当数据或指令预取异常时进入该模式，可用于虚拟存储及存储保护；
- 系统模式（sys）：运行具有特权的操作系统任务；
- 未定义指令模式（und）：当未定义的指令执行时进入该模式，可用于支持硬件协处理器的软件仿真。

ARM 微处理器的运行模式可以通过软件改变，也可通过外部中断或异常处理改变。大多数的应用程序运行在用户模式下。当处理器运行在用户模式下时，某些被保护的系统资源是不能被访问的。除用户模式以外，其余 6 种模式称为“非用户模式”或“特权模式”（Privileged Modes）。其中除去用户模式和系统模式以外的 5 种模式又称为“异常模式”（Exception Modes），常用于处理中断或异常，以及访问受保护的系统资源等情况。

2.3 寄存器组织

ARM 微处理器共有 37 个 32 位寄存器，其中 31 个为通用寄存器，其余 6 个为状态寄存

器。但这些寄存器不能被同时访问，具体哪些寄存器是可编程访问的，取决于微处理器的工作状态及具体的运行模式。但在任何时候，通用寄存器 R14～R0、程序计数器 PC 以及一个或两个状态寄存器都是可访问的。

2.3.1 ARM 状态下的寄存器组织

通用寄存器包括 R0～R15，可分为 3 类：

(1) 未分组寄存器 R0～R7

在所有运行模式下，未分组寄存器都指向同一个物理寄存器，它们未被系统用作特殊的用途。因此，在中断或异常处理进行运行模式转换时，由于不同的处理器运行模式均使用相同的物理寄存器，可能会造成寄存器中数据的破坏，这一点在进行程序设计时应注意。

(2) 分组寄存器 R8～R14

对于分组寄存器，每一次所访问的物理寄存器与处理器当前的运行模式有关。对 R8～R12 来说，每个寄存器对应两个不同的物理寄存器：当使用 fiq 模式时，访问寄存器 R8_fiq～R12_fiq；当使用除 fiq 模式以外的其他模式时，则访问寄存器 R8_usr～R12_usr。对 R13 和 R14 来说，每个寄存器对应 6 个不同的物理寄存器，其中一个是用户模式与系统模式共用，另外 5 个对应于其他 5 种不同的运行模式。

采用以下记号来区分不同的物理寄存器：

R13_<mode>

R14_<mode>

其中，mode 为以下几种模式之一：usr、fiq、irq、svc、abt、sys 和 und。

寄存器 R13 在 ARM 指令中常用作堆栈指针，但这只是一种习惯用法，用户也可使用其他寄存器作为堆栈指针；而在 Thumb 指令集中，某些指令强制性地要求使用 R13 作为堆栈指针。由于处理器的每种运行模式均有自己独立的物理寄存器 R13，在用户应用程序的初始化部分，一般都要初始化每种模式下的 R13，使其指向该运行模式的栈空间。这样，当运行的程序进入异常模式时，可将需要保护的寄存器放入 R13 所指向的堆栈；而当程序从异常模式返回时，则从对应的堆栈中恢复。采用这种方式可以保证异常发生后程序正常执行。R14 也称为“子程序连接寄存器”(Subroutine Link Register)或“连接寄存器”(LR)。当执行 BL 子程序调用指令时，R14 中存储 R15(程序计数器 PC)的备份；其他情况下，R14 用作通用寄存器。同样，当发生中断或异常时，对应的分组寄存器 R14_svc、R14_irq、R14_fiq、R14_abt 和 R14_und 用来保存 R15 的返回值。

在每一种运行模式下，都可用 R14 保存子程序的返回地址。当用 BL 或 BLX 指令调用子程序时，将 PC 的当前值拷贝给 R14；执行完子程序后，又将 R14 的值拷贝回 PC，即可完成子程序的调用并返回。以上描述可用以下指令完成：

① 执行以下任意一条指令：

```
MOV PC,LR
BX LR
```

② 在子程序入口处使用以下指令将 R14 存入堆栈：

```
STMFD SP!,{<Regs>,LR}
```

对应地，使用以下指令可完成子程序返回：

```
LDMFD SP!,{<Regs>,PC}
```

此外，R14 也可作为通用寄存器。

(3) 程序计数器 PC(R15)

寄存器 R15 用作程序计数器(PC)。在 ARM 状态下，位[1:0]为 0，位[31:2]用于保存 PC；在 Thumb 状态下，位[0]为 0，位[31:1]用于保存 PC。虽然可以用作通用寄存器，但有些指令在使用 R15 时有一些特殊限制；若不注意，则执行的结果将不可预料。在 ARM 状态下，PC 的 0 和 1 位是 0；在 Thumb 状态下，PC 的 0 位是 0。

由于 ARM 体系结构采用了多级流水线技术，因此就 ARM 指令集而言，PC 总是指向当前指令的下两条指令的地址，即 PC 的值为当前指令的地址值加 8 字节。

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

每一种运行模式下又都有一个专用的物理状态寄存器，称为 SPSR(Saved Program Status Register，备份的程序状态寄存器)。当异常发生时，SPSR 用于保存 CPSR 的当前值；从异常退出时，则可由 SPSR 来恢复 CPSR。由于用户模式和系统模式不属于异常模式，没有 SPSR，因此在这两种模式下访问 SPSR，结果是未知的。

在 ARM 状态下，任一时刻可访问上述 16 个通用寄存器和 1～2 个状态寄存器；在非用户模式下，则可访问特定模式分组寄存器。表 2-1 说明了在每种运行模式下，哪些寄存器是可以访问的，其中斜体表示该模式下的寄存器在物理上是独立的。

表 2-1　ARM 状态下的寄存器组织

usr	sys	svc	abt	und	irq	fig
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3

续表 2-1

usr	sys	svc	abt	und	irq	fig
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	*R8_fiq*
R9	R9	R9	R9	R9	R9	*R9_fiq*
R10	R10	R10	R10	R10	R10	*R10_fiq*
R11	R11	R11	R11	R11	R11	*R11_fiq*
R12	R12	R12	R12	R12	R12	*R12_fiq*
R13	R13	*R13_svc*	*R13_abt*	*R13_und*	*R13_irq*	*R13_fiq*
R14	R14	*R14_svc*	*R14_abt*	*R14_und*	*R14_irq*	*R14_fiq*
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	*SPSR_abt*	*SPSR_und*	*SPSR_irq*	*SPSR_fiq*

2.3.2 Thumb 状态下的寄存器组织

Thumb 状态下的寄存器集是 ARM 状态下寄存器集的一个子集，程序可直接访问 8 个通用寄存器(R7～R0)、程序计数器(PC)、堆栈指针(SP)、连接寄存器(LR)和 CPSR。同时，在每一种特权模式下都有一组 SP、LR 和 SPSR。Thumb 状态下的寄存器组织如表 2-2 所列。

Thumb 状态下的寄存器组织与 ARM 状态下的寄存器组织有如下对应关系：

- Thumb 状态下和 ARM 状态下的 R0～R7 是相同的；
- Thumb 状态下和 ARM 状态下的 CPSR 和所有 SPSR 是相同的；
- Thumb 状态下的 SP 对应于 ARM 状态下的 R13；
- Thumb 状态下的 LR 对应于 ARM 状态下的 R14；
- Thumb 状态下的程序计数器对应于 ARM 状态下的 R15。

表 2-2 Thumb 状态下的寄存器组织

usr	sys	svc	abt	und	irq	fig
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R13	R13	*R13_svc*	*R13_abt*	*R13_und*	*R13_irq*	*R13_fiq*
R14	R14	*R14_svc*	*R14_abt*	*R14_und*	*R14_irq*	*R14_fiq*
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	*SPSR_abt*	*SPSR_und*	*SPSR_irq*	*SPSR_fiq*

访问 Thumb 状态下的高位寄存器(Hi-registers)：在 Thumb 状态下，高位寄存器 R8～R15 并不是标准寄存器集的一部分，但可使用汇编语言受限制地访问这些寄存器，将其用作快速的暂存器。使用带特殊变量的 MOV 指令，数据可在低位寄存器和高位寄存器之间进行传送；高位寄存器的值可以使用 CMP 和 ADD 指令进行比较，或加上低位寄存器中的值。

2.3.3 程序状态寄存器

ARM 体系结构包含一个当前程序状态寄存器(CPSR)和 5 个备份的程序状态寄存器(SPSR)。备份的程序状态寄存器用来进行异常处理，其功能包括：

- 保存 ALU 中当前的操作信息；
- 控制允许和禁止中断；
- 设置处理器的运行模式。

程序状态寄存器每一位的意义如表 2-3 所列。

表 2-3 程序状态寄存器格式

条件码标志				…	控制位								
N	Z	C	V	…	I	F	T	M4	M3	M2	M2	M1	M0

(1) 条件码标志

N、Z、C、V 均为条件码标志(Condition Code Flags)位。它们的内容可被算术或逻辑运算的结果所改变,也可决定某条指令是否被执行。

在 ARM 状态下,绝大多数指令都是有条件执行的;而在 Thumb 状态下,仅有分支指令是有条件执行的。

条件码标志各位的具体含义如表 2-4 所列。

表 2-4 条件码标志的具体含义

标志位	含 义
N	当用两个补码表示的带符号数进行运算时,N=1 表示运算的结果为负数;N=0 表示运算的结果为正数或零
Z	Z=1,表示运算的结果为零;Z=0,表示运算的结果非零
C	有 4 种方法设置 C 的值: 加法运算(包括比较指令 CMN):当运算结果产生进位时(无符号数溢出),C=1;否则,C=0。 减法运算(包括比较指令 CMP):当运算产生借位时(无符号数溢出),C=0;否则,C=1。 对于包含移位操作的非加/减运算指令,C 为移出值的最后一位; 对于其他非加/减运算指令,C 值通常不改变
V	有 2 种方法设置 V 的值: 对于加/减法运算指令,当操作数和运算结果为二进制补码表示的带符号数时,V=1 表示符号位溢出; 对于其他非加/减运算指令,C 值通常不改变
Q	在 ARM v5 及以上版本的 E 系列处理器中,用 Q 标志位指示增强的 DSP 运算指令是否发生了溢出;在其他版本的处理器中,Q 标志位无定义

(2) 控制位

PSR 的低 8 位(包括 I、F、T 和 M[4:0])称为"控制位",当发生异常时这些位可被改变。如果处理器运行在特权模式下,则这些位也可由程序修改。

(3) 中断禁止位 I、F

I=1,禁止 IRQ 中断;F=1,禁止 FIQ 中断。

(4) T 标志位

该位反映处理器的运行状态。

对于 ARM 体系结构 v5 及以上版本的 T 系列处理器，当该位为 1 时，程序运行于 Thumb 状态；否则，运行于 ARM 状态。

对于 ARM 体系结构 v5 及以上版本的非 T 系列处理器，当该位为 1 时，执行下一条指令将引起未定义的指令异常；否则，运行于 ARM 状态。

(5) 运行模式位 M[4:0]

M0、M1、M2、M3 和 M4 是模式位。这些位决定了处理器的运行模式，具体含义如表 2-5 所列。

表 2-5 运行模式位 M[4:0]的具体含义

M[4:0]	处理器模式	可访问的寄存器
0b10000	用户模式	PC,CPSR,R0～R14
0b10001	快速中断模式	PC,CPSR, SPSR_fiq,R14_fiq～R8_fiq, R7～R0
0b10010	中断模式	PC,CPSR, SPSR_irq,R14_irq,R13_irq,R12～R0
0b10011	管理模式	PC,CPSR, SPSR_svc,R14_svc,R13_svc,R12～R0
0b10111	中止模式	PC,CPSR, SPSR_abt,R14_abt,R13_abt, R12～R0
0b11011	未定义指令模式	PC,CPSR, SPSR_und,R14_und,R13_und, R12～R0
0b11111	系统模式	PC,CPSR(ARM v4 及以上版本), R14～R0

由表 2-5 可知，并非所有运行模式位的组合都是有效的，其他组合结果会导致处理器进入一种不可恢复的状态。

(6) 保留位

PSR 中的其余位为保留位。当改变 PSR 中的条件码标志位或控制位时，保留位不能被改变；在程序中也不要使用保留位来存储数据。保留位将用于 ARM 版本的扩展。

2.4 异 常

正常的程序执行流程发生暂时的停止，称为“异常”(Exceptions)。例如处理一个外部的中断请求。在处理异常之前，当前处理器的状态必须保留。这样当异常处理完成之后，当前程序可以继续执行。处理器允许多个异常同时发生，它们将会按固定的优先级进行处理。ARM 体系结构中的异常，与 8 位/16 位体系结构的中断有很多相似之处，但并不完全等同。

2.4.1 ARM 体系结构所支持的异常类型

ARM 体系结构所支持的异常及具体含义如下：

复位：当处理器的复位电平有效时，产生复位异常，程序跳转到复位异常处执行。

未定义指令：当 ARM 处理器或协处理器遇到不能处理的指令时，产生未定义指令异常；可使用该异常机制进行软件仿真。

软件中断：该异常由于执行 SWI 指令而产生，可用于用户模式下的程序调用特权操作指令；可使用该异常机制实现系统功能调用。

指令预取中止：若处理器预取指令的地址不存在，或该地址不允许当前指令访问，则存储器会向处理器发出中止信号；但只有当预取的指令被执行时，才会产生指令预取中止异常。

数据中止：当处理器数据访问指令的地址不存在，或该地址不允许当前指令访问时，产生数据中止异常。

中断：当处理器的外部中断请求引脚有效，且 CPSR 中的 I 位为 0 时，产生 IRQ 异常。系统的外设可通过该异常请求中断服务。

快速中断：当处理器的快速中断请求引脚有效，且 CPSR 中的 F 位为 0 时，产生 FIQ 异常。

2.4.2 对异常的响应

当一个异常出现后，ARM 微处理器会执行以下几步操作：

① 将下一条指令的地址存入相应的连接寄存器 LR，以便程序在处理完异常后返回时能从正确的位置重新开始执行。若异常是从 ARM 状态进入的，则 LR 寄存器中保存的是下一条指令的地址（当前 PC 值＋4 或＋8，与异常的类型有关）；若异常是从 Thumb 状态进入的，则在 LR 寄存器中保存当前 PC 值的偏移量，这样，异常处理程序就无须确定异常是从何种状态进入的。例如：在软件中断异常 SWI 中，指令“MOV PC，R14_svc”总是返回到下一条指令，不管 SWI 是在 ARM 状态下执行，还是在 Thumb 状态下。

② 将 CPSR 拷贝到相应的 SPSR 中。

③ 根据异常类型，强制设置 CPSR 的运行模式位。

④ 强制 PC 从相关的异常向量地址取下一条指令执行，从而跳转到相应的异常处理程序处。

如果异常发生时，处理器处于 Thumb 状态，则当异常向量地址加载到 PC 时，处理器会自动切换到 ARM 状态。

2.4.3 从异常返回

异常处理完毕之后，ARM 微处理器会执行以下几步操作从异常返回：

① 连接寄存器 LR 的值减去相应的偏移量后送到 PC 中；

② 将 SPSR 拷贝回 CPSR 中；

③ 若在进入异常处理时设置了中断禁止位，则在此清除。

可以认为应用程序总是从复位异常处理程序开始执行的，因此复位异常处理程序无须返回。

2.4.4 各类异常的具体描述

1. 快速中断

快速中断(fiq)异常是为了支持数据传输或通道处理而设计的。在 ARM 状态下，系统有足够的私有寄存器，从而避免了对寄存器保存的需求，并减小了系统上下文切换的开销。若将 CPSR 的 F 位置 1，则会禁止 fiq 中断；若将 CPSR 的 F 位清 0，则处理器会在指令执行时检查 fiq 的输入。注意：只有在特权模式下才能改变 F 位的状态。

可由外部通过对处理器上的 nFIQ 引脚输入相应的电平来产生 fiq。不管是在 ARM 状态下还是在 Thumb 状态下进入 fiq 模式，fiq 处理程序均可执行以下指令从 fiq 模式返回：

```
SUBS PC,R14_fiq ,#4
```

该指令将寄存器 R14_fiq 的值减去 4 后，拷贝到程序计数器 PC 中，从而实现了从异常处理程序中的返回。同时，将 SPSR_mode 寄存器里的内容拷贝到当前程序状态寄存器 CPSR 中。

2. 中 断

中断(irq)异常属于正常的中断请求，可通过对处理器的 nIRQ 引脚输入低电平产生。irq 的优先级低于 fiq，当程序进入 fiq 异常时，irq 可能被屏蔽。

若将 CPSR 的 I 位置 1，则会禁止 irq 中断；若将 CPSR 的 I 位清 0，则处理器会在指令执行完之前检查 irq 的输入。同样，也只有在特权模式下才能改变 I 位的状态。

不管是在 ARM 状态下还是在 Thumb 状态下进入 irq 模式，返回代码均与 fiq 模式类似：

```
SUBS PC , R14_irq , #4
```

该指令将寄存器 R14_irq 的值减去 4 后，拷贝到程序计数器 PC 中，从而实现从异常处理程序中的返回。同时，将 SPSR_mode 寄存器里的内容拷贝到当前程序状态寄存器

CPSR 中。

3. 中　止

产生中止(abt)异常意味着对存储器的访问失败。ARM 微处理器在存储器访问周期内检查是否发生中止异常。

中止异常包括两种类型：

- 指令预取中止：发生在指令预取时；
- 数据中止：发生在数据访问时。

当指令预取访问存储器失败时，存储器系统向 ARM 处理器发出存储器中止(Abort)信号，预取的指令被记为无效。但只有当处理器试图执行无效指令时，指令预取中止异常才会发生。如果指令未被执行，例如在指令流水线中发生了跳转，则预取指令中止不会发生。若数据中止发生，则系统的响应与指令的类型有关，其返回代码也与具体的中止原因有关。

4. 软件中断

软件中断(swi)用于进入管理模式，常用于请求实现特定的管理功能。无论是在 ARM 状态下还是在 Thumb 状态下，软件中断处理程序都执行以下指令从 swi 模式返回：

```
MOVS PC , R14_svc
```

该指令恢复 PC(从 R14_svc)和 CPSR(从 SPSR_svc)的值，并返回到 swi 的下一条指令。

5. 未定义指令

当 ARM 处理器遇到不能处理的指令时，会产生未定义指令(und)异常。采用这种机制，可通过软件仿真扩展 ARM 或 Thumb 指令集。

在仿真未定义指令后，无论是在 ARM 状态下还是在 Thumb 状态下，处理器都执行以下程序返回：

```
MOVS PC, R14_und
```

该指令恢复 PC(从 R14_und)和 CPSR(从 SPSR_und)的值，并返回到未定义指令后的下一条指令。

2.4.5 异常向量

ARM 体系结构的处理器在进行异常处理时，都会根据异常的类型，从位于地址 0 处的一块区域获取异常处理的入口。各异常的入口偏移见表 2-6。

当系统运行时，异常可能会随时发生。为保证 ARM 处理器在发生异常时不至于处于未

知状态,在应用程序的设计中,首先要捕获异处理这些异常。采用的方式是在异常向量表中的相应位置放置一条跳转指令,跳转到异常处理程序。当ARM处理器发生异常时,程序计数器PC会被强制设置为对应的异常向量,从而跳转到异常处理程序;当异常处理完成以后,返回到主程序继续执行。

表2-6 异常向量地址

地址	异常	进入模式
0x00000000	复位	管理模式
0x00000004	未定义指令	未定义指令模式
0x00000008	软件中断	管理模式
0x0000000C	中止(指令预取)	中止模式
0x00000010	中止(数据)	中止模式
0x00000014	保留	保留
0x00000018	中断	中断模式
0x0000001C	快速中断	快速中断模式

第 3 章

Tornado II 集成开发环境的建立与使用

3.1 Tornado II 集成开发环境简介

Tornado II 集成开发环境是嵌入式领域里新一代的开发环境，是进行 VxWorks 嵌入式实时应用系统开发的完整平台，包括从项目工程的创建和管理到 BSP 的移植，以及从应用系统的设计到系统的调试、性能分析等。Tornado II 作为交叉开发环境运行在主机上，给嵌入式系统开发人员提供了一个不受目标机资源限制的超级开发和调试环境。Tornado II 开发系统结构如图 3-1 所示。

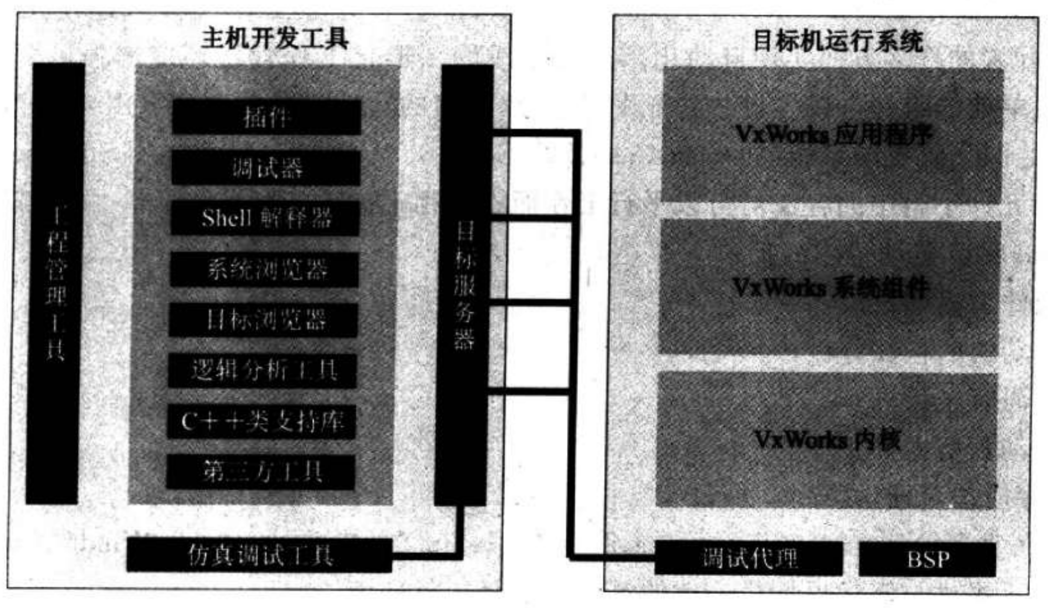


图 3-1 Tornado II 开发系统结构图

Tornado II 开发系统包含3个高度集成的部分：

- 运行在宿主机和目标机上的交叉开发工具和实用程序；
- 运行在目标机上的高性能、可裁剪的实时操作系统 VxWorks；
- 连接宿主机和目标机的多种通信方式，如以太网、串口线、ICE或ROM仿真器等。

Tornado II 软件工具包中有很多独立的工具，其中核心工具是各个 Tornado II 软件工具包都具有的。核心工具主要包括以下几种：

(1) 图形化交叉调试器

图形化交叉调试器(CrossWind Debugger/WDB)是一个远程的源代码集成调试器，支持任务级和系统级调试，支持混合源代码和汇编代码显示，支持多目标机同时调试。

这个高性能的调试器采用图形化的用户界面，使用非常方便，开发者可以成组地观察表达式的计算窗口；可在调试器的图形用户界面中迅速改变变量、寄存器和局部变量的值；可为不同组的元素设定新数值。通过信息规整和分类的方法，该调试器可有效地提供信息；还可提供开发者熟悉的 GDB 调试器引擎，这种调试引擎提供了命令行接口和图形接口，具有很强的灵活性。

开发者可以通过调试器在目标系统上产生和调试新任务，也可用调试器连接和调试已经运行的任务。

(2) 工程配置工具

工程配置工具(Project Facility/Configuration)是一个强有力的图形化工具，可对 VxWorks 操作系统及其组件进行自动配置。自动的依赖性分析、代码容量计算和自动裁剪向导，大大缩短了开发周期。

工程工具简化了 VxWorks 应用程序的组织、配置和建立工作；同时，还使工程管理和 VxWorks配置的许多方面实现自动化。这种集成的图形化工程管理环境增强了开发小组的专业技术：单独的组件可以各自独立开发，然后由小组的其他成员共享和重用。由于建立了与现在流行的源代码控制系统(如 ClearCase、SCCS、RCS、PVCS 和 MS Visual SourceSafe 等)的联系，所以允许小组中的各成员可以平行工作而不互相干扰。其特点如下：

- makefile 自动生成维护；
- 软件工程维护；
- 自动的依赖性分析；
- 代码容量计算；
- 自动裁剪向导。

(3) 集成仿真器

集成仿真器(Integrated Simulator，简称为 VxSim)支持 CrossWind、WindView 和 Browser 等工具，提供与真实目标机一致的调试和仿真运行环境。

VxSim 作为核心工具包含在各软件包中，因此允许开发者在没有 BSP、操作系统配置和

目标机硬件的情况下，使用 Tornado II 迅速开始开发工作，非常方便。

作为核心工具，包含在各软件包中的 VxSim 都是限制版本。也就是说，它并不支持网络仿真。如果想获得全部功能的 VxSim，则可根据所购买软件包的条件从 WindPower 可选工具中进行选择。

(4) 动态诊断分析工具

动态诊断分析工具 WindView 是一个图形化的动态诊断和分析工具，主要是向开发者提供目标机硬件上实际运行的应用程序的一些详细情况。这种系统级的诊断分析工具可与 VxSim一起使用。

嵌入式系统开发者经常因为无法知道系统的执行情况和软件的时间特性而感到失望。这种全功能版本的 WindView 提供了一种非常方便的手段，让开发者明确地掌握 VxWorks 应用程序的详细动态行为，并且可以使用图形化的方式显示任务、中断和系统对象相互作用的复杂关系。开发者还可使用另外一种类型的 WindView，用于监视目标硬件系统行为，通过该工具可以直观地了解系统硬件的一些特征。

(5) 编译环境

Tornado II 提供交叉编译器、汇编器、链接器，以及一些辅助工具(如 Make 工具、objcopyarm 和 objdumparm 等)，以支持 C 和 C++语言的程序编译。交叉编译器进行了许多优化，允许开发者能够迅速产生高效而简洁的代码。系统可以独立使用以下两类编译环境：

- Diab C/C++ Compiler：惟一获得 Motorola 公司白金大奖的嵌入式编译器；
- GNU C/C++ Compiler：应用最广泛的编译器。

Tornado II 提供对 C++的支持，包括：异常事件处理、标准模板库(STL，Standard Template Library)、运行类型识别(RTTI，Run-Time Type Identification)，以及静态构造器、析构器和 C++调试器。

(6) 主机目标机连接配置器

主机目标机连接配置器(Target Server Config)允许开发者轻松地设置和配置一定的开发环境，也提供对开发环境的管理功能。

(7) 目标机系统状态浏览器

目标机系统状态浏览器 Browser 是 Tornado II Shell 的一个图形化组件。Browser 的主窗口提供目标系统的全面状态总结，同时允许开发者监视独立的目标对象，如任务、信号灯、消息队列、内存分区、定时器、模块、变量和堆栈等。这些显示可根据开发者的选择，进行周期性或条件性更新。

(8) 命令行执行工具

命令行执行工具 WindSh 是 Tornado II 所独有的功能强大的命令行解释器，可以直接解释并执行 C 语言表达式，调用目标机上的 C 函数，访问系统符号表中登记的变量；同时还可直接执行 TCL 语言。

(9) 图形化核心配置工具

图形化核心配置工具(WindConfig)使用图形向导方式智能化地自动配置 VxWorks 内核及其组件参数。

以上介绍的是 Tornado II 集成的一些工具模块，这些工具会涉及整个系统软件开发的不同时期，可为不同工作提供一种非常有效而且方便的手段。后续章节中将分别说明这些工具的详细使用方法。

3.2 Tornado II 集成开发环境安装

Tornado II 集成开发环境支持多种主机平台，下面以目前使用最为广泛的 Windows 系统为例，说明该集成开发环境的安装和使用。安装一般包含两部分：Tornado II 集成开发环境和 BSP 支持包。安装过程可参照以下步骤：

① 打开安装光盘上的 setup. exe 程序，可以看到如图 3-2 所示的欢迎界面。直接单击 Next 按钮。

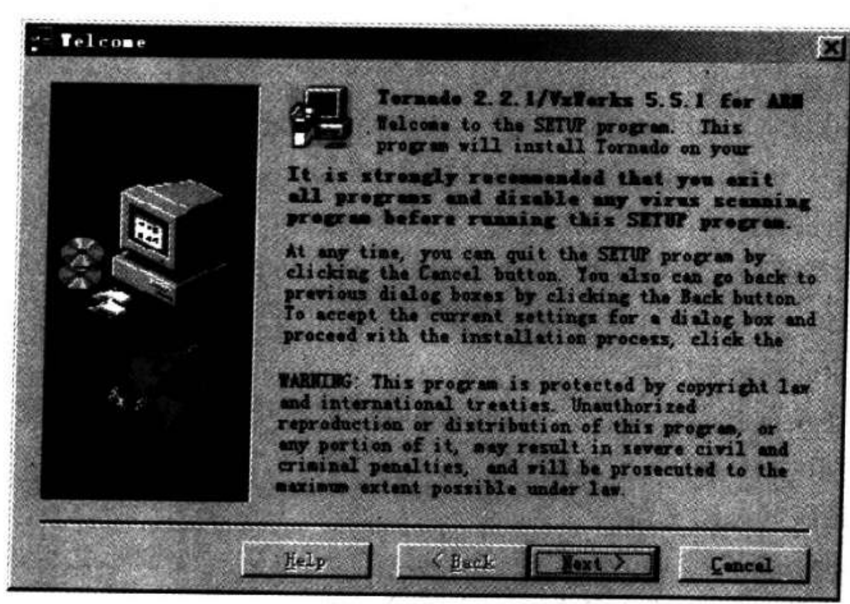


图 3-2 安装时的欢迎界面

② 安装程序会显示许可协议，如图 3-3 所示。认真阅读其中各项条款，若接受许可协议，则单击 Accept 和 Next 按钮。

③ 如图 3-4 所示，在相应位置分别填写用户名(Name)、公司名(Company)以及由风河公司提供的安装序号(Install)，并单击 Next 按钮，弹出如图 3-5 所示的界面。

④ 在图 3-5 中，可以使用系统默认的安装模式 Install Product。

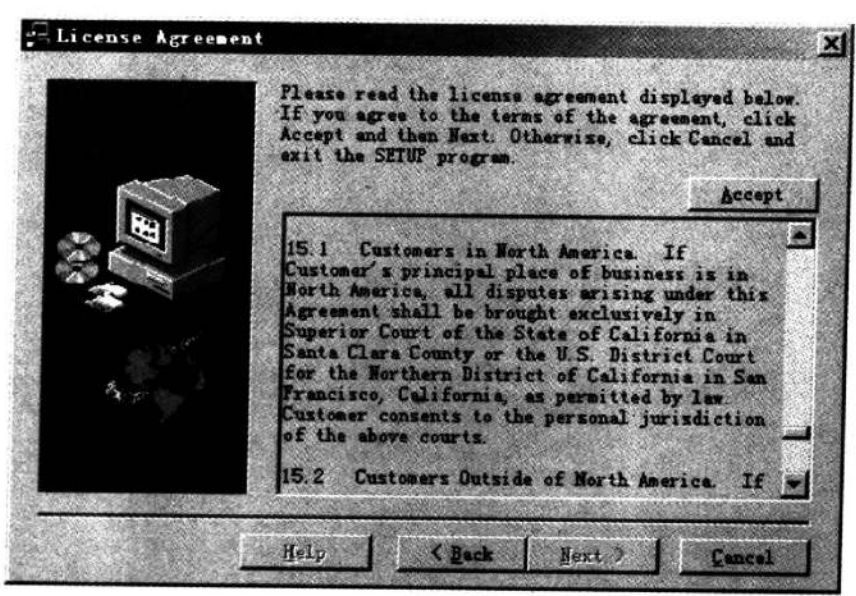


图 3-3　WindRiver 的许可协议

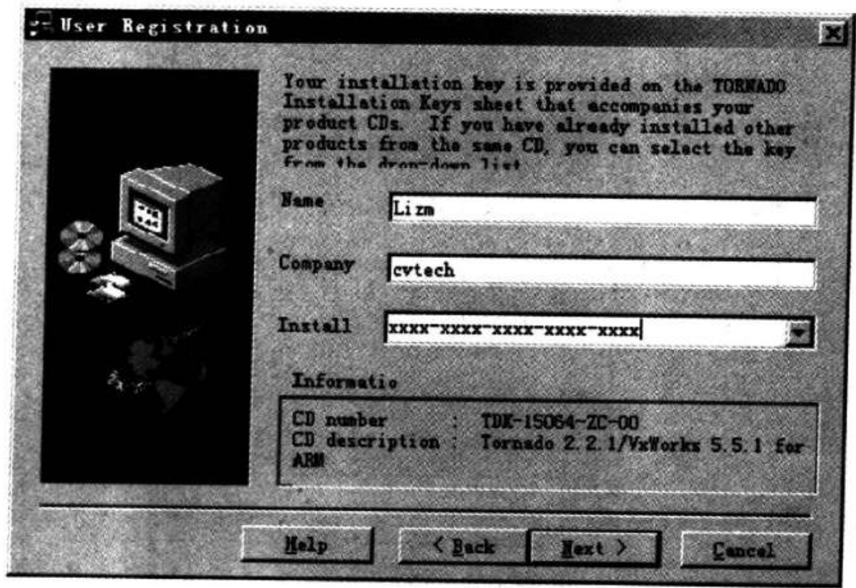


图 3-4　输入用户名、公司名和安装序号

⑤ 输入一个安装路径，如图 3-6 所示，可单击 Browser 按钮来选择安装路径，或者直接输入安装路径。

⑥ 选择安装组件。如果硬盘有足够的空间，则可不做任何修改；当然也可去掉一些组件来节省空间。其中编译器必须至少选择一个，而一些目标库则可进行调整。例如：图 3-7中，对于 S3C2410，可选择去掉 arm1020 和 arm720_t 等系统库，也可去掉一个编译器。

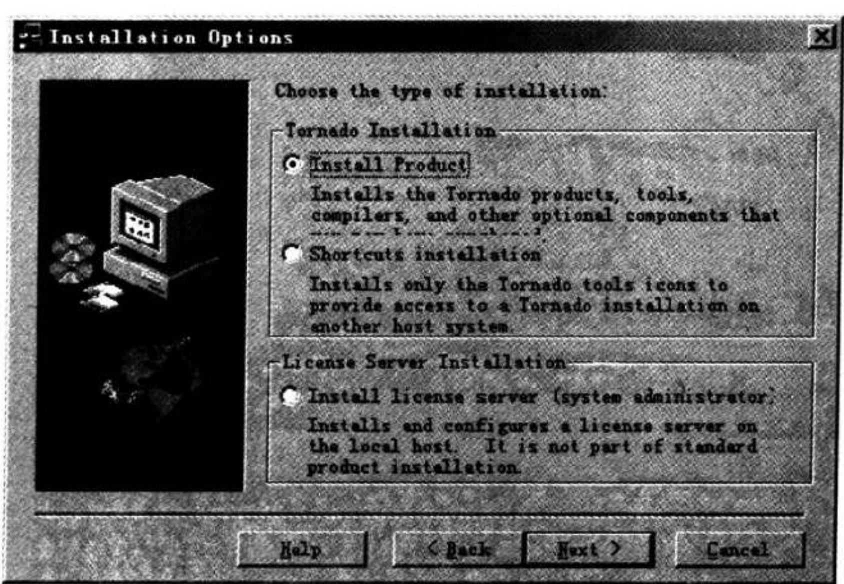


图 3-5 选择安装模式

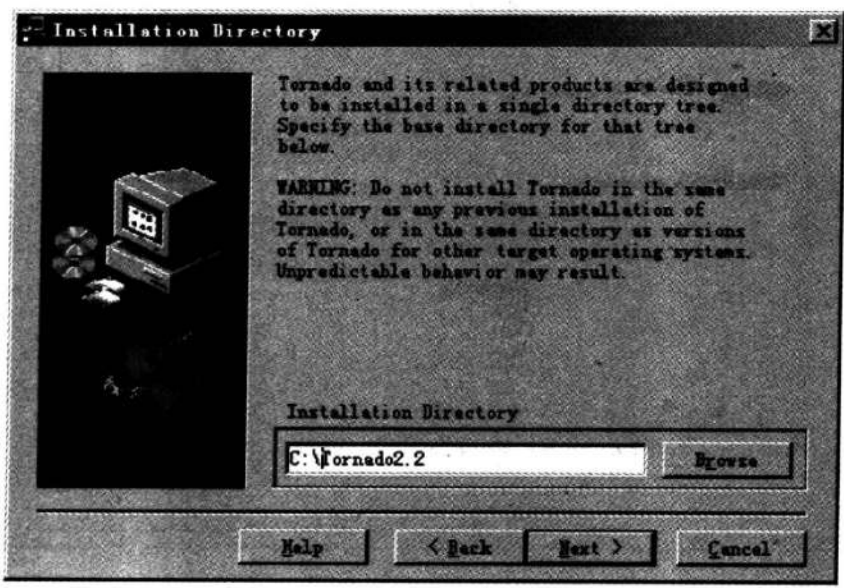


图 3-6 选择安装路径

⑦ 在后续几步中可使用系统默认设置，直接单击 Next 按钮，如图 3-8 所示。

⑧ 安装时的进度提示如图 3-9 所示。

⑨ 文件拷贝完毕后，系统会给出安装报告，如图 3-10 所示。

⑩ 最后需要配置 License 信息，可选择手工安装，并将由风河公司提供的 License 文件拷贝为＜InstallDir＞/license/WRSLicense.lic，如图 3-11 和图 3-12 所示。

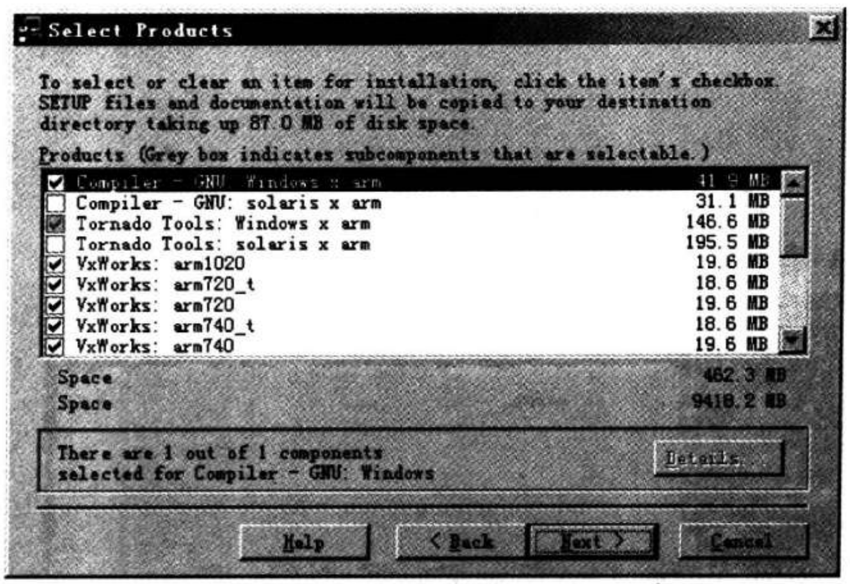


图 3-7 选择安装组件

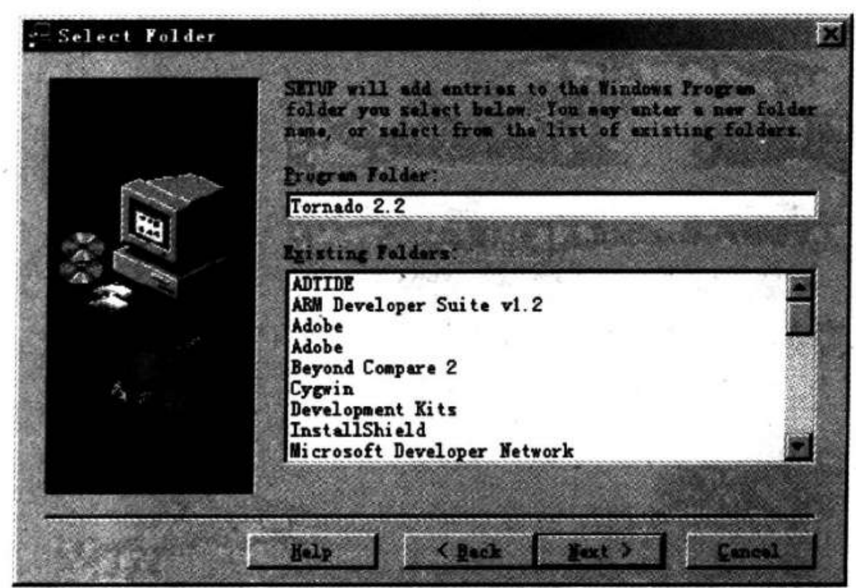


图 3-8 设置安装的程序组名

⑪ 安装 BSP 及驱动程序。运行 BSP 安装光盘中的 setup. exe 程序，阅读许可协议，按照系统的提示填写安装序列号，选择安装路径即可。

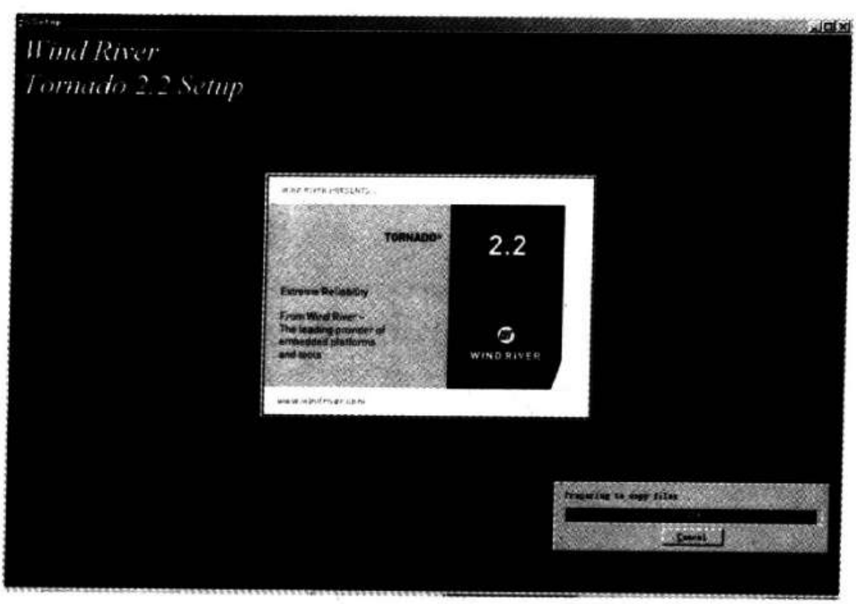


图 3-9　安装时的进度提示

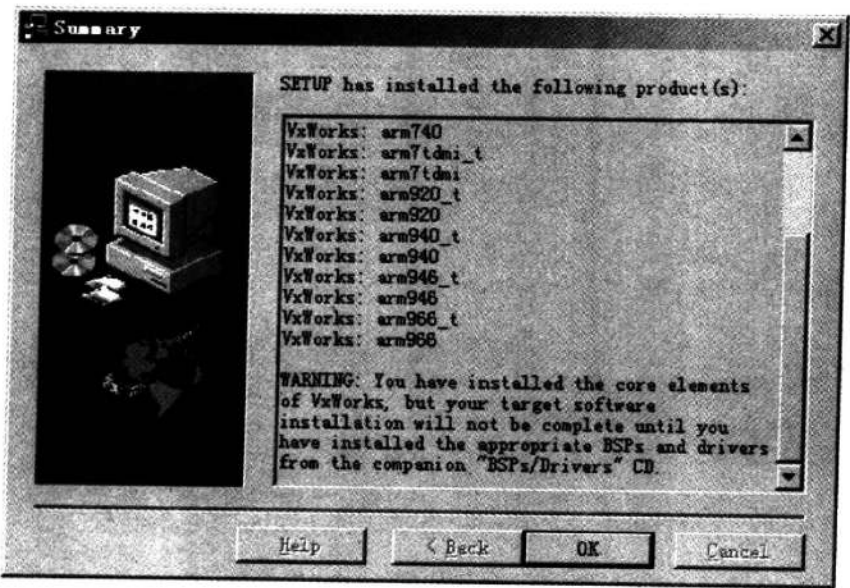


图 3-10　安装报告

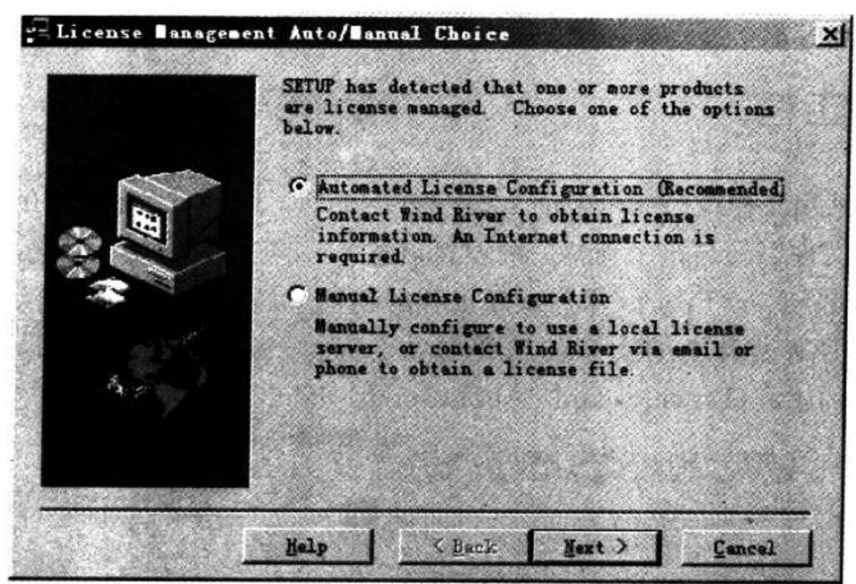


图 3-11　配置 License

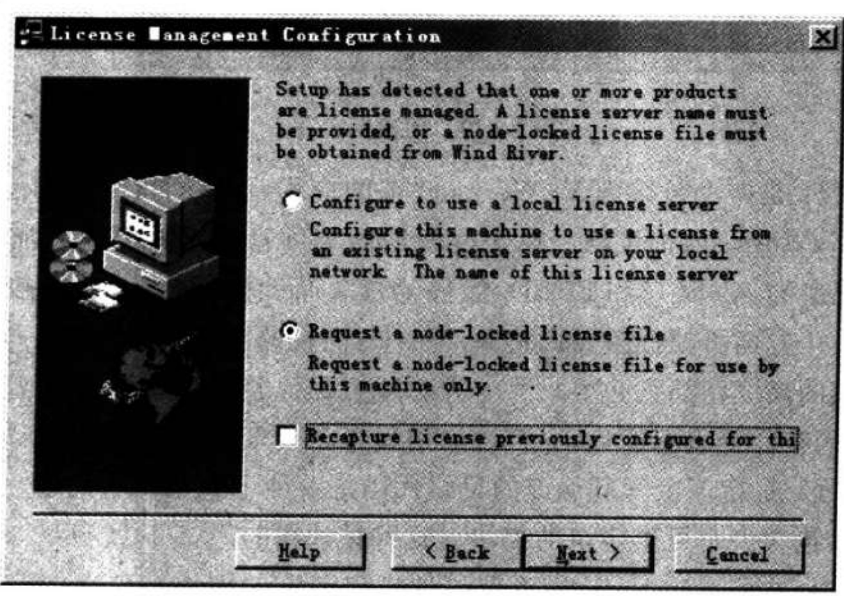


图 3-12　设置为使用 License 文件

3.3 使用 Tornado II 创建新的工程

3.3.1 新建工程

Tornado II 采用工作区(Workspace)和工程(Project)的模式来管理应用程序。一个工程可以包含源代码文件以及与该工程相关的设置信息等;而工作区就好比一个容器,可以容纳多个工程。选择 File→New Project 来启动新工程的创建过程,创建新工程的对话框如图 3-13 所示。创建工程时有两种不同类型的工程可供选择:可启动的工程(A Bootable VxWorks Image)和可下载的工程(Downloadable Application Modules)。

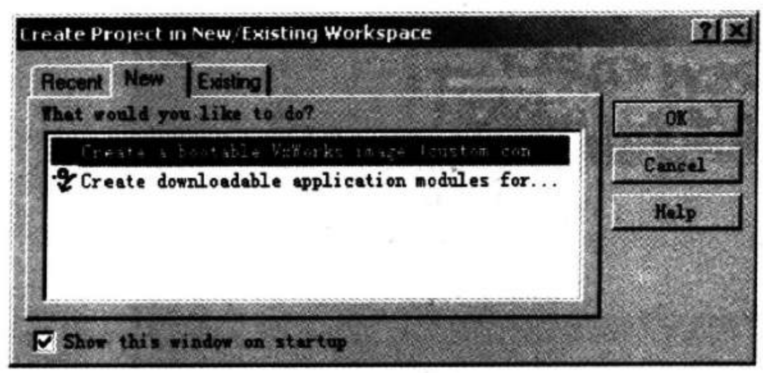


图 3-13 创建新工程

可启动的工程:是一个链接到 VxWorks 映像的应用程序。该工程中不仅包括应用程序的代码,还包括 VxWorks 内核代码。在工程管理器中可以非常方便地添加或删除一些内核组件(如操作系统中的 ANSI C 和 POSIX 组件等),从而调整 VxWorks 内核的特征。通过组件的添加或删除,一方面可根据需要调整 VxWorks 内核的代码大小(嵌入式系统的资源都非常有限,保持内核的高度精简很必要);另一方面还可保证整个系统运行所需的组件都包含在所生成的目标代码中。当目标机启动后,该应用程序会被自动加载且运行。

可下载的工程:是指该工程所编译的目标代码是一种可重定位的代码。在一个可下载的工程中,一般可以包含多个可重定位的模块。这些模块可被动态下载到一个运行有 VxWorks 内核的目标系统中,并可通过 Shell 或者调试器来启动这些模块。动态加载是 Tornado II 的一个特点。通过这项功能,可将目标模块装载到一个正在运行的系统中,与重新构建和链接整个操作系统相比,所需的时间更短,从而大大缩短了调试周期。所以在开发一个项目时,往往是先将内核移植到目标系统中;然后采用这种动态加载的手段来调试具体的应用程序;最后在应用程序调试无误后,将这些模块与 VxWorks 内核代码集成,最终形成一个可启动的代码。

对于可启动的工程，在创建时首先须正确指定所参考的 BSP，如图 3－14 所示，通过"文件浏览"对话框选择位于 D:\arm-tech\vxworks\S3C2410X\bsp_lcd 下的一个 BSP。

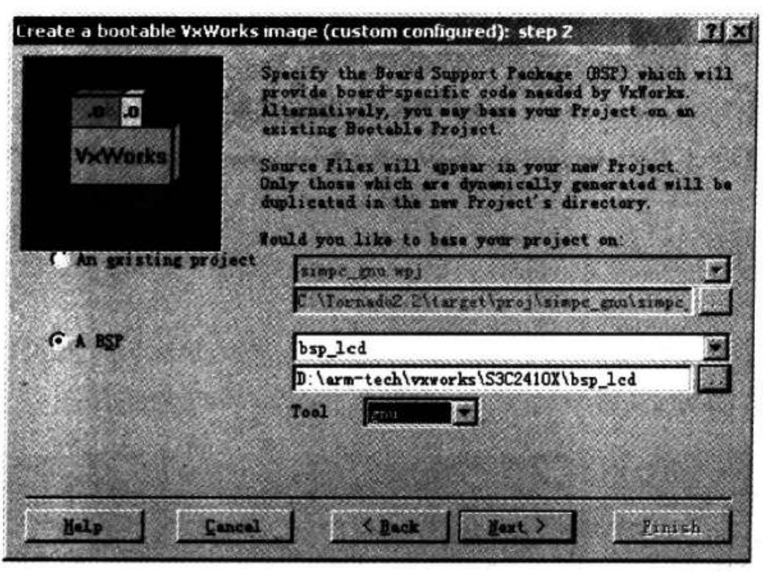


图 3－14　指定参考的 BSP

在创建工程时还须选择工具链，也就是编译器。对 ARM 这类处理器来说，可选择的工具链有两个：GNU 和 DIAB(在安装 Tornado II 时须进行选择，有的系统为了节省空间，可能省掉某个编译器，在这里就会只有某些相应的选项)。每个工具链又可分为两个版本，分别是 gnu 和 gnube、diab 和 diabbe，如图 3－15 所示。这与 ARM 处理器的存储模式有关，即大端存储(Big Endian)和小端存储(Little Endian)。

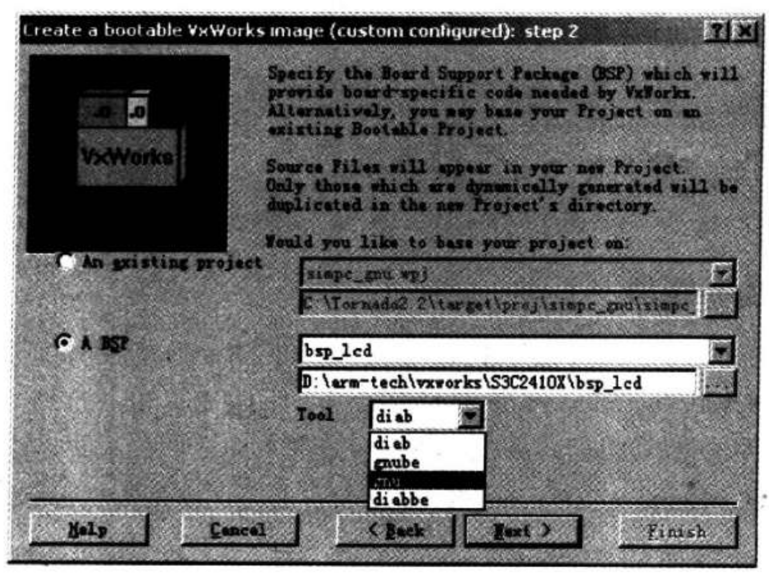
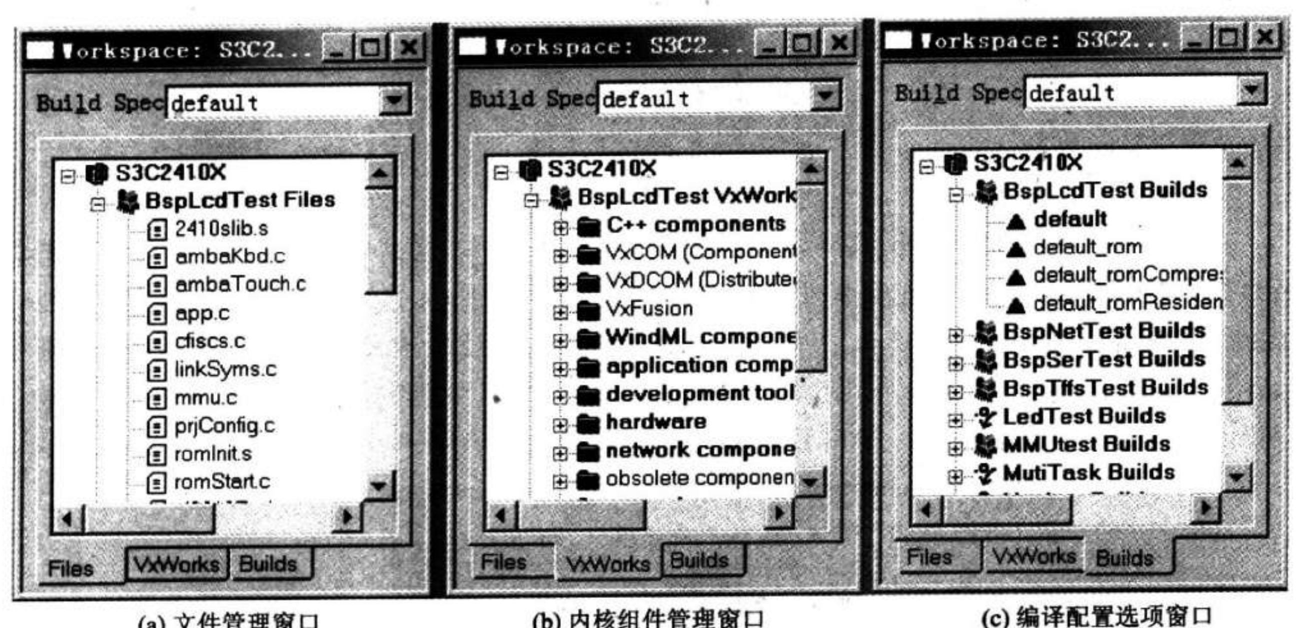


图 3－15　选择编译工具

如果使用后面有 be 后缀的编译器，则编译出的代码应该运行在大端存储模式下；而对于不带后缀的编译器，编译出的代码应该运行在小端存储模式下。大多数处理器在默认情况下都采用小端存储模式，系统会自动选择 gnu 编译器；如果目标系统采用的是大端存储模式，则需相应地将编译器修改为 gnube。当然，在创建工程时也可选择 diab 编译器。它们编译的目标代码并没有大的区别，只是在代码的语法上，特别是汇编代码的语法上，有一些小小的区别；但绝大部分代码都可在不作任何修改的情况下，使用这两个不同的编译器进行正确的编译。

3.3.2 工程管理

Tornado II 集成开发环境提供了一个非常方便且高效的图形化工程管理工具，如图 3-16 所示。该窗口包括 3 部分：文件管理窗口、内核组件管理窗口和编译配置选项窗口。



(a) 文件管理窗口　　(b) 内核组件管理窗口　　(c) 编译配置选项窗口

图 3-16　Tornado II 工程管理器

1. 文件的添加和删除

文件管理窗口主要用于一个工程所包含的代码文件的维护，包括添加和删除。添加文件可以通过选择 Project→Add/Includes→File，或者在相应工程名上右击并选择 Add Files(如图 3-17 所示)，然后通过"文件选择"(Add Source File to)对话框来完成。如果想删除工程中的某个文件，则可通过选择 Project→Remove/Exclude→File，或者右击该文件名并选择 Remove 来完成。

2. 内核组件的添加和删除

工程管理器的第二个子窗口是内核组件的管理窗口，如图 3-18 所示。该窗口所列出的

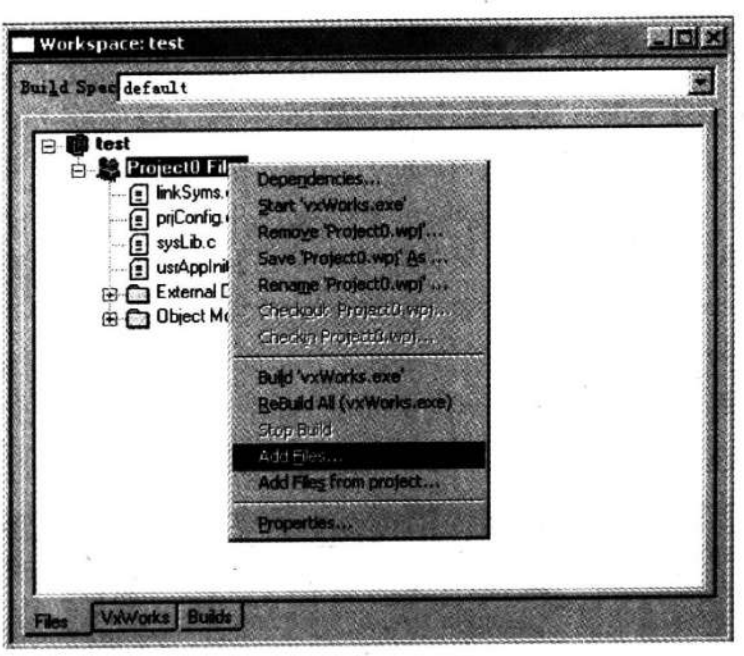


图 3-17　向工程添加文件

工程都是一些可启动的工程，即这些工程所产生的目标代码中都包含 VxWorks 内核代码，而那些可下载的工程则不会在该窗口中列出。VxWorks 内核由很多模块(前面所说的组件)组成。在工程管理器中可以很方便地调整这些组件的一些属性，添加或者删除某些组件。在调整后，系统会自动执行组件之间的依赖检查。对于一些有冲突的组件，会以红色字体提示出来。

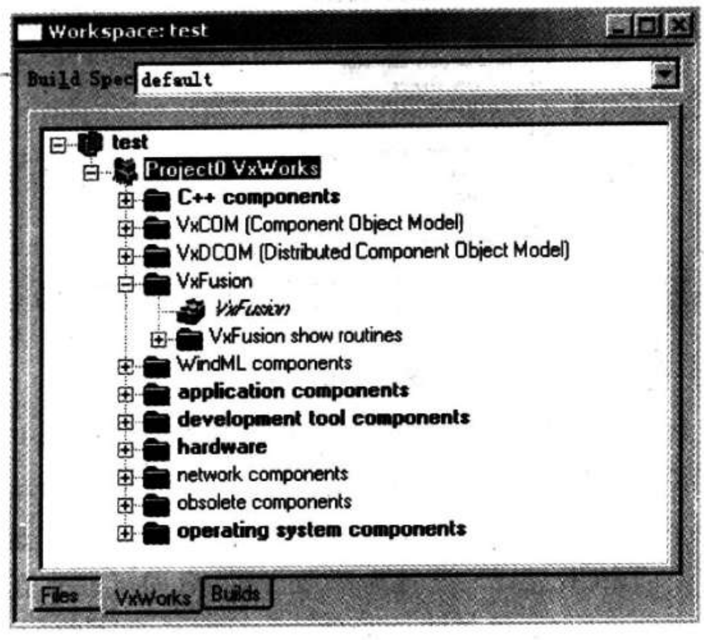


图 3-18　内核组件管理窗口

VxWorks 内核组件管理采用一个树形结构，展开树的节点即可看到与之相关组件的具体信息。如果某个组件被包含在该工程中，则该组件会以黑色粗体的形式反映出来；如果某个组件在所使用的开发平台上未被正确安装，则会以斜体的方式显示。另外，还可通过菜单或者与鼠标右键相关联的上下文菜单来修改所有组件的特性。从图 3－18 中可以看出，该工程包含了 C＋＋组件、application 组件、development tool 组件、hardware 组件和 operating system 组件，而且 VxFusion 组件未被安装。如果进一步展开该树形结构，例如选择 operating system components→kernel components，则可看到该组件的具体情况，如图 3－19 所示。从图中可以看出，该工程包含了 kernel components 的绝大部分功能（除了 read the bootline 外）。对于一个具体的应用，若无需 BSP hardware initialization；可在 BSP hardware initialization 节点处右击，并选择 Exclude 'BSP hardware initialization'，确认后即可。

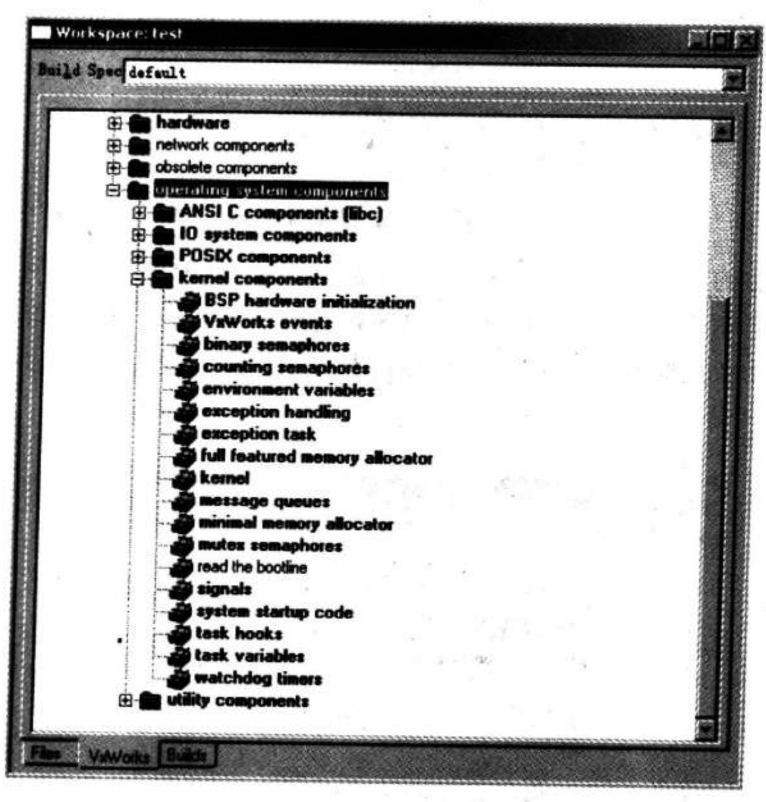


图 3－19 内核组件 kernel 的详细列表

同样，要修改某个组件的属性，也可在相应的组件上右击并选择 Properties。例如：要调整内核中 ISR_STACK_SIZE 的大小，就可以选择 operating system components→kernel components，并在 kernel 处右击并选择 Properties，在出现的对话框中选择 Params 选项卡（如图 3－20 所示），将 ISR_STACK_SIZE 设置为新值，确认后即可。要查看某个文件夹下组件的具体情况，可单击该文件夹图标，组件管理器会显示该文件夹下的所有组件及其包含情况，如图 3－19所示。

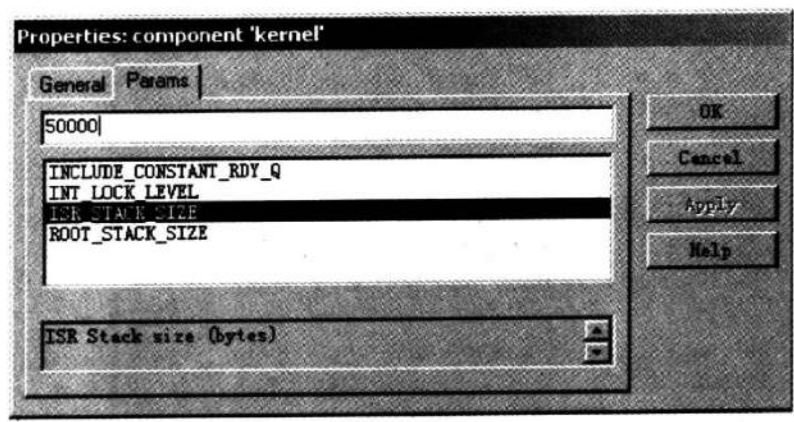


图 3-20 内核组件参数的修改

3. 编译器的配置

一个工程可有多种配置，无论使用 GNU 工具链还是 DIAB 工具链，其与编译器有关的设置都被分为 7 部分：General、Rules、Macros、C/C++ compiler、Link Order、assemble 和 linker。在工程管理器窗口（如图 3-16 所示）的 Builds 选项卡中双击配置名，可以查看该配置的一些具体情况。可启动的工程在新建时会默认生成一个缺省配置，下面以可启动的工程为例介绍一下编译器的配置。

(1) General

一个只读的选项卡，主要包括工具链和 BSP 的说明，这两方面的内容在新建工程时就已确定，如图 3-21 所示。

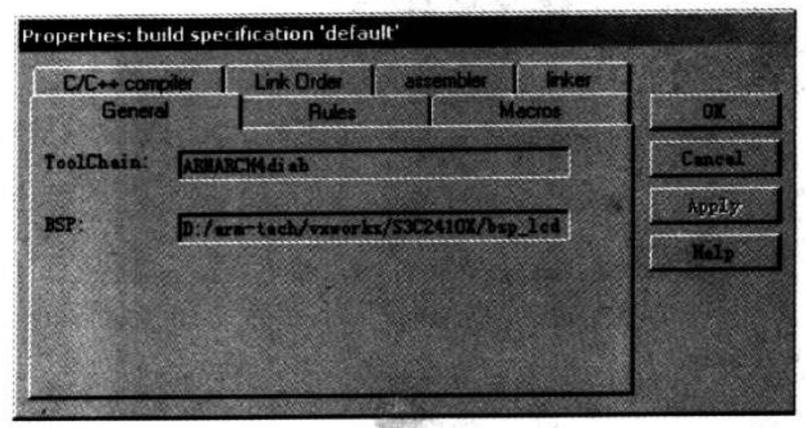


图 3-21 编译器配置信息

(2) Rules

该选项卡显示了 makefile 的规则，可在该对话框中新建、编辑或删除自定义的 makefile 规则，如图 3-22 所示。

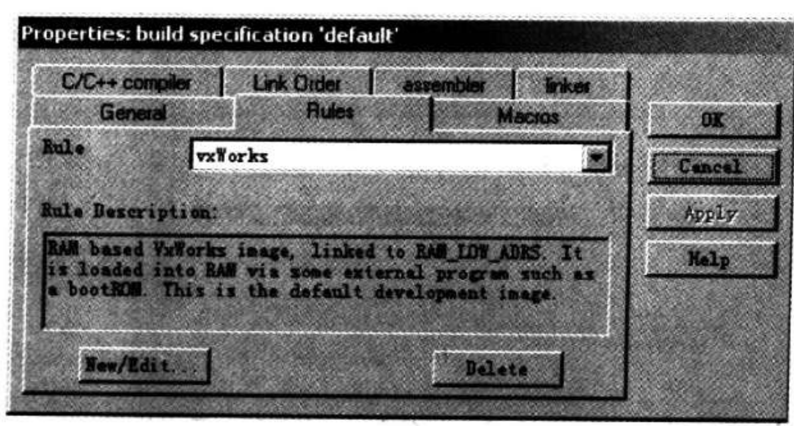


图 3-22 编译规则

(3) Macros

在工程的 makefile 文件中会定义大量的宏，这些宏有的可直接用在源代码中；有的可在编译器的解释下控制程序的编译，即条件编译。要修改这些宏定义的值，一方面可以直接修改 makefile 文件；另一方面可以借助编译配置管理器。可使用编译器配置的 Macros 选项卡来重新定义或修改这些已存在于 makefile 文件中的宏，程序清单 3-1 列出了一小段位于 makefile 文件中的宏定义。

程序清单 3-1 makefile 文件中的宏定义

```
CC_ARCH_SPEC          = -tARMLS:VxWorks55
CFLAGS                = -g -tARMLS:VxWorks55 -W:c:,...
CFLAGS_AS             = -g -tARMLS:VxWorks55 -W:c:,...
CFLAGS_AS_PROJECT     = -g -tARMLS:VxWorks55 -W:c:,...
CFLAGS_PROJECT        = -g -tARMLS:VxWorks55 -W:c:,...
CPP                   = dcc -E -Xpreprocessor-lineno-off
EXTRA_MODULES         =
LD                    = dld
LDFLAGS               = -tARMLS:VxWorks55 -X -N -Xgenerate-paddr
LD_LINK_PATH          = -L$(WIND_BASE)/target/lib/arm/ARMARCH4/diab -L$(WIND_BASE)/target/
lib/arm/ARMARCH4/common
LD_PARTIAL            = dld -tARMLS:VxWorks55 -X -r
LD_PARTIAL_FLAGS      = -tARMLS:VxWorks55 -X -r
LIBS                  = $(VX_OS_LIBS)
NM                    = nmarm -g
OPTION_DEFINE_MACRO = -D
```

```
OPTION_DEPEND        = -Xmake -dependency -w
OPTION_GENERATE_DEPENDENCY_FILE = -MD
OPTION_INCLUDE_DIR= -I
OPTION_LANG_C        = -xc
OPTION_UNDEFINE_MACRO = -U
RAM_HIGH_ADRS        = 33800000 # RAM text/data address
RAM_LOW_ADRS         = 30010000 # RAM text/data address
SIZE                 = sizearm
TOOL_FAMILY          = diab
VMA_START            = 0x$ (ROM_TEXT_ADRS)
POST_BUILD_RULE      =
```

打开编译器配置的 Macros 选项卡可看到上述宏定义，如图 3-23 所示的 makefile 文件中的宏定义 CC_ARCH_SPEC。通过 Macros 下拉列表框来选择不同的宏定义，同时也可在 Value 文本框中输入新值来修改所选定的宏，或者通过单击 Delete 按钮来删除当前宏。

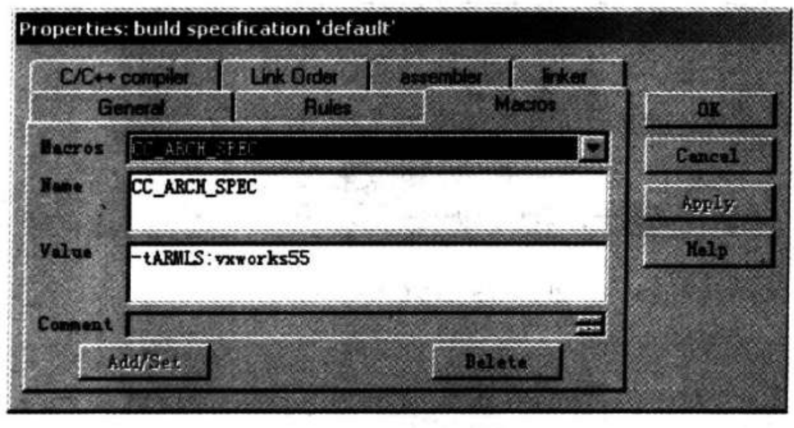


图 3-23　编译配置的宏定义修改

如果想添加一个自定义的宏，则可首先在 Name 文本框中输入宏的名称，此时位于对话框左下角的 Add/Set 按钮将变成使能状态；然后在 Value 文本框中输入宏的值；最后单击 Add/Set 按钮即可。例如：图 3-24 中添加了一个名为 MY_MACRO 的宏，其值设置为 0x1000000。

(4) C/C++ compiler

该选项卡主要显示 C 和 C++代码的一些编译参数、优化级别以及头文件的搜索路径等，如图 3-25 所示。可以在编辑框中修改这些编译参数，单击 Include paths 按钮可以添加或删除头文件的搜索路径，也可调整所包含的头文件搜索路径的搜索顺序。在编辑框中修改头文件搜索路径时要注意一点：要使用正斜杠“/”作为路径的符号，而不能使用 Windows 的反斜杠“\”。

图 3-24 添加宏定义

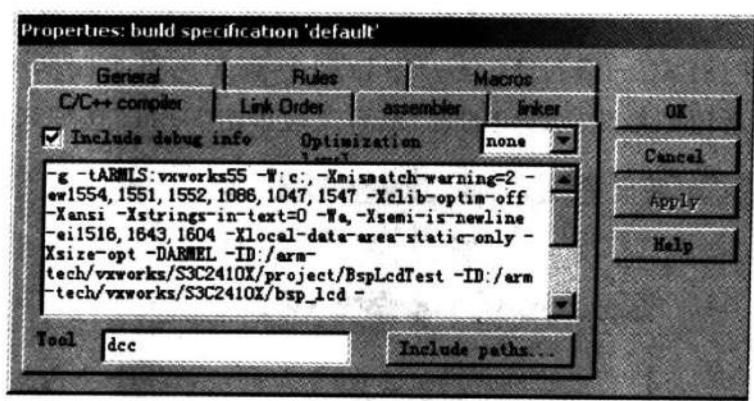


图 3-25 C/C++编译选项

(5) assemble

如图 3-26 所示，针对汇编代码，在设置时单独有 assemble 选项卡与之对应；同样，也可在该选项卡的编辑框中编辑、修改这些设置。另外，还可单击 Include paths 按钮来修改所包含头文件的搜索路径。

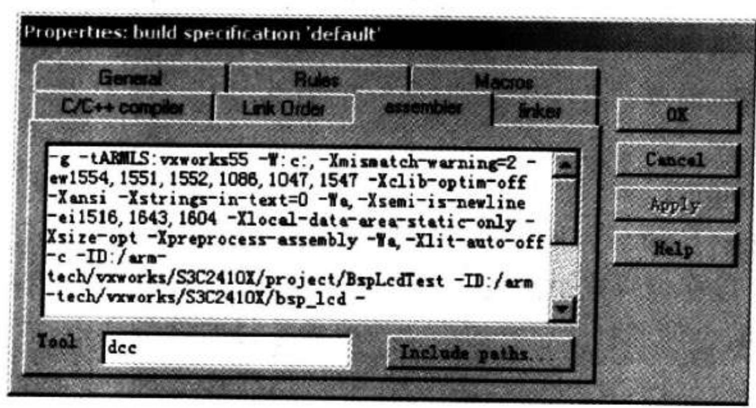


图 3-26 汇编代码编译选项

(6) Link Order

该选项卡用于设置该工程所包含的一些模块的链接顺序，如图 3－27 所示。可以选中某个模块，然后单击 Down 或 Up 按钮来调整该模块的链接顺序。对于一个可启动的工程，链接时第一个文件是不能随意指定的，该文件的第一条指令必须是该工程的入口；而对于其他模块的顺序，则无太多要求，只须保证它们之间相互调用时满足 BL 或 B 指令的跳转范围限度（±32 MB）。由于裁剪后的 VxWorks 内核大多都在 1 MB 左右，所以一般无须考虑其他模块的链接顺序。

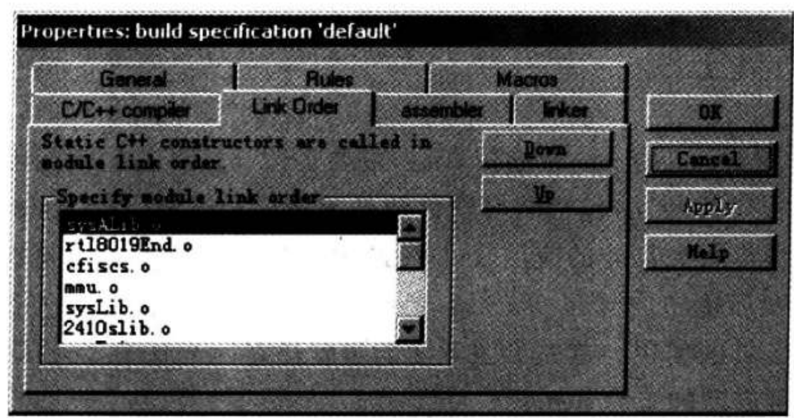


图 3－27 链接顺序

(7) linker

如果想在一个工程中使用某个现成的库，则可在该选项卡的编辑框中输入该库的名称，如图 3－28 所示。但我们推荐把库添加到前面所提到的 Macros 选项卡的宏 LIBS、EXTRA_MODULES 或 PRJ_LIBS 定义的列表中。

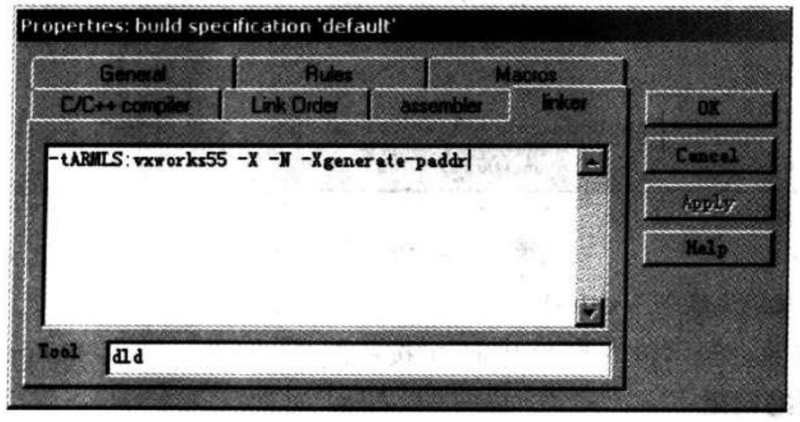


图 3－28 链接选项

Tornado II 所使用的编译器配置非常灵活，可在相关的编辑框中输入大量的编译选项。有关编译器的详细信息，可阅读《GNU ToolKit User's Guide》或《Diab C/C＋＋ Compiler

User's Guide》。

3.4 Tornado II 的调试工具

Tornado II 集成了很多工具，其中编辑器、工程管理器以及编译器已在前面做了介绍，下面介绍 Tornado II 的其他辅助工具。

3.4.1 集成仿真工具

集成仿真工具(VxSim)是一个原型仿真器，主要是使开发者在无实际目标硬件的情况下，先进行原型级应用程序的开发，包括网络设计和基于多处理器的设计。VxSim 还可使开发者在开发周期中较早地进行大部分的应用软件测试，从而能够以较小的代价纠正错误。

由于在集成开发环境中内置了 VxSim，因此可在没有硬件条件时使用该模拟器来学习 Tornado II 各辅助工具的使用。首先参照下面的步骤建立一个基于 VxSim 的工程：

(1) 新建一个可启动的工程 demo

参照图 3-29 选择 Create a bootable VxWorks image 创建一个新的可启动工程，并在图 3-30 中输入工程的名称和位置等信息。

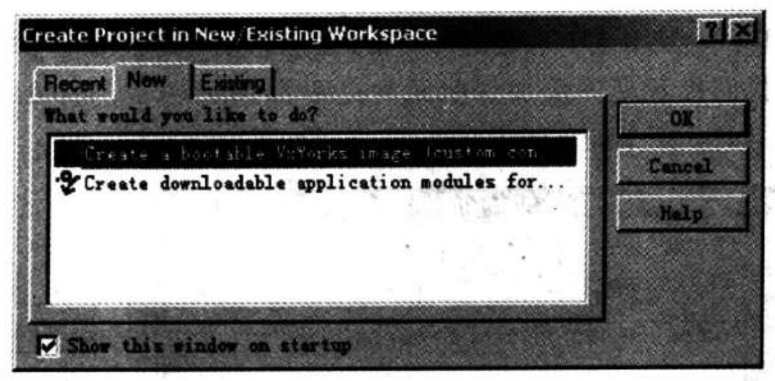


图 3-29 创建新工程

(2) 选择参考 BSP(基于 VxSim)

接下来工程向导会提示输入参考 BSP，可按照图 3-31 所示选择 simpc_gnu.wpj。

(3) 完成工程的创建

工程创建完毕后，会给出工程的信息，如工作区的位置和名称、工程的位置和名称以及参考工程，如图 3-32 所示。

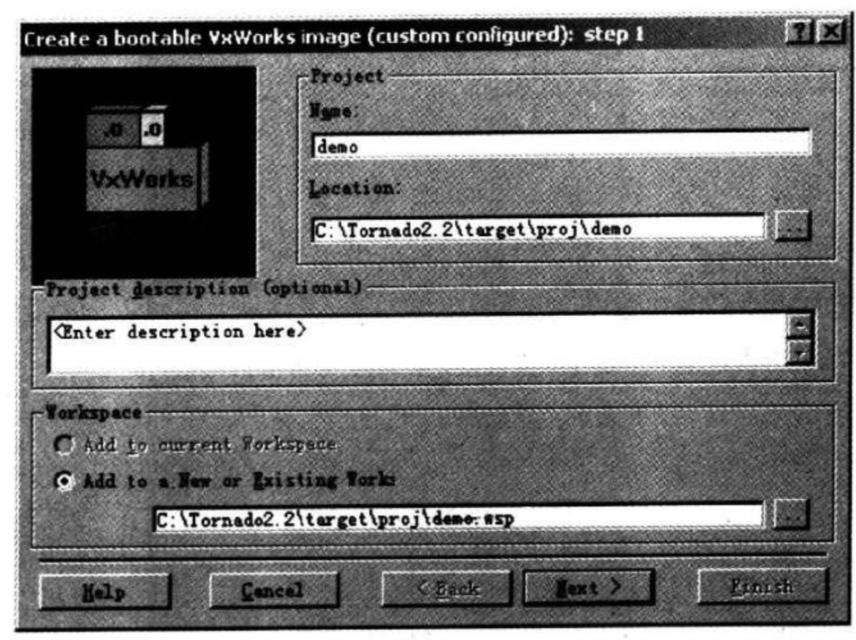


图 3－30　输入新建工程的详细信息

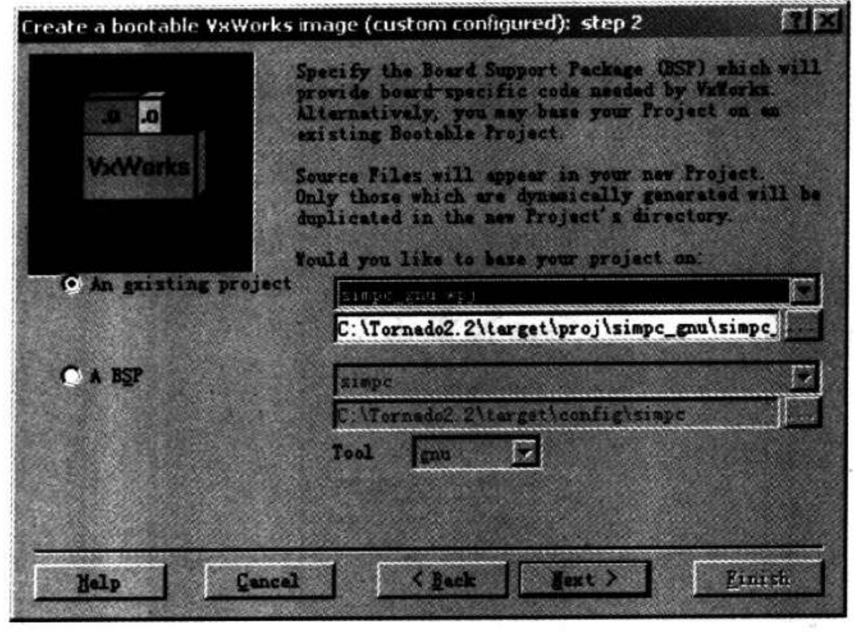


图 3－31　选择创建基于 VxSim 模板的工程

(4) 编译创建的工程

通过在工程管理器中工程名的节点处右击并选择 Build 'vxWorks. exe'来编译上面新建的工程，如图 3－33 所示。

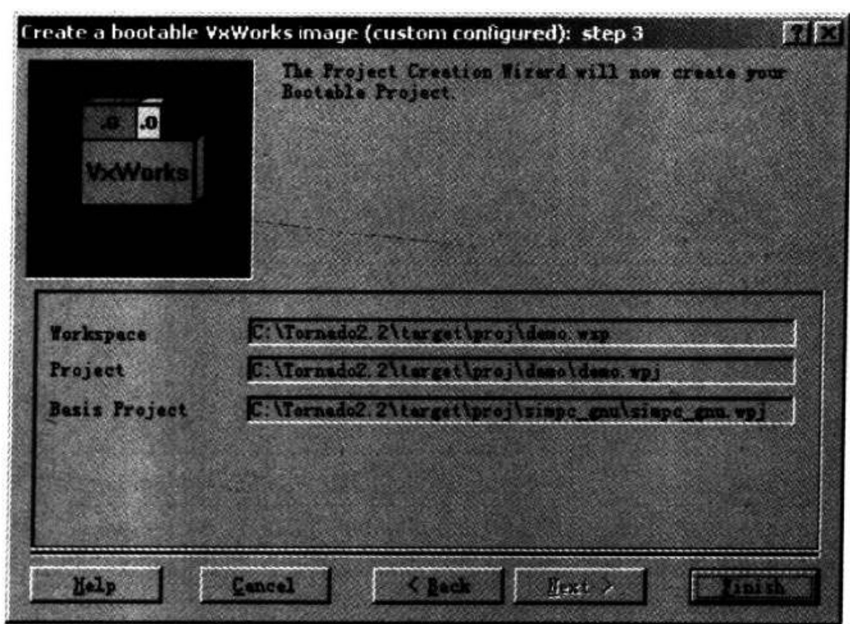


图 3-32 工程创建报告

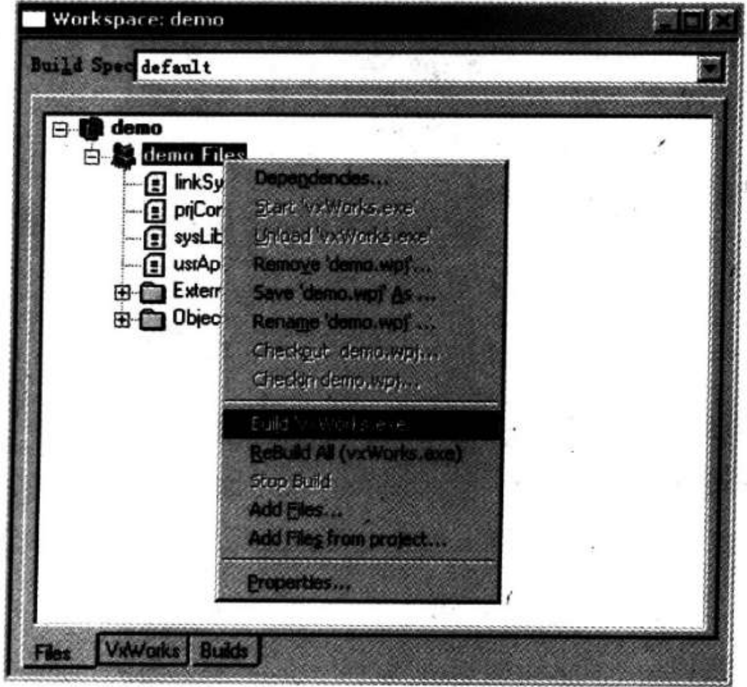


图 3-33 编译工程

(5) 再创建一个可下载的模块

为了方便，还可新建一个可下载的模块。在新建工程时选择 Create downloadable application modules(如图 3-34 所示)。这个可下载模块能够在目标机运行起来后，通过调试工具动态地下载到目标系统中。参照图 3-35，输入工程的名称和位置等信息；最后如图 3-36 所示，选择编译工具，这里选择 SIMNTgnu。

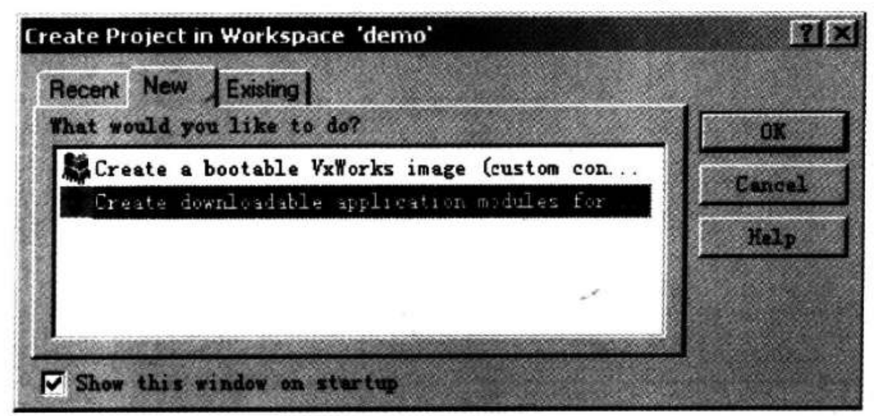


图 3-34 创建可下载模块

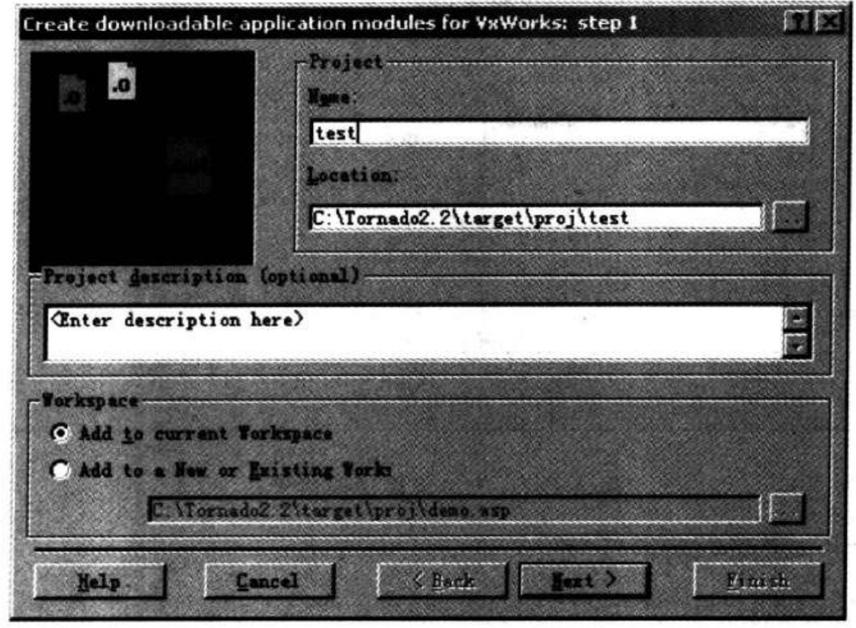


图 3-35 输入可下载模块的详细信息

(6) 在 test 可下载模块中添加测试代码

通过选择 File→New 创建测试程序(见图 3-37)。当选中复选框 Add to project 时，该新文件会自动加入到相应的工程中。输入程序清单 3-2 所列的代码备用。

图 3-36 选择新建工程的编译器

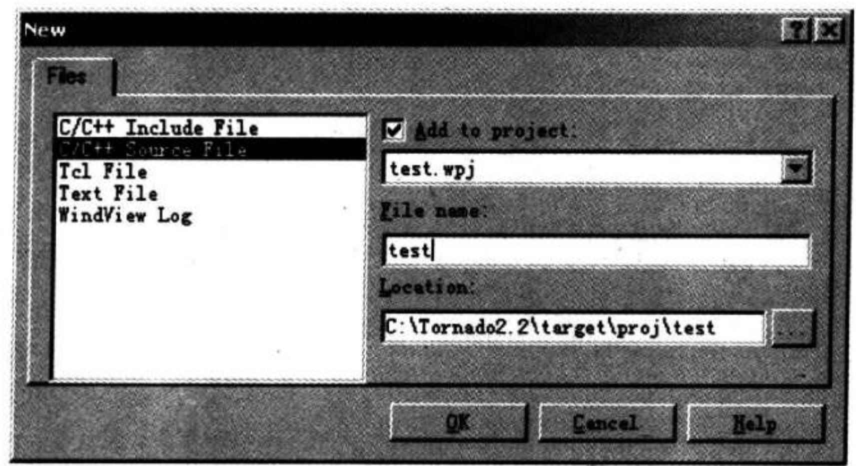


图 3-37 为工程添加测试代码

程序清单 3-2 简单的测试代码

```c
#include <stdio.h>
void add_test( int x, int y )
{
    printf(" %d + %d = %d\n", x, y, x+y );
}
```

(7) 编译 test 模块

与可启动模块一样，可通过在工程管理窗口中对应的工程名上右击并选择 Build 'test . out'来编译该模块，如图 3 - 38 所示。

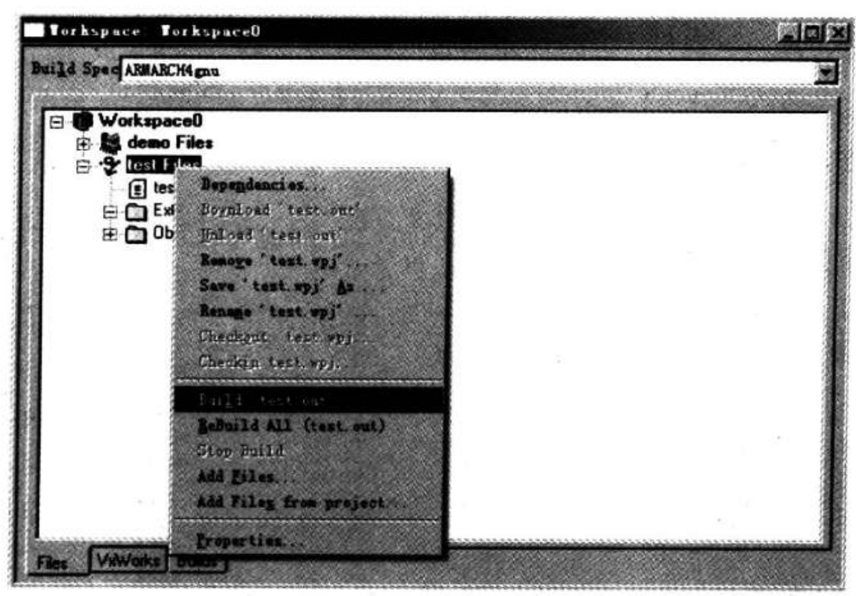


图 3 - 38 编译可下载模块

如果有硬件环境，则可参照本小节所讲的内容创建两个类似的工程供后面使用。

3.4.2 目标机服务器

目标服务器(Target Server)管理主机与目标机的通信，所有与目标机的交互都通过目标服务器来完成。它也管理主机上的目标机符号表，提供目标模块的加载和卸载。

目标代理程序是一个驻留在目标机中的任务，它起到一个桥梁作用，用于联系 Tornado 工具和目标机系统的组件。通常，目标代理程序是不可见的。

一个目标服务器必须为一个目标机而配置，并在启动主机工具之前启动该目标服务器。对于一个目标服务器，必须配置合适的通信协议，如串口通信协议 wbdserial 和网络通信协议 wbdrpc。当一个目标服务器启动后，须使用该通信协议来确定运行在目标机上的代理程序，并与其通信。

1. 配置和启动目标服务器

可通过选择 Tools→Target Server→Configure 来启动目标服务器配置程序。在 Target Server Properties 下拉列表框中有 9 个可设置的选项。在启动该配置程序后，默认情况下配置程序出现 Back End 配置选项，见图 3 - 39。

如果使用默认的 wbdrpc 后台，则须输入如下信息：

配置的名称(Description)：可任意指定，但一般使用一些能够体现目标机的命名，这样对于那些有多种配置的主机会更加方便。该配置填写完毕后，选中 Add description to menu 复选框，这样该配置的名称就会出现在 Tools→Target Server 中；当我们从菜单中选中某一个菜单项时，该服务会自动加载。

目标服务器(Target Server)：用来连接目标机名称(Target Name)或者 IP 地址(IP Address)。

如果使用 wbdserial 后台，则还须设置通信端口号(Serial Port)以及通信速率(Speed)，如图 3-40 所示。

在进行了上述配置后，可以在图 3-39 中单击 Launch 按钮来启动该服务。如果不想启动该服务，则可单击 OK 按钮来保存该配置。

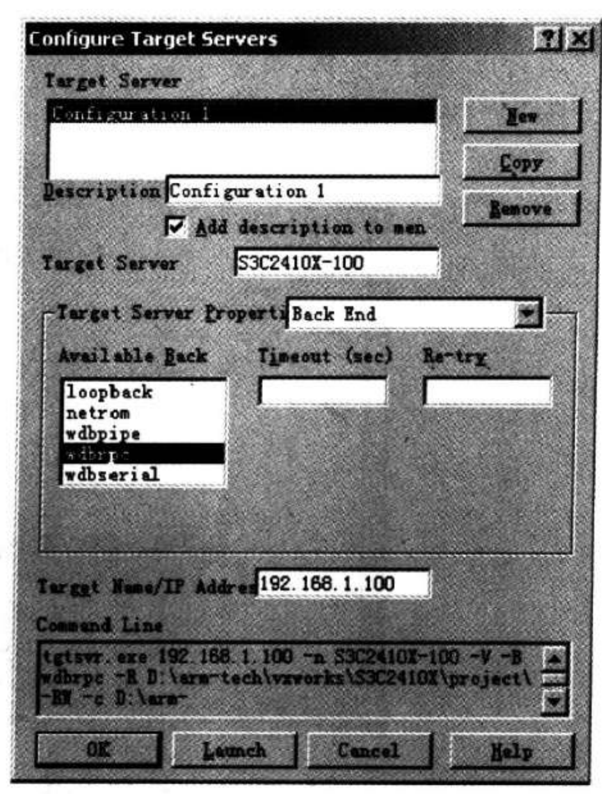


图 3-39 配置目标服务器

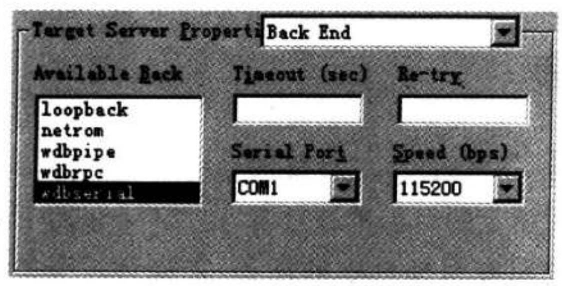


图 3-40 选择目标服务器的通信模式及参数

目标服务器启动后，在 Windows 系统的任务栏中会出现一个类似于靶的图标。双击该图标，可以看到该目标服务器的一些输出信息。如图 3-41 所示，启动了一个基于 VxSim 的目标服务器。

在图 3-40 的 Target Server Properties 下拉列表框中还有一些非常有用的配置选项，包括 Core File and Symbols、Memory Cache Size 和 Target Server File System。

Core File and Symbols：用于设置读取目标机系统上运行的内核代码的符号表信息，如图 3-42 所示。当通过 Shell 运行一些例程或者执行其他操作时，都会从该文件中读取符号调试信息，从而定位一些例程的入口以及一些变量的地址等，所以该编辑框内所指的文件应与下载

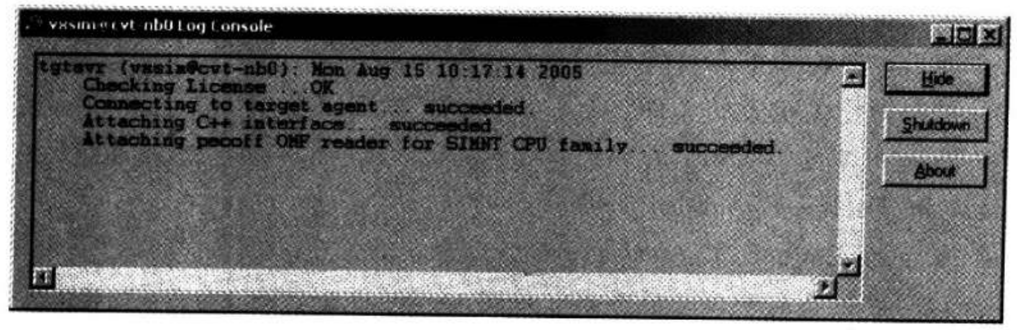


图 3-41　启动目标服务器后的输出信息

到目标机上运行的文件相对应。一般下载到目标机上运行的内核代码可以通过一些工具(如 arm-elf-objcopy)去除调试信息,但这里指定的文件要包含调试信息的文件(即去除调试信息之前的文件)。在图 3-42 中可以选择从该文件中读取所有符号(All Symbol)或者只读取全局符号(Global Symbol)。

注意: 如果该文件输入有错误,则一方面在目标服务器启动后,会出现一个 Chksum mismatch 的提示信息;另一方面,当通过 Shell 进行一些例程调用或表达式计算时,将可能得不到正确的结果。

Memory Cache Size:为了避免与目标机进行过多的数据交换,在目标机服务器中维持了一个目标机的数据缓冲。缓冲大小默认情况下为 1 MB,也可通过选择 Specify(K Bytes)来指定该缓冲的大小,如图 3-43 所示。

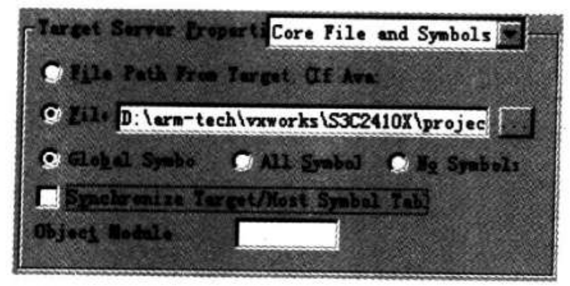


图 3-42　设置目标系统的内核符号文件

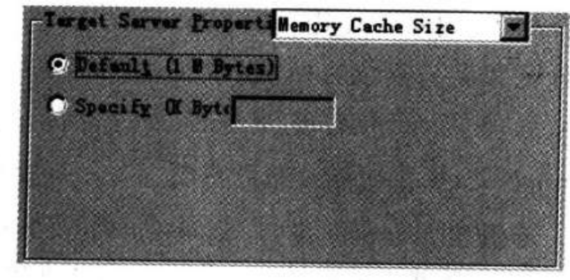


图 3-43　设置目标代理的缓冲大小

Target Server File System:简称为 TSFS,是一个简单且非常好用的文件系统。它提供了一个对主机上的文件进行全功能访问的目标机文件系统接口。为了使用该文件系统,需在内核中配置该模块,即 WBD Target Server File System 组件;同时,还须进行一些简单的配置,如图 3-44 所示。首先,选中 Enable File System 复选框;然后在 Root 对应的文本框中输入主机文件系统的目录,该目录将作为 TSFS 的根目录;最后还要指定文件系统的属性,即只读(Read only)还是可读/写(Read/Write)。配置成功后,启动相应的内核,然后在 Shell 下通过 devs 命令检查,就会发现一个/tgtsvr 设备,这就是 TSFS 文件系统。该设备的文件目录/

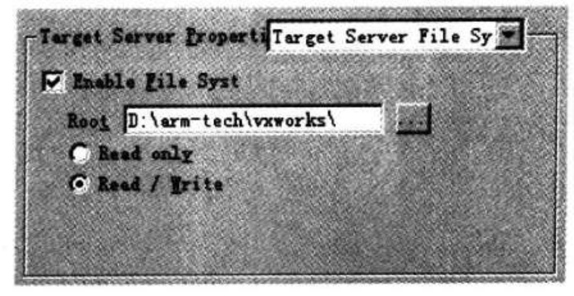


图 3-44 设置 TSFS 文件系统的根路径和存取权限

tgtsvr 实际上就等价于所输入的 Root 对应的主机目录。随后在目标机上即可像使用本地文件系统一样来操作主机上的文件。

注意：如果使用基于 VxSim 工具的工程，则可直接选择 Tools→Simulator 来启动模拟器，并在随后的 Launch Target server 对话框中单击 OK 按钮来启动目标服务器。

2. 停止目标服务器

停止目标服务器的方法有很多，简单的一种就是在任务栏中的目标服务器图标处右击并选择 Shutdown；也可双击该图标，在出现的对话框单击 Shutdown 按钮。

3.4.3 调试命令行解释器

调试命令行解释器(WindSh)是一个运行在主机上的 C 语言解释器。通过它可运行下载到目标机上的所有函数，包括 VxWorks 内核自身的函数以及应用程序所包含的函数；并且能够解释 C 语言书写的大多数表达式，允许对变量的符号访问。WindSh 外壳还能解释常规的工具命令语言(TCL)。另外，WindSh 还可实现以下所有调试功能：下载/删除软件模块；产生/删除任务；设置/删除断点；运行、单步或继续执行程序；查看/修改内存、寄存器和变量；查看任务列表、内存的使用情况以及 CPU 的利用率；查看特定的对象(任务、信号量、消息队列、内存分区和类)；复位目标机。

启动 WindSh 有多种途径，例如：在 Target Server 运行起来后，直接单击工具栏中的 Launch shell 按钮，即可弹出 WindSh 窗口，如图 3-45 所示。

若退出 WindSh 窗口，则可在 WindSh 中执行 exit 或 quit，也可使用 Ctrl+D 停止 WindSh 对话。如果 Shell 不接收输入，则可使用 Ctrl+Break 来停止 WindSh。WindSh 的功能非常强大，常用的一些有：自动补齐、函数摘要打印、数据变换、数据计算、符号表达式计算，以及内置 Shell 例程的调用等。

(1) 自动补齐

在 Shell 下输入任何符号或主机上的路径名，可使用 Ctrl+D 来补齐。例如：首先输入“c:\win”，然后输入 Ctrl+D，则可得到“c:/WINDOWS/”字符串；而且对于路径，Shell 会自动使用正斜杠替代 Windows 使用的反斜杠：

```
-> c:\win[Ctrl + D]
-> c:/WINDOWS/
```

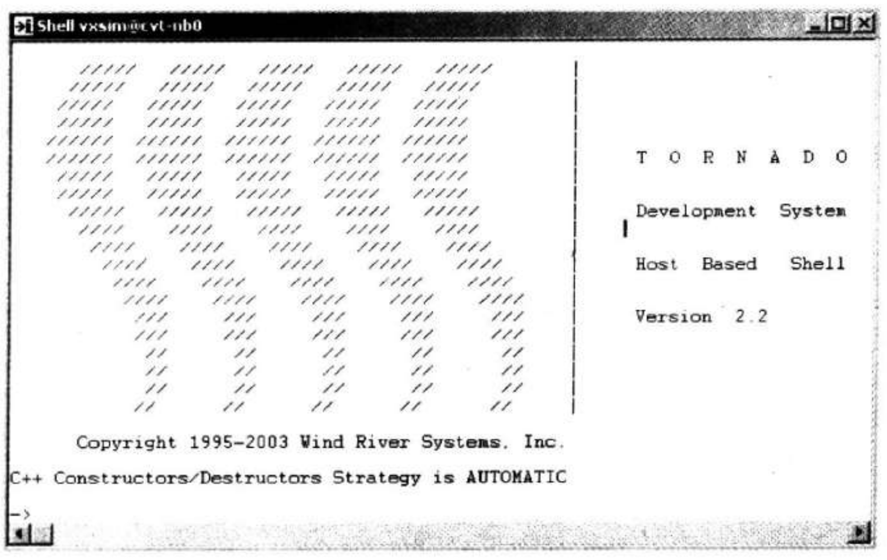


图 3-45 WindSh 窗口

另外，如果以某个字符串开头的符号或者路径存在多个匹配项，则在输入 Ctrl+D 后，系统也会将这些匹配项一一列举出来。例如：输入 prin 和 Ctrl+D，可得到以 prin 开头的所有符号的列表：

```
-> prin
printErrno   printLogo   _printf     _printOut     _printErr     _printExc
-> prin
```

(2) 函数摘要打印

一旦输入了完整的函数名，再输入 Ctrl+D，Shell 就会自动输出该函数的一些摘要。例如：输入_printf 和 Ctrl+D，即可得到_printf 函数的一些说明：

```
-> _printf
printf() - write a formatted string to the standard output stream (ANSI)
    int printf
        (
        const char * fmt,          /* format string to write */
            ...                    /* optional arguments to format string */
        )
-> _printf
```

为了显示自己所编写的函数的摘要，还须做以下工作：

- 创建包含新例程的文件，该例程应遵循 Wind River 编码规则（可参阅《VxWorks Programmer's Guide：Coding Conventions》）；

- 将该文件添加到工程中；
- 再将该文件添加到 makefile 的宏 DOC_FILES 中；
- 到达该工程的根目录，运行 make synopis。

(3) 数据变换

Shell 使用十六进制和十进制显示所有的整数和字符，也可显示字符常量、符号地址及其偏移量。例如：

```
-> 31
value = 31 = 0x1f = _major_os_version_ + 0x1b
-> '1'
value = 49 = 0x31 = '1' = _major_os_version_ + 0x2d
-> 0x1234
value = 4660 = 0x1234 = _section_alignment_ + 0x234
```

(4) 数据计算

所有的 C 运算符都可用于数据计算，并且可以使用括号“()”来设定运算的优先级，例如：

```
-> (1 + 2) * 3
value = 9 = 0x9 = _major_os_version_ + 0x5
-> (1 << 2) + 100
value = 104 = 0x68 = 'h' = _major_os_version_ + 0x64
-> 10.1 * 100
value = 1010
```

(5) 符号表达式计算

例如在下面的例子中首先添加一个变量 a，随后即可在 Shell 中引用该变量进行一些计算：

```
-> a = 10
new symbol "a" added to symbol table.
a = 0xca0e34: value = 10 = 0xa = _major_os_version_ + 0x6
-> a1 = 100
new symbol "a1" added to symbol table.
a1 = 0xca0e2c: value = 100 = 0x64 = 'd' = _major_os_version_ + 0x60
-> a * 2
value = 20 = 0x14 = _major_os_version_ + 0x10
-> x = a + a1
new symbol "x" added to symbol table.
x = 0xca0e24: value = 200 = 0xc8 = _major_os_version_ + 0xc4
```

(6) 内置 Shell 例程的调用

在 Shell 模块中,内置了很多服务例程,可以把它们分为几类:任务管理、任务信息查询、系统信息查询、调试以及目标显示等,如表 3-1 所列。

表 3-1 Shell 模块内置的常用命令

常用命令	功 能	常用命令	功 能
b	设置或显示断点情况	period	创建一个任务,周期地调用指定的例程
bd	删除一个断点	printErrno	显示指定错误代码的描述信息
bdall	删除所有断点	pwd	显示当前目录
bh	设置一个硬件断点	repeat	生成一个任务循环调用某个例程
c	恢复系统的运行	s	单步运行一个任务
cd	改变默认的目录	sp	使用默认的参数启动一个新任务
d	显示内存的数据	sps	使用默认的参数创建一个新任务,并将其置为挂起状态
h	显示或设置 Shell 历史记录的缓冲大小	td	删除指定的任务
help	显示指定例程的帮助信息	ti	显示指定任务的详细信息
l	将指定的地址进行反汇编	tr	恢复一个被挂起的任务
ld	动态加载一个可重定位模块	ts	挂起指定的任务
lkAddr	在符号表中搜索与给定地址相近的符号名	tt	显示指定任务的函数调用堆栈
lkup	列出与指定字符串相近的符号	tw	如果给定的任务被挂起,则显示引起该任务挂起的对象的信息
ls	显示指定目录的文件列表	version	显示 VxWorks 版本信息
m	修改指定的存储器	w	逐个显示所有任务的挂起信息(包括挂起状态和原因)
mRegs	修改寄存器		

3.4.4 调试器

源码级调试器(CrossWind Debugger)提供了图形和命令行两种调试方式,可进行任务级或系统级的断点设置、单步执行以及调试异常处理等。

调试器的调试引擎是 GDB 的增强版,来自免费软件基金会(FSF)的可移植符号调试器。GDB 命令的全部文档资料可见《GDB User's Guide》。

Debug 菜单提供了调试器命令的全部列表，如常见的 Step Into、Step Over、Step Out 和 Continue 等；使用键盘快捷方式也可执行相应的命令，表 3－2 描述了这些命令的具体功能。同时，在 CrossWind 工具栏中还提供了按钮(如图 3－46 所示)，它们可执行一些较常用的调试命令，打开或者关闭显示数据、存储器和堆栈信息的窗口。

表 3－2 调试器命令列表

命 令	功 能
Source Search Path	设置源代码的搜索路径
Run	在目标机上执行例行程序，并将其作为在调试器控制下的新任务。打开 RunTask 对话框，可指定须运行的例行程序的入口
Detach	将一个在调试器控制下的任务从调试器任务环境中分离
Detach and Resume	将一个在调试器控制下的任务从调试器任务环境中分离，并恢复该任务的运行
Attach	将一个指定的任务绑定到调试器控制中
Interrupt Debugger	中断任务执行
Stop Debugging	停止调试器
Breakpoints	打开断点列表窗口
Toggle Breakpoint	在编辑窗口的当前行中设置或取消一个任务级断点
Toggle Global Breakpoint	在编辑窗口的当前行中设置或取消一个全局断点
Toggle Temp Breakpoint	在编辑窗口的当前行中设置或取消一个临时断点
Step Into	单步方式执行代码的下一行。如果该语句是子程序调用，则进入函数体内部
Step Over	单步方式执行代码的下一行。如果该语句是子程序调用，则将该子程序执行完毕
Continue	继续执行程序，直至遇到下一个断点、异常，也可使用 Interrupt Debugger 将其停止
Step Out	继续执行程序，直到当前子程序返回
Watch	打开或关闭 Watch 窗口，它在程序执行的整个过程中显示指定变量的数值
Variables	打开或关闭 Variables 窗口，它显示局部变量的数值
Registets	打开或关闭 Registers 窗口，它显示目标机寄存器的数值
Back Trace	打开或关闭 Back Trace 窗口，它显示函数调用的堆栈信息
Memory	打开或关闭 Memory 窗口，它显示目标机存储器信息

图 3－46 CorssWind 工具栏

启动 CrossWind 调试器后，使用快捷键 Alt+1 可以打开命令行窗口，如图 3-47 所示。该窗口为调试器提供了一个命令行界面，它不仅可用来进行标准的调试器操作，还可进行一些图形界面无法执行的操作。

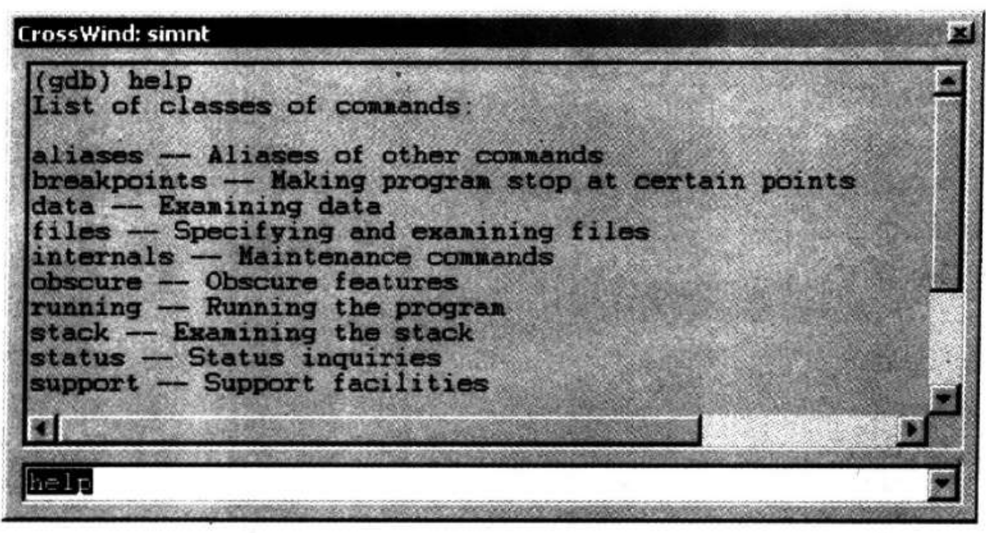


图 3-47　CrossWind 调试器命令行窗口

启动 CrossWind 的方法有两种：单击工具栏的图标，或者选择 Tools→Debugger。通过后者启动调试器时，可以选择目标服务器，如图 3-48 所示。

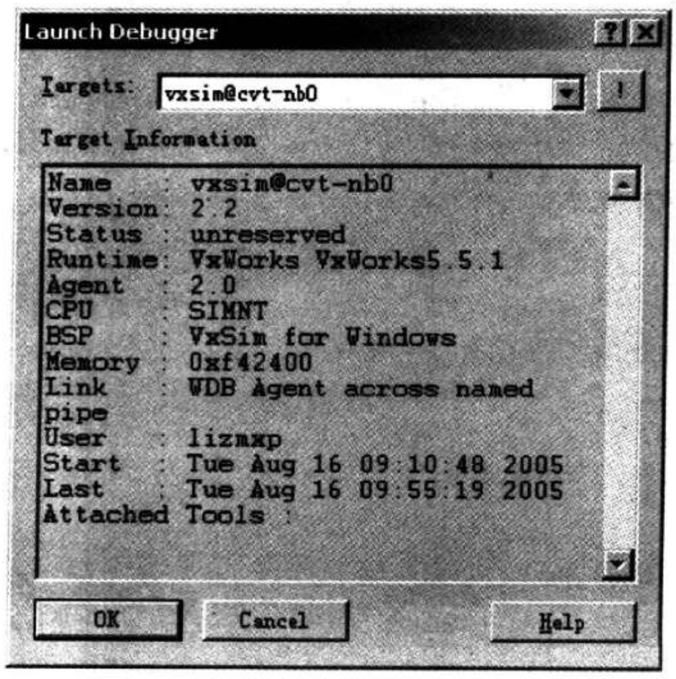


图 3-48　通过 Tools→Debuggor 启动 CrossWind

同样，要停止调试器，可以使用工具条栏的按钮，或者选择 Debug→Stop Debugging。

在进行后面的操作之前，须先创建一个可下载的测试工程，并将测试代码（如程序清单 3-3 所列）加入到该工程中。

程序清单 3-3　示例代码

```
#include "VxWorks.h"
#include "semLib.h"
#include "taskLib.h"
#include "msgQLib.h"
#include "wdLib.h"
#include "logLib.h"
#include "tickLib.h"
#include "sysLib.h"
#include "stdio.h"
/* 宏定义部分 */
#if FALSE
#define     STATUS_INFO
#endif
#define     MAX_MSG             1                  /* 消息队列中的最大消息数 */
#define     MSG_SIZE            sizeof (MY_MSG)    /* 每个消息的大小 */
#define     DELAY               100                /* 延时 100 ticks */
#define     HIGH_PRI            150                /* 高优先级任务的优先级别 */
#define     LOW_PRI             200                /* 低优先级任务的优先级别 */
#define     TASK_HIGHPRI_TEXT   "Hello from the 'high priority' task"
#define     TASK_LOWPRI_TEXT    "Hello from the 'low priority' task"
/* 类型定义 */
typedef struct my_msg
    {
    int     childLoopCount;                        /* 计数器 */
    char * buffer;                                 /* 消息数据指针 */
    } MY_MSG;
/* 全局变量 */
SEM_ID                          semId;             /* 信号量 ID */
MSG_Q_ID                        msgQId;            /* 消息队列 ID */
WDOG_ID                         wdId;              /* 看门狗 ID */
```

```
int             highPriId;                  /* 高优先级任务的 ID */
int             lowPriId;                   /* 低优先级任务的 ID */
int             windDemoId;                 /* winDemo 任务的 ID */
/* 函数声明 */
LOCAL void taskHighPri (int iteration);
LOCAL void taskLowPri (int iteration);

/*****************************windDemo——演示程序的根任务*************************/
void windDemo
    (
    int iteration
    )
    {
    int loopCount = 0;

#ifdef STATUS_INFO
    printf ("Entering windDemo\n");
#endif

    if (iteration == 0)
    iteration = 10000;

    /* 创建消息队列、信号量和看门狗定时器 */
    msgQId      = msgQCreate (MAX_MSG, MSG_SIZE, MSG_Q_FIFO);
    semId       = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
    wdId        = wdCreate ();
    windDemoId = taskIdSelf ();
    FOREVER
    {
       /* 创建子任务 */
        highPriId = taskSpawn ("tHighPri", HIGH_PRI, VX_SUPERVISOR_MODE, 4000,
               (FUNCPTR) taskHighPri, iteration,0,0,0,0,0,0,0,0,0);
        lowPriId = taskSpawn ("tLowPri", LOW_PRI, VX_SUPERVISOR_MODE, 4000,
               (FUNCPTR) taskLowPri, iteration,0,0,0,0,0,0,0,0,0);
        taskSuspend (0);          /* 挂起自己,直至别的任务将其唤醒 */

#ifdef    STATUS_INFO
```

```
        printf ("\nParent windDemo has just completed loop number %d\n",
        loopCount);
#endif /* STATUS_INFO */

        loopCount ++ ;
    }
    }

/********************************* taskHighPri——高优先级任务 ************************/
LOCAL void taskHighPri
    (
    int iteration
    )
    {
    int     ix;
    MY_MSG msg;
    MY_MSG newMsg;

    for (ix = 0; ix < iteration; ix ++ )
    {
    /* 首先触发信号量,然后检查该信号量 */
    semGive (semId);
    semTake (semId, 100);

    /* 等待信号量 */
    semTake (semId, WAIT_FOREVER);
    taskSuspend (0);                    /* 获取信号量后,将自己挂起 */

    /* 生成消息,并发送该消息 */
    msg.childLoopCount = ix;
    msg.buffer = TASK_HIGHPRI_TEXT;
    msgQSend (msgQId, (char * ) &msg, MSG_SIZE, 0, MSG_PRI_NORMAL);

    /* 尝试读取收到的消息,此处应为本任务发来的消息 */
    msgQReceive (msgQId, (char * ) &newMsg, MSG_SIZE, NO_WAIT);

#ifdef    STATUS_INFO
```

```
    printf (" %s\n Number of iterations is %d\n",
            newMsg.buffer, newMsg.childLoopCount);
#endif /* STATUS_INFO */

    /* 等待消息,此处应为低优先级任务发来的消息 */
    msgQReceive (msgQId, (char *) &newMsg, MSG_SIZE, WAIT_FOREVER);

#ifdef    STATUS_INFO
    printf (" %s\n Number of iterations by this task is: %d\n",
            newMsg.buffer, newMsg.childLoopCount);
#endif  /* STATUS_INFO */

    /* 启动看门狗定时器 */
    wdStart (wdId, DELAY, (FUNCPTR) tickGet, 1);
    wdCancel (wdId);
    }
    }

/********************************taskLowPri——低优先级的任务**********************/

LOCAL void taskLowPri
    (
    int iteration
    )
    {
    int         ix;
    MY_MSG      msg;

    for (ix = 0; ix < iteration; ix++)
    {
    semGive (semId);                            /* 触发信号量 */
    taskResume (highPriId);                     /* 唤醒高优先级任务 */

    /* 生成消息并发送 */
    msg.childLoopCount = ix;
    msg.buffer = TASK_LOWPRI_TEXT;
    msgQSend (msgQId, (char *) &msg, MSG_SIZE, 0, MSG_PRI_NORMAL);
```

```
    taskDelay (60);
    }

    taskResume (windDemoId);              /* 唤醒 windDemo 任务 */
    }
```

要调试编写的程序,可先将须调试的程序下载到目标系统中,即在 Shell 中使用 ld 命令,或者在目标代码 windDemo.o 的上下文菜单中选择 Download 'windDemo.o',如图 3-49 所示。

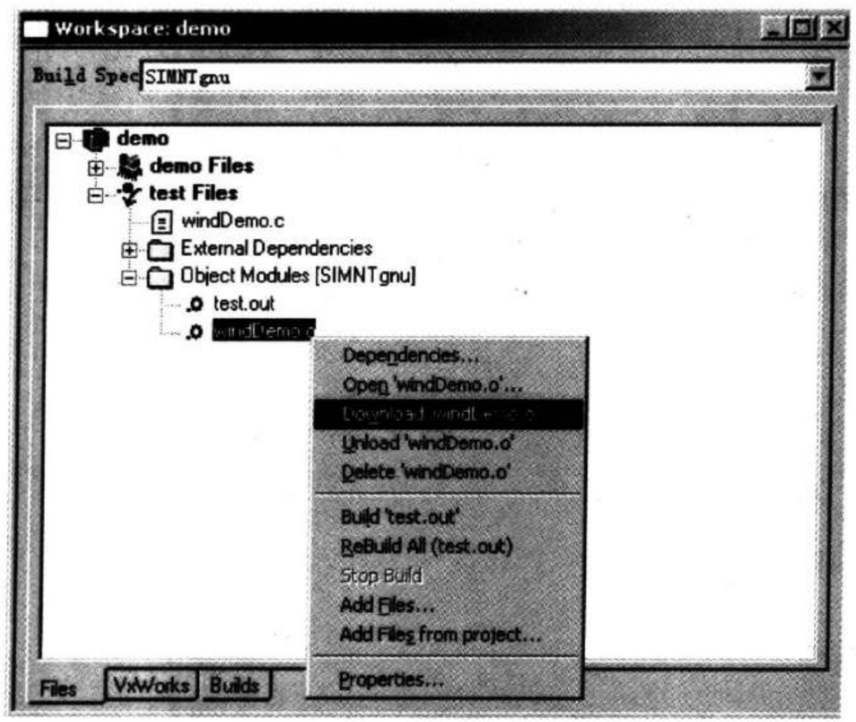


图 3-49 目标程序的下载

通常在调试过程中,会不断地修改程序。如果更新并且下载了一个同名的模块,则系统会使用新模块自动将先前下载的模块替代,而无需提前卸载。

模块下载之后,即可使用工具栏或者选择 Debug→Run 来运行指定的例程。运行前,会出现一个 Run Task 对话框,可通过该对话框指定运行的例程入口以及参数。多个参数可使用空格分开。参数可以是数字(十进制、八进制或十六进制)、字符串、字符,以及输入/输出重定向符号"<"和">";也可是 VxWorks 符号表中的全局符号。如图 3-50 所示,启动 windDemo 例程,并指定参数为 100,同时选择 Break at entry point 复选框使程序在入口处暂停。

图 3-50 显示了带有例程入口以及参数的 Run Task 对话框。如果不给出其必需的参数,则默认为 0;如果想在例程的入口处设置一个临时断点,则可选择 Break at entry point 复选框。

只要一个任务在调试器的控制之下运行,即可进行 Step Into、Step Over、Step Out 和

Stop 等调试。图 3－51 示出了在图 3－50 中单击 OK 按钮启动 windDemo 例程的情况，任务停在函数的入口处，使用快捷键 F5 可让该程序继续运行。

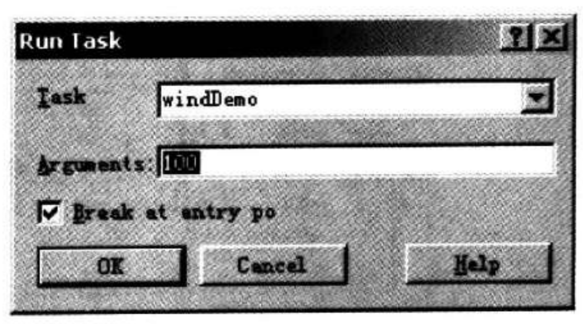


图 3－50　运行下载的程序

调试过程中，选择 Debug→Attach 可以显示出目标机中正在运行的任务列表，如图 3－52 所示。在任务列表中单击某个任务并单击 Attach 按钮，则可将指定的任务与调试器绑定起来。选择一个新任务与调试器绑定后，原先与调试器绑定的任务会自动脱离调试器的控制。

```
*
*/

void windDemo
    (
    int iteration                /* number of iterations of child code
    )
    {
    int loopCount = 0;           /* number of times through windDemo */

#ifdef  STATUS_INFO
    printf ("Entering windDemo\n");
#endif /* STATUS_INFO */

    if (iteration == 0)          /* set default to 10,000 */
        iteration = 10000;

    /* create objects used by the child tasks */

    msgQId      = msgQCreate (MAX_MSG, MSG_SIZE, MSG_Q_FIFO);
    semId       = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
    wdId        = wdCreate ();

    windDemoId  = taskIdSelf ();

    FOREVER
        {

        /* spawn child tasks to exercise kernel routines */
```

图 3－51　调试源代码窗口

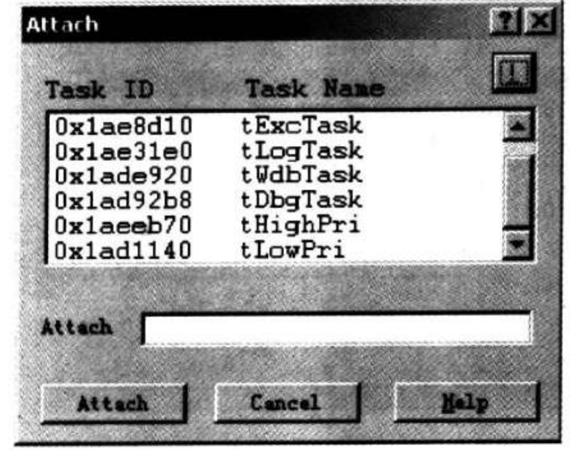


图 3－52　选择调试某个任务

为了进一步明确绑定的作用，可做如下试验：首先在 taskLowPri 和 taskHighPri 的循环体中设置断点，如图 3－53 所示。然后将调试器与 taskLowPri 任务绑定起来，执行 Continue 命令，会发现系统会在 taskLowPri 中的断点处停住，而不会停在 taskHighPri 中的断点处。如果将绑定的任务修改为 taskHighPri，那么执行 Continue 命令后，系统会在 taskHighPri 中的断点处停住。由此可知，只有与调试器绑定的任务才能被调试，所有调试命令也只会影响该任务。当然对于一个与调

试器绑定的任务，可通过 Debug→Detach and Resume 将其与调试器分离。

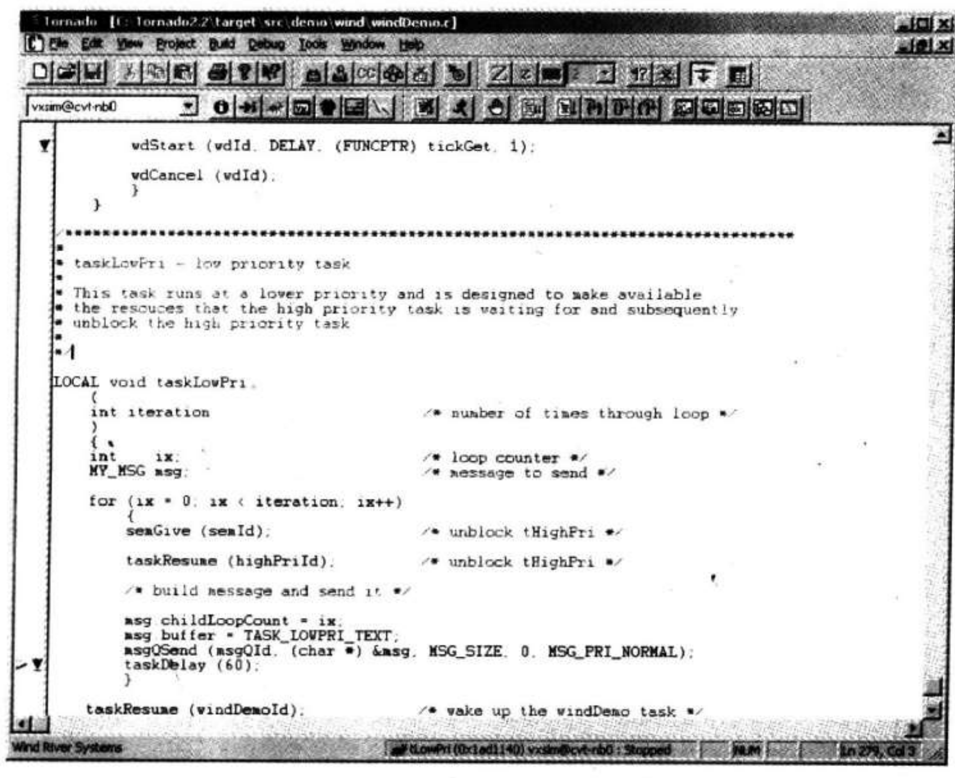


图 3-53　与调试器分离后的任务不受调试器控制

当一个与调试器绑定的任务停止运行后，可打开系统提供的一些调试窗口（如 Watch、Variable 和 Back Trace 等）来检查程序的状态；同时也可在源码窗口中选择 Mixed Source and Disassembly 来显示程序的汇编窗口。

WindDebugger 除了可以调试用户的任务外，还可调试系统任务。在这种模式下，可在多个任务之间自由切换，可以使用全局断点来打断整个系统，甚至可在系统例程（如 ISR）中检查运行情况。因此，这种模式一般被称为“系统调试模式”。为了使用系统调试模式，在内核配置中，必须选择系统调试组件 WDB System debugging。要启动系统调试模式可将调试器与系统任务绑定起来，如图 3-54 所示。

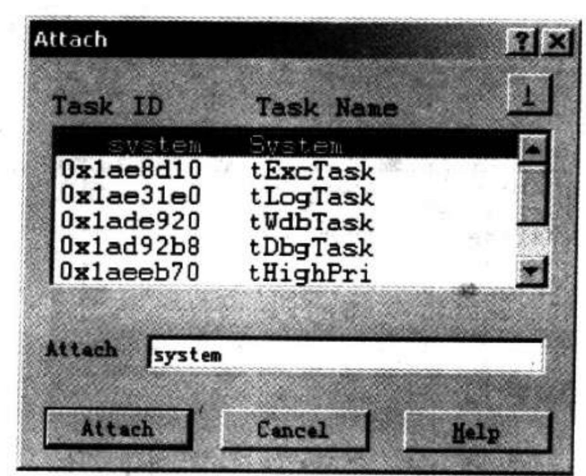


图 3-54　绑定系统任务

在系统调试模式中，GDB 线程调试工具就变得有用了。“线程”是带有一些独立上下文，但有一个共享地址空间的过程的通用术语。

在 VxWorks 中，每个任务就是一个线程；系统上下文(包括 ISR 和驱动程序)也是一个线程。GDB 用一个线程 ID 来识别每个线程，一个线程 ID 就是调试器内部的一个任意数。

可以使用 GDB 命令(见表 3-3)来管理线程。

表 3-3　GDB 的线程控制命令

线程控制命令	功能描述
info threads	为目标机系统中的每个线程显示信息摘要(包括线程 ID)
thread No	选择指定的线程为当前线程
break line spec thread No	设置一个只影响指定线程的断点

对这些命令的说明详见《GDB User's Guide: Debugging Programs with Multiple Threads》。下面讨论系统模式下，在调试一个 VxWorks 目标机的上下文中的线程命令。

1. 显示线程信息摘要

命令 info threads 显示出各线程 ID 分别与哪个 VxWorks 任务相对应。例如：在绑定系统后，立即显示出类似下面的信息：

```
(gdb)info threads
  4 task 0x4fc86     tExcTask     0x444f58 in?? ()
  3 task 0x4f9f40    tLogTask     0x444f58 in?? ()
* 2 task 0x4c7390    tNetTask     0x4151e0 in?? ()
  1 task 0x4boa24    tWdbTask     0x4184fe in?? ()
(gdb)
```

在 infothreads 的输出中，每行最左面的数字即调试器的线程 ID；左边空白处的单个星号"*"表明哪个线程是当前线程。当前的线程识别出"最为局部的"远景：报告针对任务信息的调试器命令，例如 bt 和 info reg(及其对应的调试器显示)，只适用于当前线程。线程列表的后面两列，显示了 VxWorks 的任务地址和任务名称。如果显示了系统上下文，则用一个单词 system 代替此两列。当前被内核调度的线程(一个任务或系统上下文)，用一个"+"标记在任务 ID 的右面。每行的剩余部分显示了每个线程当前堆栈帧的摘要。

2. 明确的线程切换

若要切换到一个不同的线程(使得该例程成为当前调试的例程，而不影响内核任务调度)，则使用 thread 命令。例如：

```
(gdb)thread 2
[Switching tO task 0x3a4bd8   tShell   ]
#0  0x66454 in semBTake()
```

```
(gdb)bt
#0   0x66454 in semBTake()
#1   0x66980 in semTake()
#2   0x63a50 in tyRead()
#3   0x5b07c in iosRead()
#4   0x5a050 in read()
#5   0x997a8 in ledRead()
#6   0x4a144 in execShell   ()
#7   0x49fe4 in shell   ()
(gdb)thread 3
[Switching t0 task 0x3aa9d8   tFtpdTask]
#0   0x66454 in semBTake   ()
(gdb)print/x $ i0
$ 3:0x3bdb50
```

每次切换线程，调试器就会显示出新的当前线程的任务名称。

3. 针对线程的断点

在系统模式下，无条件断点（用编辑器窗口中的图形控制设置，或者在 Debugger 窗口中用 break 命令和单个参数设置）是可以应用于全局的；任何线程在到达这样一个断点时就停止。也可设置针对线程的断点，这样在该处只有一个线程停止。

要设置一个针对线程的断点，须将线程 ID 前面的词 thread 添加到 break 命令中。例如：

```
(gdb)break printf thread 2
Breakpoint 1 at 0x56868
(gdb)  cont
Continuing.
[Switching t0  task  0x3a4bd8  +  tShell  Breakpoint 1,  0x568b8 in printf  ()
(gdb)  info threads
  8 task 0x3b8ef0    tExcTask     0xgbfd0 in qJobGet     ()
  7 task 0x3b6580    tLogTask     0xgbfd0 in qJobGet     ()
  6 task 0x3b15b8    tNetTask     0x66454 in semBTake    ()
  5 task 0x3ade80    tRlogind     0x66454 in semBTake    ()
  4 task 0x3abf60    tTelnetd     0x66454 in semBTake    ()
  3 task 0x3aa9d8    tFtpdTask    0x66454 in semBTake    ()
* 2 task 0x3a4bd8 +  tShell       0x56868 in printf      ()
  1 task 0x398688    tWdbTask     0x66454 in semBTake    ()
(gdb)bt
```

```
#0   0x56868 in printf   ()
#1   0x4a108 in execShell   ()
#2   0x49fe4 in shell   ()
```

在内部，调试器将在任何线程每次遇到此断点时取得控制；但如果线程不是命令指定的那个，那么调试器就会继续执行程序，而并不提示。

4. 隐含的线程切换

程序也许并不总是在期待的线程中被挂起。在任何线程中，当系统模式到达断点或者任何事件（如异常）发生时，调试器都会取得控制。无论目标机系统何时停止，调试器都会切换到正在执行的线程。如果新的当前线程与先前的数值不同，那么就会有以 Switching 开头的信息显示出哪个线程被挂起：

```
(gdb)  thread 2
(gdb)  cont
Continuing.
Interrupt...
Program received signal SIGINT,  Interrupt.
[Switching t0 system +]
0x5b58 in wdbSuspendSystemHere   ()
```

只要调试器没有取得控制，就可以通过选择 Debug→InterruptDebugger，或者按 Ctrl＋Break 键来中断目标机系统。当单步执行程序（使用命令 step、stepi、next 或 nexti 中的任何一个，或等价按钮）时，目标机就在它停止的地方继续执行。该目标机位于 infothreads 显示中标记有“＋”的线程中。然而，在单步执行该线程的过程中，其他线程也可能开始执行。因此，调试器可能会由于另一个线程中的事件，而在单步执行命令完成之前就停止在另外的线程中。

3.4.5 目标机浏览器

用目标机浏览器（Browser）可查看内存分配情况、任务列表、任务堆栈的使用情况、CPU 利用率，以及系统目标（如任务、消息队列和信号量等）信息等。对这些信息可周期性地进行更新。

可以通过工具栏上的按钮 来启动目标机浏览器，也可选择 Tools→Browser 来启动。在目标机浏览器左上角的下拉列表框选择 Target Information，可以看到目标机的一些相关信息，如图 3－55 所示；选择 Memory Usage，可查看目标机内存相关的信息，图 3－56 显示了 vxWorks. exe 代码、数据以及 bss 段的内存使用情况；选择 Module Information，可查看目标机运行的一些模块，如图 3－57 所示。

通过浏览器可以分析出目标机的一些错误，常见问题有如下几种：

(1) 内存泄漏

图 3－56 中，单击 Toggle Refresh 按钮进行周期刷新，如果发现存储器的分配持续增长，那么可能是由一个内存泄漏引起的。在一个运行的系统中，内存分配的持续增长是必须避免的，因为长时间的运行必将导致系统存储器耗尽，从而使系统产生一些不可预知的结果。

(2) 内存碎片

当长时间将小块内存和中等大小的内存进行交叉分配，而这些小块内存又没有被及时释放时，将会产生大量的内存碎片；而内存碎片的产生会降低系统的性能。此时借助于浏览器工具可以很好地分析该问题。如果在释放列表中存在许多小块节点，而且这些节点数目呈增长趋势，则可断定出现了过多的内存碎片。如图 3－58 所示，在释放列表中只包含 3 个节点；而在图 3－59 中，则出现很多小块内存。

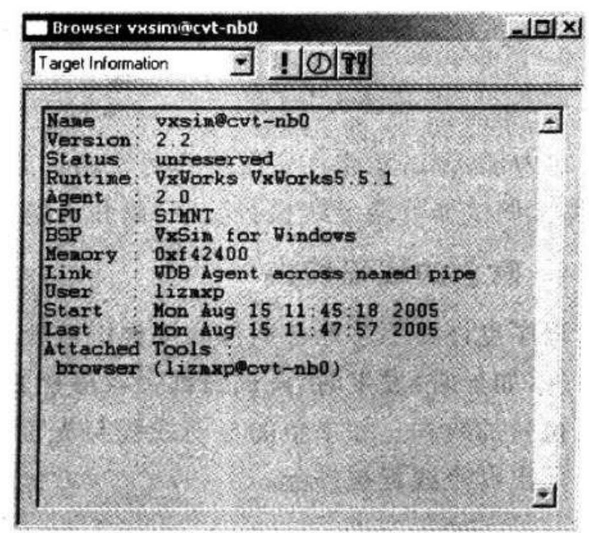


图 3－55　目标机浏览器

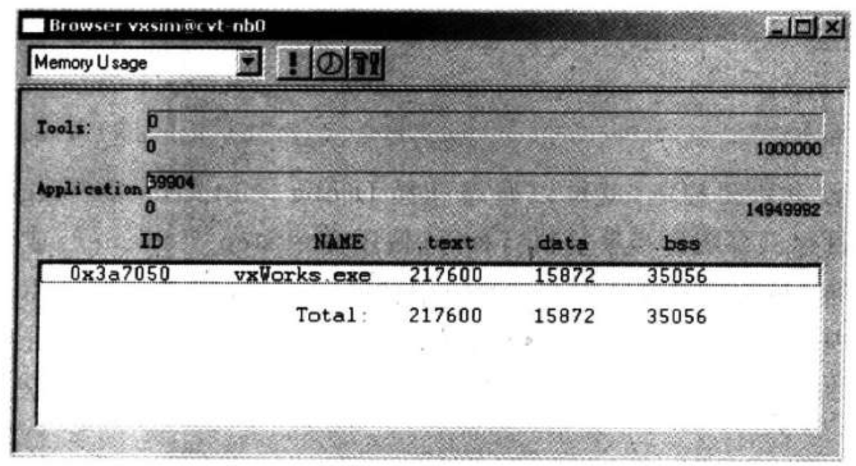


图 3－56　内存使用情况

(3) 堆栈溢出

堆栈溢出也是程序设计中的一个大问题。一个任务的堆栈溢出后，会对其他程序产生一些影响，从而使整个系统的行为变得难以解释。当遇到这种问题时，可以借助于堆栈检查工具来验证是否存在堆栈溢出。例如：在 test.c 文件中添加程序清单 3－4 所列代码。

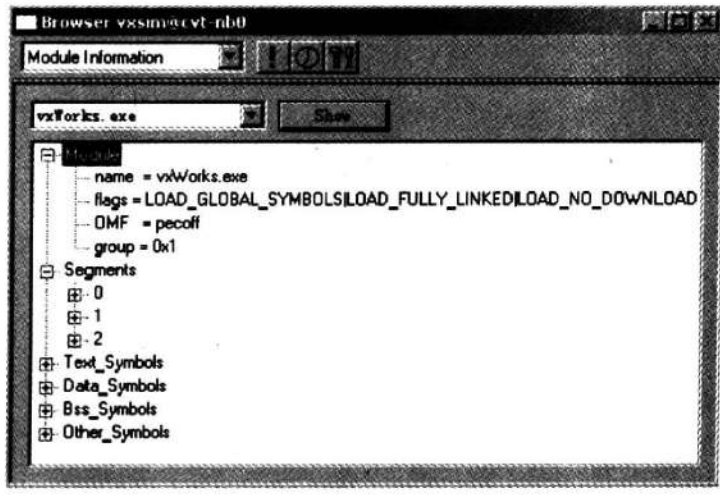


图3-57 模块信息

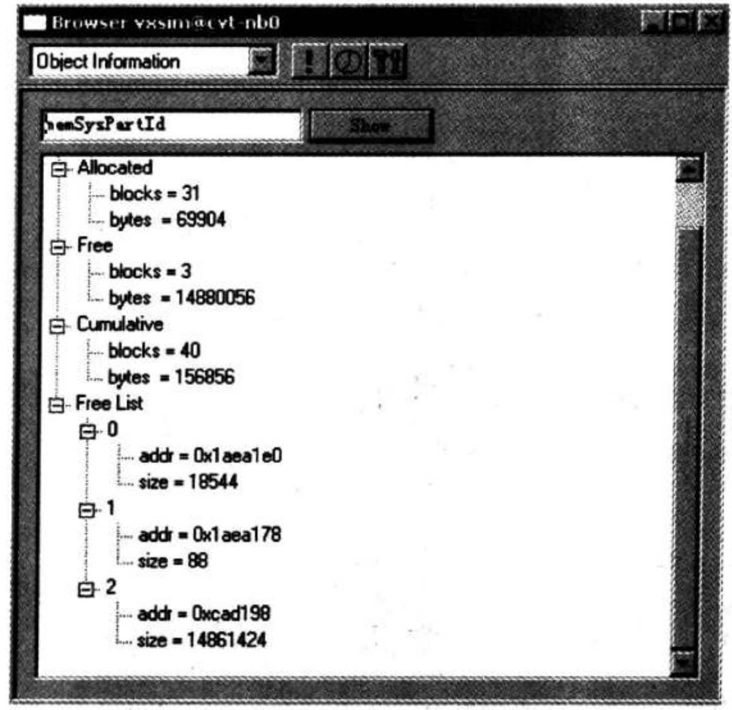


图3-58 系统的内存分区信息(无内存碎片)

程序清单3-4 堆栈溢出测试程序

```
void task0()
{
    char test[4096];
    getchar();
}
```

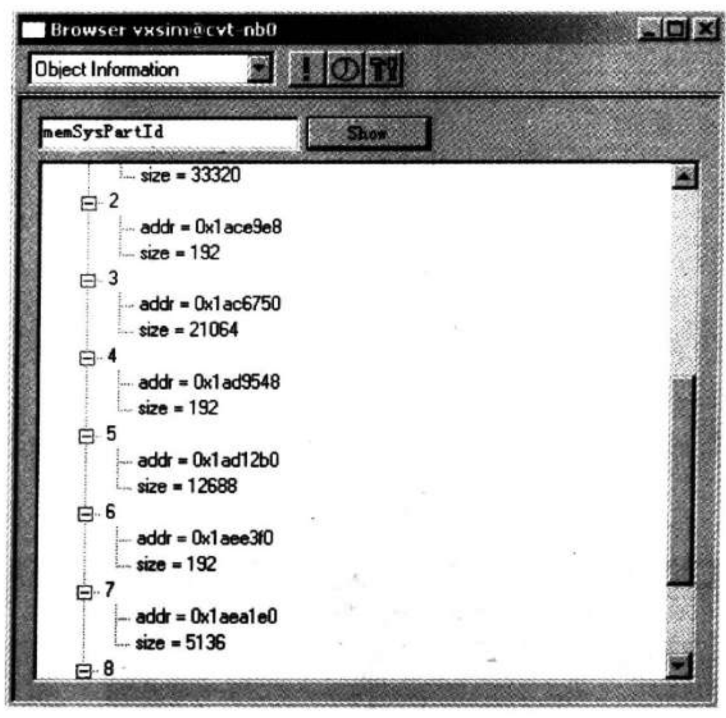


图 3-59　系统的内存分区信息(存在内存碎片)

编译后下载到目标系统中，然后在 WindSh 下使用 sp 命令以及缺省的参数生成一个任务(图 3-60 中的 slu5)，此时的堆栈使用情况如图 3-60 所示。

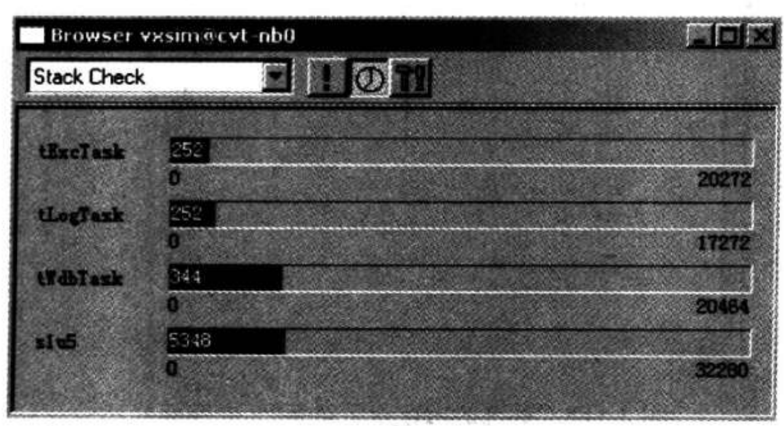


图 3-60　堆栈信息(无溢出)

随后修改代码。将函数中的字符串大小修改为 40 960，编译后再下载下去(下载前应先删除创建的任务)；同样使用 sp 生成一个任务(图 3-61 中系统将该任务命名为 slu6)，再检查堆栈，则堆栈溢出情况如图 3-61 所示。

从图 3-61 中可以看到任务 slu6 的堆栈使用量为 49 988，而堆栈的预设值仅为 32 280，

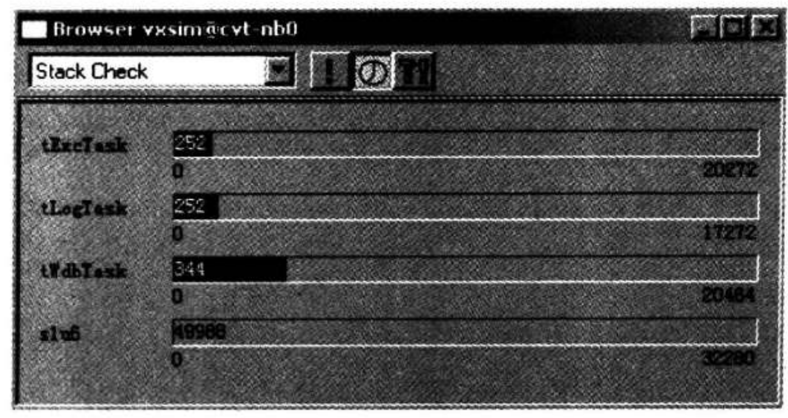


图 3-61 堆栈信息(堆栈溢出)

因此导致溢出。

(4) CPU 利用率

假设有 3 个不同优先级的任务 slu10(低优先级)、slu11(中优先级)和 slu12(高优先级)；任务 slu10 和 slu12 使用同一个信号量来访问共享资源，而 slu11 无须使用共享资源。任务 slu10 获得权限后开始对共享资源进行访问。如果此时任务 slu11 也开始运行，那么由于 slu11 的优先级高于 slu10，就会像图 3-62 中显示的一样 slu11 独占 CPU。最后导致 slu10 无法释放信号量，slu12 也无法运行，产生了优先级反转的现象。

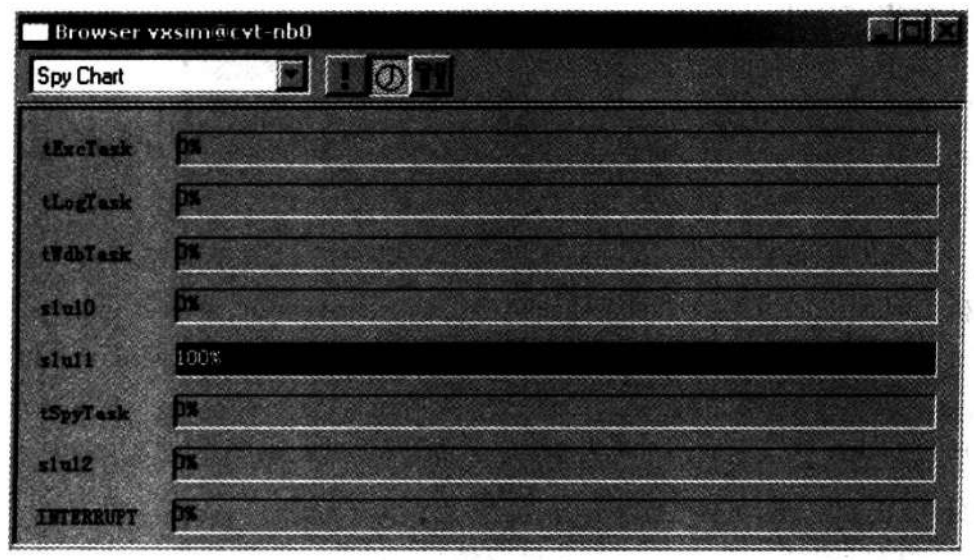


图 3-62 系统中任务的 CPU 使用情况

一旦发生了优先级反转，一方面可使用 SPY 来检查 CPU 的占用情况；另一方面可检查那些具有高优先级而处于挂起状态的任务，检查挂起的具体信号，再通过 Object Information 查

看该信号被哪个任务所占用，分析是否产生了优先级反转，并根据分析的结果对程序进行适当调整。

3.4.6 软件逻辑分析器

软件逻辑分析器(WindView)是一个图形化的动态诊断和分析工具(如图 3-63 所示)，主要向开发者提供目标机硬件上实际运行的应用程序的许多详细情况。它以图形化的方式非常明确地展示了中断处理情况、任务的 CPU 占用情况以及信号量的使用情况等，是一个分析系统整体性能的有效工具。若要使用该分析工具来分析系统性能，则需在内核配置中添加 WindView 组件。

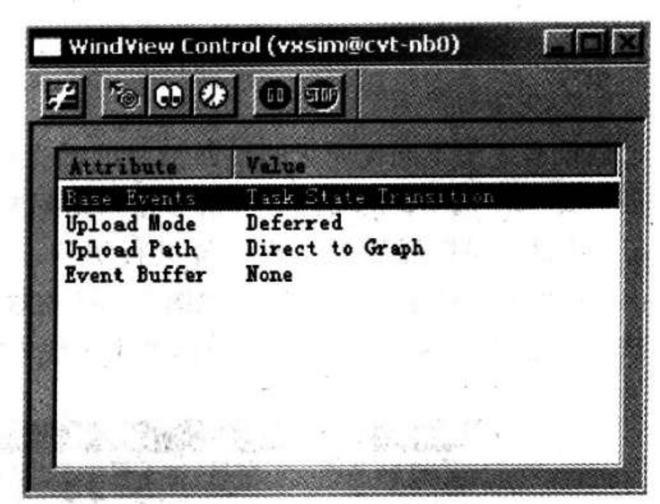


图 3-63 WindView 工具

选择 Tools→WindView→Launch，可以启动 WindView 工具，但使用前还需对其进行简单的配置。单击 WindView 控制窗口上的按钮，弹出如图 3-64 所示窗口。共有 3 个类别可以配置，分别是 Context Switch、Task State Transition 和 Additional Instrumentation。配置如下：在 Base Events 下按列表框中选择 Additional Instrumentation，并选择一些库产生 Log 事件。当选中某个库时，该库中所有对象的使用情况都会被记录下来。例如选中 semLib，则无论是内核还是应用模块，只要使用了 semaphore 对象，就会产生记录。

在图 3-64 的左下角有个 Properties 按钮，主要用于设置记录的缓冲大小以及记录上载的路径。分析工具定义了几种上载路径，分别是 Direct to Graph、Socket via TSFS、Socket via TCP/IP、File via TSFS、File via NFS，以及 3 种模式 Deferred、Post-Mortem 和 Continuous。一般选择 Direct to Graph 和 Deferred，如图 3-65 所示。

配置完毕后，在图 3-63 中单击按钮来启动该服务，再单击 Upload 按钮即可看到一个图形化的分析图，如图 3-66 所示。

从图 3-66 可以清楚地看出各任务的执行情况，以及信号量的等待情况。图中有一些有用的符号(如 、 和 等)，将鼠标停留在这些符号上时，系统会自动给出相应的提示。

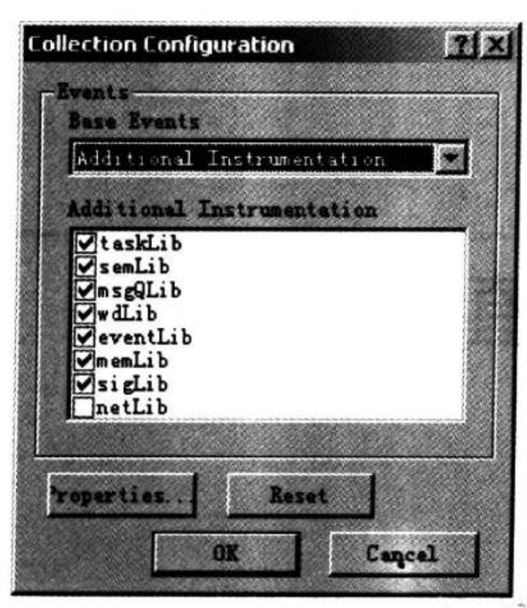


图 3-64　选择事件源

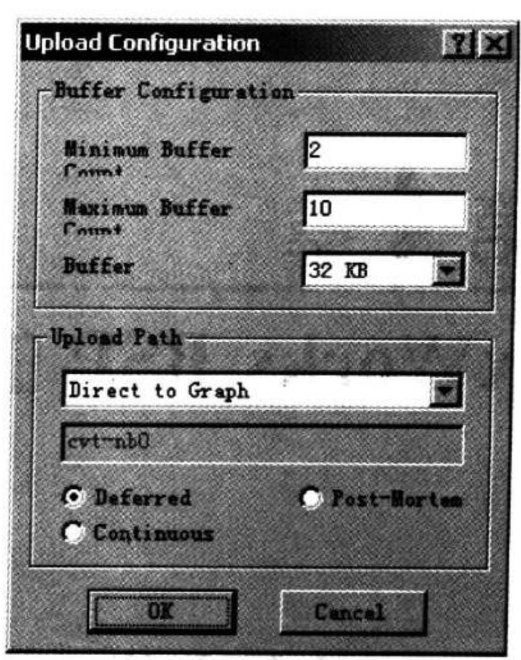


图 3-65　配置事件报告属性

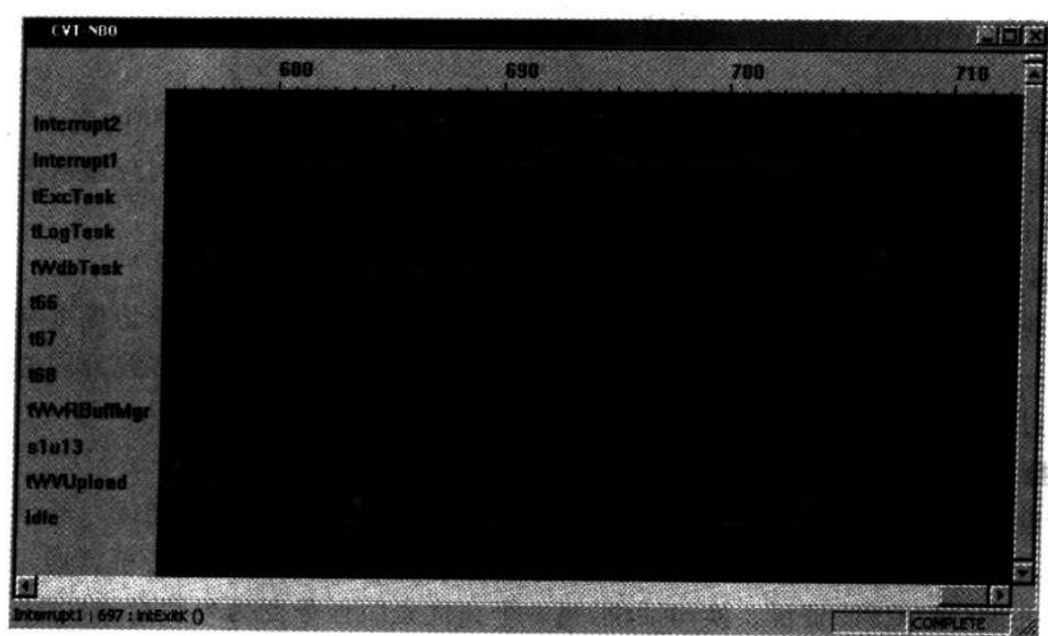


图 3-66　软件逻辑分析图

第 4 章

VxWorks BSP 的移植

4.1 VxWorks 内核的特点及 BSP 简介

4.1.1 VxWorks 内核的特点

操作系统的实时性是一个相对的概念，一般指在相同环境下，使用相同的输入，会在规定的时间内得到正确的响应。一个实时操作系统内核需要满足许多特定实时环境所提出的基本要求，包括：

(1) 多任务

由于现实世界的事件具有异步性，因此能够运行许多并发进程或任务是很重要的。多任务提供了一个较好的对现实世界的模拟，它允许对应于许多外部事件的多线程同时执行。系统内核通过适当的策略为这些任务分配 CPU，从而获得并发性。

(2) 抢占调度

现实世界的事件具有继承的优先级，在分配 CPU 时要注意这些优先级。基于优先级的抢占调度，任务都被指定了优先级；在能够执行的任务(未被挂起或正在等待资源)中，优先级最高的任务被分配 CPU 资源。换句话说，当一个高优先级任务变为可执行态，它会立即抢占当前正在运行的较低优先级的任务。

(3) 任务间的通信与同步

在一个实时系统中，可能有许多任务作为应用的一部分执行。系统必须提供这些任务间快速且功能强大的通信机制，内核也要提供同步机制，来有效地共享不可抢占的资源或临界资源。

(4) 任务与中断之间的通信

尽管现实世界的事件通常以中断方式到来，但为了提供有效的排队、优先级，减少中断延时，我们通常希望在任务级处理相应的工作。这就需要在任务级与中断级之间进行通信，完成事件的传递。

VxWorks 就是一个基于抢占式的实时操作系统，已被广泛地应用到许多行业。VxWorks 操作系统具有很多优点，例如：

(1) 高度的可靠性

操作系统的用户希望在一个工作稳定、可以信赖的环境中工作，所以操作系统的可靠性是用户首先要考虑的问题；而稳定、可靠一直是 VxWorks 的一个突出优点。自从对中国的销售解禁以来，VxWorks 以其良好的可靠性在中国赢得了越来越多的用户。

(2) 优秀的实时性

实时性是指能够在限定时间内实现规定的功能，并对外部的异步事件作出响应的能力。实时性的强弱是以实现规定功能并作出响应的时间的长短来衡量的。VxWorks 的实时性很好，其系统本身的开销很小，进程调度、进程间通信以及中断处理等系统公用程序精练而有效，因而造成的延迟很短。VxWorks 提供的多任务机制中对任务的控制采用了优先级抢占(Preemptive Priority Scheduling)和轮转调度(Round - Robin Scheduling)机制，充分保证了可靠的实时性，使同样的硬件配置能满足更强的实时性要求，为应用开发留下了更大的余地。

一般可把操作系统分为分时操作系统和实时操作系统，区别在于操作系统能否满足实时性要求。分时操作系统按照相等的时间片调度进程，轮流运行；由调度程序自动计算进程的优先级，而不是由用户控制进程的优先级。这样的系统无法实时响应外部异步事件。而实时操作系统能够在限定的时间内实现所规定的功能，并能在限定的时间内对外部的异步事件作出响应。分时系统主要应用于科学计算和一般实时性要求不高的场合；实时性系统主要应用于过程控制、数据采集、通信以及多媒体信息处理等对时间敏感的场合。

(3) 灵活的可裁剪性

用户在使用操作系统时，并不是操作系统中的每个部件都会用到。例如：图形显示、文件系统以及一些设备驱动在某些嵌入系统中往往并不需要。VxWorks 由一个体积很小的内核以及一些可根据需要进行定制的系统模块组成。VxWorks 内核最小为 8 KB，即使加上其他必要模块，所占用的空间仍然较小，且不失其实时、多任务的系统特征。基于其高度灵活性，用户可以很容易地对 VxWorks 进行定制或适当开发，来满足自己的实际应用需要。

正是上述优点使得 VxWorks 成为目前世界上应用最为广泛的嵌入式操作系统。

VxWorks 内核支持多种处理器平台，包括常见的 ARM、XScale、x86、MIPS、PPC 和 ColdFire 等。不同的目标系统需要选用不同的编译工具，同时由于嵌入式系统硬件的差异性，为了使 VxWorks 内核能够可靠地运行在目标机上，还须做一些移植工作。

下面将以武汉创维特信息技术有限公司的 JX2410 系列为例，来说明开发的一些过程。

该系统采用 Samsung 公司的 S3C2410 处理器,板载 32 MB NorFlash、16 MB NandFlash 以及 64 MB SDRAM,网卡芯片采用 RTL8019;另外在系统上还预留了两个扩展槽。要将 VxWorks 内核移植到该系统上,首先需要安装 Tornado II for ARM 的集成开发环境以及编译工具,然后才能在上面进行 BSP 的移植和调试。

4.1.2 VxWorks 的主要功能和结构

1. VxWorks 操作系统的主要功能

目前 VxWorks 操作系统的版本为 VxWorks 5.5,其核心功能模块主要有:

- 微内核 wind;
- 任务间通信机制;
- 先进的网络支持;
- 功能强大的文件系统和 I/O 管理;
- POSIX 标准实时扩展;
- C++以及其他标准支持。

这些核心功能可与 WindRiver 系统的其他附件,以及超过 400 个 Tornado II 合作伙伴的产品很好地结合在一起。

2. VxWorks 操作系统的基本结构

VxWorks 操作系统主要由以下 5 部分构成:板级支持包 BSP、微内核 wind、网络系统、文件系统以及 I/O 系统。VxWorks 内核只占用了很小的存储空间,并可高度裁剪,从而保证了系统以较高的效率运行。

(1) 板级支持包 BSP

板级支持包 BSP(Board Support Package)对各种板卡的硬件功能提供了统一的软件接口,包括硬件初始化、中断的捕捉和处理、硬件时钟和定时器管理、内存地址映射,以及内存分配等。每个板级支持包还包括一个 ROM 启动(Boot ROM)或其他启动机制。

(2) 高性能的实时操作系统核心 wind

VxWorks 的核心,被称为 wind,包括多任务调度(采用优先级抢占方式)、任务间的同步、进程间通信机制以及中断处理、看门狗和内存管理机制。一个多任务环境允许实时应用程序以一套独立任务的方式构建,每个任务都拥有独立的执行线程及其自己的一套系统资源。进程间通信机制使得这些任务的行为同步、协调。

wind 使用中断驱动和优先级的方式,缩短了上下文切换的时间和中断的时延。在 VxWorks 中,任何例程都可被启动为一个单独的任务,拥有其自己的上下文和堆栈。还有一些其他任务机制可使任务挂起、继续、删除、延时或改变优先级。

wind 核提供信号量作为任务间同步和互斥的机制。在 wind 核中有几种类型的信号量，分别针对不同的应用需求：二进制信号量、计数信号量、互斥信号量和 POSIX 信号量。所有这些信号量都是快速且高效的，它们除了被应用在开发设计过程中外，还被广泛地应用于 VxWorks 高层应用系统中。对于进程间通信，wind 核也提供了诸如消息队列、管道、套接字和信号等机制。

(3) 网络设施

网络设施提供了对其他网络和 TCP/IP 网络系统的"透明"访问，包括与 BSD 套接字兼容的编程接口、远程过程调用(RPC)、SNMP(可选项)、远程文件访问(包括客户端和服务端的 NFS 机制以及使用 RSH、FTP 或 TFTP 的非 NFS 机制)，以及 BOOTP 和 ARP 代理。无论是松耦合的串行线路、标准的以太网连接，还是紧耦合的利用共享内存的背板总线，所有的 VxWorks 网络机制都遵循标准的 Internet 协议。

(4) 文件系统

VxWorks 提供的快速文件系统适合于实时系统应用。它包括几种支持块设备(如磁盘)的本地文件系统。这些设备都使用一个标准的接口，从而使得文件系统能够被灵活地在设备驱动程序上移植。

VxWorks 也支持 SCSI 磁带设备的本地文件系统；VxWorks I/O 体系结构甚至还支持在一个单独的 VxWorks 系统上同时并存几个不同的文件系统。

VxWorks 支持多种文件系统，如 dosFs、rt11Fs、rawFs 和 tapeFs 等。

另外，普通数据文件和外部设备都统一作为文件处理。它们在用户面前具有相同的语法定义，使用相同的保护机制。这样既简化了系统设计，又便于用户使用。

(5) I/O 系统

VxWorks 提供了一个快速、灵活，与 ANSI C 兼容的 I/O 系统，包括：Unix 标准的缓冲 I/O 和 POSIX 标准的异步 I/O。

VxWorks 包括以下驱动程序：网络驱动、管道驱动、RAM 盘驱动、SCSI 驱动、键盘驱动、显示驱动、磁盘驱动和并口驱动。

4.1.3 VxWorks BSP 的简介

BSP 的英文全称为 Board Support Package，即板级支持包。其作用是针对特殊的硬件平台，为 VxWorks 内核提供操作的接口。它与内核、驱动程序以及应用程序的关系参见图 4-1。

从图 4-1 可以看出，BSP 的作用实际上就是在内核和硬件之间建立一个桥梁，以便进行数据交换。硬件驱动程序和 BSP 有着密切的关系，硬件驱动可分为常规驱动和 BSP 类型的驱动。常规驱动(如网卡驱动)可以很方便地从一个平台移植到另一个平台；而 BSP 类型的驱动

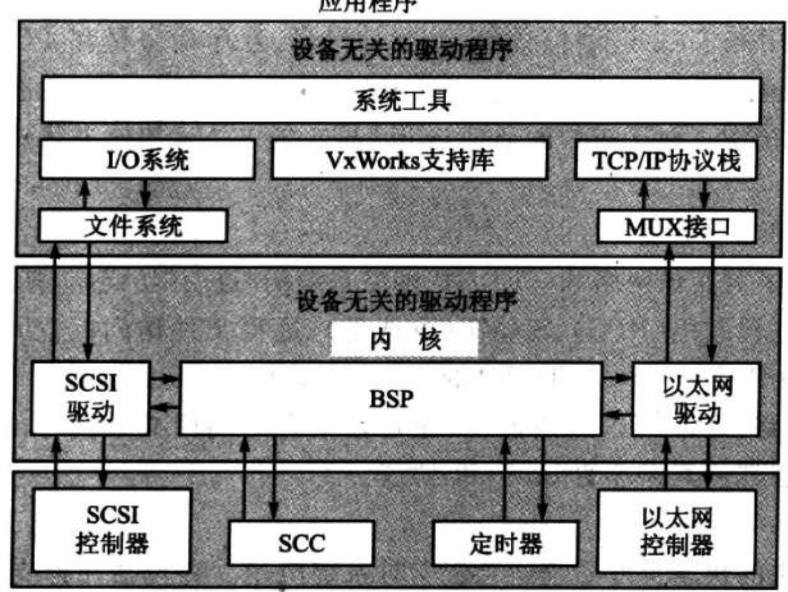


图 4-1 VxWorks 内核与 BSP

(如中断控制器驱动)则与硬件系统相互依赖。两个不同类型平台的中断控制器驱动程序是难以兼容并相互移植的。这种划分并不是绝对的，应该说无论是 BSP 还是设备驱动都是 BSP 开发人员或项目组需要完成的工作。

BSP 的开发大体上可以分为 5 个阶段：

① 配置开发环境；

② 编写 BSP 初始化代码；

③ 明确硬件的具体情况，配置一个最小的内核；

④ 启动并测试上一步创建的最小内核；

⑤ 编写其他驱动。

首先要安装开发环境，选择编译工具(例如基于 ARM 的硬件平台须选择与 ARM 相关的编译器)和调试工具，确定内核的下载机制(通过 Flash 加载或通过引导程序加载等)。

开发环境建立后，即可开始实际的编写工作。需要先编写一些初始化代码。这部分代码会在内核启动之前被执行，一般采用汇编编写。其作用就是初始化处理器及 RAM 空间，最后会把控制权交给操作系统初始化例程 usrInit()。

为了使内核能够运行起来，首先需要编写的就是中断处理程序。因为多任务操作系统依赖于定时中断，所以首要工作就是编写相应硬件的中断处理程序，包括：中断的安装、分发、使能和禁止，随后需要加入定时器的驱动。为了方便调试，还会加入串口驱动，以便启动 WDB 调试工具。

最小内核生成以后，需要对其进行测试。可以借助多种手段来完成，例如使用硬件仿真器

来下载和调试该内核。由于该内核只是一个最小系统，一般都不包含诸如网络协议、文件系统等组件，因此调试的重点就在于中断响应、定时器驱动和串口驱动这些方面了。

最后一个阶段，编写系统所用到的一些驱动，如网卡驱动和 Flash 文件系统等。这一阶段可在上面生成的最小内核上完成，同时可以借助于 WDB 工具来进行调试。

4.1.4 VxWorks BSP 的文件组织

当安装 VxWorks BSP 开发包后，BSP 相关的源代码会被拷贝到 $(WINDBASE)\target 目录下，该目录的组织见图 4-2。在编写 BSP 时，重点关心以下两个目录下的文件：target\config\all 和 target\config\bspName。严格地说，target\config\all 目录下的文件并不是 BSP 的一部分，但所有的 BSP 都会引用在该目录下定义的模块。下面具体看一下各目录的文件：

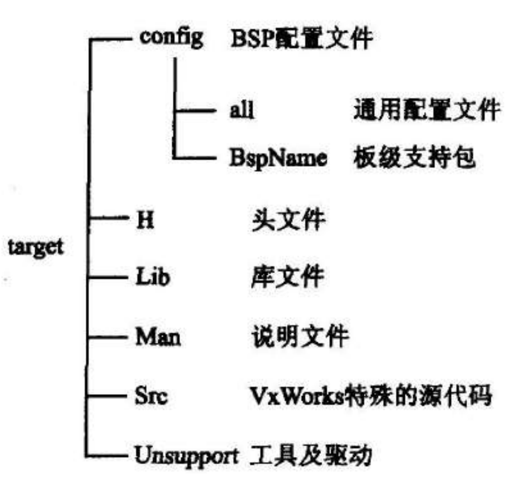


图 4-2 VxWorks BSP 文件组织

(1) target\config\all

target\config\all 目录下的文件是 VxWorks 整个软件开发工具的一部分，一般无须修改，特别是"configall.h"。这些文件对于绝大多数 BSP 都是共用的，会在创建工程时被引用一次；创建完毕后，对其修改不会影响到先前创建的工程，而只会影响以后的 BSP。该目录包含以下 6 个文件：

- bootConfig.c：引导 ROM 映像的主初始化文件；
- bootInit.c：ROM 初始化的第二步；
- configAll.h：通用的内核配置文件；
- dataSegPad.c：数据段起始位置的填充数据，通过添加这些数据来保证数据段和代码段不在同一个虚拟页中；
- usrConfig.c：VxWorks 映像的初始化代码；
- version.c：VxWorks 版本的描述信息。

(2) target\config\comps\VxWorks

该目录下的文件是 VxWorks 内核的基本组件描述文件(CDF 文件)。关于组件的一些用法在后续章节中会详细说明。

(3) target\config\comps\src

该目录下的文件代表了与内核组件相关的配置文件。

(4) target\config\BspName

该目录下的文件就是所要编写的 BSP 文件，如图 4－2 所示。这些文件往往会与系统硬件密切相关。

4.2 VxWorks 的引导过程

在编写 BSP 之前，首先需要了解整个系统的启动过程。VxWorks 内核可以分为 3 种：可加载类型内核(Loadable Image)、带有 ROM 启动功能的压缩或不压缩内核(ROM-based Image)和驻留 ROM 的内核(ROM-resident Image)。这 3 种类型的映像组织是不一样的，所以启动过程也有些区别。

可加载类型内核首先需要烧写 Flash 中的启动代码(VxWorks boot、U-boot 或其他引导程序)。该启动代码一般运行在 ARM_HIGH_ADRS 处，通过它将 VxWorks 内核加载到 RAM 中(RAM_LOW_ADRS 处)，再开始执行，如图 4－3 所示。

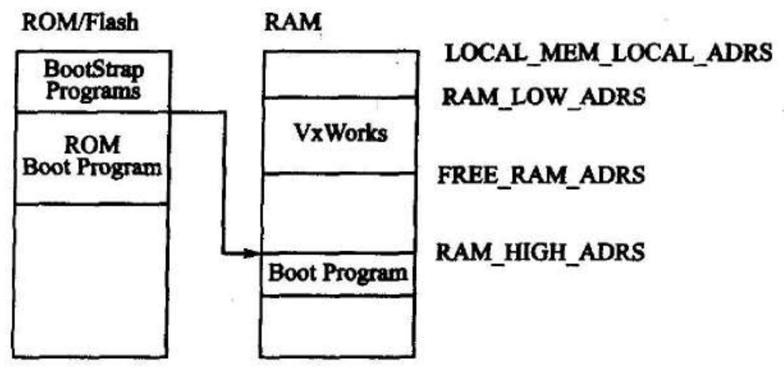


图 4－3 可加载类型 VxWorks 内核

带有 ROM 启动功能的压缩或不压缩内核可以自动执行上述拷贝及解压工作；在拷贝和解压执行完毕后，与可加载类型内核的执行流程是一样的，如图 4－4 所示。

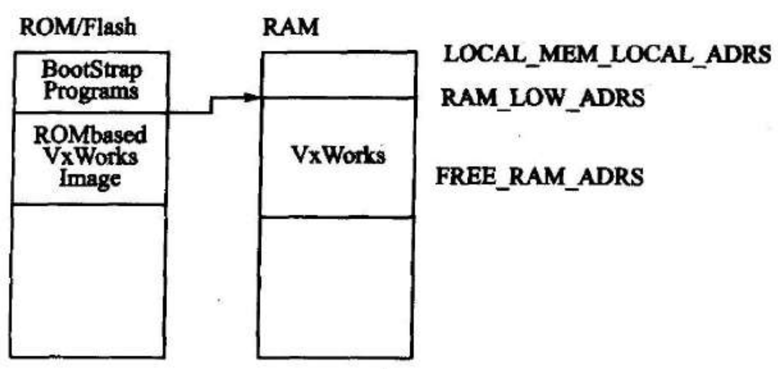


图 4－4 带 ROM 启动功能的 VxWorks 内核

驻留 ROM 的内核，与带有 ROM 启动功能的不压缩内核相似，惟一的区别就是它仅拷贝数据段，而内核代码则保留在 Flash 中并在 Flash 中执行，如图 4-5 所示。

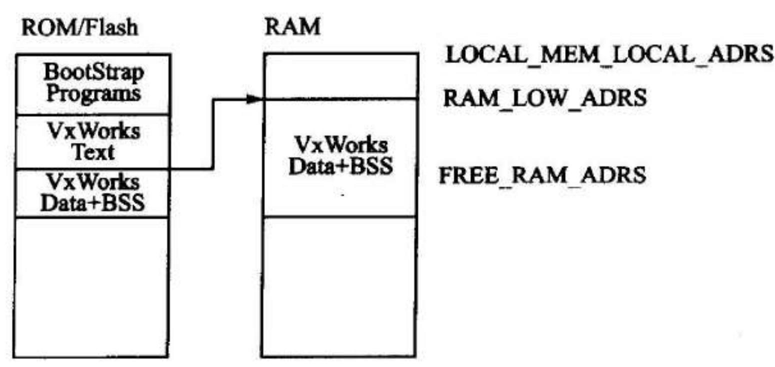


图 4-5 驻留 ROM 的 VxWorks 内核

下面以带有 ROM 启动功能的不压缩内核为例，说明整个系统的启动过程。

基于 ARM 处理器的系统，上电后会自动加载并运行位于 0 地址的指令。在 0 地址处通常会放置一条跳转指令，使其跳转到_romInit( )(初始化的入口)：

① romInit()进行处理器模式的设置，关闭中断，初始化内存以及一些必要的硬件配置；

② romInit()在执行完上述初始化后，跳转到 romStart()；

③ romStart()将 ROM 映像(启动代码或 VxWorks 内核映像)拷贝到 RAM 中；

④ 如果 ROM 映像中不含有 VxWorks 内核，那么启动代码要负责将 VxWorks 内核加载到 RAM 中。

接下来要开始运行 VxWorks 内核的入口程序 sysInit()：

① sysInit()实现一些与 romInit()例程类似的功能后，开始调用 VxWorks 内核的第一个 C 例程 usrInit()。

② usrInit()会根据 BSP 的配置，最终完成整个内核的前期初始化工作。

③ 在 usrInit()的最后，由 kernelInit()激活多任务环境，并且创建一个任务来安装设备驱动程序。同时启动设备，初始化 VxWorks 系统库，调用应用程序。

图 4-6 是 BootRom 的执行流程图，图 4-7 是 VxWorks 内核映像的启动流程图。

下面进一步说明整个启动过程中涉及的各例程：

(1) romInit( )

系统上电后，就开始执行位于 romInit.s 中的 romInit( )例程。该例程的主要工作就是初始化 CPU，配置处理器的工作模式，以及配置存储器。在执行 romStart( )之前需在该例程中完成一些前期准备工作，即执行必需的硬件初始化，关闭中断和看门狗，清除 Cache，配置存储空间，保证内存地址单元从 LOCAL_MEM_LOCAL_ADRS 到(LOCAL_MEM_LOCAL_ADRS+LOCAL_MEM_SIZE)能够可靠地读/写；最后配置启动的参数(冷启动还是热启动)，

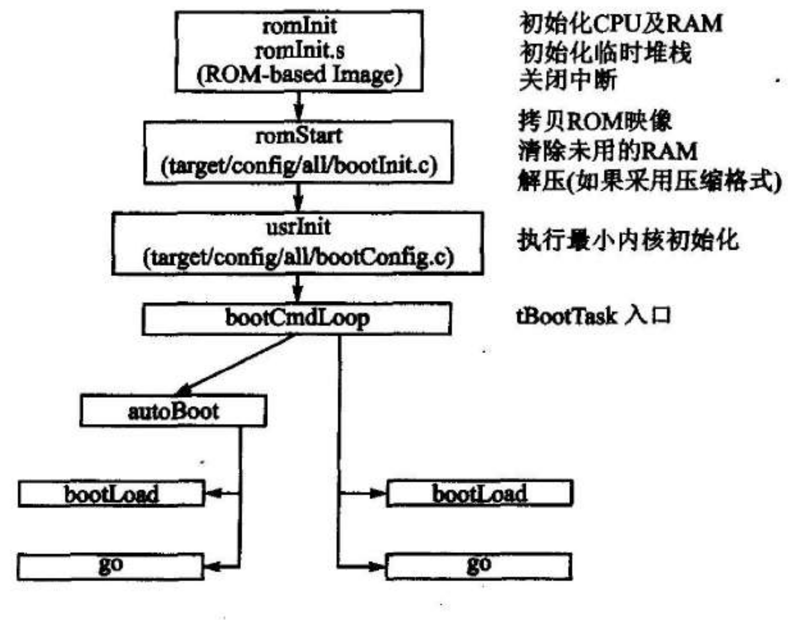


图 4-6 BootRom 执行流程图

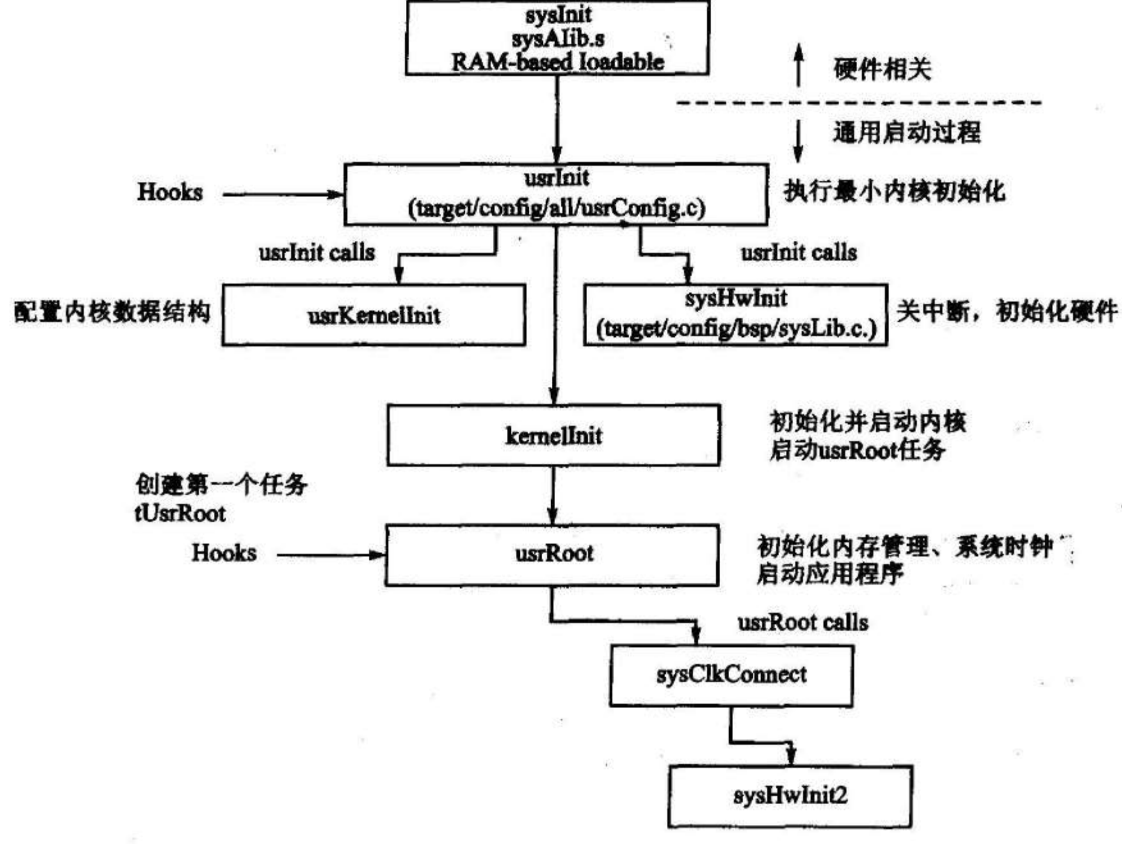


图 4-7 VxWorks 内核映像的启动流程图

并调用 bootInit. c 中的 romStart( )。另外，还需在该例程中进行一些无法使用 C 代码操作的初始化，而更完整的初始化功能可以放在 target/config/sysLib. c 中的 sysHwInit( )例程中实现。

(2) romStart( )

该例程位于 bootInit. c 中，主要功能是将 Flash 中的启动代码拷贝到 RAM 中，最后调用 ROM 或 RAM 中的 usrInit( )例程。拷贝的内容与生成的内核类型有关，但通常都需将数据段拷贝到 RAM 中。如果采用的是压缩格式，则除了将压缩代码拷贝到 RAM 中外，还需对其进行解压。在该例程的最后，调用 sysALib. s 中 sysInit( )；调用时会将启动类型作为参数传递给 sysInit( )。

(3) sysInit( )

该例程是内核映像的入口。如果不使用 bootRom 来启动 VxWorks，那么该例程就是整个映像的第一条指令，所以它必须是 sysALib. s 的第一个例程。其主要工作是：清除 Cache 中的数据，设置中断向量表，清除未决的中断，设置处理器的各寄存器，最后使用 bootType 参数调用 usrInit( )。该例程中有些工作与 romInit( )是相同的，目的是保证内核映像在运行与冷启动时，软硬件环境高度一致。因为有时可能不通过 BootRom 来启动 VxWorks，而是使用其他方式来加载内核，这样在 sysALib. s 中就必须有一些与 romInit( )相同的初始化代码。

(4) usrInit( )

该例程位于 usrConfig. c 中，用于完成最后的 CPU 初始化部分，并且关闭系统的其他硬件设备，为内核的初始化和启动准备一个单线程的环境(无中断，无任务，无线程)。一般该例程由 WindRiver 提供，而且无须修改，但在 usrInit( )中会调用函数 sysHwInit( )。sysHwInit( )例程位于 sysLib. c 中，是与系统硬件密切相关的，也是须按照相应的硬件进行设计的。需要注意的是，虽然该例程使用 C 代码编写，但由于此时操作系统的多任务机制还没正确地建立起来，所以很多工具(如信号等)都不能在该例程中使用。该例程非常重要，其具体工作包括：

- 初始化 Cache；
- 清零 BSS 段；
- 初始化中断向量；
- 调用 sysHwInit( )(在该例程中初始化硬件时，并不将硬件的中断与向量表关联起来，关联工作会在后面的 sysHwInit2( )中完成)；
- 调用 kernelInit( )。

(5) kernelInit( )

该例程由风河公司以库的形式提供，最终会创建 VxWorks 的第一个任务 usrRoot( )，并由 usrRoot( )来完成系统的最后初始化。

(6) usrRoot( )

该例程位于 usrConfig. c 中，用于完成系统最后的初始化以及所有硬件的初始化。在每个 BSP 目录下，一般都有一个 usrConfig. c 的拷贝。可以修改该 BSP 下的 usrConfig. c 文件，但建议通过修改 config. h 中的宏定义方式来改变 usrConfig. c 的行为。

当系统的多任务环境准备就绪后，首先会调用 sysClkConnect( )例程，而 sysClkConnect( )又会立即调用 sysHwInit2( )来执行在 sysHwInit( )中未完成的硬件初始化，如中断使能 intConnect( )；然后 usrRoot( )会进一步初始化定时器，并使用 sysClkEnable( )来启动定时器。一旦定时器被驱动起来，这个多任务的环境也就真正运行起来了。至此，一些内核模块、I/O 系统也都处于就绪状态，剩下的工作就是初始化并启动应用程序。

4.3 VxWorks BSP 的移植

4.3.1 Makefile

对 VxWorks 内核的整个启动过程以及系统的布局有了明确的认识之后，即可开始 BSP 的编写工作。BSP 通常会包含一些 C 代码、头文件、汇编代码、makefile 和配置文件(config. h)等，如表 4-1 所列。在硬件驱动和操作系统之间，BSP 一般都会通过一些约定的接口(驱动程序的标准接口)来进行数据的交换。当系统启动时，BSP 例程会使用一些系统调用来登记这些设备驱动程序；在系统启动后，操作系统即可根据这些登记信息，将一些操作与具体的硬件关联起来。

表 4-1 VxWorks BSP 相关的文件

文件名	说 明
README	描述整个 BSP 的一些使用说明
Makefile	编译和链接整个 BSP 的规则，如编译工具的选择、编译选项和包含文件路径等；有一些参数必须在该文件中定义
config. h	配置整个操作系统的一些宏定义
bspName. h	包含一些不能动态配置的信息，如 I/O 地址定义、中断向量定义以及设备相关的寄存器地址及编码等
romInit. s	包含 romInit()例程的代码及被 romInit()调用的子函数
sysALib. s	包含 sysInit() 例程
sysLib. c	包含一些与硬件密切相关的例程，如 sysHwInit( )和 sysHwInit2( )等
sys*Dev*. c	设备驱动初始化程序
target. nr	Tornado 2. x 的 BSP 描述文件，使用 nroff 标记语言

另外还包括几个全局变量，如表4-2所列。

表4-2 VxWorks BSP相关的全局变量

变量名称	说明
sysPhysMemDesc	描述存储器及设备的地址分配，以及存储器映射
sysPhysMemDescNumEnt	上述描述信息的条目数
sysBootLine	启动参数的地址
sysExcMsg	系统启动阶段的错误报告地址
sysFlags	启动参数

具体来讲，至少需要实现的函数如表4-3所列。

表4-3 VxWorks BSP至少需要实现的函数

函数名称	说明
sysBspRev( )	返回BSP的版本号
sysClkConnect( )	安装时钟中断处理例程
sysClkDisable( )	关闭系统的时钟中断
sysClkEnable( )	使能系统的时钟中断
sysClkInt( )	时钟中断处理例程
sysClkRateGet( )	获取当前系统时钟中断的频率
sysClkRateSet( )	设置系统时钟中断的频率
sysHwInit( )	系统硬件初始化，通常在该例程中关闭所有硬件中断
sysHwInit2( )	初始化中断驱动，挂接外设中断
sysMemTop( )	返回存储器高端物理地址
sysModel( )	返回目标系统的名称
sysSerialChanGet( )	获取指定串口设备的描述符
sysSerialHwInit( )	初始化串口设备
sysSerialHwInit2( )	安装并启动串口设备的中断
sysToMonitor( )	将控制权转交给ROM监控程序

表4-3只列出了一个函数框架，在实现时往往还需要一些子函数，这取决于设计者的实现思路。在后续章节中将围绕BSP的编写过程来讲解各函数的编写方法。

首先了解一下BSP中Makefile文件的编写。无论使用GNU编译工具还是DIAB编译工具，都需要该文件。实际上它是产生整个内核构造方法的依据。程序清单4-1是一个典型的Makefile文件。

程序清单 4-1　JX2410 VxWorks BSP Makefile

```
# 描述信息
# ……
CPU              = ARMARCH4
TOOL             = gnu
EXTRA_DEFINE     = -Wcomment -DCPU_920T \
                   -DARMMMU = ARMMMU_920T -DARMCACHE = ARMCACHE_920T
TGT_DIR = $(WIND_BASE)/target
include $(TGT_DIR)/h/make/defs.bsp
# 不要在此之前进行 make 工具相关的宏定义，因为可能被一些类似于 bsp.def
# 文件中的宏定义覆盖

TARGET_DIR    = JX2410
VENDOR        = CVTECH
BOARD         = JX2410
RELEASE       += bootrom.bin
# ROM_TEXT_ADRS、ROM_SIZE 和 RAM_HIGH_ADRS 在 config.h 和 Makefile 中都有定义
# 但这些定义必须保持一致
ROM_TEXT_ADRS     = 00000000            # ROM 入口地址
ROM_SIZE          = 02000000            # ROM 的大小
RAM_LOW_ADRS      = 30010000            # RAM 低位地址，一般为程序的入口
RAM_HIGH_ADRS     = 33f00000            # RAM 高位地址
VMA_START         = 0x $(ROM_TEXT_ADRS)

# 只有不带调试符号的 VxWorks 二进制目标码才适合于烧写到 Flash 器件中
# 带调试符号的目标文件可转换为这种类型，转换的规则可在此描述

bootrom.bin: bootrom
    -@ $(RM) $@
    $(EXTRACT_BIN) -O binary bootrom $@
bootrom_res.bin: bootrom_res
    -@ $(RM) $@
    $(EXTRACT_BIN) -O binary bootrom_res $@
bootrom_uncmp.bin: bootrom_uncmp
    -@ $(RM) $@
```

```
    $ (EXTRACT_BIN) -O binary bootrom_uncmp $@

VxWorks_rom.bin: VxWorks_rom
    -@ $(RM) $@
    $ (EXTRACT_BIN) -O binary VxWorks_rom $@

VxWorks.st_rom.bin: VxWorks.st_rom
    -@ $(RM) $@
    $ (EXTRACT_BIN) -O binary VxWorks.st_rom $@

VxWorks.res_rom.bin: VxWorks.res_rom
    -@ $(RM) $@
    $ (EXTRACT_BIN) -O binary VxWorks.res_rom $@

VxWorks.res_rom_nosym.bin: VxWorks.res_rom_nosym
    -@ $(RM) $@
    $ (EXTRACT_BIN) -O binary VxWorks.res_rom_nosym $@

#不要在此之后添加一些宏定义，否则将会导致文件依赖关系的错误
MACH_EXTRA          = rtl8019End.o
include $(TGT_DIR)/h/make/rules.bsp
```

在 Makefile 文件中主要包括以下几方面内容：

CPU：描述目标板的处理器类型。JX2410 使用的处理器 S3C2410 是一个基于 ARM 核的 CPU，版本为 ARMVv4，所以在这里指明 CPU 类型为 ARMARCH4。

TOOL：该参数用于选择编译工具。VxWorks 可以使用 GNU 和 DIAB 两种编译工具，这里指定使用 GNU 编译器来编译目标代码。

EXTRA_DEFINE：附加的编译选项。例如“-Wcomment -DCPU_920T -DARMMMU=ARMMMU_920T -DARMCACHE=ARMCACHE_920T”，其中：

— Wcomment：如果注释起始序列“/*”出现在注释中，则编译器就发出警告；

— DCPU_920T：定义 CPU 体系为 920T；

— DARMMMU=ARMMMU_920T：定义 MMU 类型为 ARMMMU_920T；

— DARMCACHE=ARMCACHE_920T：定义 Cache 类型为 ARMCACHE_920T。

这几个参数对于不同的处理器设置是不一样的，可查阅处理器文档来确定选定的处理器究竟是哪个体系结构，及其 MMU 和 Cache 属于哪一类型等。

TGT_DIR：默认设置为“$(WIND_BASE)/target”。

TARGET_DIR：默认为 BSP 所在的目录。

VENDOR：板卡生产厂商的名称。

BOARD：板卡的名称。

ROM_TEXT_ADRS：BOOT ROM 的起始地址(十六进制)，一般为 Flash 地址。

ROM_SIZE：ROM 或 Flash 的大小(十六进制)。

RAM_LOW_ADRS：VxWorks 在 RAM 中的起始地址，即入口地址。

RAM_HIGH _ADRS：非驻留 ROM 内核的启动程序加载地址。关于入口地址和高位地址的指定须参考硬件的 RAM 组织。以 JX2410 为例，其系统运行后的 RAM 分布见图 4-8。本文涉及的内存组织都将以该图描述的内容为参考。

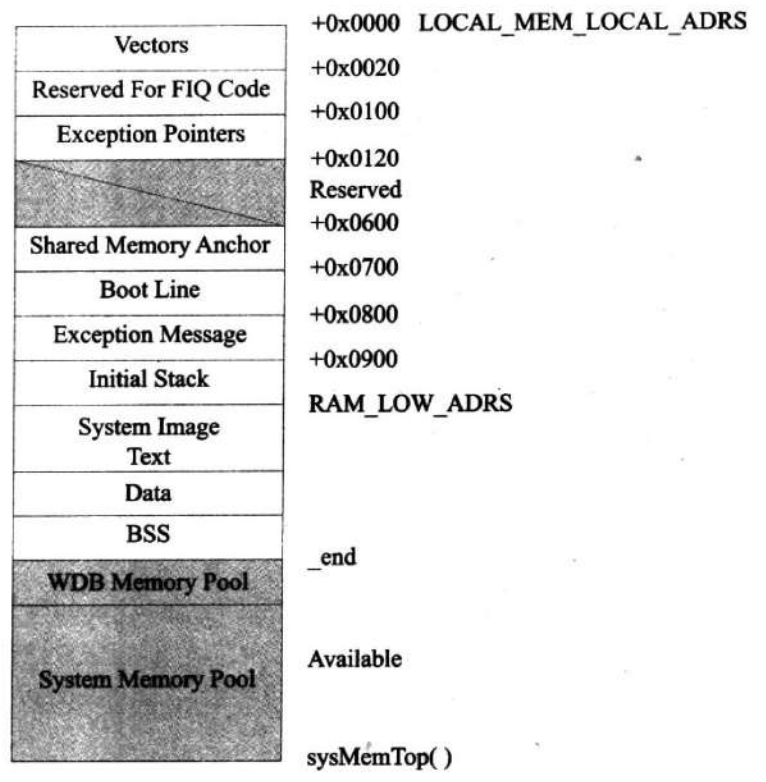


图 4-8 VxWorks 存储器参数的定义

VMA_START：一个可选参数，用于创建启动代码，如 Bootroom。注意：与 Makefile 文件中其他地址的定义不同，该地址需要以 0x 打头。

MACH_EXTRA：通过该参数可将一些附加模块与内核模块编译在一起。如果 BSP 中须使用一些不在表 4-1 中的文件，则可使用该定义将其添加到目标码中。例如程序清单 4-1中就额外定义了 3 个模块：mmu. o、2410slib. o 和 rtl8019End. o。

在上面的 Makefile 文件中，引用了两个系统提供的默认定义和编译规则，分别是："$(TGT_DIR)/h/make/defs. bsp"和"$(TGT_DIR)/h/make/rules. bsp"。

defs. bsp 的主要作用是根据不同的编译工具定义一些默认的编译参数，如 DEFAULT_RULE(默认的编译规则选项)、CFLAGS(编译参数)和 CC_INCLUDE(编译时的头文件搜索路径)等。

具体内容见程序清单 4－2。

程序清单 4－2　(TGT_DIR)/h/make/defs.bsp

```
# defs.bsp - Bsp 编译时使用的缺省编译参数
#……

# 如果在 makefile 文件中未定义编译工具,则可在此定义

ifeq     ($(TOOL_FAMILY),)
ifeq     ($(findstring gnu,$(TOOL)),gnu)
TOOL_FAMILY = gnu
else
ifeq     ($(findstring diab,$(TOOL)),diab)
TOOL_FAMILY = diab
endif
endif
endif
.SUFFIXES:  .cpp .out .cxx

# 默认情况下,编译对象为 exe
default : exe

# 多数情况下,目标码的默认规则为 VxWorks 类别所描述的规则
DEFAULT_RULE      = VxWorks

# 为了保持兼容,在此定义 BSP_NAME
BSP_NAME          =  $(TARGET_DIR)
# default flags
CFLAGS            =  $(CC_ARCH_SPEC) $(CC_COMPILER) $(CC_OPTIM) $(CC_WARNINGS) \
          $(CC_INCLUDE) $(CC_DEFINES) $(ADDED_CFLAGS) \
          $(CC_SOFT_FLOAT)
C++FLAGS          =  $(CC_ARCH_SPEC) $(C++_COMPILER) $(CC_OPTIM) $(C++_WARNINGS) \
          $(CC_INCLUDE) $(CC_DEFINES) $(ADDED_C++FLAGS) \
          $(CC_SOFT_FLOAT)
CASFLAGS          =  $(CC_ARCH_SPEC) $(OPTION_PP) $(OPTION_LANG_ASM) $(CC_INCLUDE) \
```

```
        $(CC_DEFINES)

CFLAGS_AS     = $(CC_ARCH_SPEC) $(CC_COMPILER) $(CC_OPTIM) $(CC_INCLUDE) \
        $(CC_DEFINES) $(ADDED_CFLAGS) $(OPTION_PP_AS) \
        $(OPTION_LANG_ASM)

# 与位置无关的代码需要的特殊编译参数,如 boot Init.o
CFLAGS_PIC    = $(CC_ARCH_SPEC) $(CC_COMPILER) $(CC_OPTIM) $(CC_WARNINGS) \
        $(CC_INCLUDE) $(CC_DEFINES) $(ADDED_CFLAGS) $(CC_PIC) \
        $(CC_SOFT_FLOAT)
CC_WARNINGS   = $(CC_WARNINGS_ALL)
CC_OPTIM      = $(CC_OPTIM_TARGET)
CC_INCLUDE    = $(OPTION_INCLUDE_DIR)$(UP)/h $(INCLUDE_CC) $(EXTRA_INCLUDE) \
        $(OPTION_INCLUDE_DIR)$(OPTION_INCLUDE_DIR)$(CONFIG_ALL) \
        $(OPTION_INCLUDE_DIR)$(TGT_DIR)/h \
        $(OPTION_INCLUDE_DIR)$(TGT_DIR)/src/config \
        $(OPTION_INCLUDE_DIR)$(TGT_DIR)/src/drv
CC_DEFINES    = $(OPTION_DEFINE_MACRO)CPU = $(CPU) \
        $(OPTION_DEFINE_MACRO)TOOL_FAMILY = $(TOOL_FAMILY) \
        $(OPTION_DEFINE_MACRO)TOOL = $(TOOL) \
        $(DEFINE_CC) $(EXTRA_DEFINE)
COMPILE_SYMTBL    = $(CC) $(OPTION_OBJECT_ONLY) $(OPTION_DOLLAR_SYMBOLS) $(CFLAGS)

# 一些工程管理工具需要的汇编器和 C 编译器的参数
CFLAGS_PROJECT_REMOVE = $(CC_PROJECT_OPTIMIZE_FLAGS) \
                        $(PROJECT_BSP_FLAGS_EXTRA)
CFLAGS_PROJECT_STRIPPED = $(filter-out $(CFLAGS_PROJECT_REMOVE),$(C++FLAGS))
CFLAGS_PROJECT = -g $(CFLAGS_PROJECT_STRIPPED)
CFLAGS_AS_PROJECT_STRIPPED = $(filter-out $(CFLAGS_PROJECT_REMOVE),$(CFLAGS_AS))
CFLAGS_AS_PROJECT = -g $(CFLAGS_AS_PROJECT_STRIPPED)

# 定义 ROM_LINK_ADRS 和 TGT_DIR
ifeq ($(ROM_LINK_ADRS),)
ROM_LINK_ADRS = $(ROM_TEXT_ADRS)
```

```
endif
ifeq ($(TGT_DIR),)
TGT_DIR = $(WIND_BASE)/target
endif

# 标准 BSP 模块的组成列表
MACH_DEP     = sysALib.o sysLib.o $(MACH_EXTRA) $(ADDED_MODULES)

# 链接脚本定义
include $(TGT_DIR)/h/make/defs.link

# 文档部分
DOC_FILES    = sysLib tyCoDrv sysTffs mkboot
DOC_FLAGS    = $(OPTION_DEFINE_MACRO)DOC $(OPTION_DEFINE_MACRO)INCLUDE_SCSI \
        $(OPTION_DEFINE_MACRO)INCLUDE_SHOW_ROUTINES $(EXTRA_DOC_FLAGS)

# 在此包含一些系统提供的默认规则
include $(TGT_DIR)/h/tool/$(TOOL_FAMILY)/make.$(CPU)$(TOOL)
include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)

# 编译发行版本时需要的编译过程(包括 4 部分)
RELEASE          = $(RELEASE_PRE) $(RELEASE_CMD) $(RELEASE_PRJ) $(RELEASE_POST)
RELEASE_PRE      =
RELEASE_POST     =
ifeq ($(VX_CPU_FAMILY),arm)
    RELEASE_PRJ = prj_default prj_diab
endif

ifeq ($(RELEASE_PRJ),)
    RELEASE_PRJ = prj_default
endif

# 命令行的默认参数
ifeq ($(RELEASE_CMD),)
    RELEASE_CMD = VxWorks VxWorks.st bootrom.hex
endif
```

rules. bsp 用于定义编译规则。为了理解 Make 工具如何按照指定的编译规则来编译目标，首先需要了解一些术语：

目标：需要执行的一个任务。多数情况下，它就是用户要生成的文件的名字，但也可以仅是个任务的名字。

依赖关系：两个目标之间相互依存的关系。如果修改目标 B 会造成目标 A 的修改，那么就称目标 A 依赖于目标 B，B 是 A 的先决条件。

变量：一种存储临时信息的载体。Make 中使用的变量应加上括号，如 $(TEMP)。

命令：执行任务时使用的指令，可以是一条、多条，或者没有。

规则：一条完整的规则具有以下格式：

```
目标(target) : 先决条件(prerequisites)
规则(command)
:
```

其中：只有"目标"必须要有，其他成分均可没有。一条完整的规则描述了编译一个目标的方法和依赖关系，是 Makefile 中最重要的部分。

程序清单 4-3 列出了系统提供的编译规则文件的部分内容。如果选择编译 VxWorks default，则将按照这个规则进行。

程序清单 4-3　(TGT_DIR)/h/make/ rules. bsp

```
# rules.bsp - BSP 的编译规则
# ……

################################ VxWorks ##################################
# VxWorks       - VxWorks 目标文件
# VxWorks.sym   - VxWorks 目标码的调试符号文件

VxWorks VxWorks.sym : depend.$(BSP_NAME) usrConfig.o dataSegPad.o \
        $(MACH_DEP) $(LDDEPS) $(patsubst -l%,lib%.a,$(LIBS)) $(CC_LIB)
    - @ $(RM) VxWorks VxWorks.sym
    - @ $(RM) version.o
    - @ $(RM) VxWorks.tmp ctdt.c ctdt.o
    $(CC) $(OPTION_OBJECT_ONLY) $(CFLAGS) -o version.o $(CONFIG_ALL)/version.c
    $(LD_PARTIAL) \
        -o VxWorks.tmp $(MACH_DEP) usrConfig.o version.o \
        $(LD_PARTIAL_START_GROUP) $(LD_LINK_PATH) $(LIBS) \
        $(LD_PARTIAL_END_GROUP) $(CC_LIB)
    $(NM) VxWorks.tmp | $(MUNCH) > ctdt.c
```

```
$(MAKE) CC_COMPILER="$(OPTION_DOLLAR_SYMBOLS)" ctdt.o
$(LD) $(LDFLAGS) $(LD_ENTRY_OPT) $(SYS_ENTRY) $(LD_LOW_FLAGS) \
    -o VxWorks dataSegPad.o VxWorks.tmp ctdt.o $(LD_SCRIPT_RAM)
- @ $(RM) VxWorks.tmp
$(LDOUT_HOST) VxWorks
$(LDOUT_CONV) VxWorks
$(BINXSYM) VxWorks VxWorks.sym
$(LDOUT_SYMS) VxWorks.sym
$(VXSIZEPROG) -v $(RAM_HIGH_ADRS) $(RAM_LOW_ADRS) VxWorks
```

关于 Makefile 文件更详细的说明，可参见 GNU make 的相关文档。在这一步中，仅需在了解目标板的一些配置之后，编写 Makefile 文件，而无须修改那些由系统提供的默认编译参数及规则。

4.3.2 内核配置

VxWorks 内核组件的配置可以通过 config.h 文件来定义。该文件的开始一般都会放置版本的一些声明，然后会引用 target/config/all/configAll.h。configAll.h 文件预定义了很多 VxWorks 的默认组件以及配置信息，须针对不同的硬件和不同的需求来修改这些定义。例如：在 target/config/all/configAll.h 中定义了 INCLUDE_ANSI_TIME，但如果在定制的内核中无需该组件，则可很方便地在 config.h 中使用如下语句去掉该组件：

```
#undef INCLUDE_ANSI_TIME
```

通常该文件的组织内容如下：

- BSP 版本和版本 ID 号码；
- 包含 configAll.h；
- 定义缓冲和 MMU 配置；
- 定义共享网络存储器配置；
- 定义板载 RAM 的起始地址和大小；
- 定义板载 ROM 的起始地址和大小；
- 定义非易失性存储器(NVRAM)的参数；
- 设置启动参数；
- 设置可选的邮戳定时器驱动；
- 网络设备；
- 选择 WDB 调试设备。

程序清单 4 - 4 为 JX2410 使用的 config. h 文件及其解释。

程序清单 4 -4　$ (BSP_BASE)\config. h

```
/* 文件描述 */
#ifndef INCconfigh
#define INCconfigh
#ifdef __cplusplus
extern "C" {
#endif

/* 定义 BSP 版本号 */
#define BSP_VER_1_1        1
#define BSP_VER_1_2        1
#define BSP_VERSION        "1.2"
#define BSP_REV            "/5"
#include "configAll.h"                              /* 定义 VxWorks 所有的缺省设置 */

/* 内存配置,如果定义了 LOCAL_MEM_AUTOSIZE,则 SDRAM 的大小会在 Boot 时指定 */
/* ROM_TEXT_ADRS、ROM_SIZE、RAM_HIGH_ADRS 和 RAM_LOW_ADRS 在 config.h 和 Makefile 文件中都要定
   义,且须保持一致 */
#undef  LOCAL_MEM_AUTOSIZE
#define USER_RESERVED_MEM      0                    /* 用户保留的存储空间 */
#define LOCAL_MEM_LOCAL_ADRS    0x30000000          /* RAM 的起始地址 */
#define LOCAL_MEM_BUS_ADRS      0x30000000
#define LOCAL_MEM_SIZE          (0x04000000)        /* RAM 大小为 64 MB */
#define LOCAL_MEM_END_ADRS      (LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE)
#define ROM_BASE_ADRS           0x00000000          /* Flash 基地址 */
#define ROM_TEXT_ADRS           ROM_BASE_ADRS       /* 代码在 Flash 中的起始地址 */
#define ROM_SIZE                0x00200000          /* 存放 VxWorks 的 Flash 大小 */
#define ROM_COPY_SIZE           (ROM_SIZE)
#define ROM_SIZE_TOTAL          0x02000000          /* Flash 总数 */
#define RAM_LOW_ADRS            0x30010000          /* VxWorks 入口地址 */
#define RAM_HIGH_ADRS           0x33f00000          /* BOOTROM 在 RAM 中的起始地址 */
#undef  INCLUDE_FLASH
```

```
#ifdef      INCLUDE_FLASH
#define     FLASH_SIZE              0x02000000
#define     NV_RAM_SIZE             0x00000100
#undef      NV_BOOT_OFFSET
#define     NV_BOOT_OFFSET          0
#define     FLASH_NO_OVERLAY
#define     INCLUDE_FLASH_SIB_FOOTER
#else           /* INCLUDE_FLASH */
#define     NV_RAM_SIZE             NONE
#endif          /* INCLUDE_FLASH */

/* 串口配置 */
#define     INCLUDE_SERIAL
#undef      NUM_TTY
#define     NUM_TTY                 N_SIO_CHANNELS
#undef      CONSOLE_TTY
#define     CONSOLE_TTY             0
#undef      CONSOLE_BAUD_RATE
#define     CONSOLE_BAUD_RATE       115200
#define     SERIAL_DEBUG

/* DEFAULT_BOOT_LINE: 为没有 NVRAM 的目标系统而设计,这样用户就无需在每次启动时手工输入这些
   参数,系统启动网络时 xxxEndLoad()会解释并按该定义进行加载 */
/* rtl(0,0)            RTL8019 网络设备
   cvt-nb0     主机名
   /VxWorks    主机上 TSFS 根目录下的文件
   h           主机的 IP 地址
   e           目标机的 IP 地址
   u=user pw=pswd   用户名和口令。通过网络加载、调试时,FTP 服务器和目标机的用户名和密码
   必须相同 */
#define FORCE_DEFAULT_BOOT_LINE
#define DEFAULT_BOOT_LINE \
"rtl(0,0) cvt-nb0:/VxWorks h=192.168.1.162 e=192.168.1.100 u=user pw=pswd"
#ifndef _ASMLANGUAGE
IMPORT void sysHwInit0 (void);
#endif
```

```
#define INCLUDE_SYS_HW_INIT_0
#define SYS_HW_INIT_0()          sysHwInit0 ()

/* 配置 Cache 的模式 */
#if defined(CPU_920T)  || defined(CPU_920T_T) || \
    defined(CPU_940T)  || defined(CPU_940T_T) || \
    defined(CPU_946ES) || defined(CPU_946ES_T)
#undef  USER_I_CACHE_MODE
#define USER_I_CACHE_MODE        CACHE_WRITETHROUGH
#undef  USER_D_CACHE_MODE
#define USER_D_CACHE_MODE        CACHE_COPYBACK
#endif /* defined(CPU_920T/940T/946ES) */

/* 配置 MMU 的模式 */
#if defined(CPU_720T) || defined(CPU_720T_T) || \
    defined(CPU_920T) || defined(CPU_920T_T)
#define INCLUDE_MMU_BASIC
#define INCLUDE_CACHE_SUPPORT
#endif /* defined(720T/720T_T/920T/920T_T) */

/* 去除不需要的网络驱动 */
#undef STANDALONE_NET
#undef INCLUDE_EI
#undef INCLUDE_EX
#undef INCLUDE_ENP
#undef INCLUDE_SM_NET
#undef INCLUDE_LN
#undef INCLUDE_SM_SEQ_ADDR

/* 配置内核(包括网络设备) */
#define INCLUDE_NETWORK
#define INCLUDE_END
#undef END_OVERRIDE

/* 配置 WBD 通信设备 */
#ifdef  INCLUDE_END
#undef  WDB_COMM_TYPE
```

```
#define WDB_COMM_TYPE          WDB_COMM_END
#endif  /* INCLUDE_END */

/* 配置中断模式 */
#define INT_MODE    INT_NON_PREEMPT_MODEL
#define ISR_STACK_SIZE     0x800                      /* ISR 堆栈大小 */

/* TIMESTAMP 支持 */
#undef     INCLUDE_TIMESTAMP                          /* 配置时标驱动 */
#define INCLUDE_TIMESTAMP

/* TrueFFS 支持 */
#undef     INCLUDE_TFFS
#include "JX2410.h"
#undef BSP_VTS

/* 配置一些组件 */
#ifndef INCLUDE_SHELL
#define INCLUDE_SHELL
#endif
#ifndef INCLUDE_POSIX_CLOCKS
#define INCLUDE_POSIX_CLOCKS
#endif
#ifndef INCLUDE_POSIX_TIMERS
#define INCLUDE_POSIX_TIMERS
#endif
#undef INCLUDE_WINDML                         /* 去除 WINDML 组件 */
#ifdef __cplusplus
}
#endif
#endif  /* INCconfigh */

#if defined(PRJ_BUILD)
#include "prjParams.h"
#endif
```

4.3.3 带 ROM 启动功能内核前期初始化

通常把在内核的多任务环境建立起来之前的工作称为“内核前期初始化”。参考前面提到的启动流程，该过程主要涉及以下几个例程：romInit()、romStart()、usrInit()和 usrKernelInit()。

1. romInit()

如果 VxWorks 内核的编译配置选项为 default_rom、default_romCompress 和 default_romResident 这 3 种之一时，则会首先运行 romInit()例程，参见程序清单 4-5。

程序清单 4-5　$(BSP_BASE)\romInit.s —— romInit()

```
_ARM_FUNCTION(romInit)
_romInit:
cold:
    MOV   r0, #BOOT_COLD          /* 设置启动参数 */
warm:
    B     start

    /* 设置版权信息 */
    .ascii    "Copyright 1999 - 2001 ARM Limited"
    .ascii    "\nCopyright 1999 - 2001 Wind River Systems, Inc."
    .balign 4
start:
    /* 如果是冷启动，则在执行后续操作之前延长一段时间，以便电源模块输出稳定，其他外设复位
       可靠，并处于就绪状态 */
    TEQS      r0, #BOOT_COLD
    MOVEQ     r1, #INTEGRATOR_DELAY_VALUE
    MOVNE     r1, #1
delay_loop:
    SUBS      r1, r1, #1
    BNE       delay_loop

#if defined(CPU_720T)  || defined(CPU_720T_T) || \
    defined(CPU_740T)  || defined(CPU_740T_T) || \
    defined(CPU_920T)  || defined(CPU_920T_T) || \
```

```
    defined(CPU_940T)  || defined(CPU_940T_T) || \
    defined(CPU_946ES) || defined(CPU_946ES_T)

   /* 设置 MMU 的初始状态
    *
    * MMU 控制寄存器位定义:
    *
    * bit
    *   0 M 0 MMUCR_M_ENABLE      使能/禁止 MMU
    *   1 A 0 MMUCR_A_ENABLE      使能/禁止地址对齐错误检查
    *   2 C 0 MMUCR_C_ENABLE      使能/禁止数据 Cache
    *   3 W 0 MMUCR_W_ENABLE      使能/禁止写缓冲
    *   4 P 1 MMUCR_PROG32        向前兼容 26 位地址的处理器,控制 PROG32 信号的模式
    *   5 D 1 MMUCR_DATA32        为 1 时,异常中断处理程序进入 32 位地址模式
    *   6 L 1 MMUCR_L_ENABLE      应该为 1
    *   7 B ? MMUCR_BIGEND        大小端控制 (1 为大端)
    *   8 S 0 MMUCR_SYSTEM        修改 MMU 保护
    *   9 R 1 MMUCR_ROM
    *  10 F 0 MMUCR_F             应该为 0
    *  11 Z 0 MMUCR_Z_ENABLE      应该为 0
    *  12 I 0 MMUCR_I_ENABLE      使能/禁止指令 Cache
    */

   /* mmuArmLib.h 中的 MMU_INIT_VALUE 定义
   #define MMU_INIT_VALUE (MMUCR_PROG32 | MMUCR_DATA32 | MMUCR_L_ENABLE | \
          MMUCR_ROM | MMUCR_W_ENABLE) */

   MOV r1, #MMU_INIT_VALUE
# if defined(CPU_920T) || defined(CPU_920T_T)
# if defined(INTEGRATOR_EARLY_I_CACHE_ENABLE)
   ORR r1, r1, #MMUCR_I_ENABLE            /* 使能指令 Cache */
# endif
# endif
   MCR CP_MMU, 0, r1, c1, c0, 0           /* 写 MMU CR 寄存器 */
# if defined(CPU_920T)  || defined(CPU_920T_T) || \
   defined(CPU_946ES) || defined(CPU_946ES_T)
   MOV r1, #0                             /* data SBZ */
   MCR CP_MMU, 0, r1, c7, c10, 4          /* 排空写缓冲 */
```

```
    /* 清除指令和数据 Cache */
    MCR CP_MMU, 0, r1, c7, c7, 0              /* R1 = 0 */
#endif /* defined(CPU_920T,946ES) */

#if defined(CPU_720T) || defined(CPU_720T_T) || \
     defined(CPU_920T) || defined(CPU_920T_T)
    /* 将处理器 ID 寄存器设置为 0 */
    MOV r1, #0
    MCR CP_MMU, 0, r1, c13, c0, 0
#endif /* defined(CPU_720T,920T) */
#endif /* defined(CPU_720T,740T,920T,940T,946ES) */

    /* 禁止中断并切换到 SVC 模式 */
    MRS r1, cpsr
    BIC r1, r1, #MASK_MODE
    ORR r1, r1, #MODE_SVC32 | I_BIT | F_BIT
    MSR cpsr, r1

    /* 关闭处理器的所有中断源和看门狗 */
    MOV r0,#S3C2410_WTCON                     /* 关看门狗 */
    MOV r1,#0x0
    STR r1,[r0]
    LDR r0, = S3C2410_INTMSK
    MVN r1,#0x0                               /* 屏蔽中断 */
    STR r1,[r0]
    LDR r0, = S3C2410_INTSUBMSK
    MVN r1,#0x0                               /* 屏蔽子中断 */
    STR r1,[r0]

/* 配置处理器时钟 */
#if (PLL_ON_START)
    LDR r0, = S3C2410_LOCKTIME
    LDR r1, = 0xffffff
    STR r1,[r0]
    LDR r0, = S3C2410_MPLLCON
    LDR r1, = ((M_MDIV << 12) + (M_PDIV << 4) + M_SDIV)
    STR r1,[r0]
```

```
    LDR r0, = S3C2410_CLKDIVN
    LDR r1, = ((M_HDIVN << 1) + M_PDIVN)
    STR r1,[r0]
# if (M_HDIVN)
    MRC p15,0,r0,c1,c0,0
    ORR r0,r0, # MMUCR_ASYNC
    MCR p15,0,r0,c1,c0,0
# else
    MRC p15,0,r0,c1,c0,0
    BIC r0,r0, # MMUCR_ASYNC
    MCR p15,0,r0,c1,c0,0
# endif
# endif

    /* SDRAM 初始化 */
    ADR r0, L$_sysMemConfigAdrs
    LDR r1, = S3C2410_BWSCON                /* BWSCON 地址 */
    ADD r2, r0, #52                         /* SDRAM 控制字字节数 */
SDRAM_INIT:
    LDR r3, [r0], #4
    STR r3, [r1], #4
    CMP r2, r0
    BNE SDRAM_INIT

    /* GPIO 配置 */

    /*
     * PORT A 功能定义
     * Ports   : GPA22 GPA21 GPA20 GPA19 GPA18 GPA17 GPA16 GPA15 GPA14 GPA13 GPA12
     * Signal  : nFCE nRSTOUT nFRE nFWE ALE CLE nGCS5 nGCS4 nGCS3 nGCS2 nGCS1
     * Ports   : GPA11 GPA10 GPA9 GPA8 GPA7 GPA6 GPA5 GPA4GPA3 GPA2 GPA1 GPA0
     * Signal  : ADDR26 ADDR25 ADDR24 ADDR23 ADDR22 ADDR21 ADDR20 ADDR19 ADDR18 ADDR17 ADDR16
               ADDR0
     */
    LDR r0, = S3C2410_GPACON
    LDR r1, = 0x7fffff
    STR r1,[r0]
```

```
/*
 * PORT B 功能定义
 * Ports   : GPB10 GPB9 GPB8 GPB7 GPB6 GPB5 GPB4 GPB3 GPB2 GPB1 GPB0
 * Signal  : nXDREQ0 nXDACK0 nXDREQ1 nXDACK1 nSS_KBD nDIS_OFF L3CLOCK L3DATA L3MODE nIrDATX-
           DEN Keyboard
 * Setting : INPUT OUTPUT INPUT OUTPUT INPUT OUTPUT OUTPUT OUTPUT OUTPUT OUTPUT OUTPUT
 */
LDR r0, = S3C2410_GPBCON
LDR r1, = 0x044555
STR r1,[r0]
LDR r0, = S3C2410_GPBUP              /* 禁止 GPB[10:0]的上拉 */
LDR r1, = 0x7ff
STR r1,[r0]

/*
 * PORT C 功能定义
 * Ports   : GPC15 GPC14 GPC13 GPC12 GPC11 GPC10 GPC9 GPC8 GPC7 GPC6 GPC5 GPC4 GPC3 GPC2 GPC1
           GPC0
 * Signal  : VD7 VD6 VD5 VD4 VD3 VD2 VD1 VD0 LCDVF2 LCDVF1 LCDVF0 VM VFRAME VLINE VCLK LEND
 */
LDR r0, = S3C2410_GPCCON
LDR r1, = 0xaaaaaaaa
STR r1,[r0]
LDR r0, = S3C2410_GPCUP              /* 禁止 GPC[15:0]的上拉 */
LDR r1, = 0xffff
STR r1,[r0]

/*
 * PORT D 功能定义
 * Ports   : GPD15 GPD14 GPD13 GPD12 GPD11 GPD10 GPD9 GPD8 GPD7 GPD6 GPD5 GPD4 GPD3 GPD2 GPD1
           GPD0
 * Signal  : VD23 VD22 VD21 VD20 VD19 VD18 VD17 VD16 VD15 VD14 VD13 VD12 VD11 VD10 VD9 VD8
 */
LDR r0, = S3C2410_GPDCON
LDR r1, = 0xaaaaaaaa
STR r1,[r0]
```

```
LDR r0, = S3C2410_GPDUP             /* 禁止 GPD[15:0]的上拉 */
LDR r1, = 0xffff
STR r1,[r0]

/*
 * PORT E 功能定义
 * Ports   : GPE15 GPE14 GPE13 GPE12 GPE11 GPE10 GPE9 GPE8 GPE7 GPE6 GPE5 GPE4
 * Signal  : IICSDA IICSCL SPICLK SPIMOSI SPIMISO SDDATA3 SDDATA2 SDDATA1 SDDATA0 SDCMD SDCLK
           I2SSDO
 * Ports   : GPE3 GPE2 GPE1 GPE0
 * Signal  : I2SSDI CDCLK I2SSCLK I2SLRCK
 */
LDR r0, = S3C2410_GPECON
LDR r1, = 0xaaaaaaaa
STR r1,[r0]
LDR r0, = S3C2410_GPEUP             /* 禁止 GPE[15:0]的上拉 */
LDR r1, = 0xffff
STR r1,[r0]

/*
 * PORT F 功能定义
 * Ports   : GPF7 GPF6 GPF5 GPF4 GPF3 GPF2 GPF1 GPF0
 * Signal  : nLED_8 nLED_4 nLED_2 nLED_1 nIRQ_PCMCIA EINT2 KBDINT EINT0
 * Setting : OUTPUT OUTPUT OUTPUT EINT4 EINT3 EINT2 EINT1 EINT0
 */
LDR r0, = S3C2410_GPFCON
LDR r1, = 0xaaaa
STR r1,[r0]
LDR r0, = S3C2410_GPFUP             /* 禁止 GPF[7:0]的上拉 */
LDR r1, = 0xff
STR r1,[r0]

/*
 * PORT G 功能定义
 * Ports   : GPG15 GPG14 GPG13 GPG12 GPG11 GPG10 GPG9 GPG8 GPG7 GPG6
 * Signal  : nYPON YMON nXPON XMON EINT19 DMAMODE1 DMAMODE0 DMASTART KBDSPICLK KBDSPIMOSI
 * Setting : nYPON YMON nXPON XMON EINT19 OUTPUT OUTPUT OUTPUT SPICLK1 SPIMOSI1
```

```
 * Ports    :GPG5 GPG4 GPG3 GPG2 GPG1 GPG0
 * Signal   : KBDSPIMISO LCD_PWREN EINT11 nSS_SPI IRQ_LAN IRQ_PCMCIA
 * Setting  :SPIMISO1 LCD_PWRDN EINT11 nSS0 EINT9 EINT8
 */
LDR r0, = S3C2410_GPGCON
LDR r1, = 0xff95ff9a
STR r1,[r0]
LDR r0, = S3C2410_GPGUP               /* 禁止 GPG[15:0] 的上拉 */
LDR r1, = 0xffff
STR r1,[r0]
/*
 * PORT H 功能定义
 * Ports    : GPH10 GPH9 GPH8 GPH7 GPH6 GPH5 GPH4 GPH3 GPH2 GPH1 GPH0
 * Signal   : CLKOUT1 CLKOUT0 UCLK nCTS1 nRTS1 RXD1 TXD1 RXD0 TXD0 nRTS0 nCTS0
 * Binary   : 10,  10  10 ,  11  11,  10  10 ,  10  10 ,  10  10
 */
LDR r0, = S3C2410_GPHCON
LDR r1, = 0x2afaaa
STR r1,[r0]
LDR r0, = S3C2410_GPHUP               /* 禁止 GPH[10:0]的上拉 */
LDR r1, = 0x7ff
STR r1,[r0]

/*外部中断触发信号配置*/
LDR r0, = S3C2410_EXTINT0             /* EINT[7:0],上升沿触发 */
LDR r1, = 0x44444444
STR r1,[r0]
LDR r0, = S3C2410_EXTINT1             /* EINT[15:8],下降沿触发 */
LDR r1, = 0x22222222
STR r1,[r0]
LDR r0, = S3C2410_EXTINT2             /* EINT[23:16],下降沿触发 */
LDR r1, = 0x22222222
STR r1,[r0]

/* 设置临时堆栈 */
LDR sp, L$_STACK_ADDR
MOV fp, #0
```

```
/* 跳转到 C 例程——romStart() */
LDR pc, L$_rStrtInRom
```

2. romStart()

对于带 ROM 启动的 VxWorks 内核，在执行完 romInit()例程后，便开始调用该例程。其作用就是将 VxWorks 内核映像拷贝到 RAM 中。程序清单 4-6 为非压缩核心 romStart()例程。

程序清单 4-6　非压缩核心 romStart()例程（$(WIND_BASE)target\config\comps\src\romStart.c）

```
void romStart
    (
    FAST int startType          /* 启动类型 */
    )
    {
    volatile FUNCPTR absEntry = (volatile FUNCPTR)RAM_DST_ADRS;

    /* 如果是冷启动，则首先将 RAM 清 0 */
#ifdef ROMSTART_BOOT_CLEAR
    if (startType & BOOT_CLEAR)
        bootClear();
#endif

    /* 将 VxWorks 映像拷贝到指定地址。对于非压缩内核，目标地址为 RAM_LOW_ADRS */
    copyLongs ((UINT *)ROM_DATA(binArrayStart),
            (UINT *)UNCACHED(RAM_DST_ADRS),
            (binArrayEnd - binArrayStart) / sizeof (long));
#if ((CPU_FAMILY == ARM) && ARM_THUMB)
     absEntry = (FUNCPTR)((UINT32)absEntry | 1);          /* 转换为 Thumb 调用 */
#endif   /* CPU_FAMILY == ARM */

    /* 跳转到 VxWorks 入口——sysInit() */
    absEntry (startType);
    }
```

3. usrInit()

该例程是 VxWorks 内核代码中的第一个 C 例程；根据不同的内核配置，具体代码会有所

不同。该例程位于工程目录下的 prjConfig. c 文件中。prjConfig. c 文件由 Tornado 根据配置文件动态生成，一般无需手工修改，因为在每次重新全部编译、链接整个工程时都会重新生成该文件。程序清单 4－7 是 usrInit()例程。

程序清单 4－7　$(PRJ_DIR)\prjConfig. c——usrInit( )

```
void usrInit (int startType)
    {
    sysStart (startType);                         /* 清除 BSS */
    cacheLibInit (USER_I_CACHE_MODE, USER_D_CACHE_MODE);   /* Cache 支持库初始化 */
    excVecInit ();                                /* 安装中断向量 */
    sysHwInit ();                                 /* BSP 硬件初始化,初始化串口 */
    usrCacheEnable ();                            /* 使能 Cache */
    usrKernelInit ();                             /* 多任务环境初始化 */
    }
```

sysStart 例程是 usrInit()例程调用的第一个子程序，见程序清单 4－8。除了清除 BSS 外，它还调用了 SYS_HW_INIT_0 ()，实际上该函数可以看作是一个钩子函数，目的是方便进行一些前期初始化工作(如 MMU 映射)。所以如果处理器需要在内核的其他初始化之前执行一些额外的操作，则都可放在 SYS_HW_INIT_0 ()里面。

程序清单 4－8　$(WIND_BASE)\target\config\comps\src\usrStartup. c

```
void sysStart (startType)
    {
#if (CPU_FAMILY == SPARC)
    excWindowInit ();                                   /* SPARC 窗口管理 */
#endif

#ifdef INCLUDE_SYS_HW_INIT_0
    /*
     * 在 cacheLibInit()和 BSS 段清 0 前进行一些必要的初始化
     * SYS_HW_INIT_0 () 对应于 config.h 中定义的具体例程
     * #define INCLUDE_SYS_HW_INIT_0
     * #define SYS_HW_INIT_0()     sysHwInit0 ()
     */
    SYS_HW_INIT_0 ();
#endif /* INCLUDE_SYS_HW_INIT_0 */
```

```
# ifdef CLEAR_BSS
# if (CPU_FAMILY == I960)
    extern UINT32 supervisorStack[];
    if ((UINT)supervisorStack > (UINT)edata)
    {
    UINT stackOffset = (UINT)supervisorStack + 400;
    bzero (edata, (UINT)supervisorStack - (UINT)edata);
    bzero ((char * )stackOffset, (UINT)end - (UINT)stackOffset);
    }
    else
# endif     /* CPU_FAMILY == I960 */
    bzero (edata, end - edata);             /* BSS 段清 0 */
# endif     /* CLEAR_BSS */

# if (CPU_FAMILY == PPC)
    ioGlobalStdSet (STD_IN,  ERROR);
    ioGlobalStdSet (STD_OUT, ERROR);
    ioGlobalStdSet (STD_ERR, ERROR);
# endif   /* CPU_FAMILY == PPC */

    sysStartType = startType;
    /* 设置中断向量基地址,对 ARM 体系结构的处理器无效 */
    intVecBaseSet ((FUNCPTR * ) VEC_BASE_ADRS);

# if (CPU_FAMILY == PPC) && defined(INCLUDE_EXC_SHOW)
    excShowInit ();
# endif   /* CPU_FAMILY == PPC && defined(INCLUDE_EXC_SHOW) */
    }
```

在 usrInit()调用的例程中,除了 sysHwInit()外,都由系统提供。sysHwInit0()和 sysHwInit()的具体代码参见程序清单 4-9 和程序清单 4-10。

程序清单 4-9　$ (BSP_BASE)\sysLib. c——sysHwInit0 ( )

```
void sysHwInit0 (void)
    {
# ifdef S3C2410_MMU_INIT
    mmu_Init();                  /* 如果之前没有调用 mmu_Init,则可在此执行 */
# endif
```

```
    /* 初始化 920T 体系结构的 CACHE 运行库和 MMU 运行库 */
#ifdef INCLUDE_CACHE_SUPPORT
#if defined(CPU_920T) || defined(CPU_920T_T)
    cacheArm920tLibInstall (NULL, NULL);
#endif
#endif /* INCLUDE_CACHE_SUPPORT */

#if defined(INCLUDE_MMU)
#if defined(CPU_920T) || defined(CPU_920T_T)
    mmuArm920tLibInstall (NULL, NULL);
#endif
#endif /* defined(INCLUDE_MMU) */

    return;
    }
```

程序清单 4-10　$ (BSP_BASE)\sysLib.c —— sysHwInit ( )

```
void sysHwInit (void)
    {
    /* 安装 IRQ/SVC 模式的堆栈设置程序 */
    _func_armIntStackSplit = sysIntStackSplit;

    /* 拷贝 DEFAULT_BOOT_LINE */
#ifdef  FORCE_DEFAULT_BOOT_LINE
    strncpy(sysBootLine,DEFAULT_BOOT_LINE,strlen(DEFAULT_BOOT_LINE) + 1);
#endif
#ifdef INCLUDE_SERIAL
    /* 初始化串口设备 */
    sysSerialHwInit ();        /* 仅初始化串口设备的数据结构 */
#endif /* INCLUDE_SERIAL */
}
```

4. usrKernelInit()

usrKernelInit()例程是该过程中的最后一个步骤。至此，就无需过多干预了。usrKernelInit()由系统提供，它的工作就是初始化系统的任务管理器和任务队列，并创建系统的根任务，见程序清单 4-11。

程序清单 4-11 $(WIND_BASE)\target\config\comps\src\usrKernel.c

```
void usrKernelInit (void)
    {
    classLibInit ();          /* 初始化类库 */
    taskLibInit ();           /* 初始化任务变量 */

    /* 配置内核的任务队列 */
#ifdef INCLUDE_CONSTANT_RDY_Q
    qInit (&readyQHead, Q_PRI_BMAP, (int)&readyQBMap, 256);
#else
    qInit (&readyQHead, Q_PRI_LIST);
#endif    /* !INCLUDE_CONSTANT_RDY_Q */
    qInit (&activeQHead, Q_FIFO);
    qInit (&tickQHead, Q_PRI_LIST);
    workQInit ();

    /* 激活内核并且生成系统的根任务 usrRoot */
    kernelInit ((FUNCPTR) usrRoot, ROOT_STACK_SIZE, MEM_POOL_START,
                sysMemTop (), ISR_STACK_SIZE, INT_LOCK_LEVEL);
    }
```

在 usrKernelInit()例程的最后，创建了系统中的第一个任务——usrRoot。剩下的初始化工作(如 MMU 的完全初始化、文件系统初始化、网络协议栈和网络设备的初始化，以及用户工具的初始化等)将由该任务完成；多任务环境的激活(启动定时器)也在该任务中完成。最后，usrRoot 会调用 usrAppInit ()，usrAppInit ()可以看作是操作系统在加载过程中的最后一个钩子函数；此时 VxWorks 内核基本就绪，在该函数中即可进行应用程序相关的初始化工作，并创建用户进程。usrRoot()例程见程序清单 4-12。

程序清单 4-12 $(PRJ_DIR)\prjConfig.c——usrRoot()

```
void usrRoot (char * pMemPoolStart, unsigned memPoolSize)
    {
    usrKernelCoreInit ();
    memInit (pMemPoolStart, memPoolSize);
    memPartLibInit (pMemPoolStart, memPoolSize);
    usrMmuInit ();                      /* MMU 单元初始化 */
    sysClkInit ();                      /* 系统时钟初始化 */
```

```
    selectInit (NUM_FILES);
    usrIosCoreInit ();
    usrKernelExtraInit ();
    usrIosExtraInit ();
    usrNetworkInit ();                          /* 网络系统初始化 */
    selTaskDeleteHookAdd ();
    usrToolsInit ();                            /* 软件开发工具初始化 */
    cplusCtorsLink ();
    usrAppInit ();                              /* 调用 usrAppInit() */
    }
```

4.3.4 定时器处理

系统定时器是操作系统的心脏，操作系统中任务的调度和时间的计算等都依赖于它。在 S3C2410 处理器中，片内集成了多达 6 个 PWM 定时器，可以随意选择其中两个作为系统定时器和辅助定时器。

系统定时器的初始化工作由前面提到的 usrRoot 任务调用 sysClkInit()例程完成。sysClkInit()由系统提供，位于文件 $(WIND_BASE)\target\config\comps\src\sysClkInit.c 中。该例程非常重要，其参考代码见程序清单 4-13。这里是一个分水岭：前面都没有开启中断，系统处于一种安静的模式下，而这之后会出现中断。在此之前虽然也创建了一个任务(usrRoot)，但不存在任务切换的问题，usrRoot 依然处于一个单线程的干净环境；只有在这之后才具备任务切换的能力。所以编写 BSP 时要注意一点：在多任务环境建立起来之前，不要使用一些可能引起阻塞的函数调用(如 printf 等)。因为此时一旦引起阻塞，整个系统将停留在阻塞点而无法动弹。相关信息请查阅《VxWorks BSP Developer's Guide》。

注意：*系统提供的组件中有一部分是基于源代码的，这些源代码都会在 $(PRJ_DIR)\prjConfig.c 文件中以头文件的形式被包含进来。例如在 $(PRJ_DIR)\prjConfig.c 文件中可以找到语句：#include "sysClkInit.c"。*

程序清单 4-13　$(WIND_BASE)\target\config\comps\src\sysClkInit.c

```
/*****************************************************************************
* usrClock——用户定义的系统时钟服务例程
*/
void usrClock (void)
    {
    tickAnnounce();                             /* 系统进行任务状态的检查以及任务的切换 */
```

```
    }

/ ************************************************************************************
  * sysClkInit——初始化系统时钟
  */
void sysClkInit (void)
    {
    sysClkConnect ((FUNCPTR) usrClock, 0);              /* 挂接中断例程 */
    sysClkRateSet (SYS_CLK_RATE);                       /* 设置系统时钟速率 */
    sysClkEnable ();                                    /* 启动系统时钟 */
    }
```

这里，选取 S3C2410 的定时器 0 作为系统时钟定时器；定时器 1 作为系统辅助定时器。关于 S3C2410 定时器的操作请参阅《S3C2410 user's manual》，见程序清单 4－14。

程序清单 4－14　$ (BSP_BASE) \s3c2410Timer. c

```
/ ************************************************************************************
 * sysClkInt——系统时钟的中断处理例程
 */
void sysClkInt (void)
    {
    /* 应答时钟中断 */
    sysClkIntAck();

    /* 调用系统时钟服务例程 usrClock() */
    if (sysClkRoutine != NULL)
        (* sysClkRoutine) (sysClkArg);
    }

/ ************************************************************************************
* sysClkConnect——安装系统时钟服务例程
*/
STATUS sysClkConnect
    (
    FUNCPTR routine,        /* 系统时钟服务例程 */
    int arg                 /* 参数 */
    )
```

```
    {
    if (sysClkConnected == FALSE)
        {
        /* 设置定时器为一次触发模式，且不使能定时器 */
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCON, 0x0);

        /* 调用 sysHwInit2()安装中断处理驱动,设置定时器的中断向量,并启动定时器 */
        sysHwInit2 ();
        sysClkConnected = TRUE;
        }

    /* 保存设置的时钟服务例程入口和参数 */
    sysClkArg       = arg;
    sysClkRoutine   = routine;
    return (OK);
    }

/*************************************************************************************
* 关闭系统定时器
*/
void sysClkDisable (void)
    {
    int oier;
    if (sysClkRunning)
        {
        /* 停止系统定时器 */
        SNGKS32C_TIMER_REG_READ (S3C2410_TCON,  oier);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCON, oier &~ (0x00000001));

        /* 禁止系统定时器中断 */
        SNGKS32C_TIMER_INT_DISABLE (SYS_TIMER_INT_LVL);
        sysClkRunning = FALSE;
        }
    }
/*************************************************************************************
* 使能系统定时器
```

```
 */
void sysClkEnable (void)
    {
    UINT32 oier;
    if (!sysClkRunning)
        {
        /* 设置系统定时器的分频值 */
        SNGKS32C_TIMER_REG_READ (S3C2410_TCFG0, oier);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCFG0, (oier & 0xffffff00) | 0x0063);

        /* 设置辅助定时器的分频值 */
        SNGKS32C_TIMER_REG_READ (S3C2410_TCFG1, oier);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCFG1, (oier & 0xffffff00) | 0x0011);

        /* 设置系统定时器的频率和工作方式 */
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCNTB0,((125 * 1000)/sysClkTicksPerSecond));
        SNGKS32C_TIMER_REG_READ (S3C2410_TCON,  oier);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCON,  ((oier&0xfffffff0)|0x0002));
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCON,  ((oier&0xfffffff0)|0x0009));

        /* 使能定时器中断 */
        SNGKS32C_TIMER_INT_ENABLE (SYS_TIMER_INT_LVL);
        sysClkRunning = TRUE;

        }
    }

/*******************************************************************************
* 获取系统时钟的当前频率
*/
int sysClkRateGet (void)
    {
    return (sysClkTicksPerSecond);
    }

/*******************************************************************************
```

```
 * 设置系统时钟的频率
 */
STATUS sysClkRateSet
    (
    int ticksPerSecond          /* 系统时钟的频率数 */
    )
    {
    if (ticksPerSecond < SYS_CLK_RATE_MIN || ticksPerSecond > SYS_CLK_RATE_MAX)
        return (ERROR);
    sysClkTicksPerSecond = ticksPerSecond;
    if (sysClkRunning)
        {
        sysClkDisable ();
        sysClkEnable ();
        }
    return (OK);
    }

/*******************************************************************************
 * 辅助定时器的中断处理程序
 */
void sysAuxClkInt (void)
    {
    if (sysAuxClkRoutine != NULL)
        (*sysAuxClkRoutine) (sysAuxClkArg);

    }

/*******************************************************************************
 * 安装辅助定时器
 */
STATUS sysAuxClkConnect
    (
    FUNCPTR routine,     /* 辅助定时器服务例程 */
    int arg              /* 参数 */
    )
```

```
    {
    sysAuxClkRoutine      = NULL;
    sysAuxClkArg          = arg;      /* 登记辅助定时器服务例程 */
    sysAuxClkRoutine = routine;
    return (OK);
    }

/***********************************************************************************
* 关闭辅助定时器
*/
void sysAuxClkDisable (void)
    {
    UINT32 oier;

    if (sysAuxClkRunning)
        {
        /* 停止辅助定时器 */
        SNGKS32C_TIMER_REG_READ (S3C2410_TCON, oier);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCON, oier &~(0x0100));

        /* 禁止辅助定时器中断 */
        SNGKS32C_TIMER_INT_DISABLE (AUX_TIMER_INT_LVL);
        sysAuxClkRunning = FALSE;
        }
    }

/***********************************************************************************
* 开启辅助定时器
*/
void sysAuxClkEnable (void)
    {
    UINT32 oier;
    static BOOL connected = FALSE;

    if (!connected)
        {
```

```
        intConnect (INUM_TO_IVEC (AUX_TIMER_INT_LVL), sysAuxClkInt, 0);
        connected = TRUE;
        }

    if (!sysAuxClkRunning)
        {
         /*
         * 计算辅助定时器的频率值
         */
         sysAuxClkTicks = (AUX_TIMER_CLK / sysAuxClkTicksPerSecond);
        /* 配置辅助定时器的控制寄存器,设置定时周期及工作方式 */
        SNGKS32C_TIMER_REG_READ  (S3C2410_TCFG0, oier);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCFG0, (oier & 0xffffff00) | 0x0063);
        SNGKS32C_TIMER_REG_READ  (S3C2410_TCFG1, oier);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCFG1, (oier & 0xffffff00) | 0x0011);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCNTB1,((125 * 1000)/sysAuxClkTicks));

        SNGKS32C_TIMER_REG_READ  (S3C2410_TCON,  oier);
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCON,  ((oier&0xfffff0ff)|0x0200));
        SNGKS32C_TIMER_REG_WRITE (S3C2410_TCON,  ((oier&0xfffff0ff)|0x0900));
        /* 使能辅助定时器中断 */
        SNGKS32C_TIMER_INT_ENABLE (AUX_TIMER_INT_LVL);

        sysAuxClkRunning = TRUE;
        }
    }

/*******************************************************************************
* 获取辅助定时器的当前频率
*/
int sysAuxClkRateGet (void)
    {
    return (sysAuxClkTicksPerSecond);
    }

/*******************************************************************************
```

```
 * 设置辅助定时器的频率
 */
STATUS sysAuxClkRateSet
    (
    int ticksPerSecond   /* 辅助定时器的频率数 */
    )
    {
    if (ticksPerSecond < AUX_CLK_RATE_MIN || ticksPerSecond > AUX_CLK_RATE_MAX)
        return (ERROR);

    sysAuxClkTicksPerSecond = ticksPerSecond;

    if (sysAuxClkRunning)
        {
        sysAuxClkDisable ();
        sysAuxClkEnable ();
        }
   return (OK);
   }
```

sysHwInit2()是在 sysClkConnect()中被调用的一个例程，主要用于安装系统时钟和辅助定时器的中断，以及串口等设备的中断，参见程序清单 4－15。

程序清单 4－15　$ (BSP_BASE)\sysLib.c —— sysHwInit2()

```
void sysHwInit2 (void)
    {
    static BOOL initialized = FALSE;

    if (initialized)
        return;

    /* 初始化中断驱动程序库 */
    intLibInit (S3C2410_INT_NUM_LEVELS, S3C2410_INT_NUM_LEVELS, INT_MODE);

    /* 中断驱动初始化 */
    sngks32cIntDevInit();

    /* 挂接系统时钟中断和辅助定时器中断 */
    (void)intConnect (INUM_TO_IVEC (SYS_TIMER_INT_VEC), sysClkInt, 0);
```

```
    (void)intConnect (INUM_TO_IVEC (AUX_TIMER_INT_VEC), sysAuxClkInt, 0);

#ifdef INCLUDE_SERIAL
    /* 挂接串口中断,将在串口驱动中介绍 */
    sysSerialHwInit2();
#endif /* INCLUDE_SERIAL */

    initialized = TRUE;
    }
```

4.3.5 中断处理

VxWorks 的中断处理与硬件平台密切相关,不同的处理器其中断处理方法是不一样的。针对 ARM 体系结构的处理器,系统做了一些简单的工作,即中断向量表的安装。中断向量表的安装由 excVecInit ()例程完成;excVecInit ()的工作就是在 0x00000000 地址处建立向量表。程序清单 4-16 展示了 S3C2410 的中断处理情况。实际上这里只需在中断初始化例程中实现 3 个函数 sngks32cIntLvlVecChk、sngks32cIntLvlEnable 和 sngks32cIntLvlDisable,并通过钩子函数 sysIntLvlVecChkRtn、sysIntLvlEnableRtn 和 sysIntLvlDisableRtn 来进行调用。当系统中断发生时,系统会调用由 sysIntLvlVecChkRtn 指定的处理例程;该例程执行完毕后,会返回与对应中断相关的处理函数入口;最后由系统调用这个入口来完成中断处理。

程序清单 4-16　中断处理程序

```
/*****************************************************************************
 * 中断驱动初始化
 */
void sngks32cIntDevInit (void)
    {
    int index;

    /* 安装中断驱动程序的 3 个钩子函数 */
    sysIntLvlVecChkRtn      = sngks32cIntLvlVecChk;
    sysIntLvlEnableRtn      = sngks32cIntLvlEnable;
    sysIntLvlDisableRtn     = sngks32cIntLvlDisable;

    sngks32cIntLvlEnabled   = 0x0;      /* 初始状态:禁止所有中断源 */
```

```
    /* 设置所有中断为 IRQ 模式 */
    SNGKS32C_INT_REG_WRITE (S3C2410_INTMOD,0x00000000);

    /* 屏蔽所有中断源 */
    SNGKS32C_INT_REG_WRITE (S3C2410_INTMSK,0xffffffff);

    /* 屏蔽所有子中断源 */
    SNGKS32C_INT_REG_WRITE (S3C2410_INTSUBMSK,0x7ff);

    }

/*******************************************************************************
* 中断源检测
*/
STATUS  sngks32cIntLvlVecChk
    (
    int * pLevel,
    int * pVector
    )
    {
    int newLevel;
    int intPendMask = 0x80000000;
    int count;
    UINT32 isr;

    /* 读取中断未决标志 */
    SNGKS32C_INT_REG_READ (S3C2410_INTPND, newLevel);
    newLevel &= sngks32cIntLvlEnabled ;
    if(newLevel == 0) return ERROR;

    /* 根据读取的中断标志,检查该中断的处理函数入口 */
    for (count = 0, isr = 31; count < 32; count ++)
        {
        if (intPendMask & newLevel) break;
        isr -- ;
        intPendMask >>= 1;
        }
```

```
    * pVector = isr;

    /* 清除中断 */
    SNGKS32C_INT_REG_WRITE(S3C2410_SRCPND,(1 << isr));
    SNGKS32C_INT_REG_WRITE(S3C2410_INTPND,(1 << isr));

    return OK;
    }

/************************************************************************************
* 使能指定的中断
*/
STATUS  sngks32cIntLvlEnable
    (
    int level
    )
    {
    int key;

    /* 设置中断屏蔽寄存器中的指定位 */
    key = intLock ();
    sngks32cIntLvlEnabled |= ((1 << level));

    SNGKS32C_INT_REG_WRITE (S3C2410_INTMSK, ((~sngks32cIntLvlEnabled)));

    intUnlock (key);

    return OK;
    }

/************************************************************************************
* 禁止指定的中断
*/
STATUS  sngks32cIntLvlDisable
    (
    int level
    )
    {
    int key;
```

```
/* 清除中断屏蔽寄存器中的指定位 */
key = intLock ();
sngks32cIntLvlEnabled &= ~(1 << level);
SNGKS32C_INT_REG_WRITE (S3C2410_INTMSK, ((~sngks32cIntLvlEnabled)));
intUnlock (key);
return OK;
}
```

4.4 组件管理

中断处理和定时器处理编写就绪之后，整个 BSP 基本上就可以运行了，但为了便于观察一些过程性的结果，通常还需在 BSP 中加入串口驱动。关于串口驱动程序的编写将在第 5 章中详细说明。在 BSP 发布时，还可能使用组件描述语言来描述一些配置信息。

构建 VxWorks 的传统方法是基于构造头文件的，它在快速发展的软件应用面前显得越来越低效。组件是 Tornado 使用的一种新的配置方法，以便更好地适用于越来越复杂的 VxWorks 环境。这种方法使用 Tornado 的图形配置工具和工程管理工具，为用户提供了一种可视化的配置手段。目前约有 300 个可选择的组件，组件技术已成为 VxWorks 进行工程配置的标准方法。Tornado 组件管理器如图 4-9 所示。

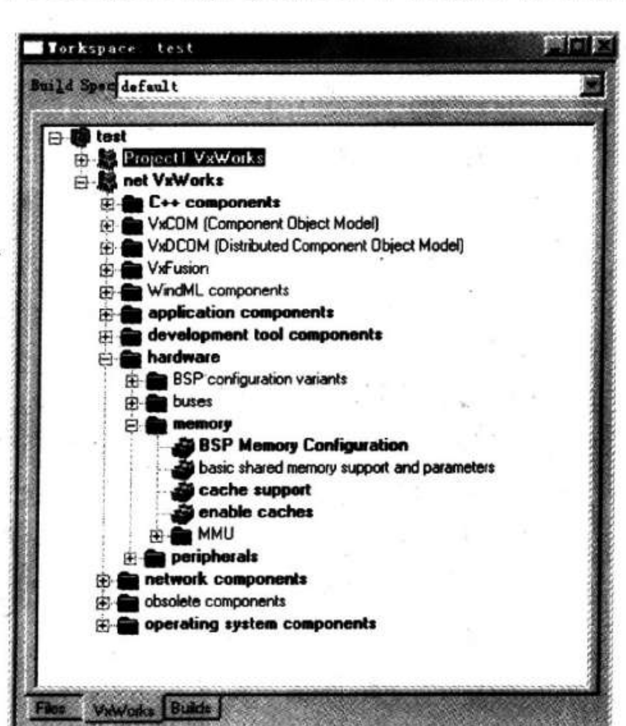


图 4-9 Tornado 组件管理器

基于组件技术的工程管理工具使用了一种描述性语言 —— 组件描述语言(CDL)，用来定义组件及其参数，以及各组件之间的关联信息。

1. 组件及组件描述语言

组件是一个可配置模块的基本单元，也是系统中最小的、可升级的单元。通过工程管理工具，用户能很方便地添加或删除一个组件，也可修改一些组件模块的参数。通常用组件描述文

件(CDF)来描述系统中的一些组件，CDF 文件的约定后缀为 cdf。每个文件可定义不只一个组件。在 Tornado 开发工具中，代码产生器能够根据用户所选择的组件配置，自动输出系统配置文件。过去用户一般都须通过修改配置文件(config.h 或 configAll.h)，来修改系统所包含的一些特征，例如声明 INCLUDE_FOO 宏来定义一个模块：

```
#define  INCLUDE_FOO
```

现在，用户可以使用 CDF 文件非常方便地定义系统的特征参数，描述一个组件，并将其配置到不同的工程中。例如：

```
component INCLUDE_FOO
{
//定义一个名为 INCLUDE_FOO 的组件模块
NAME          FOO example component
SYNOPSIS      This is just On example component
… //其他特征
)
```

组件描述语言大致有以下 4 部分：

代码：构造一个工程使用的代码。该代码可是二进制形式的目标文件或库文件，也可是源代码的合集或子集。

配置信息：提供丰富的特性参数用于描述组件，这些参数可以通过组件管理器来改变、增加或是删除。

集成信息：控制一个组件是如何集成到一个可执行目标映像中去的，例如在初始化过程中何时调用程序中所使用的这些组件；集成特性参数也定义了组件间的依赖关系。

用户介绍：对该组件的一个说明，如功能介绍和参数说明等。

组件描述语言支持很多目标类型。随着组件的增加，需要一种有效的方法来管理这些组件，因此就引入了文件夹、选集、组件、参数和初始化群组的概念，下面分别说明。

(1) 文件夹

在 Tornado 中组件模块是采用分组的形式进行管理的；而文件夹则提供了组件分组的等级。通常文件夹里面的组件是逻辑相关的，例如网络组件、内核组件、开发工具支持组件和 POSIX 组件等都是分组的组件。在工程组件管理器中，文件夹对应于工程管理窗口的一个可扩展树节点(图标■)，如图 4-9 所示。一个文件夹包含了不只一个组件，例如 ANSI 组件对应的文件夹就包含了很多相关的组件，其描述列表见程序清单 4-17。

程序清单 4-17 ANSI 组件描述列表

```
Folder FOLDER_ANSI {
NAME ANSI C components (libc)
SYNOPSIS ANSI libraries
CHILDREN INCLUDE ANSI_ASSERT\
INCLUDE ANSI_CTYPE\
INCLUDE_ANSI_LOCALE\
INCLUDE_ANSI_MATH\
INCLUDE ANSI_STDIO\
INCLUDE_ANSI_STDLIB\
INCLUDE ANSI_STRING\
INCLUDE_ANSI_TIME\
INCLUDE_ANSI_STDIO_EXTRA
DEFAULTS INCLUDE_ANSI_ASSERT INCLUDE_ANSI_CTYPE\
INCLUDE_ANSI_MATH INCLUDE_ANSI_STDIO\
INCLUDE_ANSI_STDLIB INCLUDE_ANSI_STRING\
INCLUDE_ANSI_TIME
}
```

图 4-10 示出了 ANSI 组件在 Tornado 中的配置情况。显然这种方式管理组件更加方便、直观。文件夹里面的组件可被添加或单个地删除，同时文件夹还可包含一个或更多的组件、选集和其他文件夹。一个文件夹包括以下元素：

名称(NAME)：该组件的名称，在工程组件管理器中出现在文件夹图标旁边。

描述(SYNOPSIS)：对一个文件夹的说明。例如：程序清单 4-7 中提到的 SYNOPSIS ANSI libraries，就是描述该组件为 ANSI 库。

子文件夹(CHILDREN)：描述该组件所包含的子文件夹，类似于 Windows 的文件管理。文件夹里可以包含子文件夹，也可包含文件。

默认值(DEFAULTS)：如果没有使用任何选择，就把文件夹加进去，则该组件被引用时，一些特征参数将使用这些默认值。当某个文件夹被选择加进去时，文件夹分组特性能够影响配置。这是因为组件会同时被文件夹的默认值所指定，系统会根据这些默认值分析组件间的依赖关系，从而自动改变一些系统的配置。

文件夹中组件的包含情况是可以动态修改的，用户可通过查看对应的 CDF 文件或文件夹属性对话框来检查某个文件夹所设定的默认值(如图 4-11 所示)。双击工程管理工具中的文件夹节点可以打开文件夹属性对话框，如果该文件夹中的某个组件已被包含，则该组件将以复选标记的形式出现在新打开的窗口中。为了看到一个文件夹中的所有组件信息，必须先将文件夹中的所有组件设置为不包含。

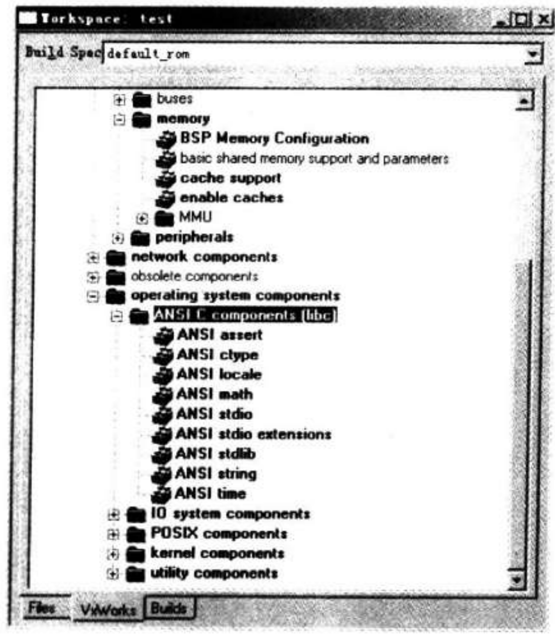


图 4-10 ANSI 组件

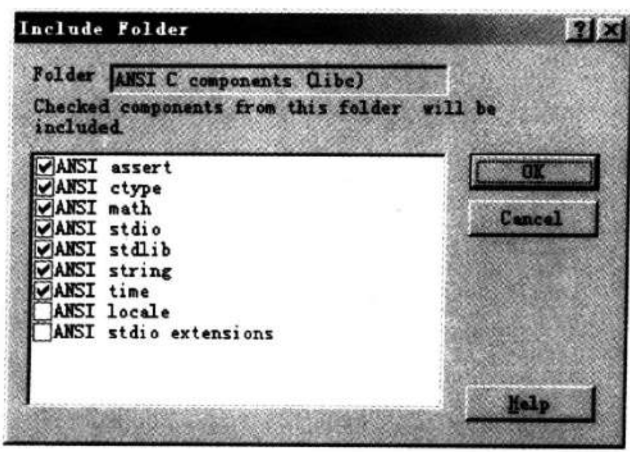
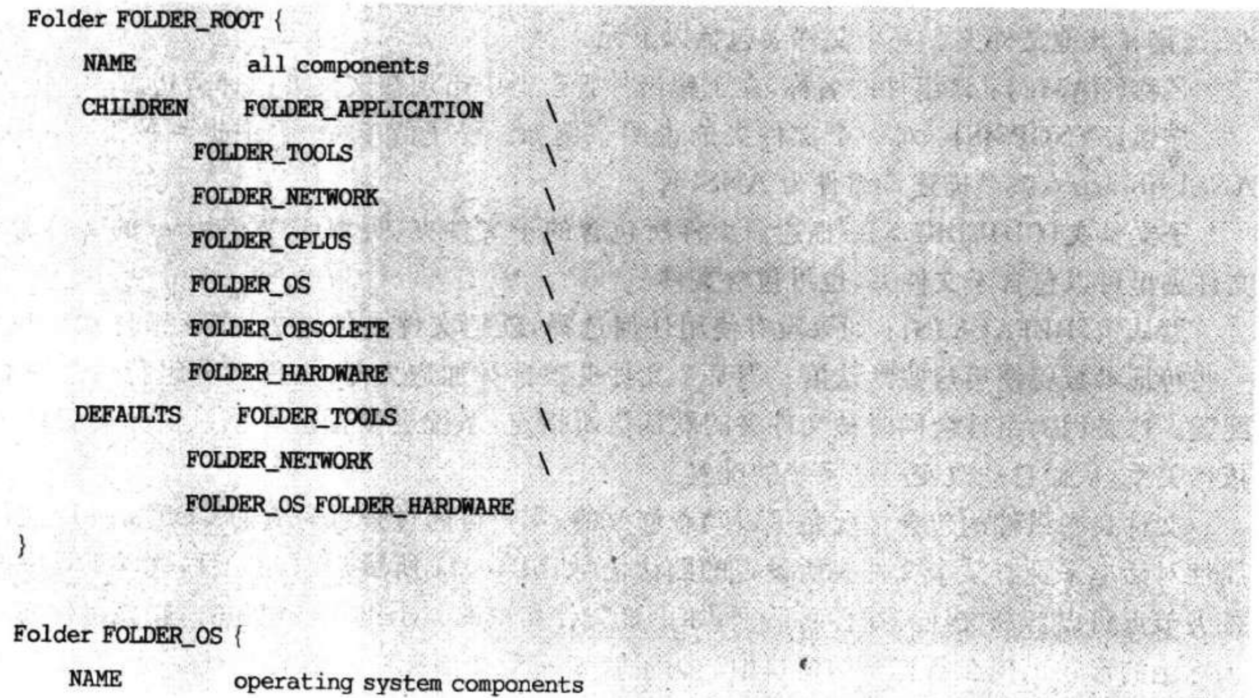


图 4-11 ANSI 文件夹中组件选择

可以通过文件 $(WIND_BASE)\target\config\comps\VxWorks\00VxWorks.cdf 查看整个系统的文件夹层次信息，程序清单 4-18 列出了该文件的一部分内容。

程序清单 4-18 $(WIND_BASE)\target\config\comps\VxWorks\00VxWorks.cdf



```
Folder FOLDER_ROOT {
    NAME        all components
    CHILDREN    FOLDER_APPLICATION      \
                FOLDER_TOOLS            \
                FOLDER_NETWORK          \
                FOLDER_CPLUS            \
                FOLDER_OS               \
                FOLDER_OBSOLETE         \
                FOLDER_HARDWARE
    DEFAULTS    FOLDER_TOOLS            \
                FOLDER_NETWORK          \
                FOLDER_OS FOLDER_HARDWARE
}

Folder FOLDER_OS {
    NAME        operating system components
```

```
    CHILDREN    FOLDER_IO_SYSTEM            \
            FOLDER_KERNEL                   \
            FOLDER_ANSI                     \
            FOLDER_POSIX                    \
            FOLDER_UTILITIES
    DEFAULTS    FOLDER_IO_SYSTEM FOLDER_KERNEL FOLDER_ANSI \
            FOLDER_UTILITIES
}

Folder FOLDER_OBSOLETE {
    NAME        obsolete components
    SYNOPSIS    will be removed next release
    CHILDREN    INCLUDE_TYCODRV_5_2
}
 :
Folder    FOLDER_ANSI {
    NAME        ANSI C components (libc)
    SYNOPSIS    ANSI libraries
    CHILDREN    INCLUDE_ANSI_ASSERT         \
            INCLUDE_ANSI_CTYPE              \
            INCLUDE_ANSI_LOCALE             \
            INCLUDE_ANSI_MATH               \
            INCLUDE_ANSI_STDIO              \
            INCLUDE_ANSI_STDLIB             \
            INCLUDE_ANSI_STRING             \
            INCLUDE_ANSI_TIME               \
            INCLUDE_ANSI_STDIO_EXTRA
    DEFAULTS    INCLUDE_ANSI_ASSERT INCLUDE_ANSI_CTYPE \
            INCLUDE_ANSI_MATH INCLUDE_ANSI_STDIO \
            INCLUDE_ANSI_STDLIB INCLUDE_ANSI_STRING \
            INCLUDE_ANSI_TIME
}

Component    INCLUDE_ANSI_STDIO_EXTRA {
    NAME        ANSI stdio extensions
    SYNOPSIS    WRS routines getw, putw, and setbuffer
    LINK_SYMS   getw putw setbuffer
```

```
}

Component INCLUDE_ANSI_ASSERT {
    NAME        ANSI assert
    LINK_SYMS    _assert
    HELP        ansiAssert
}
:
```

从程序清单 4-18 中可以非常清楚地看到文件夹的层次关系及其具体情况。例如：FOLDER_ROOT 文件夹就包含 FOLDER_APPLICATION、FOLDER_TOOLS 和 FOLDER_NETWORK 等子文件夹，同时通过 DEFAULTS 关键字指定了几个默认的子文件夹。

(2) 选　集

选集类似于文件夹，在工程管理窗口中显示为图标，它们是一个共同接口的组件。例如：串口驱动器、WindView 时标机制以及 WDB 通信接口等都是以选集的形式出现的。这些选集为相同的服务提供了多种选择，选集中的成员可以是单选也可是多选，分别为工程提供一个或多个选择。例如：当用户选择 WDB 调试模式时，即可选择 system debugging 和 task debugging 两种模式，如图 4-12 所示。

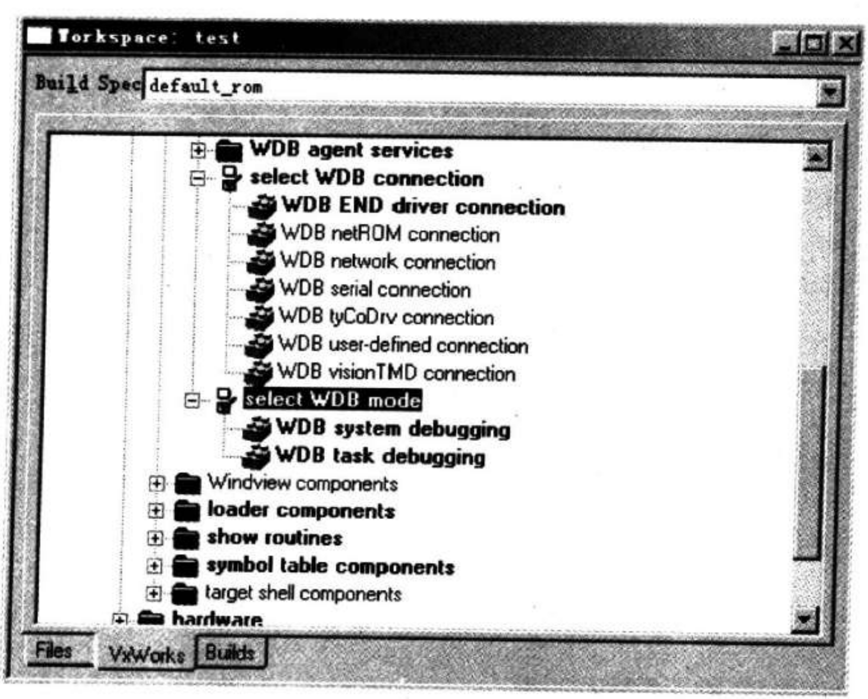


图 4-12　组件中的选集

调试模式选集对应的 CDF 描述信息见程序清单 4-19。

程序清单 4-19 调试模式选集对应的 CDF 描述信息

```
Selection SELECT_WDB_MODE {
    NAME        select WDB mode
    COUNT       1 - 2
    CHILDREN    INCLUDE_WDB_TASK                \
                INCLUDE_WDB_SYS
    DEFAULTS    INCLUDE_WDB_TASK
    HELP        tgtsvr WDB
}
:
Selection SELECT_WDB_COMM_TYPE {
    NAME        select WDB connection
    COUNT       1 - 1
    CHILDREN    INCLUDE_WDB_COMM_SERIAL          \
                INCLUDE_WDB_COMM_TYCODRV_5_2     \
                INCLUDE_WDB_COMM_NETWORK         \
                INCLUDE_WDB_COMM_NETROM          \
                INCLUDE_WDB_COMM_VTMD            \
                INCLUDE_WDB_COMM_END             \
                INCLUDE_WDB_COMM_CUSTOM
    DEFAULTS    INCLUDE_WDB_COMM_NETWORK
    HELP        tgtsvr WDB
}
:
```

选集的元素除了名称、描述和默认值之外，还有以下几方面特殊内容：

- 计数器(COUNT)：为该选集可用的选项设定一个最小值和一个最大值。如果最大、最小值都为 1，则该选集为单选。例如：程序清单 4-8 中的 SELECT_WDB_COMM_TYPE 就是单选；而 SELECT_WDB_MODE 则可为多选。
- 子选集(CHILDREN)：可选择的组件，类似于文件夹中的子文件夹。它实际上就是描述该选集向下的引用关系。

(3) 组 件

这里所指的组件实际上是一个功能部件具体实现部分的描述信息，它定义了与组件相关的源代码或目标代码、集成信息以及与该组件相关的参数。在工程管理窗口中，组件在工程管理器中的显示图标为，如图 4-12 所示。程序清单 4-20 是一个简单的组件描述。

程序清单 4-20　组件描述

```
Component INCLUDE_LOGGING {
    NAME          message logging
    MODULES       logLib.o
    INIT_RTN      logInit (consoleFd, MAX_LOG_MSGS);
    CFG_PARAMS    MAX_LOG_MSGS
    HDR_FILES     logLib.h
}
```

Tornado 能够自动检测组件间的依赖关系。当选中某个组件时，一些需与之同时加载的组件会被工程管理器自动添加到工程中；它通过分析每个相关目标模块的依赖信息来确定哪些组件需被包含其中。组件描述信息中包含大量的特征，除了名称和说明性的描述信息外，通常还有以下几个典型部分：

- 模块(MODULES)：与组件关联的目标文件，即该组件的具体目标代码；
- 头文件(HDR_FILES)：使用该组件时需要用到的相关头文件；
- 配置参数(CFG_PARAMS)：与组件相关的一些配置参数，通常为一系列的宏；
- 初始化入口(INIT_RTN)：包含该组件后需要执行的初始化入口程序；
- 依赖组件(REQUIRES)：如果该组件被包含，则必须包含一系列相关的组件；
- 排斥组件(EXCLUDES)：不能与该组件同时被包含的组件。

在组件管理中引用某个组件后，即可在程序中对组件提供的一些功能函数进行引用。例如：添加了 LOGGING 组件，则可在程序中使用该组件提供的 logMsg 进行信息输出，参见程序清单 4-21。该清单同时展示了组件的初始化方法。

程序清单 4-21　程序中组件的配置和使用（$(WINDBASE)\target\config\all\usrConfig.c）

```
#ifdef INCLUDE_LOGGING
    logInit (consoleFd, MAX_LOG_MSGS);     /* 初始化组件 */
#ifdef INCLUDE_LOG_STARTUP
    logMsg ("logging started to %s [%d], queue size %d\n",
        consoleName, consoleFd, MAX_LOG_MSGS, 4,5,6);   /* 引用组件的函数 */
    taskDelay (2);
#endif     /* INCLUDE_LOG_STARTUP */
#endif     /* INCLUDE_LOGGING */
```

(4) 参　数

参数是用户配置系统的方式之一。对于组件，通常都会有一个或多个参数来控制其行为。程序清单 4-22 为串口通信速率的参数说明。

程序清单 4-22 串口通信速率参数说明

```
Parameter WDB_TTY_BAUD {
    NAME        baud rate for WDB serial channel
    TYPE        uint
    DEFAULT     9600
}
```

参数描述部分包含以下几点：

- 数据类型(TYPE)：参数的数据类型，即 uint、bool、string 或 untyped 等。
- 默认值(DEFAULT)：出现在工程管理器组件特性窗口的参数表中。当然该默认值可被人为地修改，如图 4-13 所示。

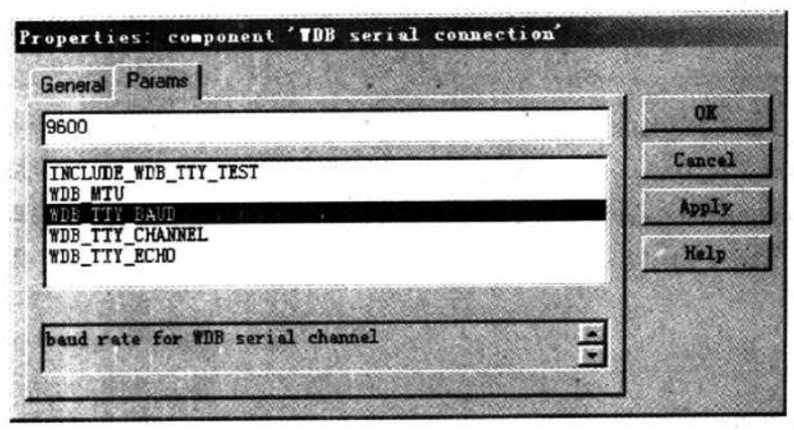


图 4-13 组件参数的修改

(5) 初始化群组

如果文件夹和选集定义的是组件的层次关系，那么初始化群组则定义的是组件的先后关系，即这些组件都被包含时，哪个先执行，哪个后执行。这实际上也是一个初始化程序和一个关于在初始化程序中何时调用那个程序的描述，这些描述称为"初始化群组定义"。程序清单4-23列出了 VxWorks 初始化群组的完整定义，通过这些信息可以看到整个系统的初始化顺序。

程序清单 4-23 初始化群组的完整定义

```
InitGroup usrInit {
    PROTOTYPE    void usrInit (int startType)
    SYNOPSIS     pre-kernel initialization
    INIT_ORDER   INCLUDE_SYS_START               \
            INCLUDE_CACHE_SUPPORT                \
            INCLUDE_EXC_HANDLING                 \
            INCLUDE_SYSHW_INIT                   \
```

```
            INCLUDE_CACHE_ENABLE            \
            INCLUDE_WINDVIEW_CLASS          \
            INCLUDE_KERNEL
}

InitGroup usrToolsInit {
    INIT_RTN      usrToolsInit ();
    SYNOPSIS      software development tools
    INIT_ORDER    INCLUDE_SPY INCLUDE_TIMEX       \
            INCLUDE_MODULE_MANAGER          \
            INCLUDE_LOADER                  \
            INCLUDE_NET_SYM_TBL             \
            INCLUDE_STANDALONE_SYM_TBL      \
            INCLUDE_STAT_SYM_TBL            \
            INCLUDE_TRIGGERING              \
            usrWdbInit usrShellInit         \
            usrWindviewInit                 \
            usrShowInit
}

InitGroup usrWindviewInit {
    INIT_RTN      usrWindviewInit ();
    INIT_ORDER    INCLUDE_WINDVIEW                \
            INCLUDE_SYS_TIMESTAMP           \
            INCLUDE_USER_TIMESTAMP          \
            INCLUDE_SEQ_TIMESTAMP           \
            INCLUDE_RBUFF                   \
            INCLUDE_WV_BUFF_USER            \
            INCLUDE_WDB_TSFS                \
            INCLUDE_WVUPLOAD_SOCK           \
            INCLUDE_WVUPLOAD_TSFSSOCK       \
            INCLUDE_WVUPLOAD_FILE           \
                        INCLUDE_WVNET
}

InitGroup usrWdbInit {
    INIT_RTN      usrWdbInit ();
```

```
    SYNOPSIS     the WDB target agent
    INIT_ORDER   INCLUDE_WDB                          \
             INCLUDE_WDB_MEM                          \
             INCLUDE_WDB_SYS                          \
             INCLUDE_WDB_TASK                         \
             INCLUDE_WDB_EVENTS                       \
             INCLUDE_WDB_EVENTPOINTS                  \
             INCLUDE_WDB_DIRECT_CALL                  \
             INCLUDE_WDB_CTXT                         \
             INCLUDE_WDB_REG                          \
             INCLUDE_WDB_GOPHER                       \
             INCLUDE_WDB_EXIT_NOTIFY                  \
             INCLUDE_WDB_EXC_NOTIFY                   \
             INCLUDE_WDB_FUNC_CALL                    \
             INCLUDE_WDB_VIO_LIB                      \
             INCLUDE_WDB_VIO                          \
             INCLUDE_WDB_BP                           \
             INCLUDE_WDB_TASK_BP                      \
             INCLUDE_WDB_START_NOTIFY                 \
             INCLUDE_WDB_USER_EVENT                   \
             INCLUDE_WDB_HW_FP                        \
             INCLUDE_WDB_TASK_HW_FP                   \
             INCLUDE_WDB_SYS_HW_FP                    \
             INCLUDE_WDB_DSP                          \
             INCLUDE_WDB_TASK_DSP                     \
             INCLUDE_WDB_SYS_DSP                      \
             INCLUDE_WDB_BANNER                       \
             INCLUDE_SYM_TBL_SYNC
}

InitGroup usrShellInit {
    INIT_RTN     usrShellInit ();
    SYNOPSIS     the target shell
    INIT_ORDER   INCLUDE_DEBUG                        \
             INCLUDE_SHELL_BANNER                     \
             INCLUDE_STARTUP_SCRIPT                   \
             INCLUDE_SHELL
```

```
}

InitGroup usrShowInit {
    INIT_RTN      usrShowInit ();
    SYNOPSIS      enable object show routines
    INIT_ORDER    INCLUDE_TASK_SHOW             \
                  INCLUDE_CLASS_SHOW            \
                  INCLUDE_MEM_SHOW              \
                  INCLUDE_TASK_HOOKS_SHOW       \
                  INCLUDE_SEM_SHOW              \
                  INCLUDE_MSG_Q_SHOW            \
                  INCLUDE_WATCHDOGS_SHOW        \
                  INCLUDE_SYM_TBL_SHOW          \
                  INCLUDE_MMU_FULL_SHOW         \
                  INCLUDE_POSIX_MQ_SHOW         \
                  INCLUDE_POSIX_SEM_SHOW        \
                  INCLUDE_HW_FP_SHOW            \
                  INCLUDE_DSP_SHOW              \
                  INCLUDE_ATA_SHOW              \
                  INCLUDE_TRIGGER_SHOW          \
                  INCLUDE_RBUFF_SHOW            \
                  INCLUDE_STDIO_SHOW
}

InitGroup usrKernelCoreInit {
    INIT_RTN      usrKernelCoreInit ();
    SYNOPSIS      core kernel facilities
    INIT_ORDER    INCLUDE_VXEVENTS              \
                  INCLUDE_SEM_BINARY            \
                  INCLUDE_SEM_MUTEX             \
                  INCLUDE_SEM_COUNTING          \
                  INCLUDE_MSG_Q                 \
                  INCLUDE_WATCHDOGS             \
                  INCLUDE_TASK_HOOKS
}

InitGroup usrKernelExtraInit {
```

```
    INIT_RTN     usrKernelExtraInit ();
    SYNOPSIS     extended kernel facilities
    INIT_ORDER   INCLUDE_HASH                   \
            INCLUDE_SYM_TBL                     \
            INCLUDE_ENV_VARS                    \
            INCLUDE_SIGNALS                     \
            INCLUDE_POSIX_AIO                   \
            INCLUDE_POSIX_AIO_SYSDRV            \
            INCLUDE_POSIX_MQ                    \
            INCLUDE_POSIX_PTHREADS              \
            INCLUDE_POSIX_SEM                   \
            INCLUDE_POSIX_SIGNALS               \
            INCLUDE_PROTECT_TEXT                \
            INCLUDE_PROTECT_VEC_TABLE
}

InitGroup usrIosCoreInit {
    INIT_RTN     usrIosCoreInit ();
    SYNOPSIS     core I/O system
    INIT_ORDER   INCLUDE_SW_FP                  \
            INCLUDE_HW_FP                       \
            INCLUDE_DSP                         \
            INCLUDE_BOOT_LINE_INIT              \
            INCLUDE_IO_SYSTEM                   \
            INCLUDE_TTY_DEV                     \
            INCLUDE_TYCODRV_5_2                 \
            INCLUDE_SIO                         \
            INCLUDE_PC_CONSOLE
    }

InitGroup usrIosExtraInit {
    INIT_RTN     usrIosExtraInit ();
    SYNOPSIS     extended I/O system
    INIT_ORDER   INCLUDE_EXC_SHOW               \
            INCLUDE_EXC_TASK                    \
            INCLUDE_LOGGING                     \
            INCLUDE_PIPES                       \
```

```
        INCLUDE_STDIO                       \
        INCLUDE_FORMATTED_IO                \
        INCLUDE_FLOATING_POINT              \
        INCLUDE_CDROMFS                     \
        INCLUDE_DOSFS                       \
        INCLUDE_RAWFS                       \
        INCLUDE_RT11FS                      \
        INCLUDE_RAMDRV                      \
        INCLUDE_SCSI                        \
        INCLUDE_FD                          \
        INCLUDE_IDE                         \
        INCLUDE_ATA                         \
        INCLUDE_LPT                         \
        INCLUDE_PCMCIA                      \
        INCLUDE_TFFS                        \
        INCLUDE_TFFS_DOSFS
}

InitGroup usrRoot {
    PROTOTYPE   void usrRoot (char * pMemPoolStart, unsigned memPoolSize)
    SYNOPSIS    entry point for post - kernel initialization
    INIT_ORDER  usrKernelCoreInit               \
        INCLUDE_MEM_MGR_FULL                    \
        INCLUDE_MEM_MGR_BASIC                   \
        INCLUDE_MMU_BASIC                       \
        INCLUDE_MMU_FULL                        \
        INCLUDE_MMU_MPU                         \
        INCLUDE_SYSCLK_INIT                     \
        INCLUDE_SELECT                          \
        usrIosCoreInit                          \
        usrKernelExtraInit                      \
        usrIosExtraInit                         \
        usrNetworkInit                          \
        INCLUDE_SELECT_SUPPORT                  \
        usrToolsInit                            \
        INCLUDE_CTORS_DTORS                     \
        INCLUDE_CPLUS                           \
```

```
            INCLUDE_CPLUS_DEMANGLER                \
            INCLUDE_HTTP                           \
            INCLUDE_USER_APPL
}
```

以 usrRoot 为例，其执行顺序是 usrKernelCoreInit、INCLUDE_MEM_MGR_FULL、INCLUDE_MEM_MGR_BASIC 等。实际代码参见程序清单 4-24。

程序清单 4-24　usrRoot 中组件的执行顺序

```
void usrRoot (char * pMemPoolStart, unsigned memPoolSize)
    {
    usrKernelCoreInit ();
    memInit (pMemPoolStart, memPoolSize);
    memPartLibInit (pMemPoolStart, memPoolSize);
    sysClkInit ();
    selectInit (NUM_FILES);
    usrIosCoreInit ();
    usrKernelExtraInit ();
    usrIosExtraInit ();
    usrNetworkInit ();
    selTaskDeleteHookAdd ();
    usrToolsInit ();
    cplusCtorsLink ();
    usrWindMlInit ();
    usrAppInit ();
    }

void usrIosExtraInit (void)
    {
    excShowInit ();
    excInit ();
    logInit (consoleFd, MAX_LOG_MSGS);
    pipeDrv ();
    stdioInit ();
    fioLibInit ();
    floatInit ();
    dosFsInit (NUM_DOSFS_FILES);
    usrDosFsInit ();
    tffsDrv ();
    }
```

2. 创建组件

当用户使用组件描述语言创建组件时必须遵循一些约定；同时在一个组件描述文件中组件描述之后，用户必须把相应的文件以及包含等级中的组件，放在合适的路径下才能保证工程管理工具读出所需的信息。当使用组件描述文件时，必须遵循以下规则：

- 组件以 INCLUDE_FOO 的形式命名；
- 文件夹以 FOLDER_FOO 的形式命名；
- 选集以 SELECT_FOO 的形式命名；
- 参数名称不能与任何其他目标类的名称相同，除此以外没有任何限制（例如：用户可以使用 FOO_XXX，但不能使用 INCLUDE_FOO）；
- 初始化群组必须以 initFoo 形式命名；
- 组件描述文件必须有 cdf 后缀；
- 所有的 CDF 文件名称以两个十进制数字开始（即 00xxxx. cdf），开始的这两个数字控制一个目录中 CDF 文件读取的顺序。

组件描述文件通常在工程创建时被读取，就用户而言，CDF 文件的读取顺序是很重要的。如果两个文件描述同一组件的同样特征，则它将读取后一个而不考虑以前的。优先权通常以两种约定方式建立：

① CDF 文件放在确定的目录下，这些目录以如下顺序读取：

- $(WIND_BASE)\target\config\comps\VxWorks：包含一些通用的组件；
- $(WIND_BASE)\target\config\comps\VxWorks\arch\arch：包含特定体系结构的组件；
- $(WIND_BASE)\target\config\bsp：包含特定 BSP 的组件；
- 工程目录：包含该工程的一些特定组件。

② 在同一个目录中，文件名头部的 2 个数字决定读取顺序。

在一个目录中，系统会根据每个 CDF 文件名的前 2 个数字来决定读取的顺序。风河公司保留开始的 50 个数字，也就是 00xxxx. cdf～49xxxx. cdf，剩下的数字（50～99）预留为第三方。这些编号的优先级是数值大的优先读取。

如果用户需要创建一个新的组件，则须根据内容的特性和优先权级别将其放在适当的路径下。创建组件的具体过程如下：

(1) 命　名

为了创建一个新的组件，首先要给它命名，并为其准备一个简单的描述信息。

```
Component INCLUDE_FOO{
NAME foo component
SYNOPSIS this is an example component
```

在声明的组件 INCLUDE_FOO 中，名称就是 foo；名称和描述仅是用户对组件的说明，而对初始化顺序和依赖性并无影响。

(2) 描述与代码相关的部分

用户要通过定义模块来描述用户组件的代码部分。如果用户组件 INCLUDE_FOO 有与其相关的目标模块或源代码，则可采用多种方式来指定这一信息。在下面的例子中，列出了 fooLib. o 和 fooShow. o：

```
MODULES fooLib.o fooShow.o
HDR_FILES foo.h
ARCHIVE fooLib.a
CONFIGLETTES fooConfig.c
```

使用 HDR_FILES 属性来指定任何与组件相关的头文件，如 foo. h。

使用 CONFIGLETTES 属性指定与配置有关的文件名称（如 fooConlig. c），配置或初始化程序中应涉及组件中的参数，否则该组件没有任何作用。

(3) 设定初始值

如果用户的组件存在初始化代码，则可使用组件目标类的 INIT_RTN 来指定初始化程序，例如：

```
INIT_RTN foolnit(argl,arg2);
```

如果用户使用组件目标以外的模块，则可通过 LINK_SYMS 获取一个外部连接：

```
LINK_SYMS _fooRtnl
```

(4) 建立初始化顺序

初始化顺序是很重要的。用户可以控制某个组件在初始化过程中何时被引用。一个被声明为某个初始化群组中的组件，默认情况下会在该群组的最后被初始化；当然，用户也可通过使用 INIT_BEFORE 来改变这种默认情况。例如：

```
_INIT_ORDER usrRoot
INIT_BEFORE INCLUDE_USER_APPL
```

在上例中，INCLUDE_FOO 被申明为 usrRoot 初始化群组的成员之一，并在 INCLUDE USER_APPL 之前被初始化。用户还可以有另一种选择，即创建一个新的初始化群组并声明 INCLUDE_FOO 为一个成员。

(5) 链接有帮助的文件

如果该组件有相关的帮助文档，则可使用 HELP 来指定相关的参考条目（HTML 格式）。例如：

```
HELP fooMan.html
```

(6) 定义从属物

使用REQUIRES、EXCLUDES和INCLUDE_WHEN来声明组件之间的依赖关系。

```
REQUIRES INCLUDE_LOGGING
EXCLUDES INCLUDE_SPY
INCLUDE_WHEN INCLUDE_POSIX_AIO INCDUDE_POSIX_MQ
```

在上例中,REQUIRES声明INCLUDE_LOGGING必须被配置;而EXCLUDES则声明了INCLUDE_SPY不能与INCLUDE_FOO同时被配置。INCLUDE_WHEN告诉系统,当组件INCLUDE_POSIX_AIO和INCLUDE_POSIX_MQ被包含时,INCLUDE_FOO也必须被包含。

(7) 列出相关参数

在组件目标中,使用CFG_PARAMS来声明所有相关的参数。例如声明参数FOO_MAX_COUNT:

```
CFG_PARAMS FOO_MAX_COUNT
```

(8) 定义参数

对于每个CFG_PARAMS中声明的参数,还必须进一步说明其类型以及默认值。

```
Parameter FOO_MAX_COUNT{
NAME Foo maximum
TYPE uint
DEFAULT 50
}
```

(9) 定义群组成员

一个组件必须与一个文件夹或一个选集联系,否则它在工程管理工具中将是不可见的。例如:

```
_CHILDREN FOLDER_ROOT
```

CHILDREN声明INCLUDE_FOO是文件夹FOLDER_ROOT的子组件。

(10) 创建组件

只有当工程工具与包含的组件联系起来时,它才会分析相关的目标文件(库文件或OBJ文件)。这就产生了一个问题:为了了解一个特殊的目标文件,工程工具需在组件真正被添加进去之前分析它们。也就是说,如果用户把ARCHIVE申明的一个组件加进去,则配置分析会在不知道ARCHIVE值的情况下完成。所以,如果用户的组件包含一个库文件以及几个目

标模块，则用户应创建一个虚拟组件。这个虚拟组件是始终要被包含的，它使工程工具知道一个新的文件应被读取；这个组件可以称为 INSTALL_FOO，它应仅包含 NAME、SYNOPSIS 和 ARVHIVE。在 INSTALL_FOO 被添加进去之前，用户不能增加来自相同组件文件的其他组件。

(11) 产生工程文件

工程工具会自动根据组件的描述信息和配置来为每个工程创建一个配置文件(prjConfig. c)。要产生这个文件，可以选择 Build→Build 来编译这个新的工程，系统会自动生成 prjConfig. c 文件。

3. 测试新组件

用户可以进行几个测试来校验组件的正确性：

(1) 核对句法和语义

这是最简单的校验测试。首先编辑文件 $(WIND_BASE)/host/resource/tcl/app－config/Project /cmpTestLib. tcl，设置一个有效的 BSP 目录名(如 mvl62)，然后运行 cmpTest：

```
>cd $(WIND_BASE)/host/resource/tcl/app_config/Project
>wtxtcl
wtxtcl>source cmpTestLib.tcl
wtxtcl>cmpTest
```

句法和语义的错误，可以通过 cmpTest 测试来排除；根据测试输出作出相应改变，直至运行该测试没有错误。

(2) 核对组件从属物

用户可以通过运行 cmpInfo 来测试用户组件中的依赖关系。例如：

```
wtxtcl> cmpInfo <component name>
```

这报告反馈给用户组件与其他组件之间的关联信息。

(3) 核对工程工具组件等级

通过核对工程工具组件等级来确认用户所添加的选集、文件夹和新组件是否被正确包含。打开"工程工具"并进入 VxWorks，查看新组件是如何出现在文件夹树中的。调用 Properties 来核对与一个组件相关的参数及其默认值。

如果用户已添加了一个包含组件的文件夹，并且已把文件夹放在用户的配置中，则工程工具等级会默认地用黑体显示那个文件夹的所有组件(即 DEFAULTS 值)。

4.5 BSP 的调试

在编写 BSP 阶段，有很多可用的调试方法，但在串口通信或网络通信驱动没有编写好之前，无法使用 Wind Debugger 来调试。这时如果没有硬件调试器，则可使用一些板载的设备(如发光管指示器和串口等)来调试 BSP，但使用一个标准的 ICE 调试工具将使调试工作变得简单。

无论使用哪一种调试方法，首先都需要在熟悉目标系统的硬件配置之后，编写一个启动代码，这对后续的工作会大有帮助。当然 Tornado 可以生成 bootRom，但要生成 bootRom 需要完成很多工作(例如编写网络驱动和文件系统等)。这些工作要比编写一个纯粹的 Bootloader 要复杂得多。在 JX2410 上，使用的是 u - boot，只要实现下载和启动 VxWorks 就足够了。关于u - boot的移植和使用，可以参考相关文档。

接下来按照 4.4 节的内容编写板级驱动：首先是时钟驱动和中断驱动；然后编写串口驱动；最后使用编写的 BSP 来生成 VxWorks 映像，再通过启动程序将其下载到目标板中，检查运行情况。对于有 ICE 调试工具的用户，也可不编写启动程序，而使用 ICE 调试工具来下载目标代码。

可在程序的特定位置添加一些调试信息，例如通过使数码管显示不同的数字来确认代码的执行情况。在 VxWorks 内核和串口驱动运行起来后，即可使用 Wind Debugger 来调试后续的驱动。关于 Wind Debugger 的使用参见第 3 章。下面的步骤用来生成一个简化的 VxWorks 内核，并且使用调试器对它进行调试。

① 将第三方提供的 BSP 拷贝到 PC 机的某个目录下。下面以目录 E:\VxWorks\S3C2410\BSP 为例来说明。

② 启动 Tordado II 集成开发环境。

③ 新建一个工程(Create a bootable VxWorks image)，如图 4 - 14 所示。

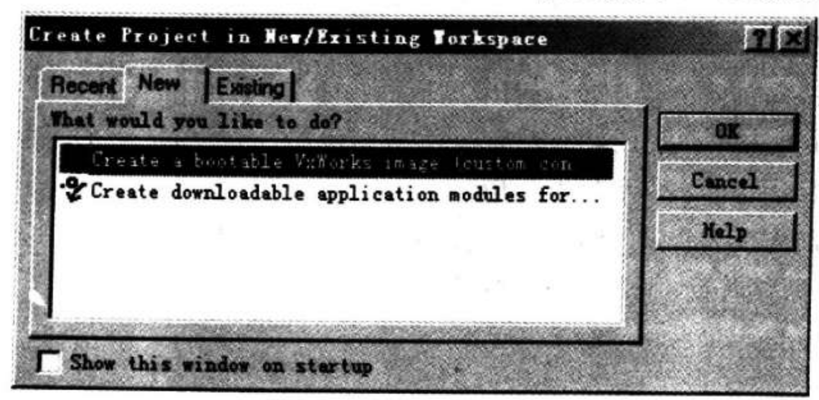


图 4 - 14 选择工程的类型

④ 输入工程的名称和位置等信息，在 E:\VxWorks\S3C2410\BspTest 下创建一个名为 BspTest 的工程，如图 4-15 所示。

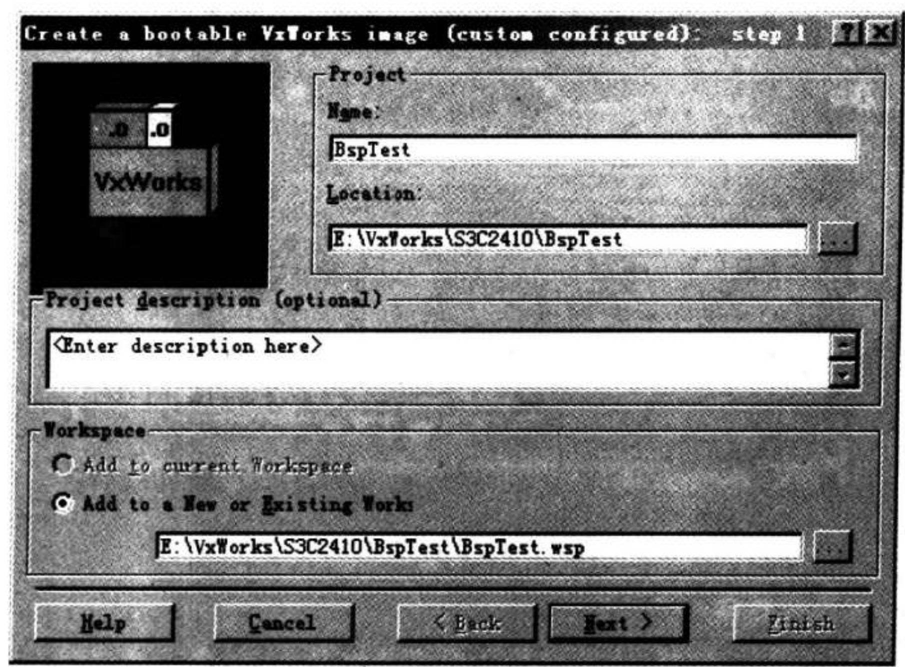


图 4-15　新建 VxWorks 工程

⑤ 选择 BSP 模板，如图 4-16 所示。

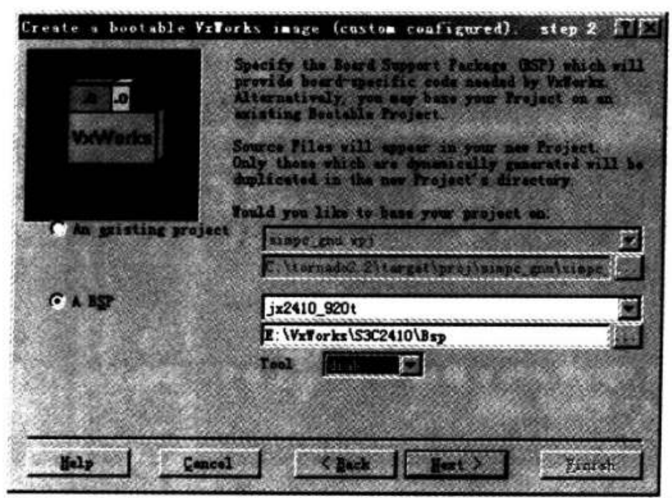


图 4-16　选择参考的 BSP 模板

⑥ 单击 Finish 按钮完成工程的创建。

⑦ 通过工程管理器，参见图 4-17 调整工程的某些属性。例如通过在 network components 选项上右击并选择 Exclude 'network components'来去掉网络支持。

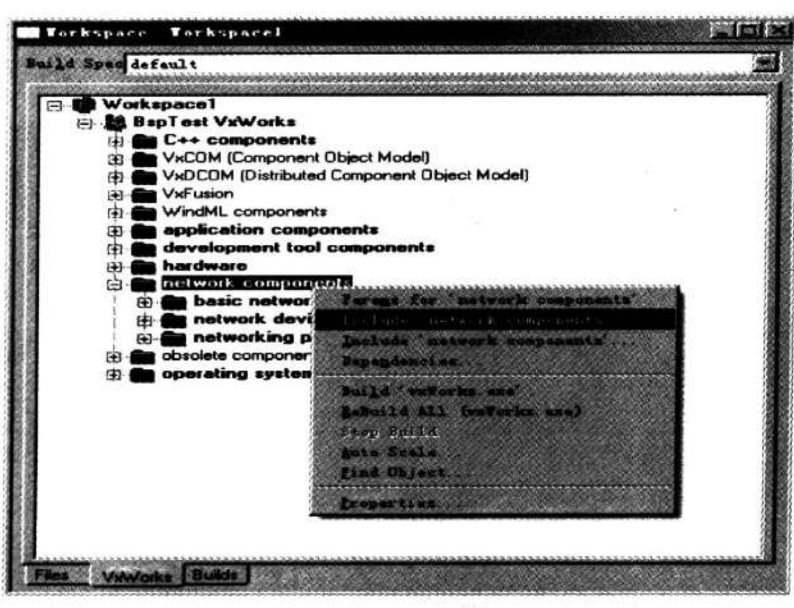


图4-17　调整系统组件

⑧ 使用Build选项来编译该工程。

⑨ 在工程编译完毕后，使用ADT将其下载到目标板上(新建一个空的工程，选择用户下载即可)。

⑩ 连接教学系统的串口0，并设置波特率为“115200，8N1”，在ADT中使用go命令运行下载的程序。可以看到如图4-18所示的运行界面。

⑪ 添加Shell功能，在Tornado工程管理器中将Shell部件添加到工程中(如图4-19所示)，再编译、下载并运行。

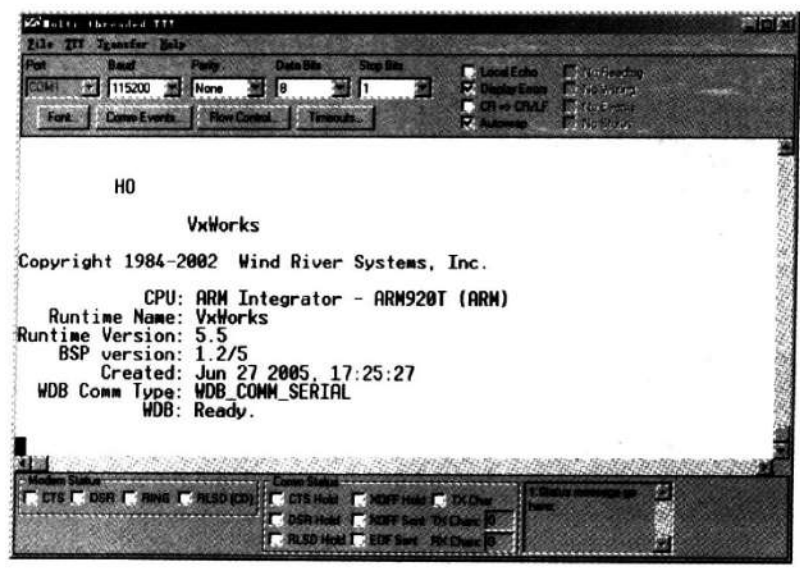


图4-18　串口辅助工具MTTTY的运行界面

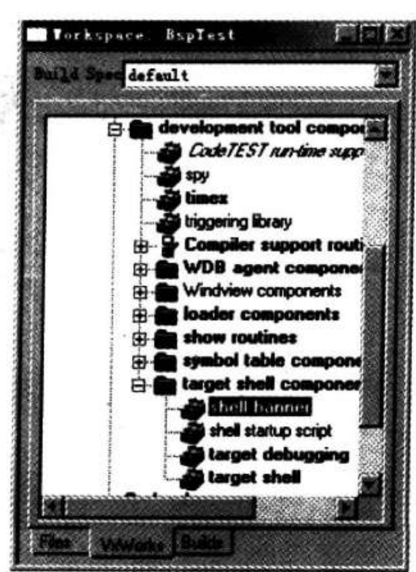


图4-19　添加Shell组件

添加 Shell 后的运行结果如图 4－20 所示，通过 Shell，在提示符“－>”下可进行一些简单的操作，例如查看进程信息和内存等。

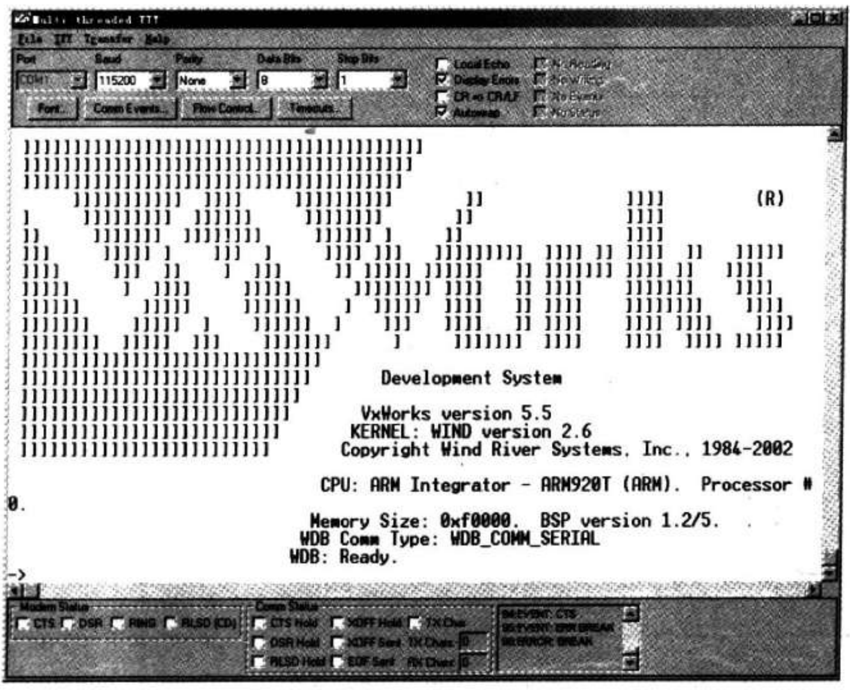


图 4－20 VxWorks Shell 界面

注意：在两次下载运行之间，需将教学系统复位。

使用 ADT 调试 BSP 的步骤如下：

① 新建一个空的工程。运行 ADT→File→New，新建一个工程，如图 4－21 所示。

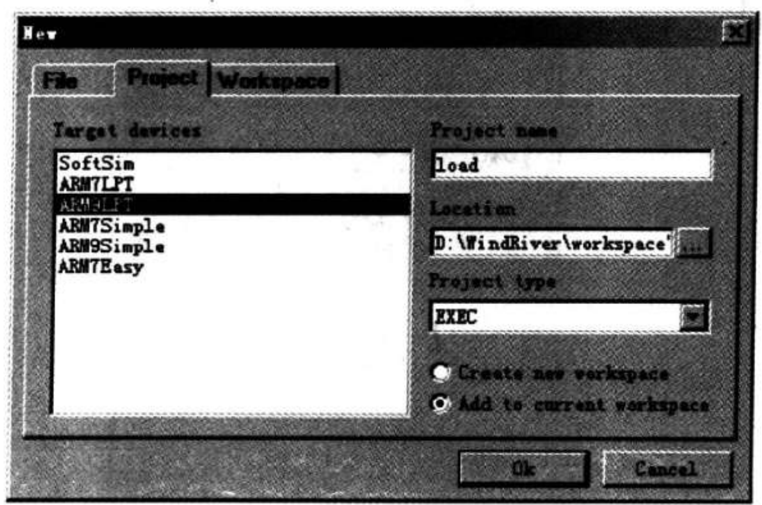


图 4－21 新建工程

② 选择下载文件。在工程的设置对话框中选择 Debug 页，配置 Download case 为 Customized case，并选择要下载的 VxWorks 目标代码(此时下载的目标地址无须设置)，如图 4－22 所示。

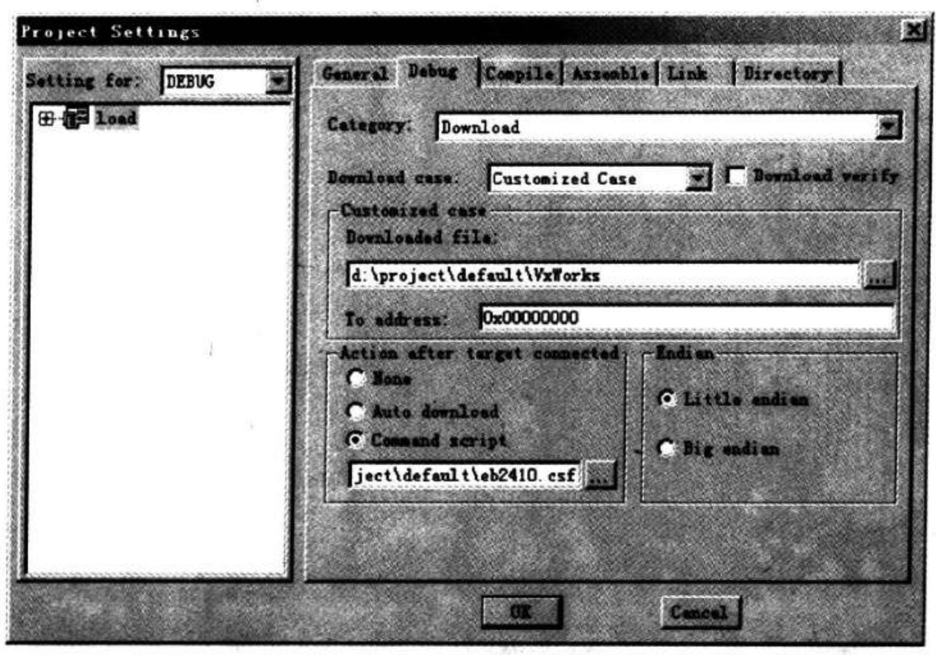


图 4－22 选择下载文件

③ 选择调试设备。根据使用的 ICE 调试设备选择驱动程序，参照图 4－23 选择 ARM9LPT。

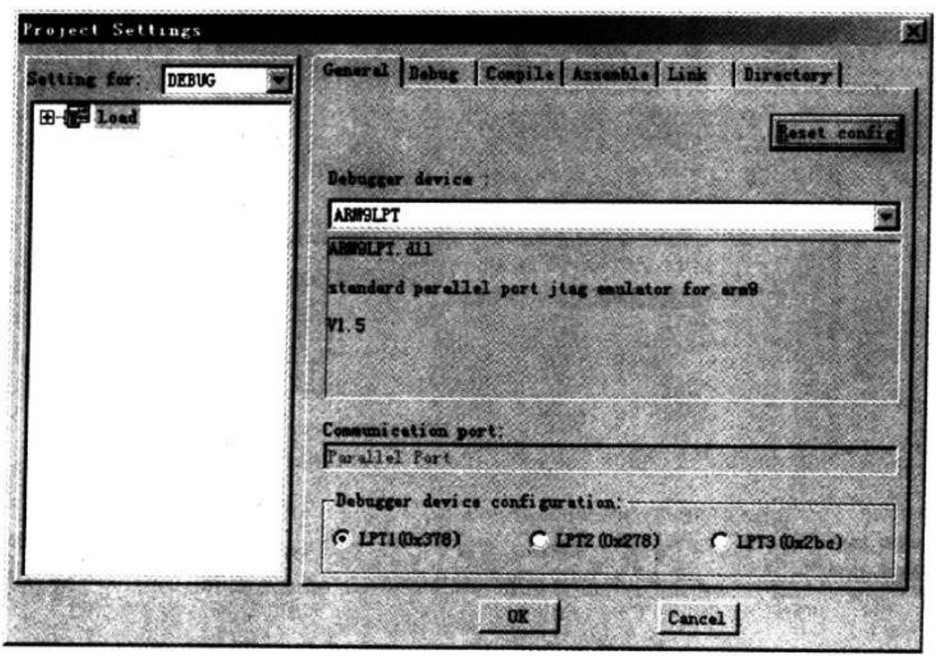


图 4－23 选择调试设备

④ 下载程序。首先与目标机建立通信连接，然后使用 download 将程序下载到目标板上，如图 4－24 所示。

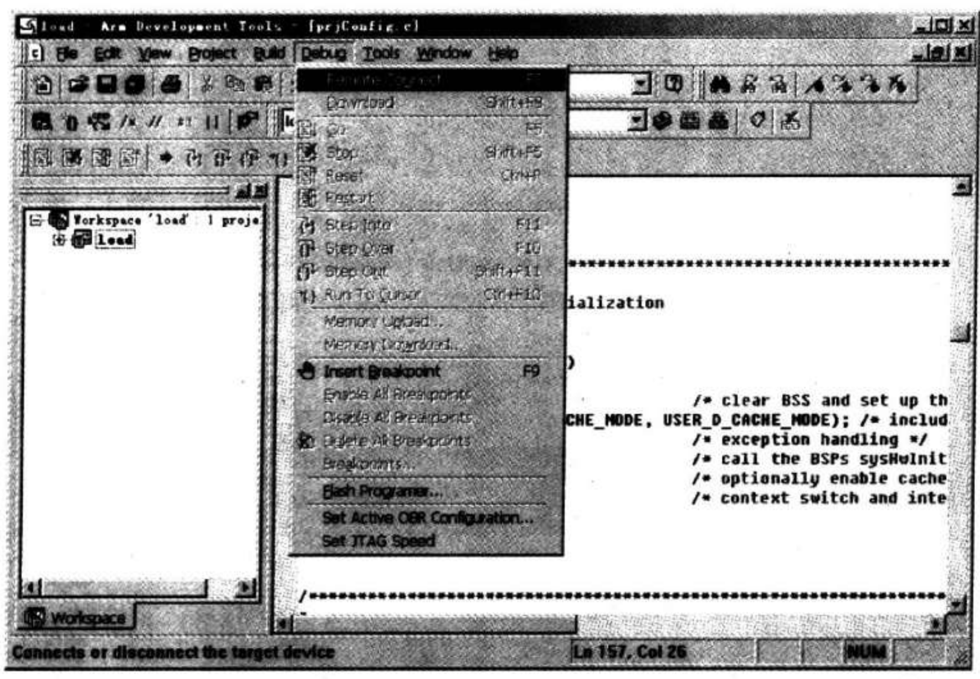


图 4－24 链接目标机和下载程序

⑤ 执行程序。选择 Debug→go 运行程序，见图 4－25。

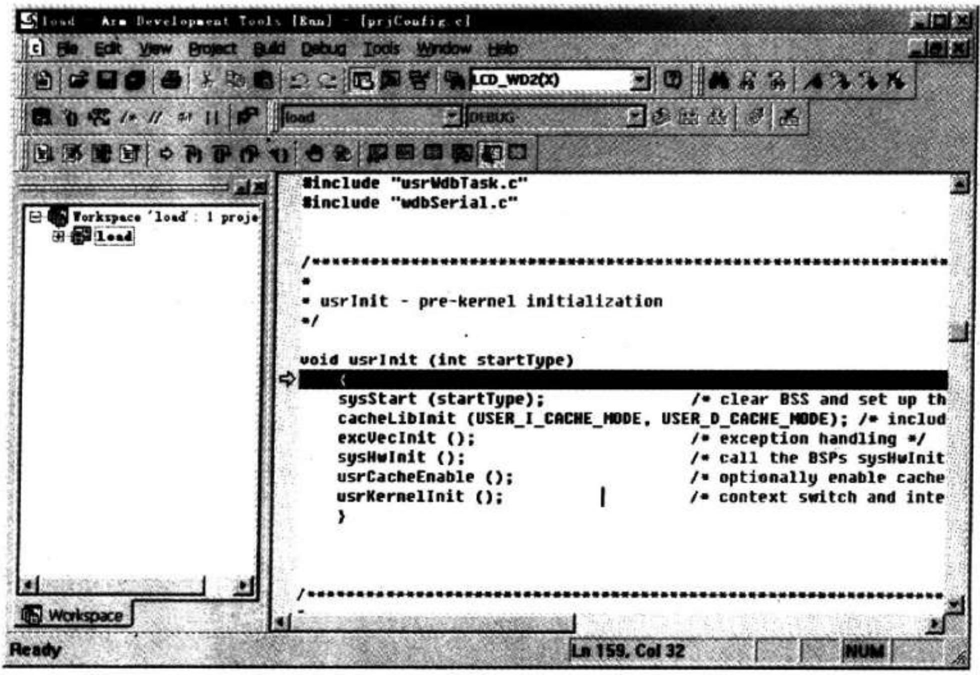


图 4－25 执行程序

⑥ 单步跟踪以及其他调试。可以使用断点功能，检查程序的执行情况（如图 4－26 所示）；也可在目标系统停止运行后，检查变量以及其他信息。

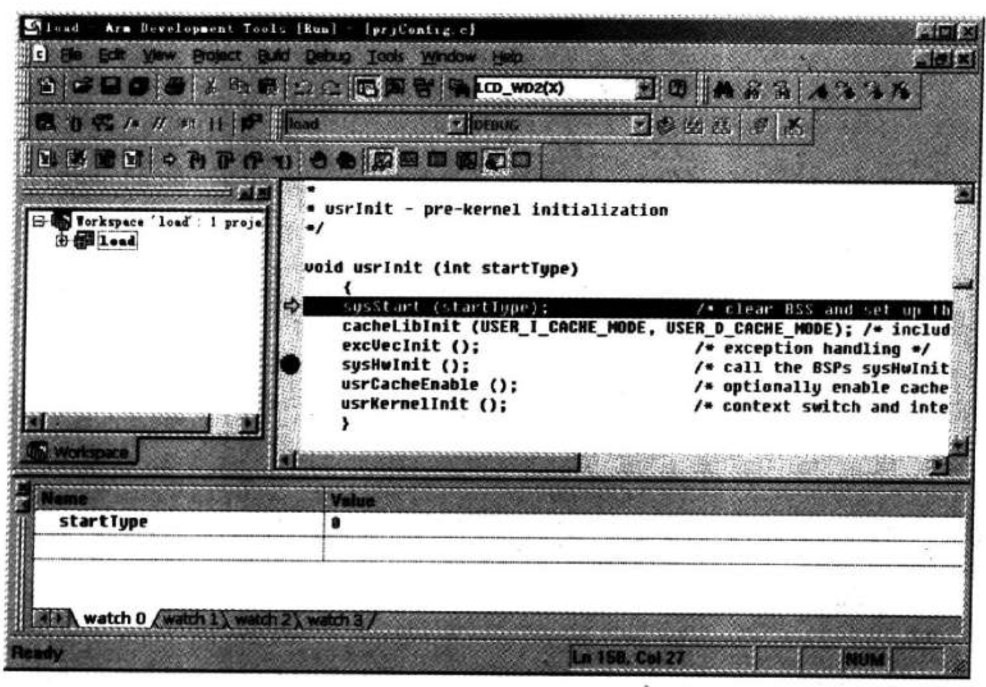


图 4－26　单步跟踪调试

在使用 ICE 硬件调试器时有一点需要注意：多数硬件仿真器不支持 MMU 的跟踪调试，所以在进行 MMU 切换时须跳过去；另外还有一些须关闭 MMU 的 Cache 功能。

第 5 章

VxWorks 驱动程序的编写

5.1 设备驱动分类及特点

多数操作系统为了屏蔽硬件的具体操作细节，并且为上层应用程序提供统一接口，都会引入驱动程序的概念，VxWorks 也不例外。图 5-1 和图 5-2 分别描述了 VxWorks 应用程序、驱动程序和硬件之间的层次关系，以及 I/O 系统与驱动程序之间的关系。

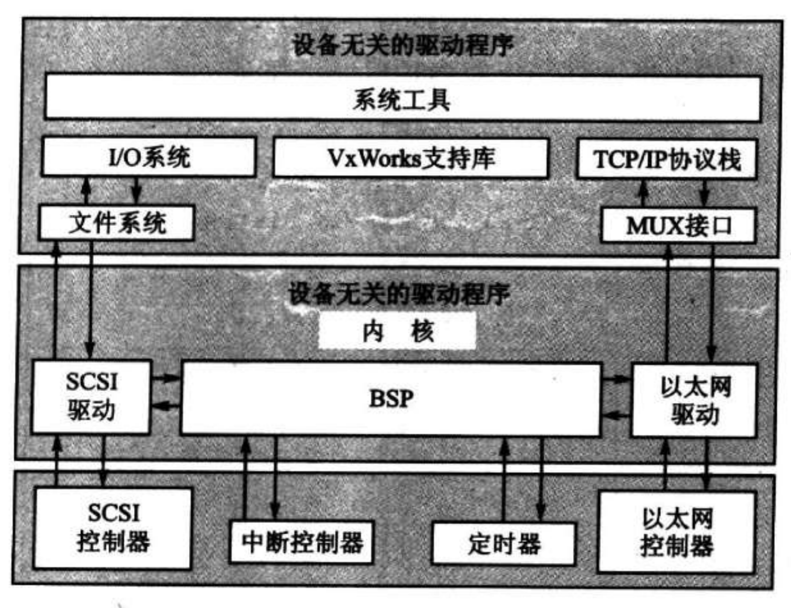


图 5-1　VxWorks 设备驱动模块图

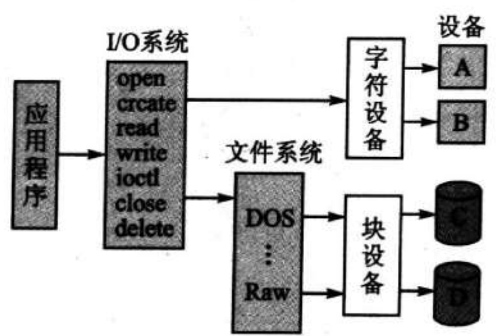


图 5-2　VxWorks I/O 系统

从图 5－2 中可以看出，VxWorks 标准设备驱动程序基本都是通过 I/O 系统来存取的。很多操作系统（如 Linux 和 QNX 等）都采用这种方法来管理设备。大体上可把设备驱动程序分为两大类：

① 基于 I/O 文件系统的设备：包括基于 I/O 系统的字符类设备和块设备。这些设备都可采用 ioLib 系统库提供的接口函数 create()、open()、close()、read()、write()和 ioctl()等来操作。VxWorks 的 I/O 系统由基本 I/O 及含缓冲的 I/O 组成，它提供标准的 C 库函数。基本 I/O 库与 Unix 兼容；而含缓冲的 I/O 则与 ANSI C 兼容。VxWorks 的 I/O 系统有其独特性，因而它比其他 I/O 系统更快速，更灵活，这在实时系统中非常重要。

② 其他特殊设备：主要指一些非基于 I/O 文件系统的设备，如串口设备和网络设备等。这些设备由于其自身特性，虽然不能通过标准 I/O 来进行存取，但也都有各自相关的标准，如图 5－3 所示。后面会通过具体的例子来详细说明这些设备的编写方法。

虽然 VxWorks 的 I/O 设备与 Unix 很类似，但二者也有区别，主要包括以下几点：

- VxWorks 的设备驱动可以动态安装和卸载；
- 除了标准的输入、输出和错误输出外，VxWorks 的文件描述符是全局的，可被任何一个任务存取；
- VxWorks 的设备驱动有优先级之分，这取决于调用它的任务的优先级，而 Unix 中的设备驱动则执行在系统模式下，无优先级之分。

当用户调用一种基本 I/O 操作时，I/O 系统将用户请求反映给特定驱动程序的相应操作函数；驱动程序的操作函数运行在调用它的任务上下文中，就像是从应用程序中直接调用该操作函数。因此，驱动程序可调用该任务可执行的任何操作，包括对其他设备进行 I/O 操作。这意味着多数驱动程序必须对关键程序代码采取一种互斥的操作机制。通常，采用的互斥机制是使用 semlib 函数库提供的信号量。

VxWorks 提供了对很多设备的驱动程序支持，如表 5－1 所列。

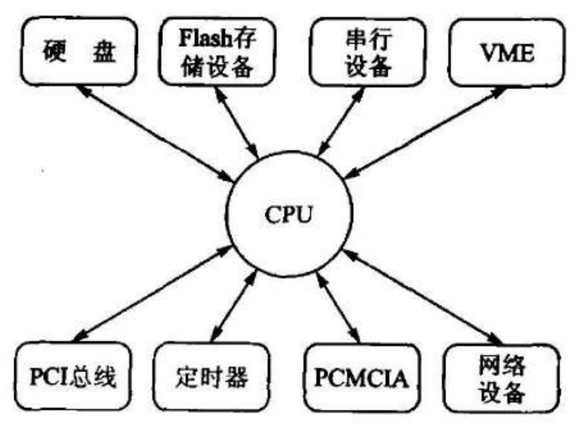


图 5－3 VxWorks 的特殊设备

表 5－1 常见的 VxWorks 设备驱动程序

驱动模块名称	说 明
ttyDrv	终端设备驱动程序
ptyDrv	伪终端设备驱动程序
pipeDrv	管道设备驱动程序
memDrv	伪存储器设备驱动程序
nfsDrv	NFS 客户端文件系统设备驱动程序
netDrv	用于远程访问的网络设备驱动程序
ramDrv	用于创建 RAM 虚拟磁盘的驱动程序
scsiLib	SCSI 接口库
其他	其他与硬件有关的设备驱动程序

(1) 终端及伪终端设备

VxWorks 提供终端(tty)和伪终端(pty)设备驱动，所谓的"伪终端"也就是终端仿真。伪终端对于应用程序很有用，最常见的如远程登陆(Remote Login Facilities)。

VxWorks 的 Serial I/O Devices 属于含缓冲的连续字节流 tty 设备。每个设备有一个输入和输出环形缓冲，从一个 tty 设备读/写数据实际上是从输入/输出环形缓冲中读/写数据。每个环行缓冲的大小要在创建设备时指定。

tty 设备有很多功能选项会影响设备的运行，这些选项可以通过设备选项控制字中的相应位来设定，例如：

```
ioctl( fd, FIOSETOPTIONS, OPT_TERMINAL );
```

在 ioLib.h 中定义了可选项的名称，对这些选项的说明见表 5-2。

表 5-2 tty 控制选项

控制标识	描 述
OPT_LINE	设置线路传输的模式
OPT_ECHO	设置回显输出
OPT_CRMOD	设置将输出的"回车"字符翻译为"换行"标志，而将"换行"字符翻译为"回车"+"换行"
OPT_TANDEM	设置软件流量控制(使用 CTRL+Q 和 CTRL+S)
OPT_7_BIT	从所有的输入字节中取出最高位
OPT_MON_TRAP	使能特殊 ROM 监控程序字符(CTRL+X)
OPT_ABORT	使能目标机 Shell 程序特殊中止字符(CTRL+Z)
OPT_TERMINAL	将上述选项位设置为 1
OPT_RAW	取消上述所有设置

tty 设备有两种工作模式：原始模式 (Raw Mode)和线性模式(Line Mode)。原始模式是缺省模式；线性模式可由表 5-2 中提到的 OPT_LINE 来选择。在原始模式下每个输入的字符可在用户输入时立即生效。一个工作在原始模式下的 tty 设备，除非控制可选项，否则将无法对正在进行的输入做修改；而对于线性模式，所有的输入字符都被保存在缓冲区中直到一个"换行"字符被输入，通常在输入的过程中可以使用特殊字符(CTRL+H/U/D 字符)来修改输入。在 Tornado 的 Shell 下的输入就是一个很好的线性模式例子，系统定义的控制字符如表 5-3 所列。

除了用 ioctl 进行 tty 模式的设置外，ioctl 例程还可实现其他功能的控制，见表 5-4。

表 5-3 系统定义的 tty 设备控制字符

特殊字符	描　述
CTRL+H	退格字符，可以连续删除当前行中的最后一个字符，直到该行的开始
CTRL+U	删除一行字符
CTRL+D	文件结束符，可以使当前行的内容导入输入缓冲区，而不必等待换行或 EOF
CTRL+C	目标机 Shell 程序中止
CTRL+X	软中断监控程序
CTRL+S	暂停输出
CTRL+Q	继续输出

表 5-4 tyLib 提供的 I/O 控制功能

控制编码	说　明
FIOBAUDRATE	设置波特率
FIOCANCEL	取消上次读/写操作
FIOFLUSH	清除缓冲区
FIOGETNAME	获取指定文件描述符对应的文件名
FIOGETOPTIONS	获取当前的选项配置
FIONREAD	获取缓冲区中未读字节数
FIONWRITE	获取输出缓冲区中的数据字节数
FIOSETOPTIONS	设置设备的选项字

(2) 管道设备

管道是利用 VxWorks 的 I/O 系统进行任务间通信的虚拟设备。消息发送者先将消息写到管道上，若有接收则从管道中读出这些消息。

可以使用 pipeDevCreate()函数来创建管道，并给定最大消息数和每个消息的最大长度。例如：

```
pipeDevCreate("/pipe/name", maxMsgs, maxLen);
```

当试图向一个满的管道写新消息时，该任务将会变为阻塞状态，直到管道中的消息被读出；当向管道写入一条超过最大消息长度的消息时，将产生错误。

VxWorks 的管道非常独特，它允许在中断服务例程中进行管道的写操作。中断服务例程通过调用 write()函数来写管道，但是一旦管道满了，由于中断服务例程无法阻塞，则相应的消息将被丢弃。管道的控制命令如表 5-5 所列。

表 5-5 pipeDrv 提供的 I/O 控制功能

编　码	说　明	编　码	说　明
FIOFLUSH	清除管道中的消息	FIONMSGS	获取管道中的消息数
FIOGETNAME	获取指定文件描述符对应的管道名	FIONREAD	获取管道中第一条消息的长度

(3) 伪存储设备

内存设备其实并非一个真正的设备，它无需任何驱动即可进行读/写操作，因此通常称为“伪内存设备”。它允许 I/O 系统把内存当作一个伪 I/O 设备来存取。在设备被创建时内存的位置和大小将被确定，这使 VxWorks 可在启动时保存数据或多个 CPU 之间共享数据。伪

内存设备不像虚拟磁盘(RAMDISK)那样需要文件系统的支持,memDrv 可以使用 I/O 调用在绝对位置上读/写数据。

为了支持伪内存设备,在配置 VxWorks 内核时,需要定义 INCLUDE_USR_MEMDRV 来添加相关组件;然后使用 memDrv()初始化设备;最后用 memDevCreate()创建设备。memDevCreate 的函数原型为:

```
memDevCreate(char * name, char * base, int length);
```

参数中的 base 是为设备分配的存储区绝对地址;而 length 则指明了存储区的容量。memDrv 的控制命令如表 5-6 所列。

表 5-6 memDrv 提供的 I/O 控制功能

编 码	说 明
FIOSEEK	设置文件的当前偏移
FIOWHERE	返回文件的当前偏移

(4) 网络文件系统设备

网络文件系统设备允许使用 NFS 协议来存取远程网络文件。VxWorks 下的 NFS 驱动作为一个 NFS 客户端来存取网络中 NFS 服务器上的文件。VxWorks 也允许运行一个 NFS 服务器来输出文件到其他远程机器上。当使用 NFS 设备可以创建、打开和存取远程的文件时,就像存取本地硬盘上的文件系统一样,屏蔽了底层网络系统的具体细节。存取远程文件系统的方法是使用 nfsMount()函数来安装指定的文件系统,并为其创建一个 I/O 设备。nfsMount带有 3 个参数:NFS 服务器的名称、主机文件系统的名称和本地文件系统的名称,例如:

```
nfsMount("cvt", "/nfsroot", "/vxNfs");
```

要使用 NFS 文件系统必须在配置内核时定义 INCLUDE_NFS。NFS 文件系统的 ioctl 控制命令参见表 5-7。

表 5-7 nfsDrv 提供的 I/O 控制功能

编 码	说 明	编 码	说 明
FIOFSTATGET	获取文件状态	FIOSEEK	设置文件指针偏移
FIOGETNAME	获取指定文件描述符对应的文件名	FIOSYNC	向远程主机刷新文件
FIONREAD	获取文件中未读取的字节数	FIOWHERE	获取当前文件指针的位置
FIOREADDIR	读取下一个目录入口信息		

(5) 非 NFS 网络设备

VxWorks 为了访问无 NFS 服务器的网络设备,提供了一个可以替换 NFS 文件系统的方法,即使用 RSH(Remote Shell Protocol)和 FTP(File Transfer Protocol)协议来完成文件的存取。这要借助于 netDrv 驱动程序的支持,而且它们都使用 nfsDrv 来实现。当一个远程文件通过 RSH 或 FTP 被打开时,整个文件会被拷贝到内存中,所以要注意文件是否过大而无法被

拷贝到本地内存。一旦文件被拷贝到内存中，则 read 和 write 只是对本地内存中的文件进行操作。当关闭文件时，如果它被改变，则会被拷贝回远程机器。

要使用 RSH 和 FTP 存取远程主机，必须先使用 netDevCreate()函数创建一个网络设备，并选择是 RSH 还是 FTP。例如：

```
netDevCreate( "cvt", "cvt", 0);
```

网络设备上的文件可被创建、打开和处理，就像本地的主机文件一样，但在该主机内要有足够的存取权限。例如：打开文件"cvt:/ftproot/test"实际上就是打开主机 cvt 上的文件"/ftproot/test"。

(6) 虚拟磁盘设备

VxWorks 的虚拟磁盘设备(RAMDISK)是将内存模拟为磁盘，将所有数据保存在内存中。内存的位置和"磁盘"的大小依赖于用 ramDevCreate()函数所得到的结果，可以通过多次运行该函数来创建多个 RAMDISK.

虚拟磁盘所使用的内存可预分配，可将该地址作为参数传递给 ramDevCreate()函数；也可由该函数自动分配。设备被创建后必须进行文件系统相关的初始化，使用 dosFsDevInit()或 dosFsMkfs()将该文件系统与一个设备名字相关联。例如：要创建一个自动分配 200 KB 内存的虚拟磁盘，512 字节的扇区，一条磁道，没有扇区偏移。设备命名为"DEV1:"，使用 dosFs 文件系统，实现方法如下：

```
BLK_DEV                    *pBlkDev;
DOS_VOL_DESC               *pVolDesc;
pBlkDev = ramDevCreate (0, 512, 400, 400, 0);
pVolDesc = dosFsMkfs ("DEV1:", pBlkDev);
```

如果 RAM 磁盘中已包括了一个磁盘映像，那么 ramDevCreate()的第一个参数即存储区中 RAM 磁盘的地址。例如：

```
pBlkDev = ramDevCreate (0xc0000, 512, 400, 400, 0);
pVolDesc = dosFsDevInit ("DEV1:", pBlkDev, NULL);
```

关于 RAMDISK 的一些具体实现将在后续章节中介绍。

(7) SCSI 接口驱动

SCSI 是一个标准外设接口，一般被应用在硬盘、光驱、软驱和磁带设备上。SCSI 块设备 driver 与 dosFs 库兼容，提供多种目标配置。SCSI 技术本身比其他设备要复杂，这里不对其进行详细分析。

5.2 字符设备驱动

5.2.1 字符设备驱动程序

字符设备是指在 I/O 传输过程中以字符为单位进行传输的设备，如键盘、鼠标和打印机等。字符设备驱动程序一般都有 7 个基本 I/O 操作函数，但也不排除某些具体的字符设备驱动程序可能忽略其中的一个或几个操作函数。这 7 个基本 I/O 操作函数及其功能是：

- create()：创建设备；
- delete()：删除设备；
- open()：打开设备；
- close()：关闭设备；
- read()：读取设备中的数据；
- write()：向设备写数据；
- ioctl()：设置设备的方式字。

驱动程序除了具有上述 7 种基本 I/O 操作函数以外，还包含以下两种操作：

① 初始化函数负责在 I/O 系统中安装驱动程序；驱动程序将相应的设备与所需的资源关联起来；然后初始化函数再执行其他必需的硬件初始化操作。

② 将驱动程序加载到 I/O 系统中，这类函数一般称为 xxDevCreate()。

对于一些简单的驱动程序，往往可能将这两个函数合二为一，例如下面将要介绍的键盘驱动程序。

在编写字符设备驱动程序时还存在一个问题：如果设备的缓冲中目前没有任何数据，但应用程序此时发起了一个 read()调用，那么此时该应用程序就有可能阻塞在这个位置上，直到缓冲中有了足够的数据。当这个应用程序只须处理一个设备的数据时，阻塞还不至于产生问题，但如果该应用程序须处理多个设备的数据时(例如同时响应键盘和触摸屏的数据)，那么阻塞在某个设备操作上就变得不可接受。为了解决类似问题，需要借助于系统提供的 select 机制。实际上这个问题并不是字符设备独有的，很多设备驱动都存在同样的问题，且处理办法与字符设备类似。

支持调用 select()函数的用户驱动程序，可使任务同时等待多个设备的输入，或者允许任务为设备执行指定的 I/O 操作设定最长等待时间。这一功能的实现需要借助于 selectLib 函数库。这个系统支持该库提供的大多数功能，但在编写驱动程序时仍须进行一些必要的操作。如果用户希望设备可以进行如下操作，那么用户的驱动程序必须支持 select()函数的调用：

① 任务需要为等待一个设备执行 I/O 操作设定时间限制。例如：可能会为一个 UDP 套

接字接收信息包操作设定一个时限。

② 一个驱动程序同时支持多个设备，而运行的任务可能会同时等待这些设备。例如：可能会为不同优先级的数据传输操作建立多个管道。

③ 任务等待某个设备的 I/O 操作，同时该设备等待其他设备的 I/O 操作。例如：一个服务器任务可能会使用管道和套接字。

在 select()函数执行时，驱动程序必须保存一个记载等待任务的表。当被等待的设备可用时，驱动程序激活所有等待该设备的任务。

如果一个设备驱动程序支持调用 select 函数，那么它必须声明一个 SEL_WAKEUP_LIST 的数据结构(通常作为设备描述符结构的一部分)，并且调用 selWakeupListInit()函数对该结构进行初始化。这一过程在驱动程序中的 xxDevCreate()函数中完成。SEL_WAKEUP_LIST 的结构成员见程序清单 5-1。

程序清单 5-1　$(WINDBASE)\target\h\private\selectLibP.h

```
typedef struct selWkNode
    {
    NODE      linkedListHooks;          /* 唤醒链表的钩子 */
    BOOL      dontFree;                 /* 该节点是否为第一个自由节点 */
    int       taskId;                   /* 任务 ID */
    int       fd;                       /* 文件描述符 */
    SELECT_TYPE    type;                /* 任务进行的操作类型 */
    } SEL_WAKEUP_NODE;

typedef struct
    {
    SEMAPHORE         listMutex;        /* 链表使用的信号量 */
    SEL_WAKEUP_NODE firstNode;          /* 唤醒链表的首节点 */
    LIST              wakeupList;       /* 唤醒链表 */
} SEL_WAKEUP_LIST;
```

程序清单 5-2 为一个使用 select 机制的驱动程序的初始化部分。

程序清单 5-2　使用 select 机制驱动程序的初始化

```
typedef struct test_dev
    {
    DEV_HDR            ioDev;
    SEL_WAKEUP_LIST    selList;
```

```
    BOOL                txRdy;
    BOOL                rxRdy;
    } TEST_DEV;
LOCAL int testDrvNum = 0;
STATUS testDevCreate
    (
    char     * name
    )
{
    TEST_DEV * pTestDev ;

    /* 必要的硬件初始化 */
    :

    /* 安装设备 */
    testDrvNum = iosDrvInstall (…);
    if (testDrvNum <= 0)
        {
        return (ERROR);
        }
    if ((pTestDev = (TEST_DEV * ) calloc (1, sizeof (TEST_DEV))) == NULL)
    {
    return (ERROR);
    }

   /* select 链表初始化 */
   selWakeupListInit (&pTestDev -> selList);

   /* 将设备添加到 I/O 系统中 */
   return (iosDevAdd (&pTestDev -> ioDev, name, testDrvNum));
}
```

当一个任务调用 select()函数后，selectLib 函数库调用驱动程序中的 ioctl()函数实现 FIOSELECT 或 FIOUNSELECT 功能。当 ioctl()函数实现 FIOSELECT 功能时，驱动程序必须执行如下操作：

① 通过调用 selNodeAdd()函数向 SEL_WAKEUP_LIST 结构中添加一个 SEL_WAKE-

UP_NODE节点，节点的内容由ioctl()函数的第三个参数提供。

② 调用selWakeupType()函数检查任务是否正等待从设备中读数据(SELREAD)，或设备是否准备好输入数据(SELWRITE)；如果设备已就绪，则驱动程序会调用selWakeup()函数，以确保调用select()函数的任务不会处于挂起状态。这样就避免当设备可用时任务仍处于阻塞状态。

③ 如果调用ioctl()函数实现FIOUNSELECT功能，那么驱动程序会调用selNodeDelete()函数将唤醒表中的SEL_WAKEUP_NODE节点删除。

④ 当某一个设备可用时，通过调用selWakeupAll()函数，唤醒所有等待该设备的任务。

程序清单5-3为一个简单例子。

程序清单5-3　支持select功能驱动程序的Ioctl

```
LOCAL int testIoctl
    (
    TEST_DEV     * pTestDev,          /* 设备指针 */
    int          request,             /* 请求代码 */
    void *       arg                  /* 参数 */
    )
    {
    switch(request)
        {
        case FIOSELECT:
            {
            SEL_WAKEUP_NODE * pNode = (SEL_WAKEUP_NODE * )arg;
            switch (selWakeupType (pNode))
                {
                case SELREAD:
                    /* 向唤醒链表中添加节点 */
                    selNodeAdd (&pTestDev -> selList, pNode);

                    /* 检查当前是否就绪,若就绪则直接唤醒等待的任务 */
                    if (pTestDev -> rxRdy) selWakeup (pNode);
                    break;
                case SELWRITE:
                    /* 向唤醒链表中添加节点 */
                    selNodeAdd (&pTestDev -> selList, pNode);
```

```
                /* 检查当前是否就绪,若就绪则直接唤醒等待的任务 */
                if (pTestDev -> txRdy) selWakeup (pNode);
                break;
            }
        }
        break;
    case FIOUNSELECT:
        {
        SEL_WAKEUP_NODE * pNode = (SEL_WAKEUP_NODE * )arg;
        switch(selWakeupType (pNode))
            {
            case SELREAD:
                /* 删除唤醒节点 */
                selNodeDelete (&pTestDev -> selList, pNode);
                break;
            case SELWRITE:
                /* 删除唤醒节点 */
                selNodeDelete (&pTestDev -> selList, pNode);
                break;
            }
        }
        break;

    case …
    ⋮
    }
}

LOCAL void testISR
    (
    TEST_DEV     * pTestDev,        /* 设备指针 */
    )
    {
    ⋮
    if (pTestDev -> rxRdy) selWakeupAll (&pTestDev -> selList, SELREAD);
    if (pTestDev -> txRdy) selWakeupAll (&pTestDev -> selList, SELWRITE);
    }
```

5.2.2 键盘驱动程序编写

1. 安装设备

任何设备驱动程序都需先通过适当的方法安装在系统中，然后才能被应用程序使用。字符类型的设备驱动程序使用 iosDrvInstall()和 iosDevAdd()来实现驱动程序的安装。

I/O 系统的功能是将用户的 I/O 请求转换成对相应驱动程序具体操作函数的调用。I/O 系统通过维护一个包括所有驱动程序操作函数的地址表来实现上述功能。通过调用 I/O 系统的 iosDrvInstall()可以动态安装驱动程序；该函数的参数就是 7 种基本 I/O 操作函数指针。iosDrvInstall()函数将这些地址写入驱动程序表的一块空闲存储区中，并返回这块存取区域的编号；编号用驱动程序号表示，可被与驱动程序相关联的设备使用。如果相应的函数指针为 0，则表示驱动程序不具备该项功能；对于非文件系统的驱动程序，close()和 delete()函数通常不起作用。iosDrvInstall()的原型是：

```
int iosDrvInstall (FUNCPTR pCreate,    FUNCPTR pDelete,    FUNCPTR pOpen,
                   FUNCPTR pClose,     FUNCPTR pRead,      FUNCPTR pWrite,
                   FUNCPTR pIoctl);
```

对字符类型设备驱动程序来说，必须首先实现设备的 7 个标准 I/O 函数，然后通过调用 iosDrvInstall()函数将设备的 I/O 函数指针作为参数传递给系统。这时系统会在设备登记信息表中记载该设备及其函数指针(如图 5-4 所示)，并返回一个设备驱动号；可以通过这个设备驱动号，进行后续相关操作。

	create	delete	open	close	read	write	ioctl
0	xxCreate	xxDelete	xxOpen	xxClose	xxRead	xxWrite	xxIoctl
1	yyCreate	yyDelete	yyOpen	yyClose	yyRead	yyWrite	yyIoctl
2	newCreate	newDelete	newOpen	newClose	newRead	newWrite	newIoctl
3							

图 5-4 VxWorks 系统的标准 I/O 函数

一些设备驱动程序能够为某一特殊种类设备的多个实例提供服务。例如：一个串行通信设备的驱动程序经常可以处理多个串行端口的通信工作。在 VxWorks 的 I/O 系统中，给设备定义了一种称为“设备头”(DEV_HDR)的数据结构；该数据结构中包括了设备名称字符串和这个设备所使用的驱动程序的编号(iosDrvInstall()返回的整数)。I/O 系统中所有设备的信息都保存在一个称为“设备表”的链表结构中。该链表中的每个元素都是设备描述符，它包括与该设备有关的数据，如设备地址、缓冲区和信号量等，而设备头则位于设备描述符的起

始部分。

当调用 iosDrvInstall()注册 I/O 函数成功后，必须使用 iosDevAdd()函数将此设备加入 I/O 系统的设备链表中，如图 5-5 所示。

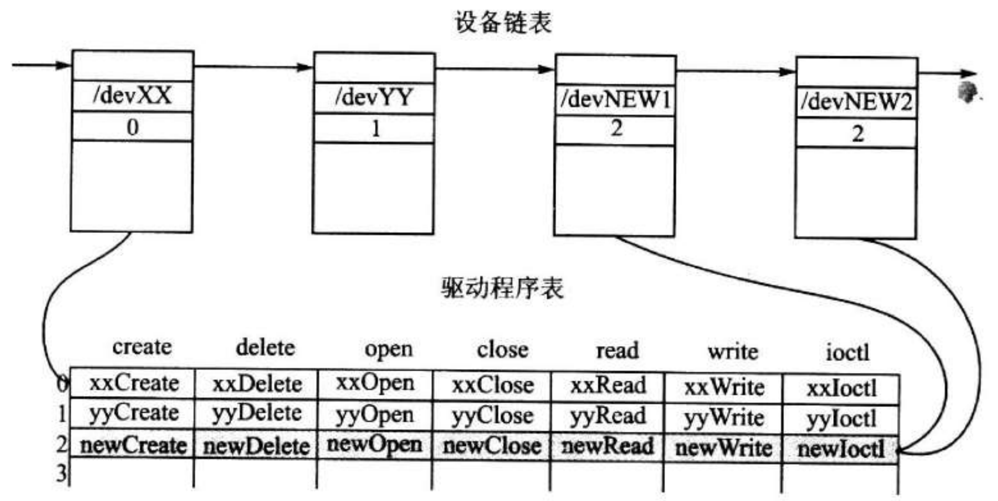


图 5-5 VxWorks 系统驱动程序安装

iosDevAdd()的原型为：

```
STATUS    iosDevAdd (DEV_HDR * pDevHdr, char * name, int drvnum);
```

其中：DEV_HDR 是一个指向该设备的描述符指针；而 DEV_HDR 结构中的第一项则指向下一个设备的描述符指针。

成功执行上述两步操作之后，相应的设备驱动就被安装到 VxWorks 中了。可以使用一些标准 I/O 系统调用(如 open、close、create、remove、read 和 write 等)对该设备进行存取。以 open 为例说明 I/O 系统的实现办法：

① 首先系统根据指定的文件名在设备链表中查找对应的描述符；

② 根据找到的设备描述符获取对应的设备编号；

③ 根据设备编号查找驱动程序表，获取对应的 open 例程的地址；

④ 最后调用驱动程序的 open 例程。

下面具体分析在 JX2410 系统中键盘驱动程序的编写和使用。这是一个非常标准的字符设备，其驱动程序编写也很简单。

JX2410 自带的键盘是一个 4×4 扫描式键盘，本身不使用中断，所以需要使用一个定时器或一个任务定时检查按键情况。在设备安装时无需一些复杂的初始化，这里将初始化程序和设备创建的程序合在一起，即用 s3c2410KbdDevCreate() 创建并安装这个设备。程序清单 5-4是 s3c2410KbdDevCreate() 例程的具体代码和说明。

程序清单 5-4　键盘设备驱动的初始化

```
STATUS s3c2410KbdDevCreate
    (
    char *   name                  /* 键盘设备的名称 */
    )
    {
    int          kbdDrvNum;        /* 安装驱动时返回的驱动程序序号 */
    DEV_HDR      *pHdr;
    char         *pName;

    /* 检查设备列表中是否存在该驱动,避免重复安装同一设备 */
    pHdr = iosDevFind (name, &pName);
    if ((pHdr != NULL) && (strcmp (name, pHdr -> name) == 0))
        {
        pKbdDevice = (KBD_DEVICE *)pHdr;
        return (OK);
        }

   /* 为键盘设备描述符申请空间 */
    pKbdDevice = (KBD_DEVICE *)malloc (sizeof (KBD_DEVICE));
    pKbdDevice -> kbdFlags = 0;

    /* 安装键盘设备 */
    kbdDrvNum = iosDrvInstall (kbdDrvOpen, (FUNCPTR) NULL, kbdDrvOpen,
                    (FUNCPTR) NULL, tyRead, tyWrite, kbdDrvIoctl);

    /* 初始化关联的 tty 设备 */
    if (tyDevInit (&pKbdDevice -> tyDev, 20, 10, kbdWriteData) != OK)
        {
        return (ERROR);
        }

    if (kbdDrvNum == ERROR)
        {
        return (ERROR);
        }
```

```
    /* 将设备驱动添加到 I/O 系统中 */
    return (iosDevAdd (&pKbdDevice -> tyDev.devHdr, name, kbdDrvNum));
    }

LOCAL int kbdWriteData
    (
    KBD_DEVICE *    pKbdDv    /* 键盘设备指针 */
    )
    {
    return (0);
    }
```

在上述代码中，安装字符设备时一般需要以下 7 个例程的入口地址：create、delete、open、close、read、write 和 ioctl。这里省略了 delete 和 close 例程，因为一旦设备打开，则无须关闭（记住：VxWorks 的一些设备句柄是全局的，对于这些设备不必频繁地打开或关闭）。同时这个键盘没有反馈指示灯，因此 write 例程也可取消。对于读/写例程，为了很好地支持 VxWorks 的 select 机制，可以借助于 tyLib 中的两个标准调用 tyRead()和 tyWrite()来完成。它们是系统提供的基于 tty 设备的读/写函数，其函数原型参见程序清单 5-5。

程序清单 5-5　tyRead 和 tyWrite 的函数原型

```
int tyRead
    (
    TY_DEV_ID pTyDev,           /* 设备 ID */
    char *      buffer,         /* 数据缓冲 */
    int         maxbytes        /* 读取的最大长度 */
    )

int tyWrite
    (
    TY_DEV_ID pTyDev,           /* 设备 ID */
    char *      buffer,         /* 数据缓冲 */
    int         nbytes          /* 缓冲中数据的字节数 */
    )

STATUS tyDevInit
    (
    TY_DEV_ID pTyDev,           /* 设备 ID */
```

```
    int         rdBufSize,          /* 用于读取的数据缓冲的大小 */
    int         wrtBufSize,         /* 用于写入的数据缓冲的大小 */
    FUNCPTR     txStartup           /* 数据发送启动例程 */
    )
```

tyLib是系统为串行设备提供的一些支持例程，这里主要使用了其两方面功能：环形缓冲和select机制。使用tyLib的环形缓冲，使得在键盘驱动中无须管理键盘缓冲区，只须使用tyRead将键盘数据导入环形缓冲；使用select机制使得应用程序可以非阻塞地检测键盘输入情况。

2. 读/写例程

由于使用系统提供的读/写例程，因此在本设备驱动的读/写例程中只须做个桥接。如程序清单5-6所列。

程序清单5-6　键盘设备驱动的读/写

```
LOCAL int kbdDrvOpen
    (
    KBD_DEVICE *    pKbdDv,         /* 设备控制符的指针 */
    char *          name,           /* 设备的名称 */
    int             mode            /* 打开的模式 */
    )
    {
    /* 打开设备时须返回一个整型的数值，该数值往往就是设备控制符的指针 */
    return ((int) pKbdDv);
    }

LOCAL STATUS kbdDrvIoctl
    (
    KBD_DEVICE *    pKbdDv,         /* 设备控制符的指针 */
    int             request,        /* 请求代码 */
    int             arg             /* 参数 */
    )
    {
    int     status = OK;

    switch (request)
        {
        case CONIOCURCONSOLE:
```

```
            break;

        case CONIOCONVERTSCAN:
            break;
        case CONIOLEDS:
            break;
        default:
            /* 将一些请求转换为 tyLib 的请求: tyIoctl */
            status = tyIoctl (&pKbdDv -> tyDev, request, arg);
            break;
        }
    return (status);
    }
```

3. 键盘值的读取

键盘的检测一般都须进行抖动消除,可采用多次扫描,通过检查结果是否一致来判断按键抖动。程序清单 5－7 中,示例代码对于键盘的某一列,使用了 4 个循环周期：第 1 个周期将其输入置为低电平;第 2 个周期获取扫描值;第 3 个周期进行抖动消除;第 4 个周期保存键值。扫描键盘的程序编写方法很多,可以自行改进。

程序清单 5－7　键盘的扫描和键值翻译

```
int  timer1_count = 0;
unsigned char output_0x10000000 = 0xff;
enum KEYBOARD_SCAN_STATUS
{
    KEYBOARD_SCAN_FIRST,
    KEYBOARD_SCAN_SECOND,
    KEYBOARD_SCAN_THIRD,
    KEYBOARD_SCAN_FOURTH
};

int row = 0;
unsigned char     ascii_key, input_key[4], input_key1[4], key_mask = 0x0f;
unsigned char *   keyboard_port_scan = (unsigned char * )0x10000000;
unsigned char *   keyboard_port_value = (unsigned char * )0x10000002;
int               keyboard_scan_status[4] =
```

```
{
    KEYBOARD_SCAN_FIRST,
    KEYBOARD_SCAN_FIRST,
    KEYBOARD_SCAN_FIRST,
    KEYBOARD_SCAN_FIRST
};

LOCAL char keyTrans
    (
    int row,
    int col
    )
    {
    char key = 0;

    switch( row )
    {
    case 0:
        if((col & 0x01) == 0) key = '0';
        else if((col & 0x02) == 0) key = 'A';
        else if((col & 0x04) == 0) key = 'B';
        else if((col & 0x08) == 0) key = 'F';
        break;
    case 1:
        if((col & 0x01) == 0) key = '7';
        else if((col & 0x02) == 0) key = '8';
        else if((col & 0x04) == 0) key = '9';
        else if((col & 0x08) == 0) key = 'E';
        break;
    case 2:
        if((col & 0x01) == 0) key = '4';
        else if((col & 0x02) == 0) key = '5';
        else if((col & 0x04) == 0) key = '6';
        else if((col & 0x08) == 0) key = 'D';
        break;
    case 3:
        if((col & 0x01) == 0) key = '1';
```

```
        else if((col & 0x02) == 0) key = '2';
        else if((col & 0x04) == 0) key = '3';
        else if((col & 0x08) == 0) key = 'C';
        break;
    default:
        break;
    }
    return key;
}
LOCAL void recvKey
    (
    int key
    )
    {
        tyIRd (&(pKbdDevice -> tyDev), key);
    }
LOCAL void kbdScan(void)
{
    int loopcnt = row, bexit = 0;
    int temp;

    /* 键盘扫描 */
    for( loopcnt = row; loopcnt < row + 4; loopcnt ++ )
    {
        if(loopcnt >= 4)
            temp = loopcnt - 4;
        else
            temp = loopcnt;
        switch(keyboard_scan_status[temp])
        {
            case KEYBOARD_SCAN_FIRST:
                /* 将 row 列置为低电平 */
                * keyboard_port_scan = output_0x10000000 & (~(0x00000001 << temp));
                keyboard_scan_status[temp] = KEYBOARD_SCAN_SECOND;
                bexit = 1;
                break;
```

```
        case KEYBOARD_SCAN_SECOND:
            /* 获取第一次扫描值 */
            input_key[temp] = ( * keyboard_port_value) & key_mask;
            if(input_key[temp] == key_mask)
            {
                /* 若无按键,则回到开始状态 */
                keyboard_scan_status[temp] = KEYBOARD_SCAN_FIRST;
            }
            else
            {
                /* 有按键 */
                keyboard_scan_status[temp] = KEYBOARD_SCAN_THIRD;
                bexit = 1;
            }
            break;
        case KEYBOARD_SCAN_THIRD:
            if ((( * keyboard_port_value) & key_mask) != input_key[temp])
            {
                keyboard_scan_status[temp] = KEYBOARD_SCAN_FIRST;
            }
            else
            {
                ascii_key = keyTrans (temp, input_key[temp]);
                keyboard_scan_status[temp] = KEYBOARD_SCAN_FOURTH;

                /* 将 row 列置为低电平 */
                * keyboard_port_scan = output_0x10000000& (~(0x1 << temp));
                bexit = 1;
            }
            break;
        case KEYBOARD_SCAN_FOURTH:
            /* 获取第一次扫描值 */
            input_key1[temp] = ( * keyboard_port_value) & key_mask;
            if(input_key1[temp] == key_mask)
            {
                /* 将按键保存在环形缓冲中 */
                recvKey(ascii_key);
```

```
                keyboard_scan_status[temp] = KEYBOARD_SCAN_FIRST;
            }else
            {
                /* 将 row 列置为低电平 */
                *keyboard_port_scan = output_0x10000000 & (~(0x1 << temp));
                bexit = 1;
            }
            break;
        }
        if(bexit) break;
    }

    row = temp;
}
void kbdPoll
    (
    int x
    )
    {
        while(1)
        {
            kbdScan();
            OSTaskDelay(10);
        }
    }
```

5.3　块设备驱动

5.3.1　块设备驱动程序

块设备是指以“块”为单位对数据进行存取的设备。它可以被随机存取，并且数据以块为单位进行传输，典型的有硬盘、光驱、软驱和磁带等。VxWorks 的块设备与字符设备有微小的差别：块设备不能与 I/O 系统直接打交道，在块设备驱动程序与 I/O 系统之间必须有文件系统，如图 5－6 所示。块设备包括 dosFs 、rawFs 和 tapeFs 等。这种层次关系允许同一个块设备上存在不同的文件系统，从而减小驱动程序中必须支持的 I/O 函数的数量。

块设备驱动必须创建一个逻辑盘或连续设备。所谓的"逻辑盘设备"可以仅为一个大的物理设备的一部分。设备驱动必须跟踪数据块的偏移,且须有将逻辑设备与物理设备的实际位置进行转换的方法。VxWorks 文件系统的块序号通常以 0 开始;一个可连续存取的设备块大小通常是可变的,大多数应用程序使用的都是可变的块大小。设备的创建函数一般来说都会分配一个设备描述符结构,用于驱动对设备的控制。该结构的第一个成员必须是 VxWorks 定义的块设备结构(BLK_DEV 或 SEQ_DEV),因为文件系统在调用设备驱动时需要使用这个块设备结构。BLK_DEV()或 SEQ_DEV()结构中应定义一些与该设备相关的变量,如块大小和块数目;还应定义一个函数列表,里面包括实现本设备读/写、控制、复位以及状态检查等例程。在 BLK_DEV 中通常还会定义一些成员来表示设备在文件系统上的特定条件和状态,如磁盘改变等。具体定义见程序清单 5-8 和 5-9。

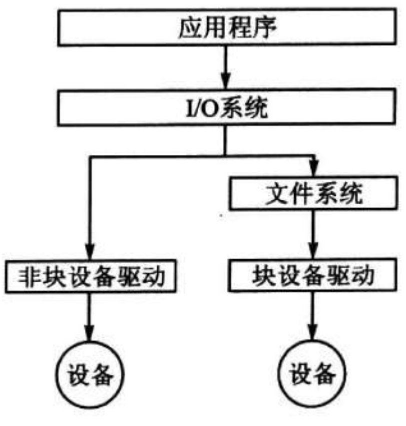


图 5-6　I/O 系统和块设备驱动

程序清单 5-8　BLK_DEV 定义

```
typedef struct          /* BLK_DEV */
    {
    FUNCPTR      bd_blkRd;              /* 数据读取的函数指针 */
    FUNCPTR      bd_blkWrt;             /* 数据写入的函数指针 */
    FUNCPTR      bd_ioctl;              /* ioctl 的函数指针 */
    FUNCPTR      bd_reset;              /* 设备复位的函数指针 */
    FUNCPTR      bd_statusChk;          /* 设备状态检查的函数指针 */
    BOOL         bd_removable;          /* 介质存在与否的标志 */
    ULONG        bd_nBlocks;            /* 设备的块数 */
    ULONG        bd_bytesPerBlk;        /* 每块的字节数 */
    ULONG        bd_blksPerTrack;       /* 每磁道的块数 */
    ULONG        bd_nHeads;             /* 设备的头数 */
    int          bd_retry;              /* I/O 存取失败时的重试次数 */
    int          bd_mode;               /* 设备的模式 */
    BOOL         bd_readyChanged;       /* 设备就绪状态改变的标志 */
    } BLK_DEV;
```

程序清单 5-9　SEQ_DEV 定义

```
typedef struct          /* SEQ_DEV */
    {
```

```
    FUNCPTR      sd_seqRd;               /* 数据读取的函数指针 */
    FUNCPTR      sd_seqWrt;              /* 数据写入的函数指针 */
    FUNCPTR      sd_ioctl;               /* ioctl 的函数指针 */
    FUNCPTR      sd_seqWrtFileMarks;     /* 写文件标记的函数指针 */
    FUNCPTR      sd_rewind;              /* 设备重新定位的函数指针 */
    FUNCPTR      sd_reserve;             /* 设备倒退的函数指针 */
    FUNCPTR      sd_release;             /* 卸载设备的函数指针 */
    FUNCPTR      sd_readBlkLim;          /* 块读取的函数指针 */
    FUNCPTR      sd_load;                /* 加载设备的函数指针 */
    FUNCPTR      sd_space;               /* 在介质上留空操作的函数指针 */
    FUNCPTR      sd_erase;               /* 设备擦除操作的函数指针 */
    FUNCPTR      sd_reset;               /* 设备复位的函数指针 */
    FUNCPTR      sd_statusChk;           /* 设备状态检查的函数指针 */
    int          sd_blkSize;             /* 设备每块的大小 */
    int          sd_mode;                /* 设备的模式 */
    BOOL         sd_readyChanged;        /* 介质就绪状态改变标志 */
    int          sd_maxVarBlockLimit;    /* 每块大小的最大值 */
    int          sd_density;             /* 设备的存储密度 */
    } SEQ_DEV;
```

当驱动程序创建块设备时，设备没有名字，无法与特定的文件系统关联；只有在设备初始化函数中选择文件系统后，才会使设备与相应的文件系统关联起来，例如使用 dosFsDevinit() 或 tapeFsDevInit()等。应用程序与设备的所有通信都是先通过 I/O 系统来访问文件系统，然后由文件系统和设备驱动程序交换数据并最终操作具体的设备，所以设备驱动程序必须提供一些设备与 VxWorks 之间的接口。基本上这些接口都是与硬件密切相关的，每个设备都有其独特的处理过程，但是提供给 VxWorks 的接口应该是统一的。

与非块存取设备驱动程序不同，每个文件系统都作为一个驱动程序安装在驱动程序表中，而块存取设备的低级设备驱动程序并不安装在这个表中。即使几个不同的低级设备驱动程序所驱动的设备安装在同一个文件系统中，每种文件系统在驱动程序表中也只会有一个入口。设备初始化使用某种文件系统后，针对该设备的所有 I/O 操作都必须通过文件系统；文件系统依次调用 BLK_DEV 和 SEQ_DEV 数据结构中指定的函数对设备执行相应的操作。块设备驱动程序的操作函数大致有以下几部分：

- 低级驱动程序初始化；
- 设备创建；
- 读/写操作；
- I/O 控制；

● 复位及状态检测。

顺序存储设备还包括一些特有的操作，如写文件标志、向后搜索、保留操作和安装/卸载等。顺序存储设备在此不详细介绍，下面以 RAMDISK 驱动程序为例来分析普通的块存储设备驱动程序编写方法。

5.3.2 RAMDISK 驱动程序编写

1. 低级驱动程序初始化

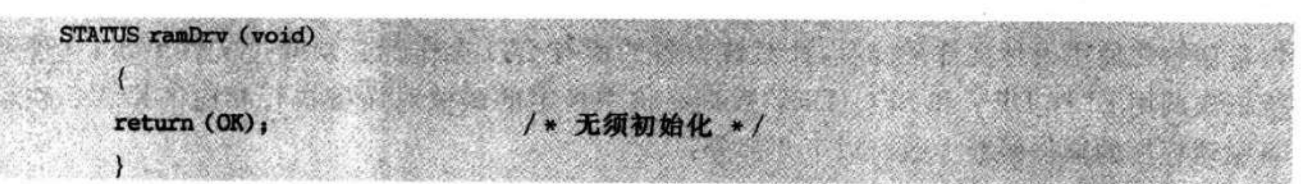
驱动程序需要一个初始化函数，它所执行的操作应该只执行一次。将某个设备的操作归为初始化代码有一个原则：初始化函数操作会影响整个设备控制器，而其他后继操作只影响特殊的设备。块设备的通用初始化函数包括：

● 初始化硬件；
● 分配和初始化设备数据结构；
● 创建互斥量(用于多任务存取)；
● 初始化中断；
● 开设备中断。

与非块存取设备驱动程序不同，块存取设备的初始化函数不是通过调用 iosDrvInstall()函数将驱动程序装入 I/O 系统的驱动程序表中。而是由文件系统将自己作为驱动程序装入驱动程序表中，并且通过使用存放于块存取设备结构 BLK_DEV 或 SEQ_DEV 中的函数地址调用实际的驱动程序。

在程序清单 5－10 所列的代码中，实际上没有执行任何操作，因为虚拟磁盘的物理设备通常是 RAM，此时已处于就绪状态，无须再进行任何初始化工作。如果是其他硬件，则还须执行前面列出的一些具体操作。

程序清单 5－10　$(WINDBASE)\target\src\usr\ramDrv.c



```
STATUS ramDrv (void)
    {
    return (OK);                        /* 无须初始化 */
    }
```

2. 设备创建

在设备创建函数中，首先要分配一个设备描述符，然后根据具体情况填写该设备描述符。程序清单 5－11 为虚拟磁盘设备创建函数的具体代码。

程序清单 5-11　$(WINDBASE)\target\src\usr\ramDrv.c

```
#define DEFAULT_DISK_SIZE   (min (memFindMax()/2, 51200))
#define DEFAULT_SEC_SIZE    512

typedef struct  /* RAM_DEV——RAMDISK 设备描述符结构 */
    {
    BLK_DEV     ram_blkdev;         /* 通用块设备结构 */
    char        * ram_addr;         /* RAMDISK 存储器地址 */
    int         ram_blkOffset;      /* 块偏移 */
    } RAM_DEV;

BLK_DEV * ramDevCreate
    (
    char        * ramAddr,          /* 存储器地址 */
    FAST int    bytesPerBlk,        /* 每块的字节数 */
    int         blksPerTrack,       /* 每磁道的块数 */
    int         nBlocks,            /* 设备的块数 */
    int         blkOffset           /* 设备块的起始偏移 */
    )
    {
    FAST RAM_DEV      * pRamDev;
    FAST BLK_DEV      * pBlkDev;

    /* 如果没有指定块大小及块数,则使用默认设置 */
    if (bytesPerBlk == 0)
    bytesPerBlk = DEFAULT_SEC_SIZE;
    if (nBlocks == 0)
    nBlocks = DEFAULT_DISK_SIZE / bytesPerBlk;
    if (blksPerTrack == 0)
    blksPerTrack = nBlocks;
    /* 为 RAM_DEV 结构分配内存空间 */
    pRamDev = (RAM_DEV * ) malloc (sizeof (RAM_DEV));
    if (pRamDev == NULL)
    return (NULL);
```

```
    /* 初始化 BLK_DEV 结构成员 */
    pBlkDev = &pRamDev -> ram_blkdev;
    pBlkDev -> bd_nBlocks       = nBlocks;          /* 块数 */
    pBlkDev -> bd_bytesPerBlk   = bytesPerBlk;      /* 每块字节数 */
    pBlkDev -> bd_blksPerTrack  = blksPerTrack;     /* 每磁道块数 */
    pBlkDev -> bd_nHeads        = 1;                /* 1 个头 */
    pBlkDev -> bd_removable     = FALSE;            /* 不可移动的设备 */
    pBlkDev -> bd_retry         = 1;                /* 出错重试次数 */
    pBlkDev -> bd_mode          = O_RDWR;           /* 设备的初始读/写模式 */
    pBlkDev -> bd_readyChanged  = TRUE;             /* 设备可用与否 */
    pBlkDev -> bd_blkRd         = ramBlkRd;         /* 块读操作例程地址 */
    pBlkDev -> bd_blkWrt        = ramBlkWrt;        /* 块写操作例程地址 */
    pBlkDev -> bd_ioctl         = ramIoctl;         /* ioctl 控制例程地址 */
    pBlkDev -> bd_reset         = NULL;
    pBlkDev -> bd_statusChk     = NULL;

    /* 如果参数没有指定内存地址,则在这里分配实际空间 */
    if (ramAddr == NULL)
    {
    pRamDev -> ram_addr = (char * ) malloc ((UINT) (bytesPerBlk * nBlocks));
    if (pRamDev -> ram_addr == NULL)
        {
        free ((char * ) pRamDev);
        return (NULL);
        }
    }
    else
    pRamDev -> ram_addr = ramAddr;
    pRamDev -> ram_blkOffset = blkOffset;
    return (&pRamDev -> ram_blkdev);
}
```

从上面的例程代码中可以看出,块设备的管理与传统的磁盘管理类似,沿用了其很多特征,如磁盘的磁头数、磁道数、每道扇区(块)数和每块字节数等。

3. 读/写操作

块设备的读/写是以块为单位进行的，其读/写例程的定义见程序清单 5 - 12。

程序清单 5 -12 读/写操作的函数定义

```
STATUS xxBlkRd
    (
    DEVICE *        pDev,                   /* 指向块设备描述符的指针 */
    int             startBlk,               /* 开始读的块的起始位置 */
    int             numBlks,                /* 需要读的块的数目 */
    char *          pBuf                    /* 存储接收的数据的缓冲指针 */
    )
STATUS xxBlkWrt
    (
    FAST RAM_DEV    * pRamDev;              /* 指向块设备描述符的指针 */
    int             startBlk;               /* 开始写的块的起始位置 */
    int             numBlks;                /* 需要写的块的数目 */
    char            * pBuffer;              /* 存储待写数据的缓冲指针 */
    )
```

RAMDISK 的读/写例程代码参见程序清单 5 - 13。

程序清单 5 - 13 $ (WINDBASE) \target\src\usr\ramDrv. c

```
LOCAL STATUS ramBlkRd (pRamDev, startBlk, numBlks, pBuffer)
    FAST RAM_DEV    * pRamDev;              /* 设备描述符指针 */
    int             startBlk;               /* 起始块号 */
    int             numBlks;                /* 读取的块数 */
    char            * pBuffer;              /* 数据缓冲区 */
    {
    FAST int        bytesPerBlk;

    bytesPerBlk = pRamDev -> ram_blkdev.bd_bytesPerBlk;

    /* 计算块偏移 */
    startBlk += pRamDev -> ram_blkOffset;
    /* 读取数据 */
    bcopy ((pRamDev -> ram_addr + (startBlk * bytesPerBlk)), pBuffer,
```

```
        bytesPerBlk * numBlks);
    return (OK);
    }

LOCAL STATUS ramBlkWrt (pRamDev, startBlk, numBlks, pBuffer)
    FAST RAM_DEV      * pRamDev;          /* 设备描述符指针 */
    int               startBlk;           /* 起始块号 */
    int               numBlks;            /* 写入的块数 */
    char              * pBuffer;          /* 待写的数据 */
    {
    FAST int          bytesPerBlk;
    bytesPerBlk = pRamDev -> ram_blkdev.bd_bytesPerBlk;

    /* 计算块偏移 */
    startBlk += pRamDev -> ram_blkOffset;

    /* 写数据 */
    bcopy (pBuffer, (pRamDev -> ram_addr + (startBlk * bytesPerBlk)),
        bytesPerBlk * numBlks);
    return (OK);
    }
```

无论是写操作还是读操作中，对于不同的文件系统，开始部分都会有一些保留块，用于文件分配表等。这些保留块由用户和系统共同决定，所以在进行操作时，都需将实际操作的块号与偏移相加形成最终的地址。在上面的代码中，设备的读/写操作非常简单，只须完成寻址和内存的拷贝。对于其他设备（如硬盘），可能需要一些控制命令来完成读/写操作，可参考具体硬件手册来实现。

4. I/O 控制

驱动程序必须提供能处理 I/O 控制请求的函数。在 VxWork 操作系统中，大部分 I/O 操作由文件系统处理，但如果出现文件系统不能识别的请求，则该操作请求就会交给驱动程序的 I/O 控制函数处理。驱动程序的 I/O 控制函数定义参见程序清单 5－14。

程序清单 5－14　Ioctl 的函数定义

```
STATUS xxIoctl
    (
```

```
    RAM_DEV      * pRamDev;            /* 设备描述符指针 */
    int          function;             /* 请求的代码 */
    int          arg;                  /* 相应的参数 */
    )
```

在 RAMDISK 的 I/O 控制函数中也未做任何实际的工作，只是简单地返回错误，如程序清单 5-15 所列。

程序清单 5-15　$(WINDBASE)\target\src\usr\ramDrv.c

```
LOCAL STATUS ramIoctl (pRamDev, function, arg)
    RAM_DEV      * pRamDev;            /* 设备描述符指针 */
    int          function;             /* 请求的代码 */
    int          arg;                  /* 相应的参数 */
    {
    FAST int     status;               /* 返回的状态值 */
    switch (function)
    {
    case FIODISKFORMAT:
        status = OK;
        break;
    default:
        errnoSet (S_ioLib_UNKNOWN_REQUEST);
        status = ERROR;
    }
    return (status);
}
```

5. 复位及状态检测

块设备驱动有单独的函数对设备进行复位，它应该只复位指定的块设备而非所有块设备。一般来说块设备驱动还应有获取设备状态的函数，以检查当前设备的状态，这对可热插拔的设备来说很重要。当设备出错或被拔出时，这个函数需要给出正确的结果。一旦检测到设备错误，文件系统就不会继续往下操作；而当插入一个新设备时，它应设置 BLK_DEV 中的 bd_readyChanged 项，此时 open 及 create 函数可以继续操作。新的设备可以自动加入系统中。在 RAMDISK 中没有实现这些操作，所以可以说这部分功能只是可选功能。其函数定义参见程序清单 5-16。

程序清单 5 -16 复位及状态检测

```
STATUS xxReset
    (
    RAM_DEV     * pRamDev;          /* 设备描述符指针 */
    )
STATUS xxStatusCheck
    (
    RAM_DEV     * pRamDev;          /* 设备描述符指针 */
    )
```

5.4 串口设备驱动

5.4.1 串口设备驱动程序

VxWorks 的串行设备不同于前面介绍的字符设备和块设备。一般的设备都是在系统最初启动时调用 xxDrv()来安装设备驱动；然后调用 xxDevCreate()将该设备描述符 xx_DEV 加入设备驱动中。在应用层使用设备时，直接通过文件描述符在设备驱动列表中查出对应的设备驱动；然后由该设备驱动提供的读/写例程进行操作。其层次关系：应用↔I/O 文件系统↔驱动程序↔硬件，非常明确。然而串行设备的层次关系与其有明显的区别。基于许多因素的考虑，VxWorks 串行设备驱动的结构如图 5 - 7 所示。

从图 5 - 7 中可以看出，系统中的串行设备驱动共分 3 层：usrConfig. c 和 ttyDrv(包括 tyLib)提供了一些对串行设备的通用操作；而 sysSerial. c 则针对具体目标系统的、串行设备相关的一些数据结构进行初始化操作；最后 xxDrv. c(在提供的 BSP 中文件名为 s3c2410Sio. c)包括了一些具体设备相关的操作(如读/写数据和设置等)。

一个包含了串行设备及其应用程序的总体模型如图 5 - 8 所示。

从图 5 - 8 中可以看出，串行设备的驱动 xxDrv 并不是直接与 I/O 系统交互的，中间存在一个 ttyDrv(包括 tyLib)。实际上，与 I/O 系统交互的不是 xxDrv 的函数而是 ttyDrv/tyLib 提供的函数。另外，Target Agent 可以和 xxDrv 交互，方便了系统的调试。

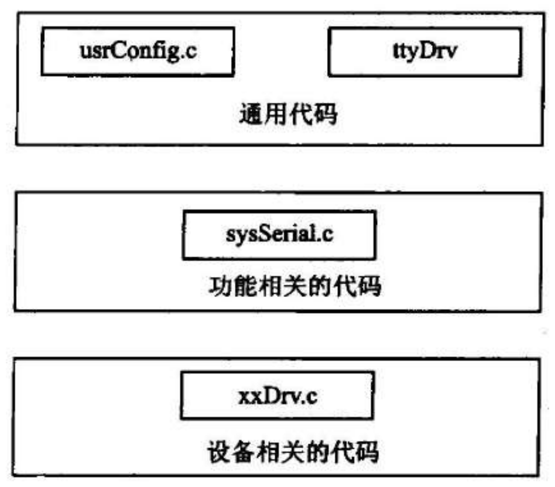


图 5-7 串行设备驱动代码的分层结构

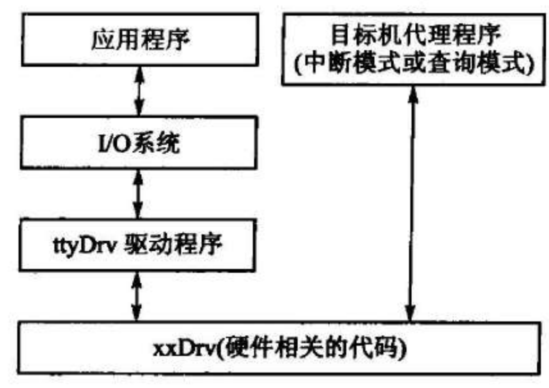


图 5-8 串行设备驱动的模型

ttyDrv(包括 tyLib)：可以称为一个虚拟的设备驱动。它只是介于 I/O 与底层具体设备驱动之间，为系统提供统一的串行设备界面。另外，还可调用具体硬件的管理驱动。总之，ttyDrv 给系统提供了一些通用的管理函数(如缓冲管理和互斥等)。所以，ttyDrv 往往可以管理多个设备。图 5-9 表明了 ttyDrv 在系统中的作用。

xxDrv：给 ttyDrv 和 target agent 提供支持。

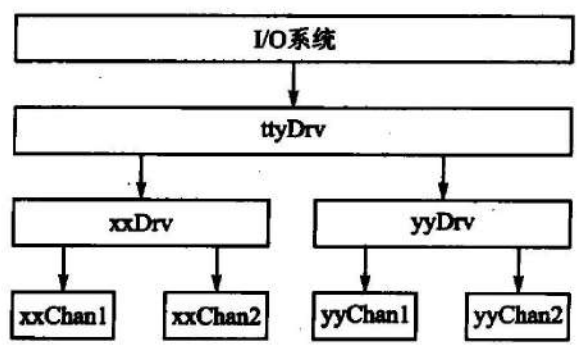


图 5-9 ttyDrv 在 I/O 系统中的作用

5.4.2 ttyDrv 的层次结构

ttyDrv(tyLib)是一个虚拟驱动，既管理与 I/O 的交互，又管理与底层硬件驱动的交互。它在与 I/O 交互时所做的工作主要有：

- I/O 系统请求初始化，例如添加设备入口，创建设备描述符；
- 处理所有 I/O 请求，如 ttyOpen、ttyIoctl、tyRead 和 tyWrite 等；
- 管理数据缓冲区；
- 管理多任务的同步和互斥。

其中：ttyDrv 负责 ttyOpen 和 ttyIoctl 的接入；而 tyLib 则负责 tyRead 和 tyWrite 的接入。图 5-10 为一个更加详细的、系统的数据流向图。

从图 5-10 中可以清楚地看出系统的层次关系。应用层往往调用通用的 ioLib 中的函数

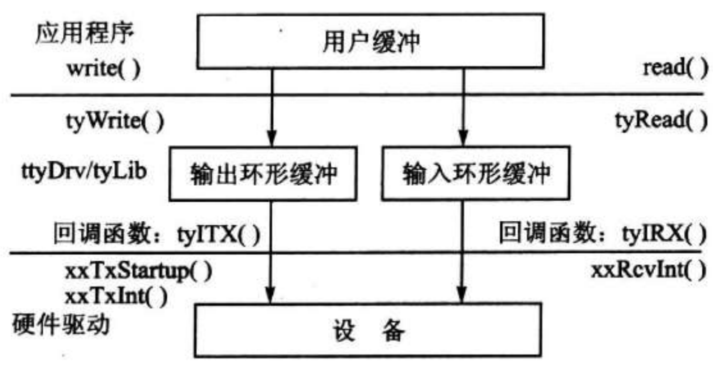


图 5-10 ttyDrv 的数据流图

read、write 和 ioctl 等。这些函数从输入的文件描述符中找到相应的设备描述符；然后找到设备驱动表，并调用设备驱动表中的处理函数。而设备驱动表中的函数 ttyDrv/tyLib(tyWrite 和 tyRead 等)则是第二层的内容。ttyDrv/tyLib 与实际的设备(xxDrv)打交道完成指定的操作。第二层和底层的通信是通过回调函数(callback)来实现的。具体来说，ttyDrv 提供两个函数(tyWrite 和 tyRead)：输出时底层驱动可以从缓冲区中读取数据；输入时底层驱动可将接收到的字符填入缓冲区。tyLib 提供的回调函数格式如下：

```
STATUS tyIRd
    (
    TY_DEV_ID pTyDev,           /* 设备描述符指针 */
    char      inchar            /* 输入的缓存指针 */
    )
STATUS tyITx
    (
    TY_DEV_ID pTyDev,           /* 设备描述符指针 */
    char *    pChar             /* 从缓存中得到的字符 */
    )
```

下面对驱动中涉及的各函数进行简要说明：

(1) ttyDrv( )

ttyDrv()调用 iosDrvInstall()将 ttyDrv 和 tyLib 中的函数安装到设备驱动表中。此函数在系统启动时由 usrRoot()调用。

(2) ttyDevCreate( )

ttyDevCreate()主要完成如下工作：

- 创建和初始化设备描述符；
- 通过调用 tyDevInit( )初始化 tyLib；
- 通过调用 iosDevAdd( )将设备添加到设备列表中；

● 给底层的设备安装回调函数；

● 启动设备的中断。

此函数也是在系统启动时由 usrRoot()调用。

(3) write()

write()函数的调用过程是：write()→tyWrite()→xxDrv。tyWrite()对所有的串行设备做同样的事情：

● 如果环形缓冲满，则阻塞；

● 从用户的缓冲中读数据到环形缓冲中；

● 在一定情况下激活 xxDrv 开始传输数据周期(一个传输周期中首先传送数据到设备，然后等待设备给出一个中断，以继续传送下一个数据)；

● xxDrv 通过调用 tyITx()来从输出环形缓冲中读数据。

(4) read()

read()函数的调用过程是：read()→tyRead()→xxDrv。tyRead()对所有的串行设备执行以下操作：

● 如果环形缓冲空，则阻塞；

● 从环形缓冲读数据到用户的缓冲中；

● 处理 X-on/X-off；

● 如果输入环形缓冲中还有字符，则启动所有阻塞的进程；

● xxDrv 调用 tyIRx()来将设备读到的字符写入输入环形缓冲中。

(5) Ioctl()

Ioctl()将函数传递到 xxIoctl()，如果设备驱动没有实现此功能，则将控制转移到tyIoctl。其关系如图 5-11 所示。

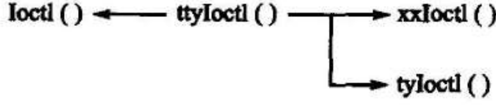


图 5-11 ioctl 控制的流向

总之，ttyDrv 和 tyLib 之所以独立为一层，主要是从代码可复用和统一界面两个角度来考虑的。代码可重用，是因为所有串行设备都有一些相同的管理工作(如缓存和信号的管理等)；同样为了给 I/O 提供统一的界面，ttyDrv 在设备驱动表中安装了统一的驱动函数，这样 I/O 和串行设备的交互就被屏蔽了。

5.4.3 S3C2410 串口驱动的编写

编写串行设备的驱动主要有以下几方面工作：

● 初始化：确定系统要支持的串行通道的个数，初始化设备的描述符，编写设备的初始化代码；

- 编写入口点函数；
- 编写设备中断服务程序(ISR)；
- 修改 sysSerial. c 文件。

1. 初始化过程

要编写设备的初始化代码，首先看一下系统初始化的大致顺序，如图 5－12 所示。

```
usrInit ()
 └→ sysHwInit () → sysSerialHwInit () → xxDevInit ()
usrRoot ()
 ├→ sysClkConnect () → sysHwInit2 () → sysSerialHwinit2 ()
 ├→ ttyDrv ()
 ├→ sysSerialChanGet ()
 └→ ttyDevCreate ()
```

图 5－12 系统初始化串行设备顺序

从图 5－12 中可以看出，串口设备的初始化涉及 2 个阶段：

(1) 在 usrInit()中进行设备的初始化

sysHwInit()→sysSerialHwInit()：进行设备资源的设置，填写 S3C2410_CHAN 结构，初始化硬件，并关闭串口的中断。

(2) 在 usrRoot 中执行进一步操作

① usrRoot()→sysSerialHwInit2()：开启中断。

② usrIosCoreInit()→ttyDrv()：进行系统调用，初始化 ttyLib；

③ usrIosCoreInit()→usrSerialInit()→ttyDevCreate()：创建串口设备，其中创建时 ttyDevCreate()的第二个参数 SIO_CHAN 需要通过 sysSerialChanGet()获得。

在文件组织上，可将 sysSerialxx()一类的函数放在 sysSerial. c 中，将剩下的其他函数放在另一个单独的文件中。

编写驱动程序，首先要根据具体的串行设备定义设备描述符的结构(xx_DEV)。该结构是注册在设备列表中的，它与描述设备的设备头(DRV_HEAD)以及设备相关的成员构成一个“设备”。对于串行设备，设备相关的成员主要是 xx_CHAN 结构，用于描述串行设备的通道信息。程序清单 5－17 为 xx_CHAN 结构的例子。

程序清单 5－17 串行通道结构描述

```
typedef struct sio_chan                    /* 串行设备 */
  {
    SIO_DRV_FUNCS * pDrvFuncs;             /* 设备驱动的一些函数指针 */
    /* 接下来可以是一些自定义的数据 */
  } SIO_CHAN;
```

```
struct sio_drv_funcs                          /* 设备驱动的函数列表,包括 5 个基本函数 */
  {
    int   (*ioctl)(SIO_CHAN *pSioChan, int cmd, void *arg);
    int   (*txStartup)(SIO_CHAN * pSioChan);
    int   (*callbackInstall)(SIO_CHAN *SioChan,int callbackType,
               STATUS(*callback)(void *, ...),void *callbackArg);
    int   (*pollInput)(SIO_CHAN *pSioChan, char * inChar);
    int   (*pollOutput)(SIO_CHAN *pSioChan,char outChar);
  };
typedef struct S3C2410_CHAN
  {
    SIO_CHAN     sio;                         /* 标准 SIO_CHAN 成员,该结构中它必须是第一个成员 */

    /* 下面是一些回调函数 */
    STATUS      (*getTxChar) ();              /* 发送处理的回调函数 */
    STATUS      (*putRcvChar) ();             /* 接收处理的回调函数 */
    void *      getTxArg;                     /* 发送回调函数的参数 */
    void *      putRcvArg;                    /* 接收回调函数的参数 */
    UINT32      *regs;                        /* 寄存器 */
    UINT8       levelRx;                      /* 接收中断号 */
    UINT8       levelTx;                      /* 发送中断号 */
    UINT8       intRxSubMask;                 /* 接收中断屏蔽码 */
    UINT8       intTxSubMask;                 /* 发送中断屏蔽码 */
    UINT32      channelMode;                  /* 模式 */
    int         baudRate;                     /* 波特率 */
  } S3C2410_CHAN;
```

程序清单 5-18 定义了两个串口设备,首先需要定义串口资源(如地址和中断)。

程序清单 5-18　JX2410 串口配置

```
typedef struct
   {
   UINT        vector;
   UINT32 *    baseAdrs;
   UINT        intLevel;
   UINT        intRxMask;
   UINT        intTxMask;
```

```
    } SYS_S3C2410_CHAN_PARAS;

LOCAL SYS_S3C2410_CHAN_PARAS devParas[] =
    {
      {   INT_VEC_UART_0, (UINT32 *)UART_0_BASE_ADR, INT_LVL_UART_0,
          INT_RX_MASK_UART_0, INT_TX_MASK_UART_0},

      {   INT_VEC_UART_1, (UINT32 *)UART_1_BASE_ADR, INT_LVL_UART_1,
          INT_RX_MASK_UART_1, INT_TX_MASK_UART_1}
    };

LOCAL S3C2410_CHAN s3c2410Chan[2];
SIO_CHAN * sysSioChans [] =
    {
    &s3c2410Chan [0].sio, /* /tyCo/0 */
    &s3c2410Chan [1].sio, /* /tyCo/1 */
    };
```

s3c2410Chan 数据结构的填写主要由 sysSerialHwInit()和 s3c2410SioDevInit()两个函数完成。其中：sysSerialHwInit()完成资源定义；而 s3c2410SioDevInit()则负责处理函数的指定。其完整代码参见程序清单 5－19。

程序清单 5－19 JX2410 串口设备初始化

```
void sysSerialHwInit (void)
    {
    int i;
    for (i = 0; i<2; i++)
    {
    /* 首先在数据结构中填写串口的资源信息 */
    s3c2410Chan [i].regs = devParas[i].baseAdrs;
    s3c2410Chan [i].baudRate = CONSOLE_BAUD_RATE;
    s3c2410Chan [i].xtal = SYS_TIMER_CLK;           /*  = PCLK */
    s3c2410Chan [i].levelRx = devParas[i].intLevel;
    s3c2410Chan [i].levelTx = devParas[i].intLevel;
    s3c2410Chan [i].intRxSubMask = devParas[i].intRxMask;
    s3c2410Chan [i].intTxSubMask = devParas[i].intTxMask;
```

```
    /* 最后调用驱动程序的初始化例程,进一步设置读/写例程等 */
    s3c2410SioDevInit(&s3c2410Chan[i]);
    }
    }

void sysSerialHwInit2 (void)
    {
    int i;

    for (i = 0; i<2; i++)
    {
    /* 只须安装并开启串口的中断 */
    (void) intConnect (INUM_TO_IVEC(devParas[i].vector),
                s3c2410SioInt, (int) & s3c2410Chan[i]);
    intEnable (devParas[i].intLevel);
    }
    }

SIO_CHAN * sysSerialChanGet
    (
    int channel            /* 串口号 */
    )
    {
    if (channel < 0 || channel >= (int)(NELEMENTS(sysSioChans)))
    return (SIO_CHAN *)ERROR;

    /* 根据串口号返回描述结构的指针 */
    return sysSioChans[channel];
    }

void sysSerialReset (void)
    {
    int i;
    for (i = 0; i < N_AMBA_UART_CHANNELS; i++)
    {
    intDisable (devParas[i].intLevel);
    }
    }
```

xxDevInit()函数是系统启动过程中首先调用的一个底层函数，被 sysSerial.c 中的 sysSerialHwInit()函数调用；用于在对目标系统的串行设备描述符初始化之后，将操作函数安装到 SIO_DRV_FUNCS 结构中。具体代码参见程序清单 5－20。

程序清单 5－20　JX2410 串口设备操作函数的安装

```
LOCAL SIO_DRV_FUNCS s3c2410SioDrvFuncs =
    {
    (int (*)())s3c2410Ioctl,
    s3c2410TxStartup,
    s3c2410CallbackInstall,
    s3c2410PollInput,
    s3c2410PollOutput
    };

void s3c2410SioDevInit
    (
    S3C2410_CHAN *    pChan                          /* 串口描述结构指针 */
    )
    {
    int oldlevel = intLock();

    /* 设置驱动程序的操作例程 */
    pChan -> sio.pDrvFuncs = &s3c2410SioDrvFuncs;
    pChan -> getTxChar    = s3c2410DummyCallback;    /* 临时回调函数 */
    pChan -> putRcvChar   = s3c2410DummyCallback;    /* 临时回调函数 */
    pChan -> channelMode = SIO_MODE_POLL;

    /* 硬件初始化 */
    s3c2410InitChannel (pChan);
    intUnlock (oldlevel);
    }

LOCAL void s3c2410InitChannel
    (
    S3C2410_CHAN *    pChan                          /* 串口描述结构指针 */
    )
    {
```

```
int         i;
UINT32      discard;

/* S3C2410 串口寄存器初始化 */
S3C2410_UART_REG_WRITE(pChan, UART_FIFO_CON,     0x00);
S3C2410_UART_REG_WRITE(pChan, UART_MODEM_CON,    0x00);
S3C2410_UART_REG_WRITE(pChan, UART_LINE_CON,     0x03);
S3C2410_UART_REG_WRITE(pChan, UART_CON,          0x245);
S3C2410_UART_REG_WRITE(pChan, UART_BAUD_DIV,(pChan -> xtal / (16 * pChan -> baudRate) - 1) );

/* 清除缓冲 */
for(i = 0;i < 16;i++)
    S3C2410_UART_REG_READ(pChan, UART_RX_DATA, discard);
s3c2410Ioctl ((SIO_CHAN *)pChan, SIO_MODE_SET, (int)pChan -> channelMode);
}
```

2. 编写处理函数

在建立了数据结构的基础上，需要编写一些具体的通信函数。这些函数包括 SIO_DRV_FUNCS 结构中的函数、输入/输出 ISR 和一个辅助函数 xxDevInt()。SIO_DRV_FUNCS 中的主要函数如表 5-8 所列。

表 5-8 串口通信的处理函数

函数名称	描 述	函数名称	描 述
xxCallBackInstall	安装高层协议的处理例程	xxTxStartup	启动一个传输过程
xxPollOutput	轮询方式输出例程	xxTxInt	发送中断处理例程
xxPollInput	轮询方式输入例程	xxRxInt	接收中断处理例程
xxIoctl	设备相关的一些控制码处理例程		

(1) 上层处理函数安装例程 xxCallBackInstall()

回调函数定义见程序清单 5-21。

程序清单 5-21 回调函数定义

```
static int s3c2410CallbackInstall
        (
        SIO_CHAN *   pSioChan,        /* 设备描述符指针 */
```

```
            int  callbackType,              /* 回调函数的类型 SIO_CALLBACK_GET_TX_CHAR */
                                            /* SIO_CALLBACK_PUT_RCV_CHAR */
            STATUS ( * callback)(),         /* 回调函数指针 */
            void *     callbackArg          /* 回调函数的参数列表 */
            )
```

功能：安装上层提供的回调函数（包括中断方式的收/发处理函数），即一旦产生中断，在驱动程序中需要调用相应的回调函数进行进一步处理。设置回调函数主要是为了便于底层驱动程序与 ttyLib 进行通信，例如将接收数据拷贝到环形缓冲中，从环形缓冲中读取发送数据等。具体代码参见程序清单 5-22。

程序清单 5-22　回调函数安装

```
static int s3c2410CallbackInstall
    (
    SIO_CHAN *  pSioChan,               /* 设备结构指针 */
    int         callbackType,           /* 回调函数的类型 */
    STATUS      ( * callback)(),        /* 回调函数的地址 */
    void *      callbackArg             /* 回调函数的参数 */
    )
    {
        S3C2410_CHAN * pChan = (AMBA_CHAN * )pSioChan;
        switch (callbackType)
        {
            case SIO_CALLBACK_GET_TX_CHAR:
                pChan -> getTxChar      = callback;
                pChan -> getTxArg       = callbackArg;
                return (OK);
            case SIO_CALLBACK_PUT_RCV_CHAR:
                pChan -> putRcvChar     = callback;
                pChan -> putRcvArg      = callbackArg;
                return (OK);
            default:
                return (ENOSYS);
        }
    }
```

(2) 设备相关的一些控制码处理例程 xxIoctl()

通常，串行设备的 Ioctl 命令如表 5-9 所列。

表 5-9　串行设备的 Ioctl 命令

命　令	描　述	命　令	描　述
SIO_BAUD_SET	设置波特率	SIO_AVAIL_MODES_GET	获取可选模式
SIO_BAUD_GET	获取当前波特率	SIO_HW_OPTS_SET	设置硬件模式
SIO_MODE_SET	设置中断/轮询模式	SIO_HW_OPTS_GET	获取硬件设置模式
SIO_MODE_GET	获取模式		

它们主要通过 switch 语句来实现对设备的操作。程序清单 5-23 描述了该函数的一个框架。

程序清单 5-23　串行设备 ioctl 操作

```
LOCAL STATUS ambaIoctl (
    SIO_CHAN *   pSioChan,                    /* 通道描述符 */
    int          request,                     /* 请求码 */
    int          arg                          /* 参数 */
    )
{
    int          oldlevel;
    STATUS       status;
    UINT32       brd;
    S3C2410_CHAN * pChan = (S3C2410_CHAN *)pSioChan;
    status = OK;

    switch (request)
    {
    case SIO_BAUD_SET:
        brd = (pChan -> xtal/(16 * arg)) - 1;    /* 计算波特率因子 */
        /* 关闭中断 */
        oldlevel = intLock ();
        /* 设置波特率 */
        S3C2410_UART_REG_WRITE(pChan, UART_BAUD_DIV,   brd );
        pChan -> baudRate = arg;
        break;

    case SIO_BAUD_GET:
```

```
        /* 返回当前串口的波特率 */
        *(int *)arg = pChan -> baudRate;
        break;

    case SIO_MODE_SET:
        /* 设置串口的工作模式:中断或轮询 */
        if ((arg != SIO_MODE_POLL) && (arg != SIO_MODE_INT))
        {
        status = EIO;
        break;
        }
        /* 关闭串口中断 */
        oldlevel = intLock ();
        if (arg == SIO_MODE_INT)
        {
        /* 中断模式:开启指定的中断 */
        S3C2410_UART_REG_WRITE(pChan, UART_FIFO_CON, 0x00);
        intEnable (pChan -> levelRx);
        *(volatile UINT32 *)(S3C2410_INTSUBMSK) &= ~pChan -> intRxSubMask;
        }
        else
        {
        /* 轮询模式:关闭指定的中断 */
        intDisable (pChan -> levelRx);
        *(volatile UINT32 *)((UINT32)(S3C2410_INTSUBMSK)) |=
                (pChan -> intTxSubMask|pChan -> intRxSubMask);
        }

        pChan -> channelMode = arg;

        intUnlock (oldlevel);
        break;

    case SIO_MODE_GET:
        /* 获取当前的工作模式 */
        *(int *)arg = pChan -> channelMode;
```

```
        break;

    case SIO_AVAIL_MODES_GET:
        /* 获取可选的工作模式 */
        *(int *)arg = SIO_MODE_INT | SIO_MODE_POLL;
        break;

    default:
        status = ENOSYS;
    }

    return status;
}
```

(3) 中断传输例程 xxTxStartup、xxTxInt()和 xxRxInt()

这部分包含了 3 个底层驱动函数的编写:启动输出函数 xxTxStartup()、发送中断处理函数 xxTxInt()和接收中断处理函数 xxRxInt()。输出部分的处理流程如图 5-13 所示。

当用户向设备写数据时,就调用 tyWrite。tyWrite 将数据写入输出环形中后,调用 xxTxStartup()启动设备的数据发送。这里只需向串口缓冲区写入一个字节,然后开启发送中断;后续字节将由发送中断处理程序来处理,具体实现见程序清单 5-24。

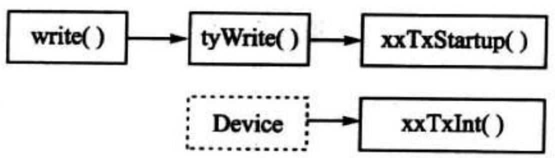


图 5-13　输出部分的处理流程

程序清单 5-24　串行设备数据发送启动

```
LOCAL int s3c2410TxStartup(
    SIO_CHAN *    pSioChan                /* 串口描述符指针 */
)
{
    S3C2410_CHAN * pChan = (S3C2410_CHAN *)pSioChan;

    if (pChan -> channelMode == SIO_MODE_INT)
    {
        s3c2410SioIntTx(pChan);
        *(volatile UINT32 *)((S3C2410_INTSUBMSK)) &= ~pChan -> intTxSubMask;
        return OK;
    }
```

```
    else
    {
        return ENOSYS;
    }
}
```

在中断模式下，缓冲区中的数据发送完毕后，设备就会产生一个中断表示可以接收下一个字符，这时就会调用中断处理程序 xxTxInt()来完成剩下的数据发送工作。输出部分中断处理代码段参见程序清单 5-25。

程序清单 5-25　串行设备数据发送中断处理

```
void s3c2410SioIntTx (
    S3C2410_CHAN *     pChan         /* 串口描述符指针 */
)
{
    char      outChar;
    UINT32     status;

    /* 首先读取串口缓冲区状态 */
    S3C2410_UART_REG_READ(pChan, UART_TXRX_STATUS, status);
    if((status & UART_TX_READY) != UART_TX_READY)    return;

    /* 使用回调函数 getTxChar()从驱动程序的环形缓冲中读取一个字符 */
    if (( * pChan -> getTxChar) (pChan -> getTxArg, &outChar) != ERROR)
    {
        /* 将读取的字符写入串口的发送缓冲区 */
        S3C2410_UART_REG_WRITE(pChan, UART_TX_DATA, outChar);
    }
    else
    {
        * (volatile UINT32 * )( (S3C2410_INTSUBMSK)) |= pChan -> intTxSubMask;
    }
}
```

接收数据部分与发送部分有一点区别。这部分只保留了输入中断处理函数，该中断处理函数在收到数据时，将寄存器中的数据通过回调函数写入驱动程序的环形缓冲区中。输入部分几个函数的调用关系如图 5-14 所示。

这部分的实现代码如程序清单 5-26 所列。

图 5-14　输入部分函数的调用关系

程序清单 5-26　串行设备数据接收中断

```
void s3c2410SioIntRx (
    S3C2410_CHAN *      pChan      /* 串口描述符指针 */
)
{
    char inchar;
    char flags;

    /* 读取状态,检查是否接收到数据 */
    S3C2410_UART_REG_READ(pChan, UART_TXRX_STATUS, flags);
    if ((flags & UART_RX_READY) == UART_RX_READY)
    {
        /* 读取串口缓冲区中的数据 */
        S3C2410_UART_REG_READ(pChan, UART_RX_DATA, inchar);

        /* 使用回调函数 putRcvChar()将读取的数据保存在驱动程序的环形缓冲区中 */
        (*pChan -> putRcvChar) (pChan -> putRcvArg, inchar);
    }
}
```

(4) 轮询方式收/发例程 s3c2410PollOutput()和 s3c2410PollInput()

前面分析了中断模式下串口设备数据的收/发处理过程,但有时也需要使用轮询的方式来处理收/发数据,例如目标机代理。目标板与主机的在线调试可通过串口通信来进行。一般情况下目标与主机之间通信模式有 3 种:

- 中断模式(Interrupt Mode):支持任务级调试;
- 查询模式(Polled Mode):支持系统级调试;
- 双模式(Bi-mode):支持两种模式之间的切换。

要执行系统级的调试,需要实现串口的轮询模式数据收/发;要支持查询模式,必须进一步实现两个设备的驱动函数 xxPollInput 和 xxPollOutput,然后修改 IoCtl()中相应的项。这两个函数的定义见程序清单 5-27。

程序清单 5-27　串行设备查询模式处理函数的原型

```
LOCAL int s3c2410PollOutput
    (
    SIO_CHAN *     pSioChan,         /* 通道描述符指针 */
    char           outChar           /* 输出字符 */
```

```
    )
LOCAL int s3c2410PollInput
    (
    SIO_CHAN *      pSioChan,           /* 通道描述符指针 */
    char *          thisChar            /* 接收缓冲区指针 */
    )
```

查询模式接收处理函数在设备有输入数据时读取数据,并将这些数据直接保存到指定的缓冲区中。具体代码参见程序清单 5-28。

程序清单 5-28　串行设备查询模式数据输入

```
LOCAL int s3c2410PollInput
(
    SIO_CHAN *      pSioChan,           /* 通道描述符指针 */
    char *      thisChar                /* 接收缓冲区指针 */
    )
{
    S3C2410_CHAN * pChan = (S3C2410_CHAN *)pSioChan;
    FAST UINT32 pollStatus;

    /* 检查接收缓冲区的状态 */
    S3C2410_UART_REG_READ(pChan, UART_TXRX_STATUS, pollStatus);
    if ((pollStatus & UART_RX_READY) != UART_RX_READY)
    return EAGAIN;
    /* 如果有数据,则读出数据 */
    S3C2410_UART_REG_READ(pChan, UART_RX_DATA, *thisChar);

    return OK;
}
```

查询模式的输出处理函数如程序清单 5-29 所列。

程序清单 5-29　串行设备查询模式数据输出

```
LOCAL int s3c2410PollOutput
    (
    SIO_CHAN *      pSioChan,           /* 通道描述符指针 */
    char     outChar                    /* 输出字符 */
    )
```

```
{
    S3C2410_CHAN * pChan = (AMBA_CHAN * )pSioChan;
    FAST UINT32 pollStatus;
    S3C2410_UART_REG_READ(pChan, UART_TXRX_STATUS, pollStatus);

    /* 检查输出缓冲区状态 */
    if ((pollStatus & UART_TX_READY) != UART_TX_READY)
    return EAGAIN;

    /* 发送数据 */
    S3C2410_UART_REG_WRITE(pChan, UART_TX_DATA, outChar);

    return OK;
}
```

5.5　网络设备驱动

5.5.1　MUX 网络设备驱动程序

VxWorks 的网络协议栈有个特征，即在数据链路层和网络协议层之间使用公共接口(API)来进行数据交互。这种接口被称为“MUX 网络接口”。在 BSD4.3 模型下，VxWorks 网络驱动程序与协议紧密结合，它们依赖于彼此的数据结构。而在基于 MUX 的模式下，网络驱动程序与协议之间没有内部交换数据，它们只通过 MUX 间接相互作用。例如：接收到数据包后，网络接口驱动程序并不直接访问协议中的任何结构；而当驱动程序准备给协议传递数据时，驱动程序会调用 MUX 辅助程序，然后此程序用于处理给协议传递数据时的具体细节。MUX 接口的作用是分解协议和网络驱动程序，从而使它们几乎独立。这种独立使添加新的驱动程序和协议变得简单。例如：若添加一个新的网络驱动，则所有现有基于 MUX 的协议均可使用新的驱动程序；同样，若添加一个新的基于 MUX 的协议，则任何现有的网络驱动均可通过 MUX 接口来访问新的协议。

图 5-15、图 5-16 和图 5-17 显示了网络协议栈、MUX 和一个网络接口驱动程序之间的 API 接口调用关系。

协议层包含以下接口：

- stackShutdownRtn()

- stackError()
- stackRcvRtn()
- stackTxRestartRtn()

MUX 会执行入口点 muxBind()、muxUnbind()和 muxDevLoad()等。MUX 使用这些入口点与网络接口驱动程序相互作用，如图 5－18 所示。当编写或加载网络接口驱动程序以使用 MUX 时，用户必须实现上面所有的接口点。

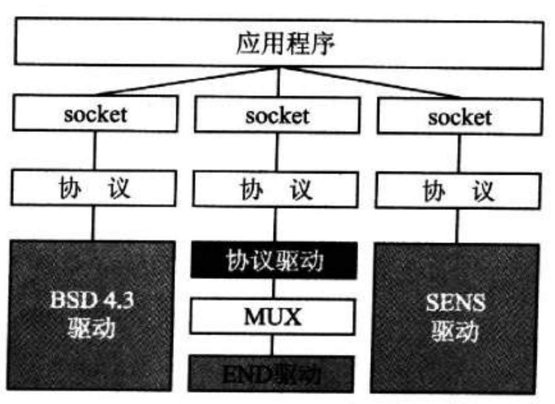


图 5－15 VxWorks 网络协议栈

当启动系统时，VxWorks 执行任务 tUsrRoot 来完成驱动程序的安装，步骤如下：

① 网络任务的工作队列初始化；

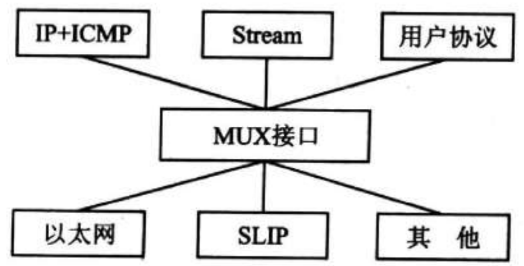


图 5－16 VxWorks 网络协议与 MUX 接口

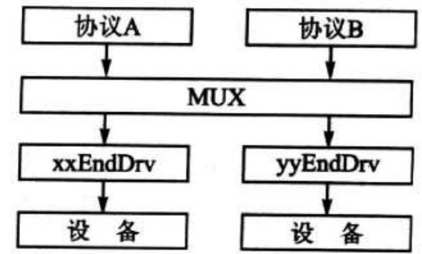


图 5－17 VxWorks MUX 驱动与网络协议

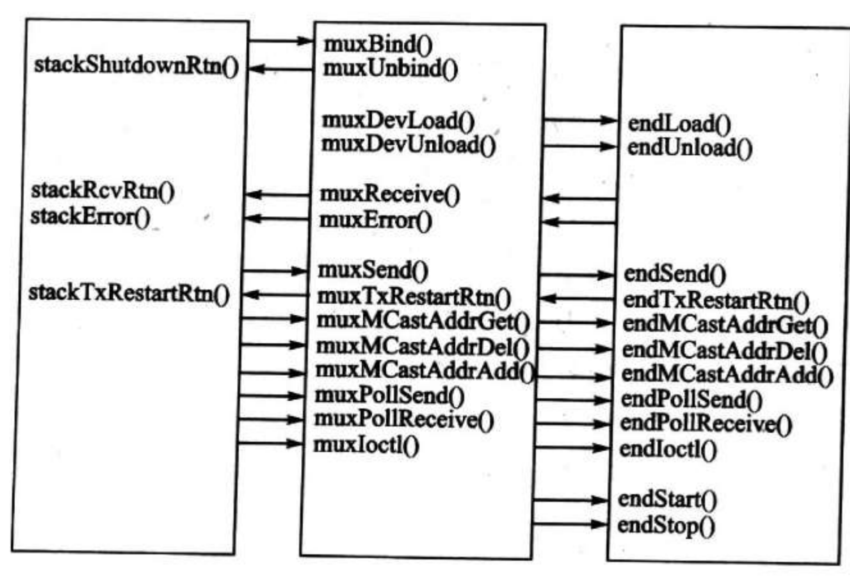


图 5－18 VxWorks MUX 驱动调用关系

② 产生 tNetTask 来处理网络任务工作队列中的条目；

③ 调用 muxDevLoad()来加载网络驱动程序；

④ 调用 muxDevStart()来启动驱动程序。

使用 tUsrRoot 调用 muxDevLoad()加载用户网络驱动程序。作为调用的输入，tUsrRoot 规定了用户驱动程序 endLoad()的入口点，该入口点类似于 xxattach()入口点。endLoad()程序用于处理任何特殊设备的初始化，并返回一个 END_OBJ 结构。用户的网络设备描述结构必须包含该结构，而且还包括一个指向 NET_FUNCS 结构的指针。

从 endLoad()返回控制值到 muxDevLoad()后，MUX 完成了 END_OBJ 结构的填写(通过给程序赋予一指针，用户的驱动程序可以传递数据包到 MUX 中)；然后 MUX 把返回的 END_OBJ 添加到 END_OBJ 结构的链表中，此链表列出了当前系统所运行的所有网络设备的信息。在 muxDevLoad()返回控制信息后，用户驱动程序装载完毕以备用。

当网络设备产生中断时，VxWorks 调用用户驱动程序先前注册的中断服务程序；中断服务程序应做尽可能少的工作来实现将信息包从本地硬件送出/取出。中断服务程序应尽量减短中断阻塞时间。它只须处理那些只需最小执行时间的任务(如差错和状态变换)，而不应包含所有任务级的耗时处理工作。

为了将一些数据包的处理工作放置在任务级，用户中断服务程序必须调用 netJobAdd() 将相应的处理程序作为输入来生成一个任务级的数据处理任务。netJobAdd()接收一个程序指针和最多 5 个附加参数，这些参数将由添加的程序引用。程序清单 5－30 为 netJobAdd() 的函数原型。

程序清单 5－30　netJobAdd 函数原型

```
STATUS netJobAdd
    {
    FUNCPTR        rountine,        /* 处理函数的指针 */
    int             param1,         /* 可选参数 1 */
    int             param2,         /* 可选参数 2 */
    int             param3,         /* 可选参数 3 */
    int             param4,         /* 可选参数 4 */
    int             param5          /* 可选参数 5 */
    }
```

当用户调用 netJobAdd()时，要把驱动程序处理数据包的入口函数作为任务的入口点；然后用 netJobAdd()把该入口函数(包括该任务的入口参数和自变量)添加到 tNetTask 的工作队列。VxWorks 使用 tNetTask 来处理任务级的网络处理。使用 netJobAdd()有两点要明确：

① 用户可使用 netJobAdd()来生成任务但不处理接收到的数据包。

② netJobAdd()为同时被网络堆栈所使用的有限资源。若其溢出，则会造成通用网络堆栈的瘫痪。

5.5.2 RTL8019 网络芯片简介

在进行驱动程序编写之前，首先要了解 JX2410 教学系统中网络芯片的操作方法。JX2410 使用的网络芯片是由台湾 Realtek 公司生产的 RTL8019AS。它是一个高集成的以太网控制器芯片，不仅集成了介质访问控制（MAC）子层和物理层的性能，而且还可方便地设计基于 ISA 总线的系统，简单地与通用处理器进行接口。另外，它还具有与 NE2000 兼容，软件移植性强以及价格低廉等优点，因而在市场上的 10 Mbps 网卡中占有相当比例。

1. 主要性能

- 适应于 Ethernet II、IEEE802.3 协议、10Base5、10Base2 以及 10BaseT；
- 支持 8 位/16 位数据总线、8 个中断申请线以及 16 个 I/O 基地址选择；
- 全双工，收/发可同时达 10 Mbps 的速率，具有休眠模式，可降低功耗；
- 内置 16 KB SRAM，用于收/发缓冲，降低了对主处理器的速度要求；
- 可连接同轴电缆和双绞线，并可自动检测所连接的媒介类型；
- 支持闪存读/写；
- 允许 4 个诊断 LED 引脚可编程输出；
- 100 脚的 TQFP 封装，缩小了主机板尺寸。

2. 工作原理

根据数据链路的不同，可将以太网络控制器内部划分为远程 DMA(Remote DMA)信道和本地 DMA(Local DMA)信道两部分。本地 DMA 完成控制器与网络线的数据交换；处理器(Host)收/发数据只需对远程 DMA 操作。当主处理器要向以太网络发送数据时，先将一帧(Frame)数据经过远程 DMA 信道，送到以太网络控制器中的发送缓冲内存(Ring Buffer)；然后发出传送命令。以太网络控制器在送出前一帧数据后，继而完成此帧的发送。以太网络控制器接收到的数据通过 MAC 比较和 CRC 校验后，由 FIFO 存到接收缓冲区中；收满一帧后，以中断或缓存器标志的方式通知处理器。

接收逻辑在接收脉冲的控制下，将串行数据组成字节送至 FIFO 和 CRC 单元；发送逻辑在发送脉冲的控制下将 FIFO 送来的字节逐步按位移出，并送到 CRC。CRC 逻辑在接收时对输入的数据进行 CRC 校验，将结果与帧尾的 CRC 比较，若不同，则该帧数据将被拒收；在发送时 CRC 对帧数据产生 CRC，并附加在数据尾传送。地址识别逻辑对接收帧的目的地址与预先设置的本地实体地址进行比较，若不同且不满足广播地址(Broadcast Address)的设定要求，则该帧数据将被拒收；FIFO 逻辑对收/发的数据进行 16 字节的缓冲，以降低对本地 DMA 请求的频率。

RTL8019 是针对 PC 总线设计的，将其应用在嵌入式设备中，须考虑其软硬件设计上的

特殊性。嵌入式设备的主处理器可利用内存映像方法，将以太网络控制器内16个I/O地址上的缓存器映像到嵌入式操作系统中，这样就可用程序来操控以太网络控制器。图5-19为RTL8019芯片寄存器映射图。

No(Hex)	Page0		Page1	Page2	Page3	
	[R]	[W]	[R/W]	[R]	[R]	[W]
00	CR	CR	CR	CR	CR	CR
01	CLDA0	PSTART	PAR0	PSTART	*9346CR*	*9346CR*
02	CLDA1	PSTOP	PAR1	PSTOP	*BPAGE*	*BPAGE*
03	BNRY	BNRY	PAR2	—	*CONFIG0*	—
04	TSR	TPSR	PAR3	TPSR	*CONFIG1*	*CONFIG1*
05	NCR	TBCR0	PAR4	—	*CONFIG2*	*CONFIG2*
06	FIFO	TBCR1	PAR5	—	*CONFIG3*	*CONFIG3*
07	ISR	ISR	CURR	—	—	*TEST*
08	CRDA0	RSAR0	MAR0	—	*CSNSAV*	—
09	CRDA1	RSAR1	MAR1	—	—	*HLTCLK*
0A	*8019ID0*	RBCR0	MAR2	—	—	—
0B	*8019ID1*	RBCR1	MAR3	—	*INTR*	—
0C	RSR	RCR	MAR4	RCR	—	*FMWP*
0D	CNTR0	TCR	MAR5	TCR	*CONFIG4*	—
0E	CNTR1	DCR	MAR6	DCR	—	—
0F	CNTR2	IMR	MAR7	IMR	—	—
10~17	Remote DMA Port					
18~1F	Reset Port					

图5-19 RTL8019芯片寄存器映射图

3. 软件设计

对于RTL8019AS的软件设计，包括芯片初始化、发送和接收3部分。

(1) 芯片初始化

首先要对网卡进行复位。18H～1FH共8个地址，为复位端口。对该端口地址的读/写任何数，都会引起网卡的复位，代码参见程序清单5-31。

程序清单5-31 RTL8019复位操作

```
void rtl8019_Reset(void)
    {
        unsigned int i;
        unsigned char temp;
        for(i=0;i<250;i++);                            /* 简单的延时 */
        temp = inportb(RTL8019_BASE + 0x1f);           /* 读网卡的复位端口 */
        outportbb(RTL8019_BASE + 0x1f,temp);           /* 写网卡的复位端口 */
    }
```

说明：RTL8019_BASE是RTL8019的基地址；RTL8019全部的寄存器地址都是由它得

到的。因此在初始化时必须赋予它正确的基准地址值，在此所使用的值为 0x18000300。网卡复位完成之后，要对网卡的工作参数进行设置，使网卡开始工作。设置网卡参数主要是对命令寄存器 CR 进行设置；CR 主要用于选择寄存器页，启动或停止远程 DMA 操作以及执行命令。网卡初始化的软件编程参见程序清单 5-32。

程序清单 5-32　RTL8019 初始化

```
void rtl8019_init()
    {
        outportb(RTL8019_BASE + 0x00,0x21);     /* 选择页 0 寄存器，网卡停止运行 */
        outportb(RTL8019_BASE + 0x01,0x4c);     /* 接收缓冲区范围 */
        outportb(RTL8019_BASE + 0x02,0x80);     /* PSTOP，构造缓冲环：0x4c～0x80 */
        outportb(RTL8019_BASE + 0x03,0x4c);     /* BNRY，设置指针 */
        outportb(RTL8019_BASE + 0x04,0x40);     /* 发送缓冲区范围 */
        outportb(RTL8019_BASE + 0x0d,0x4c);
        outportb(RTL8019_BASE + 0x0e,0xc8);     /* 设置数据配置寄存器，使用 FIFO 缓存 */
                                                /* 普通模式，8 位数据 DMA */
        outportb(RTL8019_BASE + 0x0f,0xff);     /* 清除所有中断标志位 */
        outportb(RTL8019_BASE + 0x0f,0x00);     /* 设置中断屏蔽寄存器，屏蔽所有中断 */

        /* page(1)：选择页 1 寄存器 */
        outportb(RTL8019_BASE + 0x07,0x4d);     /* 初始化当前页寄存器 */
                                                /* 指向当前正在写的页的下一页 */
        outportb(RTL8019_BASE + 0x08,0x00);     /* 设置多址寄存器 MAR0～5，均设为 0 */
        outportb(RTL8019_BASE + 0x09,0x00);
        outportb(RTL8019_BASE + 0x0a,0x00);
        outportb(RTL8019_BASE + 0x0b,0x00);
        outportb(RTL8019_BASE + 0x0c,0x00);
        outportb(RTL8019_BASE + 0x0d,0x00);
        outportb(RTL8019_BASE + 0x0e,0x00);
        outportb(RTL8019_BASE + 0x0f,0x00);
        outportb(RTL8019_BASE + 0x00,0x21);     /* 选择页 0 寄存器，网卡执行命令 */
    }
```

初始化完网卡后，需对网卡的物理地址进行设置，才能正确地接收数据包。对使用 93c46 的网卡来说，上电复位时从 93c46 读入的网卡地址不会自动写入这里，而是存放在 RTL8019 的内存地址 0x0000、0x0002、0x0004、0x0006、0x0008、0x000A 和 0x000C 中。所以先要从这 6 个内存地址中读出网卡地址。在此程序中用到 4 个寄存器：RSAR1、RSAR0、RBCR1 和

RBCR0，这 4 个寄存器是专门用于读取网卡上 RAM 的。其中：

- RSAR1：网卡上 RAM 起始地址的高 8 位；
- RSAR0：网卡上 RAM 起始地址的低 8 位；
- RBCR1：要读取的字节数的计数（高 8 位）；
- RBCR0：要读取的字节数的计数（低 8 位）。

在该程序中寄存器 RSAR1 和 RSAR0 都设置为 0，所以是从网卡上的 0x0000 地址开始读取。程序中的 RBCR1 设置为 0，RBCR0 设置为 12，因为地址 0x0000～0x00ff 的 RAM 存储类型为 word，所以要读取 12 字节。读取完物理地址后，将读取的地址写到 PAR0～5 这 6 个寄存器中。这 6 个寄存器是网卡工作时用的地址，只有符合这些寄存器中所写地址的数据包才能接收。

如果系统中没有使用 93c46 来存储该网卡地址，则在软件中应自行产生或分配一个网卡地址，写入 6 个寄存器中。

(2) 发送数据

数据的发送过程应包含 3 个步骤：封装数据包；通过远程 DMA 将数据包送到数据发送缓冲区；通过 RTL8019 的本地 DMA 将数据送入 FIFO 进行发送，代码见程序清单 5 - 33。

程序清单 5 - 33 RTL8019 FIFO 操作

```
/* 将数据写入缓冲区 */
outportb(RTL8019_BASE + 0x00,0x22);

/* 设置中断状态寄存器为 40H,清除发送完成标志 */
outportb(RTL8019_BASE + 0x07,0x40);

/* 设置远程 DMA 地址寄存器 RSAR1 和 RSAR0 为 4000H,即发送缓冲区开始地址 */
outportb(RTL8019_BASE + 0x09,0x40);
outportb(RTL8019_BASE + 0x08,0x00);

/* 设置远程 DMA 字节计数寄存器 RBCR1 和 RBCR0 为发送数据包的长度 80 */
outportb(RTL8019_BASE + 0x0a,0x50);
outportb(RTL8019_BASE + 0x0b,0x00);

/* 设置 CR 为 12H,设置命令寄存器为远程 DMA 写 */
outportb(RTL8019_BASE + 0x00,0x12);

for(i = 0;i < 80;i++)
    outport(0x10 << 1, *(buffer + i)); /* 往数据端口写入发送数据 */
```

```
/* 查询中断状态寄存器 ISR,等待远程 DMA 完成 */
temp = inportb(RTL8019_BASE + 0x07);
outportb(RTL8019_BASE + 0x0b,0x00);
outportb(RTL8019_BASE + 0x0a,0x00);

/* 设置 CR 为 22H,设置 RBCR1 和 RBCR0 为 0,远程 DMA 停止 */
outportb(RTL8019_BASE + 0x00,0x22);

/* 设置中断状态寄存器 ISR 为 40H,清除发送完标志,启动本地 DMA 将数据送出 */
outportb(RTL8019_BASE + 0x07,0x40);
outportb(RTL8019_BASE + 0x06,0x50);

/* 设置发送字节计数器为发送数据包的长度 */
outportb(RTL8019_BASE + 0x05,0x00);

/* 设置发送页面起始地址 TPSR 为 40H,即发送缓冲区开始地址的高位字节 */
outportb(RTL8019_BASE + 0x04,0x40);

/* 启动发送 */
outportb(RTL8019_BASE + 0x00,0x26);
```

(3) 接收数据

RTL8019 收到一个完整的以太网数据包后,向 CPU 发出中断请求;CPU 响应 8019 后,即进入中断服务程序并开始接收数据。

5.5.3 网络驱动程序编写

网络的各功能部件如图 5-20 所示,网络设备驱动程序实际上是处理硬件和上层协议之间的接口程序。网络传输协议层在应用程序接口和网络接口之间分发数据;网络应用协议(如 IP 协议)层在网络主机之间传输数据;而接口层使主机隶属于硬件到相同物理媒质的通信。

在 VxWorks 中,网卡驱动程序又分为 END(Enhanced Network Driver)和 BSD 两种,如图 5-20(b)所示。

在 VxWorks 中,有一部分网络驱动程序是基于 BSD Unix 版本 4.3 的。这些驱动程序都定义在一个全局例程中,那就是 xxattach 子程序。xxattach 子程序包含 5 个函数指针(见表 5-10),它们都被映射到 ifnet 结构中,且可在 IP 协议层的任何地方被调用。

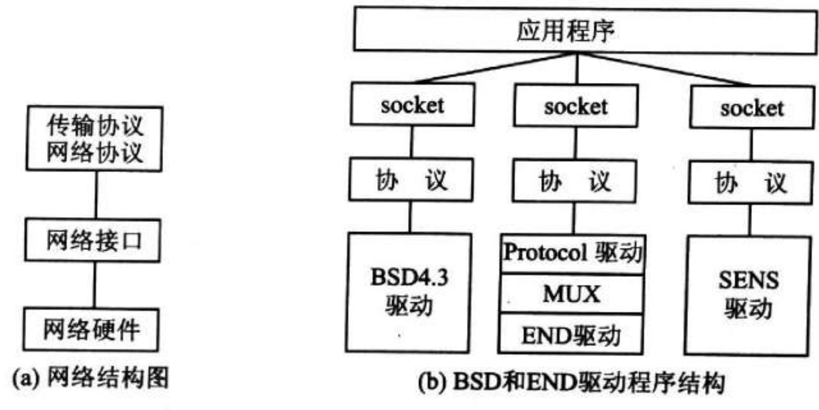


图 5－20　VxWorks 网络驱动程序结构示意图

表 5－10　BSD 网络驱动程序的例程列表

驱动程序指定函数	函数指针	功　能
xxInit( )	if_init	初始化接口
xxOutput( )	if_output	对要传输的输出分组进行排队
xxIoctl( )	if_ioctl	处理 I/O 控制命令
xxReset( )	if_reset	复位接口设备
xxWatchdog( )	if_watchdog (optional)	周期性接口例程

驱动程序入口 xxattach( )调用 ether_attach( )，将上述 5 个函数映射到 ifnet 结构中。ether_attach()调用如下：

```
ether_attach( (IFNET * ) & pDrvCtrl -> idr,
            unit,
            "xx",
            (FUNCPTR) NULL,
            (FUNCPTR) xxIoctl,
            (FUNCPTR) ether_output( ),
            (FUNCPTR) xxReset
    );
pDrvCtrl -> idr.ac_if.if_start = (FUNCPTR)xxTxStartup;
```

上述参数中，首先是一个接口数据记录(IDR, Interface Data Record)、设备编号和设备名。剩下 4 个参数就是相关驱动程序的函数指针：第一个函数指针指的是 init( )例程，该例程可有可无；第二个函数指针指的是 ioctl()接口，它允许上层来控制设备状态；第三个函数指针指的是把数据包送到物理层；最后一个函数指针指的是，如果 TCP/IP 堆栈需要复位，则它

就复位该设备。

接着下面那一句代码表示添加数据传输例程到 IDR。ether_output( )例程被调用后，传输开始例程就被 TCP/IP 协议堆栈调用。

在这个入口驱动程序中还包括设备的初始化、发送和接收描述符的初始化等。

END 驱动程序基于 MUX 模式，这也是目前在 VxWorks 操作系统上应用最广泛的一种网络驱动程序。在该模式下，网络驱动程序被划分为协议组件和硬件组件。MUX 数据链路层和网络层之间的接口用来管理网络协议接口和低层硬件接口之间的交互，从而将硬件从网络协议的细节中隔离出来，改变了以前使用输入钩子例程来处理收/发数据包的方法。链路层上的驱动程序需要访问网络(IP 或其他协议)层时，也会调用相关的 MUX 例程。值得注意的是，网络层协议和数据链路层驱动程序不能直接通信，而必须通过 MUX 接口。其整体构架如图 5-21 所示。

对 END 程序的设计，应注意以下几点：

首先应把驱动程序加入 VxWorks 系统中。VxWorks 系统在启动时，先执行任务 tUsrRoot 初始化网络，而任务 tUsrRoot 又调用函数 muxDevLoad()(muxDevLoad()函数又调用驱动程序中的 endLoad()函数)。endLoad()创建一个 END_OBJ 结构和 NET_FUNCS 结构；muxDevLoad()装载驱动程序后，就会调用 muxDevStart()(调用驱动程序中的 endStart()函数)，endStart 函数应激活驱动程序并注册中断服务程序 ISR。

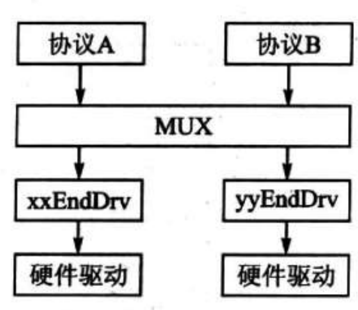


图 5-21 MUX 层的作用

利用 netBufLib 进行缓冲管理。netBufLib 允许为网络设备驱动程序和网络服务程序建立和管理一个内存池，数据被存放在簇中，通过调用 mblks 和 clblks 的数据结构形式来进行数据交换。

建立内存池，所有内存池的管理都通过 mblks、clblks 以及簇的数据结构来实现。如何配置一个内存池，取决于网络设备驱动程序和网络服务程序是否想使用这种结构。

下面具体分析 JX2410 的网络驱动程序。

1. 定义设备的描述信息

该结构中首先应包含一个 END_OBJ 结构，后面再加上一些自定义的成员。代码见程序清单 5-34。

程序清单 5-34 END_OBJ 结构

```
typedef struct rtl8019_device
    {
    END_OBJ          endObj;                        /* END_OBJ 对象 */
```

```
    END_ERR              lastError;                    /* 传递到 MUX 的错误统计信息 */
    int                  lastIntError;                 /* 中断错误统计 */
    int                  unit;                         /* 网卡单元号 */
    int                  ivec;                         /* 中断向量号 */
    int                  ilevel;                       /* 中断号(实际上与中断向量号相同) */
    int                  byteAccess;                   /* 存取模式 */
    int                  usePromEnetAddr;              /* 是否在 ROM 中读取 MAC 配置信息 */
    ULONG                base;                         /* 基地址 */
    int                  offset;                       /* 偏移 */
    char                 packetBuf [RTL8019_BUFSIZ];   /* 数据包缓冲 */
    volatile long        flags;                        /* 标志 */
    volatile UCHAR       current;                      /* 当前页序号 */
    volatile ULONG       imask;                        /* 中断掩码 */
    volatile RTL8019_STAT stats;                       /* RTL8019 状态 */
    UCHAR                enetAddr[6];                  /* 以太网 IP 地址 */
    UCHAR                mcastFilter[8];               /* 多波过滤器设置值 */
    UCHAR                nextPacket;                   /* 接收的下一包序号 */
    CL_POOL_ID           clPoolId;
    int                  configRegA;                   /* 配置寄存器 A 的值 */
    int                  configRegB;                   /* 配置寄存器 B 的值 */
} RTL8019END_DEVICE;
```

2. 驱动程序的加载

在编写 BSP 时，如果涉及网络协议，则会使用一个配置文件 configNet. h。可以通过修改这个文件来指定该设备驱动的加载入口。configNet. h 的内容参见程序清单 5 - 35。

程序清单 5 -35　JX2410 网络设备驱动程序的初始配置

```
#define RTL8019_LOAD_FUNC          rtl8019EndLoad
#define RTL8019_LOAD_STRING        "18000300:4:4:1:0:0"
IMPORT  END_OBJ * RTL8019_LOAD_FUNC(char * , void * );

END_TBL_ENTRY endDevTbl [] =
{
    { 0, RTL8019_LOAD_FUNC, RTL8019_LOAD_STRING, FALSE, NULL, FALSE},
    { 0, END_TBL_END, NULL, 0, NULL, FALSE}
};
```

在 configNet.h 配置文件中，主要涉及一个表格(END_TBL_ENTRY)的填写。这个表格中的每一项对应一个网卡的加载信息，END_TBL_ENTRY 的结构定义见程序清单 5-36。

程序清单 5-36　END_TBL_ENTRY 结构定义

```
typedef struct end_tbl_entry
    {
    int unit;                                           /* 设备编号 */
    END_OBJ * (* endLoadFunc) (char *, void *);         /* 加载函数 */
    char * endLoadString;                               /* 初始化资源字串 */
    BOOL endLoad;                                       /* 缓冲类型 */
    void * pBSP;                                        /* BSP 内部指针 */
    BOOL processed;                                     /* 处理完成与否的标志 */
    } END_TBL_ENTRY;
```

按照 END_TBL_ENTRY 的定义，在 endDevTbl 数组中声明了一个网卡。其加载程序是 rtl8019EndLoad()，资源字串是“18000300:4:4:1:0:0”。资源字串的描述与加载程序的处理有关，一般须包含：网卡基地址、网卡中断号和网卡有关的参数等。不同的网卡其资源字串会有所不同，而且排列顺序也没有硬性规定，只需在 endLoad 程序中能够得到需要的数据。在 endDevTbl 的最后一项，应使用 END_TBL_END 作为加载函数，表示 endDevTbl 的结束。程序清单 5-37 为网络驱动程序的加载代码。

程序清单 5-37　网络驱动程序的加载

```
END_OBJ * rtl8019EndLoad
    (
    char * initString,          /* 初始化资源字串 */
    void * pBSP                 /* BSP 内部指针 */
    )
    {
    RTL8019END_DEVICE     * pDrvCtrl;
    int          level;
    UCHAR        regVal;

    /* 首先检查资源字串的合法性 */
    if (initString == NULL){
    DBG_PRINTF(("EndLoad:initString is zero.\n"));
    }
    else {
```

```
DBG_PRINTF(("EndLoad:initString is %s.\n", initString));
}
if (initString == NULL)
{
DBG_PRINTF(("EndLoad:initString is zero.\n"));
return (NULL);
}
else
{
DBG_PRINTF(("EndLoad:initString is %s\n", initString));
}

/* 如果输入的资源字串为空,则使用默认值 */
if (initString[0] == '\0')
    {
    bcopy((char *)RTL8019_DEV_NAME, initString, RTL8019_DEV_NAME_LEN);
    return (NULL);
    }

/* 为设备描述符申请空间 */
pDrvCtrl = (RTL8019END_DEVICE *) calloc (sizeof(RTL8019END_DEVICE), 1);
if (pDrvCtrl == NULL)
{
DBG_PRINTF(("allocate the device structure fail! \n"));
goto errorExit;
}

/* 解析资源字串,获取硬件资源信息 */
if (rtl8019Parse (pDrvCtrl, initString) == ERROR)
{
DBG_PRINTF(("parse the init string fail! \n"));
goto errorExit;
}

/* 检测网卡芯片是否工作正常 */
if(rtl8019Verify(pDrvCtrl) != OK)
{
```

```
ERR_PRINTF(("RTL8019 not found at 0x%x! \n", pDrvCtrl -> base));
goto errorExit;
}

/*
 * 如果在资源字串中定义了两个可选的寄存器配置:configRegA 和 configRegB,
 * 则在此进行设置
 */
if (pDrvCtrl -> configRegA != 0)
    {
    /* 该操作必须为原子操作,所以关闭中断 */
    level = intLock ();
    SYS_OUT_CHAR (pDrvCtrl, ENE_CMD, CMD_PAGE0);
    SYS_IN_CHAR (pDrvCtrl, ENE_RBCR0, &regVal);
    SYS_OUT_CHAR (pDrvCtrl, ENE_RBCR0, pDrvCtrl -> configRegA);
    intUnlock (level);
    }
if (pDrvCtrl -> configRegB != 0)
    {
    level = intLock ();
    SYS_IN_CHAR (pDrvCtrl, ENE_RBCR1, &regVal);
    SYS_OUT_CHAR (pDrvCtrl, ENE_RBCR1, pDrvCtrl -> configRegB);
    intUnlock (level);
    }

/* 首先停止 RTL8019 */
SYS_OUT_CHAR (pDrvCtrl, ENE_CMD, CMD_NODMA | CMD_PAGE0 | CMD_STOP);

/*
 * 设置操作模式:8 位/16 位(取决于硬件设计)
 */
if (pDrvCtrl -> byteAccess)
SYS_OUT_CHAR (pDrvCtrl, ENE_DCON, DCON_BSIZE1 | DCON_BUS_8
        | DCON_LOOPBK_OFF);
else
SYS_OUT_CHAR (pDrvCtrl, ENE_DCON, DCON_BSIZE1 | DCON_BUS16
        | DCON_LOOPBK_OFF);
```

```
    /* 获取 BSP 为该设备提供的网络地址 */
    SYS_ENET_ADDR_GET (pDrvCtrl);

    /* 在完成硬件的检查和设置之后,下面将进行驱动程序有关对象和缓冲的处理 */
    /* 初始化 END_OBJ 结构 */
    if (END_OBJ_INIT (&pDrvCtrl -> endObj,
          (DEV_OBJ *)pDrvCtrl, (char *)RTL8019_DEV_NAME,
              pDrvCtrl -> unit, &rtl8019FuncTable,
              "rtl8019 Enhanced Network Driver") == ERROR)
    {
    DBG_PRINTF(("initialize the END parts of the structure fail! \n"));
    goto errorExit;
    }

    /* 缓冲初始化 */
    if (rtl8019MemInit (pDrvCtrl) == ERROR)
    {
    DBG_PRINTF(("Perform memory allocation/distribution fail! \n"));
    goto errorExit;
    }

    /* 初始化 MIB2 接口单元 */
    if (END_MIB_INIT (&pDrvCtrl -> endObj, M2_ifType_ethernet_csmacd,
              &pDrvCtrl -> enetAddr[0], 6,
              SIZEOF_ETHERHEADER + ETHERMTU,
              END_SPEED) == ERROR)
    {
    DBG_PRINTF(("initialize the MIB2 parts of the structure fail! \n"));
    goto errorExit;
    }

    /* 设置网卡就绪标志 */
    END_OBJ_READY (&pDrvCtrl -> endObj,
            IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST);
    return (&pDrvCtrl -> endObj);

errorExit:
```

```
    if (pDrvCtrl != NULL)
    free ((char *)pDrvCtrl);

    return (NULL);
    }
```

在上面的加载程序中首先使用了一个系统提供的处理函数 END_OBJ_INIT 来初始化 END_OBJ，其主要工作是 END_OBJ 结构的填写。END_OBJ 结构中描述了该网卡有关的一些参数和处理函数，具体定义见程序清单 5-38。

程序清单 5-38 $(WINDBASE)\target\h\end.h

```
typedef struct end_object
    {
    NODE node;
#ifndef _WRS_VXWORKS_5_X
    BOOL  nptFlagSpace;                      /* 现在没有使用该标志 */
#endif   /* _WRS_VXWORKS_5_X */
    DEV_OBJ devObject;                       /* 继承下来的设备描述符 */
    STATUS (*receiveRtn) ();                 /* 接收数据时的回调函数 */
    struct net_protocol *outputFilter;       /* 可选的输出过滤函数 */
    void * pOutputFilterSpare;               /* 输出过滤处理时使用的指针 */
    BOOL attached;                           /* 网卡被注册与否的标志 */
    SEM_ID txSem;                            /* 发送时使用的互斥信号量 */
    long flags;                              /* 通用标志 */
    struct net_funcs *pFuncTable;            /* 功能函数列表 */
    M2_INTERFACETBL mib2Tbl;                 /* MIBII 计数器. */
    LIST multiList;                          /* 多播地址链表的头 */
    int   nMulti;                            /* 多播地址链表节点的数量 */
    LIST protocols;                          /* 协议链表 */
    int snarfCount;                          /* 是否阻止数据传递给低优先级的协议 */
    NET_POOL_ID pNetPool;                    /* MUX 缓冲信息 */
#ifndef _WRS_VXWORKS_5_X
    void * pNptCookie;                       /* 现在未用 */
#endif /* _WRS_VXWORKS_5_X */
    M2_ID * pMib2Tbl;                        /* 兼容 2233 MIB 接口的对象指针 */
} END_OBJ;
```

在上面的结构中，多数成员可以使用系统提供的默认设置；而由 pFuncTable 指示的功能

函数列表中的一些函数指针规定了具体网卡的所有操作，这是需要我们填写的。net_funcs 结构定义见程序清单 5－39。

程序清单 5－39　$ (WINDBASE)\target\h\end.h

```
typedef struct net_funcs
    {
    STATUS ( * start) (END_OBJ * );                          /* start 例程 */
    STATUS ( * stop) (END_OBJ * );                           /* stop 例程 */
    STATUS ( * unload) (END_OBJ * );                         /* unload 例程 */
    int ( * ioctl) (END_OBJ * , int, caddr_t);               /* ioctl 例程 */
    STATUS ( * send) (END_OBJ *  , M_BLK_ID);                /* send 例程 */
    STATUS ( * mCastAddrAdd) (END_OBJ * , char * );          /* 多播地址添加例程 */
    STATUS ( * mCastAddrDel) (END_OBJ * , char * );          /* 多播地址删除例程 */
    STATUS ( * mCastAddrGet) (END_OBJ * , MULTI_TABLE * );   /* 多播地址获取例程 */
    STATUS ( * pollSend) (END_OBJ * , M_BLK_ID);             /* 轮询方式发送例程 */
    STATUS ( * pollRcv) (END_OBJ * , M_BLK_ID);              /* 轮询方式接收例程 */
    M_BLK_ID ( * formAddress) (M_BLK_ID, M_BLK_ID, M_BLK_ID, BOOL);
                                                             /* 地址信息检查例程 */
    STATUS ( * packetDataGet) (M_BLK_ID, LL_HDR_INFO * );
                                                             /* 数据读例程 */
    STATUS ( * addrGet) (M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID);
                                                             /* 获取数据包地址例程 */
    int ( * endBind) (void * , void * , void * , long type);
    } NET_FUNCS;
```

与上面的结构对应，需要在网卡驱动程序中定义一个相同的结构变量，来指定各互利函数的具体值（见程序清单 5－40）。当然有些函数可以忽略。

程序清单 5－40　$ (BSPBASE)\rtl8019End.c

```
LOCAL NET_FUNCS rtl8019FuncTable =
    {
    (FUNCPTR) rtl8019Start,
    (FUNCPTR) rtl8019Stop,
    (FUNCPTR) rtl8019Unload,
    (FUNCPTR) rtl8019Ioctl,
    (FUNCPTR) rtl8019Send,
    (FUNCPTR) rtl8019MCastAdd,
    (FUNCPTR) rtl8019MCastDel,
```

```
(FUNCPTR) rtl8019MCastGet,
(FUNCPTR) rtl8019PollSend,
(FUNCPTR) rtl8019PollRecv,
endEtherAddressForm,
endEtherPacketDataGet,
endEtherPacketAddrGet
};
```

在定义了上述结构之后，即可通过 endObjInit 函数来进行 END_OBJ 对象的初始化，参见程序清单 5-41。其中的参数包括上面定义的函数列表等。

程序清单 5-41　$(WINDBASE)\target\src\drv\end\endLib.c

```
STATUS endObjInit
    (
    END_OBJ *      pEndObj,                  /* END_OBJ 对象指针 */
    DEV_OBJ *      pDevice,                  /* DEV_OBJ 对象指针 */
    char *         pBaseName,                /* 网卡名称 */
    int            unit,                     /* 网卡编号 */
    NET_FUNCS *    pFuncTable,               /* 网卡操作函数列表的指针 */
    char *         pDescription              /* 网卡的描述信息 */
    )
    {
    pEndObj -> devObject.pDevice = pDevice;

    /* 首先创建发送处理时使用的互斥信号量 */
    pEndObj -> txSem = semMCreate ( SEM_Q_PRIORITY |
                  SEM_DELETE_SAFE |
                  SEM_INVERSION_SAFE);
    if (pEndObj -> txSem == NULL)
    {
    return (ERROR);
    }
    pEndObj -> flags = 0;
    lstInit (&pEndObj -> protocols);         /* 初始化(清空)该网卡有关的协议链表 */

    /* 检查并控制网卡名称的长度 */
    if (strlen(pBaseName) > sizeof(pEndObj -> devObject.name))
    pBaseName[sizeof(pEndObj -> devObject.name - 1)] = EOS;
```

```
/* 保存网卡名称 */
strcpy (pEndObj -> devObject.name, pBaseName);

/* 检查并控制网卡描述字串的长度 */
if (strlen(pDescription) > sizeof(pEndObj -> devObject.description))
pDescription[sizeof(pEndObj -> devObject.description - 1)] = EOS;
strcpy (pEndObj -> devObject.description, pDescription);

/* 设置网卡编号 */
pEndObj -> devObject.unit = unit;

/* 设置处理函数 */
pEndObj -> pFuncTable = pFuncTable;

/* 清除多播地址信息 */
lstInit (&pEndObj -> multiList);
pEndObj -> nMulti = 0;

pEndObj -> snarfCount = 0;

return (OK);
}
```

进行 END_OBJ 的初始化之后，多数操作将通过 pFuncTable 定义的函数指针，由相应的功能单元来调用。

在加载网卡驱动(endLoad)时，有项工作就是内存初始化，由 rtl8019MemInit()实现。网卡驱动中的内存管理是基于内存池的，每个 END 单元都需要有其自身的内存池。内存池一般由 mBlk 结构、clBlk 结构和内存块构成。程序清单 5－42 为 mBlk 结构和 clBlk 结构的定义。

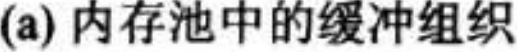
程序清单 5－42　网络驱动程序中存储器有关的结构定义

```
typedef struct clBlk
    {
    CL_BLK_LIST     clNode;         /* 指向下一节点的指针 */
    UINT            clSize;         /* 簇的大小 */
    int             clRefCnt;       /* 引用计数 */
    FUNCPTR         pClFreeRtn;     /* 释放例程 */
```

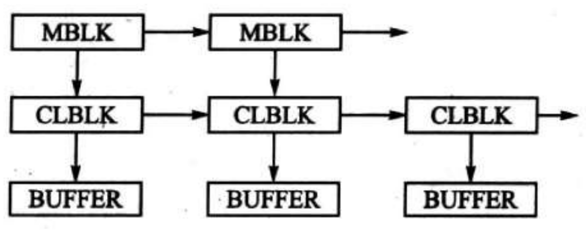
```
    int              clFreeArg1;            /* 释放例程的参数 1 */
    int              clFreeArg2;            /* 释放例程的参数 2 */
    int              clFreeArg3;            /* 释放例程的参数 3 */
    struct netPool * pNetPool;              /* netPool 指针 */
    } CL_BLK;

typedef struct mBlk
    {
    M_BLK_HDR        mBlkHdr;
    M_PKT_HDR        mBlkPktHdr;
    CL_BLK *         pClBlk;
    } M_BLK;

typedef union clBlkList
    {
    struct clBlk *   pClBlkNext;
    char *           pClBuf;
    } CL_BLK_LIST;
```

使用上述结构后，对于缓冲的访问就需要首先通过 MBLK，然后根据 CLBLK 链表读取对应的缓冲区。其排列示意图如图 5-22 所示。

MBLK	CLBLK	BUFFER
MBLK	CLBLK	BUFFER
MBLK	CLBLK	BUFFER
MBLK	CLBLK	BUFFER
MBLK	CLBLK	BUFFER
MBLK	CLBLK	BUFFER
MBLK	CLBLK	BUFFER
MBLK	CLBLK	BUFFER
...	...	...



(a) 内存池中的缓冲组织



(b) MBLK与CLBLK链表示意图

图 5-22 网络驱动的缓冲管理

在 rtl8019MemInit 首先设置内存池的参数；然后申请内存空间；再使用 netPoolInit 来形成具体的内存池。在网卡驱动程序的其他地方，如果需要进行内存分配，则都会在这个内存池中申请。申请的办法：首先使用 netClusterGet 申请缓冲区；然后使用 netClBlkGet 申请 CLBLK，在填写数据后再使用 netClBlkJoin 和 netMblkClJoin 将缓冲区、CLBLK 以及 MBLK 关联起来；最后将 MBLK 作为参数传递给处理函数。这种管理方法有很大好处，因为一个

CLBLK 可与多个 MBLK 关联，这就使得在不同协议之间传递数据变得容易，只须传递指针，而无须拷贝其中的数据。程序清单 5－43 为存储器初始化的代码。

程序清单 5－43　$(BSPBASE)\rtl8019End.c　存储器初始化

```
LOCAL STATUS rtl8019MemInit
    (
    RTL8019END_DEVICE * pDrvCtrl              /* 设备指针 */
    )
    {

    M_CL_CONFIG    eneMclBlkConfig;
    CL_DESC     clDesc;

    bzero ((char * )&eneMclBlkConfig, sizeof(eneMclBlkConfig));
    bzero ((char * )&clDesc, sizeof(clDesc));
    clDesc.clNum     = RTL8019_RX_BUFS;          /* 缓冲区的数量 */
    clDesc.clSize    = RTL8019_BUFSIZ;           /* 每个缓冲区的大小 */
    clDesc.memSize   = ((clDesc.clNum * (clDesc.clSize + 8)) + 4);
    eneMclBlkConfig.mBlkNum    = RTL8019_RX_BUFS * 2;
    eneMclBlkConfig.clBlkNum   = clDesc.clNum;

    /* 计算所需的内存大小 */
    eneMclBlkConfig.memSize =
        (eneMclBlkConfig.mBlkNum * (MSIZE + sizeof (long)))
        + (eneMclBlkConfig.clBlkNum * (CL_BLK_SZ + sizeof (long)));
    eneMclBlkConfig.memArea = (char * ) memalign(sizeof (long),
                    eneMclBlkConfig.memSize);
    if (eneMclBlkConfig.memArea == NULL)
        return (ERROR);

    /* 分配内存 */
    clDesc.memArea = (char * ) cacheDmaMalloc (clDesc.memSize);
    if (clDesc.memArea == NULL)
    return (ERROR);
    pDrvCtrl -> endObj.pNetPool = (NET_POOL_ID) malloc (sizeof(NET_POOL));
    if (pDrvCtrl -> endObj.pNetPool == NULL)
        return (ERROR);
```

```
    /* 初始化内存池 */
    if (netPoolInit (pDrvCtrl -> endObj.pNetPool, &eneMclBlkConfig,
                    &clDesc, 1, NULL) == ERROR)
        return (ERROR);

    /* 保存内存池 ID */
    pDrvCtrl -> clPoolId = clPoolIdGet (pDrvCtrl -> endObj.pNetPool,
                    RTL8019_BUFSIZ, FALSE);

    return (OK);
    }
```

3. 驱动程序相关例程

在完成加载程序编写之后，剩下的工作就是要编写一些具体的处理函数。这些函数列在加载时被登记的设备对象的结构中。下面分别说明 Ioctl、收/发处理和中断处理函数的编写。

Ioctl 主要完成网卡的启动和停止，以及 MAC 地址设置等，由函数 rtl8019Ioctl()实现，如程序清单 5-44 所列。

程序清单 5-44　$(BSPBASE)\rtl8019End.c　Ioctl

```
LOCAL int rtl8019Ioctl
    (
    void*       pCookie,     /* 设备指针 */
    int         cmd,
    caddr_t     data
    )
    {
    int error = 0;
    long value;
    RTL8019END_DEVICE * pDrvCtrl = (RTL8019END_DEVICE *) pCookie;

    switch ((UINT) cmd)
        {
        case EIOCSADDR:
            if (data == NULL)
            {
            DBG_PRINTF(("EIOCSADDR data is NULL! \n"));
            return (EINVAL);
```

```
        }
        bcopy ((char *)data, (char *)END_HADDR(&pDrvCtrl -> endObj),
        END_HADDR_LEN(&pDrvCtrl -> endObj));
        break;

    case EIOCGADDR:
        if (data == NULL)
        {
        DBG_PRINTF(("EIOCGADDR data is NULL! \n"));
        return (EINVAL);
        }
        bcopy ((char *)END_HADDR(&pDrvCtrl -> endObj), (char *)data,
        END_HADDR_LEN(&pDrvCtrl -> endObj));
        break;

    case EIOCSFLAGS:
        value = (long)data;
        if (value < 0)
        {
        value = - value;
        value-- ;
        END_FLAGS_CLR (&pDrvCtrl -> endObj, value);
        }
        else
        END_FLAGS_SET (&pDrvCtrl -> endObj, value);
        rtl8019Config (pDrvCtrl, TRUE);
        break;

    case EIOCGFLAGS:
        *(int *)data = END_FLAGS_GET(&pDrvCtrl -> endObj);
        break;

    case EIOCPOLLSTART:
        error = rtl8019PollStart (pDrvCtrl);
        break;

    case EIOCPOLLSTOP:
```

```
        error = rtl8019PollStop (pDrvCtrl);
        break;

    case EIOCGMIB2:
        if (data == NULL)
        {
        DBG_PRINTF(("EIOCGMIB2 data is NULL! \n"));
        return (EINVAL);
        }
        bcopy((char *)&pDrvCtrl -> endObj.mib2Tbl, (char *)data,
            sizeof(pDrvCtrl -> endObj.mib2Tbl));
        break;

    case EIOCGFBUF:
        if (data == NULL)
         {
         DBG_PRINTF(("EIOCGFBUF data is NULL! \n"));
         return (EINVAL);
         }
         break;

    case EIOCMULTIADD:
        error = rtl8019MCastAdd((void *)pDrvCtrl, (char *)data);
        break;

    case EIOCMULTIDEL:
        error = rtl8019MCastDel((void *)pDrvCtrl, (char *)data);
        break;

    case EIOCMULTIGET:
        error = rtl8019MCastGet((void *)pDrvCtrl, (MULTI_TABLE *)data);
        break;

    default:
        DBG_PRINTF(("unknow command %x! \n", cmd));
        error = EINVAL;
        }

    return (error);
}
```

对于数据的发送，首先要获取发送互斥信号量；然后将数据拷贝到硬件发送缓冲区中；最后启动发送即可。代码见程序清单 5-45。

程序清单 5-45　$(BSPBASE)\rtl8019End.c　数据发送

```
LOCAL STATUS rtl8019Send
    (
    void *        pCookie,            /* 设备指针 */
    M_BLK_ID      pMblk               /* 待发送的数据 */
    )
    {
    int           len;
    UCHAR         cmdStat;
    RTL8019END_DEVICE * pDrvCtrl = (RTL8019END_DEVICE *) pCookie;
    if (pDrvCtrl -> flags & (END_OVERWRITE | END_OVERWRITE2))
    {
    return (END_ERR_BLOCK);
    }

    /* 获取发送互斥信号量 */
    END_TX_SEM_TAKE (&pDrvCtrl -> endObj, WAIT_FOREVER);
    pDrvCtrl -> flags |= END_TX_BLOCKED;
    if (pDrvCtrl -> flags & END_TX_IN_PROGRESS)
    {
    int cnt;
    DBG_PRINTF (("rtl8019Send: waiting for TX_IN_PROGRESS\n"));

    /* 等待 1 s */
    cnt = sysClkRateGet ();
    while ((pDrvCtrl -> flags & END_TX_IN_PROGRESS) && (cnt-- > 0))
        taskDelay (1);
    }

    /* 关闭中断，检查缓冲覆盖标志 */
    SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, 0);
    if (pDrvCtrl -> flags & (END_OVERWRITE | END_OVERWRITE2))
    {
    DBG_PRINTF (("rtl8019Send: overwrite detected (timeout)\n"));
```

```
pDrvCtrl -> flags &= ~END_TX_BLOCKED;
END_TX_SEM_GIVE (&pDrvCtrl -> endObj);
return (END_ERR_BLOCK);
}

/* 如果上次发送超时,则重新启动芯片 */
if (pDrvCtrl -> flags & END_TX_IN_PROGRESS)
{
UCHAR tstat;
SYS_IN_CHAR (pDrvCtrl, ENE_TSTAT, &tstat);
SYS_IN_CHAR (pDrvCtrl, ENE_CMD, &cmdStat);
DBG_PRINTF (("rtl8019Send: timeout: flags = %x cmd = %02x stat = %02x\n",
        pDrvCtrl -> flags, cmdStat, tstat));
rtl8019Config (pDrvCtrl, FALSE);
}

/* 拷贝 MBLK 关联的数据,并且释放 MBLK */
len = netMblkToBufCopy (pMblk, pDrvCtrl -> packetBuf, NULL);
netMblkClChainFree (pMblk);
len = max (len, ETHERSMALL);

/* 将数据写到硬件缓冲区中 */
SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, 0x00);
rtl8019DataOut (pDrvCtrl, pDrvCtrl -> packetBuf, len, (RTL8019_TSTART << 8));

/* 强制更新写缓冲 */
CACHE_PIPE_FLUSH ();

/* 启动硬件发送数据 */
SYS_OUT_CHAR (pDrvCtrl, ENE_TSTART, RTL8019_TSTART);
SYS_OUT_CHAR (pDrvCtrl, ENE_TCNTH, len >> 8);
SYS_OUT_CHAR (pDrvCtrl, ENE_TCNTL, len & 0xff);
SYS_OUT_CHAR (pDrvCtrl, ENE_CMD, CMD_TXP | CMD_START | CMD_NODMA);

/* 标记发送处理状态 */
pDrvCtrl -> flags |= END_TX_IN_PROGRESS;
```

```
/* 使能设备的发送中断 */
pDrvCtrl -> imask |= (IM_TXEE | IM_PTXE);
SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, pDrvCtrl -> imask);

/* 释放发送信号量 */
END_TX_SEM_GIVE (&pDrvCtrl -> endObj);

/* 更新发送统计 */
END_ERR_ADD (&pDrvCtrl -> endObj, MIB2_OUT_UCAST, +1);

return (OK);
}
```

网卡驱动中大部分工作将由中断完成，中断处理的详细情况见程序清单 5－46。在中断处理例程中会根据不同的中断类型，利用 netJobAdd 生成相应的任务级处理工作。例如：对于缓冲被覆盖的错误，会通过

```
netJobAdd ( muxError, &pDrvCtrl -> endObj,&pDrvCtrl -> lastError,0, 0, 0);
```

生成错误处理；对于接收数据，则通过

```
(void)netJobAdd ((FUNCPTR)rtl8019HandleRcvInt, (int)pDrvCtrl, 0,0,0,0);
```

生成接收处理任务等。

程序清单 5－46　$(BSPBASE)\rtl8019End.c　中断处理

```
LOCAL void rtl8019Int
    (
    RTL8019END_DEVICE    *pDrvCtrl
    )
    {
    UCHAR val;
    UCHAR intStat;
    UCHAR txStat;
    UCHAR rxStat;

    pDrvCtrl -> stats.interrupts++;

    /* 读取中断状态 */
    SYS_IN_CHAR (pDrvCtrl, ENE_INTSTAT, &intStat);
```

```
intStat &= pDrvCtrl -> imask;
SYS_OUT_CHAR (pDrvCtrl, ENE_INTSTAT, intStat);

/* 读取收/发状态 */
SYS_IN_CHAR (pDrvCtrl, ENE_TSTAT, &txStat);
SYS_IN_CHAR (pDrvCtrl, ENE_RSTAT, &rxStat);

/* 更新统计计数器 */
SYS_IN_CHAR (pDrvCtrl, ENE_COLCNT, &val);
pDrvCtrl -> stats.collisions += val;
SYS_IN_CHAR (pDrvCtrl, ENE_ALICNT, &val);
pDrvCtrl -> stats.aligns += val;
SYS_IN_CHAR (pDrvCtrl, ENE_CRCCNT, &val);
pDrvCtrl -> stats.crcs += val;
SYS_IN_CHAR (pDrvCtrl, ENE_MPCNT, &val);
pDrvCtrl -> stats.missed += val;

/* 缓冲区错误检查 */
if (intStat & ISTAT_OVW)
{
UCHAR cmdStat;
END_ERR_ADD (&pDrvCtrl -> endObj, MIB2_IN_ERRS, +1);
if (!(pDrvCtrl -> flags & END_OVERWRITE2))
    {
    pDrvCtrl -> flags |= (END_OVERWRITE | END_OVERWRITE2);
    pDrvCtrl -> lastError.errCode = END_ERR_WARN;
    pDrvCtrl -> lastError.pMesg = "Overwrite";

    /* 检测到缓冲被覆盖的错误,则使用 netJobAdd 生成一个错误处理任务 */
    netJobAdd ((FUNCPTR) muxError, (int) &pDrvCtrl -> endObj,
          (int) &pDrvCtrl -> lastError,0, 0, 0);
    pDrvCtrl -> stats.overwrite ++ ;
    SYS_IN_CHAR (pDrvCtrl, ENE_COLCNT, &cmdStat);
    SYS_OUT_CHAR (pDrvCtrl, ENE_CMD, CMD_NODMA | CMD_PAGE0 | CMD_STOP);
    DBG_PRINTF (("rtl8019Int: overwrite detected\n"));

    /* 检测到缓冲恢复,则使用 netJobAdd 生成一个恢复处理任务 */
```

```
        netJobAdd ((FUNCPTR) rtl8019OverwriteRecover, (int) pDrvCtrl,
                   cmdStat, 0, 0, 0);
        }

    pDrvCtrl -> imask = 0;
    SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, pDrvCtrl -> imask);
    return;
    }

    /* 收/发错误处理 */
    if (intStat & ISTAT_RXE)
        {
    if (!pDrvCtrl -> lastIntError)
        {
        END_ERR_ADD (&pDrvCtrl->endObj, MIB2_IN_ERRS, +1);
        pDrvCtrl -> lastError.errCode = END_ERR_WARN;
        pDrvCtrl -> lastError.pMesg = "receive error";

        /* 检测到接收错误,使用 muxError 进行错误处理 */
        netJobAdd ((FUNCPTR) muxError, (int) &pDrvCtrl -> endObj,
                (int) &pDrvCtrl -> lastError,0, 0, 0);
        }
        ++pDrvCtrl -> lastIntError;
        pDrvCtrl -> stats.rerror++;
        if (rxStat & RSTAT_OVER)
            pDrvCtrl -> stats.overruns++;
        if (rxStat & RSTAT_DIS)
            pDrvCtrl -> stats.disabled++;
        if (rxStat & RSTAT_DFR)
            pDrvCtrl -> stats.deferring++;
        }
    if (intStat & ISTAT_TXE)
        {
    if (!pDrvCtrl -> lastIntError)
        {
        END_ERR_ADD (&pDrvCtrl -> endObj, MIB2_OUT_ERRS,  +1);
        END_ERR_ADD (&pDrvCtrl -> endObj, MIB2_OUT_UCAST, -1);
```

```
        pDrvCtrl -> lastError.errCode = END_ERR_WARN;
        pDrvCtrl -> lastError.pMesg = "transmit error";

        /* 检测到发送错误,使用 muxError 进行错误处理 */
        netJobAdd ((FUNCPTR) muxError, (int) &pDrvCtrl -> endObj,
                (int) &pDrvCtrl -> lastError,0, 0, 0);
        }
        ++pDrvCtrl -> lastIntError;
        pDrvCtrl -> stats.terror++;
        if (txStat & TSTAT_ABORT)
            {
            pDrvCtrl -> stats.aborts++;
            pDrvCtrl -> stats.collisions += 16;
            }
        if (txStat & TSTAT_UNDER)
            pDrvCtrl -> stats.underruns++;
        }

        /* 发送数据 */
        if (intStat & ISTAT_PTX)
            {
            /* 数据发送后,修改统计信息 */
            END_ERR_ADD (&pDrvCtrl -> endObj, MIB2_OUT_ERRS, +1);
            if (txStat & TSTAT_CDH)
                pDrvCtrl -> stats.heartbeats++;
            if (txStat & TSTAT_OWC)
                pDrvCtrl -> stats.outofwindow++;
            if (txStat & TSTAT_PTX)
                pDrvCtrl -> stats.tnoerror++;
            }

        if (!(intStat & (ISTAT_RXE | ISTAT_TXE)))
            pDrvCtrl -> lastIntError = 0;

    /* 接收处理 */
    if (intStat & ISTAT_PRX)
```

```
    {
    pDrvCtrl -> current = rtl8019GetCurr (pDrvCtrl);
    DBG_PRINTF(("rtl8019Int: input packet (flags = %x, current = %d)\n",
       pDrvCtrl -> flags, pDrvCtrl -> current));
    if (!(pDrvCtrl -> flags & END_RECV_HANDLING_FLAG))
        {
        /* 关闭接收中断 */
        pDrvCtrl -> imask &= ~IM_PRXE;
        SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, pDrvCtrl -> imask);

        pDrvCtrl -> flags |= END_RECV_HANDLING_FLAG;

        /* 生成 rtl8019HandleRcvInt 接收处理任务 */
        (void)netJobAdd ((FUNCPTR)rtl8019HandleRcvInt, (int)pDrvCtrl,
                          0,0,0,0);
        }
    }

/* 发送完毕处理 */
if ((intStat & (ISTAT_TXE | ISTAT_PTX)) != 0)
{
pDrvCtrl -> flags &= ~END_TX_IN_PROGRESS;
DBG_PRINTF (("rtl8019Int: Tx complete, blocked = %d\n",
        (pDrvCtrl -> flags & END_TX_BLOCKED) ? 1 : 0));
if (pDrvCtrl -> flags & END_TX_BLOCKED)
    {
    /* 关闭发送中断 */
    pDrvCtrl -> imask &= ~(IM_TXEE | IM_PTXE);
    SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, pDrvCtrl -> imask);

    pDrvCtrl -> flags &= ~END_TX_BLOCKED;

    /* 如果发送被阻塞,则重新启动发送 */
    netJobAdd ((FUNCPTR)muxTxRestart, (int)&pDrvCtrl -> endObj,
          0, 0, 0, 0);
    }
}
```

```
    CACHE_PIPE_FLUSH ();

    }

LOCAL void rtl8019HandleRcvInt
    (
    RTL8019END_DEVICE * pDrvCtrl
    )
    {
    int oldLevel;
    char * pBuf;
    pBuf = NULL;
    while (pDrvCtrl -> flags & END_RECV_HANDLING_FLAG)
    {
    int      len;
    CL_BLK_ID     pClBlk;
    M_BLK_ID      pMblk;
    if (pDrvCtrl -> nextPacket == pDrvCtrl -> current)
        break;
    if (pDrvCtrl -> flags & END_OVERWRITE)
        break;

    /* 申请缓冲 */
    if (!pBuf)
        {
        pBuf = netClusterGet (pDrvCtrl -> endObj.pNetPool,
                  pDrvCtrl -> clPoolId);
        if (!pBuf)
        {
        DBG_PRINTF (("rtl8019HandleRcvInt: Out of clusters! \n"));
        break;
        }
        }

    /* 读取数据 */
    len = rtl8019PacketGet (pDrvCtrl, pBuf + pDrvCtrl -> offset);
    if (len <= 0)
```

```
    {
    DBG_PRINTF (("rtl8019HandleRcvInt: bad packet! (len = %d)\n",
        len));
    END_ERR_ADD (&pDrvCtrl -> endObj, MIB2_IN_ERRS, + 1);
    break;
    }

/* 申请一个 MBLK 和 CLBLK 结构 */
pMblk = mBlkGet (pDrvCtrl -> endObj.pNetPool,
        M_DONTWAIT, MT_DATA);
if (!pMblk)
    {
    DBG_PRINTF (("rtl8019HandleRcvInt: Out of M Blocks! \n"));
    END_ERR_ADD (&pDrvCtrl -> endObj, MIB2_IN_ERRS, + 1);
    break;
    }
pClBlk = netClBlkGet (pDrvCtrl -> endObj.pNetPool, M_DONTWAIT);
if (!pClBlk)
    {
    DBG_PRINTF (("rtl8019HandleRcvInt: Out of CL Blocks! \n"));
    netMblkFree (pDrvCtrl -> endObj.pNetPool, (M_BLK_ID)pMblk);
    break;
    }

/* 将接收数据指针与 CLBLK 关联 */
netClBlkJoin (pClBlk, pBuf, len, NULL, 0, 0, 0);

/* 将 CLBLK 指针与 MBLK 关联 */
netMblkClJoin (pMblk, pClBlk);

pMblk -> mBlkHdr.mFlags    |= M_PKTHDR;
pMblk -> mBlkHdr.mLen      = len;
pMblk -> mBlkPktHdr.len    = len;
pMblk -> mBlkHdr.mData     += pDrvCtrl -> offset;

/* 记录接收的数据包 */
END_ERR_ADD (&pDrvCtrl -> endObj, MIB2_IN_UCAST, + 1);
```

```
        /* 调用上层处理函数 */
        END_RCV_RTN_CALL (&pDrvCtrl -> endObj, pMblk);
        pBuf = NULL;
        SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, pDrvCtrl -> imask);
        }

        /* 释放未用的缓冲 */
        if (pBuf)
        netClFree (pDrvCtrl -> endObj.pNetPool, (UCHAR * ) pBuf);

        /* 重新开启接收中断 */
        oldLevel = intLock ();
        pDrvCtrl -> flags &= ~END_RECV_HANDLING_FLAG;
        pDrvCtrl -> imask |= IM_PRXE;
        SYS_OUT_CHAR (pDrvCtrl, ENE_INTMASK, pDrvCtrl -> imask);

        intUnlock (oldLevel);
        }
```

在 rtl8019HandleRcvInt 函数中，最终会通过 rtl8019PacketGet 将接收的数据读出来。rtl8019PacketGet 的处理流程与硬件的操作方法有关，可参考 RTL8019 的芯片手册。程序清单 5-47 给出了数据包的接收处理代码。

程序清单 5-47　$(BSPBASE)\rtl8019End.c　数据包的接收处理

```
LOCAL int rtl8019PacketGet
    (
    RTL8019END_DEVICE      * pDrvCtrl,
    char           * pData
    )
    {
    UINT            packetSize;
    UCHAR           uppByteCnt;
    UINT8           tempPage;
    UINT8           pageCount;
    UINT            packetLen = 0;
    RTL8019_HEADER  h;
    if (pDrvCtrl -> nextPacket == pDrvCtrl -> current)
    return (0);
```

```
rtl8019DataIn (pDrvCtrl,
      (((UINT)pDrvCtrl -> nextPacket << 8) & 0x0000ffff),
      sizeof (RTL8019_HEADER), (char * ) &h);
if (h.next > pDrvCtrl -> nextPacket)
uppByteCnt = (UCHAR) (h.next - pDrvCtrl -> nextPacket);
else
uppByteCnt = (UCHAR) ((RTL8019_PSTOP - pDrvCtrl -> nextPacket)
          + (h.next - RTL8019_PSTART));
if (h.lowByteCnt > 0xfc)
uppByteCnt -= 2;
else
uppByteCnt -= 1;
h.uppByteCnt = uppByteCnt;

/* 计算当前数据包的字节数 */
packetSize = (((UINT)h.uppByteCnt << 8) + h.lowByteCnt) - 4;
pageCount = (UCHAR) ((packetSize + 4 + sizeof (RTL8019_HEADER)
          + (ENE_PAGESIZE - 1)) / ENE_PAGESIZE);
tempPage = (UCHAR) (pDrvCtrl -> nextPacket + pageCount);
if (tempPage >= RTL8019_PSTOP)
tempPage -= (RTL8019_PSTOP - RTL8019_PSTART);
if ((h.next != tempPage) ||
(h.next < RTL8019_PSTART) || (h.next >= RTL8019_PSTOP))
{
/* 接收异常 */
pDrvCtrl -> stats.badPacket ++ ;
pDrvCtrl -> stats.rerror ++ ;

/* 重新复位硬件设备 */
if (!(pDrvCtrl -> flags & (END_OVERWRITE|END_OVERWRITE2)))
    rtl8019Config (pDrvCtrl, TRUE);
return (-1);
}

/* 丢弃小包 */
if (packetSize < 60)
{
```

```
    pDrvCtrl -> stats.shortPacket ++;
    goto doneGet;
    }

    /* 丢弃大包 */
    if (packetSize > RTL8019_BUFSIZ)
    goto doneGet;
    if (h.rstat & RSTAT_PRX)
    {
    if (h.rstat & (RSTAT_DFR | RSTAT_DIS))
        {
        pDrvCtrl -> stats.badPacket ++;
        pDrvCtrl -> stats.rerror ++;
        goto doneGet;
        }
    pDrvCtrl -> stats.rnoerror ++;
    }
    else
    {
    if (h.rstat & RSTAT_DFR)
        pDrvCtrl -> stats.jabber ++;
    pDrvCtrl -> stats.rerror ++;
    goto doneGet;
    }

    packetLen = packetSize;

    /* 将数据拷贝到临时缓冲区中 */
    rtl8019DataIn (pDrvCtrl,
        ((UINT)pDrvCtrl -> nextPacket << 8) + sizeof (RTL8019_HEADER),
        packetLen,
        pData);
    DUMP_DATA((char *)pData, packetLen );

doneGet:
    pDrvCtrl -> nextPacket = h.next;
    SYS_OUT_CHAR (pDrvCtrl, ENE_BOUND,
```

```
            (pDrvCtrl -> nextPacket != RTL8019_PSTART ?
        pDrvCtrl -> nextPacket - 1 : RTL8019_PSTOP - 1));

    return (packetLen);
}
```

5.6 文件系统

VxWorks 操作系统在文件系统与设备驱动程序之间使用一种标准的 I/O 操作接口。这使得在单个 VxWorks 操作系统中可运行多个相同或不同种类的文件系统。依据这些标准接口协议，用户可为 VxWorks 操作系统编写用户自己的文件系统，并可将文件系统和设备驱动程序自由组合。

常用的文件系统包括：TSFS、dosFS 和 TrueFFS 等。

5.6.1 TSFS

目标服务器文件系统(TSFS)主要用于系统的开发和诊断，是 VxWorks 操作系统中功能最全的文件系统，但它所管理的文件存放于主机系统中。

TSFS 文件系统具有用于远程文件访问(netDrv)的网络驱动程序的所有 I/O 操作特点，而且不占用任何目标机的资源(除了目标系统和主机上目标服务器之间通信的资源)。TSFS 文件系统使用 WDB 驱动程序从 VxWorks 的 I/O 系统向目标服务器传送请求。目标服务器使用主机文件系统读取并执行这些请求。当用 TSFS 打开一个文件时，被打开的文件事实上就在主机上。可先使用 open()调用获得文件标识号，然后使用该文件标识号进行 read()和 write()的调用，这些操作实际上就是打开主机文件进行读/写。

TSFS 具有 netDrv 提供的所有 I/O 特征，无需已被配置来支持在目标机和目标服务器之间通信的资源之上的任何目标机资源。在不拷贝整个文件到目标机的情况下，自由地访问主机文件，从一个虚拟文件资源到装载一个目标模块都是允许的。

(1) 配置 TSFS

要想在目标系统中使用 TSFS，需要完成一些配置工作：

首先 TSFS 必须包括在 VxWorks 映像中。这通过在 config.h 中定义 INCLUDE_WDB_TSFS 来实现，编译的目标代码启动后会创建一个新的文件系统条目"/tgtsvr"。同时在主机上目标服务器必须被配置，包括指派主机上一个根目录给 TSFS。例如：可以设置 TSFS 根目录到主机的任何一个目录下。

(2) 文件的使用

从目标机打开主机上的文件与使用本地文件没有任何区别，通过常规的 open()函数即可。所有 I/O 请求(包括 open())都可以是同步的；直到操作完成。所提供的流控制在控制台 VIO 操作中无效。另外，这里无须通过 WTX 协议发送请求来联系 VIO 通道和一个特殊的主机文件。

考虑一个 read()调用。驱动程序传输文件的 ID(先前通过 open()调用确定的)、接收文件数据缓冲区的地址，以及指定的对目标服务器的读取长度；目标服务器在主机上声明同等作用的 read()调用，并传输数据读到目标机程序。read()的返回值和任何可能产生的错误被传递给目标机，以便文件在每一种方式下总是显示为本地文件。

(3) 套接字支持

TSFS 套接字以一种类似于其他 TSFS 文件的方法被操作，使用 open()、close()、read()、write()和 ioctl()命令。可以使用如下文件名的形式打开 TSFS 套接字："TCP:hostIP:port"和"TCP:hostname:port"。

信号量和许可变元被忽视。下面的例子说明怎样使用这些文件名：

```
fd = open("/tgtsvr/TCP:server:6164",0,0)
fd = open("/tgtsvr/TCP:150.50.50.50:6164",0,0)
```

open()调用的结果将在主机上打开一个 TCP 套接字，并且连接到指定主机(主机名)上的目标服务器。所组成的套接字是非阻塞的，可以使用 read()和 write()来读/写 TSFS 套接字。因为套接字是非阻塞的，因此如果从套接字未读到有效的数据，则 read()调用就会立刻返回一个错误和适当的错误代码。指向 TSFS 套接字的 ioctl 见 VxWorks 参考手册。套接字的配置允许 WindView 使用套接字工具，而不必请求 sockLib 和在目标机上的网络模块。

(4) 错误处理

错误可能发生在 TSFS 内不同的地方，并且都报告给目标机上最初的调用者(伴随一个适当的错误代码)。返回的错误代码与主机上的 VxWorks 错误号匹配。如果一个 WDB 错误发生，则返回一个 WDB 错误信息而非 VxWorks 错误号。

(5) 安全考虑

虽然 TSFS 与 netDrv 有许多共同之处，但就安全性而言是不同的。TSFS 中主机文件操作是为了方便用户启动目标服务器。作为引导参数，目标机的用户名没有作用。事实上，没有一个引导参数会对 TSFS 的访问特权有影响。

在这个环境下，用户几乎并不清楚 TSFS 的特权限制，因为用户 ID 和启动目标服务器的主机在每次调用时可能不同。无论如何，联系到支持 TSFS 目标服务器的任何 Tornado 工具都有与启动目标服务器的用户相同的文件访问权利。因此，默认情况下，当 TSFS 启动时目标服务器是被锁住的。

加入目标服务器启动程序，使用 TSFS 来控制访问主机文件的目标机的选项，其中包括：

① -R：设置 TSFS 的根目录。例如：指定 -R /tftpboot，对被目标服务器接收的所有 TSFS 文件名预先进行转换，例如将/tgtsvr/etc/passwd 转换成/tftpboot/etc/passwd。如果-R 未被指定，则 TSFS 不被激活，来自目标机的 TSFS 请求将不成功。在未指定-R 的情况下，重新启动目标服务器，则不可使用 TSFS。

② -RW：产生 TSFS 读/写。目标服务器将这个选项解释为审核修改操作（包括文件创建、删除或写）。如果-RW 未被指定，则默认为只读，不允许修改文件。

注意：若指定-RW，则目标服务器上的相应文件会被锁定（保留给启动它的用户）。目标服务器拥有者可以解锁，附加于其上的任何 Tornado 工具与目标服务器拥有者拥有同样的文件访问许可权。在 Unix 上启动的目标服务器会自动地被启动它的用户拥有；在 Windows 上启动的目标服务器会被环境变量 WIND_UID 指定的用户拥有，所以通常都需在 Windows 的环境变量中正确设置 WIND_UID。

5.6.2 dosFS

dosFS 是一种与 MS－DOS 兼容的文件系统，它能满足实时应用的多种要求。其主要特点如下：

- 支持层次化的文件和目录结构，能够在一个磁盘上建立一定数目的文件，并进行有效的管理；
- 可将每个文件指定为连续存储或非连续存储；
- 广泛兼容各种可存储和可检索媒体（如软盘和硬盘）；
- 可从 dosFS 文件系统中启动 VxWorks 操作系统；
- 支持 VFAT（Microsoft 公司的 VFAT 长文件名格式）和 VXLONGS（VxWorks 特有的长文件名格式）的目录结构；
- 支持 FAT12、FAT16 和 FAT32 文件分配表格式。

为了使用 dosFS 文件系统，必须首先在启动的内核映像中配置相关的组件，再建立该文件系统。步骤如下：

(1) 配置系统内核

利用 dosFS 文件系统、CBIO 和块存取设备组件配置用户系统内核，使系统支持 dosFS 文件系统。需要的组件包括：

① 必备的组件：INCLUDE_DOSFS_MAIN、INCLUDE_DOSFS_FAT 和 INCLUDE_CBIO。系统内核必须包括组件 INCLUDE_DOSFS_DIRVFAT 和 INCLUDE_DOSFS_DIR_FIXED 其中的一个或全部。除此以外，用户还须加入支持块存取设备的组件，如 INCLUDE_SCSI 或 INCLUDE_ATA 组件。

② 可选组件：INCLUDE_DOS_FS、INCLUDE_DOSFS_FMT、INCLUDE_DOSFS_CHKDSK、INCLUDE_DISK_UTIL 以及 INCLUDE_TAR。

③ 支持 CBIO 的可选组件：INCLUDE_DISK_CACHE、INCLUDE_DISK_PART 和 INCLUDE_RAM_DISK。

(2) 初始化 dosFS 文件系统

在用户执行任何操作之前，必须先初始化 dosFS 文件系统函数库 dosFsLib。如果系统中包括了必备的组件，则该过程是自动执行的。

在初始化文件系统过程中调用 iosDrvInstall()函数将驱动程序装入 I/O 系统的驱动程序表中。分配给 dosFS 文件系统的驱动程序的个数记录在一个全局变量 dosFsDrvNum 中。驱动程序表中指定了使用 dosFS 文件系统的设备执行文件操作的入口地址。

(3) 创建块存取设备

创建一个块存取设备或一个 CBIO 接口设备(rawDiskCbio)。在这个步骤中，创建一个或多个块存取设备，用户调用设备驱动程序中的功能函数创建相应的设备。所用的功能函数名称的格式是 xxxDevCreate()。其中 xxx 表示设备驱动程序的类型，如 scsiBlkDevCreate()和 ataDevCreate()。

设备驱动程序中的功能函数返回一个指向块存取设备描述字结构 BLK_DEV 的指针。该结构描述了设备的物理属性，并指定了设备驱动程序中能够在所使用的文件系统中执行的功能函数。

(4) 创建磁盘高速缓存

该步骤可选。如果用户在系统内核中加入了 INCLUDE_DISK_CACHE 组件，则可通过调用 dcacheDevCreate()函数为每个块存取设备创建一个磁盘高速缓冲区。磁盘高速缓冲区可以减小在非随机存取存储器上花费时间用于定位数据所造成的影响，但它并不适用于 RAM 存储盘和 TrueFFS 文件系统设备。

(5) 创建可用的磁盘分区

该步骤可选。如果用户在系统内核中加入了 INCLUDE_DISK_PART 组件，则可通过调用 usrFdiskPartCreate()和 dpartDevCreate()函数在磁盘上创建分区，并在分区中安装磁盘卷。程序清单 5-48 给出了一个具体例子。

程序清单 5-48　磁盘分区的初始化代码

```
STATUS usrPartDiskFsInit
    (
    void * blkDevID             /* CBIO_DEV_ID 或 BLK_DEV 指针 */
    )
    {
    const  char * devNames[] = { "/sda0a", "/sda0b" };
```

```
int      dcacheSize = 0x10000;
CBIO_DEV_ID cbio,cbio1;

/* 首先创建磁盘高速缓冲区 */
if((cbio = dcacheDevCreate(blkDevID, NULL, dcacheSize, "/sd0")) == NULL)
    return ERROR;

/* 创建分区 */
if((usrFdiskPartCreate(cbio, 3, 50, 0)) == ERROR) return ERROR;

/* 使用 FDISK 创建两个分区 */
if((cbio1 = dpartDevCreate( cbio, 2, usrFdiskPartRead)) == NULL)
    return ERROR;

/* 创建第一个文件系统 */
if(dosFsDevCreate(devNames[0], dpartPartGet(cbio1, 0), 8, 0) == ERROR)
    return ERROR;

/* 格式化第一个分区 */
if(dosFsVolFormat(devNames[0], 2, 0) == ERROR) return ERROR;

/* 创建第二个文件系统 */
if(dosFsDevCreate(devNames[1], dpartPartGet(cbio1, 0), 4, NONE) == ERROR)
    return ERROR;

/* 格式化第二个分区 */
if(dosFsVolFormat(devNames[1], 2, 0) == ERROR) return ERROR;

return OK;

}
```

(6) 创建 dosFS 文件系统设备

创建所需的 dosFS 文件系统设备。无论用户是否使用一个预先格式化的磁盘，都可以安全地创建 dosFS 文件系统设备。格式化磁盘：如果用户使用了一个未预先格式化的磁盘，则须格式化卷。

用户通过调用 dosFsDevCreate()函数（该函数内部调用 iosDrvAdd()函数）创建 dosFS 文

件系统设备。本步骤只是将设备简单地装入I/O系统中,并不执行任何I/O操作,因此未安装磁盘。直到系统执行第一次I/O操作,磁盘才被安装。

(7) 格式化磁盘

如果用户使用一个未格式化的磁盘,那么有两种方法可将其格式化。

① 直接调用dosFsVolFormat()函数。该函数需要指定文件分配表(FAT)格式和目录格式两个参数。

② 调用ioctl()函数实现FIODISKINIT功能。该功能将调用dosFsLib文件中的格式化程序。这种方法使用默认的磁盘格式和参数。

VxWorks支持的文件分配表有3种:FAT12、FAT16和FAT32。它们之间的区别在于磁盘簇号的管理方式。其中:FAT12的簇号采用12位数据表示,一般适用于小容量的磁盘;FAT16的簇号采用16位数据表示,最多可管理65 536个簇,一般情况下每个磁盘的容量要求小于8 GB,每个卷的容量小于2 GB;FAT32的簇号采用32位数据表示,可以管理容量更大的磁盘和分区。因此,对于文件系统的分配表格式,还要根据磁盘的容量来进行选择。

VxWorks支持的目录格式也有3种:VFAT(Microsoft公司的长文件名格式)、短文件名(8.3格式)以及VxWorks自己的长文件名格式(VxLong)。其中:前两种文件名格式不区分大小写;而VxWorks自己的长文件名格式则须区分大小写。VFAT格式的文件名最长为254字节,而且还兼容微软的短文件名格式;VxLong格式的文件名最长为40字节,每个文件允许的容量为4 GB。

(8) 检查磁盘完整性

用户可以调用dosFsChkDsk()函数来检测磁盘中卷的完整性,该检测过程是可选的。对于大容量磁盘,实现磁盘检测功能很耗费时间。用户在dosFsChkDsk()函数中设置的参数决定系统是否自动实现磁盘检测功能。

(9) 安装磁盘

通常,用户在对磁盘上的文件或目录第一次调用open()或create()函数时,系统将自动安装磁盘中的卷。

5.6.3 TrueFFS

TrueFFS是Tornado II开发环境中集成的一个快速闪存文件系统。它使用闪存设备来实现快速、可靠的物理存储,通过模拟VxWorks文件系统下的硬盘驱动器来屏蔽Flash操作的具体细节;且可使用标准的文件系统接口来产生和操作该文件系统,从而使得在闪存设备上执行读/写操作就像在DOS文件系统设备上操作文件一样简单。

尽管Flash存储器不可能适用于所有嵌入式系统,但由于其具有体积小,耗电少,非易失存储的特性,因此在许多环境下尤其在移动设备和手持设备领域中成为了理想的选择。Flash

存储器也有一些自身的缺点。首先,它在写之前必须要进行擦除操作,而且不能逐字节地擦除,只能以一个扇区、一个块或者整片的方式进行擦除。它的写和擦除都需较复杂的步骤才能完成,因而降低了其易使用性。另外,Flash 存储器最大的一个缺点就是寿命有限,可擦除的次数因不同厂商而有所不同,一般都在1万~10万次。一个运行在 Flash 存储器上、性能良好的块设备应能针对可移动的媒体(如 Flash 卡)处理各种复杂的情况。例如:当驱动程序正在进行写操作时,用户却把 Flash 卡抽出来了,可能会造成灾难性的后果。不过 TrueFFS 在设计时已经仔细考虑了 Flash 存储器的各种特性,以及掉电和用户非法拔卡等操作带来的后果。

TrueFFS 由1个核心层和3个功能层组成。这3个功能层分别是转换层(Translation Layer)、MTD 层(MTD Layer)和 Socket 层(Socket Layer),如图5-23所示。

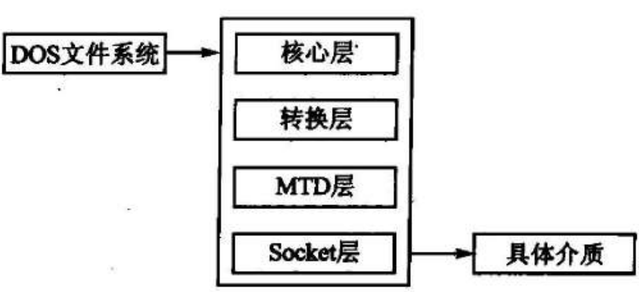


图5-23 TrueFFS 文件系统的层次结构

核心层:主要功能是将其他各层连接起来,以及将工作引向其他层并处理全局事务(如后台处理、碎片收集、计时器和其他系统资源的管理等)。

转换层:主要实现 TrueFFS 和 dosFS 之间的高级交互功能;也包含控制 Flash 映射到块、碎片回收和数据完整性所需的智能化处理功能。目前有3种不同的转换层模块可供选择。具体选择哪一种,取决于所用的 Flash 介质采用的是 NOR-based、NAND-based,还是 SSFDC-based 技术。

Socket 层:提供 TrueFFS 和板卡硬件(如 Flash 卡)的接口服务。其名字来源于用户可插入 Flash 卡的物理插槽。该层用来向系统注册 Socket 设备,检测设备拔插以及硬件写保护等。后面将详细讲解其功能。

MTD(Memory Technology Drivers)层:主要功能是实现对具体 Flash 进行读、写、擦除和 ID 识别等操作,并设置与 Flash 密切相关的一些参数。TrueFFS 已包含了支持 Intel、AMD 以及 Samsung 公司部分 Flash 芯片的 MTD 层驱动。新的芯片需要新的 MTD 支持,可以使用一个标准的接口来添加这些驱动。

为了使用 TrueFFS 文件系统,还须进行内核配置。VxWorks 系统本身已支持多种 Flash 芯片或 Flash 卡;如果使用的具体存储介质不被 VxWorks 系统支持,那么还须进行 Flash 的驱动编写。下面详细介绍具体过程。

1. 配置 TrueFFS

首先需要选择 MTD 组件,当然如果没有系统提供的 MTD 组件支持,也可自己编写一个 MTD 组件。

在系统目录 $(WIND_BASE)/target/src/drv/tffs 下包含以下 MTD 组件的源代码:

- 与 Intel、AMD、Fujitsu 以及 Sharp 公司提供的设备兼容的 MTD;
- 适用于遵守 CFI 的设备的 MTD。

表 5-11 为所支持设备的列表。

表 5-11 TrueFFS 支持的设备

设备名称	描 述
INCLUDE_MTD_CFISCS	CFI/SCS 设备
INCLUDE_MTD_CFIAMD	遵守 CFI 的 AMD 和 Fujitsu 设备
INCLUDE_MTD_128F016	Intel28F016
INCLUDE_MTD_128F008	Intel28F008
INCLUDE_MTD_AMD	AMD,Fujitsu 29F040/80/16 8 位设备
INCLUDE_MTD_WAMD	AMD,Fujitsu 29F040/80/16 16 位设备
ICLUDE MTD_INTEL28F008_BAJA	Heurikon Baja 4000 上的 Intel28F008

系统提供的这些 MTD 组件由于其适应性的需要,往往编写得较复杂。可以根据实际的存储介质,参考这些组件来编写一个适合自己,且十分精简的 MTD 组件。

在组件描述文件中定义的 MTD(即通过项目设备包含的 MTD)通常需要转换层。系统支持的转换层组件有:

- INCLUDE_TL_FTL: 支持 NOR 闪存设备;
- INCLUDE_TL_SSFDC: 支持遵守 SSFDC 规范的 NAND 设备。

组件描述文件一般都会说明转换层和 MTD 之间的相关性。因此,进行配置时,不必选择转换层,编译过程会完成这项工作。

最后还须进行内核的配置。用 TrueFFS 配置的 VxWorks 系统包括:

(1) 完全支持 dosFS 文件系统的配置

如果没有 VxWorks 兼容的文件系统 MS-DOS,则 TrueFFS 的系统配置是没有意义的。因此 dosFS 的支持以及所有相关组件都要被包含在 TrueFFS 系统中。此外,有一些其他文件系统组件不是必需的,但在某些情况下仍然很有用。这些组件增加了对文件系统一些基本功能(如 ls、cd 和 copy 等命令)的支持。

(2) 一个核心 TrueFFS 组件

所有系统都必须包含 TrueFFS 核心组件,通过定义 INCLUDE_TFFS 实现。在启动时,为了初始化该模块,通过定义该组件能保证初始化顺序正确,也能保证 Socket 驱动程序被包含在项目中。

(3) 3 个 TrueFFS 层中的相应软件模块

在 config.h 中的具体配置方法参见程序清单 5-49。

程序清单 5-49　TrueFFS 的配置方法

```
/* TrueFFS 的配置选项 */
#define        INCLUDE_TFFS
#ifdef         INCLUDE_TFFS
#define        INCLUDE_TL_FTL
#define        INCLUDE_SHOW_ROUTINES
#define        INCLUDE_TFFS_SHOW
#define        INCLUDE_DOSFS
#define        INCLUDE_MTD_CFISCS
#endif      /* INCLUDE_TFFS */
```

2. 编写 Socket 驱动

在 PCMCIASocket 启动服务后，TrueFFS 中的 Socket 驱动程序被格式化。尽管如此，它必须提供以下内容：

- 控制 Socket 电源(Socket 为 PCMCIA、RFA 或其他类型)；
- 建立内存窗口环境的标准；
- 支持板卡变更识别。

Socket 注册函数登记了这些功能部件的处理程序，这些 Socket 成员函数包括：rfaCardDetected、rfaVccOn、rfaVccOff、rfaVppOn、rfaVppOff、rfaInitSocket、rfaSetWindow、rfaSetMappingContext、rfaGetAndClearCardChangeIndicator 和 rfaWriteProtected。

程序清单 5-50 为一个 TFFS 初始化和成员登记函数。

程序清单 5-50　TrueFFS 文件系统的初始化

```
LOCAL void sysTffsInit (void)
    {
    UINT32 ix = 0;
    UINT32 iy = 1;
    UINT32 iz = 2;
    int oldTick;
    oldTick = tickGet();
    while (oldTick == tickGet())            /* 等待下一次时钟中断 */
    ;
    oldTick = tickGet();
    while (oldTick == tickGet())            /* 循环一个时钟周期 */
    {
```

```
    iy = KILL_TIME_FUNC;                /* 消耗时间 */
    ix++;
    }
    sysTffsMsecLoopCount = ix * sysClkRateGet() / 1000;
    rfaRegister ();                     /* 注册 rfa 接口 */

    }

LOCAL void rfaRegister (void)
    {
    FLSocket vol = flSocketOf (noOfDrives);
    tffsSocket[noOfDrives] =        "RFA";
    vol.window.baseAddress =        FLASH_BASE_ADRS >> 12;
    vol.cardDetected =              rfaCardDetected;
    vol.VccOn =                     rfaVccOn;
    vol.VccOff =                    rfaVccOff;
#ifdef SOCKET_12_VOLTS
    vol.VppOn =                     rfaVppOn;
    vol.VppOff =                    rfaVppOff;
#endif
    vol.initSocket =                rfaInitSocket;
    vol.setWindow =                 rfaSetWindow;
    vol.setMappingContext =         rfaSetMappingContext;
    vol.getAndClearCardChangeIndicator = rfaGetAndClearCardChangeIndicator;
    vol.writeProtected =            rfaWriteProtected;
    noOfDrives++;
    }
```

rfaCardDetected：该函数会报告在 PCMCIA 插槽中是否有与之相关的闪存板卡。对于不可删除的媒质，函数通常会返回 TRUE。实际上，支持 Tornado 的 TrueFFS 每 100 ms 就会调用该函数来检查闪存媒质是否存在。如果函数返回 FALSE，则 TrueFFS 就会将 cardChanged 设置为 TRUE。见程序清单 5－51。

程序清单 5－51　TrueFFS 文件系统卡检测处理程序

```
LOCAL FLBoolean rfaCardDetected
    (
    FLSocket vol            /* 指向对应的设备 */
```

```
    )
    {
    return (TRUE);
    }

LOCAL void rfaVccOn
    (
    FLSocket vol
    )
    {
    rfaWriteEnable ();
    }

LOCAL void rfaVccOff
    (
    FLSocket vol
    )
    {
    rfaWriteProtect ();
    }

#ifdef SOCKET_12_VOLTS
LOCAL FLStatus rfaVppOn
    (
    FLSocket vol
    )
    {
    return (flOK);
    }
LOCAL void rfaVppOff
    (
    FLSocket vol
    )
    {
    }
#endif      /* SOCKET_12_VOLTS */
```

```
LOCAL FLStatus rfaInitSocket
    (
    FLSocket vol
    )
    {
    rfaWriteEnable ();
    vol.cardChanged = FALSE;

    /* 通知内存管理单元使能相应的内存区域,并将这一位置映射到 0 地址 */
    rfaSetWindow (&vol);
    return (flOK);
    }

LOCAL void rfaSetWindow
    (
    FLSocket vol
    )
    {
    /* 基地址应该是 4 KB 对齐的 */
    vol.window.baseAddress = FLASH_BASE_ADRS >> 12;
    flSetWindowSize (&vol, FLASH_SIZE >> 12);
    }

LOCAL void rfaSetMappingContext
    (
    FLSocket vol,
    unsigned page
    )
    {
    }

LOCAL FLBoolean rfaGetAndClearCardChangeIndicator
    (
    FLSocket vol
    )
    {
    return (FALSE);
```

```
    }

LOCAL FLBoolean rfaWriteProtected
    (
    FLSocket vol
    )
    {
    return (FALSE);
    }

LOCAL void rfaWriteProtect(void)
    {
    }

LOCAL void  rfaWriteEnable (void)
    {
    }

long int flFitInSocketWindow
    (
    long int chipSize,
    int      interleaving,
    long int windowSize
    )
    {
    if (chipSize * interleaving > windowSize)
        {
        int  roundedSizeBits;
        chipSize = windowSize / interleaving;
        for (roundedSizeBits = 0; (0x1L << roundedSizeBits) <= chipSize;
             roundedSizeBits ++ )
        ;
        chipSize = (0x1L << (roundedSizeBits - 1));
        }

    return (chipSize);
    }
```

rfaVccOn：TrueFFS 可调用该函数开启操作时需要的电源。对于闪存硬件，工作电压通常为 5 V 或 3.3 V。当媒质空闲时，TrueFFS 会在操作结束后关断该电源的供给；而在访问闪存之前，TrueFFS 会使用该函数重新开启电源。

rfaVccOff：TrueFFS 可调用该函数为闪存硬件关断工作电平。当媒质空闲时，TrueFFS 会关断电源。

rfaVppOn：与 rfaVccOn 类似，只不过它管理的是写入数据时的电源。

rfaVppOff：当写入数据结束时，会调用该函数关闭编程电源。

rfaInitSocket：TrueFFS 会在访问 Socket 之前调用该函数。TrueFFS 使用该函数在访问 Socket 之前进行必要的初始化，尤其在 Socket 注册时初始化无法进行的情况下。例如：在 Socket 注册时未检查硬件，或者闪存媒质是可删除的，则该函数会检查闪存媒质并响应；如果硬件发生变化，则会设置 cardDetected 为 FALSE。

rfaSetWindow：TrueFFS 使用 window.base 来存储闪存上内存窗口的基地址；使用 window.size 来存储内存窗口的尺寸。TrueFFS 假定它是惟一能够访问内存窗口的，也就是说，在 TrueFFS 设置了这些窗口特征之后，rfaSetWindow 不允许应用程序直接改变它们，否则就会导致冲突。然而映射寄存器是个例外，因为 TrueFFS 通常在它访问闪存时重建这个映射寄存器，因而允许应用程序变换这个窗口。

rfaSetMappiUgCoUtext：TrueFFS 调用该函数设置窗口映射寄存器。该函数通过适当设置映射寄存器为某一数值来执行"滑动"。因此，该函数只有在使用滑动窗机制检查闪存的环境中(PCMCIA)才有意义。在 PCMCIA 插槽中闪存板卡也能使用该函数来访问/设置一个映射寄存器；该寄存器将有效的闪存地址移至主机的内存窗口。映射过程获取一个"板卡地址"(闪存的偏移量)，并从中产生真实的地址。如果偏移量超出了闪存的长度，则也可修改该地址，使其位于闪存的起始处。

rfaGetAndClearChangeIndicator：该函数读取硬件板卡变化状态，并将其清除。它是检查媒体变化事件的基础。如果没有这样的硬件性能，则向该函数返回 FALSE。

rfaWriteProtected：TrueFFS 调用该函数来获取媒质写保护转换的当前状态。该函数会返回媒质的写保护状态，返回 FALSE 表示可写状态。

3. 设备格式化

TrueFFS 文件系统在使用之前需要格式化。首先在 tffsDevFormatParams 结构中指定一些格式化的信息，如偏移地址和保留大小等；然后调用 tffsDevFormat 即可。参见下程序清单 5-52。

程序清单 5-52 TrueFFS 的格式化处理

```
STATUS sysTffsFormat (void)
    {
```

```
STATUS status;
tffsDevFormatParams params =
{
{0x001000001, 99, 1, 0x200001, NULL, {0,0,0,0}, NULL, 2, 0, NULL},
FTL_FORMAT_IF_NEEDED
};

status = tffsDevFormat (0, (int)&params);
return (status);
}
```

5.7 驱动程序中的数据一致性

编写设备驱动程序时应注意 Cache 一致性的问题。在开发板卡驱动程序时常常会对分配的内存所带的 Cache 产生困惑：何时应该分配带 Cache 的内存；使用 Cache 有什么好处；应注意哪些问题等。因此，在这里简单介绍一下 Cache 的有关问题。

首先 Cache 的最大的好处是，解决 CPU 与内存之间的瓶颈。现在使用的 SDRAM 的速度一般都为 3～10 ns，而 CPU 的速度已达几百兆，甚至几 GHz。显然这样的速度差异会导致 CPU 在存取内存时经常要等待数据在内存上的操作，这将致命地影响系统的性能，所以人们引进了 Cache 机制。因为成本及物理上的限制，Cache 通常都不大，只有 256 KB 或 512 KB，但速度很快，一般位于 CPU 与内存之间。它有一系列的预测算法，以保证下次 CPU 存取内存的数据在 Cache 内的高机率。这样 CPU 在存取数据时，如果数据在 Cache 内，则可直接对 Cache 进行存取，而无需与缓慢的内存打交道。

为板卡开发的驱动必须保证 Cache 的一致性，即 Cache 中的数据必须与内存中的保持同步；在使用异步方式读取内存时有可能产生内存与 Cache 失步。数据 Cache 可以减少存取内存的次数，从而提高系统的性能，它通常有两种方式：writethrough 和 copyback。writethrough 是指在写内存的同时也往 Cache 中写，可保证 Cache 在输出上一致，但不能保证在输入时一致；而 copyback 则仅将数据写到 Cache 中，它保证数据在输入和输出时都一致。

当 CPU 写一个 DMA 设备上的内存时，数据首先被写到 Cache 中；当 DMA 设备从 RAM 中传输数据时，不能保证内存中的数据与 Cache 中的一致。这样输出到设备的数据可能不是最新的；最新的数据应该在 Cache 中。可在数据传输到 DMA 设备之前将 Cache 数据刷新到内存中。

当 CPU 读一个 DMA 设备上的内存时，数据读可以来自 Cache 的高速缓冲而不是从设备

传输到内存的数据，也可以通过对Cache高速缓冲作屏蔽标记来解决读出的数据来自内存而非Cache的问题。

驱动程序可以通过分配无Cache缓冲(使用cacheDmaMalloc())，刷新以及使Cache无效来解决CPU与设备之间数据传输的Cache一致性问题。分配无Cache缓冲常用于分配静态缓冲，它需要MMU的支持。无Cache缓冲频繁地被动态分配和释放，将导致大量的内存被标记为无Cache状态。一个折中的方法是人为地刷新(使用cacheFlush()函数)和屏蔽Cache(使用cacheInvalidate())。

注意：在设备读数据之前用cacheFlush()保证数据一致；而在设备写入内存后，则用cacheInvalidate()来保证Cache与内存数据的一致。

第6章

VxWorks 应用程序的编写

6.1 VxWorks 应用程序调试环境的建立

在介绍应用程序编写的有关知识之前，需要先熟悉 VxWorks 应用程序的调试方法。后面的一些例子程序将采用这种方法进行调试。

VxWorks 应用程序的调试方法有两种：一种是与内核集成在一起进行调试；另一种是将应用程序和内核分开来进行调试，在应用程序无误后再集成到内核中。下面介绍第二种调试方法，该方法更为方便。

要将应用程序和内核分开来进行开发调试，首先须有一个可稳定运行的 VxWorks 内核。这里可使用前面做好的 BSP 来生成内核。先将可执行代码下载或烧写到目标系统中，使 VxWorks 内核运行起来；然后就是配置 Tornado。调试之前必须保证目标机已上电，并通过网络或串口与宿主机相连；下载应用程序的目标码之前还须对 Tornado 进行一些配置。

(1) 配置目标服务器

在目标系统运行起来后，即可通过选择 Tools→Target Server→Config 来新建一个配置，如图 6-1 所示。如果通过串口连接，则选择 wbdserial；如果是网络连接，则选择 wbdrpc。

单击 Launch 按钮启动该服务。这时在 Windows 任务栏将出现一个像靶一样的图标，如图 6-2 所示。

(2) 选择目标机

在目标系统运行起来，且已对 Target Server 进行了正确配置并启动了该配置之后，即可在 Tornado Launch 工具栏的第一栏选择目标机，如图 6-3 所示。

(3) 下载应用程序

在集成开发环境的工程管理窗口中选择应用程序的 Object Modules，并在二进制输出文件 *.out 上右击并选择下载 Download 'app1.out'，如图 6-4 所示。

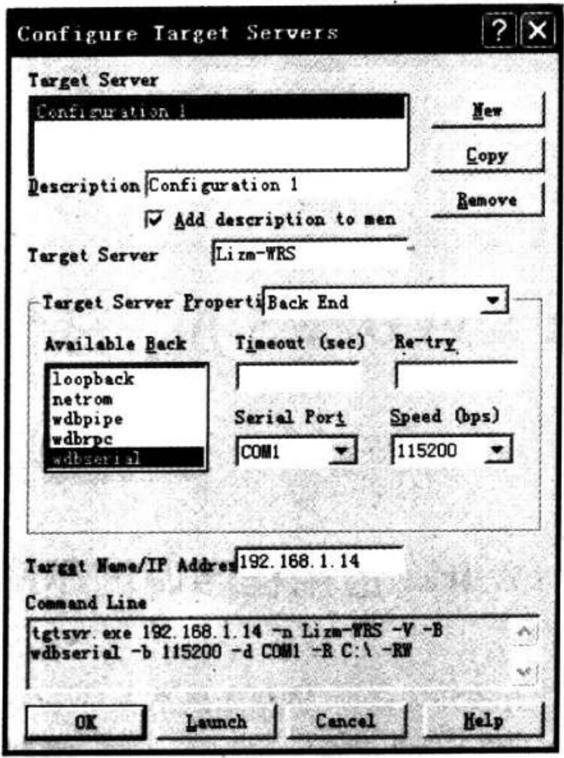


图 6-1　通信协议选择

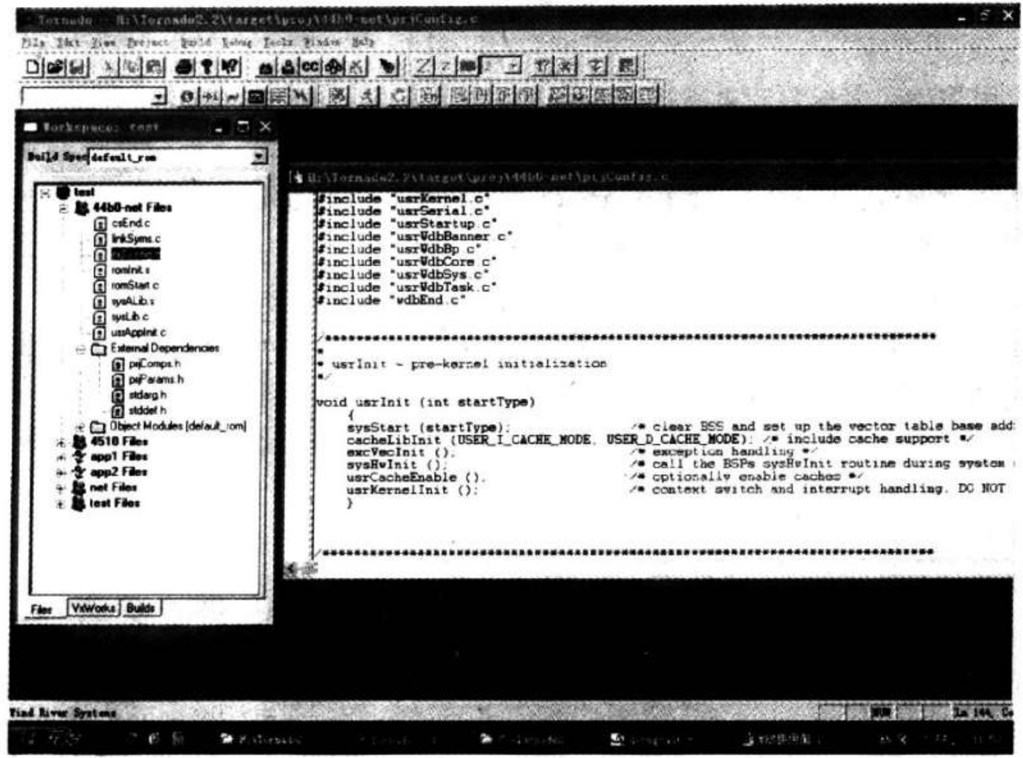


图 6-2　启动目标调试器

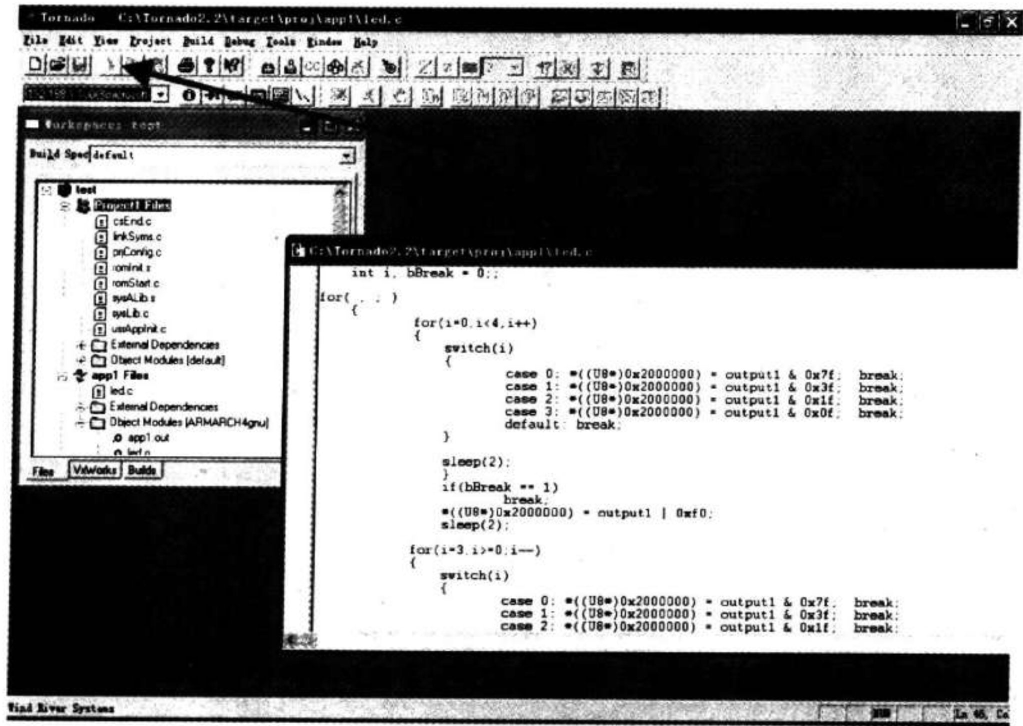


图 6-3　选择目标机

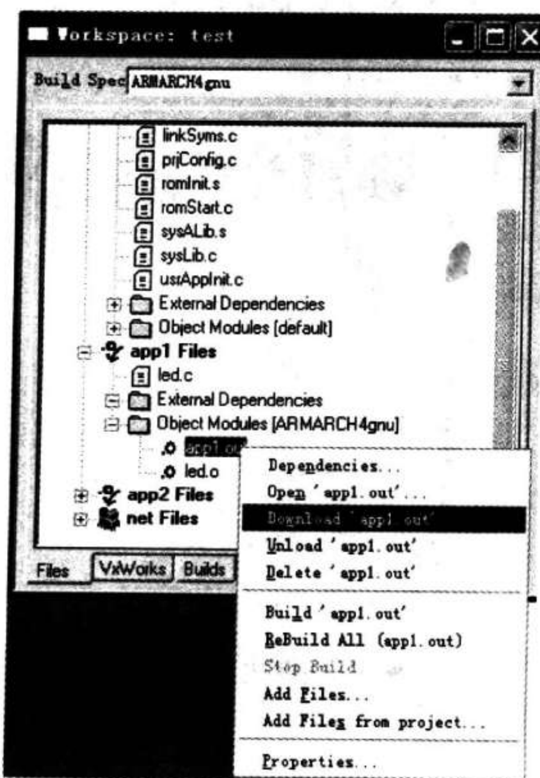


图 6-4　下载应用程序

(4) 打开调试窗口

如图 6－5 所示，启动目标调试器后，即可打开调试窗口。

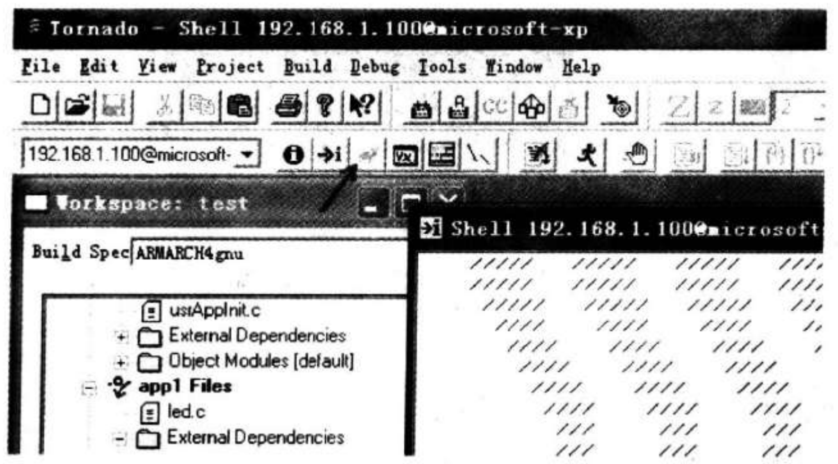


图 6－5　启动目标调试器

(5) 在源代码中设置断点并运行

可使用常用的调试手段来设置断点，运行程序等，如图 6－6 所示。

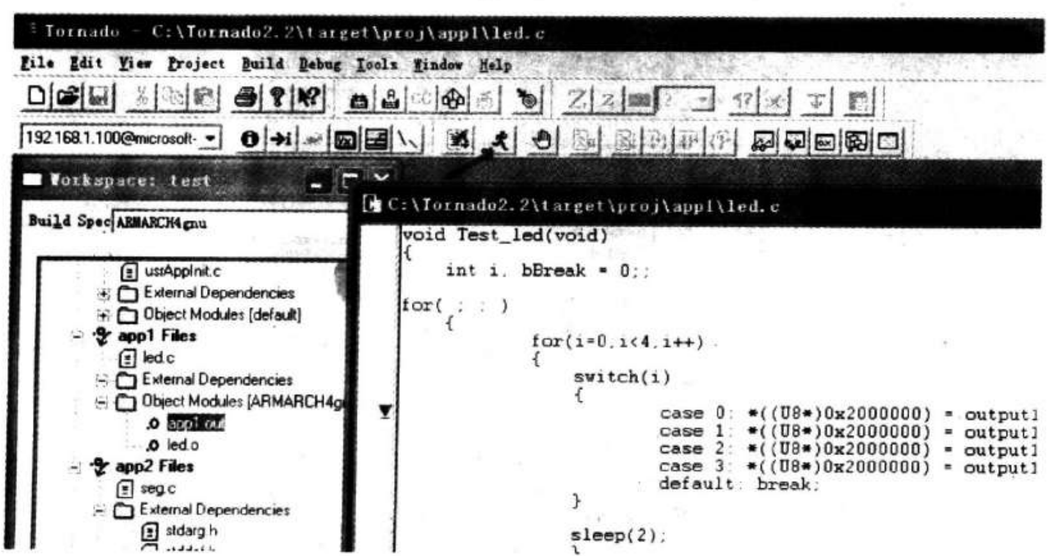


图 6－6　源代码调试窗口

输入函数名，并输入调试参数；然后选择单步调试等操作，如图 6－7 所示。

(6) 通过 Shell 运行程序

打开 Shell 窗口，并输入相应的运行命令，如图 6－8 所示。

Run Task

Task　Test_led

Arguments:

Break at entry po

OK　Cancel　Help

图 6-7　调试函数

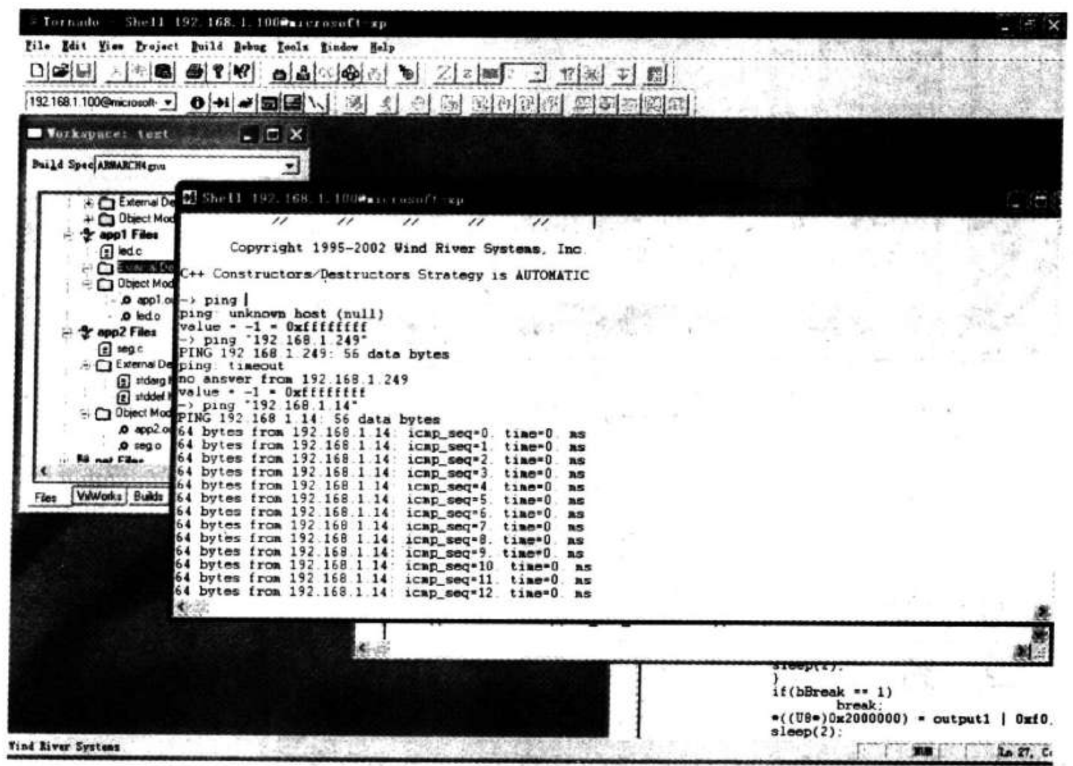


图 6-8　在 Shell 中运行程序

图 6-8 描述了在 Shell 中运行 ping 测试的情况。除了执行 Shell 模块中的命令外，还可运行系统中其他一些例程。

在调试环境建立好之后，即可使用一些常规手段（如单步运行、检查变量和检查内存等）调试编写的程序；另外也可借助于前面提到的调试工具来分析系统的一些特性，从而改进系统的性能。Browser 工具显示的目标机模块信息，如图 6-9 所示。

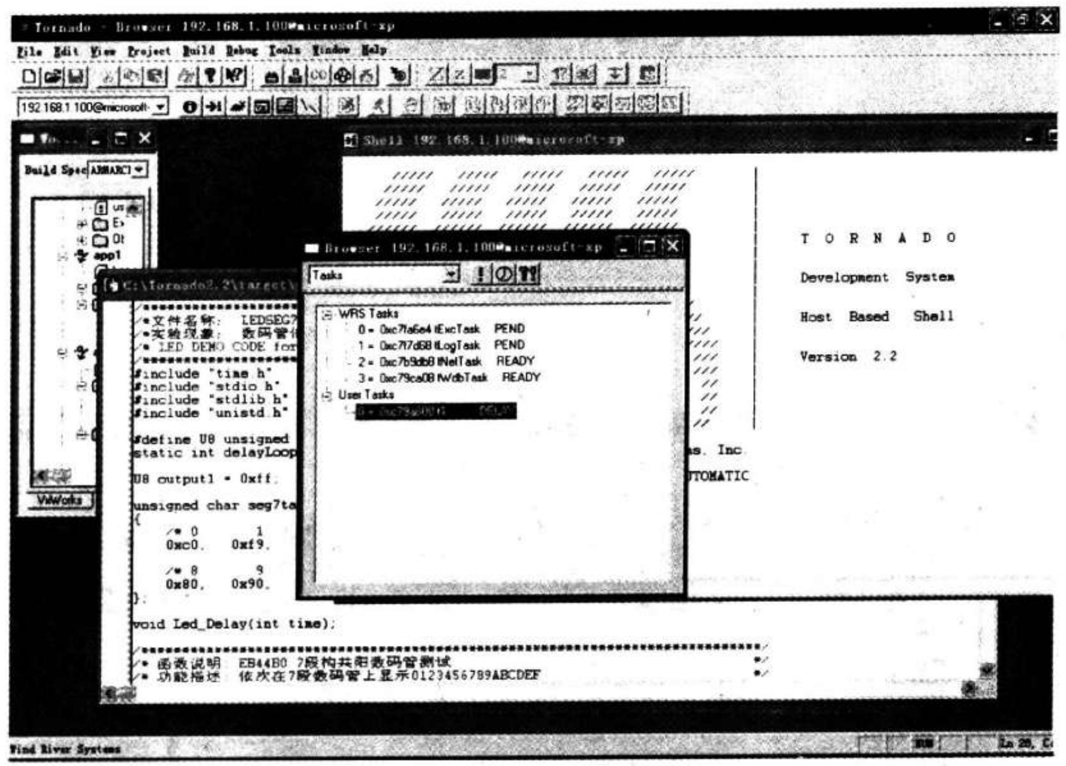


图 6-9 检查系统的性能

6.2 任务管理

6.2.1 任 务

任务是代码运行的一个实体;从系统的角度看,任务是竞争系统资源的最小运行单元。任务可使用或等待 CPU、I/O 设备以及内存空间等系统资源,并独立于其他任务,与其并发运行(宏观上如此)。VxWorks 内核使任务能快速地共享系统的绝大部分资源,同时有独立的上下文来保存各任务的信息。

1. 任务结构

多任务设计能随时打断正在执行的任务,对内外部发生的事件在确定的时间里作出响应。VxWorks 实时内核 Wind 提供了基本的多任务环境。从表面上看,多个任务同时执行;实际上,系统内核根据某一调度策略使其交替运行。系统调度器使用任务控制块的数据结构(简称 TCB)来实现任务调度功能。任务控制块用来描述一个任务,每个任务都与一个 TCB 关联。

TCB 包括了任务的当前状态、优先级、要等待的事件或资源、任务程序码的起始地址以及初始堆栈指针等信息；调度器在任务最初被激活以及从休眠态重新被激活时，会用到这些信息。

此外，TCB 还被用来存放任务的"上下文"(Context)。任务的上下文就是当一个执行中的任务被停止时所要保存的所有信息。当任务被重新执行时，必须要恢复上下文。通常，上下文即计算机当前的状态，也就是各寄存器中的内容，同发生中断时所要保存的内容一样。当发生任务切换时，将当前运行任务的上下文存入 TCB；而将要被执行的任务的上下文从其 TCB 中取出，放入各寄存器中。于是转而执行这个任务，执行的起点是上一次它在运行时被中止的位置。任务的上下文包括：

- 任务的执行点，即任务的程序计数器；
- CPU 中的寄存器；
- 动态变量和函数调用所需的堆栈；
- I/O 操作分配的标准输入/输出和标准错误输出操作；
- 一个延时定时器；
- 一个时间片定时器；
- 内核控制结构；
- 信号句柄；
- 用于调试和性能监视的值。

VxWorks 中内存地址空间不是任务上下文的一部分，所有代码运行在同一地址空间内。若每一任务需要各自的内存空间，则需可选产品 VxVMI 的支持。

2. 任务状态和状态迁移

实时系统的一个任务可有多种状态，其中最基本的状态有以下 4 种：

- 就绪态：任务只等待系统分配 CPU 资源；
- 挂起态：任务须等待某些不可利用的资源而被阻塞；
- 休眠态：如果系统无需某个任务工作，则该任务处于休眠状态；
- 延迟态：任务被延迟时所处的状态。

当系统函数对某一任务进行操作时，任务从一种状态迁移到另一种状态，如图 6-10 所示。处于任一状态的任务都可被删除。

表 6-1 列出了一些与任务状态迁移有关的系统调用。

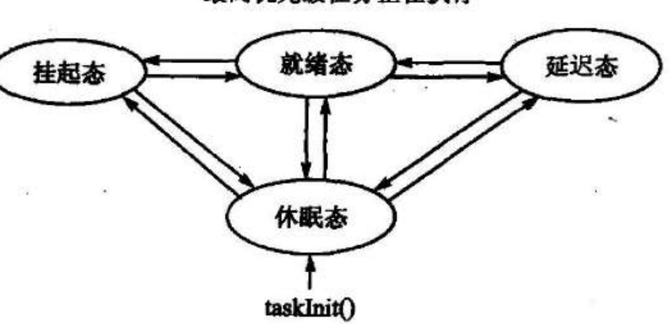


图 6-10　任务的状态及迁移

表 6-1　与任务状态迁移有关的系统调用

状态迁移	调　用	状态迁移	调　用
就绪态→挂起态	semTake()/msgQReceive()	延迟态→就绪态	expired delay
就绪态→延迟态	taskDelay()	延迟态→休眠态	taskSuspend()
就绪态→休眠态	taskSuspend()	休眠态→就绪态	taskResume()/taskActivate()
挂起态→就绪态	semGive()/msgQSend()	休眠态→挂起态	taskResume()
挂起态→休眠态	taskSuspend()	休眠态→延迟态	taskResume()

6.2.2 任务调度

多任务调度须采用一种调度算法来分配 CPU 给就绪态任务。wind 内核采用基于优先级的抢占式调度法作为其缺省策略；同时也提供了轮转调度法。

基于优先级的抢占式调度具有很多优点。它为每个任务指定不同的优先级。没有处于挂起或休眠态的最高优先级任务将一直运行下去；当更高优先级的任务由就绪态进入运行时，系统内核会立即保存当前任务的上下文，以便切换到更高优先级的任务，如图 6-11 所示。

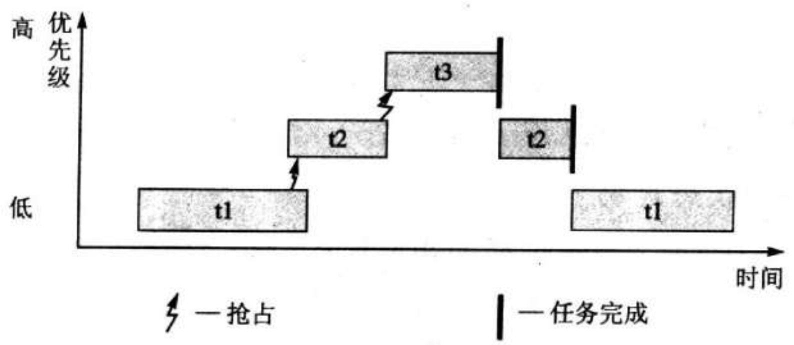


图 6-11　任务的抢占执行

VxWorks 内核将优先级划分为 256 级(0～255)。优先级 0 为最高优先级；优先级 255 为最低优先级。当任务被创建时，系统根据给定值分配任务优先级。然而，优先级也可是动态的，能在系统运行时由用户通过系统调用 taskPrioritySet()来加以改变。

轮转调度法分配给处于就绪态的每个同优先级任务一个相同的执行时间片。时间片的长度可由系统调用 KernelTimeSlice()通过输入参数值来指定。显然，每个任务都有一个运行时间计数器，任务运行时每一时间滴答加 1。一个任务用完时间片之后，即进行任务切换：停止执行当前运行的任务，将其放入队列尾部；将运行时间计数器清 0，并开始执行就绪队列中的下一个任务。当运行任务被更高优先级的任务抢占时，此任务的运行时间计数器被保存，直到该任务下次运行时，如图 6-12 所示。

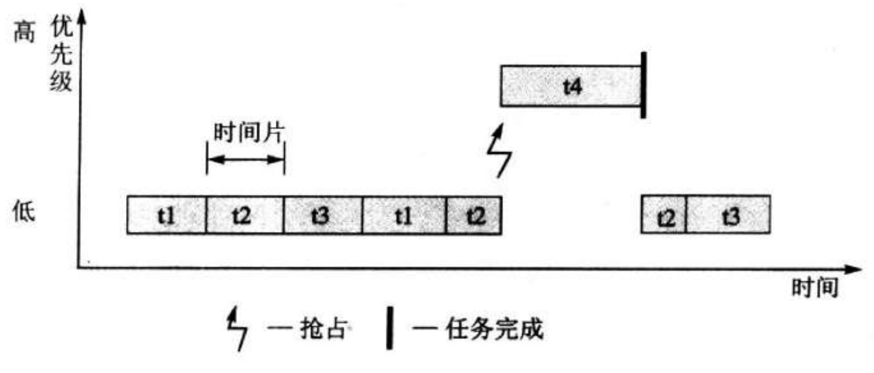


图 6-12 任务的轮转执行

(1) 关于任务的抢占禁止

Wind 内核可通过调用 taskLock()和 taskUnlock()来使调度器起作用或失效。若一个任务调用 taskLock()从而使调度器失效，则该任务运行时没有基于优先级的抢占发生；而任务被阻塞或挂起时，调度器会从就绪队列中取出最高优先级的任务运行。当设置抢占禁止的任务解除阻塞，再次开始运行时，抢占又被禁止。这种通过抢占禁止来防止任务切换的方法，对中断处理不起作用。

(2) 异常处理

在 VxWorks 中，很多情况会引起异常，如非法命令、总线或地址错误以及被零除等。VxWorks 异常处理一般是：先将引起异常的任务挂起，保存任务在异常出错处的状态值，而内核和其他任务继续执行；此时用户可借助 Tornado 调试工具，查看当前任务状态，从而确定被挂起的任务及其一些具体情况。

6.2.3 任务操纵

VxWorks 内核的任务管理提供了动态创建、删除和控制任务的功能，其实现函数主要由 taskLib 系统库提供。

1. 创建和激活任务的几个函数

函数名称：taskSpawn()

函数说明：创建并且激活一个新任务。

```
int taskSpawn
    (
    char *      name,           /* 新任务的名称 */
    int         priority,       /* 新任务的优先级 */
    int         options,        /* 新任务的模式 */
```

```
    int         stackSize,          /* 堆栈大小 */
    FUNCPTR     entryPt,            /* 任务的入口函数 */
    int         arg1,               /* 传递给入口函数的参数列表(依次为 1～10) */
    int         arg2,
    int         arg3,
    int         arg4,
    int         arg5,
    int         arg6,
    int         arg7,
    int         arg8,
    int         arg9,
    int         arg10
    )
```

函数名称： taskInit()

函数说明： 初始化新任务。

```
STATUS taskInit
    (
    WIND_TCB *      pTcb,           /* 新任务的 TCB 地址 */
    char *          name,           /* 新任务的名称 */
    int             priority,       /* 新任务的优先级 */
    int             options,        /* 新任务的模式 */
    char *          pStackBase,     /* 堆栈基地址 */
    int             stackSize,      /* 堆栈大小 */
    FUNCPTR         entryPt,        /* 任务的入口函数 */
    int             arg1,           /* 传递给入口函数的参数列表(依次为 1～10) */
    int             arg2,
    int             arg3,
    int             arg4,
    int             arg5,
    int             arg6,
    int             arg7,
    int             arg8,
    int             arg9,
    int             arg10
    )
```

函数名称：taskActivate()

函数说明：激活一个已初始化的任务。

```
STATUS taskActivate
    (
    int tid                              /* 任务的 ID 值 */
    )
```

例如：

```
id = taskSpawn (name, priority, options, stacksize,main,arg1,arg10 );
```

taskSpawn 创建一个新任务，且会根据创建时指定的参数自动为新任务申请堆栈空间，并设置初始环境（如优先级和任务的模式等），最后还会自动激活该任务。

与 taskSpawn 不同，taskInit 只创建一个新任务，该任务的堆栈需由创建时指定（预先分配）。另外 taskInit 初始化的任务是一个未激活的任务，要将这个任务变成激活状态还须使用 taskActivate 来完成。

无论使用 taskSpawn 还是 taskInit 创建新任务，都会涉及任务堆栈的问题。为一个任务精确地分配堆栈是很困难的，为了避免堆栈空间的浪费和堆栈溢出，往往需要借助于系统提供的调试工具（如 Browser）来进行分析。同时，在创建任务时，可先根据任务的具体情况将堆栈设置为一个较大的值（如 20 KB 或 100 KB），随后在运行过程中使用 checkStack()来周期性地监视堆栈使用情况，并根据监视结果得到一个较合理的值。

创建任务时，可指定一个任意长度的字串作为任务的名称。该任务名称在系统中并不一定是惟一的，但为了避免混淆，还是建议使用惟一的任务名。如果创建任务时不指定任务名，即 name 为 NULL，则此时系统会自动生成一个惟一的字串（形如 tN，其中 N 为编号）作为该任务的名称。关于任务的名称和任务 ID，系统提供了以下几个函数来进行处理：

函数名称：taskName()

函数说明：获取指定任务 ID 的任务名称。

```
char * taskName
    (
    int tid                              /* 任务 ID */
    )
```

函数名称：taskNameToId()

函数说明：获取指定任务名称的任务 ID。

```
int taskNameToId
    (
```

```
    char * name                    /* 任务名称 */
    )
```

函数名称: taskIdSelf()

函数说明: 获取本任务的 ID。

```
int taskIdSelf (void)
```

函数名称: taskIdVerify()

函数说明: 检查指定 ID 的任务是否存在。

```
STATUS taskIdVerify
    (
    int tid                    /* 任务 ID */
    )
```

taskSpawn 的第 3 个参数为新任务的模式,其取值如表 6-2 所列。

表 6-2　新任务模式的取值

宏定义	值	描　述
VX_FP_TASK	0x0008	是否需要浮点协处理器参与
VX_NO_STACK_FILL	0x0100	不采用 0xee 填充堆栈
VX_PRIVATE_ENV	0x0080	使用特定(私有)的环境执行该任务
VX_UNBREAKABLE	0x0002	禁止本任务的断点
VX_ALTIVEC_TASK	0x0400	1=ALTIVEC,协处理器支持

当创建的新任务具有以下特征之一时,必须指定 VX_FP_TASK:

- 执行了浮点操作;
- 其中调用的某个函数返回了浮点值;
- 其中调用的某个函数存在浮点类型的参数。

例如:

```
tid = taskSpawn ("tMyTask", 90, VX_FP_TASK, 20000, myFunc, 2387, 0, 0,
        0, 0, 0, 0, 0, 0, 0);
```

创建任务后,可使用以下函数获取某个任务的模式以及设置模式:

函数名称: taskOptionsGet()

函数说明: 获取任务的模式。

```
STATUS taskOptionsGet
    (
```

```
    int   tid,              /* 任务 ID */
    int * pOptions          /* 任务的模式 */
    )
```

函数名称： taskOptionsSet()

函数说明： 设置任务的模式。

```
STATUS taskOptionsSet
    (
    int tid,                /* 任务 ID */
    int mask,               /* 模式的比特掩码 */
    int newOptions          /* 待设置模式的比特掩码 */
    )
```

2. 任务信息的获取

可通过以下函数获取系统中任务的相关信息：

函数名称： taskIdListGet()

函数说明： 获取所有任务的 ID 列表。

```
int taskIdListGet
    (
    int idList[],           /* 任务 ID 数组 */
    int maxTasks            /* 任务 ID 数组的最大长度 */
    )
```

函数名称： taskInfoGet()

函数说明： 获取指定任务的信息。

```
STATUS taskInfoGet
    (
    int         tid,        /* 任务 ID */
    TASK_DESC * pTaskDesc   /* 任务描述符指针 */
    )
```

函数名称： taskPriorityGet()

函数说明： 获取指定任务的优先级。

```
STATUS taskPriorityGet
    (
    int   tid,              /* 任务 ID */
```

```
    int * pPriority             /* 返回的优先级指针 */
    )
```

函数名称：taskRegsGet()

函数说明：读取指定任务的寄存器值。

```
STATUS taskRegsGet
    (
    int        tid,             /* 任务 ID */
    REG_SET * pRegs             /* 寄存器指针 */
    )
```

函数名称：taskRegsSet()

函数说明：设置指定任务的寄存器值。

```
STATUS taskRegsSet
    (
    int        tid,             /* 任务 ID */
    REG_SET * pRegs             /* 寄存器指针 */
    )
```

函数名称：taskIsSuspended()

函数说明：检查指定任务是否处于挂起状态。

```
BOOL taskIsSuspended
    (
    int tid                     /* 任务 ID */
    )
```

函数名称：taskIsReady()

函数说明：检查指定任务是否处于就绪状态。

```
BOOL taskIsReady
    (
    int tid                     /* 任务 ID */
    )
```

函数名称：taskTcb()

函数说明：获取指定任务的控制块地址。

```
WIND_TCB * taskTcb
    (
```

```
    int tid                     /* 任务 ID */
    )
```

任务的状态等信息是动态的，所以以上函数返回的信息不一定是指定任务的当前状态信息（被挂起的任务除外）。尽管如此，但这些动态信息对分析任务的运行情况还是非常有用的。

3. 任务删除

VxWorks 的任务可以被动态删除；任务的退出有多种方法，如 exit 和 taskDelete 等。下面是用于删除任务的一些函数调用：

函数名称： exit()

函数说明： 中止调用该例程的任务，释放该任务使用的内存（仅包括任务堆栈和任务控制块）。

```
void exit
    (
    int code                    /* 任务的退出码 */
    )
```

函数名称： taskDelete()

函数说明： 中止指定的任务，释放该任务使用的内存（仅包括任务堆栈和任务控制块）。

```
STATUS taskDelete
    (
    int tid                     /* 任务 ID */
    )
```

函数名称： taskSafe()

函数说明： 保护调用该函数的任务，避免该任务被删除。

```
STATUS taskSafe (void)
```

函数名称： taskUnsafe()

函数说明： 中止任务的删除保护。

```
STATUS taskUnsafe (void)
```

注意： 使用 exit 和 taskDelete 删除任务时，不会自动释放任务动态申请的内存。

当某个任务被删除时，不会给其他任务发送相关的提醒消息。在一个多任务环境下，这一点有可能引起一些问题。例如一个任务在使用共享资源，而它又被非法删除了，这样该任务所使用的共享资源将一直被占用。使用 taskSafe 能很好地解决这个问题，一旦某个任务调用了 taskSafe，它将处于一种被保护状态（即不能被删除），直到使用 taskUnsafe 解除这种保护。典

型的保护代码参见程序清单 6－1。

程序清单 6－1　任务的安全删除

```
taskSafe();
semTake(semId,WAIT_FOREVER);          /* 获取信号量 */
...                                   /* 临界区 */
semGive(semId);                       /* 释放信号量 */
taskUnsafe();
```

4. 任务状态控制

VxWorks 任务有就绪、挂起、休眠和延迟等状态，系统提供了一些函数来控制任务的状态。一个典型的应用就是 VxWorks 的调试工具，它会频繁地挂起和恢复调试的任务；而且也只有在挂起状态下，才能检查任务的一些细节。另外当一个任务出现错误时，也须使用一种机制来重新启动该任务（嵌入式系统中除了一次性执行的初始化任务外，其他任务都应是必不可少的）等。系统提供的操作函数主要有以下几个：

函数名称： taskSuspend()

函数说明： 挂起任务。

```
STATUS taskSuspend
    (
    int tid                          /* 任务 ID */
    )
```

函数名称： taskResume()

函数说明： 恢复任务。

```
STATUS taskResume
    (
    int tid                          /* 任务 ID */
    )
```

函数名称： taskRestart()

函数说明： 重启任务。

```
STATUS taskRestart
    (
    int tid                          /* 任务 ID */
    )
```

函数名称：taskDelay()

函数说明：将任务延时指定的 ticks 数。

```
STATUS taskDelay
    (
    int ticks                          /* 延时的 ticks 数 */
    )
```

函数名称：nanosleep()

函数说明：将任务延时指定的时间长度(精度取决于系统定时器)。

```
int nanosleep
    (
    const struct timespec * rqtp,      /* 延时的时间长度 */
    struct timespec *       rmtp       /* 实际延时的时间长度 */
    )
```

taskDelay 除了实现延时功能以外，还有一项功能，就是当指定的延时 ticks 数为 0 时，它可以让系统的调度器执行一次调度，这样可使与之优先级相同的任务能够有机会运行。例如：

```
taskDelay (NO_WAIT);
```

5. 任务管理扩展

无论是创建任务、删除任务，还是切换任务过程中，系统都为用户提供了一个非常方便的扩展接口；通过这个接口，用户可以更加灵活地控制任务的行为。这些扩展接口是：

函数名称：taskCreateHookAdd()

函数说明：添加创建任务时的钩子函数。

```
STATUS taskCreateHookAdd
    (
    FUNCPTR createHook                 /* 钩子函数 */
    )
```

其中：createHook 必须声明为如下类型：

```
void createHook
    (
    WIND_TCB * pNewTcb                 /* 新任务的控制块 */
    )
```

函数名称：taskCreateHookDelete()

函数说明：删除任务创建时的钩子函数。

```
STATUS taskCreateHookDelete
    (
    FUNCPTR createHook          /* 钩子函数 */
    )
```

函数名称：taskSwitchHookAdd()

函数说明：添加任务切换时的钩子函数。

```
STATUS taskSwitchHookAdd
    (
    FUNCPTR switchHook          /* 钩子函数 */
    )
```

其中：switchHook 必须声明为如下类型：

```
void switchHook
    (
    WIND_TCB * pOldTcb,         /* 切换到后台的任务控制块指针 */
    WIND_TCB * pNewTcb          /* 获得执行权的任务控制块指针 */
    )
```

函数名称：taskSwitchHookDelete()

函数说明：删除任务切换时的钩子函数。

```
STATUS taskSwitchHookDelete
    (
    FUNCPTR switchHook          /* 钩子函数 */
    )
```

函数名称：taskDeleteHookAdd()

函数说明：添加任务删除时的钩子函数。

```
STATUS taskDeleteHookAdd
    (
    FUNCPTR deleteHook          /* 钩子函数 */
    )
```

其中：deleteHook 必须声明为如下类型：

```
void deleteHook
    (
    WIND_TCB * pTcb             /* 待删除任务的控制块指针 */
    )
```

函数名称： taskDeleteHookDelete()

函数说明： 删除任务删除时的钩子函数。

```
STATUS taskDeleteHookDelete
    (
    FUNCPTR deleteHook          /* 钩子函数 */
    )
```

并非所有系统函数都能够在任务控制的钩子函数中被调用，表 6-3 列出了钩子函数能够调用的函数库及相关例程。

表 6-3 钩子函数能够调用的函数库及相关例程

库	函 数
bLib	所有函数
fppArchLib	fppSave()，fppRestore()
intLib	intContext()，intCount()，intVecSet()，intVecGet()，intLock()，intUnlock()
lstLib	除 lstFree()之外的所有函数
mathALib	如果使用 fppSave()/fppRestore()，则可调用所有函数
rngLib	除 rngCreate()和 roundlet()之外的所有函数
taskLib	taskIdVerify()，taskIdDefault()，taskIsReady()，taskIsSuspended()，taskTcb()
vxLib	vxTas()

6.2.4 共享代码和可重入代码

在一个多任务环境中，函数的可重入性是十分重要的。可重入函数是可被多个任务调用的函数，任务在调用时不必担心数据是否会出错，如图 6-13 所示。在写函数时只须考虑尽量用局部变量（如寄存器和堆栈中的变量）；对于要使用的全局变量（例如采用关中断和信号量等）要加以保护。这样构成的函数就一定是一个可重入的函数。

此外，编译器是否有可重入函数的库，与其所服务的操作系统有关。例如：DOS 下的 Borland C 和 Microsoft C/C++等就不具备可重入的函数库，这是因为 DOS 是一个单用户、

单任务的操作系统。为了确保每个任务能控制自己的私有变量,在一个可重入的 C 函数中,需将这样的变量声名为局部变量。C 编译器将其存放在调用栈或寄存器中。

在 VxWorks 中,多个任务可调用同一子函数或函数库。VxWorks 系统动态链接工具使这相当容易,这种共享代码使系统更加高效,易于维护。

VxWorks 系统主要采用如下几种可重入技术:

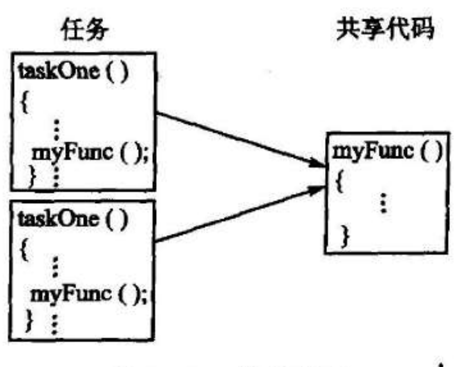


图 6-13 共享代码

1. 动态堆栈变量

许多子函数只是纯代码,除了动态堆栈变量外没有其他数据;调用程序的参数作为子函数的数据。这种子函数是完全可重入的,它们有各自的堆栈空间,因此多个任务可同时使用这种子函数,而不会互相影响,如图 6-14 所示。

2. 受保护的全局和静态变量

一些函数库包含公有数据,多个任务的同时调用很可能会导致对公有数据的破坏,因此使用起来要格外小心。系统采用信号量互斥机制来防止任务同时访问临界资源。

3. 任务变量

一些公用函数要求对于每个调用程序都有明确的全局或静态变量值。为了满足这一点,VxWorks 提供的任务变量允许 4 字节变量加入任务上下文中,当任务切换时变量的值也切换,如图 6-15 所示。任务变量可使用系统提供的一组函数来管理,这些函数是 taskVarAdd()、taskVarDelete()、taskVarSet() 和 taskVarGet()。

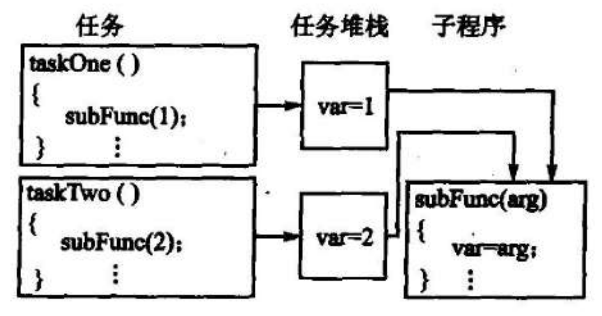


图 6-14 共享代码的重入

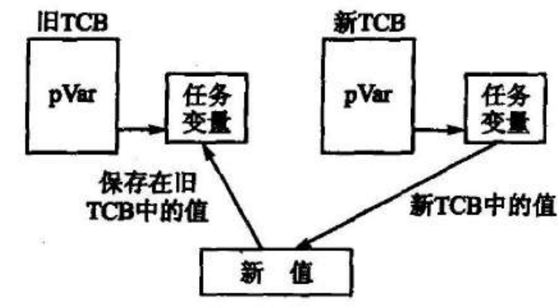


图 6-15 任务变量

编写可重入函数必须遵循以下规则:

- 将所有局部变量声明为 auto(缺省态)或寄存器型。
- 尽量不要使用 static 或 extern 变量。若有必要,则用互斥机制进行保护。

6.2.5 系统任务

VxWorks 操作系统启动后，会自动创建以下几个系统任务：

① 根任务。内核首先执行根任务 tUsrRoot。其入口点为文件 config/all/usrConfig.c 中的 usrRoot()函数，它负责初始化 VxWorks 工具，并创建注册、异常处理、网络通信和 tRlogind 等任务。一般来说，在所有初始化工作完成后，根任务 tUsrRoot 会被删除。

② 消息记录任务。消息记录任务 tLogTask 被 VxWorks 模块用于传送无需 I/O 操作的系统消息。

③ 异常处理任务。异常处理任务 tExcTask 有最高优先级，它负责系统的异常情况出错处理，不能被挂起、删除或改变优先级。

④ 网络通信任务。网络通信任务 tNetTask 负责系统级任务的网络通信。

⑤ 目标代理任务。如果目标代理程序运行在任务模式下，则目标代理任务 tWdbTask 被创建，以响应主机目标服务器的请求。

6.2.6 注意事项

VxWorks 系统是专为嵌入式实时应用而设计的模块化实时操作系统。对用户来说，一个实时应用软件由板级支持包 BSP、操作系统内核及用户选用组件，以及中断服务程序 ISR 组成。操作系统为用户提供了大量的系统调用，这是用户与操作系统的接口。针对当前开发工作和实时系统的特性，在实时应用软件的编写中要注意以下几点：

(1) 任务划分要合理

① 功能内聚性。对于功能联系较紧密的各项工作，可作为一个任务来运行。如果都以一个个任务来进行相互之间的消息通信，则会影响系统效率，不如采用任务中一个个独立的模块来完成。

② 时间紧迫性。对于对实时性要求较高的任务，要以高优先级运行，从而保证事件的实时响应。

③ 周期执行原则。对于一个需周期性执行的工作，应作为一个任务来运行，通过定时器以一定时间间隔激活任务。

(2) 防止死锁、饥饿和优先级反转

死锁：指多个任务因为等待进入对方占据的临界区而导致的不可自行恢复的运行终止。在程序设计时要注意对死锁的预防：尽量使互斥资源在相同优先级任务中使用；当须在不同优先级任务中使用时，要注意对死锁的解锁处理。

饥饿：指优先级较低的任务长期得不到系统资源（主要指 CPU 资源）而造成的任务长期无法运行。造成饥饿的主要原因是优先级较高的任务调度过于频繁或占用时间太长。合理分配任务的优先级和对较高优先级任务的合理调度是解决饥饿的关键。

任务的优先级反转：实时多任务操作系统的热门话题，是指高优先级任务因等待低优先级任务占用的互斥资源，而被较低优先级（高于低优先级但低于高优先级）的任务不断抢占的情况。有些实时多任务操作系统自身提供保护机制可对优先级反转进行预防。在操作系统未提供保护的情况下，就需要编程人员在编程时注意避免优先级反转的情况发生（例如在同一优先级内使用互斥资源），或采取相应的手段进行处理（例如动态地进行优先级提升）。

注意：VRTX 和 VxWorks 自身提供防止优先级反转机制；pSOS 未提供保护机制。

(3) 使用对象的名称访问资源

通过任务名、消息队列名和信号量名来调用这些资源，能保证应用系统可靠地运行，同时也便于程序的阅读。例如：系统调用 taskName()、taskNameToId() 和 taskIsSelf() 等，方便了用户对资源的管理，更加直观。

(4) 用户任务优先级确定

VxWorks 系统中优先级分为 256 级（0～255），其中 0 为最高优先级，255 为最低优先级。任务的优先级在任务创建时被分配，但在任务运行时可通过系统调用 taskPrioritySet() 动态改变其优先级。当操作系统在目标板上启动成功后，系统级任务已在运行，对主机与目标机之间的通信进行管理，因此用户任务优先级要低于系统级任务，一般最高为 150。同时，如何确定用户各任务优先级，如何使各任务间良好地协同工作，取决于任务的紧急程度以及实际情况，调试过程中的摸索总结也很重要。

6.3 任务通信

VxWorks 支持各任务间的通信机制，提供了多样的任务间通信方式，主要有如下几种：

- 共享内存：主要是数据的共享；
- 信号量：用于基本的互斥和任务同步；
- 消息队列和管道：用于单 CPU 的消息传送；
- Socket 和远程过程调用：用于网络间任务消息传送；
- 信号：用于异常处理。

在多处理器之间的任务也可采用共享内存对象来实现任务间通信，只是在系统配置上有所不同。下面介绍在 VxWorks 应用程序编写中常用的通信手段。

6.3.1 共享存储区

任务间通信最简单的方法是共享存储区，即相关的各任务分享属于其地址空间的同一块内存区域，如图 6-16 所示。在 VxWorks 中所有任务都存在于单一的线性地址空间内，因此任务间共享数据就变得非常容易。全局变量、线性队列、环形队列、链表和指针都可被运行在不同上下文的代码所访问。

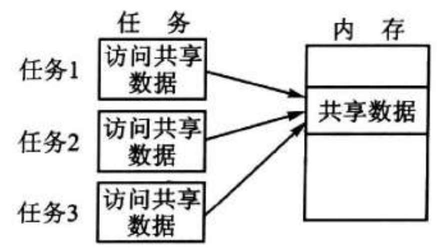


图 6-16 共享存储区

程序清单 6-2 的例子中首先定义了一个名为 globalVar 的全局变量，并分别在 task1 和 task2 中进行访问。

程序清单 6-2 共享存储区的使用

```
int globalVar;

int task1(void)
    {
    :
    globalVar++;
    :
    }

int task2(void)
    {
    :
    globalVar--;
    :
    }
```

6.3.2 互 斥

当某一地址空间用于数据交换时，为了避免冲突，对内存的锁定是非常重要的。两个或多个任务读/写某些共享数据时，最后的结果取决于任务运行的精确时序。如果控制不当，则有可能得到错误结果，所以必须采取某种手段来确保当一个任务在使用一个共享变量或文件时，其他任务不能执行同样的操作。较常用的方法主要有：关中断、禁止抢占和用信号量锁定资源

等方法。前面提到的共享存储区，实际上也是一种临界资源，因此访问时同样须进行资源的锁定。

一般来说，关中断是最有效的解决共享冲突的方法，见程序清单 6-3。但对实时应用来说，它阻止了系统对外部事件的响应，无法满足实时性的要求；同样，中断延迟也不能接受。所以如果在程序中采用了这种方式来访问共享资源，要保证访问临界区的代码处理时间尽可能短，而且在访问临界区的代码中不允许使用一些可能引起阻塞的系统调用。

程序清单 6-3　使用关中断来保护临界资源的访问

```
funcA ()
    {
    int lock = intLock();
    ...     /* 访问临界区,此时系统无法响应中断 */
    intUnlock (lock);
    }
```

如果采用抢占禁止的方法，则会稍微好些。因为采用该方法访问临界区时，中断是可被正确响应的，但它也会引发一些潜在的问题。例如：处于就绪态的高优先级任务将被暂时地挂起，直到该任务取消抢占禁止，如程序清单 6-4 所列。

程序清单 6-4　使用抢占禁止来保护临界资源的访问

```
funcA ()
    {
    taskLock ();
    ...     /* 访问临界区,此时系统可以响应中断,但不能进行任务切换 */
    taskUnlock ();
    }
```

6.3.3　信号量

VxWorks 信号量提供最快速的任务间通信机制，它主要用于解决任务间的互斥和同步。针对不同类型的问题，有以下 3 种信号量：

- 二进制信号量：使用最快捷、最广泛，主要用于同步或互斥；
- 互斥信号量：主要用于优先级继承、安全删除等；
- 计数器。

VxWorks 还提供 POSIX 信号量和多处理器上信号量。VxWorks 信号量的操作非常简单，针对不同类型的信号量，只需在创建时指定，而使用时没有任何的区别。创建信号量时，还

可指定信号量的工作模式，即采用的是先进先出(SEM_Q_FIFO)还是优先级控制(SEM_Q_PRIORITY)。其中优先级控制对于任务的实时性有很大帮助，可以保证一些重要的事件被优先处理。系统提供的信号量操纵函数有如下几种：

函数名称： semBCreate()

函数说明： 创建二进制信号量。

```
SEM_ID semBCreate
    (
    int options,                    /* 信号量模式 */
    SEM_B_STATE initialState        /* 初始状态 */
    )
```

函数名称： semMCreate()

函数说明： 创建互斥信号量。

```
SEM_ID semMCreate
    (
    int options                     /* 信号量模式 */
    )
```

函数名称： semCCreate()

函数说明： 创建计数器。

```
SEM_ID semCCreate
    (
    int options,                    /* 信号量模式 */
    int initialCount                /* 计数器初始值 */
    )
```

函数名称： semDelete()

函数说明： 删除信号量。

```
STATUS semDelete
    (
    SEM_ID semId                    /* 信号量 ID */
    )
```

函数名称： semTake()

函数说明： 获取信号量。

```
STATUS semTake
    (
```

```
    SEM_ID semId,                  /* 信号量 ID */
    int timeout                    /* 超时值 */
    )
```

函数名称： semGive()

函数说明： 释放信号量。

```
STATUS semGive
    (
    SEM_ID semId                   /* 信号量 ID */
    )
```

函数名称： semFlush()

函数说明： 解除所有阻塞在该信号量上的任务的阻塞状态。

```
STATUS semFlush
    (
    SEM_ID semId                   /* 信号量 ID */
    )
```

1. 二进制信号量

(1) 二进制信号量的获取和释放

二进制信号量可被看作是一个标志。当某个任务准备使用 semTake()获取信号量时，其结果取决于该信号量当前的状态。如果信号量处于可用状态，则该任务在获取信号量后继续运行，且该信号量同时会变为不可用状态；如果信号量处于不可用状态，则该任务将被挂起，其流程如图 6－17 所示。

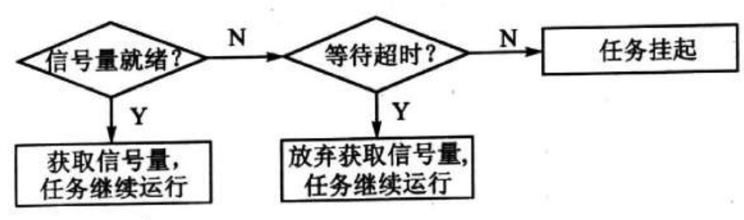


图 6－17　二进制信号量的获取

释放二进制信号量时系统也会检查信号量的当前状态。如果该二进制信号量已处于可用状态，那么系统将不进行任何操作，而释放该信号量的任务会继续运行；如果在释放时信号量处于不可用状态，那么系统会检查是否有任务在等待该信号量，若有则激活相应的任务。其流程如图 6－18 所示。

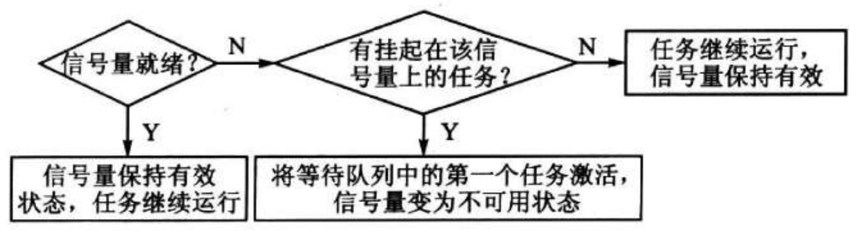


图 6-18 二进制信号量的释放

(2) 二进制信号量的用途

二进制信号量有两方面用途：互斥和同步。互斥是指不同的任务访问共享资源；同步则相当于条件执行某个任务。在用作互斥功能时，可先创建一个信号量，并将其初始化为可用状态。在访问临界资源时先通过 semTake()获取该信号量，访问完毕后就释放该信号量。程序清单 6-5 为一个简单的例子。

程序清单 6-5 使用二进制信号量进行互斥处理

```
#include "VxWorks.h"
#include "semLib.h"

SEM_ID semMutex;

void init()
    {
    /* 创建二进制信号量 */
    semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
    }

void task()
    {
    semTake (semMutex, WAIT_FOREVER);
    ...    /* 访问临界资源 */
    semGive (semMutex);
    }
```

用于同步时，往往在创建信号量时先将其设置为不可用状态；等某个条件存在时再修改信号量的状态，并激活等待该信号量的任务。示例代码参见程序清单 6-6。

程序清单 6-6　使用二进制信号量进行同步处理

```
SEM_ID semSync;

init ()
    {
    intConnect (..., eventInterruptSvcRout, ...);
    semSync = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    taskSpawn (..., task1);
    }

task1 ()
    {
    :
    semTake (semSync, WAIT_FOREVER);          /* 等待 */
    ...     /* 处理 */
    }

    eventInterruptSvcRout ()
        {
        :
        semGive (semSync);                    /* 激活等待该事件的任务 */
        :
        }
```

2. 互斥信号量

互斥信号量实际上也是一种二进制信号量，主要用于解决内部互斥问题（如优先级反转、安全删除以及资源的递归访问）。互斥信号量与二进制信号量的不同之处在于：

- 互斥信号量仅用于互斥；
- 互斥信号量仅由获取它的任务释放（即调用 semTake()的任务）；
- 不能在中断服务程序中释放；
- 不能执行 semFlush 操作。

(1) 优先级反转

系统中由于某些原因，可能出现优先级反转。如果一个低优先级的任务已获取了临界资源的访问权，则此时高优先级任务再去申请访问临界资源时，会出现高优先级任务等待低优先级任务的情况，也就是说发生了优先级反转，如图 6－19 所示。

图 6－19 中，t1 为高优先级任务；而 t3 为低优先级任务。它们同时通过一个信号量来访

问共享资源，在某一时刻，处于就绪状态的 t3 获取了信号量，并开始访问共享资源；但随后 t1 就绪，抢占了 t3 的处理器运行权，且 t3 也开始尝试获取同一个信号量。这时由于 t3 未释放该信号量，因此高优先级任务 t1 只有等待 t3 执行完毕才能获取信号量。如果此时出现一个介于 t1 和 t3 优先级之间的任务（假设为 t2）就绪，则 t2 将抢占 t3 的运行权，并可能引起灾难性的后果。况且这也不符合设计思路，因为高优先级任务被低优先级任务抢占，而低优先级任务的执行时间又未知。为了解决此类问题，互斥信号量在创建时可使用 SEM_INVERSION_SAFE 模式来避免（如图 6-20 所示）。

```
semId = semMCreate(SEM_Q_PRIORITY|SEM_INVERSION_SAFE);
```

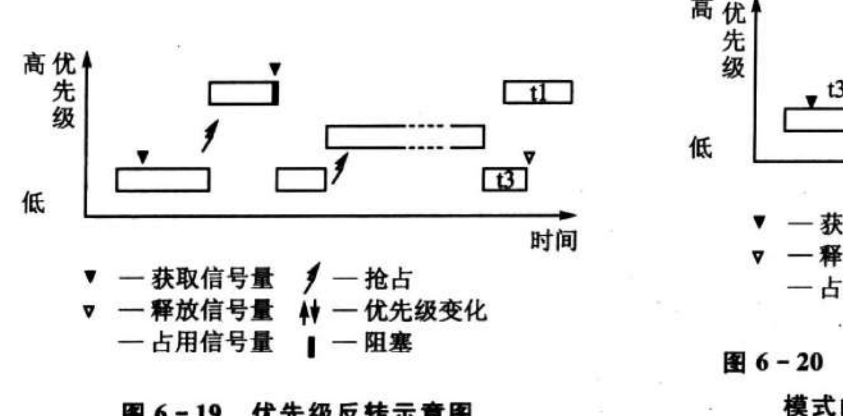


图 6-19　优先级反转示意图

图 6-20　使用 SEM_INVERSION_SAFE 模式的信号量避免优先级反转

采用 SEM_INVERSION_SAFE 创建信号量，系统可以保证当前获取资源的任务被优先执行；这个优先执行是相对的，只针对那些阻塞在该资源上的任务而言。图 6-20 中，如果低优先级任务 t3 先获取信号量，而此时高优先级任务 t1 也尝试获取信号量，那么系统会自动提升 t3 的优先级到 t1 的级别，这样可以防止在使用信号量时低优先级的任务 t3 被低于 t1 而又高于 t3 的任务打断，从而在一定程度上保证程序的执行流程。

(2) 安全删除

如果一个任务正在使用受保护的信号量进行临界区访问，那么这些任务须被保护执行，避免被意外地删除。删除一个在临界区执行的任务可能会导致意想不到的后果，资源可能因此而被破坏，同时受保护的信号量不再可用，直接终止对该资源的所有访问。taskSafe() 和 taskUnsafe() 提供了解决办法：为了安全地删除受保护的信号量，在创建它们时须指定 SEM_DELETE_SAFE 选项。当任务调用信号量获取和释放 semTake()/semGive() 时，用 SEM_DELETE_SAFE 选项创建的信号量会隐含地调用 taskSafe()/taskUnsafe()。使用这种方式，任务在占用信号量时不会被删除。

```
semId = semMCreate(SEM_Q_FIFO|SEM_DELETE_SAFE);
```

(3) 资源的递归访问

互斥信号量能够递归获取。这意味着持有信号量的任务在最终释放其之前能够多次使用 semTake()操作。递归非常适用于一组需要相互调用同时又须进行资源互斥访问的子程序。系统可以跟踪当前哪个任务持有互斥信号量,从而保证资源递归访问的合法性。

在释放信号量之前,递归获取的互斥信号量被释放和提取的次数应相等。系统通过一个计数器跟踪实现,每调用 semTake()一次计数器加 1;每调用 semGive()一次计数器减 1。程序清单 6-7 为一个资源递归访问的例子。

程序清单 6-7 资源的递归访问

```
/* includes */
#include "VxWorks.h"
#include "semLib.h"
SEM_ID mySem;

/* 创建互斥信号量 */
init ()
    {
    mySem = semMCreate (SEM_Q_PRIORITY);
    }
funcA ()
    {
    semTake (mySem, WAIT_FOREVER);          /* 获取信号量 */
    printf ("funcA: Got mutual - exclusion semaphore\n");
    ⋮
    funcB ();                               /* 调用子函数,并在子函数中再次获取信号量 */
    ⋮
    semGive (mySem);
    printf ("funcA: Released mutual - exclusion semaphore\n");
    }
funcB ()
    {
    semTake (mySem, WAIT_FOREVER);
    printf ("funcB: Got mutual - exclusion semaphore\n");
    ⋮
    semGive (mySem);
    printf ("funcB: Releases mutual - exclusion semaphore\n");
    }
```

3. 计数器

计数器是另一种实现同步和互斥的工具。其工作模式与二进制信号量类似，只是计数器会跟踪并记录使用该信号量的任务数，信号量每次被释放时计数器会加 1(如果此时没有任务阻塞在该信号量上)；而在获取该信号量后会自动减 1，当计数器减为 0 时，如果某个任务试图获取该信号量，则将被阻塞。

6.3.4 消息队列

现实的实时应用由一系列相互独立又协同工作的任务组成。信号量为任务间同步和互斥提供了高效方法。单处理器中任务间消息的传送采用消息队列。消息机制使用一个被各有关进程共享的消息队列，任务间经由该消息队列发送和接收消息，如图 6－21 所示。

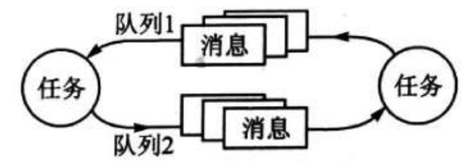


图 6－21　任务间全双工信息传送

消息队列有两种模式：基于先进先出模式和优先级模式。其建立和操作例程如下：

函数名称： msgQCreate()

函数描述： 创建并初始化消息队列。

```
MSG_Q_ID msgQCreate
    (
    int maxMsgs,                  /* 队列的最大消息数 */
    int maxMsgLength,             /* 每个消息的最大字节数 */
    int options                   /* 消息队列的模式 */
    )
```

函数名称： msgQDelete()

函数描述： 删除消息队列。

```
STATUS msgQDelete
    (
    MSG_Q_ID msgQId               /* 消息队列标识 */
    )
```

函数名称： msgQSend()

函数描述： 发送消息。

```
STATUS msgQSend
    (
```

```
    MSG_Q_ID  msgQId,          /* 消息队列标识 */
    char *    buffer,          /* 待发送的消息 */
    UINT      nBytes,          /* 消息的长度 */
    int       timeout,         /* 超时值 */
    int       priority         /* 优先级别 */
    )
```

函数名称： msgQReceive()

函数描述： 接收消息。

```
int msgQReceive
    (
    MSG_Q_ID  msgQId,          /* 消息队列标识 */
    char *    buffer,          /* 接收消息的缓冲区 */
    UINT      maxNBytes,       /* 接收缓冲区的长度 */
    int       timeout          /* 超时值 */
    )
```

程序中可使用 msgQCreate()创建一个消息队列，其参数为消息队列中能够容纳的最大消息数目以及每个消息的最大长度。根据指定的消息数目和每条消息长度，msgQCreate ()会自动申请足够的缓冲空间。在调用 msgQCreate()时，可选的参数有 3 种：

- MSG_Q_FIFO (0x00)：消息按照先进先出的方式管理；
- MSG_Q_PRIORITY (0x01)：消息使用基于优先级的方式管理；
- MSG_Q_EVENTSEND_ERR_NOTIFY (0x02)：使用该标志可在发送消息失败时，给予发送者一个出错提示，该标志默认是关闭的。

在中断处理程序和常规的任务中都可使用 msgQSend()来发送消息，如果没有任务等待该队列中的消息，那么消息将进入消息队列缓冲器；如果有任务正在等待队列中的消息，那么消息将立刻传递给第一个等待的任务。在发送消息时有个非常有用的参数(优先级别)，取值分别是：正常(MSG_PRI_NORMAL)和紧急(MSG_PRI_URGENT)。正常优先级的消息会加入消息队列尾部；而紧急优先级消息则加入消息队列的头部，这为处理一些重要事件带来了很大方便。

使用 msgQReceive()的消息接收者会不停地监视消息队列中的数据。一旦发现有消息，该任务就会从阻塞状态激活，并将第一个可用消息从消息队列中取出。在使用 msgQReceive ()时为了避免一些长时间的等待，可以通过设定超时值来实现。有两个特殊的超时值：NO_WAIT(0)和 WAIT_FOREVER，分别表示立即返回和永远等待。若为其他值，则任务将等待相应的 ticks 数。程序清单 6-8 为一个消息队列的简单例子。

程序清单 6-8 消息队列的使用

```
/*
 * 例子功能:t1 创建消息队列,并向 t2 发送消息
 * t2 在接收到消息后显示消息的内容
 */
#include "VxWorks.h"
#include "msgQLib.h"

#define MAX_MSGS (10)
#define MAX_MSG_LEN (100)
MSG_Q_ID myMsgQId;
task2 (void)
    {
    char msgBuf[MAX_MSG_LEN];
    /* 从消息队列中获取消息 */
    if (msgQReceive(myMsgQId, msgBuf, MAX_MSG_LEN, WAIT_FOREVER) == ERROR)
        return (ERROR);
    /* 显示获取的消息 */
    printf ("Message from task 1:\n %s\n", msgBuf);
    }
#define MESSAGE "Greetings from Task 1"
task1 (void)
    {
    /* 创建消息队列 */
    if ((myMsgQId = msgQCreate (MAX_MSGS, MAX_MSG_LEN, MSG_Q_PRIORITY))
                == NULL)
        return (ERROR);
    /* 发送一个普通优先级的消息。如果消息队列满,则阻塞 */
    if (msgQSend (myMsgQId, MESSAGE, sizeof (MESSAGE), WAIT_FOREVER,
                MSG_PRI_NORMAL) == ERROR)
        return (ERROR);
    }
```

VxWorks 操作系统的 show()命令可以用来显示消息队列的属性。例如:若 myMsgQId 是 Wind 消息队列,则输出将传送到标准的输出设备,并显示如下形式:

```
-> show myMsgQId
Message Queue Id: 0x3adaf0
Task Queuing: FIFO
Message Byte Len: 4
Messages Max: 30
Messages Queued: 14
Receivers Blocked: 0
Send timeouts: 0
Receive timeouts: 0
```

6.3.5 管 道

管道利用 VxWorks 的 I/O 系统，提供了一种灵活的消息传送机制，它是受驱动器（VxWorks 提供的 pipeDrv）管理的虚拟 I/O 设备。任务能调用标准的 I/O 函数打开、读出和写入管道。当任务试图从一个空的管道中读取数据，或向一个满的管道中写入数据时，任务会被阻塞。与消息队列类似，ISR 能向管道中写入信息，但不能从中读取。同时与 I/O 设备相同，管道有一个消息队列所没有的特殊调用——select()。管道的创建与文件创建一样，可以使用 pipeDevCreate 函数来创建管道，这点与消息队列类似：

```
status = pipeDevCreate ("/pipe/name", max_msgs, max_length);
```

创建管道后，即可使用标准的 I/O 函数进行读/写：

```
fd = open ("/pipe/demo", O_RDWR);
write (fd, &outMsg, sizeof (struct msg));
len = read (fd, &inMsg, sizeof (struct msg));
```

6.4 看门狗定时器管理

VxWorks 提供了一种特殊的看门狗机制，以实现在 C 例程中的延时操作。其操作例程有如下几个：

函数名称： wdCreate()

函数描述： 创建看门狗定时器。

```
WDOG_ID wdCreate (void)
```

函数名称：wdDelete()

函数描述：删除看门狗定时器。

```
STATUS wdDelete
    (
    WDOG_ID wdId                    /* 看门狗定时器标识 */
    )
```

函数名称：wdStart()

函数描述：启动看门狗定时器。

```
STATUS wdStart
    (
    WDOG_ID wdId,                   /* 看门狗定时器标识 */
    int     delay,                  /* 延时值 */
    FUNCPTR pRoutine,               /* 期满后的处理例程 */
    int     parameter               /* 期满后处理例程的参数 */
    )
```

函数名称：wdCancel()

函数描述：取消看门狗定时器。

```
STATUS wdCancel
    (
    WDOG_ID wdId                    /* 看门狗定时器标识 */
    )
```

看门狗定时器的使用很简单，首先使用 wdCreate()创建定时器；随后可以使用 wdStart()指定延时值和服务例程，并启动该定时器。程序清单 6－9 列举了一个例子，在看门狗定时器期满后会调用 logMsg 函数。

程序清单 6－9　看门狗定时器的使用

```
#include "VxWorks.h"
#include "logLib.h"
#include "wdLib.h"
#define SECONDS (3)
WDOG_ID myWatchDogId;
task (void)
    {
    /* 创建看门狗定时器 */
```

```
    if ((myWatchDogId = wdCreate( )) == NULL)
        return (ERROR);

    /* 启动看门狗定时器,并在超时后打印一些字符串 */
    if (wdStart (myWatchDogId, sysClkRateGet( ) * SECONDS, logMsg,
                "Watchdog timer just expired\n") == ERROR)
       return (ERROR);
    /* ... */
    }
```

6.5 中断管理

硬件的中断处理是实时操作系统中的一个重要问题。因为在这些系统中往往都是通过事件来驱动的,一旦发生某个事件,要求系统能够尽可能及时地作出响应;因此 VxWorks 采取了一种特殊的方法来满足这一要求,即让中断处理程序运行在单独的任务上下文中,从而缩短中断的识别和处理时间。系统库 intLib 和 intArchLib 提供了一系列中断相关的例程:

函数名称: intConnect()

函数说明: 设置指定中断的服务例程。

```
STATUS intConnect
    (
    VOIDFUNCPTR * vector,           /* 中断向量 */
    VOIDFUNCPTR   routine,          /* 中断服务例程 */
    int           parameter         /* 参数 */
    )
```

函数名称: intContext()

函数说明: 检查当前是否处于中断服务程序的运行级别。

```
BOOL intContext (void)
```

函数名称: intCount()

函数说明: 获取当前中断的嵌套数。

```
int intCount (void)
```

函数名称: intLevelSet()

函数说明: 设置中断屏蔽码。

```
int intLevelSet
    (
    int level                    /* 中断掩码 */
    )
```

函数名称: intLock()

函数说明: 关闭中断。

```
int intLock (void)
```

函数名称: intUnlock()

函数说明: 开启中断。

```
void intUnlock
    (
    int lockKey                  /* 解除中断(由 intLock 关闭) */
    )
```

函数名称: intVecSet()

函数说明: 设置中断向量。

```
void intVecSet
    (
    FUNCPTR * vector,            /* 中断向量的偏移 */
    FUNCPTR   function           /* 向量中的地址 */
    )
```

函数名称: intVecGet()

函数说明: 获取中断向量。

```
FUNCPTR intVecGet
    (
    FUNCPTR * vector             /* 中断向量的偏移 */
    )
```

中断处理的首要工作就是将某个处理程序与具体的中断关联起来，可使用 intConnect() 完成。intConnect 首先创建 SYS_INT_TBL 的结构；然后将调用函数 intVecSet()。intVecSet() 是与处理器体系结构相关的代码，其作用就是将具体的处理例程入口地址填入中断表中。intConnect() 的实现代码见程序清单 6-10。

程序清单 6-10 中断处理

```
#include "intLib.h"
#include "iv.h"
#define SYS_INT_TBL_SIZE        (255 - INUM_INTR_HIGH)

typedef struct
    {
    VOIDFUNCPTR routine;        /* 中断处理程序的入口 */
    int parameter;              /* 中断处理程序的参数 */
    } SYS_INT_TBL;

LOCAL SYS_INT_TBL sysIntTbl [SYS_INT_TBL_SIZE];  /* 中断向量表 */
LOCAL int sysInumVirtBase = INUM_INTR_HIGH + 1;

STATUS sysIntConnect
    (
    VOIDFUNCPTR * vec,
    VOIDFUNCPTR routine,
    int param
    )
    {
    FUNCPTR intDrvRtn;
    if (vec >= INUM_TO_IVEC (0) && vec < INUM_TO_IVEC (sysInumVirtBase))
        {
        /* intConnect() 的处理过程 */
        intDrvRtn = intHandlerCreate ((FUNCPTR)routine, param);
        if (intDrvRtn == NULL)
            return ERROR;
        intVecSet ((FUNCPTR *)vec, (FUNCPTR)intDrvRtn);
        }
    else
        {
        int index = IVEC_TO_INUM (vec) - sysInumVirtBase;
        if (index < 0 || index >= SYS_INT_TBL_SIZE)
            return ERROR;
        sysIntTbl [index].routine    = routine;
```

```
        sysIntTbl[index].parameter   = param;
        }

    return OK;
    }
```

使用intConnect()之后，系统会在处理中断时自动完成一些辅助工作(如寄存器备份和堆栈设置等)如图6-22所示。

所有中断服务例程都会使用同一个堆栈，这个堆栈是在系统启动时建立的，所以中断程序的堆栈必须设置为足够大，以防止最坏情况下发生堆栈溢出。实际应用过程中，可使用前面提到的checkStack()来检查堆栈使用情况。

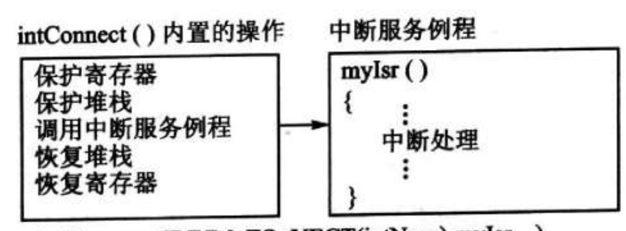


图6-22 intConnect的额外工作

中断服务程序通常都需与具体任务进行通信，VxWorks为中断服务程序提供的通信手段很丰富，常见的有以下几种：

- 共享内存和缓冲：ISR能够与任务共享变量和缓冲等；
- 信号量：ISR中能够释放(give)信号量，但不能等待信号量；
- 消息队列：ISR中能够发送消息，如果消息队列已满，则该消息会被丢弃，但不能接收消息；
- 管道：ISR中能够向管道写数据，如果管已道满，则该数据会被丢弃，但不能读取管道；
- 信号：ISR中能够向任务发送信号。

6.6 网络通信

6.6.1 网络协议

VxWorks网络协议栈是一个与BSD4.4兼容的实时TCP/IP协议栈。与基于BSD4.3的协议栈相比，它增加了完全的路由支持以及Internet的一些新特性，使得VxWorks的网络性能更加优越。VxWorks网络协议栈是一个高性能的协议栈，适合于高性能的网络交换设备以及低价格的网络接入设备，如10/100 MHz以太网交换机、广域网接入设备和ATM交换机等。软件包是可调整的，开发者可将其应用到从IP路由设备到完全TCP/IP的基于SNMP管理的应用系统中。VxWorks协议栈提供本地交换机或远程接入路由器所需的最新路由技术，可被用于千兆级以太交换机或DSL接入复用器等；另外，还支持IP多址广播、CIDR、DH-

CP、DNS 和 SNTP 等网络协议。

VxWorks 协议栈的路由引擎使用一种改进后的二叉树算法(PATRICIA),即使在很大的路由表中也可提供高速的路由查找性能。这种路由引擎可以运行更快的 IP 包传递,并提供 API(应用程序接口)用于增加或删除路由信息。VxWorks 采用 Midnight 网络公司开发的标准协议包测试过所有的路由协议。

VxWorks 网络协议栈经过仔细设计,在各类应用中的性能获得较大提高。经过测试,网络吞吐量和 CPU 占有率等性能均比上一版本有 15%～20%的提高。优化措施包括:取消在 TCP 层的数据拷贝,使用 Hash 表以及改进缓冲管理方法等。

VxWorks 协议栈完全集成了 MIB－II 支持,包括接口、IP、地址解析、ICMP、TCP 及 UDP 等。VxWorks 协议软件具有以下特性:

- 支持最新的协议,如 IP multicast、CIDR 和 RFC－1323 等;
- 可配置成 IP、“IP＋UDP”和“IP＋UDP＋TCP”;
- 可作为 DHCP 服务器、DHCP 客户端和中继代理等;
- 可作为 DNS 客户端;
- 可作为 SNTP 服务器和 SNTP 客户端;
- 支持 IP 各类服务,并为 IP 转发进行过优化;
- 支持 RIP v1 和 RIP v2;
- 可选支持 OSPF;
- 具有路由策略;
- 支持 IP/ICMP/IGMP;
- 支持 ARP/代理 ARP;
- 支持 TCP 和 UDP;
- 有 BSD 4.4 兼容的 Socket 库;
- 可作为 BOOTP 客户端;
- 可作为 RPC/NFS 服务器及客户端;
- 可作为 RSH 客户端和 Telnet 服务器;
- 可作为 RLOGIN 客户端和服务器;
- 支持 PPP/SLIP/CSLIP;
- 为 TCP 连接和路由表查询进行过优化;
- 在 TCP 和 UDP 层使用了零拷贝技术;
- 新的驱动结构,支持在同一网络设备上运行多种协议;
- 集成 MIB－II 支持。

6.6.2 套接字的使用

VxWorks提供了强大的网络功能，能与其他许多主机系统进行通信。网络完全兼容4.3 BSD，也兼容SUN公司的NFS。这种广泛的协议支持在主机和VxWorks Shell目标机之间提供了无缝的工作环境，任务可通过网络向其他系统的主机存取文件(远程文件存取)，也支持远程过程调用。

VxWorks提供了如下一些网络工具完成信息传送：

- 套接字(Socket)：完成运行在VxWorks系统上或其他系统之间任务的消息传送；
- 远程过程调用(RPC)：允许任务调用另一主机(运行的系统为VxWorks或其他)上的过程；
- 远程文件存取：VxWorks任务可采用NFS、RSH、FTP和TFTP等方式远程存取主机文件；
- 文件输出；
- 远程执行命令：VxWorks任务可通过网络激活主机系统中的命令。

VxWorks系统和网络协议的接口是靠套接字来实现的。Socket规范是得到广泛应用的、开放的、支持多种协议的网络编程接口。通信的基石是套接字，对于通信的每一端，都可找到其对应的一个名字。每个正在被使用的套接字都有其类型以及与其相关的任务。套接字存在于通信域中，通信域是为了处理一般线程通过套接字通信而引进的一种抽象概念。套接字通常与同一个域中的套接字交换数据(数据交换也可能穿越域的界限，但这时一定要执行某种解释程序)。各任务使用这个域相互之间用Internet协议来进行通信。

套接字可以根据通信性质分类。应用程序一般仅在同一类的套接字间通信；不过只要底层的通信协议允许，不同类型的套接字间也可通信。用户目前使用两种套接字：流套接字(采用TCP协议)和数据报套接字(采用UDP协议)。流套接字提供了双向的、有序的、无重复且无记录边界的数据流服务；数据报套接字支持双向的数据流，但并不保证是可靠、有序且无重复的。也就是说，一个从数据报套接字接收信息的任务有可能发现信息重复了，或者与发出时的顺序不同。数据报套接字的一个重要特点是保留了记录边界。针对这一特点，数据报套接字采用了与现在许多包交换网络(如以太网)非常类似的模型。

套接字通信的最大优点是：过程间的通信是完全对等的，无论是网络中过程的定位还是主机所运行的操作系统。一般来说，流套接字提供了可靠的、面向连接的服务，应用较广泛。

套接字通信主要支持以下函数：

- socket()：创建一个套接字；
- bind()：给套接字分配名称；
- listen()：打开TCP套接字间连接；

- accept()：完成套接字间连接；
- connect()：请求连接套接字；
- shutdown()：关闭套接字间连接；
- send()：向套接字发送数据；
- recv()：从套接字接收数据；
- select()：完成同步 I/O 传输；
- read()：从套接字读取信息；
- write()：向套接字写入信息；
- ioctl()：完成对套接字的控制；
- close()：关闭套接字。

6.6.3 网络通信程序及说明

网络通信示例见程序清单 6-11。

程序清单 6-11 网络通信示例

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "serverDemo.h"
#if defined(HOST_HP)
void bzero (s, n)
    char *s;
    int n;
{
    memset (s, '\0', n);
}
#endif

/ ************************************************************************************
 * main——服务器端程序
 */
server ()
{
```

```
int              sock, snew;                   /* 套接字描述符 */
struct sockaddr_in    serverAddr;              /* 服务器地址 */
struct sockaddr_in    clientAddr;              /* 客户机地址 */
int              client_len;                   /* 客户机地址的长度 */
char             c;
extern int       errno;                        /* Unix 错误代码 */
char             buffer[256];
long             ip;
bzero (&serverAddr, sizeof (serverAddr));
bzero (&clientAddr, sizeof (clientAddr));
sock = socket (AF_INET, SOCK_STREAM, 0);
if (sock == -1) exit (1);

/* 设置端口号及通信协议 */
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(SERVER_NUM);
printf ("\nBinding SERVER, port %d\n", serverAddr.sin_port);

if (bind (sock, (struct sockaddr *)&serverAddr, sizeof (serverAddr)) == -1)
{
    printf ("bind failed, errno = %d\n", errno);
    close (sock);
    exit (1);
}
printf ("Listening to client...\n");

/* 监听是否由客户端请求连接 */
if (listen (sock, 2) == -1)
{
    printf ("listen failed\n");
    close (sock);
    exit (1);
}

/* 接受客户端的连接 */
printf ("Accepting CLIENT\n");
client_len = sizeof (clientAddr);
```

```
        snew = accept (sock, (struct sockaddr * )&clientAddr, &client_len);
        if (snew == -1)
        {
            printf ("accept failed\n");
            close (sock);
            exit (1);
        }
        ip = ntohl(clientAddr.sin_addr.s_addr);
        printf ("CLIENT: port = %d: family = %d: addr = %d: %d: %d: %d\n",
            ntohs(clientAddr.sin_port), clientAddr.sin_family,
            (ip >> 24)&0xff,
            (ip >> 16)&0xff,
            (ip >> 8)&0xff,
            (ip)&0xff
                );

        /* 进行数据通信 */
        for (;;)
        {
            bzero (buffer, 256);
            if (recv (snew, buffer, 255, 0) == 0)
            {
                /* 如果无法收到数据,则说明客户端已关闭 */
                break;
            }

            printf("Receive : %s\n",buffer);
        }

        /* 结束程序时,须关闭已打开的 Socket 套接字 */
        close (sock);
        close (snew);
        printf ("\nclient closed, goodbye! \n");
}
```

该程序运行后会在控制台上(串口)输出如下信息:

```
Binding SERVER, port 60435
Listening to client......
```

启动与该程序配套的 Windows 测试程序，设置正确的 IP 地址，选择 Connect 后即可进行通信，如图 6-23 所示。

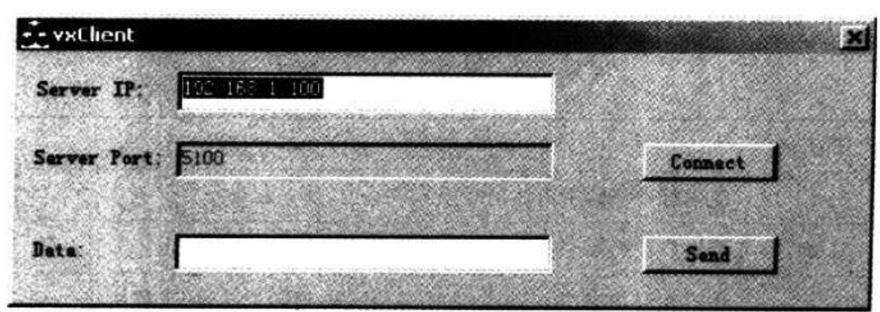


图 6-23　网络测试程序

运行在 ARM 端的服务程序会将接收到的字串数据打印到控制台上。

6.7　异常捕捉和错误处理

在程序中经常会出现一些不可避免的错误，这些错误当中有些是可以预见的，如内存分配失败、打开文件失败和除 0 等；有些则是难以预计的，如寻址错误和总线错误等。无论是哪一种错误，对于一个稳定的系统都可能造成致命的破坏，因此需对这些系统潜在的错误进行有效的检测和正确的处理。VxWorks 提供了很多方法来排除程序中的故障。

VxWorks 为错误处理提供了一个函数库——errLib，其中包含了设置和获取任务或中断错误状态的函数。大多数程序在遇到错误时都会设置一个约定的错误值，并返回 ERROR。这种机制与 Unix 的处理办法是一样的，其错误值将被设置到全局变量 errno 中。同时为了支持多任务机制以及解决中断服务程序共享上下文空间的问题，VxWorks 采取了以下两项措施：

① VxWorks 在每个任务的 TCB 中都保存独立的 errno 值，且在任务上下文切换时保存和恢复这个值，从而使得任务可在程序中随时获取和设置该错误值。

② 对于中断服务程序，VxWorks 将 errno 作为中断的进入和退出代码的一部分，所以在中断服务程序中也可直接获取和设置该值。

对于一些可以预见的错误，可在程序中进行判别，并作出合适的处理，以达到避免进一步错误发生的目的。例如：

```
if ((pNew = malloc (CHUNK_SIZE)) == NULL) return (ERROR);
```

而对于一些无法预知的非法操作，默认情况下，系统会自动保存出现非法操作时任务的状态，同时将该任务挂起；其他任务可以不受任何影响继续执行。在调试时，可通过系统提供的

调试手段以及Shell工具来处理该异常。为了更灵活地处理异常，系统还允许用户自己定义错误处理的例程，这需要借助于系统提供的信号机制。不同的异常会触发不同的信号，表6-4列举了VxWorks中所有信号的含义。

表6-4 VxWorks信号含义

信号	信号值	含义
SIGHUP	1	当运行进程的用户注销时通知该进程，使进程终止
SIGINT	2	当用户在终端中按下Ctrl+C时通知进程，使进程终止
SIGQUIT	3	用户在终端中按下Ctrl+|或Ctrl+\时通知进程，使进程终止
SIGILL	4	执行非法指令时发出的信号
SIGTRAP	5	程序调试时，程序中的陷阱指令产生的信号
SIGABRT	6	由abort系统调用所产生的中止信号
SIGEMT	7	EMT指令
SIGFPE	8	浮点操作异常产生的信号
SIGKILL	9	通过kill向进程发送的信号
SIGBUS	10	总线错误信号
SIGSEGV	11	当进程尝试访问不属于自己的内存空间而导致内存错误时产生的信号
SIGFMT	12	堆栈格式错误信号
SIGPIPE	13	管道操作时没有读，只写
SIGALRM	14	由alarm系统调用产生的时钟信号
SIGTERM	15	终端通过键盘产生的中止进程信号
SIGCNCL	16	进程取消信号
SIGSTOP	17	暂停进程
SIGTSTP	18	暂停进程(由终端的键盘产生)
SIGCONT	19	继续运行进程
SIGCHLD	20	当子进程中止时通知父进程的信号
SIGTTIN	21	前台进程读控制终端
SIGTTOU	22	后台进程写控制终端
⋮	⋮	⋮
SIGUSR1	30	用户定义的信号1
SIGUSR2	31	用户定义的信号2

表 6 - 4 中，信号 14～19 为系统任务控制有关的信号，VxWorks 中没有实现其功能，但必须定义；23～29 为系统保留的一些实时信号。信号的处理工作大致分为两部分：

- 安装信号(推荐使用 sigaction())；
- 编写信号处理函数。

使用 sigaction()安装信号可将指定的信号与某个处理例程关联起来，例如程序清单 6 - 12中将 SIGILL 信号与 sig_op 关联。

程序清单 6 - 12　信号的安装

```
struct sigaction act;
int sig  SIGILL;

sigemptyset(&act.sa_mask);
act.sa_flags = SA_SIGINFO;
act.sa_sigaction = sig_op;
if(sigaction(sig,&act,NULL) < 0)
{
    printf("install sigal error\n");
}
```

VxWorks 信号处理函数的定义为：

```
void sigHandler
(
int sig,                                  /* 信号数 */
int code,                                 /* 附加码 */
struct sigcontext * pSigContext           /* 信号产生前的任务上下文 */
)
```

程序清单 6 - 13 列出了例子中用到的 sig_op 信号处理函数的示例代码。

程序清单 6 - 13　信号处理函数

```
void sig_op(int signum,siginfo_t * info,void * myact)
{
    printf("receive signal %d", signum);
    sleep(5);
}
```

为了将多个信号处理函数合为一个，可以多次使用 sigaction()来将同一个处理函数安装到不同的信号中：

```
sigaction(SIGTERM,& sig_op,NULL);
sigaction(SIGHUP,& sig_op,NULL);
sigaction(SIGINT,& sig_op,NULL);
sigaction(SIGQUIT,& sig_op,NULL);
sigaction(SIGUSR1,& sig_op,NULL);
sigaction(SIGUSR2,& sig_op,NULL);
:
```

借助于 VxWorks 提供的信号机制，捕获和处理错误都变得非常方便。按照惯例，当被调用的程序产生错误时，调用者也会继续保留该错误值。VxWorks 的错误值采用 4 字节表示。其中：高位表明了出现错误的模块号，系统保留了模块号 0 以及与 Unix 兼容的错误定义。1～500 为 VxWorks 出现错误的具体模块编号，关于模块编号可参见 vmModNum.h；而那些大于 500 的模块号，则可断定为应用程序的。

附录

ARM 微处理器的指令系统

1. ARM 微处理器指令的分类及功能

ARM 微处理器的指令集可分为跳转指令、数据处理指令、程序状态寄存器(PSR)处理指令、加载/存储指令、协处理器指令和异常产生指令 6 类,具体的指令及功能如附录表 1 所列(表中指令为基本 ARM 指令,不包括派生的 ARM 指令)。

附录表 1　ARM 指令及功能

助记符	功能描述	助记符	功能描述
ADC	带进位加法指令	MRC	从协处理器寄存器到 ARM 寄存器的数据传输指令
ADD	加法指令	MRS	传送 CPSR 或 SPSR 的内容到通用寄存器指令
AND	逻辑"与"指令	MSR	传送通用寄存器到 CPSR 或 SPSR 的指令
B	跳转指令	MUL	32 位乘法指令
BIC	位清零指令	MLA	32 位乘加指令
BL	带返回的跳转指令	MVN	数据取反传送指令
BLX	带返回和状态切换的跳转指令	ORR	逻辑"或"指令
BX	带状态切换的跳转指令	RSB	逆向减法指令
CDP	协处理器数据操作指令	RSC	带借位的逆向减法指令
CMN	比较反值指令	SBC	带借位减法指令
CMP	比较指令	STC	协处理器寄存器写入存储器指令
EOR	逻辑"异或"指令	STM	批量内存字写入指令
LDC	存储器到协处理器的数据传输指令	STR	寄存器到存储器的数据传输指令
LDM	加载多个寄存器指令	SUB	减法指令
LDR	存储器到寄存器的数据传输指令	SWI	软件中断指令
MCR	从 ARM 寄存器到协处理器寄存器的数据传输指令	SWP	交换指令
MLA	乘加运算指令	TEQ	相等测试指令
MOV	数据传送指令	TST	位测试指令

2. 指令的条件域

当处理器工作在 ARM 状态时，几乎所有指令均根据 CPSR 中条件码的状态和指令的条件域有条件地执行。当指令的执行条件满足时，指令被执行；否则，指令被忽略。

每一条 ARM 指令包含 4 位条件码，位于指令的最高位[31:28]。条件码共有 16 种，每种条件码可用两个字符表示，这两个字符可以添加在指令助记符的后面与指令同时使用。例如：跳转指令 B 可以加上后缀 EQ 变为 BEQ，表示"相等则跳转"，即当 CPSR 中的 Z 标志置位时发生跳转。

在 16 种条件码中，只有 15 种可以使用(如附录表 2 所列)；第 16 种(1111)为系统保留，暂时不能使用。

附录表 2 指令的条件码

条件码	助记符后缀	相关标志	含 义	条件码	助记符后缀	相关标志	含 义
0000	EQ	Z	相等	1000	HI	C	无符号数大于
0001	NE	Z	不相等	1001	LS	C	无符号数小于或等于
0010	CS	C	无符号数大于或等于	1010	GE	N	带符号数大于或等于
0011	CC	C	无符号数小于	1011	LT	N	带符号数小于
0100	MI	N	负数	1100	GT	Z	带符号数大于
0101	PL	N	正数或零	1101	LE	Z	带符号数小于或等于
0110	VS	V	溢出	1110	AL	忽略	无条件执行
0111	VC	V	未溢出				

3. ARM 指令的寻址方式

所谓寻址方式就是处理器根据指令中给出的地址信息来寻找物理地址的方式。目前 ARM 指令系统支持如下几种常见的寻址方式：

(1) 立即寻址

立即寻址也称为"立即数寻址"。这是一种特殊的寻址方式，操作数本身就在指令中给出，只要取出指令也就得到了操作数。该操作数被称为"立即数"，对应的寻址方式也就称为"立即寻址"。指令示例：

```
ADD R0,R0,#1        ;R0←R0+1
ADD R0,R0,#0x3f     ;R0←R0+0x3f
```

在以上两条指令中，第二个源操作数即为立即数，要求以"#"为前缀；对于十六进制表示的立即数，还要求在"#"后加上"0x"。

(2) 寄存器寻址

寄存器寻址是利用寄存器中的数值作为操作数。这种寻址方式执行效率较高，经常被各

类微处理器采用。指令示例：

```
ADD R0,R1,R2            ;R0←R1 + R2
```

该指令的执行效果是将寄存器 R1 和 R2 的内容相加，其结果存放在寄存器 R0 中。

(3) 寄存器间接寻址

寄存器间接寻址是以寄存器中的值作为操作数的地址，而操作数本身则存放在存储器中。指令示例：

```
ADD R0,R1,[R2]          ;R0←R1 + [R2]
LDR R0,[R1]             ;R0←[R1]
STR R0,[R1]             ;[R1]←R0
```

在第一条指令中，以寄存器 R2 的值作为操作数的地址，在存储器中取得一个操作数后与 R1 相加，结果存入寄存器 R0 中。

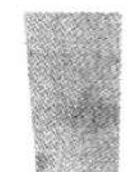
第二条指令将以 R1 的值为地址的存储器中的数据传送到 R0 中。

第三条指令将 R0 的值传送到以 R1 的值为地址的存储

(4) 基址变址寻址

基址变址寻址是将寄存器(一般称为"基址寄存器")中的 上偏移量相加，从而得到一个操作数的有效地址。变址寻址方式常用 址单元。采用变址寻址方式的指令常见有以下几种：

```
LDR R0,[R1,#4]          ;R0←[R1 + 4]
LDR R0,[R1,#4]!         ;R0←[R1 + 4],R1←R1 + 4
LDR R0,[R1],#4          ;R0←[R1],R1←R1 + 4
LDR R0,[R1,R2]          ;R0←[R1 + R2]
```



在第 1 条指令中，将寄存器 R1 的内容加上 4 形成操作数 地址，从而取得操作数并存入寄存器 R0 中。

在第 2 条指令中，将寄存器 R1 的内容加上 4 形成操作数的有效地址，从而取得操作数并存入寄存器 R0 中；然后，R1 的内容自增 4 字节。

在第 3 条指令中，以寄存器 R1 的内容作为操作数的有效地址，从而取得操作数存入寄存器 R0 中；然后，R1 的内容自增 4 字节。

在第 4 条指令中，将寄存器 R1 的内容加上寄存器 R2 的内容形成操作数的有效地址，从而取得操作数并存入寄存器 R0 中。

(5) 多寄存器寻址

采用多寄存器寻址方式，一条指令可以完成多个寄存器值的传送。这种寻址方式可以用一条指令完成最多 16 个通用寄存器值的传送。指令示例：

```
LDMIA R0,{R1,R2,R3,R4}  ;R1←[R0],R2←[R0+4],R3←[R0+8],R4←[R0+12]
```

该指令的后缀 IA 表示在每次执行完加载/存储操作后，R0 按字长度增加。因此，指令可将连续存储单元的值传送到 R1～R4。

(6) 相对寻址

与基址变址寻址方式类似，相对寻址以程序计数器 PC 的当前值为基地址，指令中的地址标号作为偏移量，将两者相加后得到操作数的有效地址。以下程序段完成子程序的调用和返回，跳转指令 BL 采用了相对寻址方式：

```
BL  NEXT                ;跳转到子程序 NEXT 处执行
 ⋮
NEXT
 ⋮
MOV PC,LR               ;从子程序返回
```

(7) 堆栈寻址

堆栈是一种数据结构，按先进后出(FILO,First In Last Out)的方式工作，使用一个称为“堆栈指针”的专用寄存器指示当前的操作位置，堆栈指针总是指向栈顶。

当堆栈指针指向最后压入堆栈的数据时，称为“满堆栈”(Full Stack)；而当堆栈指针指向下一个将要放入数据的空位置时，称为“空堆栈”(Empty Stack)。

同时，根据堆栈的生成方式，又可分为递增堆栈(Ascending Stack)和递减堆栈(Decending Stack)。当堆栈由低地址向高地址生成时，称为“递增堆栈”；当堆栈由高地址向低地址生成时，称为“递减堆栈”。这样就有 4 种类型的堆栈工作方式。ARM 微处理器支持这 4 种类型的堆栈工作方式，即：

- 满递增堆栈：堆栈指针指向最后压入的数据，由低向高生成；
- 满递减堆栈：堆栈指针指向最后压入的数据，由高向低生成；
- 空递增堆栈：堆栈指针指向下一个将要放入数据的空位置，由低向高生成；
- 空递减堆栈：堆栈指针指向下一个将要放入数据的空位置，由高向低生成。

4. ARM 指令集

(1) 跳转指令

跳转指令用于实现程序流程的跳转。在 ARM 程序中有两种方法可以实现程序流程的跳转：

- 使用专门的跳转指令；
- 直接向程序计数器 PC 写入跳转地址值。

通过向程序计数器 PC 写入跳转地址值，可以实现在 4 GB 的地址空间中任意跳转；在跳转之前结合使用“MOV LR,PC”等类似指令，可以保存将来的返回地址值，从而实现在 4 GB

连续的线性地址空间内的子程序调用。

ARM 指令集中的跳转指令可以完成从当前指令向前或向后的 32 MB 地址空间的跳转，包括的指令如附录表 3 所列。

附录表 3 跳转指令

助记符	名 称	助记符	名 称
B	跳转指令	BLX	带返回和状态切换的跳转指令
BL	带返回的跳转指令	BX	带状态切换的跳转指令

1) B 指令

格式： B{条件} 目标地址

B 指令是最简单的跳转指令。一旦遇到一个 B 指令，ARM 处理器将立即跳转到给定的目标地址，从那里继续执行。注意：存储在跳转指令中的实际值是相对当前 PC 值的一个偏移量，而不是一个绝对地址；其值由汇编器来计算(参考寻址方式中的相对寻址)。它是 24 位有符号数，左移两位后有符号扩展为 32 位，表示的有效偏移为 26 位(前后 32 MB 地址空间)。

指令示例：

```
B Label          ;程序无条件跳转到标号 Label 处执行
CMP R1,#0
BEQ Label        ;当 CPSR 寄存器中 Z 条件码置位时，程序跳转到标号 Label 处执行
```

2) BL 指令

格式： BL{条件} 目标地址

BL 是另一个跳转指令，但跳转之前，会在寄存器 R14 中保存 PC 的当前内容。因此，可通过将 R14 的内容重新加载到 PC 中，以返回到跳转指令之后的那个指令处执行。该指令是实现子程序调用的常用手段。

指令示例：

```
BL Label         ;跳转到标号 Label 处执行，同时处理器自动将当前 PC 值保存到 R14 中
```

3) BLX 指令

格式： BLX 目标地址{寄存器 Rm}

BLX 目标地址{标号}

如果使用第一种格式，Rm 的第 0 位不用作地址的一部分。Rm 的第 0 位为 1，则指令将 CPSR 中的标志 T 置位，且将目标地址的代码解释为 Thumb 代码，即进行模式切换。

对于第二种格式，BLX 指令从 ARM 指令集跳转到指令中所指定的目标地址，并将处理器的工作状态由 ARM 状态切换到 Thumb 状态。

该指令同时将 PC 的当前内容保存到寄存器 R14 中。同时，子程序的返回可通过将寄存

器 R14 值拷贝到 PC 中来完成。

4) BX 指令

格式: BX{条件} 目标地址{寄存器 Rm}

BX 指令跳转到指令中所指定的目标地址,目标地址处的指令既可是 ARM 指令,也可是 Thumb 指令。并且将根据 Rm 的第 0 位来决定模式切换,为 1 时切换为 Thumb;为 0 时切换为 ARM。

(2) 数据处理指令

数据处理指令可分为数据传送指令、算术逻辑运算指令和比较指令等,如附录表 4 所列。数据传送指令用于寄存器和存储器之间进行数据的双向传输;算术逻辑运算指令完成常用的算术与逻辑运算,该类指令不但将运算结果保存在目的寄存器中,同时更新 CPSR 中的相应条件标志位;比较指令不保存运算结果,只更新 CPSR 中相应的条件标志位。

附录表 4 数据处理指令

助记符	名 称	助记符	名 称
MOV	数据传送指令	SUB	减法指令
MVN	数据取反传送指令	SBC	带借位减法指令
CMP	比较指令	RSB	逆向减法指令
CMN	反值比较指令	RSC	带借位的逆向减法指令
TST	位测试指令	AND	逻辑"与"指令
TEQ	相等测试指令	ORR	逻辑"或"指令
ADD	加法指令	EOR	逻辑"异或"指令
ADC	带进位加法指令	BIC	位清零指令

1) MOV 指令

格式: MOV{条件}{S} 目的寄存器,源操作数

MOV 指令可完成从一个寄存器(可被移位),或将一个立即数加载到目的寄存器中。其中:S 选项决定指令的操作是否影响 CPSR 中的条件标志位;当没有 S 时,指令不更新 CPSR 中的条件标志位。

指令示例:

```
MOV R1,R0           ;将寄存器 R0 的值传送到寄存器 R1
MOV PC,R14          ;将寄存器 R14 的值传送到 PC,常用于子程序返回
MOV R1,R0,LSL#3     ;将寄存器 R0 的值左移 3 位后传送到 R1
```

2) MVN 指令

格式: MVN{条件}{S} 目的寄存器,源操作数

MVN 指令可完成从一个寄存器(可被移位),或将一个立即数加载到目的寄存器中。与 MOV 指令不同之处是在传送之前按位被取反了,即把一个被取反的值传送到目的寄存器中。

其中：S 决定指令的操作是否影响 CPSR 中的条件标志位；当没有 S 时指令不更新 CPSR 中的条件标志位。

指令示例：

```
MVN R0,#0          ;将立即数 0 取反传送到寄存器 R0 中,完成后 R0 = 0xffffffff
```

3）CMP 指令

格式：CMP{条件} 操作数 1,操作数 2

CMP 指令用于把一个寄存器中的内容与另一个寄存器中的内容或立即数进行比较,同时更新 CPSR 中的条件标志位。该指令进行一次减法运算,但不存储结果,只更改条件标志位。标志位表示的是操作数 1 与操作数 2 的关系(大,小或相等)。例如：当操作数 1 大于操作数 2 时,则此后有 GT 后缀的指令将可执行。

指令示例：

```
CMP R1,R0          ;R1 的值与 R0 的值相减,根据结果设置 CPSR 中的标志位
CMP R1,#100        ;R1 的值与 100 相减,根据结果设置 CPSR 中的标志位
```

4）CMN 指令

格式：CMN{条件} 操作数 1,操作数 2

CMN 指令用于把一个寄存器的内容与另一个寄存器的内容或立即数取反后进行比较,同时更新 CPSR 中的条件标志位。该指令实际完成操作数 1 和操作数 2 相加,并根据结果更改条件标志位。

指令示例：

```
CMN R1,R0          ;R1 的值与 R0 的值相加,并根据结果设置 CPSR 中的标志位
CMN R1,#100        ;R1 的值与 100 相加,并根据结果设置 CPSR 中的标志位
```

5）TST 指令

格式：TST{条件} 操作数 1,操作数 2

TST 指令用于把一个寄存器中的内容与另一个寄存器中的内容或立即数按位进行"与"运算,并根据运算结果更新 CPSR 中的条件标志位。操作数 1 是要测试的数据；而操作数 2 是一个位掩码。该指令一般用来检测是否设置了特定的位。

指令示例：

```
TST R1,#1          ;测试在 R1 中是否设置了最低位
TST R1,#0xffe      ;R1 的值与 0xffe 按位"与",并根据结果设置 CPSR 中的标志位
```

6）TEQ 指令

格式：TEQ{条件} 操作数 1,操作数 2

TEQ 指令用于把一个寄存器中的内容与另一个寄存器中的内容或立即数按位进行"异或"运算,并根据运算结果更新 CPSR 中的条件标志位。该指令通常用于比较操作数 1 和操作数 2 是否相等。

指令示例:

```
TEQ R1,R2              ;R1 的值与 R2 的值按位"异或",并根据结果设置 CPSR 中的标志位
```

7) ADD 指令

格式: ADD{条件}{S} 目的寄存器,操作数 1,操作数 2

ADD 指令用于把两个操作数相加,并将结果存放到目的寄存器中。操作数 1 应是一个寄存器;操作数 2 可以是一个寄存器、被移位的寄存器,或一个立即数。

指令示例:

```
ADD R0,R1,R2           ; R0 = R1 + R2
ADD R0,R1,#256         ; R0 = R1 + 256
ADD R0,R2,R3,LSL#1     ; R0 = R2 + (R3 << 1)
```

8) ADC 指令

格式: ADC{条件}{S} 目的寄存器,操作数 1,操作数 2

ADC 指令用于把两个操作数相加,再加上 CPSR 中 C 条件标志位的值,并将结果存放到目的寄存器中。它使用一个进位标志位,这样即可进行多于 32 位(如 64 位)的加法。注意:不要忘记设置 S 后缀来更改进位标志。操作数 1 应是一个寄存器;操作数 2 可以是一个寄存器、被移位的寄存器,或一个立即数。

以下指令序列完成两个 128 位数的加法。第一个数由高到低存放在寄存器 R7~R4 中;第二个数由高到低存放在寄存器 R11~R8 中;运算结果由高到低存放在寄存器 R3~R0 中。

```
ADDS R0,R4,R8          ; 加低端的字
ADCS R1,R5,R9          ; 加第 2 个字,带进位
ADCS R2,R6,R10         ; 加第 3 个字,带进位
ADC R3,R7,R11          ; 加第 4 个字,带进位
```

9) SUB 指令

格式: SUB{条件}{S} 目的寄存器,操作数 1,操作数 2

SUB 指令用于把操作数 1 减去操作数 2,并将结果存放到目的寄存器中。操作数 1 应是一个寄存器;操作数 2 可以是一个寄存器、被移位的寄存器,或一个立即数。该指令可用于有符号数或无符号数的减法运算。

指令示例:

```
SUB R0,R1,R2           ; R0 = R1 - R2
SUB R0,R1,#256         ; R0 = R1 - 256
```

```
SUB R0,R2,R3,LSL#1      ; R0 = R2 - (R3 << 1)
```

10) SBC 指令

格式： SBC{条件}{S} 目的寄存器,操作数 1,操作数 2

SBC 指令用于把操作数 1 减去操作数 2,再减去 CPSR 中 C 条件标志位的反码,并将结果存放到目的寄存器中。操作数 1 应是一个寄存器;操作数 2 可以是一个寄存器、被移位的寄存器,或一个立即数。该指令使用进位标志来表示借位,这样即可进行多于 32 位的减法。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
SUBCS R0,R1,R2      ; R0 = R1 - R2 - !C,并根据结果设置 CPSR 的进位标志
```

11) RSB 指令

格式： RSB{条件}{S} 目的寄存器,操作数 1,操作数 2

RSB 指令称为"逆向减法指令",用于把操作数 2 减去操作数 1,并将结果存放到目的寄存器中。操作数 1 应是一个寄存器;操作数 2 可以是一个寄存器、被移位的寄存器,或一个立即数。该指令可用于有符号数或无符号数的减法运算。

指令示例：

```
RSB R0,R1,R2          ; R0 = R2 - R1
RSB R0,R1,#256        ; R0 = 256 - R1
RSB R0,R2,R3,LSL#1    ; R0 = (R3 << 1) - R2
```

12) RSC 指令

格式： RSC{条件}{S} 目的寄存器,操作数 1,操作数 2

RSC 指令用于把操作数 2 减去操作数 1,再减去 CPSR 中 C 条件标志位的反码,并将结果存放到目的寄存器中。操作数 1 应是一个寄存器;操作数 2 可以是一个寄存器、被移位的寄存器,或一个立即数。该指令使用进位标志来表示借位,这样即可进行多于 32 位的减法。

指令示例：

```
RSC R0,R1,R2      ; R0 = R2 - R1 - !C
```

13) AND 指令

格式： AND{条件}{S} 目的寄存器,操作数 1,操作数 2

AND 指令用于在两个操作数上进行逻辑"与"运算,并把结果存放到目的寄存器中。操作数 1 应是一个寄存器;操作数 2 可以是一个寄存器、被移位的寄存器,或一个立即数。该指令常用于屏蔽操作数 1 的某些位。

指令示例：

```
AND R0,R0,#3            ；该指令保持 R0 的 0、1 位，其余位清 0
```

14) ORR 指令

格式： ORR{条件}{S} 目的寄存器，操作数 1，操作数 2

ORR 指令用于在两个操作数上进行逻辑"或"运算，并把结果存放到目的寄存器中。操作数 1 应是一个寄存器；操作数 2 可以是一个寄存器、被移位的寄存器，或一个立即数。该指令常用于设置操作数 1 的某些位。

指令示例：

```
ORR R0,R0,#3            ；该指令设置 R0 的 0、1 位，其余位保持不变
```

15) EOR 指令

格式： EOR{条件}{S} 目的寄存器，操作数 1，操作数 2

EOR 指令用于在两个操作数上进行逻辑"异或"运算，并把结果存放到目的寄存器中。操作数 1 应是一个寄存器；操作数 2 可是一个寄存器、被移位的寄存器，或一个立即数。该指令常用于反转操作数 1 的某些位。

指令示例：

```
EOR R0,R0,#3            ；该指令反转 R0 的 0、1 位，其余位保持不变
```

16) BIC 指令

格式： BIC{条件}{S} 目的寄存器，操作数 1，操作数 2

BIC 指令用于清除操作数 1 的某些位，并把结果存放到目的寄存器中。操作数 1 应是一个寄存器；操作数 2 可以是一个寄存器、被移位的寄存器，或一个立即数。操作数 2 为 32 位掩码，如果在掩码中设置了某一位，则清除该位；未设置的掩码位保持不变。

指令示例：

```
BIC R0,R0,#0x0B         ；该指令清除 R0 中的位 0、位 1 和位 3，其余位保持不变
```

(3) 乘法指令与乘加指令

ARM 微处理器支持的乘法指令与乘加指令共有 6 条(如附录表 5 所列)，可分为运算结果为 32 位和运算结果为 64 位两类。与前面的数据处理指令不同，指令中的所有操作数和目的寄存器必须为通用寄存器，不能对操作数使用立即数或被移位的寄存器；同时，目的寄存器和操作数 1 必须是不同的寄存器。

附录表 5　乘法指令与乘加指令

助记符	名　称	助记符	名　称
MUL	32 位乘法指令	SMLAL	64 位有符号数乘加指令
MLA	32 位乘加指令	UMULL	64 位无符号数乘法指令
SMULL	64 位有符号数乘法指令	UMLAL	64 位无符号数乘加指令

1) MUL 指令

格式： MUL{条件}{S} 目的寄存器，操作数 1，操作数 2

MUL 指令完成操作数 1 与操作数 2 的乘法运算，并把结果存放到目的寄存器中；同时可根据运算结果设置 CPSR 中相应的条件标志位。其中：操作数 1 和操作数 2 均为 32 位有符号数或无符号数。

指令示例：

```
MUL R0,R1,R2        ;R0 = R1 × R2
MULS R0,R1,R2       ;R0 = R1 × R2,同时设置 CPSR 中相应的条件标志位
```

2) MLA 指令

格式： MLA{条件}{S} 目的寄存器，操作数 1，操作数 2，操作数 3

MLA 指令完成操作数 1 与操作数 2 的乘法运算，再将乘积加上操作数 3，并把结果存放到目的寄存器中；同时可根据运算结果设置 CPSR 中相应的条件标志位。其中：操作数 1 和操作数 2 均为 32 位有符号数或无符号数。

指令示例：

```
MLA R0,R1,R2,R3     ;R0 = R1 × R2 + R3
MLAS R0,R1,R2,R3    ;R0 = R1 × R2 + R3,同时设置 CPSR 中相应的条件标志位
```

3) SMULL 指令

格式： SMULL{条件}{S} 目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2

SMULL 指令完成操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位存放到目的寄存器 Low 中，结果的高 32 位存放到目的寄存器 High 中；同时可根据运算结果设置 CPSR 中相应的条件标志位。其中：操作数 1 和操作数 2 均为 32 位有符号数。

指令示例：

```
SMULL R0,R1,R2,R3   ;R0 = (R2 × R3)的低 32 位;R1 = (R2 × R3)的高 32 位
```

4) SMLAL 指令

格式： SMLAL{条件}{S} 目的寄存器 Low，目的寄存器低 High，操作数 1，操作数 2

SMLAL 指令完成操作数 1 与操作数 2 的乘法运算，并把结果的低 32 位同目的寄存器 Low 中的值相加后又存放到目的寄存器 Low 中，结果的高 32 位同目的寄存器 High 中的值相加后又存放到目的寄存器 High 中；同时可根据运算结果设置 CPSR 中相应的条件标志位。其中：操作数 1 和操作数 2 均为 32 位有符号数。对于目的寄存器 Low，在指令执行前存放 64 位加数的低 32 位，指令执行后存放结果的低 32 位；对于目的寄存器 High，在指令执行前存放 64 位加数的高 32 位，指令执行后存放结果的高 32 位。

指令示例:

```
SMLAL R0,R1,R2,R3       ;R0 = (R2 × R3)的低 32 位 + R0;R1 = (R2 × R3)的高 32 位 + R1
```

5) UMULL 指令

格式: UMULL{条件}{S} 目的寄存器 Low,目的寄存器低 High,操作数 1,操作数 2

UMULL 指令完成操作数 1 与操作数 2 的乘法运算,并把结果的低 32 位存放到目的寄存器 Low 中,结果的高 32 位存放到目的寄存器 High 中;同时可根据运算结果设置 CPSR 中相应的条件标志位。其中:操作数 1 和操作数 2 均为 32 位无符号数。

指令示例:

```
UMULL R0,R1,R2,R3       ;R0 = (R2 × R3)的低 32 位;R1 = (R2 × R3)的高 32 位
```

6) UMLAL 指令

格式: UMLAL{条件}{S} 目的寄存器 Low,目的寄存器低 High,操作数 1,操作数 2

UMLAL 指令完成操作数 1 与操作数 2 的乘法运算,并把结果的低 32 位同目的寄存器 Low 中的值相加后又存放到目的寄存器 Low 中,结果的高 32 位同目的寄存器 High 中的值相加后又存放到目的寄存器 High 中;同时可根据运算结果设置 CPSR 中相应的条件标志位。其中:操作数 1 和操作数 2 均为 32 位无符号数。对于目的寄存器 Low,在指令执行前存放 64 位加数的低 32 位,指令执行后存放结果的低 32 位;对于目的寄存器 High,在指令执行前存放 64 位加数的高 32 位,指令执行后存放结果的高 32 位。

指令示例:

```
UMLAL R0,R1,R2,R3       ;R0 = (R2 × R3)的低 32 位 + R0;R1 = (R2 × R3)的高 32 位 + R1
```

(4) 程序状态寄存器访问指令

ARM 微处理器支持程序状态寄存器访问指令,用于在程序状态寄存器和通用寄存器之间传送数据。程序状态寄存器访问指令包括:程序状态寄存器到通用寄存器的数据传送指令(MRS)和通用寄存器到程序状态寄存器的数据传送指令(MSR)。

1) MRS 指令

格式: MRS{条件} 通用寄存器,程序状态寄存器(CPSR 或 SPSR)

MRS 指令用于将程序状态寄存器中的内容传送到通用寄存器中。一般用于以下两种情况:

① 须改变程序状态寄存器中的内容时,可用 MRS 将程序状态寄存器中的内容读入通用寄存器,修改后再写回程序状态寄存器。

② 在异常处理或进程切换时,须保存程序状态寄存器的值,可先用该指令读出程序状态寄存器中的值,然后保存。

指令示例:

```
MRS R0,CPSR             ;传送 CPSR 的内容到 R0
```

```
MRS R0,SPSR          ;传送 SPSR 的内容到 R0
```

2) MSR 指令

格式：MSR{条件} 程序状态寄存器(CPSR 或 SPSR)_<域>,操作数

MSR 指令用于将操作数的内容传送到程序状态寄存器的特定域中。其中：操作数可为通用寄存器或立即数；<域>用于设置程序状态寄存器中须操作的位。32 位程序状态寄存器可分为 4 个域：

- 位[31:24]：条件标志位域，用 f 表示；
- 位[23:16]：状态位域，用 s 表示；
- 位[15:8]：扩展位域，用 x 表示；
- 位[7:0]： 控制位域，用 c 表示。

该指令通常用于恢复或改变程序状态寄存器中的内容。在使用时，一般要在 MSR 指令中指明将要操作的域。

指令示例：

```
MSR CPSR,R0          ;传送 R0 的内容到 CPSR
MSR SPSR,R0          ;传送 R0 的内容到 SPSR
MSR CPSR_c,R0        ;传送 R0 的内容到 SPSR,但仅修改 CPSR 中的控制位域
```

(5) 加载/存储指令

ARM 微处理器支持加载/存储指令，用于在寄存器和存储器之间传送数据。加载指令用于将存储器中的数据传送到寄存器；存储指令则完成相反的操作。常用的加载存储指令包括：字数据加载指令(LDR)、字节数据加载指令(LDRB)、字数据加载指令(LDRH)、字数据存储指令(STR)、字节数据存储指令(STRB)和半字数据存储指令(STRH)。

1) LDR 指令

格式：LDR{条件} 目的寄存器,<存储器地址>

LDR 指令用于从存储器中将一个 32 位的字数据传送到目的寄存器中。该指令通常用于从存储器中读取 32 位字数据到通用寄存器，然后对数据进行处理。当程序计数器 PC 作为目的寄存器时，指令从存储器中读取的字数据被当作目的地址，从而实现程序流程的跳转。该指令在程序设计中较常用，且寻址方式灵活多样。

指令示例：

```
LDR R0,[R1]          ;将存储器地址为 R1 的字数据读入寄存器 R0
LDR R0,[R1,R2]       ;将存储器地址为 R1 + R2 的字数据读入寄存器 R0
LDR R0,[R1,#8]       ;将存储器地址为 R1 + 8 的字数据读入寄存器 R0
LDR R0,[R1,R2] !     ;将存储器地址为 R1 + R2 的字数据读入寄存器 R0,并将新地址 R1 + R2 写入 R1
LDR R0,[R1,#8]!      ;将存储器地址为 R1 + 8 的字数据读入寄存器 R0,并将新地址 R1 + 8 写入 R1
```

```
LDR R0,[R1],R2              ;将存储器地址为 R1 的字数据读入寄存器 R0,并将新地址 R1 + R2 写入 R1
LDR R0,[R1,R2,LSL #2]!     ;将存储器地址为 R1 + R2 × 4 的字数据读入寄存器 R0,并将新地址 R1 + R2 × 4
                            ;写入 R1
LDR R0,[R1],R2,LSL #2      ;将存储器地址为 R1 的字数据读入寄存器 R0,并将新地址 R1 + R2 × 4 写入 R1
```

2) LDRB 指令

格式： LDR{条件}B 目的寄存器,<存储器地址>

LDRB 指令用于从存储器中将一个 8 位的字节数据传送到目的寄存器中,同时将寄存器的高 24 位清 0。该指令通常用于从存储器中读取 8 位的字节数据到通用寄存器,然后对数据进行处理。当程序计数器 PC 作为目的寄存器时,指令从存储器中读取的字数据被当作目的地址,从而实现程序流程的跳转。

指令示例：

```
LDRB R0,[R1]               ;地址为 R1 的字节数据读入 R0,并将 R0 的高 24 位清 0
LDRB R0,[R1,#8]            ;地址为 R1 + 8 的字节数据读入 R0,并将 R0 的高 24 位清 0
```

3) LDRH 指令

格式： LDR{条件}H 目的寄存器,<存储器地址>

LDRH 指令用于从存储器中将一个 16 位的半字数据传送到目的寄存器中,同时将寄存器的高 16 位清 0。该指令通常用于从存储器中读取 16 位的半字数据到通用寄存器,然后对数据进行处理。当程序计数器 PC 作为目的寄存器时,指令从存储器中读取的字数据被当作目的地址,从而实现程序流程的跳转。

指令示例：

```
LDRH R0,[R1]               ;地址为 R1 的半字数据读入 R0,并将 R0 的高 16 位清 0
LDRH R0,[R1,#8]            ;地址为 R1 + 8 的半字数据读入 R0,并将 R0 的高 16 位清 0
LDRH R0,[R1,R2]            ;地址为 R1 + R2 的半字数据读入 R0,并将 R0 的高 16 位清 0
```

4) STR 指令

格式： STR{条件} 源寄存器,<存储器地址>

STR 指令用于从源寄存器中将一个 32 位的字数据传送到存储器中。该指令在程序设计中较常用,且寻址方式灵活多样,使用方式可参考指令 LDR。

指令示例：

```
STR R0,[R1],#8             ;将 R0 中的字数据写入以 R1 为地址的存储器中,并将新地址 R1 + 8 写入 R1
STR R0,[R1,#8]             ;将 R0 中的字数据写入以 R1 + 8 为地址的存储器中
```

5) STRB 指令

格式： STR{条件}B 源寄存器,<存储器地址>

STRB 指令用于从源寄存器中将一个 8 位的字节数据传送到存储器中。该字节数据为源寄存器中的低 8 位。

指令示例：

```
STRB R0,[R1]        ;将寄存器 R0 中的字节数据写入以 R1 为地址的存储器中
STRB R0,[R1,#8]     ;将寄存器 R0 中的字节数据写入以 R1 + 8 为地址的存储器中
```

6) STRH 指令

格式： STR{条件}H 源寄存器，<存储器地址>

STRH 指令用于从源寄存器中将一个 16 位的半字数据传送到存储器中。该半字数据为源寄存器中的低 16 位。

指令示例：

```
STRH R0,[R1]        ;将寄存器 R0 中的半字数据写入以 R1 为地址的存储器中
STRH R0,[R1,#8]     ;将寄存器 R0 中的半字数据写入以 R1 + 8 为地址的存储器中
```

(6) 批量数据加载/存储指令

ARM 微处理器所支持批量数据加载/存储指令可一次在一片连续的存储器单元和多个寄存器之间传送数据。批量加载指令用于将一片连续存储器中的数据传送到多个寄存器；批量数据存储指令则完成相反的操作。常用的加载存储指令包括：批量数据加载指令(LDM)和批量数据存储指令(STM)。

格式： LDM(或 STM){条件}{类型} 基址寄存器{!}，寄存器列表{∧}

LDM(或 STM)指令用于从由基址寄存器所指示的一片连续存储器到寄存器列表所指示的多个寄存器传送数据，该指令的常用于将多个寄存器的内容入栈或出栈。其中，{类型}分为几种情况，如附录表 6 所列。

附录表 6 {类型}常见情况

类 型	描 述	类 型	描 述
IA	每次传送后地址加 1	FD	满递减堆栈
IB	每次传送前地址加 1	ED	空递减堆栈
DA	每次传送后地址减 1	FA	满递增堆栈
DB	每次传送前地址减 1	EA	空递增堆栈

{!}为可选后缀，若选用该后缀，则当数据传送完毕之后，将最后的地址写入基址寄存器；否则，基址寄存器的内容不改变。基址寄存器不允许为 R15；寄存器列表可为 R0～R15 的任意组合。

{∧}为可选后缀，当指令为 LDM 且寄存器列表中包含 R15 时，若选用该后缀，则表示除了正常的数据传送之外，还将 SPSR 拷贝到 CPSR。同时，该后缀还表示传入或传出的是用户模式下的寄存器，而不是当前模式下的寄存器。

指令示例：

```
STMFD R13!,{R0,R4-R12,LR}      ;将寄存器(R0,R4～R12,LR)存入堆栈
LDMFD R13!,{R0,R4-R12,PC}      ;将堆栈内容恢复到寄存器(R0,R4～R12,LR)
```

(7) 数据交换指令

ARM 微处理器所支持的数据交换指令能在存储器和寄存器之间交换数据。数据交换指令有如下两条：字数据交换指令(SWP)和字节数据交换指令(SWPB)。

1) SWP 指令

格式： SWP{条件} 目的寄存器，源寄存器 1，[源寄存器 2]

SWP 指令用于将源寄存器 2 所指向的存储器中的字数据传送到目的寄存器中；同时将源寄存器 1 中的字数据传送到源寄存器 2 所指向的存储器中。显然，当源寄存器 1 和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

```
SWP R0,R1,[R2]      ;将 R2 所指向的存储器中的字数据传送到 R0,同时将 R1 中的字数据传送到
                    ;R2 所指向的存储单元
SWP R0,R0,[R1]      ;该指令完成 R1 所指向的存储器中的数据与 R0 的数据交换
```

2) SWPB 指令

格式： SWP{条件}B 目的寄存器，源寄存器 1，[源寄存器 2]

SWPB 指令用于将源寄存器 2 所指向的存储器中的字节数据传送到目的寄存器中，目的寄存器的高 24 位清 0；同时将源寄存器 1 中的字节数据传送到源寄存器 2 所指向的存储器中。显然，当源寄存器 1 和目的寄存器为同一个寄存器时，指令交换该寄存器和存储器的内容。

指令示例：

```
SWPB R0,R1,[R2]     ;将 R2 所指向的存储器中的字节数据传送到 R0,R0 的高 24 位清 0,同时将
                    ;R1 中的低 8 位数据传送到 R2 所指向的存储单元
SWPB R0,R0,[R1]     ;该指令完成将 R1 所指向的存储器中的字节数据与 R0 中的低 8 位数据交换
```

(8) 移位操作

ARM 微处理器内嵌的桶型移位器(Barrel Shifter)，支持数据的各种移位操作。移位操作在 ARM 指令集中不作为单独的指令使用，它只能作为指令格式中的一个字段，在汇编语言中表示为指令中的选项。例如：数据处理指令的第二个操作数为寄存器时，即可加入移位操作选项对它进行各种移位操作。移位操作包括如下 6 种类型：逻辑左移(LSL)、算术左移(ASL)、逻辑右移(LSR)、算术右移(ASR)、循环右移(ROR)和带扩展的循环右移(RRX)。

其中：ASL 和 LSL 是等价的，可以自由互换。

1) LSL(或 ASL)操作

格式: 通用寄存器,LSL(或 ASL) 操作数

LSL(或 ASL)可对通用寄存器中的内容进行逻辑(或算术)左移操作,按操作数所指定的数量向左移位,低位用 0 来填充。其中:操作数可是通用寄存器,也可是立即数(0~31)。

操作示例:

```
MOV R0, R1, LSL #2        ;将 R1 中的内容左移两位后传送到 R0 中
```

2) LSR 操作

格式: 通用寄存器,LSR 操作数

LSR 可完成对通用寄存器中的内容进行右移的操作,按操作数所指定的数量向右移位,左端用零来填充。其中操作数可是通用寄存器,也可是立即数(0~31)。

操作示例:

```
MOV R0, R1, LSR #2        ;将 R1 中的内容右移两位后传送到 R0 中,左端用 0 来填充
```

3) ASR 操作

格式: 通用寄存器,ASR 操作数

ASR 可对通用寄存器中的内容进行右移操作,按操作数所指定的数量向右移位,左端用第 31 位的值来填充。其中:操作数可是通用寄存器,也可是立即数(0~31)。

操作示例:

```
MOV R0, R1, ASR #2        ;将 R1 的内容右移两位后传送到 R0,左端用第 31 位的值来填充
```

4) ROR 操作

格式: 通用寄存器,ROR 操作数

ROR 可对通用寄存器中的内容进行循环右移操作,按操作数所指定的数量向右循环移位,左端用右端移出的位来填充。其中:操作数可是通用寄存器,也可是立即数(0~31)。显然,当进行 32 位循环右移操作时,通用寄存器中的值不改变。

操作示例:

```
MOV R0, R1, ROR #2        ;将 R1 中的内容循环右移两位后传送到 R0 中
```

5) RRX 操作

格式: 通用寄存器,RRX 操作数

RRX 可对通用寄存器中的内容进行带扩展的循环右移操作,按操作数所指定的数量向右循环移位,左端用进位标志位 C 来填充。其中:操作数可是通用寄存器,也可是立即数(0~31)。

操作示例:

```
MOV R0, R1, RRX #2        ;将 R1 中的内容进行带扩展的循环右移两位后传送到 R0 中
```

(9) 协处理器指令

ARM微处理器可支持多达16个协处理器,用于各种协处理操作。在程序执行的过程中,每个协处理器只执行针对自身的协处理指令,而忽略ARM处理器和其他协处理器的指令。

ARM的协处理器指令主要用于ARM处理器初始化ARM协处理器的数据处理操作,以及在ARM处理器的寄存器和协处理器的寄存器之间,或ARM协处理器的寄存器和存储器之间传送数据。ARM协处理器指令如附录表7所列。

附录表7 ARM协处理器指令

助记符	名称
CDP	协处理器数据操作指令
LDC	协处理器数据加载指令
STC	协处理器数据存储指令
MCR	ARM处理器寄存器到协处理器寄存器的数据传送指令
MRC	协处理器寄存器到ARM处理器寄存器的数据传送指令

1) CDP指令

格式: CDP{条件} 协处理器编码,协处理器操作码1,目的寄存器,源寄存器1,源寄存器2,协处理器操作码2

CDP指令用于ARM处理器通知ARM协处理器执行特定的操作。若协处理器不能成功完成该操作,则产生未定义指令异常。其中协处理器操作码1和协处理器操作码2为协处理器将要执行的操作;目的寄存器和源寄存器均为协处理器的寄存器,指令不涉及ARM处理器的寄存器和存储器。

指令示例:

```
CDP P3,2,C12,C10,C3,4        ;该指令完成协处理器P3的初始化
```

2) LDC指令

格式: LDC{条件}{L} 协处理器编码,目的寄存器,[源寄存器]

LDC指令用于将源寄存器所指向的存储器中的字数据传送到目的寄存器中。若协处理器不能成功完成传送操作,则产生未定义指令异常。其中:{L}选项表示指令为长读取操作,例如用于双精度数据的传输。

指令示例:

```
LDC P3,C4,[R0]        ;将ARM处理器的寄存器R0所指向的存储器中的字数据传送到协处理器
                      ;P3的寄存器C4中
```

3) STC指令

格式: STC{条件}{L} 协处理器编码,源寄存器,[目的寄存器]

STC 指令用于将源寄存器中的字数据传送到目的寄存器所指向的存储器中。若协处理器不能成功完成传送操作，则产生未定义指令异常。其中：{L}选项表示指令为长读取操作，例如用于双精度数据的传输。

指令示例：

```
STC P3,C4,[R0]          ;将协处理器 P3 的寄存器 C4 中的字数据传送到 ARM 处理器的寄存器 R0 所指
                        ;向的存储器中
```

4）MCR 指令

格式： MCR{条件} 协处理器编码，协处理器操作码 1，源寄存器，目的寄存器 1，目的寄存器 2，协处理器操作码 2

MCR 指令用于将 ARM 处理器寄存器中的数据传送到协处理器寄存器中。若协处理器不能成功完成操作，则产生未定义指令异常。其中：协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作；源寄存器为 ARM 处理器的寄存器；目的寄存器 1 和目的寄存器 2 均为协处理器的寄存器。

指令示例：

```
MCR P3,3,R0,C4,C5,6     ;该指令将 ARM 处理器寄存器 R0 中的数据传送到协处理器 P3 的寄存器 C4
                        ;和 C5 中
```

5）MRC 指令

格式： MRC{条件} 协处理器编码，协处理器操作码 1，目的寄存器，源寄存器 1，源寄存器 2，协处理器操作码 2

MRC 指令用于将协处理器寄存器中的数据传送到 ARM 处理器寄存器中。若协处理器不能成功完成操作，则产生未定义指令异常。其中：协处理器操作码 1 和协处理器操作码 2 为协处理器将要执行的操作；目的寄存器为 ARM 处理器的寄存器；源寄存器 1 和源寄存器 2 均为协处理器的寄存器。

指令示例：

```
MRC P3,3,R0,C4,C5,6     ;该指令将协处理器 P3 的寄存器中的数据传送到 ARM 处理器寄存器中
```

（10）异常产生指令

ARM 微处理器所支持的异常指令包括：软件中断指令（SWI）和断点中断指令（BKPT）。

1）SWI 指令

格式： SWI{条件} 24 位立即数

SWI 指令用于产生软件中断，以便用户程序能调用操作系统的系统例程；操作系统在 SWI 的异常处理程序中提供相应的系统服务。指令中 24 位立即数指定用户程序调用系统例程的类型，相关参数通过通用寄存器传递。当指令中 24 位立即数被忽略时，用户程序调用系

统例程的类型由通用寄存器 R0 中的内容决定；同时，参数通过其他通用寄存器传递。

指令示例：

```
SWI 0x02                ;该指令调用操作系统编号为 02 的系统例程
```

2) BKPT 指令

格式： BKPT 16 位立即数

BKPT 指令产生软件断点中断，可用于程序的调试。

5. Thumb 指令及应用

为兼容数据总线宽度为 16 位的应用系统，ARM 体系结构除了支持执行效率很高的 32 位 ARM 指令集以外，同时支持 16 位的 Thumb 指令集。Thumb 指令集是 ARM 指令集的一个子集，允许指令编码为 16 位长度。与等价的 32 位代码相比，Thumb 指令集在保留 32 位代码优势的同时，大大节省了系统的存储空间。

所有 Thumb 指令都有相对应的 ARM 指令，而且 Thumb 的编程模型也对应于 ARM 的编程模型。在应用程序的编写过程中，只要遵循一定的调用规则，Thumb 子程序和 ARM 程序即可互相调用。当处理器执行 ARM 程序段时，称 ARM 处理器处于 ARM 工作状态；当处理器执行 Thumb 程序段时，称 ARM 处理器处于 Thumb 工作状态。与 ARM 指令集相比，Thumb 指令集中数据处理指令的操作数仍为 32 位，指令地址也为 32 位，但 Thumb 指令集为实现 16 位的指令长度，舍弃了 ARM 指令集的一些特性。例如：大多数 Thumb 指令是无条件执行的，而几乎所有 ARM 指令都是有条件执行的，且大多数 Thumb 数据处理指令的目的寄存器与其中一个源寄存器相同。

由于 Thumb 指令的长度为 16 位，即只用 ARM 指令一半的位数来实现同样的功能，因此，要实现特定的程序功能，所需的 Thumb 指令的条数比 ARM 指令多。在一般情况下，Thumb 指令与 ARM 指令的时间效率和空间效率关系为：

- Thumb 代码所需的存储空间为 ARM 代码的 60%～70%；
- Thumb 代码使用的指令比 ARM 代码多 30%～40%；
- 若使用 32 位存储器，ARM 代码比 Thumb 代码快约 40%；
- 若使用 16 位存储器，Thumb 代码比 ARM 代码快 40%～50%；
- 与 ARM 代码相比，使用 Thumb 代码，存储器的功耗会降低约 30%。

显然，ARM 指令集和 Thumb 指令集各有优点。若对系统的性能有较高要求，则应使用 32 位的存储系统和 ARM 指令集；若对系统的成本及功耗有较高要求，则应使用 16 位的存储系统和 Thumb 指令集。当然，若两者结合使用，充分发挥其各自优点，会取得更好的效果。

参考文献

1 杜春雷. ARM体系结构与编程. 北京:清华大学出版社, 2003

2 马忠梅,等. ARM嵌入式处理器结构与应用. 北京:北京航空航天大学出版社, 2002

3 探矽工作室. 嵌入式系统开发圣经. 北京:中国青年出版社, 2002

4 陈赜,等. ARM9嵌入式技术及Linux高级实践教程. 北京:北京航空航天大学出版社, 2005

5 吕锋,等. VxWorks高级程序设计. 北京:清华大学出版社, 2003

6 孔祥营,柏桂枝. 嵌入式实时操作系统VxWorks及其开发环境Tornado. 北京:中国电力出版社, 2002

7 VxWorks下设备驱动程序及BSP开发指南. 北京:中国电力出版社, 2004

8 WindRiver Systems Inc. VxWorks BSP Developer's Reference

9 WindRiver Systems Inc. VxWorks kernel programmers guide

10 WindRiver Systems Inc. VxWorks application programmers guide

11 WindRiver Systems Inc. VxWorks for ARM architecture supplement

12 Free Software Foundation. GDB User's Guide