嵌入式 VxWorks 系统开发与应用

王学龙　编著

人 民 邮 电 出 版 社

图书在版编目（CIP）数据
嵌入式 VxWorks 系统开发与应用 / 王学龙编著. —北京：人民邮电出版社，2003.10
ISBN 7-115-11575-3

Ⅰ. 嵌… Ⅱ. 王… Ⅲ. 实时操作系统，VxWorks Ⅳ. TP316.2

中国版本图书馆 CIP 数据核字（2003）第 081796 号

内容提要

本书详细介绍了当今流行的嵌入式操作系统 VxWorks，首先概括 VxWorks 操作系统的基本知识，如任务管理、任务间通信机制、内存管理以及定时管理等内容，说明了嵌入式操作系统的实现关键。然后，结合作者多年的嵌入式系统的开发应用经验，详细阐述了 VxWorks 系统中 BSP 和应用的开发技巧，并提供了多个应用实例及分析设计。

本书适用于嵌入式系统开发人员作为参考手册使用。

嵌入式 VxWorks 系统开发与应用

◆ 编　　著　王学龙
责任编辑　刘　浩

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街 14 号
邮编　100061　　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
读者热线：010-67180876
北京汉魂图文设计有限公司制作
北京　　印刷厂印刷
新华书店总店北京发行所经销

◆ 开本：787×1092　1/16
印张：21
字数：509 千字　　2003 年 9 月第 1 版
印数：1 – 0 000 册　　2003 年 9 月北京第 1 次印刷

ISBN 7-115-11451-X/TP・3523

定价：36.00 元

前　言

当前企业应用最广泛的三大嵌入式操作系统是 pSOS、VxWorks 和 Embedded Linux。目前，大批大型高科技企业以及为数众多的中小型企业，甚至高等院校的实验室，都将使用重点放在 pSOS、VxWorks 系统上。尤其是 VxWorks 系统，已经得到了非常广泛的应用，利用嵌入式 VxWorks 系统开发出来的大型移动通信设备、ATM 交换设备、IP 交换设备、甚至医药仪器，已经在市场上稳定运行。国内，这方面的开发起步较晚，大多数开发人员处于自己摸索阶段，因此笔者想通过对自己多年开发经验的总结，帮助开发人员更好地理解 VxWorks 系统，更快地开发产品。

本书包含 7 章。

- 第 1 章介绍 VxWorks 及其开发环境 Tornado 的特点。
- 第 2 章介绍 VxWorks 系统的基本理论，全面描述 VxWorks 系统的任务管理、内存机制、中断机制、定时机制、时间机制、系统调用，并且结合自己的开发经验将描述的重点放在常用的内核机理上，便于读者迅速掌握 VxWorks 系统的基本知识。
- 第 3 章介绍 VxWorks 系统 BSP 的基本概念，开发 BSP 是开发 VxWorks 系统的关键。本章讲述如何在 VxWorks 系统上开发 BSP，并且详细探讨 BSP 与嵌入式系统开发的关系。这是开发 BSP 的入门知识。
- 第 4 章详尽介绍一个具体的 BSP 开发实例。考虑到开发一个具体的 BSP 及其驱动程序具有相当难度，笔者在本章介绍自己曾开发过的一块硬件主板的 BSP 的开发全过程。向读者展示设计 BSP 的过程，以及此 BSP 的部分代码，同时对其做详细的分析。
- 第 5 章简单介绍 VxWorks 系统的开发环境 Tornado，此环境是 VxWorks 系统的应用程序的开发环境，是组建 VxWorks 系统工程的管理中心。此部分的内容是为第 7 章阐述 VxWorks 系统应用开发实例做准备的。
- 第 6 章介绍 VxWorks 支持 GNU 的 Gcc 编译器，为了让读者对 Makefile 以及组建工程有较好的认识，这里举例向读者介绍 Makefile 的编写方法。
- 第 7 章用较大的篇幅向读者展示 VxWorks 系统的应用开发，介绍诸多应用程序的开发范例，这些开发技巧在用户开发过程中是非常实用的，完全可以直接使用或者移植到用户的实际开发中。

希望读者能够在透彻理解嵌入式系统原理的基础上，把握当前嵌入式系统应用的巨大商机，尽早加入到开发我国具有自主知识产权的嵌入式系统中，为我国的科技发展和繁荣富强作出更大贡献。

由于本人水平有限，再加上时间仓促，书中难免有不当之处，敬请读者见谅并提出批评指正意见。读者在学习本书的过程中，如有意见或发现问题，请发信至 ridgel@sina.com。

作者

2003 年 9 月

目　录

第1章　概　　述

在16位嵌入式系统应用中，由于CPU资源量较少，任务比较简单，程序员可以在应用程序中自己管理CPU资源，而不一定需要专用的系统软件。如果嵌入式系统比较复杂并且采用32位CPU时，情况就不同了。32位CPU的资源量非常大，寻址可以达到4GB空间，处理能力也非常强大，可以实现实时多任务并发处理，如果仍然沿用手工编制CPU管理程序，面对复杂应用，就很难发挥出32位CPU的处理能力，开发出高效、可靠的应用系统。

操作系统是整个系统应用的基础，它既要管理硬件又要为软件提供应用接口，所以操作系统的任何缺陷都可能引起严重而且不可预知的问题，如果每个嵌入式应用系统，从系统软件到应用软件都需要开发者自己完成，势必造成人力资源的浪费，延长开发周期，增加开发成本。如果基于嵌入式系统直接开发应用，则可很好地解决这些问题。

1.1　嵌入式实时操作系统VxWorks

VxWorks 是专门为实时嵌入式系统设计开发的操作系统软件，为程序员提供了高效的实时任务调度、中断管理、实时的系统资源以及实时的任务间通信。程序员可以将尽可能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。1983年，VxWorks成功推出以来，已顺利应用到航空、航天、医疗、通信等领域，并且在嵌入式领域占有一定的市场份额。目前，VxWorks 已经成为实际上的工业标准和军用标准，大量软硬件厂家都提供基于 VxWorks的扩展组件，因此，VxWorks 几乎可以在各种 CPU 硬件平台上提供统一的接口和一致的运行特征，应用程序不用做太多的改动就可以运行在各种CPU上，为程序员提供了一致的开发、运行环境，避免了重复劳动。

VxWorks 是一种功能强大而且复杂的操作系统，仅仅依靠人工编程调试，很难发挥它的功能并设计出可靠、高效的嵌入式系统，必须有与之相适应的开发工具。Tornado 就是为开发基于 VxWorks 的应用系统而提供的集成开发环境，Tornado 中包含的工具管理软件，可以将用户自己的代码与VxWorks的核心系统有效地组合起来，从而轻松、可靠地完成嵌入式应用开发。

1.1.1　VxWorks的应用领域

嵌入式VxWorks系统的主要应用领域如表1-1所示。

表1-1　　嵌入式VxWorks系统的主要应用领域

应用领域	示　　例
数据网络	以太网交换机、路由器、网桥、网络集线器、远程接入服务器、异步传输模式交换机、帧中继交换机

续表

应用领域	示　例
远程通信	电信专用分组交换机（PBX）和自动呼叫分配器（ACDs），CD 交换系统，蜂窝电话系统、xDSL 和电缆调制解调器
医疗设备	核磁共振扫描仪（MRI）、正电子成像扫描仪、放射理疗设备、床头监护器
消费电子	个人数字助理、机顶盒、数字电话机、应答机、可视电话、声频设备、交互式设备
交通运输	汽车发动机控制系统、汽车导航系统、交通信号控制系统、高速火车控制系统、防滑测试系统、机载娱乐设备
计算机外围设备	网络计算机、X 终端、冗余磁盘阵列数据存储系统（RAID）、I/O 控制系统
数字图像	打印机、数字复印机、传真机、多功能外围设备、数字相机
工业	机器人、测试测量设备、过程控制系统、计算机数值控制设备
航空	飞行仿真、航空机舱管理系统、卫星跟踪系统、航空电子设备
多媒体	专业视频编辑系统、电视会议

1.1.2　VxWorks 系统的特点

用户在开发包含复杂的 32 位嵌入式处理器的产品时,需要一个用来连接产品应用程序和底层硬件的操作系统。这种操作系统应该具有以下重要特点：

- 可靠性
- 高实时性
- 可裁剪性
- 可协同工作

VxWorks 操作系统是美国 WindRiver 公司于 1983 年设计开发出的一种嵌入式实时操作系统（RTOS),是 Tornado 嵌入式开发环境的关键组成部分。VxWorks 具有多达 1800 多个应用程序接口（API），适用范围较广，可用于简单的应用开发也可用于复杂的产品设计。它可靠性高，可以用于从防抱死刹车系统到星际探索的关键任务，而且具有高度的适用性，几乎可以用于所有流行的 32 位 CPU 平台。

VxWorks 操作系统的基本体系结构如图 1-1 所示。

VxWorks 实时操作系统包括微内核 WIND、高级网络支持、强有力的文件系统和 I/O 管理、C++和其他标准等核心功能。这些核心功能还可以与 WindRiver 公司的其他产品以及 WindRiver 公司 320 个合作伙伴的产品联合使用。

1．高性能的微内核设计

微内核的主要特点（Wind Microkenel）有如下几点。

- 高效的任务管理：数目无明显限制的多任务，具有 256 个优先级；具有优先级抢占和时间片轮转调度；快速、准确的上下文切换；快速灵活的任务间通讯；3 种信号量，包括二进制、计数、有优先级继承特性的互斥信号量；POSIX 管道、消息队列和信号
- 高度可裁剪性
- 动态连接和模块加载

- 高效的中断和异常事件处理
- 优化浮点支持
- 动态内存管理
- 时钟和记时工具

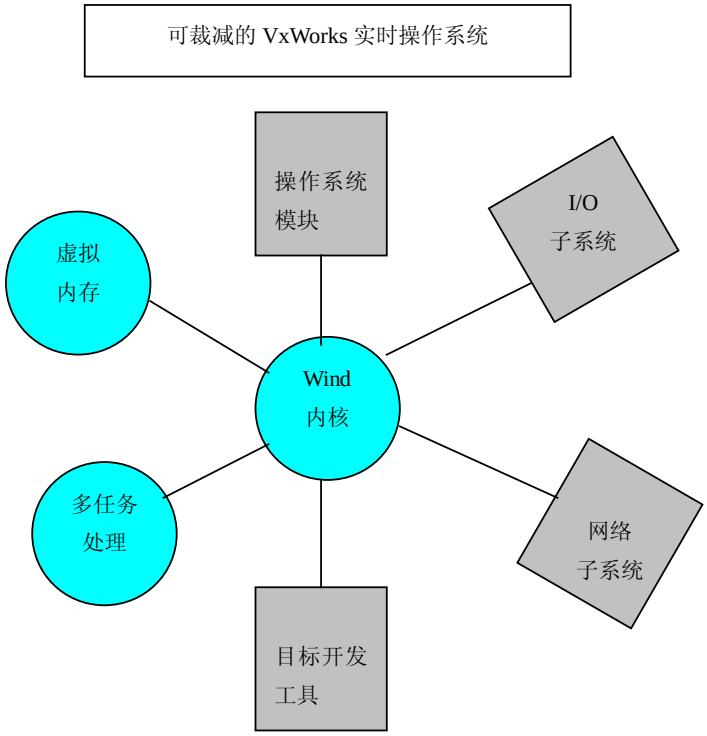

图 1-1　VxWorks 基本体系结构

2．良好的可移植性

嵌入式应用通常需要将操作系统和应用程序方便地移植到目标硬件中，如果能够将依赖于硬件的低级代码和高级的应用程序和操作系统区分开来，移植工作将会变得非常简单。VxWorks 将依赖于硬件的低级代码设计成板级支持包（BSP）。任何一个要运行 VxWorks 的硬件主板都需要相应的板级支持包，有了板级支持包的支持，移植高级代码时，只要改变相应的依赖于硬件的板级支持包即可，无须修改操作系统和应用程序。

3．良好的可裁剪性

VxWorks 的可裁剪性，可以使开发者根据自己的应用程序需要，而不是根据操作系统的需要来分配系统资源。从需要几 KB 字节内存的深层嵌入式设计到需要更多的操作系统功能的复杂的高端系统，开发者可以在 300 多个独立的模块中选择。而且，这些模块本身也是可裁剪的，所以 VxWorks 可以为用户提供粒度极小的运行环境配置。

所有这些配置选项可以通过 Tornado 的项目工具图形接口轻易地选择，或者使用 Tornado 的自动裁剪特性，自动地分析应用程序代码并选择相应的选项。

4．网络组件

VxWorks 是支持工业标准 TCP/IP 网络协议族的实时操作系统。包括：

- IP、IGMP、CIDR、TCP、UDP、ARP、RARP、RIPV1/V2
- Standard Berkeley Sockets
- Z-bufs、NFS、RPC
- PPP、BOOTP、DNS、DHCP、TFTP、FTP
- RLOGIN、TELNET、RSH

WindRiver 还支持可选的 WindNet 产品，包括 SNMP、OSPF、STREAMS 等。WindRiver 通过提供工业级最广泛的网络开发环境来加强这些核心技术，这主要是通过 WindLink for Tornado 伙伴计划来实现的。高级的网络解决方案包括：

- ATM、SMDS、FRAME RELAY、ISDN、SS7、X.25、V5 等广域网协议
- IPX/SPX、APPLETALK、SNA 局域网协议
- 分布式网络管理 RMON、CMIP/GDMO、基于 Web 的解决方案
- CORBA 分布式计算机环境

5．针对 PSOIX1003.1b 标准的兼容性

VxWorks 支持 POSIX 1003.1b 的规定和 1003.1 中有关基本系统调用的规定，其中包括进程初始化、文件和目录、I/O 初始化、语言服务、目录管理。而且 VxWorks 还支持 POSIX1003.1b 的实时扩展，包括异步 I/O、记数型信号量、消息队列、信号、内存管理和调度控制等。

6．灵活和快速的 I/O 文件特性

VxWorks 提供的快速文件系统适合于实时系统应用。它包括几种支持使用块设备（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移植到设备驱动程序上。

VxWorks 也支持 SCSI，还支持在单独的 VxWorks 系统上同时并存几个不同的文件系统。这些文件系统包括：

- POSIX 异步 I/O 和目录管理
- SCSI 支持
- 兼容 MS-DOS 文件系统
- RAW Disk 文件系统
- TrueFFS 闪存文件系统
- ISO9660 CD-ROM 文件系统
- PCMCIA 支持

1.1.3 VxWorks 的可用主机 / 目标机

1．支持的主机

- Sun-4：Sun OS 4.1.x、Solaris 2.4/2.5、Solaris 2.5.1/2.6、Solaris 2.7
- HP 9000/700：HP-UX 9.0.7、HP-UX 10.10、HP-UX 10.20
- PC：Windows 95、Windows 98、WindowsNT 3.51、WindowsNT 4.0、Windows 2000，Windows XP

2．支持的目标机

- Motorola 68K：68000、68010、68020、68030、68040、68060
- Motorola ColdFire：MCF51xx、MCF52xx、MCF53xx

- Motorola/IBM PowerPC：PPC4xx、PPC6xx、PPC7xx、MPC5xx、MPC8xx、MPC82xx
- Motorola M-Core
- Intel X86：i386、i486、Pentium、Pentium-Pro
- Intel ARM、StrongARM：SA-110、SA-1100、SA-1110
- NEC：V86x、V83x
- i960：KA/KB/CA/JX/PR、RP/RD
- NEC/LSI MIPS：R3K、R4K、Vr41xx、R4700、CW400x、CW4011、R5K
- HITACHI SH：SH-1、SH-2、SH-3、SH-4、SH-DSP、SH3-DSP
- SUN SPARC：UltraSPARC、SPARC
- ST-20
- TriCore

1.2　Tornado 开发环境

Tornado 是 Windriver 公司开发的嵌入式软件开发环境。Tornado 开发环境的最新版本是 2.2，它在延续了 Tornado 2.0 开发环境的基础上，增加了更多易于使用、性能优异的工具，因此在商业上取得了较大成功，获得了用户的好评。

Tornado 开发环境是嵌入式实时领域里最新的开发调试环境，是编写嵌入式实时应用程序的完整的软件开发平台。它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。它包含 3 个高度集成的部分：

- 运行在宿主机和目标机上的强有力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讯方式，如以太网、串口线、ICE 或 ROM 仿真器

Tornado 嵌入式集成开发环境的结构图如图 1-2 所示。

嵌入式产品的复杂度越来越高，而同时生产周期也越来越紧张，这种情况给嵌入式开发人员带来了巨大的压力。用户们则希望以更低的价格获得更多的性能。现在市场上最热门的产品应该是快速而且更加智能化、可以运行更多代码的产品。所以开发人员必须能够快速启动并且快速开发。

Tornado 是 WindRiver 公司推出的嵌入式开发环境。它保持了原来版本的特色，同时又继续不断创新。Tornado 是一个开放的，而且独立于硬件的软件解决方案，它可以运行在 90%的 32 位嵌入式处理机体系结构上，可以加速设计和开发的进程。Tornado 的开放特性使得它非常易于与第三方产品集成，而且在 WindLink 合作计划的推动下，Tornado 的工具愈加丰富和成熟，并且可以专注于互连网络和 Java 上。

Tornado 是用于实时嵌入式应用的完整的软件开发平台，可以用于 UNIX 和 Window 95/98/NT/XP，它包括 3 个高度集成的部件：

- Tornado 工具，一整套完整的交互式开发工具和实体
- VxWorks 嵌入式实时操作系统
- 主机和目标机之间全方位的通信选择

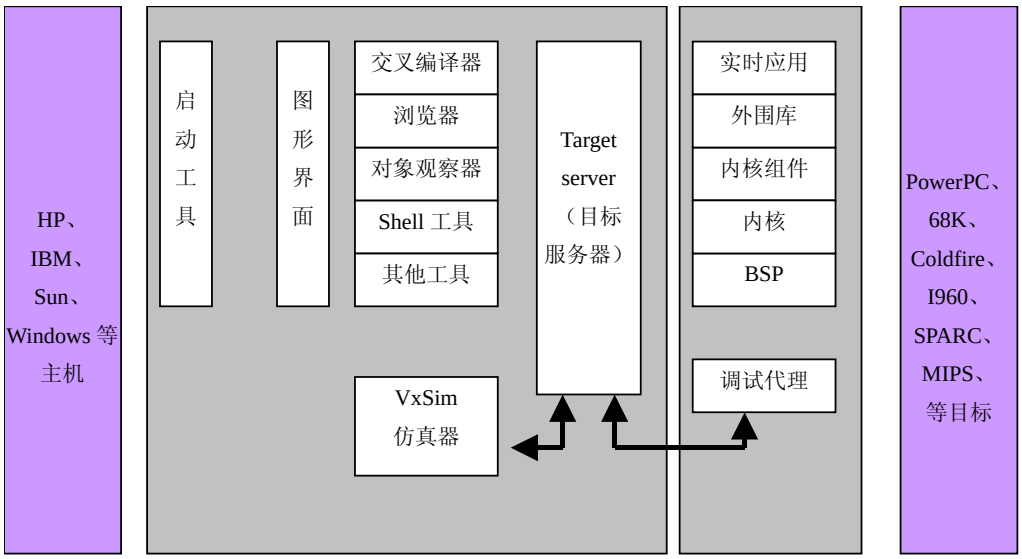

图 1-2　Tornado 嵌入式集成开发环境的结构图

Tornado 开放的体系结构以及符合企业标准的特性，使得开发人员可以轻松地开发不同厂家的系统，并且以最小的代价移植到不同的处理器上。

1.2.1　Tornado 核心工具

Tornado 核心工具包（CORE TOOLS）主要包括以下几项。

1．集成仿真器 VxSim

许多嵌入式的开发人员发现商用的硬件并不能够满足应用的要求,所以需要开发自己的、定制的硬件。在硬件开发和调试的过程中，如果能够同时开始软件的开发和调试，那么就可以有效地缩短产品的开发周期。集成仿真器（VxSim）就是这样的产品。VxSim 可以在主机的环境上模拟 VxWorks 环境，用户可以在这个模拟的环境下开发软件程序，使用 Tornado2 IDE，从而实现软硬件的同步开发。

VxSim 是完整的 Tornado2 和 VxWorks 应用的原型模拟工具。它可以允许硬件开发完成之前先行开发软件，在开发早期实现软件的测试。VxSim 还具有一定的模拟通用 I/O 设备的功能，如串口，而且可以模拟文件系统和网络。

2．软件逻辑分析仪 WindView

嵌入式的开发已经变得越来越复杂，实时任务也越来越多，如果没有复杂的系统级可视化工具支持，最终的系统几乎是不可读的。由于系统内部的运行不可知，许多开发者可能会非常困惑“为什么要用 200ms 才能接受一个键盘中断?”。

WindView 正是解决这类困惑的工具，它是一个软件逻辑分析仪，可以深入地了解任务的工作状态，同时跟踪系统的有用信息，如中断、上下文切换和任务阻塞等。所以，WindView 可以向开发者提供目标机硬件上实际应用程序的运行情况并且可以过滤掉无关的信息。

WindView 在 1994 年一经推出，立即获得了计算机界的一致好评。这是世界上第一个可以让开发人员看到正在目标机上运行的任务、中断、对象的运行情况的工具。开发人员可以在 WindView 的屏幕上看到系统中各种对象，如信号灯、消息队列、信号、任务、计时器和用

户事件的发生序列及时间。WindView有以下3个重要的特性：

- 快速集中：可以帮助用户快速地分离应用中的关键区域
- 深入分析：分析事件内部的特性
- 随处可用：无须考虑目标机的类型就可以使用WindView

3．强有力的命令行执行方式WindShell

Tornado的命令执行工具WindSh是Tornado所独有的功能强大的命令解释器，可以直接解释执行C语句表达式、调用目标机上C的函数、查看符号表中的变量，还可以直接执行TCL语言。

4．系统对象检查工具Browser

Tornado的目标机系统浏览器Browser是Tornado的一个图形化组件，它可以提供目标机中系统对象，如任务、信号灯、消息队列、内存分区、定时器、模块、变量、堆栈等系统信息，也可以显示内存的使用信息。

5．集成调试器CrossWind

CrossWind是一个源代码集成调试器，支持任务级和系统级调试，支持混合代码和汇编代码显示，支持多目标机同时调试。

CrossWind沿用了GNU Debugger 4.17版本，并且进行了VxWorks定制。在CrossWind中，开发者可以成组地观察表达式的观察窗口，可以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，可以为不同组的元素设定根值数，并且可以用分类的方法提供信息，使得调试工作效率更高。在提供GUI调试方法的同时，也提供了命令行方式，开发者可以根据自己的使用背景予以选择。

CrossWind中有两种调试方式：任务级和系统级。在任务级调试中，程序员可以在单个任务中设置断点，这种断点只对该任务有效，其他任务和系统仍然持续运行。系统级断点可以在整个系统的运行时间轴上设置断点，一个断点可以将整个系统，包括任务和操作系统全部停下来，可用于调试多任务或者是中断处理程序。

6．C/C++交叉编译工具

Tornado可以为用户提供支持C语言和C++语言的交叉编译器和类库。

编译器包括GNU的C/C++编译器和Diab C/C++编译器。类库包括VxWorks类库和tool.h类库。

7．工程管理和配置工具（Project facility/Configuration）

VxWorks中的工程管理工具是非常方便而且有用的图形化管理工具。它可以对VxWorks操作系统及其组件进行配置。组织、配置和建立VxWorks应用程序时，根据应用程序进行依赖性分析后，可以自动裁剪和配置VxWorks。Project中的检查功能还可以对开发人员定义的配置选项进行检查，并以醒目的方式标明冲突的配置。

更为有用的是，Project工具还提供了生成不同VxWorks映像的功能。开发人员可以在Project中指定生成VxWorks的映像并且加入应用程序模块，从而为产品阶段的开发提供非常便利的手段。

8．源代码分析仪（WindNavigator）

如果想透彻地了解大型C或C++程序的结构是一件非常困难的工作，如果该程序来自第三方，难度就更大了。一般而言，这类程序会包含上千个文件、函数和类。但是，开发人员

又不得不对这样的程序进行升级、维护和调试的工作。所以在 Tornado 中包含一个源代码分析仪（WindNavigator），它可以浏览源代码，提取出有用的信息后生成一个高效率的数据库，并且用图形来显示函数调用关系，快速进行代码定位。在几分钟之内，可以扫描数千行的程序。

9．动态上载/卸载器（Incremental Loader）

Tornado 的动态上载工具（Incremental Loader）可以动态地上载或卸载新增模块并且与目标机上的内核实现动态链接，所以开发人员可以不必重新下载内核以及未改动的模块。

1.2.2 WindPower 工具

Tornado 中的 WINDPOWER 工具包括：

- ScopePak
- PerformancePak
- CodeTest/Coverage
- CodeTest/Memory
- VisualSlickEdit
- Wind Foundation Classes
- Look! For Tornado

1．软件“示波器”（ScopePak）

在开发实时多任务系统程序时，开发人员非常希望可以跟踪任务的运行以便更好地掌握系统的运行，但是传统的调试工具不能实时地显示任务的运行轨迹，所以开发者只好不停地在程序中增加诸如 printf 一类的语句来跟踪任务的运行。这种方法给任务的运行增加了巨大的开销而且跟踪的效果并不理想。Wind Power 工具——ScopePak 就可以在几乎不增加系统开销的情况下，提供系统实时的运行轨迹。ScopePak 中包含两个工具：StethoScope——图形化显示系统内变量的监视工具和 TraceScope——跟踪程序运行序列的工具。

- StethoScope StethoScope 是一个实时数据收集、图形显示、归档和调试工具，主要功能是以最小的干扰跟踪并发现程序中的性能问题、失灵或错误等。它可以利用多窗口环境图形化显示变量，就好像用数字示波器显示电路信号一样；可以在最小干扰的情况下捕捉问题，调整参数，观察系统运行轨迹；可以允许开发者动态地检查和分析正在目标机上运行的实时应用程序，并且可以提供历史数据图形和直角坐标图形；可以允许开发者在程序运行的过程中改变系统的变量；可以支持多种数据类型，以高采样速率收集事后分析的数据并在产生数据的同时显示数据，也可以根据特定的时间收集数据；可以发现噪音、失灵和系统中任何信号的问题；可以为图形和数据进行归档并打印。
- TraceScope TraceScope 对于多任务环境是一个非常有用的调试工具，它可以为开发者提供函数级的运行顺序、函数在什么时候被调用、顺序和调用参数如何等。如果与 WindView 结合使用，任何一个函数的入口和出口都可以成为 WindView 的事件予以显示。TraceScope 对系统的运行几乎没有影响。

2．性能检测包 PerformancePak

实时系统的开发者总是希望获得最高的系统性能，但是又很难发现系统运行的薄弱环节。

开发者经常会遇到系统被挂起的情况，可是又不能确定是程序代码的原因还是操作系统或第三方工具的原因。在实际应用中，实时系统需要长时间稳定地工作，此时对内存使用的要求就非常高，然而内存有时是最难以控制的环节，如大量的内存被浪费，内存使用过程中发现的问题有时很隐蔽，经常是系统运行很长时间后，才能发现重大而且致命的内存问题。性能检测包（PerformencePak）就是解决这个问题的工具，它包括一个 CPU 的分析工具 ProfileScope 和内存的分析工具 MemScope。

- ProfileScope　ProfileScope 是一个分层的 CPU 运行轮廓分析仪。它可以分析 CPU 在开发者的代码、第三方库函数或操作系统中花费了多少时间。无须任何特殊的编译或硬件就可以提供全方位的精确的树型轮廓图。利用 ProfileScope，开发者可以逐个函数地分析 CPU 的运行轨迹，而且 ProfileScope 还可以指出系统效率低的环节，帮助开发者将系统的性能提高到最佳，并且发现系统的瓶颈所在。
- MemScope　MemScope 可以用图形化的方式分析内存的使用情况而且无须任何特殊的编译。MemScope 可以为开发者提供实时的详细的内存使用图，包括每一个内存块的历史调用过程。它还可以帮助开发者发现内存漏洞，全面理解内存的使用情况，从而使系统的运行更加高效和可靠。

3．代码测试器 CodeTest/Coverage

在实时嵌入式系统中，很难预料程序的运行序列，开发者难以确定系统内部的运行轨迹，而软件更是其中最难以预料的部件。在以往的嵌入式开发中，开发者通常会利用自行开发的工具或手工检查代码是否运行，这样做的结果当然不能令人满意。

WindPower 工具 CodeTest/Coverage 就是为嵌入式开发提供的嵌入式软件验证工具，有了 CodeTest/Coverage，开发者和测试者可以在开发早期验证软件的模块和集成特性，无须任何硬件的支持。CodeTest/Coverage 可以在代码运行的过程中检查代码的运行序列，准确地显示代码在特定的测试条件下的运行轨迹，为开发者提供有用的信息。

CodeTest/Coverage 可以在程序的运行过程中，交互地显示程序、函数和源代码的语句覆盖情况。CodeTest/Coverage 的独特之处还在于它可以在测试进行过程中以 XY 时间轴的方式显示覆盖趋势，如果发现覆盖峰值，可以及时终止测试，因此可以有效地缩短测试的时间。

CodeTest/Coverage 所生成的分析结果可以让开发者在实时系统的运行过程中跟踪覆盖率并观察测试的效率。开发者可以用多种方法观察覆盖信息：总结、函数、源代码及趋势。开发者可以获得真实的覆盖测量结果而不是统计采样，因此开发者获得的永远是最真实的测量结果。CodeTest/Coverage 可以跟踪数以百万计的分支，所以非常适合于当前的嵌入式应用。CodeTest/Coverage 所产生的信息有助于开发者发现无用的代码，减少死码，因此可以缩小程序的大小和复杂度，提高程序的性能。

4．CodeTest/Memory

在实时嵌入式开发过程中，跟踪并标识内存的动态分配和错误是非常重要而且非常困难的工作。一个小小的内存故障可能会引起系统严重的故障。所以，在系统失效之前及时地发现内存的漏洞是必要而且必须的。CodeTest/Memory 可以让开发者和测试者在开发早期验证软件的模块和集成特性，无须任何硬件的支持。CodeTest/Memory 可以解决嵌入式程序中最难解决的动态内存跟踪问题。它不仅可以在程序的运行过程中报告每一个申请语句所分配到的内存字节，更可以发现 20 多种分配错误，如“释放空指针”等。

有了 CodeTest/Memory，开发者可以从烦琐而低效的手工内存检查中解放出来，在发生内存错误时精确地定位源代码的位置，在明显的故障出现之前，检测并报告最复杂的内存错误。CodeTest/Memory 还可以发现最消耗内存的函数，在程序的运行过程中，观察内存分配曲线。

CodeTest/Memory 可以监视所有的内存分配以便跟踪有多少内存被分配了但是没有释放，并且利用柱型图和数据表格清楚地显示程序中所有内存语句的统计资料。CodeTest/Memory 还可以捕捉到离散的内存分配错误及错误出现时完整的程序上下文。

5. WFC 基础类库（Wind Foundationg Classes）

WFC 基础类库是为基本 I/O、数据结构、算法和 VxWorks 函数提供面向对象的接口。它使得开发者能够使用和重复使用高度优化和经过谨慎测试的代码，以最少的时间、最小的错误率、最低的成本完成实时嵌入式应用。

VxWorks Wrapper Classes 包含与 10 个 VxWorks C 函数库的 C++接口。

Toolsh++ from Rogue Wave Software 包含模板和一些不基于模板的 C++类，例如，操作类型、规范运算、单向/双向链表、文件类型和文件空间管理、队列、栈等

C++ Booch Components from Rogue Wave software 包含一组经过实时优化的基于模板的数据结构和算法，其中包括图形、列表、队列等数据结构；查询、排序工具；异常处理等。

6. Look! For Tornado

Look! For Tornado C++程序可视化调试工具，以图形化的方式动态地监视 C++程序的运行情况。揭示 C++的动态运行特性及每一行代码在运行中发挥的作用。通过实时地观察和确认对象间的关系加速调试过程，通过动态结构的观察，使开发人员可以确定程序是否符合设计要求。它可以帮助开发人员由 C 编程转移到 C++编程。

1.3 Tornado 嵌入式开发系统可选组件

Tornado 嵌入式开发系统可选组件的特点如下：

- 板级支持包开发工具 BSP Developer's Kit 板级支持包开发工具 BSP Developer's Kit 可以帮助开发者进行设计、归档和测试新设备的驱动程序和 BSP 等工作。
- 虚拟内存接口 VxVMI VxVMI 是 VxWorks 操作系统的虚拟内存接口，提供虚拟内存管理，是强有力的测试和内存管理工具，它包括文本和核心数据段、代码段及数据结构的保护、程序员接口等，可以适用于多种不同的 CPU 结构体系。
- 支持紧密耦合共享内存多处理器结构的 VxMP VxMP 是 Vxworks 操作系统的扩展，它是支持紧耦合和共享内存的多处理器结构的软件包，允许实时嵌入式应用程序的性能具有可以升级超过单 CPU 的能力，其透明的、高性能的设计可以使运行在不同 CPU 上的任务利用现有的 Vxworks 机制就可以实现相互间的同步和数据交换，例如信号量和消息队列。
- 支持紧密耦合共享内存多处理器结构的 VxDCOM VxDCOM 是 Vxworks 操作系统的扩展，它是支持紧耦合和分布式硬件的多处理器结构的软件包。
- 支持松散耦合分布式多处理器结构的 VxFUSION VxFUSION 是 Vxworks 操作系统消息

队列的扩展，它是支持松耦合和分布式硬件的多处理器结构的软件包。

- 闪存文件系统 TrueFFS for Tornado TrueFFs for Tornado 是 Tornado 开发环境中的一个集成的快速闪存文件系统，它使用一系列的嵌入式闪存设备来实现快速可靠的物理存储。这样，在闪存设备上进行读写操作就与在 DOS 文件系统设备上是一样了。

1.3.1 板级支持包 BSP Developer's Kit

开发者如果希望他们的产品尽快上市，首要的工作就是将硬件驱动起来。在实际应用中，不论这个硬件是商业的还是自行开发的，给硬件写 BSP 都不是一件简单的工作。所以开发者需要一种必需的工具轻松容易地进行自己硬件的 BSP 开发，以便将主要精力放在开发应用软件上。

板级支持包开发工具 BSP Developer's Kit 可以为开发者提供开发 BSP 所需的所有工具，包括成熟的 BSP 的模板和相关的设备驱动程序。现在，许多硬件厂商都在使用 BSP Developer's Kit 来开发他们产品的驱动程序和 BSP，这样开发者可以获得更多的支持而且很有可能获得他们所需的驱动程序。

1. BSP 开发工具包的基本选项

BSP Developer's Kit Base Option 是板级支持包的基本包，它包括以下几部分：

- 板级支持包测试工具 BSP validation test suite
- 板级支持包开发模板 Template BSP
- 驱动程序开发模板 Template Driver
- SCSI 测试工具 SCSI Test Suite

BSP validation test suite 可以检查 BSP 和驱动程序的基本功能以及相关文档中所存在的问题，它以源代码的形式提供，可供开发者维护和扩展。

Template BSP 中包括以下 CPU 类型的 BSP 模板：Motorola 68K/CPU32、Intel/AMD/Cyrixx86、MIPS、Motorola/IBM PowerPC、SPARC、Intel i960。

Template Driver 驱动程序开发模板可以为开发者提供以下设备的驱动程序模板：

- 串口
- 以太网
- SCSI 磁盘
- 中断控制器
- VME 总线
- 定时器
- 非易失存储器

SCSI 测试工具为开发者提供 SCSI 的测试程序。

BSP 文档开发工具 Documentation Set 提供如何实现设备驱动程序、如何实现目标机上 VxWorks 的最小配置。

2. 板级支持包开发工具高级包

板级支持包开发工具高级包 BSP Developer's Kit Value Option 主要面向的是愿意使用 WindRiver 公司提供的通用驱动程序源代码，并以此进行 BSP 开发的人员。BSP Developer's

Kit Value Option 中包括几乎所有的现成的标准驱动程序，例如，Ethernet、SCSI 等。其主要组件包括：

- 板级支持包的基本包 BSP Developer's Kit Base Option
- Ethernet Driver Source 驱动程序源代码
- SCSI Driver Source 驱动程序源代码

1.3.2 虚拟内存接口 VxVMI

1．产品背景及概述

在许多应用环境中，如航天、高端办公室自动化、高端全球通信及生命医学等，都需要有较高的系统可靠性。传统的、不加保护的、基于任务的实时操作系统一般是适用于嵌入式领域的。这种操作系统的特点是性能高、节约内存、编程方式非常简单而且灵活，但是这样做的后果就是整体系统比较脆弱，关键的内存区域不能抵抗未授权的访问。系统的可靠性只能通过严格的测试和严整过程保证。

可选部件 VxVMI 是解决这一问题的方案。最初，VxVMI 是针对 68K 机器而引入的，它可以允许 VxWorks 的开发者在应用程序中增加内存保护。后来，VxVMI 逐渐地在许多体系结构上都可以成功使用，它为任务提供了应用程序接口（API），允许任务为代码和数据段提供内存保护段，从而提供了测试无法保护的可靠性。

VxVMI 为用户提供了实现高可靠性的手段，可以防止系统中各个任务之间的互相破坏，但是在使用 VxVMI 时，应注意以下两点：

- VxVMI 不能防止任务破坏操作系统，只能在任务之间提供一定的隔离手段；
- VxVMI 最初是为高端的 68K 机器而设计的，至今仍然没有移植到某些系列的体系结构上。

2．产品特性

VxVMI 是为 VxWorks 操作系统增加内存保护的附件产品，可以为实时嵌入式系统中任何一个任务提供内存保护的功能。VxVMI 可以为任何一个通过 VxWorks 环境下载的应用代码提供关键资源的写保护，包括程序代码段和异常向量表，甚至包括 VxWorks 的程序代码段。因此，开发者可以专注于编写应用程序，无须考虑内存的安全问题。

VxVMI 所提供的 API 可以允许开发者按照应用程序的需要，创建和管理多个虚拟内存上下文。这些 API 可以允许从物理地址到虚拟地址的映射，修改并检查虚拟内存的状态，如设置写保护和页面的缓存，生成虚拟内存的状态报告。

其特性包括：

- 基本特性
- 文本写保护
- 内核向量表写保护
- 扩展特性
- 编程接口 API
- 将物理内存映射到虚拟内存
- 修改并检查虚拟内存状态
- 生成虚拟内存的状态报告

- 创建多个虚拟内存上下文

1.3.3　支持紧耦合共享内存多处理器结构的 VxMP

1．VxMP 产品概述

随着嵌入式领域中越来越多的用户选择 COTS（Commercial Off-the-shelf）产品，多板总线的设计也越来越多，包括 VME、MultiBus、PCI、cPCI 等，这些总线结构一般需要共享内存来实现进程间通信和同步等功能。在分布式多处理环境中，这种不对称、紧密耦合的多处理要求是一个重要的特性。

VxMP 在最早是用于 68K 机器的，随后又成功地用于其他体系结构中。VxMP 本身只是一个在 VxWorks 共享内存驱动支持下的应用实体，它可以为共享内存的多个 CPU 提供若干基于 VxWorks 中基本进程间通信机制的 API。用户可以创建这些机制的共享版本，只要利用标准 VxWorks 所提供的 API 编写应用程序即可，由 VxMP 管理所有的实现细节，用户就好像在使用本地“API”一样。

2．VxMP 产品特性

VxMP 是 Vxworks 操作系统的附加产品，它允许开发者将一个应用程序中的任务在多个 CPU 上进行剪裁。VxMP 可以实现完全透明的设计，使得 VxWorks 操作系统可以支持多处理机而无须修改代码。

VxMP 允许多个 CPU 上的任务同步或者交换数据，并提供互斥的功能。可以利用标准 VxWorks 操作系统所提供的二进制和计数型信号量、消息队列和内存分配功能。为获得最大的性能，VxMP 的数据结构都保存在可供所有 CPU 访问的共享内存中。操作共享内存中的对象和操作本地对象的方式完全一样，所以可以使用诸如 semGive, semTake 等通用的系统调用。这种透明的设计可以让 VxWorks 的开发者无须任何培训就可以实现高性能、易于移植的系统。

VxMP 的基本特点包括：

- 支持多达 20 个 CPU
- 共享二进制和计数信号量
- 共享 FIFO 消息队列
- 共享内存池和分区
- 命名服务（将符号名字翻译成对象 ID）
- 用户可配置的共享内存池大小

1.3.4　支持紧耦合分布式多处理器结构的 VxDCOM

1．VxDCOM 产品概述

嵌入式计算机产品的用户日益要求与其他设备实现轻松无缝的相互连接，尤其是与基于 PC 机的管理控制台和基于 Windows 的应用程序和工具。例如，用户希望不费力地就能将嵌入式设备中的数据插入到 Microsoft Excel 表格中进行统计计算。

VxDCOM 是将 Microsoft DCOM（分布式部件对象模型）进行剪裁后，适用于 Tornado 开发环境的产品。有了 VxDCOM，生产商可以快速地创建并压缩出可以无缝地与远程 PC 交互的嵌入式产品。例如，Windows NT 的工作站可以利用 GUI 来控制一个机器人，或者一个 VxWorks 的传感器可以在 PC 的制表软件上进行印刷。甚至，VxWorks 设备之间也可以利用标准对象接

口进行通信。

2．VxDCOM 产品特性

有了 VxDCOM 的支持,运行 VxWorks 实时操作系统的设备可以无缝地相互连接或者与基于 PC 的管理控制台进行连接。VxDCOM 支持企业标准，提供在分布式环境中可剪裁的、实时的解决方案。它具有以下主要特点：

- 与 PC 无缝集成
- 创建 OPC 服务器
- 与实现语言无关
- 紧缩的 280K 开销或更小
- 真正的 DCOM 协议
- 实时动态的线程池

1.3.5 支持松耦合分布式多处理器结构的 VxFUSION

1．VxFUSION 产品概述

嵌入式系统中已逐渐引入分布式体系结构，分布式的范围十分广泛，从紧密耦合到松散耦合、从消息队列到完全的分布式组件对象模型。VxFUSION 是 Vxworks 操作系统消息队列的扩展，相比于支持紧密耦合、共享内存配置的 VxMP 和支持紧密耦合、分布式部件对象模型的 VxDCOM，VxFUSION 支持松散耦合的分布式多处理器结构。他们分别适用于不同的应用环境，不能互相替代。

2．VxFUSION 特性及应用背景

VxFUSION 是一个轻便的、独立于介质的容错机制，这种机制建立在 Vxworks 的消息队列基础之上，主要用来开发分布式应用程序。

在 VxWorks 中，支持分布式多处理的可选件产品有以下几种：VxMP 只能允许对象利用共享内存通信；TCP/IP 可以用于网络之间的通信，但是它不适用于实时使用；VxDCOM 基于 Microsoft 的分布式部件对象模型，可以提供对远程对象的紧密耦合的访问。

VxFUSION 具有以下特点：

- 提供一个轻便的基于 Vxworks 消息队列的分布式机制。
- 它与传输介质无关，允许分布式系统通过任何传输有效地交换数据，没有对通信硬件的特殊要求。
- 可以在多节点系统中复制每个节点上的已知对象，并维护数据库，以此避免由于一对多主从结构而造成的单点故障。
- 支持系统中单点和多点消息传送。
- 实现位置透明机制，无须重新编写应用代码，对象可以在系统内无缝地传输，向对象发送消息时可以无须考虑该对象在多点系统中的位置。

1.3.6 闪存文件系统 TrueFFS for Tornado

1．TrueFFS 产品概述

越来越多的嵌入式系统希望使用高容量的闪存设备为代码和数据提供大容量而且不易失的存储。由于闪存设备的高密度和高效率，已经在许多应用环境中代替了传统的硬盘，而且闪存设备的可靠性也高于机械介质。

但闪存设备在编程过程中的使用比较复杂，例如，在重写某磁盘块时，必须先将该块上的内容擦除。而且这些磁盘块的使用寿命与编程的次数紧密相关。开发人员非常希望可以像使用磁盘一样地使用闪存，但是又不愿意做复杂的管理工作。

TrueFFS 正是解决这一问题的方案，它基于实时（RealTime）的企业标准 Flash Translation Layer，可以在闪存设备和 VxWorks 的 Dos 兼容的文件系统之间，将硬盘模拟成一个软件栈，用户只要利用标准的文件 I/O API 编写应用程序就可以读取文件。TrueFFS 可以实现块分配、碎片收集等功能，提高闪存设备的使用寿命，减少访问时间。

2．TrueFFS 产品特性

TrueFFS 具有以下功能：

- 闪存翻译层 TrueFFS 是建立在 M-SYSTEM 闪存磁盘先驱 TrueFFS 专利核心技术的数据格式之上，这种专利技术实现了 PCMCIA 闪存翻译层标准。
- 故障恢复 TrueFFS 通过监视和验证每个操作的成功与否来防止由于硬件问题或掉电所引起的可能的数据错误。TrueFFS 利用动态映射机制在不同的地方进行写操作，这样通过自动排除故障就可以保证数据的完整性。
- 磨损情况测量 TrueFSS 使用统计方法来实现磨损情况测量，这样闪存设备中的所有擦除单元都可以平均使用，最终具有相同的期望值。
- 碎片收集 TrueFSS 使用碎片收集方法可以收回不再保存有效数据的闪存块。
- 符合 FTL（闪存传输层）标准。
- 断电恢复 TrueFSS 采用的算法是建立在“写后擦除”而不是建立在“写前擦除”的基础上，这样即使断电也可以保证信息的一致性。
- 块分配 TrueFSS 采用块分配的方法，不但保证访问数据时的一致性，也减少了碎片大小，使碎片收集工作可以更有效率。

第 2 章　VxWorks 系统基本理论

2.1　VxWorks 系统概述

实时多任务操作系统是能在确定的时间内执行其功能，并对外部的异步事件作出响应的计算机系统。多任务环境允许一个实时应用作为一系列独立任务来运行，各任务有各自的线程和系统资源。VxWorks 系统提供多处理器间和任务间高效的信号量、消息队列、管道、网络透明的套接字。实时系统的另一关键特性是硬件中断处理。为了获得最快速可靠的中断响应，VxWorks 系统的中断服务程序 ISR 有自己的上下文。

VxWorks 实时操作系统由几百个相对独立的、短小精炼的目标模块组成，用户可根据需要选择适当模块来裁剪和配置系统，这有效地保证了系统的安全性和可靠性。系统的链接器可按应用的需要自动链接一些目标模块。这样，通过目标模块之间的按需组合，可得到许多满足功能需求的应用。

VxWorks 操作系统的基本构成模块包括以下部分。

- 高效的实时内核 Wind　VxWOrks 实时内核（Wind）主要包括基于优先级的任务调度、任务同步和通信、中断处理、定时器和内存管理。
- 兼容实时系统标准 POSIX　VxWOrks 提供接口来支持实时系统标准 P.1003.1b。
- I/O 系统　VxWOrks 提供快速灵活的与 ANSI-C 相兼容的 I/O 系统，包括 UNIX 的缓冲 I/O 和实时系统标准 POSIX 的异步 I/O。VxWOrks 包括以下驱动：

网络——网络设备（以太网、共享内存）
管道——任务间通信
RAM——驻留内存文件
SCSI——SCSI 硬盘，磁盘，磁带
键盘——PC x86 键盘
显示器——PC x86 显示器
磁盘——IDE 和软盘
并口——PC 格式的目标硬件

- 本机文件系统
- 文件 I/O 系统　VxWorks 的文件系统与 MS-DOS、RT-11、RAM、SCSI 等相兼容。
- 网络特性　VxWorks 网络能与许多运行其他协议的网络进行通信，如 TCP/IP、4.3BSD、NFS、UDP、SNMP、FTP 等。VxWorks 允许任务通过网络存取文件到其他系统中，并对任务进行远程调用。
- 虚拟内存（可选单元 VxVMI）　VxVMI 主要用于对指定内存区的保护，如内存块只读等，加强了系统的健壮性。
- 共享内存（可选单元 VxMP）　VxMP 主要用于多处理器上运行的任务之间的共享信号

量、消息队列、内存块的管理。

- 驻留目标工具　Tornado 集成环境中，开发工具工作于主机侧。驻留目标外壳、模块加载和卸载、符号表都可进行配置。
- Wind 基类　VxWorks 系统提供对 C++的支持，并构造了系统基类函数。
- 工具库　VxWorks 系统向用户提供丰富的系统调用，包括中断处理、定时器、消息注册、内存分配、字符串转换、线性和环形缓冲区管理以及标准 ANSI-C 程序库。
- 性能优化　VxWorks 系统通过运行定时器来记录任务对 CPU 的利用率，从而进行有效的调整，合理安排任务的运行，给定适宜的任务属性。
- 目标代理　目标代理可使用户远程调试应用程序。
- 板级支持包　板级支持包提供硬件的初始化、中断建立、定时器、内存映像等。
- VxWorks 仿真器（VxSim）　可选产品 VxWorks 仿真器，能模拟 VxWorks 目标机的运行，用于应用系统的分析。

2.2　VxWorks 系统内核及组件

Vxworks 是采用微内核结构设计的 OS（Operating System，即操作系统）。所谓微内核，字面意思是指在系统 Kernel 态运行的代码相对来说比较少，这是区别于分层结构 OS，例如 UNIX、NT 等而言的。其深层含义是，只把必须在 Kernel 模式运行的 OS 模块放在内核之中，而把可有可无的 OS 组件，例如文件系统、I/O 等统统置于内核以外，在用户模式运行。VxWorks 中，微内核称为 Wind Microkernel，是构建其他系统组件的最小操作集，其大小可以做到只有 6kB。Wind 的内核如图 2-1 所示，它主要包括 3 个部分：

- 多任务环境
- 原子信号量
- 系统时钟响应

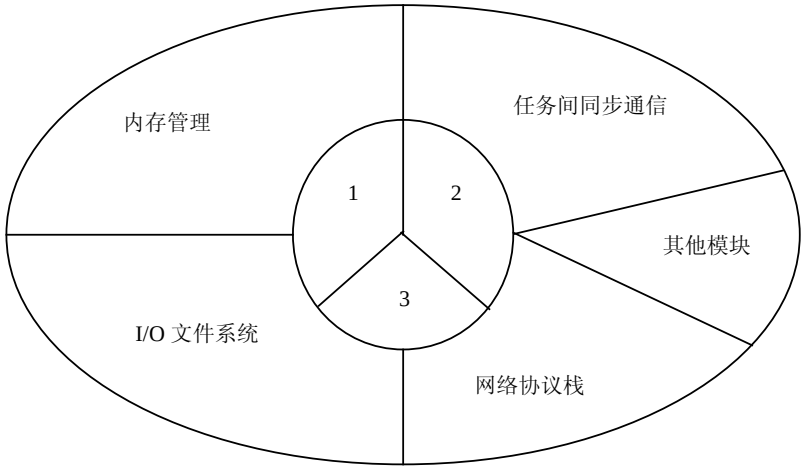

图 2-1　VxWorks 系统的微内核结构

由于真实世界的事件异步性，要求内核具有多进程并发执行的能力，Wind 的多任务环境就是一个较好的对外界的匹配，它提供了对应于真实事件的多线程执行机制。内核采用一些特定的管理调度策略分配 CPU 给多个任务，从而获得并发性。

多任务内核必须在任务之间提供同步与协调，以及对于临界资源的保护机制。这是通过原子信号量实现的，信号量在内核中实现，对于任务隐藏了其过程，因而在使用中是不可打断的。对外部封装了像 semTake、semGive 这样的系统调用，通过改变任务的执行状态实现同步协调，以及对共享资源的保护。内核设计的一个关键问题就是其性能边界，对于实时系统，设计目的不是追求大的任务吞吐量，而是响应的及时性，衡量及时性的标准是其响应延时的确定性，系统设计时要从最恶劣的条件考虑，我们可能希望系统能够始终以 50μs 执行一个函数，而不希望平均以 10μs 执行，而偶尔以 75μs 执行。

运行在 Kernel 态的内核代码不是以通常的任务方式实现的，运行时 CPU 没有相应的任务上下文，也没有抢占的机制。这种模式中的算法具有一定的确定性，为了保证 VxWorks 系统的实时性，要使得用于任务管理调度的时间开销、任务间同步、中断延时都尽可能地小。同时内核代码还应该尽可能地少，以保证 VxWorks 系统的抢占延时较小。在 MOTOLORA 的 68K 系列 CPU 芯片（20MHz 左右）上做测试，Wind 用于任务上下文切换的时间是 17μs；用于同步开销的请求和释放二值信号量仅用 8μs；内核中断延时小于 10μs。

由于采用了微内核结构，使得系统的可裁减性非常好，诸多的 OS 模块都可以实现软件的“即插即用”，同时，OS 的部分核心模块运行在用户模式，与用户应用任务处在平行的位置，用户的系统功能调用就相当于调用子函数，节省了模式转换的系统开销，提高了实时性。

以上是对 Wind 的性能上作了一个介绍，Wind 的实现实际上是一段汇编算法，它只和某种 CPU 的指令集相关，而与具体的主板结构，例如总线类型、内存芯片类型、内存大小以及内存如何定位、I/O 类型及有多少 I/O、网络接口芯片类型以及这些芯片间的连接通信方式等无关。举一个对外设作假设的例子，MS DOS 在系统的移植中，所有的外设都必须符合 BIOS 标准。VxWorks 对外设不作假设，因而移植性好。但是在 Wind 微内核和硬件之间要有一个适配层，这就是 BSP，即板级支持包，通过 BSP，完成设备驱动与 VxWorks 的集成和在具体的硬件平台上的 Vxworks 的初始化过程，使得 VxWorks 能够在特定的平台上正常运转。

Wind 内核对外部提供的系统调用符合 POSIX1003.1b 标准。对于 VxWorks 支持的不同 CPU，VxWorks 的不同版本，其提供统一的 API 编程接口。

VxWorks 内核（Wind）及其组件的基本功能可以分为如下几大类：

- 任务管理
- 事件和异步信号服务
- 信号量服务
- 消息队列服务
- 内存管理
- 中断服务程序
- 时钟管理和定时器服务
- 出错处理

在以下各节中，将对 VxWorks 内核的各类功能分别进行描述。

2.2.1　任务管理

任务是代码运行的一个映像，从系统的角度看，任务是竞争系统资源的最小运行单元。任务可以使用或等待 CPU、I/O 设备及内存空间等系统资源，并独立于其他任务，与它们一起并发运行。VxWorks 内核使任务能快速共享系统的绝大部分资源，同时有独立的上下文来控制个别线程的执行。

1．任务结构

多任务设计能随时打断正在执行着的任务，对内部和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。从表面上来看，多个任务正在同时执行，实际上，系统内核根据某一调度策略让它们交替运行。系统调度器使用任务控制块的数据结构（简记为 TCB）来管理任务调度功能。任务控制块用来描述一个任务，每一任务都与一个 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、初始堆栈指针等信息。调度器在任务最初被激活时以及在休眠态重新被激活时，要用到这些信息。

此外，TCB 还用来存放任务的“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。在任务被重新执行时，必须要恢复上下文。通常，上下文就是计算机当前的状态，也即各个寄存器的内容。如同在发生中断时所要保存的内容一样。当发生任务切换时，当前运行任务的上下文被存入 TCB，将要被执行的任务的上下文从它的 TCB 中取出，放入各个寄存器中。于是转而执行这个任务时，执行的起点是前次它在运行时被中止的位置。

在 VxWorks 中，内存地址空间不是任务上下文的一部分。所有的代码运行在同一地址空间。如果需要让每一个任务独立地运行于各自的内存空间中，必须获得可选组件 VxVMI 的支持。

2．任务状态及其跃迁

实时系统 VxWorks 的一个任务可有多种状态，最基本的状态有以下 4 种。

- 就绪态（Ready）：任务只等待系统分配 CPU 资源。
- 挂起态（Pend）：任务需等待某些不可利用的资源而被阻塞。
- 休眠态（Sleep）：如果系统不需要某一个任务工作，则这个任务处于休眠状态。
- 延迟态（Delay）：任务被延迟时所+处状态。

当系统函数对某一任务进行操作时，任务从一种状态跃迁到另一状态。处于任一状态的任务都可被删除。VxWorks 的任务跃迁如图 2-2 所示。

VxWorks 系统中任务的状态跃迁与系统调用的关系如表 2-1 所示。

表 2-1　　状态跃迁与系统调用关系表

状 态 跃 迁	系 统 调 用
就绪态 -> 挂起态	*semTake()/msgQReceive()*
就绪态 -> 延迟态	*taskDelay()*

续表

状 态 跃 迁	系 统 调 用
就绪态 -> 休眠态	*taskSuspend()*
挂起态 -> 就绪态	*semGive()/msgQSend()*
挂起态 -> 休眠态	*taskSuspend()*
延迟态 -> 就绪态	expired delay
延迟态 -> 休眠态	*taskSuspend()*
休眠态 -> 就绪态	*taskResume()/taskActivate()*
休眠态 -> 挂起态	*taskResume()*
休眠态 -> 延迟态	*taskResume()*

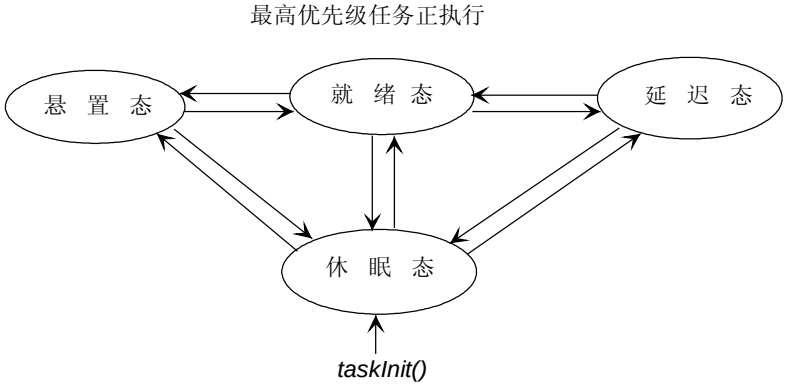

图 2-2 VxWorks 系统中任务之间的状态跃迁

3．任务调度

多任务调度须采用一种调度算法来为就绪态队列中的任务分配 CPU。Wind 内核采用基于优先级的抢占式调度法作为它的默认调度策略，同时它也提供了轮转调度算法。

基于优先级的抢占式调度具有很多优点。这种调度方法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态时，系统内核立即保存当前任务的上下文，切换到更高优先级的任务。

Wind 内核优先级划分为 256 级（0~255）。优先级 0 为最高优先级，优先级 255 为最低。当任务被创建时，系统根据给定值分配任务优先级。然而，优先级也可以是动态的，它们能在系统运行时被用户使用系统调用 taskPrioritySet()来加以改变，但不能在运行时被操作系统所改变。

在图 2-3 中，t1、t2、t3 三个任务的优先级依次从低到高，任务 t1 最先运行，但它被 t2 和 t3 打断，因此最后运行完毕。

轮转调度法分配给处于就绪态的每个同优先级的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。很明显，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，就进行任务切换，停止执行当前运行的任务，将它放入队列尾部，对运行时间计数器置零，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。

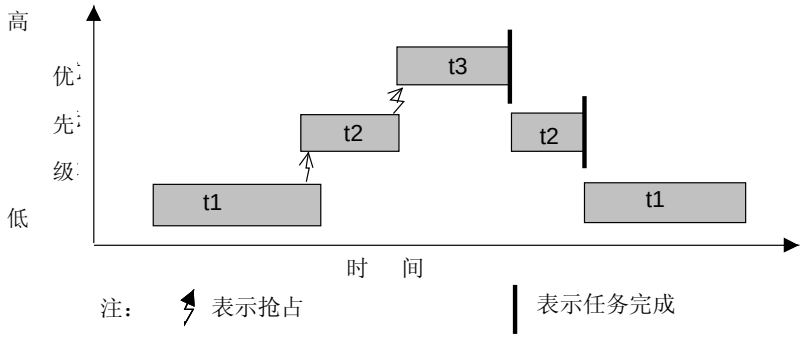

图 2-3　VxWorks 系统中高优先级任务抢占低优先级任务的实例

图 2-4 显示出了 VxWorks 系统中时间片轮转和优先级调度的混合算法。刚开始时有三个处于同一优先级的任务即 t1、t2、t3 三个任务处于就绪态队列中。但是 t1 在就绪态队列的列首。这时候 t1、t2、t3 三个任务各运行相同的时间片。一个轮回以后，具有较高优先级的任务 t4 进入就绪态队列，根据优先级抢先原理，t4 打断 t1、t2、t3 三个任务的正常运行秩序。t4 运行完毕后，t1、t2、t3 三个任务依据原来的时间片轮转调度运行。

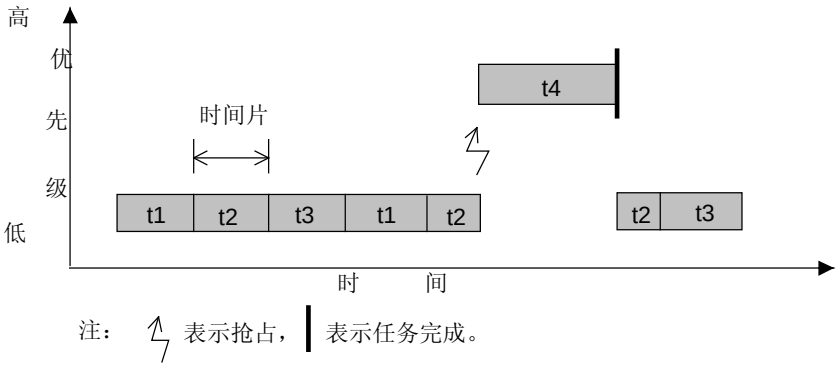

图 2-4　VxWorks 系统中时间轮转和优先级调度的混合算法

4．抢先禁止

Wind 内核可通过调用 *taskLock()*和 *taskUnlock()*来启动或禁止调度器。当一个任务调用系统调用 *taskLock()*以后，任务运行时就没有基于优先级的抢先发生。不过，如果任务被阻塞或是挂起时，调度器仍然从就绪队列中取出最高优先级的任务运行。当设置了抢先禁止的任务解除阻塞后，再次开始运行时，抢先又被禁止。这种抢先禁止可以防止任务的切换，但对中断处理不起作用。

5．异常处理

程序代码和数据的出错，是指非法命令、总线或地址错误、被零除等。VxWorks 异常处理包一般是将引起异常的任务休眠，保存任务在异常出错处的状态值。内核和其他任务继续执行。用户可借助 Tornado 开发工具查看当前任务状态，从而确定被休眠的任务。

6．任务管理

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，具体实现通过如表 2-2 所示的一些系统调用。

表 2-2　　系统调用

任务管理的系统调用	功 能 描 述
taskSpawn()	创建（产生并激活）新任务
taskInit()	初始化一个新任务
taskActivate()	激活一个已初始化的任务
taskName()	由任务 ID 号得到任务名
taskNameToId()	由任务名得到任务 ID 号
taskPriorityGet()	获得任务的优先级
taskIsSuspended()	检查任务是否被挂起
taskIsReady()	检查任务是否准备运行
taskTcb()	得到一个任务控制块的指针
taskDelete()	中止指定任务并自由内存（仅任务堆栈和控制块）
taskSafe()	保护被调用任务
taskSuspend()	挂起一个任务
taskResume()	恢复一个任务
taskRestart()	重启一个任务
taskDelay()	延迟一个任务

- taskSpawn 用于创建(产生并激活)新任务，它的原型是

```
int taskSpawn
 (
  char     *name,            /*新任务名称*/
  int      priority,           /*任务的优先级*/
  int      options,           /*任务可选项*/
  int      stackSize,          /*任务堆栈大小*/
  FUNCPTR      entryPt,      /*任务入口函数*/
  Int      arg1,               /*任务入口函数所带参数 1~10*/
  Int      arg2,      int      arg3,
  Int      arg4,      int      arg5,
  Int      arg6,      int      arg7,
  int      arg8,      int      arg9,
  int      arg10
 )
```

此调用运行成功返回任务 ID 号，否则为 ERROR。

任务可选项的几种模式如表 2-3 所示。

表 2-3　　任务可选项

名　　称	值	描　　述
VX_FP_TASK	0x8	运行带浮点的协处理器
VX_NO_STACK_FILL	0x00	不使用 0xee 填充堆栈
VX_PRIVATE_ENV	0x80	用私有环境运行任务
VX_UNBREAKABLE	0x2	任务内的断点失效
VX_SUPERVISOR_MODE	0	用户任务常用值

- taskDelete 用于删除一个任务，它的原型是

```
STATUS    taskDelete
(
 int       tid    /*删除任务的 ID 号*/
)
```

删除指定 ID 号的任务，并释放任务所占有的内存

- taskDelay 用于延迟任务，它的原型是

```
STATUS     taskDelay
(
 int       ticks /*延迟的时间滴答数*/
)
```

任务延迟为某一任务休眠一定时间提供了简单的处理方法，一般用于任务的周期性循环执行。当输入参数为 NO_WAIT（其值为零）时，表示将所延迟的任务切换到同一优先级就绪队列的尾部。

- taskSuspend 用于任务挂起，它的原型是

```
STATUS    taskSuspend
(
 int     tid     /*被挂起的任务 ID 号*/
 )
```

- taskResume 用于恢复任务，它的原型是

```
STATUS     taskResume
  (
   int     tid      /*恢复的任务 ID 号*/
)
```

7．VxWorks 系统任务

在 VxWorks 系统中，除了用户创建的任务外，系统还会自动启动一些系统任务。当 VxWorks 系统目标板加电启动成功后，有如下几个任务已开始运行。

- 根任务 tUsrRoot　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、网络通信任务和 tRlogind 等任务。一般来说，在所有的初始化工作完成后，根任务 tUsrRoot 被删除。

- 注册任务 tLogTask　注册任务 tLogTask 被 VxWorks 模块用来传送不需 I/O 操作的系统消息。
- 异常处理任务 tExcTask　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、删除或是改变优先级。
- 网络通信任务 tNetTask　网络通信任务 tNetTask 负责系统级任务的网络通信。
- 目标代理任务 tWdbTask　如果目标代理程序运行在任务模式，则目标代理任务 tWdbTask 被创建，用来响应主机目标服务器的请求。

8．任务错误状态

按照惯例，C 库函数当产生错误时将一个全局整数 errno 设置为某个合适的值，告诉系统发生了什么错误。这种惯例也是 ANSI C 标准的一部分。

VxWorks 中的 errno 由两种不同的方式同时定义。在 ANSI C 中有一个潜在的命名为 errno 的全局变量，它可以在 Tornado 开发工具中显示；errno 同时作为一个宏也定义在 errno.h 中。对 VxWorks 而言，除一个系统调用外，其他所有的系统调用均可操作 errno 宏。errno 宏定义成对函数_errno()的一次调用：

```
#define errno   ( *_errno())
```

errno()函数返回全局变量 errno 的地址(这个函数不能调用使用自身的宏 errno)。这个函数带有一个有用的特征：由于 error()是一个函数，用户可以在调试时在其中加入断点，用来测试一个特定的错误在哪儿发生。而且，C 程序可以按照正常方式设置 errno 的值。

```
Errno= someErrorNumber;
```

因此不要使用与 errno 名字相同的局部变量。

因为 VxWorks 将 errno 声明为一个全局变量，所以应用代码可以直接引用它。而且，对于 VxWorks 的多任务环境，由于每个任务都需要看到自己版本的 errno，因此 errno 更为有用。VxWorks 把 errno 作为每个任务的上下文，在每次任务上下文交换时，errno 将被同时保存与恢复。

同样，中断服务程序也需要自己版本的 errno。这是由内核提供的作为中断一部分的中断处理程序进入和退出代码自动在中断堆栈中对 errno 进行保存和恢复来实现的。因此，不考虑 VxWorks 上下文，通过直接操纵全局变量 errno，即可访问出错代码。

几乎所有的 VxWorks 函数都遵循一个约定：由函数实际返回的值来简单地指示函数操作成功与否。许多函数仅仅返回状态值 OK(＝0),有些函数正常时返回一个非负整数,例如 open()返回一个文件描述符，出错返回 ERROR，指明发生了一个错误。对于那些返回类型为指针的函数，通常用返回 NULL（＝0）来指明发生了错误。

在大多数情况下，函数在返回一个错误指示时，也将设置 errno 值来指明发生的是哪种特定错误。VxWorks 程序从不会清除全局变量 errno，因此它的值总是由最后发生的错误状态设置。如果一个 VxWorks 子程序在调用其他程序时得到一个错误，它通常返回自己的错误指

示，而不去修改 errno。因此在底层程序设置的 errno 值作为一个错误类型的指示仍然有效。

例如，中断连接函数 intConnect()，它将一个硬件中断与某一用户程序相联系，调用 malloc()来分配内存，在所分配的内存中建立中断处理程序。如果系统没有足够内存，调用 malloc()会失败，它设置 errno 来指明内存分配库 memLib 遇到这种没有足够内存（insufficient memory）用于分配的错误。Malloc()然后返回 NULL 来表明分配失败。IntConnect() 函数接受由 malloc()返回的 NULL，返回它自己的错误指示 ERROR。然而，它没有改变 errno，errno 仍然是由 malloc()设置的“insufficient memory”。

例如，

```
if ((pNew = malloc (CHUNK_SIZE)) = = NULL)
return ( ERROR) ;
```

Wind 推荐用户在自己的子程序中使用这种机制。同时设置检查 errno 也可以作为调试的一种技术手段。如果 errno 的值在错误状态号表（error-status symbol table：statSymTb）中对应一个字符串，那么调用函数 printErrno()可以显示这一字符串。

在 VxWorks 中，errno 值编码由 4 个字节组成。两个高字节表示产生错误的模块，指明产生错误的库（对应一个头文件）。另外两个字节表示错误号，指示在这个库中（头文件）特定的错误。所有 VxWorks 模块从 1～500 编码，errno 值为 0 的模块用于兼容。

头文件 targe/vwModNum.h 定义了高 16 位对应的模块（头文件）。例如 0xd0003，高 16 位是 0xd，对应的模块是 M_iosLib，对应的头文件是 target/h/iosLib.h。打开这个文件，就可以找到低 16 位错误号 3 对应的意思是 S_iosLib_INVALID_FILE_DESCRIPTOR，即无效文件描述符。

另外，如果在分配操作系统时包含了 INCLUDE_STAT_SYM_TBL 选项，那么就可以在 shell 中用 printErrno()打印错误信息。

应用程序可以使用所有其他的 errno 值，也就是说大于等于 128256（501<<16）的所有正整数和所有负数。

2.2.2　任务间通信和同步机制

VxWorks 支持各种任务间通信机制，提供了多样的任务间通信方式，主要有如下 5 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，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。
- 二进制信号，用于异常处理。

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。

1．共享内存

任务间通信的最简单的方法是采用共享内存，即相关的各个任务分享属于它们的地址空间的同一内存区域。因为所有任务都存在于单一的线性地址空间内，任务间共享数据。全局变量、线性队列、环形队列、链表、指针都可被运行在不同上下文的代码所共享。

2．互斥

当某一地址空间用于数据交换时，为了避免冲突，要对内存锁定。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，有可能得到错误值，这样必须以某种手段确保当一个任务在使用一个共享变量或文件时，其他任务不能做同样的操作。实现这样的功能主要有关中断、抢先禁止和用信号量锁定资源等方法。一般来说，关中断是最有效的解决互斥的方法。但这对于实时应用来说，它阻止系统对外部事件的响应，无法满足实时性的要求。同样，中断延迟也是不能接受的。有些实时应用一般不使用抢先禁止，大多采用信号量来解决互斥问题。

3．信号量

VxWorks 的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，VxWorks 提供以下 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、安全删除和回溯。
- 计数器　用于资源的数目较多的情况。

二进制信号量（Binary 信号量）是对应于事件的，有两种状态，置位和清除，分别代表事件的发生和结束。首先对于某种事件创建一个二值信号量，任务调用 SEMTAKE 来表示这个事件的发生，如果该事件已经发生，则该任务进入阻塞态等待，直到有某个任务调用 SEMGIVE，表示该事件已经结束，这时等待该事件的任务队列中的第一个的 SEMTAKE 返回 OK，解阻塞，其他任务仍被阻塞。

互斥信号量（Mutex 信号量）用来实现对共享资源的保护，因而是针对某个被保护的资源而创建的，比如全局变量、文件、I/O 设备等。访问资源的任务调用 SEMTAKE 获得对该资源的唯一访问权，如果资源被其他任务所占用，那么，该任务陷入阻塞态，直到资源被释放同时该任务成为等待队列中的第一个任务。拥有共享资源的任务可重复获得资源的使用权，此时，该资源的引用记数递增。Binary 和 Mutex 信号量的区别就在于所有权，所有的任务都可以发出信号表示一个事件的结束，而只有共享资源的所有者才能够释放该资源。

计数信号量（Count 信号量）在 Vxworks 中用的比较少，这是一个针对记数的信号量，已经预封装了互斥机制。

另外，为了从兼容性上考虑，VxWorks 还提供 POSIX 信号量和多处理器上信号量应用的支持。

Wind 信号量对于各种类型的信号量的控制提供了统一规范化的接口，当创建函数时要特别指明信号量类型。VxWorks 为 Wind 信号量提供的基本系统调用如表 2-4 所示。

表 2-4　　信号量的系统调用

信号量的系统调用	功 能 描 述
semBCreate()	创建（产生并激活）一个二进制信号量
semMCreate()	创建（产生并激活）一个互斥信号量
semCCreate()	创建（产生并激活）制一个计数信号量
semDelete()	中止并自由信号量
semTake()	获得信号量

续表

信号量的系统调用	功 能 描 述
semGive()	给出信号量
semFlush()	解锁所有正等待某一信号量的任务

- semBCreate 用于创建并初始化二进制信号量，它的原型是

```
SEM_ID       semBCreate
(
  int               options,          /*信号量选项*/
  SEM_B_STATE     initialState       /*信号量初始化状态值*/
)
```

信号量初始化状态值有两种：SEM_FULL（1）或 SEM_EMPTY（0）。选项参数指明被阻塞任务的入列方式：基于优先级（SEM_Q_PRIORITY）或者先进先出（SEM_Q_FIFO）。

- semCCreate 用于创建并初始化计数信号量，它的原型是

```
SEM_ID       semCCreate
(
  int          options,          /*信号量选项*/
  int          initialCount       /*信号量初始化计数值*/
)
```

选项参数指明被阻塞任务的入列方式：基于优先级（SEM_Q_PRIORITY）和先进先出（SEM_Q_FIFO）。

- semGive 用于给出信号量，它的原型是

```
STATUS       semGive
(
 SEM_ID     semId      /*所给出的信号量 ID 号*/
)
```

- semTake 用于获得信号量，它的原型是

```
STATUS       semTake
(
 SEM_ID     semId       /*所要得到的信号量 ID 号*/
 Int          timeout      /*等待时间*/
)
```

如果任务在规定时间内未得到信号量，函数 semTake 返回错误。等待时间值 WAIT_FOREVER 和 NO_WAIT 分别表示一直等待和不等待。

- semDelete 用于删除信号量，它的原型是

```
STATUS       semDelete
(
```

```
    SEM_ID      semId       /*要删除的信号量 ID 号*/
)
```

该函数释放与此信号量相关的资源，所有等待此信号量的任务将解除阻塞。

4．消息队列

现实的实时应用由一系列互相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，任务之间经由这个消息队列发送和接收消息。

VxWorks 系统中的消息队列是一个全双工的队列，同一队列上可收可发。它的逻辑模型如图 2-5 所示。

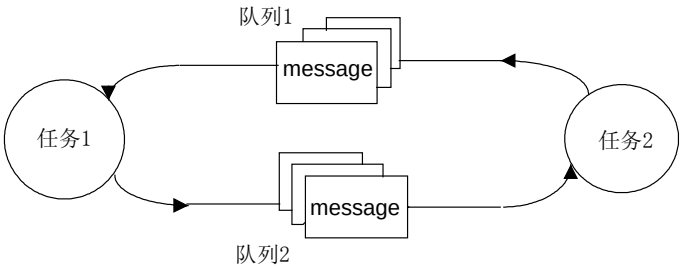

图 2-5　VxWorks 系统中任务间全双工信息传送

VxWorks 系统为 Wind 消息队列的管理提供的基本系统调用如表 2-5 所示。

表 2-5　　消息队列的系统调用

消息队列的系统调用	功 能 描 述
msgQCreate()	创建（产生并激活）消息队列
msgQDelete()	中止并自由信号量
msgQSend()	向消息队列发送消息
msgQReceive()	从消息队列接收消息

- msgQCreate 用于创建并初始化消息队列，它的原型是

```
MSG_Q_ID   msgQCreate
(
 int maxMsgs,          /*队列所能容纳的最大消息数目*/
 int maxMsgLength,     /*每一消息的最大长度*/
 int options           /*消息入列方式*/
)
```

消息入列方式有两种：MSG_Q_FIFO，先进先出，按时间先后顺序考虑；MSG_Q_PRIORITY，按消息优先级考虑。

- msgQSend 用于向一消息队列发送消息包，它的原型是

```
STATUS      msgQSend
 (
```

```
  MSG_Q_ID     msgQId,      /*所发向的消息队列名*/
  Char     *     buffer,       /*消息包所在缓冲区指针*/
  UINT          nBytes,      /*消息包长度*/
  Int     timeout,            /*等待的时间长度*/
  Int     priority            /*优先级*/
)
```

该调用将长度为 nBytes 的缓冲区 buffer 消息包发向消息队列 msgQId。如果任务正在等待接收该消息队列的消息包，消息将立即被送到第一个等待的任务。如果没有任务等待此消息，消息包被保留在消息队列中。

参数 timeout 指明当消息队列已满时，等到消息队列有空间时所等待的时间。超过该时间还没空间可用的话，消息包被舍弃。它有两个特殊值：NO_WAIT（0）表示不管消息包是否被发送立即返回；WAIT_FOREVER（-1）表示一直等待消息队列有空间可用。

参数 priority 指明发送的消息的优先级，可能值有 MSG_PRI_NORMAL(0)正常优先级，将消息置于消息队列的尾部；MSG_PRI_URGENT(1)紧急消息，将消息置于消息队列的首部。

- msgQReceive 用于接收消息，它的原型是

```
int msgQReceive
(
  MSG_Q_ID     msgQId,         /*接收消息的消息队列 ID 号*/
  Char     *     buffer,           /*接收消息的缓冲区指针*/
  UINT          maxNBytes,      /*缓冲区长度*/
  Int        timeout             /*等待时间*/
)
```

该调用从消息队列 msgQId 接收消息，将其拷贝到最大长度为 maxNBytes 的缓冲区 buffer。如果消息包长度超过 maxNBytes,多余部分被舍弃。等待时间 timeout 有两个特殊值：NO_WAIT（0）表示立即返回；WAIT_FOREVER（-1）表示一直等待消息队列有消息可取。

- msgQDelete 用于删除一个消息队列，它的原型是

```
STATUS    msgQDelete
(
 MSG_Q_ID     msgQId     /*要删除的消息队列 ID 号*/
)
```

调用此系统调用后，任何因发送或接收该消息队列的消息的任务都将解除阻塞，并返回错误 errno。

5．信号和管道

信号（signals）异步地改变了一个任务的工作流，也是一种重要的同步机制。任何任务和中断程序(ISR)都能向特定的任务发出信号，收到信号的任务立刻挂起它正在执行的线程，转而在下一时间调度运行特定任务信号处理程序。Wind 内核支持两种信号界面：UNIX BSD 形式的 signal 和兼容 POSIX 标准的 signal。在许多方面，信号类似于硬件中断。VxWorks 系统像其他经典操作系统一样，提供了一组特定信号，共 31 个。此外，VxWorks 还为用户保留了 SIGUSR1 和 SIGUSR2 两个信号。

管道用 VxWorks 的 I/O 系统提供一种灵活的消息传送机制，它是受驱动器 pipeDrv（由 VxWorks 提供）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、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，任务被阻塞。和消息队列类似，ISR 能向管道中写入信息，但不能从中读取。像 I/O 设备一样，管道有一个消息队列所没有的优势，即任务可以调用 select()等待一系列 I/O 设备上的数据。

6．网络通信机制

VxWorks 提供了强大的网络功能，能与许多其他主机系统进行通信。VxWorks 系统的网络组件完全兼容 4.3BSD，也兼容 SUN 公司的 NFS。这种广泛的协议支持在主机和 VxWorks 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，即远程文件存取，VxWorks 系统也支持远程过程调用。通过以太网，采用 TCP/IP 和 UDP/IP 协议在不同主机之间传送数据。

VxWorks 提供了如下一些网络工具完成信息传送：

- Sockets　完成运行在 VxWorks 系统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 VxWorks 或其它系统）上的过程。
- 远程文件存取　VxWorks 任务可采用 NFS、RSH、FTP、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 系统提供的网络组件结构如图 2-6 所示。

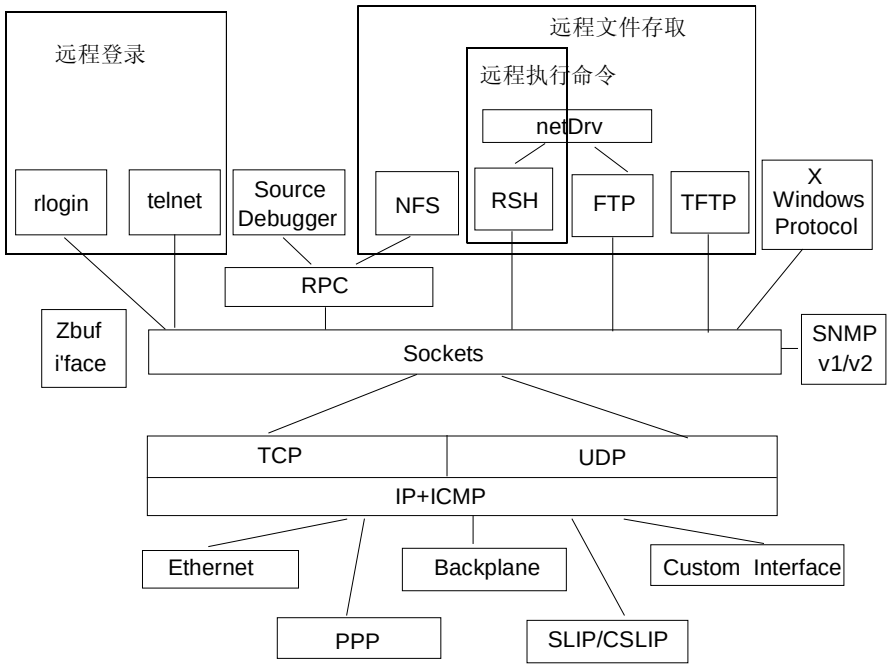

图 2-6　VxWorks 系统的网络组件框架图

Vxworks 系统和网络协议的接口是靠套接字（Sockets）来实现的。Sockets 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。通信的基础是套接字口，一个通信口是套接字口的一个端口，这个端口可以与一个端口名对应。一个正在被使用的套接字口都有它的类型和与其相关的任务。套接字口存在于通信域中。通信域是为了处理一般的线程通过套接字口通信而引进的一种抽象概念。套接字口通常和同一个域中的套接字口交换数据（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）。各个任务使用这个域互相之间用 Internet 协议来进行通信。

套接字口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口间通信。不过只要底层的通信协议允许，不同类型的套接字口间也照样可以通信。用户目前使用两种套接字口，即流套接字口（采用 TCP 协议）和数据报套接字口（采用 UDP 协议）。流套接字口提供了双向的、有序的、无重复并且无记录边界的数据流服务。数据报套接字口支持双向的数据流，但并不保证是可靠、有序、无重复的。也就是说，一个从数据报套接字口接收信息的任务有可能发现信息重复了，或者和发出时的顺序不同。数据报套接字口的一个重要特点是它保留了记录边界。对于这一特点，数据报套接字口采用了与现在许多分组交换网络（例如以太网）非常相似的模型。

套接字口（socket）通信的最大优点是过程间的通信是完全对等的，不管网络中过程的定位或主机所运行的操作系统。一般来说，流套接字口提供了可靠的面向连接的服务，应用较广泛。其应用程序时序图如图 2-7 所示。

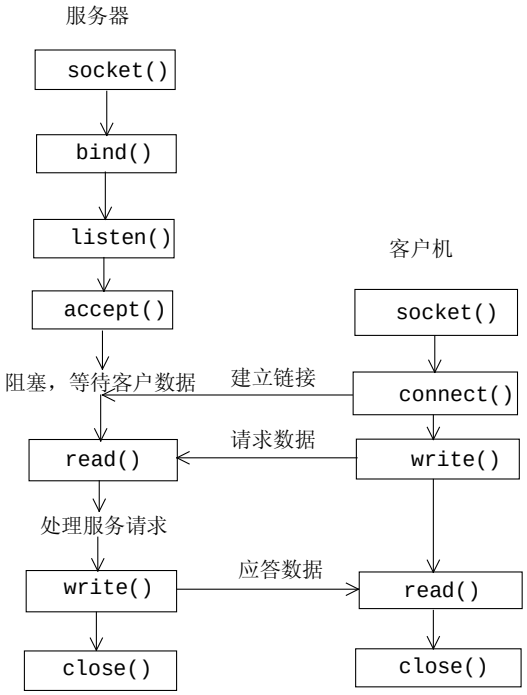

图 2-7　流套接字口应用程序基本流程图

Socket 用到的基本系统调用如表 2-6 所示。

表 2-6　　Socket 系统调用

系统调用名	功 能 描 述
socket()	创建一个套接字口
bind()	给套接字口分配名称
listen()	打开 TCP 套接字口连接
accept()	完成套接字口间连接
connect()	请求连接套接字口
shutdown()	关闭套接字口间连接
send()	向 TCP 套接字口发送数据
recv()	从 TCP 套接字口接收数据
select()	完成同步 I/O 传输
read()	从套接字口读取信息
write()	向套接字口写入信息
ioctl()	完成对套接字口的控制
close()	关闭套接字口

2.2.3　中断机制

实时系统中硬件中断处理是至关重要的,因为它是以中断方式通知系统外部事件的发生。为了快速响应中断，中断服务程序 ISR 运行在特定的空间，不同于其他任何任务，因此中断处理没有任务的上下文切换。

VxWorks 系统提供了灵活的中断处理机制，它为中断处理提供的系统调用如表 2-7 所示

表 2-7　　中断系统调用

系统调用名	功 能 描 述
intConnect()	将 C 函数和中断向量连接
intCount()	得到当前中断嵌套深度
intLevelSet()	设置程序的中断级别
intLock()	使中断禁止
intUnlock()	开中断
intVecSet()	设置异常向量
intVecGet()	得到异常向量

在 VxWorks 系统中，所有的中断服务程序使用同一中断堆栈，它在系统启动时就已根据具体的配置参数进行了分配和初始化，必须保证它的大小，以使它能满足最坏的多中断情况。中断也有缺陷：ISR 不运行在常规的任务上下文中，它没有任务控制块。对于 ISR 的基本约束就是它们不能激活那些可能使调用程序阻塞的函数，例如，它不能获取信号量，因如果该信号量不可利用，内核会试图让调用者切换到挂起态。不过，Vxworks 系统允许 ISR 给出信号量，即调用 semGive()。

一个 ISR 通常与一个或多个任务进行通信,直接或者间接地作为输入输出事务的一部分。

这种通信的本质是驱动任务执行，从而处理中断和各种情况。这与任务到任务的通信和同步基本相同，但是有如下两点不同：

首先，一个 ISR 通常作为通信或同步的发起者，它通常返回一个信号量、向队列发送一个信息包或事件给一个任务。ISR 很少作为信息的接收者，它不可以等待接收信息包或事件。

其次，ISR 内的系统调用总是立即返回 ISR 本身。例如，即使 ISR 通过发送信息包唤醒了一个很高优先级的任务，它也首先必须返回 ISR。这是因为 ISR 必须先完成。

2.2.4　定时管理机制

VxWorks 系统提供丰富的定时管理和时钟管理，具体包括以下几个方面：

- 维护系统日历时钟。
- 在任务等待消息包、信号量、事件或内存段时的超时处理。
- 以一定的时间间隔或在特定的时间唤醒或发送告警到一个任务。
- 处理任务调度中的时间片轮转。

VxWorks 系统这些功能都依赖于周期性的定时中断，离开实时时钟或定时器硬件就无法工作。

VxWorks 系统为定时管理和时钟管理提供的系统调用如表 2-8 所示。

表 2-8　　时钟系统中中的调用

系统调用名	功 能 描 述
tickAnnounce()	通知系统内核时钟“滴答”
tickSet()	设定内核时钟计数器值
tickGet()	得到内核时钟计数器值
timer_create()	创建时钟
timer_gettime()	获得时钟器给定值的当前剩余值
timer_settime()	设定时钟值
timer_connect()	连接用户函数和时钟信号
timer_cancel()	取消一个时钟
sysClkRateSet()	设置系统时钟速率

另外，VxWorks 系统还提供了看门狗定时器。看门狗定时器作为系统时钟中断服务程序的一部分，允许 C 语言函数指明某一时间延迟。一般来说，被看门狗定时器激活的函数运行在系统时钟中断级。然而，如果内核不能立即运行该函数，则函数被放入 tExcTask 工作队列中。在 tExcTask 工作队列中的任务运行在最高优先级 0。

VxWorks 系统为看门狗定时器提供的系统调用如表 2-9 所示。

表 2-9　　为看门狗定时器提供的系统调用

系统调用名	功 能 描 述
wdCreate()	分配并初始化看门狗定时器
wdDelete()	中止并解除看门狗定时器
wdStart()	启动看门狗定时器

续表

系统调用名	功 能 描 述
wdCancel()	取消当前正在计数的看门狗定时器

- wdCreate 用于创建看门狗定时器，它的原型是

```
WDOG_ID      wdCreate(void)
```

- wdStart 用于启动定时器，它的原型是

```
STATUS     wdStart
(
 WDOG_ID     wdId,        /*看门狗定时器 ID 号*/
 Int            delay,        /*延迟值，以滴答计*/
 FUNCPTR     pRoutine,     /*超时函数*/
 Int            parameter     /*超时函数的参数*/
 )
```

- wdCancel 用于取消一个当前工作的定时器，它的原型是

```
STATUS       wdCancel
(
  WDOG_ID     wdId      /*被取消的定时器 ID 号*/
)
```

该调用只是让定时器的延迟值为零来取消其工作。

- wdDelete 用于删除定时器，它的原型是

```
STATUS       wdDelete
(
  WDOG_ID     wdIdQY      /*被删除的定时器 ID 号*/
)
```

2.2.5 内存管理

Vxworks 系统的基本内存管理是采用 FLAT（平板）模式，所有进程直接寻址绝对物理地址，在进程间可以共享代码段、数据段和 BSS 段。Vxworks 系统中的内存布局如图 2-8 所示。

在 Vxworks 系统中由特定的宏定义和系统函数获得标准的系统内存地址，分别标志系统物理内存的起始地址、Vxworks 的加载地址、系统内存区的起始地址、用户保留内存的起始地址以及物理内存的容量。可以看出，在系统内存区中有一段是分配给调试代理的，专门用于 Host 与 Target 间调试信息的传递和处理，而用户任务的任务堆栈、动态内存分配则都来自于剩余的系统内存区。内存管理模块运行在用户态。举个例子，考虑该模块提供的申请内存函数 malloc()，用于分配指定大小的内存并返回一个起始指针。空闲内存是通过查找一个空闲内存块的队列找到的，在查找过程中要用信号量保护这个非抢占资源，分配过程如下：

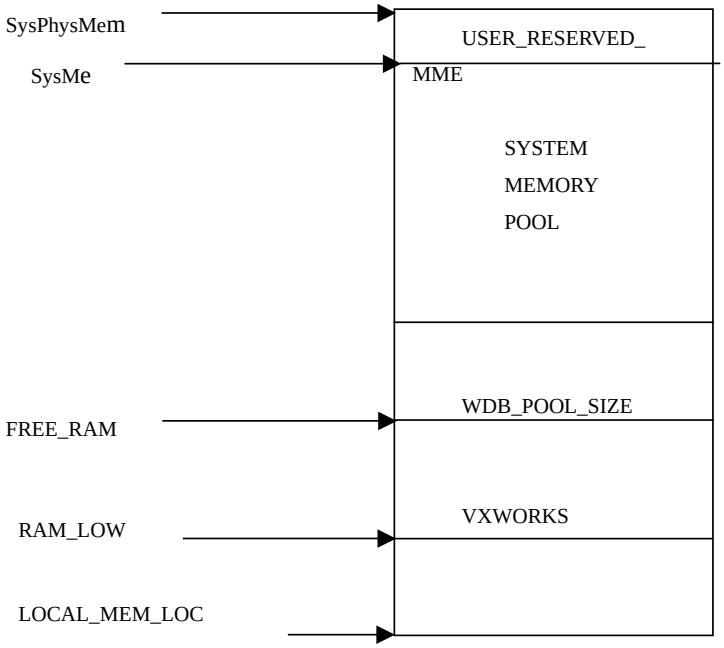

图 2-8　Vxworks 系统中的内存布局

（1）获得互斥信号量。

（2）查找内存队列链表。

（3）释放信号量。

由以上可见，对信号量的操作是原子级的，不可打断，分配时大量的时间是花费在查找链表的过程中，但这不会影响到系统效率，因为这个过程是发生在任务的上下文中，是可以被其他高优先级的任务所抢占的。而在非微内核结构的实时内核中，内存分配例程操作如下：

（1）进入内核。

（2）搜索空闲内存块链表。

（3）退出内核。

整个内存分配发生在内核级，此时任务抢占被禁止，如果高优先级的任务在此时变为就绪态，它必须等待直到内核为低优先级的任务完成内存分配。有些操作系统甚至在这段时间禁止中断。这肯定影响了系统的实时性和效率。

实时系统任务执行中，有时对于内存分配时对链表查询的时间也是不可忍受的。一般情况下，我们采用 Message Queue 来代替 malloc()，在系统启动之初就分配好可能用到的内存块，并将它们放入消息队列，那么，在以后对实时要求很高的场合中，可以直接从消息队列中获得空闲内存块的指针，避免了查找时间的耗费。

2.2.6　I/O 与文件系统

在 VxWorks 系统中,I/O 系统可分为字符 I/O 和块 I/O,文件系统是一种块 I/O 操作设备。Vxworks 里的 I/O 系统实现可以分为几个层次。底层的设备驱动、I/O 系统、其他系统模块（包括文件系统）、应用程序，如图 2-9 所示。

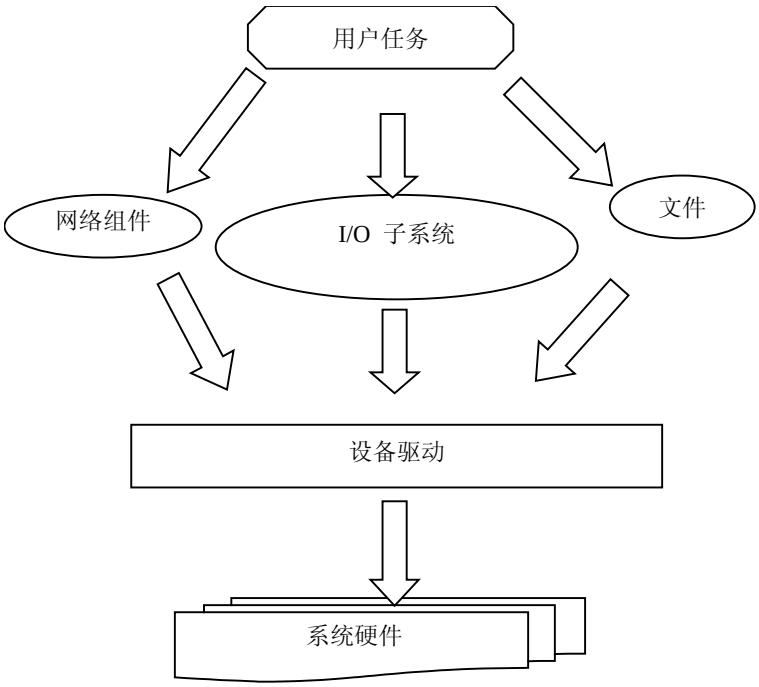

图 2-9 VxWorks 系统中 I/O 系统的位置

从图中可以看出，用户任务可以直接操作 Driver，这是在对速度有极高要求的、简单的、非面向流的场合中使用的。这种操作违反了系统结构，一般不建议使用。

I/O 是系统对于设备驱动的一个包装，通过 I/O 可以实现设备无关，并对应用提供一致的 API 接口，和 I/O 重定向、多 I/O 等待。

I/O 系统使用标准的接口调用设备驱动程序相应的函数，这些接口包括 Create()、Open()、Close()、Write()、Read()、IOCtl()、Remove()，它们分别与每个设备驱动的 xxCreate()、xxOpen、xxRemove()等相对应的。一个设备驱动在生效之前必须安装。安装就是把它提供的函数与 I/O 系统连接起来，实现 I/O 编址、操作规程、参数模型。在 Vxworks 的 I/O 子系统里面，预先安装的设备驱动只有两个，一个是串口，ttyDrv；另一个是管道，pipeDrv。

在 I/O 子系统之上还可以加入一些库文件对其进一步封装，如图 2-10 所示。

图中 STDIO 是带有缓冲区的标准 I/O 函数库，缓冲区减少了对设备驱动的访问次数，提高了速度，在 ansiStdio 库中实现。FIO 是格式化的 I/O 访问函数，不带缓冲区，在系统的 FIOLIB 库中实现。

文件系统是在 I/O 基础上实现的一个系统组件，针对的是块 I/O 操作。实现文件系统要添加一些该文件系统特有的设备访问例程，还要有块 I/O 的驱动程序，如图 2-10 所示。通过文件系统访问设备具有抽象性好的优点。目前，VxWorks 支持的文件系统有：

- MS DOS6.2 兼容文件系统 即 dosFsLib。
- 物理盘文件系统即 rawFsLib。
- rt11 文件系统即 rt11FsLib。
- ISO 9660 CdRom 即 cdromFsLib。

- SCSI 设备即 tapeFsLib。

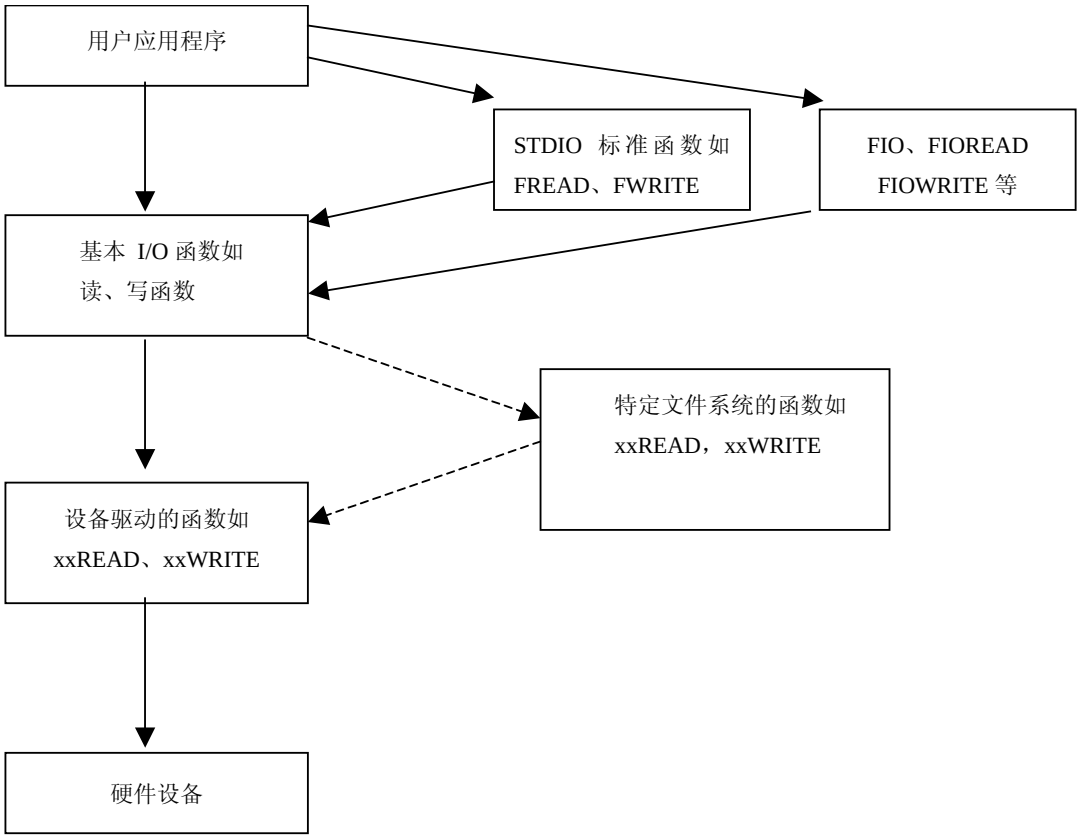

图 2-10　VxWorks 系统 I/O 子系统的封装结构

另外，提供的块设备驱动有 ramDrv、scsiDrv、tffsDrv 分别可以支持 ram,、csi 设备和 flash 设备，还支持第三方驱动。也就是说，至少可以立即在这些设备上实现以上的几种文件系统。

2.3　VxWorks 系统开发经验

VxWorks 系统是专为嵌入式实时应用而设计的模块化的实时操作系统。对于用户来说，一个实时应用软件是由板级支持包 BSP、操作系统内核及用户选用组件、中断服务程序 ISR 组成的。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制中要注重以下问题。

2.3.1　正确划分任务

1．功能内聚性

对于功能联系比较紧密的各工作可以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，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

对于实时性要求比较高的任务，要以高优先级运行，以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。

2.3.2 防止任务异常

操作系统发生死锁、饥饿或者优先级翻转都会让任务处于异常状态。死锁是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。在程序设计过程中要注意对死锁的预防，一个是尽量使互斥资源在相同优先级任务中使用，必须在不同优先级任务中使用时，要注意对死锁的解锁处理。

饥饿是指优先级较低的任务长期得不到系统资源（主要是指 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。合理的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主要策略。

任务的优先级翻转是实时多任务操作系统的热门话题，它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翻转进行预防。在操作系统未提供保护的情况下，就需要编程人员在编程的时候注意避免优先级翻转的情况发生（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（如动态的进行优先级提升）。VxWorks 系统提供了自身的防止优先级翻转机制，它主要通过限制对系统调用的使用来实现。

2.3.3 正确运用函数的可重入性

在一个多任务环境中，函数的可重入性是十分重要的。可重入函数是一个可以被多个任务调用的过程，任务在调用时不必担心数据是否会出错。在写函数时要尽量使用局部变量（例如寄存器、堆栈中的变量），对于要使用的全局变量要加以保护（例如采用关中断、信号量等），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，与它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。为了确保每一个任务控制自己的私有变量，在一个可重入的 C 函数中，须将这样的变量声明为局部变量。C 编译器将这样的变量存放在调用栈上或寄存器里。

在 VxWorks 中，多个任务可调用同一子函数或函数库。VxWorks 系统利用动态连接工具使这样做相当方便，这种共享代码让系统更加高效、更加易于维护。

VxWorks 系统主要采用如下几种可重入技术。

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，多个任务同时使用这种子函数，不会互相影响，因为它们各有自己的堆栈空间。

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，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运行代码的临界区。

3．任务变量

一些公用函数要求对于每一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 字节变量加入到任务上下文中，当任务切换时变量的值也切换。

编写可重入的函数，必须遵循以下的规则：

- 将所有的局部变量申明为 auto（缺省态）或寄存器型。
- 尽量不要使用 static 或 extern 变量。如有必要，要用互斥机制进行保护。

2.3.4　使用名称访问资源

通过任务名、消息队列名、信号量名来调用这些资源，能保证应用系统运行的可靠性，同时也便于程序的阅读。例如系统调用 taskName()、taskNameToId()、taskIsSelf()等，方便了用户对资源的管理，而且更加直观化。

2.3.5　用户任务优先级确定

VxWorks 系统中优先级分为 256 级，从 0 到 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，对于用户各任务优先级的确定，如何让各任务间良好地协同工作，有待于用户根据任务的紧急程度以及实际情况进行给定，调测过程中的摸索总结也很重要。

2.4　VxWorks 系统开发模型概述

Tornado 集成环境提供了高效明晰的图形化的实时应用开发平台，它包括一套完整的面向嵌入式系统的开发和调测工具。Tornado 环境采用主机-目标机交叉开发模型，应用程序在主机的 Windows 环境下编译链接生成可执行文件，下载到目标机，通过主机上的目标服务器与目标机上的目标代理程序的通信完成对应用程序的调测、分析。它主要由以下几部分组成：

- VxWorks——高性能的实时操作系统。
- 应用编译工具。
- 交互开发工具。

VxWorks 系统的开发模型如图 2-11 所示。

PC 机运行的是 Tornado 集成开发工具，包括编辑器、WindSh、CrossWind 交叉工具、Browser 浏览器、WindConfig 和 Target Server（目标服务器）。目标系统上运行的是应用程序（Application）、VxWorks 内核映像以及目标代理（Target Agent）。

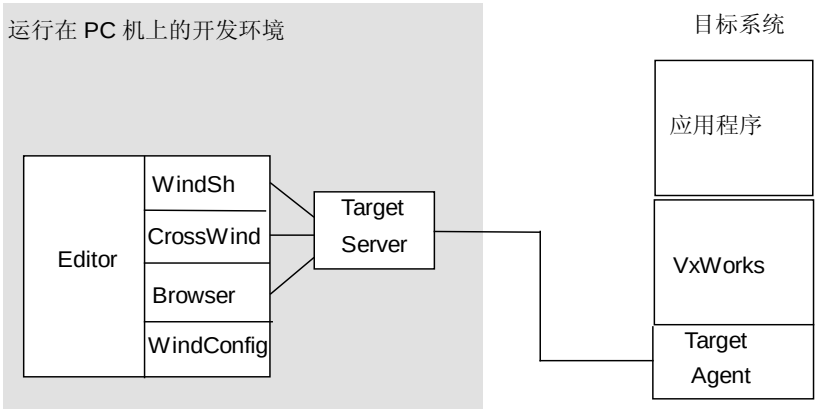

图 2-11　VxWorks 应用系统开发模型

下面对图 2-11 中 Tornado 集成环境用到的各组件功能进行简单介绍。

- Tornado 开发环境　Tornado 是集成了编辑器、编译器、调试器于一体的高度集成的窗口环境，同样也可以从 Shell 窗口下发命令和浏览。
- WindConfig：Tornado 系统配置　通过 WindConfig 可选择需要的组件组成 VxWorks 实时环境，并生成板级支持包 BSP 的配置。
- WindSh:Tornado 外壳　WindSh 是一个驻留在主机内的 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 和应用函数。Tornado 外壳还能解释常规的工具命令语言 TCL。
- 浏览器　Tornado 浏览器可查看内存分配情况、系统目标（如任务、消息队列、信号量等)。这些信息可周期性地进行更新。
- CrossWind:源码级调试器　源码级调试器 CrossWind 提供了图形和命令行方式来调试，可进行指定任务或系统级断点设置、单步执行、异常处理。
- 驻留主机的目标服务器　目标服务器管理主机与目标机的通信，所有与目标机的交互工具都通过目标服务器，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。
- Tornado 注册器　所有目标服务器在注册器中注册其提供的服务。注册器映射用户定义的目标名到目标服务器网络地址。
- VxWOrks　Tornado 包含了 VxWorks 操作系统。
- 目标代理程序　目标代理程序是一个驻留在目标机中的联系 Tornado 工具和目标机系统的组件。一般来说，目标代理程序往往是不可见的。

2.4.1　系统启动

1．启动盘的制作

在实时应用系统的开发调试阶段，往往采用以 PC 机作为目标机来调试程序。主机 PC 和目标机 PC 之间可采取串口或是网口进行连接。由于大多数目标已配有网卡，网络连接成为最

简单快速的连接方式。串口连接通信速率不高，但也有它自己的优点。系统级任务调试（如中断服务程序 ISR）需使通信方式工作在 Polled 模式，网口连接就不支持，因此可以裁剪掉系统中网络部分，以使 VxWorks 系统更小，满足目标板的内存约束。下面分别对这两种通信方式下目标机 VxWorks 系统启动盘的制作做一简要介绍。

串口通信时目标机 VxWorks 系统启动盘的制作步骤：

（1）修改通用配置文件\\Tornado\target\config\pc486\config.h。

在 config.h 文件中加入以下宏定义：

```
#undef     WDB_COMM_TYPE
#define     WDB_COMM_TYPE     WDB_COMM_SERIAL        /*定义通信方式为串口连接*/
#define     WDB_TTY_CHANNEL     1                          /*通道号*/
#define     WDB_TTY_BAUD     9600                          /*串口速率，可设置至 38400*/
```

并且修改#define DEFAULT_BOOT_LINE 中 vxWorks 为 vxWorks.st。

（2）在 Tornado 集成环境中点取 Project 菜单，选取 Make PC486，选择 Common Target，先进行 clean 操作，再选择 Boot Rom Target，进行 bootrom_uncmp 操作；再选择 VxWorks Target，进行 vxworks.st 操作。

（3）拷贝\\Tornado\target\config\pc486\bootrom_uncmp 至\\Tornado\host\bin 下。

（4）重命名文件 bootrom_uncmp 为 bootrom。

（5）准备一张已格式化的空盘插入软驱。

（6）在目录\\Tornado\host\bin 下执行命令 mkboot a: bootrom 。

（7）拷贝\\Tornado\target\config\pc486\VxWorks.*至软盘。

（8）将系统制作盘插入目标机软驱，加电启动目标机即可载入 VxWorkst 系统。

网口通信时目标机 VxWorks 系统启动盘的制作步骤：

（1）配置目标机网卡，设置其中断号和输入输出范围（I/O 地址）。

（2）修改通用配置文件\\Tornado\target\config\pc486\config.h。

针对不同的网卡，其名称不同，如 NE2000 及其兼容网卡为 ENE，3COM 以太网卡为 ELT，Intel 网卡为 EEX。

在 config.h 文件中修改相应网卡类型（如网卡为 3COM 网卡）的定义部分：

```
#define     IO_ADRS_ELT          网卡 I/O 地址
#define     INT_LVL_ELT           网卡中断号
```

并且修改#define DEFAULT_BOOT_LINE 的定义：

```
#define     DEFAULT_BOOT_LINE    \
```

"elt(0,0)主机标识名:C:\\tornado\\target\\config\\pc486\\vxWorks h=主机 IP e=目标机 IP u=登录用户名 pw=口令 tn=目标机名"

（3）主机信息的确定。主机操作系统 Win95 安装目录下有一文件 hosts.sam，向其中加入：

主机 IP 主机名

目标机 IP 目标机名

（4）在 Tornado 集成环境中点取 Project 菜单，选取 Make PC486，选择 Common Target，先进行 clean 操作；再选择 Boot Rom Target，进行 bootrom_uncmp 操作；再选择 VxWorks Target，进行 vxworks 操作。

（5）拷贝\\Tornado\target\config\pc486\bootrom_uncmp 至\\Tornado\host\bin 下。

（6）重命名文件 bootrom_uncmp 为 bootrom。

（7）准备一张已格式化的空盘插入软驱。

（8）在目录\\Tornado\host\bin 下执行命令 mkboot a: bootrom 。

（9） 启动 Tornado 组件 FTP Server，在 WFTPD 窗口中选择菜单 Security 中的 User/right...，在其弹出窗口中选择 New User...，根据提示信息输入登录用户名和口令，并且要指定下载文件 vxWorks 所在根目录；还必选取主菜单 Logging 中 Log options，使 Enable Logging、Gets、Logins、Commands、Warnings 等。

（10）将系统制作盘插入目标机软驱，加电启动目标机即可通过 FTP 方式从主机下载 VxWorkst 系统。

2. 开发主机 Tornado 环境配置

串口连接时主机 Tornado 开发环境的目标服务器配置操作如下：

（1）在 Tornado 集成环境中点取 Tools 菜单，选取 Target Server，选择 config...。

（2）在 Configure Target Servers 窗口中先给目标服务器命名。

（3）在配置目标服务器窗口中的“Change Property”窗口中 选择 Back End，在“Available Back”窗口中选择 wdbserial，再在“Serial Port”窗口中选择主机与目标机连接所占用的串口号（COM1，COM2），再在“Speed(bps)”窗口中选择主机与目标机间串口速率。

（4）在配置目标服务器窗口中的“Change Property”窗口中选择 Core File and Symbols，选择 File 为 BSP 目标文件所在目录（本例为 PC486 目录）的 VxWorks.st，并选取为 All Symbols。

（5）在配置目标服务器窗口中的“Change Property”窗口中的其它各项可根据需要选择。

网口连接时主机 Tornado 开发环境的目标服务器配置操作如下：

（1）在 Tornado 集成环境中点取 Tools 菜单，选取 Target Server，选择 config...。

（2）在 Configure Target Servers 窗口中先给目标服务器命名。

（3）在配置目标服务器窗口中的“Change Property”窗口中选择 Back End，在“Available Back”窗口中选择 wdbrpc，在“Target IP/Address”窗口中输入目标机 IP。

（4）在配置目标服务器窗口中的“Change Property”窗口中选择 Core File and Symbols，选择 File 为 BSP 目标文件所在目录（本例为 PC486 目录）的 VxWorks，并选取为 All Symbols。

（5）在配置目标服务器窗口中的“Change Property”窗口中的其他各项可根据需要选择。

2.4.2 应用系统配置

运行在目标板上的系统映像是个二进制模块。大多数情况下，用户会发现系统映像占用

空间较大。然而，用户可根据需要裁剪系统配置，降低系统占用资源。

下面针对配置系统映像从以下两方面进行说明：

- VxWorks 板级支持包（BSP）。
- VxWorks 配置文件、可选项、参数。

1．板级支持包 BSP

Tornado 目录下 config/bspname 包含板级支持包 BSP，它由运行 VxWorks 的某些硬件驱动文件组成，如有串行线的 VME 板、时钟和其他设备。文件包括：**Makefile**、**sysLib.c**、**sysSerial.c**、**sysALib.s**、**romInit.s**、***bspname*.h** 和 **config.h**。文件 **sysLib.c** 以硬件独立方式提供 VxWorks 和应用程序间的板级联系，包括：

- 初始化函数
  - 初始化硬件到已知状态
  - 标识系统
  - 初始化设备，如 SCSI 或常规设备
- 内存/地址空间函数
  - 得到板上内存大小
  - 总线地址空间
  - 设定/获得非易失性 RAM
  - 定义板的内存位图（可选）
  - 为有 MMU 的处理器定义虚拟内存到物理内存的映射
- 总线中断函数
  - 打开/关闭总线中断
  - 产生总线中断
- 时钟/定时器函数
  - 使能/不能定时中断
  - 设置定时器的周期性
  - 邮箱/位置监视函数（可选）
  - 使能邮箱/位置监视中断

在目录 config/all 的配置文件 usrConfig.c 和 bootConfig.c 负责启动库函数。设备驱动可调用内存和总线管理函数。

（1）虚拟内存

对于支持 MMU 的单板，数据结构 sysPhysMemDesc 用来定义虚拟内存到物理内存的映射。该数据一般定义在 sysLib.c 中，也有的在单独的文件 memDesc.c 中。它以数据结构 PHYS_MEM_DESC 的数组形式存在。sysPhysMemDesc 数组记录用户的系统配置。

（2）串行设备

文件 sysSerial.c 提供对目标板串口的初始化。实际的串口 I/O 设备在目录 src/drv/sio 下。ttyDrv 库使用串口 I/O 设备提供 VxWorks 的终端操作。

（3）初始化模块

romInit.s 包括汇编级初始化程序，sysALib.s 包含初始部分和具体系统的汇编级程序。

（4）板级支持包 BSP 的初始化

板级支持包 BSP 负责目标板硬件的初始化，实时内核的载入等。对于硬件初始化的顺序，大致可按表 2-10 所示形式进行。

表 2-10　　初始化

函　数	函 数 功 能	所 在 文 件
sysInit()	(a) 锁住中断；(b)禁用缓冲； (c) 用缺省值初始化系统中断表（仅 i960）； (d) 用缺省值初始化系统错误表（仅 i960）； (e) 初始化处理器寄存器到一缺省值； (f) 使回溯失效；(g)清除所有挂起中断； (h) 激活 usrInit()，指明启动类型。	sysALib.s
UsrInit()	(a) 对 bss 赋零； (b) 保存 bootType 于 sysStartType； (c) 调用 excVecInit(),初始化所有系统和缺省中断向量； (d) 依次调用 sysHwInit(),usrKernelInit(),kernelInit()	usrConfig.c
usrKernelInit()	依次调用 classLibInit(),taskLibInit(),taskHookInit(),semBLibInit(),semMLibInit(),semCLibInit(),semOLibInit(),wdLibInit(),msgQLibInit(),qInit(),workQInit()	usrKernel.c
kernelInit()	初始化并启动内核。 (a) 激活 intLockLevelSet()； (b) 从内存池顶部创建根堆栈和 TCB； (c) 调用 taskInit(),taskActivate(),用于 usrRoot()； (d) 调用 usrRoot().	kernelLib.c
UsrRoot()	初始化 I/O 系统，驱动器，设备（在 configAll.h 和 config.h 中指定） (a) 调用 sysClkConnect(),sysClkRateSet(),iosInit(),[ttyDrv()]； (b) 初始化 excInit(),logInit(),sigInit(). (c) 初始化管道,pipeDrv()； (d) stdioInit(),mathSoftInit()或 mathHardInit() (e) wdbConfig():配置并初始化目标代理机	usrConfig.c

在大多数目标板的板级支持包中，VxWorks 的入口点由两个函数：romInit()和 romStart()来完成，而非 sysInit()。具体基于 ROM 的 VxWorks 的初始化过程如下表 2-11 所示。

表 2-11　　基于 ROM 的初始化

函　　数	函 数 功 能	所 在 文 件
1. romInit()	(a) 禁止中断； (b) 保存启动类型； (c) 硬件初始化； (d) 调用 romStart()	romInit.s

续表

2. romStart()	(a) 将数据段从 ROM 拷贝到 RAM，清内存； (b) 将代码段从 ROM 拷贝到 RAM，有必要的话解压缩； (c) 调用 usrInit()	bootInit.c
3. usrInit()	初始化程序	usrConfig.c
4. usrKernelInit()	如果相应的配置文件被定义，对应函数被调用	usrKernel.c
5. kernelInit()	初始化并启动内核	kernelLib.c
6. usrRoot()	初始化 I/O 系统，驱动器，创建设备	usrConfig.c
7. Application routine	应用程序代码	Application source file

2. 配置 VxWorks

VxWorks 的配置头文件为 config/all/configAll.h 和 config/bspname/config.h. 当运行配置 VxWorks 的初始化时，这些文件被程序 usrConfig.c\bootConfig.c\bootInit.c 调用。在开发环境中，用户可能要测试几种不同的配置，或者用户想在不同情况下指明不同的目标代码。为了编译 VxWorks 满足不同情况，用户必须调整使用环境。

用户 Tornado 环境包括 3 部分：主机代码、目标代码和配置文件。缺省配置文件如表 2-12 所示。

表 2-12　　缺省配置文件

文件类型	位　　置
主机代码	$WIND_BASE/host/hosttype/bin
目标代码	TGT_DIR=$WIND_BASE/target
配置文件	CONFIG_ALL= TGT_DIR/config/all

用户可修改通用配置文件 configAll.h 和具体的目标板配置文件 config.h。许多可选特性和设备驱动用户在文件 config/all/usrConfig.c 模块中可按需调整。在这里可以选择调整的项目如表 2-13 所示。

表 2-13　　可调整的项目

宏	选　　择
INCLUDE_ADA	Ada 支持
INCLUDE_ANSI_XXX	各种 ANSI C 函数库选择
INCLUDE_BOOTP	BOOTP 支持
INCLUDE_CACHE_SUPPORT	缓冲支持

INCLUDE_CPLUS	C++支持
INCLUDE_CPLUS_XXX	各种 C++支持
INCLUDE_DEMO	使用简单的 demo 程序
INCLUDE_FTP_SERVER	FTP 服务器支持
INCLUDE_HW_FP	硬件浮点支持
INLCUDE_LOADER	驻留目标机目标模块加载包

续表

INCLUDE_LOGGING	注册工具
INCLUDE_MMU_BASIC	MMU 支持
INCLUDE_MSG_Q	消息队列支持
INCLUDE_NETWORK	网络支持
INCLUDE_POSIX_XXX	各种 POSIX 选择
INCLUDE_RLOGIN	用 rlogin 远端注册
INCLUDE_RPC	远程过程调用
INCLUDE_SEM_BINARY	二进制信号量
INCLUDE_SEM_COUNTING	计数信号量
INCLUDE_SEM_MUTEX	互斥信号量
INCLUDE_SHELL	C 语言解释器
INCLUDE_SPY	任务活动监视器
INCLUDE_WATCHDOGS	看门狗
INCLUDE_WDB	目标机代理

第 3 章　VxWorks 系统 BSP 基本概念

3.1　BSP 基础

BSP 即 Board Support Package，板级支持包。它来源于嵌入式操作系统与硬件无关的设计思想，操作系统被设计为运行在虚拟的硬件平台上。对于具体的硬件平台，与硬件相关的代码都被封装在 BSP 中，由 BSP 向上提供虚拟的硬件平台，BSP 与操作系统通过定义好的接口进行交互。BSP 是所有与硬件相关的代码体的集合，它们主要包括：

- 在一个系统被引导时，目标系统硬件初始化程序。
- 目标系统上设备的驱动程序，这些设备包括：定时器、以太网控制器、一组串行口和 SCSI 控制器等，控制这些设备的函数称为设备驱动程序。

除了用户的应用程序以外，BSP 是 VxWorks 映像的另一个源程序空间，需要用户来实现。BSP 是系统用来管理外设的部分。BSP 的初始化部分是指从系统上电复位开始直到 WIND KERNEL 和 tUsrRoot 启动的这段时间里系统的执行过程。驱动程序就是一些包含 I/O 操作的子函数。

BSP 具体的初始化包括：CPU Init、Board Init、System Init。CPU Init 初始化 CPU 的内部寄存器。Board Init 初始化智能 I/O 的寄存器，将 device 打通。System Init 为系统的运行准备数据结构，进行数据初始化。图 3-1 所示为 BSP 的初始化过程。

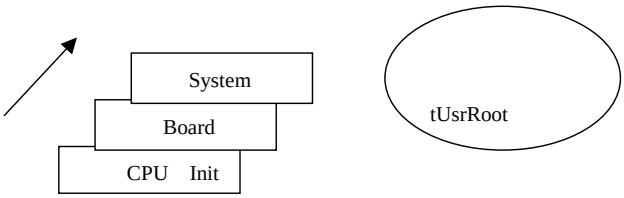

图 3-1　VxWorks 系统 BSP 的初始化过程

另外，我们可以将 VxWorks 系统的驱动程序抽象为 3 个层次：常规操作、与 I/O 的接口、与 Component 的接口。VxWorks 系统驱动程序的抽象逻辑如图 3-2 所示。

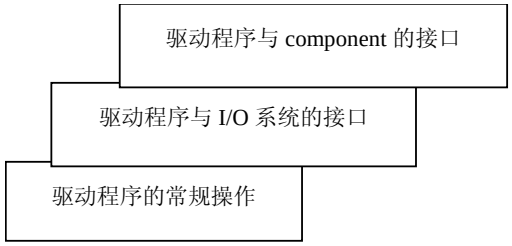

图 3-2　VxWorks 系统驱动程序的抽象逻辑

在 VxWorks 系统驱动程序的抽象逻辑中，常规操作是设备的固有操作逻辑，具有两层含义。它在微观上表现为 CPU 操作 device（设备）的寄存器。如图 3-3 所示，CPU 操作控制电路、数字电路、I/O 处理器的各种寄存器，例如对 I/O 进行编址。在宏观上它表现为具体的驱动操作。

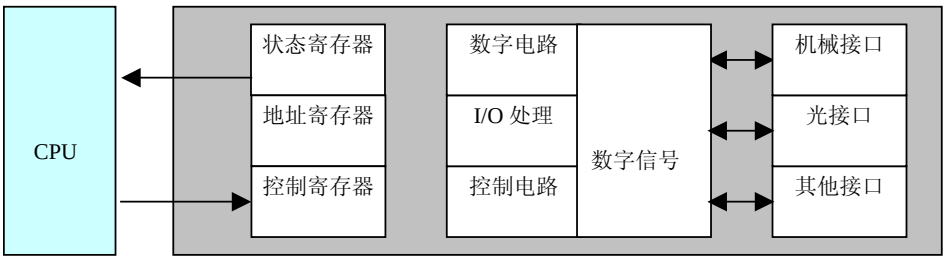

图 3-3　CPU 对设备寄存器的操作示意图

在 VxWorks 系统驱动程序的抽象逻辑中，驱动程序与 I/O 系统的接口、驱动程序与 Component 的接口有如下 3 层含义：

- I/O 管理
- 操作类型规定
- 参数规定

驱动程序（即 driver）与 I/O 系统的接口使 Driver 具有更好的层次性，驱动程序与 Component 的接口使 Driver 具有更好的抽象性。

WindRiver 公司提供大量的预制的支持许多商业主板及评估板的 BSP。同时，VxWorks 的开放式设计以及高度的可移植性使得用户在使用不同的目标板进行开发时，所做的移植工作量非常小。到目前为止，WindRiver 公司能够提供超过 200 个的 BSP，当用户在为自己的目标板开发 BSP 时，可以从 WindRiver 公司的标准 BSP 中选一个最接近的来加以修改。另外，WindRiver 还提供了 BSP Develop KIT，包括流行标准板的 BSP source code，以及 BSP 开发效果的校验工具，以方便用户。

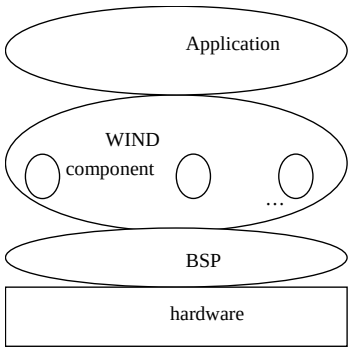

图 3-4　基于 VxWorks 系统的系统运行时结构

图 3-4 为基于 VxWorks 操作系统的系统运行时结构，说明了 BSP 在系统中的位置。从图中可以看出，BSP 向上层提供的接口有：

- 与 WIND 内核的接口。

- 与 VxWorks 系统的组件的接口。
- 与应用程序的接口（可以提供，但一般情况下不提倡使用）。

3.2　BSP 文件结构

VxWorks 系统的开发环境 Tornado 软件包安装后的目录结构如图 3-5 所示（For PowerPC 系列）。

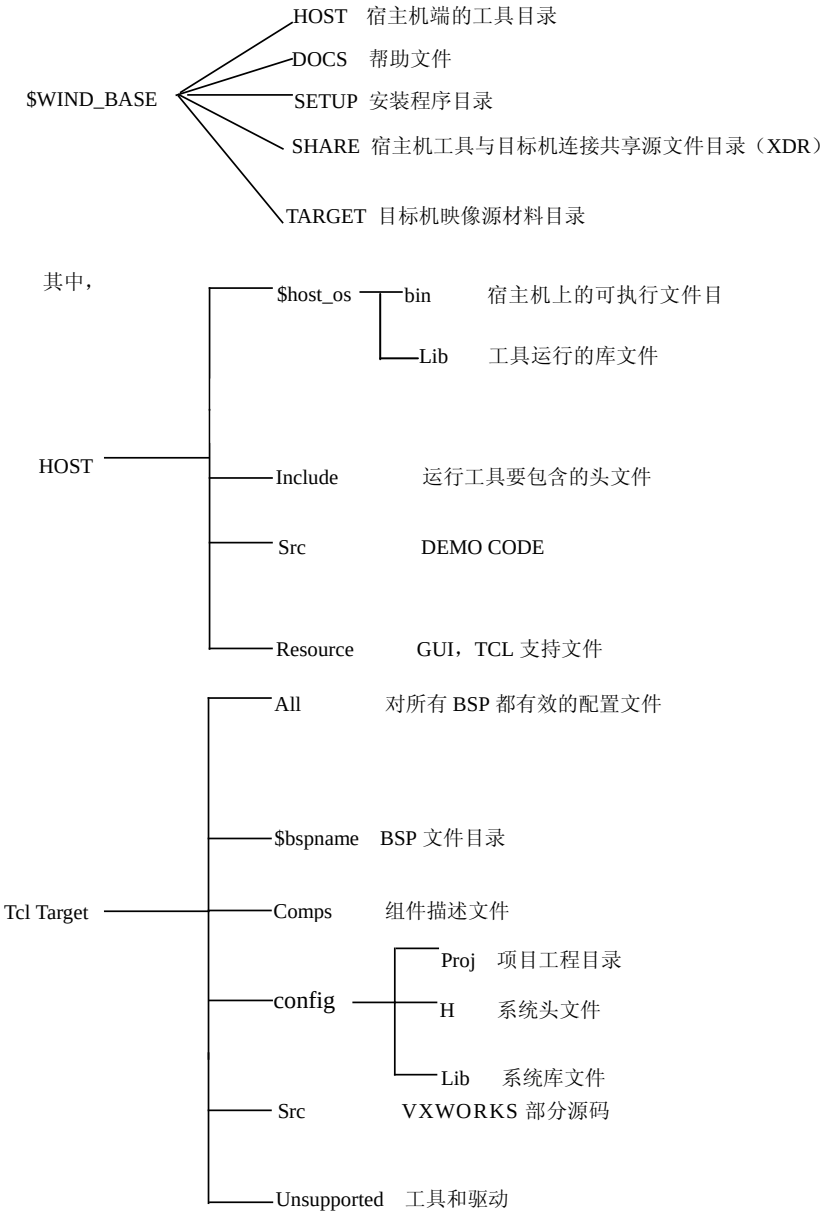

图 3-5　Tornado 软件目录结构图

3.3 VxWorks 系统的 BSP 开发过程

3.3.1 建立 BSP 开发环境

WindRiver 公司提供的软件产品分两部分：开发环境 Tornado 和实时操作系统 VxWorks。运行 Tornado 的一方为宿主机，运行 VxWorks 的一方为目标机，二者可以采用网线或串行线连接。一般不能在目标系统直接运行 WindRiver 提供的 VxWorks 映像，要根据实际的目标系统重新配置与生成，这个过程是在宿主机上完成的。在实际系统开发时，为了调试 BSP，宿主机与目标系统通过仿真器相连，其连接示意图如图 3-6 所示。

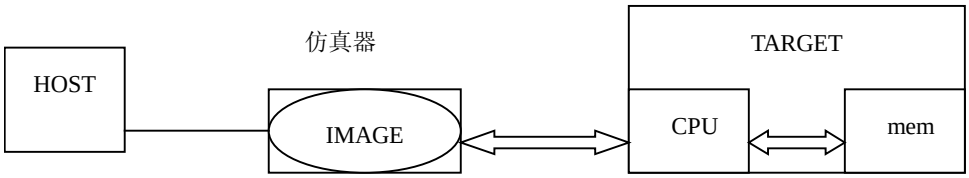

图 3-6 宿主机与目标系统的连接示意图

宿主机工作文件目录在\tornado\host 下，而目标系统的文件目录在\tornado\target 下，对 VxWorks 的配置主要是是通过对该目录下相关配置文件进行编辑与修改来完成的。

3.3.2 编辑修改 BSP 文件

开发 BSP 时，主要以目标板 CPU 的 BSP 文件为模板，在 tornado、target、config 目录下创建用户的 BSP 目录 bspname，把 tornado、target、config 下的文件和 BSP 模板文件拷贝到该目录下，建立工程，并修改相关的源文件，根据具体情况选择合适的 VxWorks 映像类型。

BSP 由 3 部分文件组成：

- 源文件、头文件，如 *bspname*.h、config.h、sysLib.c、romInit.s、 sysALib.s、sysSerial.c、configNet.h 等，以及编译描述文件 makefile。
- 派生文件。
- 二进制驱动程序模块。

BSP 的开发要根据具体目标板的硬件进行,可以根据目标系统对 tornado、target、config 目录下的下列文件进行编辑与修改。我们描述基于 ads8260 主板开发 BSP 时要修改的主要文件。

1. config\all

\tornado\target\config 目录包含的文件用来配置和生成一个专用 VxWorks 系统，它包括与目标系统相关的模块（子目录 bspname）和独立于目标系统的用户可修改的公共执行模块（子目录 all），系统配置就是在 config\all 目录包含的文件中完成的。VxWorks 的配置文件分为配置头文件和配置模块文件，通过对它们进行编辑和修改，利用 tornado 提供的工具进行配置生成满足用户目标板的操作系统映像。这里配置的头文件有\target\config\all\configAll.h 和\target\config\bspname\config.h，配置的模块文件

有\target\config\all\usrConfig.c 与\target\src\config 目录下的初始化模块。

（1）配置头文件

在配置头文件时主要是利用定义（#define）和去除 定义（#undef）实现对 VxWorks 的配置。

■ ConfigAll.h

ConfigAll.h 是全局配置头文件，一般不修改此文件，因为对其中项目可以在 config.h 文件中重新定义。该文件包含所有目标系统公用的配置参数默认定义，这些定义都可在 Config.h 中重新定义。该头文件提供了如下的选项和参数定义：

- 内核配置参数
- I/O 系统配置参数
- 网络文件系统（NFS）配置参数
- 可选的软件模块配置
- 可选的设备控制器配置
- 高速存储器方式
- 不同共享内存目标对象的最大数目
- 设备控制器的 I/O 地址、中断向量和中断级别
- 各种地址和常量

此文件还保存着 VxWorks 基本的配置宏，包括所支持的和不支持的 C++软件工具、Kernel 配置、Agent 缺省通信模式配置等。所支持的软件工具宏包括 cache、C++、C 标准库、UNIX、基本异常处理、浮点操作、浮点 I/O。

■ config.h

这是要修改的最关键的基本配置文件，其中的具体配置与实际主板密切相关。该文件中包含着系统引导（即 ROMinit 和 ROMstart 阶段）时所需要的一些最基本配置，还包含对 ads8260.h、 configall.h 等配置宏的定义或去除定义。

针对与目标板相关的头文件，用户可根据目标系统的特殊要求，编辑或修改该文件的配置参数。该头文件提供了如下的参数定义：

- 从 ROM 引导的默认的引导参数字符行
- 系统时钟和校验出错的中断向量
- 设备控制器的 I/O 地址、中断向量和中断级别
- 共享内存网络参数
- 各种地址和常量

针对硬件配置参数头文件，我们可根据目标板的具体情况配置宏定义。值得注意的是 ROM_TEXT_ADRS、ROM_SIZE、RAM_LOW_ADRS、RAM_HIGH_ADRS 要与 Makefile 文件里定义的一致，LOCAL_MEM_LOCAL_ADRS 和 LOCAL_MEM_SIZE 要保证大小合适。

另外还要修改引导行。例如，

```
#define DEFAULT_BOOT_LINE \
"motfcc(0,0)mars:/tor20ppcfcs/target/config/ads8260/vxWorks     h=10.1.1.1     e=10.1.1.51
u=vxworks pw=vxworks"
```

在 Config.h 中需要包含 ConfigAll.h 文件、ads8260.h 文件。在 Config.h 文件中还需要对缓存和内存管理单元进行配置，缓存和内存管理单元的缺省配置和体系结构有关。如果缓存模式不是用户可修改的选项，那么这些宏定义要在 ads8260.h 中定义。

在 Config.h 文件中利用如下 4 个参数定义共享内存的网络驱动器。

- SM_ANCHOR_ADRS 表示定位地址，它包含一个实际的共享内存 Pool 指针。如果板上的内存是双端口 RAM，通常分配的内存地址低位为+0x600；如果另外用一块内存板，则表示内存板的基地址。
- SM_MEM_ADRS 表示共享内存的位置。如果是 NONE，则表示内存是通过调用 malloc()函数从板上的主双端口 RAM 中动态分配的。
- SM_MEM_SIZE 表示共享内存区的大小。
- SM_INT_TYPE 表示与其他 CPU 板的通信方法。

在 Config.h 文件中对主板内存大小和地址的定义主要有以下几个宏。

- LOCAL_MEM_LOCAL_ADRS 表示板上内存区的起始地址。
- LOCAL_MEM_SIZE 表示静态内存大小。
- LOCAL_MEM_AUTOSIZE 表示动态内存的大小。
- USER_RESERVED_MEM 表示保留给用户的内存大小。
- RAM_HIGH_ADRS 表示板上的内存区的高端地址。通常是板上的内存起始地址加 0x00100000，这里的定义必须和在 *Makefile* 文件中的定义一致。

在 Config.h 文件中对主板 ROM 大小和地址的定义主要有以下几个宏。

- ROM_BASE_SIZE 表示 16 进制的 ROM 起始地址。
- ROM_TEXT_ADRS 表示 boot ROM 的入口地址。对于大多数主板，它设置为 ROM 地址区的起始地址。有些主板在开始地址处存放中断向量表，程序入口地址在其后，这部分定义也要和 Makefile 文件中的一致。
- ROM_WARM_ADRS 表示热启动时 ROM 入口地址。通常以 ROM_TEXT_ADRS 的偏移量定义。
- ROM_SIZE 表示 ROM 的大小。

此外，还要在 Config.h 文件中对非易失 RAM 参数、时间标签驱动器、外部总线地址映射网络设备和中断向量等进行定义。

■ Ads8260.h

Ads8260.h 是参数配置头文件，根据具体目标板设置系统资源如端口地址、中断号、串行接口、时钟以及 I/O 设备等。

在该文件中必须包含以下内容：

- 中断向量/中断级
- I/O 设备地址，由硬件决定的且不能由用户改变的 I/O 设备地址
- 设备寄存器各位的定义
- 系统时钟辅助时钟参数的定义，确定新的时钟是否有效
- \target\h\drv\mem 目录下的 memDev.h 文件
- \target\h\drv\intrCtl 目录下的 ppc8260Intr.h 文件

（2）配置模块文件

■ UserConfig.c

模块文件 UserConfig.c 位于目录\target\config\all\下。用户可在此定义系统配置模块源程序，包含根任务、基本的系统初始化、网络初始化、时钟中断程序等。这个文件包含了 VxWorks 映像的主要初始化程序。它还包括了 target、src、config、usrExtra.c 文件，这些文件提供了子系统的初始化和配置。其中主要函数如下：

- 函数 usrInit()，是用户定义的系统初始化程序，它是在系统 boot 后最初执行的 C 代码，由汇编程序在中断被屏蔽的情况下由 sysInit()调用。此时，kernel 不处于多任务状态。然后调用 sysHwInit()初始化硬件，设置 interrupt/exception 向量，启动 kernel multitasking（将 usrRoot()作为第一个 root task）。
- 函数 usrRoot()，是在多任务 kernel 下执行的第一个任务，其功能为执行所有最终的初始化，然后启动另外的任务。它初始化 I/O 系统，安装驱动程序，初始化设备，设置网络连接等，对于特殊配置还要求产生和加载 system symbol table（系统符号表），然后加载和产生其他需要的任务，在缺省配置下它仅初始化 VxWorks 系统的 shell 任务。
- 函数 usrClock()，是用户定义的系统时钟中断程序，其功能为在每个时钟中断时按中断优先级调用。usrRoot()函数通过调用 sysClkConnect()函数后实现安装。此函数还将调用其他需要的时钟 tick 包。如果应用程序需要任何发生在系统时钟中断级的事务，均需将其加入此函数中。
- 函数 usrDemo()，是一个没有 shell 的应用例子。此程序在 usrRoot()函数末尾产生一个任务。如果定义了 INCLUDE_DEMO,而且 INCLUDE_SHELL 没有定义在 configAll.h 或 config.h 文件中，这个例子显示了无 shell 的应用程序如何链接、加载和 ROM 化。

■ BootConfig.c

模块文件 BootConfig.c 是系统初始化文件，位于\target\src\config 目录。这个文件是所有 Boot ROM 映像的主要初始化和控制程序。实际上，它是 usrConfig.c 的一个子集。它包括完全的 Boot ROM 的 shell 任务和一个网络设备初始化表 NETIF。此模块文件主要完成两方面功能，一是实现通过网络(RSH 或 FTP)下载目标文件；二是实现用简单的字符命令集来修改或显示内存内容等功能。它是 ROM 引导 VxWorks 系统的配置模块，一般不需要用户修改。

■ BootInit.c

模块文件 BootInit.c 也是系统初始化文件，位于\target\src\config 目录。此模块的功能是根据引导方式，将 ROM 中的程序拷贝到 RAM 中，如果是从 ROM 引导，则完成系统所需工作区的初始化。这是在执行了 romInit.o 文件后第二阶段的 boot ROM 初始化程序，它主要是 RomStart()函数。它必须执行相应的解压缩程序实现映像的解压和 ROM 映像的重定位。其执行的步骤如下：

- 首先，拷贝 ROM 中的程序段和数据段到 RAM 中。
- 然后清除其他没用的 RAM 部分。
- 最后执行对 ROM 映像中压缩部分的解压缩。

■ SysLib.c

SysLib.c 是 BSP 初始化的核心代码。在这个文件中必须复位所有的硬件，使其处于初始化状态，以保证在后面中断使能之后，不会产生没有初始化的中断。主要定义的函数如下：

- sysHwInit()函数，包括设定 SPLL 的要求值， 设置 BRGCLK 的分频因子，设置周期定时器的值，复位端口 A、B、C 和 D，初始化中断，复位串行通道，设定电源管理模式等。
- SysPhyMemTop()函数 ，返回物理内存的 top 地址。
- SysMemTop()函数，返回 VxWorks 操作系统没有占用的内存中的第一个 Bytes 指针。
- SysToMonitor()函数，传送控制信息给 ROM Monitor。
- SysHwInit2()函数，完成其他的系统配置和初始化。
- SysProcNumGet()函数，返回 CPU 处理器号。
- SysProcNumSet()函数，设定 CPU 处理器号。
- SysLocalToBusAdrs()函数，转换 Local 地址到总线地址。
- SysBusToLocalAdrs()函数，转换总线地址到 Local 地址。
- SysFccEnetDisable()函数，禁用以太网控制器。
- SysFccEnetIntDisable()函数，禁用以太网接口中断。
- SysFccEnetEnable()函数，启用以太网控制器。
- SysFccEnetAddrGet()函数，返回硬件的以太网地址。
- SysFccEnetCommand()函数，发送一个命令到以太网接口
- SysCpmEnetIntEnable()函数，启用以太网接口中断。
- SysCpmEnetIntClear()函数，清除以太网接口中断。

该文件几乎包含了所有与系统相关的库函数,提供特定目标系统的可编程芯片驱动程序，如 I^2C 总线驱动、网卡驱动等。该文件提供了内存映像数据结构，用户可以编辑该结构完成特殊设备的安装。

该文件提供了一个板级接口，基于此接口 VxWorks 系统的应用程序可以独立于硬件的方式建立。文件功能大致包括：

- 初始化功能
- 存储器/地址空间映射功能
- 总线中断功能
- 时钟、计数器功能
- 信箱、定位监听功能

位于目录 config\all 下的配置模块 usrConfig.c 和 bootconfig.c 会调用本文件中的部分函数，另外设备驱动程序将使用某些存储器映射的子程序和总线函数。

- romInit.s

此文件是用汇编语言编写的初始化代码源程序，是 ROM 引导 VxWorks 和基于 ROM 版本的 VxWorks 的入口，是系统上电后运行的第一个程序。

它主要任务是设置 BOOT_COLD 参数，使系统可以顺利进入函数 romStart()。如果硬件要求尽快完成内存映射或设置特定的寄存器，则要在此函数中完成设置。一般来说，大部分硬件初始化工作是在 sysLib.c 文件的 sysHwInit()中完成的。这个文件包括的库文件有:

- VxWorks.h
- asm.h
- cacheLib.h

- config.h
- regs.h
- sysLib.h

系统初始化时首先以实地址方式运行，在初始化系统寄存器后，从实地址模式切换到保护模式。romInit.s 的主要工作由函数 romInit()完成。在 romInit()中，工作流程为：

- 屏蔽所有中断，设定 CPU。即设置 cpsr 的 I_BIT 和 F_BIT 为 1；设置中断寄存器为关模式；同时设定运行模式为 SVC32 模式。
- 保存启动方式，对于冷启动，如果 CPU 配置的是 HIGH VECTORS，就设置入口地址为 0xFFFF0000，否则设置入口地址为 0x00000000。
- 初始化堆栈指针 pc 和 sp，堆栈指针 sp 指向 STACK_ADRS。当映像驻留 ROM 时，该宏值为_sdata，当映像不驻留 ROM 时该宏值为_romInit，这两个地址经过地址映射后都指向被拷贝映像在 RAM 中的目标地址。
- 初始化 cache，屏蔽 cache。
- 根据具体目标板的需要初始化其他寄存器。
- 设定内存系统，一般需要设置 SDRAM 的片选及其刷新周期、关闭缓存、建立内存控制器等。对 SPARC 体系结构，必须激活内存管理单元（MMU）。
- 设定堆栈指针和其他在开始执行 C 代码程序前必须设置的寄存器，然后计算 romStart()函数的地址，并跳转到该函数，不再返回。
- 指针跳转到 romStart()函数并执行。

值得注意的是，在此文件中，不要进行太多的硬件初始化，大多数初始化要在后面的 sysHwInit()函数中进行。另外，即使在此文件中已经初始化的部分，在后面的程序 sysALib.s 或 sysLib.c 中还必须初始化。

■ sysALib.s

sysALib.s 文件也是汇编语言的程序，它不可由用户调用。sysALib.s 是 RAM 中的 VxWorks 操作系统启动时的入口点，这个程序最后跳转到 usrConfig.c 文件中的 usrInit()函数。它包括下列库文件：

- VxWorks.h
- asm.h
- cacheLib.h
- config.h
- regs.h
- sysLib.h

sysALib.s 文件主要的函数为 sysInit()。主要完成屏蔽外部中断，设定中断前缀，关闭指令和数据 cache，关闭外围串口、以太网等设备，启用 SDRAM 等。

本文件用汇编语言编写的与系统相关的模块包括端口读写（字节、字、长字的读写）、CPU 检测以及系统描述符的访问等。此文件与 romInit.s 文件实现的功能基本相似。

■ sysSerial.c

sysSerial.c 文件包含目标板上串行口的初始化程序，它主要是对 COM1、COM2 设备的初始化。串口 I/O 驱动程序在 src/drv/sio 目录下，ttyDrv 库使用此串口 I/O 驱动程序为 VxWorks

提供终端操作。

此文件包括一些串口的参数定义、串口的基本初始化函数、串口中断初始化函数。

此文件不直接链接，而是受 sysLib.c 文件中函数的调用。这样安排的好处在于可以将串口的初始化作为一个单独的驱动程序对待。

2. Makefile

Makefile 文件用于为特定目标系统生成 ROM 引导的 VxWorks 映像，该文件定义了系统的编译规则，生成引导映像的文件格式。在 VxWorks 系统中开发 BSP 时不必自行编写 Makefile，只需在现成的 Makefile 上做一定的定制，主要工作集中在对宏定义的修改。在 Makefile 文件里使用了将近 135 个宏，最简单的 Makefile 文件要包含以下的宏：

- CPU　目标板的 CPU 体系结构。
- TOOL　主机工具链，主机的 make 工具，例如可以为 GNU。
- TARGET_DIR　target 路径，默认为$(WIND_BASE)/target。
- VENDOR　目标厂商的名字。
- BOARD　目标主板名。
- USR_ENTRY　用户入口程序。
- ROM_TEXT_ADRS　16 进制的 Boot ROM 入口地址，与 config.h 文件定义相同。可以是为 ROM 的起始地址，也可以是中断向量表后的偏移地址。
- ROM_SIZE　16 进制的 ROM 大小。
- RAM_LOW_ADRS　加载 VxWorks 的目标地址。
- RAM_HIGH_ADRS　拷贝 Boot ROM 映像到 RAM 的目标地址。
- HEX_FLAGS：特殊结构的标记，用于产生 S-record 文件。
- ·MACH_EXTRA：扩展文件，在此用户可以加入自己的目标模块。

除此以外，Makefile 文件还需要包括以下文件：

- ·$(TGT_DIR)/h/make/defs.bsp：Vxworks 系统运行的标准变量定义。
- ·$(TGT_DIR)/h/make/make.$(CPU)$(TOOL)：提供了一系列的目标主板结构和一套编译工具，如 make. ARM7TDMI_Tgnu。
- ·$(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)：提供了与主机系统有关的定义。
- ·rules.bsp：在创建目标文件时所需要的规则。
- ·rules. $(WIND_HOST_TYPE)：指出创建目标文件时所需的从属文件表。

在编辑修改 BSP 文件时，如果不使用 all 目录下的文件而是拷贝到新目录下修改，那么需要定义与这些文件有关的宏，定义格式为：

CONFIG_ALL=$WIND_BASE/TARGET/CONFIG/newDir

上述显性定义也可以在命令行直接改变。此外，如果是将 bspname 目录下文件拷贝到新目录下修改的，则必须通过命令行修改环境变量 TGT_DIR 的值。

3.3.3　生成目标文件 bootrom 和 VxWorks 映像

bootrom 文件是从 ROM 引导 VxWorks 的引导目标模块，当建立宿主机/目标机环境时，由宿主机通过串行线或网线将 VxWorks 映像加载到目标机。该文件用于初始化目标板上包括引导硬件在内的硬件，从 ROM 上加载 VxWorks 操作系统映像，将 CPU 的控制权移交给操作系统。

VxWorks 映像可分为两类：可下载映像和可引导映像。

- 可下载映像(Loadable Image)，它实际包括两部分，一是 VxWorks，二是 boot ROM，两部分是独立创建的。其中 boot ROM 包括被压缩的 boot ROM 映像(bootrom)、非压缩的 boot ROM 映像(bootrom_uncmp)和驻留 ROM 的 boot ROM 映像(bootrom_res)3 种类型。
- 可引导映像(Bootable Image)：是将引导程序和 VxWorks 融为一体的映像，它一般是最终产品，包括不驻留 ROM 的映像和驻留 ROM 的映像两种类型。

bootrom 和 VxWorks 映像可以利用 Tornado 提供的工具，按照配置文件 config.h 有关设置自动生成，配置文件 config.h 主要定义了引导行、目标板操作系统要包含的主要模块，内存地址等参数。其中引导行（boot line）定义了引导设备、引导路径、操作系统文件名、主机/目标板 IP 地址、子网掩码、FTP 用户名和口令等参数。

在生成 bootrom 时，可在 Tornado 的集成环境下 Build 菜单中选择 Build Boot ROM 来创建指定类型的 Boot ROM，例如，如可以选择 bootrom_uncmp。

在生成 VxWorks 映像时，可在 Tornado 的集成环境下 Build 菜单中选择 standard BSP Builds 来生成 VxWorks 映像。

此外，也可以在命令行环境下利用 Makefile 创建各种映像类型。

3.3.4　基于 ROM 映像的初始化

基于 ROM 的 VxWorks 映像的初始化实际上是对系统文件中函数的调用流程。系统从 rominit()转入，直到运行 usrRoot()其才将控制权转交给应用程序。

基于 ROM 的 VxWorks 映像的初始化过程可以分成如下几步：

（1）romInit()，它在文件 romInit.s 中。

- 禁止中断
- 保存启动类型（冷/热启动）
- 硬件相关的初始化
- 转到函数 romstart()

（2）romstart()，它在文件 bootInit.c 中。

- 从 ROM 中将数据段拷贝到 RAM，将内存清 0
- 从 ROM 中将代码段拷贝到 RAM，如果是压缩的方式则解压缩
- 根据启动类型调用函数 usrInit()

（3）usrInit()，它在文件 usrConfig.c 中。

- 将 bss 未初始化的段清 0
- 将引导类型保存到 sysStartType
- 调用异常向量初始化函数 excVecInit()，初始化所有系统和默认的中断向量
- 调用系统的硬件设备初始化函数 sysHwInit()
- 调用内核库初始化函数 usrKernelInit()
- 调用内核初始化函数 kernelInit()

（4）usrKernelInit()，它在文件 usrKernel.c 中。

- classLibInit()

- taskLibInit()
- taskHookInit()
- semBLibInit()
- semMLibInit()
- semCLibInit()
- semOLibInit()
- wdLibInit()
- msgQLibInit()
- qInit()
- workQInit()

（5）kernelInit()，初始化多任务环境并启动内核，它在文件 kernelLib.c 中。

- 调用 intLockLevelSet()
- 在内存池高端创建根任务和任务控制块 TCB
- 调用任务初始化函数 taskInit()
- 调用任务激活函数 taskActivate()
- 转到函数 usrRoot()

（6）usrRoot()，它在文件 usrConfig.c 中。

初始化 I/O 系统，安装设备的驱动，根据配置文件 ConfigAll.h 和 Config.h 中的配置创建指定的设备。

（7）转入用户创建的应用程序的入口。

3.4 BSP 中设备驱动程序的开发

为了实际系统开发的方便，许多应用场合将硬件模块（或者称作主板）的设备驱动程序也封装在 BSP 中，因此在开发 BSP 的同时，还要完成设备驱动程序的开发。

在实际系统设计过程中，I/O 系统的设计目标是提供一个简单、统一、与硬件无关的接口。VxWorks 系统提供标准的 C 库支持基本 I/O 和缓冲 I/O。

在 VxWorks 应用系统中，应用软件通过文件操作的方式进入 I/O 设备。这里所说的文件包含两种方式，一是原始设备文件如串口和任务间通信的管道；二是逻辑文件如文件系统。

VxWorks 系统的设备管理使用与文件处理完全相同的、标准的系统调用方式。用户可以像操作文件一样打开、关闭和读写所有的硬件控制器。这样，VxWorks 系统的设备管理向操作系统和高层应用提供了对硬件设备的灵活的管理接口。图 3-7 示出了 VxWorks 系统中的 I/O 系统框架。从图 3-7 可以看出，要开发设备的管理接口必须至少定义如下两个接口函数：

- xxRead( )
- xxWrite( )

VxWorks 系统的设备管理可以涵盖网络通信设备管理、专用芯片的管理、特定外设的管理等。设备管理需要完成 I/O 定义、缓冲管理、设备资源分配和设备的特定处理。作为开发人员，设备管理的核心任务就是完成设备驱动程序的开发。设备驱动程序要完成的功能主要

包括：

- 设备资源的分配。
- I/O 接口的定义。
- 缓冲管理。
- 设备状态的监视。
- 设备的中断处理。

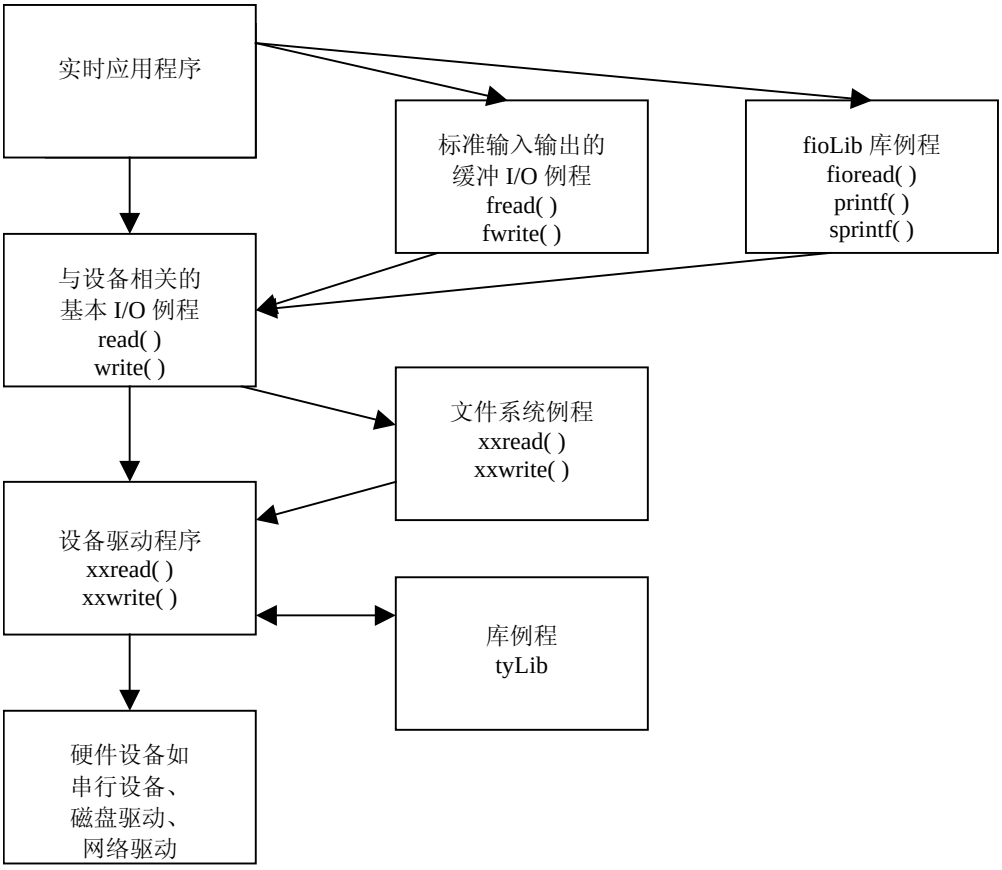

图 3-7　VxWorks 系统的 I/O 系统框架

第 4 章　VxWorks 系统 BSP 开发实例

本章所介绍的 BSP 开发实例基于某数据通信系统中的主控单板。此主控单板以 Motorola 系列处理器 MPC8260 为中央处理单元，外挂 10/100Mbit/s 以太网口、RS232 端口、I^2C 总线，实现较为复杂的控制转发和数据通信的功能。本章首先简单介绍 MPC8260 处理器以及主控单板的硬件电路配置，然后详细描述此主板 BSP 设计的细节，并对此主板主要的设备驱动程序做扼要介绍，接下来给出此板 BSP 的示例代码，最后探讨 BSP 的测试方法。

4.1　MPC8260 处理器的组成与结构

MPC8260 处理器是由 Motorola 提供的 PowerPC 系列处理器。它由两个内核即 PowerPC 603e 和通信处理模块 CPM 专用内核组成。从资源使用的角度，系统设计人员主要与 CPM 打交道，因此在介绍 MPC8260 处理器时，我们将重点放在 CPM 模块上。

4.1.1　基本功能模块

MPC8260 PowerQUICC II 是集成的 PowerPC 微处理器，它是目前较先进的集成通信微处理器。高速的嵌入式 PowerPC 内核，连同集成的网络和通信外围设备，为用户提供了一个建立高端通信系统的全新系统解决方案。MPC8260 PowerQUICC II 可以称作是 MPC860 PowerQUICC 的下一代产品，它在各方面都有较高的性能，包括更大的灵活性、扩展能力和更高的集成度。

与 MPC860 相似，MPC8260 也有 3 个主要的组成部分：嵌入式 PowerPC 内核、系统接口单元 SIU 和通信处理模块 CPM。SIU 的主要功能包括 PowerPC 到本地总线的桥接、存储控制器、总线接口（提供 60x 总线到 CPM 的接口）、L2 /Cache 接口（到 L2Cache 的简单接口）、实时时钟（每秒钟提供一个中断），以及系统功能如配置、保护、复位、时钟同步、电源管理等。

它的 CPM 是高性能的通信处理器（CP）模块，其中包括一个 32 位的 RISC 微控制器（可认为是除内核以外的另一个 CPU），分担了底层的通信处理，使 PowerPC 核可以主要进行高层的操作。这种双处理器体系结构功耗要低于传统体系结构的处理器。2 个 CPU 之间通过内部存储空间进行联系。

CPM 同时支持 3 个快速的串行通信控制器（FCC）、2 个多通道控制器（MCC）、4 个串行通信控制器（SCC）、2 个串行管理控制器（SMC）、1 个串行外围接口（SPI）和一个 I^2C 接口。

MPC8260 的基本功能模块如图 4-1 所示。

MPC8260 内部数据的流向如图 4-2 所示。

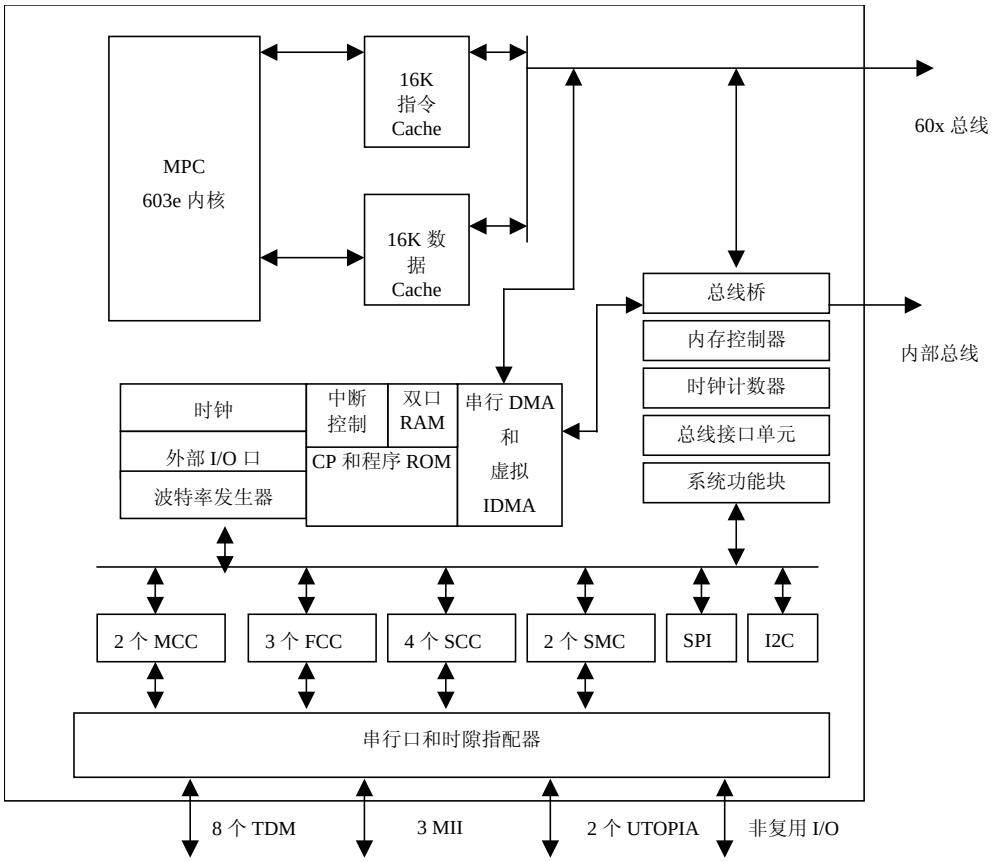

图 4-1　MPC8260 的基本功能模块

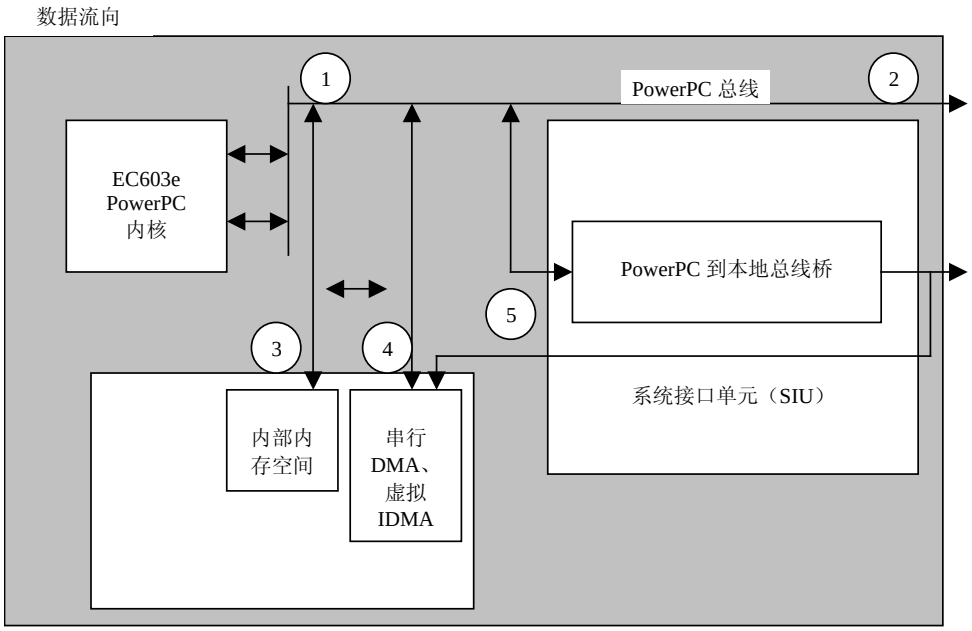

图 4-2　MPC8260 内部数据流向

下面对 MPC8260 的各部分分别进行介绍。

4.1.2　内核 603e 的组成

603e 的内部模块组成如图 4-3 所示。

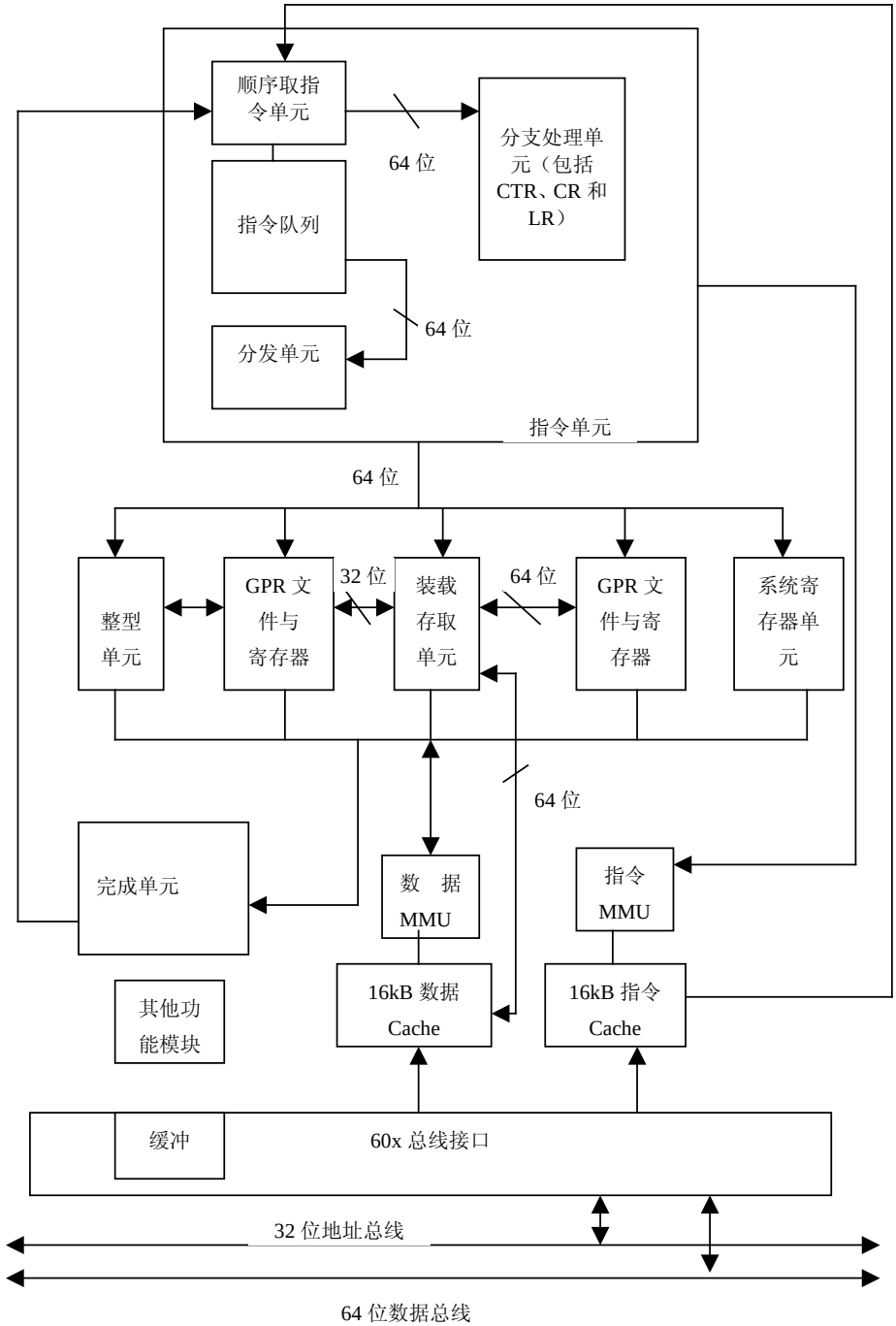

图 4-3　内核 603e 的组成模块示意图

PowerPC-603ePowerPC 处理器包括 16kB 的命令和数据缓存以及命令和数据 MMU。603e 在 100MHz 时可以达到 140MIPS（兆指令每秒），在 200MHz 时可以达到 280MIPS。其主要特点有：

- EC603e 微处理器（嵌入式 PowerPC 内核）运行频率为 100~200 MHz。
- 140.0 MIPS @ 100 MHz（Dhrystone 2.1）。
- 280.0 MIPS @ 200 MHz（Dhrystone 2.1）。
- 高性能超标量体系结构微处理器。
- CPU 非活动模式。
- 支持摩托罗拉的外部 L2 缓存芯片（MPC2605）。
- 改良的低功耗内核。
- 16kB 数据和 16kB 指令缓存。
- 存储管理单元。
- 无浮点单元。
- 通用片内处理器（COP）。

内核 603e 各部分功能如下：

- Sequential Fetcher——从指令 Cache 中取指令到指令队列。
- 分支处理单元——从 fetcher 中提取分支指令并在未解决的条件分支上使用静态分支预测，以便在估计一个条件分支时，指令单元可以从一个被预测的目标指令流取指令。
- 指令队列——保留最多 6 条指令，并在一个单个周期内从指令单元加载最多 2 条指令。
- Dispatch——以每周期两个指令的最大速度分配指令到它们各自的执行单元。
- 整数单元——执行所有整数指令。
- 加载/存储单元——执行所有加载存储指令并在 GPR 和 Cache/内存子系统间提供数据传输接口。
- 系统寄存器单元——执行各类系统级指令，包括条件寄存器的逻辑操作和移入移出专用目的寄存器指令。
- 完成单元——从分派开始跟踪指令。

4.1.3　SIU 的结构

SIU 的内部模块组成如图 4-4 所示。

SIU 各部分功能如下：

- 60x bus-to-Local Bridge——允许 603e 在本地总线上访问。
- 内存控制器——支持 12 个 memory bank.（内存槽）。
- 总线接口单元——提供 60x 总线到 CPM 的接口。
- L2 Cache 接口——提供到 L2Cache 的简单接口。
- 实时时钟——每秒钟提供一个中断。

提供的系统功能：

- 配置——配置多个系统管脚。
- 保护——硬件和软件狗。

- 复位——复位和监视。
- 时钟同步——根据外部时钟振荡器产生内部时钟。
- 电源管理——控制正常电源和低电源模式。
- JTAG——IEEE 1149.1 测试接入端口。

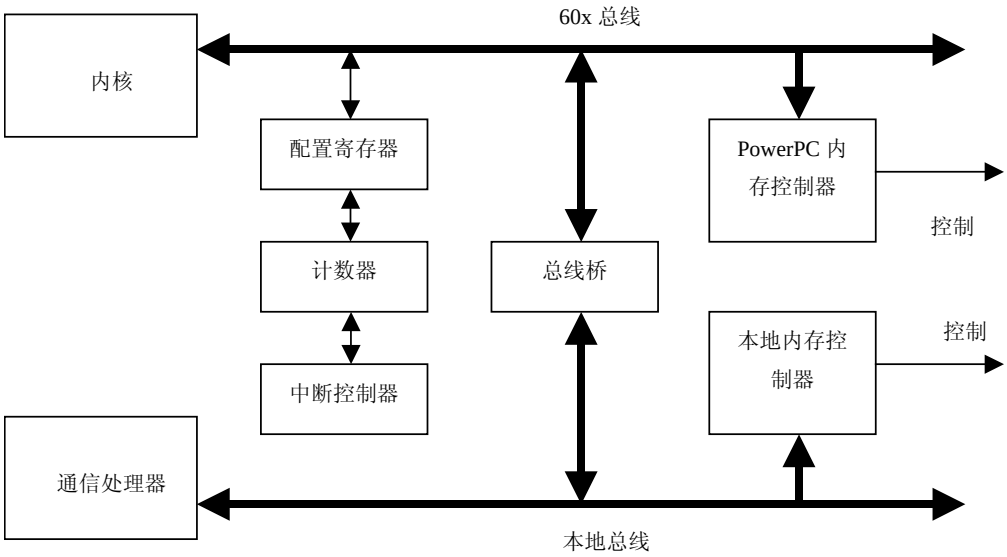

图 4-4　SIU 的模块结构示意图

4.1.4　CPM 的模块结构

CPM 的内部模块组成如图 4-5 所示。

MPC8260 高性能的通信处理模块（CPM）运行频率高达 133 MHz 或 166MHz，包括 MPC8260 中所有的通信组件，提供 3 个 FCC、2 个 MCC、4 个 SCC、2 个 SMC、1 个 SPI 和 1 个 I^2C，其主要特点有：

- PowerPC 和 CPM 可以工作在不同频率。
- 支持串行比特率达 710 Mbit/s @133 MHz。
- 并行 I/O 寄存器。
- 片内 24kB 双口 RAM。
- 两个多通道控制器（MCC），每一个支持 128 条全双工的 64kbit/s HDLC 线。
- 虚拟 DMA 功能。
- 双总线结构：1 个 64 位 PowerPC 和 1 个 32 位本地总线。
- 2 个 UTOPIA 二级主/从端口，均支持多 PHY，其中一个可以为 8/16 位的数据。
- 3 个 MII 接口。
- 8 个 TDM 接口（T1/E1），2 个 TDM 口可以无缝连接到 T3/E3。
- 内部电压 2.0V，I/O 电平为 3.3V。
- 133 MHz 功耗为：2.5 W。

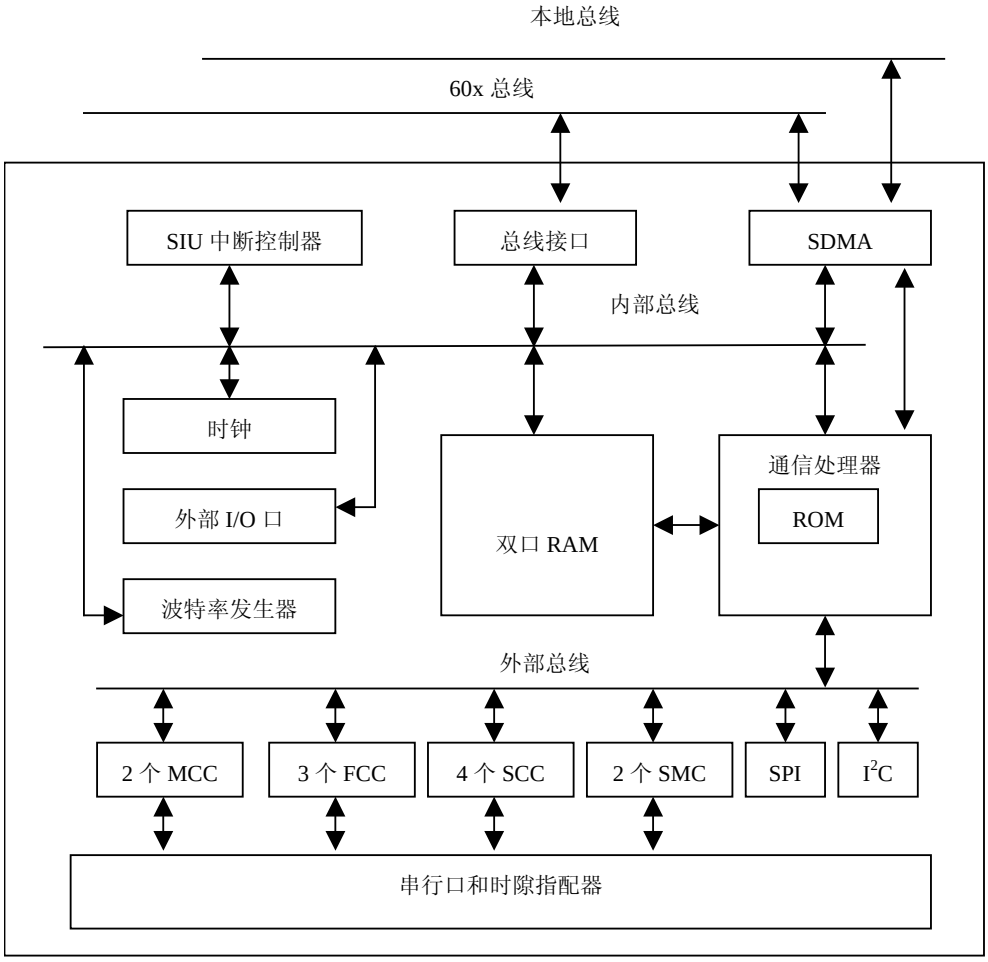

图 4-5　CPM 的模块结构示意图

MPC8260 高性能的通信处理模块 CPM 内部结构各部分功能如下：

- RISC and ROM——处理串行通信设备，虚拟 DMA，定时器等。
- Internal Memory Space——RISC 到 PowerPC 的接口，由寄存器和双口 RAM 组成。
- SDMA——用于在通信设备与 60x 总线、local 总线上的 buffer 间传递数据。
- IDMA——用于在内存—内存，内存—设备，设备—内存间传递数据。
- Interrupt Controller——处理所有的中断资源。
- Multi-channel Communication Controller——每个 MCC 在 128 个逻辑通道上传递数据，外部接口仅通过 TDM 传递数据。
- Fast Communication Controller——快速通信控制器，支持同步高速率协议：HDLC, Ethernet 和 ATM。
- Serial Communication Controller——串行通信控制器。
- Serial Management Controller——串行管理控制器。
- Serial Peripheral Interface——串行外设接口。
- Inter-Integrated Circuit——内部集成电路。

- Serial Interface——连接通信设备的物理层串行接口。
- Time Slot Assigner——多路传输来自任何通信连接（除 SPI and I^2C）的数据。
- Parallel I/O——端口 A、B、C，and D 作为 I/O 端口。
- Baud Rate Generators——内部时钟源提供给 FCC、SCC and SMC。
- Internal Timers——内部计时器。
- General Purpose Timers——通用计时器。

4.2 MPC8260 通信处理模块

MPC8260 内部的 CPM 模块是高性能的通信处理模块，其中包括一个 32 位 RISC 微控制器（可认为是除核以外的另一个 CPU），分担了底层的通信处理，使 PowerPC 核可以主要进行高层的操作处理。这两个 CPU 之间通过内部存储空间进行联系。与通信处理有关的 buffer、寄存器、BD、参数 RAM 等都存储在内部存储空间里，所以先对内部存储空间进行说明。

4.2.1 内部存储空间

CPM 中有一部分内部存储空间，共 128kB，用于存储 8260 专用寄存器、BD、参数 RAM 和数据以及一些保留区。它在 4GB 内存中的起始物理地址是由内部存储映射寄存器（IMMR）定义的，在 reset 期间 IMMR 被导入初始值（用户可在软件中修改这个值）。8260 的寄存器集包含了除 PowerPC 核以外的所有寄存器，提供了对系统的各种控制功能。

8260 具有 24kB 的静态 RAM 被配置成双口 RAM，存储有如下内容：

- BD（buffer descriptors）。
- 与每个 FCC、SCC、MCC、SMC、SPI、I^2C、IDMA 相关的参数。
- data buffers（可选，但一般是放在外部存储空间）。
- 临时存储 FCC 接收或发送到外部存储空间的数据。
- 微码。

双口 RAM 空间分配如下图 4-6 所示。

FCC、SCC、MCC、SMC、SPI、I^2C 用 BD 来控制存储数据的 buffer，发送和接收的数据都存放在一些 buffer 中。buffer 可以在内部存储空间，也可在外部存储器中，但由于存储空间的限制，一般存在外部存储空间。buffer 是通过 BD 进行索引的。BD 可位于双口 RAM 的 0x0000～0x3800 的任何位置，BD 的大小是 8 字节，其总数只受限于双口 RAM 的大小。用户定义每个串行控制器（FCC、SCC 等）使用 BD 的情况，对于用到的每个串行控制器，双口 RAM 中分别有一列 RxBD 和 TxBD，它们的起始地址分别由相应的串行控制器的参数 RAM 中的参数（如 RBASE、TBASE）确定。其起始地址值应是 8 的倍数。BD 表的最后一个 BD 的 W 位应为 1，用户可通过这些设置来确定接收和发送部分使用多少 BD。参数 RAM 中还有一个指向 BD 的指针，从一个 BD 指向下一个 BD，在任何时刻只有指针指向的 BD 是有效的。当指针指向 BD 列的尾部时，它会自动返回到 BD 列的开始。这样，只要知道 RxBD 和 TxBD 所在的位置，就可以由 RxBD 和 TxBD 找到对应的 buffer，其定位关系如图 4-7 所示。

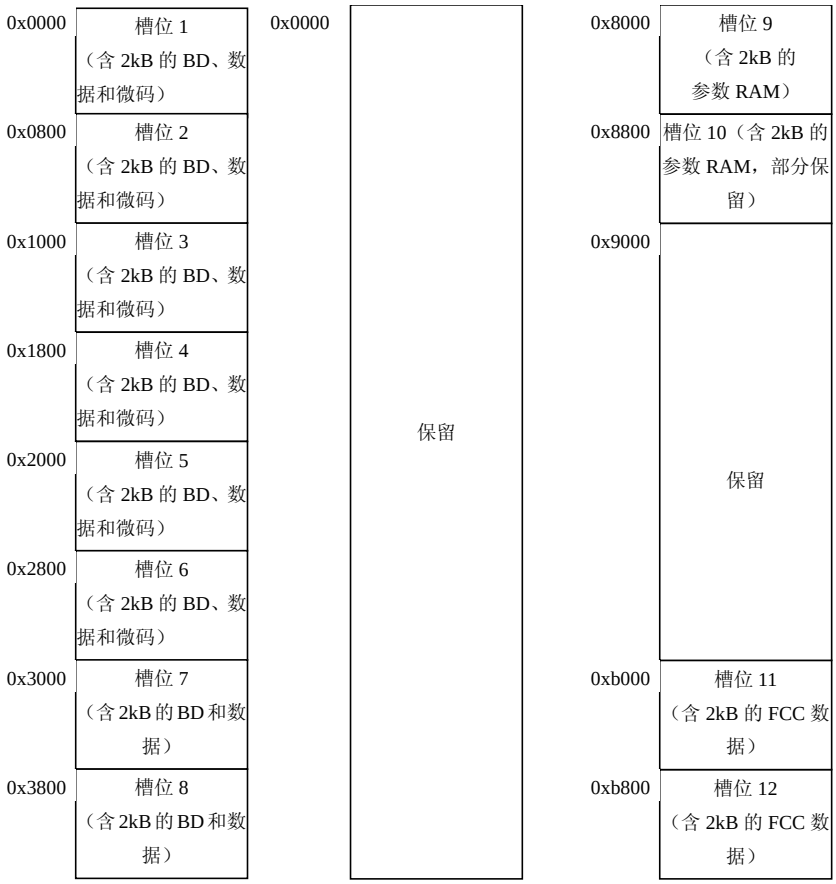

图 4-6　MPC8260 的双口 RAM 存储映射图

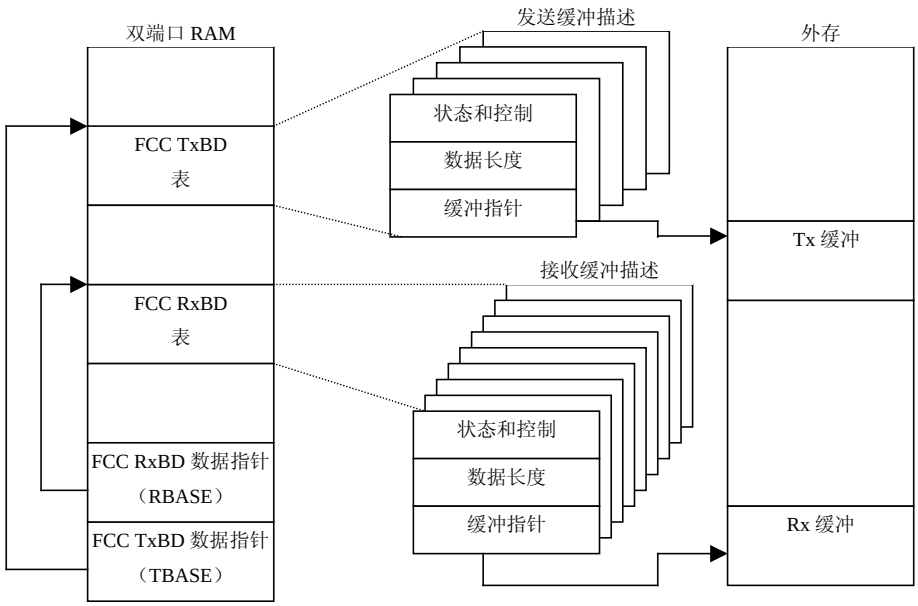

图 4-7　BD 与 buffer 间定位关系示意图

4.2.2 缓冲描述符 BD

由上面的说明可以看出，BD 对通信处理过程至关重要，下面对 BD 进行详细描述。BD 的格式都是一样的，如下表 4-1 所示。

表 4-1 BD 格式描述

地　　址	描　　述
Offset+0	状态和控制字段
Offset+2	数据长度字段
Offset+4	缓冲指针的高 16 位
Offset+6	缓冲指针低 16 位

缓冲描述符 BD 的前 16 位包含状态和控制位，用于控制和报告数据的传输状态。这些位因不同的串行控制器（FCC、SCC 等）以及所采用的不同协议而不同，每当 buffer 被发送和接收后，CPM 都要刷新这些位。

接下来的 16 位指出了发送和接收数据的长度。对于 RxBD 指的是控制器写入 buffer 的字节数，CP 在把接收数据放入相对应的 buffer 并关闭 buffer 后写入这个长度。在基于帧的协议中，这个长度指的是整个帧的长度（包括 CRC 字节），如果接收帧的长度（包括 CRC 字节）是最大接收 buffer 长度的倍数，那么最后一 BD 包含的数据长度是指整个帧的长度。最大接收 buffer 长度由参数 RAM 中的参数 MRBLR 定义，它应是 32 的倍数，对于以太网应不小于 64。对于 TxBD 指的是控制器从 buffer 中应发送的字节数，CPM 从不修改这个值。

最后的 32 位指向存储器中 buffer 开始的地址，对于 RxBD 此值应是 32 的倍数，对于 TxBD 此值无限制。

下面以 FCC 采用以太网协议为例，具体说明 RxBD 和 TxBD 的内容。FCC 用 RxBD 来报告有关每个 buffer 接收数据的信息，RxBD 的格式如图 4-8 所示。

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
偏移量＋0	E	—	W	I	L	F	—	M	BC	MC	LG	NO	SH	CR	OV	CL
偏移量＋2	数据长度															
偏移量＋4	发送数据缓冲指针															
偏移量＋6																

图 4-8　快速以太网接收 buffer RxBD。

图 4-8 中所描述 RxBD 的详细说明如表 4-2 所示。

表 4-2 RxBD 详细说明

位	名　称	描　述	说　　明
0	E	空	0 与此 BD 相关的 buffer 已满或因错误而停止接收，只要 E＝0，CP 就不用此 BD。 1 与此 BD 相关的 buffer 为空，RxBD 和 buffer 由 CP 控制，内核不能刷新此 BD

续表

位	名　称	描　述	说　　明
1	—	保留	应被清 0
2	W	帧的延伸	0 此 BD 不是 RxBD 表的最后一个。 1 此 BD 是 RxBD 表的最后一个，当此 BD 相关的 buffer 被使用后，CP 使用 BD 表的第一个 BD（由参数 RBASE 指向），此 bit 决定了 RxBD 表中 BD 的数目（以太网协议中 RxBD 表中 BD 的数目应大于 1）
3	I	中断	0　此 BD 相关的 buffer 被使用后不产生中断 1　此 BD 相关的 buffer 被使用后 FCCE[RXB]或 FCCE[RXF]被置位，如果使能这 2 个 bit 可产生中断
4	L	一帧的最后 buffer	当前 buffer 不是该帧的最后一个 buffer 当前 buffer 是该帧的最后一个 buffer，这表明该帧的结尾或接收错误（此情况下 CL、OV、CR、SH、NO 和 LG 中的一个或多个会置位），FCC 将帧长度以字节为单位写入 BD 的数据长度字段
5	F	一帧的第一个 buffer	0 当前 buffer 不是该帧的第一个 buffer 1 当前 buffer 是该帧的第一个 buffer
6	—	保留	应被清 0
7	M	遗漏	当帧是在混杂模式下接收时由 FCC 设置，内部地址匹配部分将其标记为 miss，这样当用混杂模式时，用户用此 M bit 快速确定帧的目的是否是该口，只当 RxBD[I]设置后有效。 0　帧因为地址匹配而接收 1　帧是在混杂模式下接收（地址未匹配）
8	BC	广播地址	只当帧的最后一个 buffer（RxBD[L]＝1）有效，接收帧的地址是广播地址时置位
9	MC	组播地址	只当帧的最后一个 buffer（RxBD[L]＝1）有效，接收帧的地址是组播地址时置位
10	LG	接收帧超长	当一个接收帧长度超过参数 RAM 中 MFLR 值（最大帧长）时置位
11	NO	非整字节数帧	当接收到的帧的 bit 数不能被 8 整除时置位
12	SH	短帧	当一个接收帧长度小于参数 RAM 中 MINFLR 值（最小帧长）时置位（该位只有当 FPSMR[RSH]＝1，即寄存器中设置容许接收短帧时有效）
13	CR	接收 CRC 错误	接收帧的 CRC 有错误时置位
14	OV	超速	接收 buffer 的接收速度不能达到应有速度时置位
15	CL	冲突	在帧接收过程中因为冲突而关闭帧，只当迟冲突发生时置位

FCC 用 TxBD 配置发送、指示错误以让内核知道 buffer 什么时候被使用了，TxBD 的结构如图 4-9 所示。

地址	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
偏移量＋0	R	PAD	W	I	L	TC	DEF	HB	LC	RL	RC				UN	CSL
偏移量＋2	数据长度															
偏移量＋4	发送数据缓冲指针															
偏移量＋6																

图 4-9　快速以太网发送 buffer TxBD 的示意图

图 4-9 中所描述 TxBD 的详细说明如表 4-3 所示。

表 4-3　　TxBD 详细说明

Bit	名　称	描　述	说　　明
0	R	准备好	0　与此 BD 相关的 buffer 未准备好发送，用户可以操作此 BD 及相关的 buffer，当 buffer 被发送或发生错误后 CP 清除该位 1　与此 BD 相关的 buffer 已准备好发送，buffer 或者在等待发送或在发送处理中，只要 R＝1 用户就不能改变该位或相关的 buffer
1	PAD	填充	只当 L＝1 时有效 不为短帧填充 PAD 为短帧填充 PAD 直到发送帧长度等于 MINFLR（最小帧长），PAD 字节存储在由参数 RAM 中的参数 PAD_PTR 指向的 buffer
2	W	绕回	0 此 BD 不是 TxBD 表的最后一个 1　此 BD 是 TxBD 表的最后一个，当此 BD 相关的 bufer 被使用后，CP 使用 BD 表的第一个 BD（由参数 TBASE 指向），此 bit 决定了 TxBD 表中 BD 的数目(以太网协议中 TxBD 表中 BD 的数目应大于 1)
3	I	中断	0　此 BD 相关的 buffer 被使用后不产生中断 1　此 BD 相关的 buffer 被使用后 FCCE[TXB]或 FCCE[TXE]被置位，如果使能这 2 位可产生中断
4	L	一帧的最后 buffer	0　当前 buffer 不是该帧的最后一个 buffer 1　当前 buffer 是该帧的最后一个 buffer
5	TC	发送 CRC	只当（TxBD[L]＝1）有效 在发完最后一个数据字节后立即停止发送 在发完最后一个数据字节后发送 CRC 序列
6	DEF	推迟	帧发送前没有遇到冲突但因为延误而推迟发送
7	HB	Heartbeat	完成发送后的 40 个发送串行时钟内没有发生冲突，只有当 FPSMR[HBC]＝1 时该位才会被设置

续表

Bit	名　称	描　述	说　　明
8	LC	延迟冲突	当 FPSMR[LCW]中定义的字节数(56 或 64)发送完后发生冲突，FCC

			终止发送并在发送完 buffer 后更新 LC
9	RL	重传限制	FCC 发送完 buffer 后更新 RL
10－13	RC	重传计数	记录该帧重传的次数，发送完 buffer 后更新 RC
14	UN	下溢	FCC 发送相应的 buffer 时遇到下溢时置位，发送完 buffer 后更新 UN
15	CSL	载波侦听丢失	在帧发送中载波侦听丢失则置位，发送完 buffer 后更新 CSL

4.2.3　参数 RAM

每个与 FCC、SCC 等相关的参数 RAM 在双口 RAM 中的地址都是固定的，如图 4-10 所示。

页面	相对于 RAM 基址的偏移地址	外设接口	字节大小
1	0x8000	SCC1	256
2	0x8100	SCC2	256
3	0x8200	SCC3	256
4	0x8300	SCC4	256
5	0x8400	FCC1	256
6	0x8500	FCC2	256
7	0x8600	FCC3	256
8	0x8700	MCC1	128
	0x8780	保留	124
	0x87FC	SMC1 基址	2
	0x87FE	IDMA1 基址	2
9	0x8800	MCC2	128
	0x8880	保留	124
	0x88FC	SMC2 基址	2
	0x88FE	IDMA2 基址	2
10	0x8900	保留	252
	0x89FC	SPI 基址	2
	0x89FE	IDMA3 基址	2
11	0x8A00	保留	224
	0x8AE0	RISC 时钟	16
	0x8AF0	接收数目	2
	0x8AF2	保留	2
	0x8AF4	保留	4
	0x8AF8	RAND	4
	0x8AFC	I2C 基址	2
	0x8AFE	IDMA4 基址	2
12-16	0x8B00	保留	1280

图 4-10　参数 RAM 在双口 RAM 中的地址分配

参数 RAM 包含了 FCC、SCC 等与通信处理有关的参数。有些参数 RAM 的值是在初始化时设置好的，以后一般不再改动，而有些参数 RAM 的值是由 CP 写入的。参数 RAM 可分为两部分：

一部分是与 FCC、SCC 等所采用的通信协议无关的通用参数，如 RxBD 和 TxBD 表的基地址、

接收 buffer 的最大长度、指向 RxBD 和 TxBD 的指针等。值得注意的是，发送 buffer 的最大长度是可变的，由 TxBD[data length]确定。

另一部分是与 FCC、SCC 等所采用的通信协议（如采用以太网协议）有关的专用参数，如 CAM 地址、最大、最小帧长度、单播地址、组播地址、计数器的计数值等。

4.2.4 快速以太网控制器的功能

8260CPM 中的 FCC（快速通信控制器）可通过 8260 内部寄存器独立配置来实现不同的协议如 HDLC、ATM、ETHERNET、透明传输等，同时 FCC 也有相应的物理接口如快速以太网接口 MII、ATM 接口 UTOPIA 等，当 FCC 被编程为某一特定协议或模式后，它就执行相应协议的数据链路层工作。这些设置都是在 FCC 的有关寄存器和参数 RAM 中完成的，为此先介绍 FCC 的寄存器和参数 RAM。

每个 FCC 都有如下的寄存器和参数 RAM：

- 通用模式寄存器 GFMR，与使用的协议无关，可设置其中的 MODE 位来选择需要的协议，例如当某 FCC 的 GFMRx[MODE]设置为 0x1100，则该 FCC 就执行快速以太网控制器的功能。
- 与协议相关的寄存器 FPSMR。
- 数据同步寄存器 FDSR，用于接收数据的同步，快速以太网应设置为 0xD555。
- FCC 命令发送寄存器 FTODR，可使 CP 处理新的帧而不必等待通常的 256 个串行发送时钟查询时间。
- 中断事件寄存器 FCCE。
- 中断屏蔽寄存器 FCCM，控制是否启用 FCCE 中相应的中断。
- 状态寄存器 FCC，显示 RXD 线上的实时状态。
- 参数 RAM，分成对所有协议相同部分和针对不同协议部分。

对 FCC 的初始化可按如下过程进行：

（1）写并行 I/O 口，配置、连接 I/O 引脚到 FCC。

（2）配置并行 I/O 口寄存器使能 CTS、CD 信号。

（3）配置相应的时钟到 FCC。

（4）配置 CMX 使 FCC 工作在 NMSI 模式。

（5）配置通用寄存器 GFMR（ENT、ENR 二位除外）。

（6）配置与协议相关的寄存器 FPSMR。

（7）配置数据同步寄存器 FDSR。

（8）初始化 FCC 参数 RAM 中的值。

（9）初始化 RxBD 和 TxBD。

（10）根据需要清除中断事件寄存器 FCCE 中的所有当前事件。

（11）配置中断屏蔽寄存器 FCCM 使能 FCCE 中的中断。

（12）写 CPM 中断优先级寄存器 SCPRR_H 配置 FCC 的中断优先级。

（13）清除 SIU 中断 pending 寄存器 SIPNR_L 中任何当前中断。

（14）写 SIU 中断屏蔽寄存器 SIMR_L，使能中断到 CP 中断控制器。

（15）通过 CP 命令寄存器 CPCR 初始化发送和接收参数。

（16）清除 buffer。

（17）配置通用寄存器 GFMR 中的 ENT、ENR 两位（分别是发送使能和接收使能）。

接收和发送的以太网 MAC 帧都存储在 Buffer 中，一般在外部存储空间。接收 Buffer 和发送 Buffer 中存储的 MAC 帧内容如图 4-11 所示。

图 4-11　MPC8260 存放的以太网帧格式

接收 Buffer 的大小由参数 RAM 中的参数 MRBLR 确定，它可以动态修改。发送 Buffer 的大小由 TxBD[Data Length]确定。

只要在 FCC 口外加一个物理端口器件，就可完成快速以太网发送和接收功能，FCC 与外部物理端口器件之间的连接关系如图 4-12 所示。

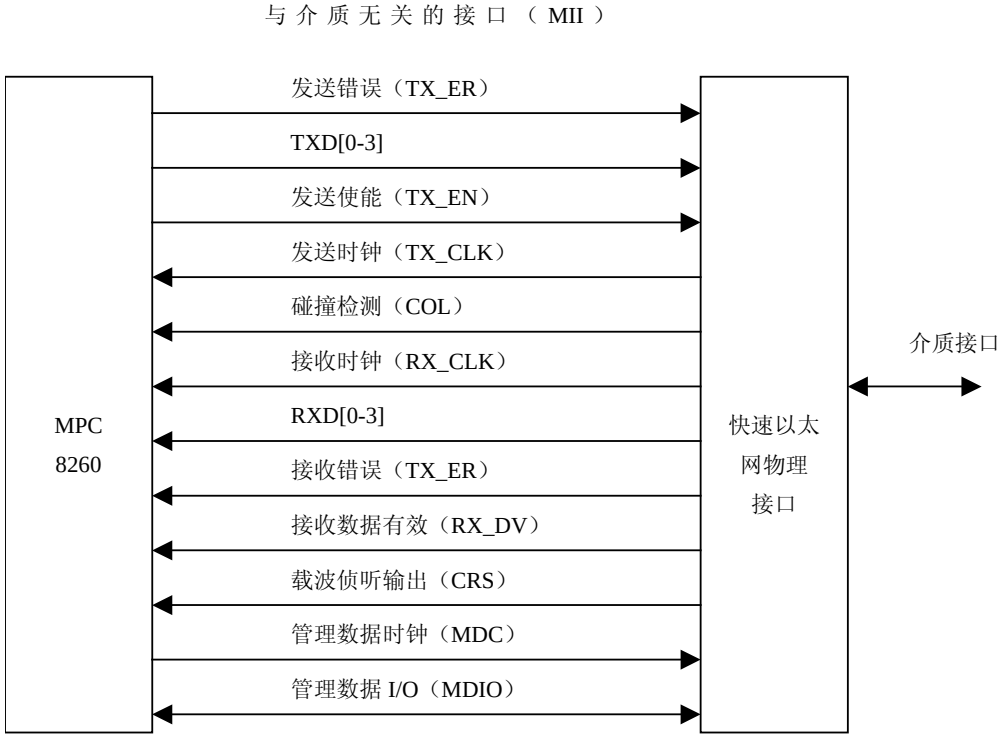

图 4-12　MPC8260 快速以太网控制器与外部物理端口器件 PHY 的连接关系

4.2.5　快速以太网控制器的接收过程

MPC8260 的快速以太网控制器主要通过 TxBD 和 RxBD 来控制发送、接收 MAC 帧，几乎不

需要内核的干预。

当内核启动接收后，FCC 进入 HUNT 模式，数据每次 4 个 bit 移入接收移位寄存器，移位寄存器中的内容与 FCC 的数据同步寄存器 FDSR 中的 SYN2 域进行比较，当比较匹配时，HUNT 模式结束，表明数据（指除去前导、帧起始定界符的数据）的开始。接收部分首先进行地址过滤，只接收指定 MAC 地址的帧。地址过滤后，当接收 FIFO 满时，FCC 发请求。CPM RISC 响应请求，同时 CP 查询下一个 RxBD 的[E]位，如果该位为 1，通过 SDMA 将数据从当前接收 FIFO 传送到当前 RxBD 指向的接收 buffer。接收过程中 FCC 检测帧长度，当帧结束时，检查 CRC 值并将其写入 buffer，将数据长度写入 BD。当接收帧完成后，将 RxBD[L]置位，将其他状态位写入 RxBD，清除 RxBD[E]，表明对应的 buffer 非空，这样直到该 BD 又由内核准备好后才会被使用。如果接收数据较长，需占用多个 buffer，则每填满一个 buffer 就清除 RxBD[E]。

如果 RxBD[I]＝1，那么与此 BD 相关的 buffer 被占用后，FCCE[RXB]或 FCCE[RXF]将置位，这两个位置位后的结果就是向内核发起中断。实际上，内核就是通过中断来处理每个帧和每个 buffer 的。硬件工作结束后，就该软件对数据进行处理了。

当 CP 检测到 RxBD 的[W]＝1，则在此 BD 相关的 buffer 被使用后，CP 使用 RxBD 表的第一个 RxBD。

用 RxBD 实现接收的过程如图 4-13 所示。

4.2.6 快速以太网控制器的发送过程

与接收 MAC 帧的过程类似，当上层应用有 MAC 帧要发送时，8260 内核启动发送控制器，当发送 FIFO 空时，FCC 发请求。CPM RISC 响应请求，同时 CP 每 256 个串行时钟查询 FCC 的 TxBD 表的第一个 TxBD 的[R]位，如果该位被置为 1，表明当前 TxBD 指向的发送 buffer 已做好发送的准备，FCC 开始取出 buffer 中数据通过 SDMA 将数据从发送 buffer 传送到当前发送 FIFO 并使能 TX_EN。这样前导、帧起始定界符、目的 MAC 地址、源 MAC 地址、长度/类型、数据依次发送。当达到当前 buffer 的结尾且 TxBD[L]=1，表明该帧结束，则将 FCS 字节附加到数据后发送出去，使 TX_EN 无效，FCC 将帧的状态位写入 TxBD 并清除[R]位，这样直到该 BD 又由内核准备好后才会被使用。如果达到当前 buffer 的结尾且 TxBD[L]=0，表明该帧由多个 buffer 组成，那么只是清除[R]位。

如果 TxBD[I]＝1，则此 BD 相关的 buffer 被使用后 FCCE[TXB]或 FCCE[TXE]被置位，如果使能这 2 个 bit 可产生中断。这样内核可以在每个帧、每个 buffer 或某特定 buffer 后被中断。

然后 CP 指向下一个 TxBD，等待[R]位被置位以处理下一个发送 buffer。当 CP 检测到 TxBD 的[W]＝1，则在此 BD 相关的 bufer 被使用后，CP 使用 TxBD 表的第一个 TxBD。在全双工模式下，因为忽略冲突，发送只要保证帧间隙（96 个串行时钟）即可。

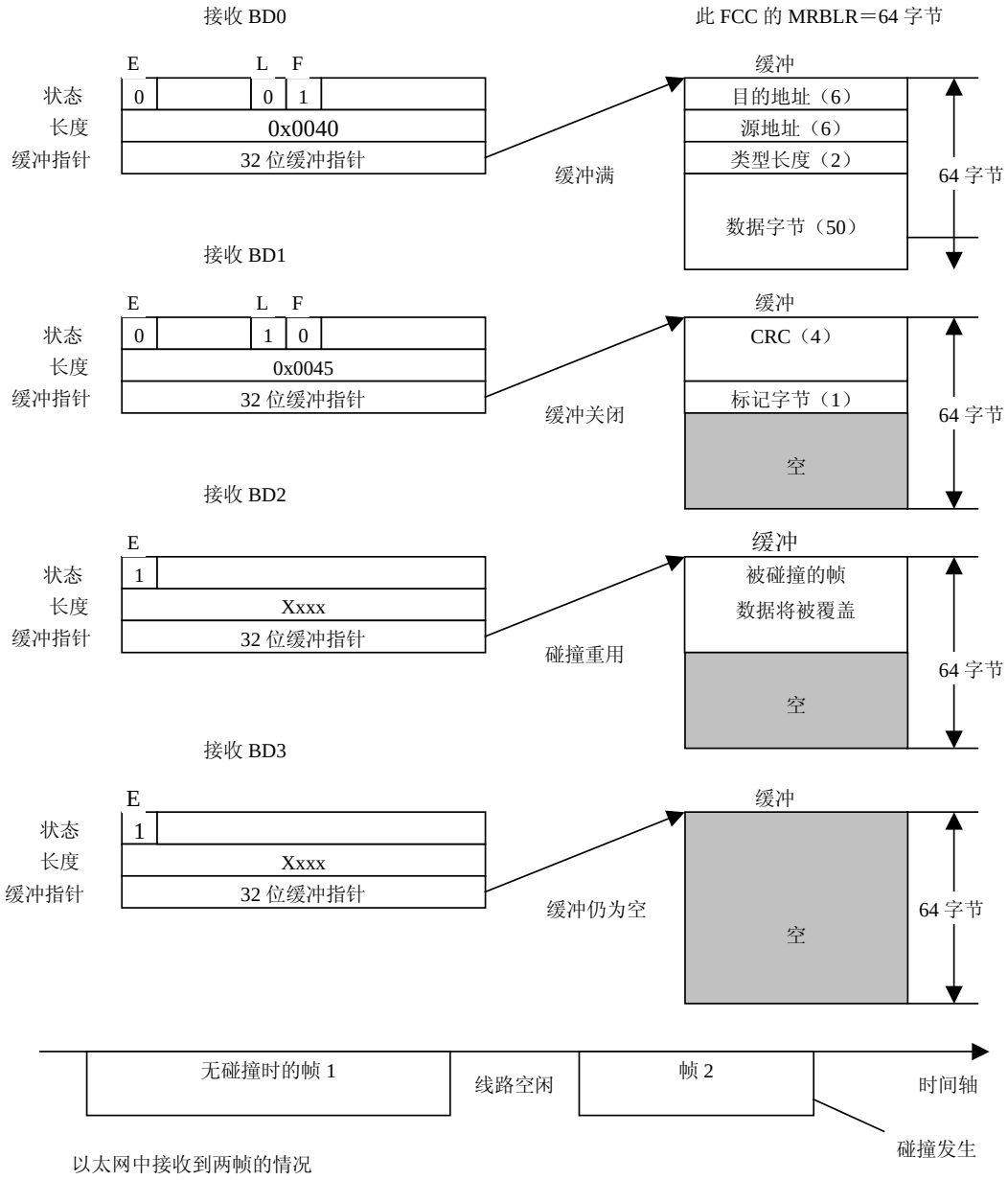

图 4-13　MPC8260 使用 RxBD 实现的以太网接收过程

4.3　MPC8260 编程特点

4.3.1　数据格式和指令格式

1．数据类型和大小

MPC8260 数据的最高位是 0 位，在最左边，最低位在最右边。所以字节 LSB 是位 7，半个

字的 LSB 是位 15，字的 LSB 位是 31。MPC8260 数据类型和大小如表 4-4 所示。

表 4-4　　数 据 类 型

数 据 类 型	大　　小
Byte	8 bits
Half word	2 byte
Word	4 bytes
Double word	8 bytes
Quad word	16 bytes

2．存放模式

MPC8260 支持两种字的存放模式：big endian 和 little endian，通过 MSR[31]控制，如表 4-5 所示。

表 4-5　　存 放 模 式

位 序 模 式	MSR[31]
big endian 模式（默认）	1
little endian 模式	0

例如， 0x54ac870b 在内存中存放如下：

big-endian :高字节放在低地址，低字节放在高地址。

0b
87
Ac
54

little-endian : 高字节放在高地址，低字节放在低地址。

54
Ac
87
0b

3．指令格式

MPC8260 指令是 32 位（word）指令格式，在内存中必须字对齐。它具有两种典型的操作格式。使用第一种格式时，源操作数是 GPR（普通目的寄存器）和 16 位立即数，操作结果存入 GPR。例如，

```
addi r3,r4,0x0760
or   r5,r2,0x80
```

其中寄存器 r1、r2…r31 对应于普通目的寄存器 GPR1、GPR2…GPR31。

当第二种格式时，两个源操作数都是 GPR，结果存入 GPR。例如，

操 作 指 令	操 作 结 果
Add RD,RA,RB	RD = RA + RB
Sub RD,RA,RB	RD = RA - RB
Mul RD,RA,RB	RD = RA * RB
Div RD,RA,RB	RD = RA / RB
Subf RD,RA,RB	RD = RB - RA

4.3.2　指令分类

MPC8260 指令按用途可分为逻辑和代数操作指令，数据加载/存储指令，分支/陷阱和寄存器指令，处理器控制指令，I/O 控制和多进程同步指令和 Cache 指令。

1．逻辑和代数操作指令

这种指令源操作数是 GPR 或 16 位数，结果存入 GPR。

这些指令包括：add　subf　neg　mul　div　rotate　shift　cmp　xor　eqv　and　nand　or　nor　ext　cntlzw。

2．数据加载/存储指令

加载（Load）指令是将内存数据放入 GPR（32 bit），如果数据类型是字节或半个字，必须将符号位扩展后再存入 GPR；存储指令是将 GPR 内容存入内存。

这类指令有 lbz　lha　lhz　lwz　lhbrx,stb　sth　stw　sthbrx　stmw　stsw 等。

3．分支/陷阱和条件寄存器指令

分支指令相当于跳转指令，有条件跳转和无条件跳转。跳到系统层可用陷阱指令。另外还有专门操作条件寄存器内部位（bit）的指令。

这类指令包括：b　bc　bccctr　bclr,trap　sc　rfi,　crand　cror　crxor　creqv。

4．处理器控制指令

处理器控制可通过 SPR（特殊目的寄存器）实现。SPR 内容不能直接用立即数写，也不能直接读入内存。只能借助于 GPR 过度。例如，将 0x0008 放入计数寄存器用下面指令：

```
li    r3,0x0008       # r3 为 GPR3
mtspr CTR, r3         # 或 mtspr 9,r3
```

5．I/O 控制和多处理器同步指令

eieio　：等待以前的 I/O loads/stores 指令执行完毕。

isync　：等待所有以前操作的完成，同时清空指令队列。

sync　：等待所有以前操作的完成。

lwarx　：多处理器对共享资源的同步。

stwcx　：多处理器对共享资源的同步。

6．分支指令和目标地址

表 4-6 给出了分支指令的语法以及目标地址的算法，目标地址可以是另一例程

Subroutine 的入口地址，也可以是本 Subroutine 内的相对地址。

表 4-6　　分支指令

语　法	分 支 操 作	目 标 地 址
B Bl	相对跳转 在 LR 中保存下条指令的地址,再做相对跳转	目标地址:LI + 分支指令地址 范围:从分支指令起 32M
Ba Bla	绝对跳转 在 LR 中保存下条指令的地址,再做绝对跳转	目标地址:LI 范围:0x00000000 起 32M
bcctr bcctrl	如果满足条件，跳转到在 CTR 中地址 保存下条指令于 LR, 如果满足条件,跳转到在 CTR 中地址	目标地址:CTR[0-29]||0b00 范围:4G 字节
Bclr bclrl	如果满足条件，跳转到 LR 中的地址 保存下条指令于 LR, 如果满足条件,跳转到 LR 中的地址	目标地址:LR[0-29]||0b00 范围:4G 字节
Bc Bcl	满足条件就相对跳转 在 LR 中保存下条指令的地址,如满足条件就做相对跳转	目标地址:LI + 分支指令地址 范围:从分支指令起 32M

从上表可看出指令中有“a”的是绝对跳转,有“c”的是条件跳转,有’“r”的返回地址在 LR 中，最后字母是’l’的要将下条指令的地址存入 LR。如果要调用其它子程序最好用 bxxl 指令，返回调用程序用 bxlr 指令。

7．程序间调用

MPC8260 子程序调用指令会把返回地址存入 LR,调用程序在调用其他程序前先将 LR 压入栈中，在调用指令之后立即把栈中地址弹出到 LR。

MPC8260 汇编代码调用 C 代码时，C 的参数都是以 unsigned long 类型传递的，如果有返回值也是 unsigned long。GPR3，GPR4…GPRn(n<32)分别对应于从左到右的实参 1,2,…n-2,返回值存入 GPR3。

8．内存访问(ram access)

对内存的访问包括向内存写（Store）数据和从内存中读（Load）数据，这些动作必须通过 GPRs 即普通目的寄存器组来完成。计算要访问内存的有效地址有如下 3 种方法：

EA = RA + D　　例如，lbzu r6,10(r8) # EA = r8 + 10 ,r8 = EA

EA = RA + RB　　例如，stw r7,r8,r9　# EA = R8 + r9

EA = RA （仅用于字符串存取）

以上 RA、RB 是 GPR0～GPR31，D 是 16 位带符号数（EA 计算时扩展到 32 位 ，EA 为计算得到的有效地址。RA=GPR0 是一种特殊情况，此时 0 将取代 RA。如果指令中包含 Update 选项，那么 RA 将被 EA 刷新。

另外，还有一种特殊的内存访问就是压栈和出栈，具体由软件维护。GPR1 用于指向栈顶的指针,压栈时先将 word 压入,然后 GPR1=GPR1-4。出栈时先将 word 弹出,然后 GPR1=GPR1+4。可见, 压栈是从高地址向低地址移动。例如，

```
Stw    r5,-12(r1)      将 r5 压入堆栈
Lwz    r5,-12(r1)      从堆栈中弹出 r5
```

4.3.3　特殊功能寄存器

MPC8260 寄存器非常重要，它们有些用于记录运算结果，有些用于控制程序的执行,还有些用于保存当前机器的状态。这些特殊功能寄存器的读写都可能有硬件的动作。

1．条件寄存器（CR）

条件寄存器（32 bit），可分为 8 个域，即从 CR0 到 CR7，每个域 4 比特。每个域的 4 个比特用于记录整数运算或比较的结果，有小于（Less than）、大于（ Greater than）、等于（Equal）和溢出（Summary overflow）4 种情况，分别对应 LT，GT，EQ 和 OV 4 个位，如图 4-14 所示。

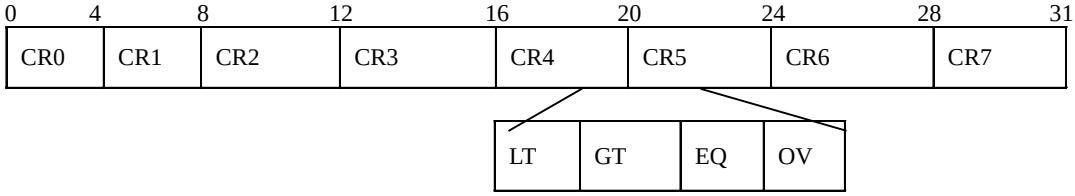

图 4-14　条件寄存器

例如，指令 cmp 4,0,r13,r14，CR4 的 4 个位设置表 4-7 所示。

表 4-7　　CR4

位	条　件	比　　较
0	LT	r13 < r14
1	GT	r13 > r14
2	EQ	r13 = r14
3	OV	XOR[SO]的复制

上面的位通常作为分支指令如 be 和 bne 指令的跳转条件。

2．异常寄存器(XER)

XER（SPR1）寄存器用于存放整数运算异常信息，比较重要的位是 SO（0 位），OV（1 位）和 CA（2 位）。表 4-8 对 3 个重要的指令异常标志做了说明。

表 4-8　　异常标志位

异常 标志位	SummaryOverflow（SO）	Overflow （OV）	Carry （CA）
置位条件	任何指令，只要产生 Overflow 就被设置。一旦设置，用 mtspr 清除	指令引起溢出，该位被设置，并立即复制到 SO。如 addo 和 addco	进位标志。Addc 和 subc 都可能引起该位变化

3．记数寄存器（CTR）

在软件编程中，CTR 可用于存放循环次数。例如下列指令控制循环 8 次：

```
        li   r3,0x0008      #将循环次数放入 GPR3
        mtspr CTR,r3        #将 8 放入 CTR
LOOP:    . . .
         . . .
        bdnz  LOOP      #CTR=CTR-1,if CTR!=0 跳到 LOOP
```

CTR 的另一用途就是存放分支指令如 bcctr 和 bcctrl 的目标地址。

4.3.4　高速缓存控制

MPC8260 高速缓存分为指令 Cache 和数据 Cache，各 4K 大小。同时分双路 128 套，每套 2 线，每线 4 个字（16 Bytes）。Icache 是只读（ReadOnly）的，Dcache 可读可写。每线有 3 个状态，如表 4-9 所示。

表 4-9　　高速缓存的状态

D(dirty)	数据只写进 Cache，未写进 ram。（仅针对 DCache）
L(lock)	只能访问，但不能被其它取代（以 tag 区分）
V(valide)	有效不为空

1．Cache 模式控制

从 MMU 角度方面考虑，程序员可通过设置 WT 和 CI 位达到控制 Cache 模式的目的。

WT（WriteThrough）:该位决定 Cache 是 WriteThrough 还是 Copyback 模式，可通过设置 MD_CTR[WTDEF]实现。其取值如表 4-10 所示。

表 4-10　　WT 含义

WT	含　义	软件实现方法
1	WriteThrough 数据既写入 Cache 又写入 ram	Lis r3,0x1000 #bit 3 置 1 mfspr r4,MD_CTR or r4,r4,r3 mtspr MD_CTR,r4
0	Copyback 数据只写入 Cache	将上面第一条语句的源操作数改为 0,其它相同

CI（Inhibated）:决定是否使用 Cache，通过设置 Mx_CTR[CIDEF]实现。例如，我们可以按如下指令编程：

```
Lis  r3,0x2000      #bit 2 置 1, MI_CTR[CIDEF]=1, Cache inhibated
mfspr r4,MI_CTR    #读 MI_CTR 到 r4
or r4,r4,r3          #使 r4[2]=1
mtspr MI_CTR,r4    #把 r4 写入 MI_CTR
```

2．Cache 操作控制

可通过 6 个特殊目的寄存器控制对 Cache 的操作，如表 4-11 所示：

表 4-11　　控　制　器

数　　据	指　　令	作　　用
DC_CST	IC_CST	Cache 控制和状态寄存器
DC_ADR	IC_ADR	Cache 地址寄存器
DC_DAT	IC_DAT	Cache 数据寄存器（只读）

其中，DC_CST 和 IC_CST 寄存器各有一命令域，通过对这个域的设置可达到控制 Cache 操作的目的。值得注意的是，必须在指令 mtspr xC_CST,r3 后跟 isync 指令，以保证 cache 的正确操作。表 4-12 列出了命令域的不同含义：

表 4-12　　域

IC_CST[4-6]	含　　义	DC_CST[4-7]	含　　义
000	保留	0000	保留
001	Cache enable	0001	设置强制 WriteThrough 位
010	Cache disable	0010	Cache enable
011	Load and lock cache block	0011	清除强制 WriteThrough 位
100	Unlock cache block	0100	Cache disable
101	Unlock all	0110	Load and lock cache block
110	Invalidate all	1000	Unlock cache block
111	保留	1001	保留
		1010	Unlock all
		1011	保留
		1100	Invalidate all
		1110	Flush cache block

为了确保复位后对 Cache 的正确操作，必须按以下步骤对 cache 操作，首先 Unlock all(解锁所有 cache)，然后 Invalidate all（将所有 cache 设置为无效），最后 Enable caches（设置 cache 使能）。

4.4　BSP 最小系统设计

本文所述主控单板的核心电路是由 MPC8260 组成的最小处理系统，它由 MPC8260、地址线/数据线/控制线的驱动、BOOT ROM、FLASH Memory 和 SDRAM 构成。BOOT ROM 存储 CPU 的 BOOT 引导程序，Flash Memory 存储驻留在本板的应用程序，SDRAM 存储软件程序及运行过程中的临时数据和消息队列以及敏感数据等。主控单板的基本框架如图 4-15 所示。

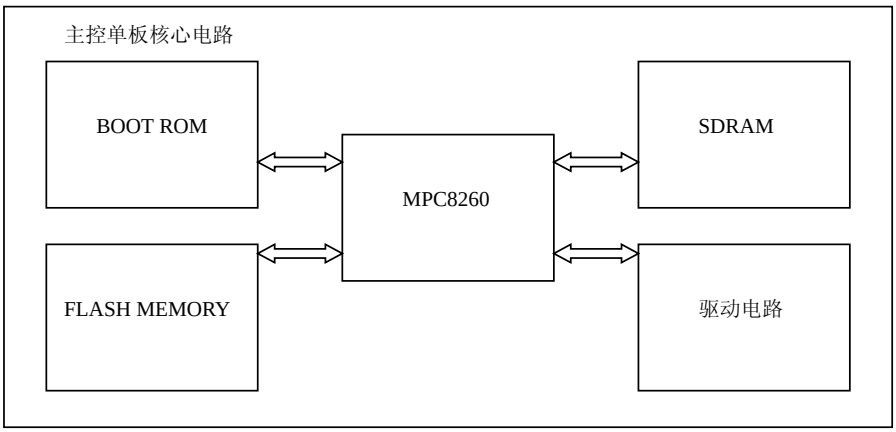

图 4-15　主控单板的基本框架

4.4.1　BOOT ROM 配置编程

1．硬件配置

BOOT ROM 用于存放系统 BOOT 程序，使用 SST 的 8M 位的 flash EPROM，它的数据线为 8 位,大小为 1024KB 便于存放占较大空间的 BOOT 程序。我们将该内存空间定位于 0xFFF00000～0xFFF80000。该 BOOT 程序只需要烧一次，因此不用再更新，它的/WE 引脚被置于无效电平。

由于系统复位后首先执行 BOOT 的程序,MPC8260 的引脚/CS0 在复位后可以被配置为有效，FLASH EPROM 的/CE 端被连到 MPC8260 的 CS0。接着，根据 FLASH EPROM 手册，并结合主控单板的时钟频率，选择适当的时序配合 Flash 的读写。

2．寄存器设置

对于 BOOTROM 来说与之相关的主要寄存器就是 Base Register(BR0)和 Option Register(OR0)。

BR0－基址寄存器

位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
字段	BA															
复位	这个值取决于硬件复位配置字															
R/W	R/W															
地址	(IMMR&FFFF0000)+0x10100															
位	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
字段	BA	—		PS		DECC		WP	MS			EME MC	ATOM		DR	V
复位	0	00		00		00		0	000			0	000		0	0
R/W	R/W															
地址	(IMMR&FFFF0000)+0x10102															

BR0－基址寄存器的描述及其设置值：（BR0=0xfff00b01）

位域	名　字	描　　述	设　置　值

0-16	BA	基地址	1111_1111_1111_00000
17-18	—	保留	00
19-20	PS	端口大小	01
21-22	DECC	数据错误改正和检查	01
23	WP	写保护	1
24-26	MS	机器检查	000
27	EMEMC	外部 MEMC 使能	0
28-29	ATOM	原子化操作	00
30	DR	数据管道化	0
31	V	有效位	1

OR0—选择寄存器：

位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
字段	SDAM＋LSDAM															
复位	0000_0000_0000_0000															
R/W	R															
地址	(IMMR&FFFF0000)+0x10104															
位	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
字段	LSDAM	BPD		ROWST			—	NUMR			PMSEL	IBID	—			
复位	0	00		000			0	000			0	0	0000			
R/W	R															
地址	(IMMR&FFFF0000)+0x10106															

OR0－选择寄存器的描述及其设置值：（OR0=0xfff80010）

位域	名　字	描　　述	设　置　值
0-11	SDAM	SDRAM 地址掩码	1111_1111_1111
12-16	LSDAM	低端 SDRAM 地址掩码	10000
17-18	BPD	每个设备的槽位数	00
19-21	ROWST	行起始地址位	000

22	—	保留	0
23-25	NUMR	SDRAM 中行地址线的数目	000
26	PMSEL	页面模式选择	0
27	IBID	禁止同一设备内槽位的交叉	1
28-31	—	保留	0000

4.4.2　程序存储区 Flash 配置

主控单板的外部程序/数据存储区采用 Flash Memory 实现，为保证其容量，我们将其配置为两片 32M 的 Flash，总共为 64MB。内存映射为 0x30000000～0x31000000，用 MPC8260 的/CS1 作为该两片 Flash 的片选。我们采用 Intel 系列的 Flash 存储器，由于 MPC8260 是 32 位高性能处理器，为了达到最高的效率，要求存储区的数据宽度为 32 位，因此选用两片 16 位数据的 Flash 组成一个 16M×32bit（64MByte）的 Flash Memory。

接下来，根据 Flash 使用手册以及主控单板的系统时钟频率为芯片配置适当的读写时序。

最后，在 BSP 中提供如下的 flash 的读写函数供上层应用代码调用。

1．SectorErase 函数

函数定义	int SectorErase (unsigned long Offset);
功能说明	擦除包含偏移地址为 Offset 的该 Sector，每片一个 Sector 的总字节数为 128K。我们将其设计为 2 片串联，所以擦除是从 offset=0 开始，并以 256KB 为一个单位
入口参数	要擦除块中的任一个地址
返回参数	返回 OK 和 FAILURE

2．WriteFlash 函数

函数定义	int WriteFlash (unsigned long Offset, unsigned long DataLen, unsigned long *BufAddr);
功能说明	写 BufAddr 中 DataLen 个字（4Bytes）进入从 flash 基地址开始偏移 Offset 的位置。
入口参数	Offset: 要写入数据的起始位置相对于 FLASH 基址的偏移地址，该 Offset 必须能被 4 整除。 DataLen: 要写入的数据的长度,其单位为字(4 bytes)； BufAddr: 指向要写入的数据的缓冲区
返回参数	返回 OK 和 FAILURE

3．ReadFlash 函数

函数定义	int ReadFlash (unsigned long Offset, unsigned long DataLen, unsigned long *BufAddr) ；
功能说明	读 Offset 中的 DataLen 个字(4 bytes)进入 BufAddr 指向的 buffer
入口参数	Offset ： flash 中的偏移地址，该 Offset 必须能被 4 整除 DataLen: 要读取的数据的长度，其单位为字(4 bytes) BufAddr: 指向存放读取数据的缓冲区
返回参数	返回 OK 和 FAILURE

CPU 在完成 FLASH 的配置以后，就可以像访问硬盘一样读写 FLASH。我们将对 FLASH 操作的程序封装在 C 代码文件 op_flash.c 中。下面给出此文件的内容，同时做扼要的分析性说明。

/********** op_flash.c 程序的开始**********/

```
/*公共宏定义部分*/

#define FLASH_UNIT_BYTE           128*2*1024
#define FLASH_UNIT_WORD          128*2*1024/4
#define FLASH_MASK                0xfffc0000       /* 块掩码*/
#define FLASH_SIZE                  256*128*2*1024
#define BUFFER_WORD_NUM         16              /* 写缓冲区模式中的字数*/

#ifndef NULL
#define NULL                         0
#endif

#ifndef ULONG
#define ULONG unsigned long
#endif

#ifndef USHORT
#define USHORT unsigned short
#endif

#define FLASH_BASE        0x30000000
#define FLASH_OK   1
#define FLASH_FAIL  0

/*操作最大等待时间*/
#define FLASH_ERASE_CHIP_LOOP        0X4000000
#define FLASH_ERASE_SECTOR_LOOP     0X4000000
#define FLASH_WRITE_LOOP              0X4000000

/*函数声明*/

extern int    FlashWrite(ULONG Offset, ULONG DataLen, ULONG *BufAddr);
extern int    FlashRead(ULONG Offset, ULONG DataLen, ULONG *BufAddr);

/*变量声明*/

/*******************************************/
/*以下两个数组太大，为防止堆栈溢出，直接声明 */
/*为静态全局变量。                          */
/******************************************* */

static ULONG BufferStart[FLASH_UNIT_WORD];
static ULONG BufferEnd[FLASH_UNIT_WORD];

/*功能函数声明*/

static int     FlashEraseChip(void);
static int Write_Flash(ULONG Offset, ULONG DataLen, ULONG *BufAddr);
int     FlashEraseSector(ULONG Offset);
```

```
/*****************************************************************************
ReadReg:      此函数用于从 8 位内存映射寄存器读取一个值。
输入参数:     Addr，ULONG 型的寄存器地址。
输出参数:     无。
返回值:       ULONG 型的寄存器值。
*****************************************************************************/
static ULONG ReadReg(ULONG *Addr)
{
     return (*Addr);
}

/*****************************************************************************
WriteReg:      此函数用于将值写入 8 位内存映射寄存器。
输入参数:      Addr，ULONG 型的寄存器地址。
               Value，ULONG 型的写入值。
输出参数:     无。
返回值:       无。
*****************************************************************************/

static void WriteReg(ULONG *Addr, ULONG Value)
{
     *Addr = Value;
     return;
}

/*****************************************************************************
FlashEraseChip:     此函数用于擦除 FLASH。
输入参数:           无。
输出参数:           无。
返回值:             FLASH_OK，表示操作成功。
                    FLASH_FAIL，表示操作失败。
*****************************************************************************/
static int FlashEraseChip(void)
{
     ULONG Loop;
     ULONG BaseAddr;
     ULONG TempReg;

     BaseAddr = FLASH_BASE;
     WriteReg((ULONG *)(BaseAddr), 0x00300030); /* 0030 是擦除芯片的命令*/
     WriteReg((ULONG *)(BaseAddr), 0x00D000D0);/*确认命令*/
     for(Loop = 0; Loop < FLASH_ERASE_CHIP_LOOP; Loop ++)
     {
          TempReg = ReadReg((ULONG *)(BaseAddr));
          TempReg &= 0x00800080;
          if(TempReg == 0x00800080)
               break;
     }
     if(Loop == FLASH_ERASE_CHIP_LOOP)
          return FLASH_FAIL;
```

```
    if(ReadReg((ULONG *)BaseAddr) & 0x00380038 != 0)
        return FLASH_FAIL;

    WriteReg((ULONG *)(BaseAddr),0x00ff00ff);
    return FLASH_OK;
}

/*****************************************************************************
FlashEraseSector:    此函数用来擦除 FLASH 的一个扇区。
输入参数:            Offset, ULONG 型的起始地址偏移。
输出参数:            无。
返回值:              FLASH_OK, 表示操作成功。
                     FLASH_FAIL, 表示操作失败。
*****************************************************************************/
int FlashEraseSector(ULONG Offset)
{
    ULONG Loop;
    ULONG BaseAddr;
    ULONG TempReg;

    BaseAddr = FLASH_BASE;
    WriteReg((ULONG *)(BaseAddr+Offset), 0x00200020);
    WriteReg((ULONG *)(BaseAddr+Offset), 0x00D000D0);
    for(Loop = 0; Loop < FLASH_ERASE_SECTOR_LOOP; Loop ++)
    {
        TempReg = ReadReg((ULONG *)(BaseAddr));
        TempReg &= 0x00800080;
        if(TempReg==0x00800080)
            break;
    }
    if(Loop == FLASH_ERASE_SECTOR_LOOP)
        return FLASH_FAIL;
    if(ReadReg((ULONG *)BaseAddr) & 0x00380038 != 0)
        return FLASH_FAIL;

    WriteReg((ULONG *)(BaseAddr), 0x00ff00ff);
    return FLASH_OK;
}

/*****************************************************************************
Delay:          此函数用来实现延迟功能。
输入参数:       NuSecs, ULONG 型的延迟微妙数。
输出参数:       无。
返回值:         无。
*****************************************************************************/
void Delay(ULONG NuSecs)
{
    ULONG Loop;
    ULONG Count;
    ULONG Seed;
```

```
     Seed = 0x1234;
    while(NuSecs--)
     {
        for (Loop=0; Loop<30; Loop++)
            Count = Loop * Seed;
     }
}

/*****************************************************************************
FlashRead:        此函数用于读取 FLASH 中的数据。
输入参数:         Offset，读取处的地址偏移量。
                  BufAddr，指向缓冲区的指针。
                  DataLen，要读取的字数，一个字表示 4 个字节。
输出参数:         无。
返回值:           FLASH_OK，表示操作成功。
*****************************************************************************/

int FlashRead(ULONG Offset, ULONG DataLen, ULONG *BufAddr)
{
     ULONG Loop;
     ULONG *Addr;

     *检查输入参数的有效性*/
     if (Offset < 0)
          return FLASH_FAIL;
     f ((Offset + DataLen*4) > FLASH_SIZE)
          return FLASH_FAIL;
     if((Offset % 4) != 0)
          return FLASH_FAIL;
     f(BufAddr == NULL)
          return FLASH_FAIL;

     Addr = (ULONG *)(FLASH_BASE + Offset);
     for(Loop = 0; Loop < DataLen; Loop ++)
          BufAddr[Loop] = Addr[Loop];
     return FLASH_OK;
}

/*****************************************************************************
Write_Flash：此函数用于将数据保存到 FLASH 中。它要求 FLASH 已经擦除。
           我们使用缓冲程序模式。
输入参数:   Offset，写入处的地址偏移量。
            BufAddr，指向缓冲区的指针。
            DataLen，要写入的字数，一个字表示 4 个字节。
输出参数:   无。
返回值:     FLASH_OK，表示操作成功。
            FLASH_FAIL，表示操作失败。
*****************************************************************************/
static int Write_Flash(ULONG Offset, ULONG DataLen, ULONG *BufAddr)
```

```
{
     ULONG  Count;
     ULONG  CurrentDataLen;
     ULONG  Loop;
     ULONG  BlockRemainWord;
     ULONG  WriteWordCount;
     ULONG  BaseAddr;
     ULONG  *Start;

     Count = 0;
     CurrentDataLen = 0;
     BaseAddr = FLASH_BASE;

     Start = (ULONG *)(BaseAddr + Offset);
     BlockRemainWord = FLASH_UNIT_WORD - ((Offset & ~FLASH_MASK)/4);
     do
     {
          WriteReg((ULONG *)(Start),0x00e800e8);

          for(Loop = 0; Loop < FLASH_WRITE_LOOP; Loop ++) /* 设置循环超时时间*/
          {
               if((ReadReg((ULONG *)(BaseAddr)) & 0x00800080) == 0x00800080)
                    break;
          }
          if(Loop == FLASH_WRITE_LOOP)
               return FLASH_FAIL;

     /************************************************/
/*判断下次写入是否会超出 FLASH 的边界，或者是否          */
/*会超出当前块的边界。                                  */
/************************************************/

          if(((CurrentDataLen + BUFFER_WORD_NUM) >= DataLen)||(BlockRemainWord
<= BUFFER_WORD_NUM))
          {
               if((CurrentDataLen + BUFFER_WORD_NUM) >= DataLen)
               {
                    WriteWordCount = (DataLen-CurrentDataLen) - 1;
               }
               if (BlockRemainWord <= (DataLen-CurrentDataLen))
               {
                    WriteWordCount = BlockRemainWord - 1;
                    BlockRemainWord = FLASH_UNIT_WORD;
               }

          }
          else
          {
               WriteWordCount = BUFFER_WORD_NUM - 1;
```

```
        /*缓冲区大小相当于 16 个字*/

        BlockRemainWord -= BUFFER_WORD_NUM;
    }

    /*终止越界的判断*/

    WriteReg((ULONG *)(Start), ((WriteWordCount<<16) + WriteWordCount));

     /*计算写入缓冲的数据长度*/

    for (Count = 0; Count <= WriteWordCount; Count ++)
    {
     *Start = BufAddr[CurrentDataLen];
     Start++;
     CurrentDataLen++;
    }

    Start--;
    WriteReg((ULONG *)Start, 0x00d000d0);

    /*移位 */

    Start++;

    /*读状态寄存器*/

    for(Loop = 0; Loop < FLASH_WRITE_LOOP; Loop ++)
    {
           if((ReadReg((ULONG *)(BaseAddr)) & 0x00800080) == 0x00800080)
                break;
      }
      if(Loop == FLASH_WRITE_LOOP)
           return FLASH_FAIL;

} while(CurrentDataLen < DataLen);

/*读状态寄存器*/

for(Loop = 0; Loop < FLASH_WRITE_LOOP; Loop ++)
{
     if((ReadReg((ULONG *)(BaseAddr)) & 0x00800080) == 0x00800080)
          break;
}
if(Loop == FLASH_WRITE_LOOP)
     return FLASH_FAIL;
if(ReadReg((ULONG *)(BaseAddr)) & 0x00380038)
     return FLASH_FAIL;

/*检查写入的数据*/
```

```
    WriteReg((ULONG *)(BaseAddr), 0x00ff00ff);      /*返回读数组模式*/
    Start = (ULONG *)(BaseAddr + Offset);
    for(Count = 0; Count < DataLen; Count ++)
    {
        if(Start[Count] != BufAddr[Count])
            return FLASH_FAIL;
    }

    return FLASH_OK;
}

/*****************************************************************************
FlashWrite:     这个函数用于将数据保存到FLASH。
输入参数:       Offset，写入处的地址偏移量。
                BufAddr，指向缓冲区的指针。
                DataLen，要写入的字数，一个字表示4个字节。
输出参数:       无。
返回值:         FLASH_OK，表示操作成功。
                FLASH_FAIL，表示操作失败。
*****************************************************************************/

int FlashWrite(ULONG Offset, ULONG DataLen, ULONG *BufAddr)
{
    int   RetVal;
    ULONG Temp;
    ULONG OffsetStart;
    ULONG OffsetEnd;
    ULONG DataLenStart;
    ULONG DataLenEnd;
    ULONG *Start;
    ULONG Loop;

    /*检查输入参数有效性*/
    if (Offset < 0)
        return FLASH_FAIL;
    if ((Offset + DataLen*4) > FLASH_SIZE)
        return FLASH_FAIL;
    if((Offset % 4) != 0)
        return FLASH_FAIL;
    if(BufAddr == NULL)
        return FLASH_FAIL;

    Start = (ULONG *)(FLASH_BASE + Offset);
    for(Loop = 0; Loop < DataLen; Loop++)
    {
        if(Start[Loop] != BufAddr[Loop])
            break;
    }
    if(Loop == DataLen)
```

```
     return FLASH_OK;

 for(Loop = 0; Loop < DataLen; Loop++)
{
     if(Start[Loop] != 0xffffffff)
          break;
}
if(Loop == DataLen)
{
     RetVal = Write_Flash(Offset, DataLen, BufAddr);
     if (RetVal == 0)
          return FLASH_FAIL;
     else
          return FLASH_OK;
}


/*保存要擦除扇区前面的部分*/
OffsetStart = Offset & FLASH_MASK;
DataLenStart = (Offset & ~FLASH_MASK) / 4;  /* word(4 byte) number */
if (DataLenStart != 0)
{
     FlashRead(OffsetStart, DataLenStart, BufferStart);
}


/*保存要擦除的扇区*/
OffsetEnd = Offset + DataLen * 4;
DataLenEnd = FLASH_UNIT_WORD - ((OffsetEnd & ~FLASH_MASK)/4);
if (DataLenEnd != FLASH_UNIT_WORD)
{
     FlashRead(OffsetEnd, DataLenEnd, BufferEnd);
}

/*擦除要写入的所有扇区*/
for (Temp = Offset; (Temp & FLASH_MASK) < OffsetEnd; Temp += FLASH_UNIT_BYTE)
{
     RetVal = FlashEraseSector(Temp);
      if (RetVal == 0)
           return FLASH_FAIL;
}

/*写入 FALSH */
if (DataLenStart != 0)
{
     RetVal = Write_Flash(OffsetStart, DataLenStart, BufferStart);
     if (RetVal == 0)
          return FLASH_FAIL;
}
```

```
    RetVal = Write_Flash(Offset, DataLen, BufAddr);
    if (RetVal == 0)
        return FLASH_FAIL;

    if (DataLenEnd != FLASH_UNIT_WORD)
    {
        RetVal = Write_Flash(OffsetEnd, DataLenEnd, BufferEnd);
        if (RetVal == 0)
            return FLASH_FAIL;
    }
    return FLASH_OK;
}
/************************ op_flash.c 程序的结尾************************/
```

4.4.3 SDRAM 初始化

主控单板中的 DRAM 配置成容量为 128MB，在设计过程全局地址空间分配时要留有足够的空间，32MB 的容量内存映射为 0x00000000～0x04FFFFFF。硬件设计时采用两片数据宽度为 16 位的 DRAM 组成 32 位的 DRAM 存储区。

由于 SDRAM（Synchronous DRAM）具有公共的系统时钟，所有的 I/O 操作与系统时钟同步，因此 SDRAM 具有更高的存取速度，得到越来越广泛的应用，是现在 DRAM 的主流产品。因此我们选择 SDRAM，元器件选择韩国 HYUNDAI 公司的 64M SDRAM。

接下来，根据 SDRAM 手册，以及主控单板的系统时钟，为芯片配置适当的设置时序、读时序、写时序以及刷新和异常时序。根据这些时序关系，我们就可以进一步配置 MPC8260 UPMA 的 RAM WORD 的数据。

SDRAM 所涉及到的寄存器主要有：BR1、OR1、MAMR 和 MPTPR。

1. Base Register（BR1）

BR1－基址寄存器：

位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
字段	BA															
复位	这个值取决于硬件复位配置字															
R/W	R/W															
地址	(IMMR&FFFF0000)+0x10108															
位	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
字段	BA	—		PS		DECC		WP	MS			EMEMC	ATOM		DR	V
复位	0	00		00		00		0	000			0	000		0	0
R/W	R/W															
地址	(IMMR&FFFF0000)+0x1010A															

BR1－基址寄存器的描述及其设置值：（BR1=0x0000 0b01）

位域	名　字	描　　述	设　置　值
0-16	BA	基地址	0000_0000_0000_00000

续表

位域	名　字	描　　述	设　置　值
17-18	—	保留	00
19-20	PS	端口大小	01
21-22	DECC	数据错误改正和检查	01
23	WP	写保护	1
24-26	MS	机器检查	000
27	EMEMC	外部 MEMC 使能	0
28-29	ATOM	原子化操作	00
30	DR	数据管道化	0
31	V	有效位	1

2. Option Register(OR1)

OR1—选择寄存器：

位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
字段	SDAM＋LSDAM															
复位	0000_0000_0000_0000															
R/W	R															
地址	(IMMR&FFFF0000)+0x1010C															
位	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
字段	LSDAM	BPD		ROWST			—	NUMR			PMSEL	IBID	—			
复位	0	00		000			0	000			0	0	0000			
R/W	R															
地址	(IMMR&FFFF0000)+0x1010E															

OR1－选择寄存器的描述及其设置值：（OR1=0xfe00 0010）

位域	名　字	描　　述	设　置　值
0-11	SDAM	SDRAM 地址掩码	1111_1110_0000
12-16	LSDAM	低端 SDRAM 地址掩码	00000
17-18	BPD	每个设备的槽位数	00
19-21	ROWST	行起始地址位	000
22	—	保留	0
23-25	NUMR	SDRAM 中行地址线的数目	000
26	PMSEL	页面模式选择	0
27	IBID	禁止同一设备内槽位的交叉	1

28-31	—	保留	0000

3. Machine A Mode Register(MAMR)

MAMR—机 A 模式寄存器

位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
字段	BSEL	RFEN	OP		—	AMA			DSA		GOCLA			GPL_A4DIS		RLFA
复位	0000_0000_0000_0100															
R/W	R/W															
地址	(IMMR&FFFF0000)+0x10170															
位	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
字段	RLFA		WLFA				TLFA				MAD					
复位	0000_0000_0000_0000															
R/W	R/W															
地址	(IMMR&FFFF0000)+0x10172															

MAMR—机 A 模式寄存器的描述及取值：（MAMR= 0x4008 4440）

位域	名 字	描 述	设 置 值
0	BSEL	总线选择	0
1	RFEN	刷新使能	1
2-3	OP	命令操作码	00
5-7	AMA	地址复用大小 A	000
8-9	DSA	禁用定时器周期	00
12	—	保留	0
10-12	GOCLA[0-2]	通用功能线	001
13-14	GPLA4DIS	禁止 GPLx4 输出端	0
15-17	RLFA	读循环字段 A	001
18-21	WLFA	写循环字段 A	0001
22-25	TLFA	定时器循环字段 A	0001
25-31	MAD	机器地址	000000

4. Memory Periodic Timer Prescaler Register(MPTPR)

MPTPR—存储器周期性定时器预分频寄存器

位	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
字段	PTP								—							
复位	0000_001x								0000_0000							
R/W	R/W															

地址	(IMMR&FFFF0000)+0x10184

MPTPR—存储器周期性定时器预分频寄存器的描述及取值：（MPTPR=0x0400）

位域	名　字	描　　述	设　置　值
0-7	PTP	周期定时器变换因子： 001x_xxxx：　变换因子为 2， 0001_xxxx：　变换因子为 4， 0000_1xxx：　变换因子为 8， 0000_01xx：　变换因子为 16， 0000_001x：　变换因子为 32， 0000_0001：　变换因子为 64， 其他值均保留。	0000_01xx
8-15	—	保留	0000_0000

4.4.4　CPU 初始化

1．系统频率

系统运行环境是外部晶振 131.072K 和 20M 外部时钟可选，假设系统要求运行于 200MHz 频率。若选用 20MHz 的外部时钟，则当系统复位时，MODCK[1:2]=11，MF=200MHz/5MHz - 1 = 39，则系统运行时钟为 5*(39+1) MHz = 200MHz。

主控单板设计采用外部时钟方案，所以配置 MODCK[1:2]=11。同时将 PLL, Low Power 和复位控制寄存器其他的比特设置为缺省值。如果希望了解其中的细节，可以查阅 MPC8260 用户手册。

2．系统保护

（1）总线监视（Bus Monitor）

当一个总线周期开始时，一个/TA 信号应该在限定的时间内产生，否则产生/TEA 信号并产生中断。我们通过设置寄存器禁用该功能。

（2）软件看门狗定时器（Software Watchdog Timer）

MPC8260 系统提供了软件 Watchdog 功能，这是通过软件看门狗定时器来实现的，它可以防止程序陷入不受控制的死循环或被挂死。当系统复位时，该功能被设置为有效，因此若没有在一定的时间内在 SWSR 寄存器中写数据以清零看门狗定时器,则当定时器溢出时会产生一个可选定的动作，即产生复位或产生一个不可屏蔽的中断。

与软件看门狗定时器相关的寄存器包括 SYPCR（System Protection Control Register ）SWSR（Software Watchdog Service Register)。

SYPCR 部分 bit 位解释如下：

- PBME=1/0　　允许/禁止 60x Bus Monitor
- SWE=1/0　　允许/禁止 Watchdog
- SWRI=1/0　　当 Watchdog 超时时产生复位/NMI（非屏蔽的中断）
- SWP=1/0　　对系统时钟预分频 2048/不预分频

- SWTC　　SWT 的计数器。

要复位 Watchdog Timer，只需对 SWSR 按顺序写入 0x556C 和 0xAA39 即可。我们在 BSP 中提供 3 个函数来支持该功能。这 3 个函数是：

- void SWTReset (void);　　清除 Watchdog
- void SWTEnable (void);　　允许 Watchdog
- void SWTStop(void);　　停止 Watchdog

一般来说，在软件调试阶段可以禁用该功能。而在软件正常运行后，可以允许该功能。

3．内部存贮器地址

PowerPC MPC8260 的内部存贮器(internal memory)用于存放控制信息。Internal Memory Map Register(IMMR)中存放的是其在系统 4GB 内存空间的映射地址。

IMMR 的结构如下：

Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Field	ISB															–
Reset	Depends on reset configuration sequence.See Section 5.4.1,"Hard Reset Configuration Word."															
R/W	R/W															
Addr	0×101AB															
Bit	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Field	PARTNUM								MASKNUM							
Reset	–															
R/W	R															
Addr	0×101AA															

我们将内部寄存器配置在系统地址的低端，因此可以将 IMMR 设置为全零的形式。

4.4.5　系统软复位

为了保证上层应用可靠运行，在 BSP 中提供一个软件复位函数：

```
void Reset (void);
```

此函数被调用以后，系统程序会从 BOOT 处重新开始运行，包括 SDRAM 的重新初始化等一系列初始化动作。

4.5　接口驱动设计

4.5.1　MPC8260 SCC1-Ethernet 接口的设计

1．Ethernet 接口的硬件配置

主控单板的 Ethernet 接口通过 MPC8260 的 SCC1 外接 MC68160 实现。该接口有两个作用，一是作为与其他硬件单板的数据通道；二是作为软件调试使用，为 HOST Computer 连接目标系统提供一个高速通道。我们的接口芯片选用了 MC68160FB，其与 MPC8260 的接口信号如图

4-16 所示。

序号	MC68160 引脚	MPC8260 引脚	备注
1	RENA	/CD	接收使能
2	RX	RXD	接收数据
3	TCLK	CLKx 任选	发送时钟
4	TENA	/RTS	发送使能
5	RCLK	CLKx 任选	接收时钟
6	CLSN	/CTS	碰撞指示
7	TX	TXD	发送数据
8	CS0：CS1：CS2	VCC：VCC：PC8/CD2	模式选择，片选
9	/LOOP	PC4/CD4	环回诊断允许

图 4-16　接口信号

ETH_TRX_BUS 中下面的信号由 MPC8260 的 I/O 口驱动。

- ETHLOOP：环回诊断允许。为 1 时，置 EEST 为环回诊断模式。正常操作时为 0。我们将其连接到 MPC8260 的 PC4 引脚。
- /ETHEN：EEST 允许（CS2）。我们将其连接到 MPC8260 的 PC8 引脚。其值为 0 表示允许 EEST，其值为 1 表示禁止 EEST。
- Ethernet RX indicator：当以太网口接收到数据时，该 LED 闪烁。
- Ethernet TX indicator：当以太网口发送数据时，该 LED 闪烁。

配合以太网收发器 MC68160，我们选用 PE68026 作为驱动器。作为电气连接，外接 RJ45 插座，再接双绞线。

2．Ethernet 接口的驱动程序设计

不管是调试阶段中使用 Ethernet 接口，还是在调试结束后的实际应用环境中，Ethernet 接口用作系统的数据通道，都要求 Ethernet 接口的驱动程序必须按 VxWorks 系统设备驱动程序的规范编写。

VxWorks 系统开发环境 Tornado 提供了 MPC8260 的 Ethernet 接口驱动程序。由于我们的 Ethernet 接口的硬件设计同样采用的是 MC68160FB 接口芯片，唯一不同在于片选 CS2，所以只需要将 PC8 配置成通用 I/O 的输出线，并置为低电平选中 CS2 即可。其他 Ethernet 接口驱动程序可以完全借鉴 ADS8260 评估板的内容。

3．Ethernet 接口驱动程序模板

在 VxWorks 系统中针对 Ethernet 接口，首先需要初始化物理端口，然后需要提供物理端口读写的驱动程序。我们可以基于 VxWorks 的 END 驱动接口规范模板编写自己的驱动程序。下面介绍这两部分程序。

初始化物理端口的工作主要在文件 sysMotFccend.C 中完成，限于篇幅，下面列出其重要部分进行分析。

```
/*******sysMotFccend.C 程序中初始化以太网端口的部分的开始**********/

/*包含头文件*/

#include "vxWorks.h"
#include "config.h"

#include "vmLib.h"
#include "stdio.h"
#include "sysLib.h"
#include "logLib.h"
#include "stdlib.h"
#include "string.h"
#include "end.h"
#include "intLib.h"
#include "miiLib.h"

/* 宏定义 */

#ifdef INCLUDE_MOT_FCC

#define MOT_FCC_NUM           0x2      /* 使用 FCC 的数目*/
#define MOT_FCC_TBD_NUM      0x40     /* TBD 的数目*/
#define MOT_FCC_RBD_NUM      0x20     /* RBD 的数目*/
#define MOT_FCC_PHY_ADDR         0x0     /*物理地址 */
#define MOT_FCC_DEF_PHY_MODE     0x2     /* 物理端口的默认操作模式*/
#define MOT_FCC_FLAGS          0x4     /* 用户标记*/

/* IMPORT 函数 */

IMPORT END_OBJ *   motFccEndLoad (char *);

/* 本地变量定义 */

/*************************************************/
/*下面这张表格表示物理端口支持的不同物理规范。可以 */
/*改变表格中项目的顺序，也可以在表格添加新的项目。*/
/*表格中项目的顺序表明了 FCC 尝试使用不同物理层规  */
/*范建立连接的协商顺序。                          */
/*************************************************/

LOCAL INT16 motFccAnOrderTbl [] =
{
 MII_TECH_100BASE_TX,     /* 100Base-T */
 MII_TECH_100BASE_T4,     /* 10Base-T */
 MII_TECH_10BASE_T,       /* 100Base-T4 */
 MII_TECH_10BASE_FD,      /* 100Base-T FD*/
 MII_TECH_100BASE_TX_FD, /* 10Base-T FD */
 -1
```

```
};

/*******************************************************************/
/* sysMotFccEndLoad －此函数用于装载 motFccEnd 驱动程序的一个实例。      */
/* 此函数装载 motFccEndmotFccEnd 驱动程序时使用两个参数。在程序中       */
/* 它查看 BCSR3 以找出系统中使用的处理器类型，然后设置适当的装载        */
/* 字符串。                                                          */
/*******************************************************************/

END_OBJ * sysMotFccEndLoad
    (
    char * pParamStr,     /* 指向初始化参数字符串的指针*/
    void * unused        /*可选参数*/
    )
    {
    /*******************************************************************/
    /* motFccEnd 驱动程序的 END_LOAD_STRING 必须是如下形式：          */
    /* <immrVal>:<fccNum>:<bdBase>:<bdSize>:<bufBase>:<bufSize>:<fifoTxBase>*/
    /* :<fifoRxBase>:<tbdNum>:<rbdNum>:<phyAddr>:<phyDefMode>:<userFlags>:*/
    /* <pAnOrderTbl>                                                   */
    /*******************************************************************/

    char * pStr = NULL;
    char paramStr [300];
    LOCAL char motFccEndParamTemplate [] =
              "0x%x:0x%x:-1:-1:-1:-1:-1:-1:0x%x:0x%x:0x%x:0x%x:0x%x:0x%x:0x%x:0x%x";
    END_OBJ * pEnd;

    if (strlen (pParamStr) == 0)
        {

/*****************************************/
/* 函数 muxDevLoad() 会调用本函数两次。        */
/* 如果调用时传入的字符串长度为 0，那么第      */
/*一次将运行到此处。                         */
/*****************************************/

        pEnd = (END_OBJ *) motFccEndLoad  (pParamStr);
        }
    else
        {

        /*********************************************/
        /*当 muxDevLoad() 第二次调用本函数时，程序将运*/
/*行到此处，这时我们将创建初始化参数字符串。          */
        /*********************************************/

        pStr = strcpy (paramStr, pParamStr);

        /*此处将到达字符串的末端*/
```

```
        pStr += strlen (paramStr);

        /* 这里完成初始化参数字符串 */

        sprintf (pStr, motFccEndParamTemplate,
          (UINT) vxImmrGet (),
          MOT_FCC_NUM,
          MOT_FCC_TBD_NUM,
          MOT_FCC_RBD_NUM,
          MOT_FCC_PHY_ADDR,
          MOT_FCC_DEF_PHY_MODE,
          &motFccAnOrderTbl,
          MOT_FCC_FLAGS
          );

        if ((pEnd = (END_OBJ *) motFccEndLoad  (paramStr)) == (END_OBJ
*)ERROR)
            {
            logMsg ("Error: motFccEndLoad  failed to load driver\n",
             0, 0, 0, 0, 0, 0);
            }
        }

    return (pEnd);
    }
#endif /* INCLUDE_MOT_FCC */

/*******sysMotFccend.C 程序中初始化以太网端口的部分的结尾**********/
```

完成对 Ethernet 物理端口的初始化以后，需要根据 END 和 MUX 接口规范编写 Ethernet 接口的驱动程序，主要用于实现数据的接收和发送。下面列出 Ethernet 驱动程序的模板。在编写其他类似的 Ethernet 接口驱动程序时，均可以借鉴此模板稍做修改即可。

```
/****************** templeEnd.c 程序的开始****************/
/*说明：此程序是 END 网络接口驱动程序的示范程序。*/

/* 包含头文件*/

#include "vxWorks.h"
#include "stdlib.h"
#include "cacheLib.h"
#include "intLib.h"
#include "end.h"                 /* 包含公共 END 结构 */
#include "endLib.h"
#include "lstLib.h"             /* 包含此文件是为了维持协议列表*/
#include "wdLib.h"
#include "iv.h"
#include "semLib.h"
```

```
#include "etherLib.h"
#include "logLib.h"
#include "netLib.h"
#include "stdio.h"
#include "sysLib.h"
#include "errno.h"
#include "errnoLib.h"
#include "memLib.h"
#include "iosLib.h"
#undef   ETHER_MAP_IP_MULTICAST
#include "etherMultiLib.h"      /* 支持多播的头文件 */

#include "net/mbuf.h"
#include "net/unixLib.h"
#include "net/protosw.h"
#include "net/systm.h"
#include "net/if_subr.h"
#include "net/route.h"

#include "sys/socket.h"
#include "sys/ioctl.h"
#include "sys/times.h"

IMPORT  int endMultiLstCnt (END_OBJ* pEnd);

/* 宏定义 */

/* 配置项目的宏定义*/

#define END_BUFSIZ      (ETHERMTU + ENET_HDR_REAL_SIZ + 6)
#define EH_SIZE          (14)
#define END_SPEED_10M     10000000      /* 速率为 10Mbps */
#define END_SPEED_100M    100000000     /* 速率为 100Mbps */
#define END_SPEED       END_SPEED_10M

/* 高速缓存的宏定义*/

#define END_CACHE_INVALIDATE(address, len) \
        CACHE_DRV_INVALIDATE (&pDrvCtrl->cacheFuncs, (address), (len))

#define END_CACHE_PHYS_TO_VIRT(address) \
        CACHE_DRV_PHYS_TO_VIRT (&pDrvCtrl->cacheFuncs, (address))

#define END_CACHE_VIRT_TO_PHYS(address) \
        CACHE_DRV_VIRT_TO_PHYS (&pDrvCtrl->cacheFuncs, (address))

/*************************************************************/
/*以下是 BSP 接口的默认宏定义。这些宏定义可以在其他文件再次定义*/
/*成用户自己的值。                                           */
/*************************************************************/
```

```
/* 将中断处理程序连接到中断向量的宏定义*/

#ifndef SYS_INT_CONNECT
#define SYS_INT_CONNECT(pDrvCtrl,rtn,arg,pResult) \
     { \
     IMPORT STATUS sysIntConnect(); \
     *pResult = intConnect ((VOIDFUNCPTR *)INUM_TO_IVEC (pDrvCtrl->ivec), \
                    rtn, (int)arg); \
     }
#endif

/* 清除中断处理程序与中断向量连接的宏定义*/

LOCAL VOID dummyIsr (void) { };

#ifndef SYS_INT_DISCONNECT
#define SYS_INT_DISCONNECT(pDrvCtrl,rtn,arg,pResult) \
     { \
     IMPORT STATUS intConnect(); \
     *pResult = intConnect ((VOIDFUNCPTR *)INUM_TO_IVEC (pDrvCtrl->ivec), \
                    dummyIsr, (int)arg); \
     }
#endif


/* 启用中断级别的宏定义*/

#ifndef SYS_INT_ENABLE
#define SYS_INT_ENABLE(pDrvCtrl) \
     { \
     IMPORT void sysLanIntEnable(); \
     sysLanIntEnable (pDrvCtrl->ilevel); \
     }
#endif

/*从 BSP 获得以太网地址的宏定义*/

#ifndef SYS_ENET_ADDR_GET
# define SYS_ENET_ADDR_GET(pDevice) \
     { \
     IMPORT unsigned char sysTemplateEnetAddr[]; \
     bcopy ((char *)sysTemplateEnetAddr, (char *)(&pDevice->enetAddr), 6); \
     }
#endif

/*快速访问内存芯片的宏定义*/

#ifndef TEMPLATE_OUT_SHORT
#define TEMPLATE_OUT_SHORT(pDrvCtrl,addr,value) \
```

```
     (*(USHORT *)addr = value)
#endif

#ifndef TEMPLATE_IN_SHORT
#define TEMPLATE_IN_SHORT(pDrvCtrl,addr,pData) \
     (*pData = *addr)
#endif

/* 从 MIB II 快速获取硬件地址的宏定义*/

#define END_HADDR(pEnd)\
          ((pEnd)->mib2Tbl.ifPhysAddress.phyAddress)

#define END_HADDR_LEN(pEnd) \
          ((pEnd)->mib2Tbl.ifPhysAddress.addrLength)


/* 数据结构类型定义 */

typedef struct
    {
    int len;
    char * pData;
    } PKT;     /* 一个简单的 DMA 数据包*/

#define TEMPLATE_PKT_LEN_GET(pPkt) (((PKT *)pPkt)->len)

typedef struct rfd
    {
    PKT *  pPkt;
    struct rfd * next;
    } RFD;     /* 简单的接收帧描述符*/

typedef struct free_args
    {
    void* arg1;
    void* arg2;
    } FREE_ARGS;

/* 驱动程序控制结构的定义*/

typedef struct end_device
    {
    END_OBJ  end;     /* 现有 END 结构的重用 */
    Int   unit;          /* 单元号*/
    int              ivec;               /* 中断向量*/
    int              ilevel;              /* 中断级别*/
    char*            pShMem;
    long  flags;
    UCHAR   enetAddr[6];             /* 以太网地址*/
```

```
    CACHE_FUNCS cacheFuncs;          /* 高速缓存函数指针 */
    FUNCPTR     freeRtn[128];
    struct free_args    freeData[128];

    CL_POOL_ID  pClPoolId;           /* 资源池 */
    BOOL    rxHandling;              /* 调度的接收任务*/
    } END_DEVICE;

/*************************************************************/
/*上述定义的结构适用于单个单元的情况，对于多单元设备驱动程序*/
/*需要对 END_DEVICE 结构进行扩充。                           */
/*************************************************************/

M_CL_CONFIG templateMclBlkConfig =     /* 网络 mbuf 配置表*/
    {
    /* mBlks 号     clBlks 号         内存区域           内存大小
      -----------     ----------        ------------         ------------    */
      0,        0,         NULL,          0
    };

CL_DESC templateClDescTbl [] =      /* 网络资源池配置表*/
    {
    /*
    内存分段大小                  数目       内存区域      内存大小
    -----------------                 ----    ------------    -----------
                                                    */
    {ETHERMTU + EH_SIZE + 2,              0,NULL,           0}
    };

int templateClDescTblNumEnt = (NELEMENTS(templateClDescTbl));

NET_POOL templateCmpNetPool;

/* 标志字段的定义*/

#define TEMPLATE_PROMISCUOUS     0x1
#define TEMPLATE_POLLING    0x2

/* 状态寄存器的位定义*/

#define TEMPLATE_RINT        0x1  /* Rx 中断挂起*/
#define TEMPLATE_TINT        0x2  /* Tx 中断挂起*/
#define TEMPLATE_RXON       0x4  /* Rx 使能 */
#define TEMPLATE_VALID_INT  0x3  /* 有一个中断挂起*/

#define TEMPLATE_MIN_FBUF   (1536)  /*第一个缓冲区的大小*/

/*用于调试的宏定义*/

#ifdef DEBUG
```

```
#define LOGMSG(x,a,b,c,d,e,f) \
     if (endDebug) \
         { \
         logMsg (x,a,b,c,d,e,f); \
         }
#else
#   define LOGMSG(x,a,b,c,d,e,f)
#endif

#undef DRV_DEBUG

#ifdef     DRV_DEBUG
#define DRV_DEBUG_OFF     0x0000
#define DRV_DEBUG_RX     0x0001
#define     DRV_DEBUG_TX     0x0002
#define DRV_DEBUG_INT     0x0004
#define     DRV_DEBUG_POLL     (DRV_DEBUG_POLL_RX |
DRV_DEBUG_POLL_TX)
#define     DRV_DEBUG_POLL_RX     0x0008
#define     DRV_DEBUG_POLL_TX     0x0010
#define     DRV_DEBUG_LOAD        0x0020
#define     DRV_DEBUG_IOCTL        0x0040
#define DRV_DEBUG_POLL_REDIR       0x10000
#define     DRV_DEBUG_LOG_NVRAM  0x20000

int     templateDebug = 0x00;
int     templateTxInts=0;

#define DRV_LOG(FLG, X0, X1, X2, X3, X4, X5, X6)                    \
     if (templateDebug & FLG)                                       \
            logMsg(X0, X1, X2, X3, X4, X5, X6);

#define DRV_PRINT(FLG,X)                                        \
     if (templateDebug & FLG) printf X;

#else

#define DRV_LOG(DBG_SW, X0, X1, X2, X3, X4, X5, X6)
#define DRV_PRINT(DBG_SW,X)

#endif

/*本地使用的函数定义*/

/* 静态函数声明*/

LOCAL void     templateReset     (END_DEVICE *pDrvCtrl);
LOCAL void     templateInt     (END_DEVICE *pDrvCtrl);
LOCAL void     templateHandleRcvInt (END_DEVICE *pDrvCtrl);
LOCAL STATUS     templateRecv     (END_DEVICE *pDrvCtrl, char* pData);
```

```
LOCAL void     templateConfig     (END_DEVICE *pDrvCtrl);
LOCAL UINT     templateStatusRead (END_DEVICE *pDrvCtrl);

/* END 特定的接口定义*/

/* 这些接口只对外部程序可见*/

END_OBJ*      templateLoad (char* initString);

LOCAL STATUS     templateStart     (END_OBJ* pDrvCtrl);
LOCAL STATUS     templateStop      (END_OBJ* pDrvCtrl);
LOCAL int          templateIoctl   (END_OBJ* pDrvCtrl, int cmd, caddr_t data);
LOCAL STATUS     templateUnload     (END_OBJ* pDrvCtrl);
LOCAL STATUS     templateSend      (END_OBJ* pDrvCtrl, M_BLK_ID pBuf);

LOCAL STATUS     templateMCastAdd (END_OBJ* pDrvCtrl, char* pAddress);
LOCAL STATUS     templateMCastDel (END_OBJ* pDrvCtrl, char* pAddress);
LOCAL STATUS     templateMCastGet (END_OBJ* pDrvCtrl, MULTI_TABLE* pTable);
LOCAL STATUS     templatePollStart (END_DEVICE* pDrvCtrl);
LOCAL STATUS     templatePollStop (END_DEVICE* pDrvCtrl);
LOCAL STATUS     templatePollSend (END_OBJ* pDrvCtrl, M_BLK_ID pBuf);
LOCAL STATUS     templatePollRcv (END_OBJ* pDrvCtrl, M_BLK_ID pBuf);
LOCAL void         templateAddrFilterSet(END_DEVICE *pDrvCtrl);

LOCAL STATUS     templateParse     ();
LOCAL STATUS     templateMemInit  ();

/* 声明功能函数表*/

LOCAL NET_FUNCS templateFuncTable =
    {
    (FUNCPTR) templateStart,         /* 用于启动设备的函数*/
    (FUNCPTR) templateStop,         /*用于停止设备的函数*/
    (FUNCPTR) templateUnload,      /* 驱动程序的卸载函数*/
    (FUNCPTR) templateIoctl,        /* 驱动程序的 Ioctl 函数*/
    (FUNCPTR) templateSend,        /* 驱动程序的发送函数*/
    (FUNCPTR) templateMCastAdd,   /* 驱动程序的增加多播的函数*/
    (FUNCPTR) templateMCastDel,   /* 驱动程序的禁用多播的函数*/
    (FUNCPTR) templateMCastGet,
    (FUNCPTR) templatePollSend,
    (FUNCPTR) templatePollRcv,
    endEtherAddressForm,
    endEtherPacketDataGet,
    endEtherPacketAddrGet          /* 获取分组地址的函数*/
    };

/*******************************************************/
/* templateLoad 函数用于初始化驱动程序和物理设备。             */
/* 此函数初始化驱动程序和物理设备到可操作状态。此函数的参数      */
/*  initString 用于传递与设备相关的所有信息。这个字符串包括的信    */
```

```
/* 息示例如下： */
/* “寄存器地址”:“中断向量”:“中断级别”:“共享内存指针”: */
/* “共享内存大小”                                          */
/*  返回值：正常情况下返回 END 对象指针，出错时返回 NULL。*/
/*******************************************************************/
END_OBJ* templateLoad
    (
    char* initString     /* 由驱动程序解析的初始化字符串*/
    )
    {
    END_DEVICE      *pDrvCtrl;

    DRV_LOG (DRV_DEBUG_LOAD, "Loading template...\n", 1, 2, 3, 4, 5, 6);

    /*申请设备描述结构*/

    pDrvCtrl = (END_DEVICE *)calloc (sizeof (END_DEVICE), 1);
    if (pDrvCtrl == NULL)
     goto errorExit;

    /* 解析初始化字符串，同时填充设备描述结构 */

    if (templateParse (pDrvCtrl, initString) == ERROR)
     goto errorExit;

    /* 从 BSP 获取以太网地址*/

    SYS_ENET_ADDR_GET(pDrvCtrl);

    /* 初始化结构重的 END 和 MIB2 部分*/

    /* The M2 element must come from m2Lib.h This template is set up
 for a DIX type ethernet device. */

    if (END_OBJ_INIT (&pDrvCtrl->end, (DEV_OBJ *)pDrvCtrl, "template",
                      pDrvCtrl->unit, &templateFuncTable,
                      "END Template Driver.") == ERROR
     || END_MIB_INIT (&pDrvCtrl->end, M2_ifType_ethernet_csmacd,
                      &pDrvCtrl->enetAddr[0], 6, END_BUFSIZ,
                      END_SPEED)
             == ERROR)
     goto errorExit;

    /* 执行内存分配*/

    if (templateMemInit (pDrvCtrl) == ERROR)
     goto errorExit;

    /* 重置和重配设备*/
```

```
    templateReset (pDrvCtrl);
    templateConfig (pDrvCtrl);

    /* 设置可读标志*/

    END_OBJ_READY (&pDrvCtrl->end,
              IFF_UP | IFF_RUNNING | IFF_NOTRAILERS | IFF_BROADCAST
              | IFF_MULTICAST);
    DRV_LOG (DRV_DEBUG_LOAD, "Done loading Template...", 1, 2, 3, 4, 5, 6);

    return (&pDrvCtrl->end);

errorExit:
    if (pDrvCtrl != NULL)
     free ((char *)pDrvCtrl);

    return NULL;
    }

/*************************************************************/
/* templateParse 函数用来拆分和解析初始化字符串。                    */
/* 此函数拆分输入的字符串，然后将值填入到设备驱动程序的控制结构     */
/* 中。VxWorks 系统的 muxLib.o 模块会自动地通过 BSP(configNet.h)  */
/* 将单元号（unit number）传递给用户的初始化字符串中。              */
/* 返回值：正常情况下返回 OK，输入参数无效时返回 ERROR 。          */
/*************************************************************/

STATUS templateParse
    (
    END_DEVICE * pDrvCtrl,
    char * initString       /* 输入字符串 */
    )
    {
    char*tok;
    char*pHolder = NULL;

    /* 下面开始拆分输入的字符串*/

    /* 来自 muxLib.o 的单元号*/

    tok = strtok_r (initString, ":", &pHolder);
    if (tok == NULL)
     return ERROR;
    pDrvCtrl->unit = atoi (tok);

    /* 中断向量 */

    tok = strtok_r (NULL, ":", &pHolder);
    if (tok == NULL)
     return ERROR;
```

```
    pDrvCtrl->ivec = atoi (tok);

    /* 中断级别 */

    tok = strtok_r (NULL, ":", &pHolder);
    if (tok == NULL)
     return ERROR;
    pDrvCtrl->ilevel = atoi (tok);

    DRV_LOG (DRV_DEBUG_LOAD, "Processed all arugments\n", 1, 2, 3, 4, 5, 6);

    return OK;
    }

/*****************************************************/
/* templateMemInit 一此函数用于初始化内存。                    */
/* 此函数用来为网络设备初始化内存，它与设备密切相关。          */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR 。             */
/*****************************************************/

STATUS templateMemInit
    (
    END_DEVICE * pDrvCtrl   /* 要初始化的设备*/
    )
    {
    int count;

    /*如果需要可以首先申请和初始化所有共享内存区域*/

/* 下面将设置一个 END 网络内存池*/

    if ((pDrvCtrl->end.pNetPool = malloc (sizeof(NET_POOL))) == NULL)
        return (ERROR);

    templateMclBlkConfig.mBlkNum = 16;
    templateClDescTbl[0].clNum = 16;
    templateMclBlkConfig.clBlkNum = templateClDescTbl[0].clNum;

    /* 计算所有 M-Blks （内存大块）和 CL-Blks（内存小段）所占用的总内存*/

    templateMclBlkConfig.memSize = (templateMclBlkConfig.mBlkNum *
                        (MSIZE + sizeof (long))) +
                    (templateMclBlkConfig.clBlkNum *
                        (CL_BLK_SZ + sizeof(long)));

    if ((templateMclBlkConfig.memArea = (char *) memalign (sizeof(long),
templateMclBlkConfig.memSize))
        == NULL)
        return (ERROR);
```

```
/*计算所有内存段的大小*/

templateClDescTbl[0].memSize = (templateClDescTbl[0].clNum *
                    (END_BUFSIZ + 8)) + sizeof(int);

/*在高速缓存安全内存中为所有内存段申请申请内存*/

templateClDescTbl[0].memArea =
    (char *) cacheDmaMalloc (templateClDescTbl[0].memSize);

if ((int)templateClDescTbl[0].memArea == NULL)
    {
    DRV_LOG (DRV_DEBUG_LOAD, "system memory unavailable\n",
      1, 2, 3, 4, 5, 6);
    return (ERROR);
    }

/* 初始化内存池*/

if (netPoolInit(pDrvCtrl->end.pNetPool, &templateMclBlkConfig,
                &templateClDescTbl[0], templateClDescTblNumEnt,
          NULL) == ERROR)
    {
    DRV_LOG (DRV_DEBUG_LOAD, "Could not init buffering\n",
      1, 2, 3, 4, 5, 6);
    return (ERROR);
    }

/*******************************************************/
/* 如果需要使用内存段存储接收到的分组，那么事先在此申请。*/
/*******************************************************/

if ((pDrvCtrl->pClPoolId = netClPoolIdGet (pDrvCtrl->end.pNetPool,
 sizeof (RFD), FALSE))
    == NULL)
    return (ERROR);

while (count < templateClDescTbl[0].clNum)
    {
 char * pTempBuf;

    if ((pTempBuf = (char *)netClusterGet(pDrvCtrl->end.pNetPool,
                                          pDrvCtrl->pClPoolId))
        == NULL)
        {
        DRV_LOG (DRV_DEBUG_LOAD, "Could not get a buffer\n",
                 1, 2, 3, 4, 5, 6);
        return (ERROR);
        }
```

```
        /*  可以在这里将指针保存到一个结构体中。*/

        }

    DRV_LOG (DRV_DEBUG_LOAD, "Memory setup complete\n", 1, 2, 3, 4, 5, 6);

    return OK;
    }

/*********************************************************/
/* templateStart －此函数用于启动一个设备。                  */
/*这个函数调用 BSP 函数为设备连接并启动中断，保证设备运行在*/
/*中断模式。                                              */
/*返回值：正常情况下返回 OK，出错时返回 ERROR 。          */
/*********************************************************/

LOCAL STATUS templateStart
    (
    END_OBJ * pDrvCtrl /* device ID */
    )
    {
    END_DEVICE
    STATUS result;

    SYS_INT_CONNECT (pDrvCtrl, templateInt, (int)pDrvCtrl, &result);
    if (result == ERROR)
     return ERROR;

    DRV_LOG (DRV_DEBUG_LOAD, "Interrupt connected.\n", 1, 2, 3, 4, 5, 6);

    SYS_INT_ENABLE (pDrvCtrl);

    DRV_LOG (DRV_DEBUG_LOAD, "interrupt enabled.\n", 1, 2, 3, 4, 5, 6);

    /*在这里可以启用中断并打开设备*/

    return (OK);
    }

/*********************************************************/
/* templateInt －此函数用于处理控制器中断。                  */
/* 当控制器的中断到来时，这个函数将在对应的中断级别上调用。  */
/* 返回值： N/A。                                         */
/*********************************************************/

LOCAL void templateInt
    (
    END_DEVICE  *pDrvCtrl   /*中断的设备*/
    )
    {
```

```
    UCHAR stat;

    DRV_LOG (DRV_DEBUG_INT, "Got an interrupt!\n", 1, 2, 3, 4, 5, 6);

    /* 读取设备状态寄存器*/

    stat = templateStatusRead (pDrvCtrl);

    /* 如果检测中断失败直接返回*/

    if (!(stat & TEMPLATE_VALID_INT)) /* 检测是否是有效的中断 */
     {
        return;    /* 直接返回，不返回任何出错消息*/
     }

    /*下面可以启用中断，清除发送或者接收中断同时清除所有出错标记*/

    /* 这里可以增加出错处理的代码*/

    /* 对于接收到的任何分组都由一个网络任务来处理*/

    if ((stat & TEMPLATE_RINT) && (stat & TEMPLATE_RXON))
        {
        if (!(pDrvCtrl->rxHandling))
            {
            pDrvCtrl->rxHandling = TRUE;
            netJobAdd ((FUNCPTR)templateHandleRcvInt, (int)pDrvCtrl,
                       0,0,0,0);
            }
        }

    /* 这里可以处理发送中断*/
    }


/*******************************************************/
/* templatePacketGet －此函数用来获取下一个接收的消息。*/
/*此函数获取下一个接收的消息失败时返回 NULL 。       */
/*返回值：正常情况下返回指向下一个接收消息的指针，    */
/*如果不能获取下一消息即消息尚未到达，则返回 NULL。*/
/*******************************************************/

char* templatePacketGet
    (
    END_DEVICE  *pDrvCtrl
    )
    {
    /* 在这里需要添加获取下一个接收消息的代码*/

    return (char *)NULL;
```

```
    }


/*******************************************************/
/* templateRecv — 此函数用于处理下一个到达的分组。       */
/* 此函数一次只能处理一个到达的分组。处理时会对分       */
/* 组做出错检查。                                       */
/* 返回值： N/A。                                       */
/*******************************************************/

LOCAL STATUS templateRecv
    (
    END_DEVICE *pDrvCtrl,
    char* pData
    )
    {
    int         len;
    M_BLK_ID     pMblk;
    char*       pCluster;
    char*       pNewCluster;
    CL_BLK_ID     pClBlk;


    /* 在此处可以添加对分组检查的代码*/

    /* 单播数据计数器增一*/

    END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_UCAST, +1);

    /*******************************************************/
/* 在将数据上传到上层协议之前，必须进行一次数据拷贝。   */
    /*******************************************************/

    pNewCluster = netClusterGet (pDrvCtrl->end.pNetPool, pDrvCtrl->pClPoolId);

    if (pNewCluster == NULL)
        {
     DRV_LOG (DRV_DEBUG_RX, "Cannot loan!\n", 1, 2, 3, 4, 5, 6);
     END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
     goto cleanRXD;
        }

    /* 获取一个内存段（cluster）*/

    if ((pClBlk = netClBlkGet (pDrvCtrl->end.pNetPool, M_DONTWAIT)) == NULL)
        {
        netClFree (pDrvCtrl->end.pNetPool, pNewCluster);
          DRV_LOG (DRV_DEBUG_RX, "Out of Cluster Blocks!\n", 1, 2, 3, 4, 5, 6);
     END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
          goto cleanRXD;
```

```
        }

    /*获取一个 内存块 ID（M_BLK_ID ）*/

    if ((pMblk = mBlkGet (pDrvCtrl->end.pNetPool, M_DONTWAIT, MT_DATA)) ==
NULL)
        {
        netClBlkFree (pDrvCtrl->end.pNetPool, pClBlk);
        netClFree (pDrvCtrl->end.pNetPool, pNewCluster);
     DRV_LOG (DRV_DEBUG_RX, "Out of M Blocks!\n", 1, 2, 3, 4, 5, 6);
          END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
     goto cleanRXD;
        }


    END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_UCAST, +1);
    len = TEMPLATE_PKT_LEN_GET ((PKT *)pData);

    pCluster = END_CACHE_PHYS_TO_VIRT (pData);

    /* 将内存段(cluster)与内存块(MBlock)连接起来*/

    netClBlkJoin (pClBlk, pCluster, len, NULL, 0, 0, 0);
    netMblkClJoin (pMblk, pClBlk);

    pMblk->mBlkHdr.mLen = len;
    pMblk->mBlkHdr.mFlags |= M_PKTHDR;
    pMblk->mBlkPktHdr.len = len;

    /* 保证分组数据的连续性*/

    END_CACHE_INVALIDATE (pMblk->mBlkHdr.mData, len);

    DRV_LOG (DRV_DEBUG_RX, "Calling upper layer!\n", 1, 2, 3, 4, 5, 6);

    /* 处理已经完成，可以清除接收缓冲*/

    /* 首先调用上层协议的处理程序*/

    END_RCV_RTN_CALL(&pDrvCtrl->end, pMblk);

cleanRXD:

    return (OK);
    }

/*************************************************************************/
/* templateHandleRcvInt 函数用来针对输入分组完成任务级中断服务。          */
/* 这个函数由中断服务程序间接调用，完成对接收消息的任务级处理。      */
/* 此函数中的双循环用于防止资源竞争。中断处理代码发现 rxHandling     */
```

```
/* 为 TRUE 时，任务代码会迅速将其设置为 FALSE。这种形式的资源竞争  */
/* 虽然不是致命的但偶尔也会造成一定的延迟。只有在下一个分组到来，    */
/* 网络处理任务再次调用此函数时竞争才会解除。                          */
* 返回值： N/A。                                                    */
/*********************************************************************/

LOCAL void templateHandleRcvInt
    (
    END_DEVICE *pDrvCtrl
    )
    {
    char* pData;

    do
        {
        pDrvCtrl->rxHandling = TRUE;

        while ((pData = templatePacketGet (pDrvCtrl)) != NULL)
            templateRecv (pDrvCtrl, pData);

        pDrvCtrl->rxHandling = FALSE;
        }
    while (templatePacketGet (pDrvCtrl) != NULL);
    }

/*********************************************************************/
/* templateSend 函数用于发送数据。                                   */
/* 驱动程序调用此函数完成数据发送。函数首先获取一个 M_BLK_ID，*/
/* 然后将发送到 M_BLK_ID。应该实现将地址信息安装到这个缓冲区，*/
/* 不过这是高层协议完成的。                                          */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。                    */
/*********************************************************************/

LOCAL STATUS templateSend
    (
    END_DEVICE * pDrvCtrl,        /* 设备指针    */
    M_BLK_ID     pMblk          /* 要发送的数据*/
    )
    {
    int         oldLevel;
    BOOL        freeNow = TRUE;

     /*****************************************************/
     /* 获取对发送程序的访问权。当系统中有多个协议栈同时发送 */
     /* 时，这样做是十分必要的。                           */
     /*****************************************************/

    if (!(pDrvCtrl->flags & TEMPLATE_POLLING))
     END_TX_SEM_TAKE (&pDrvCtrl->end, WAIT_FOREVER);
```

```
    /* 下面这种情况是常见的情况即所有数据位于一个 M_BLK_ID 中 */

    /* 在本地结构中将指针指向数据*/

    /* 提出一个发送请求*/

    if (!(pDrvCtrl->flags & TEMPLATE_POLLING))
        oldLevel = intLock ();

    /*  此处初始化设备发送*/

    /* 更新管理索引*/

    if (!(pDrvCtrl->flags & TEMPLATE_POLLING))
     END_TX_SEM_GIVE (&pDrvCtrl->end);

    if (!(pDrvCtrl->flags & TEMPLATE_POLLING))
        intUnlock (oldLevel);

    /*更新统计计数器 */

    END_ERR_ADD (&pDrvCtrl->end, MIB2_OUT_UCAST, +1);

     /**************************************/
     /* 在此处驱动程序必须释放分组。          */
     /**************************************/

    if (freeNow)
        netMblkClChainFree (pMblk);

    return (OK);
    }

/************************************************************/
/* templateIoctl －此函数用于实现驱动程序的 I/O 控制功能。     */
/* 此函数可以响应 I/O 控制请求。                              */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。          */
/************************************************************/
LOCAL int templateIoctl
    (
    END_DEVICE * pDrvCtrl,
    int cmd,              /* 控制命令码*/
    caddr_t data          /* 命令参数*/
    )
    {
    int error = 0;
    long value;

    switch (cmd)
        {
```

```
    case EIOCSADDR:
     if (data == NULL)
      return (EINVAL);
        bcopy ((char *)data, (char *)END_HADDR(&pDrvCtrl->end),
        END_HADDR_LEN(&pDrvCtrl->end));
        break;

    case EIOCGADDR:
     if (data == NULL)
      return (EINVAL);
        bcopy ((char *)END_HADDR(&pDrvCtrl->end), (char *)data,
          END_HADDR_LEN(&pDrvCtrl->end));
        break;

    case EIOCSFLAGS:
     value = (long)data;
     if (value < 0)
      {
      value = -(--value);
      END_FLAGS_CLR (&pDrvCtrl->end, value);
      }
     else
      {
      END_FLAGS_SET (&pDrvCtrl->end, value);
      }
     templateConfig (pDrvCtrl);
        break;

    case EIOCGFLAGS:
     *(int *)data = END_FLAGS_GET(&pDrvCtrl->end);
        break;

case EIOCPOLLSTART:     /*开始查询操作*/
    templatePollStart (pDrvCtrl);
    break;

case EIOCPOLLSTOP:     /* 结束查询操作 */
    templatePollStop (pDrvCtrl);
    break;

    case EIOCGMIB2:          /* 返回 MIB 统计信息*/
        if (data == NULL)
            return (EINVAL);
        bcopy((char *)&pDrvCtrl->end.mib2Tbl, (char *)data,
              sizeof(pDrvCtrl->end.mib2Tbl));
        break;
    case EIOCGFBUF:          /* 返回第一个缓冲区*/
        if (data == NULL)
            return (EINVAL);
        *(int *)data = TEMPLATE_MIN_FBUF;
```

```
            break;
        default:
            error = EINVAL;
        }

    return (error);
    }

/*************************************************************************/
/* templateConfig 函数用于重配物理接口。                                  */
/* 此函数的重配操作主要包括对接口模式的设置和多播接口列表的更新。        */
/* 返回值： N/A。                                                        */
/*************************************************************************/

LOCAL void templateConfig
    (
    END_DEVICE *pDrvCtrl
    )
    {

    /* 根据需要设置模式*/

    if (END_FLAGS_GET(&pDrvCtrl->end) & IFF_PROMISC)
     {
     DRV_LOG (DRV_DEBUG_IOCTL, "Setting promiscuous mode on!\n",
          1, 2, 3, 4, 5, 6);
     }
    else
     {
     DRV_LOG (DRV_DEBUG_IOCTL, "Setting promiscuous mode off!\n",
          1, 2, 3, 4, 5, 6);
     }

    /* 为多播设置地址过滤*/

    if (END_MULTI_LST_CNT(&pDrvCtrl->end) > 0)
     {
     templateAddrFilterSet (pDrvCtrl);
     }

    /*在此处完全关闭设备*/

    /*在此处复位设备所有的计数器和指针等*/

    /*在此处根据标记初始化硬件*/

    return;
    }

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

```
/* templateAddrFilterSet 此函数用于为多播地址设置地址过滤。            */
/* 此函数遍历多播地址表中的所有的地址然后为设备设置准确的过滤。    */
/* 多播表中的地址是由 endAddrAdd() 函数添加的。                      */
/* 返回值：N/A。                                                     */
/*********************************************************************/

void templateAddrFilterSet
    (
    END_DEVICE *pDrvCtrl     /* 要更新的设备*/
    )
    {
    ETHER_MULTI* pCurr;

    pCurr = END_MULTI_LST_FIRST (&pDrvCtrl->end);

    while (pCurr != NULL)
     {
        /*此处需要设置多播表*/

     pCurr = END_MULTI_LST_NEXT(pCurr);
     }

    /* 此处需要更新设备过滤表*/
    }

/*********************************************************************/
/* templatePollRcv 函数用于接收查询模式的分组。                      */
/* 此函数由上层程序调用，尝试从设备上接收一个分组。                  */
/* 返回值：成功时返回 OK，如果没有分组到来则返回 EAGAIN。            */
/*********************************************************************/

LOCAL STATUS templatePollRcv
    (
    END_DEVICE * pDrvCtrl,     /* 指向要查询的设备*/
    M_BLK_ID      pMblk     /* 指向接收缓冲*/
    )
    {
    u_short stat;
    char* pPacket;
    int len;

    DRV_LOG (DRV_DEBUG_POLL_RX, "templatePollRcv\n", 1, 2, 3, 4, 5, 6);

    stat = templateStatusRead (pDrvCtrl);

    /* 在此处如果没有分组到来则立即返回*/

    if (!(stat & TEMPLATE_RINT))
     {
     DRV_LOG (DRV_DEBUG_POLL_RX, "templatePollRcv no data\n", 1,
```

```
            2, 3, 4, 5, 6);
        return (EAGAIN);
    }

    /* 从设备缓冲区中获取分组和长度*/

    pPacket = NULL;
    len = 64;

    /* 上层应用必须提供一个有效的缓冲区*/

    if ((pMblk->mBlkHdr.mLen < len) || (!(pMblk->mBlkHdr.mFlags & M_EXT)))
     {
        DRV_LOG (DRV_DEBUG_POLL_RX, "PRX bad mblk\n", 1, 2, 3, 4, 5, 6);
          return (EAGAIN);
     }

    /* 此处可清除设置过的所有状态位*/

    /* 此处可以检查设备和分组的错误*/

    END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_UCAST, +1);

    /* 下面将分组拷贝到网络缓冲区中*/

    bcopy (pPacket, pMblk->m_data, len);
    pMblk->mBlkHdr.mFlags |= M_PKTHDR;      /* 设置分组头*/
    pMblk->mBlkHdr.mLen = len;                /* 设置数据长度*/
    pMblk->mBlkPktHdr.len = len;                /* 设置总长度*/

    /* 对分组的处理已经完成，在此处交给设备*/

    DRV_LOG (DRV_DEBUG_POLL_RX, "templatePollRcv OK\n", 1, 2, 3, 4, 5, 6);
    DRV_LOG (DRV_DEBUG_POLL_RX, "templatePollRcv OK\n", 1, 2, 3, 4, 5, 6);

    return (OK);
    }

/*************************************************************/
/* templatePollSend 函数用于在查询模式下发送分组。                  */
/* 上层应用程序在尝试通过设备发送数据时便调用此函数。            */
/* 返回值：正常情况下返回 OK，如果设备忙则返回 EAGAIN 。      */
/*************************************************************/

LOCAL STATUS templatePollSend
    (
    END_DEVICE*    pDrvCtrl,
    M_BLK_ID    pMblk       /* 要发送的分组*/
    )
    {
```

```
    int         len;
    u_short     stat;

    DRV_LOG (DRV_DEBUG_POLL_TX, "templatePollSend\n", 1, 2, 3, 4, 5, 6);

    /* 此处可测试 tx 即发送缓冲是否够用*/

    stat = templateStatusRead (pDrvCtrl);
    if ((stat & TEMPLATE_TINT) == 0)
     return ((STATUS) EAGAIN);

    /* 此处将网络缓冲区置入设备的发送分组中*/

    len = max (ETHERSMALL, pMblk->m_len);

    /* 在此处发送分组*/

    /*修改统计计数器*/

    END_ERR_ADD (&pDrvCtrl->end, MIB2_OUT_UCAST, +1);

    /* 数据被设备接收以后，则释放内存块*/

    netMblkClFree (pMblk);

    DRV_LOG (DRV_DEBUG_POLL_TX, "leaving templatePollSend\n", 1, 2, 3, 4, 5, 6);

    return (OK);
    }

/*******************************************************/
/* templateMCastAdd 此函数用于为设备添加一个多播地址。      */
/* 此函数为设备添加了多播地址以后，复位地址过滤器。         */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。          */
/*******************************************************/

LOCAL STATUS templateMCastAdd
    (
    END_DEVICE *pDrvCtrl,
    char* pAddress                  /* 要添加的新地址*/
    )
    {
    int error;

    if ((error = etherMultiAdd (&pDrvCtrl->end.multiList,
          pAddress)) == ENETRESET)
     templateConfig (pDrvCtrl);

    return (OK);
    }
```

```
/*********************************************************/
/* templateMCastDel 函数用于删除设备的多播地址列表。       */
/* 此函数删除的是驱动程序已经侦听的设备多播地址列表。     */
/* 删除操作完成后就复位地址过滤器                         */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。          */
/*********************************************************/

LOCAL STATUS templateMCastDel
    (
    END_DEVICE *pDrvCtrl,
    char* pAddress          /* 要删除的地址*/
    )
    {
    int error;

    if ((error = etherMultiDel (&pDrvCtrl->end.multiList,
          (char *)pAddress)) == ENETRESET)
     templateConfig (pDrvCtrl);

    return (OK);
    }

/*********************************************************/
/* templateMCastGet 函数用于获取设备的多播地址列表。       */
/* 此函数获取的是驱动程序已经侦听的设备多播地址列表。     */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。          */
/*********************************************************/

LOCAL STATUS templateMCastGet
    (
    END_DEVICE *pDrvCtrl,
    MULTI_TABLE* pTable
    )
    {
    return (etherMultiGet (&pDrvCtrl->end.multiList, pTable));
    }

/*********************************************************/
/* templateStop 函数用于停止一个设备。                     */
/* 这个函数调用 BSP 函数断开设备的中断，并停止设备在中断模  */
/* 式中的操作。                                           */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。          */
/*********************************************************/

LOCAL STATUS templateStop
    (
    END_DEVICE *pDrvCtrl    /* 要停止的设备*/
    )
    {
```

```
    STATUS result = OK;

    /*  此处停止和禁用设备*/

    SYS_INT_DISCONNECT (pDrvCtrl, templateInt, (int)pDrvCtrl, &result);

    if (result == ERROR)
     {
     DRV_LOG (DRV_DEBUG_LOAD, "Could not disconnect interrupt!\n",
          1, 2, 3, 4, 5, 6);
     }

    return (result);
    }

/*******************************************************/
/* templateUnload 函数用于从系统中卸载一个驱动程序。       */
/* 此函数首先关闭设备，然后释放驱动程序装载函数中申      */
/* 请的所有的资源。                                      */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。       */
/*******************************************************/

LOCAL STATUS templateUnload
    (
    END_DEVICE* pDrvCtrl
    )
    {
    END_OBJECT_UNLOAD (&pDrvCtrl->end);

    /* 此处可释放所有共享的 DMA 内存*/

    return (OK);
    }


/*************************************************/
/* templatePollStart 函数用于启动查询模式操作           */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。    */
/*************************************************/

LOCAL STATUS templatePollStart
    (
    END_DEVICE * pDrvCtrl
    )
    {
    int         oldLevel;

    oldLevel = intLock ();

    /* 此处需要关闭中断*/
```

```
    pDrvCtrl->flags |= TEMPLATE_POLLING;

    intUnlock (oldLevel);

    DRV_LOG (DRV_DEBUG_POLL, "STARTED\n", 1, 2, 3, 4, 5, 6);

    templateConfig (pDrvCtrl);      /*重新配置设备*/

    return (OK);
    }

/*************************************************************
/* templatePollStop 函数用于停止查询模式操作。                    */
/* 这个函数用于终止查询模式操作，即设备回到中断模式，设备中断*/
/* 将启用。此函数在切换过程中实现了中断模式以及并针对中断模式*/
/* 进行了适当的重配。                                          */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。              */
/*************************************************************/

LOCAL STATUS templatePollStop
    (
    END_DEVICE * pDrvCtrl     /* 查询的设备 */
    )
    {
    int         oldLevel;

    oldLevel = intLock ();       /* 在更新寄存器时禁用中断*/

    /* 此处需要重新启用中断*/

    pDrvCtrl->flags &= ~TEMPLATE_POLLING;

    intUnlock (oldLevel);

    templateConfig (pDrvCtrl);

    DRV_LOG (DRV_DEBUG_POLL, "STOPPED\n", 1, 2, 3, 4, 5, 6);

    return (OK);
    }

/*****************************************/
/* templateReset 函数用于复位设备。          */
/* 返回值：N/A。                          */
/*****************************************/

LOCAL void templateReset
    (
    END_DEVICE *pDrvCtrl
```

```
    )
    {
    /* 此处复位控制器*/
    }

/************************************************/
/* templateStatusRead 获取当前设备状态。          */
/* 返回值： 状态位信息。                          */
/************************************************/

LOCAL UINT templateStatusRead
    (
    END_DEVICE *pDrvCtrl
    )
    {
    /*在此处添加读取状态寄存器的代码*/
    return (0);
    }

/************* templeteEnd.c 程序的结尾*************************/
```

4.5.2　MPC8260 SMC1-RS232 接口的设计

RS232 接口是主控单板的调试端口和低速数据通信接口。在调试过程中主控单板通过 RS232 接口将单板的引导信息重定向到显示终端。系统运行过程中，主控单板可以通过 RS232 接口配置一些受控设备，通过 RS232 接口接收受监控设备所传来的测量数据，并通过 RS232 接口向受监控设备发送命令字。

RS232 接口的第二层采用 UART 异步通信协议。我们将 MPC8260 CPM 的 SMC1 配置为 UART 控制器，用于实现 RS232 总线接口，波特率定为 9600bit/s。通用异步收发（UART）协议是典型的面向字符的协议，常用于实现设备之间的低速数据通信。UART 的字符格式如图 4-17 所示。

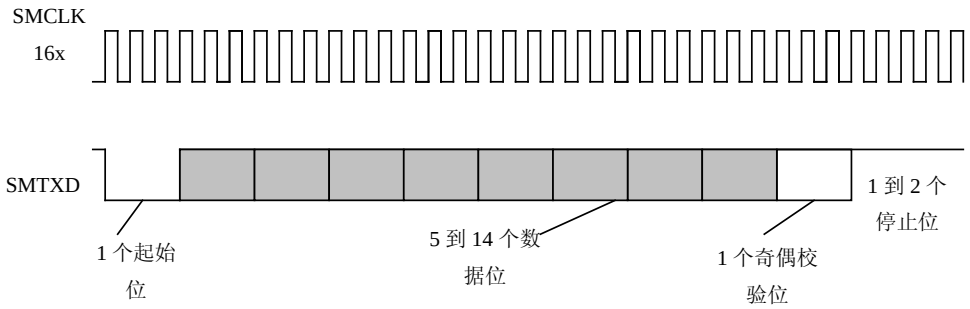

图 4-17　UART 字符格式

由于有多个设备挂接在 RS232 总线接口上，SMC1 的 UART 控制器需要支持多站结构形式。为达到这个目的，考虑到一帧由多个字符构成，我们将其第一个字符作为目的地址。同时，

将 UART 的字符格式扩展一位，以区别地址字符和正常数据字符。

UART 的多站方式支持两种模式即自动多站模式和非自动多站模式，我们选择自动多站模式。

可以通过设置 MAX-IDL（最大空闲字符）或设置多达 8 个控制字符即 CHARACTER1~8 来提供区分各帧的方法。我们使用 MAX-IDL 这种方式。

1．RS232 接口的软硬件配置

（1）硬件占用资源

RS232 接口使用 MPC8260 如表 4-13 所示的管脚资源。

表 4-13　　RS232 管脚说明

管脚	功　能	I/O	说　明
PD9	SMC1_TXD	OUT	[SMC1] RS232 发送数据
PD8	SMC1_RXD	IN	[SMC1] RS232 接收数据

（2）SMC1 配置过程

- 管脚配置。（Port D）

配置 portD9 为 TXD1，作为 SMC1 的 TXD，设定 PDPAR[DD9]=1，设定 PDPIR[DR9]=0。

配置 portD8 为 RXD3，作为 SMC1 的 RXD，设定 PDPAR[DD8]=1，设定 PDPIR[DR8]=0。

- 配置波特率发生器，使用 BRG3，设定 Baud Rate Generator Confguration Registers (BRGC3)。

设定 Clock divider ，BRGC4[CD]=24。

BRGC4[DIV16]=0，不进行 16 分频。

- 配置时钟源，设定 Serial Interface Clock Route Register（SICR）。

R3CS=010，配置 BRG3 为 TCLK。

- 配置 SDMA Configuration Register，（SDCR）
- 配置 SCC Parameter RAM

设定 Big-endian 位序模式。

选择 16bit CRC 类型。

配置最大接收缓冲区长度寄存器。

- 设置 SMC1 通用模式。
- 初始化协议规范模式寄存器。
- 初始化同步字符。

在 UART 模式下，它是用来配置分数停止位，使用缺省值。

- 配置中断：（CPIC）

清除事件寄存器

允许相应的中断

允许 SMC1 中断

清除中断挂起寄存器的相应位

配置优先级，并使能 CPM 中断。

此外，还需在发送、接收 BDs 中设置相应的允许中断的比特。

2．RS232 接口驱动程序的设计

在设计 RS232 接口的驱动程序时完全可以借鉴 VxWorks 的实现方法，因为 VxWorks 为串行接口提供了丰富的库函数。VxWorks 系统将与串口驱动有关的初始化操作封装在文件 sysserial.c 和 m8260Sio.c 中。作为对串行接口驱动的支持，VxWorks 提供了 M8260_SMC_CHAN 结构，用于存储与 SMC 端口有关的参数。下面详细分析这个数据结构。

```
typedef struct
    {
    SIO_DRV_FUNCS *pDrvFuncs;      /*驱动函数      */
    STATUS (* getTxChar) ();          /*指向发送函数 */
    STATUS (* putRcvChar) ();         /* 指向接收函数*/
    void *       getTxArg;
    void *       putRcvArg;

    VINT16      channelMode;         /* 查询或中断模式*/
    Int           baudRate;
    Int           smcNum;            /* 和该 channel 相关联的 SMC 的数目*/

    VINT32      immrVal;             /* 内部存储映射寄存器 */
    char *     pBdBase;                  /* Buffer 描述（BD）的基地址 */
    char *     rcvBufferAddr;             /* 接收 buffer 的地址 */
    char *     txBufferAddr;              /* 发送 buffer 的地址*/

    char ch;

    VINT16 *     pRBASE;           /* RxBD 的基地址 */
    VINT16 *     pTBASE;           /* TxBD 的基地址 */
    } M8260_SMC_CHAN;
```

除了提供 M8260_SMC_CHAN 结构，VxWorks 系统还提供了一个串行通道的管理结构 SIO_CHAN，用于管理所有的串行通道。串行通道的管理结构的内容如下。

```
typedef struct sio_chan
    {
    SIO_DRV_FUNCS * pDrvFuncs;
    /* 设备数据 */
    } SIO_CHAN;
```

此外，为了管理串行驱动程序，VxWorks 系统中提供了串口驱动函数管理结构，它用于管理系统中所有串行口的驱动程序实现函数。串口驱动函数管理结构的内容如下。

```
struct sio_drv_funcs
    {
    int          (*ioctl)           /*I/O 属性控制函数*/
              (
              SIO_CHAN *     pSioChan,
              Int            cmd,
              void *         arg
```

```
            );

    int          (*txStartup)      /*启动发送控制函数*/
            (
            SIO_CHAN *      pSioChan
            );

    int          (*callbackInstall)  /*回调安装函数*/
            (
            SIO_CHAN *      pSioChan,
            Int           callbackType,
            STATUS          (*callback)(void *, ...),
            void *          callbackArg
            );

    int          (*pollInput)       /*输入控制函数*/
            (
            SIO_CHAN *      pSioChan,
            char *          inChar
            );

    int          (*pollOutput)      /*输出控制函数*/
            (
            SIO_CHAN *      pSioChan,
            char            outChar
            );
    };
```

串口驱动函数管理结构内部列出了所有 I/O 驱动的基本函数。实际上串行驱动函数基本上来源于串口驱动函数管理结构。因此 VxWorks 利用内部宏定义将上述函数进行了扩展定义。

```
#define sioIoctl(pSioChan, cmd, arg) \
        ((pSioChan)->pDrvFuncs->ioctl (pSioChan, cmd, arg))
#define sioTxStartup(pSioChan) \
        ((pSioChan)->pDrvFuncs->txStartup (pSioChan))
#define sioCallbackInstall(pSioChan, callbackType, callback, callbackArg) \
     ((pSioChan)->pDrvFuncs->callbackInstall (pSioChan, callbackType, \
             callback, callbackArg))
#define sioPollInput(pSioChan, inChar) \
        ((pSioChan)->pDrvFuncs->pollInput (pSioChan, inChar))
#define sioPollOutput(pSioChan, thisChar) \
        ((pSioChan)->pDrvFuncs->pollOutput (pSioChan, thisChar))
```

分析 sysserial.c 和 m8260Sio.c 以后，可以发现，VxWorks 系统对串行口的支持主要基于一些函数，例如串行口的初始化函数 sysSerialHWinit()完成了串行口的初始化工作。在 VxWorks 系统中编写串行口的驱动程序实际上就是对这些函数稍做修改，满足具体的硬件主板配置要求。为此，我们详细分析这些函数的基本原理和实现过程。

（1）串口初始化函数：sysSerialHwInit

函数原型 void sysSerialHwInit (void)

函数说明：初始化 SMC1 为串口。

实现步骤：

① 复位并行端口，PortA，B，C，D 的寄存器都为 0。

② 设置并行端口的引脚，配置、连接 I/O 引脚到 SMC1。

③ 设置寄存器 M8260_CMXSMR（CMX SMC 时钟分配寄存器），配置 SMC1 所使用的时钟 M8260_CMXSMR(immrVal) = 0x00；初始化 BRG 为 9600 buad；将 SMC 管理结构 M8260_SMC_CHAN 做如下初始化：

```
m8260SmcChan1.channelMode = 0;
m8260SmcChan1.baudRate = DEFAULT_BAUD;
m8260SmcChan1.sccNum = 1;
m8260SmcChan1.immrVal = immrVal;
m8260SmcChan1.pBdBase = (char *) 0x4700000;
m8260SmcChan1.rcvBufferAddr = (char *) 0x04700040;
m8260SmcChan1.txBufferAddr = (char *) 0x04700060;
```

④ 禁止 SMC 的中断；复位 SMC 对应的 channel。到此初始化基本完成。

（2）SMC 初始化函数：m8260SioDevInit

函数原型：void m8260SioDevInit (M8260_SMC_CHAN *pSmcChan)

函数说明：该函数初始化 SMC 为静态。

调用参数：pSmcChan（指向 M8260_SMC_CHAN 结构）

实现步骤：

① 判定通道内部 SMC 的数目是否超过额定值，如果超过额定值，函数直接返回。

② 调用 CACHE_PIPE_FLUSH ()刷新 cache 管道。

③ 禁止接收和发送中断。

④ 设置 SMC 通道的波特率 baudRate 为 DEFAULT_BAUD。

⑤ 定位管理 MC 驱动程序函数的结构体。

（3）I/O 属性控制函数：m8260SioIoctl

函数原型：LOCAL STATUS m8260SioIoctl

```
(
M8260_SCC_CHAN *    pSmcChan,
Int                 request,
Int                 arg
)
```

函数说明： 设置或得到 baud 速率，设置或得到模式

调用参数：pSmcChan－指向 M8260_SMC_CHAN 结构的指针

request－命令代码
arg－其它参数
返回值：OK－成功

EIO－设备出错
ENOSYS－不支持的命令代码

函数实现：此函数主要采用如下 Switch 结构来实现。

```
      /******************* Switch 语句开始*****************************/
switch (request)
     {
     case SIO_BAUD_SET:
         if (arg >=  300 && arg <= 38400)
                {
                baudRate = (baudRateGenClk + (8 * arg)) / (16 * arg);
                /*得到 baud，写到 BRGC 寄存器*/
                  pSccChan->baudRate = arg;
                }
        else
             status = EIO;
         break;

     case SIO_BAUD_GET:
         * (int *) arg = pSccChan->baudRate;
         break;

     case SIO_MODE_SET:
    if (!((int) arg == SIO_MODE_POLL || (int) arg == SIO_MODE_INT))
            {
                status = EIO;
                break;
            }
          /* 锁中断 */
            oldlevel = intLock();
            /*第一次 MODE_SET 时初始化通道*/
            if (!pSccChan->channelMode)
                m8260SioResetChannel(pSccChan);

       if (arg == SIO_MODE_INT)
              {
          /*使能 SMC 中断和 SIU 中断控制器 */
               m8260IntEnable(INUM_SMC1);
            /* 使能 SMC 的接收和发送中断 */
          CACHE_PIPE_FLUSH ();
              }
       else
          {
          /* 禁止接收和发送中断 */
          CACHE_PIPE_FLUSH ();

          /* 禁止 SMC 中断和 SIU 中断 */
       m8260IntDisable(INUM_SMC1);
          CACHE_PIPE_FLUSH ();
```

```
            }
        pSccChan->channelMode = arg;
         intUnlock(oldlevel);
        break;

    case SIO_MODE_GET:
         * (int *) arg = pSccChan->channelMode;
            break;

     case SIO_AVAIL_MODES_GET:
          *(int *)arg = SIO_MODE_INT | SIO_MODE_POLL;
            break;

         default:
            status = ENOSYS;
         }

        return (status);
        }
    }

    /************** Switch 语句结尾*********************************/
```

（4）通道初始化函数：m8260SioResetChannel

函数原型：void m8260SioResetChannel (M8260_SMC_CHAN *pSmcChan)

函数说明：该函数初始化 SMC 通道，配置相关的寄存器、参数 RAM 值、BD 值。

调用参数：pSmcChan－指向 M8260_SMC_CHAN 结构的指针。

实现步骤：

① 首先锁中断，定义各寄存器和参数 RAM 的地址，刷新 Cach 管道。

② 等待直到 CPCR 寄存器的 FLG 为 0（即 CP 没有执行命令）。然后写 CPCR 寄存器执行停止发送命令。

③ 再次等待直到 CPCR 寄存器的 FLG 为 0。然后写 BRGC 寄存器使 BRG3 复位。当 BRG 复位时，调用下面的函数配置 BRG：

```
m8260SioIoctl (pSccChan, SIO_BAUD_SET, pSccChan->baudRate);
```

④ 设置接收 Buffer 的 BD（状态/控制、长度、接收 Buffer 地址）。

⑤ 设置 RxBD 表的基地址。

⑥ 设置发送 Buffer 的 BD（状态/控制、长度、接收 Buffer 地址）。

⑦ 设置 TxBD 表的基地址。

⑧ 配置 SMCM 寄存器禁止发送、接收中断。

⑨ 配置 SMCR 寄存器（除发送使能和接收使能外）。

⑩ 配置 SMC 参数 RAM 中各参数的值。

又一次等待直到 CPCR 寄存器的 FLG 为 0，然后写 CPCR 寄存器执行初始化发送参数、接收参数命令。接着配置 SMCR 寄存器（发送使能和接收使能）。

最后，解锁中断。

（5）中断连接函数：sysSerialHwInit2

函数原型：void sysSerialHwInit2 (void)

函数说明：将 SMC1 与相应的中断处理函数相联

函数实现：调用如下参数一次完成。

```
(void) intConnect (INUM_TO_IVEC(INUM_SMC1),
                   (VOIDFUNCPTR) m8260SioInt, (int) &m8260SmcChan1);
```

（6）中断处理函数：m8260SioInt

函数原型：void m8260SioInt (M8260_SMC_CHAN *pSmcChan)

函数说明：当发生 SMC 中断时调用该函数处理中断。

调用参数：pSmcChan－指向 M8260_SMC_CHAN 结构的指针。

实现步骤：

① 首先判断是否有有接收中断。如果有，则在事件寄存器中写 1 清除接收事件，这样也清除了 SIPNR 中相应的位，然后刷新 Cach 管道。

② 读 RxBD 的状态控制位，当 RxBD[E]为 0 时读出接收 Buffer 中的字符，将 RxBD[E]置 1，表示接收 Buffer 已空。

③ 刷新 Cach 管道，然后将接收到的字符传递给 tty 驱动。如果是查询模式，则 break，否则再读 RxBD，看是否接收到新字符。

④ 如果有发送中断，且不是查询模式，那么在事件寄存器中清除发送事件，这样也清除了 SIPNR 中相应的位，然后刷新 Cach 管道。如果发送函数中有要发送的字符，则一直读 TxBD[R]，直到 TxBD[R]＝0，即 buffer 没有在发送，这时可以刷新。

⑤ 调用 M8260_SCC_8_WR 将要发送的字符写到发送 Buffer，然后设置发送 buffer 的长度，将 TxBD[R]置为 1，开始发送。

⑥ SMCE 设置为清除除发送、接收外的其它中断。

（7）发送开始函数：m8260SioStartup

函数原型：

```
LOCAL int m8260SioStartup   (  M8260_SMC_CHAN *pSmcChan)
```

函数说明： 开始发送。

调用参数：pSmcChan－定义为 M8260_SMC_CHAN 结构

返回值：OK－成功；ENOSYS－当查询模式。

实现步骤：

① 首先判断是否是查询模式，如果是查询模式则返回 ENOSYS 。

② 读 TxBD[R]，如果 TxBD[R]为 0，则进一步判断发送函数中是否有要发送的字符。如果有要发送的字符，将要发送的字符写到发送 Buffer。

③ 设置发送 buffer 的长度，TxBD[R]置为 1，开始发送。

④ 刷新 Cach 管道。

⑤ 如果正常返回 OK。

（8）查询模式下字符输入函数：m8260SioPollInput

函数原型：LOCAL int m8260SioPollInput

```
(
SIO_CHAN *     pSioChan,
char *     thisChar
)
```

函数说明：查询模式下输入字符

调用参数：

- pSioChan－指向 SIO_CHAN 结构的指针
- thisChar－指向输入的字符

返回值：OK－成功；ERROR－设备错误；EAGAIN－没有接收中断或输入 Buffer 为空。

实现步骤：

① 如果 SMCE 中没有接收中断，则返回 EAGAIN。

② 读 RxBD 的状态控制位，如果 RxBD[E]为 1，则返回 EAGAIN。否则说明 RxBD[E]为 0，则从接收 Buffer 中读出字符放到参数 thischar 指向的地方。

③ 将 RxBD[E]置为 1，读 RxBD 的状态控制位，如果 RxBD[E]为 1 则清除 SMCE 中的接收中断。

④ 刷新 Cach 管道。

⑤ 如果正常返回 OK。

（9）查询模式下字符输出函数：m8260SioPollOutput

函数原型：static int m8260SioPollOutput

```
(
SIO_CHAN *     pSioChan,
Char     outChar
)
```

函数说明：查询模式下输出字符

调用参数：pSioChan－指向 SIO_CHAN 结构的指针

```
outChar－指向输出的字符
返回值：OK－成功
ERROR－设备错误
EAGAIN－没有接收中断或输入 Buffer 为空
```

实现步骤：

① 循环 M8260_SCC_POLL_OUT_DELAY 次，保证上一个字符已经发出。

② 读 TxBD 的状态控制位，如果 TxBD[R]为 1，返回 EAGAIN。

③ 清除 SMCE 寄存器中的 Tx 事件，将要发送的字符写到发送 Buffer。

④ 设置发送 buffer 的长度，将 TxBD[R]置为 1，开始发送。

⑤ 刷新 Cach 管道。

⑥ 如果正常返回 OK。

(10) 回调安装函数：m8260SioCallbackInstall

函数原型：static int m8260SioCallbackInstall

```
    (
    SIO_CHAN *  pSioChan,
    Int          callbackType,
    STATUS     (* callback)(),
    void *       callbackArg
    )
```

函数说明：安装 ISR 回调函数用于得到或输出字符。

调用参数：pSioChan－指向 SIO_CHAN 结构的指针

callbackType－回调类型

callback－回调函数

callbackArg－回调参数

返回值：OK－成功；ENOSYS－错误。

函数实现：此函数利用如下简单的 Swtich 结构实现。

```
/***************** Swtich 语句开始 *********************/
    switch (callbackType)
        {
        case SIO_CALLBACK_GET_TX_CHAR:
            pSccChan->getTxChar    = callback;
            pSccChan->getTxArg     = callbackArg;
            return (OK);
          break;

        case SIO_CALLBACK_PUT_RCV_CHAR:
            pSccChan->putRcvChar   = callback;
            pSccChan->putRcvArg    = callbackArg;
            return (OK);
           break;

        default:
            return (ENOSYS);
        }

}
      /************************ Switch 语句结尾 **********************/
```

复位函数：sysSerialReset

函数原型：void sysSerialReset (void)

函数说明：该函数调用 sysSerialHwInit()来复位串口设备

函数实现：直接调用如下函数复位。

```
          sysSerialHwInit ();
```

4.6 BSP 的调试和测试

在系统开发中，作为初始化代码和接口驱动程序的 BSP，担负着向整个上层应用软件提供虚拟运行平台的任务，其稳定性和可靠性是非常重要的，必须做好完善的调试（debugging）和测试工作（testing）。

对 BSP 来说，调试的重要工具是仿真器，如 PowerTap 和 Trace32 等。使用仿真器基本上可以找出初始化代码中的错误。但是 BSP 中接口驱动程序的调试方法与上层应用软件等同。除此之外，还有必要对驱动程序进行严格测试，测试包括以下几个部分：

- 代码质量评估　可以利用 LOGISCOPE 工具进行代码质量评估，它遵循 ISO/IEC 9126 国际标准的可维护性部分定义的 C 代码质量评估准则。主要是分析代码的复杂性系数、可移植性、可靠性和可维护性等。
- 驱动程序的功能性测试　各个接口都有相应的测试模块，由于各个接口主要完成各类不同协议的通信，所以测试其能否正确收发相应帧格式的数据包是最主要的任务，对于担负大数据量通信的通道还必须测试其数据吞吐量。
- 代码的覆盖性测试　另外，可以按照 LOGISCOPE 对 C 代码覆盖测试的原则，对 BSP 驱动程序的代码进行覆盖性测试。

下面将详细介绍示例 BSP 的测试原则、测试内容以及测试结果。

4.6.1 测试内容

我们对主控单板 BSP 的测试内容做如下安排。

1．编码和命名规范的检查，通过自查和走读相结合的方式，检查代码是否符合编码和命名规范。

2．MPC8260 基本系统的测试。

- CPU 性能的测试
- SDRAM 的测试

3．各接口驱动程序的功能性测试。

- FLASH 读写驱动程序的测试

4.6.2 测试项目及结果

1．CPU 性能测试

测试项目：MPC8260 CPU 性能的测试。

测试目的：通过测试 MPC8260 CPU 的 MIPS 数，验证硬件 MPC8260 基本系统设计的正确性以及 BSP 对 MPC8260 初始化配置，CACHE 以及 MMU 设置的正确性。

测试设计：做一个次数很大的循环，在循环内对通用寄存器进行最简单的操作，再在循环的入口和出口计时，计算出循环花费的时间，同时通过计算汇编后的指令数，便可计算出 8260CPU 的峰值 MIPS。

测试用例的设计及测试结果如表 4-14 所示。

表 4-14　　CPO 测试

序　号	测 试 用 例	测 试 结 果
1	在引导任务中做一个大循环: for (loop = 0; loop < 0x3000000; loop ++); 再在循环的入口和出口统计时间开销，计算 CPU 的 MIPS 数。此时 I-cache 是打开的。重复测试 5 次	5 次测试结果如下： CPU speed is 159.376283 MIPS. CPU speed is 159.350203 MIPS. CPU speed is 159.350203 MIPS. CPU speed is 159.376283 MIPS. CPU speed is 159.370203 MIPS.
2	同测试用例 1，只是在测试前关闭 I-cache	CPU speed is 21.7820703 MIPS.

3	在循环中添加对外存的存取操作：
	for (loop = 0; loop < 0x3000000; loop ++)
	{
	Temp = loop & 0xffff;
	}
	其中 Temp 为全局变量
		CPU speed is 128.889807 MIPS.

4	在循环中添加对外存的两次存取操作：
	for (loop = 0; loop < 0x3000000; loop ++)
	{
	Temp1 = loop & 0xffff;
	Temp2 = Temp1 & 0xff00;
	}
	其中 Temp1，Temp2 为全局变量
		CPU speed is 71.372421 MIPS.

2．SDRAM 测试

测试项目：SDRAM 的测试。

测试目的：通过对 SDRAM 的读写测试，验证硬件 MPC 8260 基本系统中 SDRAM 设计的正确性，以及 BSP 对 UPM 配置字的设置是否正确。

测试设计：在硬件初始化 SDRAM 完成后，做一个循环，在循环内对 SDRAM 的所有地址依次进行写操作，完成后再依次读出来，与写入的逐一对比，如果发现不一致，可以通过点灯告警。

测试用例的设计及测试结果如表 4-15 所示：

表 4-15　　SDRAM 测试

序　号	测 试 用 例	测 试 结 果
1	由于测试 SDRAM 必须在 SDRAM 初始化完成之后立即进行,所以要把测试代码放进 BSP 的代码中,烧进 boot 里运行。 在硬件初始化 SDRAM 完成后,做一个循环,在循环内对 SDRAM 的所有地址依次进行写操作，完成后再依次读出来，与写入的逐一对比，如果发现不一致，则发出告警。每当完成一次整个 SDRAM 的读写，就给出提示信息	在连续 12 小时的读写过程中，始终未出现告警。提示信息循环出现

3．FLASH 读写测试

测试项目：FLASH 读写驱动程序的测试。

测试目的：通过模拟上层应用调用 FLASH 读写驱动程序对 flash 进行读写操作，验证该驱动程序的正确性，包括预期功能设计的实现情况，入口参数边界的检查，各种异常情况的处理，错误返回值的验证。

测试设计：设计不同的测试用例，每个测试用例针对要测试的某个方面，分别运行并且记录结果。

测试用例的设计及测试结果如表 4-16 所示。

表 4-16　　FLASH 测试

被测试函数功能	FlashRead( )：该函数用于从 Flash 的 Offset 字节偏移开始读取长度为 DataLen 个长字的数据到 BufAddr 指向的缓冲区。 FlashWrite( )：该函数用于向 Flash 的 Offset 字节偏移开始写入长度为 DataLen 个长字的数据，该数据存放在 BufAddr 指向的缓冲区里。	
序　号	测 试 用 例	测试输出结果描述
1	输入参数为： Offset：-0x200 DataLen：10*1024 BufAddr：Buffer1[10*1024]	输出结果显示： Flash writting fail. 所得结果与程序设计功能相符
2	输入参数为： Offset：0 DataLen：10*1024 BufAddr：Buffer2[10*1024]	输出结果显示： Flash writting success. Flash reading success. Test success. 所得结果与程序设计功能相符
3	输入参数为： Offset：FLASH_SIZE - 20*1024*4; DataLen：20*1024 BufAddr：Buffer3[20*1024] 宏 FLASH_SIZE 表示 FLASH 的大小。	输出结果显示： Flash writting success. Flash reading success. Test success. 所得结果与程序设计功能相符
4	输入参数为： Offset：FLASH_SIZE - 20*1024*4 + 4 DataLen：20*1024 BufAddr：Buffer4[20*1024]	输出结果显示： Flash writting fail. 所得结果与程序设计功能相符
5	输入参数为： Offset：17777 DataLen：10*1024 BufAddr：Buffer5[10*1024]	输出结果显示： Flash writting fail. 所得结果与程序设计功能相符

续表

被测试函数功能	FlashRead( )：该函数用于从 Flash 的 Offset 字节偏移开始读取长度为 DataLen 个长字的数据到 BufAddr 指向的缓冲区。 FlashWrite( )：该函数用于向 Flash 的 Offset 字节偏移开始写入长度为 DataLen 个长字的数据，该数据存放在 BufAddr 指向的缓冲区里。	
序　号	测 试 用 例	测试输出结果描述
6	输入参数为： Offset：FLASH_SIZE - 20*1024*4; DataLen：10*1024 BufAddr：Buffer6[10*1024]	输出结果显示： Flash writting success. Flash reading success. Test success. 所得结果与程序设计功能相符
7	输入参数为： Offset：FLASH_SIZE + 2000; DataLen：10*1024 BufAddr：Buffer7[10*1024]	输出结果显示： Flash writting fail. 所得结果与程序设计功能相符

第 5 章　VxWorks 系统开发环境 Tornado

5.1　Tornado 开发环境概述

VxWorks 系统开发环境 Tornado 是一个可视化、自动化程度很高的集成环境，它能够加快产品的开发。Tornado 开发环境的显著特点是：

- 具有提示用户的向导功能。
- 可以快速配置 VxWorks 系统。
- 具有一个全集成的、简便运行的仿真器。
- 具有 WindView™集成仿真器。
- 具有新一代的图形开发环境和图形调试环境 CrossWind™调试器。
- 具有增强的用于异常处理和支持模板的 C++特性。

Tornado 的最新版本是 2.2 版，与此最新版本配套的 VxWorks 系统版本是 5.5 版。在 Tornado 开发环境中，仿真器 VxSim 运行于主机开发系统中，可以独立于 BSP 通信、独立于操作系统配置，甚至独立于目标硬件，从而可以立即开始应用程序的开发。其集成仿真器还包括了一套 WindView 系统级诊断和分析工具。WindView 提供了嵌入式软件在 VxWorks 下运行时详细可视的动态运行情况，并通过图形化显示任务、中断、系统对象。像信号量、消息队列、信号、多任务、定时器以及用户事件等系统事件都可以根据变化清晰呈现出来。针对实际目标硬件系统，WindView 可用于监控系统行为。

Tornado 集成开发环境（IDE）缩短了配置 VxWorks 的学习过程。因为它具有自动裁剪功能的“Auto Scale”向导按钮，它可以分析用户应用代码并自动裁剪 VxWorks 操作系统，加入所有当前还没有包括进去的必要组件，辨识可以完全清除的组件。Tornado 在分析和辨识操作系统相关联的可选组件以后，确认不需要的功能就给予删除，添加和删除之后，自动计算工程大小的变化，从而随时得出正确的项目大小。Tornado 图形化工程环境还能够自动创建和生成 Makefile 文件，以简化工程的建立。

CrossWind 调试器使用 gdb 做为默认调试引擎，配备了前沿开发所必需的提高产品开发效率的图形功能，支持任务级和系统级断点。它可以在任何一种状态中运行，并且可以自动切换。使用 CrossWind，开发者可以实时在系统中创建和调试对象任务，还可以把 CrossWind 放到已经运行的或者已经调试的任务中去。

Tornado 集成了多种开发工具，而且所有的开发工具在不同的应用开发阶段都可以使用。除此之外，全套工具对于目标连接策略（以太网、串行口、ICE 等）或者目标内存大小都适用。Tornado 拥有 3 个高度集成的组件，首先是 Tornado 套件，它是一个强大的交叉开发工具，可以使用在主机和目标机上；其次是 VxWorks 实时系统，它是一个高性能、可裁剪的实时操作系统，可以在目标处理器中执行任务；最后是包括以太网、串行口、ICE、ROM 仿真器等在内的各种通信软件。

Tornado 中枢体系结构支持动态链接和装载，这种能力可使开发者忽略将应用程序链接到主机核心的公共步骤，然后将其作为一个静态环境下载。这将极大缩短编辑—测试—调试的周期。而且，这样可以分享所有模块，无需再次链接主机到应用模块。可以使用对象模块添加到运行的 VxWorks 目标环境中，以用于调试或者再次配置。

Tornado 具有强大的工具环境。其中，浏览器（Browser）的主窗口显示了目标系统整个状态的摘要信息，开发者可专注于监控个别目标操作系统对象的显示，如任务、信号量、消息队列、内存划分和看门狗定时器等；外壳界面 WindSh™可以解释执行几乎所有的 C 语言表达式，包括那些在系统符号表格中可以看见的名称，如函数和变量的参考，它还具有调试功能；Tornado 的目标工具包括系统诊断、性能监控和应用原型工具。

Tornado 采用 GNU 编译器和 iostream 类库支持 C 语言和更新的 C++语言。除此之外。还提供了可选的 Wind Foundation Classes，其中包括 VxWorks 封装类，这些类封装了类似 semLib 和 taskLib 的 VxWorks 库，同时为目前没有 C++库的 VxWorks 系统提供了 C++接口；Tornado 提供了 RogueWave 软件的 h++工具，此类库已成为一种行业标准，它具有管理数据结构、收集类、字符串、规则表达式、对象和散列表的功能。

Tornado 支持各种范围内的行业标准。其中目标系统支持包括 POSIX 1003.1.b 实时扩充，ANSI C（包含浮点支持）标准和 4.4BSD TCP/IP 网络标准。Tornado 2.2 的核心是高效率的 Wind 微内核。Wind 微内核支持普遍范围内的实时功能，更重要的是它缩减了系统管理开销，可对外部事件做出迅速响应。Tornado 的 VxWorks 实时操作系统具有很强的裁剪性，开发者可拥有多达 100 多种的不同选择以创建数以百计的配置，从千字节内存的嵌入式设计到需要更多操作系统功能的复杂高端实时系统。Tornado 拥有全面的网络工具，它的 TCP/IP 网络支持最新的 Berkeley 网络协议栈如 PPP、BOOTP、DNS、DHCP、TFTP、FTP、telnet、NFS、标准 Berkeley socket 和 zbufs、IP、IGMP、CIDR、TCP、UDP、RIP V.1/V.2 等。

5.2　Tornado 开发环境的安装

实际上，Tornado 开发环境一般使用光盘安装，光盘由经销商或者代理直接提供。本节将对安装 Tornado 开发环境做详细的分步介绍。安装 Tornado 开发环境的重点分为系统安装和 License 注册。一般来说，先安装系统，然后注册 License，这样就可以使用开发环境了。不同的 Tornado 版本注册方法稍有不同，本节的注册方法重点针对 2.2 版本。

5.2.1　安装 Tornado 开发环境

Tornado 开发环境的系统安装包括主机开发系统的安装和 BSP（板级支撑包）的安装。首先需要安装开发系统，如果要做 BSP 开发，则必须在开发系统安装就绪后，完成 BSP 的安装。本节重点介绍在 Windows 2000 环境下对 Tornado 开发系统的安装，在其他主机环境如 UNIX 上安装步骤完全一样。此外，BSP 的安装步骤基本类似。

在 Windows 2000 环境下安装 Tornado 开发系统需要以下步骤。

1．在 Windows 的浏览器中打开 Tornado 开发系统的 CD-ROM，找到一级目录下面的 Setup.exe 文件，直接单击便启动了 Tornado 开发系统的安装流程。我们首先看到的是

“Welcome”窗口，在这个窗口中，安装程序提示关闭所有 Windows 应用程序，尤其是病毒扫描程序。根据提示操作，然后单击“Welcome”窗口的“Next”按钮。

2．接着，安装程序显示“README.TXT”窗口，这个窗口显示了安装 Tornado 开发系统的帮助信息，最好读完这些信息，以便顺利安装。然后单击“README.TXT”窗口的“Next”按钮。

3．这样，我们便进入了安装程序给出的“License Agreement”窗口，如图 5-1 所示。这个窗口给出遵循 Windriver 公司 License 规章制度的信息。单击“Accept”按钮，本窗口的“Next”按钮由灰变亮。然后单击“License Agreement”窗口的“Next”按钮继续安装过程。

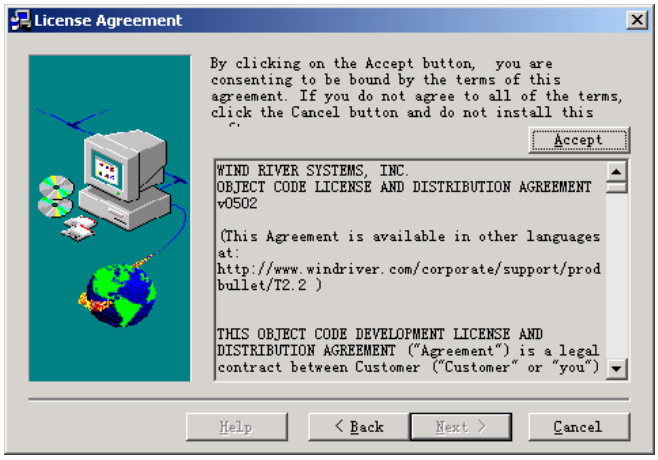

图 5-1　Tornodo 安装程序给出的“License Agreement”窗口

4．将看到“User Registration”窗口，如图 5-2 所示。“User Registration”窗口要求键入特定的注册信息和“Install key”。按图示键入“Name”和“Company”信息，然后将 Tornado 安装系统 CD-ROM 附带表单上的 25 位 Key 信息键入。最后在“User Registration”窗口中单击“Next”按钮。如果 25 位 Key 信息键入有误，系统提示再次输入正确的 25 位 Key 信息。

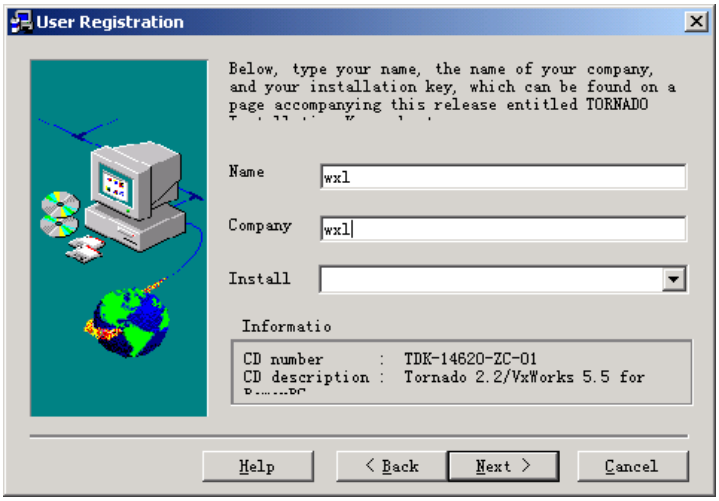

图 5-2　Tornado 安装程序的“User Registration”窗口

5．如果键入了正确的 key 信息，安装程序将从“User Registration”窗口切换到如图 5-3 所示的“Installation Options”窗口。“Installation Options”窗口提示用户选择一种安装类型。可以选择以下几种安装形式：

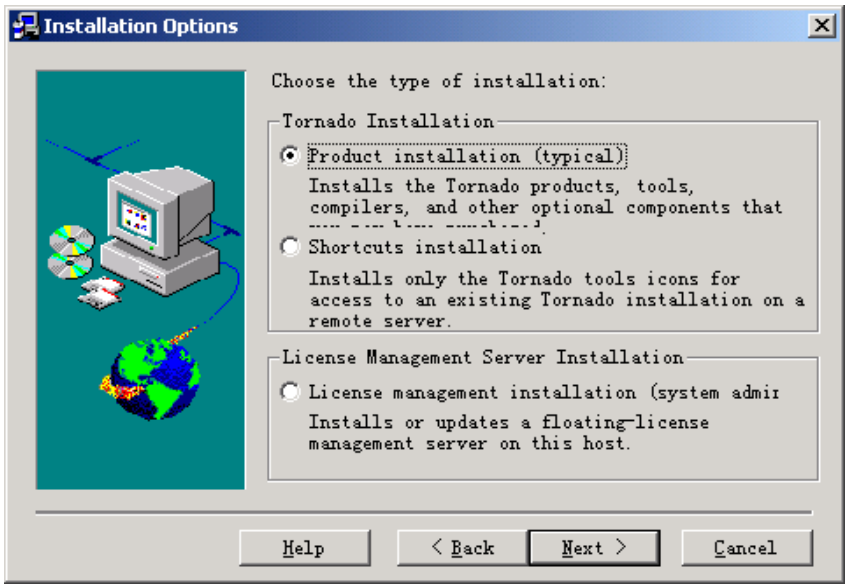

图 5-3　Tornado 安装程序的“Installation Options”窗口

- 产品安装
- 快捷安装
- License 安装

第一项用于产品的第一次安装，它可以安装 Tornado 程序的所有组件。

第二项用于日后的维护性安装或者利用远地服务器实现网络安装的模型，这种安装形式将约束本机 Tornado 开发环境的运行。

第三项用于安装 License 服务器。只有需要将本机安装成为一个 License 服务器实现 License 管理时才需要安装第三项。此选项可以安装和配置“floating”license 服务器，其他机器可以通过网络访问本机的 license 信息。

典型的安装方式即第一项“product installation”。选择典型安装方式，然后单击“Installation Options”窗口的“Next”按钮继续安装。

6．接下来，将看到 Tornado 安装程序的“Project Information”窗口，如图 5-4 所示。在这个窗口中，必须键入自己预计利用 Tornado 开发环境开发的工程信息，包括“WRS License Number”、“Project name”和“Number of tornado seat”。这些信息包括了购买 Tornado 的信息，也包括了购买者使用 Tornado 开发环境的自由度。其中 License 信息“WRS License Number”一定要与“Number of tornado seat”相匹配，因为“Number of tornado seat”限制一个 License 号的开发用户数。一个 License 号的额定开发用户数遵循所签订的商务合同的规定。单击“Project Information”窗口的“Next”按钮继续安装过程。

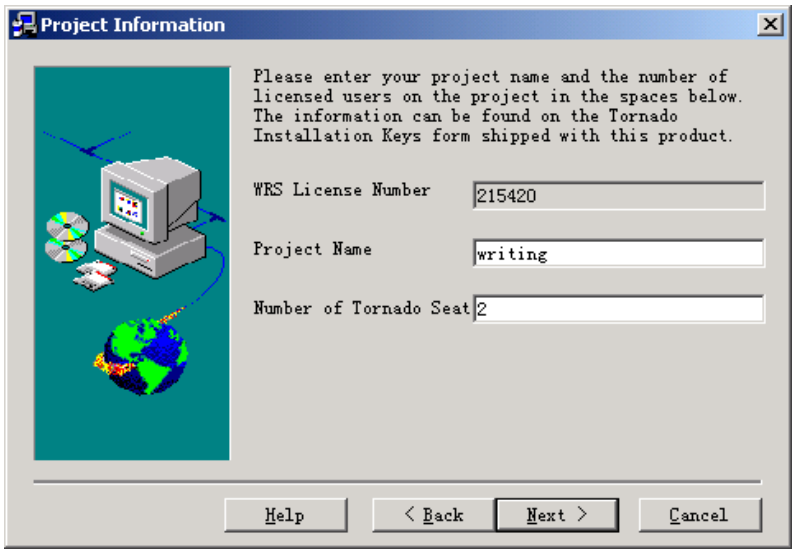

图 5-4　Tornado 安装程序的“Project Information”窗口

7．接下来，将进入 Tornado 安装程序的“select Directory”窗口，如图 5-5 所示。在这个窗口的“Destination Directory”项中填入需要将 Tornado 安装到的实际目录中。值得注意的是不要将其安装到具有空格的目录路径中，例如，目录路径 C:\Progrm Files\Tornado2.2 是非法的。此外，最好不要将 Tornado 安装到以前曾经安装过 Tornado 的目录路径下，这样可能因为版本差异或者其他原因导致系统中一些组件不能使用。也可以利用“Browse”按钮选择一个目录路径。按图 5-5 所示进行选择以后，单击“Select Directory”窗口的“Next”按钮继续下一步安装。

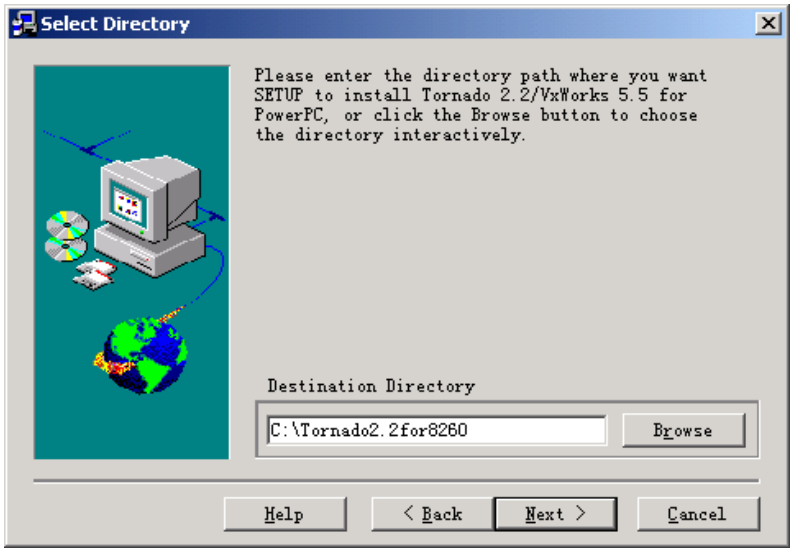

图 5-5　Tornado 安装程序的“select Directory”窗口

8．这时，将看到如图 5-6 所示的 Tornado 安装程序的“select Product”窗口。在这个

窗口中可以选择安装的产品。用户所购买的产品组件都显示在这个窗口中。通过选中产品组件左边的方框可以确认安装的产品和组件。有些产品，例如 Tornado Tools，它可能还包括几个二级组件和产品，针对这样的选项可以选择 Details 按钮实现二级组件和产品的选择。

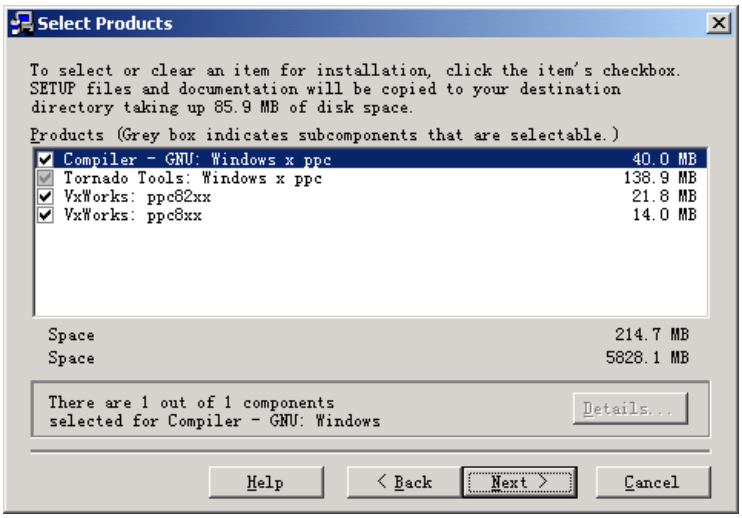

图 5-6　Tornado 安装程序的“select Product”窗口

在“Select Product”窗口中，Details 按钮的左边显示了二级组件和产品的选择个数。Detail 按钮的上方显示安装产品和组件要求的磁盘空间以及系统允许使用的磁盘空间。Tornado 开发环境的最小安装是：

- 主机系统用到的 Tornado 工具包
- 目标系统使用的处理器系列
- 目标系统的驱动及其头文件（属于 BSP）

不过，在默认情况下，安装程序已经选择了这些基本选项。值得注意的是，如果购买了若干可选组件，也可能不能一次完全安装，因为这些组件可能分布在好几个不同的 CD-ROM 上。另外，有些组件之间互相嵌套，如果用户选择了一个组件，但是没有选择它必须用到的其他组件，安装程序将给出提示性警告。发生这种情况时，只需按照提示信息选择组件便可以继续安装过程。

选择“Select Product”窗口中的“Next”按钮继续安装。

9. 接下来，进入“Select folder”窗口，如图 5-7 所示。此窗口要求用户键入“程序文件夹”名字，这个名字是显示在“开始菜单”中程序名字。可以根据需要修改安装程序给出的默认程序名。然后单击此窗口中的“Next”按钮继续安装过程。

10. 这时，安装程序将显示“Tornado Registry”窗口。在这个窗口中需要提供“Tornado Registry”的安装信息。“Tornado Registry”用于管理主机中所有的“Target server”。“Target server”用来实现 Tornado 开发环境的主机与目标系统之间的通信连接。受“Target server”管理的主机工具包括“shell”、“debugger”和“browser”。在安装 Tornado 开发环境的主机中，必须首先运行“Tornado Registry”才可以启动“Target server”。在此窗口中有以下几种选择（如图 5-8 所示）：

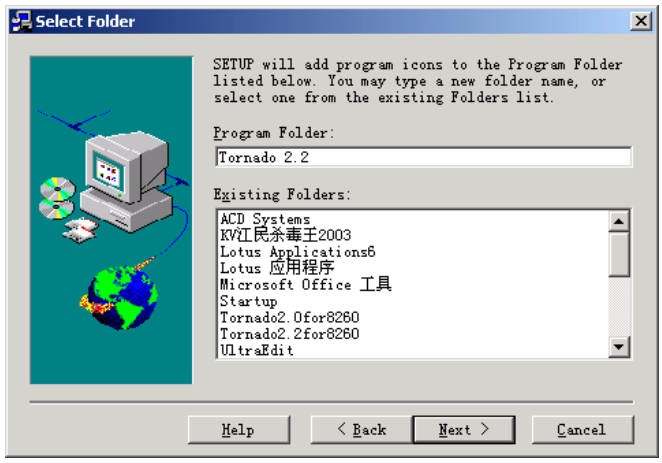

图 5-7　Tornado 安装程序的“select folder”窗口

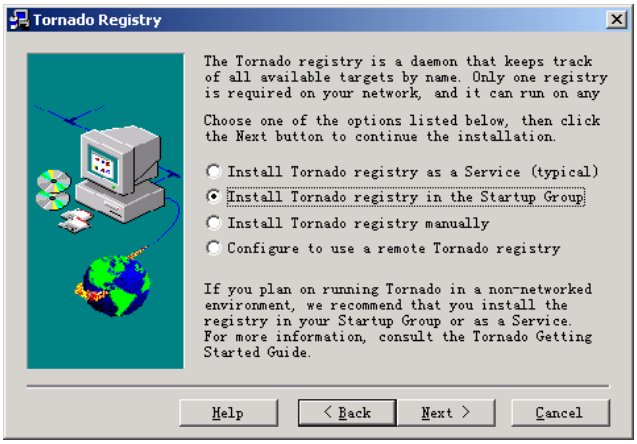

图 5-8　Tornado 安装程序的“Tornado Registry”窗口

- Install Tornado Registry as a service
- Install Tornado Registry in the StartUp Group
- Install Tornado Registry manually
- configure to use a Remote Tornado Registry

选择第一个选项后，系统启动时将自动启动“Tornado Registry”。选择第二个选项后，系统将在安装 Tornado 开发环境的用户登录时自动启动“Tornado Registry”。如果选择了第三个选项,“Tornado Registry”必须由用户根据需要手工启动。当选择第四个选项时,Tornado 开发环境使用其他主机的“Tornado Registry”。

我们选择第二个选项，然后单击“Next”继续下一步安装。

11. 接下来，可以看到“Backward Compatiblity”窗口，如图 5-9 所示。如果希望在开发环境中同时使用 Tornado 1.01 和 Tornado 2.2 工具，那么必须选中“Install portmapper for use with Tornado 1.01”。值得注意的是如果在第 10 步中选择了“Install Tornado Registry as a service”，就无法选择在开发环境中同时使用 Tornado 1.01 和 Tornado 2.2

工具。按图 5-9 所示选择后单击“Backward Compatiblity”窗口中的“Next”按钮继续安装。

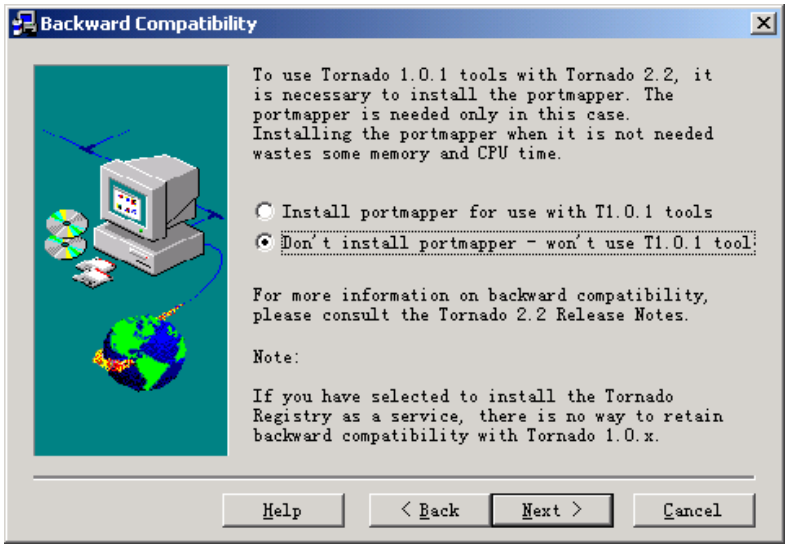

图 5-9　Tornado 安装程序的“Backward Compatiblity”窗口

12. 这时，安装程序将显示 Tornado 的“Configure Defult Application”窗口，如图 5-10 所示。在这个窗口中，可以配置与 Tornado 2.2 开发环境关联的文件应用和文件类型。在 Windows 系统安装 Tornado 2.2 开发环境时，只有具有管理员权限的用户才可以选择配置“File Types”。选择默认的文件类型，然后单击窗口中的“Next”按钮继续安装。

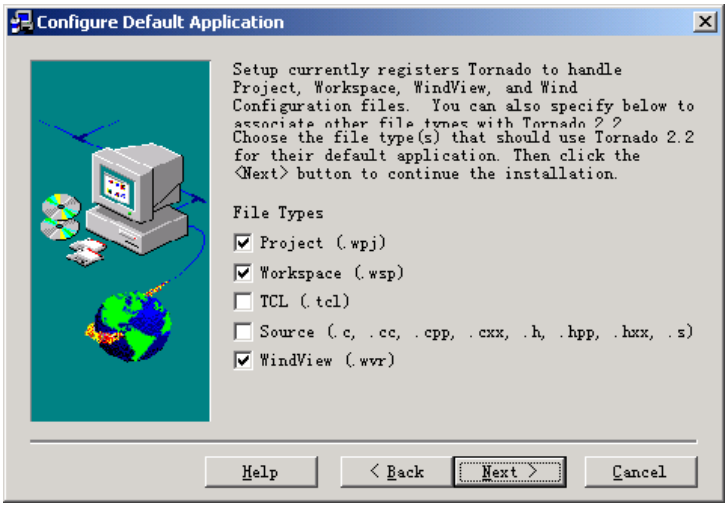

图 5-10　Tornado 安装程序的“Configure Defult Application”窗口

13. 这时，Tornado 安装程序开始拷贝文件开始系统安装。随着安装过程的进行，产品的信息将显示在 FLASH 窗口中。整个系统的安装仅需要几分钟。如果系统中运行了病毒防护程序，那么 Tornado 程序的系统安装速度将显著减慢，为了加快安装速度，最好暂时关闭病毒防护程序。当安装程序拷贝完文件以后将显示一个窗口要求用户确认 Tornado 2.2 程序已

经安装。安装完 Tornado 2.2 以后，可以根据以上步骤安装 BSP 文件。

5.2.2 注册 Tornado 开发环境

根据上一节介绍的方法虽然已经安装了 Tornado2.2 开发环境，但是还不能使用，因为还没有完成注册过程。注册完整过程可以分成以下几个步骤：

1．当我们从程序菜单中启动“Tornado2.2”时，系统将提示“注册管理出错”。同时给出如图 5-11 所示的“License Management Error”窗口。此窗口指出了注册出错的原因即缺乏 License 文件，同时说明了获得 License 文件的方法。可以单击此窗口中的“Next”按钮重新获取 License 文件，完成系统注册。其实，也可以直接运行 Tornado 安装文件所在目录中的 Setup.exe 文件，如上一节中的第 5 步所示，获取和安装 License 文件。

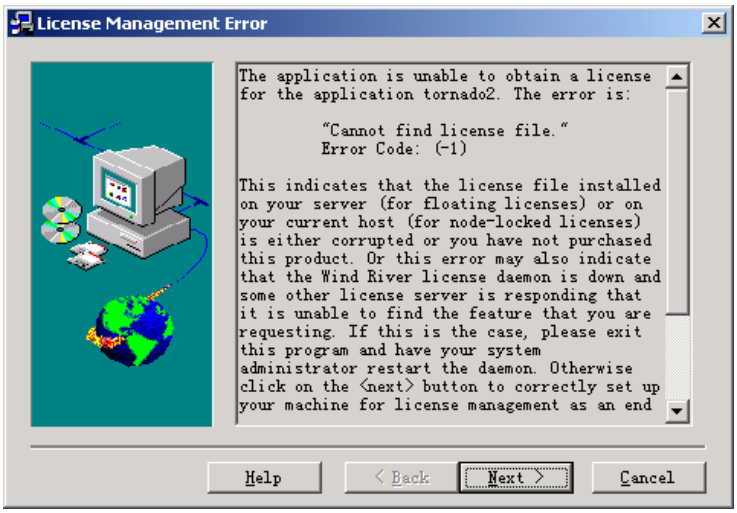

图 5-11 Tornado 环境的“License management Error”窗口

2．这时系统提示键入用户注册信息，键入用户信息后，单击“Next”按钮继续安装。

3．Tornado 安装程序将显示如图 5-12 所示的“License management Configuration”窗口。在这个窗口中可以选择是使用“floating license”，还是使用“node-locked license”。这两种注册的方法有很大的区别。

- floating license 表示从现有 License 服务器上获取 License 信息。这种 License 信息可以在主机之间共享。
- node-locked license 表示固定一个主机使用的 License 文件信息。

在具体应用中，“floating license”服务器的建立过程比较繁琐。如果用户没有建立“floating license”服务器，默认情况下，一般选择“node-locked license”。因此下面的安装步骤实际上主要针对“node-locked license”的安装。

如果用于以前曾经配置过的 License 文件，可以选择确认框“Recapture license previous configured for this machine”。然后单击此窗口的“Next”按钮。

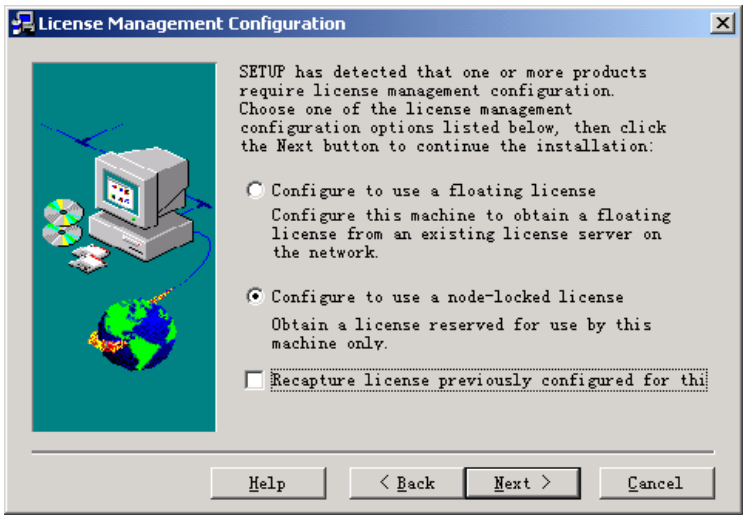

图 5-12　“License management Configuration”窗口

4．License 安装程序将给出如图 5-13 所示的“License management Installation Options”窗口。这个窗口中给出了许多有关 License 安装的选项。

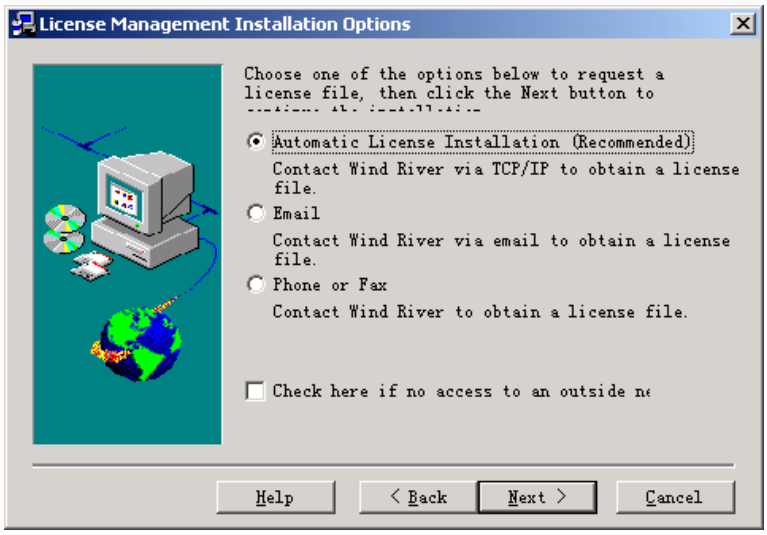

图 5-13　“License management Installation Options”窗口

- Automatic License Installation　此选项表示安装主机通过 TCP/IP 连接从产品开发商 Wind River 直接获取 License 文件，这属于自动安装 License 并完成注册的情况，这是系统推荐的方式。
- Email　此选项表示开发主机通过 Email 与产品开发商 Wind River 联系获取 License 文件，实际上，License 安装程序会将用户的注册信息和主机 ID 整理成一个电子邮件直接发送到产品开发商 Wind River 公司。
- Phone or Fax　此选项表示通过电话或者传真与产品开发商 Wind River 直接联系获取 License 文件。

- Check here if access to an outside network 表示如果安装主机不能与外部网络直接连接选择的选项。

5．如果选择第 4 步中的“Automatic License Installation”，那么安装程序将显示如图 5-14 所示的“DataBase Query Permission”信息窗口。这个窗口显示了安装主机的 Host Name 和 Host ID。此外还可以在窗口中选择“Use Disk Serial Number for host Id”即选择磁盘的卷标来表示主机。可以根据图示信息直接与 Wind River 的网站联系获取 License 注册文件，完成注册过程。

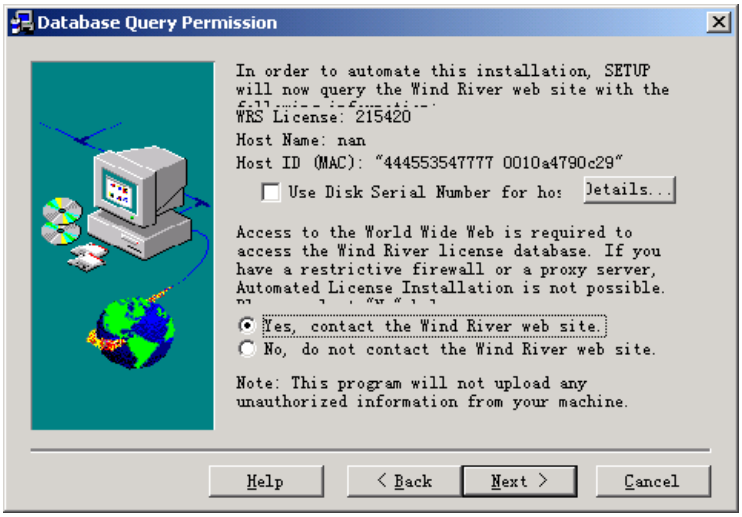

图 5-14 “DataBase Query Permission”窗口

6．如果选择第 4 步中的“Email”来获取 License 文件，那么安装程序将显示如图 5-15 所示的“Email License Configuration”信息窗口。此窗口中显示的是产品的 ID 值，我们需要确认对 License 的获取，然后单击此窗口中的“Next”按钮继续。

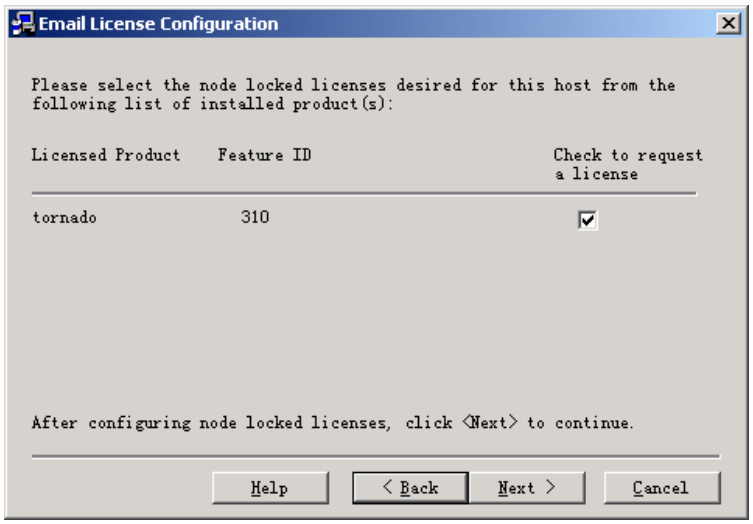

图 5-15 “Email License Configuration”信息窗口

7．将进入如图 5-16 所示“Email License”窗口。本窗口列出了如下信息：

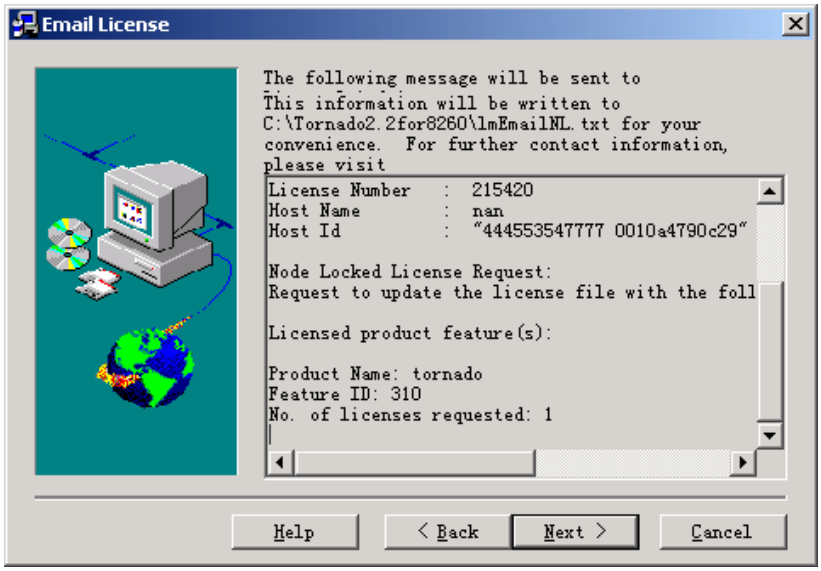

图 5-16　“Email License”窗口

- Customer Name
- Company Name
- Installation Key
- License Number
- Host Name
- Host Id

这些信息基本上代表了将要发送的邮件信息。邮件的摘要信息如下：

- Node Locked License Request（邮件的题头）
- Licensed product feature(s)
- Product Name
- Feature ID
- No. of Licenses requested

如果确认了邮件信息可以单击窗口中“Next”按钮继续下一步。

8．这时，安装程序给出如图 5-17 所示的邮件信息（Mail Information）提示窗口。在这个窗口中显示 WindRiver 接收邮件的地址，同时必须键入自己的邮件地址，以便接收到 License 文件。邮件的主题不用更改，但是可以根据具体情况更改自己使用的 Mail Protocol（邮件协议）。然后单击此窗口的“Send”按钮发送邮件。这样基本上就完成注册信息的发送。此时必须等待对端的 License 文件回应。当得到对方回应的 License 文件以后，将其安装到路径 tornado2.2\.wind\license\下便完成了整个注册过程。

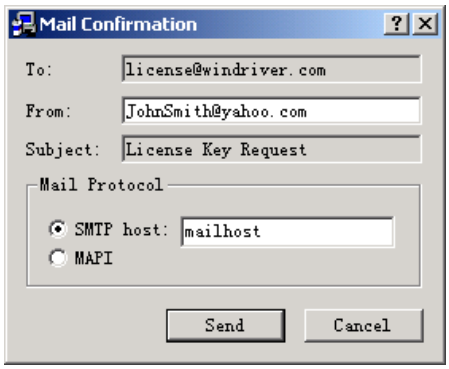

图 5-17　邮件信息（Mail Information）提示窗口

9．如果选择第 4 步中的“Phone or Fax”来获取 License 文件，那么安装程序将显示图 5-15 所示的“Email License Configuration”信息窗口。单击“Next”按钮便出现如图 5-18 所示的“Contact Wind River”窗口。根据此窗口的信息可以直接通过电话得到 License 文件信息，亦可以通过传真获得 License 文件信息。将 License 文件信息命令为 WRSLicense.lic 然后安装到路径 tornado2.2\.wind\license\下便完成了整个注册过程。

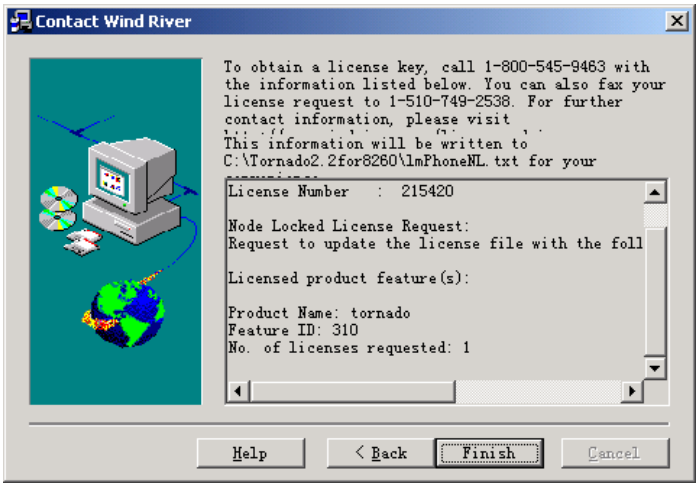

图 5-18　“Contact Wind River”窗口

5.3　初步使用 Tornado 环境

在 VxWorks 系统的 Tornado 开发环境中，可以通过创建工作区（Workspace）并在其中组建工程（Project）来启动自己的应用开发过程。在使用 Tornado 开发环境开发应用时，除了用到它的工程管理器外，还经常使用 VxWorks Simulator 来模拟目标系统进行调试。另外 Tornado 环境的 Shell 是一个很好的命令行执行工具，通过 Browser 可以观察目标系统内存的使用、任务的状态等。Tornado 环境中的 WindView 是一个软件逻辑分析仪，利用它可以看到系统中程序执行流程的图形示意。Tornado 环境的调试器（Ddebugger）非常适合调试实时

应用程序。

一般来说，调试一个实时应用程序往往需要编辑、编译连接、下载以及运行等好几个阶段的反复操作。值得注意的是，完成实时应用程序在 VxWorks Simulator 上的调试并不等于在目标系统中调试通过。在目标系统中调试需要较长的时间。

5.3.1　Tornado 工程的类型

1. bootable VxWorks image 型工程

这种工程是可下载的 VxWorks 映像，包含了必需的 VxWorks 内核、组件以及 BSP。在它的文件列表中，固定地包含了这样的一些文件：linkSyms.c、prjConfig.c、romInit.s、romStart.c、sysAlib.s、sysLib.c、usrAppInit.c。

这里，特别需要注意的是 usrAppInit.c 文件，它的函数 usrAppInit 是应用的入口点，可以在其中加入应用工程的主程序入口函数，同时也必须在文件头包含入口函数的原型说明和相关的头文件。当然，在工程编译之前，必须重新确定工程文件的依赖关系(Dependency)。这是一种生成 bootable VxWorks Application 的方法即静态包含。还有一种更简洁的方法叫做静态链接，即在函数 usrAppInit 中加入应用工程的主程序入口函数，然后静态链接已编译好的应用工程模块。这两种方法均与 VxWorks 系统的推荐方法不同。与 pSOS 的 release 版本相比，这里的 bootable VxWorks Application 可以认为是应用工程的 release 版本。

bootable VxWorks image 型工程是可配置的。配置的方法相当简便：单击工作台下部的 VxWorks 选项，然后单击该工程，展开它的组件列表。在组件列表中可以增加或删除组件，而单击右键可选择特性（Properities）项对相关组件内部的一些宏进行赋值，诸如 USER_RESERVED_MEM、LOCAL_MEM_SIZE 等。当然，在配置之前可先选择自动扩展(Auto Scale)功能，该功能将配置中没有直接调用的组件以绿色标出，但这些组件可能会被用户的应用间接调用。所以，应谨慎删除组件的操作。VxWorks 在开发者增加或删除组件时会把组件的尺寸和映像改变的大小列出，还会把它与其他组件的依赖关系一并列出。需要特别注意，防止因删除了一个组件而导致其他的组件不可用。

开发者可以修改编译规则，方法是单击工作台下部的 Builds 选项，然后单击该工程，展开它的列表。选择一个 Builds 项，右键单击选择 Properties，在弹出的对话框中选择和编辑编译规则。开发者也可以往列表中加入新的编译项目，右键单击选择 New Build 即可。这样，开发者可以使一个工程拥有几种编译规则。

除了 Tornado 自身提供的、应用于仿真器 VxSim 的工程 simpc_vx 外，所有的工程，包括系统提供的，均编译为 VxWorks 系统文件。而工程 simpc_vx 则可编译为 VxWorks.exe 文件，可以运行在 VxSim 上。

2. downloadable application modules 型工程

这种工程可编译为可下载的应用模块。仔细察看编译规则中的 rules 项，可看到 3 种类型：工程.out——编译目标文件并把它们链接而形成的文件；archive——编译目标文件并把它们放入一个名为工程.a 的文件中；objects——编译形成多个目标文件。这 3 种规则形成的文件各有用途。

应用模块可以和 VxWorks 映像静态链接一起下载，也可以当 VxWorks 映像在目标板中运

行时，动态下载链接到映像中，再通过调试器激活而运行。此工程并没有可配置的操作系统组件，仅仅只是应用。除此之外，与 bootable VxWorks image 型工程类似。

3．应用模块和 VxWorks 映像的静态链接和动态下载

在静态链接之前，应用模块应以.o 文件或.a 文件的形式存在。在 VxWorks 映像编译规则的 Macros 项中，找到 EXTRA_MODULES，令它的值为绝对路径+应用模块名(.o 文件或.a 文件)，如果要包含多个应用模块，每个模块之间用空格隔开即可。最后重新编译 VxWorks 映像工程。此为静态链接。

至于动态下载，则是先用 FTP 下载 VxWorks 映像，在 VxWorks 映像跑起来后，再通过 Tornad。集成开发环境下载应用模块，并激活运行应用模块。这里要下载的应用模块，推荐使用.out 格式。当然，.o 格式也是可以下载的，但是不推荐采用此种格式。动态下载支持在系统调试不同的时期下载应用模块，可以渐增式地调试应用模块。这对于多模块系统的调试有重要的意义。

动态下载和卸载应用模块有如下 3 种途径：

- 在要下载的.out 或.o 文件上单击右键，可以选择 LOAD 和 UNLOAD 来下载和卸载。
- 在 Shell 中使用命令 ld 和 unld 来下载和卸载。
- 在 LOOK!中使用菜单方式 LOAD 和 UNLOAD 来下载和卸载。

4．VxWorks 工程的 makefile 文件

VxWorks 工程在其工程目录下有一个 makefile 文件，用于编译和连接每个工程文件。每当创建一个 VxWorks 工程时，都会创建一个 makefile 文件，并且除了文件的 Dependency 外，工程文件的增减和编译以及工程编译项的修改将自动更新 makefile。而每一次的文件 Dependency 操作也将更新 makefile 文件中的## dependencies 项。与 pSOS 相比，VxWorks 工程的 makefile 是自动创建和修改的，也有少量的用户项（## user defined rules），这样，用户需要手工处理的工作量大大降低。

5.3.2　启动 Tornado 环境

为了开发和调试 VxWorks 系统的实时应用程序首先需要启动 Tornado 开发环境。在安装好的 Windows 环境中，通过单击“开始”菜单、程序菜单中的“Tornado2.2”以及“Tornado”即可以启动 Tornado 开发环境。当然最快的启动方式是直接单击桌面的 Tornado 快捷方式。

Tornado 开发环境在启动时一般会尝试注册，注册通过才可以使用，否则它将引导用户进入注册流程。为了保证系统启动时正常注册，可以在 Tornado 开发环境启动前检查系统是否已经启动了 Tornado Registry。

刚刚启动的 Tornado 开发环境的界面如图 5-19 所示。

Tornado 开发环境的启动界面中包含一个“创建工程”小窗口，同时包含大量工具的快捷按钮。利用“创建工程”小窗口可以创建和打开现有的工程。单击工具的快捷按钮可以直接启动 Tornado 开发环境的内嵌工具组件。

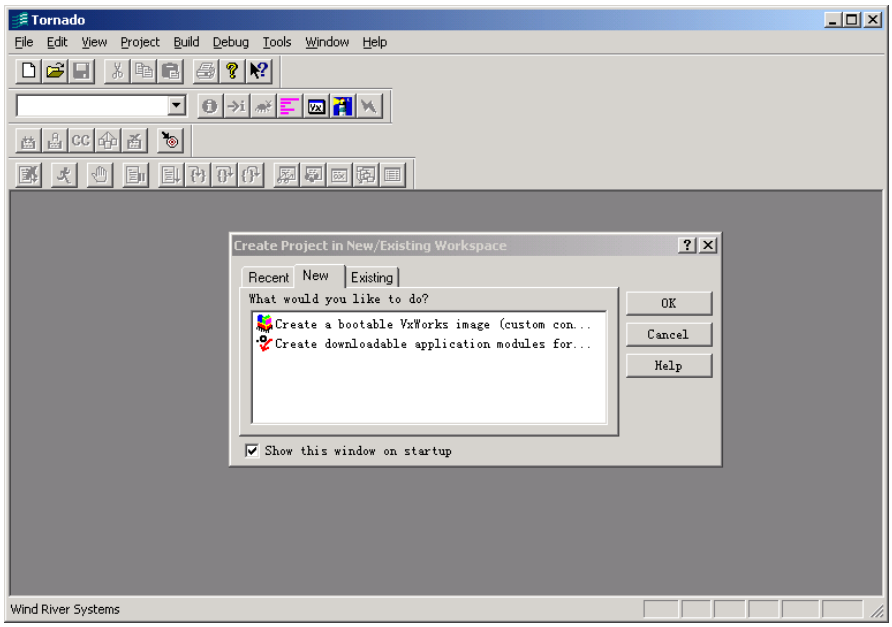

图 5-19　Tornado 开发环境的界面

5.3.3　创建工作区和工程

每个工程都必须在某个工作台下才能进行诸如增加、删除文件，编译等工作。但工程可脱离工作台存在。开发者首先必须创建工作台，然后在工作台上创建工程，或向工作台添加工程。当然，工程也可从工作台上移去，但其文件仍然在磁盘目录上，并没有删除。

由于创建一个工作台必然要创建一个工程，而在打开的工作台上也可以创建工程，并且这两种方式的步骤是一样的。所以这里只叙述第一种方式。步骤是：

1. 如果系统没有出现“Create Project in New/Exsting Workspace”小窗口，那么第一步是打开文件（File）菜单，然后选择 Open Workspace。

2. 在弹出的 Open Workspace 对话框中选择 New，将出现如图 5-20 所示的两种类型工程，bootable VxWorks image 和 downloadable application modules。第一种表示创建可引导映像，第二种表示创建可下载的应用程序模块，两者的区别在于第一种工程包含了 VxWorks 内核映像，而第二种工程仅仅包含应用程序模块。选择其中一种，单击 OK 按钮。以下的步骤对两种工程来说大同小异。这里以创建 bootable VxWorks image 为例介绍。

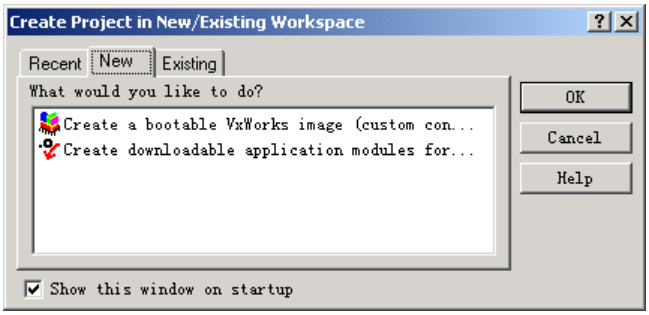

图 5-20　Tornado 开发环境的“Create Project in New/Exsting Workspace”窗口

3．如图 5-21 所示，在弹出的 Create a bootable VxWorks image (custom configured): step 1 对话框中，填入工程的名称和路径，工程的名称和路径上的名称最好一样，再填入新的工作区路径和名称。如果仅仅是创建工程，则可以选择工程要加入的工作区。最后单击“Next”按钮。

4．如图 5-22 所示，在弹出的 Create a bootable VxWorks image (custom configured): step 2 对话框中，可选择新工程是否基于一个已经存在的工程还是基于一个 BSP，其中 BSP 集中包含在目录\Tornado\target\config\下。由于不必从 BSP 生成配置信息，基于一个工程将会使新工程更快地生成，而新工程将参考原工程的源文件。新工程对源文件所做的修改将在原工程上出现，但配置的修改并不互相影响。选择基于现有工程，然后单击“Next”按钮进行下一步。

5．如图 5-23 所示，在弹出的 Create a bootable VxWorks image (custom configured): step 3 对话框中，列出了工作区、工程、工程基于已经存在的工程或 BSP 的信息，如果还需要修改，单击 Back 返回修改，否则单击“Finish”按钮。这样，新工作区和新工程就创建了。

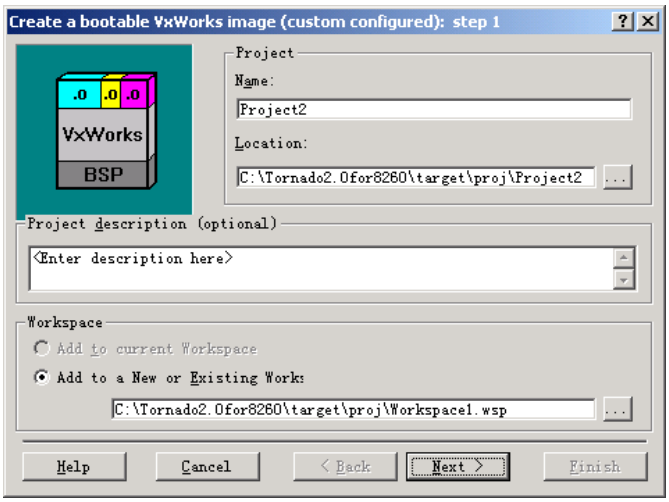

图 5-21　Tornado 开发环境中创建可引导映像的第一步

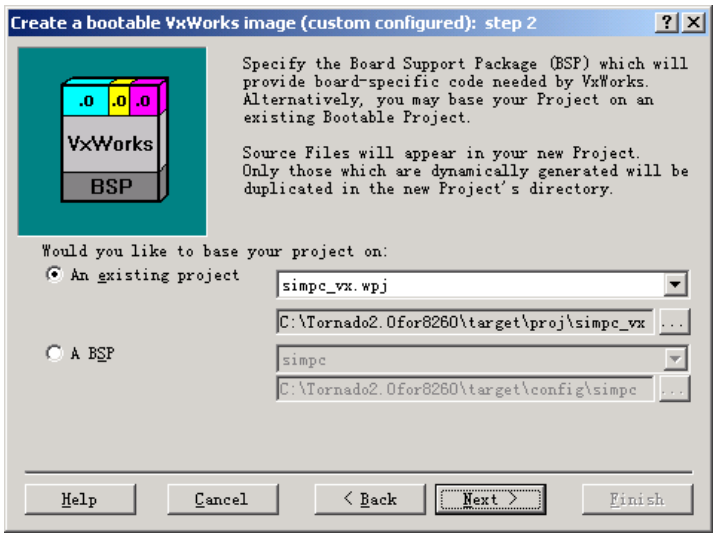

图 5-22　Tornado 开发环境中创建可引导映像的第二步

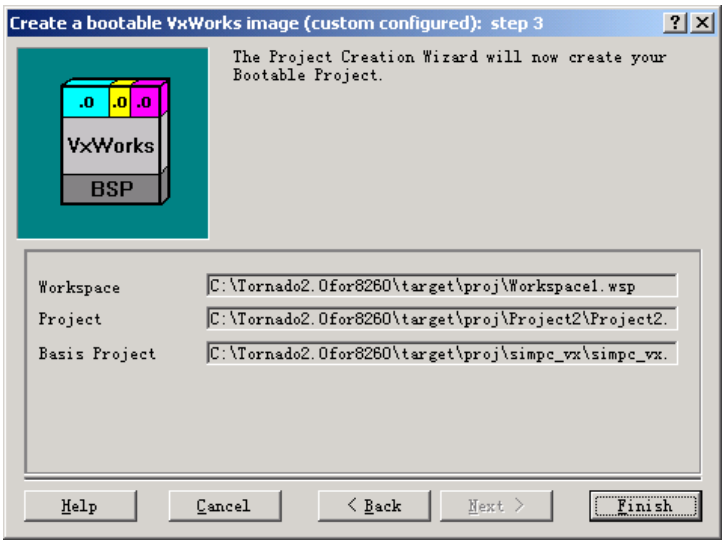

图 5-23　Tornado 开发环境中创建可引导映像的第三步

6．最后就出现了工作区窗口，如图 5-24 所示，工作区窗口中包含了工作区的名字以及包含的文件。基于此工作区窗口，便可以进行应用开发工作。

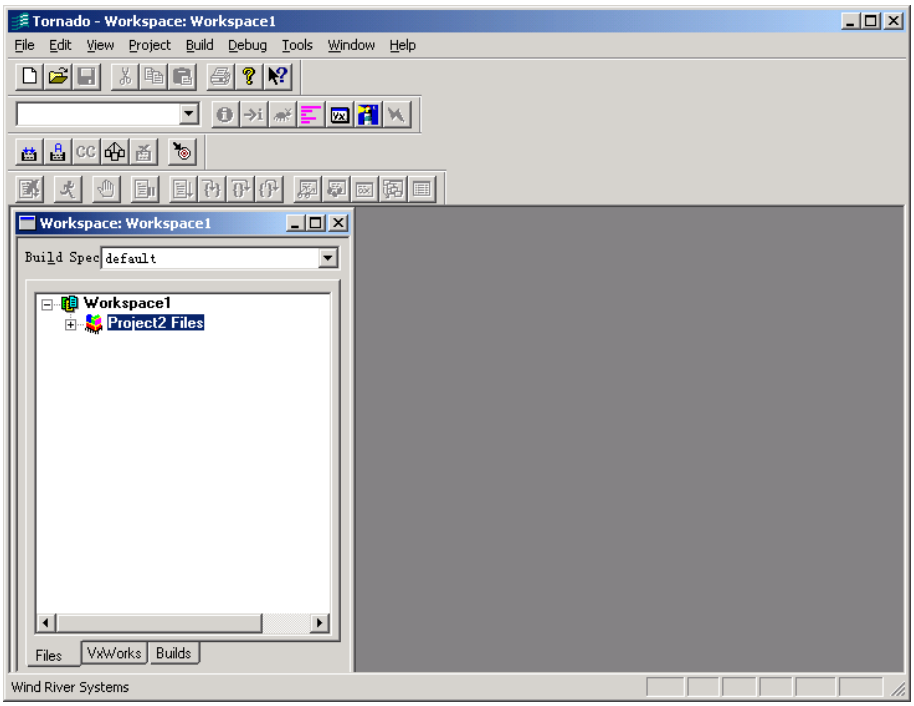

图 5-24　Tornado 开发环境的工作区窗口

在打开的工作区上创建工程时，需要在文件菜单上选择 New Project，以后的步骤与上面介绍的步骤类似。

可以在工作区创建若干个工程，这些工程之间可以共享源代码。利用工作区包含工程是组织具体开发的好方式。在大型应用项目开发过程中，许多工程师参与代码的编写和调试，初期阶段可以让每个人建立一个工程，最后将所有人开发的工程整合到一个大的工作区中。

5.3.4　添加文件到工作区和工程

在图 5-24 中，整个 Tornado 开发环境窗体界面从上到下依次是菜单栏、标准工具栏、开发工具栏、编译工具栏、调试工具栏、用户工作视图，屏幕最底端是一个注释条。开发工具栏拥有 8 个开发工具图标：Browser、Shell、Debuger、CodeTest、Look!、Wind Navigator 和 Trigger。

工作区是用户工作视图中的一个窗体，上部是被选中工程的编译规则说明，中部是工作区工程列表，下部是被选中工程的 3 个按钮。第一个是 Files，此时在工程列表中单击工程，可得到工程的文件列表；第二个是 VxWorks，只有 bootable VxWorks image 工程才有此项，此时在工程列表中单击工程，可得到工程组件列表，在这里可进行系统的配置；第三个是 Builds，说明工程的编译规则。

这里将介绍如何在工作区的工程中添加简单的程序文件。要添加的程序文件是 TaskMSG.c，它是一个简单的多任务应用程序，可以模拟一个消息收集系统，在第 7.2 节可以看到此文件的源代码。此消息收集系统中，一个任务充当消息源，一个充当消息接收者。这

两个任务是：

- sendMSG_Task 发送消息任务
- recvMSG_Task 接收消息任务

可以直接从 Tornado 开发环境的 File 菜单单击“New”项进入如图 5-25 所示的程序文件建立对话框。在此对话框中可以指明要建立程序文件的类型、文件名称、文件所在的工程以及文件在磁盘上的具体位置，然后单击 OK 按钮进入 Tornado 开发环境的程序编辑窗口，开始程序开发。

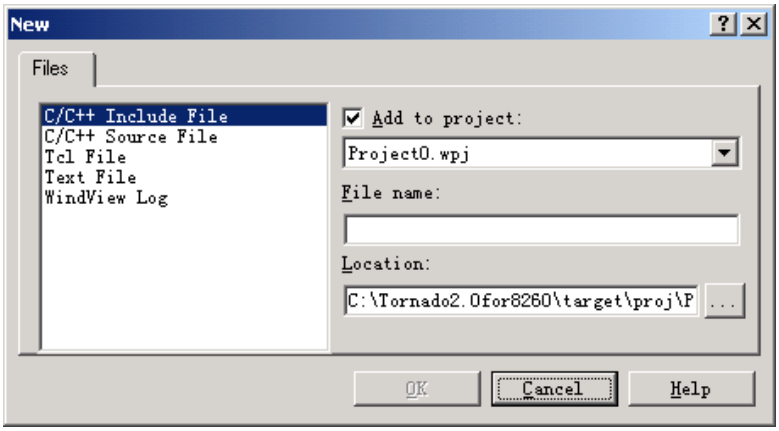

图 5-25　程序文件的建立对话框

可建立的程序文件类型有如下几种：

- C/C++ Include File，C/C++头文件。
- C/C++ Source File，C/C++源文件。
- Tcl File，Tcl 格式文件。
- Text File，文本文件。
- WindView Log，WindView 的日志文件。

有些情况下，我们已经在创建工程以前编写了程序文件。这时候可以直接在工作区窗口将已经存在的程序文件添加到工程中。例如，可以按以下步骤在 Workspace0 的 Project0 中添加程序文件 TaskMSG.c。

1．首先用鼠标右击 Workspcae0 工作区内的 Project0 选项，将出现如图 5-26 所示的浮动菜单，在这个菜单中选择“Add Files”。

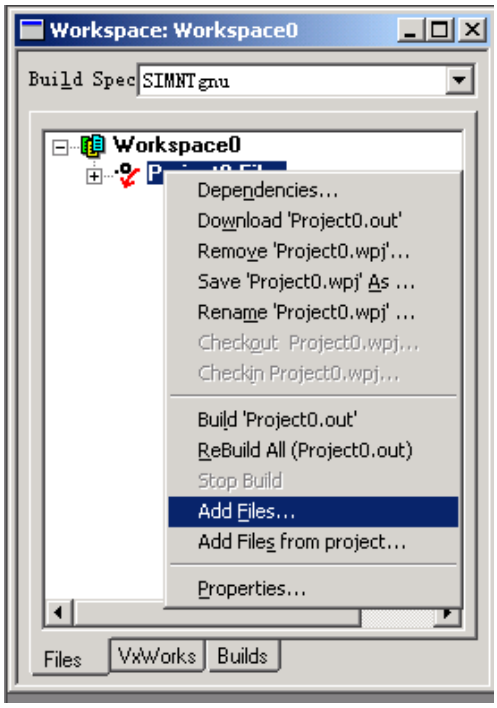

图 5-26　添加文件到工程中

2. 在出现的文件选择对话框中找到文件 TaskMSG.c，单击“Add”按钮添加到 Project0 中。

3. 在工作区的 Files 选项卡中可以看到已经添加的文件 TaskMSG.c，如图 5-27 所示。而且可以看到文件 TaskMSG.c。编译后的目标文件 TaskMSG.o。其中 Project0.out 表示整个工程编译以后的可执行二进制文件。

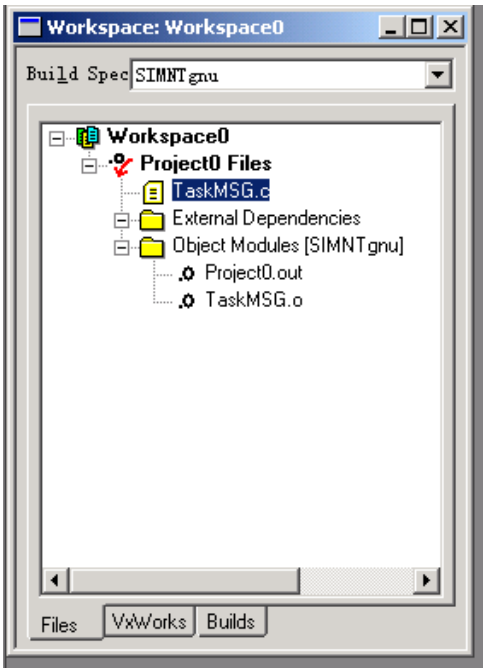

图 5-27　已经添加了文件的工程

5.3.5　编译工程

这里所说的编译不是通常的 Compile 的概念，指的是 Build 的概念，它包括了编译、汇编、连接等几个阶段的工作。开发者可以把自己编写的所有文件依次添加到工程中，使用菜单命令 Dependencies，自动计算改变文件间的依赖关系（Dependency），利用系统的自动扩展功能（Auto Scale）得到操作系统组件的使用信息，这样就可以增加删除组件进行系统配置。最后依据编译规则，对工程进行编译。

在编译工程以前，可以浏览以下工程的编译规则。首先选择工作区中 Builds 选项卡，然后单击其中的 SIMNT gnu，这样将弹出 SIMNT gnu 的属性窗口，如图 5-28 所示。这个属性窗口列出了 SIMNT gnu 的若干特性如编译规则、连接器、连接次序、汇编器、编译宏定义、C/C++编译器。我们在图 5-28 中选择“C/C++ Compiler”（C/C++编译器）选项卡即可以看到其中的 GNU 编译选项。

我们发现在 C/C++编译器选项卡中有一个确认框“Include debug info”，默认情况这个确认框是选中的，这表示在默认情况下 GNU 编译器在编译 Tornado 的工程时不使用优化选项，即并不屏蔽调试信息的输出。确认 SIMNT gnu 的属性窗口中的属性无误后，单击 SIMNT gnu 的属性窗口的“Cannel”按钮即可关闭此窗口。

编译工作区的工程时，只需右击 SIMNT gnu，然后在如图 5-29 所示的浮动菜单中选择“Build Project0”编译工程 Project0。如果是第一次编译一个源文件，Tornado 集成开发环境会弹出如图 5-30 所示的文件依赖关系（dependencies）对话框。

图 5-30 所示的文件依赖关系(dependencies)对话框中列出了需要计算依赖关系的文件，在我们创建的简单工程中，由于只有一个文件，所以图 5-30 中只列出了 TaskMSG.c。

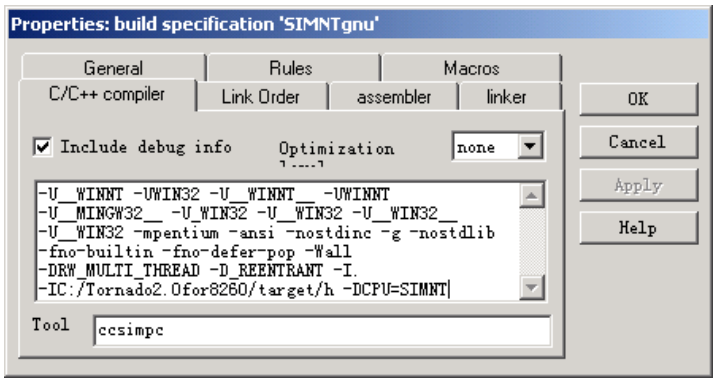

图 5-28　SIMNT gnu 的属性窗口

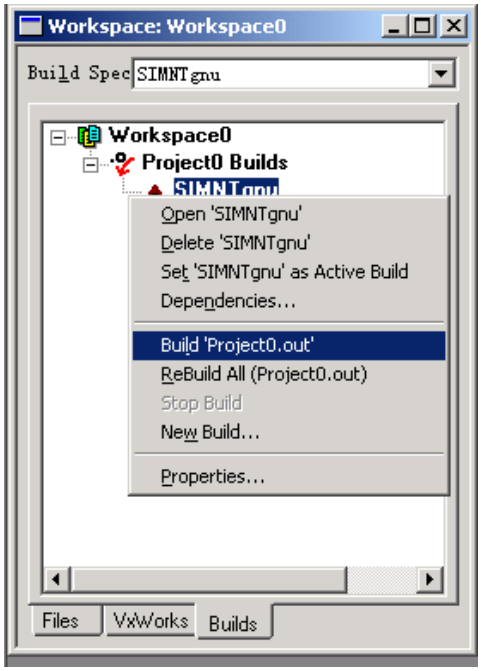

图 5-29　在 Tornado 环境中编译工程

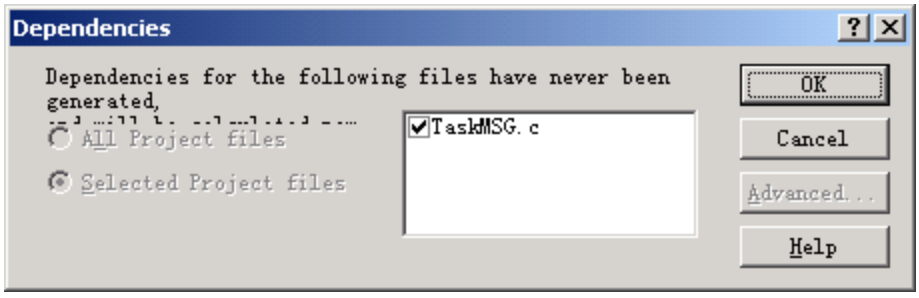

图 5-30　工程文件的依赖关系对话框

在图 5-30 中单击“OK”按钮，这样 Tornado 开发环境的编译器开始计算 Makefile 的依赖关系并开始编译程序。如果编译过程发现了外部依赖关系，它们将自动添加到工程中并列

出在工作区“Files”选项卡 的“External Dependencies”选项中。“External Dependencies”选项在工程文件中表现为一个文件夹。

编译结束以后，Tornado 开发环境在 Build Output 窗口显示编译出错信息和成功信息。例如，我们编译 TaskMSG.c 以后，Build Output 窗口输出的信息如图 5-31 所示。

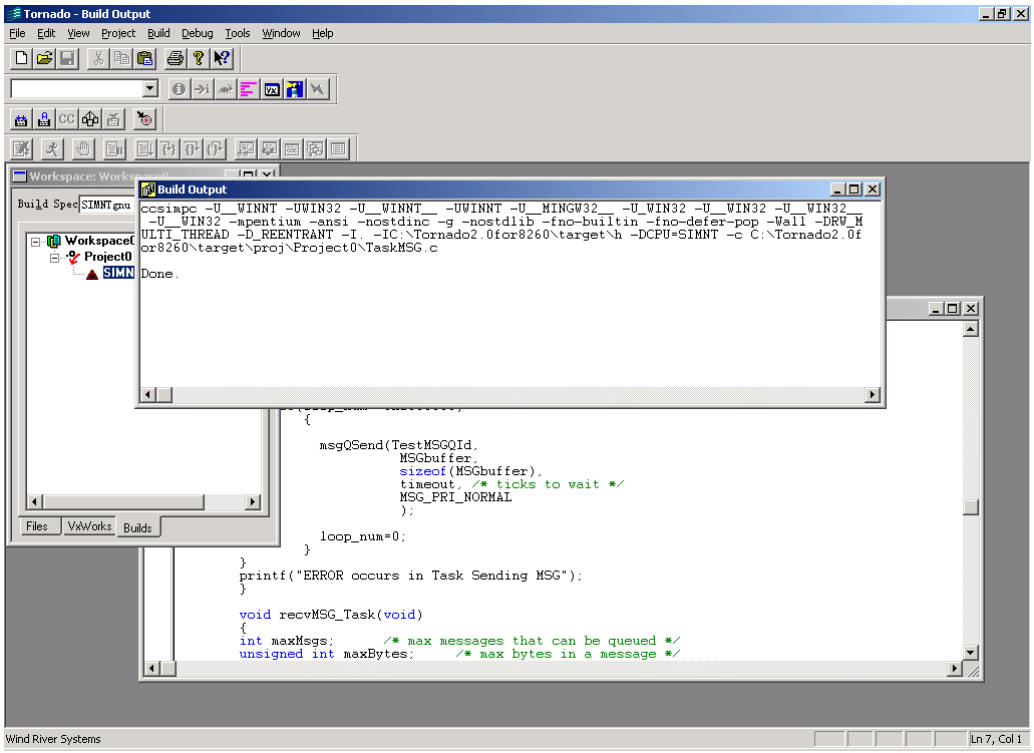

图 5-31　Tornado 开发环境的 Build Output 窗口

5.3.6　下载工程到 VxWorks 目标模拟器

前面几节已经介绍了如何创建工程，如何在工程添加文件以及如何编译工程文件，本节将介绍如何将编译好的工程文件（.out 文件）下载 VxWorks 目标模拟器（Target Simulator）中。将工程文件下载到 VxWorks 目标系统中时，可以采用完全一样的方法，唯一的不同是不需要启动 VxWorks 目标模拟器。具体下载需要以下几个步骤：

1．首先在工程文件所在工作区中找到要下载的工程文件 Project0.out，右击此工程文件，即可出现如图 5-32 所示的浮动菜单，在此菜单中选择“Download 'Project0.out'”。

2．如图 5-33 所示，这时系统将提示是否启动 VxWorks 目标模拟器。系统给出此提示的原因是，我们在前面创建工程时，选择工程基于的 ToolChain（工具链）是 SIMNTgnu。如果我们的选择的是一个针对具体目标板处理器的 ToolChain 如 PPC860gnu，那么系统将不再提示启动 VxWorks 目标模拟器，因为这种情况下，我们的工程文件将直接下载到目标系统中。单击此提示框中的“是（Y）”按钮。

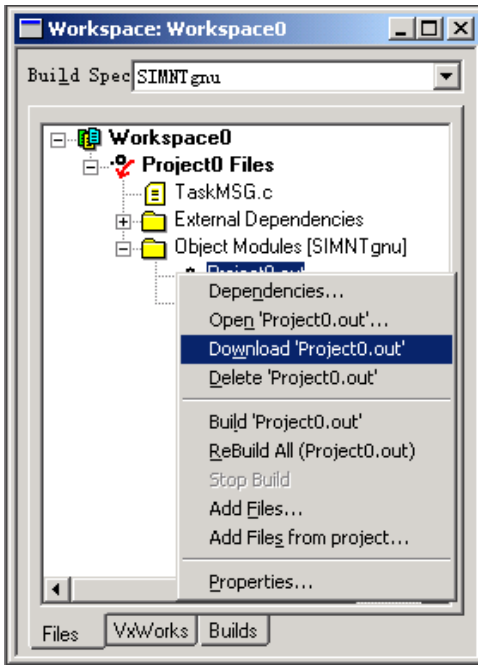

图 5-32　下载工程文件到目标系统中

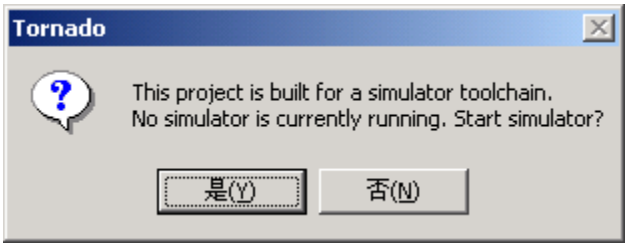

图 5-33　启动 VxWorks 目标模拟器的提示信息

3．这时开发环境将弹出如图 5-34 所示的对话框。此对话框显示出了 VxWorks 目标模拟器的可配置选项。首先可以选择 VxWorks 目标模拟器基于的 VxWorks 系统的内核映像。有如下两种选择：

- Standard Simulator，标准目标模拟器。
- Customer－build Simulator，用户自定义的目标模拟器。

一般来说，使用标准目标模拟器，这时使用的 VxWorks 系统内核映像位于默认路径下。

另外还可以选择下载到哪个处理器上进行调试和运行，这个选项只有在多处理器环境才可以配置。在单处理器环境，此选项为默认值 0。单击“OK”按钮。

4．此时我们将看到如图 5-35 所示的“VxWorks Simulator for Windows”窗口。当出现此窗口表示 VxWorks Simulator 已经启动成功。此窗口主要用于打印程序的输出信息。

5．VxWorks Simulator 已经启动成功以后，系统还会给出如图 5-36 所示的提示信息。它表示 Tornado 开发环境启动 VxWorks Simulator 时，必须绑定一个 Target Server。

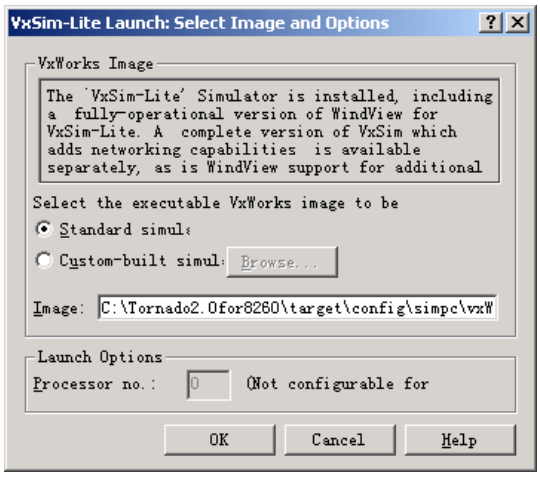

图 5-34　启动 VxWorks 目标模拟器的选项配置

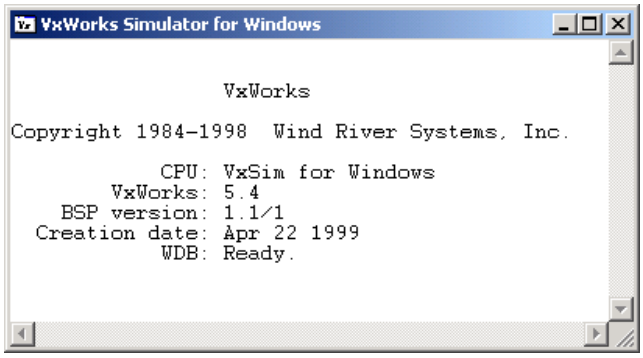

图 5-35　“VxWorks Simulator for Windows”窗口

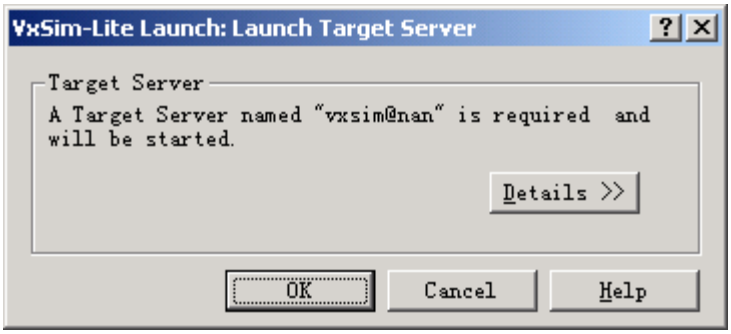

图 5-36　VxWorks 系统 Target Server 提示信息框

一般情况下，Tornado 开发环境启动的 Target Server 名为 VxSim@HostName，HostName 表示 Tornado 开发环境所在机器的名字，例如作者机器上启动的 Target Server 的默认名称是 VxSim@nan。

单击图 5-36 所示提示框的“Details”按钮可以得知 Target Server 在磁盘上的具体位置。确认启动 Target Server 以后，单击提示框中的“OK”按钮即可启动 Target Server。

值得注意的是，在 Tornado 集成开发环境中，我们一次只能启动一个 VxWorks Simulator。

因此在启动下一个 VxWorks Simulator 之前首先必须关闭当前 VxWorks Simulator，关闭的方法就是关闭 VxWorks Simulator 窗口。在 Tornado 集成开发环境中调试已经下载的工程文件时，如果 VxWorks Simulator 已经关闭，那么 VxWorks 系统 Target Server 也无法正常工作，只能选择退出。

5.3.7 在 Tornado Shell 下运行应用程序

在运行应用程序之前,有必要配置和启动 Tornado 集成开发环境中的 Debugger(调试器),以便让调试器可以自动响应应用程序中的所有异常。

Tornado 集成开发环境中的 Debugger（调试器）CrossWind 具备命令行和图形调试接口的双重功能。例如，我们可以通过点击轻松设置断点、控制程序执行流。此外 Tornado 集成开发环境中 Debugger（调试器）使用的程序列表与其编辑窗口相同，这样用户在调试程序时可以现场修改。Debugger（调试器）提供了方便的变量观察窗口、本地变量列表窗口和寄存器观察窗口。

为了配置 Debugger（调试器），选择 Tornado 集成开发环境的 Tools 菜单，找到 Options—>Debugger，单击此选项便可以出现如图 5-37 所示的 options 窗口。options 窗口中选择 Debugger 选项卡，这时我们到的是 Tornado 开发环境 Debugger 的全部属性选项。我们只需要配置“Auto Attach to Tasks”组合框，即选择“Always”，这样 Debugger 就会自动与系统启动的任务关联。然后单击此窗口的“OK”确认属性配置。

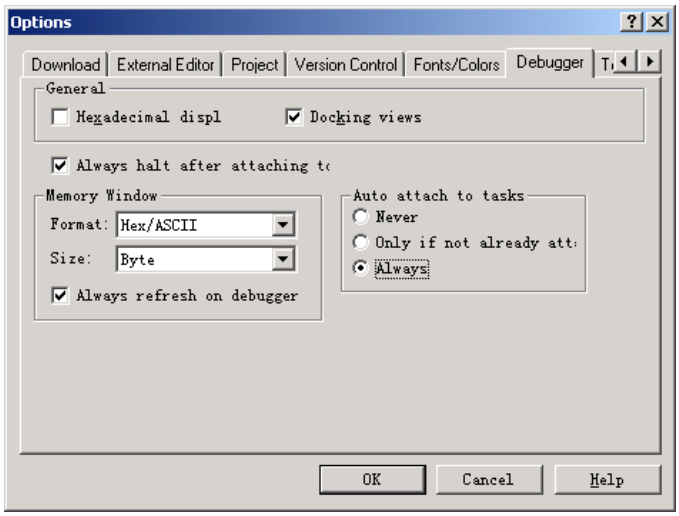

图 5-37 Tornado 环境 Debugger 的 Options 窗口

配置了 Debugger 的属性以后，可以单击选择 Tornado 集成开发环境 Tools 菜单中的 Debugger 选项启动 Debugger。更快的启动方式是直接点击形如乌龟状的快捷图标。刚启动时由于系统尚未启动任何任务，因此 Tornado 开发环境的状态栏会显示 Debugger 的状态为 Unattached，表示 Debugger 没有与任何任务关联。

Tornado 集成开发环境的 Shell 叫做 WindSh，它是一个 C 语言命令解释程序。我们可以在 Shell 上激活下载到目标板或者目标模拟器上的任何程序。Shell 也包含了它自己的一套

命令，这些命令可以用于管理任务、访问系统信息、进行调试等。可以单击选择 Tornado 集成开发环境 Tools 菜单中的 Shell 选项启动 Shell。更快的启动方式是直接点击形如“->i”的快捷图标。

已经启动的 shell 窗口如图 5-38 所示。在这个窗口中要运行 project0 工程，只需要在 Shell 提示符下面键入 project0 工程的主函数 CreateTask，然后按 Enter 键即可以运行工程。

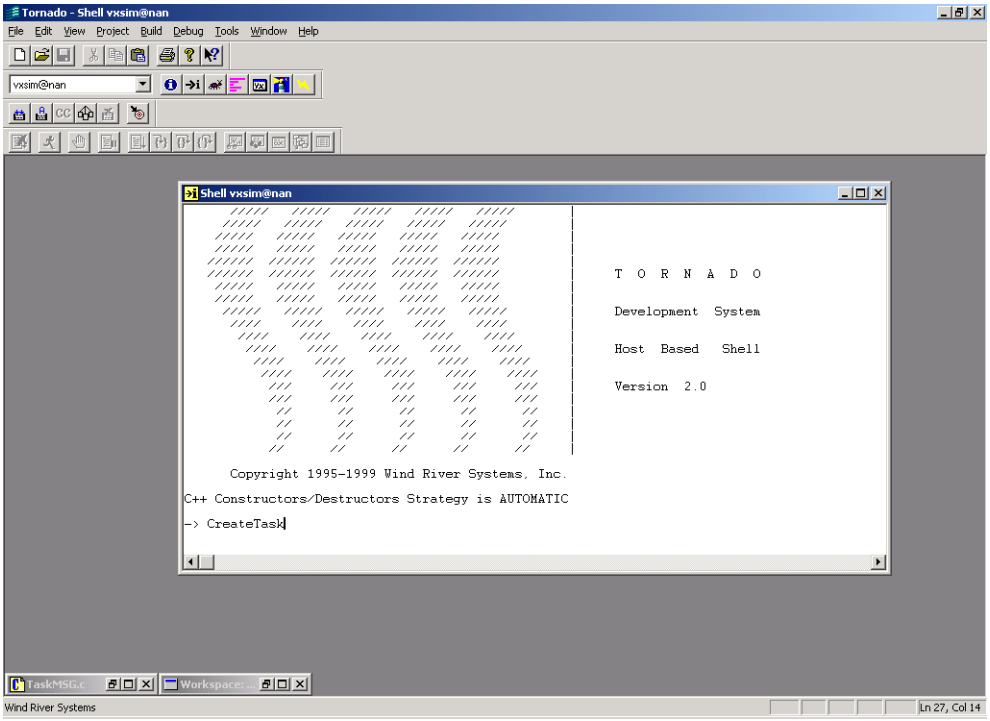

图 5-38　Tornado 集成开发环境及其 Shell 窗口

5.4　监视与调试

5.4.1　检查内存消耗

Tornado 集成开发环境的 Browser 是一个系统对象浏览器，被誉为 Tornado Shell 的图形伴侣。Browser 提供了监视目标系统状态的工具。利用 Browser 可以观察系统内活动的任务以及内存的使用等。在实际调试过程中，一般利用 Browser 检查内存的消耗。

为了启动 Browser，单击 Tornado 开发环境工具栏形如“i”的 Browser 快捷按钮。当 Browser 窗口出现时，从下拉菜单中选择“Memory Usage”并单击形如时钟的按钮即可看到如图 5-39 所示的内存消耗列表，内存消耗列表会实时刷新。

图 5-39 所示的内存消耗是前面所介绍的 TaskMSG.c 程序在 VxWorks Simulator 上运行时的情况。

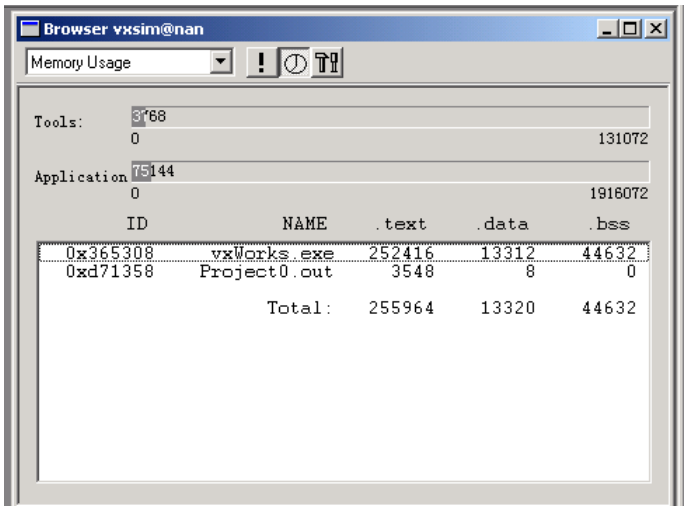

图 5-39　在 Browser 内检查内存的消耗

5.4.2　软件逻辑分析

WindView 是 Tornado 集成开发环境针对实时应用的逻辑分析仪。利用 WindView 可以动态观察到任务上下文的切换、事件的发生，也可以查看信号量、消息队列和看门狗时钟的实时状态。因此 WindView 是一个强有力的调试诊断工具。

要想启动 WindView，可以在 Tornado 集成开发环境中选择“Tools”菜单，单击其中 WindView->Launch，这时首先出现如图 5-40 所示的 WindView 的基本事件收集配置对话框。

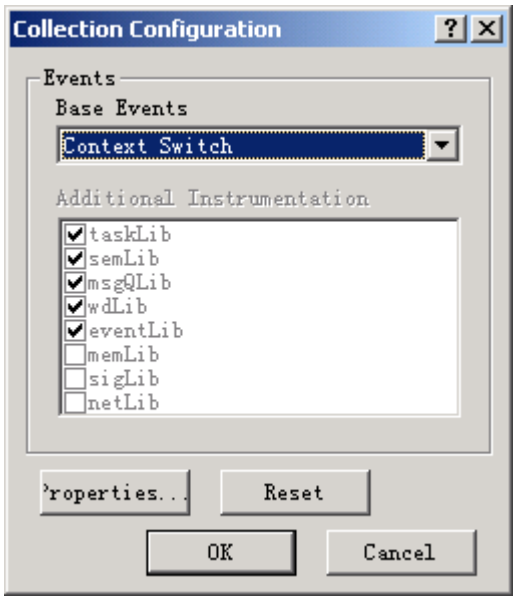

图 5-40　WindView 的基本事件收集配置对话框

在这个对话框的“Base Event”下拉菜单中可以选择如下事件：

- Context Switch，上下文切换

- Task State Transition，任务状态跃迁
- Additional Instrumentation，其他事件

在图 5-40 中，还可以选择修改收集事件的属性，如是否将收集信息直接显示到图形当中。默认情况下，收集信息将直接上传到图形文件中并显示。

在图 5-40 中选择“Additional Instrumentation”，然后单击“OK”按钮，即可看到如图 5-41 所示的 WindView 的控制窗口。这个窗口显示了目前要收集的事件信息。

要想开始收集事件信息仅需点击图 5-41 中的“Go”按钮。几秒钟以后，单击眼状按钮刷新收集的事件信息。

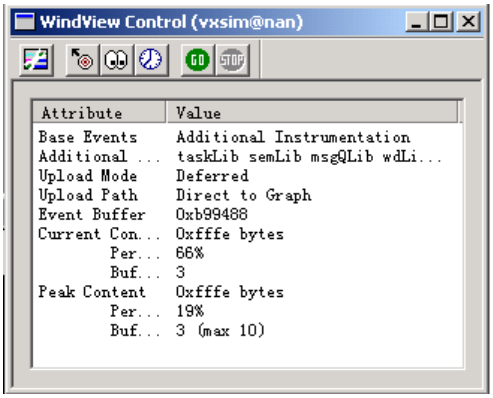

图 5-41　WindView 的控制窗口

当图 5-41 的当前事件信息的 Buffers 数目大于 2 以后，可以单击“Stop”按钮停止事件的收集工作。这时候我们发现运行 Tornado 开发环境的计算机的运行速度放慢，这是因为我们的实时应用实例程序发生了一系列事件。为了将刚才收集到的信息上传到计算机观察，首先要在 Shell 命令行上停止应用程序运行。

然后可以单击带箭头的上传按钮将事件数据上传到主机环境。这样在主机的 Tornado 开发环境上将显示事件发生的图形。

5.4.3　应用程序调试

用户在 Tornado 开发环境内开发 VxWorks 应用程序时，最关心的是如何在交叉开发环境中进行调试。Tornado 开发环境为我们提供了便捷的交叉开发和调试环境。前面我们已经介绍了如何启动 Debugger，在启动 Debugger 以后，可在 Tornado 环境提供的程序编辑窗口中直接设置应用程序的断点，进行调试。

Tornado 开发环境中设置断点的方法非常简单，将光标移动到某一行，按功能键 F9 即可以在此行设置断点。例如，在图 5-42 中在实例程序 TaskMSG.c 中的 msgSend 调用处设置了一个断点，断点的标志是窗口左侧出现向下指的箭头。

在 Tornado 开发环境中我们可以设置的断点种类如下：

- 普通断点，用于应用程序的任务级调试。
- 全局断点，用于应用程序的系统级调试。
- 临时断点，用于应用程序的临时调试。

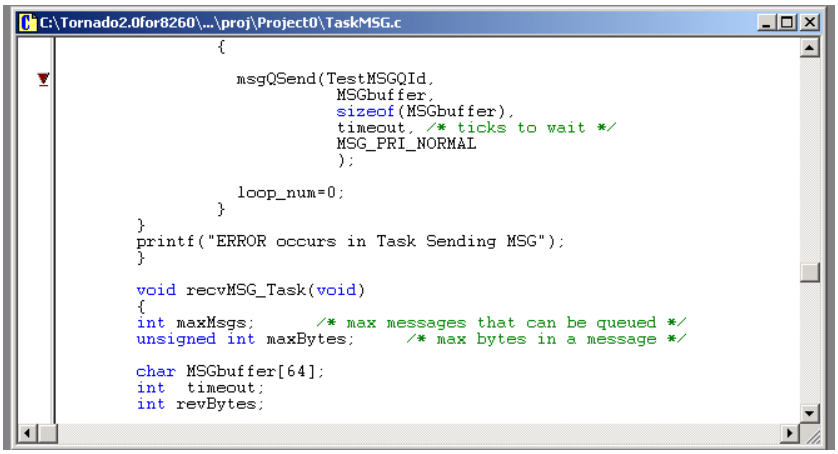

图 5-42　Tornado 环境的应用程序调试窗口

在 Tornado 环境调试应用程序时，我们有必要实时了解各个任务的运行状态和属性。在 Tornado 开发环境的 Shell 提示符下键入“i”命令然后按 Enter 键即可显示系统内各个任务的运行状态和属性。

针对我们的实例应用程序，在图 5-43 中，我们可以观察到系统任务 tExcTaskt 和 LogTask 处于挂起状态，系统调试代理 tWdbTask、send MSG 任务以及 recv MSG 任务均处于就绪状态。

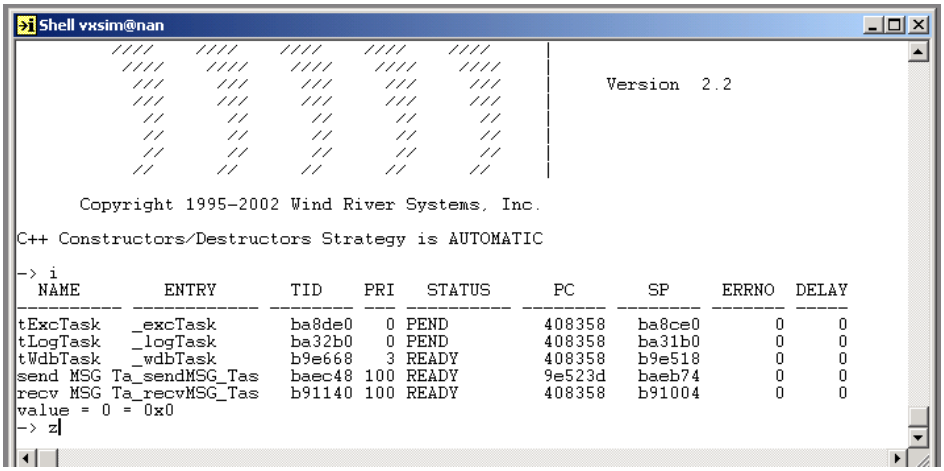

图 5-43　在 Shell 查看目标系统上的任务属性

在 Tornado 开发环境中如果借助于 VxWorks 的 Simulator（仿真器）来调试应用程序时，一般在 VxWorks Simulator 中观察输出结果，当运行我们的实例应用程序 TaskMSG.c 时，将观察到如图 5-44 所示的输出结果。

如果不借助于 VxWorks 的 Simulator（仿真器）调试应用，那么可以将应用程序映像下载到目标硬件系统中，这时应用程序的调试信息可以打印到 Tornado 开发环境自带的 FTP Server 上。用户可以直接从 Tornado 的程序菜单中选中 FTP Server 来启动它。

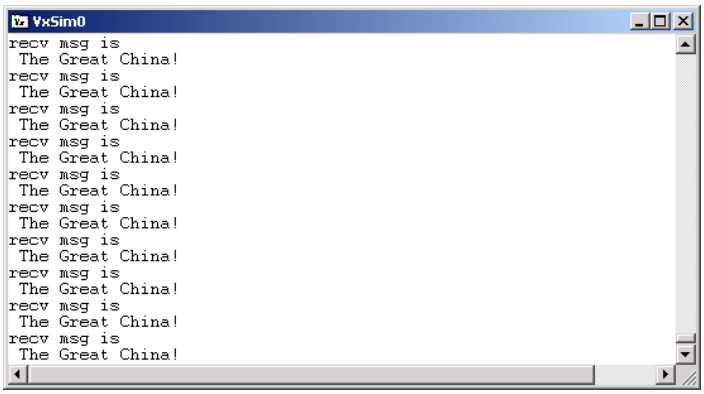

图 5-44　实例应用程序在 VxSim 上的输出结果

第 6 章　VxWorks 系统编译器

VxWorks 系统的交叉开发环境 Tornado 支持多种的编译器如 Gcc 和 Diab 编译器。其中 Gcc 是 VxWorks 默认的编译器，目前的最新版本是 2.96 版。

本章介绍 VxWorks 系统中控制软件编译过程的工具 make。make 可以自动管理软件编译的内容和方式。当使用 Gcc 或者 GNU C++开发多个源程序以后，可以借助于 make 管理项目的 Makefile 文件实现文件之间的连接和调用关系。实际开发过程中所有应用都是使用 Makefile 来实现编译和管理的。

6.1　Make 管理项目概述

当使用 GNU 中的编译语言如 Gcc、GNU C++编程开发应用程序时，绝大多数情况需要使用 make 管理项目。使用过 Borland C 的用户一定对编译源文件时输入冗长的命令很熟悉，如果不使用 make 管理项目和 Makefile，在 VxWorks 环境下编译多个源文件时要键入复杂的命令行。

make 管理项目通过把命令行保存到 Makefile 文件简化了编译工作。make 管理项目可以识别出 Makefile 中哪些文件已经修改，并且在再次编译时只编译这些文件，这样提高了编译的效率。make 管理项目还在数据库中维护当前开发工程中各个文件的依赖关系，在编译前就可以确定是否能找到所需文件。

要完成 make 管理项目的工作可以使用 Tornado 开发环境中的自动生成工具，也可以自己编写 Makefile。但是在创建大型系统工程时，往往需要用户自己维护 Makefile 文件，因此系统理解 Makefile 的细节是完全必要的。Makefile 是一个数据库文件，它以文本形式存储，其中包含的规则指明了 Make 命令怎样编译文件以及编译何种文件。一条规则包含 3 个方面的内容：make 要创建的目标文件（target），编译目标文件时需要的依赖文件列表（dependencies list），通过依赖文件创建目标文件的命令组（commands group）。此外，Makefile 的一个行中以#领头的部分全部是注释。

Make 命令在执行时按顺序查找名为 GNUmakefile、makefile 和 Makefile 文件进行编译。为了保持与 VxWorks 操作系统源代码开发的一致性，建议用户使用文件名 Makefile。一个简单的 Makefile 规则可以使用如下代码表示：

```
target:dependency file1  dependency file2[…]
      command1
      command2
      […]
```

上述规则中 target 是要创建的目标文件或者 VxWorks 系统支持格式的可执行文件。

dependency file*N*是创建 target 需要的依赖文件列表。command*N*是创建 target 时使用的命令组，是包括 make 命令在内的许多 Shell 命令的集合，每一个命令行的首字符必须是 Tab。默认情况下，当前目录就是 make 的工作目录。下面编写一个简单的 Makefile 实例，它是一个图形编辑器的 make 管理项目：

```
#a simple Makefile
painter: painter.o shape_lib.o op_lib.o
          gcc  -o painter painter.o shape_lib.o op_lib.o
painter.o : painter.c painter.h op_lib.h shape_lib.h
          gcc  -c painter.c
shape_lib.o :shape_lib.c shape_lib.h
        gcc  -c shape_lib.c
op_lib.o :op_lib.c op_lib.h
          gcc  -c op_lib.c
clean :
         rm edito *.o -
```

Makefile 文件编写好以后，在 Makefile 所在目录下键入 make 就可以编译 painter。在这个简单的示例中有好几条规则。第一条规则用于创建默认的目标 painter，它需要 3 个依赖文件即 painter.o、shape_lib.o 和 op_lib.o，编译时要用到它们。第二到四条规则是替 make 解释如何利用 Gcc 生成目标 painter 所依赖的文件 painter.o、shape_lib.o 和 op_lib.o。

读者可以发现，第一条规则中使用的 3 个依赖文件并不存在，如果在 Shell 终端利用命令行编译，Gcc 将退出编译。但是 make 管理项目在执行 Gcc 时先检查依赖文件是否存在，若不存在则先执行别的规则以生成缺少的依赖文件，最后才编译依赖性最强的目标。上述编译方式正是 make 管理项目的优势。如果在编译目标 painter 时 make 发现依赖文件 painter.o、shape_lib.o 和 op_lib.o 已经存在，则它不急于再次运行后面的规则重新编译得到它们，而是比较这些依赖文件与其对应的源文件的生成时间，如果判定有一个或者多个源文件新于这些依赖文件，make 才重新编译生成这些文件以反映相关源文件的最新变化，否则使用旧的依赖文件完成目标 painter 的编译。

在编写 Makefile 时我们会使用一些常用的目标名如 clean、install 、uninstall、dist、tags、depend、test、check、installtest 以及 installcheck。目标 clean 一般用来清除编译过程中的中间文件。而名为 install 的目标常会把最终的二进制文件、所支持的库文件和 Shell 脚本以及相关文档移到文件系统中与它对应的位置，同时设置文件的权限和所有者。uninstall 用来删除 install 目标安装的文件。dist 常常用于删除编译工作目录中旧的二进制文件和目标文件并且创建归档文件。tags 用来更新或创建程序的标记表。depend 用来设置 Makefile 文件中各个目标所需要的依赖文件列表。最后，installtest 和 installcheck 一般用于验证 install 目标的安装过程。

6.2 编写 Makefile 的规则

上一节对 Makefile 文件使用的基本方法进行了介绍，本节详细介绍使用 Makefile 的规

则，对虚拟目标、Makefile 的变量、Make 的变量、隐式规则以及模式规则逐个分析。

6.2.1　虚拟目标

在 Makefile 文件中，除了可设计普通的文件目标外，还可设计虚拟目标，这种目标并不对应实际的文件。在上一节列出的程序中，最后一个以 clean 开头的规则就是虚拟目标。clean 冒号后面是空白，说明它不需要依赖文件。由于虚拟目标没有依赖文件，它不会自动执行，要想启动虚拟目标的创建必须使用如下命令：

```
make [virtual target]
```

其中[virtual target]表示虚拟目标的名字。make 命令不自动创建虚拟目标的原因是它在执行一条规则时首先要检查依赖文件与目标的新旧关系，当没有检测到虚拟目标依赖文件时就认为虚拟目标是最新的版本，不需要重新创建。在实际编程时，往往也需要使用虚拟目标并且也不希望 make 随便创建这些虚拟目标。例如，VxWorks 源代码最上层的 Makefile 就有一个 clean 虚拟目标，在编译内核时我们就不希望它被创建，因为创建的过程与编译内核的过程正好相反，它将删除编译内核所需要的全部依赖文件。

实际上，我们也可以使用 make 的特殊目标.PHONY 来蒙骗 make 命令，让它把虚拟目标当成普通目标看待。.PHONY 目标的依赖文件的含义与普通文件一样，但是 make 命令不检查它的依赖文件而直接执行.PHONY 所对应规则的命令。因此如果使用.PHONY 来过渡，就可保证每次创建虚拟目标时都得到最新目标。这样，重新编写上一节的 Makefile 示例如下：

```
#a simple Makefile
painter: painter.o shape_lib.o op_lib.o
           gcc  -o painter painter.o shape_lib.o op_lib.o
    painter.o : painter.c painter.h op_lib.h shape_lib.h
          gcc  -c painter.c
    shape_lib.o :shape_lib.c shape_lib.h
          gcc  -c shape_lib.c
    op_lib.o :op_lib.c op_lib.h
          gcc  -c op_lib.c
    .PHONY:clean
    clean :
         rm edito *.o
```

6.2.2　Makefile 的变量

与在其他高级语言编程一样，编写 Makefile 时也可以使用变量，而且其中也有一些特殊变量。Makefile 中的变量一般使用大写，可以使用等号为它们直接定义字符串的值，而引用时只需在用小括号括起来的变量名前加上$符号。

当编写大型应用程序的 Makefile 时，其中牵涉到的依赖文件和规则繁多，如果使用变量来代表某些依赖文件的路径，则可大大简化 Makefile 文件的编写。有经验的编程人员一般在 Makefile 文件的开始就定义文件中要用到的所有变量，这样可使 Makefile 更清晰，而且方便了文件的修改。为了说明变量的使用技巧，将上面的例子再次使用变量编写如下：

```
#a simple Makefile
OBJECT= painter.o shape_lib.o op_lib.o
HEADER= painter.h op_lib.h shape_lib.h
painter: $(OBJECT)
           gcc  -o painter $(OBJECT
    painter.o : painter.c $(HEADER)
          gcc  -c painter.c
    shape_lib.o :shape_lib.c shape_lib.h
          gcc  -c shape_lib.c
    op_lib.o :op_lib.c op_lib.h
          gcc  -c op_lib.c
    .PHONY:clean
    clean :
         rm edito $(OBJECT)
```

这个例子中使用了两个变量 OBJECT、HEADER，它们分别表示头文件组合和目标文件组合。make 在读到这两个变量时会立即展开，效果与直接书写依赖文件一样。make 在展开变量时使用两种展开方式即递归展开和简单展开。递归展开指 make 在展开变量时若在其中又碰到变量就逐层展开。简单展开则只展开一次。如果要在一个变量上不断地追加字符串就应该使用简单展开，否则将陷入展开的无限循环，导致 make 命令的结束。例如：

```
GCC=gcc
GCC=$(GCC)  -O2
```

这两行代码就是为 GCC 追加字符串，执行的结果得不到 gcc -O2，因为两次引用 GCC 引起了展开死循环。对于这种情况应该使用简单展开来避免。如果采用如下形式的简单展开：

```
GCC：=gcc
GCC+=  -O2
```

就可消除变量的嵌套引用。这里第一条语句定义 GCC 为 gcc，第二条语句则实现追加，这样 GCC 的最终值为 gcc -O2。这个示例仅用来说明而已，实际上，像这么简单的语句不用变量也很简短，但它说明了什么情况该使用递归展开，什么情况该使用简单展开。

6.2.3 make 的变量

上面所说的 Makefile 的变量都是自定义变量，make 管理项目也允许在 Makefile 使用特殊变量，我们把它们叫做 make 变量。make 变量包括环境变量、自动变量和预定义变量。

这里所说的环境变量就是系统的环境变量，make 命令执行时会读取系统中这些已定义的变量，并且创建与它们意义相同的变量。当然，有些用户也许在 Makefile 文件中创建了与系统环境变量同名的变量，发生此类冲突时，在编译过程中用户定义的变量将覆盖系统的环境变量。

make 管理项目允许使用的自动变量全部以美元符号$领头，这些变量的含义参见表 6-1。

表 6-1　　自动变量的含义

变　量	说　明
$@	Makefile 中规则的目标所对应的文件名
$<	Makefile 中规则的目标所对应的第一个依赖文件名
$^	Makefile 中规则的目标所对应所有依赖文件的列表，以空格为分隔
$?	Makefile 中规则的目标所对应的依赖文件中新于目标的文件列表，以空格分隔
$(@D)	如果目标在子目录中就指目标文件的目录部分
$(@F)	如果目标在子目录中就指目标文件的文件名部分

make 管理项目支持使用的预定义变量主要用于定义程序名称以及传递给这些程序的参数和标志值。这些变量的含义如表 6-2 所示。

表 6-2　　预定义变量的含义

变　量	含　义
AR	归档维护程序名，默认值为 ar
AS	汇编程序名，默认值为 as
CC	C 语言编译程序，默认值为 cc
CPP	C 语言预处理程序，默认值为 cpp
RM	删除文件程序，默认值为“rm -f”
ARFLAGS	传递给归档维护程序的标志值，默认值为 rv
ASFLAGS	传递给汇编程序的标志值，无默认值
CFLAGS	传递给 C 语言编译程序的标志值，无默认值
CPPFLAGS	传递给 C 语言预处理程序的标志值，无默认值
LDFLAGS	传递给连接程序的标志值，无默认值

这些预定义变量在 Makefile 中可以重新定义。定义值将覆盖它们的默认含义。

6.2.4　隐式规则

上面介绍的 Makefile 规则都是用户自己声明的。其实 make 管理项目还支持自己默认的隐式规则。这样的规则有很多，它们都有特殊的用途。其中最为常用的是由源代码文件生成 Object 文件的预定义规则。这个规则实际上简化了 Makefile 文件的书写。以下面的 Makefile 为例来说明此规则的用法：

```
#a simple Makefile
OBJECT= painter.o shape_lib.o op_lib.o
painter: $(OBJECT)
            gcc  -o painter $(OBJECT

    .PHONY:clean
    clean :
         rm edito $(OBJECT)
```

此 Makefile 中目标 painter 对应的规则使用了依赖文件 painter.o、shape_lib.o 和 op_lib.o，然而此 Makefile 文件中没有生成这些依赖文件的规则。make 管理项目碰到这种情况时，会使用隐式规则来生成这些文件。以 shape_lib.o 为例进行说明，当 make 管理项目发现没有规则生成 shape_lib.o 时，首先找到与其对应的源代码文件 shape_lib.c，然后使用 gcc -c shape_lib.c -o shape_lib.o 编译生成这个 Object 文件。也就是说，对于本例，make 在生成 painter 之前会依次查找 painter.c、shape_lib.c 和 op_lib.c 以便生成 painter.o、shape_lib.o 和 op_lib.o。

实际上，由于 Object 文件可以由 C、Pascal 和 Fortran 等源文件生成，因此 make 管理项目会依次查找这些文件，直到生成所需的 Object 文件为止。如果 make 管理项目查找的不是 C 语言的源代码，那么它将使用与代码对应的编译器完成编译过程。正因为这样，在使用一种语言编程时，这个隐式规则可以为开发人员带来便利，但在混合语言编程时，这种规则可能带来麻烦。

6.2.5 模式规则

模式规则是指用户自定义的隐式规则。上面已经说过隐式规则使用得当可以为 make 管理项目的开发带来便利，因此在实际编码时可以大量地定义隐式规则。模式规则类似于普通的规则，但是它的目标和依赖文件必须带有符号%。这个符号可以与任何非空字符串匹配。例如，规则%.o:%.c 表示在这个管理项目中所有的 Object 文件都是由 C 语言源代码文件生成的。实际上，make 已经对一些模式规则进行了预定义，例如 make 对上面例子的预定义规则为：

```
%.o:%.c
$(CC) -c $(CFLAGS) $(CPPFLAGS) $< -o $@
```

此规则也表示所有的 Object 文件都由 C 源代码生成。使用此规则时，它利用自动变量$<和$@代替第一个依赖文件和 Object 文件。而变量 CC、CFLAGS 和 CPPFLAGS 使用系统中的预定义值。

6.3 Make 命 令

make 命令用于实现 make 管理项目。对于一个开发完成的项目，在其包含 Makefile 的目录下，执行 make 命令便可以实现一个应用项目的编译和连接。在键入 make 命令时，也可以带一些命令选项。make 常用的命令选项及其含义如表 6-3 所示。

表 6-3 make 常用的命令选项及其含义

变 量	含 义
-f file	指定 Makefile 的文件名
-n	打印将需要执行的命令，实际上并不执行这些命令
-Idirname	指定所用 Makefile 所在的目录

续表

变　量	含　义
-s	在执行时不打印命令名
-w	如果 make 执行时要改变目录，则打印当前的执行目录
Wfile	如果文件已经修改，就使用-n 选项显示 make 将执行的命令
-r	禁止使用 make 所有的内置规则（包括隐式规则和模式规则）
-d	打印调试信息
-I	忽略 make 规则中执行命令后返回的非 0 错误代码，这时即使有命令返回了非 0 状态代码，make 命令仍继续执行。
-k	如果某个目标编译失败，继续编译其他目标。若不指明此选项，一个目标编译失败后，make 命令就退出
-jN	N 为非 0 整数，表示每次运行的命令条数

如果使用 make 命令时碰到错误，可以带上-d 选项再次执行 make 命令以打印出调试信息。调试信息的内容十分丰富，它包括在重新编译时 make 需要检查的文件、比较的文件和比较结果、需要重新生成的文件、make 想要使用的隐式规则、make 实际使用的隐式规则以及执行的命令。通过这些调试信息，用户可以快速分析出问题的所在。

实际上如果真出现问题，make 会显示出错信息，这些出错信息在 make 的帮助文档中有详细描述。make 常见的出错信息如表 6-4 所示。

表 6-4　　make 常见的出错信息

错 误 信 息	说　明
Recursive variable 'XX' references itself (eventually). stop	此信息表示 Makefile 文件中出现对字符串 XX 的递归引用，导致了无限循环
No rule to make target 'TARGET'. stop	此信息表示 Makefile 文件中没有包含创建指定的 TARGET 需要的规则，而且也没有适当的默认规则可用
Target 'TARGET' is up to date	此信息表示指定 TARGET 的依赖文件没有变化
Illegal option-OPTION	此信息表示 make 命令在执行时碰到了不可识别的选项
COMMAND：Command not found	此信息表示 make 命令找不到 COMMAND。一般是由于命令的拼写错误或者不在系统可执行命令的路径下
Target 'TARGET' not remade because of errors	此信息表示在编译 TARGET 时出错，只有在编译时使用了-k 选项，这个信息才可能出现

6.4　Makefile 实例分析

以上几节对 make 管理项目以及它的 Makefile 文件进行了细致的分析，读者对它已经有了基本的概念，但对它的具体应用却不甚了解。因此本节将详细分析 VxWorks 系统应用中的

Makefile 文件。Makefile 文件是 VxWorks 系统可引导应用和可下载应用编译的基础，虽然它没有展示 VxWorks 系统源代码的全部编译过程，但是对它的分析将帮助读者建立起使用 Makefile 组织自己所开发应用程序的思维模型。目前大量的企业级应用在开发其 Makefile 时或多或少参考了 VxWorks 系统默认的编译组织技巧。

这里首先列出一个 VxWorks 系统可引导应用的 Makefile 文件，然后列出一个 VxWorks 系统可下载应用的 Makefile 文件，并且作出详细分析。

```
############## 可引导 VxWorks 应用 Makefile 的开始#######

## 内核信息
PRJ_DIR         = $(WIND_BASE)/target/proj/ads8260_vx
PRJ_FILE        = ads8260_vx.wpj
PRJ_TYPE        = vxWorks
PRJ_OBJS        = sysALib.o sysLib.o usrAppInit.o prjConfig.o linkSyms.o
BOOT_OBJS       = romInit.o romStart.o   #起动目标文件
BUILD_SPEC      = default
BSP_DIR         = $(WIND_BASE)/target/config/ads8260
TGT_DIR         = $(WIND_BASE)/target  #目标目录

## 设置搜索依赖关系的目录

vpath %.c $(BSP_DIR)
vpath %.cpp $(BSP_DIR)
vpath %.cxx $(BSP_DIR)

## 编译的配置信息

ifeq ($(BUILD_SPEC),default)
CPU             = PPCEC603
TOOL            = gnu
DEFAULT_RULE    = vxWorks
endif

## 组件配置信息

COMPONENTS = INCLUDE_ANSI_ASSERT \   #包含 ANSI 标准库
    INCLUDE_ANSI_CTYPE \
    INCLUDE_ANSI_LOCALE \
    INCLUDE_ANSI_MATH \
    INCLUDE_ANSI_STDIO \
    INCLUDE_ANSI_STDIO_EXTRA \
    INCLUDE_ANSI_STDLIB \
    INCLUDE_ANSI_STRING \
    INCLUDE_ANSI_TIME \
    INCLUDE_ARP_API \
    INCLUDE_BOOTP \
    INCLUDE_BOOT_LINE_INIT \
    INCLUDE_BSD_SOCKET \
```

```
INCLUDE_BUF_MGR \
INCLUDE_CPLUS \                          #包含 C++组件
INCLUDE_CPLUS_IOSTREAMS \
INCLUDE_CPLUS_STL \
INCLUDE_DHCPC_LEASE_CLEAN \
INCLUDE_DLL \
INCLUDE_END \                            #包含 END 驱动组件
INCLUDE_END_BOOT \
INCLUDE_ENV_VARS \
INCLUDE_EXC_HANDLING \
INCLUDE_EXC_TASK \
INCLUDE_FLOATING_POINT \
INCLUDE_FORMATTED_IO \
INCLUDE_HASH \
INCLUDE_HOST_TBL \
INCLUDE_ICMP \
INCLUDE_IGMP \
INCLUDE_IO_SYSTEM \                      #包含 I/O 系统
INCLUDE_IP \
INCLUDE_KERNEL \
INCLUDE_LOGGING \
INCLUDE_LOOPBACK \
INCLUDE_MEMORY_CONFIG \                  #包含内存配置
INCLUDE_MEM_MGR_BASIC \
INCLUDE_MEM_MGR_FULL \
INCLUDE_MODULE_MANAGER \
INCLUDE_MSG_Q \                          #包含消息队列组件
INCLUDE_MSG_Q_SHOW \
INCLUDE_MUX \
INCLUDE_NETDEV_NAMEGET \                 #包含网络基本库
INCLUDE_NETMASK_GET \
INCLUDE_NETWORK \
INCLUDE_NET_HOST_SETUP \
INCLUDE_NET_INIT \
INCLUDE_NET_LIB \
INCLUDE_NET_REM_IO \
INCLUDE_NET_SETUP \
INCLUDE_PIPES \                          #包含管道支持
INCLUDE_POSIX_CLOCKS \
INCLUDE_RNG_BUF \
INCLUDE_SELECT \
INCLUDE_SEM_BINARY \                     #包含信号量组件
INCLUDE_SEM_COUNTING \
INCLUDE_SEM_MUTEX \
INCLUDE_SIGNALS \
INCLUDE_SIO \
INCLUDE_STDIO \
INCLUDE_SYM_TBL \
INCLUDE_SYSCLK_INIT \
INCLUDE_SYSHW_INIT \
```

```
        INCLUDE_SYS_START \
        INCLUDE_TASK_HOOKS \
        INCLUDE_TASK_VARS \
        INCLUDE_TCP \
        INCLUDE_TFTP_CLIENT \
        INCLUDE_TIMEX \
        INCLUDE_TTY_DEV \
        INCLUDE_UDP \
        INCLUDE_USER_APPL \
        INCLUDE_WATCHDOGS \
        INCLUDE_WDB \                    #包含 WDB 调试组件
        INCLUDE_WDB_BANNER \
        INCLUDE_WDB_BP \
        INCLUDE_WDB_COMM_NETWORK \
        INCLUDE_WDB_CTXT \
        INCLUDE_WDB_DIRECT_CALL \
        INCLUDE_WDB_EVENTPOINTS \
        INCLUDE_WDB_EVENTS \
        INCLUDE_WDB_EXC_NOTIFY \
        INCLUDE_WDB_EXIT_NOTIFY \
        INCLUDE_WDB_FUNC_CALL \
        INCLUDE_WDB_GOPHER \
        INCLUDE_WDB_MEM \
        INCLUDE_WDB_REG \
        INCLUDE_WDB_START_NOTIFY \
        INCLUDE_WDB_TASK \
        INCLUDE_WDB_TASK_BP \
        INCLUDE_WDB_USER_EVENT \
        INCLUDE_WDB_VIO \
        INCLUDE_WDB_VIO_LIB
    COMPONENT_LIBS =

    include $(TGT_DIR)/h/make/defs.project

    ## 其他编译配置信息

    ifeq ($(BUILD_SPEC),default)
    AR              = arppc
    AS              = ccppc
    CC              = ccppc
    CFLAGS          = -g -mstrict-align -ansi -nostdinc -DRW_MULTI_THREAD   \  #编译选项
    -D_REENTRANT -fvolatile -fno-builtin -fno-for-scope -msoft-float -I$(PRJ_DIR) \
    -I$(WIND_BASE)/target/config/ads8260 -I$(WIND_BASE)/target/h           \
    -I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv     \
    -DCPU=PPCEC603 -DPRJ_BUILD     #目标 CPU 类型
    CFLAGS_AS       = -g -mstrict-align -ansi -nostdinc -fvolatile -fno-builtin \    #汇编选
项
    -fno-for-scope -P -x assembler-with-cpp -I$(PRJ_DIR)             \
    -I$(WIND_BASE)/target/config/ads8260 -I$(WIND_BASE)/target/h  \
    -I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
```

```
-DCPU=PPCEC603 -DPRJ_BUILD
CPP= ccppc -B$(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib / -E -P -xc
LD              = ldppc           #装载命令
LDFLAGS         = -X  -N          #装载选项
LD_PARTIAL= ccppc -B$(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib/  \
-nostdlib -r -Wl,-X
LD_PARTIAL_FLAGS = -X -r
LIBS            = $(WIND_BASE)/target/lib/libPPCEC603gnuvx.a
NM              = nmppc
OPTION_DEFINE_MACRO = -D
OPTION_INCLUDE_DIR = -I
RAM_HIGH_ADRS   = 00200000 # RAM 文本和数据地址高端
RAM_LOW_ADRS    = 00010000 # RAM 文本和数据地址低端
ROM_WARM_ADRS   = ff800108 # ROM 入口地址
SIZE            = sizeppc
POST_BUILD_RULE  =
EXTRA_MODULES   =
endif

# 添加编译行

# 添加编译行的结尾

include $(TGT_DIR)/h/make/rules.project

## 目标文件配置信息

ifeq ($(BUILD_SPEC),default)

sysALib.o:          #编译 Sys Alib.o 的命令行
    $(AS) -g -mstrict-align -ansi -nostdinc -fvolatile -fno-builtin -fno-for-scope -P -x \
assembler-with-cpp -I$(PRJ_DIR) -I$(WIND_BASE)/target/config/ads8260 \
-I$(WIND_BASE)/target/h -I$(WIND_BASE)/target/config/comps/src    \
-I$(WIND_BASE)/target/src/drv -DCPU=PPCEC603 -DPRJ_BUILD  -c \
$(WIND_BASE)/target/config/ads8260/sysALib.s -o sysALib.o

sysLib.o:          #编译 SysfLib.o 的命令行
    $(CC) -g -mstrict-align -ansi -nostdinc -DRW_MULTI_THREAD -D_REENTRANT \
-fvolatile -fno-builtin -fno-for-scope -msoft-float -I$(PRJ_DIR)         \
-I$(WIND_BASE)/target/config/ads8260 -I$(WIND_BASE)/target/h    \
-I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
-DCPU=PPCEC603 -DPRJ_BUILD  -c                                  \
$(WIND_BASE)/target/config/ads8260/sysLib.c

usrAppInit.o:      #编译 USRAppInit.o 的命令行
    $(CC) -g -mstrict-align -ansi -nostdinc -DRW_MULTI_THREAD -D_REENTRANT \
-fvolatile -fno-builtin -fno-for-scope -msoft-float -I$(PRJ_DIR)          \
-I$(WIND_BASE)/target/config/ads8260 -I$(WIND_BASE)/target/h     \
-I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
-DCPU=PPCEC603 -DPRJ_BUILD  -c $(PRJ_DIR)/usrAppInit.c
```

```
prjConfig.o:       #编译 prj Config.o 的命令行
    $(CC) -g -mstrict-align -ansi -nostdinc -DRW_MULTI_THREAD -D_REENTRANT \
-fvolatile -fno-builtin -fno-for-scope -msoft-float -I$(PRJ_DIR)          \
-I$(WIND_BASE)/target/config/ads8260 -I$(WIND_BASE)/target/h     \
-I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
-DCPU=PPCEC603 -DPRJ_BUILD  -c $(PRJ_DIR)/prjConfig.c

linkSyms.o:        #编译 linksyms.o 的命令行
    $(CC) -g -mstrict-align -ansi -nostdinc -DRW_MULTI_THREAD -D_REENTRANT \
 -fvolatile -fno-builtin -fno-for-scope -msoft-float -I$(PRJ_DIR)        \
-I$(WIND_BASE)/target/config/ads8260 -I$(WIND_BASE)/target/h    \
-I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
-DCPU=PPCEC603 -DPRJ_BUILD  -c $(PRJ_DIR)/linkSyms.c
endif

## 目标文件依赖关系

sysALib.o: $(WIND_BASE)/target/config/ads8260/sysALib.s \
    $(PRJ_DIR)/prjComps.h \
    $(PRJ_DIR)/prjParams.h

sysLib.o: $(WIND_BASE)/target/config/ads8260/sysLib.c \
    $(PRJ_DIR)/prjComps.h \
    $(PRJ_DIR)/prjParams.h

romInit.o: $(WIND_BASE)/target/config/ads8260/romInit.s \
    $(PRJ_DIR)/prjComps.h \
    $(PRJ_DIR)/prjParams.h

romStart.o: $(WIND_BASE)/target/config/comps/src/romStart.c \    #依赖一个源文件和两个头
文中
    $(PRJ_DIR)/prjComps.h \
    $(PRJ_DIR)/prjParams.h

usrAppInit.o: $(PRJ_DIR)/usrAppInit.c    #依赖单个源文件

prjConfig.o: $(PRJ_DIR)/prjConfig.c $(PRJ_DIR)/prjComps.h $(PRJ_DIR)/prjParams.h

linkSyms.o: $(PRJ_DIR)/linkSyms.c

## 用户自定义规则

romInit.o :             # romInit.o 的编译规则
$(CC) $(OPTION_OBJECT_ONLY) $(CFLAGS_AS) $(ROM_FLAGS_EXTRA) $< -o $@

romStart.o :           # romStart.o 的编译规则
    $(CC) $(OPTION_OBJECT_ONLY) $(CFLAGS) $(ROM_FLAGS_EXTRA) $< -o $@

############## 可引导 VxWorks 应用 Makefile 的结尾#######
```

```
############## 可下载 VxWorks 应用 Makefile 的开始######

## 内核信息

PRJ_DIR         = $(WIND_BASE)/target/proj/simpc_vx    #工程所在目录
PRJ_FILE        = simpc_vx.wpj
PRJ_TYPE        = vxWorks
PRJ_OBJS        = sysLib.o usrAppInit.o prjConfig.o linkSyms.o
BOOT_OBJS       =
BUILD_SPEC      = default
BSP_DIR         = $(WIND_BASE)/target/config/simpc     #BSP 所在目录
TGT_DIR         = $(WIND_BASE)/target                 #目标目录

## 设置搜索依赖关系的目录

vpath %.c $(BSP_DIR)
vpath %.cpp $(BSP_DIR)
vpath %.cxx $(BSP_DIR)

##编译配置信息

ifeq ($(BUILD_SPEC),default)
CPU             = SIMNT                              #编译仿真应用
TOOL            = gnu                                 #使用 gnu 编译器
DEFAULT_RULE    = vxWorks
endif

## 组件配置信息

COMPONENTS = INCLUDE_ANSI_ASSERT \    #包含 ANSI 标准库
    INCLUDE_ANSI_CTYPE \
    INCLUDE_ANSI_LOCALE \
    INCLUDE_ANSI_MATH \
    INCLUDE_ANSI_STDIO \
    INCLUDE_ANSI_STDIO_EXTRA \
    INCLUDE_ANSI_STDLIB \
    INCLUDE_ANSI_STRING \
    INCLUDE_ANSI_TIME \
    INCLUDE_BUF_MGR \
    INCLUDE_CACHE_ENABLE \           #包含高速编存支持
    INCLUDE_CACHE_SUPPORT \
    INCLUDE_CPLUS \                    #包含 C++组件
    INCLUDE_CPLUS_IOSTREAMS \
    INCLUDE_CPLUS_STL \
    INCLUDE_DLL \
    INCLUDE_ENV_VARS \
    INCLUDE_EXC_HANDLING \          #包含异常处理组件
    INCLUDE_EXC_TASK \
    INCLUDE_FLOATING_POINT \
```

```
INCLUDE_FORMATTED_IO \
INCLUDE_HASH \
INCLUDE_IO_SYSTEM \
INCLUDE_KERNEL \
INCLUDE_LOGGING \
INCLUDE_MEMORY_CONFIG \          #包含内存组件
INCLUDE_MEM_MGR_BASIC \
INCLUDE_MEM_MGR_FULL \
INCLUDE_MODULE_MANAGER \
INCLUDE_MSG_Q \                     #包含消息队列组件
INCLUDE_MSG_Q_SHOW \
INCLUDE_NTPASSFS \
INCLUDE_PIPES \
INCLUDE_POSIX_CLOCKS \
INCLUDE_RBUFF \
INCLUDE_RNG_BUF \
INCLUDE_SELECT \
INCLUDE_SEM_BINARY \              #包含信号量组件
INCLUDE_SEM_COUNTING \
INCLUDE_SEM_MUTEX \
INCLUDE_SEQ_TIMESTAMP \
INCLUDE_SIGNALS \
INCLUDE_SIO \
INCLUDE_STDIO \
INCLUDE_SYM_TBL \
INCLUDE_SYSCLK_INIT \
INCLUDE_SYSHW_INIT \
INCLUDE_SYS_START \
INCLUDE_TASK_HOOKS \             #包含任务组件
INCLUDE_TASK_SHOW \
INCLUDE_TASK_VARS \
INCLUDE_TIMEX \
INCLUDE_TRIGGERING \
INCLUDE_TTY_DEV \
INCLUDE_USER_APPL \
INCLUDE_WATCHDOGS \
INCLUDE_WDB \                     #包含 WDB 调试支持
INCLUDE_WDB_BANNER \
INCLUDE_WDB_BP \
INCLUDE_WDB_COMM_PIPE \
INCLUDE_WDB_CTXT \
INCLUDE_WDB_DIRECT_CALL \
INCLUDE_WDB_EVENTPOINTS \
INCLUDE_WDB_EVENTS \
INCLUDE_WDB_EXC_NOTIFY \
INCLUDE_WDB_EXIT_NOTIFY \
INCLUDE_WDB_FUNC_CALL \
INCLUDE_WDB_GOPHER \
INCLUDE_WDB_MEM \
INCLUDE_WDB_REG \
```

```
        INCLUDE_WDB_START_NOTIFY \
        INCLUDE_WDB_SYS \
        INCLUDE_WDB_TASK \
        INCLUDE_WDB_TASK_BP \
        INCLUDE_WDB_TSFS \
        INCLUDE_WDB_USER_EVENT \
        INCLUDE_WDB_VIO \
        INCLUDE_WDB_VIO_LIB \
        INCLUDE_WINDVIEW \                #包含 WINDVIEW 组件
        INCLUDE_WINDVIEW_CLASS \
        INCLUDE_WVUPLOAD_FILE \
        INCLUDE_WVUPLOAD_TSFSSOCK
    COMPONENT_LIBS =

    include $(TGT_DIR)/h/make/defs.project

    ## 其他编译配置信息

    ifeq ($(BUILD_SPEC),default)
    AR               = arsimpc
    AS               = ccsimpc
    CC               = ccsimpc
    CFLAGS= -U__WINNT -UWIN32 -U__WINNT__ -UWINNT -U__MINGW32__ \  #编译选项
    -U_WIN32 -U__WIN32 -U__WIN32__ -U__WIN32 -mpentium -ansi -nostdinc -g \
    -nostdlib -fno-builtin -fno-defer-pop -Wall -DRW_MULTI_THREAD -D_REENTRANT \
    -I$(PRJ_DIR) -I$(WIND_BASE)/target/config/simpc -I$(WIND_BASE)/target/h   \
     -I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv      \
    -DCPU=SIMNT -DPRJ_BUILD
    CFLAGS_AS = -mpentium -ansi -nostdinc -g -nostdlib -fno-builtin -fno-defer-pop -P -x \  #
汇编命令行
    assembler-with-cpp -I$(PRJ_DIR) -I$(WIND_BASE)/target/config/simpc \
    -I$(WIND_BASE)/target/h -I$(WIND_BASE)/target/config/comps/src   \
    -I$(WIND_BASE)/target/src/drv -DCPU=SIMNT -DPRJ_BUILD

    CPP =  ccsimpc -B$(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib/  \   #C++源码编译行
    -U__WINNT -UWIN32 -U__WINNT__ -UWINNT -U__MINGW32__ -U_WIN32 \
    -U__WIN32 -U__WIN32__ -U__WIN32 -E -P  - xc

    LD               = ldsimpc
    LDFLAGS          = --subsystem=windows
    LDOUT_CONV= wtxtcl  \
    $(WIND_BASE)/host/$(WIND_HOST_TYPE)/bin/simpcToExe.tcl
    LD_PARTIAL  = ccsimpc -B$(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib/ \  #装载命令行
    -U__WINNT -UWIN32 -U__WINNT__ -UWINNT -U__MINGW32__ -U_WIN32 \
    -U__WIN32 -U__WIN32__ -U__WIN32 -nostdlib -r -Wl,-X

    LD_PARTIAL_FLAGS = -r
    LD_RAM_FLAGS    = $(WIND_BASE)/host/$(WIND_HOST_TYPE)/i386-pc-mingw32/lib/crt1.o
    LIBS             = $(WIND_BASE)/target/lib/libSIMNTgnuvx.a     #用到的静态库
    NM               = nmsimpc -g
```

```
OPTION_DEFINE_MACRO = -D
OPTION_INCLUDE_DIR = -I
SIZE            = sizesimpc
VXSIZEPROG      = echo
RAM_LOW_ADRS    =
RAM_HIGH_ADRS   =
POST_BUILD_RULE  =
EXTRA_MODULES   =
endif

# 添加的编译行

#添加编译行的结尾

include $(TGT_DIR)/h/make/rules.project

## 目标文件的配置信息

ifeq ($(BUILD_SPEC),default)

sysLib.o:      #  编译 sysLib.o 的命令语句
    $(CC) -U__WINNT -UWIN32 -U__WINNT__ -UWINNT -U__MINGW32__ \
-U_WIN32 -U__WIN32 -U__WIN32__ -U__WIN32 -mpentium -ansi -nostdinc -g \
-nostdlib -fno-builtin -fno-defer-pop -Wall -DRW_MULTI_THREAD -D_REENTRANT \
-I$(PRJ_DIR) -I$(WIND_BASE)/target/config/simpc -I$(WIND_BASE)/target/h \
-I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
-DCPU=SIMNT -DPRJ_BUILD  -c $(WIND_BASE)/target/config/simpc/sysLib.c

usrAppInit.o:   #  编译 usr5AppInit.o 的命令语句
    $(CC) -U__WINNT -UWIN32 -U__WINNT__ -UWINNT -U__MINGW32__ \
-U_WIN32 -U__WIN32 -U__WIN32__ -U__WIN32 -mpentium -ansi -nostdinc -g \
-nostdlib -fno-builtin -fno-defer-pop -Wall -DRW_MULTI_THREAD -D_REENTRANT \
-I$(PRJ_DIR) -I$(WIND_BASE)/target/config/simpc -I$(WIND_BASE)/target/h \
-I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
-DCPU=SIMNT -DPRJ_BUILD  -c $(PRJ_DIR)/usrAppInit.c

prjConfig.o:     # 编译 prjConfig.o 的命令语句
    $(CC) -U__WINNT -UWIN32 -U__WINNT__ -UWINNT -U__MINGW32__ \
-U_WIN32 -U__WIN32 -U__WIN32__ -U__WIN32 -mpentium -ansi -nostdinc -g \
-nostdlib -fno-builtin -fno-defer-pop -Wall -DRW_MULTI_THREAD -D_REENTRANT \
-I$(PRJ_DIR) -I$(WIND_BASE)/target/config/simpc -I$(WIND_BASE)/target/h \
-I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
-DCPU=SIMNT -DPRJ_BUILD  -c $(PRJ_DIR)/prjConfig.c

linkSyms.o:      # 编译 LinkSyms.o 的命令语句
    $(CC) -U__WINNT -UWIN32 -U__WINNT__ -UWINNT -U__MINGW32__ \
-U_WIN32 -U__WIN32 -U__WIN32__ -U__WIN32 -mpentium -ansi -nostdinc -g \
-nostdlib -fno-builtin -fno-defer-pop -Wall -DRW_MULTI_THREAD -D_REENTRANT \
-I$(PRJ_DIR) -I$(WIND_BASE)/target/config/simpc -I$(WIND_BASE)/target/h \
-I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src/drv \
```

```
-DCPU=SIMNT -DPRJ_BUILD  -c $(PRJ_DIR)/linkSyms.c
endif

## 目标文件的配置信息

sysLib.o: $(WIND_BASE)/target/config/simpc/sysLib.c \        #SysLib.o 依赖一个源文件和两个头文件
    $(PRJ_DIR)/prjComps.h \
    $(PRJ_DIR)/prjParams.h

usrAppInit.o: $(PRJ_DIR)/usrAppInit.c

prjConfig.o: $(PRJ_DIR)/prjConfig.c $(PRJ_DIR)/prjComps.h $(PRJ_DIR)/prjParams.h

linkSyms.o: $(PRJ_DIR)/linkSyms.c

## 用户自定义规则

    ########## VxWorks 可下载应用 Makefile 的结尾############
```

6.5　Gcc 的基本概念

Gcc 是 GNU 项目的编译器组件此外 gcc 在 g77 的支持下之一，而且是 VxWorks 源代码开发的基础。它可以编译 C、C++以及 Object C 的源代码。此外在 g77 的支持下它还可以编译 Fortran 语言。以后它还会增强对其他语言如 Pascal 的支持。

结合使用 Gcc 和 Make 管理项目，可以对程序的编译过程进行深层次的控制。Gcc 编译器将编译过程分成 4 个阶段，即预处理、编译、汇编和连接。使用 Gcc 编写的代码可在任意一个编译阶段暂停编译过程并且检查相应的输出信息。Gcc 还能在生成的二进制文件中添加所需数量和种类的调试信息。在进行编译时，Gcc 可以带上不同的优化选项，这样可以根据不同的需求优化可执行代码。当然一般不在要求生成调试信息的代码内实施优化，因为优化后的代码与源代码不再一一对应，源代码中的静态变量可能被取消，控制结构可能被简化。

Gcc 是一个交叉平台编译器，能够在所处的 CPU 平台上为异构体系硬件系统开发应用程序。例如，可以在 x86 体系结构的 PC 上开发 MPC 系列处理器的应用程序。Gcc 中具有 3 个 catch-all 警告级和几十个警告。此外 Gcc 对普通的 C 和 C++进行了大量扩展，这些扩展的部分虽然降低了 Gcc 的可移植性，但是有助于方便代码的编写，有助于编译器进行代码优化，也有助于提高代码的执行效率。

为了加深读者对 Gcc 应用的了解，下面对 Gcc 的使用进行简单介绍。首先我们编写如下简单的代码示例：

```
/*sample1.c*/
#include <stdio.h>
int main()
```

```
{
   printf(“Good luck to you!”);
   return 0;
}
```

这个例子只是简单地打印一条信息。可以使用如下命令编译这个例子：

```
# gcc sample1.c  -o sample1
```

此命令要求 Gcc 编译 C 源程序并进行连接。其中的-o 参数用于指明要创建的可执行程序名为 sample1，如果不指定将创建默认的可执行文件 a.out，而创建的目标代码文件和汇编代码文件分别为 sample1.o 和 sample1.s。生成可执行程序 sample1 后可以使用如下命令运行：

```
#./sample1
# Good luck to you!
```

运行以后得到了第二行的结果。在以上编译过程中，Gcc 首先利用预处理程序展开源代码文件的宏并在其中插入包含文件的内容，然后把预处理后的代码编译成目标代码，最后利用连接程序创建名为 sample1 的二进制文件。注意，这里的二进制文件不同于传统意义上的.exe 文件。实际上也可以通过加入编译选项让以上编译过程逐步执行，这时需要以下 3 个命令才能生成可执行文件 sample1：

```
# gcc  -E sample1.c  -o sample1.cpp
#gcc  -x cpp-output  -c sample1.cpp sample1.o
#gcc sample1.o  -o sample1
```

第一个命令生成 sample1.cpp 文件，这个文件中包含了文件 stdio.h 的内容以及其他一些预处理符号信息，选项-E 可使 Gcc 在预处理完成后停止。第二个命令是由预处理文件 sample1.cpp 生成目标代码文件 sample1.o，其中的选项-c 表示编译器只编译不连接，-x 选项表示编译器从指定的步骤开始编译，此例子中是从预处理后的源代码开始编译。第三个命令就是连接目标文件生成二进制代码文件。

在由预处理文件 sample1.cpp 生成目标代码文件 sample1.o，也可以不指定目标代码文件名，默认情况下 Gcc 会通过预处理文件名加上扩展名.o 得到。Gcc 就是通过文件的扩展名来判断文件的类型并决定以何种方式处理文件的。Gcc 可以处理扩展名为以下类型的文件：

```
.c          C 语言源代码文件
.C, .cc     C++语言源代码文件
.i          预处理后的 C 语言源代码
.ii         预处理后的 C++语言源代码
.a,.so      编译后的静态库和动态库代码
.o          编译后的目标代码
.S, .s      汇编语言源代码
```

上面虽然展示了分步获得源代码文件的可执行代码的步骤，但在开发应用程序时，我们并不这样分步进行，只有在特殊情况下才控制编译过程。例如，在开发函数库或其他代码库时，只需要生成目标库文件，这时不需要进行第三步的连接步骤，因而要分步执行编译程序。此外，当程序所包含的头文件相互冲突或者头文件与程序文件冲突时需要分步编译以诊断发生冲突的原因。

在前面介绍 make 管理项目时,示范了怎样将多个源文件的目标代码连接成最终二进制文件的方法，实际上这也是分步控制编译过程形式。

6.6　Gcc　命　令

Gcc 命令是一个十分复杂的命令，它有许多命令行选项。在编译过程中或者说在开发的 Makefile 文件中正确地使用 Gcc 命令的选项可以提高编译效率，优化代码，甚至减轻程序开发的负担。在上一节的示例中我们已经使用了-c、-o 以及-x 等选项。除了这些选项外，我们经常要使用的选项及其含义如表 6-5 所示。

表 6-5　　选项及其含义

选　　项	含　　义
-g	在可执行程序中包含标准调试信息
-lFOO	连接名为 libFOO 的函数库
-static	连接静态库，即执行静态连接
-LDIRNAME	将 DIRNAME 添加到库文件的搜索目录列表中，默认情况下，gcc 只连接共享库
-IDIRNAME	将 DIRNAME 添加到头文件的搜索目录列表中
-DFOO=BAR	在命令行预处理宏 DFOO，值为 BAR
-ggdb	在可执行程序中包含只有 GNU debugger 才识别的调试信息
-O	优化编译过的代码
-ON	指定代码的优化级别。一般使用-O2 就能满足要求
-ansi	表示支持 ANSI/ISO C 的标准语法，取消 GNU 的语法扩展中与 ANSI/ISO C 标准有冲突的部分，可是这个选项不能保证生成与 ANSI/ISO 兼容的代码
-pedantic	表示允许发出 ANSI/ISO C 标准列出的所有警告
-pedantic －errors	表示允许发出 ANSI/ISO C 标准列出的所有错误
-traditional	表示支持 Kernighan & Ritchie C 语法。此语法用于定义函数
-w	表示关闭所有的警告
.Wall	表示允许发出 Gcc 提供的所有有用的警告
-werror	表示把所有警告转换为错误，这样会在警告发生时停止编译
-MW	输出一个与 make 兼容的列表
-v	显示编译过程中每步用到的命令

在编译过程可能碰到需要连接的库函数和头文件并不在标准位置下的包含文件中的情

况，这时可以使用选项-L {DIRNAME}和-I {DIRNAME}指定文件的目录，这时指定目录的搜索顺序在标准目录之前。例如，若要使用/home/wzl/目录下面的库文件和头文件编译自己的应用程序可以使用如下命令：

```
gcc sample1.c / -L/home/wzl/lib -I/home/wzl//include -o sample1 -lme
```

利用此命令编译时 gcc 首先分别在/home/wzl/lib 和/home/wzl//include 下寻找需要的库文件和头文件。此命令中的最后一个选项表示要连接的库文件为 libme.so。

Gcc 在默认情况只连接动态共享库,因此需要连接静态库文件时应该使用-static 选项声明。如果上面用到的库文件 libme 是静态库文件，则应该使用下面的命令来编译：

```
gcc sample1.c / -L/home/wzl/lib -I/home/wzl//include -o sample1 -lme -static
```

此命令中的最后一个选项表示要连接的库文件为静态库文件 libme.a。连接了静态库文件的可执行程序比连接动态共享库时要大得多。不过连接了静态库以后，程序在任何情况下都可以运行。而连接了动态共享库的程序只有在系统内包含了所需的共享库时才可以运行。

有些应用程序为了发挥静态库和共享库双方的优势，生成两类可执行程序，第一类使用静态库，而第二类使用共享库。在没有共享库的系统中运行第一类程序，这时可不受共享库的约束；而在有共享库的系统中运行第一类程序，这时可以节省系统开销。

从上面的命令选项中，我们已经看到了 Gcc 的检查出错和生成警告的功能。其中选项-pedantic 表示 Gcc 只发出 ANSI/ISO C 标准列出的所有警告，这样程序就不能使用 Gcc 扩展语法。而选项-pedantic -errors 的功能与-pedantic 相近，但它针对的是错误。选项-ansi 则支持 ANSI/ISO C 的标准语法，取消 GNU 的语法扩展中与 ANSI/ISO C 标准有冲突的部分。下面举例说明这些选项的用法。首先请看如下代码实例：

```
/*sample2.c*/
#include <stdio.h>
void main()
{
 long long i=1;
 printk(“sample2 is running over”);
 return i;
}
```

这段代码具有两个明显的缺陷，首先在定义 i 时使用了 Gcc 的扩展语法，其次在程序末尾返回 i 与函数声明为无返回值类型不符。如果使用 gcc sample2.c -o sample2 或者 gcc -ansi sample2.c -o sample2 来编译此文件不会有编译错误。原因是使用第一个命令时 Gcc 略去了返回值的错误，而第二个命令中虽然具有选项-ansi，但是它只强制 Gcc 生成标准语法要求的调试信息，最后实际编译得到的代码却不一定遵循 ANSI C 标准。但是如果使用选项-pedantic 来编译则会得到警告信息，编译可以成功；如果使用选项-pedantic -errors 来编译则会得到出错信息，编译不能成功。具体的执行过程如下：

```
# gcc -pedantic sample2.c -o sample2
```

```
 sample2.c: in function 'main':
 sample2.c:5:warning :ANSI C does not support 'long long'
# gcc  -pedantic  -errors sample2.c  -o sample2
 sample2. c: in function 'main':
 sample2.c:5:ANSI C does not support 'long long'
```

以上实例说明利用-ansi、-pedantic 和-pedantic -errors 可以实现与 ANSI C 标准的部分兼容，但不能实现完全兼容。

Gcc 的命令行选项中还有一些优化选项，这些选项可以改进程序的执行效率，不过它们要求较长的编译时间，而且在编译过程中需要更多的内存。其中-O 选项用于对代码的长度和执行时间进行优化，它相当于第一个优化级别（Option level 1）。在这个级别能执行的优化类型取决于目标处理器，但是一般包括线程跳转和延迟退出堆栈这两种优化。线程跳转优化的目的是减少跳转操作的次数。而延迟退出堆栈是指在嵌套函数调用时推迟退出堆栈的时间，要是不做这种优化，函数调用在每次完成后都需要弹出堆栈中的函数参数；做这种优化后，函数参数则暂时保留在堆栈中，直到所有递归调用完成后才同时弹出堆栈中所有积累的函数参数。而-O2 优化除了包含-O 优化的所有内容外还包括安排处理器指令时序。使用-O2 优化时，Gcc 将保证处理器在等待其他指令的结果或者来自高速缓存或内存中的数据时仍然有指令可以执行，因而它的实现与处理器的结构有关。同样-O3 优化包含了-O2 优化的所有内容，同时它还包含循环展开以及其他一些与处理器结构有关的优化内容。

根据与目标处理器相关的信息的多少，还可以使用-f {flag}选项指定需要执行的具体优化类型。常用的有 3 种类型，即-ffastmath、-finline-functions 和-funroll-loops。选项-ffastmath 用于对浮点数学运算进行优化以提高处理速度，但这种优化与 ANSI 标准相悖。选项-finline-functions 用于把简单函数像展开预处理宏一样展开。选项-funroll-loops 用来指定编译器展开在编译期间就可确定循环次数的循环。使用以上选项后，因为展开函数和确定性循环避免了部分变量的查找和函数调用工作，因而从理论上说可提高程序的执行速度。但是由于上述展开将导致可执行代码变长，这对提高程序执行速度来说又是一个负面因素。因此要仔细权衡甚至通过试验才能确定使用以上选项是否合适。

程序员开发程序总是避免不了错误，要排除错误就要进行调试，所以在本节末尾我们讨论一下 Gcc 编译器的调试选项。如果想在编译后的程序中插入调试信息以便调试，则可以使用 Gcc 的-g 和-ggdb 选项。调试选项-gN 如同优化选项-ON 一样，其中的 N 可取整数值 1、2 或 3，并且 N 越大表示在代码中插入的调试信息越多。当使用默认级别 2 时，调试信息包括扩展的符号表、行号以及局部变量信息。使用级别 1 时只生成堆栈转储需要的信息。而使用级别 3 时除包含级别 2 的调试信息外，还包含了宏定义信息。

要想使用 Gdb（GNU Debugger）来调试程序，应该使用-ggdb 选项创建其他一些调试信息以方便调试。程序中创建了这样的调试信息后，其他调试器就不能对它进行调试。原因是程序中添加了适合特定调试器如 Gdb 的信息后，程序大小将膨胀约 100 倍，这样其中绝大部分都是调试信息，而不是代码，用其他调试器对其调试已失去了意义。但是 Gcc 带上标准调试选项时，不会使代码文件的大小增加太多，一般情况下使用由-g 生成的调试信息已足够调试。

除了以上两个调试选项外，还有-p、-pg、-a 和-save-temps，它们用于将统计信息添加到二进制文件中，利用这些信息，程序员可以找到性能瓶颈。选项-p 和-pg 是在代码中创建

只有 GNU prof 才能读取的信息。而选项-a 在代码中插入记录代码块和函数累计执行次数的计数器。最后，选项-save-temps 用于保存编译过程中生成的中间文件如目标文件和汇编文件。

程序调试和代码优化是不相容的两个过程，代码优化会打乱程序的控制流使程序无法调试，因此应该完成了程序调试后再进行代码的优化工作。实际上，这里所说的优化仅指利用编译器 Gcc 所做的优化，在程序设计过程中的设计优化不在此列。程序设计中的优化也是非常重要的，应用开发人员应该通过算法优化来实现最优的程序设计。

6.7 Gcc 扩 展

Gcc 在很多方面对 ANSI C 进行了扩展，有些扩展甚至与 ANSI C 相悖。本节将简单介绍 VxWorks 系统头文件和源代码中经常出现的扩展语法，而这些语法往往是我们用得上的。如果要了解 Gcc 的详细扩展语法可以参考它的信息页（info pages）。

首先 Gcc 使用 long long 数据类型来开辟 64 位存储单元。在 x86 平台上利用 long long 定义一个变量，就要为它分配 64 位存储空间。

其次，Gcc 扩展了关键字 attribute。关键字 attribute 通过指明代码的信息来帮助 Gcc 完成优化。例如，许多标准库函数像 exit()等没有返回值，而编译器会为没有返回值的函数生成相对优化的代码，因此如果用户编写了没有返回值的函数，那么通过 attribute 来指明这一点则可生成较优化的代码。Gcc 提供了属性 noreturn 表示函数没有返回值。如果有一个名为 function_without_return()且无返回值的函数，则可以使用如下关键字和属性来定义此函数，以通知 Gcc 对此函数进行优化：

```
void function_without_return(void)  _ _attribute_ _ ((noreturn));
```

实际上这个关键字还可以用于指定变量的属性。例如可以使用属性 aligned 指定编译器在分配内存空间时按规定的边界对齐，可以使用属性 packed 指定编译器为普通变量或者结构体变量分配最小的空间。在定义结构体时如果使用了 packed 属性，Gcc 就不会为了对齐不同变量的边界而插入额外的字节。

最后，Gcc 对 case 语句也做了扩展。在 ANSI C 中，case 语句只能对应于单个值的情形。Gcc 中的 case 语句可以对应于一个范围。它使用的语法是在 case 关键字之后列出范围的两个边界值，而边界值之间以空格、省略号和空格分开。例如，如果要对整型变量 i 使用 case 语句，可以写成：

```
switch (i){
case 0 … 10:
     /*处理第一种情况的代码*/
     break;
case 11 … 100:
     /*处理第二种情况的代码*/
     break;
```

```
default:
     /*处理默认情况的代码*/
}
```

这段代码相当于 ANSI C 中的如下代码：

```
switch (i){
case 0 :
…
case10:
     /*处理第一种情况的代码*/
     break;
case 11:
…
case 100:
     /*处理第二种情况的代码*/
     break;
default:
     /*处理默认情况的代码*/
}
```

第 7 章　VxWorks 系统应用实例

前面几章对 VxWorks 系统的基本理论、底层开发、开发环境以及编译器做了详尽的介绍，作为对 VxWorks 系统介绍的重点，本章集中阐述如何在 VxWorks 系统中开发应用程序并给出应用开发的实例。本章的应用实例开发阐述了任务划分的基本思想，同时演示了如何使用 VxWorks 系统中的任务管理、信号量、MUX 接口、网络组件、中断处理、定时管理等核心机制。

7.1　VxWorks 系统中的任务划分

任务是代码运行的一个映像。从系统角度看，任务是竞争系统资源的最小运行单元。任务可以使用或等待 CPU、I/O 设备及内存空间等系统资源，并独立于其他任务，与它们一起并发运行。VxWorks 操作系统内核通过一定的指示来进行任务的切换，这些指示都来自对内核的系统调用。

在应用程序中，任务不仅在表面上具有与普通函数相似的格式，而且还有着自己较明显的特点：

- 任务具有任务初始化的起点（如获取一些系统对象的 ID 等）。
- 具有存放执行内容的私用数据区（如任务创建时明确定义的用户堆栈和堆栈）。
- 任务的主体结构表现为一个无限循环体或有明确的终止条件，它不同于函数，一般无需返回。

在设计较为复杂的多任务应用时，进行合理的任务划分对系统的运行效率、实时性和吞吐量影响极大。任务分解过细会引起任务频繁切换的开销增加，而任务分解不够彻底会造成原本可以并行的操作只能按顺序串行完成，从而减少了系统的吞吐量。为了达到系统效率和吞吐量之间的平衡与折衷，在应用设计中应遵循如下的任务分解规则（假设下述任务的发生都依赖于唯一的触发条件，如两个任务能够满足下面的条件之一，它们可以合理地分开）：

- 时间：两个任务所依赖的周期条件具有不同的频率和时间段。
- 异步性：两个任务所依赖的条件没有相互的时间关系。
- 优先级：两个任务所依赖的条件需要有不同的优先级。
- 清晰性/可维护性：两个任务可以在功能上或逻辑上互相分开。

从软件工程和面向对象的设计方法来看，各个模块（任务）间数据的通信量应该尽量小，并且最好少出现控制耦合（即一个任务可控制另一个任务的执行流程或功能）。如果必须出现，则应采取相应的措施，如任务间通信使它们实现同步或互斥，避免可能引起的临界资源冲突。

在VxWorks系统中，wind内核对应用程序中任务的数量没有具体限制，实际应用时在wind配置表中定义最大的任务数，并为存放有关数据结构分配足够的内存空间。在设计一个复杂

应用时，上面的任务分解原则仅能作初步的参考，真正设计时还需要更多的实际分析和设计经验，才能使系统达到预定性能指标和效率。

7.2　任务间通信机制

本节中我们演示一种任务间最基本的通信机制，即利用消息队列通信。我们在主函数 CreateTask 中创建两个任务 recvMSG_Task 和 sendMSG_Task。这两个任务通过消息队列 TestMSGQId 实现关联。recvMSG_Task 任务等待 sendMSG_Task 任务通过消息队列发来的消息，接到消息以后，recvMSG_Task 便在终端打印接收到的字符串。为了调整消息的发送频率，我们在发送消息的任务体中加了一个较大的 for 循环，保证消息以较慢的频率递送。

本实例程序执行流程如图 7-1 所示。

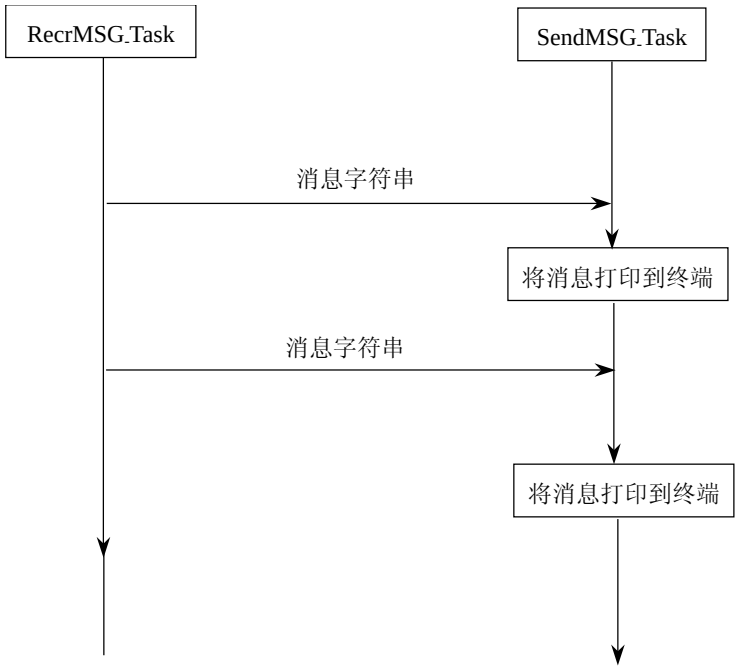

图 7-1　任务间通信程序流程

下面列出实例程序 TaskMSG.c，并做扼要的注释。

```
/***************TaskMSG.c 程序的开始**********************/
      #include "vxworks.h"
      #include "msgQLib.h"
     #include "taskLib.h"
      #include "timers.h"
     #include "string.h"
      #include "stdio.h"
```

```
        void recvMSG_Task(void);
        void sendMSG_Task(void);
        void  CreateTask(void);

        MSG_Q_ID TestMSGQId;

/***********************************************/
/* sendMSG_Task : 此函数用于实现消息发送任务。      */
/***********************************************/

        void sendMSG_Task(void)
        {
        struct Tag_timespec
        {
        time_t      tv_sec;     /* 秒 */
        long        tv_nsec;   /* 纳秒 (0 -1,000,000,000) */
        }current_time;

        int maxMsgs;      /* 队列允许的最大消息数*/
        int maxMsgLength; /* 消息中的最大字节长度 */
        int options;       /* 消息队列属性选项     */

        long loop_num;
       char MSGbuffer[64];
       int  timeout;
        unsigned long CurrentTick;


        maxMsgs=64;
        maxMsgLength=1024;
      options=MSG_Q_FIFO;
        strcpy(MSGbuffer,"The Great China!");
        timeout=100;

         TestMSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
        if(TestMSGQId==NULL)
            {
              printf("it is ERROR for Test msgQCreate!");
                 return;
            }

        for (loop_num=0;;loop_num++)
         {       /*无限发送消息*/
                 if(loop_num==0x1000000)
                  {
                   clock_gettime(CLOCK_REALTIME,(timespec *)&(current_time));
                      printf("current time is %d.\n",current_time.tv_nsec);
                      msgQSend(TestMSGQId,
                                      MSGbuffer,
                 sizeof(MSGbuffer),
```

```
                            timeout, /* ticks to wait */
                            MSG_PRI_NORMAL );
/*CurrentTick=tickGet();
                   printf("current ticks is %d;current loop num is
%d.\n",CurrentTick,loop_num);*/
               clock_gettime(CLOCK_REALTIME,current_time);
                  printf("current time is %d.\n",current_time->tv_nsec);
                  loop_num=0;
               }
    }
    printf("ERROR occurs in Task Sending MSG");
    }

/***********************************************/
/* recvMSG _Task : 此函数用于实现消息接收任务。 */
/***********************************************/

    void recvMSG_Task(void)
    {
    int maxMsgs;               /* 队列允许的最大消息数*/
     unsigned int maxBytes;      /* 消息中的最大字节长度 */

    char MSGbuffer[64];
    int  timeout;
    int  revBytes;

    maxMsgs=64;
    maxBytes=1024;
   timeout=100;

    for (;;)
    {  /*无限等待消息的到来 r*/
         revBytes=msgQReceive(TestMSGQId,MSGbuffer, maxBytes,timeout);
            if(0!=revBytes)
               printf("recv msg is \n %s\n",MSGbuffer);

    }
    printf("ERROR occurs in Task Sending MSG");
    }

/***********************************************/
/* recvMSG _Task : 此函数用于实现消息发送和接收任*/
/*                    务的创建。            */
/***********************************************/

    void  CreateTask(void)
    {
    int RetValue;

RetValue=taskSpawn("send MSG Task",100,VX_PRIVATE_ENV,512,    /#创建消息发送任务*/
```

```
(FUNCPTR)sendMSG_Task,0,0,0,0,0,0,0,0,0,0);
    if (RetValue==ERROR)
            printf("ERROR occurs in Task Creating");

    RetValue=taskSpawn("recv MSG Task",100,VX_PRIVATE_ENV,512,    /#创建接收任务*/
(FUNCPTR)recvMSG_Task,0,0,0,0,0,0,0,0,0,0);
        if (RetValue==ERROR)
             printf("ERROR occurs in Task Creating");
    }
/*********************TaskMSG.c 程序的结尾**********************/
```

7.3 Wind 内核功能

本节我们将演示如何测试 wind 内核提供的系统调用。本节介绍的程序专注于测试 wind 内核的不同功能组件如信号量的同步、互斥原语的使用，任务控制中 taskSuspend()/taskResume()控制功能的使用，消息队列的使用，看门狗（watchdog）在任务定时情况下的应用。

为了练习使用这些内核功能，我们启动了两个任务，一个高优先级任务和一个低优先级任务。高优先级任务执行的功能体需要用到一些目前没有的资源。当因为高优先级任务缺乏资源而阻塞时，低优先级任务获得处理机控制权，同时释放一些高优先级任务等待的资源。这样高优先级任务又可以运行。这样，看来简单的演示却可以测试 VxWorks 系统中的任务上下文切换、任务的调度、任务就绪队列和挂起队列的移位以及定时器队列的变化等。我们测试这些内核功能的原因是大多应用场合经常用到同步、互斥、任务控制、任务间通信以及定时器功能。这些功能体不仅是具体应用的主要组成部分，也是 VxWorks 操作系统本身的重要组成部分。

本实例中高优先级任务和低优先级任务之间交互的流程如下。

- 主程序创建信号量对象。
- 高优先级任务分配信号量，同时获得此信号量。
- 高优先级任务再次尝试获得此信号量。从而发生等待。
- 低优先级任务分配信号量，以唤醒刚刚挂起的高优先级任务。
- 高优先级任务发送消息到消息队列中，同时获得此消息。
- 高优先级任务再次尝试从消息队列中获得消息。从而发生等待。
- 低优先级任务发送消息到消息队列中，以唤醒刚刚挂起的高优先级任务。
- 最后启动 wdstart()。

任务间的交互流程图如图 7-2 所示。

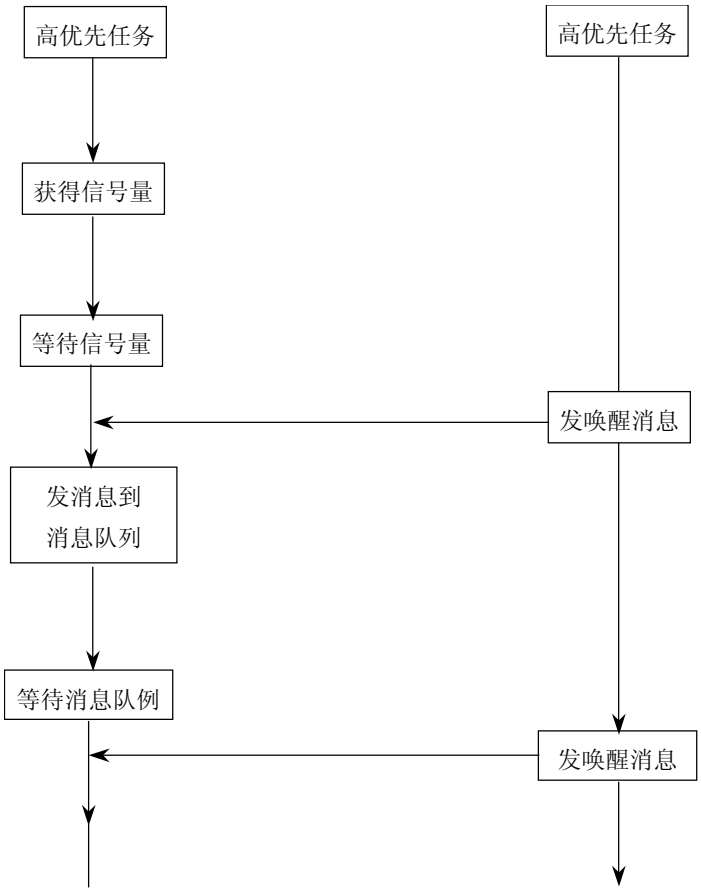

图 7-2　高低优级任务间的交互

最后我们列出本实例的应用程序代码 KernelTest.c，同时做扼要的说明。

```
/*******************KernelTest.c 程序的开始***********************/
#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"

/* defines */
#if FALSE
#define STATUS_INFO             /* 定义允许使用 printf() 调用 */
#endif

#define MAX_MSG         1     /* 队列中的最大消息数目*/
```

```
#define MSG_SIZE    sizeof (MY_MSG)    /* 消息的大小 */
#define DELAY       100       /* 100 ticks */
#define    HIGH_PRI150        /* 高优先级任务的优先级*/
#define LOW_PRI        200         /* 低优先级任务的优先级*/

#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;         /* 发送消息的任务的循环变量 */
    char * buffer;              /* 消息文本*/
    } 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 task ID */
int         lowPriId;             /* 低优先级任务的 task ID */
int         windDemoId;         /* windDemo 任务的 task ID*/

/* 函数声明 */

LOCAL void taskHighPri (int iteration);
LOCAL void taskLowPri (int iteration);

/*************************************************************/
/* windDemo  -  用于孵化子任务的父任务。                      */
/*此任务调用 taskHighPri() 和 taskLowPri()开始实际测试操作，然后将自己挂起，*/
/*任务的恢复由低优先级任务完成。                              */
/*************************************************************/

void windDemo
    (
    int iteration            /* 子任务的编号*/
    )
    {
    int loopCount = 0;             /* 通过 windDemo 循环的次数 */

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

    if (iteration == 0)           /* 将编号的默认值设为 10,000 */
     iteration = 10000;
```

```
    /* 创建子任务使用的对象*/

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

    windDemoId = taskIdSelf ();

    FOREVER
     {

    /* 孵化子任务开始执行内核例程*/

    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);      /* 以后由任务 taskLowPri 唤醒*/

#ifdef    STATUS_INFO
    printf ("\nParent windDemo has just completed loop number %d\n",
         loopCount);
#endif /* STATUS_INFO 的定义改变 */
    loopCount++;
    }
   }

/*************************************************************/
/* taskHighPri  -  高优先级任务                                */
/* 这个任务执行不同的内核函数，当资源不可获得的时候便阻塞并将 CPU     */
/* 控制权转交给下一个就绪的任务                                 */
/*************************************************************/
LOCAL void taskHighPri
    (
    int iteration            /* 循环的次数 */
    )
    {
    int    ix;                /*循环指针*/
    MY_MSG msg;              /*要发送的消息*/
    MY_MSG newMsg;             /*接收到的新消息*/

    for (ix = 0; ix < iteration; ix++)
    {
    /* 获取并给出信号量－不包括任务的上下文切换*/
    semGive (semId);
    semTake (semId, 100);        /* semTake 的超时时间为 100 tick */

    /* 获取信号量，如果信号量不可用则发生任务的上下文切换*/
```

```
    semTake (semId, WAIT_FOREVER);     /* 获取不可用的信号量*/

    taskSuspend (0);              /* 将自己挂起*/

    /* 组装消息并发送*/

    msg.childLoopCount = ix;
    msg.buffer = TASK_HIGHPRI_TEXT;

    msgQSend (msgQId, (char *) &msg, MSG_SIZE, 0, MSG_PRI_NORMAL);

    /*************************************************/
    /* 读取此任务刚刚发送的消息并打印，由于已经有消息到  */
    /* 达消息队列故将发生任务的上下文切换。                */
    /*************************************************/

    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 的定义*/

    /*********************************************************/
    /* 阻塞在消息队列上，等待低优先级任务发来消息。             */
    /* 由于消息队列中没有消息，因此会发生上下文切换。          */
    /* 当真正有消息到来时，便将其打印出来。                   */
    /*********************************************************/

    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 /* 状态信息 */

    /* 测试看门狗定时器。*/

    wdStart (wdId, DELAY, (FUNCPTR) tickGet, 1);

    wdCancel (wdId);
    }
   }

/*******************************************************/
/* taskLowPri ―这是一个低优先级的任务。                  */
/* 这个任务的运行优先级较低，设计此任务的目的是用来分配高     */
/* 优先级任务正在等待的资源，然后为高优先级的任务解除阻塞     */
/* 状态。                                 */
```

```
/*******************************************************/
LOCAL void taskLowPri
    (
    int iteration            /* 循环次数 */
    )
    {
    int    ix;                 /* 循环指针 */
    MY_MSG msg;             /*要发送的消息*/

    for (ix = 0; ix < iteration; ix++)
    {
    semGive (semId);             /* 为任务 tHighPri 解除阻塞*/

    taskResume (highPriId);

    /* 组装消息并发送*/

    msg.childLoopCount = ix;
    msg.buffer = TASK_LOWPRI_TEXT;
    msgQSend (msgQId, (char *) &msg, MSG_SIZE, 0, MSG_PRI_NORMAL);
    taskDelay (60);
    }

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

/**********************KernelTest.c 程序的结尾***********************/
```

7.4　中 断 处 理

本节的实例程序也是一个简单的多任务应用程序。介绍本节实例的意图是让 Tornado 用户迅速掌握不同的 Tornado 工具。这些工具用于调试应用程序功能。

本实例程序模拟一个数据收集系统。其中的数据来源于外部设备，当数据到来时，由中断通知。我们利用一个任务来模拟这个中断服务程序。每次数据到来时，中断服务程序分配一个信号量。实际上我们可以利用 VxWorks 系统中的看门狗定时器简单地替代这个中断服务程序。

当数据到来时，分两个阶段来处理数据。第一阶段，首先收集 NUM_SAMPLE 个数据。第二阶段，对收集的简单数据进行算术运算得到一个结果值。第一个任务通过二元信号量控制第二个任务是否开始第二阶段处理。

第二个任务得到运算结果值的处理过程受另一个任务监视，当运算结果超过安全范围时，第三个任务将打印告警信息。

本实例处理流程图如图 7-3 所示。

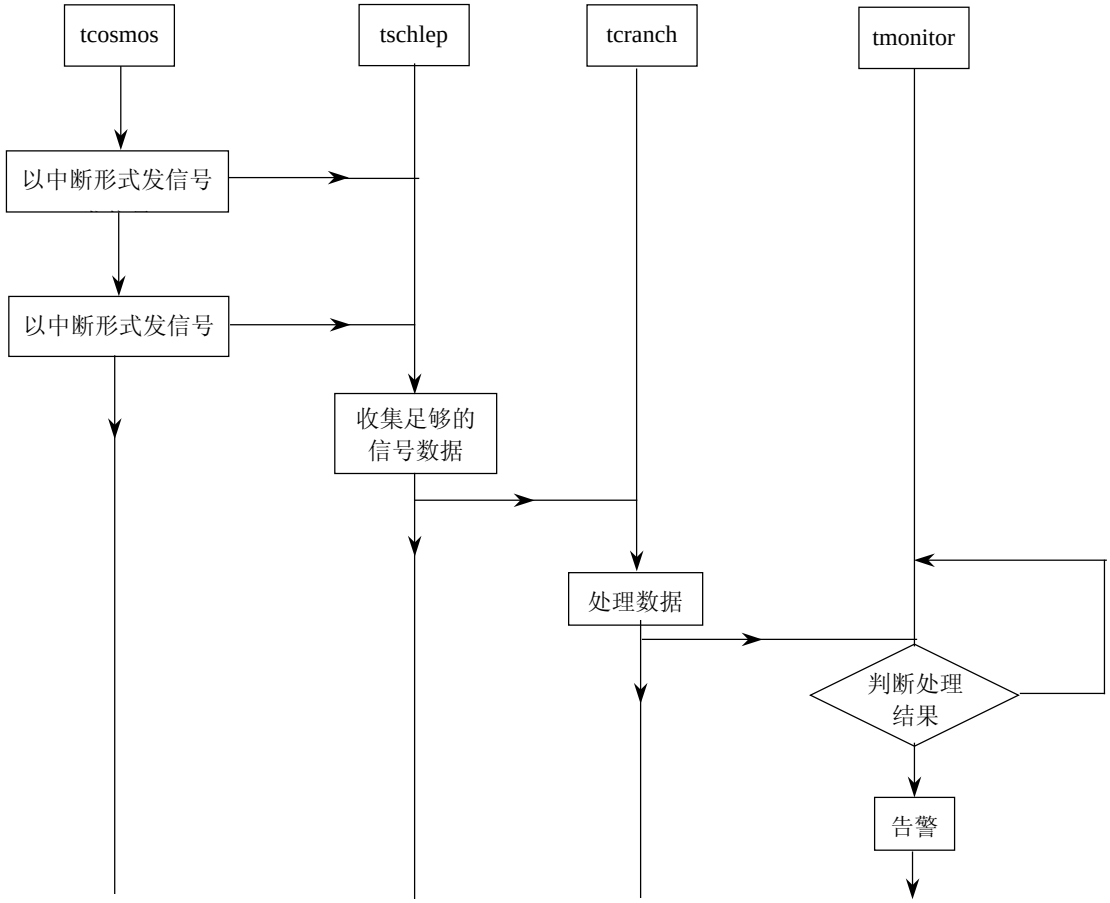

图 7-3　中断处理实例的任务间交互

最后我们列出本实例的应用程序代码，同时做扼要的说明。

```
/*****************Interrupt.c 程序的开始***************************/

/* 包含必要的组件的头文件 */

#include "vxWorks.h"
#include "stdio.h"
#include "stdlib.h"
#include "semLib.h"
#include "taskLib.h"

/* 宏定义 */

#define NUM_SAMPLE      10
#define LUCKY               7
#define HOT                 20
#define DELAY_TICKS      4
```

```
#define STACK_SIZE    20000

/*********************************************/
/* 以下是运行状态的定义，它确保程序可按步关闭的，    */
/* 同时保证没有例程去获取不再存在的信号量。                */
/*********************************************/

#define ALL_GO              0
#define COSMOS_STOP     1
#define SCHLEP_STOP     2
#define CRUNCH_STOP    3
#define ALL_STOP        4

/* 数据结构类型定义 */

typedef struct byLightning
    {
    int           data;
    int           nodeNum;
    struct byLightning * pPrevNode;
    } LIST_NODE;

/* 全局变量定义*/

int tidCosmos;            /* 各任务 ID 的定义 */
int tidSchlep;
int tidCrunch;
int tidMonitor;

int cosmicData = 0;                    /* 持有可读的数据*/
int result = 0;                         /* 保存计算的结果*/

volatile UINT8 runState = ALL_STOP;    /* 控制程序结束*/

LIST_NODE * pCurrNode = NULL;      /* 数据列表的头*/

SEM_ID dataSemId;       /* 如果有数据则给出此信号量  */
SEM_ID syncSemId;       /* 如果有数据实例可以处理，则给出此信号量*/
SEM_ID currNodeSemId;        /* 如果指向当前节点的指针可以访问，则给出此信号量。*/

/* 函数声明*/

void cosmos (void);
void nodeAdd (int data, int nodeNum);
void schlep (void);
void nodeScrap (void);
void crunch (void);
void monitor (void);
void progStop (void);
```

```
/*****************************************************/
/* progStart  -  此函数用于启动其他例程。              */
/* 它首先创建不同的信号量，然后孵化各个任务，在这个过程*/
/* 中，进行必要的出错检查。                            */
/* 返回值：  OK                          */
/*****************************************************/

STATUS progStart (void)
    {
    syncSemId = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    dataSemId = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

    currNodeSemId = semMCreate (  SEM_Q_PRIORITY
                  | SEM_INVERSION_SAFE
                  | SEM_DELETE_SAFE);

    pCurrNode = NULL;    /* 当前节点指针的起始状态*/

    /* 启动各个任务*/

    runState = ALL_GO;

    tidCosmos = taskSpawn ("tCosmos", 200, 0, STACK_SIZE,
        (FUNCPTR) cosmos,0,0,0,0,0,0,0,0,0,0);

    tidSchlep = taskSpawn ("tSchlep", 220, 0, STACK_SIZE,
        (FUNCPTR) schlep,0,0,0,0,0,0,0,0,0,0);

/*********************************************/
    /*tCranch 和 tMonitor 的*
    /*优先级不同是为了保证正确*/
    /*的执行流程*/
    /************************************/

    tidCrunch = taskSpawn ("tCrunch", 240, 0, STACK_SIZE,
        (FUNCPTR) crunch,0,0,0,0,0,0,0,0,0,0);

    tidMonitor = taskSpawn ("tMonitor", 230, 0, STACK_SIZE,
        (FUNCPTR) monitor,0,0,0,0,0,0,0,0,0,0);

    return (OK);
    }

/*********************************************************/
/* cosmos —此函数用于模拟数据到来的中断                  */
/* 此函数直接由任务调用执行，调用任务相当于一个中断服务      */
/* 程序（ISR）。中断的特点是通知某些设备有接收数据。     */
/* 此函数周期性地给出信号量 dataSemId 表示数据到来，这个       */
/* 数据可以从变量 cosmicData 中读取。                    */
/*********************************************************/
```

```
void cosmos (void)
    {
    int nadaNichtsIdx = 0;

    while (runState == ALL_GO)
    {
        if (nadaNichtsIdx != LUCKY)
            cosmicData = rand ();
        else
         {
            cosmicData = 0; /* 无限等待时必须得到有效数据*/
            nadaNichtsIdx = 0;
        }

        ++nadaNichtsIdx;

        /* 利用信号量和延迟保证只读取新到来地数据*/
        semGive (dataSemId);
        taskDelay (DELAY_TICKS);
    }

    runState = SCHLEP_STOP;
    semGive (dataSemId);
    }

/***********************************************/
/* nodeAdd - 此函数将一个新的节点链接到数据链前面。    */
/* 此函数首先申请并初始化一个新节点，然后将其挂接     */
/* 到数据链中。                          */
/* 返回值： N/A                         */
/***********************************************/

void nodeAdd
    (
    int data,
    int nodeNum
    )
    {
    LIST_NODE * node;

    if ( (node = (LIST_NODE *) malloc (sizeof (LIST_NODE))) != NULL)  /*为节点申请内存*/
    {
    node->data = data;
    node->nodeNum = nodeNum;

    semTake (currNodeSemId, WAIT_FOREVER);      /*获取信号量*/
    node->pPrevNode = pCurrNode;
    pCurrNode = node;
    semGive (currNodeSemId);                      /*释放信号量*/
    }
```

```
    else
     {
     printf ("cobble: Out of Memory.\n");
     taskSuspend (0);
     }
     }

/*******************************************************/
/* schlep - 此函数收集数据到一个实例中处理。                */
/* 此函数以周期性无限等待的方式，将获得的 NUM_SAMPLE    */
/* 实例链中然后唤醒数据处理程序。                          */
/*******************************************************/

void schlep (void)
    {
    int nodeIdx;

    FOREVER
    {
    for (nodeIdx = 0; nodeIdx < NUM_SAMPLE; nodeIdx++)
            {
        semTake (dataSemId, WAIT_FOREVER);    /*无限等待 */
        if (runState == SCHLEP_STOP)
        {
        runState = CRUNCH_STOP;
        semGive (syncSemId);
        return;
        }

        nodeAdd (cosmicData, nodeIdx);
            }

    semGive (syncSemId);          /* 唤醒数据处理程序*/
    }
   }

/*************************************************/
/* nodeScrap - 此函数用于归还一个不再用的节点            */
/*************************************************/

void nodeScrap (void)
    {
    LIST_NODE * pTmpNode;

    pTmpNode = pCurrNode;
    pCurrNode = pCurrNode->pPrevNode;
    free (pTmpNode);
    }

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

```
/* crunch - 此函数用于处理数据实例。                          */
/*此函数首先等待 schlepper 程序发来数据实例。                 */
/*然后进行简单的示意性处理。                    */
/*********************************************/

void crunch (void)
    {
    int    sampleSum =0;
    int    div;
    BOOL quit = FALSE;

    while (!quit)
    {
    semTake (syncSemId, WAIT_FOREVER);    /* 等待数据到来 */

    if (runState == CRUNCH_STOP)
        quit = TRUE;

    semTake (currNodeSemId, WAIT_FOREVER);
    /*保留对 pCurrNode 的访问权 */
    while (pCurrNode != NULL)
        {
        sampleSum += pCurrNode->data;
        div = pCurrNode->data;
        nodeScrap ();
        }

    semGive (currNodeSemId);       /* 释放对 pCurrNode 的访问*/

        result = sampleSum / div;          /* 可以进行简单的异常处理*/
/*
    if (div != 0)
            result = sampleSum/div;
*/
        sampleSum = 0;               /* 清除进行下一次处理*/
    }

    runState = ALL_STOP;
    }


/*********************************************/
/* monitor - 此函数用于检查计算的结果。               */
/*此函数首先检查计算的结果然后给出适当的告警提示。*/
/*********************************************/

void monitor (void)
    {
    int isHot = 0;
    int isNot = 0;
```

```
    while (runState == ALL_GO)
    {
        if (!isHot && result >= HOT)
            {
            isHot = 1;
          printf ("WARNING: HOT!\n");
        }
    else if (isHot && result < HOT)
        {
        isHot = 0;
        printf ("OK\n");
        }
    }
    }

/***********************************************/
/* progStop - 此函数用于终止程序。                    */
/* 当需要终止所有应用时，可以调用此函数。               */
/***********************************************/
void progStop (void)
    {
    runState = COSMOS_STOP;

   /* 等待所有程序结束*/
    while (runState != ALL_STOP)
    taskDelay (1);

/*清除信号量*/
    semDelete (dataSemId);
    semDelete (syncSemId);
    semDelete (currNodeSemId);

    printf ("BYE!TSCHUESS!ADIEU!\n");
    }
/****************** Interrupt.c 程序的结尾*****************************/
```

7.5 Sockets 通信

本节我们介绍利用 Sockets 进行网络通信的客户机/服务器应用程序。此应用中包括 3 个源程序。其中“demo.c”是一个简单的函数用于打印自己的任务 ID、任务名和启动参数。“demo.c”的功能是演示。例如，在 VxWorks 的 shell 下面可以利用其功能做如下操作得到任务的 ID 值、任务名和启动参数。

```
-> ld 1 <demo.o
    value = 0 = 0x0
    -> demo       #显示 demo 的任务 Id、任务名和启动参数
    Hello from task 0x3d6cac (shell). Startup parameter was 0.
```

```
value = 2 = 0x2
-> sp demo, $10   #指示、demo 的启动参数为 16
task spawned: id = 0x3c96d8, name = 1
value = 3970776 = 0x3c96d8
-> Hello from task 0x3c96d8 (1). Startup parameter was 16.

-> repeat 4, demo, $a      #将 demo 的作务 Id、任务名和启动参数打印 4 次
task spawned: id = 0x3c96d8, name = 2
value = 0 = 0x0
-> Hello from task 0x3c96d8 (2). Startup parameter was 10.
Hello from task 0x3c96d8 (2). Startup parameter was 10.
Hello from task 0x3c96d8 (2). Startup parameter was 10.
Hello from task 0x3c96d8 (2). Startup parameter was 10.
->
```

“client.c”是本客户机/服务器应用中的客户端程序，它运行在 VxWorks 系统中。如果将 “demo.c”和“client.c”两个程序重新编译在一起可以使用如下命令：

```
> make CPU=<CPU_TYPE>
```

例如，

```
> make CPU=SPARC
```

通过编译我们可以将这两个 VxWorks 目标文件“demo.o”和“client.o”安装到目录 $WIND_BASE/target/lib/objSPARCgnutest/中。在 VxWorks 系统中键入如下命令来装载和启动 client 程序：

```
-> ld < client.o
client "wrs"
```

如果相应的 server 程序已经启动，那么在 VxWorks 系统中键入字符以后，将得到 Unix 系统上所运行 Server 的回应。按下 ctrl-D 键可以终止会话。

本实例中介绍的另外一个源程序是 server.c，它运行在 unix 环境，当然也可以运行在 Red hat Linux 上。如果对 server.c 有任何改动可以利用如下命令来编译：

```
> cc server.c -o server
```

配合 VxWorks 系统，我们在 UNIX 系统开发环境的 Shell 提示符下面键入如下命令启动 server：

```
> server
```

client 和 Server 的执行流程图如图 7-4 所示。

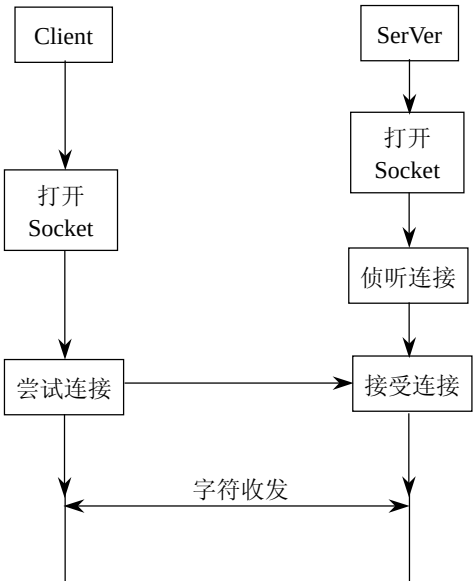

图 7-4　client 和 SerVer 的交互流程

最后我们列出本实例的应用程序代码 demo.c、client.c 和 server.c，同时做扼要的说明。

```
/*************** demo.c 程序的开始*********************/

/*包含头文件*/

#include "vxWorks.h"
#include "taskLib.h"
#include "stdio.h"


/*******************************************************/
/* demo  -  此函数是一个简单的演示 Socket 通信的程序。                */
/* 此函数会打印调用任务的 Task id 、任务名和启动参数。                  */
/*******************************************************/

void demo (param)
    int param;

    {
    /* 找出任务自身的参数，然后打印*/

    printf ("Hello from task %#x (%s). Startup parameter was %d.\n",
        taskIdSelf (), taskName (taskIdSelf()), param);
    }
/****************demo.c 程序的结尾************************/

/***************client.c 程序的开始***************************/
```

```
/* 说明：client.c 程序是简单的 client/server 网络演示程序的客户端任务*/

/*包含头文件*/
#include "vxWorks.h"
#include "fioLib.h"
#include "stdio.h"
#include "unistd.h"
#include "string.h"
#include "usrLib.h"
#include "errnoLib.h"
#include "hostLib.h"
#include "sockLib.h"
#include "socket.h"
#include "inetLib.h"
#include "in.h"

#define    SERVER_NUM    1100    /* 套接字通信的端口号*/

IMPORT char sysBootHost [];    /* VxWorks 保存引导主机的名字 */

/*******************************************************/
/* 将 'clientSock'设置成一个全局变量是为了保证运行在 VxWorks 上*/
/*的客户端程序出错时，套接字可以可靠关闭              */
/*******************************************************/

int clientSock;        /* 为 server 打开的套接字*/

/*******************************************************/
/* client - 此函数是客户端任务的入口。                 */
/* 这是一个简单的客户端程序。它与 server 以套接字的形式相连。    */
/* 它的任务是从标准输入设备读取字符然后通过套接字将它们发送    */
/* 到 server 端。如果读到了“0”字符相当于接收到了 EOF。    */
/* server 运行在 UNIX 主机上，而 client 运行在 VxWorks 上。    */
/* 返回值： OK 或者 ERROR                              */
/*******************************************************/
STATUS client (hostName)
    char *hostName; /* 运行 server 的主机名， 0 表示引导主机 */

    {
    struct sockaddr_in    serverAddr;    /* server 地址*/
    struct sockaddr_in    clientAddr;    /* client 地址*/
    int        nBytes;              /* 从 stdin 读取的字节数 */
    char    c;

    if (hostName == NULL)
     hostName = sysBootHost;

/***********************************/
 /* 给数据结构 sock_addr 全部赋 0。             */
 /* 此操作必须在进行套接字调用以前执行。*/
```

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

    bzero ((char *) &serverAddr, sizeof (serverAddr));
    bzero ((char *) &clientAddr, sizeof (clientAddr));

    /*************************************/
/* 打开一个套接字。我们使用 ARPA 网络地*/
/*址格式和流式套接字，详细格式见 socket.h*/
/*************************************/

    clientSock = socket (AF_INET, SOCK_STREAM, 0);

    if (clientSock == ERROR)
    return (ERROR);

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port   = htons(SERVER_NUM);

    /* 获取 server 的网络地址*/

    if ((serverAddr.sin_addr.s_addr = inet_addr (hostName)) == ERROR &&
    (serverAddr.sin_addr.s_addr = hostGetByName (hostName)) == ERROR)
    {
    printf ("Invalid host: \"%s\"\n", hostName);
    printErrno (errnoGet ());
    close (clientSock);
    return (ERROR);
    }

    printf ("Server's address is %x:\n", htonl (serverAddr.sin_addr.s_addr));

    if (connect (clientSock, (struct sockaddr *)&serverAddr,
            sizeof (serverAddr)) == ERROR)
    {
    printf ("Connect failed:\n");
    printErrno (errnoGet ());
    close (clientSock);
    return (ERROR);
    }

    printf ("Connected...\n");

    /* 重复从标准输入设备中读取字符并发送给套接字口直到碰到 EOF */

    while (TRUE)
    {
    /* 读取字符*/

    if ((nBytes = read (STD_IN, &c, 1)) == 0)
        {
```

```
        /* 当读取到EOF;退出循环. */
        break;
        }
    else if (nBytes != 1)
        {
        printf ("CLIENT read error, %d bytes read\n", nBytes);
        printErrno (errnoGet ());
        break;
        }

    /* 发送字节流到server */

    if (send (clientSock, &c, 1, 0) != 1)
        {
        printf ("CLIENT write error:\n");
        printErrno (errnoGet ());
        }
    }

    /* 从VxWorks 侧关闭套接字*/

    close (clientSock);

    printf ("\n...goodbye\n");
    return (OK);
    }
/*************** client.c 程序的结尾*********************/

/**************server.c 程序的开始*****************************/
/* 说明：server.c 程序用于处理 client/server 网络应用中的服务器端操作     */

static char *copyright = "Copyright over";

/*包含头文件*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#if     defined(HOST_HP)
void bzero (s, n)
    char *s;
    int n;

    {
    memset (s, '\0', n);
    }
#endif

#ifndef  SERVER_NUM
#define    SERVER_NUM    1100    /*应用套接字通信的端口号*/
```

```
#endif

/*******************************************************/
/* main - 此函数是服务器端处理的主程序。                    */
/* 这是一个简单的服务器端程序，它通过套接字与客户端通信。     */
/* 它读取接收到的字符，一次读取一个。然后将它们回应到标准     */
/* 输出设备上。当客户端退出时，此服务器程序也因为从套接字中*/
/*读取了“0”字符而跟着退出。                               */
/*******************************************************/

main ()
    {
    int            sock, snew;     /* 套接字标识 */
    struct sockaddr_in    serverAddr;        /* server 地址 */
    struct sockaddr_in     clientAddr;        /* client 地址*/
    int            client_len;     /* clientAddr 的长度 */
    char        c;
    extern int        errno;               /* 作为 UNIX 出错处理的参考*/


/*************************************/
/* 给数据结构 sock_addr 全部赋 0。            */
 /* 此操作必须在进行套接字调用以前执行。*/
 /*************************************/

    bzero (&serverAddr, sizeof (serverAddr));
    bzero (&clientAddr, sizeof (clientAddr));

 /*************************************/
/* 打开一个套接字。我们使用 ARPA 网络地*/
/*址格式和流式套接字，详细格式见 socket.h*/
/*************************************/

    sock = socket (AF_INET, SOCK_STREAM, 0);

    if (sock == -1)
    exit (1);

    /* 设置网络地址，然后绑定，这样客户端就可以连接*/

    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(SERVER_NUM);

    printf ("\nBinding SERVER\n", serverAddr.sin_port);

    if (bind (sock, (struct sockaddr *)&serverAddr, sizeof (serverAddr))
                                == -1)
    {
    printf ("bind failed, errno = %d\n", errno);
    close (sock);
```

```
    exit (1);
    }

    /* 侦听客户端的连接*/

    printf ("Listening to client\n");
/* 利用标准的 socket 函数 listen()侦听连接*/
    if (listen (sock, 2) == -1)
    {
    printf ("listen failed\n");
    close (sock);
    exit (1);
    }

    /* 客户端已经连接，故接受连接并接收字符*/

    printf ("Accepting CLIENT\n");

    client_len = sizeof (clientAddr);
/* 利用标准的 socket 函数 accept()接受连接*/
    snew = accept (sock, (struct sockaddr *)&clientAddr, &client_len);

    if (snew == -1)
     {
     printf ("accept failed\n");
     close (sock);
     exit (1);
     }

    printf ("CLIENT: port = %d: family = %d: addr = %lx:\n",
         ntohs(clientAddr.sin_port), clientAddr.sin_family,
         ntohl(clientAddr.sin_addr.s_addr));

    /* 重复接收字符并将其他发送到标准输出设备上*/

    for (;;)
     {
     if (recv (snew, &c, 1, 0) == 0)
         {
         /* 客户端已经消失*/
         break;
         }

     putchar (c);
     }

    /*  在 UNIX 侧关闭套接字 */

    close (sock);
    close (snew);
```

```
    printf ("\n...goodbye\n");
    }
/*************** server.c 程序的结尾*************************/
```

7.6 任务多实例应用

本节介绍一个 VxWorks 系统下的图像处理应用程序。目的是向大家介绍 VxWorks 系统中多任务处理的高级话题即任务的多实例（instance）应用。任务的实例化是复杂应用如数据通信、移动通信、路由/交换等中的热门话题。任务的实例化的前提是系统中存在大量相同的事务处理，例如，程控交换机接入的话音用户基本上执行一模一样的流程。因此可以在 VxWorks 系统中编写一份任务程序代码，而且为同类事务中每一个事务创建一个任务。这样不仅可以实现任务之间的负荷分担，而且可以大大简化应用程序的设计过程。

本程序实现一个具有自我调节功能的平面图着色算法。此算法的核心是，给定一个平面图，此算法可以从由六种颜色组成的集合中挑选一种颜色分配到平面图的任意节点上，而且保证平面图上所有相邻节点的颜色不同。

处理方法主要基于两个断言。首先，任何一个平面图都可以通过一种算法转化为有向图。其次，对这样的有向图总可以找到一种算法对其着色，保证相邻节点不出现重复的颜色。

本应用程序分为主机侧程序和目标系统程序两部分。主机侧的程序用于控制和显示目标系统程序的执行过程。两者以 client/server 的形式共存。限于篇幅我们不介绍主机侧的图形界面程序。

在主机侧，必须利用如下脚本在 Tornado 环境中安装环境变量。

```
############################################################
#!/bin/sh
# 首先检查环境变量

wb_message="this environment variable must be set to the root location \
of the Tornado tree."
wr_message="this environment variable must be set to the host type \
on which the Tornado Registry daemon is running."

: ${WIND_BASE?$wb_message} ${WIND_REGISTRY?$wr_message}

usage="usage: vxColor tgtsvr [-V.erbose]"

TCL_LIBRARY=${WIND_BASE}/host/tcl/tcl
export TCL_LIBRARY        #此环境变量标识 TCL 库的路径

TCLX_LIBRARY=${WIND_BASE}/host/tcl/tclX
export TCLX_LIBRARY       #此环境标识 TCLX 库的路径
```

```
TK_LIBRARY=${WIND_BASE}/host/tcl/tk
export TK_LIBRARY          #此环境变量标识 TK 库的路径

TKX_LIBRARY=${WIND_BASE}/host/tcl/tkX
export TKX_LIBRARY         #此环境变量标识 TKX 库的路径

# usefull proc
msg()           # 打印出错信息到标准输出
{
    echo "$@" >&2
}

# 检查命令行
if ( (test $# -lt 1) || (test $# -gt 2) ) then
    msg ""
    msg "$usage"
    msg ""
    exit 0
fi

demoFile=$WIND_BASE/host/src/demo/color/demoHost.tk      #   定位 demo File
#根据命令行内容获得所需参数
if (test $# -eq 2) then
    wtxwish -f $demoFile $1 $2
else
    wtxwish -f $demoFile $1
fi
############################################################
```

在目标系统侧，本应用程序创建的任务主要有 graphNodeColoring、graphControlEnd、graphControl 和 graphStartColoring。其中 graphNodeColoring 是一个可以实例化的任务，每一个任务负责为平面图的一个节点着色。graphControlEnd 任务负责监视任务的终结，当任务异常终止时它通知主机侧然后终止本身。GraphControl 任务是一个图像监视任务，它负责控制各图像节点任务的执行。所有 graphNodeColoring 任务均是由 graphNodeCreate 创建的，当系统判定要为某个节点着色时便启动一个单独的 graphNodeColoring 任务。任务 graphStartColoring 用于请求所有 graphNodeColoring 任务开始本节点的着色工作。

在目标系统侧，应用程序除了用到了 VxWorks 系统内核提供的任务、消息队列等之外，还使用几个复杂的函数例如 graphNeighbourTalk 、graphConnexionCreate 和 graphFullDuplexCreate 等。这些函数当中有些实现了复杂的算法。为了便于大家理解，我们在后续的代码介绍中将进行详细的说明。

在目标系统侧，任务间的执行流程如图 7-5 所示。

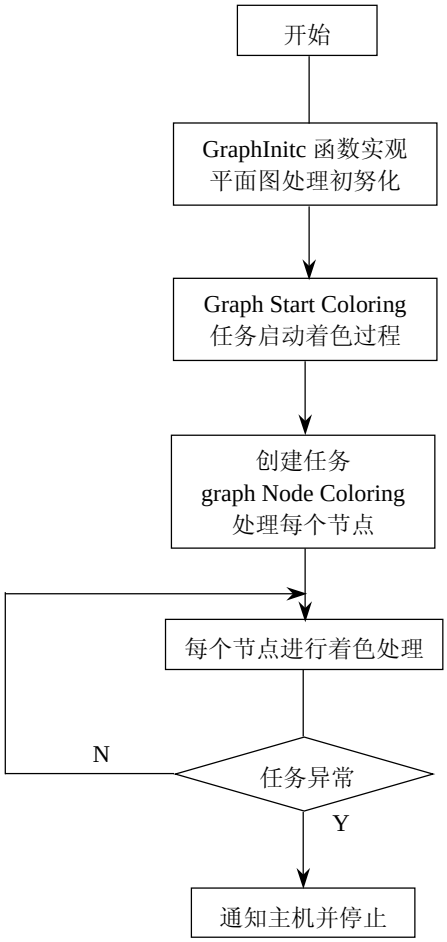

图 7-5　着色处理的简单流程。

在目标系统侧，本应用程序只设计了两个函数即 vxColor.h 和 vxColor.c。下面列出这两个程序，并做扼要注释说明。

```
/****************** vxcolor.h 程序的开始*********************/
/* 说明：vxColor.h 程序是 vxColor.c 的头文件*/

/* 宏定义*/

#define    MAX_NODE    100                /*图形中的最大节点数 */
#define    MAX_CONNEX    10                    /* 最大节点连接数 */
#define    MAX_MSG        1                /* 消息队列中的最大消息数 */
#define    MSG_SIZE    sizeof (NODE_ATTRIB)        /* 消息大小 */
#define    MAX_COLOR    6            /* 颜色的最大数目*/
#define    MAX_OUTDEGREE    5    /* 节点最大分角数*/
#define    OUTDEGREE_INIT    (MAX_OUTDEGREE + 1)
#define    PAUSE_CLK_RATIO    10      /* clckRate 因子*/
#define    DEMO_EOLM    127           /* 行结束标志*/
```

```
#define    CONT_PRIORITY    200            /* 控制程序的任务优先级*/
#define    NODE_PRIORITY    200           /* 节点处理程序的任务优先级*/
#define    NODE_STBL_INIT  -1            /* stable 字段的初始值*/
#define    NODE_UNSTABLE    0        /* stable 字段的非静态值*/
#define    NODE_STABLE    1          /* stable 字段的静态值 */

/* 以下各堆栈的大小定义与 CPU 类型和调试模式有关。*/

#if (defined (CPU_FAMILY) && (CPU_FAMILY==MC680X0))
#define    NODE_STACK    500        /*节点任务的堆栈大小 */
#define    NODE_STACK_DBG    1000  /* 调试模式时节点任务的节点任务的堆栈大小*/
#define    CONT_STACK    500        /* graphControl 任务的堆栈大小*/
#define    CONT_STACK_DBG    1000  /* 调试模式时 graphControl 任务的堆栈大小*/
#endif  /* CPU_FAMILY == MC680X0 */

#if (defined (CPU_FAMILY) && (CPU_FAMILY==SIMSPARCSUNOS))
#define   NODE_STACK 2500      /*节点任务的堆栈大小 */
#define   NODE_STACK_DBG 5000 /* 调试模式时节点任务的节点任务的堆栈大小*/
#define   CONT_STACK 2500      /* graphControl 任务的堆栈大小*/
#define   CONT_STACK_DBG 5000 /* 调试模式时 graphControl 任务的堆栈大小*/
#endif  /* CPU_FAMILY == SPARC */

#if (defined (CPU_FAMILY) && (CPU_FAMILY==I80X86))
#define   NODE_STACK 500        /*节点任务的堆栈大小 */
#define   NODE_STACK_DBG 1000  /* 调试模式时节点任务的节点任务的堆栈大小*/
#define   CONT_STACK 500        /* graphControl 任务的堆栈大小*/
#define   CONT_STACK_DBG 1000  /* 调试模式时 graphControl 任务的堆栈大小*/
#endif  /* CPU_FAMILY == I80X86 */

#if (defined (CPU_FAMILY) && (CPU_FAMILY==I960))
#define   NODE_STACK 1000       /*节点任务的堆栈大小 */
#define   NODE_STACK_DBG 2000  /* 调试模式时节点任务的节点任务的堆栈大小*/
#define   CONT_STACK 1000       /* graphControl 任务的堆栈大小*/
#define   CONT_STACK_DBG 2000  /* 调试模式时 graphControl 任务的堆栈大小*/
#endif  /* CPU_FAMILY == I960 */

#if (defined (CPU_FAMILY) && (CPU_FAMILY==SIMSPARCSOLARIS))
#define   NODE_STACK 5000       /*节点任务的堆栈大小 */
#define   NODE_STACK_DBG 10000    /* 调试模式时节点任务的节点任务的堆栈大小*/
#define   CONT_STACK 5000       /* graphControl 任务的堆栈大小*/
#define   CONT_STACK_DBG 10000    /* 调试模式时 graphControl 任务的堆栈大小*/
#endif  /* CPU_FAMILY == SIMSPARCSOLARIS */

/* 堆栈大小的默认值 */
#ifndef NODE_STACK
#define   NODE_STACK  2500        /*节点任务的堆栈大小 */
#define   NODE_STACK_DBG  5000   /* 调试模式时节点任务的节点任务的堆栈大小*/
#define   CONT_STACK  2500        /* graphControl 任务的堆栈大小*/
#define   CONT_STACK_DBG  5000   /* 调试模式时 graphControl 任务的堆栈大小*/
#endif /* not def NODE_STACK */
```

```
#if (defined (CPU_FAMILY) && (CPU_FAMILY==I960) && (defined __GNUC__))
#pragma  align 1                /*要求 gcc 不用优化对齐*/
#endif  /* CPU_FAMILY==I960 */

/* 数据结构类型定义*/

typedef struct ids /* ID 定义*/
    {
    INT32 tid;        /* 任务 id */
    INT32nid;        /* 节点 id */
    } IDS;

typedef struct node_attrib  /* NODE_ATTRIB 的定义 */
    {
    INT32 color;        /* 节点的颜色 */
    INT32 Xvalue;          /* 节点的 Xvalue 值 */
    } NODE_ATTRIB;           /* 节点属性*/

typedef struct connect_info /* CONNECT_INFO 结构的定义*/
    {
    INT32dir;             /* 连接方向*/
    IDS      tnid;        /* 连接节点的 tid & nid */
    NODE_ATTRIB    att;        /* 连接节点的属性*/
    MSG_Q_ID    wMQId;       /* 写消息队列的 id */
    MSG_Q_ID    rMQId;        /* 读消息队列的 id */
    } CONNECT_INFO;           /* 连接节点信息*/

typedef struct gnode_st /* GNODE 的定义 */
    {
    INT32        stable;            /* 颜色稳定性标记    */
    IDS               tnid;    /* 节点的 tid & nid */
    NODE_ATTRIB        att;       /* 节点的颜色和 Xvalue 值 */
    INT32     pc;             /* 节点程序指针 */
    INT32     oD;             /* 节点分角*/
    INT32     cNum;                /* 节点连接数*/
    CONNECT_INFO    cArray [MAX_CONNEX];    /* 节点连接数组*/
    struct gnode_st *  pNext;           /* 指向下一个节点的指针 */
    } GNODE;                  /* 全局节点信息 */

typedef struct control /* CONTROL 结构的定义*/
    {
    char        steadyState;                    /* 图形是否稳定的标志*/
    char        controlAlive;                   /* 控制任务是否活动的标志*/
    char    coloringStop;                       /* 是否请求停止的标志*/
    char        pauseOn;                       /* 暂停请求的标志*/
    INT32       graphControlId;             /* graphControl 任务的任务 Id */
    INT32       nodeFailure;                /* 节点处理任务的出错标记 */
    SEM_ID    nodeJobDoneSem;        /* 控制程序的同步信号量*/
    SEM_ID      createAckSem;              /* 请求节点创建的信号量*/
```

```
    SEM_ID    restartSem;        /* 启动着色和控制同步的信号量*/
    SEM_ID    connexionsDoneSem;  /* 同步信号量*/
    SEM_ID    controlerFlushSem;        /* 控制程序的刷新信号量*/
    SEM_ID      dataBaseMutex;         /* 数据库互斥信号量 */
    } CONTROL;            /* 全局控制信息*/

typedef struct data_base /* DATA_BASE 的定义 */
    {
    CONTROL  cT;        /* 控制信息块*/
    INT32    nNum;            /* 图形节点号*/
    GNODE *     pFirstGnode;        /* 指向第一个节点的指针*/
    } DATA_BASE;             /* 图形信息*/

/*关闭 i960 cpu 系列的对齐要求*/

#if (defined (CPU_FAMILY) && (CPU_FAMILY==I960) && (defined __GNUC__))
#pragma align 0
#endif  /* CPU_FAMILY==I960 */

/*函数声明，主要声明了 GUI API */

INT32    graphColorUpdate (DATA_BASE * pdB, char * colorMemBlkAdd);  /*图形颜色更新*/
INT32    graphInit (INT32 nodNumber, char * colorMemBlkAdd, BOOL debugMode);  /*图形初始
化*/
INT32    graphStartColoring (DATA_BASE * pdB);  /*图形着色开始*/
INT32     graphStop (DATA_BASE * pdB);        /*图形着色停止*/

/****************** vxcolor.h 程序的结尾**********************/

/****************** vxcolor.c 程序的开始*****************/
/*说明：vxColor.c 程序是图形处理程序的源程序*/

/* 包含文件*/

#include "vxWorks.h"
#include "taskLib.h"
#include "msgQLib.h"
#include "tickLib.h"
#include "kernelLib.h"
#include "sysLib.h"
#include "string.h"
#include "vxColor.h"

/* 除 GUI API 以外的函数定义 */

INT32    graphConnexionCreate (DATA_BASE * pdB, GNODE * pNd, GNODE * pNd2);
BOOL    graphConsistanceTest (DATA_BASE * pdB);
void  graphControl (DATA_BASE * pdB);
INT32    graphFullDuplexCreate (DATA_BASE * pdB, GNODE * pNd, GNODE * pNd2);
void  graphItoa (INT32 integerToConvert, char resultingString[]);
```

```
INT32     graphNeighbourTalk (GNODE * pNode);
GNODE * graphNodeByIdGet (DATA_BASE * pdB, INT32 nodeId);
void  graphNodeColoring (INT32 nodeId, GNODE * pNode, DATA_BASE * pdB);
INT32    graphNodeColoringStep (GNODE * pNode);
INT32    graphNodeCreate (DATA_BASE * pdB, GNODE * pNode, INT32 nodeId,
         INT32 stackSize);

/*****************************************/
/* graphNodeColoring －节点着色任务。          */
/* 图形中的每个节点都必须执行此任务。     */
/*****************************************/

void graphNodeColoring
    (
    INT32 nodeId,         /* 节点的 id */
    GNODE *  pNode,     /* 此任务处理的节点*/
    DATA_BASE * pdB     /* 指向数据库的指针 */
    )
    {
    /* 初始化随机数发生器*/

    srand (tickGet () + taskIdSelf ());

    /* 节点的初始化，初始 Xvalue 随机选取 */

    pNode->tnid.tid    = taskIdSelf ();
    pNode->tnid.nid    = nodeId;
    pNode->cNum     = 0;
    pNode->pc         = 0;
    pNode->stable     = NODE_STBL_INIT;
    pNode->oD         = OUTDEGREE_INIT;
    pNode->att.color    = 0;
    pNode->att.Xvalue    = rand ();

    /* 向控制程序进行起始汇报*/

    semGive (pdB->cT.createAckSem);

    /************************************************/
    /* 在进入循环以前，首先等待任务 graphStartColoring 发来的*/
    /*所有连接已经建立的消息。                      */
    /************************************************/

    semTake (pdB->cT.connexionsDoneSem, WAIT_FOREVER);

    FOREVER
     {
     /* 每个节点阻塞在由控制程序递送来的同一个信号量上。*/

     semTake (pdB->cT.controlerFlushSem, WAIT_FOREVER);
```

```
     /* 执行一个单一的着色算法处理，如果出错则向控制程序告警*/

        if (graphNodeColoringStep (pNode) == ERROR)
         pdB->cT.nodeFailure = 1;

     /* 作为同步，向控制程序发送一个结束着色的信号*/

     semGive (pdB->cT.nodeJobDoneSem);
     }
    }

/***********************************************/
/* graphNodeColoringStep 一此函数处理着色的单一步骤  */
/*返回值： OK 或者 ERROR                          */
/***********************************************/

INT32 graphNodeColoringStep
    (
    GNODE * pNode            /* 当前节点*/
    )
    {
    INT32ix;            /* 索引号*/
    INT32outDegree;     /* 此节点的分角*/
    INT32colConf;              /* 是否发生颜色冲突 */
    INT32col [MAX_COLOR];    /* 颜色检查数组*/
    INT32    XvalueMax;       /*节点最大值*/

    /* 判断节点与其所有邻节点的交换是否已经完成*/

    if (pNode->pc < pNode->cNum)
        {
    /*****************************************************/
    /* 继续进行扫描过程，即继续与本节点相连的节点交换信息。       */
    /*****************************************************/

    /* 与邻节点的会话*/

    if (graphNeighbourTalk (pNode) == ERROR)
        return ERROR;

    /* 修改程序指针*/

    (pNode->pc)++;
        }

    /* 当与邻节点的会话结束以后开始下面的处理*/

    else
        {
```

```
    /*********************************************************/
/*采用如下方案进行图形转换和着色处理。首先计算 outdgree，*/
/* 判断是否可以根据图形理论将一个平面图（PlanarGraph）转化为    */
/* 一个有向图(Directed  Acyclic Graph )。然后计算各个连接节点的*/
/* 方向以及节点的 outDegree。标识连接方向时采用“+”和“－”     */
/* 分别表示正向连接和反向连接。从当前节点的角度上来说，从本节*/
/* 点起始连接到其他连接节点时可以以正向来标识，其他连接节点到*/
/* 达本节点的连接可以以反向来标识。“+”和“－”方向也可以理解*/
/* 为'outcoming'(外向)和 'incoming'（内向）。当前节点的 outdegree 定*/
/* 义外向连接节点的数目，也可以叫做节点外向分角数。          */
/*********************************************************/

    XvalueMax = 0;
       outDegree = 0;

    for (ix = 0; ix < pNode->cNum; ix++)
        {

/*****************************************************/
 /* 基于 Xvalues 建立连接方向。             */
/*****************************************************/

        if ((pNode->att.Xvalue < pNode->cArray[ix].att.Xvalue) ||
        ((pNode->att.Xvalue == pNode->cArray[ix].att.Xvalue) &&
         (pNode->tnid.nid < pNode->cArray[ix].tnid.nid)))
        {
        pNode->cArray[ix].dir = '+';
        outDegree++;

        /* 存储当前的最大 Xvalue */

        if (pNode->cArray[ix].att.Xvalue > XvalueMax)
            XvalueMax = pNode->cArray[ix].att.Xvalue;
            }
        else
        pNode->cArray[ix].dir = '-';
        }

    pNode->oD = outDegree;

     /*判断是否将图形转换为有向图的*/

        if (outDegree > MAX_OUTDEGREE)

/*******************************************************/
/* PG to DAG conversion: 当前节点的 Xvalue 与其他邻节点的    */
/*Xvalue 相比具有最小的值。因此当前节点的 Xvalue 的改变      */
/*将影响到下一次 outdegree 的计算。                    */
 /*******************************************************/
```

```
        pNode->att.Xvalue = XvalueMax + 1;
else
    {
    /*******************************************************/
    /* 着色处理的方法是判断是否发生颜色冲突，然后更具需要更新    */
    /*  颜色和稳定标志。                                   */
    /*******************************************************/

    /* 初始化冲突检查标志为 false */

        colConf = 0;

    /* 初始化颜色标志 */

for (ix = 0; ix < 6; ix++)
    col[ix] = 'y';

    /* 检查颜色冲突 */

for (ix = 0; ix < pNode->cNum; ix++)
     {

     /* 只处理外向邻节点*/

         if (pNode->cArray[ix].dir == '+')
         {
         /* 当连接节点具有相同的颜色，便发生了颜色冲突*/

          if (pNode->att.color == pNode->cArray[ix].att.color)
             colConf = 1;

          /* 标记连接节点使用的颜色*/

             col [pNode->cArray[ix].att.color] = 'n';
           }
        }

    /* 如果发生了颜色冲突更新颜色和稳定标记*/

    if ( colConf == 1 )
    {
    /* 查询第一个未用的颜色*/

    ix = 0;

    while(ix < 6 && col[ix] == 'n')
        ix++;

    if (ix == 6)
        return ERROR;
```

```
            else
                pNode->att.color = ix;

            pNode->stable = NODE_UNSTABLE;
                }
            else
            pNode->stable = NODE_STABLE;
            }

        /* 复位当前节点的程序指针"program counter" */

        pNode->pc = 0;
            }
        return OK;
        }

    /*************************************************************************/
    /* graphNeighbourTalk 一此函数完成与邻节点的会话。                          */
    /* 此函数完成当前活动节点与其邻接节点的信息交互。交互方式可以采用客户机      */
    /* 服务器模式。但是为了简单和方便我们采用读写模式。活动节点向所有邻节点      */
    /* 写消息和读消息。只有来自邻节点的相关消息保存在本地拷贝中，其他消息全部    */
    /* 舍弃。                                          */
    /* 返回值： OK 或者 ERROR                                  */
    /*************************************************************************/
    INT32 graphNeighbourTalk
        (
        GNODE *  pNode       /* 当前节点 */
        )
        {
        NODE_ATTRIB  att;    /* 节点属性 */
        INT32ix;               /* 连接节点索引*/
        INT32msgNum;        /* 消息队列中的消息数*/

        att = pNode->att;

        /* 写向所有的连接节点*/

        for (ix = 0; ix < pNode->cNum; ix++)
         {
        if ((msgNum = msgQNumMsgs (pNode->cArray[ix].wMQId)) == ERROR)
            return ERROR;
        if (msgNum < MAX_MSG)
            {
            if (msgQSend (pNode->cArray[ix].wMQId, (char *) &att, MSG_SIZE, \
             WAIT_FOREVER, MSG_PRI_NORMAL) == ERROR)
             return ERROR;
           }
          }

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

```
    /* 读所有连接节点，并拷贝有效信息。*/
    /***********************************/

    for (ix = 0; ix < pNode->cNum; ix++)
    {
    if ((msgNum = msgQNumMsgs (pNode->cArray[ix].rMQId)) == ERROR)
        return ERROR;
    if (msgNum > 0)
        {
        if (msgQReceive (pNode->cArray[ix].rMQId, (char *) &att, \
        (UINT) MSG_SIZE, WAIT_FOREVER) == ERROR)
        return ERROR;
       if (ix == pNode->pc)
        pNode->cArray[ix].att = att;
        }
    }
    return OK;
    }

/*****************************************************/
/* graphConsistanceTest 一判断图形颜色的连续性。              */
/* 返回值： 如果图形连续返回 TRUE，否则返回 FALSE。             */
/*****************************************************/

BOOL graphConsistanceTest
    (
    DATA_BASE * pdB    /* 指向数据库的指针*/
    )
    {
    INT32     i;              /* 节点索引*/
    GNODE *    pNode;    /* 指向当前节点的指针*/
    GNODE *    pNodec;   * 指向当前连接节点的指针*/

    pNode = pdB->pFirstGnode;
    while (pNode != NULL)
     {
     for(i = 0; i < pNode->cNum; i++)
         {
         /* 判断颜色冲突*/

         if ((pNodec = graphNodeByIdGet (pdB, pNode->cArray[i].tnid.nid))
         == NULL)
         return FALSE;

         if (pNode->att.color == pNodec->att.color)
          return FALSE;
         }
    pNode = pNode->pNext;
        }
    return TRUE;
```

```
    }


/*******************************************************/
/* graphControlEnd 一此函数用于监视图形处理的终止。        */
/* 当查询全局状态变量发现控制任务异常终止时，此函数通知主机 */
/* 侧的显示程序，然后自己退出。用到的全局变量是      */
/* pdB->cT.controlAlive                          */
/*******************************************************/

void graphControlEnd
    (
    DATA_BASE * pdB /* 指向数据库的指针*/
    )
    {
    pdB->cT.controlAlive = 0;
    taskDelete (pdB->cT.graphControlId);
    }


/*********************************************/
/* graphControl 一图形处理监督任务。          */
/*此任务用于控制节点任务的处理操作。          */
/*********************************************/
void graphControl
    (
    DATA_BASE * pdB /* 指向数据库的指针*/
    )
    {
    GNODE * pNode;        /* 指向当前节点的指针 */
    INT32delay;        /* 等待延迟*/
    INT32ix;      /* 节点索引 */

    /* 设置controlAlive 标志，主机可以查询此信息*/

    pdB->cT.controlAlive = 1;

    /* 初始化 steadyState 标志 */

    pdB->cT.steadyState = 0;

    /* 初始化 pdB->cT.pauseOn 标志*/

    pdB->cT.pauseOn = 0;

    /* 初始化 pdB->cT.coloringStop 标志*/

    pdB->cT.coloringStop = 0;

    /*初始化 pdB->cT.nodeFailure 标志 */
```

```
    pdB->cT.nodeFailure = 0;

    /*等待直到 graphStartColoring 传来信号量 restartSem */

    semTake (pdB->cT.restartSem, WAIT_FOREVER);

    delay = (sysClkRateGet () / PAUSE_CLK_RATIO);

    FOREVER
    {
    /* 检查节点的失败 */

    if ( pdB->cT.nodeFailure == 1)
        graphControlEnd (pdB);

    /* 检查停止请求*/

    if ( pdB->cT.coloringStop == 1 )
        goto stopLabel;

/* 检查暂停请求*/

    if ( pdB->cT.pauseOn == 1 )
        {
        taskDelay (delay);
        goto pauseLabel;
        }

    /* 为所有节点解除阻塞*/

    if (semFlush (pdB->cT.controlerFlushSem) == ERROR)
        graphControlEnd (pdB);

    /* 在节点任务之后重新调度启动控制任务*/

    taskDelay (0);

    /* 检查稳定性*/

    pNode = pdB->pFirstGnode;
    while (pNode != NULL && pNode->stable == NODE_STABLE)
        pNode = pNode->pNext;

    if (pNode == NULL && graphConsistanceTest (pdB))
        {
        /* 节点已经稳定，而且图形颜色连续。*/

        pdB->cT.steadyState = 1;
```

```
stopLabel:

        /*尝试得到同步信号，失败则挂起。*/

        semTake (pdB->cT.restartSem, WAIT_FOREVER);

        /* unblocks all the nodes in an atomic way */

        if (semFlush (pdB->cT.controlerFlushSem) == ERROR)
            graphControlEnd (pdB);

        pdB->cT.steadyState = 0;
        }

/*****************************************************/
/* 等待所有节点完成当前着色步骤，对于正确地完成着色来说，      */
/* 这样做并不是必要的，但是在调试模式下，为了监视一个节点      */
/* 任务并让控制任务等待一个单一的着色步骤的完成，这样做比      */
/* 较方便。                                                  */
/*****************************************************/

    for (ix = 0; ix < pdB->nNum; ix++)
        semTake (pdB->cT.nodeJobDoneSem, WAIT_FOREVER);

pauseLabel:
    }
    }

/*************************************************/
/* graphNodeCreate - 此函数用于创建节点任务          */
/* 为一个给定的节点孵化一个任务。                    */
/* 返回值： OK 或者 ERROR。                          */
/*************************************************/

INT32 graphNodeCreate
    (
    DATA_BASE * pdB,        /* 指向数据库的指针*/
    GNODE *    pNode,       /*指向新节点的指针 */
    INT32    nodeId,            /* 创建节点的 Id   */
    INT32    stackSize          /* 节点任务的堆栈大小*/
    )
    {
    char    name[9];   /* 任务名字符串*/
    char    idNum[4];

    /* 组装任务名*/

    graphItoa (nodeId, idNum);
    strcpy (name, "tNode");
    strcat (name, idNum);
```

```
/*********************************************/
/* 孵化 graphNodeColoring 任务，并赋予它较低的优先级。        */
/* 在 shell 下孵化此任务，让它的优先级为 100。                 */
/*********************************************/

    pNode->tnid.tid =
    taskSpawn ((char *) name, NODE_PRIORITY, VX_SUPERVISOR_MODE, \
           stackSize, (FUNCPTR) graphNodeColoring, nodeId, (INT32) pNode, \
           (INT32 ) pdB, 0, 0, 0, 0, 0, 0, 0);

    if (pNode->tnid.tid == ERROR)
    return ERROR;

    return OK;
    }

/*******************************************/
/*graphItoa  - 此函数将整数转化为字符串。      */
/*******************************************/

void graphItoa
    (
    INT32n,      /*  要转化的正整数值*/
    Char  s[]    /* 转化结果字符串*/
    )
    {
    INT32       i, j, c; /*转化操作过程中用到的索引*/

    i = 0;
    do
    {
    s[i++] = n%10 + '0';
    }
    while ((n /= 10) > 0);

    s[i] = '\0';

    /* 完成字符反转 */

    for (i = 0, j = strlen (s)-1; i < j; i++, j--)
    c = s[i], s[i] = s[j], s[j] = c;
    }

/*********************************************************/
/*graphNodeByIdGet 一此函数以节点 ID 访问数据库中的节点项       */
/* 此函数在以节点 ID 访问数据库后会做适当的检查，然后返回节点块。*/
/* 返回值：  正常情况下返回找到的节点，否则返回 NULL。          */
/*********************************************************/
```

```
GNODE * graphNodeByIdGet
    (
    DATA_BASE * pdB,     /* 指向数据库的指针*/
    INT32nodeId           /* 节点 id */
    )
    {
    GNODE *  pNode;     /* 指向要检查的节点的指针*/

    /* 以 nodeId 为参数查找节点链表*/

    pNode = pdB->pFirstGnode;
    while (pNode != NULL && pNode->tnid.nid != nodeId)
     pNode = pNode->pNext;

    /* 返回在数据库中找到的节点的指针，如果没有找到则返回 NULL */

    return pNode;
    }

/*******************************************************/
/* graphFullDuplexCreate－此函数建立两个节点之间的双向连接              */
/* 返回值 ： OK 或者 ERROR。                                  */
/*******************************************************/
INT32 graphFullDuplexCreate
    (
    DATA_BASE * pdB,      /* 指向数据库的指针*/
    GNODE *    pNode1,   /* 指向第一个节点的指针*/
    GNODE *    pNode2   /* 指向第二个节点的指针*/
    )
    {
    /* 创建两个连接 */

    if (graphConnexionCreate (pdB, pNode1, pNode2) == ERROR ||
        graphConnexionCreate (pdB, pNode2, pNode1) == ERROR)
    return ERROR;

    /*更新数据库*/

    (pNode1->cNum)++;
    (pNode2->cNum)++;

    /*向控制任务汇报已经成功建立双向连接*/

    semGive (pdB->cT.createAckSem);

    return OK;
    }

/*******************************************************/
/* graphConnexionCreate －此函数用于在两个节点之间建立单向连接。        */
```

```
/* 返回值 ： OK 或者 ERROR。                                          */
/*************************************************************/

INT32 graphConnexionCreate
    (
    DATA_BASE * pdB,      /* 指向数据库的指针*/
    GNODE * pNode1,      /* 指向第一个节点的指针*/
    GNODE * pNode2     /* 指向第二个节点的指针*/
    )
    {
    MSG_Q_IDmqId;      /* 使用的消息队列 id */

    pNode1->cArray [pNode1->cNum].tnid = pNode2->tnid;

    /*将方向变量 dir 初始化为没有方向*/

    pNode1->cArray [pNode1->cNum].dir = '0';

    /* 判断连接节点数是否超过限制*/

    if ((pNode1->cNum + 1) == MAX_CONNEX)
     return ERROR;

    /* 为节点 1 -> 节点 2 之间的连接创建消息队列*/

    if ((mqId = msgQCreate (MAX_MSG, MSG_SIZE, MSG_Q_PRIORITY)) == NULL)
     return ERROR;

    /* 填充数据库*/

    pNode1->cArray [pNode1->cNum].wMQId = mqId;
    pNode2->cArray [pNode2->cNum].rMQId = mqId;
    return OK;
    }

/*************************************************************/
/*graphColorUpdate  -  此函数用于更新区域内的颜色。                  */
/* 当收到主机请求时，将新的颜色指配给区域。新颜色从申请的内存块中*/
/*读取，并由主机写给目标。此函数属于 GUI API。                      */
/* 返回值：  OK.                                          */
/*************************************************************/
INT32 graphColorUpdate
    (
    DATA_BASE * pdB,     /* 指向数据库的指针*/
    char * blkAdd           /* 颜色内存块的起始地址*/
    )
    {
    GNODE *  pNode;          /* 指向节点的指针*/
    char * pColor;         /* 颜色内存块中的操作指针*/
```

```
    pColor = blkAdd;
    pNode = pdB->pFirstGnode;

    /* 指配新的颜色然后将稳定标记置为不稳定*/

    while (pNode != NULL)
    {
    pNode->att.color = *pColor;
    pNode->stable = NODE_UNSTABLE;

    pColor++;
    pNode = pNode->pNext;
    }
    return OK;
    }

/*************************************************************************/
/* graphInit 一此函数用于初始化图形着色工作。                              */
/*此函数创建用到的信号量，申请需要的内存块，初始化数据库，创建节点          */
/*以及节点之间的连接，然后孵化控制任务。在这个过程中，需要从图形文件   */
/*中获取图形数据。此函数属于 GUI API.                                    */
/* 返回值：正常情况下返回数据库的地址否则返回 NULL。                      */
/*************************************************************************/

INT32 graphInit
    (
    INT32nodeNumber,    /* 创建区域的数目*/
    char *      blkAdd,           /* 颜色块的起始地址*/
    BOOL     debugMode    /* 是否是调试模式的标志*/
    )
    {
    char *      pArea;             /* 指向操作块的指针*/
    INT32       ix;               /* 区域索引*/
    INT32stackSize;  /* 任务堆栈*/
    DATA_BASE * pdB;   /* 指向数据库的指针*/
    GNODE *  pNode;
GNODE *  pNode1,        /* 指向第一个节点的指针*/
    GNODE *  pNode2        /* 指向第二个节点的指针*/
    GNODE *  prevNode; /* 指向前一个节点的指针*/

    /* 数据库的动态内存申请*/

    if ((pdB = calloc (1, sizeof (DATA_BASE))) == NULL)
    return NULL;

    /* 信号量的创建*/

    /* 为了避免节点任务的无限制运行，使用二元信号量作同步*/

    pdB->cT.controlerFlushSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
```

```
    if (pdB->cT.controlerFlushSem == NULL)
     return NULL;

    /*使用互斥信号量保护图形数据库*/

    pdB->cT.dataBaseMutex = semMCreate (SEM_Q_FIFO);
    if (pdB->cT.dataBaseMutex == NULL)
     return NULL;

    /*使用二元信号量作同步保证节点和连接的串行创建*/

    pdB->cT.createAckSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    if (pdB->cT.createAckSem == NULL)
     return NULL;

    /*使用二元信号量作同步 graphControl 和 graphStartColoring */

    pdB->cT.restartSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    if (pdB->cT.restartSem == NULL)
     return NULL;

/*使用二元信号量作同步保证节点任务只有在连接建立的情况下才开始工作*/

    pdB->cT.connexionsDoneSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    if (pdB->cT.connexionsDoneSem == NULL)
     return NULL;

    /*使用计数型信号量保持控制任务的主循环与节点任务完成的同步*/

    pdB->cT.nodeJobDoneSem = semCCreate (SEM_Q_FIFO, 0);
    if (pdB->cT.nodeJobDoneSem == NULL)
     return NULL;

    /* 初始化数据库 */

    pdB->cT.controlAlive = 0;
    pdB->nNum = nodeNumber;
    pdB->pFirstGnode = NULL;

    /* 检查节点数是否超出 MAX_NODE 的限制 */

    if (pdB->nNum == MAX_NODE-1)
     return NULL;

    /* 申请保存节点信息的内存*/

    /* 申请第一个节点的内存*/

    if ((pdB->pFirstGnode = calloc(1, sizeof(GNODE))) == NULL)
     return ERROR;
```

```
    prevNode = pdB->pFirstGnode;

    for (ix = 1; ix < nodeNumber; ix++)
     {
        /* 循环体内申请单一节点内存 */

        if ((pNode = calloc(1, sizeof(GNODE))) == NULL)
         return ERROR;
     pNode->pNext = NULL;
     prevNode->pNext = pNode;
     prevNode = pNode;
     }

    /* 顺序创建所有节点*/

    ix = 1;
    pNode = pdB->pFirstGnode;
    stackSize = (debugMode ? NODE_STACK_DBG : NODE_STACK);
    while (pNode != NULL)
     {
    if (graphNodeCreate (pdB, pNode, ix, stackSize) == ERROR)
         return NULL;

        semTake (pdB->cT.createAckSem, WAIT_FOREVER);
    pNode = pNode->pNext;
    ix++;
    }

/*****************************************************/
/* 主机将图形数据库写到目标中以 blkAdd 为起始的内存块上。   */
/*下面根据数据库中的信息创建所有节点及节点间的连接。          */
/*****************************************************/

    pArea = blkAdd;

    for (ix=1; ix <= nodeNumber; ix++)
     {
    /* 读取当前行的第一个节点 Id */

        if ((pNode1 = graphNodeByIdGet (pdB, *pArea)) == NULL)
         return ERROR;

    pArea++;

    /* 读取所有连接的节点*/

    while ((*pArea) != DEMO_EOLM)
        {
        if ((pNode2 = graphNodeByIdGet (pdB, *pArea)) == NULL)
```

```
         return ERROR;

        if (graphFullDuplexCreate (pdB, pNode1, pNode2) == ERROR)
         return NULL;

            semTake (pdB->cT.createAckSem, WAIT_FOREVER);
        pArea++;
        }

    /* 跳过行结束标记*/

    pArea++;
    }

/*****************************************************/
/* 孵化任务 graphControl，它的优先级较低。可以在 shell 以优    */
/*先级 100 孵化此任务，图形控制任务的优先级小于节点任务。 */
/*****************************************************/

    stackSize = (debugMode ? CONT_STACK_DBG : CONT_STACK);
    pdB->cT.graphControlId =
     taskSpawn ((char *) "tDemoCtr", CONT_PRIORITY, VX_SUPERVISOR_MODE, \
              stackSize, (FUNCPTR) graphControl, (INT32) pdB , 0, 0,  \
              0, 0, 0, 0, 0, 0, 0);
    if (pdB->cT.graphControlId == ERROR )
     return NULL;

    /* 返回数据库的地址*/

    return (INT32) pdB;
    }

/*****************************************************/
/* graphStartColoring 一此函数请求所有节点任务启动它们的工作       */
/* 此函数启动所有节点处理任务，属于本应用的 GUI API。本函数     */
/* 使用的全局变量是 pdB->cT.coloringStop。              */
/* 返回值： OK 或者 ERROR.                    */
/*****************************************************/

INT32 graphStartColoring
    (
    DATA_BASE * pdB /* 指向数据库的指针*/
    )
    {
    /* 检查控制任务的活动性*/

    if (pdB->cT.controlAlive == 0)
     return ERROR;

    /* 复位全局标记*/
```

```
    pdB->cT.coloringStop = 0;

    /* 为所有节点解除阻塞*/

    if (semFlush (pdB->cT.connexionsDoneSem) == ERROR)
     return ERROR;

    /* 恢复控制任务*/

    if (semGive (pdB->cT.restartSem) == ERROR)
     return ERROR;

    return OK;
    }

/*********************************************************************/
/*graphStop 一全局退出函数。                                         */
/*此函数杀死节点任务、控制任务、消息队列和信号量，同时释放所有内存资源  */
/*返回值： OK 或者 ERROR.                                           */
/*********************************************************************/
INT32 graphStop
    (
    DATA_BASE * pdB
    )
    {
    GNODE *  pNode;    /* 指向当前节点的指针*/
    GNODE *  pPrev;     /* 指向前一节点的指针*/
    INT32jx;  /* 连接节点的索引*/

    /* 删除节点处理任务*/

    pNode = pdB->pFirstGnode;
    while (pNode != NULL)
    {
    if (taskDelete (pNode->tnid.tid) == ERROR)
        return ERROR;
    pNode = pNode->pNext;
    }

    /*删除图形使用的消息队列*/

    pNode = pdB->pFirstGnode;
    while (pNode != NULL)
    {
    for (jx = 0; jx < pNode->cNum; jx++)
        {
        if (pNode->cArray[jx].tnid.nid > pNode->tnid.nid)
         {
            if (msgQDelete (pNode->cArray[jx].rMQId) == ERROR ||
```

```
                msgQDelete (pNode->cArray[jx].wMQId) == ERROR)
            return ERROR;
        }
       }
    pNode = pNode->pNext;
    }

    /* 释放图形节点占用的空间 */

    pNode = pdB->pFirstGnode;
    while (pNode != NULL)
     {
    pPrev = pNode;
    pNode = pNode->pNext;
    free (pPrev);
    }

    /* 删除控制任务*/

    if (pdB->cT.controlAlive == 1)
        if (taskDelete (pdB->cT.graphControlId) == ERROR)
         return ERROR;

    /* 删除所有信号量 */

    if (semDelete (pdB->cT.dataBaseMutex)     == ERROR ||
     semDelete (pdB->cT.controlerFlushSem) == ERROR ||
     semDelete (pdB->cT.connexionsDoneSem) == ERROR ||
     semDelete (pdB->cT.createAckSem)      == ERROR ||
     semDelete (pdB->cT.restartSem)        == ERROR ||
     semDelete (pdB->cT.nodeJobDoneSem)    == ERROR)
        return ERROR;

    /* 释放数据库空间*/

    free (pdB);

    return OK;
    }

/********************** vccolor.c 程序的结尾**********************/
```

7.7　C++　应　用

本文前面已经介绍了 VxWorks 系统对 C++的支持。这里介绍一个实际的 C++例子。

本 C++实例中实现了“object factory”。首先给各个类指定一个可读性较强的名字，并将它们登记到一个“全局类注册表”中。然后依据各个对象名创建对象，同时登记到对象注

册表中。本实例中介绍的类和对象的等级关系如图 7-6 所示。

在本实例中，通过创建对象，可以接触到下列不同的 C++功能：

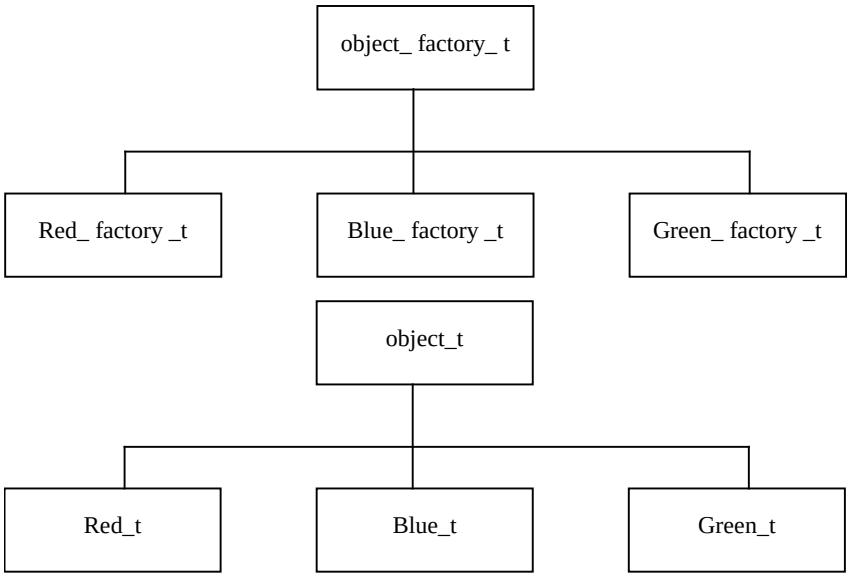

图 7-6　C++实例中类和对象的等级关系

- 标准模板库（Standard Template Library）　使用一个基本的模板来创建不同的注册表。
- 用户定义模板（User defined templates）　我们的用到的类注册表和对象注册表都是基于一个基本的注册表类型。
- 运行时类型检查（Run Time Type Checking）　我们在实例中提供一个函数来判定注册对象的类型。
- 异常处理（Exception Handling）　在类型判定过程中，当碰到“wrong”类型时，必须进行 C++异常处理。

在本 C++实例中，我们提供了一个简单的 C 接口 testFactory()，这是一个对象测试函数。在 Wind shell 下可以通过这个接口实现简单的测试。下面列出执行测试的全过程。

```
-> classRegistryShow       # 显示类注册表内容
Showing Class Registry ...
Name     Address
============================================================
blue_t     0x6b1c7a0
green_t 0x6b1c790
red_t      0x6b1c7b0

-> objectRegistryShow       # 显示对象注册表内容
Showing Object Registry ...
Name     Address
============================================================
```

```
-> objectCreate "green_t", "bob"      # 创建一个名为 bob 的 green_t 对象
Creating an object called 'bob' of type 'green_t'

-> objectCreate "red_t", "bill"         # 创建一个名为 biu 的 red_t 对象
Creating an object called 'bill' of type 'red_t'

-> objectRegistryShow                 # 再次显示对象注册表内容
Showing Object Registry ...
Name      Address
============================================================
bill      0x6b1abf8
bob      0x6b1ac18

-> objectTypeShowByName "bob"     # 显示名字"bob"的对象类型
Looking up object 'bob'
Attempting to ascertain type of object at 0x6b1ac18      # 查询是否为 red_t 类型
Attempting a dynamic_cast to red_t ...
dynamic_cast threw an exception ... caught here!         #出现异常
Attempting a dynamic_cast to blue_t ...                  #查询是否为 blue_t 类型
dynamic_cast threw an exception ... caught here!          #出现异常
Attempting a dynamic_cast to green_t ...                 #查询是否为 green_t 类型
Cast to green_t succeeded!                              #成功查询
green.
```

测试的简单流程如图 7-7 所示。

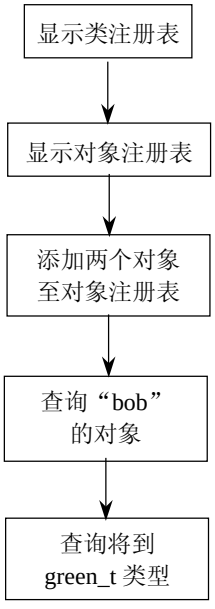

图 7-7　测试的简单流程

本 C++应用实例只编写了两个程序文件即 factory.h 和 factory.c。下面列出这两个程

序，并做扼要注释说明。

```
/******************factory.h 程序的开始***************/
/*说明： factory.h 头文件是 factory.cpp 程序文件的类声明  */

/*包含头文件*/

#include <vxWorks.h>
#include <iostream.h>
#include <string>
#include <typeinfo>
#include <map>

/*对象声明*/
struct object_t
    {
    virtual void method () {}
    };

struct red_t : object_t
    {
    };

struct blue_t : object_t
    {
    };

struct green_t : object_t
    {
    };

struct object_factory_t
    {
    virtual object_t* create () = 0;
    };

struct red_factory_t : object_factory_t
    {
    red_t* create () { return new red_t; }
    };

struct blue_factory_t : object_factory_t
    {
    blue_t* create () { return new blue_t; }
    };

struct green_factory_t : object_factory_t
    {
    green_t* create () { return new green_t; }
    };
```

```
    /*********************************************/
    /* registry_t<T> —注册 T 类型的对象。            */
    /* 这里将用户可读的名字映射为对象指针。         */
    /********************************************* */

    template <class T> class registry_t
        {
    private:
        typedef map <string, T*> map_t;
        map_t registry;
    public:
        void insert (string objectName, T* pObject);  /* 添加一个类型为 pObjcct 的 objectName
对象*/
        T* lookup (string objectName);            /* 查询对象的类型*/
        void list ();
        };

    /* object_registry_t 表示继承 object_t 的对象的注册*/

    typedef registry_t <object_t> object_registry_t;

    /* class_registry_t 表示类的注册*/

    class class_registry_t : public registry_t <object_factory_t>
        {
    public:
        object_t* create (string className) ;
        };

    /*********************************************/
    /* 下面主要是模板的方法定义。将模板方法定义放到头文*/
    /* 件中可为应用带来方便，便于在使用时随时实例化。 */
    /*********************************************/

    /*******************************************************/
    /* registry_t<T>::insert 一此函数用于注册一个对象。              */
    /* 此函数注册由 pObject 和 'objectName'.确定的对象。              */
    /* 返回值：N/A                                  */
    /*******************************************************/

    template <class T>
    void registry_t<T>::insert
        (
        string objectName,
        T* pObject
        )
        {
        registry [objectName] = pObject;
        }
```

```
/*******************************************************/
/* registry_t<T>::lookup －此函数用于通过名字查找一个对象。     */
/* 此函数在注册表中查找'objectName'，然后返回一个对象指针。     */
/* 返回值：正常情况下返回对象指针，否则返回 NULL。          */
/*******************************************************/

template <class T>
T* registry_t<T>::lookup
    (
    string objectName
    )
    {
    return registry [objectName];
    }


/*************************************************/
/* registry_t<T>::list－此函数列出注册表中的对象。     */
/* 返回值： N/A                          */
/*************************************************/

template <class T>
void registry_t<T>::list ()
    {
    cout << "Name \t" << "Address" << endl;
    cout << "================================================" << endl;
    for (map_t::iterator i = registry.begin ();
         i != registry.end (); ++i)
        {
        cout << i -> first << " \t"
             << "0x" << hex << (int) i -> second << endl;
        }
    }

/* 函数声明*/

/* objectCreate 函数用于创建一个给定类型的对象*/
object_t* objectCreate (char* className, char* objectName);

/* objectTypeShowByName 函数用于确定一个注册对象的类型*/
void objectTypeShowByName (char* objectName);

/* objectRegistryShow 函数用于显示全局对象注册库的信息*/
void objectRegistryShow ();

/* classRegistryShow 函数用于显示全局类注册库的信息*/
void classRegistryShow ();

/****************** factory.h 程序的结尾*********************/
```

```
/*******************factory.cpp 程序的开始*****************/
/* 说明：factory.cpp 程序用于实现一个对象库。*/

/* 包含头文件*/
#include "factory.h"

/* 本地变量 */

/* 指向全局类注册库的指针*/
LOCAL class_registry_t* pClassRegistry;

/* 指向全局对象注册库的指针*/
LOCAL object_registry_t* pObjectRegistry;

/*********************************************************/
/* testFactory －此函数用于测试注册库的运行。                    */
/*********************************************************/

void testFactory ()
{
    cout << "classRegistryShow ()" << endl;
    classRegistryShow ();
    cout << "objectRegistryShow ()" << endl;
    objectRegistryShow ();
    cout << "objectCreate (\"green_t\", \"bob\")" << endl;
    objectCreate ("green_t", "bob");
    cout << "objectCreate (\"red_t\", \"bill\")" << endl;
    objectCreate ("red_t", "bill");
    cout << "objectRegistryShow ()" << endl;
    objectRegistryShow ();
    cout << "objectTypeShowByName (\"bob\")" << endl;
    objectTypeShowByName ("bob");
}

/*********************************************************/
/* class_registry_t::create －此函数以 className 创建一个对象。     */
/* 此函数首先以 className 查找注册表。如果存在则利用注册      */
/* 类库中的类型创建一个对象，如果不存在则返回 NULL 值。     */
/*返回值： 指向新创建的对象的指针，失败时返回 NULL。       */
/*********************************************************/

object_t* class_registry_t::create
    (
    string className
    )
    {
    object_factory_t* pFactory = lookup (className);
    if (pFactory != NULL)
        {
```

```
            return lookup (className)-> create ();
            }
        else
            {
            cout << "No such class in Class Registry. " << endl;
            return NULL;
            }
        }

    /*******************************************************/
    /* classRegistryGet －此函数用于从全局类注册表获得引用值。    */
    /* 此函数创建并返回一个全局类注册表的引用值。          */
    /* 返回值： 全局类注册表的引用。                  */
    /*******************************************************/

    LOCAL class_registry_t& classRegistryGet ()
        {
        if (pClassRegistry == NULL)
            {
            pClassRegistry = new class_registry_t;
            pClassRegistry -> insert ("red_t", new red_factory_t); /*将“red_t”类添加到
red_factory_t*/
            pClassRegistry -> insert ("blue_t", new blue_factory_t);  /*将“bkte_t”类添加到
blue_factory_t*/
            pClassRegistry -> insert ("green_t", new green_factory_t);  /*将“green_t”类添加
到 green_factory_t*/
            }
        return *pClassRegistry;
        }

    /*****************************************************/
    /* objectRegistryGet－此函数用于从全局对象注册表获得引用值。*/
    /* 此函数创建并返回一个全局对象注册表的引用值。          */
    /* 返回值： 全局对象注册表的引用。                  */
    /*****************************************************/

    LOCAL object_registry_t& objectRegistryGet ()
        {
        if (pObjectRegistry == NULL)
            {
            pObjectRegistry = new object_registry_t;
            }
        return *pObjectRegistry;
        }

    /*****************************************************/
    /* objectCreate－创建一个给定类型的对象。            */
    /* 此函数使用全局类注册表中以 className 注册的类创建一个新    */
    /* 的对象，同时以 objectName 将此对象注册到全局类注册表。  */
    /* 返回值：对象类型。                    */
```

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

object_t* objectCreate
    (
    char* className,
    char* objectName
    )
    {
    cout << "Creating an object called '" << objectName << "'"
         << " of type '" << className << "'" << endl;
    object_t* pObject = classRegistryGet().create (className);
    if (pObject != NULL)
        { /*和 pObject 的添加函数插入一个 objectNane 对象*/
        objectRegistryGet().insert(objectName, pObject);
        }
    else
        {
        cout << "Could not create object. Sorry. " << endl;
        }
    return pObject;
    }

/*****************************************************************/
/* isRed , isBlue, isGreen 一这些函数均表示对对象的引用。        */
/* 此函数尝试进行 dynamic_cast，如果成功则返回 TRUE，否则捕捉异常结果同时 */
/* 返回 FALSE。                                    */
/* 返回值： TRUE 或者 FALSE。                        */
/*****************************************************************/

/* isRed 函数 */

LOCAL BOOL isRed (object_t& anObject)
    {
    try
        {
    cout << "Attempting a dynamic_cast to red_t ..." << endl;
        dynamic_cast<red_t&> (anObject);
        cout << "Cast to red_t succeeded!" << endl;
        return TRUE;
        }
    catch (exception)
        {
        cout << "dynamic_cast threw an exception ... caught here!" << endl;
        return FALSE;
        }
    }

/* isBlue 函数*/

LOCAL BOOL isBlue (object_t& anObject)
```

```
    {
    try
        {
    cout << "Attempting a dynamic_cast to blue_t ..." << endl;
        dynamic_cast<blue_t&> (anObject);
        cout << "Cast to blue_t succeeded!" << endl;
        return TRUE;
        }
    catch (exception)
        {
        cout << "dynamic_cast threw an exception ... caught here!" << endl;
        return FALSE;
        }
    }

/* isGreen 函数*/

LOCAL BOOL isGreen (object_t& anObject)
    {
    try
        {
    cout << "Attempting a dynamic_cast to green_t ..." << endl;
        dynamic_cast<green_t&> (anObject);
        cout << "Cast to green_t succeeded!" << endl;
        return TRUE;
        }
    catch (exception)
        {
        cout << "dynamic_cast threw an exception ... caught here!" << endl;
        return FALSE;
        }
    }

/*******************************************************/
/* objectTypeShow 一此函数用于确定一个对象的类型。          */
/* 此函数利用动态类型检查来判断一个对象的类型。        */
/* 返回值： N/A                          */
/*******************************************************/

LOCAL void objectTypeShow (object_t* pObject)
    {
    cout << "Attempting to ascertain type of object at " << "0x" << hex
         << (int) pObject << endl;
    if (isRed (*pObject))
        {
        cout << "red." << endl;
        }
    else if (isBlue (*pObject))
        {
        cout << "blue." << endl;
```

```
        }
    else if (isGreen (*pObject))
        {
        cout << "green." << endl;
        }
    }

/*******************************************************/
/* objectTypeShowByName －此函数用于确定一个注册对象的类型。*/
/*此函数首先以'objectName'查询全局对象注册表，然后将相应对象    */
/*打印出来。                              */
/* 返回值： N/A                              */
/*******************************************************/

void objectTypeShowByName
    (
    char* objectName
    )
    {
    cout << "Looking up object '" << objectName << "'" << endl;
    object_t *pObject = objectRegistryGet ().lookup (objectName);
    if (pObject != NULL)
        {
        objectTypeShow (pObject);
        }
    else
        {
        cout << "No such object in the Object Registry." << endl;
        }

    }

/*****************************************************/
/* objectRegistryShow－此函数用于显示全局对象注册表中的内容。*/
/* 返回值： N/A                                      */
/*****************************************************/

void objectRegistryShow ()
    {
    cout << "Showing Object Registry ..." << endl;
    objectRegistryGet ().list ();
    }

/*****************************************************/
/* classRegistryShow －此函数用于显示全局类注册表中的内容。    */
/* 返回值： N/A                          */
/*****************************************************/

void classRegistryShow ()
    {
```

```
    cout << "Showing Class Registry ..." << endl;
    classRegistryGet ().list ();
}

/******************factory.cpp 程序的结尾******************/
```

7.8　数据报应用

本节我们介绍一个简单的 UDP 数据报应用，演示如何利用 UDP 数据报在处理器之间交互数据信息。此 UDP 数据报应用包含两个源程序，一个是 dgMain.c，运行 在 Unix 主机（如 Linux）上，另一个是 dgTest.c，运行在 VxWorks 系统上。

本实例展示了如何在 VxWorks 系统和 Unix 系统（如 Linux）之间发送和接收 UDP 数据报。我们首先让接收程序在 VxWorks 系统或者 UNIX 系统上初始化，然后接收程序在分配的端口上等待分组数据的到来。接下来，我们让发送数据的程序在 UNIX 系统或者 VxWorks 系统上开始向对端发送数据。当接收程序接收到了一定量的数据分组，它将停止接收。

我们设计的发送和接收程序支持以下可选项：

-v　用于指示打印详细的调试信息。

-b　广播标志，用于指示发送程序 dgSender 广播数据分组。

-h　主机名标志，指示发送数据的网络位置。

-p　端口标志，用于选择通信的 UDP 端口号，默认是 1101。

-n　用于指示发送数据分组的数目，默认为 5 个

-help 用于指示打印上述信息。

例如，如果我们需要看到调试信息并且指示主机名，可以使用如下命令分步启动数据接收和发送程序：

```
> dgReciver -v
> dgSender -h localhost -v
... output from dgReceiver
```

为了创建 VxWorks 目标文件，我们可以在开发主机上使用如下命令：

```
> make CPU=<CPU_TYPE>
```

例如，利用命令 make CPU=SPARC 就可以将 VxWorks 目标文件 "dgTest.o" 安装到目录 $WIND_BASE/target/lib/ objSPARCgnutest/中。

为了在 Unix 主机上创建 UDP 数据报的发送和接收程序（即 dgSender 和 dgReceiver），可以利用如下命令：

```
> cc -o dgSender dgMain.c dgTest.c
> cc -o dgReceiver dgMain.c dgTest.c
```

本应用实例只编写了两个程序文件即 dgMain.c 和 dgTest.c。下面列出这两个程序，并做扼要注释说明。

```
/****************** dgmain.c 程序的开始**********************/
/* 说明：dgMain.c 程序是运行在 UNIX 上测试数据报收发的主程序。*/

/*包含头文件*/
```

```
#if 0
#include "vxWorks.h"
#include "sys/types.h"
#include "netdb.h"
#include "inetLib.h"
#include "in.h"
#else
#include <sys/types.h>
#include <netdb.h>
#include <netinet/in.h>

#define IMPORT extern
#define LOCAL static
#define BOOL int
#define ERROR -1
#define NULL 0
#endif

#if    !defined(HOST_HP)
#include <sys/socket.h>    /*对于 HP/UX 7.0 , 不包含此头文件*/
#endif

#if    !defined(HOST_MIPS)
#include <sys/ioctl.h>
#else
#include <bsd43/sys/ioctl.h>
#endif

#define    TORF(x)    ((x) ? "True" : "False")

#define PACKET_NUM    500        /* 最大分组数 */

IMPORT BOOL broadcast;    /* 当为 TRUE 时, dgSender 将开始广播*/
IMPORT BOOL verbose;     /* 当为 TRUE 时, 将打印调试信息*/
IMPORT int dgPort;
IMPORT int dgPackets;

IMPORT BOOL is42;
#ifdef POLL
IMPORT int fionread;        /*Unix 的 FIONREAD 值不同于 VxWorks */
#endif

IMPORT int errno;

char sysBootHost [40];        /* VxWorks 引导的主机名和 Unix 主机名*/

/*程序的宏定义*/

#define    SENDNAME    "dgSender"
#define    RECVNAME    "dgReceiver"
```

```
LOCAL char *toolname;

LOCAL char *dgSendUsage =
    "usage: %s [-help] [-b] [-h host] [-p port] [-n packets] [-v]\n";
LOCAL char *dgRecvUsage =
    "usage: %s [-help] [-p port] [-n packets] [-v]\n";

LOCAL char *dgSendHelp =
"show defaults\n\
-b    broadcast         = %s\n\
-h    host              = %s\n\
-p    port number       = %d\n\
-n    number of packets = %d\n\
-v    verbose           = %s\n";

LOCAL char *dgRecvHelp =
"show defaults\n\
-p    port number       = %d\n\
-n    number of packets = %d\n\
-v    verbose           = %s\n";

#if     defined(HOST_HP)
void bzero (s, n)
    char *s;
    int n;

    {
    memset (s, '\0', n);
    }
#endif

/*******************************************************/
/* main  -  此函数用于发送和接收 UDP 数据报，它运行在主机上。      */
/* -help        显示默认值                  */
/* -b         广播或者只发送                      */
/* -h         主机侧广播或者只发送                 */
/* -n            端口号（默认值为 1101）    */
/* -p            分组数（默认值为 5）       */
/* -v        是否显示其它信息                    */
/*******************************************************/

void main (argc, argv)
    int argc;
    char *argv [];

    {
    char **argp = argv;
    BOOL send;
```

```
toolname  = *argp;
send = strcmp (SENDNAME, toolname) == 0;

gethostname (sysBootHost, sizeof (sysBootHost));    /*获取主机名*/

while (--argc > 0)
{
argp++;
switch (argp [0][0])
    {
    case '-':
    if (strcmp (*argp, "-help") == 0)
        {
        if (send)
        printf (dgSendHelp, TORF(broadcast), sysBootHost,
            dgPort, dgPackets, TORF(verbose));
        else
        printf (dgRecvHelp,
            dgPort, dgPackets, TORF(verbose));
        exit (0);
        }
    switch (argp [0][1])
        {
        case 'b':            /*广播标志*/
        if (!send)
            goto l_usage;
        broadcast = !broadcast;
        break;
        case 'h':            /*主机名标志*/
        if (!send)
            goto l_usage;
        if (argc-- > 0)
            strcpy (sysBootHost, *++argp);  /*填写引导主机名参数*/
        else
            {
            printf ("%s: missing <%s>\n", toolname, "host");
            goto l_usage;
            }
        break;
        case 'n':     /*指示发送分组的个数*/
        if (argc-- > 0)
            {
            dgPackets = atoi (*++argp);    /*将接收字符转化为整型*/
            if (dgPackets > PACKET_NUM)
            {
            printf ("%s: too many packets <%d>\n",
                toolname, dgPackets);
            goto l_usage;
            }
            }
```

```
            else
               {
               printf ("%s: missing <%s>\n", toolname, "packet");
               goto l_usage;
               }
            break;
           case 'p':                /*端口标志*/
            if (argc-- > 0)
               dgPort = atoi (*++argp);
            else
               {
               printf ("%s: missing <%s>\n", toolname, "port");
               goto l_usage;
               }
            break;
           case 'v':                /*打印调试信息*/
           verbose = !verbose;
           break;
           default:
           printf ("%s: bad flag <%s>\n", toolname, *argp);
           goto l_usage;
           }
        break;

        default:
         printf ("%s: no such option <%s>\n", toolname, *argp);      /*其他调选项无效*/
         goto l_usage;
        }
    }

    if (argc > 0)
     {
     printf ("%s: no such option <%s> ...\n", toolname, *argp);
     goto l_usage;
    }

#ifdef POLL
    fionread = FIONREAD;
#endif

#ifndef    SO_BROADCAST
    /* BSD 4.2 不要求打开广播*/
    is42 = TRUE;
#endif    /* SO_BROADCAST */

    if (send)
     dgSender (sysBootHost, dgPort, dgPackets);
    else
     dgReceiver (dgPort, dgPackets);
```

```
    exit (0);

l_usage:
    printf ((send ? dgSendUsage : dgRecvUsage), toolname);
    exit (0);
}

/*****************************************************/
/*     hostGetByName - 此函数为 VxWorks 兼容函数,          */
/*    用于返回长整型 inet 地址。                  */
/*****************************************************/

long hostGetByName (hostname)
    char *hostname;

    {
    struct hostent *destHost;

    /*获得给定主机的 IP 地址*/

    if ((destHost = (struct hostent *) gethostbyname (hostname)) == NULL)
     {
    printf ("%s:non existant host <%s>\n", toolname, hostname);
    return ((u_long)ERROR);
    }

    return (*(u_long *)destHost->h_addr);
    }
/*****************************************************/
/* errnoGet -此函数为 VxWorks 兼容函数,用于返回错误号。     */
/*****************************************************/

int errnoGet ()

    {
    return (errno);
    }
/******************** dgmain.c 程序的结尾*****************************/

/******************** dgtest.c 程序的开始  ************************/
/* 说明:dgTest.c 程序测试数据报的发送和接收。*/

/* 包含头文件 */

#ifdef CPU
#include "vxWorks.h"
#include "sys/types.h"
#include "ioLib.h"
#include "fioLib.h"
#include "stdio.h"
```

```
#include "unistd.h"
#include "string.h"
#include "usrLib.h"
#include "errnoLib.h"
#include "hostLib.h"
#include "sockLib.h"
#include "socket.h"
#include "inetLib.h"
#include "in.h"
#else
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BOOL int
#define LOCAL static
#define IMPORT extern
#define ERROR -1
#define NULL 0
#define TRUE 1
#define FALSE 0
#endif

#define    TORF(x)    ((x) ? "True" : "False")

#define PACKET_SIZE    100   /* UDP 分组的大小*/
#define PACKET_NUM    500      /* UDP 分组的最大数*/

IMPORT char sysBootHost[];        /* VxWorks 引导主机名*/

typedef struct    /* 分组结构定义 */
    {
    int sequence;
    int spare;
    char data [PACKET_SIZE - 2 * sizeof (int)];
    } PACKET;

BOOL broadcast       = FALSE;    /* 当为 TRUE 时，dgSender 将开始广播*/
BOOL verbose       =FALSE;      /* 当为 TRUE 时，将打印调试信息*/
BOOL terminate       = FALSE;    /*当为 TRUE 时，dgReceiver 将退出*/
BOOL is42            = FALSE;   /*当为 FALSE 时，socket 用于广播*/
int dgPort           = 1101;
int dgPackets        = 5;
#ifdef POLL
int fionread         = FIONREAD;
#endif


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

```
/* dgSender  - 此函数用于发送 UDP 数据报                */
/*******************************************************/

void dgSender (host, port, packets)
    char *host;
    int port;
    int packets;

    {
    struct in_addr destNet;
    u_long inetaddr;
    struct sockaddr_in sockaddr;
    int optval = 1;
    PACKET packet;
    int ix;
    int s;

    if (host == NULL)
    host = sysBootHost;

    if ((inetaddr = hostGetByName (host)) == ERROR &&   /* 验证主机名的有效性*/
     (inetaddr = inet_addr (host)) == ERROR)
    {
    printf ("invalid host: <%s>\n", host);
    return;
    }
    /*判断是否需要广播*/
    if (broadcast)
    {
#ifdef CPU
    struct in_addr tmp;
    /*构造广播地址*/
    tmp.s_addr = inetaddr;
    destNet  = inet_makeaddr (inet_netof (tmp), INADDR_BROADCAST);
    inetaddr = destNet.s_addr;
#else
#if (WIND_HOST_TYPE==iris)
    struct in_addr tmp;
    /*构造广播地址*/
    tmp.s_addr = inetaddr;
    destNet  = inet_makeaddr (inet_netof (tmp), INADDR_BROADCAST);
    inetaddr = destNet.s_addr;
#else
    destNet = inet_makeaddr (inet_netof (inetaddr), INADDR_BROADCAST);
#endif
#endif

    if (verbose)
        printf ("broadcast ");
    }
```

```
    if (verbose)
     printf ("inet address = %#x\n", htonl(inetaddr));

    s = socket (AF_INET, SOCK_DGRAM, 0);    /* 获得一个 udp 套接字*/

    if (s < 0)
    {
    printf ("socket error (errno = %#x)\n", errnoGet ());
    return;
    }

    if (!is42)
    {
    /* BSD 4.2 不要求打开广播*/

    if (setsockopt (s, SOL_SOCKET, SO_BROADCAST,
            (caddr_t) &optval, sizeof (optval)) < 0)
        {
        printf ("setsockopt error (errno = %#x)\n", errnoGet ());
        return;
        }
    }

    if (port == 0)
     port = dgPort;

    if (packets == 0)
     packets = dgPackets;
/* 首先为分组内存区域清零*/
    bzero ((char *) &packet, sizeof (PACKET));
    bzero ((char *) &sockaddr, sizeof (sockaddr));

    sockaddr.sin_family      = AF_INET;
    sockaddr.sin_addr.s_addr = inetaddr;
    sockaddr.sin_port        = htons (port);

    /* 发送数据报*/

    for (ix = 0; ix < packets; ix++)
     {
    packet.sequence = htonl (ix);

    /* 发送 */

    if (sendto (s, (caddr_t) &packet, sizeof (packet), 0,
            (struct sockaddr *)&sockaddr, sizeof (sockaddr)) == -1)
        {
        printf ("sendto error on packet # %d (errno = %#x)\n",
             ix, errnoGet ());
```

```
            close (s);
            return;
            }
        }

        printf ("sent %d packets\n", ix);

        close (s);
        }
    /*******************************************************/
    /* dgReceiver -此函数用于接收数据报分组          */
    /*******************************************************/

    void dgReceiver (port, packets)
        int port;
        int packets;

        {
        struct sockaddr_in sockaddr;
        int sockaddrLen = sizeof (sockaddr);
        int nBytes;
        int ix;
        BOOL missing;
        char packetList [PACKET_NUM];    /* 接收分组的布尔型数组*/
        PACKET packet;
        int nPacketsReceived = 0;
        int s = socket (AF_INET, SOCK_DGRAM, 0);     /*打开一个 socket * /

        if (s < 0)
        {
        printf ("socket error (errno = %#x)\n", errnoGet ());
        return;
        }

    #ifdef    REUSE
        if (setsockopt (s, SOL_SOCKET, SO_REUSEADDR,
                  (caddr_t) &optval, sizeof (optval)) < 0)     /*设置 socket 选项 * /
        {
        printf ("setsockopt error (errno = %#x)\n", errnoGet ());
        return;
        }
    #endif

        if (port == 0)
         port = dgPort;

        if (packets == 0)
         packets = dgPackets;

        /* 清除分组列表*/
```

```
    /* 为分组的内存区域清空 */
    bzero ((char *) packetList, sizeof (packetList));
    bzero ((char *) &sockaddr, sizeof (sockaddr));

    sockaddr.sin_family      = AF_INET;
    sockaddr.sin_port        = htons (port);
    /* 绑定 socket */
    if (bind (s, (struct sockaddr *) &sockaddr, sizeof (sockaddr)) == ERROR)
     {
     printf ("bind error (errno = %#x)\n", errnoGet ());
     return;
     }
    /* 获取 socket 的各字 */
    if (getsockname (s, (struct sockaddr *) &sockaddr, &sockaddrLen) == ERROR)
     {
     printf ("getsockname error (errno = %#x)\n", errnoGet ());
     return;
     }

    terminate = FALSE;

    while (packets > nPacketsReceived)
     {
#ifdef    POLL
     if (ioctl (s, fionread, &nBytes) == ERROR)
         {
         printf ("ioctl error (errno = %#x)\n", errnoGet ());
         close (s);
         return;
         }

     if (nBytes > 0)
#endif    /* POLL */
         {/*从 socket 接收字符*/
         nBytes = recvfrom (s, (caddr_t) &packet, sizeof (PACKET), 0,
                     (struct sockaddr *) &sockaddr, &sockaddrLen);

         if (verbose)
          printf ("received packet # %d\n", ntohl(packet.sequence));

         packetList [ntohl(packet.sequence)] = TRUE;
         nPacketsReceived++;
         }
     if (terminate)
        {
        printf ("terminated\n");
        break;
         }
    }
```

```
        missing = FALSE;
        nPacketsReceived = 0;
        for (ix = 0; ix < packets; ix++)
         {
         if (packetList[ix] == TRUE)
             nPacketsReceived++;
         else
             {
             missing = TRUE;
             printf ("%d\n", ix);
             }
         }

        printf ("\n%smissing packets\n", missing ? "" : "no ");
        printf ("%d packets received\n", nPacketsReceived);

        close (s);
    }

    /*************************************/
    /* dgHelp  - 此函数用于显示帮助          */
    /*************************************/

    void dgHelp ()

        {
        printf ("dgHelp                              show this list\n");   /* 打印信息 */
        printf ("dgReceiver [port],[packets]        receive datagrams\n");    /* 打印接受由文
据报 */
        printf ("dgSender [host],[port],[packets]   send datagrams\n");    /* 打印发送由文据
报 */
        printf ("\n");
        printf ("broadcast               = %s\n", TORF(broadcast));
        printf ("host                    = %s\n", sysBootHost);
        printf ("port number             = %d\n", dgPort);
        printf ("number of packets       = %d\n", dgPackets);
        printf ("verbose                 = %s\n", TORF(verbose));
        printf ("\n");
        }
    /******************** dgtest.c 程序的结尾*****************************/
```

7.9　虚拟内存设备驱动

本节我们介绍一个虚拟内存设备驱动程序的编写。VxWorks 系统的 I/O 系统可以通过这个驱动程序将硬件系统的内存当作虚拟 I/O 设备直接访问。当我们在创建虚拟 I/O 设备时，指定了内存的位置和大小。当我们需要在 VxWorks 引导期间保存关键数据或者需要在 CPU 之间共享数据时，使用虚拟 I/O 设备是非常方便的。

此外，这个虚拟内存设备驱动程序可以将一些文件编译为 VxWorks 二进制映像，然后将它们挂接到一个文件系统上。这是通过 VxWorks 系统传递一些不宜改变的文件的简便方法。当然在将文件编译为 VxWorks 二进制文件时，首先需要将这些文件转换为 C 语言源程序的数组，这可以使用 memdrvbuild 功能函数实现。例如，具有集成 web server 的系统就可以使用虚拟内存技术将某些 HTML 以及与其相关的链接文件编译为 VxWorks 二进制映像。

本节介绍的虚拟内存驱动程序利用 I/O 系统调用为硬件系统中绝对内存位置的字节流读写提供了一种简单而又高效的方法。也可以利用本驱动程序实现一个简单的只读文件系统。

实现只读文件时，可以进行简单的目录搜索和 I/O 属性操作。

此驱动程序中的大多数函数只能通过 VxWorks 系统的 I/O 接口系统访问。但是有 4 个函数是可以直接调用的。这 4 个函数中，函数 memDrv()用于初始化此驱动程序，函数 memDevCreate() 和 memDevCreateDir()用于创建设备，而函数 memDevDelete()则用来删除设备。

下面详细介绍 memDevCreate()函数。如果要创建一个设备“/mem/cpu0/”访问本地处理器的整个内存，我们可以使用如下调用方法：

```
memDevCreate ("/mem/cpu0/", 0, sysMemTop())
```

设备将以指定的名字、起始内存位置和大小创建。上述调用以 sysMemTop()为大小创建设备，所以可以访问到本地处理的所辖内存。针对内存打开文件描述符时可以使用系统调用 open()。打开文件后，我们可以用文件的字节偏移声明一个伪文件名，或者在开始就打开原始文件，然后声明一个查找的位置。例如，如下对 open()的调用将可以读取以十进制偏移量 1000 为起始的内存：

```
fd = open ("/mem/cpu0/1000", O_RDONLY, 0)
```

再举一个例子，假设系统中配置了两个 CPU，而且 CPU 之间通过 1M 字节的双端口内存实现互相访问。第一个 CPU 映射到 VMEbus 的地址时 0x00400000 (4M)，第二个 CPU 的地址是 0x00800000 (8M)，双端口内存地址是 0x00c00000 (12 M)。在每个 CPU 上，我们都可以创建 3 个设备。在第一个 CPU，可做如下创建：

```
memDevCreate ("/mem/local/", 0, sysMemTop())
memDevCreate ("/mem/cpu1/", 0x00800000, 0x00100000)
memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

而在第二个 CPU，可做如下创建：

```
memDevCreate ("/mem/local/", 0, sysMemTop())
memDevCreate ("/mem/cpu0/", 0x00400000, 0x00100000)
memDevCreate ("/mem/share/", 0x00c00000, 0x00100000)
```

如果第一个 CPU 上有一个本地磁盘，那么它上面的数据或者目标模块可以从第一个 CPU 传递到第二个 CPU。为了完成这个工作首先在第一个 CPU 上做如下操作：

```
copy </disk1/module.o >/mem/share/0
```

然后在第二个 CPU 运行如下装载命令：

```
ld </mem/share/0
```

接着，我们详细介绍一下 memDevCreateDir()函数。此函数可以为多个文件创建内存设备。而且此函数创建的内存设备用来组织一个目录下面的多个文件。目录数组的入口记录了文件各种信息。其实，目录下面也允许有别的目录，不过这时是把它当作一个文件对待的。这样目录与目录之间可能形成多层嵌套。memDevCreateDir()函数提供的灵活的目录嵌套机制允许用户可以在 VxWorks 系统中直接创建和安装文件系统，当然这种文件系统只能提供只读形式的访问。文件系统的内部结构可以利用主机上专门的内存驱动工具软件创建。memDevCreateDir()函数在得到目录数组入口项进行内存设备创建的时候,并不将目录数组入口项拷贝到函数内的局部变量中，因为 memDevCreateDir()函数引用的是目录数组入口项的指针。因此我们不能随意修改目录数组入口项的内容。

下面介绍函数 memDevDelete()。这个函数用于删除一个包含单个文件或者文件集合的内存设备。删除内存设备时只需要传入设备名字。例如，要删除利用函数调用 memDevCreate ("/mem/cpu0/", 0,sysMemTop())创建的内存设备，可以利用如下调用实现：

```
memDevDelete ("/mem/cpu0/");
```

在使用此驱动程序时，必须首先调用函数 memDrv( )完成初始化工作，而且这个函数仅需在执行读写操作或者调用函数 memDevCreate()之前调用一次。例如，我们可以在 VxWorks 系统中提供的 usrRoot( )函数中调用，当然在系统启动流程的稍后一点调用也是可行的。

VxWorks 系统支持的 dosFs 文件系统支持在 ioLib.h 中定义的 ioctl() 功能，在使用这些 I/O 属性功能时必须提供文件描述符。

本驱动程序编写的代码文件为 memDrv.c，下面列出此文件并对其做详细说明。

```
/******************* memDrv.c 程序的开始*****************************/
/*说明： memDrv.c 程序用来实现虚拟内存设备*/

/*包含头文件*/

#include "vxWorks.h"
#include "ioLib.h"
#include "iosLib.h"
#include "stat.h"
#include "dirent.h"
#include "memLib.h"
#include "errnoLib.h"
#include "string.h"
#include "stdlib.h"
#include "stdio.h"
#include "memDrv.h"
```

```
typedef struct mem_drv_direntry
    {
    char * name;                /* 入口名字，相当一个挂接点*/
    char * base;           /* 起始地址*/
    struct mem_drv_direntry *pDir;   /*包含文件的目录 */
    int length;                 /* 文件的字节长度或者目录下的文件数*/
    } MEM_DRV_DIRENTRY;


typedef struct                  /* MEM_DEV 结构是内存设备描述符 */
    {
    DEV_HDR devHdr;
    MEM_DRV_DIRENTRY dir;     /* memDrv 设备的内容 */
    int allowOffset;            /* 允许文件以偏移量打开的标志 */
    } MEM_DEV;


typedef struct                  /* MEM_FILE_DESC 结构是内存文件描述符*/
    {
    MEM_DEV *pDevice;           /* 文件的内存设备 */
    MEM_DRV_DIRENTRY *pDir;  /* 文件的目录入口 */
    int offset;                 /* 当前位置 */
    int mode;                  /* 模式包括 O_RDONLY、 O_WRONLY 和 O_RDWR */
    } MEM_FILE_DESC;


LOCAL int memDrvNum;      /* 内存驱动的数目*/


/* 函数声明 */

LOCAL MEM_DRV_DIRENTRY *memFindFile ();
LOCAL MEM_FILE_DESC *memOpen ();
LOCAL STATUS memFileStatGet ();
LOCAL int memRead ();
LOCAL int memWrite ();
LOCAL int memClose ();
LOCAL STATUS memIoctl ();

extern STATUS  memDrv (void);
extern STATUS  memDevCreate (char *name, char *base, int length);
extern STATUS  memDevCreateDir (char * name, MEM_DRV_DIRENTRY * files, int
numFiles);
extern STATUS  memDevDelete (char *name);


/*******************************************************/
/* memDrv－此函数用于安装内存驱动。                    */
```

```
/* 此函数的任务是初始化内存驱动程序。在调用此驱动的其他函数之前 */
/* 必须首先调用此函数。                                    */
/* 返回值：正常时返回 OK，如果 I/O 系统不能安装驱动返回 ERROR。 */
/********************************************** ********/

STATUS memDrv (void)

    {
    if (memDrvNum > 0)
     return (OK);    /* 驱动已经安装，返回 OK */

    memDrvNum = iosDrvInstall ((FUNCPTR) memOpen, (FUNCPTR) NULL,
                    (FUNCPTR) memOpen, memClose,
                    memRead, memWrite, memIoctl);

    return (memDrvNum == ERROR ? ERROR : OK);
    }
/*********************************************************/
/* memDevCreate 一此函数用于创建一个内存设备。                */
/* 此函数创建的内存设备包含一个单一的文件，分配给设备的内存是绝    */
/* 对内存。它起始于“base”，内存的大小由参数“length”指定。          */
/* 返回值：正常情况下返回 OK，如果内存不足或者 I/O 系统不可添加设   */
/* 备则返回 ERROR，ERRNO 为 S_ioLib_NO_DRIVER                */
/*********************************************************/

STATUS memDevCreate
    (
    char * name,         /* 设备名字 */
    char * base,      /* 设备所用的内存起始地址 */
    int    length        /* 设备所用的内存字节长度 */
    )
    {
    STATUS status;
    FAST MEM_DEV *pMemDv;

    if (memDrvNum < 1)
        {
        errnoSet (S_ioLib_NO_DRIVER);
        return (ERROR);
        }

    if ((pMemDv = (MEM_DEV *) calloc (1, sizeof (MEM_DEV))) == NULL)
     return (ERROR);

    pMemDv->dir.name  = "";
    pMemDv->dir.base  = base;
    pMemDv->dir.pDir  = NULL;
    pMemDv->dir.length = length;
    pMemDv->allowOffset = 1;
```

```
    /*****************************************/
    /* 声明字节、字以及长字方式的访问。              */
/*****************************************/

    status = iosDevAdd ((DEV_HDR *) pMemDv, name, memDrvNum);

    if (status == ERROR)
     free ((char *) pMemDv);

    return (status);
    }

/*********************************************************/
/* memDevCreateDir 一此函数是内存设备的目录创建函数。        */
/* 返回值：正常情况下返回 OK，如果内存不足或者 I/O 系统不可     */
/* 添加设备则返回 ERROR，ERRNO 为 S_ioLib_NO_DRIVER。          */
/*********************************************************/

STATUS memDevCreateDir
    (
    char * name,                  /* 内存设备名字 */
    MEM_DRV_DIRENTRY * files,      /* 目录数组的入口 */
    int    numFiles               /* 目录数组入口的数目 */
    )
    {
    STATUS status;
    FAST MEM_DEV *pMemDv;

    if (memDrvNum < 1)
        {
        errnoSet (S_ioLib_NO_DRIVER);
        return (ERROR);
        }

    if ((pMemDv = (MEM_DEV *) calloc (1, sizeof (MEM_DEV))) == NULL)
     return (ERROR);

    pMemDv->dir.name  = "";
    pMemDv->dir.base  = NULL;
    pMemDv->dir.pDir  = files;
    pMemDv->dir.length = numFiles;

    /*******************************************/
    /*    对于打开的文件，偏移量直接赋 0。      */
    /*******************************************/
    pMemDv->allowOffset = 0;


    status = iosDevAdd ((DEV_HDR *) pMemDv, name, memDrvNum);
```

```
    if (status == ERROR)
     free ((char *) pMemDv);

    return (status);
    }

/*********************************************************/
/* memDevDelete －此函数用于删除一个内存设备。                  */
/* 返回值：正常情况下返回 OK，如果设备则不存在返回 ERROR。          */
/*********************************************************/

STATUS memDevDelete
    (
    char * name          /* 内存设备名字 */
    )
    {
    DEV_HDR * pDevHdr;

    /* 获取内存设备名字所对应设备的指针 */

    if ((pDevHdr = iosDevFind (name, NULL)) == NULL)
        return (ERROR);

    /* 在 VxWorks 系统的 I/O 系统中删除这个设备。 */

    iosDevDelete (pDevHdr);

    /* 释放设备指针 */

    free ((MEM_DEV *) pDevHdr);

    return (OK);
    }

/*******************************************************/
/* memFindFile －此函数用来通过文件名形式查找内存文件。    */
/* 返回值： 正常情况下返回查找到的文件描述符，如果文件名是    */
/* 一个无效值则返回 ERROR                    */
/*******************************************************/

LOCAL MEM_DRV_DIRENTRY *memFindFile
    (
    MEM_DEV *pMemDv,
    char *name,              /* 要查找的文件名字 */
    MEM_DRV_DIRENTRY *pDir,       /* 指向内存设备描述符的指针 */
    int *pOffset                  /* 返回文件偏移 */
    )
    {
    MEM_DRV_DIRENTRY *pFile = NULL;
```

```
    *pOffset = 0;

    /* 禁止文件名等于 NULL */
    if (name == NULL)
     name = "";
    while (*name == '/')
     ++name;

    if (strcmp (pDir->name, name) == 0)
     {
     pFile = pDir;
     }
    else if (strncmp (pDir->name, name, strlen (pDir->name)) == 0)
     {
     int index;

     name += strlen (pDir->name);
     if (pDir->pDir != NULL)
         {
         /*在目录下面查找指定的文件 */
         for (index = 0; index < pDir->length; ++index)
          {
          pFile = memFindFile (pMemDv, name, &pDir->pDir[index], pOffset);
          if (pFile != NULL) break;
          }
         }
     else if (pMemDv->allowOffset)
         {
         int off = 0;

         while (*name == '/')
          ++name;
         if (sscanf (name, "%d", &off) == 1)
          {
          pFile = pDir;
          *pOffset = off;
          }
         }
     }

    return pFile;
}

/*******************************************************/
/* memOpen  -  此函数用来打开一个内存文件。            */
/* 返回值:  正常情况下返回查找到的文件描述符，如果文件名是    */
/* 一个无效值则返回 ERROR，其 ERRNO 是 EINVAL。       */
/*******************************************************/

LOCAL MEM_FILE_DESC *memOpen
```

```
    (
    MEM_DEV *pMemDv,      /* 指向内存设备描述符的指针 */
    char *name,           /* 要打开的文件名字 */
/***************************************/
/* 这里支持的文件访问模式主要有如下三种：     */
/*O_RDONLY 表示只读访问模式。         */
/*O_WRONLY 表示只写访问模式。         */
/*O_RDWR) 表示既可读又可写的访问模式        */
/***************************************/

    int mode
    )
    {
    MEM_DRV_DIRENTRY *pFile = NULL;
    MEM_FILE_DESC *pMemFd = NULL;
    int offset = 0;
    int isDir = 0;

    pFile = memFindFile (pMemDv, name, &pMemDv->dir, &offset);

    if (pFile != NULL)
     {
     /* 目录只能以只读模式打开 */
     isDir = (pFile->pDir != NULL);
     if (!isDir || mode == O_RDONLY)
         {
         /* 获取一个空闲的文件描述符 */
         pMemFd = (MEM_FILE_DESC *) calloc (1, sizeof (MEM_FILE_DESC));
         if (pMemFd != NULL)
          {
          pMemFd->pDevice = pMemDv;
          pMemFd->pDir = pFile;
          pMemFd->offset = offset;
          pMemFd->mode   = mode;
          }
         }
     }

    if (pMemFd != NULL)
     return pMemFd;
    else
     {
     errnoSet (EINVAL);
     return (MEM_FILE_DESC *) ERROR;
     }
    }

/*************************************************************/
/* memFileStatGet 一这个函数用来获取文件的状态信息。          */
/* 此函数受 ioctl() 函数调用。ioctl()调用此函数时必须使用函数代码      */
```

```
/* FIOFSTATGET。由于 ioctl()调用此函数时已经传入了一个状态数          */
/* 据结构，因此，此函数直接将目录入口信息即文件信息填入这个结构      */
/* 然后返回。                                                        */
/* 返回值：正常情况下为 OK，出错时为 ERROR。                         */
/* 此函数的 ERRNO 为 EINVAL。                                        */
/*******************************************************************/

LOCAL STATUS memFileStatGet
    (
    MEM_FILE_DESC * pfd,              /* 指向文件描述符的指针 */
    struct stat    * pStat                    /* 用于填入状态信息的结构 */
    )
    {
    MEM_DEV *pMemDv = pfd->pDevice;       /* 指向设备的指针*/
    MEM_DRV_DIRENTRY *pDir = pfd->pDir;  /* 指向文件信息的指针 */
    int isDir = 0;

    if (pStat == NULL || pDir == NULL)
     {
     errnoSet (EINVAL);
     return ERROR;
     }

    bzero ((char *) pStat, sizeof (struct stat)); /* 将 stat 结构清零 */
    isDir = (pDir->pDir != NULL);

    /* 填充 stat 结构*/

    pStat->st_dev     = (ULONG) pMemDv;
    pStat->st_ino     = 0;              /* 无文件序列号*/
    pStat->st_nlink   = 1;               /* 只有一条连接*/
    pStat->st_uid     = 0;               /* 无用户 ID */
    pStat->st_gid     = 0;               /* 无组 ID */
    pStat->st_rdev    = 0;               /* 无特定设备 ID */
    pStat->st_size    = isDir ? 0 : pDir->length;   /* 字节表示的文件大小*/
    pStat->st_atime   = 0;           /* 无上次访问时间*/
    pStat->st_mtime   = 0;                /* 无上次修改时间*/
    pStat->st_ctime   = 0;
    pStat->st_attrib  = 0;                /* 文件属性字节*/


/*************************************************/
/* 将文件的访问权限设置为既可读又可写。对于目录     */
/* 还加上可执行（搜索）的权限。                    */
/*************************************************/

    pStat->st_mode |= (S_IRUSR | S_IRGRP | S_IROTH |
                S_IWUSR | S_IWGRP | S_IWOTH);
    if (isDir)
     pStat->st_mode |= (S_IXUSR | S_IXGRP | S_IXOTH);
```

```
    /* 判断是目录还是文件 */
    if (isDir)
     pStat->st_mode |= S_IFDIR;      /* 判定是目录 */
    else
     pStat->st_mode |= S_IFREG;          /* 判定是一个普通文件 */

    return (OK);
}

/*************************************************************/
/* memRead一 此函数用于读取一个内存文件。                    */
/* 返回值：正常情况下返回读取的字节数，这个值可能比请求读取的字节数小。*/
/*  如果文件是以只写模式打开则返回 ERROR。如果文件中没有数据则返回 0。*/
/*  此函数的 ERRNO 是 EINVAL 和 EISDIR。                     */
/*************************************************************/

LOCAL int memRead
    (
    MEM_FILE_DESC *pfd,              /* 要写入字节的文件描述符*/
    char *buffer,                    /* 用于写入的缓冲区*/
    int maxbytes                    /* 读入到缓冲区中的最大字节数*/
    )
    {
    MEM_DRV_DIRENTRY *pDir = pfd->pDir;   /* pointer to file info */
    int lLeft;

    /* 如果模式不对或者它是一个目录则返回错误*/
    if (pfd->mode == O_WRONLY)
     {
     errnoSet (EINVAL);
     return (ERROR);
     }
    if (pDir == NULL || pDir->pDir != NULL)
     {
     errnoSet (EISDIR);
     return (ERROR);
     }

    /* 计算还剩多少字节要读取*/
    lLeft = pDir->length - pfd->offset;
    if (lLeft < 0) lLeft = 0;

/*****************************************/
/* 将最大读取字节数 maxbytes 限定为缓冲区的长度    */
/* 和文件内所剩字节数这两个值中的较小者。          */
/*****************************************/

    if (maxbytes > lLeft) maxbytes = lLeft;
```

```
    if (maxbytes > 0)
        {
        bcopy (pDir->base + pfd->offset, buffer, maxbytes);
        pfd->offset += maxbytes;
        }

    return (maxbytes);
}

/*************************************************************/
/* memWrite 一此函数用于写入一个内存文件。                    */
/* 返回值：正常情况下返回已经写入的字节数，如果传递的字节数超出  */
/* 内存的边界或者打开文件的模式是只读模式则返回 ERROR，其 ERRNO  */
/* 是 EINVAL 分别是 EISDIR。                                  */
/*************************************************************/

LOCAL int memWrite
    (
    MEM_FILE_DESC *pfd,             /* 要写入字节的文件描述符*/
    char *buffer,                   /* 用于写入的缓冲区*/
    int nbytes                          /* 从缓冲区中写入的字节数*/
    )
    {
    MEM_DRV_DIRENTRY *pDir = pfd->pDir; /* 指向文件信息的指针*/
    int lLeft;

    /* 如果模式不对或者它是一个目录则返回错误*/
    if (pfd->mode == O_RDONLY)
     {
     errnoSet (EINVAL);
     return (ERROR);
     }
    if (pDir == NULL || pDir->pDir != NULL)
     {
     errnoSet (EISDIR);
     return (ERROR);
     }

    /* 计算还剩多少字节要写入*/
    lLeft = pDir->length - pfd->offset;
    if (lLeft < 0) lLeft = 0;

    /* 如果写入太多，则对其进行裁减*/
    if (nbytes > lLeft) nbytes = lLeft;

    if (nbytes > 0)
        {
     bcopy (buffer, pDir->base + pfd->offset, nbytes);
     pfd->offset += nbytes;
        }
```

```
    return (nbytes);
}

/*************************************************************/
/* memIoctl 一此函数用于实现特定的设备控制功能。                    */
/* 此函数实现的控制功能仅限于以下几项：FIONREAD, FIOSEEK,          */
/* FIOWHERE, FIOGETFL, FIOFSTATGET, 和 FIOREADDIR。          */
/* 返回值：正常情况下返回 OK，如果搜索超出了内存的边界则返回     */
/*  ERROR，其 ERRNO 可以是 EINVAL 和 S_ioLib_UNKNOWN_REQUEST    */
/*************************************************************/

LOCAL STATUS memIoctl (pfd, function, arg)
    FAST MEM_FILE_DESC *pfd;       /* 用于控制的描述符 */
    FAST int function;                  /* 函数代码*/
    int   arg;               /* 其他可扩展传入的变量 */

    {
    MEM_DRV_DIRENTRY *pDir = pfd->pDir;  /* 指向文件信息的指针 */
    DIR *dirp;
    int isDir = 0;
    int status = OK;

    if (pDir == NULL)
     {
     errnoSet (EINVAL);
     status = ERROR;
     }
    else
     {
     isDir = (pDir->pDir != NULL);
     switch (function)
         {
         case FIONREAD:          /* 读取功能 */
          if (isDir)
              {
              errnoSet (EINVAL);
              status = ERROR;
              }
          else
              {
              *((int *) arg) = pDir->length - pfd->offset;
              }
          break;

         case FIOSEEK:            /* 查找功能 */
          if (isDir || arg > pDir->length)
              {
              errnoSet (EINVAL);
              status = ERROR;
```

```
            }
         else
            {
            pfd->offset = arg;
            }
         break;

        case FIOWHERE:
         if (isDir)
             status = 0;
         else
             status = pfd->offset;
         break;

        case FIOGETFL:
         *((int *) arg) = pfd->mode;
         break;

        case FIOFSTATGET:       /* 获取状态 */
         status = memFileStatGet (pfd, (struct stat *) arg);
         break;

        case FIOREADDIR:       /* 目录读取功能 */

/*******************************************/
/* 要求的目录入口的索引通过 dirp->dd_cookie 传递。     */
/* 我们在 dirp->dd_dirent 中返回它的名字。          */
/*******************************************/

         dirp = (DIR *) arg;
         if (!isDir || dirp->dd_cookie >= pDir->length)
             {
             errnoSet (EINVAL);
             status = ERROR;
             }
         else
             {
             strcpy (dirp->dd_dirent.d_name, pDir->pDir[dirp->dd_cookie].name);
             ++dirp->dd_cookie;
             }
         break;

        default:
         errnoSet (S_ioLib_UNKNOWN_REQUEST);
         status = ERROR;
         break;
        }
     }

    return (status);
```

```
}

/*************************************************************/
/* memClose 一此函数用于关闭一个内存文件。                    */
/* 返回值:  正常情况下返回 OK，如果文件不可关闭或者找不到入口则      */
/*             返回 ERROR。                                  */
/*************************************************************/

LOCAL STATUS memClose (pfd)
    MEM_FILE_DESC *pfd;   /* 要关闭文件的文件描述符*/

    {
    free (pfd);
    return OK;
}

/******************** memDrv.c 程序的结尾*****************************/
```

7.10　RamDisk 驱动

本节我们介绍一个 RamDisk 的驱动程序开发实例。本驱动程序可以模拟一个磁盘驱动程序，但是，实际上我们将所有数据保存在内存中，而不是在所谓的“磁盘”中。

当我们创建虚拟“磁盘”时，直接指定“磁盘”所在的内存位置以及所使用内存的大小。以这种形式创建的 RAM 磁盘非常实用。例如，可以利用 RAM 磁盘共享不同处理器之间关键数据，可以利用 RAM 磁盘保存 VxWorks 系统引导时的关键数据。

本驱动程序中的大部分函数只能通过 VxWorks 系统来访问。但是有两个函数可以在上层应用程序中直接调用。第一个可直接调用的函数是 ramDrv( )，此函数没有实现什么功能，主要用于与真正的磁盘设备驱动程序兼容，提供一个初始化入口。如果仅仅使用 RAM 形式的“磁盘”，也可以不调用 ramDrv( )。第二个函数是 ramDevCreate()，在创建 RAM 形式的“磁盘”时，必须调用这个函数。

下面详细介绍函数 ramDevCreate()。用于创建 RAM 磁盘的内存可以事先申请。如果是这样的话，ramDevCreate()的参数 ramAddr 可以是事先申请到的内存设备地址。要是没有事先申请内存，在利用 ramDevCreate()创建 RAM 磁盘时，它将使用 malloc()调用自动申请内存，在这种情况下,可将输入参数 ramAddr 直接设置为 0。ramDevCreate()的输入参数 bytesPerBlk 表示 RAM 磁盘上每个逻辑块的大小。如果传输的 bytesPerBlk 值为 0，那么将逻辑块的大小设置为默认值 512。ramDevCreate()的输入参数 blksPerTrack 表示 RAM 磁盘上每个逻辑磁道上的逻辑块数。如果参数 blksPerTrack 的传入值为 0，那么每个磁道上的逻辑块数设置为 nBlocks，就是说整个磁盘定义为只有一个磁道。ramDevCreate()的输入参数 nBlocks 表示整个磁盘的大小（单位为块）。如果 nBlocks 传入的参数为 0，那么 RAM 磁盘的大小便是默认值。RAM 磁盘大小默认值的确定步骤是这样的，首先确定系统中最大内存区域的字节大小 K，然后取 K 与 51 200 字节两个值中的较小者作为 M。默认磁盘的块数目为 M 除以参数 bytesPerBlk

得到的整数值。ramDevCreate()的输入参数 blkOffset 声明是以块为单位的偏移量，它表示当读写 RAM 磁盘设备时从何处开始。在进行磁盘访问时，这个偏移量需要加到文件系统传入的块数上。VxWorks 系统的文件系统总是使用 0 作为设备开始。如果调用 ramDevCreate()函数时传入了有效的 ramAddr 参数，即声明了确切的地址。那么偏移量可以正常使用。否则参数 blkOffset 直接传入 0 即可。调用 ramDevCreate()函数后，正常情况下将返回指向块设备结构的指针，如果用于创建 RAM 磁盘或者设备结构的内存申请不到则返回 NULL。

当调用 ramDevCreate()函数创建了 RAM 磁盘以后，必须将 RAM 磁盘与一个确切的名字和文件系统（如 dosFs、 rt11Fs 和 rawFs）关联在一起。这可以利用文件系统的设备初始化程序或者创建文件系统的程序如 dosFsDevInit()和 dosFsMkfs()来完成。调用 ramDevCreate()函数时返回的指向块设备结构的指针。这个块设备 BLK_DEV 结构包含了描述磁盘设备物理属性的字段，也包括描述 ramDrv 驱动程序函数地址的字段。BLK_DEV 结构的地址必须通过文件系统的设备初始化程序或者创建文件系统的程序传递给要创建的文件系统（如 dosFs、rt11Fs 和 rawFs）中。只有在 RAM 磁盘与一个确切的名字和文件系统关联在一起以后，才可以使用 RAM 磁盘设备。

下面举一些调用 ramDevCreate()函数创建 RAM 磁盘的例子。在这个例子中，将自动申请内存，创建一个 200K 字节大小的 RAM 磁盘。此 RAM 磁盘中只有一个磁道，其块的大小为 512 字节，无块偏移量。此 RAM 磁盘设备指定的设备名为“DEV1”，在其上初始化的文件系统是 dosFs，创建的代码如下：

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

dosFsMkfs() 程序以默认参数调用 dosFsDevInit()函数在磁盘上初始化文件系统，不过真正的初始化工作是 VxWorks 系统中 I/O 系统的 ioctl()函数完成的，其函数调用代码为 FIODISKINIT 。如果 RAM 磁盘的内存已经包含了一个在其他地方创建的磁盘映像，那么在调用 ramDevCreate()函数时，第一个输入参数应该等于内存的地址，而且其他后续用于格式化的输入参数如 bytesPerBlk、blksPerTrack、nBlocks 和 blkOffset 必须等于创建磁盘映像时使用的值。例如，

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

在这个例子中，必须使用 dosFsDevInit()函数代替上一个例子的 dosFsMkfs()函数，因为文件系统已经在磁盘上存在，不可再次初始化。如果要在一个已经引导的 VxWorks 系统上的内存位置创建一个 RAM 磁盘，那么按这个例子进行操作是相当方便的。这样 RAM 磁盘上的内容将被保留。上面介绍的例子是针对 dosFs 的，要利用 rt11FsDevInit()和 rt11FsMkfs()创建 rt11Fs 文件系统或者利用 rawFsDevInit()创建 rawFs 文件系统关联，操作方法基本相同。

RAM“磁盘”设备一旦创建，就必须与一个设备名字和文件系统关联。这里谈到的文件系

统可以是 VxWorks 系统支持的默认文件系统，也可以是用户的自定义文件系统。

函数 ramDrv( )几乎不执行任何实际操作，编写此函数的目的是保持与其他磁盘驱动程序的兼容。一般的磁盘设备驱动程序必须包含一个用于完成初始化工作的初始化函数。在编译 VxWorks 映像前，可在 VxWorks 系统的 usrConfig.c 中利用初始化函数将 RAM 磁盘设备的驱动程序与物理设备连接起来。除此用途外，其他场合可以不调用这个函数。

函数 ramBlkRd()用来从 RAM 磁盘卷中读出一个或多个块。读取的起始块号由输入参数 startBlk 确定。此函数在读取数据时首先计算字节偏移，然后直接将 RAM 磁盘中的数据拷贝到缓冲区中。如果在创建 RAM 磁盘的过程中，调用 ramDevCreate()时声明了具体的块偏移量，那么函数 ramBlkRd()在读取操作之前，首先会将块偏移量与参数 startBlk 相加确定读取的真正起始地址。

函数 ramBlkWrt()用来向一个特定 RAM 磁盘卷中写入一个或多个块。写入时起始的块号由输入参数 startBlk 确定。此函数在写入时首先计算字节偏移，然后直接将缓冲区中的数据拷贝到 RAM 磁盘中。如果在创建 RAM 磁盘的过程中，调用 ramDevCreate()时声明了具体的块偏移量，那么函数 ramBlkWrt()在写入操作之前，首先会将块偏移量与参数 startBlk 相加确定真正的起始地址。

本实例介绍的 RAM 驱动程序像真正的磁盘驱动程序一样，可以响应 VxWorks 系统 I/O 系统中 ioctl()函数的调用，这样就为设备接口的属性控制提供了较好的交互方式。当文件系统不能处理特定的 ioctl()调用请求时，它就将其传递给 RAMDISK 驱动程序。虽然我们的 RAMDISK 驱动程序并不控制真正的磁盘，但是驱动程序确实可以处理 FIODISKFORMAT 这样的调用请求，为了方便起见，此调用总是返回 OK。驱动程序对其他 ioctl()函数的属性控制调用请求返回错误，同时设置调用任务的“errno”为 S_ioLib_UNKNOWN_REQUEST。

本驱动程序编写的代码文件为 ramDrv.c，下面列出此文件并对其做详细说明。

```
/****************** ramDrv.c 程序的开始*****************************/
/*说明： ramDrv.c 程序实现了一个简单的 RAM 磁盘驱动程序。*/

/*包含头文件*/

#include "vxWorks.h"
#include "blkIo.h"
#include "ioLib.h"
#include "iosLib.h"
#include "memLib.h"
#include "errnoLib.h"
#include "string.h"
#include "stdlib.h"
#include "stdio.h"

#define DEFAULT_DISK_SIZE  (min (memFindMax()/2, 51200))
#define DEFAULT_SEC_SIZE   512

typedef struct                  /* RAM_DEV 是 RAM 磁盘设备的描述符结构*/
    {
    BLK_DEV  ram_blkdev;        /* 通用块设备结构/
```

```
    char    *ram_addr;              /* RAM 磁盘的内存位置*/
    int         ram_blkOffset;      /* 这个设备从 ram_addr 起始的块偏移量 */
    } RAM_DEV;


/* 函数声明*/

LOCAL STATUS ramIoctl ();
LOCAL STATUS ramBlkRd ();
LOCAL STATUS ramBlkWrt ();


/*************************************************************/
/* ramDrv －此函数用于初始化 RAM 磁盘驱动程序。                */
/*此函数是为了兼容其他磁盘驱动而创建的，这里不完成任何具体操作。    */
/*返回值：总是返回 OK。                                      */
/*************************************************************/

STATUS ramDrv (void)

    {
    return (OK);            /* 不需要进行特别的初始化工作 */
    }

/*************************************************************/
/* ramDevCreate －此函数用来创建一个 RAM 磁盘设备。            */
/* 返回值： 正常情况下将返回指向块设备结构的指针，如果用于创建 RAM    */
/* 磁盘或者设备结构的内存申请不到则返回 NULL。                 */
/*************************************************************/

BLK_DEV *ramDevCreate
    (
    char  *ramAddr,
    FAST int   bytesPerBlk,
    int        blksPerTrack,
    int        nBlocks,
    int        blkOffset
    )


    {
    FAST RAM_DEV    *pRamDev;
    FAST BLK_DEV *pBlkDev;


    /*设置默认值 */

    if (bytesPerBlk == 0)
     bytesPerBlk = DEFAULT_SEC_SIZE;
```

```
if (nBlocks == 0)
 nBlocks = DEFAULT_DISK_SIZE / bytesPerBlk;

if (blksPerTrack == 0)
 blksPerTrack = nBlocks;


/* 为设备申请 RAM_DEV 结构的存储空间 */

pRamDev = (RAM_DEV *) malloc (sizeof (RAM_DEV));
if (pRamDev == NULL)
 return (NULL);                    /* 内存不足 */


/* 初始化 RAM_DEV 中的 BLK_DEV 结构 */

pBlkDev = &pRamDev->ram_blkdev;

pBlkDev->bd_nBlocks      = nBlocks;
pBlkDev->bd_bytesPerBlk  = bytesPerBlk;
pBlkDev->bd_blksPerTrack = blksPerTrack;

pBlkDev->bd_nHeads       = 1;
pBlkDev->bd_removable    = FALSE;
pBlkDev->bd_retry         = 1;
pBlkDev->bd_mode        = O_RDWR;
pBlkDev->bd_readyChanged = TRUE;

pBlkDev->bd_blkRd       = ramBlkRd;       /* 读取块的函数 */
pBlkDev->bd_blkWrt      = ramBlkWrt;  /* 写入块的函数 */
pBlkDev->bd_ioctl        = ramIoctl;     /* I/O 控制函数 */
pBlkDev->bd_reset        = NULL;
pBlkDev->bd_statusChk    = NULL;


/* 初始化设备结构的其他字段 */

if (ramAddr == NULL)
 {
 pRamDev->ram_addr = (char *) malloc ((UINT) (bytesPerBlk * nBlocks));

 if (pRamDev->ram_addr == NULL)
     {
     free ((char *) pRamDev);
     return (NULL);                /* 内存不足 */
     }
 }
else
 pRamDev->ram_addr = ramAddr;
```

```
    pRamDev->ram_blkOffset = blkOffset;      /* 块的偏移量 */

    return (&pRamDev->ram_blkdev);
    }

/*************************************************************/
/* ramIoctl 一此函数用于完成特定的设备控制功能。            */
/* 此函数在文件系统不能处理 ioctl()中的特定功能时被调用。    */
/* 调用此函数是如果使用的函数代码是 FIODISKFORMAT ，那么此 */
/* 函数总是返回 OK，这是因为 RAM 磁盘设备根本不需要格式化。  */
/* 其他调用均返回 ERROR。                                    */
/* 返回值：正常情况下返回 OK，出错时返回 ERROR。             */
/* 此函数相关的 ERRNO 是 S_ioLib_UNKNOWN_REQUEST。          */
/*************************************************************/

LOCAL STATUS ramIoctl (pRamDev, function, arg)
    RAM_DEV *pRamDev;             /* 指向设备结构的指针 */
    Int        function;               /* 调用函数代码 */
    Int        arg;                /* 其他扩展变量 */

    {
    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);
    }

/*********************************************************/
/* ramBlkRd 一此函数用来从 RAM 磁盘卷中读出一个或多个块。  */
/* 返回值：总是返回 OK。                                  */
/*********************************************************/

LOCAL STATUS ramBlkRd (pRamDev, startBlk, numBlks, pBuffer)
    FAST RAM_DEV    *pRamDev;
    int            startBlk;
    int            numBlks;
    char      *pBuffer;

    {
```

```
    FAST int       bytesPerBlk;   /* 每块中的字节数 */


    bytesPerBlk = pRamDev->ram_blkdev.bd_bytesPerBlk;

    /*增加磁盘偏移量*/

    startBlk += pRamDev->ram_blkOffset;

    /* 读取块中的数据 */

    bcopy ((pRamDev->ram_addr + (startBlk * bytesPerBlk)), pBuffer,
       bytesPerBlk * numBlks);

    return (OK);
    }

/*******************************************************/
/* ramBlkWrt －此函数用来向 RAM 磁盘卷中写入一个或多个块。     */
/* 返回值：总是返回 OK。                                  */
/*******************************************************/

LOCAL STATUS ramBlkWrt (pRamDev, startBlk, numBlks, pBuffer)
    FAST RAM_DEV     *pRamDev;
    int              startBlk;
    int              numBlks;
    char       *pBuffer;

    {
    FAST int       bytesPerBlk;   /* 每块中的字节数 */


    bytesPerBlk = pRamDev->ram_blkdev.bd_bytesPerBlk;

    /* 增加磁盘偏移量 */

    startBlk += pRamDev->ram_blkOffset;

    /* 写磁盘块 */

    bcopy (pBuffer, (pRamDev->ram_addr + (startBlk * bytesPerBlk)),
       bytesPerBlk * numBlks);

    return (OK);
}

/****************** ramDrv.c 程序的结尾*****************************/
```

7.11 WDB 应 用

VxWorks 系统可以使用 WDB 代理与 Tornado 开发环境进行通信。这是目标机与开发主机通信的重要方式。本节将介绍 WDB 代理的程序实现原理并分析实现的细节。

WDB 代理与 Tornado 开发环境使用包含在 WDB_COMM_IF 结构中的两个函数指针来完成通信功能。牵涉到的两个函数分别是 sendto()和 rcvfrom()。实际上，WDB 代理使用简单的 UDP/IP 协议栈作为传输的承载。WDB 代理支持两种操作模式，一种是中断模式，一种是查询模式。这两种模式可以在系统运行过程中动态改变。

在 WDB 代理初始化的过程中，将调用函数 udpCommIfInit()初始化 a WDB_COMM_IF 数据结构。初始化以后，指向此结构的指针将传递给 WDB 代理的函数 wdbInstallCommIf()，这样就完成了通信接口的安装。

值得注意的是，当运行在目标板的 WDB 代理与 Tornado 开发环境进行通信时，不一定非要调用 udpCommIfInit()来完成初始化工作。因此，读者完全可以自己定义一个通信端口初始化函数。但是自定义的初始化函数必须要完成 WDB_COMM_IF 数据结构的初始化，并且将其安装到 WDB 代理中。WDB 代理初始化 WDB_COMM_IF 数据结构的目的完全是为了调用标准的 UDP 套接字函数。

本节介绍的套接字应用实际上只支持“中断”模式，并不支持查询模式。这主要是为了防止任何查询例程对 VxWorks 系统内核的访问。正因为这样，本节列出的 WDB 通信实例实际上只支持任务模式的代理例如 tWDB。

在实际系统开发时，使用 WDB 代理只需要在 VxWorks 安装目录的“..\target\src\wdb”中替换或者改变其中的源代码文件 wdbUdpLib.c 和 wdbUdpSockLib.c。这样目标板中 VxWorks 系统的 WDB 代理便会按照自己的设计要求与 Tornado 开发环境进行通信。

WDB 应用只需要两个源程序 UdpLib.c 和 UdpSockLib.c。前者实现通信端口的封装，后者完成 UDP 套接字的通信应用。下面列出这两个源程序，同时给出详尽的说明。

```
/******************** wdbUdpLib.c 程序的开始 ********************/
/*说明：wdbUdpLib.c 程序是 WDB 的通信接口，以 UDP 数据报为基础。*/

/*包含头文件*/

#include "wdb/wdb.h"
#include "wdb/wdbCommIfLib.h"
#include "wdb/wdbLibP.h"
#include "wdb/wdbRtIfLib.h"
#include "wdb/wdbUdpLib.h"
#include "wdb/wdbMbufLib.h"
#include "string.h"
#include "netinet/in_systm.h"
#include "netinet/in.h"
#include "netinet/ip.h"
#include "netinet/udp.h"
```

```
#define IP_HDR_SIZE    20
#define UDP_HDR_SIZE  8
#define UDP_IP_HDR_SIZE IP_HDR_SIZE + UDP_HDR_SIZE

#define MILLION        1000000
#ifndef MIN
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif

/* 全局变量定义 */

static u_int    commMode;     /* 通信模式主要是查询模式和中断模式 */
static int    agentIp;              /* 网络字节位序 */

static struct mbuf *   pInputMbufs;
static WDB_DRV_IF * pWdbDrvIf;

static void (*udpHookRtn) (u_int arg);
static u_int udpHookArg;

static void *    readSyncSem;
static void *    writeSyncSem;
static char udpIpHdrBuf [128];  /* 头部使用 128 字节的缓冲 */

/* 函数定义 */

static int  udpRcvfrom  (WDB_COMM_ID commId, caddr_t addr, u_int len,    /* UDP 接收函数 */
               struct sockaddr_in *pAddr, struct timeval *tv);
static int  udpSendto  (WDB_COMM_ID commId, caddr_t addr, u_int len,     /* UDP 发送函数 */
               struct sockaddr_in *pAddr);
static int  udpModeSet    (WDB_COMM_ID commId, u_int newMode);      /* 模式设置函数 */
static int  udpCancel  (WDB_COMM_ID commId);
static int  udpHookAdd    (WDB_COMM_ID commId, void (*rout)(), u_int arg);  /* 钩联函数添
加 */

extern int wdbCksum (char * pData, int nBytes);

/*******************************************************************/
/* udpCommIfInit - 此函数用于初始化 WDB 通信函数指针。          */
/*******************************************************************/

STATUS udpCommIfInit
    (
    WDB_COMM_IF * pWdbCommIf,         /* 要安装的函数的指针 */
    WDB_DRV_IF *  pDrvIf
    )
    {
    static initialized = FALSE;

    pWdbCommIf->rcvfrom    = udpRcvfrom;
```

```
    pWdbCommIf->sendto = udpSendto;
    pWdbCommIf->modeSet = udpModeSet;
    pWdbCommIf->cancel  = udpCancel;
    pWdbCommIf->hookAdd = udpHookAdd;
    pWdbCommIf->notifyHost = NULL;

    pWdbDrvIf     = pDrvIf;

    commMode = WDB_COMM_MODE_INT;  /* 默认通信模式是中断模式 */

    if (pWdbRtIf->semCreate != NULL)
     {
     readSyncSem = pWdbRtIf->semCreate();  /* 读同步信号量 */
     writeSyncSem = pWdbRtIf->semCreate();  /* 写同步信号量 */
     }

    if (initialized)
     return (OK);

    initialized = TRUE;

    return (OK);
    }


/*************************************************************/
/* udpModeSet 一此函数用于设置 UDP 通信模式，可设置的模式的是查询模    */
/* 式 POLL 和中断模式 INT。                                    */
/*返回值： 正常情况下返回 OK，否则如果设置的模式不支持则返回 ERROR。*/
/*************************************************************/

static int udpModeSet
    (
    WDB_COMM_ID commId,
    u_int newMode
    )
    {
    if (pWdbDrvIf->modeSetRtn == NULL)
     return (ERROR);

    if ((*pWdbDrvIf->modeSetRtn)(pWdbDrvIf->devId, newMode) == ERROR)
     return (ERROR);

    commMode = newMode;

    return (OK);
    }

/************************************************/
/* udpRcv 一此函数用于从驱动程序接收 UDP/IP 数据报。    */
```

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

void udpRcv
    (
    struct mbuf * pMbuf       /* 接收缓冲 */
    )
    {
    /* 将数据保存以供下一步处理 */

    if (pInputMbufs != NULL)        /* 只允许一个请求排队 */
     {
     wdbMbufChainFree (pInputMbufs);
     return;
     }
    pInputMbufs        = pMbuf;

    /* I 判断通信模式，然后调用输入 hook 函数 */

    if ((commMode == WDB_COMM_MODE_INT) && (udpHookRtn != NULL))
     {
     (*udpHookRtn) (udpHookArg);
     return;
     }

    /* 判断是否是正常任务模式，然后为任务解除阻塞 */

    if (commMode == WDB_COMM_MODE_INT)
     {
     pWdbRtIf->semGive (readSyncSem);
     }
    }

/*************************************************************/
/* udpRcvfrom ―此函数用于读取一个 UDP/IP 数据报。              */
/* 此函数读取数据报时已经去除了 UDP/IP 头部。此函数假定从驱动程序 */
/* 中接收到的 mbuf 链中的第一个 mbuf 含有 UDP/IP 头部。        */
/*************************************************************/

static int udpRcvfrom
    (
    WDB_COMM_ID   commId,
    caddr_t    addr,
    u_int  len,
    struct sockaddr_in *pAddr,
    struct timeval *tv
    )
    {
    struct udphdr *   pUdpHdr;
    struct ip *           pIpHdr;
    u_int       nBytes;
```

```
    /* WDB_COMM_MODE_INT 相当于“任务模式”。使用信号量等待数据。*/

    if (commMode == WDB_COMM_MODE_INT)
     pWdbRtIf->semTake (readSyncSem, tv);

    /* 如果不是“任务模式”，则查询驱动程序直到数据到来或者定时超时 */

    else
     {
     while ((volatile)pInputMbufs == NULL)
         (*pWdbDrvIf->pollRtn) (pWdbDrvIf->devId);
     }

    /* 判断是否有数据 */

    if (pInputMbufs == NULL)
     {
     return (0);
     }

    /* 如果有数据到来，将分组拆分为小单元 */

    pIpHdr  = mtod (pInputMbufs, struct ip *);
    pUdpHdr = (struct udphdr *)((int)pIpHdr + IP_HDR_SIZE);

    /* 如果这个分组的目的地不是 WDB 代理，则舍弃 */

    if ((pIpHdr->ip_p      != IPPROTO_UDP) ||
     (pUdpHdr->uh_dport != htons(WDBPORT)) ||
     (pInputMbufs->m_len < UDP_IP_HDR_SIZE))
     {
     nBytes = 0;
        goto done;
     }

    /* 保存答复地址信息 */

    agentIp         = pIpHdr->ip_dst.s_addr;
    pAddr->sin_port  = pUdpHdr->uh_sport;
    pAddr->sin_addr  = pIpHdr->ip_src;

    /* 将 RPC 数据拷贝到缓冲区 */

    pInputMbufs->m_len -= UDP_IP_HDR_SIZE;
    pInputMbufs->m_data += UDP_IP_HDR_SIZE;
    wdbMbufDataGet (pInputMbufs, addr, len, &nBytes);

done:
    /* 释放 mbuf 链 */
```

```
    wdbMbufChainFree (pInputMbufs);
    pInputMbufs = NULL;

    return (nBytes);
    }

/*******************************************************/
/* udpCancel －此函数用于取消 udpRcvfrom()函数的 UDP 接收。*/
/*******************************************************/

static int udpCancel
    (
    WDB_COMM_ID commId
    )
    {
    pWdbRtIf->semGive (readSyncSem);

    return (OK);
    }

/*********************************************/
/* udpSendto －此函数用于发送一个答复分组，      */
/* 前提是已经添加了 UDP/IP 头部。           */
/*********************************************/

static int udpSendto
    (
    WDB_COMM_ID commId,          /* WDB 通信标识 */
    caddr_t addr,
    u_int len,
    struct sockaddr_in pAddr            /* socket 地址 */
    )
    {
    static u_short    ip_id;
    struct udphdr *     pUdpHdr;
    struct ip *         pIpHdr;
    STATUS        status;
    struct mbuf *     pDataMbuf;       /* 数据缓冲 */
    struct mbuf *     pHeaderMbuf;     /* 头部缓冲 */

    /* 申请一个 mbuf 对 */

    pDataMbuf = wdbMbufAlloc();
    if (pDataMbuf == NULL)
     return (0);

    pHeaderMbuf = wdbMbufAlloc ();
    if (pHeaderMbuf == NULL)
     {
```

```
 wdbMbufFree (pDataMbuf);
 return (0);
 }

/* 将答复数据放置到 mbuf 中 */

wdbMbufClusterInit (pDataMbuf, addr, len, NULL, 0);

if (commMode == WDB_COMM_MODE_INT)
 {
 pDataMbuf->m_extFreeRtn = pWdbRtIf->semGive;
 pDataMbuf->m_extArg1   = (int)writeSyncSem;
 }

/* 将头部数据放置到另一个 mbuf 中 */
wdbMbufClusterInit (pHeaderMbuf, udpIpHdrBuf, len, NULL, 0);

pHeaderMbuf->m_data +=  40;
pHeaderMbuf->m_len =    UDP_IP_HDR_SIZE;
pHeaderMbuf->m_type =   MT_DATA;
pHeaderMbuf->m_next  =  pDataMbuf;

pIpHdr    = mtod (pHeaderMbuf, struct ip *);
pUdpHdr  = (struct udphdr *)((int)pIpHdr + IP_HDR_SIZE);
/* 填写 IP 头部 */
pIpHdr->ip_v        = IPVERSION;
pIpHdr->ip_hl    = 0x45;
pIpHdr->ip_tos   = 0;
pIpHdr->ip_off   = 0;
pIpHdr->ip_ttl    = 0x20;
pIpHdr->ip_p     = IPPROTO_UDP;
pIpHdr->ip_src.s_addr = agentIp;
pIpHdr->ip_dst.s_addr = pAddr->sin_addr.s_addr;
pIpHdr->ip_len  = htons (UDP_IP_HDR_SIZE + len);
pIpHdr->ip_id   = ip_id++;
pIpHdr->ip_sum  = 0;
pIpHdr->ip_sum  = ~wdbCksum ((char *)pIpHdr, IP_HDR_SIZE);

pUdpHdr->uh_sport    = htons(WDBPORT);
pUdpHdr->uh_dport    = pAddr->sin_port;
pUdpHdr->uh_sum     = 0;
pUdpHdr->uh_ulen     = htons (UDP_HDR_SIZE + len);

/* 通知驱动程序发送数据 */

if (pWdbDrvIf == NULL)
 return (ERROR);
status = (*pWdbDrvIf->pktTxRtn) (pWdbDrvIf->devId, pHeaderMbuf);
if (status == ERROR)
 return (ERROR);
```

```
    /* 在返回之前等待驱动程序完成发送 */

    if (commMode == WDB_COMM_MODE_INT)
     pWdbRtIf->semTake (writeSyncSem, NULL);

    return (UDP_IP_HDR_SIZE + len);
    }

/*******************************************************/
/* udpHookAdd 一此函数用来为 UDP 添加一个 Hook 函数。    */
/*UDP 的 Hook 函数又叫钩联函数，在 UDP 数据报到来时触发。      */
/*******************************************************/

static int udpHookAdd
    (
    WDB_COMM_ID commId,
    void  (*rout)(),
    u_int  arg
    )
    {
    udpHookRtn = rout;
    udpHookArg = arg;

    return (OK);
    }


/******************** wdbUdpLib.c 程序的结尾*****************************/

/******************** wdbUdpSockLib.c 程序的开始 ******************/
/*说明： wdbUdpSockLib.c 程序是 WDB 通信接口中的 UDP 套接字部分。*/

/*包含头文件*/

#include "vxWorks.h"
#include "stdio.h"
#include "net/protosw.h"
#include "sys/socket.h"
#include "net/socketvar.h"
#include "net/route.h"
#include "net/if.h"
#include "netinet/in_systm.h"
#include "netinet/in.h"
#include "netinet/ip.h"
#include "netinet/udp.h"
#include "arpa/inet.h"
#include "sockLib.h"
#include "string.h"
#include "selectLib.h"
```

```
#include "ioLib.h"
#include "pipeDrv.h"
#include "netLib.h"
#include "intLib.h"
#include "iv.h"
#include "wdb/wdb.h"
#include "wdb/wdbSvcLib.h"
#include "wdb/wdbCommIfLib.h"
#include "wdb/wdbLibP.h"

/* 外部函数声明 */

extern void rtalloc();
extern void rtfree();

/* 静态全局变量 */

static int   wdbUdpSock;            /* WDB 代理读取的套接字 */
static int   wdbUdpCancelSock;  /* 取消读取 wdbUdpSock 的标志 */

/* 内部函数的声明*/

static int  wdbUdpSockRcvfrom  (WDB_COMM_ID commId, caddr_t addr, uint_t len,
                    struct sockaddr_in *pAddr, struct timeval *tv);
static int  wdbUdpSockSendto   (WDB_COMM_ID commId, caddr_t addr, uint_t len,
                    struct sockaddr_in *pAddr);
static int  wdbUdpSockModeSet  (WDB_COMM_ID commId, uint_t newMode);
static int  wdbUdpSockCancel    (WDB_COMM_ID commId);

/*************************************************************/
/* wdbUdpSockIfInit 一此函数初始化 WDB 通信函数指针。        */
/*************************************************************/

STATUS wdbUdpSockIfInit
    (
    WDB_COMM_IF *   pWdbCommIf
    )
    {
    struct sockaddr_in srvAddr;

    pWdbCommIf->rcvfrom   = wdbUdpSockRcvfrom;  /* 接收接口 */
    pWdbCommIf->sendto = wdbUdpSockSendto;       /* 发送接口 */
    pWdbCommIf->modeSet = wdbUdpSockModeSet;    /* 模式设置接口 */
    pWdbCommIf->cancel  = wdbUdpSockCancel;
    pWdbCommIf->hookAdd = NULL;
    pWdbCommIf->notifyHost = NULL;

    /*创建套接字 wdbUdpSocket */

    if ((wdbUdpSock = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
```

```
        {
        printErr ("Socket failed\n");
        return (ERROR);
        }

    /* 将创建的套接字绑定到 WDB 代理的端口号上 */

    bzero ((char *)&srvAddr, sizeof(srvAddr));
    srvAddr.sin_family      = AF_INET;
    srvAddr.sin_port        = htons(WDBPORT);
    srvAddr.sin_addr.s_addr = htonl(INADDR_ANY);
    if (bind (wdbUdpSock, (struct sockaddr *)&srvAddr, sizeof(srvAddr)) < 0 )
        {
        printErr ("Bind failed\n");
        return (ERROR);
        }

    /* 创建套接字 wdbUdpCancelSocket */

    if ((wdbUdpCancelSock = socket (AF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
        {
        printErr ("Socket failed\n");
        return (ERROR);
     }

    /* 与对端套接字口连接，并写入数据到 wdbUdpSocket 。*/

    bzero ((char *)&srvAddr, sizeof(srvAddr));
    srvAddr.sin_family      = AF_INET;
    srvAddr.sin_port        = htons(WDBPORT);
    srvAddr.sin_addr.s_addr = inet_addr ("127.0.0.1");

    if (connect (wdbUdpCancelSock, (struct sockaddr *)&srvAddr, sizeof(srvAddr))
                == ERROR)
     {
        printErr ("Connect failed\n");
     return (ERROR);
     }

    return (OK);
    }


/*************************************************************/
/* wdbUdpSockModeSet－此函数用于设置通信模式。这里只支持中断模式。      */
/*************************************************************/

static int wdbUdpSockModeSet
    (
    WDB_COMM_ID commId,
```

```
    uint_t newMode
    )
    {
    if (newMode != WDB_COMM_MODE_INT)
     return (ERROR);

    return (OK);
    }

/*******************************************************/
/* wdbUdpSockRcvfrom 一此函数用于从 UDP 套接字读取数据。    */
/* 此函数在从 UDP 套接字读取数据时使用了超时设置。          */
/*******************************************************/

static int wdbUdpSockRcvfrom
    (
    WDB_COMM_ID commId,
    caddr_t    addr,
    uint_t len,
    struct sockaddr_in *pSockAddr,
    struct timeval *tv
    )
    {
    uint_t     nBytes;
    int        addrLen = sizeof (struct sockaddr_in);
    struct fd_setreadFds;
    static struct route  theRoute;
    struct sockaddr_in      sockAddr;

    /* 以超时方式等待数据*/

    FD_ZERO (&readFds);
    FD_SET  (wdbUdpSock, &readFds);

    if (select (wdbUdpSock + 1, &readFds, NULL, NULL, tv) < 0)
        {
        printErr ("wdbUdpSockLib: select failed!\n");
        return (0);
        }

    if (!FD_ISSET (wdbUdpSock, &readFds))
        {
        return (0);             /* 超时前没有等到数据 */
        }

    /* 超时前等到了数据，下面读取数据 */

    nBytes   = recvfrom (wdbUdpSock, addr, len, 0,
              (struct sockaddr  *)pSockAddr, &addrLen);
```

```
    if (nBytes < 4)
     return (0);

/* 判断是否需要调整全局变量 wdbCommMtu 的大小 */

    if (theRoute.ro_rt == NULL)
     {
     sockAddr = *pSockAddr;
     sockAddr.sin_port = 0;
     theRoute.ro_dst = * (struct sockaddr *) &sockAddr;
     rtalloc (&theRoute);
     if (theRoute.ro_rt != NULL)
         {
         if (wdbCommMtu > theRoute.ro_rt->rt_ifp->if_mtu)
          wdbCommMtu = theRoute.ro_rt->rt_ifp->if_mtu;
         rtfree (theRoute.ro_rt);
         }
     }

    return (nBytes);
    }

/*******************************************************/
/* wdbUdpSockCancel －此函数用于取消函数 wdbUdpSockRcvfrom()      */
/* 针对套接字的数据接收。                                 */
/*******************************************************/

static int wdbUdpSockCancel
    (
    WDB_COMM_ID commId
    )
    {
    char      dummy;

    netJobAdd (write, wdbUdpCancelSock, (int)&dummy, 1, 0, 0);

    return (OK);
    }

/*****************************************************/
/* wdbUdpSockSendto－此函数用于将数据写到 UDP 套接字中。    */
/*****************************************************/

static int wdbUdpSockSendto
    (
    WDB_COMM_ID commId,
    caddr_t addr,
    uint_t len,
    struct sockaddr_in * pRemoteAddr   /* 对编 socket 地址 */
    )
```

```
    {
    uint_t     nBytes;

    nBytes = sendto (wdbUdpSock, addr, len, 0, (struct sockaddr *)pRemoteAddr,
              sizeof (*pRemoteAddr));

    return (nBytes);        /* 返回发送的字节数 */
    }

/******************** wdbUdpSockLib.c 程序的结尾 ************************/
```

7.12 任务软调度实例一

VxWorks 系统提供了基于任务调度的灵活机制。但在许多具体的应用场合，不可能无限制的创建任务来处理复杂的应用，因为任务的数量过多，将造成操作系统的上下文切换过于频繁，这样实际上降低了系统 CPU 的利用率。本节我们将给出一个在单个任务中实现多事务处理的实例。在任务的基础上我们将引入“进程”（Process）的概念，利用进程完成任务中的单元事务。即在 VxWorks 系统调度任务执行的基础上，任务将处理器资源直接分配给进程来享用，在任务内实现“软调度”。

本实例的核心思想是让 VxWorks 的任务挂载在内核创建的消息队列上，当发现有消息时立即转入任务内的进程调用，查询进程消息队列内是否有消息存在。为此我们在系统创建 3 个任务和 4 个进程。第一个任务是 sendMSGTask，它用于为应用系统中其他任务和进程发送消息。第二个和第三个任务分别是 SCHETASK1 和 SCHETASK2，创建此两任务的目的是利用这两个任务调度内部进程运作。挂接在 SCHETASK1 上的进程是 Process1 和 Process2，而挂接在 SCHETASK2 上的进程是 Process3 和 Process4。

设计本实例的目的是为了验证挂接在任务下的进程相互间可以公平共享处理器资源。因此我们将调度消息的频率打印在终端上，如果各个进程打印在终端（如 VxWorks Simulator for Windows 输出窗口）上的调度频率数据大小相近，那么可以判定进程之间实现了公平调度。

本实例的运行流程图如图 7-8 所示。

本实例运行以后，可以在 VxWorks Simulator for Windows 输出窗口上看到如下打印结果：

```
Process1MSG number is 511
Process2MSG number is 1022
Process3MSG number is 511
Process4MSG number is 1022
……
Process1MSG number is 47012
Process2MSG number is 47523
Process3MSG number is 47012
```

```
Process4MSG number is 47523
```

由以上输出结果，可以判定进程之间随着被调度次数的增多，其消息调度次数越来越相近，这样可以认为任务间进程的调度是公平的。

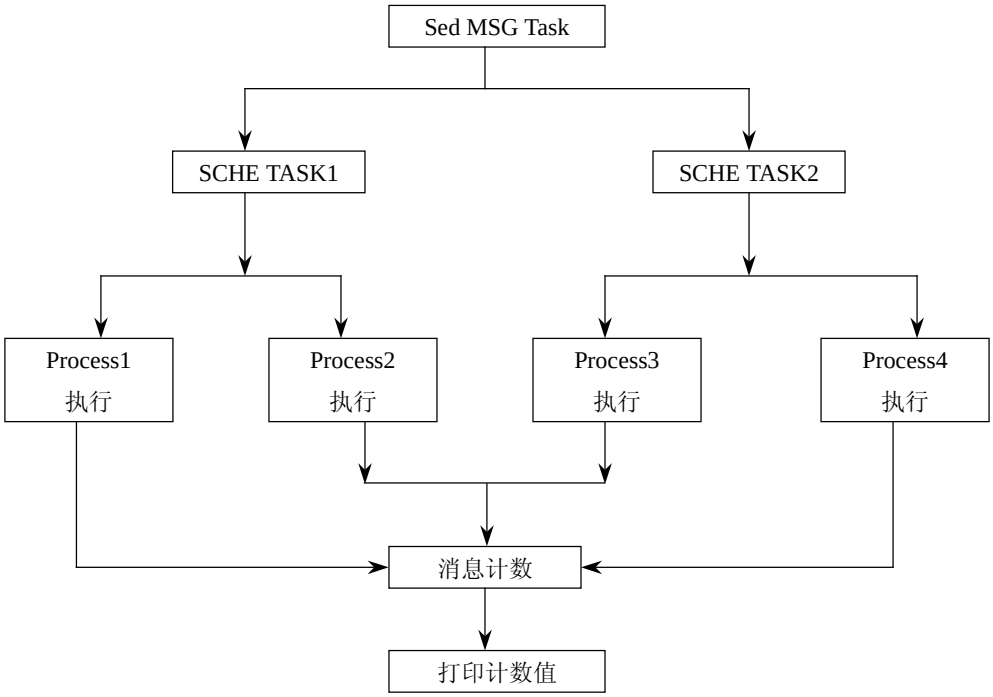

图 7-8　进程执行流程

本节介绍的任务软调度是以 VxWorks 系统的消息队列为基础的，虽然可以实现任务内和任务间各个进程的调度平衡，但是它们的调度效率比较低，原因在于我们调用操作系统中消息队列的系统调用过于频繁。为此，下一节我们将介绍一个效率更高的平衡调度实例，实现更高效的任务软调度。

本实例只有一个源程序 Softsche1.c。下面列出 Softsche1.c 的代码并进行详尽说明。

```
/****************** Softsche1.c 程序的开始*****************/
/* 说明：此应用程序用于演示任务软调度的第一种方案。*/

    /*包含 VxWorks 头文件*/

    #include "vxworks.h"
    #include "msgQLib.h"
   #include "taskLib.h"
    #include "timers.h"
   #include "string.h"
    #include "stdio.h"
    #include "stdlib.h"
    #include "math.h"
```

```
/*函数定义*/
void sendMSG_Task(void);    /* 发送消息任务 */
void CreateTask(void);
void ScheduleTask(int taskId);  /* 调度任务 */
/*各进程入口*/
void Process1Entry(void);
void Process2Entry(void);
void Process3Entry(void);
void Process4Entry(void);

/*宏定义*/
#define MAXMSG_NUM    0x01FF
#define MAXMSG_LENGTH 0x0020
#define MOD_CONSTANT  0x010

#define Process1IdMSG 0x001
#define Process2IdMSG 0x010
#define Process3IdMSG 0x011
#define Process4IdMSG 0x1000

/*最大随机值的定义*/
#ifdef RAND_MAX
#undef RAND_MAX
#define RAND_MAX 128
#endif

int MaxMsgBytes;  /* 消息最大字节数 */
int timeout;        /* 等待消息的时间 */
int maxMsgs;      /* 消息队列中的最大消息数 */

/*这些缓冲区用于接收和发送消息*/
char MSGbuffer[MAXMSG_LENGTH];
char MSGbufferA[MAXMSG_LENGTH];
char MSGbufferB[MAXMSG_LENGTH];
 char MSGbuffer1[MAXMSG_LENGTH];
 char MSGbuffer2[MAXMSG_LENGTH];
 char MSGbuffer3[MAXMSG_LENGTH];
 char MSGbuffer4[MAXMSG_LENGTH];

/*消息的调度次数 */
unsigned long int Process1Msg;
unsigned long int Process2Msg;
unsigned long int Process3Msg;
unsigned long int Process4Msg;

/*任务和进程使用的消息队列*/
MSG_Q_ID ScheduleTask1MSGQId;
MSG_Q_ID ScheduleTask2MSGQId;
MSG_Q_ID ScheduleProcess1MSGQId;
```

```
    MSG_Q_ID ScheduleProcess2MSGQId;
    MSG_Q_ID ScheduleProcess3MSGQId;
    MSG_Q_ID ScheduleProcess4MSGQId;

/***********************************************/
/*sendMSG_Task－此函数用于发送消息。       */
/*此函数的功能是周期性地将消息递送到任务或者进程等待 */
/*的消息队列中。                          */
/***********************************************/

    void sendMSG_Task(void)
    {
    int maxMsgLength; /* 消息中允许地最大字节数 */
    int options;       /* 消息队列地选项 */

    long loop_num;
    int randnum;

    maxMsgs=64;
    maxMsgLength=1024;
   options=MSG_Q_FIFO;

   /*任务 SCHETASK1 使用的消息队列*/
    ScheduleTask1MSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
       if(ScheduleTask1MSGQId==NULL)
       {
         printf("ERROR occurs in task1 msgQCreate!");
            return;
       }

   /*任务 SCHETASK2 使用的消息队列*/
    ScheduleTask2MSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
       if(ScheduleTask2MSGQId==NULL)
       {
         printf("ERROR occurs in task2 msgQCreate!");
            return;
       }

 /*进程 PROCESS 1 使用的消息队列*/*/
   ScheduleProcess1MSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
       if(ScheduleProcess1MSGQId==NULL)
       {
         printf("ERROR occurs in Process1 msgQCreate!");
            return;
       }

/*进程 PROCESS 2 使用的消息队列*/
    ScheduleProcess2MSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
       if(ScheduleProcess2MSGQId==NULL)
       {
```

```
        printf("ERROR occurs in Process2 msgQCreate!");
           return;
      }

   /*进程 PROCESS 3 使用的消息队列*/*/
  ScheduleProcess3MSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
      if(ScheduleProcess3MSGQId==NULL)
      {
        printf("ERROR occurs in Process3 msgQCreate!");
           return;
      }

   /*进程 PROCESS 4 使用的消息队列*/*/
  ScheduleProcess4MSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
      if(ScheduleProcess4MSGQId==NULL)
      {
        printf("ERROR occurs in Process4 msgQCreate!");
           return;
      }

 for (loop_num=0;;loop_num++)     /*消息发送循环*/
  {
    if(loop_num == 0x2f5e34c6)
     {
        randnum = rand();
        randnum = fmod(randnum,MOD_CONSTANT);   /* 获取随机数 */
        if(randnum == 0)
        {
           randnum = MOD_CONSTANT;
        }
        strcpy(MSGbuffer,"randnum");
       msgQSend(ScheduleTask1MSGQId,      /* 发送随机数 */
                    MSGbuffer,
                    sizeof(MSGbuffer),
                    timeout, /* 等待的 tick 数 */
                MSG_PRI_NORMAL
                 );
        randnum++;

        randnum = fmod(randnum,MOD_CONSTANT);  /* 获取随机数 */
        if(randnum == 0)
        {
           randnum = MOD_CONSTANT;
        }
        strcpy(MSGbuffer,"randnum");
       msgQSend(ScheduleTask2MSGQId,    /* 发送随机数 */
                       MSGbuffer,
                    sizeof(MSGbuffer),
              timeout, /* 等待的 tick 数 */
                        MSG_PRI_NORMAL
```

```
                );
 randnum++;

 randnum = fmod(randnum,MOD_CONSTANT);     /* 获取随机数 */
  if(randnum == 0)
  {
     randnum = MOD_CONSTANT;
  }
  strcpy(MSGbuffer,"randnum");
msgQSend(ScheduleProcess1MSGQId,     /* 发送随机数 */
                 MSGbuffer,
              sizeof(MSGbuffer),
        timeout, /* 等待的 tick 数 */
                 MSG_PRI_NORMAL
               );
 randnum++;

 randnum = fmod(randnum,MOD_CONSTANT);    /* 获取随机数 */
  if(randnum == 0)
  {
     randnum = MOD_CONSTANT;
  }
  strcpy(MSGbuffer,"randnum");
msgQSend(ScheduleProcess2MSGQId,        /* 发送随机数 */
                 MSGbuffer,
              sizeof(MSGbuffer),
        timeout, /* 等待的 tick 数 */
                  MSG_PRI_NORMAL
               );
 randnum++;

 randnum = fmod(randnum,MOD_CONSTANT);  /* 获取随机数 */
  if(randnum == 0)
  {
     randnum = MOD_CONSTANT;
  }
  strcpy(MSGbuffer,"randnum");
msgQSend(ScheduleProcess3MSGQId,     /* 发送随机数 */
              MSGbuffer,
           sizeof(MSGbuffer),
     timeout, /* 等待的 tick 数 */
               MSG_PRI_NORMAL
                );
 randnum++;

 randnum = fmod(randnum,MOD_CONSTANT);     /* 获取随机数 */
  if(randnum == 0)
  {
     randnum = MOD_CONSTANT;
  }
```

```
            strcpy(MSGbuffer,"randnum");
           msgQSend(ScheduleProcess4MSGQId,     /* 发送随机数 */
                       MSGbuffer,
                       sizeof(MSGbuffer),
                        timeout, /* 等待的 tick 数 */
                           MSG_PRI_NORMAL
                        );

           loop_num=0;
          }
       }
       printf("A ERROR occurs in Task Sending MSG");
       }

/************************************************/
/*CreateTask 一此函数是本应用程序的主函数。      */
/*此函数创建本应用程序的所有任务包括 sendMSG_Task、*/
/*ScheTask1 和 ScheTask2。                       */
/************************************************/

       void  CreateTask(void)
       {
       int RetValue;

       timeout = 100;
       MaxMsgBytes = MAXMSG_LENGTH;   /* 消息的最大长数 */
       /* 初始化消息计数器 */
       Process1Msg = 0;
       Process2Msg = 0;
       Process3Msg = 0;
       Process4Msg = 0;

      RetValue=taskSpawn("sdMSGTask",     /* 发送消息的任务的初始化 */
                  100,
                  VX_PRIVATE_ENV,
                  512,
                  (FUNCPTR)sendMSG_Task,
                  0,0,0,0,0,0,0,0,0,0);
       if (RetValue==ERROR)
                printf("ERROR occurs in sendMSG Task Creating");

       RetValue=taskSpawn("SCHETask1",    /* 调务任务 1 的初始化 */
                          100,
                          VX_PRIVATE_ENV,
                          512,
                          (FUNCPTR)ScheduleTask,
                          1,0,0,0,0,0,0,0,0,0);/*第一个参数是任务参数*/
      if (RetValue==ERROR)
                printf("ERROR occurs in SCHETask1 Creating");
```

```
    RetValue=taskSpawn("SCHETask2",     /* 调务任务 2 的初始化 */
                       100,
                       VX_PRIVATE_ENV,
                       512,
                       (FUNCPTR)ScheduleTask,
                       2,0,0,0,0,0,0,0,0,0); /*第一个参数是任务参数*/
    if (RetValue==ERROR)
              printf("ERROR occurs in SCHETask2 Creating");
    }

/*****************************************************/
    /*ScheduleTask一本函数实现了调度任务功能。        */
/*此函数实现的调度任务直接调度挂接在它下面的进程执行。     */
    /*调度任务的原则是实现进程间的公平调度。          */
    /*****************************************************/

    void ScheduleTask(int taskId)
    {
    int tId;
    int revBytes;

    tId = taskId;

/*如果任务 Id 为 1，只调度 process1 和 process2 执行。*/

    if (tId == 1)
    {
         for (;;)
          {
          revBytes=msgQReceive(ScheduleTask1MSGQId,  /* 读取消息队列 */
                               MSGbufferA,
                               MaxMsgBytes,
                               timeout);
               if(0!=revBytes)

               {
                    revBytes=msgQReceive(ScheduleProcess1MSGQId,
                                         MSGbuffer1,
                                         MaxMsgBytes,
                                         timeout);
               if (0 != revBytes)
               {
                   Process1Entry();   /* 运行进程 1 */
               }
                revBytes=msgQReceive(ScheduleProcess2MSGQId,
                                         MSGbuffer2,
                                         MaxMsgBytes,
                                         timeout);
               if (0 != revBytes)
               {
```

```
                    Process2Entry();    /*运行进程 2 */
                }
            }
                printf("ERROR occurs in SCHETask1 Scheduling MSG");

            }
        }

    /*如果任务 Id 为 2，只调度 process3 和 process4 执行*/

        else if(tId == 2)
        {
            for (;;)
            {
            revBytes=msgQReceive(ScheduleTask2MSGQId,
                                 MSGbufferB,
                                 MaxMsgBytes,
                                 timeout);
                if(0!=revBytes)

                {
                    revBytes=msgQReceive(ScheduleProcess3MSGQId,
                                         MSGbuffer3,
                                         MaxMsgBytes,
                                         timeout);
                if (0 != revBytes)
                {
                    Process3Entry();    /* 运行进程 3 */
                }
                revBytes=msgQReceive(ScheduleProcess4MSGQId,
                                         MSGbuffer4,
                                         MaxMsgBytes,
                                         timeout);
                if (0 != revBytes)
                {
                    Process4Entry();   /* 运行进程 4 */
                }
            }
                printf("ERROR occurs in SCHETask2 Scheduling MSG");
            }
        }

        else
        {
            /*没有其他调度任务*/
        }

        }

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

```
/*Process1Entry－此函数是进程 Process1 的入口。      */
/* 此进程用于计算消息的调度频率。                    */
/************************************************/

     void Process1Entry(void)
     {
       int revBytes;
       int loop;
       int msgnum;
       int old_process1Msg;

       old_process1Msg = Process1Msg;

      /*获取消息队列中的消息数*/

       msgnum = msgQNumMsgs(ScheduleProcess1MSGQId);
       if(msgnum == MAXMSG_NUM)
       {
          printf("Process1MSGQ excess its Limit" );
          return;
       }

       /*遍历消息队列计算消息的调度次数*/

       for(loop = 0;loop <msgnum ;loop ++)
       {
           revBytes=msgQReceive(ScheduleProcess1MSGQId,
                                MSGbuffer1,
                                MaxMsgBytes,
                                timeout);

       /*增加相应的计数器*/

           if ( strcmp(MSGbuffer1,"1") )
           {
                 Process1Msg++;   /* 增加进程 1 的消息计数器 */
           }
           if ( strcmp(MSGbuffer1,"2") )
            {
                  Process2Msg++;   /* 增加进程 2 的消息计数器 */
            }
            if ( strcmp(MSGbuffer1,"3") )
            {
                  Process3Msg++;   /* 增加进程 3 的消息计数器 */
            }
           if ( strcmp(MSGbuffer1,"4") )
            {
                  Process4Msg++;   /* 增加进程 4 的消息计数器 */
            }
```

```
        }

        /*如果消息调度次数有所增加，则打印到终端*/

        if(old_process1Msg != Process1Msg)
        {
           printf("Process1MSG number is %ld\n",Process1Msg);
        }

       return;
       }


/************************************************/
/*Process2Entry－此函数是进程 Process2 的入口。          */
/* 此进程用于计算消息的调度频率。                        */
/************************************************/

     void Process2Entry(void)
     {
     int revBytes;
     int loop;
     int msgnum;
     int old_process2Msg;

     old_process2Msg = Process2Msg;

      /*获取消息队列中的消息数*/

     msgnum = msgQNumMsgs(ScheduleProcess2MSGQId);
     if(msgnum == MAXMSG_NUM)
     {
     printf("Process2MSGQ excess its Limit" );
     return;
     }

     /*遍历消息队列计算消息的调度次数*/

     for(loop = 0;loop <msgnum ;loop ++)
     {
        revBytes=msgQReceive(ScheduleProcess2MSGQId,
                             MSGbuffer2,
                             MaxMsgBytes,
                             timeout);

  /*增加相应的计数器*/

          if ( strcmp(MSGbuffer2,"1") )
          {
             Process1Msg++;
```

```
            }
            if ( strcmp(MSGbuffer2,"2") )
            {
               Process2Msg++;
            }
            if ( strcmp(MSGbuffer2,"3") )
            {
               Process3Msg++;
            }
            if ( strcmp(MSGbuffer2,"4") )
            {
               Process4Msg++;
            }

        }

        /*如果消息调度次数有所增加，则打印到终端*/

        if(old_process2Msg != Process2Msg)
        {
             printf("Process2MSG number is %ld\n",Process2Msg);
        }

        return;
        }

/************************************************/
/*Process3Entry－此函数是进程 Process3 的入口。          */
/* 此进程用于计算消息的调度频率。                      */
/************************************************/

      void Process3Entry(void)
      {
        int revBytes;
        int loop;
        int msgnum;
        int old_process3Msg;

        old_process3Msg = Process3Msg;

       /*获取消息队列中的消息数*/

        msgnum = msgQNumMsgs(ScheduleProcess3MSGQId);
        if(msgnum == MAXMSG_NUM)
        {
          printf("Process3MSGQ excess its Limit" );
          return;
        }

        /*遍历消息队列计算消息的调度次数*/
```

```
        for(loop = 0;loop <msgnum ;loop ++)
        {
            revBytes=msgQReceive(ScheduleProcess3MSGQId,
                                 MSGbuffer3,
                                 MaxMsgBytes,
                                 timeout);

        /*增加相应的计数器*/

            if ( strcmp(MSGbuffer3,"1") )
            {
                  Process1Msg++;
            }
           if ( strcmp(MSGbuffer3,"2") )
            {
                  Process2Msg++;
            }
            if ( strcmp(MSGbuffer3,"3") )
            {
                  Process3Msg++;
            }
           if ( strcmp(MSGbuffer3,"4") )
           {
                  Process4Msg++;
           }

      }

/*如果消息调度次数有所增加，则打印到终端*/

        if(old_process3Msg != Process3Msg)
        {
           printf("Process3MSG number is %ld\n",Process3Msg);
        }

       return;
       }


/*********************************************/
/*Process4Entry－此函数是进程 Process4 的入口。    */
/* 此进程用于计算消息的调度频率。                */
/*********************************************/

     void Process4Entry(void)
     {
     int revBytes;
     int loop;
     int msgnum;
```

```
int old_process4Msg;

old_process4Msg = Process4Msg;

 /*获取消息队列中的消息数*/

msgnum = msgQNumMsgs(ScheduleProcess4MSGQId);
if(msgnum == MAXMSG_NUM)
{
printf("Process4MSGQ excess its Limit" );
return;
}

/*遍历消息队列计算消息的调度次数*/

for(loop = 0;loop <msgnum ;loop ++)
{
   revBytes=msgQReceive(ScheduleProcess4MSGQId,
                        MSGbuffer4,
                        MaxMsgBytes,

                        timeout);

/*增加相应的计数器*/

     if ( strcmp(MSGbuffer4,"1") )
     {
        Process1Msg++;
     }
     if ( strcmp(MSGbuffer4,"2") )
     {
        Process2Msg++;
     }
     if ( strcmp(MSGbuffer4,"3") )
     {
        Process3Msg++;
     }
     if ( strcmp(MSGbuffer4,"4") )
     {
        Process4Msg++;
     }

}

/*如果消息调度次数有所增加，则打印到终端*/

if(old_process4Msg != Process4Msg)
{
printf("Process4MSG number is %ld\n",Process4Msg);
}
```

```
    return;
    }

/***************** Softschel.c 程序的结尾*****************/
```

7.13 任务软调度实例二

上一节介绍了一个基本的任务软调度的实例，同时指出了实例中的缺陷。本节的意图是介绍一个更高效的任务软调度的实例。实际上，两个实例各有优缺点。上节介绍的实例虽然在时间上有一定的滞后，但是它可以完成进程的公平调度，而且使用 VxWorks 系统的消息队列让程序的实现变得异常简单。本节介绍的任务软调度实例既可以实现进程的公平调度，又具有较高的时间特性。但是本节介绍的进程不使用 VxWorks 系统提供的消息队列。因此要求调度任务自己完成各个进程的消息分配，这样就需要自己编写和控制进程的消息队列，从而增加了编程的难度。

本实例的核心思想是让 VxWorks 的任务挂载在内核创建的消息队列上，当发现有消息时立即将收到的消息转发给任务内的进程，同时不断查询进程消息队列内是否有消息存在以便及时处理。同上一节实例，我们在系统也创建 3 个任务和 4 个进程。第一个任务是 sendMSGTask，它用于为应用系统中其他任务和进程发送消息。第二个和第三个任务分别是 SCHETASK1 和 SCHETASK2，创建此两任务的目的是利用这两个任务调度内部进程运作。系统三个任务具有相同的优先级。挂接在 SCHETASK1 上的进程是 Process1 和 Process2，而挂接在 SCHETASK2 上的进程是 Process3 和 Process4。

我们设计本实例的目的是为了验证挂接在任务下的进程相互间可以公平共享处理器资源并且具有较高的效率。我们将进度调度的频率打印在终端上，如果各个进程打印在终端（如 VxWorks Simulator for Windows 输出窗口）上的调度频率大小相近，那么可以判定进程之间实现了公平调度。同时我们可以在根据输出窗口的打印频率判定其时间特性。

本实例运行以后，可以在 VxWorks Simulator for Windows 输出窗口上看到如下打印结果：

```
Process1 is scheduled 10 times
Process2 is scheduled 10 times
Process3 is scheduled 10 times
Process4 is scheduled 10 times
Process1 is scheduled 20 times
Process2 is scheduled 20 times
Process3 is scheduled 20 times
Process4 is scheduled 20 times
……
Process1 is scheduled 8000 times
Process2 is scheduled 8000 times
Process3 is scheduled 8000 times
```

```
Process4 is scheduled 8000 times
```

由以上输出结果，我们可以判定进程之间的被调度次数基本上保持持平。不过，其中的调度次数偶尔有一定漂移,但是幅度不大于 10 次,这样可以认为任务间进程的调度是公平的。而且通过对打印频率的判断得知其时间特性优于上一节的实例。

本节介绍的任务软调度不以 VxWorks 系统的消息队列为基础，通过自己创建的简单消息队列完成消息的分配和管理，因而既可以实现任务内和任务间各个进程的调度平衡，又具有较高的调度效率。

本实例只有一个源程序 Softsche2.c。下面列出 Softsche2.c 的代码并进行了详尽的说明。

```
/****************** Softsche2.c 程序的开始*****************/
/* 说明：此应用程序用于演示任务软调度的第二种方案。*/

/*********************************************************/
/* 在此方案中，进程使用自定义的消息队列替换方案一中的  */
/*  VxWorks 消息队列。                                   */
/*********************************************************/

     /*包含 VxWorks 头文件*/

     #include "vxworks.h"
     #include "msgQLib.h"
    #include "taskLib.h"
     #include "timers.h"
    #include "string.h"
     #include "stdio.h"
     #include "stdlib.h"
     #include "math.h"

     /*函数声明*/
     /*所有任务入口的声明*/

     void sendMSG_Task(void);
     void CreateTask(void);
     void ScheduleTask(int taskId);

     /*所有进程入口的声明*/

     void Process1Entry(void);
     void Process2Entry(void);
     void Process3Entry(void);
     void Process4Entry(void);

/*特定应用函数的声明*/

     int  DataModel(int data1);
     void PostMessageOnProcess(int processNo,int messageNo);
```

```
int  MessageValid(int revMsg,int processNo);
void PrintfStatistic(int Process,int ProcessId);
int  GetMessage(int processNo);

/*宏定义*/

#define MAXMSG_NUM  0x04FF
#define MAXMSG_LENGTH  64
#define PRINT_FREQ      10

#ifndef SUCCESS
#define SUCCESS        1
#endif
#define FALSE       0

#ifdef RAND_MAX
#undef RAND_MAX
#define RAND_MAX      256
#endif
#define MOD_SEED       128
#define FIRST_TASK      1
#define SECOND_TASK   2
#define PROCESSNUM    4

/*消息结构的定义*/
typedef struct Tag_Message
{

/*****************************************/
/*messageNO(1-64)发送给 process 1          */
/*messageNO(65-128) 发送给 process 2       */
/*messageNO(129-192) 发送给 process 3      */
/*messageNO(193-256) 发送给 process 4      */
/*****************************************/

 char messageNO;
 char messagePara1;
 char messagePara2;
 char messagePara3;
}T_Message;

/*下面是发送给进程的消息单元的定义*/

typedef struct Tag_MessageElement
{
 char messageNO;
 char pPara;
 int prevMessage;
 int nextMessage;
 int IsUsed;
```

```
      int prevIdle; /*闲态队列中的上一元素*/
      int nextIdle; /*闲态队列中的下一元素*/
     }T_MessageElement;

    /*******************************************************/
     /*数据结构 T_PCD 用于保存本应用所创建进程的控制管理数据*/
/*实际上，它的大部分内容是管理消息队列的字段。          */
    /*******************************************************/

     typedef struct Tag_PCD
     {

/*管理有效消息队列的字段*/

     int messageSum;
     int firstMessage;
     int lastMessage;

/*管理闲态消息队列（或闲态队列）的字段*

     int idleSum;
     int firstIdle;
     int lastIdle;
     T_MessageElement messageArray[MAXMSG_NUM];
     }T_PCD;

     T_PCD ProcessPCDArray[PROCESSNUM];

int maxMsgLength;    /* 消息最大字节数 */
     int timeout;          /* 等待消息的时间 */
     int maxMsgs;        /* 消息队列中的最大消息数 */

     /*用于接收和发送消息的缓冲区*/

     char MSGbuffer1[4];
     char MSGbuffer2[4];
     char MSGbuffer3[4];
     char MSGbuffer4[4];

   /*进程调度计数器*/

     unsigned  int Process1Id;
     unsigned  int Process2Id;
     unsigned  int Process3Id;
     unsigned  int Process4Id;


   /*VXWorks 消息队列只用于调度任务*/

     MSG_Q_ID ScheduleTask1MSGQId;
```

```
    MSG_Q_ID ScheduleTask2MSGQId;


/*******************************************************/
/*sendMSG_Task－此函数用于发送消息。           */
/*此函数的功能是周期性地将消息递送到任务等待的消息队列中。     */
/*******************************************************/

    void sendMSG_Task(void)
    {

    int options;       /* 消息队列选项 */
   long loop_num;
    int randnum;
   char MSGbuffer[4];
   T_Message sendMessage;

    sendMessage.messageNO = 0;
    sendMessage.messagePara1 = 0;
    sendMessage.messagePara2 = 0;
    sendMessage.messagePara3 = 0;

   options=MSG_Q_FIFO;

/*SCHETASK1 使用的消息队列*/

    ScheduleTask1MSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
       if(ScheduleTask1MSGQId==NULL)
       {
         printf("A ERROR occurs in task1 msgQCreate!");
            return;
       }

/*SCHETASK2 使用的消息队列*/

    ScheduleTask2MSGQId= msgQCreate( maxMsgs,maxMsgLength,options);
       if(ScheduleTask2MSGQId==NULL)
       {
         printf("A ERROR occurs in task2 msgQCreate!");
            return;
       }

  for (loop_num=0;;loop_num++)
   {
     if(loop_num == 0x0F34E078)
     {
          randnum = rand();
          sendMessage.messageNO = (char)DataModel(randnum);

          memcpy(MSGbuffer,&(sendMessage),sizeof(sendMessage));
```

```
            msgQSend(ScheduleTask1MSGQId,
                            MSGbuffer,
                            sizeof(MSGbuffer),
                         timeout, /* 等待的 tick 数 */
                    MSG_PRI_NORMAL
                          );

             randnum++;
             sendMessage.messageNO = (char)DataModel(randnum);

             memcpy(MSGbuffer,&(sendMessage),sizeof(sendMessage));
            msgQSend(ScheduleTask2MSGQId,
                        MSGbuffer,
                        sizeof(MSGbuffer),
                    timeout, /* 等待的 tick 数 */
                    MSG_PRI_NORMAL
                      );

             loop_num = 0;
           }
        }
        printf("A ERROR occurs in the Task sending MSG \n");
        }

/***********************************************/
/*CreateTask 一此函数是本应用程序的主函数。          */
/*此函数创建本应用程序的所有任务包括 sendMSG_Task、      */
/*ScheTask1 和 ScheTask2。                      */
/***********************************************/

        void  CreateTask(void)
        {
        int RetValue;
        int loop_i1;
        int loop_i2;

        /*为各进程初始化调度计数器*/

        Process1Id = 0;
        Process2Id = 0;
        Process3Id = 0;
        Process4Id = 0;

        timeout = 100;
        maxMsgs = MAXMSG_NUM; /*最大消息数目*/
        maxMsgLength = MAXMSG_LENGTH;

        /*初始化进程消息块的闲态队列*/

        for(loop_i1 = 0; loop_i1 < PROCESSNUM; loop_i1++)
```

```
        {
             ProcessPCDArray[loop_i1].idleSum = MAXMSG_NUM;
             ProcessPCDArray[loop_i1].firstIdle = 0;
             ProcessPCDArray[loop_i1].lastIdle = MAXMSG_NUM-1;
             for (loop_i2 = 1;loop_i2 < MAXMSG_NUM-1;loop_i2++)
             {
                ProcessPCDArray[loop_i1].messageArray[loop_i2].prevIdle = loop_i2 - 1;
                ProcessPCDArray[loop_i1].messageArray[loop_i2].nextIdle = loop_i2 + 1;
             }
             ProcessPCDArray[loop_i1].messageArray[0].prevIdle = 0;
             ProcessPCDArray[loop_i1].messageArray[0].nextIdle = 1;

             ProcessPCDArray[loop_i1].messageArray[MAXMSG_NUM-1].prevIdle =
    MAXMSG_NUM-2;
             ProcessPCDArray[loop_i1].messageArray[MAXMSG_NUM-1].nextIdle =
    MAXMSG_NUM-1;

        }

       RetValue=taskSpawn("sendMSGTask",
                     100,
                     VX_PRIVATE_ENV,
                     512,
                      (FUNCPTR)sendMSG_Task,
                     0,0,0,0,0,0,0,0,0,0);
        if (RetValue==ERROR)
    {
             printf("A ERROR occurs in sendMSG Task Creating");
    }

        RetValue=taskSpawn("SCHETask1",
                           100,
                           VX_PRIVATE_ENV,
                           512,
                           (FUNCPTR)ScheduleTask,
                           1,0,0,0,0,0,0,0,0,0);/*the first parameter is taskId*/
        if (RetValue==ERROR)
    {
            printf("A ERROR occurs in SCHETask1 Creating");
    }

        RetValue=taskSpawn("SCHETask2",
                           100,
                           VX_PRIVATE_ENV,
                           512,
                           (FUNCPTR)ScheduleTask,
                           2,0,0,0,0,0,0,0,0,0); /*the first parameter is taskId*/
        if (RetValue==ERROR)
    {
            printf("A ERROR occurs in SCHETask2 Creating");
```

```
}
    }

/*************************************************/
/*ProcessEntry1－此函数是进程 process1 的入口。    */
/*process1 用于消耗有效消息队列中的消息和记录消耗的    */
/*次数，消耗的次数表示了 process 1 的调度信息。此函数以*/
/*10 为频率间隔将调度信息打印到终端。        */
/*************************************************/

    void ProcessEntry1(void)
    {
        int revMsg;
        revMsg = GetMessage(1);

        if (MessageValid(revMsg,1))
        {
         Process1Id++;
        }
        PrintfStatistic(1,Process1Id);
        return;
    }

/*************************************************/
/*ProcessEntry2－此函数是进程 process2 的入口。    */
/*process2 用于消耗有效消息队列中的消息和记录消耗的    */
/*次数，消耗的次数表示了 process 2 的调度信息。此函数以*/
/*10 为频率间隔将调度信息打印到终端。        */
/*************************************************/

    void ProcessEntry2(void)
    {
        int revMsg;
        revMsg = GetMessage(2);

        if (MessageValid(revMsg,2))
        {
         Process2Id++;
        }
        PrintfStatistic(2,Process2Id);
        return;
    }

/*************************************************/
/*ProcessEntry3－此函数是进程 process3 的入口。    */
/*process3 用于消耗有效消息队列中的消息和记录消耗的    */
/*次数，消耗的次数表示了 process 3 的调度信息。此函数以*/
/*10 为频率间隔将调度信息打印到终端。        */
/*************************************************/
```

```
void ProcessEntry3(void)
     {
         int revMsg;
         revMsg = GetMessage(3);

         if (MessageValid(revMsg,3))
         {
          Process3Id++;
         }
         PrintfStatistic(3,Process3Id);
         return;
     }

/************************************************/
/*ProcessEntry4－此函数是进程 process4 的入口。    */
/*process4 用于消耗有效消息队列中的消息和记录消耗的    */
/*次数，消耗的次数表示了 process 4 的调度信息。此函数以*/
/*10 为频率间隔将调度信息打印到终端。        */
/************************************************/

void ProcessEntry4(void)
     {
         int revMsg;
         revMsg = GetMessage(4);

         if (MessageValid(revMsg,4))
         {
          Process4Id++;
         }
         PrintfStatistic(4,Process4Id);
         return;
     }

/******************************************************/
/*ScheduleTask－本函数实现了调度任务功能。            */
/*此函数实现的调度任务直接调度挂接在它下面的进程执行。    */
/*调度任务的原则是实现进程间的公平调度。              */
/******************************************************/

     void ScheduleTask(int taskId)
     {
          char MSGbufferA[4];
          char MSGbufferB[4];
          int timeOut;
          int revBytes;
          int imessageNO;
          int tId;

          timeout = 100;
          tId = taskId;
```

```
            switch(tId)
            {
            case FIRST_TASK:
              {
                 for(;;)
                 {

                      /*如果 Process 1 有消息要处理，调用 processEntry1 */

                      if(ProcessPCDArray[1].messageSum !=0)
                      {
                        ProcessEntry1();
                      }

                    /*如果 Process 2 有消息要处理，调用 processEntry2 */

                      if(ProcessPCDArray[2].messageSum !=0)
                      {
                        ProcessEntry1();
                      }

/*如果无消息要处理，则检查任务的消息队列*/

                     revBytes = msgQReceive(ScheduleTask1MSGQId,
                                            MSGbufferA,
                                            maxMsgLength,
                                            timeOut);
                            imessageNO = (int)MSGbufferA[0];
                            if((imessageNO > 0) &&(imessageNO <= MOD_SEED))
                            {
                                   if((imessageNO > 0)
&&(imessageNO <= MOD_SEED/2))
                                   {

/* 为 Process1 张贴消息 */

                                    PostMessageOnProcess(1,imessageNO);
                                   }
                                   else
                                   {
                                     PostMessageOnProcess(2,imessageNO);
                                   }
                            }
                 }

                 printf("A ERROR occurs in task scheduling");
                 break;
              }
            case SECOND_TASK:
```

```
            {
               for(;;)
               {

                    /*如果 Process 3 有消息要处理，调用 processEntry3 */

                    /if(ProcessPCDArray[3].messageSum !=0)
                    /{
                    /  ProcessEntry3();
                    /}

                    //*如果 Process 4 有消息要处理，调用 processEntry4 */

                    /if(ProcessPCDArray[4].messageSum !=0)
                    /{
                    /  ProcessEntry4();
                    /}

/*如果无消息要处理，则检查任务的消息队列*/

                    /revBytes = msgQReceive(ScheduleTask2MSGQId,
                                        MSGbufferB,
                                        maxMsgLength,
                                        timeOut);
                              imessageNO = (int)MSGbufferB[0];
                              if((imessageNO > MOD_SEED) &&
(imessageNO <= RAND_MAX))
                             {
                                    if((imessageNO > MOD_SEED) &&
(imessageNO <= (MOD_SEED+RAND_MAX)/2))
                                   {
                                    PostMessageOnProcess(3,imessageNO);
                                   }
                                   else
                                   {
                                     PostMessageOnProcess(4,imessageNO);
                                   }
                             }
               }

               printf("A ERROR occurs in task scheduling");
               break;
             }

           default:
            {
               printf("A ERROR occurs in task scheduling");
               break;
            }
           }
```

```
    return;

    }

    /************************************/
    /* DataModel 一此函数用于计算数据的模。      */
/************************************/

    int DataModel(int data1)
    {
       int randnum;

       randnum = data1;
       randnum = fmod(randnum, MOD_SEED);
       if (randnum == 0)
       {
        randnum = MOD_SEED;
       }
       return randnum;
    }

/*****************************************************************/
/* PostMessageOnProcess一此函数为给定的进程张贴一条消息。                */
/* 从闲态队列的头部获取一个消息块，并且将它放置在有效          */
/* 消息队列的尾部。                                  */
    /*****************************************************************/

    void PostMessageOnProcess(int processNo,int messageNo)
    {
       int process;
       int message;
       int messageBlock;
       int last;
       int first;

       process =    processNo;
       message =   messageNo;

       if((ProcessPCDArray[process].messageSum >=MAXMSG_NUM)
||(ProcessPCDArray[process].idleSum <= 0))
       {
             return ;
       }

       messageBlock = ProcessPCDArray[process].firstIdle;

       /*修改 PCD*/

       ProcessPCDArray[process].messageSum++;
       last = ProcessPCDArray[process].lastMessage;
```

```
        ProcessPCDArray[process].lastMessage = messageBlock;

        ProcessPCDArray[process].firstIdle =
            ProcessPCDArray[process].messageArray[messageBlock].nextIdle;
        ProcessPCDArray[process].idleSum--;

        /*修改空闲队列，摘取它的第一个元素。*/

        first = ProcessPCDArray[process].firstIdle;
        ProcessPCDArray[process].messageArray[first].prevIdle = first;
        if(ProcessPCDArray[process].idleSum == 1)
        {
              ProcessPCDArray[process].messageArray[first].nextIdle = first;
        }

        /*修改有效消息队列，在消息队列的尾部添加一元素。*/

        ProcessPCDArray[process].messageArray[messageBlock].prevMessage = last;
        ProcessPCDArray[process].messageArray[messageBlock].nextMessage = messageBlock;
        ProcessPCDArray[process].messageArray[last].nextMessage = messageBlock;

/*设置消息元素的属性。*/

        ProcessPCDArray[process].messageArray[messageBlock].IsUsed = 1;
        ProcessPCDArray[process].messageArray[messageBlock].messageNO = message;

        return;
    }

/*******************************************/
/*GetMessage一此函数用于从给定的进程获得消息，*/
/* 并将其消耗。从有效消息队列的头部获取一个消 */
/* 息块，并且将它放置在闲态队列的尾部。      */
/*******************************************/

    int GetMessage(int processNo)
    {
      int process;
       int message;/*返回消息*/
       int messageBlock;
       int last;
       int first;

       process =    processNo;

       if((ProcessPCDArray[process].messageSum < 0) ||
           (ProcessPCDArray[process].idleSum < 0))
       {
              return FALSE;
       }
```

```
        /*从有效消息队列中获得第一个消息。*/

        messageBlock = ProcessPCDArray[process].firstMessage;

        /*修改 PCD*/

        if(ProcessPCDArray[process].messageSum == 0)
        {
             ProcessPCDArray[process].firstMessage = 0;
        }

        last = ProcessPCDArray[process].lastIdle;
        ProcessPCDArray[process].lastIdle = messageBlock;
        ProcessPCDArray[process].idleSum++;

        ProcessPCDArray[process].firstMessage =
            ProcessPCDArray[process].messageArray[messageBlock].nextMessage;
        ProcessPCDArray[process].messageSum--;

        /*修改空闲队列，在空闲队列的尾部添加一元素。*/

        ProcessPCDArray[process].messageArray[last].nextIdle = messageBlock;
        ProcessPCDArray[process].messageArray[messageBlock].prevIdle = last;
        ProcessPCDArray[process].messageArray[messageBlock].nextIdle = messageBlock;


        /*修改有效消息队列，摘取消息队列的第一个元素。*/

        first = ProcessPCDArray[process].firstMessage;
        ProcessPCDArray[process].messageArray[first].prevMessage = first;
        if(ProcessPCDArray[process].messageSum == 1)
        {
          ProcessPCDArray[process].messageArray[first].nextMessage = first;
        }

/* 设置消息单元的属性。*/

        ProcessPCDArray[process].messageArray[messageBlock].IsUsed = 0;
        message = ProcessPCDArray[process].messageArray[messageBlock].messageNO;

        return message;
    }

    /*************************************************************/
    /*MessageValid一此函数为一个给定的进程进行消息的有效性验证。*/
    /*************************************************************/

    int MessageValid(int revMsg,int processNo)
    {
```

```
            int imessage;
            int iprocess;

            imessage = revMsg;
            iprocess = processNo;

            switch(iprocess)
            {

             /*发送到 process4 的消息。*/

            case 4:
            {
                if ((imessage > (MOD_SEED+RAND_MAX)/2)&&
    (imessage <= RAND_MAX))
               {
                    return SUCCESS;
               }
               else
               {
                    return FALSE;
               }
               break;
            }

    /*发送到 process3 的消息。*/

            case 3:
            {
               if ((imessage > MOD_SEED)&&
    (imessage <= (RAND_MAX+MOD_SEED)/2))
               {
                    return SUCCESS;
               }
               else
               {
                    return FALSE;
               }
               break;
            }

    /*发送到 process2 的消息*/

            case 2:
            {
               if ((imessage > MOD_SEED/2)&&(imessage <= MOD_SEED))
               {
                    return SUCCESS;
               }
               else
```

```
            {
                return FALSE;
            }
            break;
        }

/*发送到 process1 的消息*/

        case 1:
        {
            if ((imessage > 0)&&(imessage <= MOD_SEED/2))
            {
                return SUCCESS;
            }
            else
            {
                return FALSE;
            }
            break;
        }

        default:
        {
            return FALSE;
            break;
        }

       }
    return FALSE;
    }

    /*********************************************/
    /*PrintfStatistic 一此函数用于输出调试信息。          */
    /*********************************************/

    void PrintfStatistic(int Process,int ProcessId)
    {
        int count;
        int pro;

        pro = Process;
        count = Process1Id;

/*判断是否输出，防止打印过于频繁*/

        if((fmod(count, PRINT_FREQ) == 0)&&(count != 0))
        {
         printf("Process %d is scheduled %d times. \n",pro,count);
        }
        return;
```

```
    }

/********************* sofsche2.c 程序的结尾*****************************/
```