VxWorks开发人员指南丛书

WIND RIVER

VxWorks程序员指南

[美]Wind River 著

王金刚 高 伟 苏 琪 丁大尉 姜 平 译

清华大学出版社

VxWorks 开发人员指南丛书

组稿编辑：曾　刚　文稿编辑：肖　丽　封面设计：秦　铭

《VxWorks 程序员指南》

《VxWorks 网络程序员指南》

《Tronado 用户指南》

《VxWorks BSP 开发人员指南》

ISBN 7-302-06853-4

9 787302 068532 >

定价：37.00 元

VxWorks 开发人员指南丛书

VxWorks 程序员指南

[美] Wind River 著

王金刚 高 伟 苏 琪 丁大尉 姜 平 译

清 华 大 学 出 版 社

北 京

内 容 简 介

本书是《VxWorks 开发人员指南丛书》之一——VxWorks 程序员指南，根据 Wind River 公司的技术文档“VxWorks Programmer's Guide”翻译而成。

主要内容包括：VxWorks 简介、操作系统基础知识、POSIX 标准接口、输入/输出接口技术、局部文件系统、目标调试工具的使用、C++开发技术、闪存模块驱动设计与应用、VxDCOM 应用程序、分布式消息队列、标准存储对象、虚拟内存接口等内容。

本书语言通畅、条理清晰、内容详细，主要针对从事以 VxWorks 操作系统为基础内核的嵌入式系统开发人员，可作为了解 VxWorks 并且将其应用到开发项目中的指导手册。

图书在版编目（CIP）数据

VxWorks 程序员指南/（美）风河公司著，王金刚等译. 北京：清华大学出版社，2003
（VxWorks 开发人员指南丛书）
ISBN 7-302-06853-4

Ⅰ. V… Ⅱ. ①风… ②王… Ⅲ. 实时操作系统，VxWorks-指南 Ⅳ. TP316.2-62

中国版本图书馆 CIP 数据核字（2003）第 055306 号

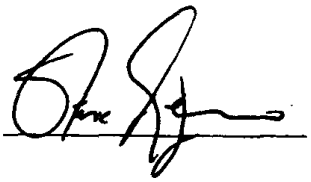
出 版 者：清华大学出版社　　**地　　址**：北京清华大学学研大厦
http://www.tup.com.cn　　**邮　　编**：100084
社 总 机：010-62770175　　**客户服务**：010-62776969
组稿编辑：曾　刚
文稿编辑：肖　丽
封面设计：秦　铭
版式设计：茹桂兰
印 刷 者：清华大学印刷厂
发 行 者：新华书店总店北京发行所
开　　本：185×260　**印张**：26.5　**字数**：602 千字
版　　次：2003 年 8 月第 1 版　2003 年 8 月第 1 次印刷
书　　号：ISBN 7-302-06853-4/TP・5084
印　　数：1～5000
定　　价：37.00 元

授 权 声 明

"Wind River Systems, Inc. does not endorse this publication, nor has Wind River reviewed its contents for accuracy. This publication is the sole creation of Tsing Hua University Press, which bears sole responsible for its contents. Wind River has granted the Tsing Hua University Press a limited right to use the marks VXWORKS, TORNADO, VXSIM, CROSS WIND, WIND SH, WINDVIEW, WIND and WIND RIVER for identification purposes only. This grant does not indicate that Wind River has endorsed the contents of the book. VXWORKS, TORNADO, VXSIM,CROSS WIND, WIND SH, WINDVIEW, WIND and WIND RIVER are trademarks and/or registered trademarks of Wind River Systems, Inc. and are used with permission in this publication."

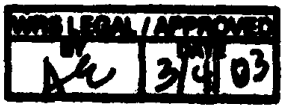
Wind River Systems, Inc.授权清华大学出版社编译出版本技术文档，由清华大学出版社完全负责该技术文档的编译、编辑、审校、出版；此出版物的中文版版权为清华大学出版社单独拥有；清华大学出版社对此出版物内容完全负责。Wind River Systems, Inc.已授权清华大学出版社在此出版物中使用其注册商标：VXWORKS、TORNADO、VXSIM、CROSS WIND、WIND SH、WINDVIEW、WIND 和 WIND RIVER。以上授权说明清华大学出版社承认并保证此出版物的内容。VXWORKS、TORNADO、VXSIM、CROSS WIND、WIND SH、WINDVIEW、WIND 和 WIND RIVER 等 Wind River Systems, Inc.的商标和（或）注册商标在此出版物中的使用已得到了 Wind River Systems, Inc.的许可。

WIND RIVER SYSTEMS, INC.

By: 

Name, title STEVE KENNEDY
GROUP VICE PRESIDENT

Date 3.4.03



译　序

VxWorks 是美国风河系统公司（Wind River System 公司，WRS）推出的高性能实时操作系统。WRS 公司组建于 1981 年，是一个专门从事实时操作系统开发与生产的软件公司，该公司在实时操作系统领域具有世界公认的领导地位。

VxWorks 是一个运行在目标机上的高性能、可裁减的嵌入式实时操作系统。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中。由于 VxWorks 的开放式结构及其对工业标准的支持使开发者只需做最少的工作就可有效地适合于不同的用户要求，越来越受到广大电子行业的青睐。

VxWorks 操作系统自传入中国，短短几年来，逐渐进入了国内通信、国防、工业控制、医疗设备等嵌入式实时应用领域，在国内拥有了较多的用户。特别是最近两年，VxWorks 操作系统越来越多地占据了国内嵌入时实时应用市场。

本书讲述了 VxWorks 实时操作系统以及 VxWorks 工具在实时应用中的使用方法，重点说明基本的函数功能和设备，并介绍了可选产品和相关技术。

本书布局如下：第一章对全书的内容作了简要的介绍，解释了如何查询 VxWorks 操作系统和 Tornado 开发环境的相关文件。第二章讨论 VxWorks 运行环境中处于核心部分的任务设备、任务间通信及中断处理机制。第三章主要介绍 VxWorks 内核中遵守 POSIX 标准的接口，以及专门为 VxWorks 设计的接口。第四章详细描述 VxWorks 操作系统中的 I/O 系统。第五章讲述 VxWorks 操作系统在文件系统与设备驱动程序之间使用的一种标准 I/O 操作接口以及如何在单个 VxWorks 操作系统中运行多个相同或不同种类的文件系统。第六章对基于目标机的设备进行详细讨论。第七章提供了 C++开发的相关内容，这些信息适用于使用 Wind River 公司 GNU 和 Diab 工具的 VxWorks 操作系统。第八章简要介绍闪存，并描述建立一个支持 TrueFFS 系统的步骤。第九章主要讨论 VxDCOM 设备。第十章介绍 VxWorks 分布式信息队列。第十一章详细阐述 VxWorks 共享内存对象及其内部需要考虑的问题，并提供配置和解决问题的方法。第十二章讲述 VxWorks 提供的虚拟内存接口。

本书由王金刚、高伟、苏琪、丁大尉和姜平翻译，参加编译工作的人员还有熊辉、宫霄霖、杨锡励等。由于 VxWorks 嵌入式系统在国内的应用还处于起始阶段，所以相关的书籍和材料有限，再加上时间仓促以及编译人员水平有限，不足之处在所难免。希望读者对本书提出宝贵意见。

译者

2003.4.30

目　录

第1章 引 言

1.1 概 要

本指南介绍 VxWorks 实时操作系统以及 VxWorks 工具在实时应用发展中的使用方法。本指南集中介绍了基本的函数功能和设备，并介绍了可选产品和技术，主要内容如下：

- 基本的操作系统函数
- POSIX 标准接口
- I/O 系统
- 本地文件管理，包括 dosFs
- 目标工具，如命令解释器（shell），基于目标机的装载器和目标机符号表
- 使用 GNU 和 Diab 工具链的 C++开发
- 快闪存储设备接口（TrueFFS）
- COM 和 DCOM（VxDCOM）
- 分布式消息队列（VxFusion）
- 共享存储体（VxMP）
- 虚拟存储接口（VxVMI）

本章介绍如何查询 VxWorks 操作系统和 Tornado 开发环境的相关文件。此外，本章还介绍本指南中的 Wind River 公司的客户服务和文档规范。

1.2 相关的文件资源

关于 VxWorks 库和例行程序的详细信息，请参见《VxWorks API 参考》。对于目标机数据结构的专门信息请参阅《VxWorks architecture supplements》[①]和《VxWorks BSP 参考》。

在《VxWorks 网络编程指南》中描述了 VxWorks 网络函数。

关于 VxWorks 和 Tornado 较早版本中的移植应用、BSP、驱动程序以及 Tornado 项目的相关内容，请参考《Tornado 移植指南》。

---

① 例如，VxWorks for PowerPC Architecture Supplement, VxWorks for Pentium Architecture Supplement, VxWorks for MIPS Architecture Supplement, and VxWorks for ARM Architecture Supplement。

在安装和使用 Tornado 开发环境时，可参考下列文件：

《Tornado 入门指南》提供了安装 Tornado 开发环境和相关可选产品的信息。

《Tornado 用户指南》提供了关于建立开发环境和使用 Tornado 工具开发 VxWorks 应用程序的详细内容。它不仅包括用本书中描述的不同组件配置 VxWorks 系统的信息，而且还包括建立和运行这些系统的信息。

对于 Tornado 文档的完整描述请参考《Tornado 入门指南》：文档指南。

1.3 VxWorks 配置和建立

本文档描述了 VxWorks 的特征和配置选项，但是没有讨论系统机制，即配置和编译以 VxWorks 为基础的系统机制。用于配置和编译应用程序的工具和程序请参考《Tornado 用户指南》和《Tornado 用户参考》。Tornado 提供了这些配置和编译的 GUI 工具和行命令工具。

⚠ **注意**：与《VxWorks API 参考》一样，本书中 VxWorks 组件是通过系统配置文件所用名称作为标识，例如 INCLUDE_FOO。类似的，配置参数也是通过诸如 NUM_FOO_FILES 形式的配置参数名来识别的。

若使用命令行配置工具和相关的配置参数，那么组件名可直接用于组件区分并进行 VxWorks 系统配置。该方法也适用于配置参数名。

若在 Tornado IDE 中使用 GUI 配置机制，一个简单的搜索工具用组件名能够在 GUI 中搜索出组件。一旦找到所需组件，就能够通过 GUI 访问组件参数。

1.4 Wind River 代码约定

Wind River 公司拥有自己的代码约定，Tornado 及 VxWorks 文档的例子就体现出了该约定。这些约定为从源程序编码生成 Tornado 和 VxWorks API 联机参考文档提供了基础。根据这些约定，可用 Tornado 安装的工具生成各种 HTML 格式的 API 文档。有关详细信息，请参考《Tornado 用户指导》：代码约定。

1.5 文 档 约 定

本节描述了指南中的文档约定。

1. 印刷协定

VxWorks 文档使用列于表 1-1 中的约定来区分不同要素。在表示子程序名时经常使用圆括号，例如：printf( ).

表 1-1　印刷约定

术　语	例　子
文件，路径名	/etc/hosts
库，驱动器	memLib, nfsDrv
主机工具	more, chkdsk
子程序	semTake( )
启动命令	p
代码显示	main ( );
键盘输入	make CPU=MC68040 ...
显示输出	value = 0
用户提供的参数	name
组件和参数	INCLUDE_NFS
C 关键字，cpp 指示词	#define
键盘上已命名的键	RETURN
控制字符	CTRL+C
小写缩写词	fd

2. 交叉引用

对于本书中出现的子程序、库或工具的相关引用请参见《VxWorks API 参考》（关于目标程序或库），或《Tornado 用户指南》（关于主机工具）中的相关条目。关于其他书籍的引用在每章中都有标注，并采用了——书名：章节名——形式；例如，《Tornado 用户指南》：工作站。

对于如何访问在线文档，请参考《Tornado 入门指南》：文件指导。

3. 目录路径名

所有 VxWorks 文件都位于 Tornado 安装目录下的目标路径（和子路径）中，由于安装目录是由用户决定的，路径将采用以下格式：安装目录/目标路径。

例如：将 Tornado 安装在 UNIX 主机的/home/tornado 目录下，或者安装在 Windows 主机上的 C:\tornado 目录下，若需要识别的整个文件路径为安装目录/target/h/vxWorks.h，那么根据本指导手册分别在 UNIX 主机上安装为/home/tornado/target/h/vxWorks.h；在 Windows 主机上安装为 C:\tornado\target\h\vxWorks.h。

对于 UNIX 用户，安装目录等同于 Tornado 环境变量 WIND_BASE。

⚠ **注意**：本书中 VxWorks 系统默认使用“\”作为 UNIX 和 Windows 文件名的路径分隔符。

第 2 章　基本操作系统

2.1　简　　介

现代实时系统是在多任务和任务间通信的基础上建立起来的。一个多任务的环境允许将实时应用构造成一组独立的任务，每个任务拥有各自的线程和一套系统资源。为了协调任务间的行为，任务间的通信设备允许这些任务通过同步和通信操作协调各自的活动。在 VxWorks 操作系统中，任务间通信设备包括信号量、消息队列、管道以及网络套接字等设备。

在实时系统中，处理中断是另一个主要功能，这是因为中断是将外部事件通知系统的重要方式。为了能得到较快的中断响应，VxWorks 操作系统里中断服务程序（ISR）在一个专门的上下文中执行，是处于任务的上下文之外。

本章讨论了 VxWorks 运行环境中处于核心部分的任务设备、任务间通信及中断处理机制。此外 VxWorks 操作系统中也可使用 POSIX 实时扩展接口。详细信息请参考“第 3 章 POSIX 标准接口”。

2.2　VxWorks 任务

通过协调，把应用程序组织成相互独立的程序是非常必要的。在执行时每个程序都被称之为任务。VxWorks 操作系统中，任务可以直接地或者以共享方式访问大多数系统资源，为了维护各自的线程，每个任务必须保持有足够的上下文环境。

⚠ **注意**：POSIX 标准包括“线程”的概念，该概念类似于任务，但比任务有更多的特性。用户要了解详细信息，请参考“3.4 POSIX 线程”。

2.2.1　多任务

多任务提供一种机制，用于响应现实世界中多重的、离散的事件。VxWorks 实时内核——Wind 提供了基本的多任务环境。多任务构造出多线程并发执行的假象，但实际上，系统内核是根据某个调度算法交错执行的。每个任务拥有各自的上下文，即拥有各自的 CPU

环境和系统资源（指任务被内核调度执行时所使用的资源）。在上下文切换时，任务的上下文保存在任务控制块（TCB）中。

任务的上下文包括：

- 任务的执行点，即任务的程序计数器
- CPU 中的寄存器和浮点寄存器（可选）
- 动态变量和函数调用所需的堆栈
- I/O 操作分配的标准输入、标准输出和标准错误输出操作
- 一个延时定时器
- 一个时间片定时器
- 内核控制结构
- 信号句柄
- 用于调试和性能监视的值

VxWorks 操作系统中内存地址空间是一个非常重要的资源，由于其所有代码在一个单一的公共地址空间内执行。为每个任务建立独立的存储空间，需要从虚拟地址向物理地址映射的机制，该机制仅在使用可选产品 VxVMI 时有效。详细信息请参考“第 12 章虚拟存储接口”。

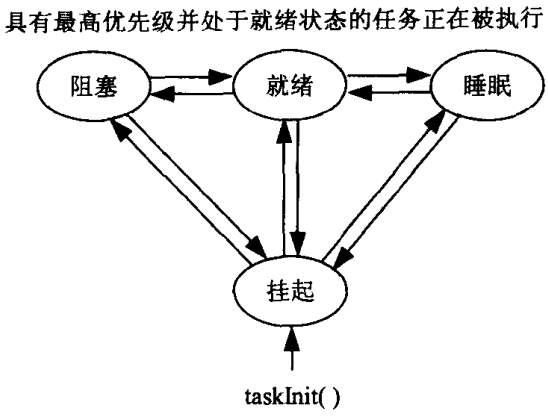
2.2.2 任务状态转变

在操作系统里内核负责维护每个任务的当前状态。若应用程序调用了内核程序，任务将会从一个状态改变到另一个状态。任务创建时处于挂起状态。必须激活一个创建的任务才能使其进入就绪状态。激活阶段相当快，因此应用程序应该先创建任务，并且及时地将其激活。另一种方法就是使用发起任务（spawning）的原语，调用一个函数就能创建并激活任务；任务可以在任何一种状态被删除。

表 2-1　任务状态符号

状 态 符 号	描　　述
就绪（READY）	该状态时任务仅等待 CPU 的状态，不等待其他任何资源
阻塞（PEND）	任务由于一些资源不可用而被阻塞时的状态
睡眠（DELAY）	处于睡眠的任务状态
挂起（SUSPEND）	该状态时任务不执行，主要用于调试用。挂起仅仅约束任务的执行，并不约束状态的转换，因此 pended-suspended 状态时任务可以解锁，delayed-suspended 状态时任务可以唤醒
DELAY + S	既处于睡眠又处于挂起的任务状态
PEND + S	既处于阻塞又处于挂起的任务状态
PEND + T	带有超时值处于阻塞的任务状态
PEND + S + T	带有超时值处于阻塞，同时又处于挂起的任务状态
state + I	任务处于 state 且带有一个继承优先级

表 2-1 中描述了在使用 Tornado 开发工具时常见状态符号；图 2-1 是对应了内核（Wind）状态的状态图。



ready	→	pended	semTake( )/msgQReceive( )
ready	→	delayed	taskDelay( )
ready	→	suspended	taskSuspend( )
pended	→	ready	semGove( )/msgQSend( )
pended	→	suspended	taskSusPend( )
delayed	→	ready	expired delay
delayed	→	suspended	taskSuspend( )
suspended	→	ready	taskResume( )/taskActivate( )
suspended	→	pended	taskResume( )
suspended	→	delayed	taskResume( )

图 2-1　任务状态转换

2.2.3　Wind 任务调度

多任务系统需要使用一个调度算法把 CPU 分配给就绪的任务。在 Wind 内核中，默认算法是基于优先级的抢占式调度算法。当然也可使用轮转调度算法。这两种算法都依赖于任务的优先级。Wind 内核里有 256 种优先级，优先级从 0 到 255，优先级 0 为最高，优先级 255 为最低。在创建任务时，需要分配给任务一个优先级。调用 taskPrioritySet( )可以改变任务的优先级。这种动态地改变任务优先级的功能可以使任务跟踪现实世界的优先关系变化。控制任务调度函数列于表 2-2。

表 2-2　任务调度控制函数

调　用	描　述
kernelTimeSlice( )	控制轮转调度
taskPrioritySet( )	改变任务优先级
taskLock( )	禁止任务调度
taskUnlock( )	允许任务调度

POSIX 还提供了一个调度接口，详细信息请参考“3.5POSIX 调度接口”。

1. 基于优先级的抢占式任务调度

使用基于优先级的抢占式任务调度算法，当一个新任务优先级高于系统当前执行任务的优先级时，它将抢占 CPU 执行。因此，系统内核将确保 CPU 分配给处于就绪状态的具有最高优先级的任务执行。这意味着如果某个任务比当前任务的优先级高，并处于就绪状态，那么系统内核将立刻保存当前执行任务的上下文，并切换到高优先级任务的上下文中去。例如，在图 2-2 中，任务 t1 被优先级高的任务 t2 抢占，但任务 t2 同样又被优先级更高的任务 t3 抢占，那么当 t3 结束后，t2 将继续执行；同样当 t2 完成执行后，t1 将继续执行。

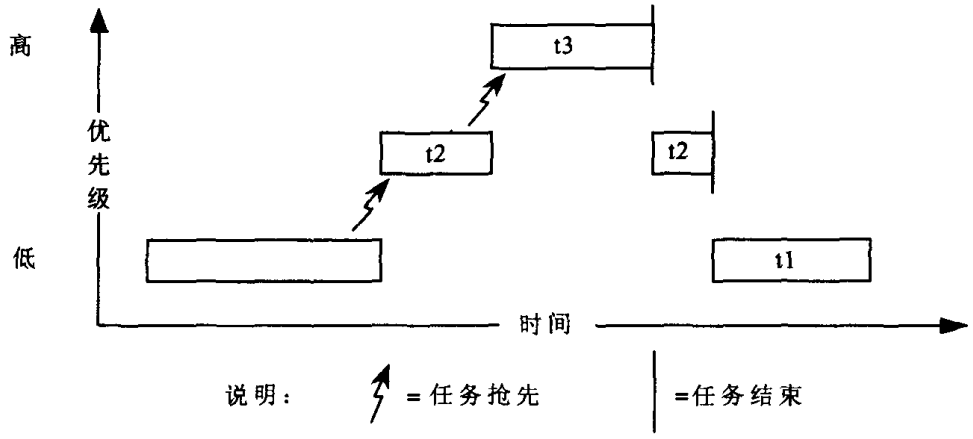


图 2-2 优先级抢占

这种调度算法的缺点是：当多个相同优先级的任务需要共享一台处理器时，如果某个执行的任务永不阻塞，那么它将一直独占处理器，其他相同优先级的任务就没有机会执行。但是使用轮转式调度算法可以解决这一点。

2. 轮转式调度

当所有相同优先级的任务处于就绪状态时，轮转算法倾向于平均使用 CPU。轮转调度算法对于所有相同优先级的任务，通过时间片获得相同的 CPU 处理时间。在一组相同优先级的任务里，每个任务将在一个规定的时间间隔或时间片内执行。

调用 kernelTimeSlice( )将启用轮转调度算法，其参数为时间片的长度或者一段时间间隔，即某个任务放弃 CPU 给另一个相同优先级的任务执行之前，系统允许该任务执行的时间长度。因此任务的执行状态不断轮转，每个任务各自执行一段相等的时间。在相同的优先级队列中，在所有任务都执行一遍之前，没有任务会得到第二块时间片。

在大多数系统里，并不一定需要使用轮转调度算法。但在相同代码被多份复制执行时，例如在用户接口任务内执行时，需要使用轮转调度算法。

如果在使用轮转调度算法的基础上，对当前执行的任务使用抢占算法，系统时间片计数器将增加该任务的可执行时间长度。当规定时间段结束时，时间片计数器将清零，并根据优先级把该任务放到相同优先级任务队列的尾部。新加入到优先级队列的任务放在队列尾部，该任务的时间片计数器初始化值为零。

使用轮转调度算法既不影响任务上下文的切换性能，也不影响其他内存的分配。

在任务的执行时间片内，如果该任务被阻塞或者被更高优先级的任务抢占，那么将保存其时间片计数值，并且在其重新执行时恢复计数。对于抢占情况，当抢占的高优先级任务完成执行后，只要没有其他更高优先级任务抢占执行，那么原任务将继续执行。而对于任务阻塞情况，根据任务优先级将其放在队列尾部。在轮转调度时，若禁止使用抢占，那么执行任务的时间片计数值维持不变。

当出现一个系统时钟标记（tick）时，无论该任务在整个时间片内是否执行，时间片计数值将自动增加。由于可能被高优先级的任务抢占，或者被中断服务程序占用了 CPU 处理时间，任务执行的全部 CPU 处理时间可能与预先分配的时间片有偏差。

图 2-3 为三个相同优先级任务 t1、t2 和 t3 的轮转调度。任务 t2 被一个更高优先级的任务 t3 抢占。当 t4 执行结束后，t2 将在其中止处继续执行。

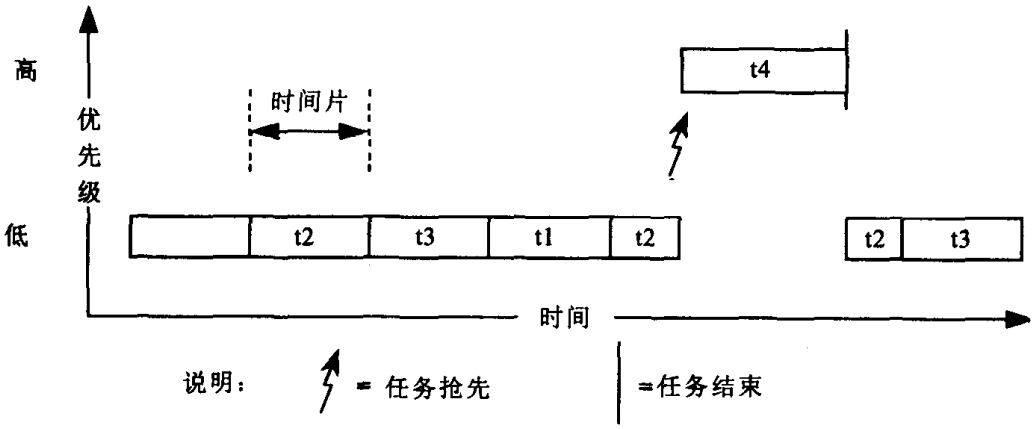


图 2-3　轮转调度

3. 抢占上锁

通过调用 taskLock( )和 taskUnlock( )函数，可以禁止使用 Wind 内核调度程序或启用 Wind 内核调度程序。当任务调用 taskLock( ) 函数时，将禁止使用调度程序，若该任务正在执行时，不会发生基于优先级的抢占。

但是，如果任务在执行中被阻塞或者挂起，调度程序将选择有资格执行且优先级最高的任务执行。当抢占上锁的任务被解除阻塞并且重新开始执行时，抢占再一次被禁止。

注意，抢占上锁只能阻止任务的上下文切换，并不禁止中断。

抢占上锁可以实现互斥操作，但需保证抢占上锁时间尽可能短。详细信息请参考“2.3.2 互斥”。

4. taskLock( )和 intLock( )比较

使用 taskLock( )函数时不产生互斥操作。一般说来，若由于硬件引起的中断，系统将最终返回到任务中。但是一旦遇到阻塞，任务将会闭锁，因此从程序返回之前，需要调用 taskUnlock( )函数。

当任务访问一个变量或者数据结构时，由于该变量或数据结构可能也会被中断服务程序访问，可以使用 intLock( )实现互斥。在单处理器的处理环境中，intLock( )函数可以实现操作过程的最小化。操作过程保持最小化是最好的情况，这意味着只有一些代码没有函数调用过程。如果调用过程太长，将直接影响中断响应时间，增加系统的不确定性。

5. 驱动程序支持的任务优先级

所有应用任务的优先级应该在 100~250 之间；但是驱动程序支持的任务（与中断服务程序关联的任务）优先级能够位于 51~99。这些任务是很重要的；例如，某个任务从芯片内复制数据失败，那么设备将不能获得数据[①]。系统函数 netTask( )的优先级是 50，所以用户使用任务的优先级应分配在 50 以上，如果用户的任务出错，网络连接就会出现死机，并阻止 Tornado 工具进行调试。

2.2.4 任务控制

本节概述了基本的 VxWorks 操作系统中的任务函数，这些函数都包含在 VxWorks 操作系统库 taskLib 中。这些函数提供了任务的创建、控制以及恢复任务信息的方法。关于 taskLib 的详细信息，请参考《VxWorks API 参考》的相关条目。

对于交互式操作，用户可以通过主机或者目标机命令解释器（Shell）来控制 VxWorks 操作系统中的任务；请参考《Tornado 用户指南》：Shell 和本手册中“第 6 章目标机工具”的内容。

1. 创建和激活任务

列于表 2-3 中的函数用于创建任务。

taskSpawn( )函数的参数为：新任务名（一个 ASCII 字符串），优先级，可选字段，堆栈尺寸，入口函数地址及 10 个传给入口函数的启动参数：

```
id = taskSpawn(name, priority, options, stacksize, main, arg1, arg10 );
```

函数 taskSpawn( )创建新任务的上下文，包括分配堆栈和建立用于调用带有特定参数的程序（一个普通的子程序）的任务环境。新任务在指定函数的入口处开始执行任务。

[①] 例如，一个网络接口，一个高级数据链路控制等。

表 2-3　任务创建函数

调　用	描　　述
taskSpawn( )	Spawns 创建并激活一个新任务
taskInit( )	初始化一个新任务
taskActivate( )	激活一个初始化的任务

taskSpawn( ) 函数包含分配、初始化和激活等一些低级操作。初始化和激活功能由 taskInit( ) 函数和 askActivate( ) 函数完成。建议用户只有对分配或激活需要更多控制时再使用这些函数。

2. 任务堆栈

如果不去分析系统配置，确切的说很难知道有多少堆栈空间可分配。为了避免堆栈溢出和破坏任务堆栈，可使用下列方法。在最初分配堆栈空间时，分配比预先估计的空间大一些；例如，对于 20KB 到 100KB 范围，根据应用类型可分配大一点的空间，然后周期性地调用 checkStack( ) 函数监控；若可以安全使用更小的空间，将修改分配空间的尺寸。

3. 任务名和 ID 号

当发起一个任务后，可以指定一个任意长度的 ASCII 字符串为任务名。VxWorks 操作系统返回一个长度为 4 字节并指向任务数据结构的任务 ID 号。大多数 VxWorks 操作系统程序使用任务 ID 号来定位任务。VxWorks 操作系统约定任务 ID 号为 0 值时表示任务调用。

VxWorks 操作系统不需要惟一的任务名，但为了避免混乱，建议使用惟一的任务名。为了充分发挥 Tornado 开发工具的特点，任务名不应该与全局函数名和变量名冲突。为了避免冲突，VxWorks 操作系统使用了任务命名规则：所有从目标机启动的任务以字母 t 开头，而从主机启动的任务以字母 u 开头。

有时用户不想对一些或全部的应用任务命名。如果给 taskSpawn( )函数的任务名参数提供一个空指针，那么 VxWorks 操作系统将给任务分配惟一的名字 tN，这里 N 是一个随机命名任务递增的十进制整数。

列于表 2-4 中的 taskLib 库函数用于任务 ID 号和任务名管理。

表 2-4　任务名和 ID 函数

调　用	描　　述
taskName( )	得到与任务号相关的任务名
taskNameToId( )	寻找与任务名相关的任务 ID
taskIdSelf( )	获得调用任务的 ID 号
taskIdVerify( )	检查一个特定任务的存在性

4. 任务选项

当启动一个任务后，可以传递一个或多个列于表 2-5 中参数选项；其结果是由指定选

项上“逻辑或”操作来决定的。

表 2-5 任务选项

名 称	十六进制	描 述
VX_FP_TASK	0x0008	用浮点运算协处理器来执行
VX_NO_STACK_FILL	0x0100	不要用 0xee。填充堆栈
VX_PRIVATE_ENV	0x0080	用私有环境执行任务
VX_UNBREAKABLE	0x0002	禁止断点
VX_DSP_TASK	0x0200	1 = DSP 协处理器支持
VX_ALTIVEC_TASK	0x0400	1 = ALTIVEC 协处理器支持

在任务创建时，若需执行下列操作则要包括 VX_FP_TASK 选项：

- 实行浮点操作
- 调用返回浮点值的函数
- 调用以浮点值为参数的函数

例如：

```
tid = taskSpawn ("tMyTask", 90, VX_FP_TASK, 20000, myFunc, 2387, 0, 0, 0,
0, 0, 0, 0, 0, 0);
```

一些函数在内部执行浮点操作。VxWorks 操作系统文档明确地指出对于这一类函数需要使用 VX_FP_TASK 选项。

任务发起后，通过使用列于表 2-6 中的函数，可以检查或修改任务的选项，目前只有 VX_UNBREAKABLE 选项可以被修改。

表 2-6 任务选项函数

调 用	描 述
taskOptionsGet( )	检查任务选项
taskOptionsSet( )	设置任务选项

5. 任务信息

在调用表 2-7 中的函数时，可以根据任务上下文获得任务的信息。由于任务状态是动态的，除非知道任务处于睡眠状态（即挂起），否则不能获得当前信息。

表 2-7 任务信息函数

调 用	描 述
taskIdListGet( )	用 ID 填充一组所有激活的任务
taskInfoGet( )	得到任务的信息
taskPriorityGet( )	查看任务的优先级

续表

调　用	描　述
taskRegsGet( )	检查任务寄存器（不能使用当前任务时）
taskRegsSet( )	设置任务寄存器（不能使用当前任务时）
taskIsSuspended( )	检查任务是否处于挂起状态
taskIsReady( )	检查任务是否处于就绪状态
taskTcb( )	获得任务控制块的指针

6. 任务删除和删除安全

从系统中能够动态地删除任务。VxWorks 操作系统包含列于表 2-8 中用于删除任务以及保护任务免于被删除的函数。

表 2-8　任务删除函数

调　用	描　述
exit( )	终止任务调用，释放内存 （仅对于任务堆栈和控制模块）①
taskDelete( )	终止指定任务，释放内存 （仅对于任务堆栈和控制模块）
taskSafe( )	保护调用任务免于删除
taskUnsafe( )	解除任务删除保护

① 在任务终止时，不释放任务执行期间被分配的内存。

警告：*必须确保任务正常删除。删除任务前，应释放任务所占有的全部共享资源。*

在任务创建时，如果指定的入口程序返回，任务将隐含调用 exit( ) 函数；当调用 taskDelete( ) 函数，一个任务能够删除自身或另一个任务。

当某个任务被删除时，不会通知其他任何任务。taskSafe( ) 函数和 taskUnsafe( ) 函数用于防止任务被意外删除。taskSafe( ) 函数用于保护任务，防止被其他任务删除。当任务在临界区域执行，或者使用某种关键资源时，这种保护是必要的。

注意：*在任务执行结束时，可以使用 VxWorks 操作系统中的事件操作发送一个事件。详细信息，请参考 “2.4 VxWorks 事件”。*

例如，任务可能使用信号量对某个数据结构进行互斥访问。在临界区域内执行时，任务可能会被其他任务删除；由于被删除任务不能完成临界区的访问，该数据结构可能处于一种破坏或者不协调的状态。进一步说，由于信号不可能被任务释放，对于其他任务而言，该临界资源将不再可用，本质上已经被冻结。

为了避免上述情况，调用 taskSafe( ) 函数可以保护获得信号量的任务。任何任务若需删除被 taskSafe( ) 函数保护的任务，将进入阻塞状态。在完成临界资源使用后，受保护的任务调用 taskUnsafe( ) 函数将解除自身保护，并允许被其他任务删除。

为了支持嵌套的删除保护，使用一个计数器来跟踪 taskSafe( ) 函数和 taskUnsafe( ) 函数被调用的次数。仅在计数值为零时允许进行删除操作，即上述两个函数调用的次数相同。保护操作仅对当前任务有效。一个任务不能对另一个任务实施保护操作使其免于被删除。

下面的代码段表明了如何使用 taskSafe( ) 函数和 taskUnsafe( ) 函数去保护一个临界代码区域：

```
taskSafe ( );
semTake (semId, WAIT_FOREVER);   /* 阻塞直至信号量可用 */
.
.  /* 邻居区域代码 */
.
semGive (semId);                 /* 释放信号量 */
taskUnsafe ( );
```

如上例表明，删除操作的安全性经常与互斥紧密联系在一起。为了在使用中更方便和更有效，操作系统中的一种特殊信号量——互斥信号量提供了对删除操作的安全保障。详细信息请参考“互斥信号量”。

7. 任务控制

表 2-9 中的函数对任务的执行提供了直接控制。

表 2-9 任务控制函数

调 用	描 述
taskSuspend( )	挂起任务
taskResume( )	恢复任务执行
taskRestart( )	重新启动任务
taskDelay( )	延迟任务，延迟单位为“tick”
nanosleep( )	延迟任务，延迟单位为纳秒

VxWorks 操作系统的调试功能要求提供挂起和恢复任务执行的函数，这些函数用于冻结任务的状态并进行检查。

由于某些破坏性错误，任务在执行期间需重新启动。重新启动机制——taskRestart( ) 函数，使用原有创建参数重新创建一个任务。

延迟参数为任务提供一个简单的睡眠机制，任务延时常用于轮询机制的应用中。例如，下面调用无需考虑时钟速率，将使任务延时半秒：

```
taskDelay (sysClkRateGet ( )/2);
```

函数 sysClkRateGet( )返回系统时钟的速率，单位为 tick/秒。POSIX 中与 taskDelay( ) 函数对应使用的函数是 nanosleep( )，后者可以直接规定延时的时间单位，两者仅单位不同，使用效果相同，都依赖于系统时钟。详细信息，请参考“3.2 POSIX 时钟和计时器”。

另一方面，taskDelay( ) 函数把作为调用者的任务移动到相同优先级队列的尾部。特别是，当调用 taskDelay(0) 时，将会把 CPU 交给系统中其他相同优先级的任务。

```
taskDelay (NO_WAIT);        /* 允许其他相同优先级的任务运行 */
```

延时为零时，只能调用 taskDelay( ) 函数；函数 nanosleep( )中的延时参数禁止为零。

⚠ **注意**：ANSI 和 POSIX 的 APIs 类似。

系统时钟资源典型值为 60Hz（每秒 60 次）。对于时钟延时单位来说该频率太低了，最好能达到 100Hz 或者 120Hz。由于周期性延时是一种有效的轮询机制，因此用户可以考虑使用事件驱动技术来替代。

2.2.5　任务扩展函数

为了允许其他相关的任务函数加入到系统中去，VxWorks 操作系统提供“hook”函数。该函数允许任务在创建、删除和上下文交换时调用附加的函数。任务控制块（TCB）里仍有多余空间可用作任务上下文的应用扩展。

表 2-10 中的函数为“hook”函数，详细信息请参考 taskHookLib 中的相关条目。

表 2-10　任务创建、上下文交换和删除

调　用	描　述
taskCreateHookAdd( )	增加一个在每个任务创建时都调用的函数
taskCreateHookDelete( )	删除一个以前加入的任务创建函数
taskSwitchHookAdd( )	增加一个在每个任务切换时都调用的函数
taskSwitchHookDelete( )	删除一个以前加入的任务切换函数
taskDeleteHookAdd( )	增加一个在每个任务被删除时都调用的函数
taskDeleteHookDelete( )	删除一个以前加入的任务删除函数

在使用“hook”函数时，注意下列限制事项：

- 执行任务切换的“hook”函数必须保证不在任何 VM 上下文中执行，而在内核上下文中执行（与中断服务程序类似）。
- 执行任务交换的“hook”函数禁止依赖于当前任务的信息，或者调用依赖于任何信息的函数，例如函数 taskIdSelf( )。

- 一个执行交换的"hook"函数，即使已经执行了析构程序，仍然禁止使用 taskIdVerify（pOldTcb）机制决定是否删除"hook"函数。但需要改变其他的一些状态信息，例如，在删除"hook"函数时可使用能被执行交换的"hook"函数发现的 NULL 指针。

TaskCreate 类中的"hook"函数在创建任务的上下文里执行，任何新对象都属于创建任务的保护域，或者属于创建任务。因此，为了防止在创建任务终止时使用不需要的对象，需要给新对象分配所有权。

由于用户安装的执行交换的"hook"函数是在内核上下文中被调用的，因此它不能调用所有的 VxWorks 操作系统函数。表 2-11 总结了执行交换的"hook"函数中能被调用的函数；一般说来，通常不涉及到内核的函数都可被调用。

表 2-11 交换函数可调用的函数

库	函 数 调 用
bLib	所有函数
fppArchLib	fppSave( )，fppRestore( )
intLib	intContext( )，intCount( )，intVecSet( )，intVecGet( )，intLock( )，intUnlock( )
lstLib	除 lstFree( )外的所有函数
mathALib	如果使用了 fppSave( )/fppRestore( )，所有函数均可调用
rngLib	除 rngCreate( )和 roundlet( )外所有函数
taskLib	taskIdVerify( )，taskIdDefault( )，task 中断服务程序 ready( )，taskIsSuspended( )，taskTcb( )
vxLib	vxTas( )

⚠ **注意**：对于 POSIX 扩展信息，请参考"第 3 章 POSIX 标准接口"。

2.2.6 任务的错误状态：errno

按照惯例，C 库函数设置一个单一的全局整形变量 errno，当函数发生错误时，将 errno 设置成一个适当值，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。

1. errno 的分层定义

VxWorks 操作系统中，errno 用两种不同的定义方式。作为 ANSI C 里的一个潜在的全局变量 errno，它能够在 Tornado 开发工具中利用变量名显示，请参考《Tornado 用户指南》。Errno 还以宏方式在 errno.h 中被定义，该宏定义除了一个函数以外对 VxWorks 操作系统中的其他函数均可见。该宏被定义成调用函数__errno( )，并返回全局变量 error 的地址，（该函数不能调用自身），这一功能产生了一个有用的特征：由于__errno( )是一个函数，用户在调试时可加入断点，来决定何处发生了错误。

但是宏 errno 结果是全局变量 errno 的地址，C 程序要用标准方法来设置 errno 值：

```
errno = someErrorNumber;
```

由于一些其他有关 error 的应用方式，请不要使用与 errno 相同名字的局部变量。

2. 任务各自的 errno 值

在 VxWorks 操作系统里，潜在的全局变量 errno 是一个已定义的全局变量，它能够直接被与 VxWorks 操作系统（或者在主机上，或者在下载时）相连接的应用代码所使用。但是，对于使用 VxWorks 操作系统多任务环境里的 errno，每个任务必须参照自身的 errno 版本。因此在每次发生上下文切换时，errno 被系统内核保存并且恢复成每个任务上下文的一部分。类似的中断服务函数（ISR）也需参照各自的 errno 版本。

作为由内核自动提供（请参考“2.6.1 中断处理连接程序”）的中断进入和退出代码的一部分，这是在中断堆栈中通过保存和恢复 errno 完成的。因此，无需考虑 VxWorks 操作系统中的上下文，错误代码可以直接通过对全局变量 errno 的操作来存储或查询。

3. 出错返回约定

几乎所有的 VxWorks 函数都遵循一个约定：由函数的实际返回值来表明函数操作成功与否。许多函数仅返回状态值 OK(0)或者 ERROR(-1)。有些函数操作正常时，返回一个非负的整数（例如，函数 open( )返回一个文件描述符），有时返回 ERROR 表明一个错误状态。对于指针函数经常返回 NULL(0)来表示一个错误。在大多数情况下，一个函数返回这类的错误并设置 errno 为一特别值，表明发生了哪种错误。

VxWorks 函数不对全局变量 errno 清零，因此 errno 值总由最后的错误状态设置。当 VxWorks 子函数调用另一个函数发生错误时，经常返回给它自己一个错误指示而不修改 errno。因此，底层程序设置的 errno 值作为错误的指示值仍然有效。

例如，连接硬件中断与用户函数的 VxWorks 函数 intConnect( )，调用 malloc( )来分配内存，并在分配的内存中建立中断驱动程序。如果系统没有足够的内存，malloc( ) 调用失败，此时它设置error为一个代码——表示内存分配库memLib中发现内存分配不足。malloc( )返回 NULL 表明发生了错误。中断连接函数 intConnect( )收到从 malloc( )中返回的错误信息 NULL，然后返回给它自己一个错误指示——ERROR，但它没有改变 errno，errno 仍然是由 malloc( )设置代码表示“内存不足”。例如：

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

Wind 推荐用户在自己的子程序里使用这种机制，设置并检查 errno 作为调试的技术手段。若 errno 值对应错误状态符号表 statSymTbl 中的某个字符串，那么函数 printErrno( )就能够显示出与 errno 相关的这一字符串常量。错误状态值和建立 statSymTbl 的详细信息，请参考 errnoLib 相关条目。

4. 出错状态值分配

大多数情况下，VxWorks errno 值编码模块由两个高字节构成，对于单个错误值至少使用 2 个字节。所有 VxWorks 模块编码范围都在 1~500 之间；errno 值为 0 用于资源兼容。

应用程序可以使用其他所有 errno 值（即大于或等于 128256 的正数和所有负数）。对于使用本协定定义和代码 errno 值的详细信息，请参考 errnoLib 相关条目。

2.2.7 任务异常处理

程序中代码或数据错误会导致硬件异常，诸如非法指令、总线或地址错误、除数为 0 等。VxWorks 操作系统的异常处理包处理所有这些异常。默认异常处理程序将挂起导致异常的任务，并保存任务发生异常点的状态。内核和其他任务将继续执行而不会中断。异常描述被传送到可检查任务挂起状态的 Tornado 开发工具中，详细信息请参考《Tornado 用户指南》：Shell 详细信息。

通过信号函数，任务能够激活它们自己的某种硬件异常处理程序。如果任务提供了一个异常信号处理程序，那么将不再使用上述默认异常处理程序。用户自定义的信号处理程序往往对于恢复严重破坏的事件非常有效。用来定义程序中恢复控制点的函数 setjmp( )，以及在信号异常处理时恢复上下文的函数 longjmp( )，都是典型的自定义函数。注意，longjmp( ) 函数用来恢复任务信号状态。

与硬件异常一样，信号也可用来发送软件异常信号。详细描述请参考“2.3.7 信号”以及 sigLib 相关条目。

2.2.8 共享代码和重入

VxWorks 操作系统里，单个子程序或子程序库被许多不同任务调用是很普遍的现象。例如，许多任务可以同时调用 printf( )，但是系统里仅有一份复制。单份代码被多个任务执行称为代码共享。VxWorks 动态连接设备很容易实现代码共享。代码共享可使系统效率更高，更易维护，请参照图 2-4。

代码共享必须是可重入的。如果一份复制的函数被多个任务同时调用且不发生冲突，那么该函数是可重入的。在子函数修改全局或者静态变量时，由于仅存在一份复制数据和代码，所以这是一种典型的冲突。子程序使用这些变量地址可能在不同任务上下文中重叠，或者干扰了其他任务的执行。

VxWorks 操作系统里，大多数函数是可重入的。但若存在一个对应于命名为 someName_r( )的函数，someName( ) 因作为函数重入的版本将认为是不可重入的。例如，ldiv( )有一个对应函数 ldiv_r ( )，则 ldiv( )是不可重入的。

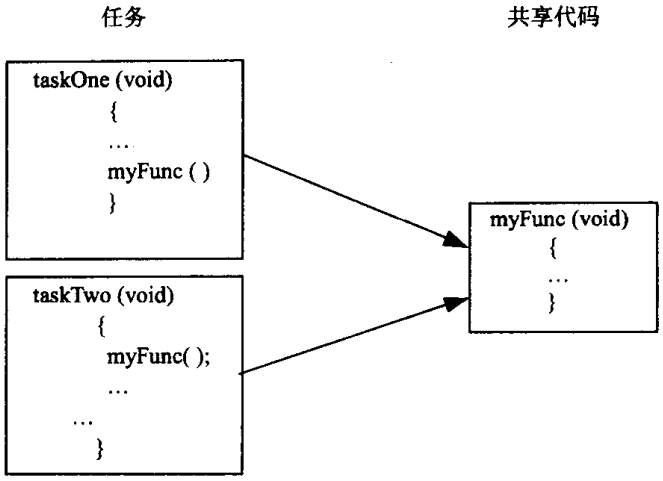


图 2-4　代码共享

VxWorks 操作系统中的 I/O 和驱动程序是可重入的，但应用设计时仍需谨慎使用。对于缓冲 I/O，建议对每个任务使用文件指针缓冲器。由于 VxWorks 文件描述符表是全局的，所以在驱动级可能有来自于不同任务用流加载的缓冲。这可能是你所希望的，也可能不是，这依赖于应用的本质。例如，包驱动程序能够混合不同任务的流，这是因为每个包头部信息指明了目的地址。

大部分 VxWorks 函数使用下列重入技术：

- 动态堆栈变量
- 被信号保护的全局和静态变量
- 任务变量

在编写被多个任务调用的应用代码时，建议使用这些技术。

⚠ **注意**：大多数情况重入代码不是最好的办法，如果调用中断服务程序时，应使用二进制信号量来保护临界段代码，或者使用函数 intLock( )及 intUnlock( )。

虽然逻辑上 Init( ) 函数应该仅被调用一次，但实际上会被多次调用。作为一般规则，为了保持状态信息函数应该避免使用静态变量。但 Init( ) 函数却是一个例外，这里使用了一个返回给 Init( ) 函数成功与否的静态变量是合适的。

1. 动态堆栈变量

许多子函数是纯（pure）代码，除动态堆栈变量以外没有自己的数据。它们以调用者提供的数据为参数进行操作；库 lstLib 就是一个典型例子。在每个子函数调用时，它们操作在调用者提供的 lists 和 nodes 上。

这种子程序本质上是可重入的。多任务能够同时调用这类函数，由于每个任务有各自的堆栈，因此任务间不会相互影响。请参照图 2-5。

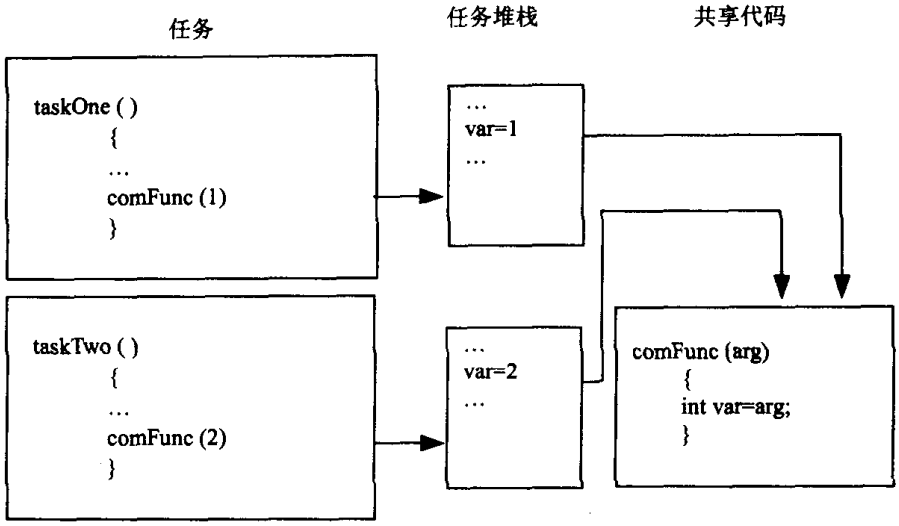


图 2-5　堆栈变量和共享代码

2. 受保护的全局和静态变量

对于公共数据，一些库进行封装访问。由于库函数本质上不是可重入的，所以使用时需要小心。当多个任务同时调用库中函数时，对公共变量的访问可能发生相互冲突。这类库必须借助互斥机制，禁止任务同时在代码临界区域内执行，明确地实现重入。通常使用的互斥机制是 semMLib 提供的 mutex 信号量和“互斥信号量”中描述的互斥信号量。

3. 任务变量

一些可被多个任务同时调用的程序，对每次任务的调用需要不同值的全局变量或静态变量。例如，几个任务可能用同一个全局变量访问相同的私有内存缓冲器。

为了解决这种情况，VxWorks 提供了一种称为任务变量的机制，这种机制允许在任务的上下文中加入 4 个字节的变量，因此在发生任务切换或从它自身切换时，这种变量的值也将进行切换。典型地，几个任务声明相同的变量作为一个任务变量：每个任务都能够把单个的存储地址作为各自的私有变量；请参照图 2-6。这个功能是由 taskVarLib 库中函数 taskVarAdd( )，taskVarDelete( )，taskVarSet( )和 taskVarGet( )实现的。

要非常小心地使用这种机制。由于必须保存和恢复变量值使之成为任务上下文的一部分，每个任务变量将使任务上下文切换时间增加几微秒。考虑到可以收集一个模块的所有任务变量，并存放于一个动态分配的结构中，然后通过一个指针间接地对该结构进行所有的访问，该指针可作为所有使用这个模块任务的任务变量。

4. 使用相同主程序的多任务

使用 VxWorks 操作系统，可以使用相同的主程序发起几个任务。每个发起操作使用各

自的堆栈和上下文建立一个新任务。每个发起操作还能够给新任务传递入口程序的不同参数。在这种情况下，在任务变量中描述的重入规则适用于整个任务。

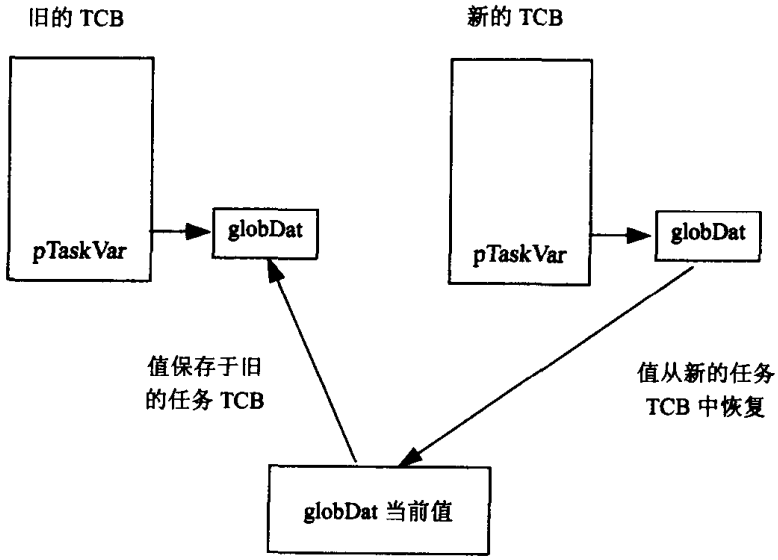


图 2-6　任务变量和上下文切换

这种用法对于相同程序用不同的设置参数并发执行是非常有用的。例如，一个特定的监控类型装备程序可能要发起多次来监控装备中的不同部分。主程序的参数能够指明装备中那部分需要发起任务来监控。

在图 2-7 中，机械臂的多个连接点使用相同的代码。任务调用 joint( )来操纵连接点。连接点的编号（jointNum）表明机械臂上的哪个连接点需要进行操作。

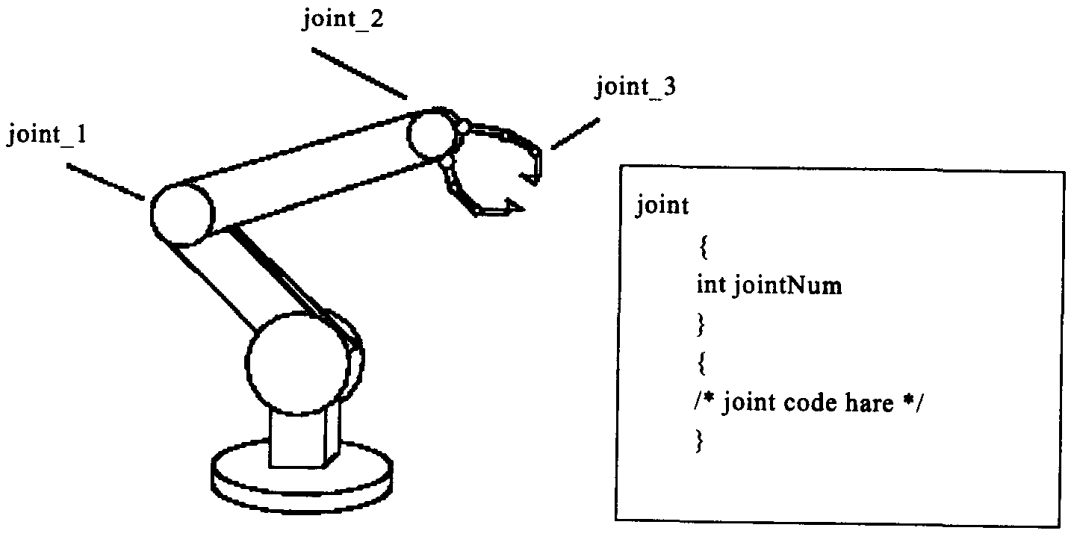


图 2-7　多个任务使用相同的代码

2.2.9 VxWorks 操作系统任务

对于 VxWorks 操作系统的配置，可以包括如下一系列系统任务：

1. 根任务：tUsrRoot

根任务是内核执行的首个任务。根任务的入口点是安装目录/target/config/all/usrConfig.c 下函数 usrRoot( )，该函数可初始化 VxWorks 操作系统的大部分程序，发起诸如日志任务、异常处理任务、网络任务和 tRlogind 后台程序。正常情况下根任务在所有初始化结束后，终止任务并且被删除。

2. 日志任务：tLogTask

日志任务 tLogTask，在当前任务上下文中不执行输入/输出操作时，被 VxWorks 操作系统模块用来记录系统信息。详细信息请参考“4.5.3 信息记录”和 logLib 相关条目。

3. 异常处理任务：tExcTask

异常处理任务 tExcTask，提供 VxWorks 异常处理包，完成在中断中不能执行的功能。它同样适用于当前任务上下文中不能执行的情况，例如任务自我中断。它必须拥有系统的最高优先级。禁止被挂起、删除或改变任务的优先级。详细信息请参考 excLib 相关条目。

4. 网络任务：tNetTask

tNetTask 后台用于 VxWorks 网络任务级程序处理。通常配置 INCLUDE_NET_LIB 组件的 VxWorks 操作系统可以发起网络任务。

5. 目标代理任务：tWdbTask

若目标代理被设置为任务模式执行，将创建目标代理任务 tWdbTask。它处理 Tornado 目标服务器的请求。对于服务器信息，请参考《Tornado 用户指南》：概要。用 INCLUDE_WDB 组件配置的 VxWorks 操作系统包括目标代理功能。

6. 可选组件的任务

如果定义了相关的配置常数，将创建下述的 VxWorks 操作系统任务。详细信息，请参考《Tornado 用户指南》：配置和建立。

7. tShell

如果在 VxWorks 配置中包含了目标机命令解释器（shell），那么目标机命令解释器将作为任务被发起。任何一个从目标机命令解释器调用的程序或任务，将不是发起而是在 tShell 上下文中执行。详细信息，请参考“第 6 章目标工具”。用 INCLUDE_SHELL 组件配置的 VxWorks 操作系统包括目标机命令解释器。

8. tRlogind

如果在 VxWorks 操作系统的配置中包含了目标机命令解释器和 rlogin 功能，该后台允许远程用户登录到 VxWorks 操作系统上。它将接收另一个 VxWorks 操作系统或者主机系统的远程登录请求，并发起 tRlogInTask 和 tRlogOutTask 两个任务。一旦远程用户登录，这些任务就存在于系统中。在远程会议时，shell（以及任何其他任务）的输入/输出需要给远程用户重定向。通过使用虚拟的 VxWorks 操作系统终端驱动程序 ptyDrv，将提供给用户一个类似于 tty 的接口。详细信息，请参考“4.7.1 串行 I/O 设备（终端和伪终端设备）”和 ptyDrv 相关条目。用 INCLUDE_RLOGIN 组件配置的 VxWorks 操作系统包含 rlogin 功能。

9. tTelnetd

如果在 VxWorks 操作系统的配置中包含了目标机命令解释器和 telnet 功能，该后台允许远程用户使用 telnet 登录到 VxWorks 操作系统。它将接收另一个 VxWorks 操作系统或者主机系统的远程登录请求，并发起输入任务 tTelnetInTask 和输出任务 tTelnetOutTask。一旦远程用户登录，这些任务就存在于系统中。在远程会议时，目标机命令解释器（以及任何其他的任务）的输入/输出需要给远程用户重定向。通过使用虚拟的 VxWorks 操作系统终端驱动程序 ptyDrv，将提供给用户一个类似于 tty 的接口。详细信息，请参考“4.7.1 串行 I/O 设备（终端和伪终端设备）”和 ptyDr 相关条目。用 INCLUDE_TELNET 配置的 VxWorks 操作系统包含 telnet 功能。

10. tPortmapd

如果在 VxWorks 配置中包含了 RPC 功能，作为一个服务器，该后台将完成运行在同一台机器上的 RPC 服务中心注册功能。RPC 客户请求 tPortmapd 后台程序寻找如何联系不同类型的服务器。用 INCLUDE_RPC 组件配置的 VxWorks 操作系统包括 portmap 功能。

2.3　任务间通信

“2.2 VxWorks 任务”中描述的多任务通信就是任务间通信功能。这些任务间通信功能协调多个独立任务间的活动。

VxWorks 操作系统提供了一套丰富的任务间通信机制，包括：

- 共享内存，数据的简单共享
- 信号量，基本的互斥和同步
- Mutexe 和条件变量，使用 POSIX 接口时互斥与同步操作
- 消息队列和管道，同一个 CPU 内任务间消息的传递
- Sockets 和远程程序调用，任务间透明的网络通信
- 信号，用于异常处理。

可选产品 VxMP 和 VxFusion，提供了多个 CPU 的任务间通信。请参考“第 11 章共享内存对象”和“第 10 章分布式信息队列”。

2.3.1　共享数据结构

任务间最常用的通信方法是访问共享数据结构。因为在 VxWorks 操作系统中所有任务存在于一个单独的线性地址空间中，所以任务间共享数据结构是很容易实现的，请参照图 2-8。全局变量、线性缓冲、环形缓冲、连接链和指针都可以被运行在不同上下文中的代码直接引用。

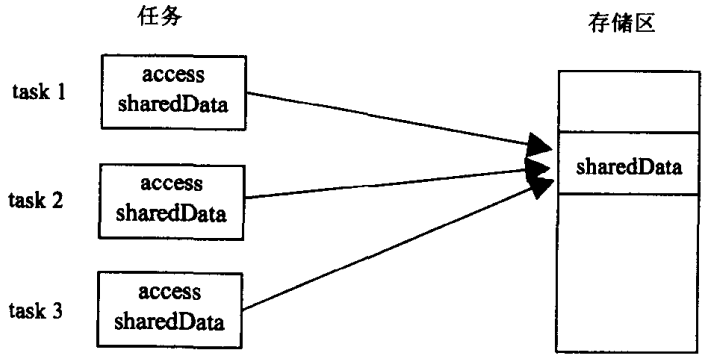


图 2-8　共享数据结构

2.3.2　互斥

当共享地址空间进行简单的数据交换时，为避免竞争，需要对内存进行互锁。实现资源互斥访问的方法很多，不同之处仅在于互斥的范围；例如，禁止中断、禁止抢占以及信号量对资源的上锁。

对于 POSIX 互斥体详细信息，请参考“3.7　POSIX 互斥体（Matexes）和条件变量”。

1. 中断上锁和响应时间

对于互斥使用最强有力的办法是禁止中断。这种上锁的方法保证了对 CPU 的单独访问。

```
funcA ( )
{
int lock = intLock( );
.
.    /* 禁止中断的代码临界区域*/
intUnlock (lock);
}
```

但是这种方法涉及到使用中断服务程序的互斥，由于它在上锁期间阻止系统对外部事件响应，因此对于大部分实时系统，不能作为一种通用的互斥方法。一旦要求紧急响应外部事件时，此时的中断响应时间将难以接受。然而在使用中断服务程序时，有时又需要中断上锁，在这种情况下应保持中断上锁时间尽量短。

警告： 中断上锁时不要调用 VxWorks 操作系统函数。强行使用会导致意外的中断。

2. 抢占上锁和响应时间

当不允许其他任务抢占当前执行任务时，禁止抢占提供了一种较小限制性的互斥，此时，中断服务程序仍然能够执行。

```
funcA ( )
{
taskLock ( );
.
.  /*禁止中断的代码临界区域*/
.
taskUnlock ( );
}
```

但是使用这种方法可能会造成系统实时性得不到充分的保证。尽管高优先级任务本身没有涉及到临界区，然而该任务只能在上锁任务离开临界区后才能够执行，由于使用该种互斥机制简单，因此使用时确保较短的响应时间。“2.3.3 信号量”中讨论的信号量是一个更好的机制。

警告： 临界区代码禁止阻塞。否则，抢占将被重新激活。

2.3.3 信号量

在 VxWorks 操作系统中，信号量被高度优化，并提供了最快的任务间通信机制。信号量是互斥和任务同步的最主要的手段，如下所述：

对于互斥，信号量可对共享资源访问进行互锁，并提供了比“2.3.2 互斥”中讨论的禁止中断和抢占上锁更精确的互斥程度。

对于同步，信号量可协调外部事件与任务之间的执行。

VxWorks 操作系统中三种类型的 Wind 信号量适合于解决不同类型的问题：

二进制

最快最通用的信号量，适用于同步和互斥。

互斥

为解决内在互斥问题、优先级继承、删除安全以及递归问题等而最优化的一种特殊二进制信号量。

计数

类似于二进制信号量，但其跟踪信号量被释放的次数，适用于单个资源多个实例需要保护的情况。

VxWorks 操作系统不仅为系统设计提供了 Wind 信号量，而且为程序的可移植性也提供了 POSIX 信号量。一个可供选的信号量库提供了 POSIX 兼容接口，详细信息请参考“3.6POSIX 信号量”。

这里描述的信号量主要用于单个 CPU 的使用。可供选产品 VxMP 提供了多个处理器间使用的信号量，详细信息请参考“第 11 章共享存储体”。

1. 信号量控制

Wind 信号量对每种类型的信号量的控制不是通过定义一整套控制程序实现的，而是提供了一套统一接口用于信号量的控制。信号量类型仅由创建函数确定。表 2-12 列出了信号量控制函数。

表 2-12　信号量控制函数

调　用	描　　述
semBCreate( )	分配并初始化一个二进制信号量
semMCreate( )	分配并初始化一个互斥信号量
semCCreate( )	分配并初始化一个计数器信号量
semDelete( )	终止并释放一个信号量
semTake( )	获取一个信号量
semGive( )	提供一个信号量
semFlush( )	解锁所有正在等待信号量的任务

函数 semBCreate( )，semMCreate( )和 semCCreate( )返回一个信号量 ID，该 ID 为随后其他信号量控制函数的使用提供句柄。在建立信号量时就已经确定队列的类型，等待信号量的任务可以根据优先级顺序（SEM_Q_PRIORITY）或者先进先出顺序（SEM_Q_FIFO）排队。

警告： semDelete( )调用将终止一个信号量，并释放所有相关内存。为避免删除时其他任务仍需使用该信号量，应小心进行删除操作，尤其是互斥信号量的删除。除非是获得信号量的同一个任务删除它，否则不要删除该信号量。

2. 二进制信号量

使用二进制信号量能够满足两种任务的协调需要：互斥和同步。二进制信号量需要的系统开销最小，因而特别适用于高性能的需求。在“互斥信号量”中讨论的互斥信号量也是一种二进制信号量，但它用于解决内在互斥的问题。在不需要使用互斥信号量的高级特性时，二进制信号量仍可用于互斥。

二进制信号量可被用作为一个标志：资源可用（full）或者不可用（empty）。当一个任务调用 semTake( )函数提取二进制信号量时，其结果依赖于被调用时信号量是可用的（full）还是不可用的（empty）；请参考图 2-9。如果信号量是可用的（full），调用 semTake( )将使信号量变为不可用，同时任务将继续执行。如果信号量不可用（empty），调用 semTake( ) 的任务将被放置到一个阻塞队列中，处于等待信号量可用的状态。

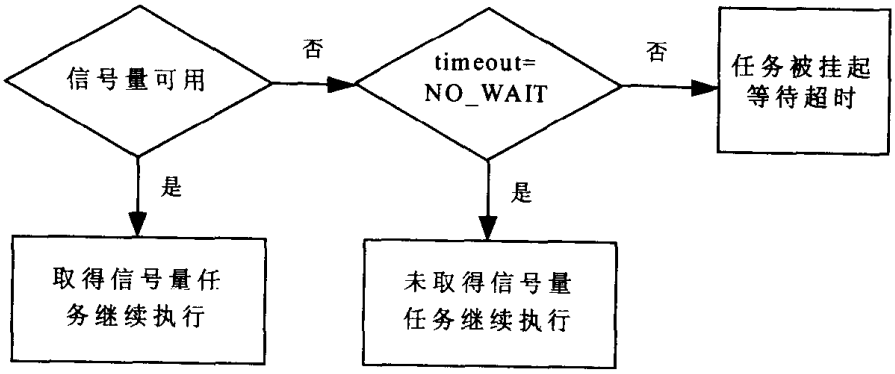


图 2-9　提取信号量

当任务调用 semGive( ) 函数释放二进制信号量时，其结果依赖于被调用时信号量是可用的（full）还是不可用的（empty），请参考图 2-10。如果信号量是可用的（full），调用信号量将不产生任何影响。如果信号量不可用（empty）并且没有任务在等待它，那么信号量将变为可用（full）。如果信号量不可用（empty）并且有任务在等待它，那么阻塞队列中第一个任务将解除阻塞，同时信号量仍为不可用（empty）。

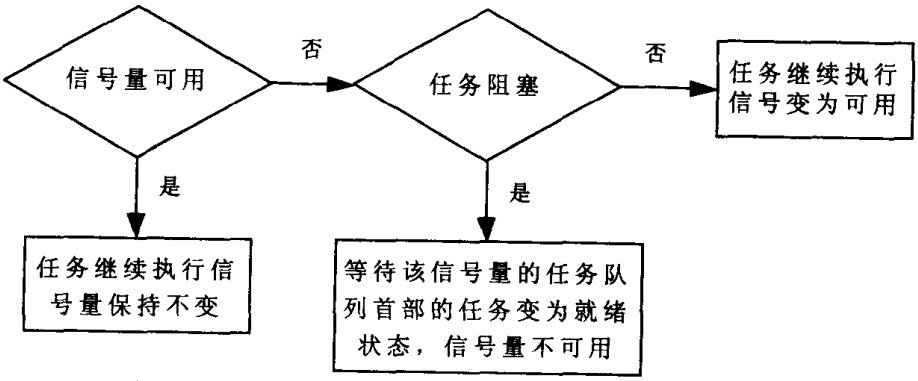


图 2-10　释放信号量

（1）互斥

二进制信号量能有效地对共享资源的访问进行互锁。与禁止中断和抢占上锁不同，二进制信号量仅限制了相关资源的互斥范围。使用该技术时需创建信号量来保护资源，信号量在最初创建时状态为可用（full）。

```
/* includes 语句 */
#include "vxWorks.h"
#include "semLib.h"
SEM_ID semMutex;
/* 创建一个二进制信号量,该信号量最初为可用的*
 *阻塞在信号量上,按优先级顺序等待信号量  */
semMutex = semBCreate (SEM_Q_PRIORITY, SEM_FULL);
```

当任务访问资源时，首先必须获得信号量。只要任务持有信号量，其他所有需要访问该资源的任务将被阻塞。当该任务结束资源访问时释放信号量，允许其他任务访问资源。

因此，对资源的所有访问需要互斥时，可使用函数 semTake( )和 semGive( )来实现。

```
semTake (semMutex, WAIT_FOREVER);
.
.  /* 临界区域,任何时候仅单个任务可以访问*/
.
semGive (semMutex);
```

（2）同步

当信号量用于任务同步时，信号量可用作任务等待的一个状态或事件。初始时信号量不可用。任务或中断服务程序通过释放信号量来表明事件的发生（对于完整的中断服务程序的讨论，请参考“2.6 中断服务代码：中断服务程序”）。调用 semTake( )函数提取信号量的其他任务处于等待状态，这些等待任务被阻塞，直至事件发生，并释放信号量。

注意，信号量在互斥和同步时状态次序不同。对于互斥，信号量初始时为可用（full），每个任务先提取信号量，然后再释放。但是对于同步，信号量初始时为不可用（empty），每个任务都需要等待提取被其他任务释放的信号量。

在例 2-1 中，init( ) 函数建立了一个二进制信号量，把中断服务程序与一个事件相连接，并发起一个任务去处理该事件。函数 task1( )一直运行到调用 semTake( ) 函数。在这个点上，除非有事件发生并引起中断服务程序调用函数 semGive( )，否则将一直处于阻塞状态。当完成中断服务程序后，task1( ) 函数恢复执行，处理该事件。在一个专用的任务上下文中处理这种事件具有一个优点：中断级上的任务处理少，因而减少了中断响应时间。这种事件的处理非常适合于实时系统应用。

例 2-1：使用信号量实现任务同步

```
/* 本例显示了信号量使用于同步的用途. */
/* includes 语句 */
#include "vxWorks.h"
#include "semLib.h"
#include "arch/arch/ivarch.h" /* 用结构体类型代替 arch */
SEM_ID syncSem;                /* 同步信号量 ID */
init (
   int someIntNum
    )
   {
    /* 连接中断服务函数*/
    intConnect (INUM_TO_IVEC (someIntNum), eventInterruptSvcRout, 0);

   /* 建立信号量 */
    syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

 /* 发起用于同步的任务. */
taskSpawn ("sample", 100, 0, 20000, task1, 0,0,0,0,0,0,0,0,0,0);
    }

task1 (void)
{
...
semTake (syncSem, WAIT_FOREVER); /* 等待事件发生 */
printf ("task 1 got the semaphore\n");
...   /* 启动事件 */
}

eventInterruptSvcRout (void)
{
  ...
  semGive (syncSem);        /* 让任务 1 启动事件*/
  ...
}
```

广播同步允许所有阻塞在同一个信号量上的任务自动解除阻塞。这种同步方式正确的应用行为经常需要一个任务集合，在集合中任何一个任务有机会处理更多事件之前，任务集合将整体地处理一个事件。semFlush( )函数是通过解除信号量上所有任务的阻塞状态从而实现同步的。

3. 互斥信号量

互斥信号量是一种用于解决内在互斥问题的特殊的二进制信号量，包括优先级倒置，删除安全以及资源的递归访问。

互斥信号量的基本行为与二进制信号量一致，不同之处如下：

它仅用于互斥；

它仅能由提取它（即调用 semTake( )）的任务释放；

不能在中断服务程序中释放；

semFlush( )函数操作非法。

（1）优先级倒置

图 2-11 表示了优先级的倒置状态。

当一个高优先级任务需要等待一段不确定时间，让低优先级任务完成时，需要发生优先级倒置。考虑到如图 2-11 所示的情况，t1、t2 和 t3 分别是优先级为高、中和低的三个任务。t3 占有由二进制信号量保护的某种资源，从而进入临界状态；当 t1 抢占了 t3，由于需要提取相同的信号量从而引发竞争时，t1 被阻塞了。如果能保证 t1 在 t3 正常完成该资源访问之前被阻塞，将不会出现资源抢占，也就不会出现上述问题。但是，由于低优先级任务容易被中等优先级的任务（如 t2）抢占，t3 可能会再次被阻塞。一旦发生这样的情况，受阻塞的 t1 可能又要等待一段不确定的时间。

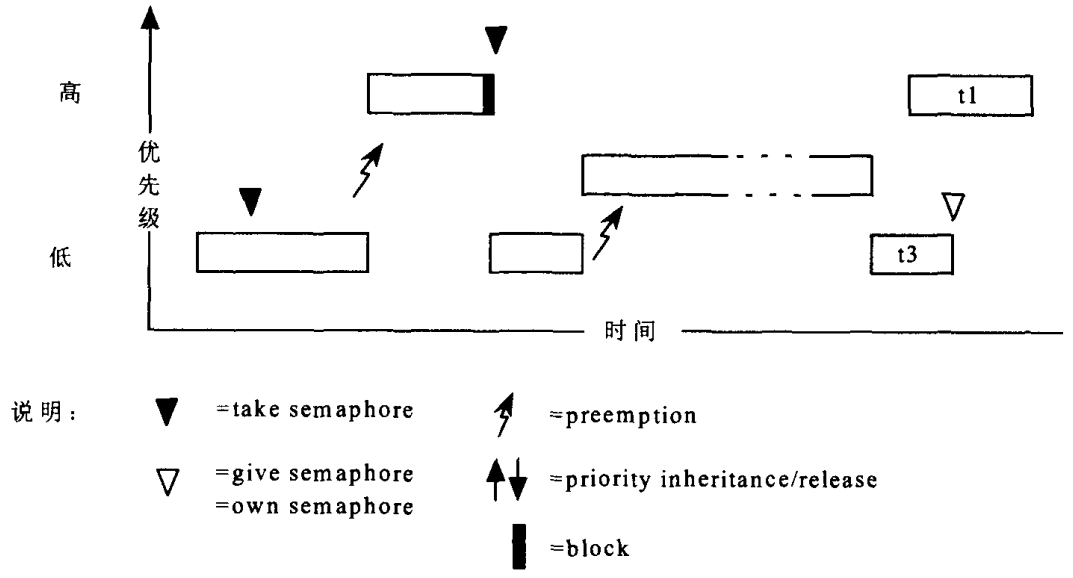


图 2-11　优先级倒置

互斥信号量选项 SEM_INVERSION_SAF 能够继承优先级算法。优先级继承协议确保在资源阻塞的所有任务中优先级最高的且拥有资源执行资格的任务将优先执行。一旦任务的优先级被提高，它以提高后的优先级执行；直到释放其占有的全部互斥信号量后，该任务将返回到正常或者标准的优先级。因此“继承的任务”被保护以防止被任何中间优先级

的任务所抢占。该选项要求与优先级队列（SEM_Q_PRIORITY）一起使用。

图 2-12 中，当 t1 因信号量阻塞时，通过把 t3 优先级提升到 t1 优先级，从而解决了优先级倒置问题。这样不仅保护了 t3；而且间接地保护 t1，以防止被 t2 抢占。

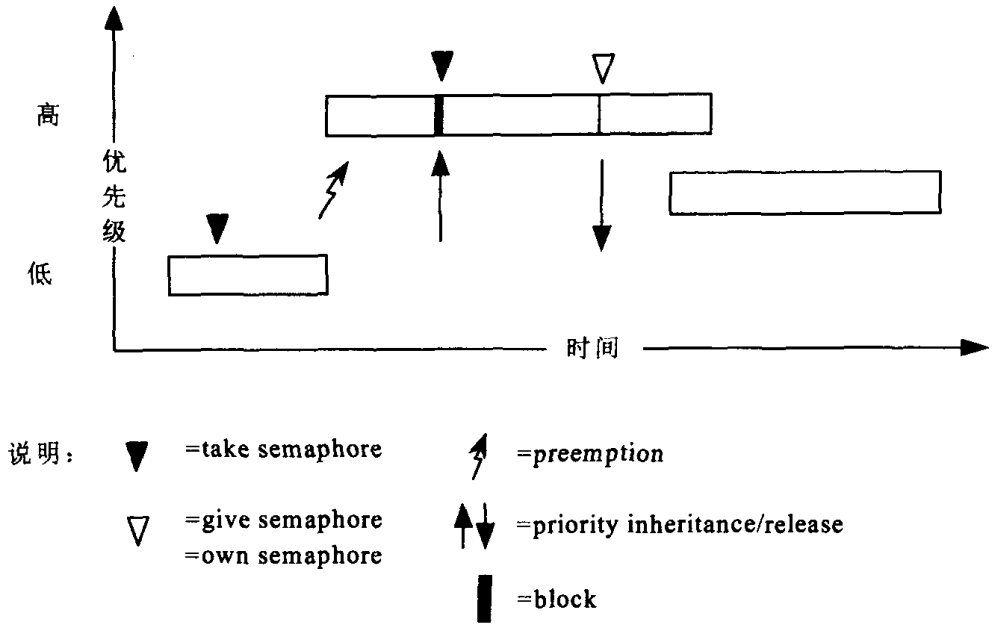


图 2-12　优先级继承

下例用优先级继承算法创建了一个互斥信号量：

```
semId = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);
```

（2）删除安全

互斥的另一个问题涉及到任务的删除。一个受信号量保护的临界区域内经常需要保护执行任务避免被意外地删除。删除一个在临界区执行的任务可能会导致意想不到的后果。资源可能处于破坏状态，同时受保护的信号量不再可用，直接终止对该资源的所有访问。

原语 taskSafe( )和 taskUnsafe( )提供了一种任务删除的方法。但是在使用互斥信号量选项 SEM_DELETE_SAFE 时，每次使用 semTake( )将隐含调用 taskSafe( )，使用 semGive( )将隐含调用 taskUnsafe( )。使用这种方式，任务在占用信号量时不会被删除。该选项由于最终代码进入内核的部分较少，因此比使用原语 taskSafe( )和 taskUnsafe( )更加有效。

```
semId = semMCreate (SEM_Q_FIFO | SEM_DELETE_SAFE);
```

（3）递归资源访问

互斥信号量能够递归获得。这意味着持有信号量的任务在最终释放其之前能够多次地提取。递归非常适用于一组需要相互调用的子程序同时又需进行资源互斥访问的情况。由于系统保持跟踪当前哪个任务持有互斥信号量，因此这是能够实现的。

在释放信号量之前，递归获取的互斥信号量被释放和提取的次数应该相等。这通过一个计数器跟踪实现，每调用 semTake( )一次计数器加一，每调用 semGive( )一次计数器减一。

例 2-2：互斥信号量的递归使用

```
/* 函数 A 访问某资源时需要提取 mySem;
* 函数 A 需要调用函数 B，函数 B 也需要提取 mySem  */

/* includes */
#include "vxWorks.h"
#include "semLib.h"
SEM_ID mySem;

/* 创建一个互斥信号量. */
init ( )
    {
    mySem = semMCreate (SEM_Q_PRIORITY);
    }
funcA ( )
    {
    semTake (mySem, WAIT_FOREVER);
    printf ("funcA: Got mutual-exclusion semaphore\n");
    ...
    funcB ( );
    ...
    semGive (mySem);
    printf ("funcA: Released mutual-exclusion semaphore\n");
    }
funcB ( )
    {
    semTake (mySem, WAIT_FOREVER);
    printf ("funcB: Got mutual-exclusion semaphore\n");
    ...
    semGive (mySem);
    printf ("funcB: Releases mutual-exclusion semaphore\n");
    }
```

4. 计数器信号量

计数器信号量是实现任务同步和互斥的另一种手段。计数器信号量与二进制信号量相似，它还跟踪信号量被释放的次数。每释放一个信号量，计数器加一；每提取一个信号量，计数器减一。当计数器计数为零时，试图提取信号量的任务将被阻塞。与二进制信号量一样，如果信号量被释放时存在被阻塞的任务，那么被阻塞的任务将被解除阻塞。然而，与二进制信号量不同的是，如果信号量被释放时不存在阻塞的任务，那么计数器将加一。这意味着一个被释放二次的信号量，可以无阻塞地被提取二次。表 2-13 列出了初始值为 3 的

计数器信号量被任务提取和释放时的时间顺序。

表 2-13　计数器信号量示例

信号量调用	调用后计数值	调 用 过 程 行 为
semCCreate( )	3	计数值为 3，信号量初始化
semTake( )	2	提取信号量
semTake( )	1	提取信号量
semTake( )	0	提取信号量
semTake( )	0	任务阻塞，等待直到信号量可用
semGive( )	0	等待任务将释放信号量
semGive( )	1	无等待信号量的任务，计数值加一

计数器信号量适用于保护多份复制的资源。例如，可以使用一个初始值为 5 的计数器信号量来协调 5 个磁带驱动器工作；或使用初始值为 256 的计数器信号量来实现一个有 256 个入口的环形缓冲器。计数器信号量的初始值以参数形式用于 semCCreate( )中。

5. 特定信号量选项

Wind 标准信号量接口包括两个特定的选项。这些选项不适用于“3.6　POSIX 信号量”中描述的与 POSIX 兼容的信号量。

（1）超时

信号量不可超时，超时可作为阻塞状态时另一种解决方法。提取信号量可限制在一段特定的时间内；若在该时间段内没有提取信号量，则意味着操作失败。

这种行为由 semTake( )的参数控制的；该参数为指定的 tick 时间数目，表示任务在阻塞状态等待的时间。若任务在分配时间内成功提取信号量，semTake( )返回 OK。当信号量规定的超时值已过，semTake( )返回 ERROR，系统设置 errno 值。

一个以 NO_WAIT(0)为参数的 semTake( )调用意味着系统不需要等待，若调用不可用，设置 errno 值为 S_objLib_OBJ_UNAVAILABLE。一个参数为正超时值的 semTake( )调用，若调用同样不可用，则返回 S_objLib_OBJ_TIMEOUT。一个以 WAIT_FOREVER(-1)为超时值的调用表示无限期的等待。

（2）队列

Wind 信号量能够对阻塞在信号量上的任务进行排队。它们基于先进先出顺序或者优先级顺序之一进行排队，请参考图 2-13。

优先级顺序更好地维护了系统优先级构造的意图，但却以调用 semTake( )进行优先级的排序为代价。一个先进先出队列则不需要排队开销，而且能保证恒定的时间性能。使用 semBCreate( )、semMCreate( )或 semCCreate( )建立信号量时确定队列类型。使用优先级继承选项（SEM_INVERSION_SAFE）的信号量必须选择优先级顺序队列。

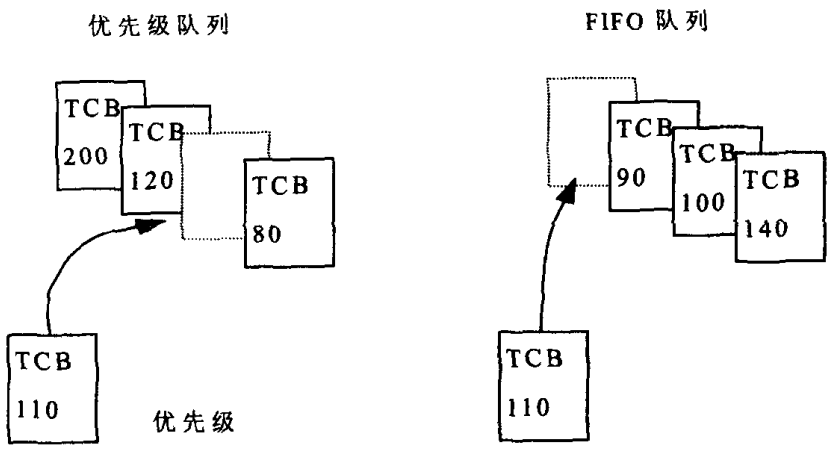


图 2-13　任务队列类型

6. 信号量和 VxWorks 事件

本节描述如何使用信号量处理 VxWorks 事件，当然也可使用 VxWorks 其他结构体处理 VxWorks 事件，详细信息请参考“2.4 VxWorks 事件”。

（1）使用事件

如果任务请求给其发送事件，那么可用信号量来给任务发送事件。为能获得信号量发送的事件，任务通过信号量调用 semEvStart( )来寄存；从那点开始，每次调用 semGive( ) 来释放信号量；只要没有其他任务阻塞在该信号量上，信号量将给已寄存的任务发送事件。调用 semEvStop( )，信号量停止发送事件。

任何时候每个信号量仅能寄存一个任务。使用 eventLib 库中的函数，能够恢复信号量给任务发送事件。对于信号量发送事件的详细信息，请参考 semEvStart( )。

在某些应用中，信号量建立者希望知道何时信号量发送事件将会失败。如果任务通过信号量寄存，且随后在取消任务寄存之前被删除，那么这种假想能够实现。在这种情况下，一个给定操作可能会导致信号量给已经被删除的任务发送事件。这种做法显然不适合。如果用 SEM_EVENTSEND_ERROR_NOTIFY 选项建立信号量，给定的操作将返回一个错误，否则 VxWorks 将自动处理错误。

使用 eventReceive( )函数，任务可能被信号量发送的事件阻塞。如果删除该信号量，与阻塞在信号量上的任务一样，阻塞在事件上的任务将返回到就绪状态。

（2）退出 VxWorks API

VxWorks 事件执行时不要求跟踪当前任务寄存的所有资源。因此，资源能够给不存在的任务发送事件。例如，一个任务可能被删除或可能进行了自析构操作，但仍然寄存在接收事件的资源内，这种错误只能在释放资源时发现，通过 semGive( )汇报返回 ERROR。然而在这种情况下，错误并不意味着没有释放信号量，或者信息没有被合理地传递；仅仅意味着资源不能给寄存的任务发送事件。这种行为与当前 VxWorks 操作系统里存在的行为不同；但与 pSOS 消息队列和信号量的行为相同。

（3）性能影响

当任务被信号量阻塞时，调用 semGive( )不产生性能影响。但不是该情况时（例如，信号量空闲），因为必须给任务发送事件，调用 semGive( )需要花费更长的时间。而且调用可能挂起等待事件的任务，即意味会出现抢占调用；甚至出现无任务等待信号量的情况。

任务等待从信号量发送事件时，因需唤醒任务，所以 semDestroy( )的性能会受到影响。注意，在该情况下不需要再发送事件。

2.3.4　消息队列

现代实时应用常常构造成一套独立但是相互合作的任务。对于同步和任务的互锁，信号量提供一种高速机制；但是为了允许合作任务间相互通信，经常需要一种更高级的机制来满足要求。在 VxWorks 操作系统里，单个 CPU 里任务间的主要通信方式时使用消息队列。（VxWorks 分布式消息队列组件在处理器和传输媒体间提供了共享的消息队列，请参考“第 10 章分布式消息队列”）

消息队列对允许数目可变、长度可变的消息排队。任务和中断服务程序能够发送消息给消息队列，同时任务还能从消息队列接收消息。

多个任务能够向同一个消息队列发送和接收消息。两个任务间的全双工通信一般需要两个消息队列，每个方向一个消息队列，请参考图 2-14。

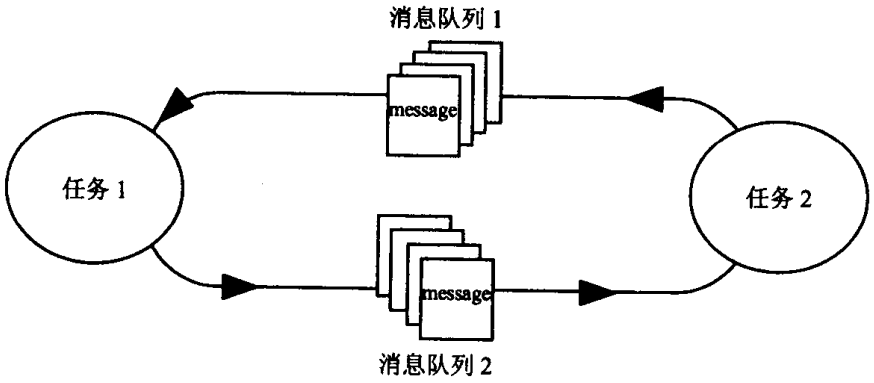


图 2-14　使用消息队列实现全双工通信

在 VxWorks 操作系统中有两个消息队列的子程序库。其中一个是 msgQLib，它提供专门用于设计的 Wind 消息队列；另一个是 mqPxLib，它与 POSIX 标准（1003.1b）的实时扩展兼容。对于两种消息队列设计上的差别，请参考“3.5.1 POSIX 和 Wind 调度方法比较”。

1. Wind 消息队列

Wind 消息队列建立、删除和使用的函数请参考表 2-14。该库提供了先入先出（FIFO）顺序排列的消息；但在有两个优先级时，高优先级的消息放在队列头部是一个例外。

表 2-14　Wind 消息队列控制

调　用	描　述
msgQCreate( )	分配并初始化一个消息队列
msgQDelete( )	终止并释放一个消息队列
msgQSend( )	向一个消息队列发送消息
msgQReceive( )	从一个消息队列接收消息

msgQCreate( )创建一个消息队列，其参数为消息队列中能够排列的最大消息数目以及每个消息的最大长度。根据指定的消息数目和长度，msgQCreate( )分配足够的缓冲空间。

调用 msgQSend( )，任务或中断服务程序把消息发送给一个消息队列。如果没有任务等待该队列中的消息，那么消息将进入消息队列缓冲器；如果已有任务在等待队列中的消息，那么消息将立刻传递给第一个等待的任务。

调用 msgQReceive( )，任务从消息队列中接收到一个消息。如果消息队列缓冲器中的消息可用，那么第一个可用的消息立刻被释放，并返回给调用者。如果没有可用消息，那么调用任务将被阻塞，并加入到等待消息的任务队列中。在创建队列的可选参数中，既可选择任务优先级顺序，也可选择先入先出顺序。

（1）超时

函数 msgQSend( )和 msgQReceive( )都可以设置超时参数。发送消息时，超时的含义指在没有可用空间进行消息排队时，从缓冲空间等待到其可用时所需的 tick 长度。在接收一个消息时，超时指从没有直接可用的消息等待到有可用消息所需的 tick 长度。和信号量一样，超时参数值可以为特殊值 NO_WAIT(0)，含义为立即返回；或者为 WAIT_FOREVER（-1），含义为程序永远等待。

（2）紧急消息

msgQSend( )允许指定下列消息的优先级为参数：正常（MSG_PRI_NORMAL），或紧急（MSG_PRI_URGENT）。正常优先级消息加入到消息队列尾部；紧急优先级消息加入到消息队列头部。

例 2-3：Wind 消息队列

```
/*本例中,任务 t1 创建消息队列,并给任务 t2 发送了消息.任务 t2 从队列中接收消息并简要地显
示消息. */

/* includes */
#include "vxWorks.h"
#include "msgQLib.h"

/* defines */
#define MAX_MSGS (10)
#define MAX_MSG_LEN (100)
```

```
MSG_Q_ID myMsgQId;

task2 (void)
    {
    char msgBuf[MAX_MSG_LEN];

    /* 从队列中获得消息,若需要将等待直至msg可用 */
    if (msgQReceive(myMsgQId, msgBuf, MAX_MSG_LEN, WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* 显示消息 */
    printf ("Message from task 1:\n%s\n", msgBuf);
    }

#define MESSAGE "Greetings from Task 1"
task1 (void)
    {
   /* 创建消息队列 */
    if ((myMsgQId = msgQCreate (MAX_MSGS, MAX_MSG_LEN, MSG_Q_PRIORITY))
  == NULL)
  return (ERROR);

  /* 若队列满(full),发送一个优先级消息 */
  if (msgQSend (myMsgQId, MESSAGE, sizeof (MESSAGE), WAIT_FOREVER,
                MSG_PRI_NORMAL) == ERROR)
      return (ERROR);
}
```

2. 消息队列显示属性

VxWorks 操作系统的 show( )命令是用来显示消息队列的重要属性，对于每种消息队列，例如，若 myMsgQId 是 Wind 消息队列，那么输出将传送到标准的输出设备，并显示如下形式：

```
-> show myMsgQId
Message Queue Id    : 0x3adaf0
Task Queuing        : FIFO
Message Byte Len    : 4
Messages Max        : 30
Messages Queued     : 14
Receivers Blocked   : 0
```

```
Send timeouts       : 0
Receive timeouts    : 0
```

3. 消息队列模式的服务器和客户机

实时系统经常构造成任务的客户机/服务器使用模式。在此模式里，服务器经常接收来自客户机的请求，执行一些任务并返回应答。这些请求和应答经常形成任务间的通信。在 VxWorks 操作系统里，消息队列或管道（请参考“2.3.5 管道”）是实现这种模式的常用方法。

例如，客户机/服务器通信可以实现为图 2-15 所示的形式。服务器任务建立一个消息队列，接收来自客户机的消息请求。客户机任务建立一个消息队列，接收从服务器返回的应答。每个请求消息包括一个含有客户机应答消息队列的 msgQId 域。服务器任务轮询它的请求消息队列，读取请求消息，执行请求，给客户机应答消息队列返回应答。

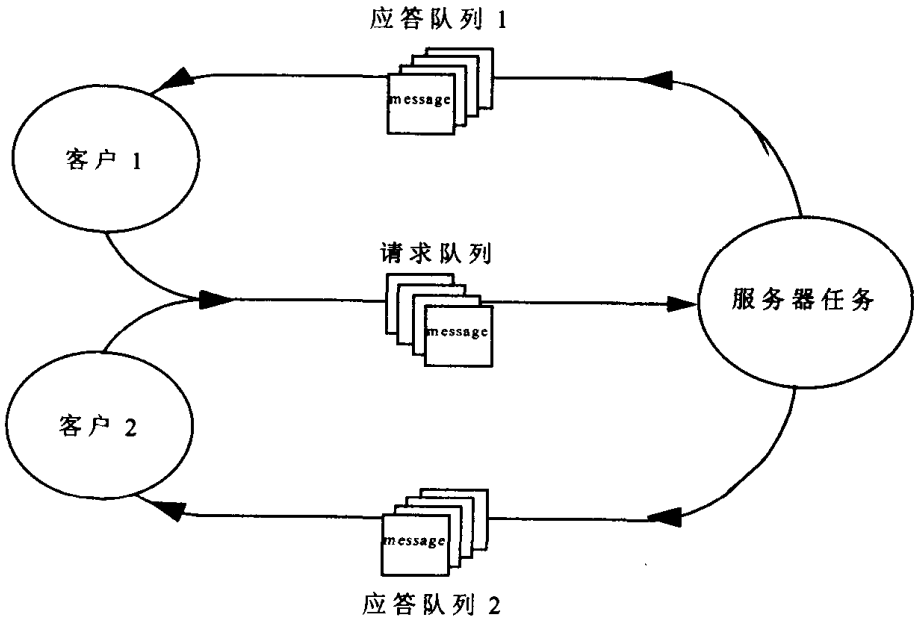


图 2-15　使用消息队列的客户机/服务器通信

用管道代替消息队列，或在特定应用中使用专用设计的结构体同样可以实现该功能。

4. 消息队列和 VxWorks 事件

本节描述了用消息队列处理 VxWorks 事件，当然也可使用 VxWorks 其他结构体处理 VxWorks 事件，详细信息请参考“2.4 VxWorks 事件”。

（1）使用事件

如果任务请求发送某个事件，消息队列可以给任务发送事件。为得到消息队列发送的事件，任务必须通过消息队列调用 msgQEvStart( )函数来寄存。从那点起，每次消息队列接收到消息且没有任务阻塞在其上时，消息队列给寄存的任务发送事件。调用 msgQEvStop( )函数，消息队列停止发送事件。

任何时候每个消息队列仅能寄存一个任务。调用 eventLib 库中的函数，能够恢复消息队列给任务发送事件。消息队列发送事件的详细信息请参考 msgQEvStart( ) 函数的相关条目。

某些应用中，消息队列建立者希望知道何时消息队列发送事件将会导致失败。如果任务通过消息队列寄存，且随后在取消任务寄存之前被删除，那么这种假想能够实现。在这种情况下，一个给定操作可能导致消息队列给已被删除的任务发送事件。这种做法显然不适合。如果用 SG_Q_EVENTSEND_ERROR_NOTIFY 选项建立消息队列，已给操作返回一个错误，否则 VxWorks 将自动处理错误。

调用 eventReceive( ) 函数，任务可能会被消息队列发送的事件所阻塞。如果删除消息队列，与阻塞在消息队列上的任务一样，阻塞在事件上的任务返回到就绪状态。

（2）退出 VxWorks API

VxWorks 事件的执行不要求跟踪当前任务寄存的所有资源。因此，资源能够给不存在的任务发送事件。例如，一个任务可能被删除或进行了自析构操作，但仍然寄存在接收事件的资源内；这种错误只能在释放资源时发现，通过 msgQSend( )汇报返回 ERROR。然而这种情况下，错误并不意味着没有释放消息队列，或没有合理地传递信息，仅意味着资源不能给寄存的任务发送事件。该行为与当前 VxWorks 操作系统里使用的行为不同，但与 pSOS 消息队列存在的行为相同。

（3）性能影响

任务被消息队列阻塞时，调用 msgQSend( )不产生性能影响。但若不是该情况，由于必须给任务发送事件，调用 msgQSend( )需要花费更长的时间。而且调用可能挂起等待事件的任务，即意味着会出现抢占调用；甚至出现无任务等待消息队列的情况。

任务等待消息队列发送事件时，因需唤醒任务而使函数 msgQDestroy( )性能受到影响。注意，该情况下无需发送事件。

2.3.5　管道

管道使用 VxWorks 操作系统中的 I/O 系统，并提供替换消息队列的接口。管道是由驱动程序 pipeDrv 管理的虚拟 I/O 设备。函数 pipeDevCreate( )创建管道设备以及与该管道相连的底层消息队列。调用时需要指定管道创建名称、能排列的消息最大数目及每个消息的最大长度等信息。

```
status = pipeDevCreate ("/pipe/name", max_msgs, max_length);
```

创建管道正常时是命名的 I/O 设备。任务能够使用标准 I/O 函数对管道进行打开、读取或写入等操作，另外也可以调用函数 ioctl。与其他 I/O 设备一样，除非管道有可用数据，从一个空管道读取数据时任务将被堵塞；同样，除非有可用的空间，任务向满管道进行写操作时也将被堵塞。与消息管道类似，中断服务程序能够向管道写入，但不能从管道读取。

作为 I/O 设备，管道提供了一个消息队列不具备的重要特性——使用函数 select( )。该函数允许任务等待一个 I/O 设备的数据直至其可用。该函数还能够与其他不同步的 I/O 设备一起工作，包括网络套接字和串行设备。因此使用 select( ) 函数，任务能够在几个管道、套接字和串行设备的结合体上等待数据。请参考“4.3.8 基于多文件描述符的挂起操作：选择功能”。

管道可以执行客户机/服务器模式的任务间通信，请参考消息队列模式的服务器和客户机。

2.3.6 任务间网络通信

1. 套接字

VxWorks 操作系统里，套接字是穿越网络的任务间通信的基本形式。套接字是任务间通信终端，数据从一个套接字传送到另一个套接字。在建立套接字时需指定数据传输的互联网通信协议。VxWorks 支持互联网的 TCP 协议和 UDP 协议。VxWorks 套接字功能与 BSD 4.4 UNIX 资源兼容。

TCP 使用的流套接字是可靠的受保护的双工数据通信。在一个流套接字通信中，两个套接字是“相连”的，这允许两个套接字间线路在每个方向上有可靠的字节流传输。所以，TCP 经常被用作虚拟电路协议。

UDP 提供了一个简单却更有效的通信方式。在 UDP 通信里，数据在套接字间以间隔的无连接的单个带地址的数据包形式传输。一个程序建立一个带地址数据包，并把地址绑定在一个特定的端口上。不存在 UDP“连接”的概念。在网络主机上，任何一个 UDP 套接字能够给标有互联网地址和端口的其他 UDP 套接字发送消息。

套接字通信最大优点就是它的同一性。如果不考虑互联网上程序地址或执行的操作系统，程序间套接字通信完全相同的。在单个 CPU 内程序能够穿越背板、以太网或任何网络上连接的结合体进行通信。套接字通信能够发生在 VxWorks 任务和主机系统程序间。在所有的情况下，除了速度外，套接字通信与应用程序类似。

详细信息，请参考《VxWorks 网络编程指南》：网络 APIs 和 sockLib 相关条目。

2. 远程程序调用（RPC）

远程程序调用（RPC）功能是允许一台机器上的某个程序调用另一个可执行程序，这里另一个可执行的程序既可在同台机器上，也可在远程机器上。RPC 内在地使用套接字作为底层的通信机制。因此使用 RPC 通信时，VxWorks 任务和主机系统程序可调用与之相连接的其他 VxWorks 操作系统或主机上的程序。

如上节在消息队列和管道中所讨论的，许多实时系统也构建成任务的客户机/服务器模式。在此模式里，客户机任务首先向服务器任务请求服务，然后等待服务器应答。RPC 规

范化了该形式，并提供了请求和返回应答的标准协议。另外，RPC 中还包括帮助产生客户机接口函数和服务器框架的工具。

RPC 详细信息，请参考《VxWorks 网络编程指南》：RPC（远程程序调用）。

2.3.7　信号

VxWorks 支持软件信号功能。信号可以异步改变任务的控制流程。任何任务或中断服务程序可以向指定任务发送信号。接收到信号的任务立即挂起当前的执行线程，在下次调度执行时转而执行指定的信号处理程序。信号处理程序在接收任务的上下文中执行，并使用任务的堆栈。即使在任务被阻塞时，仍可调用信号处理程序。

与通用任务间通信机制相比，信号机制更适合于错误和异常的处理。通常信号处理程序可作为中断处理程序看待，任何导致调用程序阻塞的函数均不能在信号处理程序中调用。由于信号是异步的，当发生特定的信号时很难预测哪个资源不可用。为安全起见，信号处理程序仅能调用那些在中断处理程序中安全使用的函数；仅在保信号处理程序不会出现死锁时，可以调用其他程序。

Wind 内核支持两种类型的信号接口：UNIX BSD 风格的信号和 POSIX 兼容信号。POSIX 兼容信号接口包括 POSIX 标准 1003.1 中指定的基本信号接口，以及从 POSIX 1003.1b 中扩展出来的队列信号接口。详细信息请参考“3.9　POSIX 队列信号”。为了简化设计，建议在一个应用程序中使用一种类型接口，不要混合使用不同接口。

关于信号的详细信息，请参考 sigLib 相关条目。

⚠ **注意**：VxWorks 操作系统中 sigLib 的执行不要求对下列信号操作强加任何特殊限制，这些信号为 SIGKILL，SIGCONT 和 SIGSTOP；但在 UNIX 中却有所限制，例如不能在 SIGKILL 和 SIGSTOP 中调用 signal( )。

1. 基本信号函数

VxWorks 默认地使用基本的信号组件 INCLUDE_SIGNALS。该组件使用 sigInit( ) 函数自动对信号进行初始化。表 2-15 列出了基本信号函数。

表 2-15　基本的信号调用（BSD and POSIX 1003.1b）

POSIX 1003.1b 兼容调用	UNIX BSD 兼容调用	描　述
signal( )	signal( )	指定信号的处理程序
kill( )	kill( )	向任务发送信号
raise( )	N/A	向自身发送信号
sigaction( )	sigvec( )	检查或设置信号的处理程序
sigsuspend( )	pause( )	挂起任务直至任务提交

续表

POSIX 1003.1b 兼容调用	UNIX BSD 兼容调用	描　述
sigpending( )	N/A	恢复一组用于传递而被阻塞的信号
sigemptyset( ) sigfillset( ) sigaddset( ) sigdelset( ) sigismember( )	sigsetmask( )	设置信号屏蔽
sigprocmask( )	sigsetmask( )	设置阻塞信号的屏蔽
sigprocmask( )	sigblock( )	增加到一组阻塞的信号中

函数名 kill( )起源于 UNIX BSD 中原始的接口。虽然这些接口不相同，但是 BSD 风格的信号和基本 POSIX 信号的功能却是类似的。

信号在很多方面类似于硬件中断。基本信号功能提供了 31 种不同的信号。调用 sigvec( ) 和 sigaction( )函数为特定的信号指定一个信号处理程序，这和调用 intConnect( ) 函数为中断指定一个中断处理程序类似。调用函数 kill( )给任务发送一个信号，这类似于发生中断。函数 sigsetmask( )和 sigblock( )或 sigprocmask( )可有选择地屏蔽信号。

信号发生通常与硬件中断相联系。例如总线出错、非法指令以及浮点数异常都可能产生某种信号。

2. 信号配置

在 VxWorks 操作系统中，基本信号功能默认包括使用 INCLUDE_SIGNALS 组件。

2.4　VxWorks 事件

在 VxWorks 5.5 中引进了类似于 pSOS 事件功能的 VxWorks 事件。VxWorks 事件包含在标准 VxWorks 功能中，并被用作向 VxWorks 操作系统提供 pSOS 事件端口。本节首先简要总结了 VxWorks 事件；接着详细描述了 pSOS 事件和 VxWorks 事件，并对两种事件及它们的应用编程接口（API）进行了比较。

⚠ **注意**：本节使用了术语事件（event）来描述 pSOS 和 VxWorks 事件；不要与 WindView 事件混淆。

VxWorks 事件是一种在任务和中断处理程序间，或任务和 VxWorks 结构体间的通信方式。在 VxWorks 事件上下文中，这些结构体被用作为资源，包括信号量和消息队列。只有任务能够接收事件；然而任务、中断处理程序或资源都可以发送事件。

任务必须使用资源寄存才能从资源接收事件。资源只有处于空闲状态时才能发送事件。任务和资源间是点到点的通信，这表明仅寄存的任务才能从资源接收事件。在这种情况下，由于事件是面向任务的，所以事件与信号类似。但是一个任务可以等待多个资源的事件，因此事件可以等待某个信号量直到其可用，也可以在一个消息队列中等待消息。

与信号不同，事件本身就是同步的，即在等待发生事件时，接收事件的任务必须处于堵塞或挂起状态。当任务接收到所需事件时，与调用函数 msgQReceive( )或 semTake( )一样，堵塞的任务继续执行。因此与信号不同，事件不需要处理程序。

任务也可以等待与资源无连接的事件。这些事件是指从另一个任务或中断服务程序发送的事件。接收这些事件的任务不需要寄存；发送事件的任务或中断服务程序仅需知道接收事件的相关信息。例如，类似于中断服务程序释放信号量，仅需知道存在任务需要获得该信号量。

对于不同的任务，每个事件的含义是不同的。例如，当事件 eventX 被接收时，它会被接收的每个任务解释成不同的含义。同时，只要任务接收到某个事件后，如果再给相同任务发送该事件，那么事件将会被忽略。因此，不可能对发送给某个任务的事件跟踪计数。

警告： 由于事件不能保存，两个独立的应用程序可能会在一个任务里使用相同的事件。作为一种预防，使用 VxWorks 事件的中间设备应公布一份使用清单。

2.4.1 pSOS 事件

本节描述了 pSOS 事件的功能。此功能提供基本的 VxWorks 事件，但没有详细描述其行为。详细信息请参考 VxWorks 对 pSOS 事件的扩展。

1. 发送和接收事件

在 pSOS 操作系统里，可以从资源给任务发送事件，从中断服务程序给任务发送事件，或直接在两个任务间发送事件。任务、中断服务程序以及资源都使用同一个应用编程接口——ev_send( )来发送事件。

对于从资源接收事件的任务来说，任务必须用资源寄存，而且请求资源在空闲时发送一系列指定的事件；这种资源可以是信号量，也可以是消息队列。资源在空闲时，给寄存的任务发送事件；而寄存的任务可能正在等待该事件，也可能不在等待该事件。

如上所述，任务也可从另一个任务接收事件。例如，若两任务同意在它们之间发送事件，当 taskA 完成执行时，可以给 taskB 发送一个指定事件，通知 taskB 其已经完成执行。与从资源发送的事件一样，接收的任务可能正在等待该事件，也可能不在等待该事件。

2. 等待事件

任务能够从一个或多个资源等待多个事件。每个资源可以发送多个事件，同样任务也

可以等待接收一个或多个事件。例如，任务正在等待事件 1－事件 10，这里事件 1－事件 4 来自于信号量，事件 5－事件 6 来自于消息队列，事件 7－事件 10 来自于另一任务。

一个任务在等待事件时可以指定一个超时值。

3. 事件的寄存

从资源接收事件时，资源只能寄存一个任务。如果另一个任务随后用同样的资源寄存，那么不会通知原先寄存的任务就自动解除了原有的寄存。这种行为与 VxWorks 事件不同，详细信息请参考 VxWorks 对 pSOS 事件的扩展。

当任务使用资源寄存时，即使任务寄存时资源空闲，也不会立即给任务发送事件。事件只在下次资源空闲时发送。例如，没有任务等待一个信号量，那么信号量将在其被释放后的下次发送事件。（被释放不意味着资源空闲，请参考“空闲资源”）。这种行为用于 VxWorks 事件配置的详细信息，请参考 VxWorks 对 pSOS 事件的扩展。

4. 空闲资源

当资源给任务发送事件表明空闲时，不意味着资源的空闲状态可以保留。因此，从资源等待事件的任务在资源空闲时被解除阻塞；但同时资源也可能被取走。如果任务随后想要占有资源，将不能保证资源仍然可用。

如上所述，一个资源仅在它空闲时发送事件。空闲和释放并不是同义的。为了澄清它们的区别，如果信号量被释放，它确实是空闲的。然而如果在释放时，另一个任务正在等待，那么将不是空闲的。因此，对于两个或两个以上的任务持续交换资源所有权的情况，资源不处于空闲状态，所以资源将不会发送事件。

5. pSOS 事件的应用编程接口

pSOS 事件的应用编程接口函数列于表 2-16 中。

表 2-16 pSOS 事件应用编程接口函数

函　数	描　述
ev_send( )	给任务发送事件
ev_receive( )	等待事件
sm_notify( )	寄存一个被信号量告知可用的任务
q_notify( )	寄存一个被消息队列告知有消息到来的任务
q_vnotify( )	寄存一个被可变长度的消息队列告知有消息到来的任务

2.4.2 VxWorks 事件

VxWorks 事件执行以 pSOS 事件为基石。本节首先描述了用于讨论 VxWorks 事件的重要术语，然后与 pSOS 功能进行比较，并详细描述了 VxWorks 事件。

1. 空闲资源定义

资源发送事件的一个重要概念是资源在空闲时发送事件。因此，对于 VxWorks 事件如何定义资源空闲是非常重要的。

（1）互斥信号量

当一个互斥信号量被释放并且在其上没有任务阻塞时，互斥信号量被认为是空闲的。例如，调用 semGive( )函数，如果另一个任务在调用该函数时因相同的信号量被堵塞，那么信号量不发送事件。

（2）二进制信号量

当没有任务占有或等待一个二进制信号量时是空闲的。

（3）计数器信号量

一个计数器信号量在其计数值非零且其上没有堵塞任务时是空闲的，因此事件不能作为一种计算释放信号量次数的机制。

（4）消息队列

队列中有消息存在，且没有等待该队列中消息而堵塞的任务，那么消息队列是空闲的，因此事件不能作为消息队列发送消息数目的计算机制。

2. VxWorks 对 pSOS 事件的扩展

执行 VxWorks 事件时，需要对基本的 pSOS 功能进行扩充。本节描述了那些扩充和配置选项，并比较了 VxWorks 事件和 pSOS 事件的结果行为特性。

（1）单任务资源寄存

如同“2.4.1 pSOS 事件”中所涉及的，在 pSOS 系统中一个任务用资源寄存发送 pSOS 事件时，它会无意地取消另一个已用该资源寄存的任务寄存；这就阻止了第一个任务从寄存资源中接收事件。因此，第一个用该资源寄存的任务将无限期地被阻塞。

为了解决这一问题，VxWorks 事件提供了一个选项，在该选项中如果另一个任务已经用某个资源寄存了，则不允许第二个任务用该资源再寄存。如果第二个任务用该资源寄存，将返回一个错误。在 VxWorks 操作系统里，可以使用 VxWorks 或 pSOS 行为来配置寄存机制。

（2）立即发送选项

如同在“事件寄存”中所涉及的，当一个 pSOS 任务用资源寄存时，即使寄存时资源处于空闲状态，也不会立即给任务发送事件。对于 VxWorks 事件，默认行为与之相同。然而，VxWorks 事件提供了一个选项，即若该资源在寄存时处于空闲状态，该选项允许任务请求资源立即给其发送事件。

（3）自动取消寄存选项

有时存在下列情形：任务仅需从资源接收一次事件，然后取消寄存。pSOS 执行过程需要任务在从资源接收任务后明确地取消寄存。VxWorks 执行提供一个选项，该选项可以通

知资源仅发送一次事件，然后在发送后自动取消寄存。

（4）自动解除资源堵塞

当删除资源（一个信号量或者消息队列）时，调用函数 semDelete( ) 和 msgQDelete( ) 解除所有任务的挂起。在任务等待被删除资源发送事件时，该措施保护任务避免无限期的堵塞。然后任务继续执行，导致任务挂起的函数 eventReceive( )返回一个 ERROR 值。另外，可参考“退出 VxWorks 应用编程接口”。

3. 任务事件寄存器

每个任务有各自的事件域或空间，即指任务事件寄存器。任务事件寄存器是一个用作存储事件的 32 位任务域，通常这些事件来自于资源、中断服务程序或其他任务。

任务事件寄存器不可以直接进行访问。任务、中断服务程序或资源通过给相应任务发送事件来填充任务事件寄存器。一个任务能够给自身发送事件，填充的是自身的事件寄存器。事件 25 到 32（VXEV25 或 0x01000000 到 VXEV32 或 0x80000000）用作系统保留用，VxWorks 用户不可以使用这些事件。表 2-17 描述了影响事件寄存器内容的函数。

表 2-17　事件寄存器使用函数

函　数	描　述
eventReceive( )	通过选择选项，清除或者保留事件寄存器的内容
eventClear( )	清除事件寄存器的内容
eventSend( )	复制事件到事件寄存器中
semGive( )	使用信号量寄存任务时，复制事件到事件寄存器中
msgQSend( )	使用消息队列寄存任务时，复制事件到事件寄存器中

4. VxWorks 事件应用编程接口

VxWorks 事件应用编程接口的详细信息，请参考 eventLib，semEvLib 和 msgQEvLib。

Show 函数

为了调试使用事件的系统，库 taskShow，semShow 和 msgQShow 分别显示不同的事件信息。

库 taskShow 显示下列信息：

事件寄存器的内容

必要事件

调用 eventReceive( )时的函数指定选项

库 semShow( )和 msgQShow( ) 显示下列信息：

用于接收事件的任务寄存

资源发送到相应任务的事件

函数 semEvStart( )或 msgQEvStart( )的传输选项

2.4.3　API 比较

为全面地描述函数行为，VxWorks 事件 API 修改了 pSOS 事件 API。从资源进行寄存或取消寄存，pSOS API 使用了一套通告（notify）的函数。但这些函数名没有准确反映资源的行为，它们要么发送事件，要么不发送事件。因此 VxWorks API 更准确地描述了通知资源发送事件或停止发送事件的请求。这种行为通过调用函数 semEvStart( ) 和 msgQEvStart( )来启动资源发送事件，调用 semEvStop( )和 msgQEvStop( )来停止发送事件。

表 2-18 比较了 VxWorks 和 pSOS 事件 API 的相似和不同之处。

表 2-18　事件比较

VxWorks 函数	pSOS 函数	注　释
eventSend	ev_send	直接端口
eventReceive	ev_receive	直接端口
eventClear		VxWorks 中的新功能
semEvStart	sm_notify	SemEvStart 等价于用非零事件参数调用 sm_notify
semEvStop	sm_notify	SemEvStop 等价于用事件参数为 0 调用 sm_notify
msgQEvStart	q_vnotify	msgQEvStart 等价于用非零事件参数调用 q_notify
msgQEvStop	q_vnotify	msgQEvStop 等价于用事件参数为 0 调用 q_notify
	q_notify	VxWorks 没有一个固定长度的消息队列机制

2.5　看门狗定时器

VxWorks 包括一个看门狗定时器机制，它允许任何 C 函数与一个特定的时间延时器联系。看门狗定时器作为系统时钟中断服务程序的一部分来维护。详细的 POSIX 定时器信息，请参考“3.2 POSIX 时钟和计时器”。

被看门狗定时器调用的函数通常作为系统时钟中断级的中断服务代码来执行。但如果内核由于某种原因不能立即执行函数（例如一个优先中断或者内核状态），函数将放在 tExcTask 工作队列中。tExcTask 工作队列中的函数以 tExcTask（通常是 0）优先级来执行。

中断服务程序的限制适用于看门狗定时器使用的相关函数。表 2-19 中函数由库 wdLib 提供。

表 2-19　看门狗定时器函数调用

调　用	描　述
wdCreate( )	分配并初始化一个看门狗定时器
wdDelete( )	终止并释放一个看门狗定时器

续表

调　用	描　述
wdStart( )	启动一个看门狗定时器
wdCancel( )	取消当前的一个计数的看门狗定时器

看门狗定时器由 wdCreate( )函数创建，由 wdStart( ) 函数启动，函数参数为：延迟 tick 数、需要调用的 C 程序以及传递给该函数的一个参数。一旦指定 tick 数延时结束，函数将使用指定的参数进行调用。在定时器结束计时之前的任何时间内，调用 wdCancel( ) 函数取消看门狗定时器的执行。

例 2-4：看门狗定时器

```
/* 创建一个看门狗定时器,并设置它在 3 秒内关闭.*/

/* includes 语句 */
#include "vxWorks.h"
#include "logLib.h"
#include "wdLib.h"

/* defines 语句 */
#define  SECONDS (3)

WDOG_ID myWatchDogId;
task (void)
    {
    /* 创建看门狗定时器 */
    if ((myWatchDogId = wdCreate( )) == NULL)
        return (ERROR);

    /* 设置关闭定时器的事件(单位:秒),并给 stdout 打印信息*/
    if (wdStart (myWatchDogId, sysClkRateGet( ) * SECONDS, logMsg,
                 "Watchdog timer just expired\n") == ERROR)
        return (ERROR);
    /* ... */
    }
```

2.6 中断服务代码：中断服务程序

硬件中断处理是实时系统中最重要的部分，因为系统经常通过中断与外部事件相互交

互。为了尽快地响应中断，VxWorks 中断处理程序（中断服务程序）在所有任务上下文之外的一个特殊上下文内执行。因此，中断处理不涉及到任务上下文的切换。表 2-20 列出了库 intLib 和 intArchLib 中提供的中断处理函数。

表 2-20　中断处理函数

调　用	描　述
intConnect( )	设置中断处理的 C 程序
intContext( )	如果是从中断级调用，返回真
intCount( )	获得当前中断嵌套深度
intLevelSet( )	设置处理器的中断屏蔽级
intLock( )	禁止中断
intUnlock( )	重新允许中断
intVecBaseSet( )	设置向量基地址
intVecBaseGet( )	得到向量基地址
intVecSet( )	设置异常向量
intVecGet( )	获得异常向量

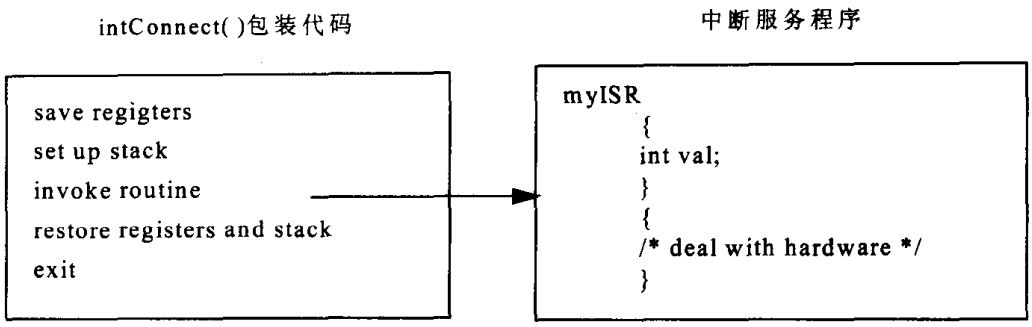
对于使用 MMU 的底板，可选产品 VxVMI 为中断向量表提供了写保护。请参考“第 12 章　虚拟内存接口”。

2.6.1　中断处理连接程序

系统硬件中断处理程序可替代 VxWorks 中断程序。VxWorks 提供 intConnect( )函数允许 C 函数与任何中断处理相联系；该函数参数为：与之相连的中断矢量的字节偏移量、连接到 C 函数的地址并传递给该函数的一个参数。当使用这种方法建立的位矢量发生中断时，连接的 C 函数在中断级用指定参数进行调用。中断处理完成后连接 C 函数返回。用该方法连接到中断的函数就是中断处理程序。

中断实际上不能够直接与 C 函数相联系。但函数 intConnect( )不仅建立了少量存储在寄存器中的代码，而且用被传输的参数建立了一个堆栈入口（或一个特殊的中断堆栈，或当前的任务堆栈），另外还能调用连接函数。从调用函数返回时，intConnect( ) 函数恢复寄存器和堆栈，并退出中断，请参考图 2-16。

对于使用 VME 底板的目标底板，BSP 提供了两个标准函数用于控制 VME 总线中断，即函数 sysIntEnable( )和 sysIntDisable( )。



intConnect (INUM_TO_IVEC (someIntNum), myISR, someVal);

图 2-16　使用 intConnect( ) 建立函数

2.6.2　中断堆栈

所有中断服务程序使用相同的中断堆栈。该堆栈由系统启动时根据指定配置参数来定位和初始化。为能够处理最坏情况下的中断嵌套，必须分配足够大的中断堆栈空间。

但是，一些体系结构不允许使用分离的中断堆栈。在这种体系结构里，中断服务程序使用中断任务的堆栈。如果遇到这样的体系结构，必须建立一个任务拥有充足堆栈的空间，用来处理最坏情况下嵌套中断和正常嵌套调用。用户请参考自身使用的 BSP，从而确定使用的体系结构是否支持不分离的中断堆栈。

开发过程中，使用 checkStack( )函数可以观察如何关闭中断堆栈以及中断服务程序如何逐渐地占用堆栈空间。

2.6.3　编写和调试中断服务程序

调用中断服务程序函数存在着很多的限制。例如，在应用中断服务程序时不能使用 printf( )，malloc( )和 semTake( )函数，但是可以使用 semGive( )，logMsg( )，msgQSend( ) 和 bcopy( )这样的函数。详细信息请参考“2.6.4 中断服务程序特殊限制”。

2.6.4　中断服务程序的特殊限制

许多 VxWorks 函数在中断服务程序中使用时仍存在许多重要限制。这些限制是由于中断服务程序不在一个固定的任务上下文中执行，而且没有任务控制块，因此所有中断服务程序必须共享一个单独的堆栈。

由于上述原因，中断服务程序基本限制为禁止调用导致调用者堵塞的函数。例如，禁止试图使用一个信号量，因为只要信号量不可用，内核将把调用者切换到挂起状态。但是

中断服务程序能够释放信号量，以及等待信号量的任务。

由于存储器函数 malloc( )和 free( )都要求获得信号量，中断服务程序不能调用 malloc( )和 free( )，也不能调用其他调用 malloc( )和 free( )的函数。例如，中断服务程序不能调用任何用于创建或删除的函数。

中断服务程序也不能通过 VxWorks 驱动程序来执行 I/O 操作。虽然在 I/O 系统里没有内在约束，但是，由于大多数的设备驱动器可能会堵塞等待设备的调用者，因此它们需要一个任务上下文。VxWorks 管道驱动器是非常重要的设备，它设计用于中断服务程序的写操作，但这是一个例外。

VxWorks 提供了一个记录功能，允许向系统任务平台打印文本信息。这个机制是专门为中断服务程序使用而设计的，同时它也是从中断服务程序打印信息的最常用方法。详细信息请参考 logLib 相关条目。

中断服务程序同时禁止调用浮点协处理器函数。在 VxWorks 操作系统里，由 intConnect( ) 函数建立的中断驱动代码不能保存和恢复浮点寄存器。因此中断服务程序不能包含浮点指令。若中断服务程序需要使用浮点指令，则必须明确地保存和恢复 fppArchLib 中函数浮点协处理器的寄存器。

所有 VxWorks 函数库，像连接链和环形缓冲库，都可以被中断服务程序使用。如前所述（“2.2.6 任务的错误状态：errno”），全局变量 errno 被保存并恢复成中断进入和退出代码的一部分，这些进入和退出代码是由 intConnect( ) 函数实现的。因此 errno 能够被中断服务程序访问并修改成其他的代码。表 2-21 列出了能够从中断服务程序中被调用的函数。

表 2-21　能被中断服务程序调用的函数

库	函　数
bLib	所有函数
errnoLib	errnoGet( )，errnoSet( )
fppArchLib	fppSave( )，fppRestore( )
intLib	intContext( )，intCount( )，intVecSet( )，intVecGet( )
IntArchLib	intLock( )，intUnlock( )
LogLib	logMsg( )
LstLib	除 lstFree( )外的所有函数
mathALib	如果使用 fppSave( )/fppRestore( )，所有函数均可
msgQLib	msgQSend( )
pipeDrv	write( )
rngLib	除 rngCreate( )和 rngDelete( )外的所有函数
selectLib	selWakeup( )，selWakeupAll( )
semLib	semFlush( )，除了使用互斥信号量的 semGive( )
sigLib	kill( )
taskLib	taskSuspend( )，taskResume( )，taskPrioritySet( )，taskPriorityGet( )，taskIdVerify( )，taskIdDefault( )，task 中断服务程序 ready( )，taskIsSuspended( )，taskTcb( )

续表

库	函　数
tickLib	tickAnnounce( )，tickSet( )，tickGet( )
tyLib	tyIRd( )，tyITx( )
vxLib	vxTas( )，vxMemProbe( )
wdLib	wdStart( )，wdCancel( )

2.6.5　中断级异常

当任务导致硬件异常，例如一个非法指令或总线发生错误时，任务将被挂起，但系统其他部分继续工作。当中断服务程序导致这类异常时，由于中断服务程序没有挂起的上下文，系统将没有安全可靠的资源用来处理异常。VxWorks 在低端内存的某个特定区域存储了这种异常的描述，并执行系统重启作为该异常处理的替代方法。

VxWorks 启动程序在低端内存测试上述描述异常是否存在，如果发现则将在系统的控制平台上显示出来：在启动 ROM 中的 e 命令将重新显示该异常的描述。请参考《Tornado 用户指南》：建立和重新启动。

一则上述异常实例信息显示如下：

```
workQPanic: Kernel work queue overflow
```

在中断级，内核以极高的频率调用时，经常会发生这种异常。这种异常通常与中断信号的清除或者一个类似的驱动问题有关。

2.6.6　保留高中断级

在本节前面所述的 VxWorks 中断的支持对绝大多数应用是可接收的。但是，有时对于临界运动控制等需要低级控制的事件，或系统失败响应等情况，需要保留最高中断级来确保这些事件的零延时。为了获得零延时响应，VxWorks 提供了 intLockLevelSet( )函数来设置中断互锁级别。如果不指明级别，那么默认值为处理器支持的最高级别。对于特定系统，intLockLevelSet( )执行的详细信息，请参考合适的 *VxWorks architecture supplement*。

警告：为防止屏蔽硬件中断级，请核查硬件生产商的文献。

2.6.7　在高中断级上中断服务程序的附加限制

中断服务程序与不能上锁的中断级（或比 intLockLevelSet( )设置更高的中断级，或由硬件设置的非屏蔽中断级）相连接时有特殊限制：

- 仅 intVecSet( )函数能连接中断服务程序。
- 中断服务程序不能调用任何依赖于中断上锁的 VxWorks 操作系统函数。除了重新启动外，有效的办法为中断服务程序不调用任何与之相关的 VxWorks 函数。

对于结构体的详细信息，请参考 VxWorks architecture supplement 文献。

警告：除了重新启动功能需 NMI 以外，不推荐在 VxWorks 函数中使用任何 NMI。标有“中断安全”的函数并不表明使用 NMI 是安全的；实际上，这些函数经常是 NMI 不能够调用的函数（因为它们都是使用 intLock( ) 函数来获得中断安全条件的典型函数）。

2.6.8　中断与任务通信

虽然 VxWorks 支持与中断级执行的中断服务程序直接相连，但中断事件经常涉及到任务级代码。许多 VxWorks 设备不能使用中断级代码，包括从 I/O 到除了管道外的任何设备。从中断服务程序到任务级的通信可使用如下技术：

1. 共享内存和环缓冲器

使用任务级代码，中断服务程序能够共享变量、缓冲器和环缓冲器。

2. 信号量

中断服务程序可以释放任务提取和等待时的信号量（除了互斥信号量和 VxMP 共享信号量外）。

3. 消息队列

中断服务程序能够给消息队列发送任务接收的消息（除了使用 VxMP 的共享消息队列）。如果队列为满（full），消息将被丢弃。

4. 管道

中断服务程序能够向管道写入任务读取的消息。任务和中断服务程序能够向相同的管道写入消息。但是，如果管道为满（full），因中断服务程序不允许堵塞，从而写入的消息将被丢弃。中断服务程序禁止在管道上调用除了 write( )外的任何 I/O 函数。

5. 信号

中断服务程序给任务发送信号，从而异步调度信号处理程序。

第 3 章　POSIX 标准接口

3.1　简　　介

对于实时系统扩展（1003.1b）的 POSIX 标准为内核设备提供了一套接口。为提高应用的可移植性，VxWorks 内核“Wind”包括了 POSIX 接口以及专门用于 VxWorks 设计的接口。

本章使用限定词“Wind”来表示为 VxWorks 操作系统 Wind 内核设计的设备。例如，在“3.6.1　POSIX 和 Wind 信号量比较”中 POSIX 信号信和 Wind 信号量的比较中讨论了 Wind 信号量和 POSIX 信号量的差别。

在 aioPxLib 库中可以使用 POSIX 异步输入/输出（AIO）函数。VxWorks AIO 函数执行满足 POSIX 1003.1b 标准。详细信息，请参见“4.6 异步输入/输出操作”。

3.2　POSIX 时钟和计时器

时钟是一种以秒或纳秒为单位来记录时间的软件结构(结构 timespec 定义在 time.h 中)。软件时钟由系统时钟 tick 更新。VxWorks 提供了 POSIX 1003.1b 标准时钟和计时器接口。

POSIX 标准提供了一种可识别多个虚拟时钟的方法，但只有系统的实时时钟是必需的。VxWorks 系统中不支持虚拟时钟。

在时钟和计时器函数中，系统的实时时钟在 time.h 中定义为 CLOCK_REALTIME。VxWorks 提供了访问系统实时时钟的函数，详细信息请参见 clockLib 相关条目。

POSIX 计时器提供了在未来某时刻任务对其自身发送信号的函数，这些函数用于创建、设置和删除一个计时器。详细信息请参见 timerLib 相关条目。当计时器执行完成后，向任务发送默认信号 SIGALRM。如果要安装一个在计时器结束后执行的信号处理程序，可以调用函数 sigaction( )（请参见“2.3.7 信号”)。

例 3-1：POSIX 计时器

```
/* 本例创建了一个新的定时器,并将其存储在 timerid 中。*/

/* includes */
```

```
#include "vxWorks.h"
#include "time.h"

int createTimer ( void)
    {
    timer_t timerid;

    /* 创建定时器 */
    if ( timer_create ( CLOCK_REALTIME, NULL, &timerid) == ERROR)
        {
        printf ( "create FAILED\n");
        return ( ERROR);
        }
    return ( OK);
    }
```

nanosleep( )作为 POSIX 的一个附加功能，与 Wind 中 taskDelay( )类似，用来指定一个以秒或纳秒为单位的睡眠或延迟时间，但 taskDelay( )单位为 tick。不过两者精度相同，都由系统时钟速率决定，只是单位不同而已。

3.3　POSIX 内存上锁接口

许多操作系统执行内存分页和交换功能，这些技术通常将内存块数据复制到磁盘，或从磁盘复制回内存，同时允许使用比物理内存更大的虚拟内存。但是，由于执行时存在严重的、不可预测的延时，实时系统一般不使用分页和交换功能，因此 Wind 内核从不使用它们。

但是，实时扩展的 POSIX 1003.1b 标准使用了执行分页和交换功能的操作系统。在该系统里，执行实时的应用能够使用 POSIX 锁页功能（*page-locking*）来阻止某些内存模块被分页和交换。

为实现其他符合 POSIX 标准的系统和 VxWorks 系统间程序的可移植性，VxWorks 使用了一个 POSIX 锁页函数；因为内存通常是上锁的，所以该函数在 VxWorks 系统中不会产生不利的影响。

列于表 3-1 中的 POSIX 锁页函数是内存管理库 mmanPxLib 的一部分。在 VxWorks 中使用时，因为所有页都保存在内存中，所以这些函数仅返回一个值 OK(0)。

表 3-1　POSIX 内存管理函数调用

调　用	在系统中使用分页和交换的作用
mlockall( )	锁住一个任务使用的所有内存页
munlockall( )	解锁一个任务使用的所有内存页
mlock( )	锁住一个指定的内存页
munlock( )	解锁一个指定的内存页

用 INCLUDE_POSIX_MEM 组件配置的 VxWorks 系统包含 mmanPxLib 库。

3.4　POSIX 线程

POSIX 线程与任务相似，但有一些附加的特性，包括区别于任务 ID 的线程 ID。

3.4.1　POSIX 线程属性

POSIX 的特性称为属性。每个属性包含一组值，以及一组恢复和设置这些值的访问函数。在一个属性对象 pthread_attr_t 内，创建线程时可以指定所有的线程属性。一些情况下可以动态地在运行线程中修改属性值。

POSIX 属性和对应的访问函数描述如下。

1. 堆栈大小 stacksize

stacksize 属性指定了使用堆栈的大小，堆栈尺寸值在每页分界线上可四舍五入。

属性名：stacksize

默认值：使用为 taskLib 设置的默认堆栈尺寸

访问函数：pthread_attr_getstacksize( )和 pthread_attr_setstacksize( )

2. 堆栈地址 stackaddr

stackaddr 属性指定了一个分配给用户的内存区域作为线程的堆栈区域。由于默认值为 NULL，在创建线程时系统为线程分配一个堆栈。

属性名：stackaddr

默认值：NULL

访问函数：pthread_attr_getstackaddr( )和 pthread_attr_setstackaddr( )

3. Detach 状态

Detachstate 属性描述了线程的状态。使用 POSIX 线程，在线程退出前，线程的创建程

序不会被阻塞（请参见《*VxWorks API 参考*》中的 pthread_exit( )和 pthread_join( )），在这种情况下，创建的线程是可连接的；否则就是一个分离的线程。调用 pthread_detach( )把可连接的线程动态转化为分离的线程。

属性名：detachstate

可能值：PTHREAD_CREATE_DETACHED 和 PTHREAD_CREATE_JOINABLE。

默认值：PTHREAD_CREATE_JOINABLE

访问函数：pthread_attr_getdetachstate( )和 pthread_attr_setdetachstate( )

动态访问函数：pthread_detach( )

4. 竞争域 Contentionscope

Contentionscope 属性描述了线程如何竞争使用资源（如 CPU）。在 VxWorks 系统里，所有任务都竞争使用 CPU，所以，竞争在系统内是广泛的。虽然 POSIX 允许使用两个值，但是只有 PTHREAD_SCOPE_SYSTEM 适用于 VxWorks 系统。

属性名：contentionscope

可能值：只有 PTHREAD_SCOPE_SYSTEM（PTHREAD_SCOPE_PROCESS 不适用于 VxWorks 系统）

默认值：PTHREAD_SCOPE_SYSTEM

访问函数：pthread_attr_getscope( )和 pthread_attr_setscope( )

5. 继承调度 inheritsched

inheritsched 属性决定线程创建时是使用继承父线程的调度参数，还是使用明确指定的参数。

属性名：inheritsched

可能值：PTHREAD_EXPLICIT_SCHED 或 PTHREAD_INHERIT_SCHED

默认值：PTHREAD_INHERIT_SCHED

访问函数：pthread_attr_getinheritsched( )和 pthread_attr_setinheritsched( )

6. 调度策略 Schedpolicy

Schedpolicy 属性描述了线程的调度策略，仅在 inheritsched 属性值为 PTHREAD_EXPLICIT_SCHED 时，该调度策略有效。

属性值：schedpolicy

可能值：SCHED_FIFO（基于优先级的抢占调度）和 SCHED_RR（基于优先级的轮转调度）

默认值：SCHED_RR

访问函数：pthread_attr_getschedpolicy( )和 pthread_attr_setschedpolicy( )

由于 inheritsched 属性的默认值为 PTHREAD_INHERIT_SCHED，将不再默认使用

schedpolicy 属性。详细信息，请参见“3.5.3 获得并显示当前调度策略”。

7. 调度参数 Schedparam

Schedparam 属性描述了线程的调度参数，仅在 inheritsched 属性值为 PTHREAD_EXPLICIT_SCHED 时，该调度参数有效。

属性名：schedparam

值的范围：0~255

默认值：使用为 taskLib 设置的默认任务优先级

访问函数：pthread_attr_getschedparam( )和 pthread_attr_setschedparam( )

动态访问函数：pthread_getschedparam( )和使用线程 ID 的 pthread_setschedparam( )；或者 sched_getparam( )和使用任务 ID 的 sched_setparam( )

由于 inheritsched 属性默认值为 PTHREAD_INHERIT_SCHED，将不再默认地使用 schedparam 属性。详细信息，请参见“3.5.2 获得和设置 POSIX 任务优先级”。

8. 建立 pthread 时指定属性

下面是使用默认属性和指定属性建立线程的几个例子：

例 3-2：使用指定调度属性建立 pthread

```
pthread_t tid;
pthread_attr_t attr;
int ret;
pthread_attr_init( &attr);

/* 用指定属性设置 inheritsched */
pthread_attr_setinheritsched( &attr, PTHREAD_EXPLICIT_SCHED);

/*设置 schedpolicy 属性值 SCHED_FIFO */
pthread_attr_setschedpolicy( &attr, SCHED_FIFO);

/*创建线程*/
ret = pthread_create( &tid, &attr, entryFunction, entryArg);
```

例 3-3：使用默认属性建立 pthread

```
pthread_t tid;
int ret;

/* 用 NULL 属性指派默认值来创建 pthread */
ret = pthread_create( &tid, NULL, entryFunction, entryArg);
```

例 3-4：为 pthread 指派用户堆栈

```
pthread_attr_init( &attr);

/* 为线程分配堆栈内存空间*/
stackbase = malloc( 2 * 4096);

if ( stackbase == NULL)
    {
    printf( "FAILED: mystack: malloc failed\n");
    exit( -1);
    }

/* 给基地址设置堆栈指针*/
stackptr = ( void *)(( int)stackbase);

/*设置 stackaddr 属性*/
pthread_attr_setstackaddr( &attr, stackptr);

/* 设置 stacksize 为 4096 */
pthread_attr_setstacksize( &attr, ( 4096));
/* 设置 schedpolicy 属性为 SCHED_FIFO */
pthread_attr_setschedpolicy( &attr, SCHED_FIFO);

/*创建 pthread */
ret = pthread_create( &tid, &attr, mystack_thread, 0);
```

3.4.2　线程私用数据

当线程访问私用数据时，POSIX 使用了 key 来访问数据。一块访问区域可以通过调用 pthread_key_create( )建立，并通过调用 pthread_key_delete( ) 释放。建立一块区域后，可调用 pthread_getspecific( )和 pthread_setspecific( )进行访问。其中 pthread_key_create( )有一个析构功能 destructor function 的选项，当已建线程退出时，如果 key 对应的值为非零，则调用该函数。

3.4.3　线程取消

POSIX 提供了一种取消机制 cancellation，该机制合理地终止线程；取消机制存在两种类型：同步和异步。同步取消机制通过线程外在的检查，确定线程是直接取消的，还是调

用一个包含取消点的函数来实现的。异步取消机制终止线程的执行，调用一个类似于信号[①]的处理程序。

使用取消机制时，可调用的函数列于表 3-2 中。

表 3-2　取消线程函数

函　数	含　义
pthread_setcancelstate( )	实现或禁止取消操作
pthread_setcanceltype( )	选择同步还是异步取消
pthread_cleanup_push( )	在取消线程时寄存被调用的函数
pthread_cleanup_pop( )	在取消线程时对被调用的函数解除寄存，然后继续调用函数

当通过 pthread_cleanup_push( )和 pthread_cleanup_pop( )取消线程时，线程可以寄存被调用函数，或取消其寄存。pthread_cleanup_pop( )在取消寄存时可以有选择地调用函数。

3.5　POSIX 调度接口

列于表 3-3 中的 POSIX 1003.1b 调度函数由 schedPxLib 提供。这些函数可以使用一个可移植的接口，从而获得和设置任务的优先级，获得调度策略以及任务最大和最小的优先级；如果轮转调度有效，还将获得时间片的长度。本节描述了如何使用这些函数，首先从 POSIX 和 Wind 调度方法的一系列次要方面进行比较。

表 3-3　POSIX 调度函数调用

调　度	描　述
sched_setparam( )	设置任务的优先级
sched_getparam( )	获得指定任务的调度参数
sched_setscheduler( )	设置任务的调度策略和参数
sched_yield( )	放弃 CPU
sched_getscheduler( )	获得当前的调度策略
sched_get_priority_max( )	获得最大的优先级
sched_get_priority_min( )	获得最小的优先级
sched_rr_get_interval( )	如果轮转调度，获得时间片长度

用 INCLUDE_POSIX_SCHED 组件配置的 VxWorks 系统，包含 POSIX 调度函数库 schedPxLib。

[①] 异步取消实际上是由一个特殊信号——SIGCANCEL 实现的；对于该信号，用户应小心使用以免阻塞。

3.5.1　POSIX 和 Wind 调度方法比较

POSIX 和 Wind 调度函数的区别如下：

（1）POSIX 调度基于进程，而 Wind 调度基于任务。

（2）POSIX 标准使用了 FIFO 调度术语，VxWorks 文档使用了基于优先级的抢占式调度。两者仅表达不同，都使用了相同的基于优先级的策略。

（3）POSIX 在进程到进程的基础上应用调度算法；Wind 的调度算法应用于整个系统，即所有任务既可使用轮转调度，也可使用基于优先级的抢占调度。

（4）POSIX 的优先级编号方法与 Wind 相反。在 POSIX 中编号越大，其优先级也越高，但是在 Wind 中编号越小，其优先级越高，0 编号是最高的优先级。相应的，使用 POSIX 调度库 schedPxLib 与其他所有 VxWorks 组件使用的优先级不匹配。用户可以不使用默认值，设置全局变量 posixPriorityNumbering 为 FALSE 从而解决这一问题。如果使用该设置，schedPxLib 使用 Wind 的编号策略（小编号对应高优先级），并且它的优先级编号与 VxWorks 其他组件使用的 Wind 优先级相匹配。

3.5.2　获得和设置 POSIX 任务优先级

sched_setparam( )和 sched_getparam( )分别设置和获得一个任务的优先级。两个函数使用都需要获得一个任务 ID 和 sched_param 结构（定义在安装目录/target/h/sched.h 下）。一个 ID 为 0 的任务可以设置或者获取调用任务的优先级。

在调用 sched_setparam( )时，该函数结构的成员 sched_priority 需要指定新任务的优先级。sched_getparam( )把指定任务的当前优先级填充到 sched_priority 中。

例 3-5：获得和设置一个 POSIX 任务的优先级

```
/* 本例设置了调用任务优先级为 150,并核实了该优先级。为了能从壳( shell)运行,以任务形
式: -> sp priorityTest 发起。 */

/* includes */
#include "vxWorks.h"
#include "sched.h"

/* defines */
#define PX_NEW_PRIORITY 150

STATUS priorityTest ( void)
    {
```

```
struct sched_param myParam;

/* 给需求的优先级初始化参数结构 */

myParam.sched_priority = PX_NEW_PRIORITY;
if ( sched_setparam ( 0, &myParam) == ERROR)
    {
    printf ( "error setting priority\n");
    return ( ERROR);
    }

/* 通过获得的任务优先级来核实预先设置的优先级,确保它们有相同优先级值。 */

if ( sched_getparam ( 0, &myParam) == ERROR)
    {
    printf ( "error getting priority\n");
    return ( ERROR);
    }

if ( myParam.sched_priority != PX_NEW_PRIORITY)
    {
    printf ( "error - priorities do not match\n");
    return ( ERROR);
    }
else
    printf ( "task priority = %d\n", myParam.sched_priority);

return ( OK);
}
```

函数 sched_setscheduler( )用于设置单个 POSIX 进程的调度策略和优先级，在大多数情况下它与单个 Wind 任务通信。在 VxWorks 内核中，由于内核禁止使用不同的任务调度策略，所以该函数仅控制任务的优先级。若任务调度策略与当前系统调度策略一致，则与函数 sched_setparam( )行为类似，sched_setscheduler( )仅设置优先级；否则 sched_setscheduler( )返回一个错误值。

改变调度策略的惟一方法就是改变所有任务的策略；不过没有提供实现这一想法的函数调用。为设置系统的调度策略，可以使用轮转调度中描述的 Wind 函数——kernelTimeSlice( )。

3.5.3　获得并显示当前调度策略

POSIX 函数 sched_getscheduler( )返回当前调度策略。在 VxWorks 系统中有两个有效的调度策略：基于优先级的抢占式调度（POSIX 术语为 SCHED_FIFO）和基于优先级的轮转调度（POSIX 术语为 SCHED_RR）。详细信息请参见调度策略。

例 3-6：获得 POSIX 调度策略

```
/* 本例获得调度策略,并显示该策略. */

/* includes */

#include "vxWorks.h"
#include "sched.h"

STATUS schedulerTest ( void)
    {
    int policy;

    if (( policy = sched_getscheduler ( 0)) == ERROR)
        {
        printf ( "getting scheduler failed\n");
        return ( ERROR);
        }

    /* sched_getscheduler 返回 SCHED_FIFO 或 SCHED_RR */

    if ( policy == SCHED_FIFO)
        printf ( "current scheduling policy is FIFO\n");
    else
        printf ( "current scheduling policy is round robin\n");

    return ( OK);
    }
```

3.5.4　获得调度参数：优先级限制和时间片

函数 sched_get_priority_max( )和 sched_get_priority_min( )分别返回最大和最小可能的

POSIX 优先级。

若能够使用轮转调度，调用函数 sched_rr_get_interval( )可以指定当前的时间片长度。该函数参数为 timespec 结构（定义在 time.h 中）中的指针，并向该结构的对应元素写入每个时间片的秒数或纳秒数。

例 3-7：获得 POSIX 轮转时间片

```
/* 下例检查了轮转调度可以使用,获得时间片长度,并显示时间片*/

/* includes */

#include "vxWorks.h"
#include "sched.h"

STATUS rrgetintervalTest ( void)
    {
    struct timespec slice;

    /* turn on round robin */

    kernelTimeSlice ( 30);

    if ( sched_rr_get_interval ( 0, &slice) == ERROR)
        {
        printf ( "get-interval test failed\n");
        return ( ERROR);
        }

    printf ( "time slice is %l seconds and %l nanoseconds\n",
            slice.tv_sec, slice.tv_nsec);
    return ( OK);
    }
```

3.6 POSIX 信号量

POSIX 定义了命名和未命名的信号量，虽然两者性质相同，但是在接口使用上却有着轻微的不同。对于命名和未命名的信号量，POSIX 信号量库提供了用于创建、打开和破坏

的函数。在打开一个命名信号量时，为其指定一个符号名[②]，作为其他命名信号量接受的参数。semPxLib 提供的 POSIX 信号量函数列于表 3-4 中。

表 3-4　POSIX 信号量函数

调　用	描　述
semPxLibInit( )	初始化 POSIX 的信号量库（非 POSIX）
sem_init( )	初始化一个未命名信号量
sem_destroy( )	破坏一个未命名信号量
sem_open( )	初始化/打开一个命名信号量
sem_close( )	关闭一个命名信号量
sem_unlink( )	移去一个命名信号量
sem_wait( )	对一个信号量上锁
sem_trywait( )	仅当它未上锁时，锁上一个 POSIX 信号量
sem_post( )	对信号量解锁
sem_getvalue( )	获得信号量的值

用 INCLUDE_POSIX_SEM 组件配置的 VxWorks 系统，包含 POSIX 库 semPxLib 的信号量函数。VxWorks 系统中包含 POSIX 信号量时，默认地调用初始化函数 semPxLibInit( )。

3.6.1　POSIX 和 Wind 信号量比较

POSIX 信号量属于计数信号量，即它们会跟踪信号量被释放的次数。除下列 Wind 信号量提供的附加特征外，Wind 信号量机制与 POSIX 中信号量机制类似；而且下列特性起重要作用时，Wind 信号量性能更加优越。

- 优先级继承
- 任务删除安全性
- 对于单任务多次获取信号量的能力
- 互斥信号量的所有权
- 信号量超时
- 队列机制选择

POSIX 术语等待（或上锁）和通信（或解锁）分别对应于 VxWorks 中术语提取和释放。用于上锁、解锁和获得信号量值的 POSIX 函数适用于命名和未命名信号量。

sem_init( )和 sem_destroy( )仅用于对未命名信号量的初始化和删除。调用 sem_destroy( )终止一个未命名信号量，并释放所有相关内存。

[②] 一些主机操作系统，例如 UNIX，在进程中共享对象需要符号名；这是因为在此类操作系统中进程通常不共享内存。在 VxWorks 系统中，不需要命名信号量，因为所有内核对象都有惟一标识符。但是，使用 POSIX 类型的命名信号量提供了一种决定对象 ID 的简便方法。

sem_open( )，sem_unlink( )和 sem_close( )仅用于打开和关闭命名信号量。sem_close( )和 sem_unlink( ) 联合作用于命名信号量，与 sem_destroy( )作用于未命名信号量时的效果相同，即终止信号量，并释放相关内存。

警告： 删除信号量时，尤其是删除互斥信号量，应防止删除其他任务仍需使用的该信号量。除非任务删除时已经成功地锁住信号量，否则不要删除信号量。类似于命名信号量，仅由打开它们的任务关闭这些信号量。

3.6.2 未命名信号量使用

在使用未命名信号量时，任务通常会对信号量进行内存分配和初始化。信号量使用定义在 semaphore.h 中的数据结构 sem_t 表示。信号量初始化函数 sem_init( )允许指定初始值。

对信号量初始化后，任务能够使用该信号量；调用 sem_wait( )（阻塞时）或 sem_trywait( )（非阻塞时）对信号量上锁，而 sem_post( )用于信号量的解锁。

信号量既可用于同步，又可用于互斥。因此当信号量用于同步时，信号量值被初始化为零（上锁状态）；调用 sem_wait( )时等待同步的任务被阻塞。调用 sem_post( )，执行同步的任务解除阻塞。如果阻塞在信号量上的任务是惟一等待的任务，那么任务将被解除阻塞，进入就绪状态执行。若还有其他任务阻塞在该信号量上，拥有最高优先级的任务将首先解除阻塞状态。

信号量用于互斥时，它被初始化为大于零的值，即表明资源是可用的。因此，第一个对信号量上锁的任务将不会被阻塞；但当信号量值初始化为 1 时，任务进入阻塞状态。

例 3-8：POSIX 未命名信号量

```
/* 本例使用未命名信号量实现调用任务和它发起的任务( tSyncTask)之间的同步。为从 shell
运行,作为任务 -> sp unnameSem 发起 */

/* includes */

#include "vxWorks.h"
#include "semaphore.h"

/* 正向说明( forward declaration) */
void syncTask ( sem_t * pSem);

void unnameSem ( void)
    {
    sem_t * pSem;
```

```
    /* reserve memory for semaphore */
    pSem = ( sem_t *) malloc ( sizeof ( sem_t));

    /* 初始化信号量为不可用 */
    if ( sem_init ( pSem, 0, 0) == -1)
        {
        printf ( "unnameSem: sem_init failed\n");
        free (( char *) pSem);
        return;
        }

    /* 创建同步任务 */
    printf ( "unnameSem: spawning task…\n");
    taskSpawn ( "tSyncTask", 90, 0, 2000, syncTask, pSem);

    /* 为与 syncTask 同步做准备 */
    /* 对 sem 解锁 */
    printf ( "unnameSem: posting semaphore - synchronizing action\n");
    if ( sem_post ( pSem) == -1)
        {
        printf ( "unnameSem: posting semaphore failed\n");
        sem_destroy ( pSem);
        free (( char *) pSem);
        return;
        }
    /* 完成执行 - 破坏信号量 */
    if ( sem_destroy ( pSem) == -1)
      {
      printf ( "unnameSem: sem_destroy failed\n");
      return;
      }
    free (( char *) pSem);
    }

void syncTask
    (
    sem_t * pSem
    )
    {
    /* 从 unnameSem 等待同步 */
```

```
if ( sem_wait ( pSem) == -1)
    {
    printf ( "syncTask: sem_wait failed \n");
    return;
    }
else
    printf ( "syncTask:sem locked; doing sync'ed action…\n");

/* 处理执行结果*/
}
```

3.6.3 命名信号量的使用

函数 sem_open( )既可打开已有信号量，也可建立新的信号量。组合使用下列标志，可以创建一个需要的信号量。

1. O_CREAT

若信号量不存在，建立该信号量。（若信号量存在，成功打开与否依赖于是否使用了 O_EXCL。）

2. O_EXCL

信号量仅在初建时可以打开；若信号量已经存在，则打开失败。

根据标志设置以及访问的信号量是否存在，调用 sem_open( )的结果列于表 3-5 中。由于单独使用标志 O_EXCL 无意义，所以没有将其单独列出。

表 3-5　sem_open( )调用结果

标 志 设 置	若信号量存在	若信号量不存在
无	打开信号量	函数调用失败
O_CREAT	打开信号量	建立信号量
O_CREAT 和 O_EXCL	函数调用失败	建立信号量

一旦初始化，POSIX 命名信号量后将一直处于可用状态，直至该信号量被破坏。任务能够标示将要被破坏的信号量，除非所有任务都关闭了该信号量，否则信号量将一直保存在系统里。

若 VxWorks 配置包含 INCLUDE_POSIX_SEM_SHOW 组件，可从壳（shell）中调用 show( )来显示 POSIX 信号量消息[③]。

[③] 这不是一个 POSIX 函数，它也不是设计应用于编程的，而是应用于 Tornado 壳（shell）（请参见《Tornado 用户指南》：shell 详细信息。）

```
-> show semId
value = 0 = 0x0
```

输出结果传输到标准输出设备，并使用两个等待信号量的被阻塞任务提供 POSIX 信号量 mySem 的信息。

```
Semaphore name          :mySem
sem_open( ) count        :3
Semaphore value         :0
No. of blocked tasks    :2
```

当一组合作任务使用同一个命名信号量时，其中一个任务首先调用 sem_open( )建立并初始化信号量，其中标志需设置为 O_CREAT；其后任何任务使用该信号量时，可使用相同的信号量名调用 sem_open( )打开它（但不需要再设置 O_CREAT）。任何已经打开该信号量的任务可以调用 sem_wait( )（阻塞状态）或 sem_trywait( )（非阻塞状态）上锁，或调用 sem_post( )解锁，从而使用该信号量。

为了删除信号量，所有使用信号量的任务必须调用 sem_close( )关闭它；其中必须有一个任务解除与该信号量的连接。调用 sem_unlink( )解除连接，将从信号量命名管理表中删除该信号量名。删除信号量名后，当前打开该信号量的任务仍可使用它，但不允许新任务打开该信号量。下次某个任务试图打开该信号量时若不设置标志 O_CREAT，那么操作将会失败。在最后一个任务关闭该信号量时，该信号量消失。

例 3-9：POSIX 命名信号量

```
/*
 * 本例中,nameSem( )创建了一个用于同步的任务,新任务 tSyncSemTask 阻塞在 nameSem( )
中创建的信号量上。
 *一旦同步发生,两个任务都关闭信号量,
 * nameSem( )解除与信号量的连接。为了从 shell 运行任务,以任务形式 -> sp nameSem,
"myTest" 发起。 */
/* includes */
#include "vxWorks.h"
#include "semaphore.h"
#include "fcntl.h"

/* 正向说明( forward declaration) */
int syncSemTask ( char * name);

int nameSem
    (
```

```
char * name
)
{
sem_t * semId;

/* 创建一个命名信号量,并初始化为 0*/
printf ( "nameSem: creating semaphore\n");
if (( semId = sem_open ( name, O_CREAT, 0, 0)) == ( sem_t *) -1)
    {
    printf ( "nameSem: sem_open failed\n");
    return;
    }

printf ( "nameSem: spawning sync task\n");
taskSpawn ( "tSyncSemTask", 90, 0, 2000, syncSemTask, name);

/* 为与 syncSemTask 同步做准备*/

/* 释放信号量 */
printf ( "nameSem: posting semaphore - synchronizing action\n");
if ( sem_post ( semId) == -1)
    {
    printf ( "nameSem: sem_post failed\n");
    return;
    }

/* 运行结束 */
if ( sem_close ( semId) == -1)
    {
    printf ( "nameSem: sem_close failed\n");
    return;
    }

if ( sem_unlink ( name) == -1)
    {
    printf ( "nameSem: sem_unlink failed\n");
    return;
    }

printf ( "nameSem: closed and unlinked semaphore\n");
```

```
    }

int syncSemTask
    (
    char * name
    )
    {
    sem_t * semId;

    /* 打开信号量 */
    printf ( "syncSemTask: opening semaphore\n");
    if (( semId = sem_open ( name, 0)) == ( sem_t *) -1)
        {
        printf ( "syncSemTask: sem_open failed\n");
        return;
        }

    /* 阻塞,并等待从 nameSem 实现同步*/
    printf ( "syncSemTask: attempting to take semaphore...\n");
    if ( sem_wait ( semId) == -1)
        {
        printf ( "syncSemTask: taking sem failed\n");
        return;
        }

    printf ( "syncSemTask: has semaphore, doing sync'ed action ...\n");

    /* 结果输出 */

    if ( sem_close ( semId) == -1)
        {
        printf ( "syncSemTask: sem_close failed\n");
        return;
       }
    }
```

3.7 POSIX互斥体（Mutexes）和条件变量

应用POSIX标准（1003.1c）时互斥体和条件变量相互兼容。它们本质上与互斥和二进

制信号量（实际执行时）有着相同的功能；互斥体和条件变量与 pthreadLib 是同时可用的。类似于 POSIX 线程，互斥体和条件变量有相关的属性。

Mutex 属性存储在一个 pthread_mutexattr_t 数据类型中，该数据类型包含两个属性：协议（protocol）和优先级单元（prioceiling）。

1. 协议（protocol）

Protocol 互斥体属性描述了互斥体如何处理互斥信号量的优先级倒置问题。

属性名：protocol

可能值：PTHREAD_PRIO_INHERIT 和 PTHREAD_PRIO_PROTECT

访问函数：pthread_mutexattr_getprotocol( )和 pthread_mutexattr_setprotocol( )

调用 SEM_Q_PRIORITY 和 SEM_PRIO_INHERIT 选项下 semMCreate( )，可以建立一个继承优先级的互斥信号量。用优先级保护值创建的互斥信号量涉及到优先级单元的概念，这是互斥体的另一个属性。

2. 优先级单元（prioceiling）

prioceiling 属性是在 protocol 属性设置为 PTHREAD_PRIO_PROTECT 时建立的互斥体 POSIX 优先级单元。

属性名：prioceiling

可能值：任何有效的（POSIX）优先级值

访问函数：pthread_mutexattr_getprioceiling( )和 pthread_mutexattr_setprioceiling( )

动态访问函数：pthread_mutex_getprioceiling( )和 pthread_mutex_setprioceiling( )

⚠ **注意**：POSIX 优先级编号方法与 Wind 优先级编号相反。请参见“3.5.1 POSIX 和 Wind 调度方法比较”。

一个优先级单元通过下列条件定义：

- 任何需要获得 mutex 的线程，若优先级比单元优先级高，则不能获得该 mutex。
- 任何优先级低于单元优先级的线程，在占有 mutex 期间，可把其优先级提高到单元优先级。
- 一旦线程释放 mutex 时，其优先级将恢复到原值。

3.8 POSIX 消息队列

由 mqPxLib 提供的 POSIX 消息队列函数列于表 3-6 中。

表 3-6　POSIX 消息队列函数

调　用	描　述
mqPxLibInit( )	初始化 POSIX 消息队列库（非 POSIX）
mq_open( )	打开一个消息队列
mq_close( )	关闭一个消息队列
mq_unlink( )	删除一个消息队列
mq_send( )	给消息队列发送消息
mq_receive( )	从消息队列获得消息
mq_notify( )	给任务发送信号表明消息队列里有消息在等待
mq_setattr( )	设置消息队列属性
mq_getattr( )	获得消息队列属性

为配置 VxWorks 系统使其包含 POSIX 消息队列函数，需要包括 INCLUDE_POSIX_MQ 组件。初始化函数 mqPxLibInit( ) 使 POSIX 消息队列函数可用，当系统包含 INCLUDE_POSIX_MQ 组件时该初始化函数会自动被调用。

3.8.1　POSIX 和 Wind 消息队列比较

除了 POSIX 消息队列提供具有优先级的消息外，POSIX 消息队列与 Wind 消息队列非常相似。两者的差别总结在表 3-7 中。

表 3-7　消息队列的特征比较

特　征	Wind 消息队列	POSIX 消息队列
消息优先级水平	1	32
阻塞任务队列	基于优先级，或 FIFO	基于优先级
超时接受	可选择	不可用
任务通知	不可用	可选择（单任务）
关闭/接触连接语义	否	是

若 POSIX 消息队列是从另一个 1003.1b 兼容系统中移植到 VxWorks 系统中的，则它同样是可移植的。这就意味着使用 POSIX 消息队列时不用改变代码，从而也减少了端口效应。

3.8.2　POSIX 消息队列属性

一个 POSIX 消息队列有如下属性：

- 一个可选的 O_NONBLOCK 标志

- 消息队列中最大的消息数目
- 最大的消息尺寸
- 当前队列中消息数目

任务调用 mq_setattr( )能够设置或清除 O_NONBLOCK 标志（但该标志不属于其他的属性）；调用 mq_getattr( )可获得所有属性值。

例 3-10：设置和获得消息队列属性

```
/* 本例设置了 O_NONBLOCK 标志,并检查消息队列的属性*/

/* includes */
#include "vxWorks.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"

/* defines */
#define MSG_SIZE    16

int attrEx
    (
    char * name
    )
    {
    mqd_t           mqPXId;             /* mq 描述符*/
    struct mq_attr  attr;               /* 队列属性结构*/
    struct mq_attr  oldAttr;            /* 旧队列的属性*/
    char            buffer[MSG_SIZE];
    int             prio;

    /*创建阻塞状态的读写队列*/
    attr.mq_flags = 0;
    attr.mq_maxmsg = 1;
    attr.mq_msgsize = 16;
    if (( mqPXId = mq_open ( name, O_CREAT | O_RDWR , 0, &attr))
        == ( mqd_t) -1)
        return ( ERROR);
    else
        printf ( "mq_open with non-block succeeded\n");

    /*在队列上改变属性 - 启用 non_blocking*/
    attr.mq_flags = O_NONBLOCK;
    if ( mq_setattr ( mqPXId, &attr, &oldAttr) == -1)
```

```
        return ( ERROR);
    else
        {
        /* 附加检查 - oldAttr 不应该包含 non_blocking. */
        if ( oldAttr.mq_flags & O_NONBLOCK)
            return ( ERROR);
        else
            printf ( "mq_setattr turning on non-blocking succeeded\n");
        }

    /*接收消息 - 不存在消息,但接收消息时不应该被阻塞*/
    if ( mq_receive ( mqPXId, buffer, MSG_SIZE, &prio) == -1)
        {
        if ( errno != EAGAIN)
            return ( ERROR);
        else
            printf ( "mq_receive with non-blocking didn't block on empty
queue\n");
        }
    else
        return ( ERROR);

    /* 使用 mq_getattr 检查调用成功与否 */
    if ( mq_getattr ( mqPXId, &oldAttr) == -1)
        return ( ERROR);
    else
        {
        /* 测试可能得到的值*/
        if ( !( oldAttr.mq_flags & O_NONBLOCK) || ( oldAttr.mq_curmsgs !=
0))
            return ( ERROR);
        else
            printf ( "queue attributes are:\n\tblocking is %s\n\t
                    message size is: %d\n\t
                    max messages in queue: %d\n\t
                    no. of current msgs in queue: %d\n",
                    oldAttr.mq_flags & O_NONBLOCK ? "on" : "off",
                    oldAttr.mq_msgsize, oldAttr.mq_maxmsg,
                    oldAttr.mq_curmsgs);
        }

    /* 清除 - 关闭并解除 mq 连接*/
```

```
    if ( mq_unlink ( name) == -1)
        return ( ERROR);
    if ( mq_close ( mqPXId) == -1)
        return ( ERROR);
    return ( OK);
    }
```

3.8.3 显示消息队列属性

对于 POSIX 或者 Wind 消息队列，VxWorks 中 show( )将显示关键的消息队列属性。为了获得 POSIX 消息队列信息，配置 VxWorks 系统时需包括 INCLUDE_POSIX_MQ_SHOW 组件。

例如，若 mqPXId 是一个 POSIX 消息队列：

```
-> show mqPXId
value = 0 = 0x0
```

其输出结果将发送到标准输出设备，并显示如下：

```
Message queue name          : MyQueue
No. of messages in queue    : 1
Maximum no. of messages     : 16
Maximum message size        : 16
```

当 myMsgQId 是一个 Wind 消息队列时，其输出结果如下：

```
-> show myMsgQId
Message Queue Id    : 0x3adaf0
Task Queuing        : FIFO
Message Byte Len    : 4
Messages Max        : 30
Messages Queued     : 14
Receivers Blocked   : 0
Send timeouts       : 0
Receive timeouts    : 0
```

⚠ **注意**：内置 show( )函数处理 Wind 消息队列，请参见《Tornado 用户指南》：内置函数的 Shell 信息。也可以使用 Tornado 浏览器来获得 Wind 消息队列信息，请参见《Tornado 用户指南》：浏览器详细信息。

3.8.4　用消息队列通信

在一组任务利用 POSIX 消息队列通信之前，其中一个任务必须调用设置标志为 O_CREAT 的 mq_open( )建立消息队列。建立消息队列后，其他任务使用创建的队列名可以打开该消息队列，进而在该队列上发送或接收消息。仅在第一个任务用 O_CREAT 标志打开队列后，其他任务才能打开该队列进行下列操作：接收消息（O_RDONLY），或发送消息（O_WRONLY），或同时发送接收消息（O_RDWR）。

调用 mq_send( )把消息放入队列中。如果某个任务在队列满状态时把消息放入队列中，那么该任务进入阻塞状态，直到有其他任务从队列中读取消息后才能解除阻塞，在队列中放入消息。为了防止调用 mq_send( )时阻塞，打开消息队列时需要设置 O_NONBLOCK。这样，即使队列处于满状态，也不会阻塞执行的任务,而是由 mq_send( )返回-1 值，并设置 errno 为 EAGAIN，并允许重新进行调用，或使用其他的方法。

消息优先级为 mq_send( )的一个参数；优先级范围从 0（最低优先级）到 31（最高优先级）；详细信息请参见“3.5.1　POSIX 和 Wind 调度方法比较”。

当任务调用 mq_receive( )来接收消息时，任务接收的是当前队列中具有最高优先级的消息。对于具有相同优先级的多个消息，队列中的第一个消息被首先接收（FIFO 顺序）。若队列状态为空时，任务将处于阻塞直到队列中放入新的消息。

为了避免调用 mq_receive( )时阻塞，打开消息队列需使用 O_NONBLOCK。在这种情况下，当任务需要从空状态的队列中读取消息时，mq_receive( )将返回-1 值，设置 errno 为 EAGAIN。

调用 mq_close( )关闭一个消息队列。关闭不会破坏消息队列，但确定任务以后不再使用该队列。调用函数 mq_unlink( )破坏一个队列。解除消息队列的连接不会立即破坏队列，但会从队列名管理表中移走该队列名，而且任何任务将不能够再次打开它。当前打开队列的任务可以继续使用；当最后一个任务关闭一个无连接队列时，队列被破坏。

例 3-11：POSIX 消息队列

```
/* 本例中 mqExInit( )函数发起两个使用消息队列进行通信的任务 */

/* mqEx.h - 消息头文件*/

/* defines */
#define MQ_NAME "example Message Queue"

/* 正向说明( forward declaration) */
void receiveTask ( void);
void sendTask ( void);
```

```
/* testMQ.c - 使用 POSIX 消息队列的实例*/

/* includes */
#include "vxWorks.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"
#include "mqEx.h"

/* defines */
#define HI_PRIO     31
#define MSG_SIZE    16

int mqExInit ( void)
    {
    /* create two tasks */
    if ( taskSpawn ( "tRcvTask", 95, 0, 4000, receiveTask, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0) == ERROR)
        {
        printf ( "taskSpawn of tRcvTask failed\n");
        return ( ERROR);
        }

    if ( taskSpawn ( "tSndTask", 100, 0, 4000, sendTask, 0, 0, 0, 0,
                0, 0, 0, 0, 0, 0) == ERROR)
        {
        printf ( "taskSpawn of tSendTask failed\n");
        return ( ERROR);
        }
    }

void receiveTask ( void)
    {
    mqd_t    mqPXId;          /* msg queue descriptor */
    char     msg[MSG_SIZE];   /* msg buffer */
    int      prio;            /* priority of message */

    /* 打开使用默认属性的消息队列*/
    if (( mqPXId = mq_open ( MQ_NAME, O_RDWR | O_CREAT, 0, NULL))
        == ( mqd_t) -1)
        {
        printf ( "receiveTask: mq_open failed\n");
```

```
        return;
        }

    /*从队列中读消息*/
    if ( mq_receive ( mqPXId, msg, MSG_SIZE, &prio) == -1)
        {
        printf ( "receiveTask: mq_receive failed\n");
        return;
        }
    else
        {
        printf ( "receiveTask: Msg of priority %d received:\n\t\t%s\n",
                prio, msg);
        }
    }
/* sendTask.c - mq 发送的实例 */
/* includes */
#include "vxWorks.h"
#include "mqueue.h"
#include "fcntl.h"
#include "mqEx.h"

/* defines */
#define MSG   "greetings"
#define HI_PRIO 30

void sendTask ( void)
    {
    mqd_t    mqPXId;          /* msg 队列描述符 */

    /* 打开 msg 队列；应该已经存在使用默认属性的队列*/

    if (( mqPXId = mq_open ( MQ_NAME, O_RDWR, 0, NULL)) == ( mqd_t) -1)
        {
        printf ( "sendTask: mq_open failed\n");
        return;
        }

    /*向队列写操作*/
    if ( mq_send ( mqPXId, MSG, sizeof ( MSG), HI_PRIO) == -1)
        {
        printf ( "sendTask: mq_send failed\n");
```

```
        return;
        }
    else
        printf ( "sendTask: mq_send succeeded\n");
    }
```

3.8.5 通知任务有消息在等待

任务可以调用 mq_notify( )，向空状态的队列发出请求，在队列有消息到来时进行通告。这种方法的优点在于能够避免任务阻塞，或者可以轮询等待消息。

空状态队列中放入消息时，调用 mq_notify( )给任务发送一个指定信号。这种机制使用 POSIX 发送带有其他数据信号的方式，例如，允许信号携带一个队列标识符（请参见 3.9 POSIX 队列信号）。

设计 mq_notify( )机制的目的在于当新消息可用时通知任务。若消息队列已有可用消息，在更多消息到来时不再发送通知信号。若存在另一个调用 mq_receive( )而阻塞的任务，而同时其他的任务没有阻塞，那么调用 mq_notify( ) 也不会给寄存的任务发送通知。

通知只适用于单个任务：每个队列一次仅寄存一个未来要通知的任务。一旦队列需要给某个任务发通知时，直到通知请求被许可或者被取消，调用 mq_notify( )将不能对任务进行寄存。

一旦队列给任务发送了通知，就满足了通知请求，而且队列与该任务也就没有了特殊的联系；即对于每个 mq_notify( )的请求队列仅发送一次通知信号。为了让该任务继续接受通知信号，最好的途径就是在接收通知信号的同一个信号处理程序中调用 mq_notify( )，从而尽可能重新发出通知请求。

若指定 NULL 代替通知信号，将取消通知请求；只有当前寄存的任务能够取消它自己的通知请求。

例 3-12：通知任务，队列中有消息在等待

```
/*  本例中任务通过 mq_notify( )发现一个的空队列中何时有等待的消息。  */
/* includes */
#include "vxWorks.h"
#include "signal.h"
#include "mqueue.h"
#include "fcntl.h"
#include "errno.h"
/* defines */
#define QNAM        "PxQ1"
#define MSG_SIZE     64          /* 限制消息的尺寸 */
```

```
/* 正向说明( forward declarations) */
static void exNotificationHandle ( int, siginfo_t *, void *);
static void exMqRead ( mqd_t);
/*
 * exMqNotify -使用 mq_notify( )的实例
 *
 * 该函数表明了如何使用 mq_notify( )获得通知
 * 通过队列中新消息发送信号。为简化实例,使用了单一的任务来发送和接收消息。*/
int exMqNotify
    (
    char * pMess                /* 返回给自己的消息内容*/
    )
    {
    struct mq_attr    attr;                 /* 队列属性结构*/
    struct sigevent   sigNotify;            /* 通告*/
    struct sigaction  mySigAction;          /* 信号处理程序 */
    mqd_t             exMqId                /* 消息队列 id */
    /* 降低完整检查的复杂性,避免溢出 msg 缓冲器尺寸 */
    if ( MSG_SIZE <= strlen ( pMess))
        {
        printf ( "exMqNotify: message too long\n");
        return ( -1);
        }
    /*
     * 安装信号处理程序产生通过信号,填入到 sigaction 数据结构中,并传递给
     * sigaction( )。由于处理程序需要 siginfo 数据结构作参数,
     * SA_SIGINFO 标志被设置为 sa_flags。 */
   mySigAction.sa_sigaction = exNotificationHandle;
   mySigAction.sa_flags     = SA_SIGINFO;
   sigemptyset ( &mySigAction.sa_mask);
   if ( sigaction ( SIGUSR1, &mySigAction, NULL) == -1)
        {
        printf ( "sigaction failed\n");
        return ( -1);
        }
   /*
    * 创建一个消息队列- 把所需的消息尺寸和 no.号填入到 mq_attr structure 中 ,
    * 并传递给 mq_open( )。*/
   attr.mq_flags  = O_NONBLOCK;             /* 设置非阻塞 */
   attr.mq_maxmsg = 2;
   attr.mq_msgsize = MSG_SIZE;
```

```
    if (( exMqId = mq_open ( QNAM, O_CREAT | O_RDWR, 0, &attr)) ==
         ( mqd_t) - 1 )
        {
        printf ( "mq_open failed\n");
        return ( -1);
        }
    /*
     * 建立通告的信号：填入一个 sigevent 数据结构中，并传递给 mq_notify( )。队列的
ID * 以参数传递给信号处理程序*/
    sigNotify.sigev_signo       = SIGUSR1;
    sigNotify.sigev_notify      = SIGEV_SIGNAL;
    sigNotify.sigev_value.sival_int = ( int) exMqId;

    if ( mq_notify ( exMqId, &sigNotify) == -1)
        {
        printf ( "mq_notify failed\n");
        return ( -1);
        }
    /*
     * 刚刚创建的消息队列为空状态；
     * 在请求通知时，一个更高优先级任务在队列中可能放入了消息；若此时在队列中存在消息，
那么 mq_notify( )将不做任何事情，因此下面将恢复队列中存在的消息。*/
    exMqRead ( exMqId);
    /*
     * 现在，消息队列为空，将在下次有消息到来时接收一个信号
     * 发送一个消息来唤醒用于通告的处理程序；把获得通告的任务转变为在队列中放置消息的任
务比较笨拙，这里简化了实例，实际应用中远不止这些内容。 */
    if ( mq_send ( exMqId, pMess, 1 + strlen ( pMess), 0) == -1)
        {
        printf ( "mq_send failed\n");
        return ( -1);
        }
    /* Cleanup */
    if ( mq_close ( exMqId) == -1)
        {
        printf ( "mq_close failed\n");
        return ( -1);
        }
    /* More cleanup */
    if ( mq_unlink ( QNAM) == -1)
        {
        printf ( "mq_unlink failed\n");
```

```
        return ( -1);
        }

    return ( 0);
    }
/*
* exNotificationHandle -处理读消息。
*
* 该函数是一个信号处理程序,它从消息队列中读消息。*/
static void exNotificationHandle
    (
    int         sig,            /* 信号数量*/
    siginfo_t * pInfo,          /* 信号信息*/
    void  *     pSigContext     /* 未使用的 ( 为posix用) */
    )
    {
    struct sigevent   sigNotify;
    mqd_t             exMqId;
    /* 获得siginfo数据结构外的消息队列 ID */
    exMqId = ( mqd_t) pInfo->si_value.sival_int;
    /*
     * 再次请求发送通告信号,每次发送通告信号结束后,需要重新设置。*/
    sigNotify.sigev_signo = pInfo->si_signo;
    sigNotify.sigev_value = pInfo->si_value;
    sigNotify.sigev_notify = SIGEV_SIGNAL;

    if ( mq_notify ( exMqId, &sigNotify) == -1)
        {
        printf ( "mq_notify failed\n");
        return;
        }
    /* 读消息 */
    exMqRead ( exMqId);
    }
/*
 * exMqRead -读消息
 * 该函数接受并显示当前 POSIX 消息队列中的所有的消息。假设队列为 0_NONBLOCK*/
static void exMqRead
    (
    mqd_t       exMqId
    )
    {
```

```
char        msg[MSG_SIZE];
int         prio;
/*
 * 读操作- 在消息中通过一个循环来读消息
 * 在一个空消息队列发送消息时,仅发送通告的信号。例如,一个高优先级的任务在发送消息
时,可能有多种消息。由于消息队列打开时用 O_NONBLOCK 标志,最终该循环退出时,errno 设置成
EAGAIN( 即在一个空消息队列上调用了一次 mq_receive( ))。*/
    while ( mq_receive ( exMqId, msg, MSG_SIZE, &prio) != -1)
        {
        printf ( "exMqRead: received message: %s\n",msg);
        }

    if ( errno != EAGAIN)
        {
        printf ( "mq_receive: errno = %d\n", errno);
        }
    }
```

3.9 POSIX 队列信号

给任务发送信号时，sigqueue( )可以替换 kill( )。两者的主要区别如下：

（1）sigqueue( )包含一个指定应用值，该值作为信号的一部分发送给任务；该值可以提供与信号处理程序相关的上下文。该值属于 sigval 类型（定义在 signal.h 中）；信号处理程序（signal handler）在其参数的 si_value 域发现一个结构——siginfo_t，POSIX sigaction( )的扩展允许寄存可以接受附加参数的信号处理程序。

（2）sigqueue( )可以对发送给任一任务的多个信号进行排队。相对而言，即使在信号处理程序运行之前到来多个信号，函数 kill( )也只能传递单个信号。

VxWorks 在 SIGRTMIN 中为应用保留了七个连续编号的信号，对于 POSIX 1003.1b 来说这些保留信号是必要的，但对指定的信号值却不作要求；对于可移植性，指定这些信号作为 SIGRTMIN 中的偏移量（例如，SIGRTMIN +2 表示第三个保留信号编号）。所有通过调用 sigqueue( )传递的信号都通过编号来排序，低编号信号排列在高编号信号的前面。

POSIX 1003.1b 还引入了另一种接受信号的方法。sigwaitinfo( ) 不同于 sigsuspend( )或 pause( )，是因为 sigwaitinfo( )允许应用不使用信号处理程序寄存机制来响应信号；当寄存信号可用时，函数 sigwaitinfo( )返回该信号值作为结果；即使有信号处理程序寄存时，也不调用信号处理程序。sigtimedwait( )用法与 sigwaitinfo( )用法类似，只增加了超时功能。

信号的详细信息，请参见 sigLib 相关条目。表 3-8 列出了 POSIX 1003.1b 队列信号的调用。

表 3-8　POSIX 1003.1b 队列信号的调用

调　用	描　述
sigqueue( )	发送一个队列信号
sigwaitinfo( )	等待一个信号
sigtimedwait( )	使用超时等待一个信号

使用 INCLUDE_POSIX_SIGNALS 组件配置 VxWorks 系统来包含 POSIX 队列信号。该组件调用 sigqueueInit( )将自动初始化 POSIX 队列信号。sigqueueInit( )为 sigqueue( )分配缓冲器，因为 sigqueue( )需要一个缓冲器存储当前队列信号。若无可用缓冲器，sigqueue( )调用失败。

第 4 章　输入/输出系统

4.1　简　　介

VxWorks 操作系统中的 I/O 系统可以提供简单、统一、与任何设备无关的接口。这些设备包括：

- 面向字符设备，例如显示终端或者通信线
- 随机块存取设备，例如磁盘
- 虚拟设备，例如程序内部的通信管道和套接字
- 控制和监视设备，例如数字和模拟的 I/O 设备
- 可以访问远端设备的网络设备

VxWorks 操作系统中的 I/O 系统包括基本 I/O 系统和缓冲 I/O 系统，操作系统为每种 I/O 系统提供不同的 C 语言函数库。基本 I/O 系统使用与 UNIX 标准兼容的 C 语言函数库；缓冲 I/O 系统使用与 ANSI-C 标准兼容的 C 语言函数库。VxWorks 操作系统的 I/O 系统内部采用特殊设计，具有比其他 I/O 系统更快速、兼容性更好的特性。这些特性对于实时系统是很重要的。

本章共分四个部分。在第一部分中介绍了文件和设备的概念，并从用户的角度介绍基本 I/O 系统和缓冲 I/O 系统；在第二部分中具体介绍几种特殊的设备；第三部分详细介绍 VxWorks 操作系统中 I/O 系统的内部结构；最后介绍 VxWorks 操作系统如何支持 PCMCIA（个人计算机存储器卡接口适配器）和 PCI（个人计算机接口）。

图 4-1 说明了 VxWorks 操作系统中 I/O 系统的各个组成部分之间的关系。本章将介绍除了文件系统和网络两部分以外的所有组成部分（文件系统将在第 5 章中介绍，网络系统将在 VxWorks 操作系统网络编程手册中介绍）。

⚠ **注意**：图 4-1 中的虚线表示 CBIOS（定制的基本输入输出系统）块存取设备在文件系统调用驱动程序的过程中是可选的。

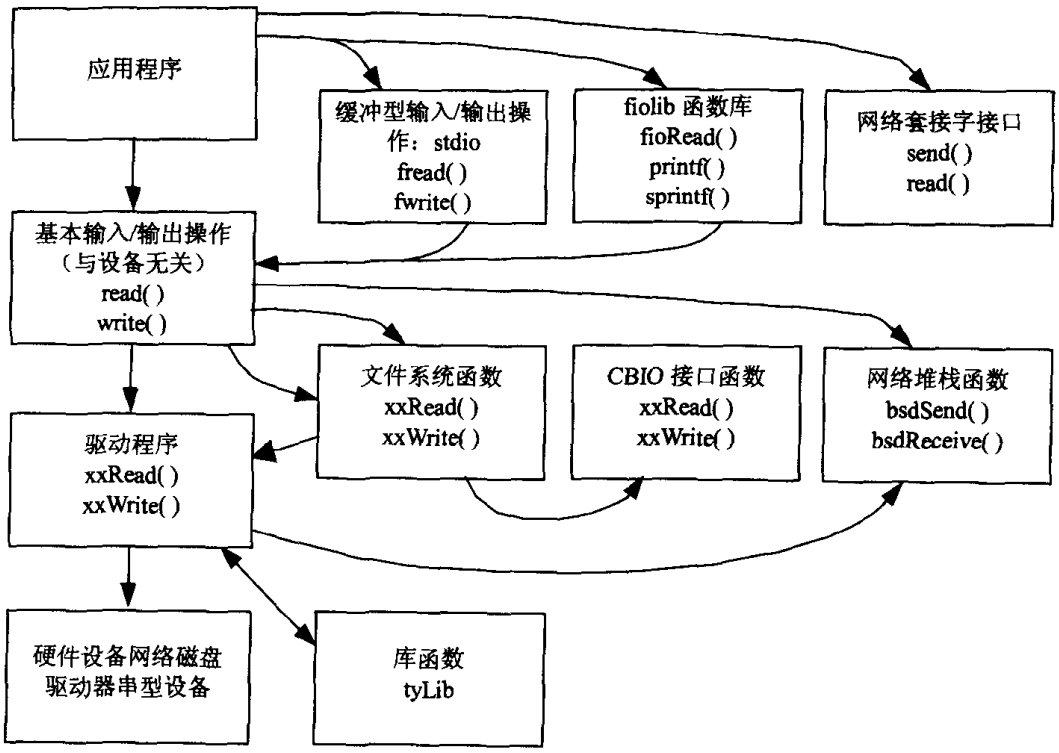


图 4-1　VxWorks 操作系统中 I/O 系统的整体结构

4.2　文件、设备和驱动程序

在 VxWorks 操作系统中，应用程序通过打开指定的文件来操作 I/O 设备。一个指定的文件表示：

- 一个非结构化的原始设备，如一个串行通信通道或者一个任务间的管道
- 在具有文件系统的结构化的随机存储设备上的一个逻辑文件

举例说明：

/usr/myfile

/pipe/mypipe

/tyCo/0

例 1 表示在一个名为/usr 上的一个称为 myfile 的文件；

例 2 表示一个已命名的管道（通常管道名以/pipe 开头）；

例 3 表示一个实际的串行通道；

在 VxWorks 操作系统中，对于以上不同对象的 I/O 操作方法是相同的。即使这些操作

对象代表不同的实际对象，但它们都称为文件。

在系统中，由称为驱动程序的程序模块负责控制各种设备。通常，对 I/O 系统的操作并不需要深入了解各种设备和驱动程序的运行方式。但是 VxWorks 操作系统中的 I/O 系统对控制各种指定的设备具有很强的灵活性。驱动程序既可以运行于传统的方式下，又可以在特定的应用中采用不同的方式。详见“4.7 VxWorks 操作系统中的设备”。

虽然所有的 I/O 操作都指向已命名的文件，但它也分为两种不同的类型：基本操作和缓冲操作。这两种类型的操作区别在于数据缓冲的方式和调用 I/O 操作的方式不同。这两种类型的操作将在以后的章节中进行介绍。

4.2.1 文件名称和默认设备类型

文件名必须是一个有效字符串。一个非结构化的设备由其设备名标识。在一个文件系统设备中，用设备名和紧跟其后的文件名表示一个文件。例如，/tyCo/0 可能表示一个特定的串行 I/O 通道，而 DEV1:/file1 可能表示在设备 DEV1 上一个名为 file1 的文件。

当在一次 I/O 调用中使用某一文件名时。I/O 系统会寻找具有相匹配文件名的设备，然后 I/O 功能函数就指向这个设备。

如果找不到相匹配的文件名设备时，I/O 功能函数就会指向一个默认设备。用户能够将系统中的任何设备设置为默认设备，包括不指定任何设备，此时系统就会因为没有找到相匹配的设备而返回一个错误。用户可以通过使用 ioDefPathGet( )函数获取当前默认设备的类型，也可以通过使用 ioDefPathSet( )函数设置默认设备的类型。

通常在系统初始化时，非块存取设备在加入 I/O 系统时被指定文件名。而块存取设备是在它们被初始化使用某一指定文件系统时被指定文件名。VxWorks 操作系统中的 I/O 系统对于给设备指定的文件名并没有任何限制。除了在寻找匹配的设备名和文件名时，I/O 系统并不翻译设备或者文件名。

如何命名设备和文件名有一些习惯方法：除了非 NFS（网络文件系统）格式的网络设备和 VxWorks 操作系统的 DOS 设备（dosFs）以外，大部分的设备名以斜线（/）开头。

通常，以斜线（/）开始的名称表示基于 NFS 文件系统的网络设备，例如：/usr。

不基于 NFS 文件系统的网络设备名是在远程主机名后加一冒号，例如：host:

设备名中其余的部分是远程系统目录中的文件名。

使用 dosFs 格式的文件系统设备名经常以大写字母、数字的组合形式加一冒号构成，例如：DEV1:

注意：在 dosFs 格式设备中的文件和目录名经常用反斜线（\）来分割，它在使用中可以与斜线（/）互换。

警告：因为在 I/O 系统中使用简单的字符匹配方法来识别设备名，因此不能单独将斜线（/）用作设备名。

4.3　基本 I/O 接口

基本 I/O 接口是 VxWorks 操作系统中最低级别的 I/O 接口。它在标准 C 语言库中与 I/O 原语兼容。在 VxWorks 操作系统中共有 7 种基本 I/O 操作可供调用，如表 4-1 所示。

表 4-1　基本 I/O 操作程序

函 数 名 称	功 能 介 绍
creat( )	创建一个文件
delete( )	删除一个文件
open( )	打开一个文件（可以同时创建文件）
close( )	关闭一个文件
read( )	从一个已打开的文件中读取数据
write( )	向一个已打开的文件写入数据
ioctl( )	对文件执行特殊的控制操作

4.3.1　文件描述符

在基本 I/O 接口的操作中，文件通过相应的文件描述符（fd）来表示。一个文件描述符（fd）就是通过调用 open( )或者 creat( )函数而返回的一个整数值。其他的基本 I/O 接口函数通过将文件描述符（fd）作为一个参数来指定目的文件。

文件描述符（fd）在系统中是全局可见的。例如，任务 A 和 B 对文件描述符（fd）为 7 的文件各调用一次 write( )函数时，它们将对同一个文件（设备）进行操作。

当函数 open( )打开一个文件时，系统将分配一个文件描述符（fd），并将它传递给函数的调用者。当文件被关闭后，相应的文件描述符（fd）会被系统删除。在 VxWorks 操作系统中，文件描述符（fd）的个数是有限的。为了避免在使用中出现文件描述符（fd）的个数超出规定限制的情况，需要关闭那些不再使用的文件，从而释放资源。在 I/O 系统初始化时，设定可以使用的文件描述符（fd）的个数。

通常，只有当文件被关闭后系统才删除文件描述符（fd）。

4.3.2 标准输入设备、标准输出设备和标准错误输出设备

以下三个文件描述符（fd）的值作为默认值有特殊的意义：

0 = 标准输入设备

1 = 标准输出设备

2 = 标准错误输出设备

这些文件描述符（fd）的值不会出现在 open( ) 或 creat( )函数的返值中，但它们可以作为非直接的参考值重定向到其他任何已打开文件的文件描述符（fd）。

这些文件描述符（fd）能使得任务和功能模块与它们实际的 I/O 操作相独立。如果一个功能模块将它的输出设备定向到标准输出设备（fd = 1），此时不需要修改该功能模块就能将它的输出设备重定向到任何文件或设备上。

VxWorks 操作系统允许两级重定向操作：第一级操作是三个具有全局可见性的标准文件描述符（fd）；第二级操作是任何单独的任务可以用它自己的文件描述符代替全局可见性的标准文件描述符（fd），而且此项操作只会影响该任务本身。

1. 全局重定向操作

当 VxWorks 操作系统初始化时，全局可见性的标准文件描述符（fd）指向系统控制台。当开始一个新任务时，它没有被分配一个指向特定任务的文件描述符（fd），而是使用全局标准文件描述符（fd）。

全局可见性的标准文件描述符（fd）可以通过调用 ioGlobalStdSet( )函数而被重定向。该操作用到的参数有两个：一个是被重定向的全局可见性的标准文件描述符（fd），另一个是重定向的目标文件描述符（fd）。例如，如下的函数调用将全局可见性的标准输出文件（fd = 1）重定向到一个已打开的文件（fd = fileFd）。

```
ioGlobalStdSet (1, fileFd);
```

从此以后，系统中的所有任务的标准输出都写入那个文件中。例如任务 tRlogind 在远程登录期间调用 ioGlobalStdSet( )函数通过网络重定向 I/O 操作。

2. 任务的重定向操作

不同的任务可以通过调用 ioTaskStdSet( )函数进行重定向操作。该操作有三个参数：第一个是调用 ioTaskStdSet( )函数的任务的任务号（ID = 0 表示该任务本身）；第二个是被重定向的全局可见性的标准文件描述符（fd）；第三个是重定向的目标文件描述符（fd）。例如，一个任务可以通过如下方法将标准输出文件（fd = 1）重定向到一个已打开的文件（fd = fileFd）。

```
ioTaskStdSet (0, 1, fileFd);
```

其他所有任务都不受此次重定向操作的影响，同样该任务也不受以后其他任务重定向操作的影响。

4.3.3　打开和关闭文件操作

当对一个设备进行 I/O 操作之前，必须通过调用 open( )函数或者 creat( )函数（该函数将在下一小节中介绍）获得相应的文件描述符（fd）。open( )函数需要的参数包括文件名、文件访问方式、文件模式。例如：

```
fd = open ("name", flags, mode);
```

文件访问方式标志（flag）如表 4-2 所示。

表 4-2　文件访问方式标志

标　志　名	标志值（Hex）	说　　明
O_RDONLY	0	以只读方式打开文件
O_WRONLY	1	以只写方式打开文件
O_RDWR	2	以读写方式打开文件
O_CREAT	200	建立一个新文件
O_TRUNC	400	删除该文件

警告：open( )函数的第三个参数对于 VxWorks 操作系统中的应用是必须的，但对于其他操作系统中的应用来说是可选的。当该函数的第三个参数无效时参数值必须设为零。当程序由 UNIX 操作系统移植到 VxWorks 操作系统上时尤其要注意这个问题。

文件模式参数用于在如下操作中指定文件或建立子目录的模式（又称允许位）：

- 通常，用户只能通过调用 open( )函数来打开已经存在的设备和文件。但是，对 NFS 格式的网络设备和 dosFs 格式的设备，用户在调用 open( )函数时，可以通过在文件访问方式标志项中增加 O_CREAT 参数来同时新建并打开一个文件。对于 NFS 格式的设备，调用 open( )函数时还须指定文件模式参数。例如：

```
fd = open ("name", O_CREAT | O_RDWR, 0644)
```

- 对于 NFS 格式和 dosFs 格式的设备，用户可以同时使用文件访问方式标志参数 O_CREAT 和文件模式参数 FSTAT_DIR 来新建一个子目录。对于 dosFs 格式的设备，其他的文件模式参数是无效的。

如果 open( )函数调用成功，它将会返回一个文件描述符（fd）。以后对于该文件的 I/O 操作将使用这个文件描述符（fd）。这个文件描述符（fd）与系统运行的任务无关。某一个

任务可以打开一个文件，然后另一个任务可以使用相应的文件描述符（fd）（例如：对于管道的操作）。直到调用 close( ) 函数将该文件关闭，其相应的文件描述符（fd）一直有效。例如：

```
close (fd);
```

从某种意义上说，对于文件的 I/O 系统被刷新（全部输出）以及文件描述符（fd）不能被其他任务用于其他应用中。但是同一个文件描述符（fd）的值可以被 I/O 系统在后续的 open( )函数调用中再次使用。

因为文件描述符（fd）在默认的情况下并不是与任务相连的，因此当退出或删除一个任务时，除非该任务拥有它所打开的文件，否则这些文件不会自动关闭。因此当任务不再需要某些文件时，建议用户及时关闭相应文件。这样做的另一个原因是，VxWorks 操作系统对于能够同时打开的文件数目有限制。

4.3.4 新建和删除文件操作

面向文件的设备必须既能够打开已存在的文件，也能够新建和删除文件。

creat( )函数的功能是在一个面向文件的设备上新建一个文件并返回一个文件描述符（fd）。对于 creat( )函数的使用，除了将文件名参数改为新建文件名以外，其使用方法与 open( )函数相同，而且 creat( )函数返回一个指向新文件的文件描述符（fd）。例如：

```
fd = creat ("name", flag);
```

delete( )函数的功能是在一个面向文件的设备上删除一个已命名的文件，例如：

```
delete ("name");
```

不要在文件处于打开状态时试图删除文件。

如果在 creat( )函数中使用了面向非文件系统的设备名时，它的功能就如同 open( )函数一样，同样 delete( )函数功能无效。

4.3.5 读操作和写操作

在利用 open( )或 creat( )函数获得文件描述符（fd）后，任务可以通过调用 read( ) 函数和 write( )函数从相应文件中读出数据和写入数据。read( ) 函数的参数有文件描述符（fd）、用于接收读出数据的缓冲区地址和读出数据的字节数。例如：

```
nBytes = read (fd, &buffer, maxBytes);
```

read( ) 函数在实际执行时会等待从指定的文件中读取数据，并返回实际读取数据的字节数。对于文件系统设备，如果实际读取数据的字节数少于指定读取的字节数，随后被调

用的 read( ) 函数会返回一个 0 值，表示已经到了文件尾部。对于非文件系统设备，即使有可以读取的数据，实际读取数据的字节数仍可以少于指定读取的字节数，随后被调用的 read( ) 函数可以返回一个 0 值，也可以返回一个非 0 值。这种情况适用于串行设备和 TCP 套接字，它们都需要连续调用 read( ) 函数来读取指定字节数的数据（详见 fioLib 库中的 fioRead( )函数介绍）。如果调用 read( ) 函数返回一个出错值（-1），则表示本次操作出错。

使用 write( )函数时需要的参数有文件描述符（fd），存储输出数据缓冲区的地址以及输出数据的字节数。

```
actualBytes = write (fd, &buffer, nBytes);
```

虽然输出数据还没有被写入到设备中，但 write( )函数在执行时会将数据按线性顺序输出（这个过程与驱动程序有关）。write( )函数的返回值表示被写入数据的字节数。如果返回值与函数的参数值不同，则表示程序出现了错误。

4.3.6　文件截取操作

有时有必要删除文件的一部分数据。当文件按照写入模式被打开后，用户可以调用 ftruncate( )函数改变该文件的大小。该函数的参数包括文件描述符（fd）和所需的文件长度。

```
status = ftruncate (fd, length);
```

如果文件截取操作成功，ftruncate( )函数返回 OK 标志。如果指定的文件长度比实际文件长度大或文件描述符（fd）所指向的设备不能执行该操作，则 ftruncate( )函数返回 ERROR 标志并将错误号设为 EINVAL。

ftruncate( )函数是 POSIX 1003.1b 标准的一部分，但该函数只是与 POSIX 标准部分兼容。因为文件被创建和修改的次数不会被更新，所以只有与 DOS 系统兼容的文件系统函数库 dosFsLib 支持 ftruncate( )函数。系统中只有包括 INCLUDE_POSIX_FTRUNCATE 组件后才提供文件截取功能。

4.3.7　I/O 控制操作

ioctl( )函数提供了一种自由的方式执行其他基本 I/O 功能函数不能执行的操作，例如可以读取能够输入的实际数据量、读取某一文件系统的基本信息，以及在文件的任意位置读/写数据等。该函数所需要的参数包括：文件描述符（fd）、表示所执行功能的代码以及一个与所执行功能有关的参数。

```
result = ioctl (fd, function, arg);
```

下面的语句是通过执行 FIOBAUDRATE 功能将一个 tty 设备传输波特率设置为 9600。

```
status = ioctl (fd, FIOBAUDRATE, 9600);
```

在“4.7 VxWorks 操作系统中的设备”中将介绍 ioctl( )函数针对不同设备的使用方法。在头文件 ioLib.h 中定义了在 ioctl( )函数中可以使用的功能代码。用户欲了解更多内容，请查阅参考手册中关于相应驱动程序或文件系统的内容。

4.3.8 基于多文件描述符的挂起操作：选择功能

VxWorks 操作系统通过利用多文件描述符的挂起功能提供与 UNIX 和 WINDOWS 操作系统相兼容的选择功能。库文件 selectLib 即提供任务级支持——允许任务等待多个设备中的一个变为活动状态，又提供对设备驱动程序的支持——使驱动程序能够检测是否有任务因等待对设备进行 I/O 操作而处于等待状态。应用程序中必须包括头文件 selectLib.h 才能实现该功能。

任务级应用不仅支持任务同时等待对多个设备进行 I/O 操作，还能够指定等待过程的最长时间。一个利用多文件描述符执行选择操作的例子是客户机/服务器模型。在该模型中服务器同时对本地和远端的客户机提供服务。在服务器上运行的任务利用管道与本地客户机通信，利用套接字与远端客户机通信。它必须尽可能快地响应客户机的请求。如果服务器仅仅因为等待多个通信流中其中一个的响应而处于挂起状态，那么它在收到第一个响应之前将不会响应其他通信流传来的请求。这种情况将会导致对本地请求的响应产生延迟。选择功能将使服务器不论正在处理哪个通信流，都将同时监视所有套接字和管道中传来的服务请求信息。这样可以很好地解决上述问题。

在数据变为有效或设备变为可写状态之前，任务一直处于拥塞状态。通过 select( )函数可以知道何时一个或多个文件描述符变为有效状态或发生了超时的情况。通过 select( )函数可以指定某一个任务等待哪些文件描述符变为活动状态。在 select( )函数中使用位域标志设置有关文件的读/写状态。通过 select( )函数返回值中位域标志的改变情况反映相应的文件描述符是否有效。表 4-3 中列出建立和操纵这些位域标志的宏函数。

表 4-3 选择功能中的宏函数

宏 函 数 名	功 能 介 绍
FD_ZERO	将所有标志位设置为‘0’
FD_SET	对指定文件描述符的标志位设置为‘1’
FD_CLR	对指定文件描述符的标志位设置为‘0’
FD_ISSET	如果指定标志位值为‘1’，则返回‘1’；否则返回‘0’

在应用程序中可以通过调用 select( )函数使得字符型 I/O 设备（如：管道、串行设备、套接字）具有上述功能。如果用户想了解编写支持 select( )函数的设备驱动程序方面的信息，请查阅 select( )函数的应用范例。

例 4-1：选择功能

```
/* selServer.c - 演示选择功能
* 在本例中,一个任务充当服务器,它使用两个管道通信。一个具有正常的优先级,
*另一个具有更高的优先级。服务器同时打开这两个管道并等待它们传送的信息。
*/

#include "vxWorks.h"
#include "selectLib.h"
#include "fcntl.h"

#define MAX_FDS 2
#define MAX_DATA 1024
#define PIPEHI   "/pipe/highPriority"
#define PIPENORM "/pipe/normalPriority"

/*********************************************************************
* selServer - 读取两个管道中出现的有效数据
* 打开两个管道的文件描述符,从管道中读取有效数据。服务器的程序假定管道已由其他
* 任务或命令解释器(shell)建立。从命令解释器中测试该部分程序需执行如下操作:
*  -> ld < selServer.o
*  -> pipeDevCreate ("/pipe/highPriority", 5, 1024)
*  -> pipeDevCreate ("/pipe/normalPriority", 5, 1024)
*  -> fdHi = open   ("/pipe/highPriority", 1, 0)
*  -> fdNorm = open ("/pipe/normalPriority", 1, 0)
*  -> iosFdShow
*  -> sp selServer
*  -> i
* 此时,用户会看到 selServer 处于挂起状态。用户可以向任意一个管道中写入数据并
* 使 selServer 显示输入的数据。
*  -> write fdNorm, "Howdy", 6
*  -> write fdHi, "Urgent", 7
*/

STATUS selServer (void)
    {
    struct fd_set readFds;      /* bit mask of fds to read from */
    int     fds[MAX_FDS];      /* array of fds on which to pend */
    int     width;            /* number of fds on which to pend */
    int     i;                /* index for fd array */
    char    buffer[MAX_DATA]; /* buffer for data that is read */

/* 打开文件描述符 */
```

```
    if ((fds[0] = open (PIPEHI, O_RDONLY, 0)) == ERROR)
        {
        close (fds[0]);
        return (ERROR);
        }
    if ((fds[1] = open (PIPENORM, O_RDONLY, 0)) == ERROR)
        {
        close (fds[0]);
        close (fds[1]);
        return (ERROR);
        }
/* 程序处于循环状态,读取数据并对客户端程序提供服务 */
    FOREVER
        {
        /* clear bits in read bit mask */
        FD_ZERO (&readFds);

/* 初始化控制位 */
        FD_SET (fds[0], &readFds);
        FD_SET (fds[1], &readFds);
        width = (fds[0] > fds[1]) ? fds[0] : fds[1];
        width++;

/* 程序处于悬挂状态,等待文件描述符变为可用状态 */
    if (select (width, &readFds, NULL, NULL, NULL) == ERROR)
        {
        close (fds[0]);
        close (fds[1]);
        return (ERROR);
        }

/* 从可用的文件描述符中读取数据 */
    for (i=0; i< MAX_FDS; i++)
        {
        /* check if this fd has data to read */
        if (FD_ISSET (fds[i], &readFds))
            {
            /* typically read from fd now that it is ready */
            read (fds[i], buffer, MAX_DATA);
            /* normally service request, for this example print it */
            printf ("SELSERVER Reading from %s: %s\n",
                    (i == 0) ? PIPEHI : PIPENORM, buffer);
```

```
        }
    }
}
}
```

4.4 缓冲型 I/O 设备：stdio

VxWorks 操作系统中的 I/O 设备库包含一个与 UNIX 和 Windows 操作系统中的 stdio 包相兼容的缓冲型 I/O 设备包，并且完全支持 ANSI C 语言。配置了 ANSI 标准组件的 VxWorks 操作系统提供对缓冲型 I/O 操作的支持。

⚠ **注意**：传统上包括在 stdio 包中的 printf( )、sprintf( )和 sscanf( )函数在 VxWorks 操作系统属于不同的包。这些函数将在“4.5 其他格式化 I/O 操作”一节中介绍。

4.4.1 使用 stdio 设备

虽然 VxWorks 操作系统中的 I/O 操作工作效率较高，但系统仍需花费一些与每一个低层函数调用有关的额外开销。第一种情况是为了实现某一个功能，I/O 系统先调用与设备无关的功能函数（如 read( )、write( )等），然后再调用与驱动程序相关的低层函数。第二种情况是大多数驱动程序都会采用互斥或队列机制来避免发生因同时调用一个驱动程序而导致多个用户程序互相影响的情况。

因为 VxWorks 操作系统的源程序运行速度非常快，所以这些额外的系统开销非常小。但是如果一个应用程序在读取每一个字符时都分别调用 read( )函数，例如：

```
n = read (fd, &char, 1);
```

那么在处理每一个字符时都会带来额外的系统开销。为了使这类 I/O 操作更有效、更灵活，stdio 包使用了一种缓冲机制，即只对大量的缓冲数据进行读/写操作。这种缓冲过程对于应用程序而言是透明的，而且它是由 stdio 包中的函数和宏自动控制的。如果用户想用 stdio 包处理一个文件，那么就应使用 fopen( )函数，而不是 open( )函数打开文件（许多 stdio 包中的函数都是以字母 f 开头）。例如：

```
fp = fopen ("/usr/foo", "r");
```

fopen( )函数的返回值是一个文件指针（fp）。它指向已打开的文件、相应的缓冲区及其指针的句柄。一个文件指针实际上是指向 FILE 类型的数据结构的指针（也就是说，文件指针必须声明为 FILE *）。相比较而言，低层 I/O 操作函数通过文件描述符（fd）来确认一个

文件，其数据类型是整型。实际上，文件指针所指向的 FILE 数据结构中包括标识已打开文件的文件描述符。一个已打开的文件描述符可以通过调用 fdopen( )函数与一个缓冲区连接，例如：

```
fp = fdopen (fd, "r");
```

在通过调用 fopen( )函数打开一个文件后，可以使用 fread( )或 fwrite( )函数批量读出或写入数据，或者使用 getc( ) 和 putc( )函数一次只读出或只写入一个字符。

使用上述的函数或宏执行从一个文件中读出或写入数据操作是十分有效的。这些函数通过一个在读/写数据时增加的指针访问缓冲区。当读缓冲区为空或写缓冲区为满时，函数才调用低层的读/写操作程序。

警告： 在 stdio 操作中的缓冲区和指针对于某一个具体的任务来说是专有的。它们之间不采用信号量或互斥机制连接，因为这样做降低了专有缓冲区机制的效率。因此多个任务不能同时对同一个 stdio 操作中的 FILE 类型指针进行 I/O 操作。

调用 fclose( )函数可以析构 FILE 类型的缓冲区。

4.4.2 标准输入设备、标准输出设备和标准错误输出设备

如 4.3 节基本 I/O 操作中所述，系统为标准输入设备、标准输出设备和标准错误输出设备指定了三种文件描述符（0，1，2）。当一个任务使用标准文件描述符（stdin，stdout 和 stderr）执行缓冲型 I/O 操作时，系统会自动建立相应的 stdio 操作的 FILE 类型缓冲区。每一个使用标准 I/O 操作文件描述符的任务都拥有各自的 stdio 操作的 FILE 类型缓冲区。当任务退出时，析构各自的缓冲区。

4.5 其他格式化 I/O 操作

本节介绍其他几种格式化 I/O 操作的函数及其功能。

4.5.1 特例：printf( )，sprintf( )和 sscanf( )函数

printf( )，sprintf( )和 sscanf( )函数通常作为标准 stdio 包的一部分，但 VxWorks 操作系统中的应用程序在调用这些函数时并不使用 stdio 包。操作系统使用 fioLib 函数库中的一种

自制的、格式化的、非缓冲型的接口对 I/O 系统进行操作。注意这些函数支持 ANSI 标准指定的一些功能，但 printf( )函数是一个非缓冲型函数。

由于这些程序运行时有上述特点，因此整个 stdio 包作为可选项在 VxWorks 操作系统中可以被删除而不影响程序的运行。如果用户想执行带有缓冲功能的 printf 类型的输出操作，可以在应用程序中调用 fprintf( )函数。

虽然 sscanf( )函数可以在应用程序中不包括 stdio 包，而只有 fioLib 文件时使用，但 scanf( )函数却不能。

4.5.2　其他函数：printErr( )和 fdprintf( )函数

在 fioLib 文件中提供了另外两种具有格式化操作但是非缓冲型的输出函数。printErr( )函数在功能上与 printf( )函数相似。不同之处在于前者是将格式化的字符串输出到标准错误输出设备上（fd=2），而后者可以将其输出到指定的文件描述符所代表的设备上。

4.5.3　信息记录

由 logLib 文件提供的另一个高层 I/O 操作是在没有任务上下文或不对任务上下文进行 I/O 操作时允许记录格式化的信息。信息的格式和参数通过一个信息队列传递到执行记录的任务中。该任务负责格式化和输出信息。这种信息记录的方式在如下几种应用中十分有用：（1）在中断程序中需要记录信息；（2）不希望当前执行 I/O 操作的任务产生延迟；（3）不希望使用当前的任务堆栈进行信息格式化操作（这种操作会占用大量堆栈空间）。除非在系统启动时调用 logInit( )函数或在系统工作时动态地调用 logFdSet( )函数，否则信息记录操作所输出的信息将会在控制台上显示。

4.6　异步输入/输出操作

异步输入/输出操作（AIO）能够在执行普通的内部处理时，同时执行输入/输出操作。当 I/O 操作与具体的任务间逻辑上是独立的时候，异步输入/输出操作使得用户能够将 I/O 操作与具体的任务分离。

异步输入/输出操作的优点是极大地提高系统处理的效率。即任何时候，只要系统资源有效，异步输入/输出功能就能开始 I/O 操作，而不必等待一些专门事件（如完成一个独立的操作）的发生。异步输入/输出操作减少了一些由同步输入/输出操作引起的不必要的任务拥塞现象，而且它能够减少输入/输出和内部处理过程中的资源竞争现象，并加快了处理速度。

VxWorks 操作系统中的异步输入/输出操作符合 POSIX 1003.1b 标准。如果想在 VxWorks 操作系统中加入异步输入/输出操作功能，需要在系统内核中加入 INCLUDE_POSIX_AIO 和 INCLUDE_POSIX_AIO_SYSDRV 组件。所有 VxWorks 操作系统中执行异步输入/输出操作的设备都需要辅助异步输入/输出系统驱动程序，而第二个配置常数使得该驱动程序有效。

4.6.1 POSIX 标准的异步输入/输出程序

aioPxLib 库文件提供符合 POSIX 标准的异步输入/输出程序。就像处理其他文件一样使用 open( )函数打开一个文件。然后使用 open( )返回的文件描述符调用异步输入/输出程序。在表 4-4 中列出了 POSIX 标准的异步输入/输出程序（其中包括两个有关的非 POSIX 标准的程序）。

当 VxWorks 操作系统内核中包括 INCLUDE_POSIX_AIO 组件后，系统就包含了 POSIX 标准的异步输入/输出组件。此时默认的 VxWorks 操作系统初始化程序会自动调用 aioPxLibInit( )函数。

aioPxLibInit( )函数只需一个参数，即同时允许调用 lio_listio( )函数的最大个数。该参数的默认值为常数 MAX_LIO_CALLS。当参数值为 '0' 时（默认情况），其值从常数 AIO_CLUST_MAX 中获取（该常数在文件 installDir/target/h/private/aioPxLibP.h 中定义）。

当 VxWorks 操作系统内核中包括 INCLUDE_POSIX_AIO 和 INCLUDE_POSIX_AIO_SYSDRV 组件后，异步输入/输出系统的驱动程序 aioSysDrv 在默认情况下由 aioSysInit( ) 函数初始化。驱动程序 aioSysDrv 的作用是建立与任何特定的设备驱动程序无关的请求队列。因此，用户可以在 VxWorks 操作系统中对任何设备驱动程序使用异步输入/输出操作。

表 4-4 异步输入/输出操作函数

函 数 名 称	功 能 介 绍
aioPxLibInit( )	初始化 AIO 功能函数库（非 POSIX 标准）
aioShow( )	显示 AIO 功能要求（非 POSIX 标准）注
aio_read( )	初始化异步读操作
aio_write( )	初始化异步写读操作
aio_listio( )	初始化个数最多为 LIO_MAX 的 AIO 功能请求
aio_error( )	在一个 AIO 操作中寻找错误状态值
aio_return( )	在一个已完成的 AIO 操作中寻找返回状态值
aio_cancel( )	取消一个 AIO 操作
aio_suspend( )	等待一个 AIO 操作完成、被中断或超时

1：该函数不被编译进 Tornado 软件中的 shell 程序中。用户要在 shell 程序中使用该函数，必须在 VxWorks 操作系统内核中加入 INCLUDE_POSIX_AIO_SHOW 组件。当用户调

用该函数时，输出结果直接传送到标准输出设备上。

aioSysInit( )函数需要三个参数：（1）可以同时执行的异步输入/输出任务数；（2）任务的优先级；（3）运行任务的堆栈容量。如果执行异步输入/输出操作任务的数目等于系统要求的数目，则各个任务可以并行执行。默认的参数值为：MAX_AIO_SYS_TASKS，AIO_TASK_PRIORITY 和 AIO_TASK_STACK_SIZE。

当任何传递到 aioSysInit( )函数的参数值为‘0’时，函数就会用以下三个常量：AIO_IO_TASKS_DFLT，AIO_IO_PRIO_DFLT 和 AIO_IO_STACK_DFLT 代替相应参数，所有常量在文件“installDir/ target/h/aioSysDrv.h”中定义。表 4-5 列出了上述参数和常量名以及相应的定义文件。

表 4-5　AIO 操作初始化函数及其相关内容

函数名称	aioPxLibInit( )	aioSysInit( )		
参数名称	MAX_LIO_CALLS	MAX_AIO_SYS_TASKS	AIO_TASK_PRIORITY	AIO_TASK_STACK_SIZE
参数的默认值	0	0	0	0
当参数值为‘0’时使用的常数	AIO_CLUST_MAX	AIO_IO_TASKS_DFLT	AIO_IO_PRIO_DFLT	AIO_IO_STACK_DFLT
常数的默认值	100	2	50	0x7000
定义常数默认值的文件名（存储在“installDir/target”目录下）	h/private/aioPxLibP.h	h/aioSysDrv.h	h/aioSysDrv.h	h/aioSysDrv.h

4.6.2　异步输入/输出操作控制块

每运行一个异步输入/输出功能函数都产生一个描述异步输入/输出操作的控制块（aiocb）。调用功能函数时必须为控制块分配与单次异步输入/输出操作相关的存储空间。两个同时运行的异步输入/输出操作不能使用同一个控制块。如果用户试图这样做，将会产生不可预料的结果。

在系统响应一个异步输入/输出操作请求的过程中，会使用控制块及其相应的数据缓冲区。因此用户申请一个异步输入/输出操作请求之后，在调用 aio_return( )函数之前不能修改相应的控制块。因为该函数释放用于修改或重用的控制块。

在 aio.h 文件中定义了异步输入/输出操作控制块的结构，它包括以下字段：

aio_fildes 字段：用于 I/O 操作的文件描述符；

aio_offset 字段：从文件开始处计算的偏移量；

aio_buf 字段：AIO 操作所需的缓冲区地址；
aio_nbytes 字段：读/写数据的字节数；
aio_reqprio 字段：本次 AIO 操作请求的优先级；
aio_sigevent 字段：完成一次操作时返回的信号（可选项）；
aio_lio_opcode 字段：调用 lio_listio( )函数所完成的操作；
aio_sys_p 字段：VxWorks 操作系统中特殊数据（非 POSIX 标准）的地址；
用户欲了解全部定义和其他重要信息，请查阅有关 aioPxLib 文件的内容。

警告：如果一个应用程序为异步输入/输出操作控制块分配了堆栈区，那么该程序必须在返回之前调用 aio_return( )函数释放该控制块。

4.6.3 使用异步输入/输出操作

aio_read( )、aio_write( )或 lio_listio( )函数可以初始化异步输入/输出操作。第三个函数允许用户同时提出多个异步输入/输出操作请求（读/写请求）。一般来说，在提出异步输入/输出操作请求之后，由这些函数初始化的操作并不立即响应。因此函数的返回值并不反映实际操作的输出值。但可以反映出所提出的请求是否被响应，即函数是否将该请求放入最终执行的操作队列中。

在输入/输出操作执行之后，会返回一个值表示该操作成功或失败。用户可以使用两个函数来得到该信息：aio_error( )和 aio_return( )函数。用户通过调用 aio_error( )函数可以获得异步输入/输出操作状态（成功、失败或正在运行中），通过调用 aio_return( )函数可以获得单独一个操作的返回值。当一个异步输入/输出操作完成后，它的错误状态是 EINPROGRESS 标志。用户通过调用 aio_cancel( )函数可以取消一个异步输入/输出操作。

1. 带有周期性完成状态检测功能的异步输入/输出操作

下面的程序范例在异步输入/输出操作中使用管道。该程序执行的过程是：首先创建一个管道，然后提出一个异步读操作请求，在验证读操作的请求正在被处理之后再提出一个异步写操作请求。通常在同步读操作过程中，任务会因对一个空的管道而产生阻塞，不会继续执行写操作。但是在异步输入/输出操作的情况下，用户可以先初始化一个读操作然后再初始化一个写操作。程序中提出一个异步写操作请求后，任务将周期性地循环检测所有的异步输入/输出操作请求，直至所有的操作完成为止。因为异步输入/输出操作的控制块存放于堆栈区，因此程序在从 aioExample( )函数返回前必须调用 aio_return( )函数。

例 4-2：异步输入/输出操作

```
/* aioEx.c - 使用异步输入/输出操作的程序 */

/* 头文件 */
```

```
#include "vxWorks.h"
#include "aio.h"
#include "errno.h"

/* 常量定义 */

#define BUFFER_SIZE 200

/*******************************************************************
* aioExample - 使用AIO 库 * 该程序演示AIO库的基本功能。
* 返回值:如果操作成功返回OK标志,否则返回ERROR标志。
*/

STATUS aioExample (void)
    {
    int      fd;
    static char   exFile [] = "/pipe/1stPipe";
    struct aiocb  aiocb_read; /* read aiocb */
    struct aiocb  aiocb_write; /* write aiocb */
    static char *  test_string = "testing 1 2 3";
    char     buffer [BUFFER_SIZE]; /* buffer for read aiocb */

pipeDevCreate (exFile, 50, 100);

if ((fd = open (exFile, O_CREAT | O_TRUNC | O_RDWR, 0666)) ==
    ERROR)
    {
    printf ("aioExample: cannot open %s errno 0x%x\n", exFile, errno);
    return (ERROR);
    }

printf ("aioExample: Example file = %s\tFile descriptor = %d\n",
     exFile, fd);

/* 初始化读/写操作的异步输入/输出操作控制块(aiocb) */
bzero ((char *) &aiocb_read, sizeof (struct aiocb));
bzero ((char *) buffer, sizeof (buffer));
   aiocb_read.aio_fildes = fd;
   aiocb_read.aio_buf = buffer;
   aiocb_read.aio_nbytes = BUFFER_SIZE;
   aiocb_read.aio_reqprio = 0;
```

```
bzero ((char *) &aiocb_write, sizeof (struct aiocb));
    aiocb_write.aio_fildes = fd;
    aiocb_write.aio_buf = test_string;
    aiocb_write.aio_nbytes = strlen (test_string);
    aiocb_write.aio_reqprio = 0;

/* 初始化读操作 */
if (aio_read (&aiocb_read) == -1)
    printf ("aioExample: aio_read failed\n");

/* 验证正在执行读操作 */
if (aio_error (&aiocb_read) == EINPROGRESS)
    printf ("aioExample: read is still in progress\n");

/* 向管道中写数据 - 必须能够完成读操作 */
    printf ("aioExample: getting ready to initiate the write\n");
    if (aio_write (&aiocb_write) == -1)
        printf ("aioExample: aio_write failed\n");

/* 等待完成读/写操作 */
    while ((aio_error (&aiocb_read) == EINPROGRESS) ||
           (aio_error (&aiocb_write) == EINPROGRESS))
        taskDelay (1);

/* 输出读取的数据 */
    printf ("aioExample: message = %s\n", buffer);

/* 结束操作 */
    if (aio_return (&aiocb_read) == -1)
        printf ("aioExample: aio_return for aiocb_read failed\n");
    if (aio_return (&aiocb_write) == -1)
        printf ("aioExample: aio_return for aiocb_write failed\n");

close (fd);
    return (OK);
}
```

2. 检测异步输入/输出操作完成状态的方法

任务可以采用下列方法检测异步输入/输出操作是否完成：

- 周期性检测 aio_error( )函数的结果。如上例所示，程序检测异步输入/输出操作的

状态，直到不出现 EINPROGRESS 为止。

- 使用 aio_suspend( )函数将任务悬挂起来，直到异步输入/输出操作完成为止。
- 当异步输入/输出操作完成时使用信号量通知其他函数。

下面的程序除了使用信号量通知写操作完成以外，与 aioExample( )函数的功能相似。如果用户在 shell 程序中测试该程序范例，需要将其优先级设置的比 AIO 系统任务低，这样可以保证程序运行时不会阻塞 AIO 请求。

例 4-3：使用信号量执行异步 I/O 操作

```
/* aioExSig.c - 使用信号的异步输入/输出操作程序 */
/* 头文件 */
#include "vxWorks.h"
#include "aio.h"
#include "errno.h"
/* 常量定义 */
#define BUFFER_SIZE   200
#define LIST_SIZE    1
#define EXAMPLE_SIG_NO  25 /* signal number */
/* 声明 */

void mySigHandler (int sig, struct siginfo * info, void * pContext);

/*******************************************************************
* aioExampleSig - 使用 AIO 库.
*
* 该程序演示 AIO 库的基本功能。
* 如果程序从命令解释器中运行,应输入如下命令:
*  -> sp aioExampleSig
*
* 返回值:如果操作成功返回 OK 标志,否则返回 ERROR 标志。
*/
STATUS aioExampleSig (void)
    {
    int          fd;
    static char         exFile [] = "/pipe/1stPipe";
    struct aiocb        aiocb_read;       /* read aiocb */
    static struct aiocb  aiocb_write;      /* write aiocb */
    struct sigaction     action;          /* signal info */
    static char *       test_string = "testing 1 2 3";
    char         buffer [BUFFER_SIZE];     /* aiocb read buffer */
pipeDevCreate (exFile, 50, 100);
```

```
if ((fd = open (exFile, O_CREAT | O_TRUNC| O_RDWR, 0666)) == ERROR)
    {
    printf ("aioExample: cannot open %s errno 0x%x\n", exFile, errno);
    return (ERROR);
    }

printf ("aioExampleSig: Example file = %s\tFile descriptor = %d\n",
        exFile, fd);
/* 为 EXAMPLE_SIG_NO 建立信号句柄 */
    action.sa_sigaction = mySigHandler;
    action.sa_flags = SA_SIGINFO;
    sigemptyset (&action.sa_mask);
    sigaction (EXAMPLE_SIG_NO, &action, NULL);
/* 初始化读/写操作的异步输入/输出操作控制块(aiocb) */
    bzero ((char *) &aiocb_read, sizeof (struct aiocb));
    bzero ((char *) buffer, sizeof (buffer));
    aiocb_read.aio_fildes = fd;
    aiocb_read.aio_buf = buffer;
    aiocb_read.aio_nbytes = BUFFER_SIZE;
    aiocb_read.aio_reqprio = 0;
bzero ((char *) &aiocb_write, sizeof (struct aiocb));
    aiocb_write.aio_fildes = fd;
    aiocb_write.aio_buf = test_string;
    aiocb_write.aio_nbytes = strlen (test_string);
    aiocb_write.aio_reqprio = 0;
/* 建立 info 信号 */
    aiocb_write.aio_sigevent.sigev_signo = EXAMPLE_SIG_NO;
    aiocb_write.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
    aiocb_write.aio_sigevent.sigev_value.sival_ptr =
                    (void *) &aiocb_write;
/* 初始化读操作 */
  if (aio_read (&aiocb_read) == -1)
      printf ("aioExampleSig: aio_read failed\n");
/* 验证正在执行读操作 */
    if (aio_error (&aiocb_read) == EINPROGRESS)
      printf ("aioExampleSig: read is still in progress\n");
/* 向管道中写数据 - 必须能够完成读操作 */
    printf ("aioExampleSig: getting ready to initiate the write\n");
    if (aio_write (&aiocb_write) == -1)
      printf ("aioExampleSig: aio_write failed\n");
/* 结束操作 */
    if (aio_return (&aiocb_read) == -1)
```

```
        printf ("aioExampleSig: aio_return for aiocb_read failed\n");
    else
        printf ("aioExampleSig: aio read message = %s\n",
                aiocb_read.aio_buf);
close (fd);
    return (OK);
    }
void mySigHandler
    (
    int       sig,
    struct siginfo * info,
    void *     pContext
    )

    {
    /* 输出读取的数据 */
    printf ("mySigHandler: Got signal for aio write\n");

    /* 结束写操作 */
    if (aio_return (info->si_value.sival_ptr) == -1)
        {
        printf ("mySigHandler: aio_return for aiocb_write failed\n");
        printErrno (0);
        }
    }
```

4.7　VxWorks 操作系统中的设备

VxWorks 操作系统中的 I/O 系统有很强的灵活性，允许用户为 7 种 I/O 功能指定设备驱动程序。VxWorks 操作系统中的所有设备驱动程序都具有前面几节所介绍的一般特点，但它们又有不同之处。本节将介绍这些不同之处。

表 4-6 列出了 VxWorks 操作系统中的驱动程序。

表 4-6　VxWorks 操作系统中的驱动程序

模 块 名 称	驱动程序介绍
ttyDrv	终端设备驱动程序
ptyDrv	伪终端设备驱动程序
pipeDrv	管道设备驱动程序
memDrv	伪存储设备驱动程序

续表

模 块 名 称	驱动程序介绍
nfsDrv	NFS 系统客户机驱动程序
netDrv	用于远程文件访问的网络驱动程序
ramDrv	用于创建 RAM 存储盘的 RAM 驱动程序
scsiLib	SCSI 接口库
-	其他与硬件有关的驱动程序

4.7.1 串行 I/O 设备（终端和伪终端设备）

VxWorks 操作系统提供了多种终端和伪终端设备驱动程序（tty 和 pty 驱动程序）。tty 驱动程序用于实际的终端设备，pty 驱动程序用于模拟的终端设备。这些伪终端设备将在诸如远程登录等应用中发挥作用。

VxWorks 操作系统中的 I/O 设备是一种缓冲型的串行字节流设备。每一个设备都有一个环形缓冲区用于输入和输出操作。从一个 tty 型设备中读取数据实际是从输入环形缓冲中提取数个字节数据；向一个 tty 型设备中写数据实际是向输出环形缓冲中加入数个字节数据。在操作系统初始化过程中创建设备时，指定每个环形缓冲区的大小。

⚠ **注意**：在本节的剩余部分中，术语 tty 用于表示 tty 和 pty 设备。

1. Tty 设备的功能选项

Tty 设备有很多功能选项可以影响设备的运行。这些参数是通过在设备选项字中的控制位来设定的，用户通过调用 ioctl( )函数执行 FIOSETOPTIONS 功能设置控制位（见“I/O 控制功能”）。下例是在 Tty 设备上设置除了 OPT_MON_TRAP 功能以外的所有功能。

```
status = ioctl (fd, FIOSETOPTIONS, OPT_TERMINAL & ~OPT_MON_TRAP);
```

表 4-7 列出了所有的可选项，在文件 ioLib.h 中定义了可选项的名称。用户欲了解更多信息，请查阅有关 tyLib 文件的内容。

表 4-7 Tty 设备中的可选项

名 称	说 明
OPT_LINE	选择线性传输模式（见“原始模式和线性模式”）
OPT_ECHO	向同一输出通道回应输入的字符
OPT_CRMOD	将输入的 RETURN 字符翻译成 NEWLINE 标志；将输出的 NEWLINE 标志翻译成 RETURN-LINEFEED 字符
OPT_TANDEM	响应软件流量控制字符 CTRL+Q 和 CTRL+S （XON and XOFF）
OPT_7_BIT	从所有输入字节中取出最高位

续表

名　称	说　明
OPT_MON_TRAP	使特殊 ROM 软中断监控程序字符有效，默认为 CTRL+X
OPT_ABORT	使特殊目标机 SHELL 程序终止字符，默认为 CTRL+Z（改选项只有在系统中使用 SHELL 程序时有效）
OPT_TERMINAL	将上述选项位设为‘1’
OPT_RAW	不设置上述选项位

2. 原始模式和线性模式传输

一个 tty 设备在工作中必定处于一种工作模式：或者是原始模式（非缓冲型），或者是线性模式。系统默认状态是原始模式，可以通过在设备选项字中设置 OPT_LINE 控制位将工作状态改为线性模式。

在原始模式下，每一个输入的字符一旦从输入设备中输入，接收方就立即获得该字符。在原始模式下用户在 tty 设备上进行读操作时，可以从输入环形缓冲区中读取尽可能多的字符（只是受用户输入环形缓冲区大小的限制）。输入方式只能由 tty 设备选项控制字改变。

在线性模式操作中，所有输入的字符先存储在缓冲区中，当输入 NEWLINE 字符后，整行字符（包括 NEWLINE 字符）同时导入环形存储区中。当从一个 tty 设备以线性模式进行读操作时，根据用户读操作缓冲区的容量，可以从环形存储区中读取两行的字符信息。输入的内容可以由以下特殊字符改变：CTRL+H（退格），CTRL+U（删除一行）和 CTRL+D（文件结束），这些特殊字符将在“tty 设备中的特殊字符”一节中介绍。

3. tty 设备中的特殊字符

当 tty 设备工作在线性模式下（即 OPT_LINE 控制位为‘1’），则下列的特殊字符才起作用：

- 退格字符（默认为 CTRL+H）的功能是连续删除当前行中的最末一个字符直至到该行的开始。该操作是响应空格字符后连续的退格字符。
- 删除一行字符（默认为 CTRL+U）的功能是删除当前行中的内容。
- 文件结束字符（默认为 CTRL+D）的功能是使当前行的内容导入输入环形缓冲区中，而不必等待输入 NEWLINE 或 EOF 字符。如果 EOF 字符出现在一行的第一个字符处，从该行中读取了零个字符，通常表示到达了文件的尾部。
- 如果如下的字符与操作选项中的控制位一起使用能产生一些特殊效果：
- 软件流量控制字符 CTRL+Q 和 CTRL+S（XON 和 XOFF）。当设备在输入字符中读取了 CTRL+S 时，会将相应通道的输出转为挂起状态，当继续收到 CTRL+S 字符后，又会继续输出操作。相反的，当 VxWorks 操作系统的输入缓冲区几乎装满时，向发送信息方输出 CTRL+S，通知对方挂起传输操作。当缓冲区容量足够大后，系统会输出 CTRL+Q，通知对方继续传输操作。通过设置选项控制位 OPT_TANDEM 可以使软件流量功能有效。

- ROM 存储区软中断监控程序字符（默认为 CTRL+X）。该字符是一个驻留 ROM 存储区的软中断监控程序。它的作用十分强大，所有正常的 VxWorks 操作系统中的工作都被挂起，计算机系统完全受监控程序控制。监控程序的功能不同，有些可以使 VxWorks 操作系统从中断处重新启动，有些则不能。通过设置选项控制位 OPT_MON_TRAP 可以使软中断监控程序功能有效。
- 目标机命令解释器（shell）终止字符（默认为 CTRL+C）。该字符可以在目标机 shell 程序进入不友好状态时重新启动目标机 shell 程序，比如程序调用了一个无效的信号量或进入了无限循环状态。通过设置选项控制位 OPT_ABORT 可以使目标机 shell 程序终止功能有效。

大部分特殊功能对应的特殊字符可以通过 tyLib 函数改变定义，见表 4-8 所列。

表 4-8　tty 设备中的特殊字符

特 殊 字 符	说　明	修改定义的程序
CTRL+H	退格（删除字符）	tyBackspaceSet( )
CTRL+U	删除一行	tyDeleteLineSet( )
CTRL+D	文件结束	tyEOFSet( )
CTRL+C	目标机 shell 程序终止	tyAbortSet( )
CTRL+X	软中断监控程序	tyMonitorTrapSet( )
CTRL+S	挂起输出	无
CTRL+Q	继续输出	无

4. I/O 操作控制函数

表 4-9 介绍了在文件 ioLib.h 中定义的 ioctl( )函数中控制 tty 设备的功能。用户欲了解更多信息，请查阅参考手册中有关 tyLib、ttyDrv 文件 and ioctl( )函数的内容。

表 4-9　tyLib 文件支持的 I/O 操作控制函数

功 能 名 称	功 能 说 明
FIOBAUDRATE	对指定参数设置传输速率
FIOCANCEL	取消一次读/写操作
FIOFLUSH	丢弃输入和输出缓冲区中的所有数据
FIOGETNAME	获取指定文件描述符对应的文件名
FIOGETOPTIONS	返回当前设备选项字的内容
FIONREAD	获取输入缓冲区中的未读字节数
FIONWRITE	获取输出缓冲区中的字节数
FIOSETOPTIONS	设置设备选项字的内容

⚠ **注意**：通过调用 ioctl( )函数执行 SIO_HW_OPTS_SET 功能可以改变设备驱动程序的硬件配置（比如，改变停止位的位数或奇偶校验的方式）。

以为大多数驱动程序并不支持这个命令，因此用户需要将配置参数添加到用户自己的 BSP 程序中的串行设备驱动程序中（存放于“installDir/target/src/drv/sio”）。如何使用这个配置命令取决于用户电路板上的串行芯片的具体功能。在文件“installDir/target/h/sioLib.h”中定义了配置硬件设备所需的 POSIX 标准的常量。

4.7.2　管道设备

管道是任务间通过 I/O 系统通信的一种虚拟设备。一个任务可以向管道写信息，其他的任务可以读取这些信息。由 pipeDrv 文件负责管理管道设备。管道设备使用核心信息队列进行实际信息传输操作。

1. 创建管道设备

通过调用管道创建函数来创建管道。

```
status = pipeDevCreate ("/pipe/name", maxMsgs, maxLength);
```

新建的管道同时可以容纳信息的数目由参数 maxMsgs 定义。当任务试图向一个已经达到最大信息数目的管道写新信息时，该任务将变为拥塞状态直到有一条信息被读出。管道中每条信息的最大长度由参数 maxLength 定义。试图写入一条长于定义的信息将会产生错误。

2. 从中断服务程序向管道写信息

VxWorks 操作系统允许中断服务程序像任务级程序一样采用同样方法向管道中写信息。许多 VxWorks 操作系统中的操作不能被用于中断服务程序，比如除了向管道以外不允许中断服务程序向其他输出设备输出数据。但中断服务程序可以通过管道与任务进行通信。这样就可以从中断服务程序中调用上述操作功能。中断服务程序通过调用 write( )函数向管道中写信息，而且任务和中断服务程序能向同一个管道写信息。但是如果目标管道空间已满，中断服务程序因为不能进入挂起状态而直接将待写信息丢弃。中断服务程序除了调用 write( )函数以外，不能使用其他 I/O 操作函数对管道进行操作。用户欲了解有关中断服务程序的内容，请查阅“2.6 中断服务代码：中断服务程序”一节的内容。

3. I/O 操作控制函数

表 4-10 列出了在 ioctl( )函数中控制管道设备的功能函数。在文件 ioLib.h 中列出了这些功能函数的定义。用户欲了解更多信息，请查阅用户手册中有关 pipeDrv 文件和 ioLib 文件中 ioctl( )函数的内容。

表 4-10 pipeDrv 文件支持的 I/O 操作控制函数

功 能 名 称	功 能 说 明
FIOFLUSH	丢弃管道中所有信息
FIOGETNAME	获取指定文件描述符对应的管道名
FIONMSGS	获取管道中的信息数目
FIONREAD	获取管道中第一条信息的字节数

4.7.3 伪存储设备

memDrv 驱动程序允许 I/O 系统访问存储器就像访问伪 I/O 设备一样。当这类设备新建时需指定存储区位置和容量。当需要在 VxWorks 操作系统的启动程序间保存数据或需要在不同 CPU 间共享数据时，上述特点很有用。memDrv 驱动程序不像 ramDrv 驱动程序一样需要一个文件系统。ramDrv 驱动程序必须依靠文件系统对存储区进行控制，而 memDrv 驱动程序则通过 I/O 操作函数就能够按照存储区的绝对地址进行高级别读/写操作。

1. 安装存储设备驱动程序

当 VxWorks 操作系统内核包含 INCLUDE_USR_MEMDRV 组件时，系统通过调用 memDrv( )函数自动地初始化存储设备驱动程序。例如

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

为设备分配的存储区是以基地址开始的绝对寻址存储区。参数 length 决定了存储区的容量。用户欲了解有关存储区驱动程序的信息，请查阅“VxWorks 操作系 API 参考手册”中关于 memDrv( )、memDevCreate( )和 memDevCreateDir( )函数的内容，也可以查阅“Tornado 工具参考手册”中关于主机工具 memdrvbuild 的内容。

2. I/O 操作控制函数

表 4-11 列出了 ioctl( )函数支持的存储区驱动程序。在文件 ioLib.h 中列出了这些功能函数的定义。

表 4-11 memDrv 文件支持的 I/O 操作控制函数

功 能 名 称	功 能 说 明
FIOSEEK	在文件中设置当前字节偏移量
FIOWHERE	返回当前在文件中的位置

用户欲了解更多信息，请查阅关于 memDrv 文件和 ioLib 文件中 ioctl( )函数的内容。

4.7.4　网络文件系统（NFS）设备

网络文件系统（NFS）设备可以通过 NFS 协议存取远程主机上的文件。NFS 协议负责指定从远程主机上读取文件的客户端软件和在远程主机上发送文件的服务器端软件。

驱动程序 nfsDrv 作为 VxWorks 操作系统中的 NFS 客户端，用于读取在网络上任何 NFS 服务器端上的文件。而且 VxWorks 操作系统允许用户启动一个 NFS 服务器向其他系统输出文件。用户可查阅“VxWorks 操作系统网络编程手册”中有关访问文件的内容。

用户可以使用 NFS 设备创建、打开、访问位于远程主机上的文件。就如同对本地磁盘上的文件系统进行操作一样。这种操作被称为网络透明化。

1. 从 VxWorks 操作系统中安装远程 NFS 文件系统

用户通过调用 nfsMount( )函数可以在本地创建一个 NFS 文件系统并创建一个相应的 I/O 设备，然后就可以访问位于远程主机上的 NFS 文件系统了。这个函数需要三个参数：（1）运行 NFS 文件系统服务器的主机名；（2）远程主机的文件系统名；（3）本地文件系统名。

在下例中，函数在主机‘mars’中安装名为‘/usr’的系统，本地系统名为‘/vxusr’。

```
nfsMount ("mars", "/usr", "/vxusr");
```

上例使用指定的本地名称（‘/vxusr’）创建了一个 VxWorks 操作系统 I/O 设备。如果本地名称指定为 NULL，则本地名称与远程文件系统名相同。

在安装了远程文件系统之后，访问远程文件系统就像访问本地文件系统一样。因此在上述函数调用完成之后，打开文件‘/vxusr/foo’实际上是打开主机‘mars’上的文件‘/usr/foo’。

服务器必须能够输出位于其上的远程文件系统，但是 NFS 服务器只能输出本地文件系统，用户需调用相关命令察看在服务器上哪些是本地文件系统。NFS 系统要求认证参数来确认用户进行远程访问。用户可以调用 nfsAuthUnixSet( )和 nfsAuthUnixPrompt( )函数设置这些参数。

用户要使系统支持 NFS 客户端的功能，需要在系统内核中加入 INCLUDE_NFS 组件。

在“VxWorks 操作系统网络编程指导”中详细介绍安装和输出 NFS 文件系统以及访问认证操作。用户也可查阅有关 nfsLib 和 nfsDrv 文件以及 SUN 微系统公司的有关 NFS 的文档。

2. NFS 用户端的 I/O 控制函数

表 4-12 列出了 ioctl( )函数支持 NFS 用户端设备的功能。在文件 ioLib.h 中列出了这些功能的定义。用户欲了解更多信息，请查阅有关 nfsDrv 文件和 ioLib 文件中 ioctl( )函数的内容。

表 4-12 nfsDrv 文件支持的 I/O 控制函数

功 能 名 称	功 能 说 明
FIOFSTATGET	获取文件状态信息
FIOGETNAME	获取指定文件描述符对应的文件名
FIONREAD	获取文件中未读取字节数
FIOREADDIR	读取下一个目录入口信息
FIOSEEK	在文件中设置当前字节偏移量
FIOSYNC	向远程 NFS 文件刷新数据
FIOWHERE	返回文件中当前字节偏移量

4.7.5 非 NFS 网络设备

VxWorks 操作系统也支持利用远程外壳协议（RSH）或文件传输协议（FTP）访问远程主机上的文件。这些网络设备需要 netDrv 驱动程序支持。当使用 FTP 或 RSH 协议打开一个远程文件时，整个文件都会被复制到本地存储区中。因此采用这种方式所能打开的文件大小受本地存储区容量的限制。对文件的读写操作实际上是对本地存储区中的文件进行操作。当关闭该文件时，如果文件被修改过，则会将本地存储区中的文件复制到远程主机中。

一般来讲，因为 NFS 设备不将等待操作的文件全部复制到本地存储区，因此它比 RSH 或 FTP 设备在使用上的灵活性和操作性更好。但是并不是所有主机上的操作系统都支持 NFS 协议。

1. 创建网络设备

用户要想使用 RSH 或 FTP 协议访问远程主机上的文件，则必须首先调用 netDevCreate( ) 函数创建一个网络设备。该函数需要三个参数：（1）网络设备的名称；（2）网络设备要访问的主机名；（3）进行网络通信所使用的协议，‘0’表示采用 RSH 协议，‘1’表示采用 FTP 协议。

在下例中，netDevCreate( )函数在远程主机‘mars’上创建了一个 RSH 设备‘mars:’。通常网络设备名是由远程主机名后加一个冒号（:）组成，如：

```
netDevCreate ("mars:", "mars", 0);
```

在一个网络设备上的文件可以像在本地磁盘上一样执行创建、打开等操作。因此，打开文件‘mars:/usr/foo’实际上是打开主机‘mars’上的‘/usr/foo’文件。

⚠ **注意**：一个网络设备允许用户访问远程主机上的任何文件或设备，而一个 NFS 文件系统只允许用户访问指定的远程文件系统。

如果通过 RSH 或 FTP 协议访问远程主机上的文件，那么必须在本地主机和远程主机上建立起允许访问和用户认证机制。在“VxWorks 操作系统的网络编程指导”中的“访问文件”一节中详细介绍了如何创建和配置网络设备，用户也可以参考有关 netDrv 文件的内容。

2. I/O 操作控制函数

除了 FIOSYNC 和 FIOREADDIR 功能以外，RSH 和 FTP 设备在调用 ioctl( )函数时具有与 NFS 设备一样的功能。在文件 ioLib.h 中定义了这些功能。用户欲了解更多信息，请查阅有关 netDrv 文件和 ioctl( )函数的内容。

4.7.6　CBIO 接口

核心缓冲区块存取输入/输出（CBIO）组件‘INCLUDE_CBIO_MAIN’提供了针对多种文件系统（dosFs 和 rawFs 等）的接口，用户在使用这些文件系统时需要在系统内核中包含该组件。

- INCLUDE_CBIO_DCACHE：提供磁盘缓冲区功能；
- INCLUDE_CBIO_DPART：提供磁盘卷分区功能；
- INCLUDE_CBIO_RAMDISK：提供 RAM 存储盘功能；

上述组件将在下面几节中介绍，用户还可以参考“VxWorks 操作系统 API 参考手册”中关于 cbioLib、dcacheCbio、dpartCbio 和 ramDiskCbio 文件的内容。

1. CBIO 磁盘缓冲区

CBIO 磁盘缓冲区用于使磁盘的 I/O 操作更有效而且能自动检测磁盘更换操作。

（1）磁盘 I/O 操作效率

磁盘缓冲区通过以下技术可以减少访问磁盘的次数以及提高对磁盘读写操作的速度。

- 在 RAM 存储区中保存了最近使用（MRU）的磁盘数据，因此用户从存储区而不是从磁盘中访问最经常使用的数据。
- 从磁盘上一次读取多少数据取决于磁盘缓冲区的预读特性，这是一个可以改变的磁盘缓冲区参数。
- 因为数据首先被写到存储区中，因此磁盘写操作的性能大大提高。而且通过磁盘缓冲区的另一个参数（延时写）使得系统有规律地更新数据。

由于大大减少了磁盘驱动器花费在寻找数据上的时间，因此可以提高系统操作性能。使得磁盘用于读/写数据的时间尽可能多。换句话说，用户可以更有效的使用磁盘的数据传输速率。

磁盘缓冲功能提供了一系列用户可以调整的参数（如延时写、预读、旁路阈值）来满足不同文件系统的工作量。

如果系统意外关闭，延时写功能可能会导致未写入磁盘的数据丢失。为了使这种情况

造成的损失最小，磁盘缓冲区会定期刷新磁盘。超过指定时间后，缓冲区中被修改的数据存入磁盘保存。

发生数据丢失时的最坏的情况是丢失所有被修改的数据，例如，假如设定的更新时间为 2 秒，这个时间既能存储足够多数据满足磁盘写操作优化的效果，也能保证在发生灾难情况时丢失的数据尽量少。用户可以将更新时间设为零秒，这样所有修改过的数据将立即存储到磁盘中。

磁盘缓冲区功能允许用户调节磁盘操作性能和存储区的消耗：为磁盘缓冲区分配了越多的存储区，读取数据的“命中率”越高，意味着提高了文件系统的性能。

（2）旁路阈值

另一个可以修改的参数是旁路阈值，它决定了在一次数据操作中不使用磁盘缓冲区功能的最小数据量。

当应用程序进行大量数据读/写操作时，用户数据缓冲区会绕过磁盘缓冲区与磁盘控制器之间直接传递数据。当用户同时使用旁路阈值功能和连续文件分配与访问功能时（调用 ioctl( )函数的 FIOCONTIG 功能和使用 open( )函数的 DOS_O_CONTIG 标志），可以产生同 rawFs 文件系统相同的操作性能。

用户欲了解磁盘缓冲区功能中可修改参数的详细情况，请查阅“VxWorks 操作系统 API 参考手册”中有关 dcacheDevTune( )函数的内容。用户欲了解磁盘缓冲区功能的全部情况，请查阅中有关 dcacheCbio 文件的内容。

（3）磁盘更换检测功能

磁盘缓冲区功能也提供了自动对磁盘更换操作进行检测的功能。磁盘更换检测功能基于两个先决条件：（1）磁盘的惟一性；（2）更换磁盘所需时间。

第一个假设是指磁带或磁盘存储器的第一个分区，即启动分区是惟一的。通常存储媒体在制造的时候已经被预先格式化，在启动分区中包含了一个 32 位由制造商设置的连续的 ID 号。当一个存储媒体的卷 ID 号有效时，dosFs 文件系统的格式化工具会保留这个 ID 号。当卷 ID 号无效时，格式化工具会根据当时的时间重新设置一个卷 ID 号。

第二个假设是指系统默认用户需要 2 秒钟时间更换一个磁盘。如果磁盘驱动器在 2 秒或更长的时间内处于停止状态，系统就会重新检查磁盘的启动分区，并与先前的纪录比较。如果比较的结果不一致，文件系统模块会发现这个变化，并采取相应的措施。首先，卸载原先的磁盘卷，然后将已打开的文件描述符标记为不可用。此时如果发现在驱动器中有一个磁盘，则安装新磁盘卷。该操作能够导致执行可选的自动一致性检测功能，用于检测由于在磁盘更新过程中拆除前一个磁盘而导致的任何结构上的不一致性（请查阅“VxWorks 操作系统 API 参考手册”中关于 dosFsDevCreate( )函数的内容）。

⚠ **注意**：*磁盘缓冲区功能是为旋转媒体存储设备设计的而不能用于 RAM 存储盘或 TrueFFS 文件系统盘中。*

2. CBIO 接口磁盘分区管理器

通常 VxWorks 操作系统的目标系统和 PC 机上运行的 Windows 操作系统共享固定磁盘和可移动存储器。因此 dosFs 文件系统支持 PC 机形式的磁盘分区功能。由两个功能模块提供分区管理机制：

dpartCbioLib ：该函数库提供了一种独立于各种特定分区表格式的一般分区处理机制；

usrFdiskPartLib ：该函数库负责解释大多数 PC 机上的分区表内容（主要由 FDISK 工具创建的分区表）。它是以源程序形式发布的，因此可以根据分区表格式的不同作修改。而且函数库中还有创建 PC 形式分区表的软件（与 FDISK 相似）。本版软件支持的最多分区数是 4 个。

DpartCbioLib 模块通过调用用户提供的分区表解释函数处理固定磁盘和可移动磁盘上的分区。当更换可移动磁盘时，系统调用该函数。当系统初始化时，用户必须指定在驱动器上希望建立的分区个数。并且为每一个分区设定一个逻辑驱动器名。如果用户在系统中插入的可移动磁盘上的分区个数大于用户指定的个数，则系统只能访问可移动存储器上指定个数的分区，其余的分区都不能访问。

3. CBIO 接口 RAM 存储盘

在一些应用中，用户希望文件系统虽然在没有磁盘或传统存储媒体时也可以组织和访问数据。CBIO 接口的 RAM 存储盘工具使得用户可以利用文件系统访问 RAM 存储区中的数据。RAM 存储盘既能在易失性存储器也能在非易失性存储器中创建。

⚠ **注意**：ramDiskCbio 函数库通过 CBIO 接口实现 RAM 存储盘，而 ramDrv 函数库通过使用 BLK_DEV 结构实现 RAM 存储盘。

用户欲了解更多信息，请查阅“VxWorks 操作系统 API 参考手册”中关于 ramDiskCbio 函数库的内容。

4. CBIO 接口设备支持的 I/O 控制函数

表 4-13 列出了 ioctl( )函数所支持的 CBIO 接口设备的功能函数。

表 4-13　CBIO 接口设备支持的 I/O 控制函数

功 能 名 称	功 能 说 明
CBIO_RESET	使设备重启
CBIO_STATUS_CHK	检测设备状态
CBIO_DEVICE_LOCK	阻止拆卸磁盘
CBIO_DEVICE_UNLOCK	允许拆卸磁盘
CBIO_DEVICE_EJECT	卸载设备

续表

功 能 名 称	功 能 说 明
CBIO_CACHE_FLUSH	刷新缓冲区
CBIO_CACHE_INVAL	刷新磁盘并使磁盘失效
CBIO_CACHE_NEWBLK	分配一个临时的存储块

4.7.7 块存取设备

块存取设备是由一系列可独立访问的数据块组织起来的设备。最普通的块存取设备就是磁盘。在 VxWorks 操作系统中术语“块”表示在设备中可寻址的最小单位，对于大多数磁盘设备，与 VxWorks 操作系统中“块”相当的术语是“扇区”。

VxWorks 操作系统对于块存取设备的接口与其他 I/O 设备的接口稍有不同。块存取设备不是与 I/O 系统相连，而是通过一些底层驱动程序与某一文件系统相连，而文件系统再与 I/O 系统通信。这种通信形式使得某一底层驱动程序可以被不同的文件系统调用，进而减少了驱动程序必须支持的 I/O 操作函数的数目。块存取设备的底层驱动程序内部应用将在“4.9.4 块存取设备”一节中介绍。

1. 块存取设备上的文件系统

VxWorks 操作系统提供了与 MS-DOS 文件系统兼容的文件系统来控制块存取设备，而且 VxWorks 操作系统为简单原始磁盘文件系统、SCSI 接口磁带设备、CD-ROM 接口设备和闪存设备提供了多个函数库。将在本书的“第 5 章本地文件系统”中介绍上述文件系统的用法（同时还将介绍目录服务器文件系统）。用户可参考“VxWorks 操作系统 API 参考手册”中有关 dosFsLib、rawFsLib、tapeFsLib、cdromFsLib 和 tffsDrv 文件的内容。

2. 块存取设备：RAM 磁盘驱动器

由 ramDrv 函数库实现的 RAM 磁盘驱动器是在存储区中模拟了磁盘设备。当调用 ramDevCreate( )函数创建 RAM 磁盘设备时需指定存储区的地址以及模拟磁盘的容量。用户可以多次调用该函数以便创建多个 RAM 磁盘设备。

用于 RAM 磁盘设备的存储区可以预先分配好并将地址传递给 ramDevCreate( )函数，也可以通过调用 malloc( )函数从系统存储区中自动分配出一块区域。在创建 RAM 磁盘设备后，用户必须调用文件系统的设备创建函数和格式化函数，为新设备指定一个名称和相应的文件系统（如 dosFs 或 rawFs 文件系统）。例如对于 dosFs 文件系统可以调用 dosFsDevCreate( ) 和 dosFsVolFormat( )函数。有关设备的信息通过 BLK_DEV 结构传递给相应的文件系统，指向该结构的指针由 RAM 磁盘设备的创建函数产生。

在下面的例子中，通过自动分配存储区系统创建了一个 200KB 容量的 RAM 磁盘设备，每个扇区 512 个字节，只有一个磁道，没有扇区偏移量。该设备的名称是 DEV1:，并且采用 dosFs 文件系统。

```
BLK_DEV             *pBlkDev;
DOS_VOL_DESC         *pVolDesc;
pBlkDev = ramDevCreate (0, 512, 400, 400, 0);
if (dosFsDevCreate ("DEV1:", pBlkDev, 20, NULL) == ERROR)
{
   printErrno( );
};

/* 如果用户想格式化磁盘执行如下操作: */
if (dosFsVolFormat ("DEV1:", 2, NULL) == ERROR)
{
   printErrno( );
}
```

如果 RAM 磁盘中已经包括了一个磁盘映像，那么 ramDevCreate( )函数的第一个参数是存储区中 RAM 磁盘的地址，而且格式化参数也必须与已有的磁盘映像一致。例如：

```
pBlkDev = ramDevCreate (0xc0000, 512, 400, 400, 0);
if (dosFsDevCreate ("DEV1:", pBlkDev, 20, NULL) == ERROR)
{
   printErrno( );
}
```

在上例中，只需调用 dosFsDevCreate( )函数。因为在磁盘上已经建立了文件系统，并不需要重新格式化。如果创建 RAM 磁盘的地址与上一次 VxWorks 操作系统启动时的地址相同，那么 RAM 磁盘中的内容不变。

⚠ **注意**：ramDiskCbio 函数库通过 CBIO 接口控制 RAM 磁盘，而 ramDrv 函数库通过 BLK_DEV 结构接口控制 RAM 磁盘。

用户欲了解有关控制 RAM 磁盘驱动器更多的信息，请查阅关于 ramDrv 文件的内容。用户欲了解有关文件系统的信息，请查阅“第 5 章　本地文件系统”中的内容。

3. SCSI 接口驱动器

SCSI 标准外设接口允许系统连接多种硬盘、光盘、软盘、磁带设备和 CD-ROM 设备。SCSI 接口块存取设备驱动器与 dosFs 文件系统库兼容，而且对于目标机的配置还有如下优点：

非网络环境下本地大容量数据存储；

比以太网更快的 I/O 接口流量；

虽然 VxWorks 操作系统仍可以配置原先的 SCSI 接口功能（现在称为 SCSI-1 接口），但现在已经用 SCSI-2 接口标准代替原先的 SCSI 接口。VxWorks 操作系统可以通过 SCSI-2 接口标准控制符合 SCSI-1 接口的应用，比如在 SCSI-1 接口标准下建立的文件系统。但如果应用程序直接使用 scsiLib.h 文件中定义的符合 SCSI-1 接口标准的数据结构，为了与 SCSI-2 接口标准兼容则需重新修改和再编译。

VxWorks 操作系统中的 SCSI 接口包括两个模块：一个是与设备无关的 SCSI 接口；另一个是支持特定 SCSI 接口设备的控制器。ScsiLib 函数库中的函数用于实现与设备无关的 SCSI 接口，由特定 SCSI 接口设备的函数库负责 SCSI 接口设备控制器的配置。在 sysLib.c 文件中提供了对单独目标设备额外的支持。

（1）配置 SCSI 接口驱动器

表 4-14 列出了与 SCSI 接口驱动器有关的组件。

表 4-14　SCSI 接口和相关的组件

组 件 名 称	组 件 说 明
INCLUDE_SCSI	使系统支持 SCSI 接口
INCLUDE_SCSI2	使系统支持 SCSI-2 接口的扩展功能
INCLUDE_SCSI_DMA	使 SCSI 接口具有 DMA 传输功能
INCLUDE_SCSI_BOOT	允许系统从 SCSI 接口设备上启动
SCSI_AUTO_CONFIG	在 SCSI 接口总线上自动配置和定位所有目标设备
INCLUDE_DOSFS	使系统支持 DOS 文件系统
INCLUDE_TAPEFS	使系统支持磁带文件系统
INCLUDE_CDROMFS	使系统支持 CD-ROM 文件系统

用户可以通过将 INCLUDE_SCSI 组件添加到系统内核中，使 VxWorks 操作系统支持 SCSI-1 接口标准。如果用户要使用 SCSI-2 接口标准，还需添加 INCLUDE_SCSI。

在不同的 BSP 程序中相应地定义了自动配置 SCSI 接口、DMA 传输功能和从 SCSI 接口设备上启动的方法。如果用户需要改变这些配置，需要参考 sysScsiConfig( )函数和文件“installDir/target/src/config/usrScsi.c”。

警告：使 VxWorks 操作系统支持 SCSI-2 接口标准会大大增加程序映像所需的存储区容量。

（2）配置 SCSI 接口总线 ID 号

必须为支持 SCSI-2 接口标准用户板的 SCSI 接口启动器定义一个独一无二的 SCSI 接口总线 ID 号。SCSI-1 接口标准在同一时间内只支持一个总线启动器，并将它的 SCSI 接口总线 ID 号设为 7。SCSI-2 接口标准支持多个总线启动器，最多达到 8 个。因此必须为每一

个总线启动器设定一个惟一的 ID 号。sysScsi.c 文件中的 sysScsiInit( )函数从 0~7 的数字中选择一个作为驱动器初始化程序（如 ncr710CtrlInitScsi2( )函数）的参数。用户欲了解更多内容，请查阅有关驱动器初始化程序的内容。如果一个 SCSI 总线连接多个用户板，而且所有用户板使用同一个 BSP 程序，那么用户必须为每一块用户板指定一个惟一的 SCSI 接口总线 ID 号，并将不同版本的 BSP 程序分别编译。

（3）调整 SCSI 总线启动时 ROM 存储区容量

如果系统内核中包含 INCLUDE_SCSI_BOOT 模块，则一些用户板可能会需要更大的 ROM 存储区。

（4）SCSI 总线子系统的结构

SCSI 总线子系统支持符合 SCSI-1 标准协议和 SCSI-2 标准协议的函数库和驱动程序，包括下列 6 个与 SCSI 总线控制器无关的函数库：

- scsiLib：根据板级支持包（BSP）程序的配置，提供将 SCSI 访问请求与 SCSI-1 标准函数库（scsi1Lib）或 SCSI-2 标准函数库（scsi2Lib）连接的功能。
- scsi1Lib：当系统内核中只包含 INCLUDE_SCSI 组件时，提供符合 SCSI-1 标准的函数和接口（见“配置 SCSI 接口驱动器”）。
- scsi2Lib：提供符合 SCSI-2 标准函数库以及创建和删除物理设备的程序。
- scsiCommonLib：提供符合所有 SCSI 接口设备的命令。
- scsiDirectLib：提供用于直接访问设备（磁盘）的函数和命令。
- scsiSeqLib：提供用于顺序访问块存取设备（磁带）的函数和命令。

符合 SCSI-2 标准并与控制器无关的功能函数分别位于 scsi2Lib、scsiCommonLib、scsiDirectLib、scsiSeqLib 函数库中。应用程序可以直接访问这些 SCSI-2 标准函数库的接口。但 scsiSeqLib 函数库只用于与 tapeFs 文件系统有关的应用，而 scsiDirectLib 函数库只用于与 dosFs 和 rawFs 文件系统有关的应用。符合 SCSI-1 标准的应用程序可以在 SCSI-2 标准下直接使用，但符合 SCSI-1 标准的设备驱动程序则不能。

如果 VxWorks 操作系统的目标板使用 SCSI 接口控制器，则它还需要一个与控制器有关的设备驱动程序。这些驱动程序和与控制器无关的 SCSI 函数库一起，提供包含在与控制器相关的函数库中的控制器配置和初始化函数。例如，设备驱动程序库 wd33c93Lib、wd33c93Lib1 和 wd33c93Lib2 支持西部数据公司的 WD33C93 系列 SCSI 接口控制器。符合 SCSI-1 标准的函数（wd33c93CtrlCreate( )）和符合 SCSI-2 标准的函数（wd33c93CtrlCreateScsi2( )）为了简化配置过程而分别封装于不同的函数库中。在 sysLib.c 文件中还包含有一些支持单目标板的程序。

（5）启动和初始化

当 VxWorks 操作系统内核中包含 INCLUDE_SCSI 组件后就可以支持 SCSI 标准接口。系统启动程序通过调用 sysScsiInit( )和 usrScsiConfig( )函数初始化 SCSI 接口。sysScsiInit( )函数负责初始化 SCSI 接口控制器以及启动中断处理程序。在“installDir/target/src/config/usrScsi.c”文件中的 usrScsiConfig( )函数负责配置物理设备。该文件包括一个系统配置过程的一个例子，程序中包括：

- 调用 scsiPhysDevCreate( )函数定义物理设备；
- 调用 scsiBlkDevCreate( )函数创建逻辑分区；
- 调用 dosFsDevCreate( )函数设置文件系统；

如果用户没有使用 SCSI_AUTO_CONFIG 参数，则需修改 usrScsiConfig( )函数来反映实际的配置情况。用户欲了解有关上述函数的更多信息，请查阅关于 scsiPhysDevCreate( )、scsiBlkDevCreate( )、dosFsDevCreate( )和 dosFsVolFormat( )函数的相关内容。

（6）与设备有关的配置参数

SCSI 接口函数库具有以下默认功能：

- SCSI 接口信息传递；
- 解除连接；
- 最小周期和最大 REQ/ACK 延迟；
- 命令标记队列；
- 长字节数据传输；

如果设备共用上述默认功能，用户程序不必设置与设备有关的参数。但如果用户需要配置一个具有与默认功能不同的设备，应该使用 scsiTargetOptionsSet( )函数修改参数值。如下所示，这些参数是 SCSI_OPTIONS 结构中的一些字段。在 scsi2Lib.h 文件中定义了 SCSI_OPTIONS 结构。用户为了适应特定的 SCSI 设备和应用程序可以修改一些或全部参数。

```
typedef struct                  /* SCSI_OPTIONS - 用户可修改的选项     */
    {
    UINT     selTimeOut;        /* 设备选择的超时值(us)                */
    BOOL     messages;          /* FALSE 标记表示不使用 SCSI 信息      */
    BOOL     disconnect;        /* FALSE 标记表示不使用解除连接        */
    UINT8     maxOffset;        /* 最大同步传输延迟(0 表示异步传输)    */
    UINT8     minPeriod;        /* 最小同步传输周期(x 4 ns)            */
    SCSI_TAG_TYPE tagType;      /*  默认的命令标记                     */
    UINT     maxTags;           /* 最大的命令标记数(0 表示无)          */
    UINT8     xferWidth;        /* SCSI 单元的长字节数据传输宽度       */
    } SCSI_OPTIONS;
```

VxWorks 操作系统支持很多种 SCSI 设备，每一种设备都有各自符合 SCSI-2 接口标准的特点。用户可以通过定义一个 SCSI_OPTIONS 结构，并设置其中的相关字段来设置与设备有关的选项。在设置完相应的字段之后，用户可以调用 scsiTargetOptionsSet( )函数使修改生效。例 4-5 演示了如何使用 SCSI_OPTIONS 结构配置一个设备。

用户需要在初始化 SCSI 子系统之后，在初始化 SCSI 物理设备之前调用 scsiTargetOptionsSet( )函数。用户欲了解有关配置选项的设置和使用方面的信息，请查阅关于 scsiTargetOptionsSet( )函数的内容。

警告： *在初始化物理设备之后调用* scsiTargetOptionsSet( )*函数会导致异常错误。*

SCSI 子系统将每一次 SCSI 命令请求当作一个 SCSI 事件处理。这种处理方式要求 SCSI 子系统选择一个设备。不同的 SCSI 设备对于系统选择的相应时间不同，一些情况下，需要对 selTimeOut 字段的默认值作适当修改。

如果某一个设备不支持 SCSI 信息传递功能，则将表示信息传递功能的布尔字段设置为 FALSE 标志。同样，如果一个设备不支持断开/再连接功能，则表示该方式的布尔字段设置为 FALSE 标志。

SCSI 子系统自动地尝试调整同步数据传输参数。但是如果一个 SCSI 设备不支持同步数据传输功能，用户需要将 maxOffset 字段的值设为 '0'。通常，SCSI 子系统数图调节 VxWorks 操作系统的目标机上的控制器所支持的最小数据传输周期和最大 REQ/ACK 延迟功能的参数，这样可以使两个设备间数据传输速度达到最大。但数据传输速度还受一些电器特性的影响，比如电缆长度、电缆的质量、设备终端的性能等。因此为了达到更快的数据传输速度用户有可能需要减小 maxOffset 或 minPeriod 字段的值。

TagType 字段使用如下宏函数定义了用户所希望使用的命令标记队列的方式：

- SCSI_TAG_UNTAGGED
- SCSI_TAG_SIMPLE
- SCSI_TAG_ORDERED
- SCSI_TAG_HEAD_OF_QUEUE

用户欲了解有关可以使用的命令标记队列方式的信息，请查阅“小型计算机接口（SCSI-2）”中关于 ANSI X3T9-I/O 接口说明中的内容。

MaxTags 字段负责设置对于特定 SCSI 设备的有效命令标记的最大数目。

SCSI 子系统在初始化时会自动调节与目标设备进行长字节数据传输的参数值。就像 SCSI 中 ANSI 规范中所说明的，在调节长字节数据传输的参数值通常发生在同步数据传输参数设置之前。因为调节长字节数据传输的参数值会将任何先前设置的同步数据传输参数复位。但是如果一个 SCSI 设备不支持长字节数据传输而且在初始化该设备时出现问题，则必须将 xferWidth 字段的值设为 '0'。默认情况下，SCSI 子系统会调节 VxWorks 操作系统目标板上 SCSI 控制器所支持的最大数据传输宽度，以便使两个设备间的默认数据传输速度达到最大值。用户欲了解有关实际函数调用方式的信息，请查阅关于 scsiTargetOptionsSet( ) 函数的内容。

（7）SCSI 配置实例

下面的几个例子介绍了几种不同 SCSI 设备的配置方法。例 4-4 介绍了一个简单块存取设备的配置方法。例 4-5 介绍了如何选择特殊选项以及如何使用 scsiTargetOptionsSet( )函数。例 4-6 介绍了如何配置一个磁带设备以及配置磁带文件系统。例 4-7 介绍了如何配置一个

SCSI 设备进行同步数据传输。例 4-8 介绍了如何配置 SCSI 总线 ID 号。这些例子既可以在 usrScsiConfig( )函数中使用，也可以在用户自己定义的配置程序中使用。

例 4-4：配置 SCSI 接口驱动器

在下面的例子中，通过修改 usrScsiConfig( )函数反映新的系统配置。在新配置中包括一个 SCSI 接口磁盘，它的总线 ID 号是 4，逻辑单元号（LUN）是 0，在磁盘中使用 dosFs 文件系统（分配的磁盘容量为 0x20000 块）和 rawFs 文件系统（占据剩余的磁盘容量）。在 usrScsiConfig( )函数中反映出这些改动。

```
/* 将一个温式硬盘配置成总线 ID 号为 4,逻辑单元号为 0 */
 if ((pSpd40 = scsiPhysDevCreate (pSysScsiCtrl, 4, 0, 0, NONE, 0, 0, 0))
       == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    }
else
    {
    /* 创建块存取设备 - 一个使用 dosFs 文件系统,另一个使用 rawFs 文件系统 */

    if (((pSbd0 = scsiBlkDevCreate (pSpd40, 0x20000, 0)) == NULL) ||
        ((pSbd1 = scsiBlkDevCreate (pSpd40, 0, 0x20000)) == NULL))
        {
        return (ERROR);
        }

    /* 初始化 dosFs 文件系统和 rawFs 文件系统 */

    if ((dosFsDevInit ("/sd0/", pSbd0, NULL) == NULL) ||
        (rawFsDevInit ("/sd1/", pSbd1) == NULL))
        {
        return (ERROR);
        }
    }
```

如果在用户的配置过程中存在问题，可以在 usrScsiConfig( )函数的开始处加入如下几行程序，这样可以获得 SCSI 总线活动的一些信息。

```
#if FALSE
scsiDebug = TRUE;
scsiIntsDebug = TRUE;
#endif
```

不要在函数内部声明全局变量 scsiDebug 和 scsiIntsDebug，因为用户需要从 shell 程序中设置或重新设置这些变量。请参考“Tornado 用户手册”中关于 shell 程序的内容。

例 4-5：将 SCSI 接口磁盘驱动器配置成支持异步数据传输功能和不支持命令标记队列功能。

在本例中，程序将一个 SCSI 接口磁盘设备配置成不支持同步数据传输功能和命令标记队列功能。通过调用 scsiTargetOptionsSet( )函数关闭这些功能。磁盘设备的 SCSI ID 号为 2，LUN 号为 0。

```
int             which;
SCSI_OPTIONS     option;
int             devBusId;

devBusId = 2;
which = SCSI_SET_OPT_XFER_PARAMS | SCSI_SET_OPT_TAG_PARAMS;
option.maxOffset = SCSI_SYNC_XFER_ASYNC_OFFSET;
/* => 值为 0,在 scsi2Lib.h 文件中定义 */
option.minPeriod = SCSI_SYNC_XFER_MIN_PERIOD;  /* 在 scsi2Lib.h 文件中定义 */
option.tagType = SCSI_TAG_UNTAGGED;      /* 在 scsi2Lib.h 文件中定义 */
option.maxTag = SCSI_MAX_TAGS;

if (scsiTargetOptionsSet (pSysScsiCtrl, devBusId, &option, which) == ERROR)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: could not set options\n", 0, 0, 0, 0,
        0, 0);
    return (ERROR);
    }

/* 将 SCSI 磁盘设备配置成总线 ID 号设为 devBusI,逻辑单元号设为 0。 */

if ((pSpd20 = scsiPhysDevCreate (pSysScsiCtrl, devBusId, 0, 0, NONE, 0, 0,
        0)) == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    return (ERROR);
    }
```

例 4-6：配置磁带设备

用户可以通过使用 scsiCommonLib 函数库中的通用命令和 scsiSeqLib 函数库中的顺序存储设备命令控制 SCSI 磁带设备。这些命令使用一个指向在 seqIo.h 文件中定义的 SCSI 顺序存储设备结构 SEQ_DEV 的指针。用户欲了解有关控制 SCSI 磁带设备的信息，请查阅

关于这些函数库的内容。

在程序中，将 SCSI 磁带设备的总线 ID 号设为 5，LUN 号设为 0。而且在程序中包括创建物理设备指针、设置顺序存储设备和初始化 tapeFs 文件系统设备的命令。

```
/* 将 Exabyte 8mm 磁带设备配置成总线 ID 号为 5,逻辑单元号为 0  */
if ((pSpd50 = scsiPhysDevCreate (pSysScsiCtrl, 5, 0, 0, NONE, 0, 0, 0))
       == (SCSI_PHYS_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n");
    return (ERROR);
    }

/* 将该物理设备配制成顺序存储设备 */
if ((pSd0 = scsiSeqDevCreate (pSpd50)) == (SEQ_DEV *) NULL)
    {
    SCSI_DEBUG_MSG ("usrScsiConfig: scsiSeqDevCreate failed.\n");
       return (ERROR);
    }

/* 配置磁带设备 */
pTapeConfig = (TAPE_CONFIG *) calloc (sizeof (TAPE_CONFIG), 1);
pTapeConfig->reWind = TRUE;   /* 这是一个可倒带的设备 */
pTapeConfig->blkSize = 512;   /* 使用 512 字节固定块存取单元 */

/* 初始化 tapeFs 文件系统设备 */
if (tapeFsDevInit ("/tape1", pSd0, pTapeConfig) == NULL)
    {
    return (ERROR);
    }

/* 直接使用 scsiSeqLib 库接口执行倒带操作 */
if (scsiReWind (pSd0) == ERROR)
    {
    return (ERROR);
    }
```

例 4-7：将 SCSI 磁盘设备配置成同步数据传输模式（具有非默认偏移量和周期值）

在本例中将一个 SCSI 接口磁盘设备配置成支持同步数据传输模式，其中偏移量和周期值由用户设定。设置的周期值是 25，因为 SCSI 接口时间单位为 4ns，故实际周期应为 4×25=100ns；同步偏移量设为 2。注意用户应根据各自的硬件环境调整参数值。

```
int                 which;
SCSI_OPTIONS          option;
int                 devBusId;

devBusId = 2;

    which = SCSI_SET_IPT_XFER_PARAMS;
    option.maxOffset = 2;
    option.minPeriod = 25;

    if (scsiTargetOptionsSet (pSysScsiCtrl, devBusId &option, which) ==
        ERROR)
        {
        SCSI_DEBUG_MSG ("usrScsiConfig: could not set options\n",
                        0, 0, 0, 0, 0, 0)
        return (ERROR);
        }

    /* 将 SCSI 磁盘设备配置成总线 ID 号为 devBusId ,逻辑单元号为 0。*/

    if ((pSpd20 = scsiPhysDevCreate (pSysScsiCtrl, devBusId, 0, 0, NONE,
                                     0, 0, 0)) == (SCSI_PHYS_DEV *) NULL)
        {
        SCSI_DEBUG_MSG ("usrScsiConfig: scsiPhysDevCreate failed.\n")
        return (ERROR);
        }
```

例 4-8：改变 SCSI 接口控制器的总线 ID 号

用户可以通过修改 sysScsi.c 文件中的 sysScsiInit( )函数，从而改变 SCSI 接口控制器的总线 ID 号。通过调用 xxxCtrlInitScsi2( )函数可以将 SCSI 接口总线 ID 号设为 0~7 之间的任意值。xxx 表示 SCSI 接口控制器的名称。SCSI 接口控制器总线 ID 号的默认值为 7。

疑难解答

■　SCSI-1 接口标准与 SCSI-2 接口标准的不兼容性

符合 SCSI-1 接口标准的应用程序可能会在 SCSI-2 接口标准的环境下不能正常工作。因为在 scsi2Lib.h 文件中定义的数据结构，如 SCSI_TRANSACTION 和 SCSI_PHYS_DEV 已经被改变了。这种不兼容性只有当直接使用这些数据结构时才会发生。

发生这种情况时，用户可以选择将系统配置成只支持 SCSI-1 接口标准或根据 scsi2Lib.h 文件中定义的数据结构修改应用程序。为了在已改变的数据结构中增加几个新字段，有些应用程序只需重新编译，而有些应用程序则需修改后再编译。

■ SCSI 接口总线故障

可能有很多原因会导致用户的 SCSI 接口总线处于挂起状态，下面介绍几种通常的情况：

用户的传输电缆有故障，这是最常出现的原因；

用户的传输电缆超过了 SCSI-2 接口标准中规定的最长距离（6 米），导致 SCSI 接口信号的电器性能改变；

用户的传输电缆终端处理不正确。在 SCSI-2 接口标准中规定可以在传输电缆两端提供终端电源；

最小数据传输周期过小和最大 REQ/ACK 延迟时间过长。用户可以调用 scsiTargetOptionsSet( )函数调整这些参数；

驱动程序试图对一个不支持长字节传输的设备调整相应参数。因为设备不支持长字节传输功能，因此驱动程序不能调整这些不匹配的参数。用户可以调用 scsiTargetOptionsSet( )函数为特定的 SCSI 接口设备设置合适的 xferWidth 字段值。

4.7.8 套接字

在 VxWorks 操作系统中，套接字是网络通信的基础。一个套接字是任务间通信的一个端点，数据从一个套接字发往另一个套接字。套接字不能通过标准 I/O 操作函数创建或打开，而是通过调用 sockLib 函数库中的 socket( )函数创建，并调用函数库中的其他函数进行连接和访问。但是在流式套接字（使用 TCP 协议）创建和连接之后，可以像标准 I/O 设备一样，调用 read( )、write( )、ioctl( )和 close( )函数进行访问。通过 socket( )函数返回的套接字句柄相当于 I/O 系统中的文件描述符（fd）。

VxWorks 操作系统中套接字的程序在源程序上与符合 BSD 4.4 标准的 UNIX 操作系统的套接字函数和 Windows 操作系统的套接字网络标准（Winsock 1.1）兼容。在“VxWorks 操作系统网络编程指南”中的“网络 API 函数”一节中将介绍这些函数。

4.8 VxWorks 操作系统与主机操作系统中 I/O 系统的区别

在 VxWorks 操作系统中的大多数普通的 I/O 操作与 UNIX 和 Windows 操作系统中的 I/O 操作在源程序上是完全兼容的，但也有如下不同之处：

■ 设备配置方式

在 VxWorks 操作系统中，设备驱动程序可以动态地加载和卸载。

■ 文件描述符

在 UNIX 和 Windows 操作系统中，文件描述符（fd）对于每一个进程来说是惟一的；在 VxWorks 操作系统中，文件描述符（fd）是可以被任何任务访问的全局变量。但标准输入、标准输出和标准错误输出除外，它们可由任务指定。

■　I/O 控制

传递给 ioctl( )函数的参数在 UNIX 操作系统和 VxWorks 操作系统中可以不一样。

■　驱动程序

在 UNIX 操作系统中，设备驱动程序是运行在系统模式下而且不能被抢占。在 VxWorks 操作系统中，设备驱动程序可以被先占有，因为设备驱动程序运行在调用它们的任务上下文中。

4.9　内 部 结 构

VxWorks 操作系统与其他大多数操作系统的区别在于对用户 I/O 请求的响应方式上。VxWorks 操作系统将这个过程分为与设备无关的 I/O 系统和设备驱动程序本身。

在多数操作系统中，设备驱动程序提供一些函数完成一些低层的 I/O 操作，如从/向一个字符设备中读/写一系列字节。高层的协议，如面向字符设备的通信协议，则运行在 I/O 系统中与设备无关的部分上。用户的请求在传递给驱动程序之前主要由 I/O 系统处理。

这种响应方式主要是为了使驱动程序的使用更加容易，并使得各种设备的运行方式尽可能相同。但它有一些缺点，驱动程序的编写者经常因为现有的 I/O 系统不能提供一些可以选择的通信协议而遇到困难。在实时操作系统中，有时因为对某个设备的通信流量严格要求或设备与标准模式发生冲突而需要完全避开标准协议。

在 VxWorks 操作系统中，用户的请求在传递给设备驱动程序之前，I/O 系统将进行尽可能少地处理。VxWorks 操作系统中的 I/O 系统只起到了将用户请求与相应的驱动程序相连的开关作用。每个驱动程序能够以适合设备的方式处理原始的用户请求，而且驱动程序的编写者还能够通过调用一些高级的子程序库在字符设备和块存取设备上使用标准协议。因此驱动程序编写者既可以很容易地写出适用于大多数设备的标准驱动程序（只包括一些与设备有关的代码），也可以根据需要以非标准方式响应用户请求。VxWorks 操作系统中的 I/O 系统能很好的支持这两方面应用。

VxWorks 操作系统中的 I/O 系统中有两种基本设备类型：块存取设备和字符型设备（又叫非块存取设备，见图 4-8 所示）。块存取设备用于存储文件系统。它们是一种随机存储设备，数据以块为单位进行传输，块存取设备有硬盘、软盘等。字符型设备是那些不按数据块为单位进行存储的设备，如串行接口、图形输入设备等。

如前几节中所讲，VxWorks 操作系统中的 I/O 系统包括三种主要的基本组件：驱动程序、设备和文件。以下几节将详细介绍这三种基本组件，重点主要放在字符型设备驱动程序及其在块存取设备上的应用。因为块存取设备的驱动程序必须与 VxWorks 操作系统的文

件系统同时使用，因此它们在组织结构上有些不同，见“4.9.4 块存取设备”。

> ⚠ **注意**：本节的介绍主要是为了解释 VxWorks 操作系统中 I/O 设备的结构，以及强调在为 VxWorks 操作系统编写 I/O 设备驱动程序时应注意的事项，并不是完整地介绍如何编写设备驱动程序。用户欲了解有关这方面更详细的信息，请查阅“VxWorks 操作系统 BSP 编程手册”。

例 4-9 介绍了一个假定的驱动程序的简化的程序代码，这个例子将作为以后讨论的基础。该程序是一个典型的面向字符设备的驱动程序。

在 VxWorks 操作系统中，每一个驱动程序都有一个惟一的、简洁的缩写形式，如 net 或 tty，作为该驱动程序的前缀。下面驱动程序例子的缩写是 xx。

例 4-9：假设的驱动程序

```
/*
 * xxDrv - 驱动器初始化函数
 * xxDrv( ) 函数初始化驱动器。它通过调用 iosDrvInstall 安装驱动器。
 * 分配数据结构,连接中断服务程序,初始化硬件 */

STATUS xxDrv ( )
  {
  xxDrvNum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl);
  (void) intConnect (intvec, xxInterrupt, ...);
  ...
  }
/*************************************************************************
*
 * xxDevCreate - 设备创建函数
 *
 * 调用该函数可以添加设备。
 * 其他与驱动器有关的参数包括缓冲区容量、设备地址。程序通过调用 iosDevAdd 将设备加入
 * I/O 系统中。它还可以为设备分配和初始化数据结构,初始化信号量。初始化设备硬件等。
*/

STATUS xxDevCreate (name, ...)
  char * name;
  ...
  {
  status = iosDevAdd (xxDev, name, xxDrvNum);
  ...
```

```
    }

  /*
   * 下面的程序调用了基本 I/O 操作函数。
   * xxOpen( )函数的返回值只对该驱动程序有意义,并且作为参数传递给
   * 其他 I/O 操作函数。
   */

  int xxOpen (xxDev, remainder, mode)
    XXDEV * xxDev;
    char * remainder;
    int mode;
    {
    /* 串行设备不能包括文件名部分 */

    if (remainder[0] != 0)
      return (ERROR);
    else
      return ((int) xxDev);
    }

  int xxRead (xxDev, buffer, nBytes)
    XXDEV * xxDev;
    char * buffer;
    int nBytes;
    ...
  int xxWrite (xxDev, buffer, nBytes)
    ...
  int xxIoctl (xxDev, requestCode, arg)
    ...
  /*
   * xxInterrupt - 中断服务程序
   *
   * 大多数驱动程序提供处理相应设备产生的中断。通过调用 intConnect(通常在 xxDrv 中)将中
   * 断服务程序与中断连接。它们可以收到一个在调用 intConnect 时指定的参数
  */

  VOID xxInterrupt (arg)
    ...
```

4.9.1 驱动程序

对于一个非块存取设备的驱动程序有 7 种基本 I/O 操作函数：creat( )、delete( )、open( )、close( )、read( )、write( )和 ioctl( )。虽然一些设备不具备操作的相应操作函数会被忽略掉，但总的来说这种类型的设备驱动程序包含上述 7 种操作函数。

驱动程序可以有选择的允许任务等待多个文件描述符的活动。这种操作可以通过调用驱动程序中的 ioctl( )函数实现，见“select( )函数的应用”。

块存取设备的驱动程序不是与 I/O 系统直接相连，而是将文件系统作为接口。文件系统执行大多数 I/O 操作，而驱动程序仅仅执行一些如读/写数据块、复位设备、执行 I/O 控制、检查设备状态等操作。在“4.9.4 块存取设备”一节中将介绍块存取设备驱动程序的一些特殊要求。

当用户调用一种基本的 I/O 操作程序时，I/O 系统将用户请求反映给特定驱动程序的相应操作函数（这个过程将在以下几节中介绍）。驱动程序的操作函数运行于调用它的任务上下文中，好像是从应用程序中直接调用该操作函数。因此，驱动程序可调用该任务可以执行的任何操作，包括对其他设备进行 I/O 操作。这意味着多数驱动程序必须对关键程序代码采取一种互斥的操作机制。通常采用的互斥机制是 semLib 函数库提供的信号量操作。

驱动程序除了具有上述 7 种基本 I/O 操作函数以外，还包含以下三种操作：

初始化函数负责在 I/O 系统中安装驱动程序，驱动程序将相应的设备与所需要的中断操作连接起来，然后初始化函数再执行任何必须的硬件初始化操作。这类函数一般称作 xxDrv( )。

将由驱动程序服务的设备加载到 I/O 系统中。这类函数一般称作 xxDevCreate( )。

将由驱动程序服务的设备的中断操作相联系的中断级程序。

1. 驱动程序表和安装驱动程序

I/O 系统的功能是将用户的 I/O 请求与相应驱动程序中的相应操作函数相连。I/O 系统通过维护一个包括每个驱动程序中每个操作函数的地址表来完成上述工作。通过调用 I/O 系统的内部函数 iosDrvInstall( )可以动态的安装驱动程序。该函数的参数就是新驱动程序中 7 种基本 I/O 操作函数的地址。iosDrvInstall( )函数将这些地址写入驱动程序表中的一块空闲存储区中，并返回这块存取区域的标志。标志用驱动程序号表示，可以被与驱动程序相关联的设备使用。

空地址(0)可以分配给 7 个基本操作函数中的几个，表示驱动程序不具备该项功能。对于非文件系统的驱动程序而言，close( )和 delete( )函数通常不起作用。

VxWorks 操作系统中的文件系统（如 dosFsLib 文件系统）在驱动程序表中包含了它们自己的入口地址，这些入口地址是在文件系统库初始化时创建的。

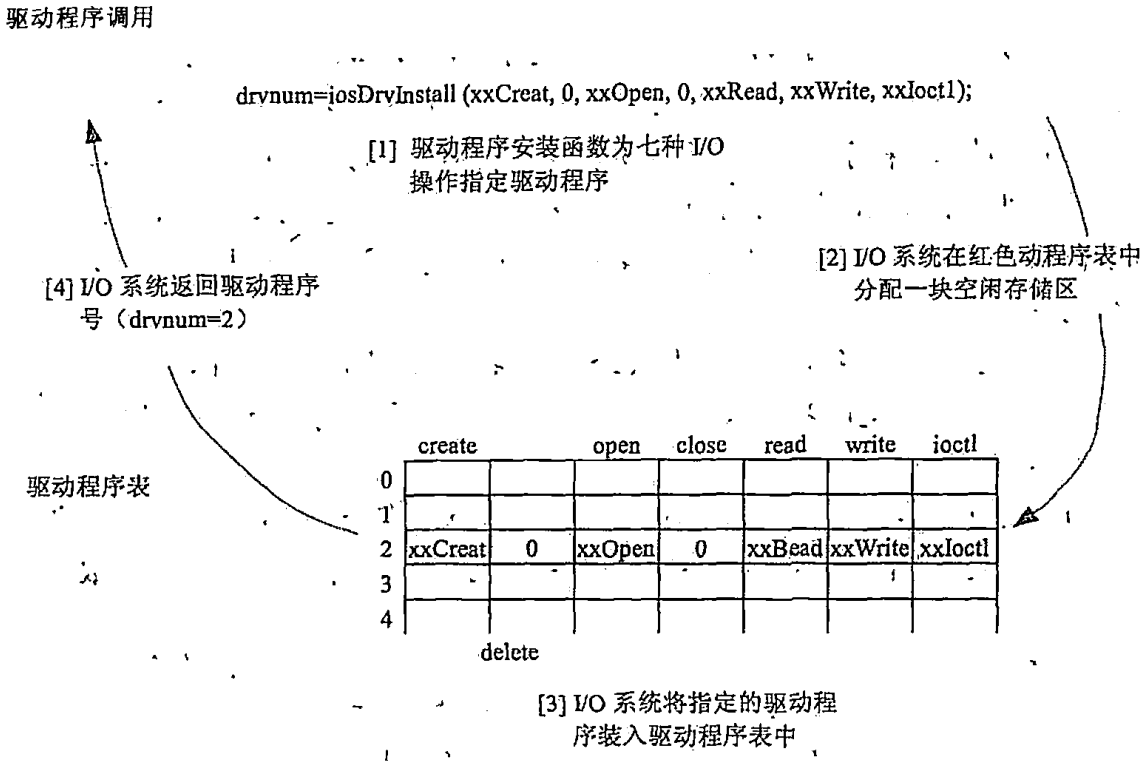


图 4-2　初始化非块存取设备的驱动程序

2. 安装驱动程序的程序范例

图 4-2 演示了在初始化程序 xxDrv( )运行时，驱动程序范例和 I/O 系统所执行的操作。

驱动程序首先调用 iosDrvInstall( )函数为驱动程序的 7 种基本 I/O 操作函数指定地址，然后 I/O 系统将执行如下操作：

在驱动程序表中设定下一块可用存储区（在本例中是 slot 2）；

将驱动程序中的操作函数地址写入驱动程序表中；

将存储区号作为新安装的驱动程序号；

4.9.2　驱动设备

一些设备驱动程序能够为某一特殊种类设备的多个实例提供服务。例如，一个串行通信设备的驱动程序经常可以处理多个参数不同（如设备地址）的通信通道。

在 VxWorks 操作系统的 I/O 系统中，给设备定义了一种称为设备头（DEV_HDR）的数据结构。在这个数据结构中包括了设备名称字符串和这个设备所使用的驱动程序的编号。I/O 系统中的所有设备的设备头都保存在一个称为设备表的驻留内存的链表结构中。设备头位于由单独设备驱动程序决定的一个更大的数据结构的起始部分。这个更大的数据结构称为设备描述符。它包括了其他与设备有关的数据，如设备地址、缓冲区、信号量等。

1. 设备列表和增加设备

用户可以通过调用内部 I/O 操作函数 iosDevAdd( )向 I/O 系统动态地增加非块存取设备。iosDevAdd( )函数所需的参数包括：（1）新设备的设备描述符地址；（2）设备名；（3）为该设备服务的驱动程序的个数。由驱动程序指定的设备描述符只要以设备头结构开始，那么它就可以包括与设备有关的任何信息。设备头中不需要装入驱动程序，只需包括设备有关的信息。iosDevAdd( )函数会将设备名和驱动程序号写入设备头结构中，并将它加入系统设备列表中。

用户通过调用设备所需文件系统的设备初始化程序（dosFsDevCreate( )或 rawFsDevInit( )函数）就可以向 I/O 系统添加一个块存取设备，然后设备初始化程序自动调用 iosDevAdd( )函数。

iosDevFind( )函数可以用来确定设备的结构（通过获取一个指向 DEV_HDR 的指针），以及验证设备表中存在的设备名。下面的程序介绍了如何使用 iosDevFind( )函数。

```
char *  pTail;                          /* 指向 devName 结构尾部的指针 */
char devName[6] = "DEV1:";              /* 设备名 */
DOS_VOLUME_DESC *  pDosVolDesc;         /* 第一个成员是 DEV_HDR */
    ...
    pDosVolDesc = iosDevFind(devName, (char**)&pTail);
    if (NULL == pDosVolDesc)
        {
        /* 错误:设备名不存在或默认设备不存在 */
        }
    else
        {
        /*
         * pDosVolDesc 是一个有效的 DEV_HDR 结构指针,pTail 指向 devName 的起
         * Check devName against pTail to determine if it is
         *始处。检查 pTail 和 devName 来决定是否是默认名称或指定的 devName。
         */
        }
```

2. 添加设备

在图 4-3 中，驱动程序范例中的设备创建函数 xxDevCreate( )通过调用 iosDevAdd( )函数向 I/O 系统添加设备。

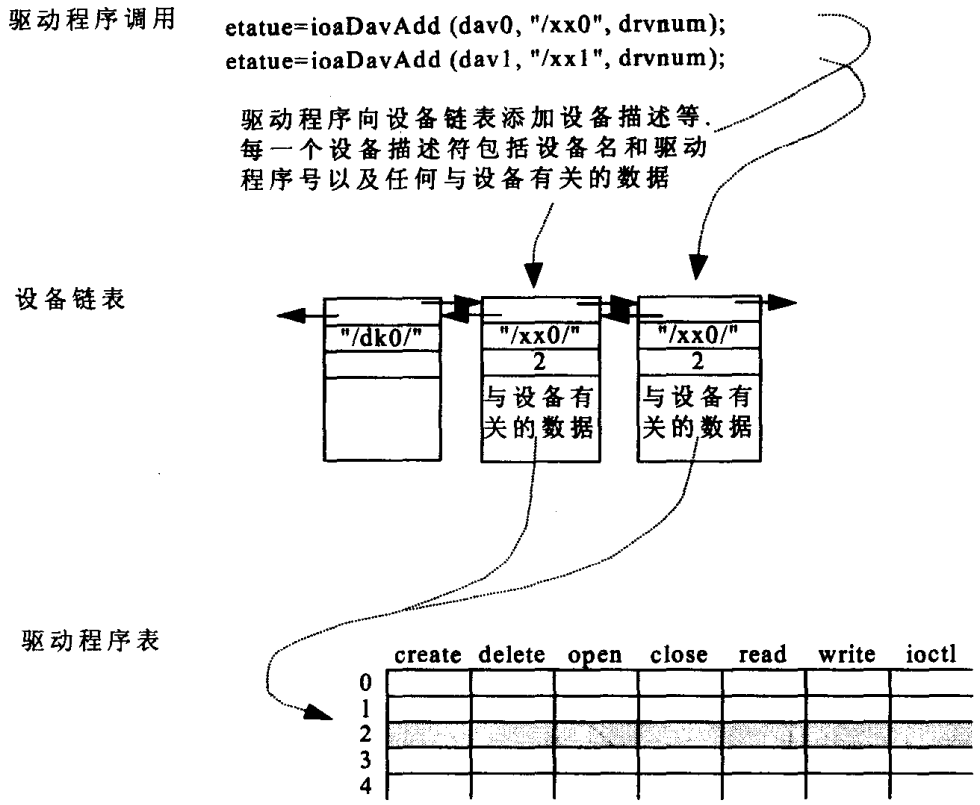


图 4-3　向 I/O 系统中添加设备

4.9.3　文件描述符

一个设备同时可以打开多个文件描述符（fd）。设备驱动程序可以维护超出 I/O 系统设备信息范围以外与文件描述符有关的附加信息。特别是当一个设备同时打开多个文件时，所打开的每一个文件描述符都产生与文件相关的信息（如文件偏移量）。用户也可以使用一个非块存取设备，如 tty，打开多个文件描述符。这时通常不会产生附加信息，而且向任何文件描述符写数据会产生相同的结果。

1. 文件描述符表

文件通过调用 open( ) 函数（或 creat( )函数）打开。I/O 系统在设备列表中寻找与调用者指定的文件名（或一个初始字符串）相匹配的设备名，如果找到相匹配的设备名，I/O 系统会提取相应位置的设备头中所包含的驱动程序号并从相应驱动程序表中调用执行打开操作的程序。

I/O 系统必须使在以后 I/O 操作中使用的文件描述符和为该文件描述符服务的驱动程序建立联系。而且驱动程序必须将每一个文件描述符与一些数据结构相关联。对于非块存取设备，是将设备描述符与一些数据结构相关联。

I/O 系统使用文件描述符表来维护这些关系。表中包括驱动程序号和一个由驱动程序确定的 4 字节的值。这个值是由驱动程序中执行打开操作的程序返回的内部描述符。它可以取任何非负数，用于驱动程序识别文件。在应用程序级的 I/O 操作中，当调用驱动程序其他 I/O 操作函数（read( )、write( )、ioctl( )和 close( )）时，驱动程序用这个值代替文件描述符。

2. 打开文件

在图 4-4 和图 4-5 中，用户程序调用 open( )函数打开一个名为‘/xx0’的文件。I/O 系统执行了如下操作：

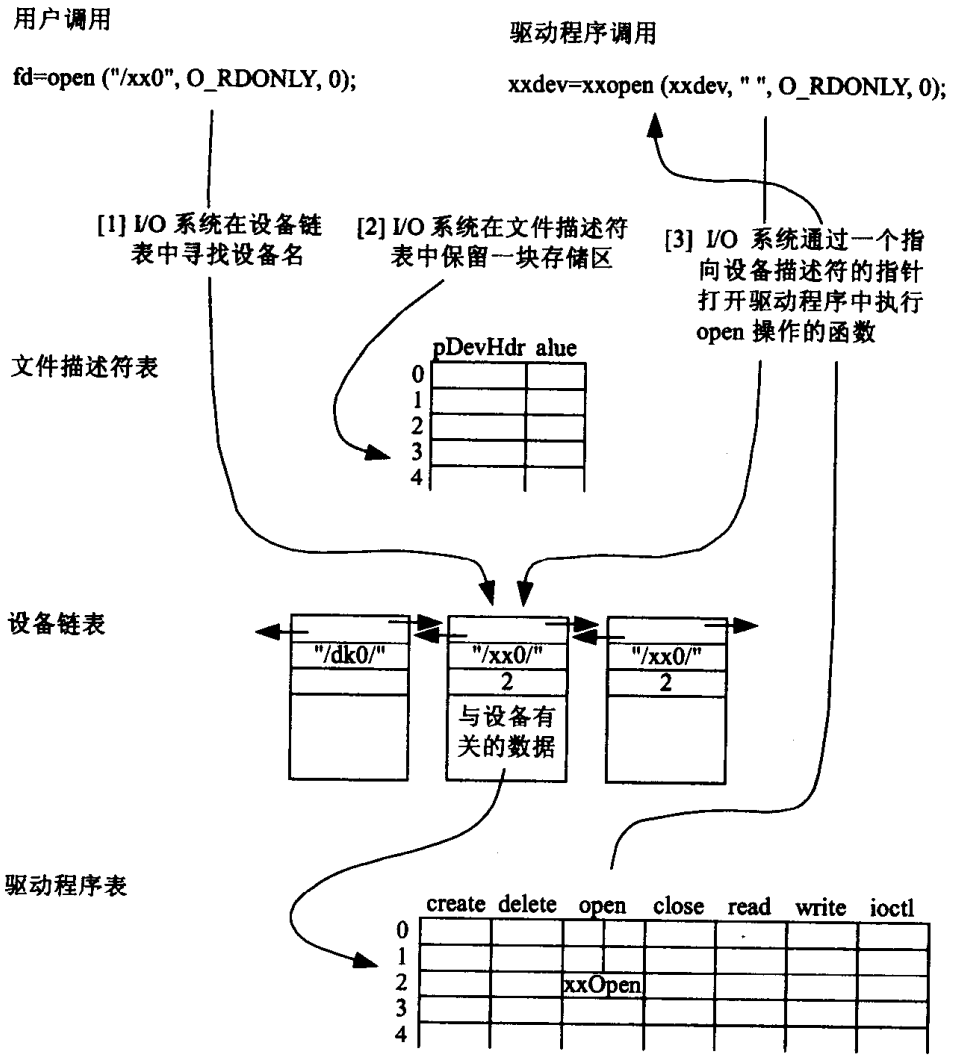


图 4-4　调用 I/O 操作函数——open( )（第一部分）

I/O 系统在设备列表中寻找与指定文件名（原始字符串）相匹配的设备名。在本例中，存在一个完全匹配的设备名。

I/O 系统在文件描述符表中占用一块区域创建一个新的文件描述符对象。如果打开操作成功，系统将使用这个文件描述符。

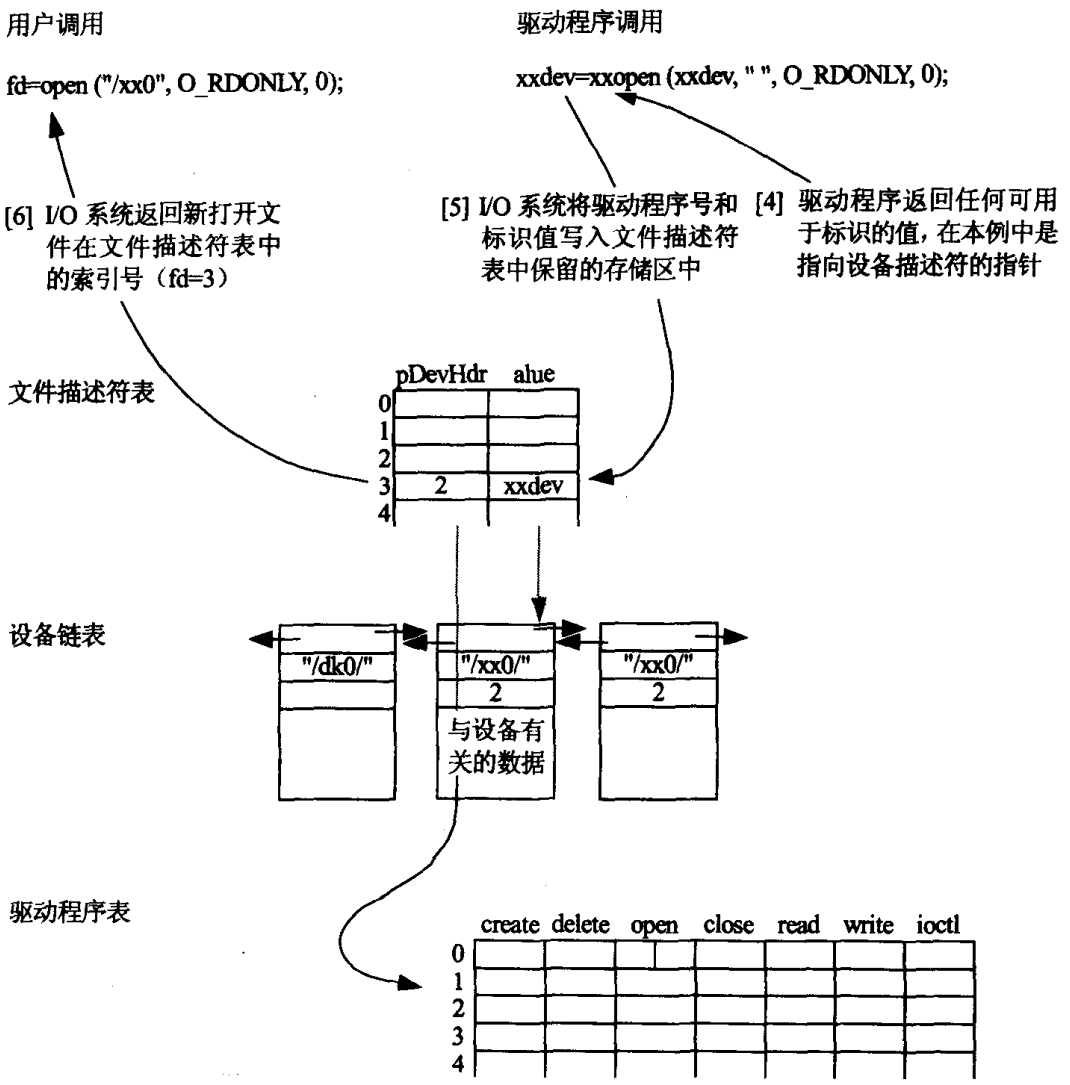


图 4-5　调用 I/O 操作函数——open( )（第一部分）

I/O 系统然后寻找驱动程序中执行打开操作的 xxOpen( )函数的地址，并调用该函数。注意 xxOpen( )函数所需的参数由用户最初传递给 open( )函数，然后由 I/O 系统在传递给 xxOpen( )函数。xxOpen( )函数的第一个参数是一个指向由 I/O 系统在文件名查询中确定的设备描述符指针。第二个参数是由用户指定的文件名在删去与设备名相匹配的字符串后的余项。在本例中由于文件名与设备名完全匹配，因此传递给驱动程序的余项是一个空字符。因此驱动程序不用解释这个余项。在对块存取设备的操作中，这个余项是设备上的一个文件的名称。在对非块存取设备的操作中，就同本例一样，应该是一个空字符，若不是则出错。第三个参数是文件访问标志。在本例中是 O_RDONLY，即文件是以只读方式打开的。最后一个参数是最初传递给 open( )函数的文件模式。

当 I/O 系统运行 xxOpen( )函数后，该函数会提供一个用于确定新打开文件的返回值。在本例中，这个返回值是一个指向设备描述符的指针。该值会在随后涉及对已打开文件的 I/O 操作中提供给驱动程序。注意如果仅返回了一个设备描述符，那么该设备的驱动程序

将不能区别针对同一设备的多个已打开的文件。对于非块存取设备的驱动程序，通常是这样的。

然后将 I/O 系统 xxOpen( )函数返回的驱动程序号和返回值写入新的文件描述符中。

写入文件描述符中的返回值只对驱动程序有意义，而且对于 I/O 系统而言，它是随机的。

最后，I/O 系统将文件描述表中存储区的索引传递给用户程序。

3. 从文件中读取数据

在图 4-6 中，用户程序调用 read( )函数从文件中读取数据。指定的文件描述符就是该文件在文件描述符表中的索引。I/O 系统使用表中保存的驱动程序号寻找该驱动程序用于执行读操作的函数 xxRead( )。然后调用 xxRead( )函数并同时将确认值传递给该函数。这个确认值是由驱动程序中执行打开操作的函数 xxOpen( )产生并保存在文件描述符表中的。在本例中，这个值是指向设备描述符的指针。然后驱动程序中执行读操作的函数从指定设备中读取所需的数据。用户程序调用 write( ) 和 ioctl( )函数时的执行过程与上述情况相似。

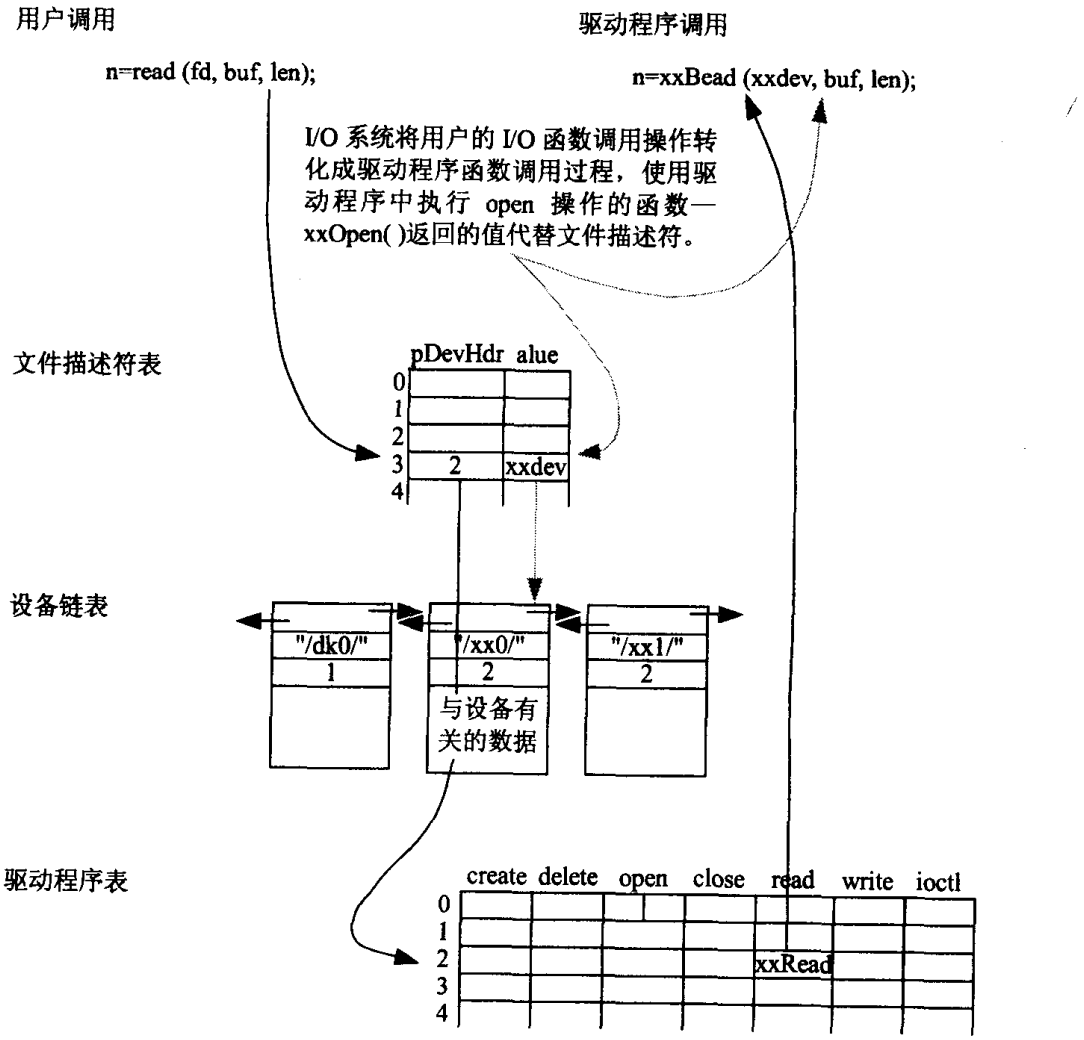


图 4-6　调用 I/O 操作函数——read( )

4. 关闭文件

用户程序通过调用 close( )函数来终止一个文件的使用。就像 read( )函数一样，I/O 系统使用保存在文件描述符表中的驱动程序号定位驱动程序执行关闭操作的函数。在驱动程序范例中并未指定执行关闭操作的函数，因此 I/O 系统不会调用任何函数。但是 I/O 系统直接将与该文件对应的文件描述符表中的存储区标记为可用。以后任何针对于该文件描述符的操作将产生错误。但如果再次调用 open( )函数，则仍可以使用这块存取区。

5. 使用 select( )函数

支持调用 select( )函数的用户驱动程序可以使任务等待多个设备的输入，或者允许任务为设备可以执行指定的 I/O 操作设定最长等待时间。因为 selectLib 函数库提供大多数功能，所以编写一个支持调用 select( )函数的驱动程序很简单。如果用户希望设备可以进行如下操作，那么用户的驱动程序必须支持调用 select( )函数。

- 任务需要为等待一个设备进行 I/O 操作设定最长时间限制。例如，可能会为一个 UDP 套接字接收信息包操作设定一个时限。
- 一个驱动程序同时支持多个设备，而运行的任务可能会同时等待这些设备。例如，可能会为不同优先级的数据传输操作建立多个管道。
- 任务等待某个设备的 I/O 操作，同时该设备等待其他设备的 I/O 操作。例如，一个服务器任务可能会使用管道和套接字。

在 select( )函数执行时，驱动程序必须保存一个说明什么任务等待什么设备活动的表。当被等待的设备可用时，驱动程序激活所有等待该设备的任务。

如果一个设备驱动程序支持调用 select( )函数，那么它必须声明一个 SEL_WAKEUP_LIST 的数据结构（通常作为设备描述符结构的一部分），并且调用 selWakeupListInit( )函数对这个结构进行初始化。这一过程在驱动程序中的 xxDevCreate( )函数中完成。当一个任务调用 select( )函数后，selectLib 函数库调用驱动程序中的 ioctl( )函数执行 FIOSELECT 或 FIOUNSELECT 功能。当 ioctl( )函数执行 FIOSELECT 功能时，驱动程序必须进行如下操作：

通过调用 selNodeAdd( )函数向 SEL_WAKEUP_LIST 结构中增加一个 SEL_WAKEUP_NODE 节点，该节点的内容由 ioctl( )函数的第三个参数提供；

调用 selWakeupType( )函数检查任务是否正等待从设备中读数据（SELREAD）或设备是否准备好输入数据（SELWRITE）；

如果设备已经就绪（由 selWakeupType( )函数决定用于输入或输出），驱动程序会调用 selWakeup( )函数以确保调用 select( )函数的任务不会处于挂起状态。这样就避免当设备可用时任务仍处于拥塞状态；

如果调用 ioctl( )函数执行 FIOUNSELECT 功能，那么驱动程序会调用 selNodeDelete( )函数将唤醒表中的 SEL_WAKEUP_NODE 节点删除。

当某一个设备可用时，通过调用 selWakeupAll( )函数可以将所有等待该设备的任务拥

塞状态解除。虽然这种情况一般发生在驱动程序的中断服务程序中，它也能发生在任何其他地方。例如，当某个管道中有剩余空间存储数据时，该管道的驱动程序会从它的 xxRead( ) 函数中调用 selWakeupAll( )函数激活等待执行写操作的任务。同样，当某个管道中有数据时，该管道的驱动程序会从它的 xxWrite( )函数中调用 selWakeupAll( )函数激活等待执行读操作的任务。

例 4-10：使用 select( )函数功能的驱动程序

```
/* 该段程序演示了驱动程序如何支持 select( )函数。在该例中,驱动程序将等待设备的
 * 任务变为就状态绪。
*/

/* myDrvLib.h - 驱动程序的头文件 */

typedef struct     /* MY_DEV */
    {
    DEV_HDR     devHdr;                 /* 设备头 */
    BOOL        myDrvDataAvailable;      /* 表示数据可以被读取 */
    BOOL        myDrvRdyForWriting;      /* 表示数据可以被写入 */
    SEL_WAKEUP_LIST selWakeupList;    /* 选择功能列出的处于挂起状态的任务 */
    } MY_DEV;

-------------------------------------------------------------------------

/* myDrv.c - 驱动程序中支持 select( )函数的程序 */

#include "vxWorks.h"
#include "selectLib.h"

/* 首先创建并初始化设备 */

STATUS myDrvDevCreate
    (
    char *  name,                     /* 创建设备的名称 */
    )
    {
    MY_DEV * pMyDrvDev;               /* 指向设备描述符的指针 */
    ... additional driver code ...

    /* 为 MY_DEV 分配存储区 */
    pMyDrvDev = (MY_DEV *) malloc (sizeof MY_DEV);
    ... additional driver code ...
```

```
    /* 初始化 MY_DEV */
    pMyDrvDev->myDrvDataAvailable=FALSE
    pMyDrvDev->myDrvRdyForWriting=FALSE

    /* 初始化唤醒表 */
    selWakeupListInit (&pMyDrvDev->selWakeupList);
    ... additional driver code ...
    }

/* 执行读/写操作的 ioctl 功能 */

STATUS myDrvIoctl
    (
    MY_DEV * pMyDrvDev,              /* 指向设备描述符的指针 */
    int      request,                /* ioctl 功能 */
    int      arg                     /* 存放应答信号 */
    )
    {
    ... additional driver code ...

    switch (request)
        {
        ... additional driver code ...

        case FIOSELECT:

            /* 向唤醒表添加节点 */

            selNodeAdd (&pMyDrvDev->selWakeupList, (SEL_WAKEUP_NODE *) arg);

            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELREAD
                && pMyDrvDev->myDrvDataAvailable)
                {
                /* 检查数据有效性,确认任务不处于挂起状态 */
                selWakeup ((SEL_WAKEUP_NODE *) arg);
                }
            if (selWakeupType ((SEL_WAKEUP_NODE *) arg) == SELWRITE
                && pMyDrvDev->myDrvRdyForWriting)
                {
```

```
                /* 检查设备可写,确认任务不处于挂起状态 */
                selWakeup ((SEL_WAKEUP_NODE *) arg);
                }
            break;

        case FIOUNSELECT:

            /* 从唤醒表中删除节点 */
            selNodeDelete (&pMyDrvDev->selWakeupList, (SEL_WAKEUP_NODE *)
arg);
            break;

            ... additional driver code ...
        }
    }

/* 该部分程序实际使用 select( )功能执行读/写操作 */

void myDrvIsr
    (
    MY_DEV * pMyDrvDev;
    )
    {
    ... additional driver code ...

    /* 如果存在可读取的数据,唤醒所有处于挂起状态的任务 */

    if (pMyDrvDev->myDrvDataAvailable)
        selWakeupAll (&pMyDrvDev->selWakeupList, SELREAD);

    /* 如果设备可写,唤醒所有处于挂起状态的任务 */

    if (pMyDrvDev->myDrvRdyForWriting)
        selWakeupAll (&pMyDrvDev->selWakeupList, SELWRITE);
    }
```

6. 高速缓冲存储区的一致性问题

带有高速缓冲存储区的用户板的驱动程序必须保证缓冲区中内容的一致性。缓冲区中内容的一致性意味着高速缓冲存储区中的数据必须与 RAM 存储区中的数据同步或一致。如果对 RAM 存储区进行异步数据访问（DMA 设备操作或 VME 总线操作），那么高速缓冲存储区中的数据与 RAM 存储区中的数据有可能不一致。高速缓冲存储区的作用是通过减

少访问 RAM 存储区的次数来提高系统性能。图 4-7 展示了 CPU、数据高速缓冲存储区、RAM 存储区和一个 DMA 设备间的关系。

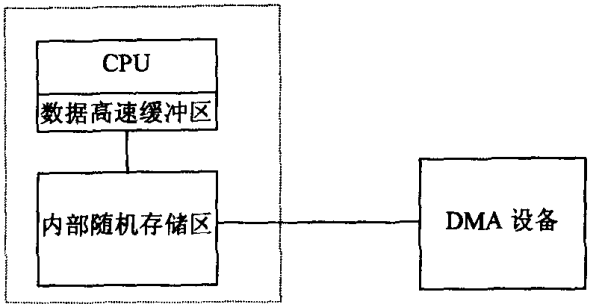


图 4-7　高速缓冲存储区的一致性

数据高速缓冲存储区可以运行于如下两种操作模式：同时写（writethrough）和回写（copyback）。

在同时写模式下，同时向数据高速缓冲存储区和 RAM 存储区写入数据。这样就保证了缓冲存储区在写入时的数据一致性，但不能保证数据读出时的一致性。在备份模式下，新数据只写入数据高速缓冲存储区。这样就能保证在读/写数据时的一致性。

如果 CPU 指定一个 DMA 设备向 RAM 存储区中写数据，那么数据将首先被写入数据高速缓冲存储区中。当 DMA 设备从 RAM 存储区中读数据时，将不能保证 RAM 存储区中的数据被缓冲存储区中的数据刷新过，这样输出至设备中的数据可能仍是旧数据，新数据仍保存在缓冲存储区中。解决这种数据不同步的问题可以通过一种机制，即确保在 RAM 存储区中的数据输出到 DMA 设备之前已经被刷新过。

当 CPU 通过一个 DMA 设备从 RAM 存储区中读数据时，如果缓冲存储区中标记所要读取的数据无效，则从缓冲存储区中读取数据，该数据并不是从设备中写入 RAM 存储区中的数据。解决数据不同步的方法是保证当缓冲存储区中的数据标记为无效时，从 RAM 存储区而不是从缓冲存储区中读取数据。

驱动程序可以通过两种方法解决缓冲存储区的一致性问题：（1）分配安全预取缓冲存储区，即一些标记成不可预取的缓冲存储区；（2）当从设备中读出或写入数据时都刷新预取缓冲存储区并将其标记为无效。对于静态缓冲存储区而言，分配安全预取缓冲存储区的方法是有效的，但这通常需要 MMU 单元。对于动态缓冲存储区，如果频繁地分配和释放非预取缓冲存储区，则会导致大量存储区被标记成非预取。使动态缓冲存储区保持数据一致性的方法是手工将非缓冲存储区标记为无效或者手工对非缓冲存储区进行刷新。

cacheFlush( )函数和 cacheInvalidate( )函数用于手工刷新缓冲存储区和将其标记为无效。在设备读数据之前，调用 cacheFlush( )函数刷新 RAM 存储区中的数据，这样可以保证设备读取新数据。当设备将新数据写入 RAM 存储区后，调用 cacheInvalidate( )函数将相应的缓冲存储区标记为无效。这样可以保证当 CPU 读取数据时，缓冲存储区已经被 RAM 存储区中的数据刷新了。

例 4-11：DMA 传输

```
/* 该程序演示 DMA 操作。在设备输出数据之前，程序通过调用 cacheFlush( )函数刷新高速缓冲
* 区。在一次读操作中，在数据传送完毕后，必须调用 cacheInvalidate( )函数使高速缓冲区无效
*/

#include "vxWorks.h"
#include "cacheLib.h"
#include "fcntl.h"
#include "example.h"
void exampleDmaTransfer   /* 1 表示读操作, 0 表示写操作 */
    (
    UINT8 *pExampleBuf,
    int exampleBufLen,
    int xferDirection
    )
    {
    if (xferDirection == 1)
        {
        myDevToBuf (pExampleBuf);
        cacheInvalidate (DATA_CACHE, pExampleBuf, exampleBufLen);
        }

    else
        {
        cacheFlush (DATA_CACHE, pExampleBuf, exampleBufLen);
        myBufToDev (pExampleBuf);
        }
    }
```

将分配安全预取缓冲存储区的方法与刷新预取缓冲存储区并将其标记为无效的方法相结合，可能会使驱动程序更有效。思路是只有当绝对需要时才执行刷新操作或将缓冲存储区标记为无效。通过调用 cacheDmaMalloc( )函数可以解决静态缓冲区的一致性问题。这个函数初始化一个在 cacheLib.h 文件中定义的 CACHE_FUNCS 数据结构，这个数据结构指向了用于保持缓冲区一致性的、执行刷新和标记为无效操作的函数。

宏函数 CACHE_DMA_FLUSH 和 CACHE_DMA_INVALIDATE 使用这个数据结构优化执行刷新和标记为无效操作函数的调用过程。因为如果当 CACHE_FUNCS 结构中指向执行刷新和标记为无效操作函数的指针为空时，会认为此时缓冲区处于一致状态，因此不会调用相应函数。

驱动程序使用虚拟地址而设备使用物理地址，因此必须将物理地址传给设备，当驱动

程序访问内部存储区时，它必须使用虚拟地址。

设备驱动程序在将地址内容传递给相应设备时，必须使用宏函数CACHE_DMA_VIRT_TO_PHYS 将虚拟地址转换成物理地址。同样有时也需要使用宏函数 CACHE_DMA_PHYS_TO_VIRT 将物理地址转换成虚拟地址。因为这两个过程既浪费时间又是不确定的，所以应尽可能避免。

例 4-12：地址转换操作

```
/* 下面的程序演示了一个驱动程序执行地址转换操作。程序分配一块带缓存安全保证的缓冲区，并
 * 填写数据。然后将数据写入设备中。它使用 CACHE_DMA_FLUSH 功能保证数据及时刷新。然后
 * 驱动程序读取新的数据并使用 CACHE_DMA_INVALIDATE 来保证缓冲区中的数据一致性。
 */

#include "vxWorks.h"
#include "cacheLib.h"
#include "myExample.h"
STATUS myDmaExample (void)
    {
    void * pMyBuf;
    void * pPhysAddr;

    /* 分配一块带缓存安全保证的缓冲区 */
    if ((pMyBuf = cacheDmaMalloc (MY_BUF_SIZE)) == NULL)
    return (ERROR);

     fill buffer with useful information

    /* 在将数据写入设备之前刷新缓冲区中的数据 */
    CACHE_DMA_FLUSH (pMyBuf, MY_BUF_SIZE);

    /* 将虚地址转换成物理地址 */
    pPhysAddr = CACHE_DMA_VIRT_TO_PHYS (pMyBuf);

    /* 设备从内部存储区读数据 */
    myBufToDev (pPhysAddr);
     wait for DMA to complete
     ready to read new data

    /* 设备向内部存储区写数据 */
    myDevToBuf (pPhysAddr);
     wait for transfer to complete
```

```
/* 将物理地址转换成虚地址 */
pMyBuf = CACHE_DMA_PHYS_TO_VIRT (pPhysAddr);

/* 使高速缓冲区无效 */
CACHE_DMA_INVALIDATE (pMyBuf, MY_BUF_SIZE);
 use data

/* 操作结束后释放存储区 */
if (cacheDmaFree (pMyBuf) == ERROR)
    return (ERROR);
return (OK);
}
```

4.9.4 块存取设备

1. 概述

在 VxWorks 操作系统中，块存取设备的接口与其他 I/O 设备的接口稍有不同。块存取设备的驱动程序不是与 I/O 系统直接相连，而是与文件系统相互作用。文件系统再与 I/O 系统相连。自从支持 SCSI-1 协议之后，允许操作系统同 dosFs 和 rawFs 文件系统一样可以直接访问块存取设备，而且 VxWorks 操作系统支持符合 SCSI-2 协议的顺序存储设备。顺序存储设备是一种由只能按一定顺序读/写的独立数据块组成的设备。当执行写操作时，写入的数据块按顺序加到存储媒体的尾部，即不能替换媒体中部的数据块。但是可以对媒体的任何位置进行读操作。这种对顺序存储媒体进行数据访问的过程与其他块存取设备不同。

图 4-8 展示了块存取设备和非块存取设备（字符设备）的 I/O 操作层次模型。这种层次模型允许不同的文件系统使用同一个块存取设备驱动程序。这样就能减少驱动程序中必须支持的 I/O 操作函数的数量。

一个块存取设备的驱动程序必须能够提供一种方法来建立一种逻辑块存取设备的数据结构。对于直接访问的块存取设备，结构名为 BLK_DEV；对于顺序访问的块存取设备，结构名为 SEQ_DEV。这两种数据结构以一种普通的方式对设备进行描述，只对设备采用的文件系统必须了解的一般特点进行说明。数据结构中的有些字段表示设备的各种物理配置变量，例如数据块的大小、数据块的总数等。数据结构中的其他字段指定了设备驱动程序中用于控制设备的函数（读数据块、写数据块、进行 I/O 控制、复位设备、检查设备状态）。数据结构中还包括驱动程序用于向文件系统传递一些状态信息的字段（如更换磁盘）。

当驱动程序新建一个块存取设备后，该设备既没有名称也没有任何文件系统与它相联接。这些信息通过设备的文件系统初始化程序设置（如 dosFsDevInit( ) 或 tapeFsDevInit( ) 函数）。

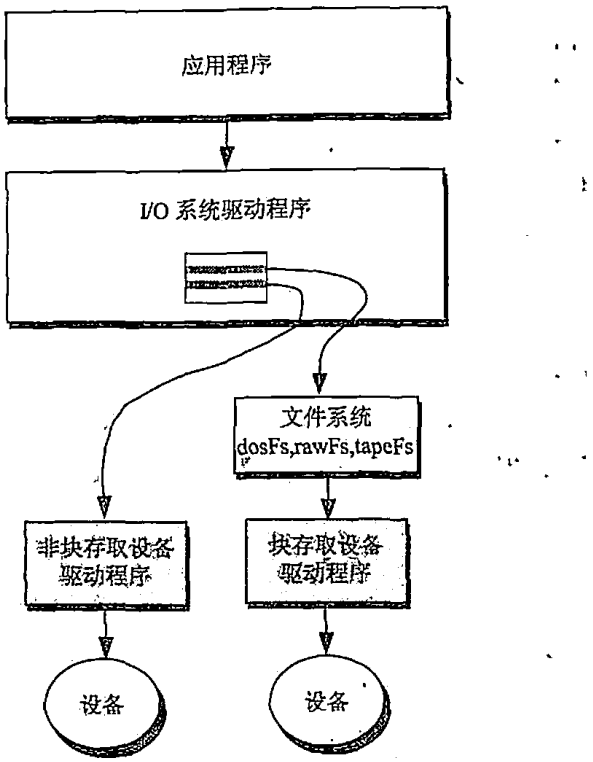


图 4-8　非块存取设备和块存取设备

与非块存取设备驱动程序不同，块存取设备的低级设备驱动程序并不安装在 I/O 系统的驱动程序表中。在 VxWorks 操作系统中，每一个文件系统都作为一个驱动程序安装在驱动程序表中。即使几个不同的低级设备驱动程序所驱动的设备安装在同一个文件系统中，每种文件系统在驱动程序表中只有一个入口。

在设备被初始化使用某种文件系统后，针对该设备的所有 I/O 操作都必须通过文件系统。文件系统依次调用 BLK_DEV 和 SEQ_DEV 数据结构中指定的函数对设备执行相应的操作。

一个块存取设备的驱动程序必须提供一个设备与 VxWorks 操作系统之间的接口。VxWorks 操作系统需要一系列功能函数，而且还需要基于附加功能的单独设备。在建立 VxWorks 操作系统驱动程序时，已使用设备的用户手册和该设备其他的驱动程序是非常有用的。

下面几部分介绍了建立与 VxWorks 操作系统中的文件系统的标准接口相关的低级块存取设备驱动程序所必需的组件。

2. 低级驱动程序初始化组件

驱动程序通常需要一个通用的初始化函数。这个函数包括那些只执行一次的操作，不包括为驱动程序的每个设备分别执行的操作。总的说，初始化函数的操作影响整个设备控制器，而后续的操作只影响某个个别的设备。

块存取设备驱动程序中通用的操作包括：

- 初始化硬件；
- 分配并初始化数据结构；
- 建立信号量；
- 初始化中断向量；
- 允许中断操作。

初始化函数执行的操作完全与使用的设备（控制器）有关，VxWorks 操作系统对初始化函数没有任何要求。

不像非块存取设备驱动程序，块存取设备的初始化函数不是通过调用 iosDrvInstall( ) 函数将驱动程序装入 I/O 系统的驱动程序表中，而是文件系统将自己作为驱动程序装入驱动程序表中，并且通过使用存放于块存取设备结构 BLK_DEV 或 SEQ_DEV 中的函数地址调用实际的驱动程序。

3. 设备创建函数

驱动程序必须提供一个创建逻辑磁盘设备或顺序存储设备的函数。一个逻辑磁盘设备可能是一个更大的物理设备的一部分。如果是这种情况，设备驱动程序必须能保存任何块地址偏移量或采取其他能识别与逻辑设备有关的物理位置的方法。VxWorks 操作系统经常将以零开始的数据块个数作为设备的开始地址。一个顺序存储设备所拥有的数据块的数目既可以是可变的也可以是固定的。大多数应用中使用的设备具有固定数目的数据块。

设备创建函数通常分配了一个设备描述符结构用于使设备驱动程序可以管理设备。设备描述符中的第一个字段必须是一个 VxWorks 操作系统块存取设备结构（BLK_DEV 或 SEQ_DEV）。因为该块存取设备结构的地址是由文件系统在调用驱动程序时传递过来的，所以放在设备描述符结构的第一个字段中，而且还能根据该块存取设备结构的地址确定设备描述符。

设备创建函数必须初始化 BLK_DEV 或 SEQ_DEV 结构中的字段。表 4-15 列出了 BLK_DEV 结构中的字段及其初始化值。

表 4-15 BLK_DEV 结构中的字段

字段名称	字段值
bd_blkRd	从设备中读取数据块的驱动程序的地址
bd_blkWrt	向设备中写入数据块的驱动程序的地址
bd_ioctl	执行设备 I/O 控制操作的驱动程序的地址
bd_reset	执行设备复位操作的驱动程序的地址（如果没有该程序，则值为 NULL）
bd_statusChk	执行检查设备状态操作的驱动程序的地址（如果没有该程序，则值为 NULL）
bd_removable	该字段值表示设备是否是可移动的。TRUE 表示设备是可移动的（如软盘）；FALSE 表示设备是不可移动的
bd_nBlocks	设备上的数据块总数
bd_bytesPerBlk	设备上每个数据块包含的字节数

续表

字 段 名 称	字　段　值
bd_blksPerTrack	设备上每个磁道包含的数据块个数
bd_nHeads	设备中的磁头个数
bd_retry	当出现读/写操作失败时重复操作的次数
bd_mode	设备的工作模式（写保护状态）；通常值为 O_RDWR
bd_readyChanged	该字段值表示设备的就绪状态是否改变。TRUE 表示就绪状态已经改变；初始化时将该字段值设为 TRUE 将导致安装磁盘。

表 4-16 列出了 SEQ_DEV 结构中的字段及其初始化值。

表 4-16　SEQ _DEV 结构中的字段

字段名称	字　段　值
sd_seqRd	从设备中读取数据块的驱动程序的地址
sd_seqWrt	向设备中写入数据块的驱动程序的地址
sd_ioctl	执行设备 I/O 控制操作的驱动程序的地址
sd_seqWrtFileMarks	向设备中写入文件标志的驱动程序的地址
sd_reWind	对顺序存储设备执行倒带操作的驱动程序的地址
sd_reserve	对顺序存储设备执行抢占式操作的驱动程序的地址
sd_release	对顺序存储设备执行释放操作的驱动程序的地址
sd_readBlkLim	从顺序存储设备中读取数据块容量限制的驱动程序的地址
sd_load	对顺序存储设备执行加载/卸载操作的驱动程序的地址
sd_space	对顺序存储设备执行向前/向后转动至文件结束/记录结束标志处操作的驱动程序的地址
sd_erase	对顺序存储设备执行擦除操作的驱动程序的地址
sd_reset	对顺序存储设备执行复位操作的驱动程序的地址（如果没有该程序，则值为 NULL）
sd_statusChk	对顺序存储设备执行状态查询操作的驱动程序的地址（如果没有该程序，则值为 NULL）
sd_blkSize	该字段值表示顺序存储设备上的数据块的大小。如果该值为‘0’则表示顺序存储设备使用可变数据块
sd_mode	顺序存储设备的工作模式（写保护状态）
sd_readyChanged	该字段值表示设备的就绪状态是否改变。TRUE 表示就绪状态已经改变；初始化时将该字段值设为 TRUE 将导致安装顺序存储设备。
sd_maxVarBlockLimit	可变数据块的最大容量
sd_density	该字段值表示顺序存储媒介的密度

设备创建函数返回 BLK_DEV 或 SEQ_DEV 结构的地址，然后文件系统设备初始化时调用这个地址用于确认不同设备。

不像非块存取设备驱动程序，块存取设备的设备创建函数并不是通过调用 iosDevAdd( )

函数将设备装入 I/O 系统设备表中。这个工作是在文件系统设备初始化函数中完成的。

4. 读操作（适用于直接存储设备）

驱动程序必须提供能执行从设备中读取一个或多个数据块操作的函数。对于直接存储的设备，读数据块操作必须包括如下的参数和返回值：

```
STATUS xxBlkRd
    (
    DEVICE *  pDev,
    int       startBlk,
    int       numBlks,
    char *    pBuf
    )
```

⚠ **注意**：在本例和以后的例子中，函数名以 xx 开始。带有这种名称的函数只是一个范例，并不能用于用户的设备驱动程序中。VxWorks 操作系统只根据函数地址辨认函数而不是根据函数名称。

pDev

该参数是一个指向驱动程序的设备描述符结构的指针，数据类型是 DEVICE。通过该参数确认设备。实际上文件系统传递的是相应 BLK_DEV 结构的地址，因为 BLK_DEV 结构在设备描述符中是第一个字段。

startBlk

该参数表示从设备中开始读数据块的起始位置。文件系统经常将数据块为零的位置作为设备的开始处。驱动程序在计算该起始位置是必须加入逻辑设备的偏移量。

numBlks

该参数表示需要读取的数据块个数。如果完成读操作的硬件设备不支持一次读取多个数据块，则驱动程序必须能够通过多次循环操作模拟批量读取操作。

pBuf

该参数用于存放所读取数据的存储区的地址。

如果读操作成功，则该函数返回 OK 标志，否则返回 ERROR 标志。

5. 读操作（适用于顺序存储设备）

驱动程序必须提供一个能执行从指定设备读取一定数量字节数据操作的函数，所读取的数据经常被认为是正处于存储媒介的读/写磁头所处的位置。该函数必须包括如下的参数和返回值：

```
STATUS xxSeqRd
```

```
(
DEVICE *  pDev,
int       numBytes,
char *    buffer,
BOOL      fixed
)
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针，数据类型是 DEVICE，通过该参数确认设备。实际上文件系统传递的是相应 SEQ _DEV 结构的地址，因为 SEQ_DEV 结构在设备描述符中是第一个字段。

numBytes

该参数表示读取数据的字节数。

buffer

该参数是指向存放所读取数据的存储区地址的指针。

fixed

该参数表示根据文件系统的指定，操作函数从顺序存储设备中读取固定大小的数据块还是可变大小的数据块。如果该参数值为 TRUE，则表示读取固定大小的数据块。

如果读操作成功，则该函数返回 OK 标志，否则返回 ERROR 标志。

6. 写操作（适用于直接存储设备）

驱动程序必须提供能执行向设备中写入一个或多个数据块操作的函数。该函数的定义与读操作函数的定义相似。对于直接存储的设备，写数据块操作必须包括如下的参数和返回值：

```
STATUS xxBlkWrt
    (
    DEVICE *  pDev,
    int       startBlk,
    int       numBlks,
    char *    pBuf
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

startBlk

该参数表示从设备中开始写数据块的起始位置。

numBlks

该参数表示需要写入的数据块个数。如果完成写操作的硬件设备不支持一次写入多个

数据块，则驱动程序必须能够通过多次循环操作模拟批量写入操作。

pBuf

该参数用于存放待写入数据的存储区的地址。

如果写操作成功，则该函数返回 OK 标志，否则返回 ERROR 标志。

7. 写操作（适用于顺序存储设备）

驱动程序必须提供能执行向设备中写入一定数量字节数据操作的函数。所写入的数据经常被认为是正处于存储媒介的读/写磁头所处的位置。该函数必须包括如下的参数和返回值：

```
STATUS xxWrtTape
    (
    DEVICE *  pDev,
    int       numBytes,
    char *    buffer,
    BOOL      fixed
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

numBytes

该参数表示写入数据的字节数。

buffer

该参数是指向存放待写入数据的存储区地址的指针。

fixed

该参数表示根据文件系统的指定，操作函数从顺序存储设备中读取固定大小的数据块还是可变大小的数据块。如果该参数值为 TRUE，则表示读取固定大小的数据块。

如果写操作成功，则该函数返回 OK 标志，否则返回 ERROR 标志。

8. I/O 控制函数

驱动程序必须提供能处理 I/O 控制请求的函数。在 VxWorks 操作系统中，大多数超出基本文件处理功能的 I/O 操作由 ioctl( )函数处理。这些 I/O 操作的大部分过程是由文件系统处理。但是如果出现文件系统不能识别的请求，则该操作请求就会交给驱动程序的 I/O 控制函数处理。

驱动程序的 I/O 控制函数定义如下：

```
STATUS xxIoctl
    (
    DEVICE *  pDev,
```

```
    int         funcCode,
    int         arg
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

funcCode

该参数表示 ioctl( )函数的功能代码。在文件 ioLib.h 中定义了 VxWorks 操作系统中的 I/O 控制功能代码。其他用户定义的功能代码可以由用户的设备驱动程序调用。在本手册的“第 5 章本地文件系统”中介绍了 dosFs、rawFs 和 tapeFs 文件系统支持的各种 I/O 控制功能。

arg

一些 ioctl( )函数的特殊功能专用的参数，并不是所有 ioctl( )函数的功能都使用此参数。

驱动程序的 I/O 控制函数采用一种开关式的工作模式，即根据不同的功能代码执行不同的操作。因此 I/O 控制函数必须提供一个默认的执行状态，以防止出现不能识别的功能代码。当出现这种情况时，I/O 控制函数向 S_ioLib_UNKNOWN_REQUEST 结构中写入错误号并返回一个 ERROR 标志。

驱动程序的 I/O 控制函数如果成功处理了操作请求，则返回一个 OK 标志，否则返回 ERROR 标志。

9. 设备复位操作

驱动程序经常提供一个对指定设备执行复位操作的函数，但这个函数不是必须有的。当 VxWorks 操作系统的文件系统第一次安装磁盘设备或磁带设备，或者当第一次读/写操作失败后，系统会调用复位函数。

驱动程序的设备复位函数定义如下：

```
STATUS xxReset
    (
    DEVICE *  pDev
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

当调用复位函数时，该函数将对设备和控制器进行复位。如果条件允许，它将不会复位其他设备。如果驱动程序对设备成功执行复位操作之后，函数将返回 OK 标志，否则返回 ERROR 标志。

如果设备不需要复位操作，则驱动程序中可以省略该函数。这时，由设备创建函数向 BLK_DEV 或 SEQ_DEV 结构中的 xx_reset 字段写入 NULL 标志。

⚠ **注意**：在本例和下面的例子中，在 BLK_DEV 和 SEQ_DEV 结构中除了开始字母 bd_或 sd_不同以外，各对应字段名称的其余部分是相同的。这样，开始字母就用 xx_表示，即 xx_reset 字段代表 bd_reset 字段和 the sd_reset 字段两种情况。

10. 状态检测函数

如果驱动程序提供一个执行检测设备状态或其他初级操作的函数，那么文件系统在运行每一个 open( )或 creat( )函数之前都会调用该函数。

状态检测函数的定义如下：

```
STATUS xxStatusChk
    (
    DEVICE *  pDev
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

如果打开或新建操作可以执行，则该函数返回 OK 标志。如果函数检测到设备存在一个错误，它会向 errno 变量写入一个表示该错误的数值并返回 ERROR 标志。如果函数返回 ERROR 标志，文件系统将终止后续操作。

状态检测函数的主要作用是检测设备（该类设备只有在插入新磁盘后才能发现磁盘被更换）上的磁盘更换过程。如果该函数确认出现了一个新磁盘，它会向 BLK_DEV 结构中的 bd_readyChanged 字段写入 TRUE 标志，然后返回一个 OK 标志。使得打开或新建操作可以继续执行。文件系统会自动安装新磁盘（参见“就绪状态”）

同样状态检测函数也能检测磁带设备的更换过程。该函数可确定是否装入一个新的磁带设备。如果出现一个新的磁带设备，该函数会向 SEQ_DEV 结构中的 sd_readyChanged 字段写入 TRUE 标志，后返回一个 OK 标志。使得打开或新建操作可以继续执行。当一个磁带设备的文件描述符正处于打开状态时，设备驱动程序不能卸载该磁带设备或将磁带从设备中弹出。

如果设备驱动程序不需要状态检测函数，则设备创建函数将 BLK_DEV 或 SEQ_DEV 结构中的 xx_statusChk 字段的值设为 NULL。

11. 具有写保护功能的存储媒介

设备驱动程序可能会检测出磁盘或磁带设备正处于写保护状态。如果发生这种情况，驱动程序将 BLK_DEV 或 SEQ_DEV 结构中的 xx_mode 字段的值设为 O_RDONLY。该项操作可以在任何时候执行（甚至可以在设备已经初始化运行某个文件系统之后）。文件系统

会根据 xx_mode 字段的值不允许对设备进行写操作，直到 xx_mode 字段的值改为 O_RDWR 或 O_WRONLY。

12. 检测就绪状态

设备驱动程序只要发现了设备的就绪状态有变化就通知文件系统，这些变化可能是更换了一个软盘、或者更换了一个磁盘、或者是其他一切可能导致文件系统重新安装磁盘的情况。

驱动程序通过将 BLK_DEV 或 SEQ_DEV 结构中的 xx_readyChanged 字段的值设为 TRUE 表示设备的就绪状态发生变化。文件系统识别该标志后，通过后续对磁盘进行 I/O 系统初始化操作重新安装磁盘。然后文件系统再将 xx_readyChanged 字段的值设为 FALSE。设备驱动程序从不清除该字段的 TRUE 标志。

将 xx_readyChanged 字段的值设为 TRUE 与调用文件系统中的就绪状态改变函数（如调用 ioctl( )函数执行 FIODISKCHANGE 功能）的效果是一样的。

一个可选的状态检测函数可以提供一种便捷的方法用于判断就绪状态改变过程，特别是用于那些直到装入新磁盘才发现磁盘被更换了的设备。如果状态检测函数发现了一个新磁盘，那么它会将 xx_readyChanged 字段的值设为 TRUE。文件系统在每一次执行打开或创建文件的操作前都调用这个函数。

13. 写文件标志函数（适用于顺序存储设备）

顺序存储设备的驱动程序必须提供能执行向磁带设备上写文件标志操作的函数。该函数必须包括以下参数：

```
STATUS xxWrtFileMarks
    (
    DEVICE *  pDev,
    int       numMarks,
    BOOL      shortMark
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

numMarks

该参数表示顺序写入的文件标志的个数。

shortMark

该参数表示文件标志的类型（长型或短型）。如果 shortMark 参数的值是 TRUE，则文件标志的类型是短型。如果文件标志正确地写入磁带设备，则该函数会返回 OK 标志；否则会返回 ERROR 标志。

14. 执行倒带操作函数（适用于顺序存储设备）

顺序存储设备的驱动程序必须提供能对磁带设备中的磁带进行倒带操作的函数。该函数定义如下：

```
STATUS xxReWind
    (
    DEVICE *  pDev
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

该函数能够对磁带设备中的磁带进行倒带操作。如果操作成功，该函数会返回 OK 标志，否则返回 ERROR 标志。

15. 执行保留操作的函数（适用于顺序存储设备）

顺序存储设备的驱动程序必须提供一种保护程序，它可以在主机执行保留操作时使物理磁带设备处于专用状态。磁带设备会一直维持这个状态直到主机执行释放操作或遇到外部干预。该函数定义如下：

```
STATUS xxReserve
    (
    DEVICE *  pDev
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

如果磁带设备成功地执行了该操作，函数返回 OK 标志；如果磁带设备不能执行该操作或发生错误，则返回 ERROR 标志。

16. 执行释放操作的函数（适用于顺序存储设备）

该函数释放主机对于一个磁带设备所执行的保留操作。该磁带设备可以再次被同一台主机或其他主机执行保留操作。该函数执行与保留操作相反的操作，如果驱动程序中包括执行保留操作的函数，则必须包括执行释放操作的函数。该函数定义如下：

```
STATUS xxReset
    (
    DEVICE *  pDev
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

如果磁带设备成功地执行了该操作，函数返回 OK 标志；如果磁带设备不能执行该操作或发生错误，则返回 ERROR 标志。

17. 读取数据块限制的函数（适用于顺序存储设备）

读取数据块限制的函数可以查询一个磁带设备的物理数据块大小的限制，然后将该数据大小的限制传递给文件系统，文件系统就能为用户程序设定一个数据块大小的范围。该函数定义如下：

```
STATUS xxReadBlkLim
    (
    DEVICE *  pDev,
    int       *maxBlkLimit,
    int       *minBlkLimit
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

maxBlkLimit

该参数表示磁带设备所能处理的数据块容量的最大值。

minBlkLimit

该参数表示磁带设备所能处理的数据块容量的最小值。

当该函数获得了数据块容量限制，就返回 OK 标志，否则返回 ERROR 标志。

18. 安装/卸载函数（适用于顺序存储设备）

顺序存储设备必须提供一个能在物理磁带设备上执行安装/卸载磁带盘操作的函数。安装意味着由文件系统安装磁带卷，这个过程通常与 open( )或 creat( )函数一起执行。但只有当文件系统想从磁带设备中弹出磁带盘时才能卸载设备。安装/卸载函数定义如下：

```
STATUS xxLoad
    (
    DEVICE *  pDev,
    BOOL      load
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

load

该参数决定对磁带盘执行安装还是卸载操作的一个布尔变量。如果该参数值为 TRUE，则磁带盘执行安装操作；如果该参数值为 FALSE，则磁带盘执行卸载操作。

如果安装/卸载操作成功，该函数返回 OK 标志，否则返回 ERROR 标志。

19. 搜索函数（适用于顺序存储设备）

顺序存储设备必须提供一个能够在磁带存储媒介向前或向后执行搜索操作的函数。磁带移动的距离取决于系统执行搜索操作的种类。总的来说，磁带的搜索方式可依据记录结束标志、文件结束标志或其他与设备有关的标志。

基本搜索函数定义如下，该函数模型中还可以加入其他参数。

```
STATUS xxSpace
    (
    DEVICE *  pDev,
    int       count,
    int       spaceCode
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

count

该参数指定了搜索的方向。正数表示从磁带设备当前位置向前搜索，负数表示向反方向搜索。

spaceCode

该参数定义了磁带设备在磁带存储媒介上执行搜索操作所参照的标记种类。基本的标记种类包括记录结束标志和文件结束标志，但是不同的磁带设备可能会支持更复杂的搜索标记，可以使磁带设备更有效的控制磁带存储媒介。

如果磁带设备能够按指定的方向和搜索标记执行搜索操作，该函数返回 OK 标志，否则返回 ERROR 标志。

20. 擦除函数（适用于顺序存储设备）

顺序存储设备必须提供一个能够将磁带上记录的数据擦除的函数，该函数定义如下：

```
STATUS xxErase
    (
    DEVICE *  pDev
    )
```

pDev

该参数是一个指向驱动程序的设备描述符结构的指针。

如果操作成功，该函数返回 OK 标志，否则返回 ERROR 标志。

4.9.5　驱动程序支持库

表 4-17 中列出的子程序库对于用户编写设备驱动程序有所帮助。使用这些函数库，大多数符合标准协议的设备驱动程序可以仅包含几页与设备有关的代码，用户可以参考每一个库函数手册来了解详细内容。

表 4-17　VxWorks 操作系统中的驱动程序支持库

函数库名称	说　明
errnoLib	错误状态函数库
ftpLib	ARPA 文件传输协议函数库
ioLib	I/O 接口函数库
iosLib	I/O 系统函数库
intLib	中断支持函数库
remLib	远程命令函数库
rngLib	环行缓冲区子程序函数库
ttyDrv	终端驱动程序函数库
wdLib	看门狗定时器函数库

4.10　PCMCIA 接口

一个 PCMCIA 卡可以插入笔记本电脑中用于连接诸如调制解调器和外部硬件设备等设备。VxWorks 操作系统提供了应用程序、BSP 程序、驱动程序，使得此系统可以运行于支持 PCMCIA 接口硬件设备的目标板上（如 pcPentium、pcPentium2、pcPentium3 系统）。

VxWorks 操作系统支持 2.1 版本的 PCMCIA 协议。但该协议不支持套接字服务程序或插卡服务程序。VxWorks 操作系统并不需要这两种服务程序。该版本协议中还包括芯片驱动程序和函数库。用户还可以获得基于非 Intel Pentium 结构 VxWorks 操作系统的 PCMCIA 函数库和驱动程序的源代码。

用户要在系统中支持 PCMCIA 协议，需要在 VxWorks 操作系统的核心保护域中加入 INCLUDE_PCMCIA 组件。用户欲了解有关 PCMCIA 应用的更多信息，请查阅“VxWorks 操作系统 API 参考手册”中关于 pcmciaLib 和 pcmciaShow 的内容。

4.11 外部设备互连接口：PCI

外部设备互连接口（PCI）是用于连接外设和个人电脑的一种总线标准，已用于 Pentium 系列和其他系列电脑中。PCI 接口包括用于 CPU 和相对慢速外设通信的缓冲区，这样可以使得它们之间进行异步通信。用户要了解有关 PCI 应用的更多信息，请查阅“VxWorks 操作系统 API 参考手册”中关于 pciAutoConfigLib、pciConfigLib、pciInitLib 和 pciConfigShow 的内容。

第 5 章　本地文件系统

5.1　简　　介

VxWorks 操作系统在文件系统与设备驱动程序之间使用一种标准的 I/O 操作接口。这样使得在单个 VxWorks 操作系统中可以运行多个相同或不同种类的文件系统。依据这些标准接口协议，用户可以为 VxWorks 操作系统编写用户自己的文件系统，并可以将文件系统和设备驱动程序自由组合。

本章将介绍下面列出的几种文件系统，并介绍如何组织、配置和使用这些文件系统。

- dosFs 文件系统：适用于块存取设备（磁盘）的实时操作，与 MS-DOS 文件系统兼容。
- rawFs 文件系统：提供了一种简单的原始文件系统。该文件系统将整个磁盘当作一个单独的大文件。
- tapeFs 文件系统：适用于不使用标准文件或目录结构的磁带设备。实际上将磁带盘当作一个原始设备并将整个磁带盘当作一个大文件。
- cdromFs 文件系统：允许应用程序从按照 ISO 9660 标准文件系统格式化的 CD-ROM 设备上读取数据。
- TSFS（Target Server File System）目标服务器文件系统：通过使用 Tornado 软件中的目标服务器，使得目标机可以访问主机系统中的文件。

VxWorks 操作系统通过使用可选产品——TrueFFS 文件系统，支持闪存设备。用户欲了解更多内容，请参考“第 8 章闪存块存取设备驱动程序”。

5.2　与 MS-DOS 兼容的文件系统：dosFs 文件系统

dosFs 文件系统是一种与 MS-DOS 文件系统兼容的文件系统，它能满足实时应用的多种要求。其主要特点如下：

- 支持层次化的文件和目录结构，能够在一个磁盘上建立一定数目的文件并进行有效地管理；
- 可以将每一个文件指定为连续存储或非连续存储；
- 广泛兼容各种可存储和可检索媒体（如软盘、硬盘）；

- 可以从 dosFs 文件系统中启动 VxWorks 操作系统；
- 支持 VFAT（微软公司的 VFAT 长文件名格式）和 VXLONGS（VxWorks 公司的长文件名格式）的目录结构；
- 支持 FAT12、FAT16 和 FAT32 文件分配表格式；

用户要了解 dosFs 文件系统的 API 参考内容，请查阅“VxWorks 操作系统 API 参考手册”中关于 dosFsLib 和 dosFsFmtLib 文件的内容，以及关于 cbioLib、dcacheCbio 和 dpartCbio 文件的内容。

用户要了解 MS-DOS 操作系统，请查阅微软公司参考手册。

⚠ **注意**：本章所讨论的 dosFs 文件系统使用术语“扇区”来表示磁盘上最小可寻址单位，这个术语的含义与大多数关于 MS-DOS 操作系统的文件中的定义一致。但在 VxWorks 操作系统中，磁盘上最小可寻址单位通常称作“块”，并且将一个磁盘设备称作块存取设备。

5.2.1 建立 dosFs 文件系统

本节将分几个步骤介绍在 VxWorks 操作系统中建立一个 dosFs 文件系统的过程，并用流程图表示出来。

在建立 dosFs 文件系统的过程中包括如下几个主要步骤，它们对应于 VxWorks 操作系统在硬件设备（如 SCSI 磁盘）和 I/O 系统之间的不同组件层（图 5-1 中虚线间的部分）。

步骤 1：配置系统内核

利用 dosFs 文件系统、CBIO 和块存取设备组件配置用户系统内核，该步骤将在“5.2.2 配置用户系统”中介绍。

步骤 2：初始化 dosFs 文件系统

如果在用户的工程中已经包括了所需组件，则该步骤将自动执行。该步骤将在“5.2.3 初始化 dosFs 文件系统”中介绍。

步骤 3：创建块存取设备

创建一个块存取设备或一个 CBIO 接口设备（ramDiskCbio）。该步骤将在“5.2.4 创建块存取设备”中介绍。

步骤 4：创建磁盘高速缓冲区

创建磁盘高速缓冲区的过程是可选的，磁盘高速缓冲区只用于旋转存储媒体。该步骤将在“5.2.5 创建磁盘高速缓冲区”中介绍。

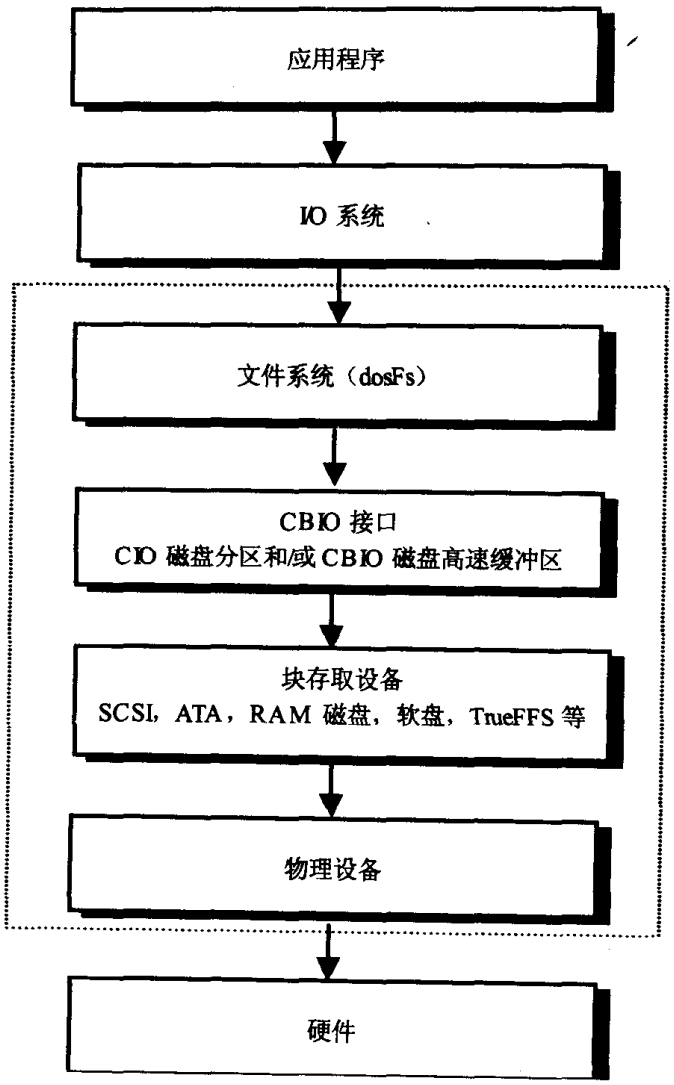


图 5-1　将 VxWorks 操作系统配置成使用 dosFs 文件系统

步骤 5：创建可用的磁盘分区

创建和安装磁盘分区的过程是可选的。该步骤将在“5.2.6 创建和使用磁盘分区”中介绍。

步骤 6：创建 dosFs 文件系统设备

创建所需的 dosFs 文件系统设备。不论用户是否使用一个预先格式化的磁盘，都可以安全地创建 dosFs 文件系统设备。该步骤将在“5.2.7 创建 dosFs 文件系统设备”中介绍。

步骤 7：格式化磁盘卷

如果用户使用了一个未预先格式化的磁盘，则需格式化卷。该步骤将在“5.2.8 格式化磁盘卷”中介绍。

步骤 8：检查磁盘上卷的完整性

用户可以调用 dosFsChkDsk( )函数检测磁盘中卷的完整性，该检测过程是可选的。对大容量磁盘执行磁盘检测功能很费时间。用户在 dosFsChkDsk( )函数中设置的参数决定系统是否自动执行磁盘检测功能。用户欲了解详细信息，请查阅 VxWorks 操作系统参考手册中关于 dosFsChkDsk( )函数的内容。

步骤 9：安装磁盘

通常用户在对磁盘上的文件或目录第一次调用 open( )或 creat( )函数时，系统将自动安装磁盘中的卷。该步骤将在“5.2.9 安装磁盘卷”中介绍。

5.2.2 配置用户系统

用户要使 VxWorks 操作系统支持 dosFs 文件系统，需要使系统内核包括所需的组件。

1. 必须包括的组件

系统内核中必须包括下列组件：

INCLUDE_DOSFS_MAIN——dosFsLib（2）

INCLUDE_DOSFS_FAT——dosFs FAT12/16/32 FAT handler

INCLUDE_CBIO——CBIO API module

系统内核中必须包括下列一个或全部组件：

INCLUDE_DOSFS_DIR_VFAT——Microsoft VFAT direct handler

INCLUDE_DOSFS_DIR_FIXED——Strict 8.3 & VxLongNames directory handler

除此以外，用户还需加入支持块存取设备的组件，如：INCLUDE_SCSI 或 INCLUDE_ATA 组件。最后还需加入用户系统中所需的任何组件。

2. 支持 dosFs 文件系统的可选组件

系统内核中包括的支持 dosFs 文件系统的可选组件有：

INCLUDE_DOSFS——usrDosFsOld.c wrapper layer

INCLUDE_DOSFS_FMT——dosFs2 file system formatting module

INCLUDE_DOSFS_CHKDSK——file system integrity checking

INCLUDE_DISK_UTIL——standard file system operations, such as ls, cd, mkdir, xcopy, and so on

INCLUDE_TAR——the tar utility

3. 支持 CBIO 的可选组件

系统内核中包括的支持 CBIO 的可选组件有：

INCLUDE_DISK_CACHE——CBIO API disk caching layer
INCLUDE_DISK_PART——disk partition handling code
INCLUDE_RAM_DISK——CBIO API RAM disk driver

5.2.3　初始化 dosFs 文件系统

在用户执行任何操作之前，必须先初始化 dosFs 文件系统函数库 dosFsLib。如果系统中包括了必须的组件，则这个过程是自动执行的。

在初始化文件系统过程中调用 iosDrvInstall( )函数将驱动程序装入 I/O 系统的驱动程序表中。分配给 dosFs 文件系统的驱动程序的个数记录在一个全局变量 dosFsDrvNum 中。驱动程序表中指定了使用 dosFs 文件系统的设备执行文件操作的入口地址。

5.2.4　创建块存取设备

在这个步骤中，创建一个或多个块存取设备，用户调用设备驱动程序中的功能函数创建相应的设备。所用的功能函数名称的格式是 xxxDevCreate( )，其中 xxx 表示设备驱动程序的类型，如：scsiBlkDevCreate( )或 ataDevCreate( )。

设备驱动程序中的功能函数返回一个指向块存取设备描述字结构 BLK_DEV 的指针。该结构描述了设备的物理属性，并指定了设备驱动程序中能够在所使用的文件系统中执行的功能函数。用户欲了解有关块存取设备的更多内容，请查阅“4.9.4 块存取设备”一节。

5.2.5　创建磁盘高速缓冲区

如果用户在系统内核中加入了 INCLUDE_DISK_CACHE 组件，就可以通过调用 dcacheDevCreate( )函数为每一个块存取设备创建一个磁盘高速缓冲区。磁盘高速缓冲区可以减少在旋转存储媒体上用于定位数据所花费时间的影响，它并不适用于 RAM 存储盘或 TrueFFS 文件系统设备。在例 5-1 中介绍了如何创建磁盘高速缓冲区。

5.2.6　创建和使用磁盘分区

如果用户在系统内核中加入了 INCLUDE_DISK_PART 组件，就可以通过调用 usrFdiskPartCreate( )和 dpartDevCreate( )函数在磁盘上创建分区，并在分区中安装磁盘卷。

下面的两个程序例子介绍了如何创建和使用分区。在第一个程序中创建磁盘分区并格式化磁盘。在第二个程序中对已经格式化的磁盘创建分区管理器。

例 5-1：创建磁盘设备，在其之上进行分区并创建卷

本例利用一个指向块存取设备的指针，创建了三个分区，并为这些分区建立了分区句柄和 dosFs 文件系统设备句柄。然后调用 dosFsVolFormat( )函数对这些分区进行格式化。

```
STATUS usrPartDiskFsInit
    (
    void * blkDevId             /* 数据类型为 CBIO_DEV_ID 或 BLK_DEV*   */
    )
    {
    const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c" };
    int dcacheSize = 0x30000 ;
    CBIO_DEV_ID cbio, cbio1 ;
    /* 创建磁盘高速缓冲区 */

    if((cbio = dcacheDevCreate(blkDevId, NULL, dcacheSize, "/sd0"))
       == NULL )
       return ERROR ;

    /* 创建分区 */

    if((usrFdiskPartCreate (cbio, 3, 50, 45)) == ERROR)
       return ERROR;

    /* 使用 FDISK 工具创建分区管理器，共有三个分区 */

    if((cbio1 = dpartDevCreate( cbio, 3, usrFdiskPartRead )) == NULL)
       return ERROR;

    /* 创建第一个文件系统，可同时打开 8 个文件并使用 CHKDSK 功能 */

    if(dosFsDevCreate( devNames[0], dpartPartGet(cbio1, 0), 8, 0 ) == ERROR)
       return ERROR;

    /* 创建第二个文件系统，可同时打开 6 个文件并使用 CHKDSK 功能 */

    if(dosFsDevCreate( devNames[1], dpartPartGet(cbio1, 1), 6, 0 ) == ERROR)
       return ERROR;

    /* 创建第三个文件系统，可同时打开 4 个文件，不使用 CHKDSK 功能 */

    if(dosFsDevCreate( devNames[2], dpartPartGet(cbio1, 2), 4, NONE )
      == ERROR)
       return ERROR;
```

```
    /* 格式化第一个分区 */

    if(dosFsVolFormat (devNames[0], 2, 0) == ERROR)
        return ERROR;

    /* 格式化第二个分区 */

    if(dosFsVolFormat (devNames[1], 2, 0) == ERROR)
        return ERROR;

    /* 格式化第三个分区 */

    if(dosFsVolFormat (devNames[2], 2, 0) == ERROR)
        return ERROR;

    return OK;
    }
```

例 5-2：访问已分区的磁盘

在本例中对一个已有三个分区的磁盘进行系统配置。注意 ATA 接口硬盘组件允许自动安装在组件参数中指定个数的分区。

```
STATUS usrPartDiskFsInit
    (
    void * blkDevId             /* 数据类型为 CBIO_DEV_ID 或 BLK_DEV* */
    )
    {
    const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c" };
    int dcacheSize = 0x30000 ;
    CBIO_DEV_ID cbio, cbio1 ;

    /* 创建磁盘高速缓冲区 */

    if((cbio = dcacheDevCreate(blkDevId, NULL, dcacheSize, "/sd0"))
      == NULL )
        return ERROR ;

    /* 使用 FDISK 工具创建分区管理器，共有三个分区 */

    if((cbio1 = dpartDevCreate( cbio, 3, usrFdiskPartRead )) == NULL)
```

```
    return ERROR;

/* 创建第一个文件系统，可同时打开 8 个文件并使用 CHKDSK 功能 */

if(dosFsDevCreate( devNames[0], dpartPartGet(cbio1, 0), 8, 0 )
  == ERROR)
    return ERROR;

/* 创建第二个文件系统，可同时打开 6 个文件并使用 CHKDSK 功能 */

if(dosFsDevCreate( devNames[1], dpartPartGet(cbio1, 1), 6, 0 ) == ERROR)
    return ERROR;

/* 创建第三个文件系统，可同时打开 4 个文件，不使用 CHKDSK 功能 */

if(dosFsDevCreate( devNames[2], dpartPartGet(cbio1, 2), 4, 0)
  ==ERROR)
    return ERROR;

return OK;
}
```

5.2.7　创建 dosFs 文件系统设备

用户通过调用 dosFsDevCreate( )函数（该函数内部调用 iosDrvAdd( )函数）创建 dosFs 文件系统设备。本步骤只是将设备简单地装入 I/O 系统中，并不执行任何 I/O 操作。因此并未安装磁盘。

直到系统执行第一次 I/O 操作，磁盘才被安装。用户欲了解更多内容，请查阅“5.4.4 安装磁盘卷”一节。

5.2.8　格式化磁盘卷

如果用户使用一个未被格式化的磁盘，那么有两种方法可以对其进行格式化。

直接调用 dosFsVolFormat( )函数。该函数需要指定文件分配表（FAT）的格式和目录的格式两个参数（将在下面介绍）。

调用 ioctl( )函数执行 FIODISKINIT 功能。该功能将调用 dosFsLib 文件中的格式化程序。该方法使用默认的磁盘格式和参数。

用户欲了解更多内容，请查阅“VxWorks 操作系统 API 参考手册”中关于

dosFsVolFormat( )和 ioctl( )函数的部分。

MS-DOS 和 dosFs 文件系统在文件分配表的格式和目录格式上提供了多种可选的形式。下面将介绍的几种形式是完全独立的。

> **警告：** 如果用户使用的磁盘上已经使用 MS-DOS 工具（如 FORMAT）建立了 MS-DOS 格式的启动区、文件分配表和根目录，用户就可以调用 dosFsDevCreate( )函数建立一个 dosFs 文件系统设备，而不需要再调用 dosFsVolFormat( )函数，否则文件系统中的数据结构将再次被初始化（格式化）。

1. 文件分配表（FAT）格式

一个磁盘卷上的文件分配表格式是在格式化过程中指定的。通过传递给 dosFsVolFormat( )函数的参数（如：磁盘卷容量或其他用户的设定值）确定文件分配表的格式。表 5-1 列出了文件分配表的选项。

表 5-1　文件分配表的格式

格　式	入口地址长度	适 用 范 围	磁 盘 容 量
FAT12	簇号用 12 比特表示	只用于小容量磁盘，簇个数少于 4084	一般情况下每个簇只有两个扇区
FAT16	簇号用 16 比特表示	用于簇个数少于 65524 的磁盘	一般情况下整个磁盘容量少于 8GB，每个卷容量少于 2GB

2. 目录格式

文件系统的目录格式有三种：

- MSFT 格式长文件名（VFAT）

该目录格式支持对大小写不敏感的长文件名，最长可达 254 个字符。这种目录格式兼容短文件名格式的磁盘。文件系统默认的目录格式是 MSFT 格式长文件名。

- 短文件名（8.3 格式）

该目录格式是对大小写不敏感的 MS-DOS 格式的文件名（8.3 格式）。其中文件名称最多占 8 个字符，文件扩展名占 3 个字符。

- VxWorks 格式长文件名（VxLong）

风河公司产品在支持 MSFT 格式长文件名之前已经支持具有专利技术的 VxWorks 格式长文件名，并且现有产品仍支持装有旧的 dosFs 文件系统 VxLong 目录结构的磁盘。该文件格式最多支持 40 个对大小写敏感的字符（包括所有的 ASCII 字符）。在 MS-DOS 格式文件名中表示文件扩展名的点字符（·）在 dosFs 文件系统 VxLong 格式长文件名中没有特殊含义。

⚠ **注意**：VxWorks 格式长文件中文件容量字段有效位是 40 比特，允许文件容量超过 4GB。

警告：如果用户使用 VxWorks 格式长文件名，那么磁盘将不与 MS-DOS 文件系统兼容。建议用户只有在 VxWorks 操作系统初始化的磁盘上存储系统使用的本地数据时才使用这种文件名格式。

5.2.9 安装磁盘卷

通常在对磁盘上的文件或目录第一次执行 open( )或 creat( )函数时，系统会自动安装磁盘卷。通过调用 ioctl( )函数执行特定功能也可以完成同样操作。

⚠ **注意**：即使在系统启动时存在未被格式化的扇区，或驱动器中未插入可移动磁盘，dosFs 文件系统的初始化过程仍可以成功执行。即使驱动器中未插入可移动磁盘并且该磁盘的配置信息和参数都未知，系统仍可以成功启动并初始化所有的设备。

5.2.10 例子

本小节提供了一个关于上述几小节中讨论的步骤的程序范例。在本例中使用了多个配置参数和设备类型。这些参数和类型有一定的真实性并可以应用于大多数块存取设备。在第一个例子中使用了一个 ATA 接口磁盘。在程序中详细介绍了在 shell 程序中运行的命令（包括用户的输入命令和 shell 程序的输出信息）。在第二个例子中列出了创建并格式化一个 RAM 存储磁盘所需的操作步骤。在最后一个例子中介绍如何初始化已被格式化的 SCSI 接口磁盘。

⚠ **注意**：因为 I/O 系统使用简单的子字符串匹配方式识别设备名，因此文件系统中不能使用单独的斜杠（/）作为文件名，否则系统可能会发生异常错误。

例 5-3：使用 dosFs 文件系统初始化一个 ATA 接口磁盘

本例介绍如何将一个 ATA 接口磁盘初始化成使用 dosFs 文件系统。在例子中列出了通过 VxWorks 操作系统的 shell 工具输入的命令和系统的输出信息。虽然例子中使用 ATA 接口磁盘，但初始化步骤适用于其他块存取设备。

步骤 1：创建块存取设备

创建一个块存取设备（BLK_DEV）用于控制主 ATA 接口控制器（零号控制器）上的主 ATA 硬盘（零号驱动器）。该块存取设备使用整个磁盘。

```
-> pAta = ataDevCreate (0, 0, 0, 0)
new symbol "pAta" added to symbol table.
pAta = 0x3fff334: value = 67105604 = 0x3fff344 = pAta + 0x10
->
```

pAta 现在是一个块存取设备类型的指针（BLK_DEV *），ataDevCreate( )函数返回一个有效值。如果返回值是标志 NULL（0x0），则表明 ataDevCreate( )函数中发生了错误。通常是因为 BSP 配置程序参数或硬件配置参数有错。

该步骤用于任何 BLK_DEV 结构的设备，如：flash 存储盘、SCSI 接口存储盘等。如果用户欲了解相关信息，请查阅有关 ataDevCreate( )函数的内容。

步骤 2：创建磁盘高速缓冲区

本步骤在磁盘上创建一个可选的磁盘数据高速缓冲 CBIO 接口层。

```
-> pCbioCache = dcacheDevCreate (pAta, 0, 0, "ATA Hard Disk Cache")
new symbol "pCbioCache" added to symbol table.
pCbioCache = 0x3ffdbd0: value = 67105240 = 0x3fff1d8
->
```

pCbioCache 是一个指向 CBIO_DEV_ID 结构的指针，用于控制带有 CBIO 接口的磁盘高速缓冲层。如果用户欲了解更多信息，请查阅有关 dcacheDevCreate( )函数的内容。

步骤 3：创建分区

在磁盘设备上创建两个分区，各占用 50%的磁盘空间。该步骤只能在磁盘第一次初始化过程中执行。如果磁盘上已存在分区信息，则不能执行本操作，否则会破坏磁盘上已有的数据。

```
-> usrFdiskPartCreate (pCbioCache, 2, 50, 0, 0)
value = 0 = 0x0
->
```

如果用户欲了解更多信息，请查阅“VxWorks 参考手册”中有关 usrFdiskPartLibCbio( )函数的内容。

在该步骤中，可以用块存取设备指针 pAta 代替 pCbioCache 指针。这样会通过由

usrFdiskPartCreate( )函数内部生成的一个包装器调用 BLK_DEV 结构的程序。

步骤 4：显示分区信息

到目前为止，用户可以显示由 usrFdiskPartCreate( )函数创建的分区。

```
-> usrFdiskPartShow (pAta)
Master Boot Record - Partition Table
-------------------------------------
Partition Entry number 00     Partition Entry offset 0x1be
Status field = 0x80          Primary (bootable) Partition
Type 0x06: MSDOS 16-bit FAT >=32M Partition
Partition start LCHS: Cylinder 0000, Head 001, Sector 01
Partition end   LCHS: Cylinder 0245, Head 017, Sector 39
Sectors offset from MBR partition 0x00000027
Number of sectors in partition 0x00262c17
Sectors offset from start of disk 0x00000027
Master Boot Record - Partition Table
-------------------------------------
Partition Entry number 01     Partition Entry offset 0x1ce
Status field = 0x00          Non-bootable Partition
Type 0x06: MSDOS 16-bit FAT >=32M Partition
Partition start LCHS: Cylinder 0000, Head 018, Sector 01
Partition end   LCHS: Cylinder 0233, Head 067, Sector 39
Sectors offset from MBR partition 0x00262c3e
Number of sectors in partition 0x00261d9e
Sectors offset from start of disk 0x00262c3e
Master Boot Record - Partition Table
-------------------------------------
Partition Entry number 02     Partition Entry offset 0x1de
Status field = 0x00          Non-bootable Partition
Type 0x00: Empty (NULL) Partition
Master Boot Record - Partition Table
-------------------------------------
Partition Entry number 03     Partition Entry offset 0x1ee
Status field = 0x00          Non-bootable Partition
Type 0x00: Empty (NULL) Partition
value = 0 = 0x0
->
```

注意，上面的输出信息显示磁盘上已经建立了两个分区，而且还有两个分区表入口地址空闲。

⚠ **注意**：指向 CBIO 设备 ID 号的指针 pCbioCache 可以代替块存取设备指针 pAta 作为参数传递给 usrFdiskPartShow( )函数。这样会使得该函数调用 CBIO 接口磁盘缓冲区程序访问磁盘设备。

步骤 5：创建分区句柄

在本步骤中用户在磁盘上创建分区句柄。当用户使用了磁盘高速缓冲接口层后，分区句柄代码经常显示在磁盘高速缓冲接口层信息之后。

```
-> pCbioParts = dpartDevCreate (pCbioCache, 2, usrFdiskPartRead)
new symbol "pCbioParts" added to symbol table.
pCbioParts = 0x3ffd92c: value = 67099276 = 0x3ffda8c = pCbioParts + 0x160
->
```

调用 dpartDevCreate( )函数时通知分区层接口需要在磁盘上建立两个分区（分区安装程序最多可以处理 24 个分区），而且用户指定分区管理器（dpartCbio）使用 FDISK 方式的分区安装程序 usrFdiskPartRead( )函数。

用户欲了解更多信息，请查阅有关 dpartDevCreate( )函数的内容。

步骤 6：创建 dosFs2 文件系统

本步骤在每一个分区上创建 dosFs2 文件系统，其中函数中最后一个参数用于配置整体 chkdsk 功能。

```
-> dosFsDevCreate ("/DOSA", dpartPartGet (pCbioParts, 0), 16, 0)
value = 0 = 0x0
-> dosFsDevCreate ("/DOSB", dpartPartGet (pCbioParts, 1), 16, -1)
value = 0 = 0x0
->
-> devs
drv name
0 /null
1 /tyCo/0
1 /tyCo/1
6 ahostname:
3 /DOSA      <---- First Partition
3 /DOSB      <---- Second Partition
value = 25 = 0x19
->
```

本步骤定义了每个磁盘卷的信息并将它们添加到 I/O 系统中。在本步骤中并未格式化

每个磁盘卷，没有执行任何对于磁盘的 I/O 操作，而且未安装磁盘卷。

用户要了解更多的信息，请查阅“VxWorks 操作系统参考手册”中关于 dosFsLib 文件的内容。

步骤 7：按 DOS 格式格式化磁盘卷

本步骤将磁盘卷按 DOS 格式格式化。该步骤只能在磁盘卷第一次初始化时执行一次。如果 DOS 格式的磁盘卷已经被初始化（被格式化），用户可以跳过此步。下面的例子使用默认的参数对系统的磁盘卷进行格式化。

```
-> dosFsVolFormat ("/DOSA", 0, 0)
Retrieved old volume params with %100 confidence:
Volume Parameters: FAT type: FAT16, sectors per cluster 32
2 FAT copies, 0 clusters, 153 sectors per FAT
Sectors reserved 1, hidden 39, FAT sectors 306
Root dir entries 512, sysId (null)  , serial number 8120000
Label:"          " ...
Disk with 2501655 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT16, sectors per cluster 64
2 FAT copies, 39082 clusters, 153 sectors per FAT
Sectors reserved 1, hidden 39, FAT sectors 306
Root dir entries 512, sysId VXDOS16 , serial number 8120000
Label:"          " ...
value = 0 = 0x0
-> dosFsVolFormat ("/DOSB", 0, 0)
Retrieved old volume params with %100 confidence:
Volume Parameters: FAT type: FAT16, sectors per cluster 32
2 FAT copies, 0 clusters, 153 sectors per FAT
Sectors reserved 1, hidden 39, FAT sectors 306
Root dir entries 512, sysId (null) , serial number 9560000
Label:"          " ...
Disk with 2497950 sectors of 512 bytes will be formatted with:
Volume Parameters: FAT type: FAT16, sectors per cluster 64
2 FAT copies, 39024 clusters, 153 sectors per FAT
Sectors reserved 1, hidden 39, FAT sectors 306
Root dir entries 512, sysId VXDOS16 , serial number 9560000
Label:"          " ...
value = 0 = 0x0
->
```

用户要了解更多的信息，请查阅“VxWorks 操作系统参考手册”中关于 dosFsFmtLib 文件的内容。

步骤 8：访问磁盘卷

现在装有 dosFs 文件系统的磁盘卷已经可以访问。注意，磁盘卷/DOSA 将开始执行完成 chkdsk 功能的默认程序，而磁盘卷/DOSB 将不执行类似程序。用户可以通过 dosFsDevCreate( )函数的第 4 个参数设置 Autochk 功能。

```
-> ll "/DOSA"
/DOSA/  - disk check in progress ...
/DOSA/  - Volume is OK
total # of clusters:       39, 085
# of free clusters:        39, 083
# of bad clusters:         0
total free space:          1, 221 Mb
max contiguous free space:  1, 280, 671, 744 bytes
# of files:                0
# of folders:              0
total bytes in files:       0
# of lost chains:          0
total bytes in lost chains: 0
Listing Directory /DOSA:
value = 0 = 0x0
-> ll "/DOSB"
Listing Directory /DOSB:
value = 0 = 0x0
->
-> dosFsShow "/DOSB"
volume descriptor ptr (pVolDesc):       0x3f367c0
cache block I/O descriptor ptr (cbio):  0x3f37be0
auto disk check on mount:               NOT ENABLED
max # of simultaneously open files:     18
file descriptors in use:                0
# of different files in use:            0
# of descriptors for deleted files:     0
# of  obsolete descriptors:             0
current volume configuration:
- volume label:                NO LABEL ; (in boot sector: )
- volume Id:                   0x9560000
- total number of sectors:     2, 497, 950
- bytes per sector:            512
- # of sectors per cluster:    64
- # of reserved sectors:       1
```

```
- FAT entry size:            FAT16
- # of sectors per FAT copy:   153
- # of FAT table copies:      2
- # of hidden sectors:        39
- first cluster is in sector # 339
- Update last access date for open-read-close = FALSE
- directory structure:        VFAT
- root dir start sector:      307
- # of sectors per root:      32
- max # of entries in root:    512
FAT handler information:
------------------------
- allocation group size:      4 clusters
- free space on volume:       1, 278, 771, 200 bytes
value = 0 = 0x0
->
```

从上面显示的信息中，用户可以了解磁盘卷/DOSB 的参数。到此为止，带有文件系统的磁盘卷已经安装成功并可以进行操作。

如果用户要从一个 ATAPI 接口 CD-ROM 驱动器上操作一个 ATA 接口硬盘或访问 CD-ROM 文件系统，可以使用 usrAtaConfig( )函数。该函数一次可以执行多步操作。用户要了解更多的信息，请查阅有关 usrAtaConfig( )函数的内容。

例 5-4：创建并格式化 RAM 磁盘卷

下面的例子将建立一个特定容量的 RAM 磁盘，并将其格式化成使用 dosFs 文件系统。在该例中不是使用 dosFsLib 函数库进行操作，而是使用 ramDiskCbio 功能模块。

```
STATUS usrRamDiskInit
    (
    void                        /* 没有参数 */
    )
    {
    int ramDiskSize = 128 * 1024 ;    /* 共128KB，每个扇区128 bytes */
    char *ramDiskDevName = "/ram0" ;
    CBIO_DEV_ID cbio ;

    /* 每个扇区128 bytes 128 bytes/sec，每个磁道17个扇区，自动分配 */

    cbio = ramDiskDevCreate(NULL, 128, 17, ramDiskSize/128, 0) ;

    if( cbio == NULL )
        return ERROR ;
```

```
    /* 创建文件系统，可以同时打开 4 个文件，不支持 CHKDSK 功能 */

    dosFsDevCreate( ramDiskDevName, cbio, 4, NONE );

    /* 格式化 RAM 磁盘，忽略其中的内容 */

    dosFsVolFormat( cbio, DOS_OPT_BLANK | DOS_OPT_QUIET, NULL );

    return OK;
}
```

例 5-5：初始化 SCSI 磁盘驱动器

下面的例子将一个 SCSI 磁盘初始化成装有文件系统的单个磁盘卷(假定磁盘已经被格式化)。

```
STATUS usrScsiDiskInit
    (
    int scsiId                      /* SCSI 设备的 id 号 */
    )
    {
    int dcacheSize = 128 * 1024 ;     /* 128KB 磁盘高速缓冲区 */
    char *diskDevName = "/sd0" ;      /* 磁盘设备名 */
    CBIO_DEV_ID cbio;                /* 指向 CBIO_DEV 的指针 */
    BLK_DEV *pBlk;                   /* 指向 BLK_DEV 的指针*/
    SCSI_PHYS_DEV *pPhys ;           /*指向一个 SCSI 接口物理设备的指针 */

    /* 创建 SCSI 接口物理设备 */

    if ((pPhys = scsiPhysDevCreate
        (pSysScsiCtrl, scsiId, 0, 0, NONE, 0, 0, 0))== NULL)
        {
        printErr ("usrScsiDiskInit: scsiPhysDevCreate SCSI ID %d failed.\n",
                scsiId, 0, 0, 0, 0, 0);
        return ERROR;
        }

    /* 创建块存取设备 */

    if( (pblk = scsiBlkDevCreate(pPhys, 0, NONE )) == NULL )
        return ERROR;
```

```
/*  使 ids 号小于 10，并检查设备名与 id 号的一致性 */

diskDevName[strlen(diskDevName)-1] += scsiId ;

/* 创建磁盘高速缓冲区 */

if((cbio = dcacheDevCreate(pblk, NULL, dcacheSize, diskDevName))
  == NULL )
   return ERROR ;

/* 创建文件系统，可以同时打开 10 个文件，支持 CHKDSK 功能 */

dosFsDevCreate( diskDevName, cbio, 10, 0 );

return OK;
}
```

5.2.11 对磁盘和磁盘卷进行操作

本小节将讨论有关磁盘和磁盘卷的问题。

用户欲了解更多关于 ioctl( )函数所支持功能的信息，请查阅“5.2.16 dosFsLib 文件支持的 I/O 控制功能”。

1. 使用 Ready-Change 功能宣布更换磁盘

用户可以使用 Ready-Change 功能通知有一个磁盘正被更换。当磁盘 ready-status 有一个变化时，dosFsLib 库会认为必须在下一个 I/O 操作之前重新安装磁盘。使用下面的方法可以宣布发生了 Ready-Change。

调用 ioctl( )函数执行 FIODISKCHANGE 功能。

该操作会使设备驱动程序将 BLK_DEV 结构中的 bd_readyChanged 字段标志设为 TRUE。这与直接通知 dosFsLib 库的效果一样。

使用 cbioRdyChgdSet( )函数在 CBIO 接口层中设置 ready-changed 标志位。

2. 访问磁盘卷的配置信息

dosFsShow( )可以用于显示磁盘卷的配置信息。dosFsVolDescGet( )函数可以获得或验证一个指向 DOS_VOLUME_DESC 结构的指针。用户欲了解更多信息，请查阅相关内容。

3. 同步磁盘卷

当磁盘执行同步操作时，磁盘缓冲区所有已被修改的数据都被写入磁盘中，因此磁盘

中的数据被更新了。该操作包括将数据写入文件中、更新目录和 FAT 信息。为了避免丢失数据，磁盘应在卸载之前执行同步操作。用户欲了解更多信息，请查阅“VxWorks 操作系统参考手册”中有关 close( )和 dosFsVolUnmount( )函数的内容。

5.2.12　目录操作

本小节将讨论有关目录的问题。

1. 创建子目录

对于使用 FAT32 目录结构的文件系统，可以在任何时候、任何目录中创建新的子目录。对于使用 FAT12 和 FAT16 目录结构的文件系统，除非根目录中已经达到最大目录数，也能在任何时候、任何目录中创建新的子目录。新建子目录的方法有如下几种：

使用 ioctl( )函数执行 FIOMKDIR 功能：新建子目录的名称将作为一个参数传递给 ioctl( )函数。

使用 open( )函数：当调用该函数新建子目录时，标志参数中 O_CREAT 选项必须设置为打开，模式参数中 FSTAT_DIR 选项必须设为‘1’。调用 open( )函数后会返回一个描述新目录的文件描述字。用户可以使用该文件描述字对新目录执行只读或关闭操作（当不再需要该目录时）。

使用 usrFsLib 函数库中的 mkdir( )函数。

使用以上几种方法之一新建子目录时，必须指定目录名。该目录名可以是一个带有完全路径的名称或是相对于当前目录的一个名称。

2. 删除子目录

一个将被删除的子目录必须是空目录（除了“.”“..”路径以外）。根目录永远不能被删除。删除子目录有以下几种方法：

使用 ioctl( )函数执行 FIORMDIR 功能，同时指定被删除目录的名称。而且使用的文件描述字可以指向磁盘卷中的任何文件或目录，也可以指向整个磁盘卷本身。

使用 remove( )函数，同时指定被删除目录的名称。

使用 usrFsLib 函数库中的 rmdir( )函数。

3. 读取目录内容

用户可以调用 opendir( )、readdir( )、rewinddir( )和 closedir( )函数在装有 dosFs 文件系统的磁盘卷中寻找所需目录。

用户还可以调用 fstat( )或 stat( )函数获取关于指定文件更多的信息。除了返回标准文件信息以外，这些函数还能返回相应目录中表示文件属性的字节。用户欲了解更多信息，请查阅“VxWorks 操作系统参考手册”中有关 dirLib 文件的内容。

5.2.13　文件操作

本小节将讨论有关文件的问题。

1. 文件的 I/O 操作

使用 creat( )，remove( )，write( )和 read( )这几个 VxWorks 操作系统标准 I/O 操作函数可以创建、删除、读/写一个 dosFs 文件系统设备上的文件。用户欲了解更多信息，请查阅“VxWorks 操作系统参考手册”中有关 ioLib 文件和“4.3 基本 I/O 接口”的内容。

2. 文件的属性

dosFs 文件系统的目录中表示文件属性的字节包括一组标志位，其中的每一位表示一个文件特征。表 5-2 列出了表示文件属性的字节中每一位的含义。

表 5-2　文件属性字节中的标志

标 志 名 称	标志值（16 进制）	说　明
DOS_ATTR_RDONLY	0x01	只读文件
DOS_ATTR_HIDDEN	0x02	隐藏文件
DOS_ATTR_SYSTEM	0x04	系统文件
DOS_ATTR_VOL_LABEL	0x08	卷标
DOS_ATTR_DIRECTORY	0x10	子目录
DOS_ATTR_ARCHIVE	0x20	存档文件

DOS_ATTR_RDONLY 标志

如果该标志位为‘1’，则使用 open( )函数打开的文件不能执行写操作。如果 open( )函数中指定了参数 O_WRONLY 或 O_RDWR，则函数会将错误代码写入 S_dosFsLib_READ_ONLY 结构中。

DOS_ATTR_HIDDEN 标志

dosFsLib 文件将忽略该标志，不执行任何操作。例如在执行目录搜索操作时，系统将报告具有该标志的目录。

DOS_ATTR_SYSTEM 标志

dosFsLib 文件将忽略该标志，不执行任何操作。例如在执行目录搜索操作时，系统将报告具有该标志的目录。

DOS_ATTR_VOL_LABEL 标志

该标志是卷标标志位，表示该目录中包含 dosFs 文件系统卷标。卷标不是必需的，如果存在，则每一个磁盘卷只能有一个卷标并且存在于根目录中。当时用 readdir( )函数读取目录内容时并不显示卷标。用户要获得卷标信息只能通过调用 ioctl( )函数执行

FIOLABELGET 功能。通过调用 ioctl( )函数执行 FIOLABELSET 功能，可以为卷标设置（重新设置）任何少于 11 个字母的字符串作为名称。在调用 ioctl( )函数的过程中可以使用指向该磁盘卷的任何已打开的文件描述字。

DOS_ATTR_DIRECTORY 标志

该标志表示相应目录是一个子目录，而不是一个文件。

DOS_ATTR_ARCHIVE 标志

该标志是用于存档的标志位，当文件被创建或被修改后，该标志位被置‘1’。其他程序使用该标志位搜索修改过的文件并将它们存档。由于 VxWorks 操作系统不清除该标志位，因此执行上述操作的程序必须清除该标志位。

除了子目录和卷标标志位以外，所有文件属性字节中的标志位都能通过调用 ioctl( )函数执行 FIOATTRIBSET 功能而被设置或清除。在属性要修改的文件被打开之后调用该函数。在执行 FIOATTRIBSET 功能时指定的文件属性字节的值可以直接复制。可以使用 stat( )或 fstat( )函数保存原有的标志值和确定新的标志值，并通过位与和位或操作进行修改。

例 5-6：设置 dosFs 文件系统文件属性

在本例中只将一个 dosFs 文件系统中的文件设为只读，没有修改其他属性。

```
STATUS changeAttributes
    (
    void
    )
    {
    int          fd;
    struct stat   statStruct;

    /* 打开文件 */

    if ((fd = open ("file", O_RDONLY, 0)) == ERROR)
        return (ERROR);

    /* 获取目录信息 */

    if (fstat (fd, &statStruct) == ERROR)
        return (ERROR);

    /* 设置只读标志 */

    if (ioctl (fd, FIOATTRIBSET, (statStruct.st_attrib | DOS_ATTR_RDONLY))
        == ERROR)
        return (ERROR);
```

```
/* 关闭文件 */

close (fd);
return (OK);
}
```

⚠ **注意**：用户还可以使用 attrib( )函数修改文件属性。用户欲了解更多信息，请查阅有关 usrFsLib 文件的内容。

5.2.14 分配磁盘空间的方法

dosFs 文件系统使用下面的方法分配磁盘空间。系统选择前两种方法的依据是执行写操作的数据块大小不一样。最后一种方法必须通过用户在程序中指定。

- 单簇分配方法

该方法是以一个簇（最小的磁盘空间分配单位）为分配单位。当执行写操作时的数据量小于一个簇的容量时，系统自动选择该方法。

- 簇集分配方法（连续空间分配方法）

该方法为文件分配一组连续的簇空间，称为上下文。该方法分配的存储空间几乎是连续的。当文件写入的数据量大于一个簇的容量时，系统默认选择该方法。

- 绝对连续空间分配方法

该方法只使用空间上绝对连续的簇。因为是否使用该方法取决于现有的这种类型的空间的容量，因此只在两种情况下使用：（1）在新建一个文件之后立即分配数据空间；（2）从已被分配了连续空间的文件中读数据。使用该方法可能会产生磁盘碎片。

对于上述任何方法，用户可以调用与 POSIX 标准兼容的 ftruncate( )函数或调用 ioctl( )函数执行 FIOTRUNC 功能释放未使用的空间。

1. 选择一种磁盘空间分配方法

在大多数情况下，簇集分配方法比绝对连续空间分配方法更好。因为前一种方法使用几乎连续空间分配方式，能达到几乎最佳的文件访问速度，而且该方法可以极大地减少因采用绝对连续空间分配方法而导致产生磁盘碎片的危险。

绝对连续空间分配方法可以达到磁盘本身的数据传输速率，但这仅比采用簇集分配方法访问文件的速度稍微快一点，而且会经常产生磁盘碎片。因为磁盘在使用一段时间之后，不可能再有足够的连续空间可以继续使用。因此并不能保证对于按绝对连续空间分配方法打开或新建的文件，可以在已有的数据段之后连续地写入新数据。

建议用户执行对性能敏感的操作时，应用程序对磁盘空间的使用应限制在全部磁盘空间 90%以内。当将数据写入磁盘上最后的剩余空间时，不可避免的会产生磁盘碎片，这将严重影响系统性能。

2. 使用簇集分配方法

dosFs 文件系统根据存储媒介的物理特征定义了簇集的大小。对于每一个确定的存储媒介，这个值是固定的。由于定位数据这种降低系统性能的操作经常发生，因此希望一个文件中连续的部分在磁盘中也存储于连续的簇中。当簇集的容量足够大，以至于在一次操作中数据定位所需的时间与执行读/写操作所用的时间相比可以忽略不计时，系统采用簇集分配方法。这种方法有时称为几乎连续空间分配方法，是因为在连续的簇集之间数据定位所需的时间被极大地减少。

因为在一个磁盘卷上的所有大文件都被希望按一组上下文方式写入，而且一旦将它们删除，就可以释放一些上下文用于以后新建的文件，因此只要有可用于存储文件的空间，就存在可以使用的上下文。因此簇集分配方法有效地防止产生磁盘碎片（文件被分配了一些在磁盘上处于较分散位置的较小存储单元）。访问一个带有碎片的文件可能会很费时间，这取决于数据分散的程度。

3. 使用绝对连续空间分配方法

一个连续的文件存储于一系列连续的磁盘扇区中。绝对连续空间分配方法试图为指定的文件或目录分配连续的存储空间，这样可以优化访问文件的操作。用户可以在创建一个文件时采用绝对连续分配方式，或对一个采用这种方式建立的文件再按这种方式执行打开操作。

用户欲了解关于 ioctl( )函数更多的信息，请查阅“5.2.16 dosFsLib 文件支持的 I/O 控制功能”一小节中有关内容。

■ 为一个文件分配连续的存储空间

用户可以采用如下下步骤为一个新建的文件分配连续的存储空间：

使用通常的方式调用 open( )或 creat( )函数新建一个文件。

调用 ioctl( )函数。将 open( )或 creat( )函数返回的文件描述字作为文件描述字参数的值，将功能代码参数设为 FIOCONTIG，将以字节方式表示的所需连续存储区的容量作为第三个参数。

然后，文件分配表会在磁盘中寻找合适的区域。如果找到，该区域就会分配给新建的文件。然后新建的文件就可以关闭或执行其他 I/O 操作。在 ioctl( )函数中使用的文件描述字对于该文件必须是惟一的。应用程序经常在向文件写入数据前调用 ioctl( )函数，执行 FIOCONTIG 功能。

如果用户要获得可用的最大连续存储空间。则需使用 CONTIG_MAX 参数表示连续存储空间的容量。例如：

```
status = ioctl (fd, FIOCONTIG, CONTIG_MAX);
```

■ 为子目录分配空间

子目录也可以按照相同的方式分配连续的磁盘空间：

如果是调用 ioctl( )函数执行 FIOMKDIR 功能建立的子目录，必须执行打开操作来获得一个指向它的文件描述字。

如果是调用 open( )函数新建的目录，则函数返回的文件描述字可以直接使用。

当为一个目录分配连续存储空间时，目录内容必须为空（除了“•”“··”路径以外）。

■　打开并使用一个连续存储的文件

访问一个有碎片的文件需要查询 FAT 表中存放簇地址的链表。但如果文件被认为是连续的，系统可以使用一种增强的方法提高性能。这个方法适用于任何连续存储的文件，不论它们是否是使用 FIOCONTIG 功能建立的。当打开一个文件时，系统会检查文件的完整性。如果文件是连续存储的，文件系统会记录下关于文件的必要信息，以避免以后再次访问 FAT 表。这种操作方式消除了数据定位时间，提高了系统性能。

当用户打开一个连续的文件时，可以通过向 open( )函数设定 DOS_O_CONTIG_CHK 标志，人为地指定该文件是连续的。这样会提醒文件系统从 FAT 表中检索分配给该文件的连续的存储区域。

■　程序范例

用户可以调用 ioctl( )函数执行 FIONCONTIG 功能寻找设备上最大的连续存储空间。也可以调用 dosFsConfigShow( )函数显示该信息。

例 5-7：在一个 dosFs 文件系统的设备上寻找最大的连续存储空间

在本例中，最大连续存储空间的数值（按字节计算）被复制到 ioctl( )函数中作为第三个参数的整型指针中。

```
STATUS contigTest
    (
    void                    /* 没有参数 */
    )
    {
    int count;               /* 最大连续存储空间数 */
    int fd;                  /* 文件描述符 */

    /* 以源模式打开设备 */

    if ((fd = open ("/DEV1/", O_RDONLY, 0)) == ERROR)
        return (ERROR);

    /* 寻找最大连续存储空间 */

    ioctl (fd, FIONCONTIG, &count);

    /* 关闭设备并显示最大连续存储空间 */
```

```
close (fd);
printf ("largest contiguous area = %d\n", count);
return (OK);
}
```

5.2.15 灾难恢复和磁盘卷的一致性问题

DOS文件系统本质上很容易受由于某种类型磁盘更新过程中断而导致的数据结构不一致性的影响。导致更新过程中断的情况包括电源故障、固定磁盘中系统的意外崩溃或人为地拆除磁盘。

⚠ **注意**：DOS文件系统是一个不考虑容错性能的文件系统。

文件的不一致性是由于在文件系统中单独一个文件的数据会存放于磁盘上的三个不同的区域。这三个区域是：

- 文件分配表中的文件链表。它存放于接近磁盘开始的区域。
- 目录路径。可以存放于磁盘中的任何地方。
- 包含文件数据的簇。可以存放于磁盘中的任何地方。

由于这三个区域不可能在异常中断之前被全部更新，因此 dosFs 文件系统包含一个可选的集成的一致性检测机制。可以用于检测磁盘的不一致性以及从不一致性状态中复原。例如，当一个文件正被删除时将磁盘拆除，一致性检测功能会完成文件删除工作；当一个文件正被建立时发生了异常中断，一致性检测功能会认为文件并未被建立。总之，一致性检测功能会尽量纠正不一致性操作。

用户要使用一致性检测功能，应手动向 dosFsDevCreate( )函数中加入 autoChkLevel 参数或调用 chkdsk( )函数。如果系统配置了一致性检测功能，则在下列情况时会调用该功能：

- 安装一个新的磁盘卷；
- 对于固定磁盘进行系统初始化；
- 插入了一个新的可移动磁盘；

⚠ **注意**：一致性检测功能会使系统的速度降低，特别是第一次访问磁盘时。

5.2.16 dosFsLib 文件支持的 I/O 控制功能

dosFs 文件系统支持 ioctl( )函数中的各种功能。在文件 ioLib.h 中定义了这些功能及其相应的常数值。

用户欲了解更多信息，请查阅有关 dosFsLib 文件和 ioLib 文件中 ioctl( )函数的内容。

表 5-3 列出了 dosFsLib 文件支持的 I/O 控制功能。

表 5-3 dosFsLib 文件支持的 I/O 控制功能

功 能 名 称	常数值（十进制）	说 明
FIOATTRIBSET	35	设置 dosFs 文件系统目录中文件属性字节内容
FIOCONTIG	36	为文件或目录分配连续的磁盘空间
FIODISKCHANGE	13	宣布更换存储媒介
FIODISKFORMAT	5	格式化磁盘
FIODISKINIT	6	将磁盘卷初始化成使用 dosFs 文件系统
FIOFLUSH	2	刷新磁盘内容
FIOFSTATGET	38	获取文件状态信息
FIOGETNAME	18	获取指定文件描述字所对应的文件名
FIOLABELGET	33	获取磁盘卷标名称
FIOLABELSET	34	设置磁盘卷标名称
FIOMKDIR	31	创建新目录
FIOMOVE	47	移动一个文件（不重命名）
FIONCONTIG	41	获取设备上最大连续存储空间容量
FIONFREE	30	获取磁盘卷上剩余空间容量
FIONREAD	1	获取文件中未读取字节数
FIOREADDIR	37	读取下一个目录内容
FIORENAME	10	重命名文件或目录
FIORMDIR	32	删除一个目录
FIOSEEK	7	在文件中设置当前字节偏置量
FIOSYNC	21	除了重新读取缓冲区文件数据以外与 FIOFLUSH 功能相同
FIOTRUNC	42	将一个文件截取到指定长度
FIOUNMOUNT	39	卸载一个磁盘卷
FIOWHERE	8	获取当前在文件中的字节位置

5.3 使用 SCSI 设备从本地 dosFs 文件系统启动

VxWorks 操作系统可以从一个本地 SCSI 设备中启动。在用户系统可以启动之前，必须新建一个包含 SCSI 接口设备库的 ROM 启动程序，应该在启动程序中包含 INCLUDE_SCSI，INCLUDE_SCSI_BOOT 和 SYS_SCSI_CONFIG 模块。

在写完 ROM 中的启动应用程序之后，用户可以准备将 dosFs 文件系统作为启动设备。最简单的方法是将 SCSI 设备分区，使 dosFs 文件系统从磁盘的第零块开始运行。然后用户可以编译新的系统程序，将其下载到 SCSI 启动设备中，并从该处启动新的 VxWorks 操作系统。下面列出了详细步骤。

警告：在启动设备中，dosFs 文件系统中的目录名称必须以斜线开始和结束（如在下面例子中出现的/sd0/）。对于 dosFs 文件系统中的命名习惯来说是个特例，并且与 NFS 系统中的设备名不能以斜线结束的要求不一致。

步骤 1：创建 SCSI 设备

调用 scsiPhysDevCreate( )函数创建 SCSI 设备（见“SCSI 驱动器”一节），并将其初始化成使用 dosFs 文件系统（见“5.2.3 初始化 dosFs 文件系统”一节）。通过修改文件“installDir/target/bspName/sysScsi.c”来反映用户对于 SCSI 设备的配置。

步骤 2：重新编译用户系统

需要重新编译用户系统。

步骤 3：复制 VxWorks 系统运行程序

将 VxWorks 操作系统运行文件复制到驱动器中。在下面的例子中一个 VxWorks 操作系统中的任务调用 copy( )函数，该函数需要两个参数：

第一个参数是执行所需的源文件名。源文件是 VxWorks 操作系统的运行程序。源主机名为‘tiamat:’，源文件名是‘C:/vxWorks’。传递给 copy( )函数的参数是连接在一起的形式‘tiamat:C:/vxWorks’。

第二个参数是目的文件名。在本地目标机上 SCSI 磁盘设备中的 dosFs 文件系统名为‘/sd0’，目标文件名为‘vxWorks’。同样，作为一个整体‘/sd0/vxWorks’传递给 copy( )函数。当从一个 SCSI 磁盘设备上启动目标机时，启动程序应将‘/sd0/vxWorks’指定为运行文件。

```
-> sp (copy, "tiamat:c:/vxWorks", "/sd0/vxWorks")
task spawned: id = 0x3f2a200, name = t2
value = 66232832 = 0x3f2a200
Copy OK: 1065570 bytes copied
```

步骤 4：复制系统符号表

根据运行程序的配置信息，系统可能会需要 vxWorks.sym 文件作为系统符号表。因此按同样方式复制 vxWorks.sym 文件。运行程序 VxWorks 从相同的路径中下载 vxWorks.sym 文件。

```
-> sp (copy, "tiamat:c:/vxWorks.sym", "/sd0/vxWorks.sym")
task spawned: id = 0x3f2a1bc, name = t3
value = 66232764 = 0x3f2a1bc
Copy OK: 147698 bytes copied
```

步骤 5：检查复制的文件

在本步骤中，系统列出已复制的文件，确保无误。

```
-> sp (ll, "/sd0")
task spawned: id = 0x3f2a1a8, name = t4
value = 66232744 = 0x3f2a1a8
->
Listing Directory /sd0:
-rwxrwxrwx  1 0      0         1065570 Oct 26 2001 vxWorks
-rwxrwxrwx  1 0      0          147698 Oct 26 2001 vxWorks.sym
```

步骤 6：重新启动并更改系统参数

重新启动系统，并更改系统启动参数。SCSI 设备的启动参数格式如下：

```
scsi=id, lun
```

其中 id 是启动设备的 ID 号，lun 是逻辑单元号（LUN）。为了使目标机能够使用网络，还需在其他字段中加入以太网设备（如 ln 代表 LANCE 公司产品）。

在下面的例子中，系统从一个 SCSI ID 号为 2，LUN 号为 0 的 SCSI 设备上启动。

```
boot device          : scsi=2, 0
processor number     : 0
host name            : host
file name            : /sd0/vxWorks
inet on ethernet (e) : 147.11.1.222:ffffff00
host inet (h)        : 147.11.1.3
user (u)             : jane
flags (f)            : 0x0
target name (tn)     : t222
other                : ln
```

5.4　原始文件系统：rawFs 文件系统

VxWorks 操作系统提供了一个最小化的文件系统——rawFs 文件系统。该文件系统只需提供最基本的磁盘 I/O 操作功能。由 rawFsLib 文件支持的 rawFs 文件系统将整个磁盘卷当作一个单一的大文件。

虽然 dosFs 文件系统在不同程度上也提供类似功能，但如果不需要更复杂的操作功能，rawFs 文件系统在系统所占空间和操作性能上更优越。

用户要在 VxWorks 操作系统中使用 rawFs 文件系统，必须在系统内核中包含 INCLUDE_RAWFS 组件，并设置参数 NUM_RAWFS_FILES 的值，即希望打开文件描述字的最大个数。

5.4.1 磁盘组织形式

rawFs 文件系统对于磁盘中的数据并不采取任何组织形式。它不保存目录信息，因此不将磁盘划分成指定的文件区域，在 rawFs 文件系统的设备中调用 open( )函数时只需指定设备名而不需附加文件名。

整个磁盘的存储空间当作一个文件，可以被任何在该设备上打开的文件描述字使用。所有对于磁盘的读/写操作都使用相对于磁盘上第一个数据块的偏移地址。

5.4.2 初始化 rawFs 文件系统

在进行任何操作前，必须通过调用 rawFsInit( )函数初始化 rawFs 文件系统库——rawFsLib。该函数只需一个参数，即可以同时打开的 rawFs 文件系统的文件描述字的最大个数。该参数用于分配一组文件描述字，每次打开一个 rawFs 文件系统设备时使用一个文件描述字。

rawFsInit( )函数同时在 I/O 系统驱动程序表中为 rawFs 文件系统建立了一个项目（通过调用 iosDrvInstall( )函数）。该项目指定了使用 rawFs 文件系统的设备执行操作的入口点。为 rawFs 文件系统分配的驱动程序存放在全局变量 rawFsDrvNum 中。

通常在 VxWorks 操作系统启动时，由 usrRoot( )任务调用 rawFsInit( )函数。

5.4.3 将设备初始化成使用 rawFs 文件系统

在初始化 rawFs 文件系统之后，下一步需要建立一个或几个应用设备。这些设备是由设备驱动程序中的设备建立函数（xxDevCreate( )）创建。该函数返回一个指向块存取设备描述字结构（BLK_DEV）的指针。BLK_DEV 结构描述了设备物理方面的信息，并指定设备驱动程序中可供文件系统调用的函数。用户欲了解有关块存取设备更多信息，请查阅“4.9.4 块存取设备”小节中的内容。

块存取设备在刚建立之后，没有指定设备名，也没有与文件系统相连。要想将一个已经建立的块存取设备初始化成使用 rawFs 文件系统，必须将该设备与 rawFs 文件系统相连并且分配一个设备名。rawFsDevInit( )函数执行上述操作。该函数的参数是用于确认设备的设备名和一个指向该设备的描述字结构（BLK_DEV）的指针。

```
RAW_VOL_DESC *pVolDesc;
BLK_DEV      *pBlkDev;
pVolDesc = rawFsDevInit ("DEV1:", pBlkDev);
```

调用rawFsDevInit( )函数为设备分配指定的名称，并将设备写入I/O系统的设备表中(通过调用 iosDevAdd( ))。给函数还为设备分配并初始化一个文件系统磁盘卷描述字，并返回一个指向磁盘卷描述字的指针。该指针用于在特定的文件系统调用过程中确定磁盘卷。

注意，将磁盘初始化成使用 rawFs 文件系统并未将磁盘格式化，若想格式化磁盘需调用 ioctl( )函数执行 FIODISKFORMAT 功能。

⚠ **注意**：因为磁盘上没有文件系统结构，所以并不需执行磁盘初始化功能（FIODISKINIT）。但 rawFs 文件系统为了与其他文件系统兼容，也具有 ioctl( )函数中的该项功能。这时 ioctl( )函数不会执行任何操作，并始终返回 OK 标志。

5.4.4 安装磁盘卷

通常在第一次调用 open( )或 creat( )函数时，系统会自动安装磁盘卷（特定的 ioctl( )函数功能也能执行该操作)。在 ready-change 操作之后第一次访问磁盘时会自动地再次进行磁盘卷安装操作。

警告：因为 I/O 系统使用简单的子字符串匹配方式识别设备名，因此文件系统不能使用斜线（/）作为文件名，否则将会发生异常错误。

5.4.5 文件 I/O 操作

要想在一个装有 rawFs 文件系统的设备上执行 I/O 操作，首先应调用 open( )函数打开该设备（也可以使用 creat( )函数，虽然并未新建任何文件)。然后可以使用标准 I/O 操作函数 write( )和 read( )从该设备上读/写数据。用户欲了解更多内容，请查阅“4.3 基本 I/O 接口”。

通过调用 ioctl( )函数执行 FIOSEEK 功能，设置指向文件描述字的字符型指针，即设置执行读/写操作所需的以字节为单位的偏移量。

对于同一个设备可以同时打开多个文件描述字。这些文件描述字必须仔细地管理，以免其中一个文件描述字修改了正被其他文件描述字使用的数据。大多数情况下，使用 FIOSEEK 功能将这些文件描述字的指针指向磁盘不同的区域。

5.4.6 更换磁盘

更换可移动磁盘（如软盘）时，必须通知 rawFs 文件系统。系统提供了两种方法：（1）调用 rawFsVolUnmount( )函数；（2）利用 ready-change 功能。

1. 卸载磁盘卷

声明更换磁盘的第一种方法是在卸载磁盘之前调用 rawFsVolUnmount( )函数。如果可能，该函数将刷新所有已修改的文件描述字缓冲区（见“磁盘卷同步”），并将所有处于打开状态的文件描述字标记为不可用。在下一次 I/O 操作中重新安装磁盘。调用 ioctl( )函数执行 FIOUNMOUNT 功能与使用 rawFsVolUnmount( )函数效果相同。任何处于打开状态的文件描述字在 ioctl( )函数中都可以使用。

在 I/O 操作中，试图使用处于不可用状态的文件描述字将产生 S_rawFsLib_FD_OBSOLETE 错误。通常使用 close( )函数释放处于不可用状态的文件描述字。该操作即使在发生上述错误时也能释放文件描述字。

中断服务程序不能直接调用 rawFsVolUnmount( )函数，因为当设备可用时该操作可能会处于挂起状态。中断服务程序可以向一个卸载磁盘卷的任务传递一个信号量（注意中断服务程序可以直接调用 rawFsReadyChange( )函数，参考“使用 Ready-Change 功能宣布更换磁盘”）。

当调用 rawFsVolUnmount( )函数时，该函数试图将缓冲区中的数据写入磁盘中。这种操作不适用于直到插入新磁盘才发现更换磁盘的情况，因为此时缓冲区中的旧数据会写入新磁盘中。在这种情况下，就像“使用 Ready-Change 功能宣布更换磁盘”一节中所介绍的一样使用 rawFsReadyChange( )函数。

如果在磁盘被拆卸之后才调用 rawFsVolUnmount( )函数，数据刷新操作会失败。但文件描述字仍旧标记为不可用，而且磁盘会被标记为需要重新安装。rawFsVolUnmount( )函数不会产生错误标志。为了避免在这种情况下丢失数据，在拆卸磁盘之前要对磁盘进行同步化操作（参考“磁盘卷同步操作”）。

2. 使用 Ready-Change 功能宣布更换磁盘

宣布更换磁盘的第二种方法是利用 Ready-Change 功能。RawFsLib 文件认为：磁盘就绪状态中的一个变化意味着磁盘在下一次 I/O 操作之前需要重新安装。有三种方式可以宣布就绪状态变化：

- 直接调用 rawFsReadyChange( )函数；
- 调用 ioctl( )函数执行 FIODISKCHANGE 功能；
- 使设备驱动程序将 BLK_DEV 结构中的 bd_readyChanged 字段的标志改为 TRUE，该方法与直接通知 rawFsLib 作用相同。

宣布就绪状态改变不会使缓冲区中的数据写入磁盘。它只是将相应磁盘卷标记为需要

重新安装。因此需写入文件的数据可能会丢失。可以在宣布就绪状态改变之前对磁盘进行同步化操作来避免这种错误。综合使用磁盘同步和宣布就绪状态改变两种方法可以提供除了将文件描述字标记为不可用功能以外，rawFsVolUnmount( )函数的所有功能。

就绪状态改变功能可以从中断服务程序中调用，因为该操作不写入数据或进行其他可能导致延时的操作。

对于只需在插入新磁盘后才检测更换磁盘的设备来说，块存取设备驱动程序中状态检测函数可以很好地判断这种设备的就绪状态改变。（该函数在 BLK_DEV 结构的 bd_statusChk 字段中定义）。在文件系统执行就绪状态检测操作之前，在每次执行 open( )或 creat( )函数的开始时调用该函数。

对于任何一种就绪状态检测方法不可能每次更换磁盘时都能发觉，并在更换磁盘之前关闭与相应磁盘卷有关的文件描述字。

3. 磁盘卷同步操作

当执行磁盘卷同步操作时，缓冲区中所有已修改过的数据都写入物理设备中，完成磁盘内容更新。对于 rawFs 文件系统，已修改的数据指打开的文件描述字缓冲区中的数据。

为了避免丢失数据，需要在更换磁盘之前执行磁盘卷同步操作。根据调用 rawFsVolUnmount( )函数的时间，用户需要特意执行磁盘卷同步操作。

当调用 rawFsVolUnmount( )函数时，系统试图在拆卸磁盘之前对设备进行同步操作。如果此时磁盘未被拆卸而且仍可操作，系统会自动执行同步操作，不需用户干预。

但如果拆卸磁盘之后再调用 rawFsVolUnmount( )函数，很明显已经不能执行同步操作，而且 rawFsVolUnmount( )函数会丢弃缓冲区中的数据。因此在拆卸磁盘之前先调用 ioctl( )函数执行 FIOSYNC 功能（操作员可以通过命令完成该操作）。在 ioctl( )函数中可以使用任何已打开的文件描述字。该操作将所有文件描述字缓冲区中已修改过的数据写入磁盘中。

5.4.7 rawFsLib 文件支持的 I/O 控制功能

表 5-4 列出了 rawFs 文件系统所支持的 ioctl( )函数的功能。在文件 ioLib.h 中定义了这些功能。用户欲了解更多信息，请查阅有关 rawFsLib 文件和 ioLib 文件中 ioctl( )函数的内容。

表 5-4 rawFsLib 文件支持的 I/O 控制功能

功 能 名 称	常数值（十进制）	功 能 介 绍
FIODISKCHANGE	13	宣布更换存储媒介
FIODISKFORMAT	5	格式化磁盘
FIODISKINIT	6	将磁盘卷初始化成使用 rawFs 文件系统
FIOFLUSH	2	功能与 FIOSYNC 相同
FIOGETNAME	18	获取指定文件描述字对应的设备名

续表

功 能 名 称	常数值（十进制）	功 能 介 绍
FIONREAD	1	获取设备中未读取的字节数
FIOSEEK	7	设置设备的字节偏移量
FIOSYNC	21	刷新所有已修改的文件描述字缓冲区
FIOUNMOUNT	39	卸载磁盘卷
FIOWHERE	8	获取当前设备的字节偏移量

5.5　磁带文件系统：tapeFs 文件系统

tapeFs 文件系统库 tapeFsLib 为不使用标准文件或目录结构的磁带设备提供了基本的服务函数。磁带盘与一个源设备很相似，因为可以将磁带盘看作是一个大文件，在这个大文件中的任何数据组织方式都是属于高层结构的。

为了使 VxWorks 操作系统支持 tapeFs 文件系统，需要在系统内核中加入 INCLUDE_TAPEFS 组件。

⚠ **注意**：使用 tapeFs 文件系统时，系统必须具有 SCSI-2 功能，请查阅“SCSI 驱动器”一书内容。

5.5.1　磁带中的组织结构

tapeFs 文件系统对于磁带上的数据不采用任何组织结构。系统不维护目录信息，不会将磁带分成不同的文件区域，不使用任何文件名。对于 tapeFs 文件系统的设备执行打开操作时只需指定设备名，不需附加文件名。

此带设备上打开的任何文件描述字都能访问设备中的全部存储区。所有对于磁带设备的读/写操作都使用相对磁带头部的偏移量作为参数。当一个文件配置成可倒带的设备并第一次打开时，系统从磁带的开始处（BOM）执行相应的操作（见“将设备初始化成使用 tapeFs 文件系统”）。因此，所有关于磁带设备的操作都是与那时磁头的位置有关。tapeFs 文件系统并不维护定位信息。

5.5.2　初始化 tapeFs 文件系统

为了使磁带文件系统可以使用一个物理磁带设备，必须在该设备上初始化 tapeFs 文件系统并在系统中新建一个磁带设备。这样系统才能执行正常的 I/O 操作。

通过调用 tapeFsInit( )函数初始化 tapeFs 文件系统库 tapeFsLib。一个磁带文件系统可以控制多个磁带设备，但是对于每一个磁带设备只允许分配一个文件描述字。因此用户不能在同一个磁带设备上同时打开两个文件。

tapeFsInit( )函数向 I/O 系统驱动程序表中加入一个 tapeFs 文件系统的入口（通过调用 iosDrvInstall( )函数）。该入口指定了在一个使用 tapeFs 文件系统的设备上执行 tapeFs 类型操作的功能函数指针。系统分配给 tapeFs 文件系统的驱动程序号存放在一个全局变量 tapeFsDrvNum 中。

如果执行 tapeFsInit( )函数初始化一个磁带设备，系统会自动调用 tapeFsDevInit( )函数，因此并不需要用户特意进行磁带文件系统初始化操作。

1. 将设备初始化成使用 tapeFs 文件系统

一旦 tapeFs 文件系统已经被初始化，下一步需要建立一个或多个可以使用该文件系统的设备。该操作通过调用顺序存储设备创建函数 scsiSeqDevCreate( )来实现。该函数返回一个指向顺序存储设备描述字结构 SEQ_DEV 的指针。SEQ_DEV 结构描述了设备物理方面的性质以及 tapeFs 文件系统可以调用的功能函数。用户欲了解有关顺序存储设备的内容，请查阅手册中关于“scsiSeqDevCreate( )函数”、“配置 SCSI 驱动器”和“4.9.4 块存取设备”部分的内容。

在顺序存储设备建立成功后，并未被分配一个设备名或相连的文件系统。通过调用 tapeFsDevInit( )函数为顺序存储设备指定一个设备名和分配一个文件系统可以将该设备初始化成使用 tapeFs 文件系统。该函数的参数有：（1）磁带卷名，用于确定设备；（2）一个指向 SEQ_DEV 结构的指针；（3）一个指向已被初始化的磁带设备配置结构 TAPE_CONFIG 的指针，该结构内容如下：

```
typedef struct  /* TAPE_CONFIG tape device config structure */
    {
    int blkSize;
    BOOL rewind;
    int numFileMarks;
    int density;
} TAPE_CONFIG;
```

目前在上述定义中，只有两个字段 blkSize 和 rewind 被使用。如果 rewind 字段标志为 TRUE，则磁带设备在调用 close( )函数关闭文件时会将磁带绕回磁带开始处（BOM）。如果该字段标志为 FALSE，则关闭文件操作将不影响磁带设备中读/写磁头的位置。

blkSize 字段指定实际磁带设备中数据块的大小。指定了数据块容量之后，每一次读/写操作中传输的数据量都按 blkSize 字段中的定义为单位。磁带设备可以执行固定块容量或可变块容量数据传输操作。在 blkSize 字段中对它们的区别也做出了定义。

用户欲了解有关初始化 tapeFs 文件系统设备的内容，请查阅“VxWorks 操作系统 API 参考手册”中关于 tapeFsDevInit( )函数的内容。

2. 固定块存取设备系统或可变块存取设备系统

根据实际物理设备的功能，可以创建一个用于固定块容量数据传输或可变块容量数据传输的磁带文件系统。通常在文件系统中建立磁带设备时，即在调用 tapeFsDevInit( )函数之前，决定采用何种数据传输类型（固定块容量或可变块容量）。容量是零的数据块表示可变块容量数据传输。

一旦特定的磁带设备设置了数据块大小，一般不再改变。可以调用 ioctl( )函数执行 FIOBLKSIZESET 和 FIOBLKSIZEGET 功能获得和设置一个物理设备的数据块容量。

注意对于固定块容量数据传输方式，磁带文件系统对一个数据块容量的数据进行缓存。如果在打开一个文件后修改物理设备的数据块容量的定义，则应将该文件先关闭再打开，这样才能使修改生效。

例 5-8：配置磁带设备

配置磁带设备有很多方式。在下面的程序中，将一个磁带设备数据块的容量设为 512 字节，并将 rewind 字段的标志设为 TRUE。

```
/* 在其他地方分配的全局变量 */

SCSI_PHYS_DEV *   pScsiPhysDev;

/* 局部变量 */

TAPE_VOL_DESC *  pTapeVol;
SEQ_DEV  *       pSeqDev;
TAPE_CONFIG      tapeConfig;

/* 初始化程序 */

tapeConfig.blkSize  = 512;
tapeConfig.rewind   = TRUE;
pSeqDev    = scsiSeqDevCreate (pScsiPhysDev);
pTapeVol   = tapeFsDevInit ("/tape1", pSeqDev, tapeConfig);
```

通过调用 tapeFsDevInit( )函数给设备分配指定的名称并将该设备加入 I/O 系统设备表中（通过调用 iosDevAdd( )）。该函数返回一个指向包含磁带卷的配置信息和状态信息描述字结构的指针。

5.5.3 安装磁带卷

在执行 open( )函数时系统自动执行安装磁带卷的操作。安装操作已经暗含再打开操作之中。

> **警告：** 因为 I/O 系统使用简单的子字符串匹配方式识别设备名，因此文件系统中不能使用单独的斜杠（/）作为设备名，否则系统可能会发生异常错误。

TapeFs 文件系统中的磁带设备只能执行两种类型操作：读操作（O_RDONLY）和写操作（O_WRONLY），不存在读/写操作。在使用 open( )函数打开一个文件时需指定操作类型。

5.5.4 文件 I/O 操作

在对一个 TapeFs 文件系统中的设备执行 I/O 操作之前，该设备必须先由 open( )函数打开，然后使用标准 I/O 操作函数 write( )和 read( )对设备上的数据进行读写操作。用户欲了解更多信息，请查阅“4.7.7 块存取设备”一节内容。

通过调用 ioctl( )函数执行 MTWEOF 功能可以向磁带设备中写入文件结束标志。用户欲了解更多信息，请查阅“5.5.6 tapeFsLib 文件支持的 I/O 控制功能”一节内容。

5.5.5 更换磁盘

当更换可移动存储媒介时（如更换数据磁带），必须通知 TapeFs 文件系统。由 tapeFsVolUnmount( )函数控制卸载磁盘的过程。

在拆卸一个磁带之前首先要卸载。在卸载磁带之前要先关闭已打开的文件描述字。执行关闭操作会将缓冲区中的数据写入磁带中。这样就使文件系统和磁带中的数据保持同步。通过调用 ioctl( )函数执行 FIOFLUSH 或 FIOSYNC 功能可以在关闭文件前执行刷新或同步数据操作。

在关闭了所有打开的文件之后，在拆除磁盘之前，要调用 tapeFsVolUnmount( )函数。一旦磁带设备被卸载了，在下一次 I/O 操作之前必须使用 open( )函数将其重新安装。

中断处理程序不能直接调用 tapeFsVolUnmount( )函数。因为虽然设备可用，但调用该函数也可能使中断处理程序进入挂起状态。中断处理程序可以通过设置信号量提示一个任务卸载磁带卷。

5.5.6　tapeFsLib 文件支持的 I/O 控制功能

表 5-5 列出了 tapeFs 文件系统所支持的 ioctl( )函数中的功能。在文件 ioLib.h、seqIo.h 和 tapeFsLib.h 中列出了它们的定义。用户欲了解更多信息，请查阅“VxWorks 操作系统 API 参考手册”中关于 tapeFsLib 和 ioLib 文件以及 ioctl( )函数的内容。

表 5-5　tapeFsLib 文件支持的 I/O 控制功能

功能名称	常数值（十进制）	功能说明
FIOFLUSH	2	刷新所有已修改过的文件描述字缓冲区
FIOSYNC	21	与 FIOFLUSH 功能相同
FIOBLKSIZEGET	1001	通过向磁带设备发送一个驱动器命令获取其数据块实际容量。将该值与 SEQ_DEV 数据结构中的值比较
FIOBLKSIZESET	1000	设置磁带设备上和 SEQ_DEV 数据结构中数据块容量值
MTIOCTOP	1005	对磁带设备执行类似 UNIX 中 MTIO 的操作。操作类型和操作次数通过 MTIO 结构传递给 ioctl( )函数。在表 5-6 中定义了 MTIO 操作

其中 MTIOCTOP 功能与 UNIX 系统中的 MTIOCTOP 功能兼容。在调用 ioctl( )函数执行 MTIOCTOP 功能时，需要的参数是一个指向 MTOP 结构的指针。该结构包括如下两个字段：

```
typedef struct mtop
    {
    short mt_op;
    int  mt_count;
} MTOP;
```

mt_op 字段包括执行 MTIOCTOP 操作的类型，在表 5-6 中定义了这些操作类型。mt_count 字段内容是执行 mt_op 字段中规定类型操作的次数。

表 5-6　MTIOCTOP 操作

功 能 名 称	常数值（十进制）	功 能 说 明
MTWEOF	0	写文件结束记录或文件标志
MTFSF	1	向前跳过一个文件标志
MTBSF	2	向后跳过一个文件标志
MTFSR	3	向前跳过一个数据块
MTBSR	4	向后跳过一个数据块
MTREW	5	将磁带回绕到开始处

续表

功能名称	常数值（十进制）	功能说明
MTOFFL	06	将磁带回绕到开始处并使设备处于脱机状态
MTNOP	07	不执行操作，只设置 SEQ_DEV 结构中的状态值
MTRETEN	08	夹紧磁带（只用于盒式磁带）
MTERASE	09	擦除磁带中的数据
MTEOM	10	将磁带绕到结尾处
MTNBSF	11	向后倒带至文件开始处

5.6 CD-ROM 文件系统：cdromFs

cdromFs 文件系统库 cdromFsLib 支持应用程序读取按照 ISO 9660 文件系统标准格式化的 CD-ROM 盘中的数据。为了使 VxWorks 操作系统支持 cdromFs 文件系统，需要在系统内核中加入 INCLUDE_CDROMFS 组件。

在初始化 cdromFs 文件系统并将其安装至一个 CD-ROM 块存取设备上之后，用户可以使用标准 POSIX 协议 I/O 操作访问设备上的数据。可执行的操作有：open( )、close( )、read( )、ioctl( )、readdir( )和 stat( )，但调用 write( )函数会产生错误。

cdromFs 文件系统支持同时对多个驱动器、多个打开的文件进行访问。cdromFs 文件系统允许用户在路径名中使用 '/' 和 '\'。不建议用户使用 '\'，因为以后的系统版本可能会不支持这种形式。

cdromFs 文件系统可以使用任何标准 BLK_DEV 结构访问 CD-ROM 中的文件系统。基本的初始化过程与在一个 SCSI 设备中安装 cdromFs 文件系统相似。

用户欲了解有关使用 cdromFs( )函数的信息，请查阅“VxWorks 操作系统 API 参考手册”中关于 cdromFsLib 文件的内容。

例 5-9：创建并使用 CD-ROM 块存取设备

下面的例子介绍了如何在一个 CD-ROM 驱动器上创建一个块存取设备、建立一个 cdromFsLib 设备、安装文件系统和访问设备上的数据。

步骤 1：为 CD-ROM 设备配置用户环境

在用户的工程中加入 INCLUDE_CDROMFS 组件，同时加入其他需要的组件（包括 SCSI/ATA 接口等）。用户欲了解更多信息，请查阅“5.2.2 配置用户系统”一节中的内容。

如果用户使用的是一个 ATAPI 接口设备，就需要对 ataDrv 和 ataResources 结构作相应修改。必须按照用户的硬件平台进行配置。

步骤 2：创建一个块存取设备

根据用户使用设备的类型，选用合适的函数创建一个块存取设备。下面的例子介绍了

如何在辅 ATA 控制器上创建一个主 ATAPI 接口设备。

```
-> pBlkd = ataDevCreate(1, 0, 0, 0)
new symbol "pBlkd" added to symbol table.
pBlkd = 0x3fff334: value = 67105604 = 0x3fff344 = pBlkd + 0x10
```

步骤 3：在 I/O 系统中创建一个 CD-ROM

必须先创建一个块存取设备，然后调用 cdromFsDevCreate( )函数（在其内部调用 iosDrvInstall( )函数），将合适的驱动程序加入 I/O 驱动程序表中。

```
-> cdromFsDevCreate("/cd", pBlkd)
value = 67105456 = 0x3fff2b0
-> devs
drv name
0 /null
1 /tyCo/0
1 /tyCo/1
5 ala-petrient:
6 /vio
7 /cd
value = 25 = 0x19
-> cd "/cd"
value = 0 = 0x0
```

其中 cd 命令在不执行 I/O 操作的情况下改变当前工作目录，因此在安装存储媒体之前可以使用。

步骤 4：安装设备

现在运行 cdromFsVolConfigShow( )函数检查一下前面几步中建立的设备是否已经安装。函数执行的结果显示设备还未安装。

```
-> cdromFsVolConfigShow "/cd"
device config structure ptr     0x3fff2b0
device name                     /cd
bytes per blkDevDrv sector      2048
no volume mounted
value = 18 = 0x12
```

设备只有安装后才能被访问。因为 cdromFs 文件系统设备在第一次执行 open( )函数时安装，因此调用 open( )函数或执行其他使用 open( )函数的操作会安装设备。下例中 ls 命令

将安装设备并显示设备的内容。

```
-> ls "/cd"
/cd/.
/cd/..
/cd/INDEX.HTML;1
/cd/INSTRUCT.HTML;1
/cd/MPF
/cd/README.TXT;1
value = 0 = 0x0
```

步骤 5：检查配置信息

用户可以使用 cdromFsVolConfigShow( )函数检查 CD-ROM 设备的配置信息。

```
-> cdromFsVolConfigShow "/cd"
device config structure ptr     0x3fff2b0
device name                   /cd
bytes per blkDevDrv sector      2048
Primary directory hierarchy:
standard ID                     :CD001
volume descriptor version       :1
system ID                       :LINUX
volume ID                       :MPF_CD
volume size                     :611622912 = 583 MB
number of logical blocks        :298644 = 0x48e94
volume set size                 :1
volume sequence number          :1
logical block size              :2048
path table size (bytes)         :3476
path table entries              :238
volume set ID                   :
volume publisher ID             :WorldWide Technolgies
volume data preparer ID         :Kelly Corday
volume application ID           :mkisofs v1.04
copyright file name             :mkisofs v1.04
abstract file name              :mkisofs v1.04
bibliographic file name         :mkisofs v1.04
creation date                   :13.11.1998  14:36:49:00
modification date               :13.11.1998  14:36:49:00
expiration date                 :00.00.0000  00:00:00:00
effective date                  :13.11.1998  14:36:49:00
value = 0 = 0x0
```

5.7 目标服务器文件系统：TSFS

目标服务器文件系统（TSFS）用于系统开发和诊断目的，它是VxWorks操作系统中功能最全的文件系统，但是它所管理的文件存放于主机系统中。

TSFS文件系统提供了用于远程文件访问（netDrv）的网络驱动程序的所有I/O操作特点，而且不占用任何目标机的资源（除了目标系统和主机上的目标服务器之间通信所需的资源）。TSFS文件系统使用WDB驱动程序从VxWorks操作系统中的I/O系统向目标服务器传送请求。目标服务器使用主机文件系统读取并执行这些请求。当用户使用TSFS文件系统打开一个文件，该文件实际位于主机上，接下来利用open( )函数返回的文件描述字进行read( )and write( )操作都是对主机上已打开的文件进行读/写操作。

同Tornado 1.0版本中VIO驱动程序一样，TSFS文件系统中VIO驱动程序针对于文件的I/O操作，而不是针对控制台操作。TSFS文件系统支持netDrv文件提供的所有I/O操作，而且除了用于目标机与目标服务器之间通信的资源以外，不占用任何其他目标机的资源。该文件系统允许用户程序在不向目标机复制整个文件的情况下随意访问主机中的文件，允许在一个虚拟的文件源中装载一个对象模块并向函数（如moduleLoad( )和copy( )）提供文件名。

每一个I/O操作请求，包括调用open( )函数都是同步执行的，即在操作完成之前调用目标机的任务处于拥塞状态。这样就提供了在控制台的VIO操作中不能提供的流量控制。而且将VIO通道与一个指定主机连接时并不需要发布WTX协议的请求，所需信息已经包含在文件名中。

分析调用read( )函数的过程。驱动程序向目标服务器传递文件的ID号（通过open( )函数建立）、用于接收文件数据的缓冲区地址和读取数据的长度。目标服务器通过在主机上产生相同的read( )函数调用过程来响应目标机的请求，并向目标机中的程序传送读取的数据。由read( )函数产生的返回值以及可能产生的错误号都传送至目标机中，好像所读取的文件存放于目标机中。

用户欲了解更多信息，请查阅“VxWorks操作系统API参考手册”中关于wdbTsfsDrv的内容。

1. 对套接字的支持

TSFS文件系统中对于套接字的操作方式与对文件的操作方式相似，也使用open( )、close( )、read( )、write( )和ioctl( )函数。可以使用下列中的一种文件名形式打开一个TSFS文件系统套接字。

```
"TCP:hostIP:port"
"TCP:hostname:port"
```

函数中的标志字和参数可以忽略。下面的例子介绍如何使用这两种文件名：

```
fd = open("/tgtsvr/TCP:phobos:6164"0, 0)    /* 打开套接字并与服务器 phobos 建立
连接 */
fd = open("/tgtsvr/TCP:150.50.50.50:6164", 0, 0)  /* 打开套接字并与服务器
150.50.50.50 建立连接    */
```

在上例中调用 open( )函数的结果是在主机上打开一个 TCP 套接字并将它与目标服务器套接字中的指定端口相连，目标服务器中的套接字是以主机名或主机 IP 地址的形式提供的。被打开的套接字是非块存取型的，可以使用 read( )和 write( )函数读/写该套接字。因为该套接字是非块存取型的，当套接字中没有可读取的数据时，执行 read( )函数会立即返回一个出错标志及错误号。对于 TSFS 文件系统套接字中使用 ioctl( )函数的方法将在“VxWorks 操作系统 API 参考手册”中的 wdbTsfsDrv 中讨论。上述套接字的配置方法允许 VxWorks 操作系统在目标机中没有 sockLib 文件和网络组件的情况下使用套接字功能。

2. 错误处理

在 TSFS 文件系统中任何时候都可能产生错误。系统会将错误标志及其错误代码一起传回目标机上的函数调用者。产生的错误代码是在 VxWorks 操作系统错误列表中与主机上产生的错误最相近的代码值。如果发生了一个 WDB 错误，那么将传回 WDB 错误信息，而不是一个 VxWorks 操作系统错误代码。

3. 配置 TSFS 文件系统

用户要在 VxWorks 操作系统中使用 TSFS 文件系统，需要在系统内核中加入 INCLUDE_WDB_TSFS 组件，这样就创建了/tgtsvr 文件系统。主机系统中的目标服务器也必须配置成使用 TSFS 文件系统。这样会使主机为 TSFS 文件系统分配一个根目录（见“安全事项”中有关目标服务器的-R 选项的内容）。例如在一台个人电脑主机中用户可以将 TSFS 文件系统的根目录设置成“c:\myTarget\logs”。

配置结束后，打开目标机中的文件“/tgtsvr/logFoo”会导致目标服务器打开主机中的文件“c:\myTarget\logs\logFoo”。该文件的文件描述字将送回目标机中的调用程序。

4. 安全事项

虽然 TSFS 文件系统在很多方面与 netDrv 文件相同，但在安全事项方面不同。在 TSFS 文件系统中，对主机中文件的操作是由目标服务器中的用户发起的，作为启动参数传递给目标服务器的用户名没有任何作用。实际上，对于 TSFS 文件系统的访问权限，任何启动参数都不起作用。

在这种情况下，用户对于 TSFS 文件系统实际的权限不是很清楚。因为用于启动目标服务器的用户 ID 号和主机名在每次启动时都不一样。默认情况下，任何与支持 TSFS 文件系统的目标服务器相连的 Tornado 工具在访问任何文件时与用户启动目标服务器时具有的

权限相同。但目标服务器可以被加锁（使用 L 选项），以限制访问 TSFS 文件系统。

可以将下面的选项加入目标服务器启动程序中，用于控制目标机通过 TSFS 文件系统访问主机中的文件。

-L 选项：锁定目标机服务器

该选项限制具有同一个用户 ID 号（UNIX 系统）或 WIND_UID 号（WINDOWS 系统）的程序访问目标服务器。

-R 选项：设置 TSFS 文件系统的根目录

例如，目标服务器收到的所有文件名前加有一个“-R /tftpboot”字符串，这样“/tgtsvr/etc/passwd”就会映射成“/tftpboot/etc/passwd”。如果在启动程序中不指定“-R”选项，TSFS 文件系统就不会生效，而且目标机发出的 TSFS 文件系统请求就会出错。如果目标服务器在重新启动时没有指定“-R”选项，TSFS 文件系统也会失效。

-RW 选项：使 TSFS 文件系统变为可读/写状态

目标服务器认为可以执行修改操作（包括对文件建立、删除和写入）。如果未指定“-RW”选项，目标服务器的默认状态是只读状态，不允许对文件进行修改。

注意：用户欲了解有关目标服务器和 TSFS 文件系统的内容，请查阅“Tornado 工具手册”中关于 tgtsvr 的内容。用户欲了解有关目标服务器中特定选项的内容，请查阅“Tornado 用户手册”中关于“目标机管理器”的内容。

第 6 章　目标机工具

6.1　简　　介

Tornado 开发系统提供了一整套在主机上安装并运行的开发工具。这种方法保留了目标机内存和资源。然而，许多情况下需要基于目标机的 shell，以及一个基于目标的动态加载器、基于目标机的调试设备或一个基于目标机的系统符号表。这一章详细讨论了基于目标机的设备。

在以下情况，基于目标的工具可能是非常有用的：

- 当通过一个简单串口调试调度系统时；
- 当开发和调试网络协议时，从目标机的视角观察网络是有意义的；
- 为了创建一个动态可配置、能够从目标机的磁盘或通过网络下载模块的系统。

基于目标机的工具是独立的，例如，可以在没有目标加载器时使用目标机 shell，反之亦然。然而，为了充分发挥其他工具的功能，系统符号表是必要的。

有时，可能会同时用到基于主机和基于目标机的开发工具。这时，需要使用附加设备使两种环境保持一致的系统视图。详细内容请看“6.4.4 使用 VxWorks 系统符号表”。

本章简要介绍了这些基于目标的设备并概述了有关最常用的 VxWorks 显示函数。

大多数情况下，基于目标机的设备与 Tornado 主机的相应设备有着相同的工作方式。详细内容请看《Tornado 用户指南》的相应章节。

6.2　基于目标机的 shell

大多数情况下，基于目标机的 shell 工作方式与主机的 shell 相同。有关主机 shell 的详细信息以及主机与目标机 shell 之间的不同，请看《Tornado 用户指南》：shell.，也可以参照 *VxWorks* 关于 dbgLib，shellLib 以及 usrLib 的 API 参考。

6.2.1　主机和目标机 shell 的不同

目标机和主机 shell 之间的主要不同是：

- Tornado 主机 shell 提供额外的命令。
- 无论是主机 shell 还是目标机 shell 都包含一个 C 编译器，主机还提供一个 Tcl 编译器，两种 shell 都提供编辑模式。
- 对于任一个指定的目标机可以有多个主机 shell 起作用，而对于一个目标机一次只有一个目标 shell 起作用。
- 主机 shell 提供虚拟 I/O，而目标机 shell 不提供虚拟 I/O。
- 只要 WDB 目标代理被包含在系统中，主机 shell 就可以执行。目标机和基于目标机的符号表以及模块加载器必须通过包含适当的组件，使其配置成 VxWorks 镜像。
- 目标机 shell 的输入和输出被默认导向同一个窗口，通常与电路板串口的控制器连接。[①]对于主机 shell，这些标准 I/O 流则被导向主机 shell 的同一个窗口。
- 主机 shell 能够不消耗目标机的资源而完全在主机上执行许多控制和信息功能。
- 主机 shell 使用主机资源来完成大多数的功能，从而与目标机保持独立，这意味着主机能够从外部对目标机进行操作。然而目标机 shell 必须在目标机上进行操作，这意味着有一些工作是目标机无法完成的。例如，为了在目标机上完成中断调用，必须要使用 sp( )。另外，当使用目标机 shell 时，可能会发生任务优先级的冲突。

警告：shell 命令使用时必须与函数原型相一致，否则会导致系统暂停（例如使用没有参数的 ld( )）

- 目标机能正确编译函数中的代字操作符，而主机 shell 不能。例如，下面在目标机中被用户 panloki 执行的命令能够正确定位于主机系统上的/home/panloki/foo.o 。

```
-> ld < ~/foo.o
```

- 当目标机遇到表达式中的字符串时，目标机会为字符串分配内存，包括字符串终止符和一些附加的字头。[②]字符串的数值是分配给字符串的内存地址。例如，下面的表达式在目标机存储空间中分配了 12 个字节，将这个字符串输入内存（包括 0 终止符）并将字符串的地址分配给 x:

```
-> x = "hello there"
```

下面的表达式将释放目标机存储器中所分配的存储空间（内存管理的有关信息请看 memLib 参考）

```
-> free （x）
```

进一步，即使字符串没有赋予标识符，内存仍会被永久分配给该字符串。例如，下面的表达式使用了未被释放的内存：

---

① 只要有适当的硬件，一旦 shell 运行 ioGlobalStdSet（），标准输入和输出就可以重新定向。

② 分配的内存数量是不同结构的最小分配单元加上内存块的头部所占用的内存开销。

```
-> printf ("hello there")
```

这是因为如果只是为字符串临时分配内存，一个字符串会传递到作为任务发起的程序，当目标机执行并试图访问该字符串时，目标机 shell 可能已经释放了保留字符串的临时内存（或该临时内存已被重用）。

在使用目标机 shell 进行长期开发工作之后，存储字符串占用的内存越来越多，最终将不得不重新启动目标机。

如果用到字符串，主机 shell 也可以在目标机上分配内存。然而，它不会在目标机上为那些在主机级别执行的命令分配内存（例如 lkup( )，ld( )等）。

6.2.2 用目标机 shell 配置 VxWorks

为了要生成目标机 shell，必须配置 VxWorks 来包含 INCLUDE_SHELL 组件。

必须用组件配置 VxWorks 来获得符号表的支持。（请参见“6.4.1 配置 VxWorks 系统符号表”）

INCLUDE_SHELL_BANNER

显示 shell 标号。

INCLUDE_DEBUG

包含 shell 调试设备。

INCLUDE_DISK_UTIL

包含文件函数如 ls 和 cd。

INCLUDE_SYM_TBL_SHOW

包含符号表显示函数，如 lkup。

包含组件对于模块加载器和卸载器是有用的。（请参见“6.3.1 配置 VxWorks 加载器”）这些组件对于加载和卸载的 usrLib 命令是必要的。（请参见“6.2.4 从目标机 shell 加载和卸载目标模块”）

Shell 任务（tshell）是用 VX_UNBREAKABLE 选项来生成的。因此，不能在任务中设置断点，这是因为 shell 中的断点会使用户无法与系统保持一致。任何从目标机 shell 导入而不是生成（spawned）的程序或任务会在 tshell 的上下文中运行。

一次只有一个目标机 shell 能在 VxWorks 系统上运行。目标机 shell 的 parser 是不可重入的，因为它需要调用 UNIX 工具 yacc 来完成。

6.2.3 使用目标机 shell 的帮助和控制字符

你可以将下列命令键入 shell 显示 help:

```
-> help
```

用 dbgHelp 显示有关调试的命令。

下面的目标机命令列出了所有可用的帮助函数：

```
-> lkup "Help"
```

目标机 shell 有自己的一套终端控制字符，而主机 shell 是从引入它的主机窗口中继承设置。表 6-1 列出了目标机的终端控制字符。前四个是默认值，通过使用 tylib 中的程序能够映射到不同的按键。

表 6-1　目标机 shell 终端控制符

命　令	描　述
CTRL+C	中断或重启 shell
CTRL+H	删除字符（空格）
CTRL+Q	继续输出
CTRL+S	临时暂停输出
CTRL+U	删除一整行
CTRL+X	重新启动（trap to ROM 监视器）
ESC	在输入模式和编辑模式之间往返（只对于 vi 模式）

Shell 的行编辑命令与主机 shell 相同。

6.2.4　从目标机 shell 加载和卸载目标模块

目标模块可以用模块加载器动态地加载入 VxWorks。下面是一个 shell 典型的加载命令，其中用户下载 appl.o 到 appBucket 域：

```
[appBucket] -> ld < /home/panloki/appl.o
```

ld( )命令从文件或标准输入中加载目标模块到一个指定的保护域。在加载过程中将重新解析模块中的外部引用。

一旦应用模块被加载到目标机内存中，能够从 shell 中直接调用模块中的子程序，或将其作为任务发起，或将其连接至中断等。用程序来做什么取决于下载目标模块的标记（flags）（全局标识符或全部标识符的可视性），有关 ld 的详细内容请参见《VxWorks API 参考》usrLib 条目。

模块可以用 reld( )进行重载，这样会在加载新的模块之前卸载具有相同名字的模块。有关 reld 的详细内容请参见《VxWorks API 参考》unldLib 条目。

卸载一个代码模块会从目标机的符号表中清除该模块的符号，从模块清单中清除代码块描述符和 section 描述符，并从存储区中清除相应的 section。

有关目标机加载器和卸载器特性的详细内容请参见“6.3 基于目标机的加载器”。

6.2.5 调试目标机 shell

只要用 INCLUDE_DEBUG 组件配置 VxWorks，目标机 shell 就会具有与主机 shell 相同任务级的调试应用程序。有关调试命令的详细内容请参见《Tornado 用户指南》：shell 和《VxWorks API 参考》的 dbgLib 条目。

使用系统模式来调试目标机 shell 的应用程序是不允许的。

6.2.6 终止目标机 shell 正在执行的程序

有时需要中止 shell 对一条语句的求值过程。例如，一个被调用的函数会发生冗余循环、暂停或等待信号三种情况。这些情况的发生是因为调用过程中指定参数出错、函数执行过程中出错或疏忽了调用程序的后果。在这些情况下，通常需要中止并重新启动目标机 shell 任务。这可以通过按下键盘上特定的目标机 shell 中止符来实现，默认情况下为 CTRL+C。这时目标机 shell 的任务从起始进入点重新执行。注意中止符可以通过调用函数 tyAbortSet( ) 改变。

当重新启动时，目标机 shell 将自动重新分配系统标准输入、输出流，使其与目标机 shell 最初时相同。

如果以下条件为真，中止功能就会工作：

- 调用 dbgInit( )（见“6.2.5 调试目标机 shell”）
- excTask( ) 正在运行（见《Tornado 用户指南》：*配置和构建*）
- 键盘装置支持的驱动程序（所有 VxWorks 提供的驱动器都提供这种支持）

调用函数 ioctl( ) 可以将设备的中止功能选项激活。ioctl( ) 通常在文件 usrConfig.c 的根任务中。有关激活目标机 shell 中止符的内容请参见“tty 选项”。

而且，输入的表达式可能使目标机 shell 发生严重错误，诸如总线/地址错误或优先权冲突。这样的错误通常会导致冲突任务挂起，但允许进一步调试。

然而，当目标机 shell 任务导致这类错误时，VxWorks 会自动重新启动目标机 shell，否则无法进行进一步调试。请注意，由于这个原因，与使用断点和单步（调试）一样，在调试时以任务方式发起一个函数而不是直接从目标机 shell 中调用它是很有益的。

无论目标机 shell 是由于什么原因（致命错误或从终端退出）中止的，任务的迹都会自动显示出来，它显示了中止时目标机 shell 执行到什么地方。

请注意，发生冲突的程序会跳出系统部分，而目标机 shell 状态在中止时不会被清除。例如，目标机 shell 可能已经提取了信号量，但中止时信号量不会被自动释放。

6.2.7　使用远程登录进入目标机 shell

1. 来自主机的远程登录：telnet 和 rlogin

当 VxWorks 最初启动时，目标机 shell 的终端通常是系统的控制器。只要包含 VxWorks 项目设备视图中的 INCLUDE_TELNET 组件（请参见《Tornado 用户指南》：项目），就能通过网络使用 telnet 从主机进入目标机 shell。在定义 INCLUDE_TELNET 过程中会发起 tTelnetd 任务。为了通过网络进入目标机 shell，请在主机上输入下列命令（*targetname* 表示 VxWorks 系统目标机的名称）：

```
% telnet "targetname"
```

UNIX 主机系统也使用 rlogin 从主机进入目标机 shell。包含 VxWorks 项目设备视图中的 INCLUDE_RLOGIN 组件发起 tRlogind 任务。但是请注意，VxWorks 并不支持使用 telnet 或 rlogin 方式从 VxWorks 系统访问主机。

显示在系统控制器上的信息表明用户正在通过 telnet 或 rlogin 进入目标机 shell，而无法再从控制器进入该目标机 shell。

如果远程登录目标机 shell，那么在系统控制器上键入字符是无效的。这是因为目标机 shell 是单用户系统，它只允许从系统控制器或单一的远程登录进入目标机 shell 而不允许同时使用以上两种方法。当在控制器上时，使用 shellLock( ) 应用程序可以防止有人远程登录。

```
-> shellLock 1
```

为了下次远程登录时目标机 shell 有效，可以键入：

```
-> shellLock 0
```

为了结束目标机 shell 远程登录连接，可以从目标机 shell 中调用 logout( ) 函数。为了结束 rlogin 连接，只需键入“~”和“.”并将其作为一行中惟一的字符：

```
-> ~.
```

2. 远程登录安全：

当远程登录 VxWorks 时，VxWorks 将提示键入一个登录用户名和密码：

VxWorks login: *user_name*

Password: *password*

包含 VxWorks 项目设备镜像中的 INCLUDE_SECURITY 组件将会启动远程登录安全功能。系统提供的默认登录用户名和密码分别是 *target* 和 *password*。可以使用函数 loginUserAdd( ) 来改变用户名和密码，如下所示：

```
-> loginUserAdd "fred", "encrypted_password"
```

为了获得加密的密码，可以使用主机系统上的工具 vxencrypt。它会提示键入密码然后显示加密后的形式。

为了定义一组登录用户名，可以包含启动脚本（startup script）中的一组 loginUserAdd( ) 命令，并在系统启动后运行该脚本，或者将这组 loginUserAdd( ) 命令包含到文件 usrConfig.c 中，然后重新构建 VxWorks。

> ⚠ **注意**：*用户名和密码只用于远程登录进入 VxWorks 系统。它们不会影响 VxWorks 与远程系统的网络连接。请参见《VxWorks 网络编程指南》：* rlogin 和 telnet，*主机连接应用*。

在启动时可以通过设置启动行标志参数中的标志位为 0x20 使远程登录安全功能失效（请参见“Tornado 启动”），也可以通过消除 VxWorks 项目设备视图中的 INCLUDE_SECURITY 使该功能失效。

6.2.8 分配 Demangler

Wind River 公司的目标机 shell 作为整套 VxWorks 开发工具中的一部分，包括 demangler 库的 unlinked 版本。这个库于 1991 年 6 月为 GNU 库通用的公共认证 Version 2 所认证。该版本能够在 www.gnu.org 网址或通过从 WindSurf 网站上将源程序下载到 GNU 工具链中。在这个认证下，demangler 库可以在开发过程中不受限制，但是 VxWorks 开发者应该意识到：库的通用公共认证限制了库与应用代码的链接，并且不允许将其转让给第三方。

如果 VxWorks 开发者希望分配 Wind River 公司的目标机 shell 时不受通用公共库认证限制，那么 demangler 库不应该被包含在项目中。

为了将 demangler 从 Tornado 项目构建的内核中消除，只需删除“C++symbol demangler”组件。

为了从 BSP 构建的内核中删除 demangler，只需在 config.h 中定义 INCLUDE_NO_CPLUS_DEMANGLER。

删除这个库不会影响 Wind River 目标机 shell 的运行，但这样会减少 C++标识符和符号的可读性。

6.3 基于目标机的加载器

VxWorks 允许在目标系统运行的时候加载代码。这个操作称为加载或下载，使你能够安装应用程序或扩充操作系统。

下载的代码可以是一组函数，可以由其他代码使用（等价于其他操作系统中的库）；或者也可以是应用程序，可以由一项或一组任务执行。能够被下载的代码单元被称为目标模块。

下载个体目标模块的能力极大地增加了开发过程的灵活性，使开发过程能够以几种不同的方式进行。这种功能的主要用途是在开发过程中卸载、重编译和在开发过程中重载目标模块。另一种方法是将已开发代码与 VxWorks 镜像连接起来，重建这一镜像，并重新启动目标机，这样每一次已开发代码都要被重新编译一遍。

加载器使你能够动态地扩展操作系统，因为一旦代码被加载，这些代码与编译到启动镜像中的已编译代码就没有任何区别。

最后，你可以为每次下载配置不同的加载器，使其机动地为下载的模块分配内存，从而灵活使用目标机的内存。加载器不仅能够动态地为下载的代码分配内存，并释放已卸载模块的内存，而且调用函数能够指定已分配的内存地址。这使得用户能够更加容易地控制内存中的代码分布。有关详细内容请参见“6.3.5 指定加载模块的内存分配”。

VxWorks 目标机加载器的功能包括两部分：其一是加载器，将目标模块的内容安装至目标机系统内存；其二是卸载器，即卸载目标模块。另外，加载器依赖于系统符号表提供的信息。

> ⚠ **注意**：基于目标的加载器通常容易与导入加载器混淆，导入加载器用来在内存中安装内核镜像。尽管这两种工具执行相近的功能，并使用相同的支持代码，它们实际上是两个独立的实体。导入加载器的加载只能完成镜像，而不能完成内存的再分配。

6.3.1 配置 VxWorks 加载器

默认情况下，加载器没有包含在 VxWorks 镜像中。为了使用目标加载器，就必须用 INCLUDE_LOADER 组件来配置 VxWorks。INCLUDE_LOADER 组件将在以后的章节中进一步讨论，VxWorks 的 API 参考中的 loadLib 条目也有所涉及。加入这个组件将自动包含其他的一些组件，这些组件将共同提供完整的加载器功能。

INCLUDE_MODULE_MANAGER

具有管理下载模块以及获得相关信息的功能。详细内容请参见 VxWorks 参考手册中的 moduleLib 条目。

INCLUDE_SYM_TBL

具有存取符号的功能。有关详细内容请参见“6.4 基于目标机的符号表”以及《VxWorks 参考手册》中的 symLib 条目。有关配置符号表的内容请参见“6.4.1 配置 VxWorks 系统符号表”。

INCLUDE_SYM_TBL_INIT

规定了初始化系统符号表的方法。

包含加载器并不意味着自动的包含卸载器。为了将卸载器加入系统，需要包含以下组件：

INCLUDE_UNLOADER

具有卸载模块的功能。有关卸载的其他内容请参见《VxWorksAPI 参考》中的 unldLib 条目。

INCLUDE_SYM_TBL_SYNC

使主机-目标机符号表和模块同步化。

警告： 如果除了 Tornado 主机工具还需要使用基于目标机的符号表和加载器，为了使主机-目标机符号表和模块同步，就必须用 INCLUDE_SYM_TBL_SYNC 组件来配置 VxWorks。有关信息请参见“6.4.4 使用 VxWorks 系统符号表”。

6.3.2 目标机-加载器 API

表 6-2 和表 6-3 列出了用于模块下载和卸载的 API 应用程序和 shell 命令。

表 6-2 用于加载和卸载目标模块的应用程序

程　序	描　述
loadModule( )	加载目标模块
loadModuleAt( )	将目标模块加载到指定的内存位置
unldByModuleId( )	通过指定模块名来卸载目标模块
unldByNameAndPath( )	指定模块名和路径来卸载目标模块

请注意，所有的加载器函数可以从 shell 或代码中直接调用。但是，shell 命令通常只用于 shell 中而不能用于函数。[③]通常，shell 命令处理的一些辅助操作，如打开、关闭一个文件，可以使运行结果和出错信息显示在控制器上。

表 6-3 加载和卸载目标模块的 shell 命令

命　令	描　述
ld( )	加载目标模块
reld( )	通过指定文件名或模块名卸载或重载一个目标模块
unld( )	通过指定文件名或模块名来卸载一个目标模块

[③] 在未来的版本中，调用 shell 的命令将不再被支持。

这些应用程序和命令的使用将在下面讨论。

有关详细内容请参见《VxWorks API 参考》中的 loadLib，unldLib 和 usrLib 条目，以及“6.3.3 加载器选项总结”。

6.3.3 加载器选项总结

加载器的行为可以通过传递给 loadLib 和 unldLib 函数的加载标志符来控制。这些标志符可以组合起来（如使用 OR 操作符），尽管其中一些标志是互斥的。表 6-4，表 6-5 和表 6-6 将这些选项分类列出。

表 6-4　符号注册的加载器选项

选　项	十六进制	描　述
LOAD_NO_SYMBOLS	0x2	模块中没有符号在系统符号表中注册，因此不能链接代码模块。当模块不用于以后的链接操作时，这个选项可以起到扩展系统的作用
LOAD_LOCAL_SYMBOLS	0x4	只有模块中本地的符号在系统的符号表中注册。因此不能链接该代码模块的公共符号。这个选项本身不起什么作用，但它是 LOAD_ALL_SYMBOLS 的基本选项之一
LOAD_GLOBAL_SYMBOLS	0x8	只有模块中的全局符号在系统符号表中注册。不能链接代码模块的私有符号。这是加载器在 loadFlags 参数为 NULL 时的默认状态

表 6-5　代码模块可视性加载器选项

选　项	十六进制	描　述
HIDDEN_MODULE	0x10	代码模块对于函数 moduleShow( )或 Tornado 工具是不可见的。当自动加载模块不应对用户是可见的，该选项对于扩展系统是有用的

表 6-6　处理公共符号的加载器选项

选　项	十六进制	描　述
LOAD_COMMON_MATCH_NONE	0x100	使公共符号保持独立并只对目标模块是可视的。该选项阻止现有符号的任何匹配。除非设置 LOAD_NO_SYMBOLS，公共符号才被加入符号表。该选项为默认选项
LOAD_COMMON_MATCH_USER	0x200	在系统符号表中寻找一个匹配的符号，但只考虑用户模块中的符号而不是起始引导镜像中的符号。如果没有匹配的符号存在，该选项作用与 LOAD_COMMON_MATCH_NONE 相同
LOAD_COMMON_MATCH_ALL	0x400	在系统符号表中寻找一个匹配的符号，考虑范围是所有的符号。如果没有匹配的符号存在，该选项的作用与 LOAD_COMMON_MATCH_NONE 相同

对于选项 LOAD_COMMON_MATCH_USER 和 LOAD_COMMON_MATCH_ALL 如果存在匹配的符号，则处理符号的优先级顺序如下：首先是 bss 段中的符号，然后是 data 段中的符号。如果在某一类型段中存在匹配的符号，将使用最新加入目标机服务器符号表中的符号。表 6-7 为针对断点和沟函数的卸载器选项。

表 6-7 针对断点和沟函数的卸载器选项

选　项	十六进制	描　述
UNLD_KEEP_BREAKPOINTS	0x1	当代码模块被卸载时会留下断点，这样有利于调试，否则所有断点会在卸载模块时会被系统删除

6.3.4 加载 C++模块

如果对于工具链产生文件的加工不充足，加载器就无法正确地处理它们。有关用于下载的 C++文件的说明请参见“7.3.1 细化 C++应用模块”。

当下载和卸载 C++模块时，它可能包括必须自动执行的代码。这些代码由类的静态实例的构造函数和析构函数组成。必须在代码模块执行之前执行构造函数。相似的，析构函数必须在代码模块不再使用之后，以及该模块被卸载之前执行。总的来说，在加载时执行构造函数以及在卸载时执行析构函数是最简单的方法。然而出于调试的目的，用户可能愿意将构造函数和析构函数与加载、卸载步骤分离开来。

由于这个原因，加载器和卸载器关于构造函数和析构函数的行为是可配置的。当 C++策略设置为 AUTOMATIC（1），构造函数或析构函数会在下载或卸载时执行。当 C++策略被设置为 AUTOMATIC（0），构造函数和析构函数不会被加载器和卸载器执行。当 C++策略为 MANUAL 时，函数 cplusCtors( )和 cplusDtors( )可以用来执行关于某一代码模块的构造函数和析构函数。

函数 cplusXtorSet( ) 可以改变 C++运行时的策略。默认设置为 AUTOMATIC。

有关的更多信息请参见《VxWorks API 参考》的 cplusLib 和 loadLib 条目。

6.3.5 指定加载模块的内存分配

默认情况下，加载器分配必要的内存来保持代码模块，也可以使用 loadModuleAt( )命令将目标模块的 text，data 和 bss 段中的任何一个安装在内存中的指定位置。如果指定一个段的地址，那么调用函数就必须在调用该下载程序之前为这个段分配足够的空间。如果没有指定地址，加载器会为所有三种段分配一个足够大的内存空间。

对于任何没有指定地址的段，加载器会为它们分配内存（使用 memPartAlloc( ) 或对于队列式内存使用 memalign( )）。基地址可以设置为 LOAD_NO_ADDRESS 的值，这种情况下，一旦这个段被安装在内存中，加载器会用该段的实际地址取代 LOAD_NO_ADDRESS

的值。

当运用具有 ELF 目标模块格式的结构时，情况会变得复杂。可重定位 ELF 目标文件中信息的基本单位是 section。为了使内存碎片最少，加载器将这些 section 集聚起来，使它们的组合在逻辑上与 ELF 的 segment 相同。简单起见，这些 section 的组合也可以称之为 segment。详细信息请参见“ELF 目标模块格式”。

VxWorks 加载器生成了三种段：text，data 和 bss。当把 section 集聚起来形成 segment 时，这些 section 会以它们在 ELF 文件中的顺序进行排列。有时，在 section 之间有必要增加额外的空间来满足所有 section 排列的需要。在为一个或更多的 segment 分配内存空间时，必须确保有足够的空间使得所有的 section 能够正确地排列。（section 排列要求会在 ELF 格式中进行介绍。二进制应用程序 readelfarch 和 objdumparch 可以用来获取排列信息。）

另外，各个 section 之间填空位的数量取决于基地址的排列。为了保证在预先不知道基地址的情况下有足够的空间，可以分配内存块来满足 segment 中最大 section 的排列要求。例如，数据段（segment）包含了 128 和 264 字节排列空间的 section，根据上面的原则，将分配 264 字节的内存。

在 VxWorks 中其他目标模块格式的基本信息单元与具有单一 text、data 或 bss 单元的 VxWorks 模式十分近似。因此，当处理使用 a.out 或 PECOFF 文件格式的结构时，在 section 之间分配额外空间的问题就不再出现了。

无论安装在什么地方，这些 section 都会被卸载器删除，所以不需要特殊的指令来卸载在指定地址下载的模块。然而，如果基地址被指定在调用程序中，那么作为卸载过程的一部分，卸载器不会释放用于保存 segment 的内存空间。由于分配内存（allocation）是由调用函数完成的，de-allocation 也必须由调用函数完成。

6.3.6　影响加载器行为的限制

下面将介绍下载模块的原则。

1. 可重定位目标文件

可重定位文件是目标文件，其中 text 和 data section 具有临时形式，也就是说一些地址是未知的。可执行文件能够被完全链接，并在一个指定的地址运行。在许多操作系统中，可重定位目标模块是一个位于源文件（.c，.s，.cpp）和可执行文件之间的中间环节，只有可执行文件可以被下载和运行。这种由工具链（toolchains）生成的可重定位文件具有扩展名 a .o。

为了构建一个用于下载的可执行镜像，程序的执行地址和外部定义符号的地址（如库函数）必须是已知的。由于 VxWorks 视图和内存中下载代码的布局是灵活的，因此在主机上运行的编译器无法获得这些信息元。所以，基于目标机加载器处理的代码必须是可重定位方式，而不是可执行方式。

一旦安装在系统内存中，由目标模块的代码、数据和符号组成的实体就被称为代码模块。有关安装代码模块的信息请参见《VxWorks API 参考》moduleLib 下的函数条目。

2. 目标模块格式

可重定位和可执行文件有几种标准格式。在 Tornado 开发环境中，每一个系统支持的目标结构都有一种简单的首选目标模块格式。对于大多数结构而言，首选的格式是可执行的并且是可链接的格式（ELF）。当然也有例外，如 68K 工具链生成的 a.out 目标文件；NT 模拟工具链生成的 pecoff 目标文件。

VxWorks 加载器能够处理工具链生成的大多数目标文件。

⚠ **注意**：并非所有由一种结构所支持的重定位操作一定都是加载器支持的。通常这只会影响用户编写的汇编代码。

3. ELF 目标模块格式

一个可重定位 ELF 目标文件必须有两类元素组成：头部（headers）和节（sections）。headers 描述 sections，而 sections 包含已安装的文本和数据。

一个可执行 ELF 文件是段（segments）的集合，而这些 segment 由 section 组成。VxWorks 加载器将可重定位目标文件集中起来，这与工具链生成可执行 ELF 文件的过程类似。最终的镜像由一个文本段，一个数据段和一个 bss 段组成。（通常一个 ELF 可执行文件对于每种类型会有多个段，但是 VxWorks 加载器使用更简单的模式，即每种类型至多一个段）加载器将以下 section 安装进系统的内存：

文本 section：保持应用程序的指令

数据 section：保持应用程序初始化数据

bss section：保持应用程序未初始化数据

只读数据 section：保持应用程序的常量数据

只读数据 section 被加载器放置在文本段中。

4. a.out 目标模块格式

a.out 格式的目标文件十分简单，与抽象的 VxWorks 代码模块非常相近。与 ELF 文件类似，它们都包含头（headers）和节（sections）。在文件格式的文字表达中，a.out 文件的“sections”有时指 sections，有时指 segments，这种情况甚至发生在同一个工作中，而“sections”确切的意思取决于上下文的含义。由于一个 a.out 文件中只能有一个文本 section，一个数据 section 和一个 bss 的 section，所以 a.out 文件中的 section 等同于 segment。

a.out 目标模块的文本、数据和 bss 节（section）就成为下载代码模块的文本、数据和 bss 段（segment）。

5. PECOFF 目标模块格式

在 VxWorks 中，PECOFF 目标模块格式只应用于 Windows 模拟器结构。

PECOFF 目标模块中的基本信息单元也称为 section。一个 section 的大小受到 PECOFF 文件格式的限制。因此，尽管并不多见，在一个 PECOFF 文件中可以有多个文本或数据 section。

VxWorks 目标机加载器只能处理至多含有一个文本 section、一个数据 section 和一个 bss section 的目标模块。这些 section 成为 VxWorks 代码模块的文本、数据和 bss 的 segment。

6. 链接和引用解析

VxWorks 加载器执行的任务与传统链接器相同，因为它也是为执行环境准备目标模块的代码和数据。这还包括模块的代码和数据与其他代码数据的链接。

然而，加载器与一个传统的链接器不同，因为加载器直接在目标机系统的内存中完成这项工作，而不是生成一个输出文件。

此外，加载器使用 VxWorks 系统中已存在的应用程序和变量而不是使用库文件来为下载的目标模块重新定位。系统符号表（请参见“6.4.4 使用 VxWorks 系统符号表”）用来存储已经安装在系统中的函数以及变量的名字和地址。这样做的缺点是一旦符号安装到系统符号表中，它们就能被任何一个下载模块链接。而且，当试图解析模块中未定义的符号时，加载器会使用在目标机镜像中编译的所有全局符号，和所有以前下载模块的全局符号。通常作为下载过程中的一部分，模块提供的所有全局符号都在系统符号表中注册。你可以使用 LOAD_NO_SYMBOLS 下载标志来越过这一步（见表 6-4）。

系统符号表允许发生名字冲突，例如，假设系统中有一个名为 func 的符号，又有一个 func 符号作为下载的一部分被加入到系统符号表中。从这时起，所有对 func 的链接都指向最近下载的符号（“6.4.1 配置 VxWorks 系统符号表”）

7. 加载的顺序原则

VxWorks 加载器按照顺序原则加载代码模块。也就是说，每一个独立的代码模块需要独立的下载。假设一个用户有两个称为 A 模块和 B 模块的代码模块，A 模块引用了 B 模块中的符号。用户既可以使用基于主机的链接器将 A 模块和 B 模块组合成一个模块，也可以先下载 B 模块再下载 A 模块。

当代码模块被下载时，他们会不可逆的链接到现有的环境中。也就是说，一旦建立从模块到外部符号的链接，该链接不会在卸载和重载模块时发生改变。

因此，当下载模块时，必须要考虑模块之间的关联，以保证每一个新模块的引用既能够用 VxWorks 镜像中经过编译的代码解析，也可以用已经下载到系统中的模块来解析。否则，会导致解析代码不完全，这在下载过程的末期将妨碍未定义符号的引用。为了查错，加载器会显示一列缺失符号。这种解析不完全的代码不应该被执行，因为当试图执行一条重定位不当的指令时，系统的行为是无法预测的。

通常，如果下载失败，这种只安装了一部分的代码会被删除。然而，如果只是一些符号未被解析，该代码不会被自动卸载。这时的用户能够检查下载失败的结果，甚至能够执行已完全解析的那部分代码。因此，含有未解析符号的代码模块只能用独立的卸载命令 unld( )来删除。

请注意，VxWorks 加载器顺序原则意味着卸载一个用于解析另一个 code module 的代码模块，会导致引用不存在的代码和数据。如果执行这种悬空引用（dangling references ）的代码，结果是不可预测的。

8. 解析公共代码

公共代码向 VxWorks 加载器提出了挑战，因为它面对的不再是传统的链接器。参考下面的例子：

```
#include <stdio.h>

int willBeCommon;

void main (void) {}
    {
    ...
    }
```

符号 willBeCommon 未被初始化，所以技术上它是一个未定义的符号。这种情况下，许多编译器会生成一个‘common’符号。

ANSI C 允许多个目标代码定义具有相同名称的未初始化的全局符号。链接器用于解决不同模块中符号的引用，这是通过符号的链接，而不是符号的惟一实例化来实现的。如果不同的引用规定不同的大小，链接器会用最大的尺寸定义一个符号，然后将所有的引用与该符号链接。

所有的模块在链接发生时都已经存在，这并不是很难。然而，由于 VxWorks 模块是被顺序下载的，在独立的下载操作中，加载器无法知道那一个模块会被下载，其后又会遇到哪一组公共代码。因此，VxWorks 目标加载器有一个选项用来控制公共代码的链接。loadLib API 函数的默认行为就是相同对待公共符号与没有匹配的引用（LOAD_COMMON_MATCH_NONE），结果每一个下载模块有其自己的符号复制。其他可能的行为，允许把公共符号看成是符号表中没有任何匹配的符号（LOAD_COMMON_MATCH_ALL），或者看成是在初始启动镜像中的没有任何匹配的符号（LOAD_COMMON_MATCH_USER）。

这个规定的公共符号匹配选项可以在每一个使用 loadLib API 的调用过程中设定。当对加载器把不同的公共匹配行为混在一起时要特别小心。挑选一个匹配行为并用它来进行所有的下载会更为安全。有关每一个选项下匹配行为的详细内容请参见表 6-6。

⚠ **注意**：请注意 shell 下载命令 ld，它可以控制处理公共符号以及控制不同的默认行为。详细内容请参见 “usrLib 参考条目”。

6.4　基于目标机的符号表

符号表是一个存储信息的数据结构，这些信息描述了所有模块中的函数、变量和常量以及 shell 中生成的所有变量。使用符号表库能够操作两种不同类型的符号表：用户系统表和最常用的系统符号表。

1. 符号项

表中的每一个符号都包含三部分：

name

name 是一个字符串，来源于源代码中的名称。

value

value 通常是符号所指元素（element）的地址：既可以是函数的地址，也可以是变量的地址（即变量内容的地址）。

type

type 提供了有关符号的附加信息。对于系统符号表中的符号，它的类型是 installDir/target/h/symbol.h 中定义类型中的一种。例如，SYM_UNDF，SYM_TEXT 等。对于用户符号表，该字段可以是用户定义的。

2. 符号更新

当模块下载到目标机或从目标机上卸载时，符号表都会更新。可以使用列于表 6-4 中的加载器选项来控制存储在符号表中的精确信息。

3. 查找符号库

可以很容易在所有的符号表中查找特定的符号。为了从 shell 中查找，可以使用 lkup( )。详细内容请参见 “lkup( )参考条目”。也可以使用符号库 API 进行检索，可以根据地址、名字、类型以及将用户提供的功能应用于符号表中每一个符号的操作，在符号表中进行搜索。详细内容请参见 “symLib 参考条目”。

6.4.1　配置 VxWorks 系统符号表

对于所有的符号表基本配置都是必需的。这种配置提供了符号表库，这对于用户符号

表来说是足够的。然而，这种配置并不足以生成系统符号表。本节将描述基本配置和生成系统符号表所需的附加配置。

有关用户符号表的信息请参见“6.4.6 生成用户符号表”。有关系统符号表的内容请参见“6.4.4 使用 VxWorks 系统符号表”。

1. 基本配置

对于一个符号表来说，包含组件 INCLUDE_SYM_TBL 是最基本的配置。它提供了基本符号表库 symLib（不等同于系统符号表）以及足够的配置来生成用户符号表。符号表库组件包括配置选项，使用户能改变符号表的宽度并控制是否允许名字冲突的发生。

2. 散列表宽度

INCLUDE_SYM_TBL 组件包含了一个允许用户改变默认符号表宽度的配置参数。参数 SYM_TBL_HASH_SIZE_LOG2 定义了符号表的散列表宽度。它为正值并且是 2 次方幂。SYM_TBL_HASH_SIZE_LOG2 的默认值为 8；这样，符号表的默认宽度是 256。参数值越小，需要内存就越少，但会降低查找的效率，因而耗费更多的时间。有关改变参数值的内容请参见《Tornado 用户指南》：配置和构建。

3. 处理名称冲突

系统符号表 sysSymTbl 可以配置成允许名称冲突或不允许名称冲突。用于调用 symTblCreate( ) 的标志符决定重叠的名称是否能够进入符号表。如果设置为 FALSE，符号名只允许出现一次。当允许发生名称冲突时，在具有相同名称的几个符号中，当按名字查找符号表时，返回的是最新加入的符号。

4. 系统符号表配置

为了包含内核中现有符号的信息，从而使 shell、加载器和调试设备正常工作，必须生成并初始化系统符号表。初始化时，可以包含下列组件来生成一个系统符号表：

INCLUDE_STANDALONE_SYM_TBL.

生成一个内部系统符号表，其中系统符号表和 VxWorks 镜像包含在同一个模块中。这种符号表将在“6.4.2 生成一个内部系统符号表”中介绍。

INCLUDE_NET_SYM_TBL .

生成一个分立的系统符号表，如下载到 VxWorks 系统的.sym 文件。这种符号表将在 6.4.3 生成一个可下载的系统符号表中介绍。

在系统初始化、系统符号表初次建立时表中不包含符号。符号必须在运行时加到表中。这些组件中的每个组件都以不同方式添加符号。

⚠ **注意**：在 BSP 目录下而不是在项目设备（project facility）中生成符号表时，只包含组件 INCLUDE_STANDALONE_SYM_TBL 是不够的。我们需要编译 vxWorks.st.，这种配置面向 standalone 镜像，不会对目标机

上的网络设备进行初始化。为了在使用 standalone 符号表的同时也可使用目标机网络设备，需要用项目设备（project facility）来配置镜像。详细内容请参见《Tornado 用户指南》: “Projects”。

6.4.2　生成一个内部系统符号表

一个内部系统符号表将信息复制至包装代码（wrapper code），然后这些代码在构建系统时被编译并与内核建立链接。

1. 生成系统符号信息

一个内部系统符号表依靠应用程序 makeSymTbl 来获取符号信息。这个应用程序使用 GNU 函数 nmarch 生成有关包含在镜像中符号的信息。然后，它将这些信息制备成文件 symTbl.c ，该文件包含一个名为 standTbl 的 SYMBOL 类型的数组（见符号条目）。数组中的每一项都有符号名和类型字段集。地址（value）字段不会被 makeSymTbl 填充。

2. 编译和链接符号文件

symTbl.c 文件像普通的.c 文件一样编译，并与 VxWorks 镜像的其他部分建立链接。作为普通链接过程的一部分，工具链链接器会将每一个全局符号的正确地址写入数组。当构建完成时，镜像中的符号信息就准备好了，它是作为 VxWorks 符号的一个全局数组。在系统初始化过程中，内核镜像被加载到目标机内存后，来自全局数组的信息会被用来构建系统符号表。

在下列文件中（只在构建符号表之后）可以发现 standTbl 数组的定义。这些文件是作为构建 VxWorks 镜像的一部分生成的。

installDir/target/config/*BSPname*/symTbl.c
用于直接从 BSP 目录中的镜像建立

installDir/target/proj/projDir/buildDir/symTbl.c
用于使用项目设备时的镜像建立

3. 使用内部系统符号表的优点

尽管这种方法生成了 VxWorks 镜像（模块文件），该镜像比没有符号时大。这种内部符号表有一些优点，它们在下载链接很慢时会起作用。这些优点包括：

如果不使用目标加载器，它能够节省内存，因为不需要把目标加载器（以及相关的组件）配置进系统中。

它不需要目标机与主机接通（与可下载系统符号表不同）。

它比下载两个文件更快（镜像和.sym 文件），因为文件的网络操作[④]比数据传输到内存

[④] 使用 open（），seek（），read（）和 close（）。

花费更长的时间。

对于没有网络连接、基于 ROM 的扩展系统是很有用的，但是需要 shell 作为用户接口。

6.4.3 生成一个可下载的系统符号表

可下载符号表在一个分立的目标模块文件中被建立。这个文件从系统镜像被单独下载到系统中，同时也将信息复制到符号表中。

1. 生成.sym 文件

可下载符号表使用 vxWorks.sym 文件而不使用 symTbl.c 文件。vxWorks.sym 文件是通过 objcopy 函数删除符号信息以外的所有 section 来生成的。这些符号信息放在一个名为 vxWorks.sym 的文件中。

VxWorks 镜像使用 ELF 目标模块格式的结构，vxWorks.sym 文件也是 ELF 格式，并只包含一个 SYMTAB section 和一个 STRTAB section。通常，vxWorks.sym 文件与其他由工具链编译的其他目标文件具有相同的目标格式。

2. 下载文件.sym

在启动和初始化过程中，加载器直接调用 loadModuleAt( )来下载 vxWorks.sym 文件。为下载 vxWorks.sym 文件，加载器使用了当前默认的装置（请参见“4.2.1 文件名称和默认设备类型”）。

为了下载 VxWorks 镜像，加载器在下载时也使用了当前的默认装置。因此，下载 vxWorks.sym 文件的默认装置可能是同一装置。这是因为默认装置可以由其他运行的初始化代码设置或重置。进行这种设置或重置只在下载 VxWorks 镜像之后，同时在下载符号表之前。

然而，在标准 VxWorks 配置中，由于不包含 customized 系统初始化代码，在下载 vxWorks.sym 文件时的通常用协议 rsh 或 ftp 将网络设备设置为默认装置。

3. 使用可下载系统符号表的优点

使用可下载符号表时要求具有加载器组件。因此，系统必须有足够的内存用于加载器以及其相关的组件。使用可下载符号表的优点总结如下：

- 如果加载器需要用于其他用途，仍然很方便。
- 比建立一个内部系统符号表速度更快。

6.4.4 使用 VxWorks 系统符号表

一旦初始化，VxWorks 系统符号表就会包含启动编译镜像中所有全局符号的完整的名称和地址。在目标机上需要这些信息来激活目标工具库的全部功能。例如：

目标机工具保持系统符号表，这样它就能为所有静态编译到系统中或是动态下载的代码包含数据名称和地址信息。（可以使用选项 LOAD_NO_SYMBOLS 隐藏下载模块，这样它们的符号就不会出现在系统符号表中，请参见表 6-5）

如果非操作系统代码需要系统符号库提供设备，就需要生成一个符号表并使用上述的符号库对其进行操作。

当以下任一情况发生时，符号会动态地加入系统符号表或从表中删除：

- 当加载或卸载模块时
- 当使用 shell 动态生成变量时
- 目标机的符号表通过 wdb 代理实现与主机中的符号表保持一致

系统符号表与其他目标及工具的确切关系如下所述：

加载器

加载器需要系统符号表，而加载器对于系统符号表不是必需的。基于目标机的加载器和代码模块管理需要系统符号表，这是因为加载的代码必须要与目标机上已经存在的代码进行链接。

调试工具

基于目标的符号调试，设备和用户命令如 i 和 tt，需要依靠系统符号表提供有关任务进入点、调用堆栈的符号内容等信息。没有系统符号表，它们仍然可以继续使用，但它们的效用将大大降低。

目标机 shell

目标机 shell 严格意义上并不需要系统符号表，但如果没有符号表，目标机 shell 的功能发挥将受到很大的限制。目标机 shell 需要系统符号表中函数的符号名来运行这些函数。基于目标机的 shell 使用系统符号表来执行 shell 命令、调用系统函数以及编辑全局变量。目标机 shell 也包含 usrLib 库，该库包含有 i，ti，sp，period 和 bootChange 等命令。

wdb 代理

作为与主机之间符号同步的一部分，wdb 代理将符号加入系统符号表。

> ⚠ **注意**：如果你选择同时使用基于主机的和基于目标机的工具，你可以使用同步的方法来保证基于主机和目标机的工具共享同一组符号。同步化只能够应用于符号和模块，无法应用于其他信息如断点。

6.4.5　基于主机和目标机的符号表同步化

另一个符号表——目标机服务器符号表——位于主机上，并由主机使用和保存。在本

章中，该符号表即为主机符号表。主机符号表的存在不依赖于目标机上是否有系统符号表存在。有一种依赖于 wdb 代理的可选机制，该机制可以用来同步化主机和目标机符号表信息（INCLUDE_SYM_TBL_SYNC）。为了激活主机上的这种机制，目标机服务器必须使用 -s 选项启动。有关为目标服务器设置该选项的详细内容请参见《Tornado 用户指南》。

6.4.6 生成用户符号表

尽管一个用户应用可以对系统符号表中的符号进行操作，但不推荐这种行为。向符号表中加入符号或从表中删除符号设计通过操作系统库来完成。因此，系统符号表的任何其他行为都会干扰操作系统的正常运行，符号的加入甚至会对模块的链接造成负面的和不可预料的影响。

因此，用户定义的符号不应加入到系统符号表。如果应用需要一个自己的符号表，这时需要生成一个用户符号表。详细内容请参见《VxWorks API 参考》的 symLib 条目。

6.5 显 示 函 数

VxWorks 包含了系统信息函数，该函数显示特定对象或服务的相关系统状态。然而，它们在调用时显示的只是系统服务的概况，不能反映系统的当前状态。如果要使用这些函数，必须要定义相应的宏配置（请参见《Tornado 用户指南》：项目）。当调用它们时，它们的输出会被送到标准输出设备。表 6-8 列出了常用的系统显示函数。

表 6-8 显示函数

调 用	描 述	宏 配 置
envShow( )	显示 stdout 上指定任务的环境	INCLUDE_TASK_SHOW
memPartShow( )	显示分区块和统计数据	INCLUDE_MEM_SHOW
memShow( )	系统内存显示函数	INCLUDE_MEM_SHOW
moduleShow( )	显示所有下载模块的统计数字	INCLUDE_MODULE_MANAGER
msgQShow( )	信息队列显示函数（POSIX 和 Wind）	INCLUDE_POSIX_MQ_SHOW INCLUDE_MSG_Q_SHOW
semShow( )	信号量显示函数（POSIX 和 Wind）	INCLUDE_SEM_SHOW INCLUDE_POSIX_SEM_SHOW
show( )	通用显示函数	
stdioShow( )	标准 I/O 文件指针显示函数	INCLUDE_STDIO_SHOW
taskSwitchHookShow( )	任务转换列表显示函数	INCLUDE_TASK_HOOKS_SHOW
taskCreateHookShow( )	任务生成列表显示函数	INCLUDE_TASK_HOOKS_SHOW

续表

调　用	描　述	宏 配 置
taskDeleteHookShow( )	任务删除列表显示函数	INCLUDE_TASK_HOOKS_SHOW
taskShow( )	显示任务控制块的内容	INCLUDE_TASK_SHOW
wdShow( )	Watchdog 显示函数	INCLUDE_WATCHDOGS_SHOW

浏览系统信息的另一种方法是使用 Tornado 浏览器，它可以配置为定期更新系统信息。这种工具的详细信息请参见《Tornado 用户指南》中的浏览器部分。

VxWorks 也包含网络信息显示函数，《VxWorks 网络编程指南》一书对此有详细描述。这些函数可以通过在 *VxWorks* 配置中定义 INCLUDE_NET_SHOW 来实现初始化。请参见《*Tornado* 用户指南》：配置和建立。表 6-9 列出了经常调用的网络显示函数。

表 6-9　网络显示函数

调　用	描　述
ifShow( )	显示链接的网络接口
inetstatShow( )	显示 Internet 协议套接字的所有现行连接
ipstatShow( )	显示 IP 统计数据
netPoolShow( )	显示网络缓冲区统计数据
netStackDataPoolShow( )	显示网络堆栈 pool 数据
netStackSysPoolShow( )	显示网络堆栈系统 pool 数据
mbufShow( )	报告 mbuf 统计数据
netShowInit( )	初始化网络显示函数
arpShow( )	显示系统 ARP 表中的入口
arptabShow( )	显示已知的 ARP 入口
routestatShow( )	显示路由统计数据
routeShow( )	显示主机和网络路径表
hostShow( )	显示主机表
mRouteShow( )	显示路由表入口

6.6 常 见 问 题

本节列举了在使用目标机工具时经常会遇到的问题。

1. 目标机 shell 调试未触发断点

在目标机调用的函数中设置了一个断点，但断点没有被触发，为什么？

解决方法

不要直接运行应用函数，使用 taskSpawn( ) 运行用户的应用函数。

解释

目标机 shell 任务运行时会默认 VX_UNBREAKABLE 选项，那些直接从目标机 shell 的 prompt 命令调用的函数只能在目标机 shell 任务的上下文中执行。因此，在直接调用的函数中设置的断点不会被触发。

2. 内存不足

加载器报告内存不足无法下载模块。然而检查内存表明可用内存是足够的。发生了什么事情？我该怎样做？

解决方法

使用不同的设备下载文件。从通过 NFS 安装的主机文件系统下载目标模块只需要容纳文件一个复制的内存（加上少量的 header 内存开销）。

解释

基于目标机的加载器通过 VxWorks 文件管理的透明机制来调用设备驱动程序，这种机制会调用 open，close 和 ioctl。如果使用基于目标机的加载器通过网络下载模块（与从基于目标机系统磁盘上下载相反）下载目标模块所需内存取决于以怎样的方式访问远程文件系统。这是因为根据用于下载的设备调用会进行不同的操作。

对于某些设备，I/O 库会在目标机内存中对文件进行复制。如果要在使用这样驱动程序的设备上下载文件，就需要足够的内存来同时存放文件的两个复制。首先，整个文件会在打开时复制到本地内存的缓存区中，并且文件被链接到 VxWorks 时又会被复制到内存中。然后，这个复制会执行 seek 和 read 操作。因此，使用这些驱动程序会需要足够的内存来容纳下载文件的两个复制，同时需要少量内存用于加载操作或 header 内存开销。

3. “重定位不当”错误信息

当下载时，会出现以下类型的出错信息：

"Relocation does not fit in 26 bits."

出错信息中出现的实际数字取决于结构（26 或 23 等）。这种错误意味着什么？我该怎样做？

解决方法

使用 Diab 编译器的-Xcode-absolute-far 和 GNU 编译器适当的“长调用” 选项 -m*longCallOption* 来重新编译目标文件。

解释

一些结构包含少于 32 位、用来调用内存中邻近位置的指令。使用这些指令比经常用 32 位指令调用内存中的邻近位置更为有效。

当函数在内存中的位置超出了少于 32 位的指令所能达到的范围时，编译器在调用这些函数时会出现问题。例如，如果这样的指令调用 printf，只有当目标代码位于内核代码附近时，下载操作才会成功，否则将无法下载目标代码。

4. 符号缺失

从主机上下载的代码模块中的符号不会出现在目标机 shell 中，反之亦然。主机 shell 生成的符号对于目标机 shell 是不可见的，一样的，目标机 shell 生成的符号对于主机 shell 也是不可见的。这种情况为什么发生？应该怎样做使它们显示出来？

解决办法

查看目标机服务器是否与主机符号同步化，这种符号同步化是否编译进入镜像。详细内容请参见“6.4.5 基于主机和目标机的符号表同步化”。

解释

同步化机制可以在主机和目标机上分别激活。

5. 加载器使用的内存过多

包含目标机加载器会导致可用内存变少，怎样获得更多的内存？

Solution 解决方法

使用主机工具（windsh、主机加载器等）而不是目标机工具，并将所有的目标机工具从 VxWorks 镜像中删除。详细内容请参见《Tornado 用户指南》。

解释

包含目标机加载器的同时也会包含系统符号表。这种符号表中包含编译过的 VxWorks 镜像中每一个全局符号的 name，address 和 type。

使用基于目标机的加载器会占用应用程序的内存，这会对基于目标机加载器所需的符号表造成很大影响。

6. 符号表不可用

系统符号表无法下载到目标机上。如果不能按名称调用函数，怎样才能使用目标机 shell 解决这个问题？

解决方法

使用函数和数据的地址而不是它们的符号名。这些地址可以使用 nm*arch* 函数从主机上的 VxWorks 镜像中获得。

以下是一个 Unix 主机的例子：

> nm*arch* vxWorks | grep memShow

0018b1e8 T memShow

0018b1ac T memShowInit

使用这个信息通过地址从目标机 shell 中调用这个函数。（当通过地址调用时圆括号是必需的）

```
-> 0x0018b1e8 ( )
 status     bytes      blocks     avg block    max block
------ --------- -------- ---------- ----------
current
 free   14973336    20     748666     12658120
 alloc  14201864    16163      878       —
cumulative
 alloc  21197888   142523      148       —
 value = 0 = 0x0
```

第 7 章　C++语言开发

7.1　简　　介

本章提供了 C++语言开发信息，这些信息适用于使用了 Wind River 公司 GNU 和 Diab 工具的 VxWorks 操作系统。

警告： GNU C++和 Diab C++的二进制文件不兼容。对于使用 Tornado 工具的 C++文献，请参考介绍工具的章节（本手册或《Tornado 用户指南》）。

7.2　在 VxWorks 系统下使用 C++语言

警告： 需要使用 VX_FP_TASK 选项发起使用 C++语言的 VxWorks 任务。不使用 VX_FP_TASK 选项可能导致调试困难，或在运行时出现浮点寄存器的意外毁坏。默认情况下，从类似于 Wind 命令解释器（Shell）的 Tornado 工具和调试装置等发起的任务将自动激活 VX_FP_TASK 选项。

7.2.1　实现 C++语言访问的 C 语言代码

若要从用户的 C 语言代码中引入一个 C++语言符号（非过载的全局符号），必须使用 extern "C"把该符号和 C 语言代码原型进行链接。

```
#ifdef __cplusplus
extern "C" void myEntryPoint ( );
#else
void myEntryPoint ( );
#endif
```

另外，可以使用 C++语言代码能够访问 C 语言符号的语法。在 VxWorks 操作系统中，C 语言符号可自动实现 C++访问，这是因为 VxWorks 操作系统的头文件中使用了这种声明机制。

7.2.2 加入支持组件

VxWorks 操作系统默认情况下内核仅支持最低限度的 C++组件。为增强 C++功能，可加入下面的一个或多个组件：

1. 基本支持组件

- INCLUDE_CTORS_DTORS：（默认地包含于内核中）为了确保在内核启动时能调用编译器生成的初始化函数，包括 C++静态对象的初始化程序。
- INCLUDE_CPLUS：包括 C++应用的基本支持。常与 INCLUDE_CPLUS_LANG 同时使用。
- INCLUDE_CPLUS_LANG：包括 C++语言特征支持，例如建立、删除和异常处理。

2. C++库组件

下面列出了可以使用的 C++语言库组件。

（1）GNU-标准库支持的组件

对于 GUN 编译器，包括如下组件：

- INCLUDE_CPLUS

包含在 VxWorks 操作系统中运行 C++语言的所有基本要求。该组件能够进行下载和运行编译以及细化 C++的模式。

- INCLUDE_CPLUS_STL

包含标准模板库的基本组件。

- INCLUDE_CPLUS_STRING

包含字符串类型库的基本组件。

- INCLUDE_CPLUS_IOSTREAMS

包含输入/输出数据流库的基本组件。

- INCLUDE_CPLUS_COMPLEX

包含复杂类型库的基本组件。

- INCLUDE_CPLUS_IOSTREAMS_FULL

包含所有数据流库，该组件必须并自动地包括组件——INCLUDE_CPLUS_IOSTREAMS。

- INCLUDE_CPLUS_STRING_IO

包含字符串 I/O 功能，该组件必须并自动地包括组件——INCLUDE_CPLUS_STRING 和 INCLUDE_CPLUS_IOSTREAMS。

- INCLUDE_CPLUS_COMPLEX_IO

包含有复杂数字对象输入/输出功能的基本组件；该组件必须并自动地包括组件——INCLUDE_CPLUS_IOSTREAMS 和 INCLUDE_CPLUS_COMPLEX。

（2）Diab-标准库支持的组件

对于 Diab 编译器，所有 C++库函数功能都被压缩到单个组件——INCLUDE_CPLUS_IOSTREAMS 中，它包含所有库功能。

配置 VxWorks 操作系统时需要包括或排除的组件的详细信息，请参考《Tornado 用户指南》：项目工程。

7.2.3　C++组合器

使用目标机命令解释器（shell）登录时，需要包括组件 INCLUDE_CPLUS_DEMANGLER。添加组合器（demangler）允许目标机命令解释器（shell）的符号表返回用户可阅读形式的 C++语言符号名。若使用主机上的 Tornado 工具，可以不包括组合器。配置组合器的详细信息，请参考“6.2.8 分配 Demangler”。

> ⚠ **注意**：若要使用目标机命令解释器和 C++语言，但不需要包括组合器，可以在 config.h 中加入 INCLUDE_NO_CPLUS_DEMANGLER 组件，并用命令行形式编译 BSP 程序。使用工程方式进行编译，可以简单地从工程配置中删除“C++符号组合器”。

7.3　初始化和确定静态目标

本节讨论了使用静态 C++对象编程的问题。这些问题包括一个附加的主机处理步骤，及调用静态构造体和析构体的特殊策略。

7.3.1　细化（munch）C++应用模块

在将 C++模块下载到 VxWorks 目标机之前，必须在主机上经过附加的处理步骤，该步骤通常称为细化，细化执行的任务如下：

- 初始化静态对象。
- 确保 C++程序按正确顺序运行，对所有静态对象调用合适的构造体和析构体。
- （仅对于 GNU/ELF/DWARF）把由-fmerge-templates 生成的任何“linkonce”部分分解为程序段和数据段（请参考-fmerge-templates）。

细化应在程序编译之后且下载之前执行。

下面的例子使用三种工具编译了一个 C++应用源文件——hello.cpp；对.o 文件执行细化操作，编译生成的 ctdt.c 文件，并将应用程序与 ctdt.o 文件链接生成一个可下载的模块——hello.out。

1. 使用 GNU 工具

下列代码（包含命令和注释）使用 GNU 工具对 hello.cpp 文件执行上述操作。

```
# Compile
ccppc -mcpu=604 -mstrict-align  -O2 -fno-builtin -InstallDir/target/h \
    -DCPU=PPC604 -DTOOL_FAMILY=gnu -DTOOL=gnu -c hello.cpp

# Run munch
nmppc hello.o | wtxtcl installDir/host/src/hutils/munch.tcl \
    -c ppc > ctdt.c

# Compile ctdt.c file generated by munch
ccppc -mcpu=604 -mstrict-align  -fdollars-in-identifiers -O2 \
-fno-builtin -IinstallDir/target/h \
    -DCPU=PPC604 -DTOOL_FAMILY=gnu -DTOOL=gnu  -c ctdt.c

# Link hello.o with ctdt.o to give a downloadable module (hello.out)
ccppc  -r  -nostdlib  -Wl,-X  -T  installDir/target/h/tool/gnu/ldscripts/
link.OUT \
-o hello.out hello.o ctdt.o
```

⚠ **注意**：-T .../link.OUT 选项分解了输入文件中任何“linkonce”部分（详细信息请参考-fmerge-templates）。它不应该在 68k 系列、VxSim Solaris 或 VxSim PC 上使用。但写成如下形式：cc68k -r -nostdlib -Wl，-X -o hello.out hello.o ctdt.o。

2. 使用 Diab

下列代码（包含命令和注释）使用 Diab 工具对 hello.cpp 文件执行上述操作。

```
# Compile
dcc -tMCF5307FS:vxworks55 -W:c:,-Xmismatch-warning=2 \
-ew1554,1551,1552,1086,1047,1547 -Xclib-optim-off -Xansi \
-Xstrings-in-text=0 -Wa,-Xsemi-is-newline -ei1516,1643,1604 \
-Xlocal-data-area-static-only -ew1554 -XO -Xsize-opt -InstallDir/target/h \
```

```
    -DCPU=MCF5200 -DTOOL_FAMILY=diab -DTOOL=diab -c hello.cpp

# Run munch
nmcf hello.o | wtxtcl <italic>installDir</italic>/host/src/hutils/munch.tcl \
-c cf > ctdt.c

# Compile ctdt.c file generated by munch
dcc -tMCF5307FS:vxworks55 -Xdollar-in-ident -XO -Xsize-opt -Xlint \
-I<italic>installDir</italic>/target/h \
    -DCPU=MCF5200 -DTOOL_FAMILY=diab -DTOOL=diab -c ctdt.c

# Link hello.o with ctdt.o to give a downloadable module (hello.out)
dld -tMCF5307FS:vxworks55 -X -r -r4 -o hello.out hello.o ctdt.o
```

3. 使用通用规则

若使用 VxWorks 操作系统中的 Makefile 定义，可制定一个简单的细化规则（对 CPU 和 TOOL 宏编写合适的定义），此规则适用于支持 GUN 和 Diab 工具的所有体系结构。

```
CPU     = PPC604
TOOL    = gnu

TGT_DIR = $(WIND_BASE)/target

include $(TGT_DIR)/h/make/defs.bsp

default : hello.out
%.o : %.cpp
    $(CXX) $(C++FLAGS) -c $<

%.out : %.o
    $(NM) $*.o | $(MUNCH) > ctdt.c
    $(CC) $(CFLAGS) $(OPTION_DOLLAR_SYMBOLS) -c ctdt.c
    $(LD_PARTIAL) $(LD_PARTIAL_LAST_FLAGS) -o $@ $*.o ctdt.o
```

在细化、下载及链接后，将调用静态构造体和析构体。该步骤将在下节叙述。

7.3.2　交互式调用静态构造体和析构体

VxWorks 操作系统提供了两个调用静态构造体和析构体的策略，描述如下：

■ 自动策略

这是一个默认策略。静态构造体在模块下载到目标对象之后，以及模块登录程序返回给调用程序之前执行。静态析构体仅在模块卸载前执行。

■ 手动策略

在下载模块之后、运行应用程序之前，需用户手动调用静态构造体；在任务完成运行之后、卸载模块之前，需用户手动调用静态析构体。通过 cplusCtors( )调用静态构造体；通过 cplusDtors( )调用静态析构体。这些函数以模块为参数，因此，静态构造体和析构体是模块和模块间的调用。此外可以对所有当前登录的静态构造体或析构体进行无参数调用。

警告：使用手动策略时，每个模块的构造体仅运行一次。

在调用构造体和析构体时，使用 cplusXtorSet( )将改变调用策略。调用 cplusStratShow( )汇报当前策略信息。

本节涉及到的函数详细信息，请参考联机参考手册中应用编程接口的相关条目。

7.4 使用 GNU C++编程

由 Tornado IDE 提供的 GNU 编译器支持 ANSI C++标准中大部分语言特征。尤其是，GNU 编译器支持模板实例化、异常处理、运行类型信息和命名空间等功能。关于 GUN 工具支持的功能，将在下面的章节里详细讨论。

GUN 编译器及相关工具的完整文档，请参考《GNU 工具箱用户指南》。

7.4.1 模板实例化

GNU C++工具支持三种不同的模板实例化策略，描述如下：

1. -fimplicit-templates

这是一种隐性实例化策略选项。使用该策略，模板代码在每个使用的模块里显示出来，模块使用时，模板上的工作代码必须是可用的。也就是使用模板功能体，以及在头文件里对代码进行声明。隐性实例化的优点在于其简单性，以及默认在 VxWorks 操作系统的 makefiles 中使用；缺点在于它可能导致代码重复，并使应用程序很大。

2. -fmerge-templates

除了模板实例化和内联（inline）函数的非线性复制被放入特殊“linkonce”部分以外，这种模块与-fimplicit-templates 类似。链接器将重复的部分合并，所以每个实例化模块在输

出文件中仅出现一次，从而-fmerge-templates 模板避免了-fimplicit-templates 模板的代码膨胀问题。

⚠ **注意**：只有 ELF/DWARF 目标对象支持本标志的功能，该标志可以被 68k 和模拟器编译器（cc68k，ccsimso 以及 ccsimpc）接受，但不起任何作用。

警告：VxWorks 动态登录程序不直接支持“linkonce”部分。但“linkonce”部分在登录前必须被并入，而且要转化成标准的文本和数据形式。这是通过“7.3.1 细化 C++应用模块”中描述的一个特殊的链接步骤实现的。

3. -fno-implicit-templates

这是一种显性实例化策略选项。使用该策略要明确对任何需要的模板进行实例化。显性实例化的优点是基本上控制需要实例化的模板部分，避免代码膨胀；其缺点是要明确对每一个模块进行实例化。

4. -frepo

这是第三种实例化模块策略选项。该策略使用时需对每个模块操纵模板实例化数据库。编译器为每个相应目标文件生成一个.rpo 文件，该文件提供了所有能够被使用和被演示的模板实例化对象。然后，链接封装器 collect2 更新.rpo 文件，通知编译器如何安排放置这些实例化对象，并重建一个有效的目标文件。第一步完成链接时间的花费可以忽略不计，这是因为编译器继续在相同文件中放置模板实例化对象。该策略的优点在于它结合了隐性实例化的简单性和手动实例化模板时较小的继承性。

（1）程序

模板头文件必须包含模板体，如果模板体当前存储在.cpp 文件中，#include theTemplate.cpp 语句必须加入到 theTemplate.h 文件中。

使用-frepo 选项建立一个完整文件时需要创建.rpo 文件，该文件告诉编译器哪个模板需要实例化。应使用 ccarch，而不使用 ldarch 来调用链接的步骤。

随后，单个模块能够与正常时一样编译（但需使用-frepo 选项，并且无其他的模块标志）。

当有新模块实例化时，与项目相关部分必须重建，并更新为.rpo 文件。

（2）下载顺序

在模块使用下载符号前，Tornado 工具的动态链接需要模块包含一个下载符号的定义。使用-frepo 选项可能不清楚那个模块包含该定义。因此应提前链接并下载到链接对象中。

（3）实例

本例使用了一个标准 VxWorks 操作系统 BSP 生成文件（makefile），使用了 6
标机。

例 7-1：生成文件实例

```
make PairA.o PairB.o ADDED_C++FLAGS=-frepo
/* 虚拟链接到实例化模板*/
cc68k -r -o Pair PairA.o PairB.o

//Pair.h

template <class T> class Pair
{
public:
    Pair (T _x, T _y);
    T Sum ( );

protected:
    T x, y;
};

template <class T>
Pair<T>::Pair (T _x, T _y) : x (_x), y(_y)
{
}

template <class T>
T Pair<T>::Sum ( )
{
    return x + y;
}

// PairA.cpp
#include "Pair.h"

int Add (int x, int y)
{
    Pair <int> Two (x, y);
    return Two.Sum ( );
}
```

```
// PairB.cpp
#include "Pair.h"

int Double (int x)
{
    Pair <int> Two (x, x);
    return Two.Sum ( );
}
```

7.4.2　异常处理

GNU C++编译器默认支持多线程的安全异常处理。使用编译器标志-fno-exceptions 关闭异常处理的支持。

本节讨论下面有关异常处理内容：

- 用于异常预防模块（pre-exception）的代码的影响
- 使用异常处理的开销
- 如何安装用户的终端处理程序

1. 使用异常预防模块

根据 C++编译的异常预防模式可以进行代码编写。例如，调用 new 可以检查返回失败值为零的指针。若在提高异常处理能力的版本中不能正确编译代码时，请遵循下列简单原则进行检查：

- 使用 new（nothrow）命令。
- 在 iostreams 对象不要明确打开异常功能。
- 不要使用字符串或把它们包含在“try { } catch(...){ }”模块中。
- 这些原则由下面的规则产生。
- 在库建立以及特殊 iostreams 对象中启用异常功能时，除非定义了 IO_THROW，否则 GNU 输入/输出流不使用 throw。默认情况下认为无异常。异常必须明确地启用每个需要抛掷的 iostate 标志。
- 在 basic_string 类（这种“string”是一种专用字符串）的某些使用方法中，标准模板库（STL）不产生异常。

2. 异常处理开销

异常处理产生高额开销。在堆栈释放时为支持自动对象析构，例如对于建立自动对象（在堆栈基础上）和析构体的函数，编译程序必须插入附加代码。

以附加代码为例，下面阐述了在 PowerPC 604 目标机上（mv2604 BSP）如何使用异常处理。在执行指令时采取了计数方法。下面是需要加入到异常处理中的指令：

- 为执行“throw 1”和相连的“catch(...)”，需要 1235 条指令
- 为使用析构体对自动变量和临时对象寄存或解除寄存，需要执行 14 条指令
- 对于每个使用异常处理的非内联函数，建立异常处理需要执行 29 条指令
- 在遇到第一个异常处理构造体时（try，catch，throw 或寄存自动变量，以及寄存临时变量），执行 947 条指令

实例代码如下：

```
                          first time   normal case
void test( )  {           // 3+29          3+29
    throw 1;              // 1235          1235     total time to printf
}

void doit( ) {            // 3+29+947      3+29
    try {                 // 22            22
       test( );               // 1              1
    } catch (...) {
       printf("Hi\n");
    }
}

struct A { ~A( ) { } };

void local_var ( ) {      //               3+29
    A a;                  //               14
}                         //               4
```

使用-fno-exceptions 选项关闭异常处理功能。若关闭该功能，将删除所有异常处理开销。

3. 未能处理（unhandled）异常

根据 ANSI C++标准要求，未能处理异常最终将调用 terminate( )。该函数默认地挂起异常任务，并给操作平台发送一个警告消息。调用定义在头文件 exception 中 set_terminate( )，用户可以安装自己的终止处理的程序。

7.4.3 Run-Time 类型信息

GNU C++编译器支持 Run-time 类型信息（RTTI）特征。默认情况将启用该特征，给含有类和虚拟函数的 C++程序加入一小部分开销。如果不使用该特征，可以调用-fno-rtti 关闭。

7.4.4　命名空间（Namespaces）

GNU C++编译器支持命名空间。根据 C++标准用户可以为自己的代码使用命名空间。

C++标准最新版本根据系统头文件定义了名字，这些名字位于一个称为 std 的“命名空间”中。该标准要求在标准头文件中指明使用哪些名字。对于 std 命名空间，GNU C++编译器可以接收新格式，但不使用它们。这是因为 GNU C++把 std 命名空间置成全局命名空间。

这意味着当前的 GNU C++编译器是过渡性的，因为它根据标准编译继承的代码和根据标准编译新的代码。注意，在新语法标准下编写代码时，std 命名空间中的标志为全局性的，因此它们在全局命名空间中必须惟一。

例如，最新标准中下列代码在技术上无效，但在通用 GNU C++编译器下仍可进行编译：

```
#include <iostream.h>
int main( )
    {
        cout << "Hello, world!" << endl;
    }
```

下列三个实例显示了当前 C++标准如何表达该代码，这些实例也可以在通用 GUN C++编译器下编译：

// 例 1

```
#include <iostream>
int main( )
    {
        std::cout << "Hello, world!" << std::endl;
    }
```

// 例 2

```
#include <iostream>
using std::cout;
using std::endl;
int main( )
    {
        cout << "Hello, world!" << endl;
    }
```

```
// 例 3

    #include <iostream>
    using namespace std;
    int main( )
        {
            cout << "Hello, world!" << endl;
        }
```

注意，GNU C++可以使用 using 指令，但并不是必需的。这一点也适用于标准 C 代码。例如，下列两个代码实例可以通过编译：

```
#include <stdio.h>

void main( )
    {
        int i = 10;
        printf("%d", &i");
    }
```

或者

```
#include <cstdio>

void main( )
    {
        int i = 10;
        std::printf("%d", &i");
    }
```

7.5 使用 Diab C++编程

由 Tornado IDE 提供的 Diab C++编译器使用 Edison 设计集团（EDG）的 C++语言；Diab C++编译器全面支持 ANSI C++标准，除了在“7.5.1 模板实例化”中涉及到的限制。

下节简要地描述 Diab 编译器支持的模块实例化、异常处理和 run-time 类型信息。对于 Diab 编译器和相关工具的完整文档，请参考《Diab C/C++编译器用户指南》。

7.5.1　模板实例化

根据标准，可执行除了下列情况以外的函数和类模板。

- 模板不执行 export 关键字。
- 一个类成员模板的局部专用程序不能加到类定义的外部。

控制模板实例化有两种途径。默认情况为模板隐性实例化，即无论何时使用模板都通过编译器进行实例化。若要求更好的控制模板实例化时，-Ximplicit-templates-off 选项通知编译器仅对源代码里明确调用的部分进行模板实例化，例如：

```
template class <int>;    // Instantiate A<int> and all member functions.
template int f1(int);     // Instantiate function int f1(int).
```

Diab C++选项在下面总结了多种模板实例化的控制。这些选项的详细信息，请参考《Diab C/C++编译器用户指南》。

1. -Ximplicit-templates

默认实例化每处使用的模板。

2. -Ximplicit-templates-off

仅在代码中明确要求实例化时对模板进行实例化。

3. -Xcomdat

模板隐性实例化时，把每个生成的代码或数据部分标注为 comdat。链接器把标注为 comdat 的相同实例分解成内存中单一的实例。

VxWorks 不能下载包含 comdat 的目标文件，但可以在 Diab 链接器中使用-Xcomdat-on，然后下载可执行文件。

4. -Xcomdat-off

最终目标文件中静态实体是由模块实例化和内联（inline）函数生成的，这些静态实体能导致静态成员函数和类变量多种实例化。该选项是默认使用的。

7.5.2　异常处理

Diab C++编译器默认支持线程的安全异常处理。为取消异常处理支持，可使用编译器标志-Xexceptions-off。

Diab 异常处理模式是表驱动的，若没有抛掷某个特定异常，系统将花费少量的运行开

销，但异常处理确实增加了开销。

7.5.3 Run-Time 类型信息

Diab C++编译器支持 C++ Run-time 类型信息（RTTI）。默认情况将启用这种语言特征，使用编译器标志-Xrtti-off 将终止使用该特性。

7.6 使用 C++库

VxWorks 组件中提供了 iostream 类库、字符串和复杂数字类，并且可使用任一种工具来配置。这些库描述如下：

1. 字符串及复杂数字类

这些类是新标准 C++类库的一部分。对于 Diab 工具，在头文件中完整地提供了这些类。对于 GUN 工具，可以使用 INCLUDE_CPLUS_STRING 和 INCLUDE_CPLUS_COMPLEX 配置 VxWorks 操作系统。用户可以有选择性地使用 INCLUDE_CPLUS_STRING_IO 和 INCLUDE_CPLUS_COMPLEX_IO，为这些类包含 I/O 功能。

2. iostream 类库

使用 INCLUDE_CPLUS_IOSTREAMS 可以给 VxWorks 操作系统配置 iostream 类库。

对于 GNU 工具，iostream 类库头文件位于“installDir/host/hostType/include/g++-3”目录下。对于 Diab 工具，iostream 类库头文件位于“installDir /host/diab/include/cpp”目录下。

警告： 每个编译器自动为这些库包括含有头文件的目录。使用时不要求给编译器加入这些目录，包括路径。如果系统对丢失的头文件进行警告，那么可能是由于使用了-nostdinc 标志；该标志不能用于目前的 Tornado 版本。

为了使用该库，在应用程序的某个模块的 vxWorks.h 头文件后应包含一个或多个头文件。最频繁使用的头文件是 iostream.h，但其他的头文件也可使用，请参考类似于 Stroustrup 的 C++参考书。

标准 iostream 对象（cin，cout，cerr 和 clog）是全局性的，即对于任何特定任务都不是私有的。无需考虑使用它们的任务或模块数量便能够正确地初始化；它们能够安全地用于多种任务，只要这些任务有相同的 stdin，stdout 和 stderr 定义。但当不同的任务有不同标准的 I/O 文件描述符时，不能够安全地使用；在这种情况下，应根据应用情况使用互斥

操作。

私有标准的 iostream 对象的影响可以进行模拟：建立一个属于相同类的新的 iostream 对象作为标准 iostream 对象，（例如 cin 作为 istream_withassign）；并给它分配一个与合适文件描述符紧密联系的新的 filebuf 对象。新 filebuf 和 iostream 对象对于调用任务都是私有性的，其目的就是确保没有其他任务能够意外地破坏它们。

```
ostream my_out (new filebuf (1));          /* 1 == STDOUT */
istream my_in (new filebuf (0), &my_out); /* 0 == STDIN;
                                            * TIE to my_out */
```

对于 iostream 类库的详细信息，请参考《GNU 工具箱用户指南》。

3. 标准模板库（STL）

对于这两种工具，标准模板库完全由整套头文件组成。在 VxWorks 操作系统中不要求包含特殊的 run-time 组件。

（1）GNU 工具的标准模板库

用于 VxWorks 系统的 GNU STL 端口在类级别上是线程安全的，即若有两个任务需要使用同一个集装箱（container）对象（信号量可用作这种对象，请参考“2.3.3 信号量”），用户必须提供直接上锁机制。但是对于属于相同 STL 集装箱类的不同对象可以并行访问。

异常处理程序关闭时，客户编译的代码中可使用 C++标准模块库。含有如下含义：

- 对于所有调用者进行地合理检查，例如界限（bounds）检查，对已抛掷的异常将不采取任何处理措施。当启用优化操作时，相当于移除了检查操作。
- 对于已抛掷 bad_alloc 异常情况的内存耗尽处，将记录下列消息（若包括记录功能）。

```
"STL Memory allocation failed and xceptions disabled -calling terminate"
```

并且任务调用 terminate 操作。此行为仅适合于默认分配符；用户可以自由地为不同的行为定义用户风格的分配符。

（2）Diab 工具的标准模板库

Diab C++库完全与 ANSI C++一致，除了下列少数异常以外，即 wchar 不完全支持 Diab C++库，Diab C++库完全与 ANSI C++一致。Diab 标准模板库组件是线程安全的。

7.7　运行事例演示

Factory 实例演示了不同的 C++特征，包括标准模板库，用户定义模板 run-time 类型信息以及异常处理。该实例位于 installDir/target/src/demo/cplusplus/factory 下。

用户要创建、编译、链接及运行该演示程序，可以使用位于《Tornado 用户指南》：工程中的 Tornado 工程工具，也可使用下列显示的行命令。

对于 factory 程序，系统内核中必须包含下列组件：

INCLUDE_CPLUS

INCLUDE_CPLUS_LANG

INCLUDE_CPLUS_IOSTREAMS

另外，对于 GUN（仅 GUN），还需包含下列组件：

INCLUDE_CPLUS_STRING

INCLUDE_CPLUS_STRING_IO

用户要从 Tornado IDE 中加入组件，请参考《Tornado 用户指南》中的“工程”部分。通过命令行编译 factory 程序，仅需将 factory 源文件复制到 BSP 目录下，显示如下：

```
cd installDir/target/config/bspDir
cp installDir/target/src/demo/cplusplus/factory/factory.* .
```

然后，建立一个包含 factory 实例的启动程序，按如下显示执行：

```
make ADDED_MODULES=factory.o
```

然后启动目标机。

建立一个包含 factory 实例的可下载应用程序，按如下显示执行：

```
make factory.out
```

然后，从 WindSh 中下载 factory 模块，按如下显示执行：

```
ld < factory.out
```

最后，运行 factory 演示程序，在命令解释器（shell）上键入：

```
-> testFactory
```

所有读者希望参考的文档都包含在演示程序源代码的注释中。

第 8 章　闪存块设备驱动程序

可选 TrueFFS 组件

8.1　简　　介

支持 Tornado 的 TrueFFS 是可选产品，它提供了面向多种闪存设备的块设备接口。TrueFFS 是 M-Systems FLite 2.0 版本兼容的 VxWorks 产品。该系统可重进入（reentrant），线程安全（thread-safe），并支持所有装载 VxWorks 的 CPU 结构。

本章首先简要介绍闪存，并描述建立一个支持 TrueFFS 系统的步骤。本章的主要部分将详细介绍各个步骤，包括编写 socket 驱动程序和 MTD 组件。最后，本章将描述闪存块设备器件的功能。

⚠ **注意**：本版本 TrueFFS 产品是面向 VxWorks 的块设备驱动程序，它提供的 MS-DOS 兼容文件系统保证了与 VxWorks 的兼容性。

8.1.1　选择 TrueFFS 作为媒质

TrueFFS 应用程序能从闪存中读、写，其过程如同从磁性介质的机械硬盘驱动器上的 MS-DOS 文件系统中读写一样。然而，内在的磁性介质却大不相同。尽管这些不同对于高级开发者来说是完全透明的，但能够意识到这一点，在设计一个嵌入式系统的时候是十分重要的。

闪存能够无限地存储数据，即使掉电时这些数据也不会丢失。闪存的物理组件由一些固态器件组成[①]，这些器件只消耗很少的能量并可以多次擦写。而且，闪存非常适用于移动设备、手持设备以及高可靠、信息不易失的嵌入式系统，而这些要求对于机械硬盘显得很苛刻。

但是，闪存设备具有有限的可擦除周期，并且 TrueFFS 只支持对称分区，这使得闪存的使用寿命受到了限制。此外，闪存不具备一些对于磁介质块设备驱动器来说很普通的特

[①] 意味着它们没有移动部分。

征。读和写的时间不相等是闪存的一个典型特征，通常读比写更快一些。详细内容请参见“8.13 闪存功能”。而且，TrueFFS 不支持 ioctl。

闪存相对于磁介质硬盘驱动器的特点是它对于某些应用来说是理想的，但却不适用于其他的一些应用。

> **注意**：尽管可以在任意大小的内存中进行写操作，从字节到字或双字，但是只能在块（blocks）中进行擦除。延长闪存寿命的最好方法是尽量保证内存中所有的块的磨损程度相同。

> **注意**：TrueFFS 可选产品不支持分区表。

> **警告**：VxWorks 在处理 I/O 申请时，相对于 I/O 顺序更偏重任务优先级。所以，如果一个较低优先级的任务和一个较高优先级的任务在资源很少的时候共同申请 I/O 服务，即使较低优先级的任务先发出请求，较高优先级的任务一旦可执行将首先获得资源。对于 VxWorks 而言，闪存也是资源。

8.1.2 TrueFFS 层

TrueFFS 由一个核心层和三个功能层——转换层、存储技术驱动程序（MTD）层和 socket 层——组成（如图 8-1 所示），三个功能层具有源代码形式或二进制形式，或者同时具有两种形式。有关这些层功能的详细信息请参见“8.13 闪存功能”。

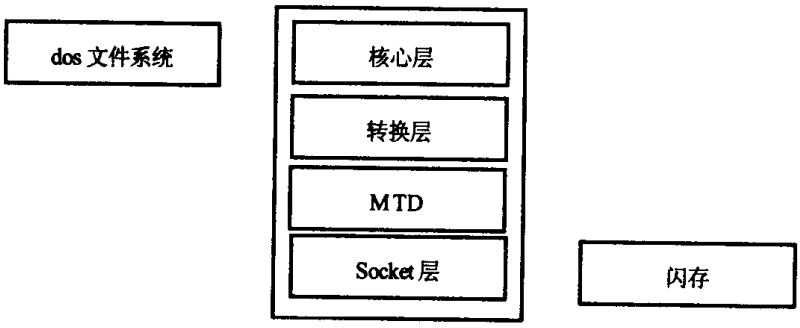


图 8-1　TrueFFS 具有分层结构

■　核心层

该层将其他各层连接起来，并将工作引向其他层并处理全局事务，如后台处理（backgrounding）、碎片收集、计时器和其他系统资源。核心层只具有二进制形式。

■　转换层

该层包含了存储媒质的文件系统视图与闪存擦除区之间的映射。块分配映射是完成磨损测平和错误检测功能的基本构件。转换层具有特殊媒质（NOR or SSFDC）。转换层只具有二进制形式。

■　MTD 层

MTD 执行闪存媒质的低级编程（映射、读、写和擦除），MTD 具有源代码和二进制形式。

■　Socket 层

Socket 层提供 TrueFFS 与板级硬件之间的接口，提供指定板硬件的存取程序。进行电源管理、卡片识别、窗口管理和 socket 注册。Socket 驱动程序只具有源代码形式。

8.2　构建支持 TrueFFS 的系统

本节将概述开发过程，并描述怎样配置和构建一个支持 TrueFFS 的 VxWorks 系统。配置和构建过程需要 VxWorks 具有可引导和可下载的应用程序。

步骤 1：选择一个 MTD 组件

从 TrueFFS 产品提供的 MTD 中选择适合硬件的一款，也可以自己编写一个 MTD。详细内容请参见“8.3 选择 MTD 组件”。

步骤 2：确定 Socket 驱动程序

确定现在的 socket 驱动程序是目前流行的版本。socket 驱动程序是源代码组件，在 sysTffs.c.文件中执行。对于一些 BSP，socket 驱动程序需要在 BSP 目录下完全定义。否则，就需要将一个包含骨架代码的类属（generic）文件传送到硬件中。详细内容请参见“8.4 确定 socket 驱动程序”。

步骤 3：配置系统

添加适当的组件来配置系统，使其支持 TrueFFS。至少，也要添加支持 dosFs 和四个 TrueFFS 层的组件。详细内容请参见“8.5 配置和建立项目”。

⚠ **注意**：组件的描述中使用了缩写 TFFS 而不是 TRUEFFS。

步骤 4：构建项目

在建立系统之前，需要更新支持 MTD 的二进制数据，在 BSP 目录下的 Socket 驱动程序文件必须是目前流行的版本。详细内容请参见“8.5.7 建立系统项目”。

步骤 5：引导目标机以及格式化驱动程序

先启动目标机，然后从 shell 中格式化驱动程序。详细内容请参见“8.6 设备格式化”。

⚠ **注意**：在闪存中保留一个区域来存储启动代码，请参见“8.7 创建用于编写启动镜像的区域”。

步骤 6：安装驱动程序

在 TrueFFS 闪存驱动程序上安装 VxWorksDOS 文件系统。详细内容请参见“8.8 安装驱动器”。

步骤 7：检测驱动程序

检测驱动程序。

一种方法是将一个文本文件从主机（或从其他存储介质中）复制到目标机的闪存文件系统中，然后将文件复制至控制器或一个临时文件中，进行对比并确定其内容。

```
%->@copy "host:/home/panloki/.cshrc" "/flashDrive0/myCshrc"
Copy Ok: 4266 bytes copied
Value = 0 = 0x0
%->@copy "/flashDrive0/myCshrc"
...
...
...
Copy Ok: 4266 bytes copied
Value = 0 = 0x0
```

⚠ **注意**：复制命令需要 dosFs 支持的组件进行适当配置。详细内容请参见“可选 dosFs 组件”。

8.3 选择 MTD 组件

installDir/target/src/drv/tffs 目录中包含有以下 MTD 组件的源代码：

- 与 Intel，AMD，Fujitsu 以及 Sharp 提供的设备兼容的 MTD;
- 适用于遵守 CFI 的设备的两个类属 MTD。

为了更好地支持 out-of-box 经验，这些 MTD 应适用于更多的设备和总线结构。结果这

些驱动程序与那些经过特别编写，从而为指定运行环境编址的驱动程序相比，体积更大，速度更慢。如果驱动程序的性能和大小不能符合要求，可以进行适当的修改来满足需要。“8.12 编写 MTD 组件”一节将特别讨论这个问题。

MTD 的完整列表，请参见“8.11 使用 MTD 支持的闪存设备”，此外，“8.11.1 支持常用闪存接口（CFI）”中详细描述了 CFI-MTD，并评价了这些驱动程序是否与所使用支持 TrueFFS 的设备相匹配。通常可以通过他们的 JEDEC 名称识别不同的设备。只要适用于闪存设备，就可以使用这个 MTD。这些驱动程序具有二进制形式，所以除非已经对代码进行修改，否则不必对 MTD 源代码进行编译。

⚠ **注意**：有关 MTD 列表以及将该 MTD 组件加到系统项目中的详细内容请参见“8.5.4 包含 MTD 组件”。

8.4　确定 Socket 驱动程序

包含在系统中的 Socket 驱动程序必须在 bsp 中使用。某些 BSP 包含 Socket 驱动程序，而其他的 BSP 则不是这样。Socket 驱动程序文件是 sysTffs.c，而且如果存在，它应该位于 bsp 目录中。如果 bsp 提供一个 Socket 驱动程序，用户可以使用它。

如果 bsp 不提供这个文件，请参考“8.10 编写 Socket 驱动程序”一节所描述的过程，本节解释了怎样将一个存根版本（stub version）传送到硬件中。

另一种情况下，建立过程需要 bsp 目录中有一个目前适用的 Socket 驱动程序（sysTffs.c）。有关详细信息请参见“8.5.6 加入 Socket 驱动程序”。

8.5　配置和建立项目

用 TrueFFS 配置的 VxWorks 系统包括：

- 完全支持 dosFs 文件系统的配置；
- 一个核心 TrueFFS 组件，INCLUDE_TFFS；
- 三个 TrueFFS 层中的每一个至少有一个软件模块。

既可以从命令行也可以使用 Tornado IDE 项目设备来配置和建立系统。当选择其中一种方式来配置、建立支持 TrueFFS 的系统时，请考虑以下标准：

- 从命令行配置和建立需要编辑包含组件列表以及初始化参数的文本文件，并需要调用 make 程序建立一个系统镜像。这个过程尽管比使用项目设备配置系统要快，但需要提供相关组件的详细内容。

- 通过 Tornado 项目设备来配置和建立系统则提供了一种简单、准确的方法加入所需的组件，尽管这种建立过程与使用命令行相比要花费更多的时间。

无论使用那种配置、建立方式，都要考虑到 Socket 驱动程序或 MTD 甚至两者都没有被提供的情况。驱动程序需要注册，MTD 也需要适当的组件描述。有关详细内容请参见"8.10 编写 Socket 驱动程序"以及"8.12.4 定义 MTD 为组件"。

有关配置过程的详细内容请参见《Tornado 用户指南》：配置和建立，以及《Tornado 用户指南》：项目。

⚠ **注意**：由于 MTDs 和 Socket 驱动程序，支持 Tornado 的 TrueFFS 向系统中加入了新的资源。MTD 在 target/src/drv/tffs 中，Socket 驱动程序则是在每个支持 TrueFFS 的 BSP 中 target/config/bspname 目录下的 sysTffs.c 文件中定义的。

8.5.1 包含文件系统组件

如果没有 VxWorks 兼容文件系统 MS-DOS，TrueFFS 的系统配置是没有意义的。因此 dosFs 的支持以及所有相关的组件都要被包含在 TrueFFS 的系统中。有关这个文件系统和支持组件的信息，请参见"5.2.2 配置用户系统"。

此外，有一些其他的文件系统组件不是必需的，但在某些情况下仍然很有用。这些组件增加了对文件系统的一些基本功能（如 ls，cd，copy 等命令）的支持。

8.5.2 包含核心组件

所有的系统都必须包含 TrueFFS 核心组件 INCLUDE_TFFS。

在启动时间（boot time），为了初始化该产品，定义该组件能保证初始化过程的正确顺序。也能保证 Socket 驱动程序被包含在项目中。（见"8.5.6 加入 Socket 驱动程序"）

可以使用项目设备来包含这个组件，如图 8-2 所示或是在 config.h 中定义 INCLUDE_TFFS。

8.5.3 包含应用程序组件

本节描述了 TrueFFS 应用程序组件、它们的目的以及默认配置选项。创建一个高效的 TrueFFS 系统并不需要改变这些组件的默认配置。图 8-2 所示为从项目设备加入 TrueFFS 组件。

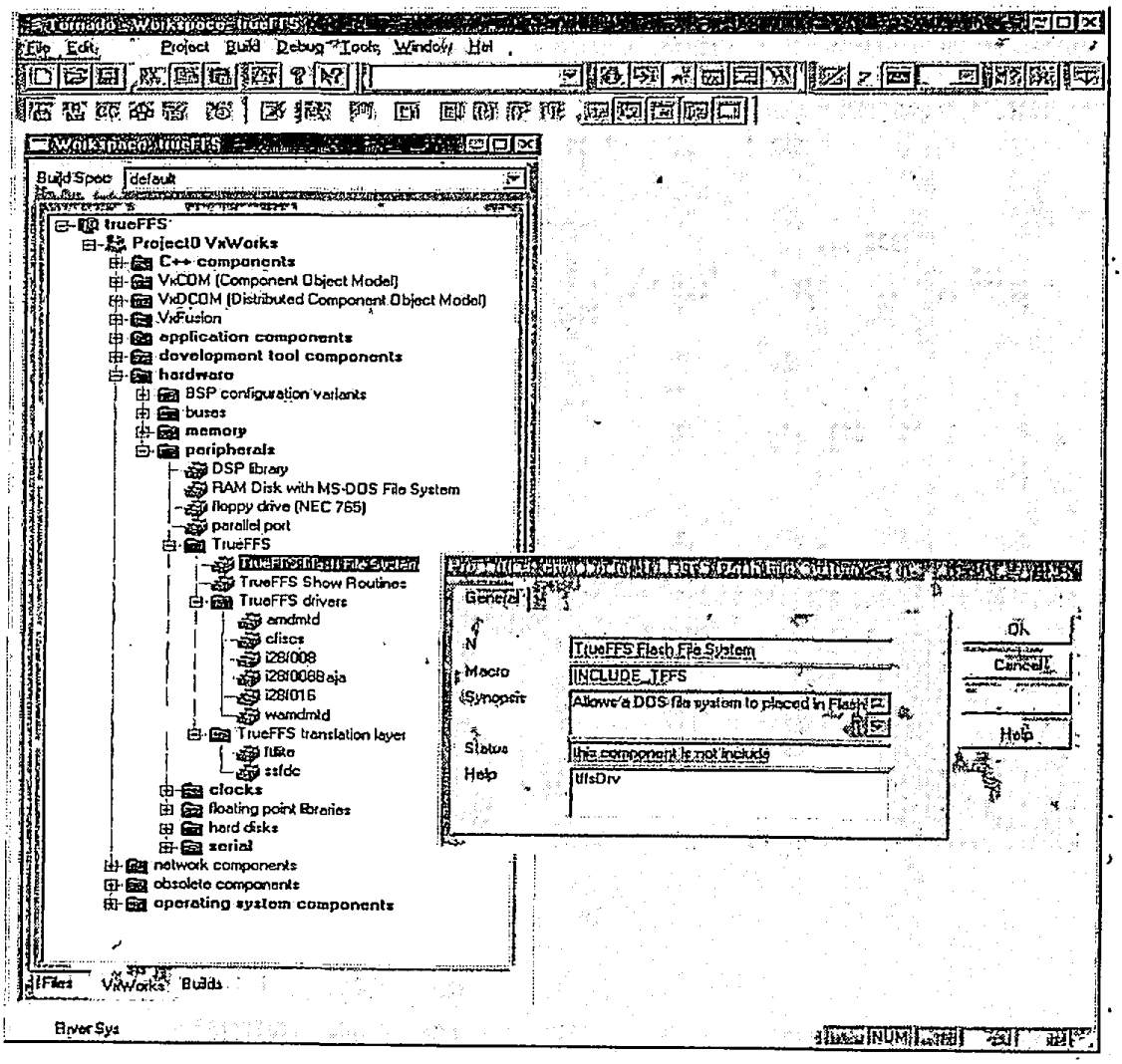


图 8-2　从项目设备中加入 TrueFFS 组件

INCLUDE_TFFS_SHOW

包含这个组件会加入两个配置显示程序 tffsShow( )和 tffsShowAll( )。tffsShow( ) 程序为指定的 Socket 接口显示器件信息。有助于用户决定在编写启动镜像时所需要的擦除单元数目。tffsShowAll( )程序为所有注册在 VxWorks 中的 Socket 接口提供相同的信息。tffsShowAll( )程序在 shell 中按照注册顺序列出系统中的驱动器。默认情况下该组件不会被包含。可以从项目设备或通过定义来包含它。

⚠ **注意**: INCLUDE_TFFS_BOOT_IMAGE 在 Socket 驱动程序 sysTffs.c 中被默认定义。它通常用于配置基于闪存的启动镜像。定义这个常量会自动将 tffsBootImagePut( ) 包含在 sysTffs.o 文件中。这个程序被用来向闪存中写入启动镜像（请参见“8.7.3 在闪存中编写启动镜像”）。

8.5.4 包含 MTD 组件

将适用于闪存的 MTD 组件加入到系统项目中。如果编写了自己的 MTD，请将 MTD 定义为组件，并且确保该 MTD 组件被正确地定义。可以通过两种方法加入 MTD 组件，其一是通过项目设备，其二是通过在 Socket 驱动程序文件中定义 MTD 来进行命令行建立（请参见“8.5.7 建立系统项目”）。无论采用那种方法，配置 MTD 组件以及建立项目必须是相同的。

由 TrueFFS 提供，并适用于 Intel，AMD，Fujitsu 和 Sharp 生产的闪存设备的 MTD 组件如下所示：

INCLUDE_MTD_CFISCS
CFI/SCS 设备，详细内容见“CFI/SCS 闪存支持”。
INCLUDE_MTD_CFIAMD
遵守 CFI 的 AMD 和 Fujitsu 设备，详细内容见“AMD/Fujitsu CFI 闪存支持”。
INCLUDE_MTD_I28F016
Intel 28f016; 详细内容见 Intel 28f016 闪存支持。
INCLUDE_MTD_I28F008
Intel 28f008; 详细内容见 Intel 28f008 闪存支持。
INCLUDE_MTD_AMD
AMD，Fujitsu: 29F0{40，80，16} 8-bit 设备，详细内容见“AMD，Fujitsu 闪存支持”。
INCLUDE_MTD_WAMD
AMD，Fujitsu 29F0{40，80，16} 16-bit 设备
INCLUDE_MTD_I28F008_BAJA
Heurikon Baja 4000 上的 Intel 28f008

在组件描述文件中定义的 MTD（也就是通过项目设备包含的 MTD）通常需要转换层。如果是从命令行建立的或是编写了自己的 MTD，需要包含转换层。

⚠ **注意**：尽管组件也可以在 config.h 中被定义，但我们不推荐这种做法。因为这会与项目设备配置发生冲突。

8.5.5 包含转换层

我们根据闪存媒质所采用的技术对转换层进行选择。NOR 和 NAND 是闪存的两种类型。本产品只支持遵守 SSFDC 规定的 NAND 器件。详细内容请参见“8.11.3 获得片上磁盘的支持”。

转换层只具有二进制形式。下面列出了转换层的组件：

INCLUDE_TL_FTL，支持 NOR 闪存设备的转化层。如果能够在闪存中执行代码，那么器件使用的是 NOR 逻辑。

INCLUDE_TL_SSFDC，转换层支持遵守东芝固态软盘控制器规范的设备。TrueFFS 支持惟一遵守 SSFDC 规范的 NAND 设备。

组件描述文件说明了转换层和 MTD 之间的相关性。因此，当通过项目设备进行配置的时候，不必选择一个转换层，建立过程会完成这项工作。

如果不使用项目设备，需要选择正确的转换层。因为当使用 MTD 从命令行进行配置和建立时，只是在条件语句#ifndef PROJECT_BUILD 里将转换层定义到 sysTffs.c 文件中，并未选择转换层。详细内容请参见“有条件编译”。

详细内容请参见“8.12.4 定义 MTD 为组件”。

8.5.6 加入 Socket 驱动程序

为了将 socket 驱动程序包含到项目中，BSP 目录中必须要有 Socket 驱动程序的一个流行版本。

包含 Socket 驱动程序的过程显得相对简单。通过将 TrueFFS 核心组件 INCLUDE_TFFS 包含到项目中，建立过程会在 BSP 目录中检查 Socket 驱动程序文件 sysTffs.c，并将该文件包含到系统项目中。

如果 BSP 不提供 Socket 驱动程序，请查看“8.10 编写 Socket 驱动程序”所描述的过程。

8.5.7 建立系统项目

建立系统项目既可以通过命令行，也可以通过 Tornado IDE。在建立系统项目的时候，请考虑下面两个问题：

有条件编译

文件 sysTffs.c 在条件子句#ifndef PROJECT_BUILD 中定义了常量。默认情况下，这些常量包含了支持 TrueFFS 产品所提供的所有 MTD 组件的定义。PROJECT_BUILD 语句有条件地包含所有支持命令行建立的常量（即把它们加到 sysTffs.o 文件中），并把这些常量排除在建立项目设备之外（因为从 GUI 包含了它们）。因此，配置 MTD、转换层组件和建立项目都必须使用相同的方式。

如果从命令行建立，并希望存储在内存空间，可以取消那些硬件不支持的 MTD 的常量定义。

更新组件

在建立项目之前，MTD 的二进制数据需要更新，而且 BSP 目录中的 the sysTffs.c 必须是适用版本。

MTD 具有源代码和二进制形式。当编写自己的 MTD 时，我们推荐重建目录来将源代码转换到二进制码。通过这种方法，二进制数据被放置在正确的位置，并被加到适当的库中。如果 *installDir*/target/src/drv/tffs/*.c 组中的任何文件比库 *installDir*/target/lib/*archFamily*/*arch*/common/libtffs.a 中的相应文件要新，则需要在建立项目之前重建它们。

8.6 设备格式化

启动系统。在系统启动并注册 Socket 驱动程序后，调出 shell 程序。从 shell 中运行 tffsDevFormat( )格式化闪存来使用 TrueFFS。这个函数定义在 tffsDrv.h 文件中，该函数具有两个变量，一个是驱动器号，另一个是格式化变量。

tffsDevFormat（int *tffsDriveNo*，int *formatArg*）;

⚠ **注意**：即使还没有与闪存相关的块设备驱动程序，也能格式化闪存介质。

警告：如果在与文件系统共用启动代码的设备上运行 tffsDevFormat( )，运行结束后电路板（board）上不会留有启动代码。除非采用其他方法重新装入丢失的代码，否则该电路板无法使用。但是这样做会使用户通过格式化创建的文件系统被破坏。

8.6.1 规定驱动器号

第一个变量 tffsDriveNo 是驱动器号（Socket 驱动器号）。驱动程序号的识别需要格式化的闪存介质，该号码是由 Socket 驱动程序注册的顺序决定的。大多数系统只有一个闪存驱动程序，但 TrueFFS 最多支持五个闪存驱动器。驱动器号被分配给目标机硬件上的闪存驱动程序，分配的顺序是由启动期间 Socket 驱动程序在 sysTffsInit( )中注册的顺序决定的。第一个被注册的是驱动器 0，第二个是驱动器 1，以此类推到驱动器 4。因此，Socket 注册顺序决定了驱动器号（这个过程的详细内容请参见“Socket 注册”），可以使用这个号码在格式化时指定驱动器。

8.6.2 对设备进行格式化

第二个变量 formatArg，是一个指向结构 tffsDevFormatParams 的指针。这个结构描述了驱动程序是如何被格式化的。结构 tffsDevFormatParams 被定义在 *installDir*\target\h\tffs\tffsDrv.h 中，如下所示：

```
typedef struct
    {
     tffsFormatParams  formatParams;
     unsigned          formatFlags;
    }tffsDevFormatParams;
```

第一个成员 formatParams 具有 tffsFormatParams 类型。

第二个成员 formatFlags 是无符号整型。

TFFS_STD_FORMAT_PARAMS 宏。

为了方便从目标机 shell 中调用 tffsDevFormat( )，可以将 0 发送到第二个变量 formatArg。这样做会用到一个宏，它定义了结构 tffsDevFormatParams 的默认值。宏 TFFS_STD_FORMAT_PARAMS 定义了用于格式化闪存磁盘驱动程序的默认值。这个宏定义在 tffsDrv.h 中，如下所示：

```
#define TFFS_STD_FORMAT_PARAMS {{0, 99, 1, 0x100001, NULL, {0, 0, 0, 0}, NULL,
2, 0, NULL}, FTL_FORMAT_IF_NEEDED}
```

如果第二个变量 formatArg 是 0，tffsDevFormat( )会从这个宏中使用默认值。

宏将数值传递给结构 fsDevFormatParams 的第一、第二个成员。如下：

```
formatParams = {0,99,1,0x100001,NULL,{0,0,0,0},NULL,2,0,NULL}
formatFlags = FTL_FORMAT_IF_NEEDED
```

这些默认值以及其他结构成员的变量的含义请参见下文：

formatParams 成员

formatParams 成员是 tffsFormatParams 类型。这个结构以及宏 FFS_STD_FORMAT_PARAMS 使用的默认值被定义在 *installDir*\target\h\tffs\tffsDrv.h 中。

如果使用 STD_FORMAT_PARAMS 宏，为使用 TrueFFS，该宏的默认值会格式化整个闪存媒质。改变 formatParams 的最重要的原因是为了支持一个启动区域。如果想创建一个排斥 TrueFFS 的启动镜像区域，只需要改变结构 tffsFormatParams 中的第一个成员来更改这些默认值，详细内容请参见“8.7 创建用于编写启动镜像的区域”。

formatFlags 成员

tffsDevFormatParams 结构的第二个成员 formatFlags，决定了格式化驱动器的方式。表 8-1 列出了 formatFlags 的几个可能的数值：

表 8-1　formatFlags 选项

宏	数　值	含　义
FTL_FORMAT	1	FAT 和 FTL 格式化
FTL_FORMAT_IF_NEEDED	2	FAT 格式化，当需要时，需要对 FTL 格式化
NO_FTL_FORMAT	0	只是 FAT 格式化

默认的宏 TFFS_STD_FORMAT_PARAMS 将 FTL_FORMAT_IF_NEEDED 作为数值传递给变量。

8.7 创建用于编写启动镜像的区域

尽管 TrueFFS 的转换服务在管理文件系统相关数据方面有不少优势，但这些服务如同一个启动设备一样会使闪存的应用复杂化。惟一可行的解决方法是先创建一个排斥 TrueFFS 的启动镜像区域，然后将启动镜像写到这个区域中。本节首先介绍了这个过程的技术细节，然后介绍了怎样创建这个区域，最后介绍了怎样在该区域中编写启动镜像。

8.7.1 写保护闪存

TrueFFS 要求所有与文件系统相互作用的闪存器件（包括启动镜像区和 NVRAM 区），不能通过 MMU 写保护。这是因为该产品的正常工作需要所有发送给器件的命令能够到达，而写保护功能阻止非面向写操作的命令到达器件，从而影响了正常工作。相关信息请参见“rfa 写保护”。

然而，可以保留一个未被文件系统填充的空区域。在对器件进行格式化时，TrueFFS 允许用户指定一个空区域，从而支持启动代码。空区域通常被保留在闪存的起始处。在一些情况下，结构端口需要空区域位于闪存的末端。这可以通过设置识别过程中 MTD 的尺寸（即通知 MTD 内存不够）来完成，然后通知格式化调用不需要空区域。TrueFFS 不关心空区域是怎样管理的，也不会被任何伪造所影响。

8.7.2 创建启动镜像区域

为了创建启动镜像区域，需要格式化闪存使 TrueFFS 段在一定偏移后开始。这样，就在未被格式化的闪存中创建了一个空区域，从而保留了启动镜像区。如果愿意更新启动镜像，可以在这个空区域中编写一个启动镜像。相关内容见“8.7.3 在闪存中编写启动镜像”。

1. 偏移格式化

为了在偏移之后格式化闪存，需要初始化结构 tffsFormatParams 为启动试图在闪存设备中留下一定空间。这时需要为结构的 bootImageLen 成员指定某个数值，该数值至少与启动镜像所占的内存数量一样大。bootImageLen 成员制定了偏移量并在偏移之后对闪存媒质进行格式化以使用 TrueFFS。有关 bootImageLen 以及结构其他成员的详细内容请参见头文件 *installDir*\target\h\tffs\tffsDrv.h。

在偏移量之前由 bootImageLen 决定的区域不支持 TrueFFS。这个特殊的区域对于启动

镜像是必要的，因为正常的 TrueFFS 转换和磨损测评功能与镜像程序及其依赖的启动镜像的需要并不兼容。当 tffsDevFormat( ) 格式化闪存的时候，它会注意到偏移量，然后进行擦除并从高于偏移量的起始地址开始对所有的擦除单元进行格式化。包含偏移量地址的擦除单元被保留下来，不进行格式化。因此在偏移量地址之前的所有数据被保留下来。

有关磨损测评的详细内容请参见“8.13 闪存功能”。

2. 使用 BSP 帮助函数

一些 BSP 提供一个可选的、与 BSP 匹配的帮助函数 sysTffsFormat( )，可以从外部调用它来创建或保留启动镜像区。这个函数首先建立一个指向结构 tffsFormatParams 的指针。该结构已经被初始化，其中在偏移地址格式化的 bootImageLen 被赋值，然后它调用了 tffsDevFormat( )。

一些 BSP（如 ads860 BSP）包括 sysTffsFormat( )函数，该函数为启动镜像保留了 0.5 MB 空间。下面是一个例子：

```
STATUS sysTffsFormat (void)
    {
    STATUS status;
    tffsDevFormatParams params =
        {
#define HALF_FORMAT
/* lower 0.5MB for bootimage,upper 1.5MB for TFFS */

#ifdef  HALF_FORMAT
    {0x800001,99,1,0x100001,NULL,{0,0,0,0},NULL,2,0,NULL},
#else
    {0x0000001,99,1,0x100001,NULL,{0,0,0,0},NULL,2,0,NULL},
#endif  /* HALF_FORMAT */

    FTL_FORMAT_IF_NEEDED
    };

/*假定驱动器 0 是 SIMM */

    status = tffsDevFormat (0, (int) &params) ;
    return (status) ;
    }
```

有关使用 sysTffsFormat( )的例子，请参见 *installDir*/target/src/drv/tffs/sockets 目录中的 Socket 驱动程序。如果 BSP 不提供 sysTffsFormat( )函数，那么需要创建一个类似的函数，或将适当的变量传递给 tffsDevFormat( )。

8.7.3 在闪存中编写启动镜像

如果创建了一个启动镜像区域，需要编写启动镜像到闪存设备中。这时，可以使用tffsBootImagePut( )，它会绕过 TrueFFS（及其转换层）直接写到闪存中的任意位置。然而，因为 tffsBootImagePut( )依靠对 tffsRawio( )的调用，一旦设置 TrueFFS 的卷标（volume），将无法使用这个函数。

警告：因为 tffsBootImagePut( )允许直接写到闪存中的任意位置，可能或造成重复写入并损坏闪存的 TrueFFS-管理区域。有关小心使用该应用程序的详细内容请参见 VxWorks API 参考中的 tffsBootImagePut( )条目。

tffsBootImagePut( )函数在 *installDir*/target/src/drv/tffs/tffsConfig.c 中定义，如下：

```
STATUS tffsBootImagePut
    (
    int     driveNo,                /* TFFS 驱动器号 */
    int     offset,                 /* 闪存板卡中的偏移量 */
    char *  filename                /* 启动镜像的二进制形式*/
    )
```

该函数有以下变量：
driveNo 参数与输入到格式化程序中的驱动器号相同。
offset 偏移参数是闪存中从镜像写入开始的实际偏移量。
filename 参数是指向启动镜像的指针（bootApp 或启动 ROM 镜像）。

注意：bootImagePut( )函数的详细描述，请参见 *installDir*/target/src/drv/tffs/tffsConfig.c。

8.8 安装驱动器

接下来，使用 usrTffsConfig( ) 函数将 VxWorks DOS 文件系统安装在 TrueFFS 闪存驱动器上。该函数在 *installDir*/target/config/comps/src/usrTffs.c 中定义：

```
STATUS usrTffsConfig
    (
    int     drive,                  /* TFFS 驱动器号   */
    int     removable,              /* 不可删除闪存媒质为 0   */
    char * fileName                 /* 安装点 */
    )
```

该函数由三个参数：

Drive 参数指定了 TFFS 闪存驱动程序的驱动器号。有效值为 0 和 BSP 中的 Socket 接口号码。

Removable 参数指定了媒质是否是可删除的，0 表示不可删除，1 表示可删除。

FileName 参数指定了安装点，例如'/tffs0/'。

下面的例子运行 usrTffsConfig( )将一个驱动器连接至 dosFs，然后运行 devs，列出所有的驱动程序：

```
% usrTffsConfig 0,0,"/flashDrive0/"

% devs
drv      name
0        /null
1        /tyCo/0
1        /tyCo/1
5        host:
6        /vio
2        /flashDrive0/
```

实际上，usrTffsConfig( ) 调用了其他函数来传递输入的参数。

在这些函数中，tffsDevCreate( )在 Socket 驱动程序的顶部创建了一个 TrueFFS 块设备。这个函数中有一个号码（0~4）标志着 TrueFFS 块设备创建在哪一个 Socket 驱动程序的顶部。tffsDevCreate( )使用这个号码作为 FLSocket 结构数组的索引。稍后这个号码将作为 dosFs 的驱动号。

在创建 TrueFFS 块设备后，dcacheDevCreate( ) 和 dosFsDevCreate( )相继调用。这个函数将 dosFs 安装到设备上。安装 dosFs 之后，用户就能从闪存上读和写操作，就象从一个标准硬盘驱动程序上进行上述操作一样。

8.9　运行 shell 命令

下面的每一个例子都假设已经建立 VxWorks 并启动目标机。

1. 具有板上闪存阵列和启动镜像的 assabet

这个例子使用 sysTffsFormat( )来格式化基于插件板的闪存，并保留启动镜像区。它不会更新启动镜像，所以不会调用 tffsBootImagePut( )。然后，它设置非可删除 RFA 媒质的驱动程序号为 0。

在目标机 shell 提示处，输入以下命令：

```
-> sysTffsFormat
-> usrTffsConfig 0,0,"/RFA/"
```

2. 具有基于插件板的闪存数组和一个 PCMCIA 插口的 ads860

这个例子格式化 RFA 和 PCMCIA 闪存，并将其作为两个驱动程序。

本例中的一行通过调用帮助函数 sysTffsFormat( )（该函数保留启动镜像区域）对基于插件板的闪存进行格式化。本例并没有更新启动镜像。然后安装驱动程序，将其设置为 0，并将 0 传递给 usrTffsConfig( )的第二变量，这是因为 RFA 是非可删除的。

本例的最后一行格式化闪存，传递默认的格式化数值给 tffsDevFormat( )以格式化整个驱动程序。然后再安装那个驱动程序。因为 PCMCIA 是可删除闪存，所以 1 被传递给 usrTffsConfig( )的第二个变量（请参见“8.8 安装驱动器”有关 usrTffsConfig( )变量的内容）。

将一个闪存卡插入 PCMCIA 插座。在目标机 Shell 提示处输入下面的命令：

```
-> sysTffsFormat
-> usrTffsConfig 0,0,"/RFA/"
-> tffsDevFormat 1,0
-> usrTffsConfig 1,1,"/PCMCIA1/"
```

3. 具有基于插件板的闪存阵列、不具有启动镜像的 mv177

本例使用 tffsDevFormat( )的默认参数对基于插件板的闪存进行格式化，请参见“8.6 设备格式化”。然后，它安装驱动程序，设置驱动程序号为 0 并提示闪存是非可删除的。

在目标机 shell 提示处键入下列命令：

```
-> tffsDevFormat 0,0
-> usrTffsConfig 0,0,"/RFA/"
```

4. 具有使用 INCLUDE_PCMCIA 的两个 PCMCIA 插口的 x86

本例使 PCMCIA 闪存格式化为两个驱动程序。每一个驱动程序格式化调用都不会保留启动镜像区域。然后，它安装驱动程序，第一个标号为 0，第二个标号为 1。PCMCIA 是一个可删除媒质。

将一个闪存卡插入每一个 PCMCIA 插座。在目标机 SHELL 提示处输入下面的命令：

```
-> tffsDevFormat 0,0
-> usrTffsConfig 0,1,"/PCMCIA1/"
-> tffsDevFormat 1,0
-> usrTffsConfig 1,1,"/PCMCIA2/"
```

8.10　编写 Socket 驱动程序

Socket 驱动程序在文件 sysTffs.c 中被执行。如果 BSP 不包含 Socket 驱动程序的存根版本，TrueFFS 将为这些 BSP 提供该版本。如果要编写 Socket 驱动程序，下面所列出的 Socket 驱动程序文件的几个重要内容需要注意：

- sysTffsInit( )是主函数，该函数调用了 Socket 注册函数。
- *xxx*Register( )是 Socket 注册函数，它将函数分配给 socket 结构的成员函数。
- 由 Socket 注册函数分配的函数。
- 反映硬件的宏值。

在这个存根文件中，所有函数都需要声明。其中大多数函数都是完整定义的，尽管一些使用了需要修改的类属或假定的宏。

存根文件中的 Socket 注册函数只是为 RFA 驱动程序编写的。PCMCIA Socket 驱动程序没有注册函数的存根版本。如果正在为 RFA 编写 Socket 驱动程序，可以使用这个存根文件并参考“8.10.1　传送 Socket 驱动程序存根文件”所描述的步骤。如果正在编写 PCMCIA Socket 驱动程序，请看 *installDir/target/src/drv/tffs/sockets/pc386-sysTffs.c* 中的例子以及“8.10.2　理解 Socket 驱动程序功能”中的内容。

⚠ **注意**：有关其他 RFA Socket 驱动程序的例子请参见 *installDir*/target/src/drv/tffs/sockets。

8.10.1　传送 Socket 驱动程序存根文件

如果 BSP 不提供 Socket 驱动程序，需要编写自己的 Socket 驱动程序。当运行 build 时，Socket 驱动程序的存根文件 sysTffs.c，从 *installDir*/target/config/comps /src 中被复制到 BSP 目录下。也可以在运行 build 之前将该版本手动复制到 BSP 目录下。无论怎样，只有复制到 BSP 目录中的文件会被编辑，原始的存根文件不会被改变。

这个存根版本是起点，它会帮助传送 Socket 驱动程序到 BSP 中。尽管如此，它包含不完全的代码不能被编译。用户需要作如下修改（这些修改并不多且全部用/* TODO */标注）。

- 用反映硬件的正确数值代替“fictional”宏值，如 FLASH_BASE_ADRS。然后删除下面列出的行：

```
#error "sysTffs: Verify system macros and function before first use"
```

- 为 BSP 支持的每一个附加的设备向注册程序增加调用。因此，如果只有一个设备，可以跳过此步。详细内容请看“调用 Socket 注册函数”。
- 回顾两个标有/* TODO */的函数的执行，可以不必为他们增加代码。详细内容请参见“执行 Socket 结构成员函数”。

警告：不要编辑 *installDir*/target/config/comps /src 中的 sysTffs.c 存根版本 的原始复制，因为在以后还需要传送它们。

1. 调用 Socket 注册函数

sysTffs.c 中的主函数是 sysTffsInit( )，它在启动事件被自动调用。该函数的最后一行为每一个系统支持的设备调用 Socket 注册函数。sysTffs.c 存根文件调用了 Socket 注册函数 rfaRegister( )。

如果 BSP 只支持一个（RFA）闪存设备，不需要编辑这个部分。然而，如果 BSP 支持几个闪存设备，就需要编辑存根文件来为每一个 Socket 的注册函数增加调用。需要这样做的地方在 sysTffsInit( )函数中用/* TODO */注视标明。

如果有几个 Socket 驱动程序，可以将每一个 *xxx*Register( )调用封装在预处理条件语句中，如下所示：

```
#ifdef  INCLUDE_SOCKET_PCIC0
    (void) pcRegister (0,PC_BASE_ADRS_0) ;    /* socket 0 上的闪存板卡*/
#endif  /* INCLUDE_SOCKET_PCIC0 */

#ifdef  INCLUDE_SOCKET_PCIC1
    (void) pcRegister (1,PC_BASE_ADRS_1) ;    /*  socket 1 上的闪存板卡*/
#endif  /* INCLUDE_SOCKET_PCIC1 */
```

在定义了 BSP 的 sysTffs.c 文件中的常量后，就能使用它们有选择地控制在编译时间包含在 sysTffsInit( )中的调用。

2. 执行 Socket 结构成员函数

Socket 驱动程序存根文件也包含了 rfaRegister( ) 函数的执行，该过程将函数分配到 FLSocket 结构的成员函数，TrueFFS 使用该结构来存储用于处理硬件与闪存设备接口的数据和函数指针。大多数情况下，需要了解分配给 FLSocket 结构的函数而不必关心 FLSocket 结构。一旦这些函数被执行，就不能直接调用它们。它们自动被 TrueFFS 调用。

所有被注册程序分配给 Socket 结构成员函数的函数在 Socket 驱动程序存根模块中得到定义。但是，只有 rfaSocketInit( ) 和 rfaSetWindow( )函数是不完整定义的。当编辑存

根文件时，请注意代码中的#error 标号以及/* TODO */注释。它们表明在哪儿以及怎样修改代码。

下面列出了由注册程序分配的一列函数，并描述了其中的每一个函数是怎样在存根文件中执行的。下面还列出了两个需要注意的函数，并描述了它们是怎样被执行的。

> ⚠ **注意**：有关每一个函数功能的详细介绍请参见“8.10.2 理解 Socket 驱动程序功能”。然而，这些信息对于传送 Socket 驱动程序并不是必要的。

rfaCardDetected

该函数在 RFA 环境中通常返回 TRUE，因为这个设备是不可删除的。该函数在存根文件中的执行是完全的。

rfaVccOn

为了有利于退出（exit），Vcc 必须是已知的。在 RFA 环境中 Vcc 始终为 ON。该函数只是一个包装函数。尽管它在存根文件中的执行是完全的，仍然需要增加代码。

当把转换 Vcc 为 on，只有 Vcc 稳定在适当的操作电压时，VccOn( )函数才会返回。如果必要，函数需要用空循环延迟执行或调用 flDelayMsec( ) 函数直至 Vcc 稳定下来。

rfaVccOff

Vcc 在 RFA 环境中通常为 ON。这个函数只是一个包装函数，并且在存根文件中执行完全。

rfaVppOn

为了有利于退出（exit），Vpp 必须是已知的，并且在 RFA 环境中通常为 ON。该函数不是可选的并且必须被执行。因此，不可以删除这个函数。尽管该函数在存根文件中的执行是完全的，仍然可以增加代码，如下所述。

当 Vpp 转换为 on 时，除非 Vpp 稳定在适当的电压，否则 VppOn( )函数不可以返回。如果必要，VppOn( )函数需要用空循环延迟执行或调用 flDelayMsec( ) 函数直至 Vpp 稳定下来。

rfaVppOff

Vpp 假定在 RFA 环境中始终为 ON。该函数在存根文件中是执行完全的。然而，它不是可选的，并且必须被执行。因此，不可以删除这个函数。

rfaSocketInit

包含一条/* TODO */语句。该函数在每一次初始化 TrueFFS（访问驱动程序）时被调用，并确保闪存处于一个可用的状态下（即板级初始化）。如果有什么需要在访问驱动程序之前做的，那么需要在该函数中执行访问。详细内容请参见“nrfaSocketInit”。

rfaSetWindow

包含一条/* TODO */语句。

该函数使用了在存根文件中设置的 FLASH_BASE_ADRS 和 FLASH_SIZE v 数值。只

要这些数值是正确的，那么该函数在存根文件中的执行是完全的。

TrueFFS 调用该函数对 Window 结构的关键成员进行初始化（Window 结构是 FLSocket 结构的一个成员）。对于大多数硬件，setWindow 函数实现了以下功能：

按照 4 KB 页面设置 window.baseAddress 为基地址。

调用 flSetWindowSize( )在 4 KB 单元中指定窗口的大小。实际上，对 flSetWindowSize( ) 的调用设置了窗口大小、窗口基地址以及窗口的当前页面。

该函数设置了当前的窗口硬件性能：基地址、大小、速度和总线宽度。所需要的设置在 vol.window 结构中给出。如果窗口大小的设置无法达到 vol.window.size 的要求，那么窗口大小的值应该尽量大一些。无论哪种情况，vol.window.size 应该包含实际窗口大小尺寸（在 4KB 单元中）。

详细内容请参见“nrfaSetWindow”和“Socket 窗口及地址映射”。

警告： *在具有多个 nrfaSetWindow 驱动程序的系统中（处理多个闪存设备），要保证每一个 nrfaSetWindow 的窗口基地址不同。另外，窗口尺寸必须要保证窗口不会重叠。*

rfaSetMappingContext

TrueFFS 调用这个函数设置窗口映射注册。因为基于插板的闪存阵列通常将整个闪存映射在内存中，他们不需要这个功能。在存根文件中它是一个包装函数，所以它的执行是完全的。

rfaGetAndClearChangeIndicator

通常在 RFA 环境中返回 FALSE，因为设备是不可删除的。这个函数在存根文件中的执行是完全的。

rfaWriteProtected

该函数在 RFA 环境中通常返回 FALSE。这个函数在存根文件中的执行是完全的。

8.10.2 理解 Socket 驱动程序功能

在 PCMCIA Socket 启动服务之后，TrueFFS 中的 Socket 驱动程序被格式化。尽管如此，它们必须提供以下内容：

- 控制 Socket 电源（Socket 为 PCMCIA，RFA 或其他类型）；
- 建立内存窗口环境的标准；
- 支持板卡变更识别。

Socket 初始化函数。

本节描述了 Socket 注册，Socket 成员函数以及这些函数设置的开窗和地址映射。这些信息对于存根 RFA 文件的传送是不必要的。然而，它对于编写 PCMCIA socket 驱动程序是

有用的。

1. Socket 注册

注册函数执行的第一个任务是为 Socket 结构分配驱动程序号。该任务在存根文件中执行完全。只需注意格式化驱动程序时的驱动程序号（“8.6.1 规定驱动器号”）。

驱动程序号是 FLSocket 结构的一个预分配阵列的索引号码。与驱动程序相关的驱动程序号由注册的顺序决定，如 rfaRegister( ) 函数的第一行代码所示：

```
FLSocket vol = flSockeOf (noOfDrives);
```

这里，noOfDrives 表示与系统相连的正在运行的驱动程序数目。函数 flSocketOf( ) 会向 Socket 结构返回一个指针用来描述内存容量，该指针随系统每调用一个 Socket 注册函数而加 1。这样，Socket 结构中的 TrueFFS 核被分配给系统支持的 5 个驱动程序[②]。当 TrueFFS 调用那些处理硬件接口需要的函数时，它就会使用驱动程序号作为索引来访问 Socket 硬件（即一个闪存设备）。

2. Socket 成员函数

rfaCardDetected

该函数会报告在 PCMCIA 插槽中是否有与之相关的闪存板卡。对于不可删除的媒质，函数通常会返回 TRUE。实际上，支持 Tornado 的 TrueFFS 每 100 毫秒就会调用该函数检查闪存媒质是否还在。如果函数返回 FALSE，TrueFFS 就会设置 cardChanged 为 TRUE。

rfaVccOn

TrueFFS 可以调用这个函数开启 Vcc，Vcc 是操作电平。对于闪存硬件，Vcc 通常为 5 或 3.3 伏特。当媒质空闲时，TrueFFS 会在操作结束之后关断 Vcc 来保持电源。在访问闪存之前，TrueFFS 会使用该函数重新开启电源。

然而，当 Socket 储集（polling）为当前设置时，系统会引入延迟 Vcc 关闭机制，即 Vcc 将会在至少一个间隔过后被关断。如果快速执行几个访问闪存的操作，Vcc 会在操作期间保持开启；当 TrueFFS 进入相对空闲的状态时，Vcc 才会关断。

rfaVccOff

TrueFFS 可以调用这个函数为闪存硬件关断工作电平。当媒质空闲时，TrueFFS 会关断 Vcc。然而当 Socket 储集（polling）为当前设置时，Vcc 会在延迟之后关断。这样，如果快速执行访问操作的标示符序列，在执行期间会保持 Vcc。Vcc 只有当 TrueFFS 进入一个相对空闲的状态才会被关断。在 RFA 环境中 Vcc 通常为 ON。

rfaVppOn

该函数不是可选的而且必须要执行。TrueFFS 会调用这个函数来管理编程电平 Vpp。对于闪存芯片，Vpp 通常为 12V。因为不是所有的闪存芯片都需要这个电平，只有当 SOCKET_12_VOLTS 被定义时才会包含该成员。

[②] TrueFFS 最多只能支持 5 个驱动程序。

Vpp 必须保持直至结束编程，而且在 RFA 环境中它始终为 ON。

⚠ **注意**：只有拥有 TrueFFS 核心源代码的用户能更改宏 SOCKET_12_VOLTS。

rfaVppOff

TrueFFS 调用该函数关闭闪存芯片的编程电平（Vpp 通常为 12 Volts）。因为不是所有的闪存芯片都需要这个电平，该成员只有在 SOCKET_12_VOLTS 被定义之后才会被包含。该函数不是可选的，而是必须执行的。Vpp 在 RFA 环境中通常为 ON。

rfaSocketInit

TrueFFS 会在它访问 Socket 之前调用这个函数。TrueFFS 使用该函数在访问 Socket 之前进行必要的初始化，尤其在 Socket 注册时初始化无法进行的情况下。例如，如果在 Socket 注册时没有检查硬件，或者如果闪存媒质是可删除的，这个函数会检查闪存媒质并响应，包括如果硬件发生变化会设置 cardDetected 为 FALSE。

rfaSetWindow

TrueFFS 使用 window.base 来存储闪存上内存窗口的基地址，使用 window.size 来存储内存窗口的尺寸。TrueFFS 假定它是惟一能够访问内存窗口的，也就是说，在 TrueFFS 设置了这些窗口特征之后，它不允许应用程序直接改变它们，如果这样做就会导致冲突。然而映射寄存器（mapping register）是个例外，因为 TrueFFS 通常在它访问闪存的时候重建这个映射寄存器，所以应用程序可以变换这个窗口。但是，不要从中断函数中这样做。

rfaSetMappingContext

TrueFFS 调用该函数设置窗口映射寄存器。该函数通过适当设置映射寄存器为某一数值来执行“滑动”。因此，这个函数只有在使用滑动窗机制检查闪存的环境中（PCMCIA）才有意义。在 PCMCIA 插槽中的闪存板卡也能使用这个函数来访问/设置一个映射寄存器，该寄存器将有效的闪存地址移至主机的内存窗口。映射过程获取一个 “板卡地址”（闪存的偏移量），并从中产生真实的地址。如果偏移量超出了闪存的长度，它也可以包装该地址，使其位于闪存的起始处。由上可知，闪存的大小是 Socket 驱动程序中必需的一个实体。一旦进入 setMappingContext，vol.window.currentPage 就被映射到窗口中的页面（这意味着它是由最后一次调用 setMappingContext 映射的）。

rfaGetAndClearChangeIndicator

该函数读取硬件板卡变化显示，并将其清除。它是检查媒质-变化事件的基础。如果没有这样的硬件性能，则向该函数返回 FALSE（设置该函数指针为 NULL）。

rfaWriteProtected

TrueFFS 调用该函数来获取媒质写保护转换的当前状态。该函数会返回媒质的写保护状态，而且对于 RFA 环境通常返回 FALSE。详细内容请参见“8.7.1 写保护闪存”。

3. Socket 窗口和地址映射

FLSocket 结构（定义于 *installDir*/target/h/tffs/flsocket.h）包含有一个 window 状态结构。

如果正在传送 Socket 驱动程序，下面的有关这个 window 结构的背景信息会在执行 *xxx*SetWindow( ) 和 *xxx*SetMappingContext( )时有用。

窗口的概念来源于 PCMCIA 领域，它明确地表达了主机总线适配器的含义。

- PCMCIA 总线在主机地址范围内是完全可视的。
- PCMCIA 地址范围中只有一段（segment）在主机地址空间是可视的。
- 主机地址空间中只有一段（segment）对于 PCMCIA 是可视的。

为了支持这些概念，PCMCIA 使用了“窗基址寄存器”。可以改变它来调整窗口的视角。在典型的 RFA 环境中，器件逻辑为 NOR，窗口尺寸为插件板上闪存的大小。在 PCMCIA 环境中，窗口尺寸是特定的（implementation-specific）。Don Anderson 所著的《PCMCIA 系统结构》一书很好地解释了这个概念。

8.11　使用 MTD 支持的闪存设备

本节列出了本产品的 MTD 所支持的闪存设备。

8.11.1　支持常用闪存接口（CFI）

TrueFFS 支持的设备包括 Intel 使用可裁减指令集的设备和使用 AMD 指令集的设备。因此，为了增加代码的可读性，分别由两个 MTD 支持不同的指令集，只需配置系统来包含两个 MTD，即可支持两种指令集（请参见“8.5.4　包含 MTD 组件”）。两种命令集如下所示：

Intel/Sharp 指令集

这是适用于 SCS 指令集的 CFI 规范。这种 MTD 的驱动程序文件是 *installDir*/target/src/drv/tffs/cfiscs.c。Intel 网站中的 *Application Note 646* 提供了对 Intel/Sharp 指令集的支持。

AMD/Fujitsu 指令集

这是适用于 SCS 指令集的嵌入式程序算法和灵活区段结构。驱动程序文件是 *installDir*/target/src/drv/tffs/cfiamd.c。有关这种 MTD 支持的细节请参见“AMD/Fujitsu CFI 闪存支持”。

1. 公共功能

两种驱动程序均支持 8、16 位设备，以及 8-和 16-bit 间隔。宏配置用来控制这些以及其他配置事务，并且它必须要特别定义。如果欲修改代码，一定要重新配置宏。请留意下面的宏：

```
INTERLEAVED_MODE_REQUIRES_32BIT_WRITES
```

对于具有 16-bit 间隔并需要支持“Write to Buffer”指令的系统，这个宏必须被定义。

SAVE_NVRAM_REGION

在支持 TrueFFS 的系统中，每个闪存设备的最后一个擦除区被保留，用于启动参数的 Non-Volatile 存储。

CFI_DEBUG

用 DEBUG_PRINT 定义的 I/O 函数使驱动程序更加详尽。

BUFFER_WRITE_BROKEN

支持缓冲器尺寸大于 1 的系统，但不支持一次写入多个 byte 或 word。当该宏被定义时，强制缓冲器大小为 1。

DEBUG_PRINT

如果定义该宏，使用它的值能使驱动程序更加详细。

⚠ **注意**：这些宏只能通过在原文件中定义来进行配置，而不能通过 IDE 项目工具。

2. CFI/SCS 闪存支持

定义于 cfiscs.c 中的 MTD 支持遵守 CFI/SCS 规范的闪存组件。CFI 代表公共闪存接口，SCS 代表可裁减指令集。CFI 询问闪存组件特征的一种标准方法。SCS 是建在 CFI 规范基础上的第二层。这使得一个 MTD 能以普通方式（common）处理所有的 CFI/SCS 闪存技术。

CFI/SCS 联合规范已经被 Intel 和 Sharp 公司采纳，并且从 1997 年将该规范应用于所有的闪存组件。

CFI 文档可以从以下网址下载：

http://www.intel.com/design/flcomp/applnots/292204.htm

或者在 http://www.intel.com/design 中查找 CFI。

在 BSP 的 sysTffs.c 文件中定义 INCLUDE_MTD_CFISCS，从而将这个 MTD 包含在支持 Tornado 的 TrueFFS 中。

在最近的一些目标机插板上不存在 NVRAM（non-volatile RAM）电路，而 BSP 开发者习惯将高端闪存当作 NVRAM 使用。每个闪存中最后的擦除块用于构成这个区域。CFISCS MTD 通过提供常数 SAVE_NVRAM_REGION 给编译器来支持这个概念。由于闪存的大小等于擦除块大小乘以它的数目，如果该 MTD 被定义，闪存的尺寸会减小。NVRAM 会被保留下来，并且不会被重新写入。尤其是 ARM BSP 将闪存用作 NVRAM 来存储启动镜像。

3. AMD/Fujitsu CFI 闪存支持

在 AMD 和 Fujitsu 器件中，灵活区段结构（也称为启动块设备）只是在擦除块时被支持。然而，TrueFFS 核心和转换层并不了解启动块中的细分情况，这是因为 MTD 实现这种细分对于 TrueFFS 核心和转换层是透明的。29LV160 器件的数据层是由 35 个区段组成。然而，TrueFFS 核心和转换层将其中的 4 个启动块区段仅作为一个区段（64 KB）。这样 TrueFFS 核心只能探测到 32 个区段。因此，支持启动镜像的代码也不知道启动块无法为它提供直接

的支持。

AMD 和 Fujitsu 器件包括 Top and Bottom 导入设备。然而 CFI 询问过程并不能区分以上两种器件。这样，为了确定导入块的类型，驱动程序使用了 JEDEC 器件 ID。如果器件用这种 ID 注册，支持的器件数量会受到限制，并需要确认正在使用的器件是否在注册表中列出。

8.11.2　支持其他的 MTD

如果不使用 CFI-MTD，我们也提供了其他的 MTD：

1. Intel 28F016 闪存支持

在 i28f016.c 中定义的 MTD 支持 Intel 28F016SA 和 Intel 28F008SV 两种闪存器件。这种 MTD 能够辨认并支持任何基于这些芯片的闪存阵列或板卡。这种 MTD 支持用于访问 BYTE-模式 28F016 组件、数值为 2 和 4 的交错因子（interleaving factors）。

对于 WORD-模式组件访问，只支持非交叉模式（non-interleaved）。支持的闪存媒质包括：

Intel 2+ PC 系列板卡

M-系统 2+ PC 系列板卡

在 BSP 的 sysTffs.c 文件中定义 INCLUDE_MTD_I28F016，从而将这个 MTD 包含在支持 Tornado 的 TrueFFS 中。

2. Intel 28F008 闪存支持

在 I28F008.c 中定义的 MTD 支持 Intel 28F008SA、Intel 28F008SC 和 Intel 28F016SA/SV（具有 8-bit 可兼容模式）闪存组件。任何基于这些芯片的闪存阵列或板卡都能够由这种 MTD 辨认并支持。但是，该 MTD 不支持 28F016SA/SV 的 WORD-模式 （只支持 BYTE-模式）。这个 MTD 也支持所有的交错因子（1，2，4，...）。尽管 MTD 同时访问的闪存组件不能超过 4 个，但多于 4 的交错因子（Interleaving factor）也会被辨认出来。支持的闪存媒质包括：

M-系统 D-系列 PC 板卡

M-系统 S-系列 PC 板卡

Intel 2（8-mbit family）系列 PC 板卡

Intel 2+（16-mbit family）系列 PC 板卡

Intel Value 系列 100 PC 板卡

Intel 小型板卡

M-系统 PC-FD，PC-104-FD，Tiny-FD 闪存

在 BSP 的 sysTffs.c 文件中定义 INCLUDE_MTD_ I28F008，从而将这个 MTD 包含在支

持 Tornado 的 TrueFFS 中。

3. AMD/Fujitsu 闪存支持

在 amdmtd.c 中定义的 MTD 支持 AMD 的 C 系列和 D 系列闪存技术家族中的 AMD 闪存组件，也支持相应的 Fujitsu 闪存组件。支持的闪存类型有：

Am29F040（JEDEC IDs 01a4h，04a4h）

Am29F080（JEDEC IDs 01d5h，04d5h）

Am29LV080（JEDEC IDs 0138h，0438h）

Am29LV008（JEDEC IDs 0137h，0437h）

Am29F016（JEDEC IDs 01adh，04adh）

Am29F016C（JEDEC IDs 013dh，043dh）

任何基于这些芯片的闪存阵列或板卡都能够由这种 MTD 辨认并支持。这个 MTD 也支持交数值为 1，2，4 的叉因子。被支持的闪存媒质包括：

AMD and Fujitsu C-系列 PC 板卡

AMD and Fujitsu D-系列 PC 板卡

AMD and Fujitsu 小型板卡

在 BSP 的 sysTffs.c 文件中定义 INCLUDE_MTD_AMD，从而将这个 MTD 包含在支持 Tornado 的 TrueFFS 中。

8.11.3 获得片上磁盘的支持

以前，NAND 器件的指令具有以下两种形式之一：SSFDC/ Smart 媒质器件和 M-系统中的片上磁盘。它们分别由两个独立的转换层支持。为了向 M-系统提供增加片上磁盘的能力，必须要对产品进行特殊的优化使其不影响其他器件，而且必须直接从 M-系统获得对 M-系统器件的支持，同时 Tornado 产品不再支持这些器件。Tornado 的这个版本只支持遵守 SSFDC 规范的 NAND 器件。请参见“8.5.5 包含转换层”。

8.12 编写 MTD 组件

MTD 是一个软件模块，它为 TrueFFS 提供数据以及指针（该指针指向为闪存编程的函数）。所有的 MTD 都必须提供以下三种函数：写函数、擦除函数和识别函数。MTD 模块使用识别函数来辨别哪一种闪存器件的类型与该 MTD 相匹配。如果在写自己的 MTD，需要将它定义为一个组件并注册识别函数。

有关 MTD 的源代码请参见 *installDir*/target/src/drv/tffs 目录。

8.12.1　编写 MTD 识别函数

TrueFFS 提供一个闪存结构，其中包含每个闪存部分的信息。识别过程负责正确地建立闪存结构。

> ⚠ **注意**：由 M-Systems 或 Wind River 开发的许多 MTD 都提供了源代码形式，以演示怎样编写 MTD（in *installDir*/target/src/drv/tffs）。本节提供了编写 MTD 的一些其他的相关知识。

在为闪存阵列创建逻辑块器件的过程中，TrueFFS 会寻找一个与闪存器件匹配的 MTD。TrueFFS 从每个 MTD 调用识别函数，直至其中一个报告与闪存器件匹配。最先报告匹配的 MTD 将被采用。如果没有合适的 MTD，TrueFFS 会退回到默认的只读 MTD，它通过从 Socket 窗口中复制闪存器件中的内容进行读取。

MTD 识别函数的调用优先于 MTD 中其他函数。一个 MTD 识别函数具有以下形式：

FLStatus *xxx*Identify（FLFlash vol）

在 MTD 识别函数中，必须先察看器件的类型，怎样做取决于硬件。如果器件类型与这个 MTD 不匹配，返回 failure。否则，设置 FLFlash 结构的成员。如下所示：

每个 MTD 的识别函数必须注册在 mtdTable（定义于 *installDir*/target/src/drv/tffs/tffsConfig.c）中。每当卷标（volume）被安装，识别函数的列表就会被遍历来找到适合 volume 的 MTD。这对于 hot-swap 器件更为有利，因为以前确认过的器件一定是与特定 volume 相符合的惟一器件。

器件识别可以通过多种方式实现。如果器件遵守 JEDEC（Joint Electronic Device Engineering Council）或 CFI（Common Flash Interface）标准，就能够使用它们的识别过程。可以让 MTD 识别器件的多个版本（或者惟一一个版本）。

1. 初始化 FLFLash 结构成员

在识别过程的末期，ID 函数需要设置 FLFLash 结构中除了 Socket 成员以外的所有数据元。Socket 成员由 TrueFFS 中的内部函数设置。FLFlash 结构定义于 *installDir*/h/tffs/flflash.h。结构的成员列于下方：

type

闪存硬件的 JEDEC ID。这个成员由 MTD 的识别函数设置。

erasableBlockSize

已连接的闪存硬件的可擦除块尺寸（以 bytes 为单位）。该结构成员只考虑了交错因子。这样在 MTD 中设置这个值的时候，代码通常如下：

vol.erasableBlockSize = *aValue* * vol.interleaving;

这里 *aValue* 是不存在交叉存取的闪存芯片的可擦除块尺寸。

chipSize

用来组建闪存阵列的闪存芯片之一的存储容量（以 bytes 为单位）。这个值通过使用 flFitInSocketWindow( )全局函数由 MTD 设置。

noOfChips

用于组建闪存阵列的闪存芯片数目。

interleaving

闪存阵列的交错因子。这是跨越数据总线的器件数目。例如，在一个 32-bit 的总线上，我们可以有四个 8-bit 器件或两个 16-bit 器件。

flags

0~7 为 TrueFFS 使用而保留（它使用这些标示符来寻迹，如查询 volume 安装状态）。8~15 为 MTD 的使用而保留。

mtdVars

如果 MTD 使用这个字段，该字段需要由 MTD 识别函数初始化为指向一个私有存储区域。这是特定的实例。例如，假设有基于 I28F016 闪存的 Intel RFA；假如还有一个 PCMCIA 插座，并且要向其中插入一个具有相应闪存板卡，那么两种器件会使用相同的 MTD，而且 mtdVars 用作特定实例的变量，使 MTD 可以在系统中多次使用。

socket

这个成员是一个指向 FLSocket 结构的指针。该结构包含了数据以及指向 Socket 层函数的指针。TrueFFS 需要 socket 层函数管理闪存硬件的插板接口。在这个结构中引入的函数会在注册 Socket 驱动程序时安装（见“8.10　编写 Socket 驱动程序”）。而且，因为 TrueFFS 使用这些 Socket 驱动程序函数来访问闪存硬件，所以必须在运行初始化该结构的 MTD 注册函数之前，注册 Socket 驱动程序。

map

一个指向闪存映射函数的指针。该映射函数将闪存映射至内存区。TrueFFS 初始化 map 使其指向默认的映射函数，该函数适用于所有的 NOR（linear）闪存类型。这个默认函数可以通过简单的 Socket 映射来映射闪存。闪存应通过对采用 map-through-copy 函数的引用来取代这个指针。

read

一个指向闪存读函数的指针。一进入 MTD 识别函数，这个成员就已经被初始化使其指向一个默认读函数，该函数适于所有的 NOR（linear）闪存类型。该函数是通过从映射窗口中复制来实现从闪存中读取的。如果这适于闪存器件，请保持 read 不变。否则，MTD 识别函数会更新这个结构成员使其指向一个更适宜的函数。

write

一个指向闪存写函数的指针。由于存在不当写操作的危险，默认函数为这个成员返回

一个写保护错误。MTD 识别函数必须为这个成员提供一个更适宜的函数指针。

erase

一个指向闪存擦除函数的指针。由于存在不当擦除操作的危险，默认函数为这个成员返回一个写保护错误。MTD 识别函数必须为这个成员提供一个更适宜的函数指针。

setPowerOnCallback

该指针指向一个在闪存硬件电源启动后 TrueFFS 要执行的函数。当试图安装一个闪存器件时，TrueFFS 会调用该函数。不要将 FLFlash 结构的这个成员与 FLSocket 结构的 powerOnCallback 成员相混淆。对于许多闪存器件而言，这样的函数并不是必要的。然而，定义于 *installDir*/target/src/drv/tffs/nfdc2048.c 中的 MTD 会用到这个成员。

Return Value

识别函数会返回 flOK 或一个定义于 flbase.h 中的错误代码。源代码如下：

```
FLStatus myMTDIdentification
     (
    FLFlash vol
    )
    {
    /* 进行识别*/

    /* 如果识别失败,返回 error */

    return flOK;
    }
```

在设置了以上这些成员后，该函数会返回 flOK。

2. Call Sequence 调用顺序

识别函数会更新 FLFlash 结构，FLFlash 结构会完成 FLSocket 结构的初始化，而 FLFlash 结构中调用了 FLSocket 结构。

8.12.2　编写 MTD 映射函数

当 RAM 缓冲器被用于窗口时，MTD 需要提供一个映射函数。在本版本中，没有专门的 MTD 为这一类器件服务。如果正在使用的器件需要这种支持，需要将一个映射函数加到 MTD 中并分配一个指针，该指针指向 FLFlash.map 中的映射函数。该函数有三个变量：一个指向 volume 结构的指针、一个板卡地址和一个字段长度，并且该函数会返回一个空指针。图 8-3 所示为相应的 MTD 简图。

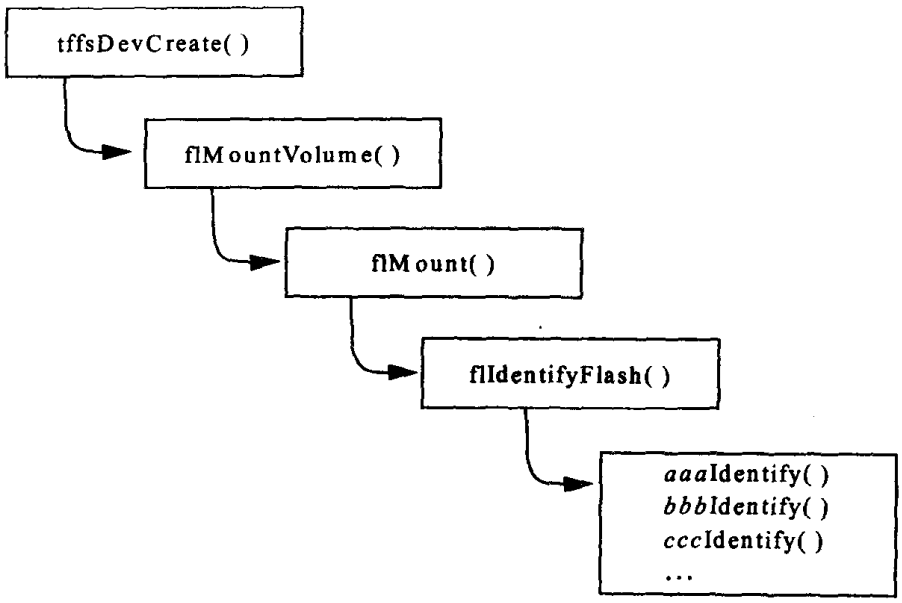


图 8-3　识别闪存相应的 MTD

```
static void FAR0 * Map
     (FLFlash vol,
     CardAddress address,
     int length
    )
    {
    /* 执行函数 */
    }
```

8.12.3　编写 MTD 读、写和擦除函数

读、写和擦除函数应该尽可能具有一般性。这意味着它们应该：

- 一次只能读、写或擦除一个字符、一个字或一个长字（long word）；
- 能够处理一个非标准的读或写；
- 能够处理跨越芯片边界的读、写或擦除操作。

当编写这些函数时，可能会使用 MTD 帮助函数 flNeedVpp( )，flDontNeedVpp( )和 flWriteProtected( )。这些函数的接口如下：

FLStatus　flNeedVpp（FLSocket vol）

void　flDontNeedVpp（FLSocket vol）

FLBoolean flWriteProtected（FLSocket vol）

如果需要为芯片开启 Vpp 时，可以使用 flNeedVpp( )。该函数启动了一个计数器 FLSocket.VppUsers，然后调用在 FLSocket.VppOn 中引用的函数。在调用 flNeedVpp( )之后，

检查它的返回状态来确认是否成功开启 Vpp。

当处理需要 Vpp 的写操作或擦除操作时，调用 flDontNeedVpp( )使计数器 FLSocket.VppUsers 减 1。计数器 FLSocket.VppUsers 是延迟关闭系统的一部分。当芯片状态为忙时，TrueFFS 保持芯片被持续供电，当芯片状态为闲时，TrueFFS 会关闭电平[③]。

使用 flWriteProtected( )来检查闪存器件是否被写保护。在检查是否允许向板卡编写之前 MTD 的写和擦除函数不能进行任何 flash 编程。如果板卡被写保护，布尔函数 flWriteProtected( )会返回 TRUE，否则会返回 FALSE。

1. 读函数

如果闪存器件能直接被映射到闪存中，从器件中读取数据会变得很简单。TrueFFS 提供了一个默认函数，它执行重映射、简单的内存复制来从指定区域获取数据。然而，如果映射是通过缓冲器来实现的，就必须提供自己的读函数。

2. 写函数

写函数必须在闪存中指定地址写一个特定的块。它的变量包括一个指向闪存器件的指针、要写入的闪存地址和缓冲器长度。最后一个参数是布尔型的，如果设置为 TRUE 则表明在许可写请求之前目标区域不能被擦除。该函数声明为 static 是因为只有 volume 描述符调用该函数。

```
static FLStatus myMTDWrite
     (
    FLFlash vol,
    CardAddress address,
    const void FAR1 *buffer,
    int length,
    FLBoolean overwrite
    )
    {
    /* 写函数 */
    return flOK;
    }
```

写函数要完成以下几点：

- 查看器件是否被写保护；
- 调用 flNeedVpp( )来开启 Vpp。
- 在写操作之前，通常将板卡地址映射到 flashPtr 中。

当完成写函数时，通过缓冲器以一种适当方式重复，如果只允许写入字或双字，请查

[③] 一个 MTD 无需接触 Vcc，TrueFFS 在调用 MTD 函数之前就已经开启了 Vcc。

看缓冲器地址和板卡地址是否匹配。如果不匹配将返回一个错误提示。

正确的算法通常会按照以下的顺序：

- 在板卡地址引入“write setup”指令；
- 将数据复制到该地址；
- 在状态注册器处循环直至状态返回 OK 或直至时间结束。

器件数据层通常会为这种算法提供流图。AMD 器件也需要按照一种 unlock 顺序运行。

写函数负责确认过去写的内容与缓冲器中正在写的内容是否一样。文件 flsystem.h 具有用于此种目的的比较函数。

3. 擦除函数

擦除函数必须一个或多个具有某一尺寸的邻近块（blocks）。这个函数被提供了一个闪存 volume 指针、第一个可擦除块的号码以及可擦除块的数目。源程序如下：

```
Static FLStatus myMTDErase
     (
    FLFlash vol,
    int     firstBlock,
    int     numOfBlocks
    )
    {
    volatile UINT32 * flashPtr;
    int      iBlock;

    if  (flWriteProtected (vol.socket) )
              return flWriteProtected;
    for  (iBlock = firstBlock; iBlock < iBlock + numOfBlocks; Iblock++)
        {
        flashPtr = vol.map  (&vol,iBlock * vol.erasableBlockSize,0) ;

        /* 执行擦除操作 */

        /* 确认擦除成功 */

        /* 如果失败返回 flWriteFault  */
        }
    return flOK;
    }
```

可擦除块的号码作为输入，使用 FLFlash 结构的 erasableBlockSize 成员来将这个号码转换为闪存阵列的偏移量。

8.12.4　定义 MTD 为组件

一旦完成了这个 MTD，就需要将其作为组件加入到系统项目中，传统上 MTD 组件被命名为 INCLUDE_MTD_someName，例如 INCLUDE_MTD_USR。可以通过项目设备来包含 MTD 组件或通过在 Socket 驱动程序文件 sysTffs.c 中定义该 MTD 组件，实现命令行配置和建立来包含这个 MTD 组件。

1. 将 MTD 加入到项目设备中

为了让 MTD 为项目设备所辨认，MTD 的组件描述是必需的。为了使用项目设备将 MTD 组件加入系统中，请编辑 *installDir*\target\config\comps\VxWorks\00tffs.cdf 文件来包含它。MTD 组件使用以下形式定义于该文件中：

```
Component INCLUDE_MTD_type
{
    NAME            name
    SYNOPSIS        type devices
    MODULES         filename.o
    HDR_FILES       tffs/flflash.h tffs/backdrnd.h
    REQUIRES        INCLUDE_TFFS \
                    INCLUDE_TL_type
}
```

一旦在 00tffs.cdf 文件中定义了自己的 MTD 组件，下一次运行 Tornado 的时候就会在项目设备中发现该组件。

2. 在 Socket 驱动程序文件中定义 MTD

对于一个命令行配置和建立，可以在 Socket 驱动程序文件 sysTffs.c 中定义 MTD 来包含该 MTD 组件，如下所示：

```
#define INCLUDE_MTD_USR
```

如条件编译所述，将 MTD 组件加入到那些定义于条件语句之间的 MTD 列表中。然后，为 MTD 定义正确的转换层。如果两个转换层都在 Socket 驱动程序文件中定义，取消那个不使用的转换层定义。其他的例子请参见 *installDir*\target\src\drv\tffs\sockets 中的 *type*-sysTffs.c 文件。

警告：在更改定义之前请确认用户有正确的 sysTffs.c 文件。有关详细信息请参见“8.10.1　传送 Socket 驱动程序存根文件”。

8.12.5 注册识别函数

每一个 MTD 的识别函数都必须在 mtdTable[ ]中定义。每安装一个 volume，TrueFFS 会查找表中是否有与 volume（闪存器件）相应的 MTD。对于每一个为系统定义的组件，TrueFFS 会执行从 mtdTable[ ]中引用的识别函数。定义于 *installDir*/target/src/drv/tffs/tffsConfig.c 中的当前 mtdTable[ ]为：

```
MTDidentifyRoutine mtdTable[] =          /* MTD 表 */
    {
#ifdef  INCLUDE_MTD_I28F016
    i28f016Identify,
#endif                      /* INCLUDE_MTD_I28F016 */

#ifdef  INCLUDE_MTD_I28F008
    i28f008Identify,
#endif                      /* INCLUDE_MTD_I28F008 */

#ifdef  INCLUDE_MTD_AMD
    amdMTDIdentify,
#endif                      /* INCLUDE_MTD_AMD */

#ifdef  INCLUDE_MTD_CDSN
    cdsnIdentify,
#endif                      /* INCLUDE_MTD_CDSN */

#ifdef  INCLUDE_MTD_DOC2
    doc2Identify,
#endif                      /* INCLUDE_MTD_DOC2 */

#ifdef  INCLUDE_MTD_CFISCS
    cfiscsIdentify,
#endif                      /* INCLUDE_MTD_CFISCS */
    };
```

如果写一个新的 MTD，需要将它的识别函数在 mtdTable[ ]中列出。例如：

```
#ifdef INCLUDE_MTD_USR
    usrMTDIdenitfy,
#endif                      /* INCLUDE_MTD_USR */
```

应该用上述的条件包含语句包含组件名。控制上述条件包含的符号常量定义于 BSP config.h 文件中。使用这些常量，终端用户能够有条件地包含特定的 MTD。

将 MTD 识别函数加到这个表中时，也应该将一个新常量加入 BSP 的 config.h 中。

8.13　闪 存 功 能

本节讨论了闪存的功能，以及它保护数据完整、延长媒质寿命、支持发现错误的方法。

8.13.1　块分配以及数据串

为了满足块驱动程序的需要，TrueFFS 将闪存映射到邻近的存储块中，文件系统可以在这些存储块上进行读、写入数据。这些块的标号，最小为 0，最大不超过这些分程序块的总数。现在，TrueFFS 惟一支持的文件系统是 dosFs（请参见“5.2　与 MS-DOS 兼容的文件系统：dosFs 文件系统”）。

1. 程序块分配算法

为了更有效地实现数据检索，TrueFFS 使用了灵活的分配策略，即将相关数据集中到只擦除单元中的相邻区域。这些数据组可以是由文件的多个部分组成的程序块。TrueFFS 集中相关数据遵循优先次序：

（1）首先，在相同的擦除单元中保留一组物理上连续的空程序块。

（2）如果上述步骤失败，就要保证这组程序块全都位于同一个擦除块中。

（3）如果上述步骤失败，最后 TrueFFS 会将这组程序块分配到拥有最多空间的擦除单元中。

2. 集中数据的好处

以这种方式集中相关数据有以下几个优点：

缩短检索时间

在某些情况下，TrueFFS 需要通过一个小的内存窗口访问闪存，集中相关数据会减少将物理块映射到窗口的调用的数目。对于顺序访问的文件，缩短了检索时间。

分段存储最小化

集中相关数据减少了分段存储，这是因为删除一个文件会解放多个能够重新声明的完整的程序块。

加快碎片收集

分段程序的最小化就意味着加快了碎片收集。

静态文件程序块定位

在磨损测平算法决定移动静态区域时，为属于静态文件的程序块定位会使转换这些程序块变得很方便。

8.13.2 读和写操作

闪存不同于其他普通磁介质机械磁盘的特征之一就是它向介质中写入数据的方式。使用传统的磁存储介质，向一个已经写入数据的存储区域中再次写入新的数据会清除现有数据；而闪存则不然。本节描述了闪存是怎样从内存中读取以及向内存中写入数据的。

1. 从程序块中读取

从数据块中读取是直接的。文件系统申请某个块的内容。作为响应，TrueFFS 将这个块的标号转换为闪存中的地址信息，并在其中检索数据并将这些数据返回文件系统。

2. 向空数据块写入数据

如果目标块以前没有被写操作，可以直接向其中写入数据。TrueFFS 将块号码转换为闪存中的相应部分，并向其中写入数据。然而，如果写操作申请改变块中的内容，情况会变得更复杂。

如果任何写操作失败，TrueFFS 会进行第二次写操作。有关详细内容请参见写操作过程中的恢复。

⚠ **注意**：在闪存中存储数据需要使用生产厂商提供的程序设计算法，它定义在 MTD 中。因此，向闪存中写入数据通常是指在闪存中进行程序设计。

3. 向已写块中写入数据

如果写操作申请一个已经包含有数据的闪存区域，TrueFFS 会找到一个不同的、可写的闪存区域代替它，该替代区域已经被清空并能够接受数据。然后 TrueFFS 将新的数据写入那个空区域。在数据被安全写入之后，TrueFFS 会更新它的 block-to-flash 映射结构，使程序块映射为包含已修改数据的区域。这个映射信息甚至在发生故障期间都会保存下来。有关故障恢复以及映射信息的详细内容请参见“恢复映射信息”。

8.13.3 擦除循环和碎片收集

实际上，写操作是与擦除操作相联系的，因为数据不能 over-written。数据必须擦除，然后那些已擦除单元必须在写操作之前被重新声明。本节会描述这个过程以及闪存过度擦除的后果。

1. 擦除单元

一旦数据被写入闪存的一个区域中，对数据块进行修改会使得闪存中相同大小的区域中数据不再有效。这些区域在擦除之前是不可写入的。然而，擦除操作不能在个别字节或数据块进行。擦除限于在更大的区域（擦除单元）中进行。这些擦除单元的大小取决于不同的闪存技术，但通常为 64 KB。

2. 重新声明已擦除块

为了声明那些不再包含有效数据的程序块，TrueFFS 使用了一个称为碎片收集的机制。该机制从源擦除单元中复制所有有效数据，并将其复制到另一个称为转换单元的擦除区域。TrueFFS 然后更新块到闪存（block-to-flash）的映射并向后擦除源（擦除）单元。尽管位于闪存中不同的位置，虚拟块包含有相同的数据。

有关引起碎片收集的算法细节请参见“碎片收集”。有关碎片收集和故障恢复的详细内容请参见“碎片收集过程中的恢复”。

3. 过编程

如果闪存的一个区域不停进行擦除和重新写入操作，它就会进入一个过编程状态，在这种状态下，要花费很长的时间来响应写操作申请。尽管能够自我修复，但闪存的寿命会大大缩短。[④]最终，闪存会先出现零星的擦除错误，不久这种擦除错误会变得越来越频繁，直至无法擦除，从而无法执行写操作。因此，闪存会限制在同一区域擦除和重写入的次数。这个数值称为循环极限（*cycling limit*），这取决于不同的闪存技术，但大致分布在每个擦除块 10 万次到 100 万次[⑤]。

8.13.4　优化方式

如上所述，闪存并不是一个可以使用无限次的存储介质。闪存中每个擦除单元的可擦除次数虽然不少但却是有限的。最终，闪存会进入只读状态。为了尽可能延长闪存的使用寿命，TrueFFS 使用了无用单元收集算法，以及一种称为磨损测评的技术。

1. 磨损测评

有一种缓解“过编程”现象的方法能够平衡整个介质的使用，这样闪存各个部分会平均磨损，不会有一个部分出现过度使用的情况。这种技术成为磨损测评技术，它能够极大地延长闪存的使用寿命。为了完成磨损测评功能，TrueFFS 使用了 block-to-flash 的转换系统，该系统以动态维护的映射为基础。当程序块更改、移动或在收集无用单元时，这种映射都会发生改变。

---

④ 已经存在的数据是可读的。

⑤ 这个数字是近似的，不应该作为一个准确的数字。

2. 静态文件闭锁

当发生改变的 blocks 重映射至闪存中新的位置，会发生某种程度的磨损测评。然而，一些存储在闪存中的数据本质上是静态的，这意味着如果只在修改过程中进行磨损测评，存储静态数据的闪存区域根本不参加闪存中各个部分的循环使用。这种情况称为静态文件闭锁，它加重了闪存中其他区域的磨损程度，结果使得闪存中区域的循环使用变得更加频繁。如果不采取相应的措施，这种情况会极大地缩短闪存媒质的使用寿命。

TrueFFS 通过强制静态区域的转换来克服静态文件闭锁现象。因为 block-to-flash 映射是动态的，TrueFFS 能够以一种文件系统可视的方式来管理这些磨损测评转换。

3. 算法

由于磨损测评所需要的数据移动，实现绝对的磨损测评会对性能产生负面影响。为了避免这种影响，TrueFFS 执行一种并不绝对的磨损测评算法。该算法使各个单元的擦除次数近似相等。因此从长远看，应该在不降低性能的前提下，使各擦除单元具有相同的擦除次数。由于具有可观的可擦除次数，磨损测评这种方法是很好的。

4. Dead Locks（死锁）

最后，TrueFFS 磨损测评算法进一步改进，从而跳出故障模式“*dead locks*”。一些简单的磨损测评算法在很长时间内仅仅在两个或多个单元之间进行触发转换—“flip-flop” 而忽视了其他单元。TrueFFS 使用的磨损测评机制能够停止这样的循环。为了实现良好的磨损测评性能，TrueFFS 至少需要 12 个擦除单元。

5. 无用存储单元收集

正如“重新声明已擦除块”中描述的那样，无用存储单元收集技术为循环使用、对已擦除块进行再次声明。然而，如果这些无用的存储单元收集过于频繁，会使磨损测评算法失效并降低闪存磁盘的整体性能。因此，无用存储单元收集技术使用了一种依靠块分配的算法（“块分配算法”），并且该算法只是在需要时使用。

块分配算法包含了一组空的连续数据块，这组数据块位于同一个擦除单元中。当这组数据（存储块）块变得太小时，块分配算法会启动无用单元收集算法，该算法能够发现并重声明最匹配下列要求的擦除单元：

- 无用单元存储块数目最多；
- 该单元擦写次数最少；
- static 程度最大

除了这些可测量的标准，无用单元收集算法也应用于随机选择过程。这保证了重声明的过程平均地覆盖整个媒质，而不会由于应用程序使用数据的方式而发生变化。

8.13.5　TrueFFS 中的故障恢复

在以下情况中可能会发生故障：

- 在对来自于文件系统的写申请进行响应的时候；
- 在无用存储单元收集过程中；
- 在擦除操作过程中；
- 在格式化过程中。

除了向闪存第一次写入新数据时出现的故障以外，TrueFFS 能够从所有故障中恢复过来。这时，这些新数据会丢失。然而，一旦数据成功写入闪存中，它就能够不受掉电的影响。所有闪存中的数据是可恢复的，而且文件和磁盘的目录结构被保留下来。事实上，电源中断或发生故障的负面结果只是需要重新开始未完成的无用存储单元收集操作。本节描述了 TrueFFS 中“故障发生以及从故障中恢复”。

1. 写操作过程中的恢复

一个写操作或擦除操作会因为硬件问题或掉电而发生故障。如上所述，TrueFFS 使用了“先写后擦除”的算法，该算法规定直至更新操作成功完成以前的数据才会被擦除。为了防止可能的数据丢失，TrueFFS 监视并确认每一个写操作是否成功。为了这个目的，TrueFFS 使用了一个注册器，在注册器中实际写入的数据被重新读取出来，并与用户数据进行比较。如果操作完成，新的 sector 是有效的。如果更新失败，原有的数据也不会丢失或收到某种损坏。

TrueFFS 确认每一个写操作，并在任何失败之后自动尝试二次写入数据到闪存中的不同区域。这样就自动完成了故障恢复从而保证了数据的完整性。这种 write-error 恢复机制在闪存介质快要达到它的使用寿命时显得更加有效。这时闪存中写/擦除故障发生得更加频繁，但是用户能观察到的只是性能大幅度降低（由于重新写入的需要）。

2. 恢复映射信息

TrueFFS 会在闪存中存储关键的映射信息，这样在中断期间（如掉电或闪存介质被拆卸）闪存就不会丢失数据。然而，TrueFFS 使用了基于 RAM 的映射表来跟踪闪存中的内容。当重新接电或重新连接媒质时，这个基于 RAM 的闪存映射表会使用基于闪存的信息进行重建（或确认）。

注意：映射信息可能位于媒质的任何位置。但是，闪存中的每一个擦除单元都保留了一个可预测位置的头信息（header information）。通过在每一个擦除单元中进行仔细地交叉检查，TrueFFS 能够重建或确认闪存映射表的 RAM 复制。

3. 在无用存储单元收集过程中的故障恢复

在发生故障后，任何发生故障时正在进行的收集工作必须重新开始。无用单元区域是被一些已经由主机删除的 sections 占据的空间。TrueFFS 会重新声明无用单元空间，它首先将数据从一个转换单元移至另一个转换单元，然后擦除原来的单元。

如果持续的闪存故障阻碍了写操作移动数据，或者如果不能擦除原始单元，无用单元收集操作会失败。为了尽可能避免写操作的转换部分发生故障，TrueFFS 会格式化闪存介质来保留两个以上的转换单元。这样，即使面向一个转换单元的写操作失败，TrueFFS 会重新尝试向不同的转换单元进行写操作。如果所有的转换单元都失败了，媒质不再接收新的数据，变成了一个只读器件。然而，这不会直接影响用户数据，因为这些数据已经被妥善保存起来。

4. 在格式化过程中的故障恢复

在一些情况下，当闪存被首次格式化时，闪存媒质的 sections 是无法使用的，这是因为这些 sections 是不可擦除的。只要这些坏的单元数目不超过转换单元的数目，媒质从整体上来说是可用的并且能被格式化。惟一值得注意的是格式化后闪存的存储量会减少。

第 9 章　VxDCOM 应用

COM 支持和可选组件 VxDCOM

9.1　简　　介

VxDCOM 是在 VxWorks 上实现 COM 和分布式 COM（DCOM）的技术。VxDCOM 既是这种技术的名称也是该类可选产品的名称。在 VxWorks 标准设备支持基本 COM，VxDCOM 可选产品则增加了 DCOM 功能。

COM 代表组件对象模型（Component Object Model），这是针对基于组件的对象通信的二进制规范。通过 Wind River 的 VxDCOM 技术，样板文件代码能够自动执行，从而使 COM 和 DCOM 组件的创建变得极为方便。因此，VxDCOM 使用户编写实时嵌入系统软件的分步式对象应用变得更简单。

VxDCOM 文档吸收了 COM 技术应用知识，并将重点放在 VxDCOM 设备上，这些设备用于创建在 VxWorks 上执行的服务器应用程序。本章以及《Tornado 用户指南》：构建 COM 和 DCOM 应用程序讨论了相关内容。其中《Tornado 用户指南》中详细介绍了创建 VxDCOM 应用程序的步骤，并介绍了运行 VxDCOM 向导、生成的项目以及建立和扩充 VxDCOM 应用程序的过程。

本章涉及下列内容，包括参考材料和设计问题。

- VxDCOM 技术的概述
- WOTL（Wind 对象模板库 Object Template Library）类
- 自动生成的 WOTL 框架代码
- Wind IDL 编译器和指令行选项
- 定义在自动生成文件中 IDL 的结构和含义
- VxWorks 系统中用于 DCOM 支持的配置参数
- 实时扩展和 OPC 接口
- 编写执行代码的简单介绍和实例
- VxDCOM 和 ATL 的比较

本章中演示了一个包括 Visual Basic 和 C++客户机的 DCOM 服务器应用程序。该示例位于 installDir/host/src/vxdcom/demo/MathDemo 目录中。

9.2 COM 技术概述

COM 是对象（COM 组件）间通信协议的规范。COM 组件是 COM 客户机/服务器应用程序的基本构件。COM 组件通过显示 COM 接口为客户机应用提供服务。总的来看，COM 接口是方式原型的集合，它提供给 COM 客户机一种经过良好定义的，具有连贯性的功能或服务。

9.2.1 COM 组件和软件可重用性

COM 组件使用 CoClasses 类进行实例化。CoClasses 包括一个或多个对 COM 接口的继承。CoClass 执行从这些接口继承的方法。尽管接口本身严格定义了它提供的服务，CoClass 执行代码实现的细节对客户端是完全透明的。实际上客户端意识不到 COM 改变客户端的应用程序，就可以更新或重新使用组件。

下面详细描述了 COM 的基本原理。

1. COM 接口

接口是一个纯方式原型的名称集合。接口名称反映了该方式接口的功能，习惯上以大写字母 I 开头。因此 Imalloc 代表一个分配、释放和管理内存的接口。相似的，Isem 可能是一个封装信号量功能的的接口。

作为 COM 技术的一部分，基本接口服务定义在 COM 库中。与 VxDCOM 在一起的 COM 和 DCOM 库是 VxDCOM 支持的 COM 技术所需要的一些基本接口。

> ⚠ **注意**：这些 COM 和 DCOM 库由 C++模板类库（template class library）使用，并与 VxDCOM 封装在一起。有关 API 的定义请见 comLib.h 和 dcomLib.h。

作为开发者，需要定义具有自己风格的接口。接口定义必须遵循 COM 规范，COM 规范包括应用于接口、接口的使用方式、方式参数和返回类型的描述性属性和规范（见“接口定义”）。只有严格遵守该规范，COM 接口才能成为客户端和服务器之间的一个默认协定，也才能合理地制定通信协议。

> ⚠ **注意**：接口定义是纯原型的，它与只包含纯虚拟方法的抽象 C++类相似。事实上，用来编写 VxDCOM 的对象模板库（WOTL）也是使用相同方法实现的接口。

客户端和服务器之间的约定提供某种服务，但它并不知道如何去实现该服务。实现的细节隐藏于 CoClass 中。

2. CoClasses

CoClasses 是一些声明接口并执行接口方式的类。CoClass 的定义包括了它所有支持的接口声明。CoClass 执行代码必须执行 CoClass 声明的所有接口的全部方式。一旦 CoClass 被实例化，它就成为一个 COM 组件或服务器。客户机向接口请求服务，如果服务器支持这些接口，就会通过接口与客户机应用程序进行通信。

3. 接口指针

在 COM 模型中，指向 COM 接口的指针用于处理 COM 客户机和服务器之间的通信协议。这些接口指针负责在 COM 客户机和 COM 服务器之间传输数据。客户机应用程序使用 COM 接口指针向 COM 服务器特定接口请求服务，使其能访问那些接口并调用那些接口的方式。因为 COM 规范和通信协议是以指针为基础的，所以它们具有二进制标准。

由于 COM 技术使用了二进制标准，理论上，COM 组件能够用任何语言编写并在任意的操作系统上运行，与此同时，COM 组件之间仍然能够进行通信。这样，COM 技术为软件开发者提供了更大的灵活性和组件的可重用性，尤其在将应用程序传送到不同的操作系统时变得更为方便。

⚠ **注意**：近来，在 VxDCOM 下，惟一支持 COM 服务器的语言是 C 和 C++，然而支持 DCOM 服务器只有 C++。

4. VxDCOM 工具

根据 COM 规范，对接口和 CoClasses 的完整定义显得详细而又冗长。然而，由于规范遵守标准规则，VxDCOM 工具能够自动生成代码，从而使定义过程变得很方便。例如，使用 VxDCOM 向导，可以方便地命名接口，然后从预先定义的列表中选择方式参数类型和属性。向导会自动生成正确方式的返回类型、接口和 CoClass 定义。

当建立应用程序时，编译器 widl（Wind Interface Definition Language）生成用于服务器注册和排列（marshaling）的附加代码。这样在开发 COM 应用程序时，只需使用这些工具，编写客户端和服务器的执行代码，并建立项目（project）。

9.2.2　VxDCOM 和实时分布式技术

COM 提供了一个公共框架体描述软件组件的行为以及是否易于访问。扩展基本 COM 技术，即超越进程和机器边界进入分布式对象领域需要一个 RPC 网络协议。为此，DCOM

使用了 RPC（ORPC）对象（一个微软的有关 DCE-RPC 的扩展规范）。ORPC 使用排列好的接口指针作为 COM 组件之间的通信协议，并规定了显示、传送和保存组件接口参考信息的方法。COM 在它的基本接口（定义于标准 COM 库）中提供了这个功能。

VxDCOM 技术支持 VxWorks 的标准设备部分——基本 COM 接口以及 VxDCOM 的可选产品部分——DCOM 网络协议。这些功能的实现记录在 VxDCOM 的 COM 和 DCOM 库中。他们包含了用于嵌入式系统开发和支持 ORPC 的 COM 和 DCOM 相关接口。

⚠ **注意**：有关这些接口的详细内容，请见相应的.idl 文件，如 vxidl.idl。对于 API 文件，请见 comLib.h 和 dcomLib.h。

VxDCOM 支持进程内（in-process）和远程服务器模式。可以在 VxWorks 目标机上编写服务器，该服务器能够为运行于其他 VxWorks 目标机或运行 Windows NT 的 PC 上的客户端应用程序提供服务。DCOM 协议能够使嵌入式智能系统（如长途通讯装置、工业控制器、办公室外围设备）的开发者将 COM 编程环境扩展到局域网，甚至扩展到 Internet，而不必关心网络环境。

⚠ **注意**：VxDCOM 不支持目标机上 VxWorks 到 Windows NT 的连接，但支持从 NT 到 VxWorks 的响应。只要取消所有 NT 安全设置，VxDCOM 就支持 VxWorks 到 NT 连接。详细内容请见《Tornado 用户指南》: 识别服务器以及《Tornado 用户指南》: 注册、扩展并运行应用程序。

9.3 使用 Wind 目标模板库

Wind 目标模板库（WOTL）是一个为编写 VXDCOM 客户端与服务器程序设计的 C++ 模板类库。它与 ATL（Microsoft COM 模板类库）子模板资源兼容，并在 ATL 上模块化。WOTL 是一个可以用来编写客户端和服务器代码的框架。

可以运行 VxDCOM 向导创建 COM 组件。该向导产生输出文件。这些输出文件为头文件和 WOTL 中包含框架代码的执行源文件。使用这个框架代码，可以完成服务器和可选客户端应用的执行细节。

《Tornado 用户指南》: 建立 COM 和 DCOM 应用描述了哪些文件可以用来编写适合应用类型的执行代码。“9.10　编写 VxDCOM 服务器和客户端应用”以及“9.10.3　编写客户端代码”中将进一步讨论细节内容。本章中的代码片断是从 CoMathDemo 例子中摘录的，它位于 installDir/host/src/VxDCOM/demo/MathDemo 目录中。

9.3.1　WOTL 模板类的分类

可以用 WOTL 定义三种 VxDCOM 模板类。它们根据类是否使用 CLSID，以及类在任意时间是否只有一个实例来区分。CLSID 是类的惟一标识符，它允许通过类 factory 对类进行外部实例化。

⚠ **注意**：类 factory 是在 CLSID 基础上对 CoClasses 进行实例化的对象。

三种 WOTL 模板类如下所示：

True CoClasses

存在真 COM 类，即它们已经注册了 CLSIDs 并且通过类 factories 进行实例化。CcomCoClass 模板类用来声明这些类。VxDCOM 向导使用该模板类为 COM 组件的 CoClass 定义生成框架代码。

Singleton classes

除了只有一个实例的 Singleton 类，它们都是真 CoClasses。这样，IClassFactory::CreateInstance( ) 每次调用都会返回相同的实例。为声明这样的类，可以使用类定义中宏 DECLARE_CLASSFACTORY_SINGLETON。

Lightweight classes

技术上，这些类并不是真 COM 类。该模板类没有相关的 CLSID（类标识值），这样，使用一个默认类- factory 创建机制来进行实例化。这些类不是 DCOM 类。为了声明这样的类，需要使用 CcomObject 类。可以使用这些类来创建内部对象，从而提高代码面向对象的能力。

下面将详细讨论如何使用 WOTL 类和类模板。头文件 installDir/target/h/comObjLib.h 中声明了 WOTL 类。一些其他的帮助类，如定义于 installDir/target/h/comObjLibExt.h 的 VxComBSTR，有关详细信息，请见“9.11　比较 VxDCOM 和 ATL 执行”。

9.3.2　CoClass 真模板类

VxDCOM 向导会为 CoClass 自动生成模板类定义，其中包括 CoClass 执行的从基本类或接口引申的定义。下面将解释这种代码。

1. CComObjectRoot – Iunknown 执行支持类

CcomObjectRoot 是 VxDCOM 类中的基本类，能够提供安全任务 Iunknown 实现，还支持可累加（aggregatable）对象。为了声明一个能够为一个或多个 COM 接口提供具体实现的类，需要该种类继承 CcomObjectRoot 和相应接口。下例声明了一个类 Cexample，该类

支持接口 Iexample，并执行该接口的方式。

```
class CExample: public CComObjectRoot,public IExample
    {
        // 私有对象数据
    public:
        // 执行,包括 IExample 方式
    };
```

作为 Iunknown 的执行类，所有声明的 WOTL 执行类都应该包括 CcomObjectRoot 基本类。

2. CComCoClass – CoClass 类模板

CcomCoClass 是实现 CoClass 的模板类，它包括了一个类使用的公开 CLSID 以及一个或多个公开接口。这些类创建的对象是真 COM 对象，这是因为通过使用 CLSID 以及调用 COM 或 DCOM API 的创建函数 CoCreateInstance( )和 CoCreateInstanceEx( )可以对这些对象进行外部实例化。

CcomCoClass 将 class factory 类和注册机制所需的功能封装起来，由 class factory 导出的类继承这些功能。CcomClassFactory 是执行标准 IClassFactory COM 接口的 class factory 类模板。该接口允许在运行时使用 CLSIDs 创建对象。该类的声明遵循标准 WOTL 格式，继承 CcomObjectRoot 和使用的接口。

3. CoClass 定义

对 CoClasses 的定义是由向导自动生成的。这些 CoClasses 由以下各项导出：
WOTL 基本类，CComObjectRoot
所有用于 CoClasses 的 WOTL 基本类模板，CComCoClass
主接口，由 CoClass 实现
所有由 CoClass 实现的附加接口

4. 定义示例

下例展示了头文件 CoMathDemoImpl.h 中的服务器 CoClass 的定义继承。

```
class CoMathDemoImpl
    : public CComObjectRoot,
      public CComCoClass<CoMathDemoImpl,&CLSID_CoMathDemo>
      ,public IMathDemo
      ,public IEvalDemo
    {
        // 定义体
    };
```

为应用程序生成的服务器执行头文件包含 CoClass 的声明，该声明与从 CcomObjectRoot、从 CcomCoClass一主接口、以及从任何附加的接口中导出的 CoClass 类似。请注意，WOTL 支持多种继承，包括多个接口的继承，所以用户的 CoClass 定义来源于其实现的每一个接口。

5. 使用 CLSID 实例化

CcomCoClass 类模板使用 CoClass 和 CLSID 的名称创建类的实例。CoMathDemo 示例代码中的参数&CLSID_CoMathDemo 代表标识 CoClass 的 GUID(全局惟一标识符)。CoClass 的 CLSID 为 CLSID_basename，它是由 widl 编译器生成的，位于文件 basename_i.c 中。这样可以使用参数&CLSID_basename。

⚠ **注意**：*CLSID_basename*（类名称）声明为 const，它代表.idl 文件编辑生成 CoClass 的 GUID。*CLSID_basenam* 用于服务器头文件、执行文件和客户端执行文件。当对 CoClass 使用 CLSID 时，请使用 const。

9.3.3 Lightweight 对象类模板

由于 Lightweight 类在创建 COM 对象时不需要 CLSID，因此它们不是真 CoClasses。Lightweight 类通过使用接口对应用程序的面向对象设计进行内部优化。

CcomObject 是 WOTL 的 lightweight 对象类模板。CcomObject 是封装模板类，用于创建 lightweight 对象的类。已模板化的类 CcomObject，其实参为实际执行类，实际执行类来源于 CcomObjectRoot，它能够继承 Iunknown 接口支持。

在接下来的 CcomObject 模板类实例中，Cexample 是在上面例子中定义的类。它来源于 CcomObjectRoot，并执行 Iexample 接口：

```
CComObject<CExample>
```

Lightweight 类没有 CLSID，这样就不能用通常的 class-factory 方式进行外部实例化。因此，CcomObject 提供了默认的 class-factory 执行方法。这些类调用函数 CComObject<CExample>::CreateInstance( )不使用 CoCreateInstance( )进行实例化。该函数与 IClassFactory::CreateInstance( )有相同的变量。

VxDCOM 向导不会为这种模板类生成定义。为了创建一个 lightweight 类对象，只需把定义和执行代码加入到头文件和源文件中。

9.3.4 单一实例类宏

为了将一个类定义为 singleton，即只含一个实例的类，需要在类定义中声明

DECLARE_CLASSFACTORY_SINGLETON 语句。

```
class CExample
    : public CComObjectRoot,
      public CComCoClass<CExample,&CLSID_Example>,
      public IExample
    {
        // 私有对象数据
    public:
        DECLARE_CLASSFACTORY_SINGLETON( );
        // 执行,包括 IExample
    };
```

当使用该宏声明一个类的时候，每次调用 IClassFactory::CreateInstance( )都返回相同的实例。VxDCOM 向导不会为这种模板类生成定义。为了创建一个单一实例类对象，只需简单地将定义和执行代码加入头文件和源文件中。

9.4 阅读 WOTL-生成的代码

本节的目的在于熟悉向导生成的输出文件中的代码。大多数情况下，执行服务器或客户端程序不需要修改这些代码。然而，WOTL 包含许多提高方便性和可阅读性的帮助宏。本节讨论了它们的定义，它们能够帮助用户更容易的阅读 WOTL 代码，并更好地理解 VxDCOM 是怎样实现 COM 技术的。本节还总结了生成的接口、CoClass 定义以及包含这些接口和定义的文件。

9.4.1 WOTL CoClass 定义

用于服务器 CoClass 的定义由执行头文件 basenameImpl.h 中的向导生成。该头文件包括两个基本的文件：

WOTL 头文件，installDir/target/h/comObjLib.h

widl 生成的接口方式原型头文件，basename.h.

如果必要，根据执行的代码可以包含其他的头文件。例如，CoMathDemoImpl.h 头文件如下所示：

```
/* CoMathDemoImpl.h - 自动生成 COM 类头部 */

#include "comObjLib.h"     // COM-对象模板库
```

```
#include "CoMathDemo.h"    // IDL-输出接口 defs
#include <string>
```

Widl 生成的接口原型头文件 basename.h（如上所述），会声明接口为具有纯虚拟函数（pure virtual functions）的抽象 C++类。然后接口的方式在执行它们的 CoClass 中重定义为虚拟 STDCALL 函数，该函数返回 HRESULT。本质上，CoClass 定义为执行纯抽象基本类（即接口）的虚拟方式。

9.4.2　生成文件中使用的宏定义

有三种自动生成的 WOTL 文件在其定义中使用了宏。这些文件是：

- 由 widl 生成的接口原型头文件，basename.h。
- 由向导生成的服务器 CoClass 头文件，basenameImpl.h。
- 由向导生成的服务器执行文件，basenameImpl.cpp。

在这些文件中使用的宏定义在头文件 installDir\target\h\comLib.h 中。下面是五个定义：

#define STDMETHODCALLTYPE　STDCALL

#define STDMETHODIMP　　HRESULT STDMETHODCALLTYPE

#define STDMETHODIMP_（type）　type STDMETHODCALLTYPE

#define STDMETHOD（method）　virtual HRESULT STDMETHODCALLTYPE method

#define STDMETHOD_（type，method）　virtual type STDMETHODCALLTYPE method

这些宏将接口方式的定义映射到.idl 文件、头文件和执行文件中。本节将在下面讨论该过程的细节，表 9-1 列出了阅读文件时这些映射的主要参考信息。

表 9-1　映射到文件的接口方式定义

文　件	含　义	生成代码使用的工具	方式原型格式
basename.idl	IDL 接口定义	wizard	HRESULT routine_name（参数）
basename.h	C++ 接口方式原型	widl	虚拟 HRESULT STDMETHODCALLTYPE routine_name（参数）= 0
basenameImpl.h	C++ CoClass 方式原型	wizard	STDMETHOD（方式）（参数）
basenameImpl.cpp	C++ CoClass 方式定义	wizard	STDMETHODIMP coclass :: routine_name（params）{ // 执行代码}

⚠ **注意**：如果需要人工编辑 C++头文件 *basename*Impl.h，可以把 CoMathDemo 文件或其他向导生成文件作为例子使用。

1. 将 IDL 定义映射到接口头文件原型

使用这些宏来映射接口方式原型，使.idl 文件中的每一个接口定义为虚拟 STDCALL 方式，并在生成的接口头文件 basename.h 中返回 HRESULT。下面在.idl 文件中进行语法定义：

```
在 basename.h 头文件中
HRESULT routine_name (parameters) ;
变成:
virtual HRESULT STDMETHODCALLTYPE routine_name (parameters) = 0;
```

2. 将接口原型映射到 CoClass 方式定义

进一步，在 comBase.h 中宏 STDMETHODCALLTYPE 被定义为 STDMETHOD:

#define STDMETHOD（method） virtual HRESULT STDMETHODCALLTYPE method

所以，对于向导中定义的每个接口，basenameImpl.h 文件中的最终原型是：

STDMETHOD（method）（parameters）;

3. 在执行文件中定义 CoClass 方式

服务器执行文件使用宏 STDMETHODIMP 表示一个 STDCALL 方式，该方式返回 HRESULT。因此，在生成执行文件中该方式的定义与 CoMathDemoImpl.cpp 文件中一样，具有如下形式：

STDMETHODIMP coclass :: routine_name（params）{ // implementation code }

9.4.3 接口映射

WOTL 在类定义中使用了 COM_MAP 类型的接口映射。这些宏与在 ATL 中使用的宏类似，并且是 Iunknown 方式 QueryInterface( )执行的一部分。COM_MAP 宏的 WOTL 库定义位于 WOTL 头文件 comObjLib.h 中。

COM_MAP 宏定义了一个名为 qi_impl( )的函数，该函数在运行过程中获取所需的接口指针。除此之外，CoClass 中 COM_MAP 的 layout 与 ATL 映射中的 COM_MAP 相同。然而，COM_MAP 只支持 COM_INTERFACE_ENTRY 和 COM_INTERFACE_ENTRY_IID 入口函数。

CoMathDemoImpl.h 头文件在方式的公共定义末端包含一个接口映射。CoClass 执行的两个接口都在接口映射中定义。

```
// COM 接口映射
BEGIN_COM_MAP (CoMathDemoImpl)
    COM_INTERFACE_ENTRY (IMathDemo)
```

```
    COM_INTERFACE_ENTRY (IEvalDemo)
END_COM_MAP( )
```

对于 CoClass 接口，生成的 CoClass 头文件有相同的接口映射定义[①]。

9.5　配置 DCOM 性能参数

可以配置 DCOM 性能参数来增加 DCOM 支持。下面将描述这些参数的意义和用途，以及每个参数的可能值和默认设置的范围。

有关如何改变这些参数，请见《Tornado 用户指南》:建立 COM 和 DCOM 客户端和服务器应用。

VXDCOM_AUTHN_LEVEL .

规定了应用所需的认证级别。

Values

0 =默认值，无需 authentication。

1 =无需 authentication

2 = RPC_CN_AUTHN_LEVEL_CONNECT，意味着应用必须创建一个用户 ID 和密码（使用 vxdcomUserAdd( ) API）使未来的连接能够被识别。

Default

0

VXDCOM_BSTR_POLICY .

设置 BSTRs 配置。

Values

TRUE = BSTRs 配置为计数串（字节数组），第一个字节规定了串的长度，其余的字节由 ASCII 字符表示。

FALSE = BSTRs 作为 BSTRs（即宽字符串）进行配置，这是双字节 unicode 字符串。

Default

FALSE

VXDCOM_CLIENT_PRIORITY_PROPAGATION .

在基本 DCOM 功能中增加优先级方案和实时扩展优先级配置。

Values

[①] 默认的 Iunknown 通常是表中的第一个 COM_INTERFACE_ENTRY。因此，通过定义，该条目必须由 Iunknown 导出，否则 static_cast( )就会失败。

TRUE =传送客户端优先级，使服务器能够与客户机以相同的优先级运行。

FALSE =服务器以自己的优先级运行，与客户机的优先级无关。

Default

TRUE

详细内容，请见“9.8.2 在 windows 上配置客户端优先级传送”。

VXDCOM_DYNAMIC_THREADS .

规定了在时间高峰期能够分配的附加线程的数目。

Values

范围是 0..32

Default

30

VXDCOM_OBJECT_EXPORTER_PORT_NUMBER .

在重启系统，设置参数从而配置 ObjectExporter 端口数目。如果这个参数被设置为 0，将动态分配端口数目。

Values

Range is 1025..65535（short int values）

Default

65000

VXDCOM_SCM_STACK_SIZE .

在建立时，规定了服务控制管理（SCM）任务的堆栈尺寸。该参数主要用于配置 PPC60x 目标机结构，这样能够创建一个比其他标准结构体更大的堆栈框架。

在大多数结构体中能够使用默认值。

Default

30K

如果在 Tscm 任务中发现了堆栈异常或数据异常，可以使用浏览器检查堆栈空间是否占满，如果占满，需要增加这个数值。

VXDCOM_STACK_SIZE .

规定了服务器线程集合中线程的堆栈尺寸。

Values

推荐范围是 16K...128K

Default

16K

如果在 tCOM 任务中发现了数据异常或堆栈异常，可以使用浏览器检查堆栈空间是否

占满，如果占满，需要增加这个数值。

VXDCOM_STATIC_THREADS .

规定服务器线程集合中预先分配的线程数目。

Values

范围是 1..32

Default

5

VxDCOM 启动了一组已初始化的线程来加速 CoClass 的启动时间。通过将该值设置为系统运行的 CoClasses 平均数来加速 CoClass 的初始化。

VXDCOM_THREAD_PRIORITY .

规定了服务器线程集合中的线程默认优先级。详细内容，请见“9.8.3 使用线程集合 Threadpools”。

Values

与 VxWorks 任务优先级的范围相同。

Default

150

9.6　使用 Wind IDL 编译器

Wind IDL 编译器 widl 是一个命令行工具，它集成在 Tornado IDE 中。这样，建立一个 DCOM 项目时，widl 将自动运行。

9.6.1　命令行格式

可以从命令行中运行 widl，但是，如果.idl 是项目的一部分，这样做就会显得多余。运行 widl 的命令行格式如下：

widl [-IincDir] [-noh] [-nops] [-dep] [-o outPutDir] idlFile[.idl]

IdlFile 必须具有.idl 扩展名或无扩展名的形式。其他命令行转换是可选的，列于下方：

-I incDir

将指定的包含目录加入头文件查找路径中

-noh

不生成头文件

-nops

不生成代理/存根文件

-dep

生成 GNU-类型从属文件

-o outputDir

从 widl 中将输出写入到指定的输出目录中

9.6.2 已生成代码

widl 编译由向导生成的.idl 文件时，会生成附加代码，这些代码被加入由向导生成的下列三个空文件中：basename_i.c，basename.h，和 basename_ps.cpp。

basename.tlb（DCOM only）

类型库，用于 Windows 注册表中注册服务（在运行 nmakefile 建立 Windows DCOM 客户端程序时该文件由 MIDL 生成的。只有 DCOM 项目需要使用该文件）

basename_ps.cpp（DCOM only）

代理/存根代码，用于客户机和服务器间接口方式参数的配置。

basename.h

接口原型，它以 VxDCOM 向导生成的.idl 文件中的接口定义为基础。

basename_i.c

GUIDs（unique identification numbers）与.idl 文件元素的结合体（例如，每个接口都有一个相关 IID 或接口标示符，每个 CoClass 都有惟一的 CLSID 或类标示符等）。

警告： widl 工具只能为定义在已编译的 IDL 文件中的接口生成 proxy/stub 代码，而不能为定义在输入 IDL 文件中的接口服务。如果要为那些接口生成 proxy/stub 代码，就必须对那些定义每一个接口的 IDL 文件独立进行编译。*installDir*\target\h\vxidl.idl 中的 IDL 文件则是个例外，其 proxy/stub 代码程序在 DCOM 支持组件中。

9.6.3 数据类型

Widl 工具能够编译自动数据类型和非自动数据类型。widl 用 VxDCOM 对这两类数据类型编辑的详细内容如下：

1. 自动数据类型

自动数据类型指下列描述的简单类型，接口指针，VARIANT 和 BSTR。

long，long *，int，int *，short，short *，char，char *，float，float *，double，double *

整型值必须明确地声明为上面的一种类型或一种指针类型。

IUnknown *，IUnknown **

包括接口指针类型。

BSTR，BSTR *

BSTR 为宽字符，即 16 位无符号整型，用于创建字符串。因为 COM API 以宽字符串为参数，所以宽字符串被作为 BSTR 进行传送。可以在 DCOM 配置选项参数中修改读取或传送 BSTR 的方式。

如果需要在标准 8 位 ASCII 类型和宽字符串之间进行转化，可以使用以下函数：

comWideToAscii( )

将一个宽字符串转换为 ASCII

comAsciiToWide( )

将 ASCII 转换为宽字符串

⚠ **注意**：VxDCOM 也支持 ATL 宏，OLE2T 和 T2OLE。然而，这些宏调用 alloca( ) 会占用当前堆栈的内存，即这些宏若使用一个内循环将会消耗整个堆栈的空间。建议使用定义于 comObjLibExt.h 中的 VxComBSTR 类。详细内容请见“9.11.7　VxComBSTR”。

VARIANT，VARIANT *

包含枚举 VARTYPE 的所有值，这些值是有效的并且由 VARIANT 中的 VxDCOM 所支持，包括 SAFEARRAY。SAFEARRAY 支持的详细内容，请见“具有 VARIANTS 的 SAFEARRAY”。

2. 非自动数据类型

Widl 能够正确编译的非自动数据类型有：

simple array

具有固定编译时间常数大小的数组

fixed-size structure

这是一个结构，它的成员或是自动数据类型、简单数组，或是其他固定尺寸的结构体。

conformant array

这是一个数组，使用 size_is 属性时，其大小在运行时由其他参数、结构域（field）或表达式决定。

conformant structure

该结构大小可变，其最终成员是由另一个结构成员决定的构造数组（conformant array）。

string
一个指向有效字符类型（单字节或双字节类型）的指针，具有[string]属性。

 **警告：** widl 不支持 union 类型。

如果使用了一个非自动数据类型，那么需要建立一个代理 DLL。可以使用 midl 生成 DLL，然后建立 DLL。有关注册代理 DLL 的详细内容请见《Tornado 用户指南》：在 Windows 中注册代理 DLL。

如果因为使用 OPC 接口而采用非自动数据类型，那么需要将 DCOM_OPC 支持组件加到系统中，并在 windows 上安装 OPC 代理 DLL。详细内容，请见《Tornado 用户指南》：OPC 函数支持。

3. 具有 VARIANTS 的 SAFEARRAY

只有在 VARIANT 中进行配置，VxDCOM 才会支持 SAFEARRAY，否则不支持 SAFEARRAY。VxDCOM 既支持基于 SAFEARRAY 的 COM 技术应用，也能够将 SAFEARRAY 从 DCOM 服务器配置到 DCOM 客户端。当应用程序使用 SAFEARRAY 时，请注意以下有关对 COM 和 DCOM 应用的特殊支持和限制的章节。

4. COM 支持

在 COM 下使用 SAFEARRAY 时，具有以下 SAFEARRAY 支持：
对下列类型的多维 SAFEARRAY 支持
VT_UI1
VT_I2
VT_I4
VT_R4
VT_R8
VT_ERROR
VT_CY
VT_DATE
VT_BOOL
VT_UNKNOWN
VT_VARIANT
VT_BSTR
对 VARIANT 的 SAFEARRAY 支持，其中包括 VARIANT 能够包含 SAFEARRAY
支持最小程度的 SAFEARRAY API。

警告： 由 SafeArrayAccessData( )返回的内存结构无法保证与 Microsoft API 返回的内存结构完全一样。

5. DCOM 支持

在 DCOM 下使用 SAFEARRAY 时，下面是一组 SAFEARRAY 支持的特征和限制。

对以下类型的单维 SAFEARRAY 支持。

VT_UI1

VT_I2

VT_I4

VT_R4

VT_R8

VT_ERROR

VT_CY

VT_DATE

VT_BOOL

VT_BSTR

VxDCOM 只支持不大于 16K 的 SAFEARRAY。

由于 Microsoft DCOM 的 fragment 包长度超过 4 KB，在从 Microsoft 平台传送 SAFEARRAY 时，其大小不应超过 4 KB。

6. HRESULT 返回值

所有的接口方式都需要一个 HRESULT 返回类型。HRESULT 是所有 ORPC 方式返回的 32 位数值，它用于处理 RPC 异常。通过使用 ORPC 和 HRESULT 返回类型，从开发者的角度看，COM 技术能够提供一个虚拟透明的目标通信过程。这是因为，只要客户端代码返回一个 HRESULT，无论是进程内还是远程访问。客户端应用程序以相同的透明方式访问所有的对象，事实上，进程内服务器与客户端应用程序通过调用 C++虚拟方式进行通信，而远程服务器与客户端应用程序则是通过使用 proxy/stub 代码和调用当地 RTC 服务器进行通信。然而，由于它在 COM 机制中是自发的，所以这个过程对程序员来说是透明的。惟一区别是远程函数调用要消耗更多的内存[②]。

VxDCOM 向导为你定义的所有方式生成一个 HRESULT 返回类型。当编写执行代码时，可以查阅头文件 installDir\target\h\comErr.h 获取 HRESULT 错误常数的全面列表。表 9-2 列出了最常见的 HRESULT 数值。

[②] 由于不需要支持跨进程（cross-process）的和本地 RPC 的服务器模型，VxDCOM 只支持进程内部的和远程的服务器模型。

表 9-2 常见 HRESULT 数值

数 值	含 义
S_OK	成功（0x00000000）
E_OUTOFMEMORY	（0x80000002） 没有足够的内存完成调用
E_INVALIDARG	（0x80000003） 一个或多个变量无效
E_NOINTERFACE	（0x80000004） 没有支持的接口
E_ACCESSDENIED	（0x80070005） 一个受保护的操作由于不满足安全权限而失败
E_UNEXPECTED	（0x8000FFFF） 未知的，但相对严重的错误
S_FALSE	（0x00000001）故障

9.7 阅读 IDL 文件

IDL（接口定义语言）是 COM 用来定义元素（接口、方式、类型库、CoClasses 等）的规范。包含这些定义的文件用 IDL 编写，并且习惯上用.idl 扩展名标示。VxDCOM 为工具能够正确地工作而采用这些扩展名。因此，向导生成的接口定义文件具有这种扩展名。

9.7.1 IDL 文件结构

.idl 文件结构包括接口，CoClass 和类型库定义，其中的每一个都包含声明部分和实现部分。CoClass 定义本身是库实现的一部分。这种结构是标准的，尽管也会遇到一些与之稍有不同的.idl 文件。

以下是一个具有多个接口定义的.idl 文件典型格式，其中库定义包含 CoClass 定义，而且 CoClass 定义包含了它所执行的接口。

```
//加入 directives，typedefs，constant 声明及其他定义
[attributes] interface interfacename: base-interface {definitions};
[attributes] interface interfacename: base-interface {definitions};
//附加接口定义.
[attributes] library libname {definitions};
```

下面例子中的.idl 文件来自 CoMathDemo 函数（在 installDir/host/src/VxDCOM/MathDemo 目录中），它显示了.idl 文件的结构 该文件定义了两个接口 IMathDemo 和

IevalDemo，一个类型库 CoMathDemoLib 和一个执行以上两个接口的 CoClass—CoMathDemo。

```
#ifdef _WIN32
import "unknwn.idl";
#else
import "vxidl.idl";
#endif

[
    object,
    oleautomation,
    uuid (A972BFBE-B4A9-11D3-80B6-00C04FA12C4A) ,
    pointer_default (unique)
]
interface IMathDemo:IUnknown
    {
    HRESULT pi ([out,retval]double* value) ;
    HRESULT acos ([in]double x,[out,retval]double* value) ;
    HRESULT asin ([in]double x,[out,retval]double* value) ;
    HRESULT atan ([in]double x,[out,retval]double* value) ;
    HRESULT cos ([in]double x,[out,retval]double* value) ;
    HRESULT cosh ([in]double x,[out,retval]double* value) ;
    HRESULT exp ([in]double x,[out,retval]double* value) ;
    HRESULT fabs ([in]double x,[out,retval]double* value) ;
    HRESULT floor ([in]double x,[out,retval]double* value) ;
    HRESULT fmod ([in]double x,[in]double y,[out,retval]double* value) ;
    HRESULT log ([in]double x,[out,retval]double* value) ;
    HRESULT log10 ([in]double x,[out,retval]double* value) ;
    HRESULT pow ([in]double x,[in]double y,[out,retval]double* value) ;
    HRESULT sin ([in]double x,[out,retval]double* value) ;
    HRESULT sincos ([in]double x,
                    [out]double* sinValue,
                    [out]double* cosValue) ;
    HRESULT sinh ([in]double x,[out,retval]double* value) ;
    HRESULT sqrt ([in]double x,[out,retval]double* value) ;
    HRESULT tan ([in]double x,[out,retval]double* value) ;
    HRESULT tanh ([in]double x,[out,retval]double* value) ;
    };
[
    object,
    oleautomation,
```

```
    uuid (4866C2E0-B6E0-11D3-80B7-00C04FA12C4A),
    pointer_default (unique)
]
interface IEvalDemo:IUnknown
    {
    HRESULT eval ([in]BSTR str,[out,retval]double* value);
    HRESULT evalSubst ([in]BSTR str,
                       [in]double x,
                       [out,retval] double* value);
    };

[
    uuid (A972BFC0-B4A9-11D3-80B6-00C04FA12C4A),
    version (1.0),
    helpstring ("CoMathDemo Type Library")
]

library CoMathDemoLib
    {
    importlib ("stdole32.tlb");
    importlib ("stdole2.tlb");

    [
        uuid (A972BFBF-B4A9-11D3-80B6-00C04FA12C4A),
        helpstring ("CoMathDemo Class")
    ]
    coclass CoMathDemo
        {
        [default] interface IEvalDemo;
        interface IMathDemo;
        };
};
```

下面将讨论.idl 文件各部分的格式。

1. 输入提示

Import 提示出现在接口定义之前，它规定了另一个.idl 文件或头文件。这些输入的文件包含了一些定义（如 typedefs，常数声明和接口定义），它们可以在输入的 IDL 文件中被引用。所有的接口继承了 Iunknown 接口。因此，所有的接口定义将有条件地包含 WIN32 头文件 unknwn.idl，或 vxidl.idl，它们在目标机上运行的时候会在 VxDCOM 下执行 Iunknown。

2. 接口定义

.idl 文件中的接口定义规定了客户端应用于服务器对象之间的实际约定。它描述了一个接口 header 和一个接口 body 中的每一个接口的特征。接口定义的格式如下：

```
[attributes] interface interfacename: base-interface {definitions};
```

3. 接口 Header

接口 header 是接口定义的开始部分。接口 header 由方括号中的信息和跟有接口名称的关键词 interface 组成。方括号中的信息是描述接口整体特征的属性列表。该信息对于整个接口来说是全局的（相对应用于接口方式的属性而言）。描述接口- [object]，[oleautomation]，[uuid]和[pointer-default] –的属性是 VxDCOM 接口定义的标准属性，将在“9.7.2　定义属性”中介绍。

4. 接口 Body

接口 body 是接口定义中由{ }括号括起来的部分。该部分包含了远程数据类型和接口的方式原型。它能够有选择地不包含或包含多个输入列表，编译指示，常数声明，一般声明和函数声明。

5. 库和 CoClass 定义

类型库和 CoClass 定义遵循与接口定义相同的格式。库定义格式为：

[attributes] library libname {definitions};

在库名称之前有属性描述，其后跟着一组由{ }括住的定义。关键词 library 表明编译器能够生成类型库。类型库包括库 block 中每一个元素的定义，和在外部定义的每个元素的定义。CoClass 定义包含在{definitions}中，或库 block 的 body 部分；并且 CoClass 有自己的 header 定义和 body 定义。

[version]属性是接口的一个特殊版本。[version]属性确保客户端与服务器端软件兼容。

[helpstring]属性规定了一个以零为终止符的字符串，该字符串包含帮助文本，对本例而言，它描述应用于类型库的元素。[helpstring]属性能用于—类型库、输入库、接口、模块—或 CoClass 语句、typedefs、性质和方式。

CoClass 语句用于定义 CoClass 及其支持的接口。CoClass 定义与接口定义相似。它由一组属性组成，包括[uuid]属性（代表 CLSID），CoClass 关键词，CoClass 名称和一个定义体。CoClass 执行的所有接口都列在 CoClass 的 body 中，这样就必须将 CoClass 执行的附加接口加到那个列表中。[③]也需要将那些接口加到服务器执行头文件中的 CoClass 定义中，如“9.10　编写 VxDCOM 服务器和客户端应用”所述。

[③] 这个列表规定了一整套 CoClass 执行的接口，包括输入和输出的。

9.7.2 定义属性

接口属性规定了接口某种特性。接口属性收集在括号中的以逗号为间隔的列表里。属性列表通常位于列表中属性所描述的对象之前。属性也能应用于库和 CoClasses。然而，某些属性或属性的联合体对于某一种定义是有效的，但对于另一种定义却未必如此（请见“VxDCOM 的属性限制以及库和 CoClass 定义”）。有关属性的详细内容请见《Microsoft COM 规范》。

1. IDL 文件属性

当运行 D/COM 应用向导时，生成的接口定义会默认包含下列属性。如果想对它们进行修改，就必须遵守语言的约束（请见“VxDCOM 属性限制”）。

[object]

[object]属性是一个指定 COM 接口（而不是 RPC 接口）的 IDL 扩展名。这个属性告诉 IDL 编译器特别为 COM 接口生成所有的 proxy/stub 代码，并为每一个在.idl 文件中定义的库 block 生成一个类型库（请看库和 CoClass 定义）。所有的 VxDCOM 接口定义都会在定义中注明这个属性。

[oleautomation]

[oleautomation]属性表明接口是自动操作的。向导生成的 VxDCOM 接口定义由属性进行声明，属性规定了参数和返回类型是自动类型。

[pointer_default]

[pointer_default]属性为几乎所有指针规定了默认指针属性，但被用作 top-level 参数（如用作函数参数的指针）的指针例外，这些指针被自动指定为 ref 指针。[pointer_default]属性的格式如下：

pointer_default（ptr | ref | unique）

这里 ptr，ref，或 unique 可以被规定为:

ptr

将指针指定为完全指针，具有 C 语言指针的全部功能，包括 aliasing。

ref

指定一个引用指针，能够提供数据的地址。引用指针不能为 NULL。

unique

允许指针为 NULL，但不支持 aliasing。

[pointer_default]属性能够应用于函数返回的指针。对于作为 top-level 参数的指针（如用作函数参数的个别指针），必须提供适当的指针属性。例如：

HRESULT InterfaceMethod（ [unique] VXTYPE* ptrVXTYPE ）;

在这个例子中，指针属性会取代出现在接口头部的默认属性[pointer_default]。

[pointer_default]属性在.idl 文件中是可选的，并且只有当函数返回一个未定义指针类型或当函数包含一个具有多个星号的参数（*）时，该属性才是必需的。

[uuid]

[uuid]接口属性指定了一个单一通用标示符（UUID），它被分配给接口，并将该接口与其他接口区分开来。COM 技术依赖这些单一值作为识别组件和接口、以及 COM 系统中子对象 sub-objects（如接口指针和类型库）的方式。做为.idl 文件的一部分，向导为每一个接口、类型库和 CoClass 定义生成一个 UUID。COM 接口——即对于以[object]接口属性为标示的接口——需要[uuid]属性来决定客户端是否能与服务器连接在一起。它能够区分公共接口的不同版本，使不同的开发商能够引入新的特性而不必冒兼容性的风险。

⚠ **注意**：一个 UUID 指定了一个 128-bit 的数值，并且该数值是惟一确定的。实际的数值代表了 GUID、CLSID 或 IID。

2. VxDCOM 的属性限制

为 VxDCOM 接口定义自动生成的惟一接口属性有[object]，[oleautomation]，[pointer-default]和[uuid]。使用这些接口属性的限制如下：

VxDCOM 不支持所有的接口类型；这样，任何 VxDCOM 不支持的接口类型所定义的属性都将无法使用。例如，由于 VxDCOM 不支持 Idispatch，[dual]属性就无法使用。

一些属性是被禁止的，例如，[version]属性不能用于 COM 接口。[version]属性能够在 RPC 接口的多个版本中确定一个版本。MIDL 编译器不支持 COM 接口的多个版本，所以[object]属性（该属性会指定一个接口作为 COM 接口）也不能包含[version]属性。

对于一整列接口属性，它们的含义和有效的结合，请见 Microsoft 文档记录。

⚠ **注意**：如果要创建现有 COM 接口的新版本，可以使用接口继承。引入的 COM 接口具有不同的 UUID，但继承了接口成员函数，状态代码和基本接口的接口属性。

3. 接口方式参数的定向属性

接口方式参数的定向属性表明了数据在各种方式之间传输的方向。两个基本的定向属性是[in]和[out]。以下是参数属性的四种组合：

[in]

[in]属性规定参数由调用者向被调用者传输，也就是说，数据只能由调用者发送，例如客户端（caller）到服务器（callee）。

[out]

[out]属性规定了参数是由被调用者向调用者传输，即从服务器到客户端。由于数据通过[out]参数返回，通常它们是指针类型。

[in，out]

[in，out]能够对一个参数使用[in]和 [out]两种属性。如果要在一个参数上使用两种定向属性，那么就要在两个方向上对参数进行复制，这样，发送数据进行修改并将其传送返回调用者（caller）的过程是通过指向数据目标位置的指针来完成的。参数用这种属性组合来定义会增加调用的内存开销。

[out，retval]

与[out] 属性组合的[retval]属性标示参数值为函数的返回值。这个属性必要的，因为所有 COM 接口方式的返回值必须是 HRESULT。这样，对于一些语言的应用如 Visual Basic，需要提供附加返回值。有关 HRESULT 返回值的详细内容请见“HRESULT 返回值.”。

9.8 增加实时扩展

对于 DCOM 服务器应用，VxDCOM 提供了实时扩展，从而改善应用程序的性能。实时扩展是优先级方案和线程集。我们将在下面进行讨论。

9.8.1 使用 VxWorks 上的优先级方案

优先级方案用来控制在 VxWorks 上运行的服务器对象的调度。通过设置宏 AUTOREGISTER_COCLASS 的参数可以在类注册时选择优先级方案。宏的第二和第三个变量规定了优先级方案的细节。由 MathDemo 的向导自动生成的 AUTOREGISTER_COCLASS 宏的例子如下所示：

```
AUTOREGISTER_COCLASS (CoMathDemoImpl, PS_DEFAULT, 0) ;
```

在服务器执行文件中生成的定义看上去与反映服务器名的第一个变量相似。

1. 第二个参数优先级方案

AUTOREGISTER_COCLASS 宏的第二个参数规定了优先级方案。这个变量的三个可能的数值为：

PS_SVR_ASSIGNED

该数值表明服务器通常会以一个给定的优先级运行。该优先级在对象注册的时候分配给每一个对象。例如 AUTOREGISTER_COCLASS 宏的一个参数。

PS_CLNT_PROPAGATED

这个数值表明，用于远程调用的工作线程应该以发出申请的客户端优先级运行。这与在同一个节点上同一个线程调用操作的过程相似。

PS_DEFAULT

该数值表明，当即将到来的请求不包含传播优先级时，可以指定一个默认优先级，并

将其应用于工作线程。在 PS_CLNT_PROPAGATED 情况下，当 VXDCOM_ORPC_EXTENT 不存在时，-1 值表明应该使用默认的服务器优先级（在项目设备中配置）。

2. 第三个参数优先级

第三个变量规定了分配给服务器（例如 PS_SVR_ASSIGNED 方案）或客户端（PS_CLNT_PROPAGATED 方案）的优先级。另外，当 VXDCOM_ORPC_EXTENT（其中包含有优先级）不在申请中时，它会指定一个优先级。

如果客户端是一个 VxWorks 目标机，客户端的优先级通过 RPC 调用传输到服务器，该过程使用了 DCOM 协议定义规定的 ORPC 扩展机制。这样，服务器与客户端任务在相同的优先级上执行方式。

9.8.2　在 Windows 上配置客户端优先级传送

当配置 VxDCOM 项目时，客户端优先级被自动传送，而且这个过程是透明的。客户端优先级传送可以被关闭（见“9.5　配置 DCOM 性能参数”），这样，每个申请会节省一些字节。

使用 Windows，你必须创建一个 IchannelHook 接口来传输优先级，即向即将发出的申请增加一个优先级(以 ORPC_EVENT 的形式)。向其他的操作系统传输一个适当的 VxWorks 优先级仍然是用户的责任。只有 ClientGetSize( )和 ClientFillBuffer( )函数必须执行。其他的 IchannelHook 接口方式 ClientNotify( )，ServerNotify( )，ServerGetSize( )和 ServerFillBuffer( )可以是空函数。下面的例子中，代码执行 ClientGetSize( )和 ClientFillBuffer( )方式：

```
void CChannelHook::ClientGetSize
    (REFGUID uExtent,REFIID riid,ULONG* pDataSize)
    {
    if (uExtent == GUID_VXDCOM_EXTENT)
        *pDataSize = sizeof (VXDCOMEXTENT) ;
    }

void CChannelHook::ClientFillBuffer
    (REFGUID uExtent,REFIID riid,ULONG* pDataSize,void* pDataBuffer)
    {
    if (uExtent == GUID_VXDCOM_EXTENT)
        {
        VXDCOMEXTENT *data = (VXDCOMEXTENT*) pDataBuffer;
        getLocalPriority ( (int *) & (data->priority) ) ;
        *pDataSize = sizeof (VXDCOMEXTENT) ;
        }
    }
```

GUID_VXDCOM_EXTENT 和 VXDCOMEXTENT 被定义在 installDir\target\h\dcomLib.h 中。希望传输优先级的 Windows 编程员必须#include 这个文件。一旦 执行 hook 函数，它就必须使用 CoRegisterChannelHook( )函数在 Windows COM 运行时间进行注册。

9.8.3 使用线程集合 Threadpools

线程集合是 VxWorks 运行时间库具有的一组任务，为了执行目标方式，这组任务会处理 DCOM 申请。因为少量的任务能够处理大量的申请，所以线程集合优化了系统的性能并增加了系统的可裁减性[④]。

如果集合中所有的任务都处于忙状态，新的任务将动态地加入到集合中来处理那些无暇应接的行为。这些任务会在工作负荷减少时又被系统重新利用。

如果所有的线程集中的静态任务处于忙状态，而且动态任务的数量已经达到最大，那么一个队列会存储在一个未服务的申请中。当任务变得空闲时，它们就会处理队列中的申请。如果队列已满，调用任务会返回一个警告信息。

因为线程集合是内核的一部分，所以线程集合参数是可配置的。

9.9 使用 OPC 接口

VxDCOM 支持定义于以下文件中的 OPC 接口，这些 OPC 接口使用相应的类属名作为标识。

installDir\target\h\opccomn.idl	常用接口
installDir\target\h\opcda.idl	访问数据
installDir\target\h\opc_ae.idl	Alarms and Events

一些 OPC 接口服务使用了数组和匿名结构数据类型。为了支持这些类型这些文件用条件标记符进行了少量的改动。

如果你应用了 OPC 接口，你就需要使用 non-automation 的数据类型，他们需要在.idl 文件中进行额外的编辑。详细内容，请见《Tornado 用户指南》：建立 COM 和 DCOM 应用。

OPC 文件的 VxDCOM 版本会传送 installDir\target\h\vxidl.idl 文件。该文件定义了一个 OPC 应用所需的接口。对于 OPC 协议所需的每一个接口而言，VxDCOM 需要一个相应的 proxy/stub 接口进行远程访问。为了使用这种 proxy/stub 代码，必须增加 DCOM_OPC 支持组件，请见《Tornado 用户指南》：建立 COM 和 DCOM 应用。

---

④ 这与调用中心相似，其中一组固定数目的操作符为即将到来的调用服务。事实上，假如给定平均调用频率和长度，Erlang 计算器能够优化操作符的合理数目。

9.10　编写 VxDCOM 服务器和客户端应用

本节讨论了编程问题并提供了样本代码用于编写 VxDCOM 服务器和客户端应用程序。

9.10.1　编程

当编写 VxDCOM 应用程序时，有几个需要注意的编程问题，描述如下：

1. 使用 VX_FP_TASK 选项

应该使用 VX_FP_TASK 选项创建所有 COM 线程的主函数。

⚠ **注意**：由于 COM 是单线程的，主函数生成的任何 COM 对象会在主函数线程中运行。

2. 避免使用虚拟基类

接口的每一次执行必须有自己的 vtable 复制，这包括 Iunknown 方式指针部分。如果使用虚拟继承改变这种设计，会干扰 COM 接口映射。

因此，当定义 COM 和 DCOM 类时，不要使用虚拟继承。例如，不使用以下定义：

```
class CoServer : public virtual IFoo,
                 public virtual IBar
    {
    // 类的实现
    };
```

3. 区分 COM 和 DCOM 的 API

VxDCOM 支持进程内服务器和远程服务器。当编写服务器时，请注意 COM 和 DCOM 组件的主要区别是使用的 COM API 函数的版本。COM 程序的 DCOM 版本通常添加 Ex.扩展名。特别的，必须使用 DCOM 应用的 COM CoCreateInstance( )函数的 CoCreateInstanceEx( )版本。详细内容请见“Microsoft COM”文档中的 COM 和 DCOM 库。

9.10.2　编写服务程序

本节的其余部分描述了 CoMathDemo 的服务器代码，重点描述了为客户端提供服务的

方式。这些方式是客户端和服务器之间的接口。本节中使用的代码来自于以下文件：

- CoMath.idl 接口定义文件
- CoMathDemoImpl.h 头文件
- CoMathDemoImpl.cpp 执行文件

有关执行的细节，请见 CoMathDemo 源文件。

1. 服务器接口

CoMathDemo 服务器组件执行两个接口，ImathDemo 和 IEvalDemo，这两个接口都直接来自于 Iuknown。

ImathDemo 接口定义了 19 个普通数学方式，这些方式列于下方的接口定义中。

```
interface IMathDemo : IUnknown
    {
    HRESULT pi ([out,retval]double* value) ;
    HRESULT acos ([in]double x,[out,retval]double* value) ;
    HRESULT asin ([in]double x,[out,retval]double* value) ;
    HRESULT atan ([in]double x,[out,retval]double* value) ;
    HRESULT cos ([in]double x,[out,retval]double* value) ;
    HRESULT cosh ([in]double x,[out,retval]double* value) ;
    HRESULT exp ([in]double x,[out,retval]double* value) ;
    HRESULT fabs ([in]double x,[out,retval]double* value) ;
    HRESULT floor ([in]double x,[out,retval]double* value) ;
    HRESULT fmod ([in]double x,[in]double y,[out,retval]double* value) ;
    HRESULT log ([in]double x,[out,retval]double* value) ;
    HRESULT log10 ([in]double x,[out,retval]double* value) ;
    HRESULT pow ([in]double x,[in]double y,[out,retval]double* value) ;
    HRESULT sin ([in]double x,[out,retval]double* value) ;
    HRESULT sincos ([in]double x,[out]double* sinValue,
                    [out]double* cosValue) ;
    HRESULT sinh ([in]double x,[out,retval]double* value) ;
    HRESULT sqrt ([in]double x,[out,retval]double* value) ;
    HRESULT tan ([in]double x,[out,retval]double* value) ;
    HRESULT tanh ([in]double x,[out,retval]double* value) ;
    };
```

IevalDemo 定义了两个方式 eval( )和 evalSubst( )。

```
interface IEvalDemo : IUnknown
    {
    HRESULT eval ([in]BSTR str,[out,retval]double* value) ;
    HRESULT evalSubst ([in]BSTR str,
                       [in]double x,
```

```
                    [out,retval]double* value) ;
    };
```

eval( )方式有一个 BSTR，它包含一个代数表达式 str，并返回双精度类型的数值。请注意，eval( )方式收到的变量由[in]参数属性定义，返回的数值由[out]和[retval]参数属性定义。[retval]属性允许需要返回值的语言使用这种方式。参数属性的详细内容请见《Tornado 用户指南》：COM 工具。

这个 evalSubst( )方式也具有一个包含代数表达式的 BSTR，并返回一个双精度类型的数值。另外，evalSubst( )也会用一个给定值来取代变量 x。

执行代码表明 eval( ) 和 evalSubst( )会返回一个 S_OK（成功解码表达式时）或 E_FAIL（出错时）。

2. 客户端的相互作用

CoMathDemoClient 是通过一个求值表达式的参数来创建的。客户端先访问服务器的 IevalDemo 接口，然后调用 Iuknown 的 QueryInterface( )访问 ImathDemo 接口。客户端代码调用 IevalDemo 的 eval( )方式，将其传递给表达式 str 进行求值，并返回给引用数值&result。然后，编写客户端代码来调用 IMathDemo 接口的 pi( )，sin( )和 cos( )方式（请见“9.10.4　询问服务器”）。

9.10.3　编写客户端代码

由于 DCOM 在结构之间是透明的，这样就不必区分客户应用中的进程内和远程调用对象而编写特殊代码。可以编写使用 COM 接口服务的代码，而不必考虑执行接口对象的网络位置。MathDemo 函数，如下所示，使用了相同的用户客户端代码。只有由 VxDCOM 向导生成的 ifdef 语句决定了客户端是 COM 还是 DCOM，以及对于 DCOM 客户端来说，它是用于 VxWorks 上还是 Windows NT 上。

该函数示范了怎样编写一个简单的 COM 或 DCOM 客户端。MathDemo 函数创建了一个 COM 对象并用这个对象执行简单的算术计算。

1. 决定客户端类型

客户端代码的第一个部分包含了#define 和#include 指令，它们大都是由 VxDCOM 向导自动生成的。用于_WIN32 alt*.*文件和 comOjbLib.h 文件的#include 指令是手动加入的，因为这些函数使用了 CComBSTR。同样，可以为函数需要的代码加入必要的头文件。

```
/* 包含头文件*/

#ifdef _WIN32
```

```
#define _WIN32_WINNT 0x0400
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "CoMathDemo.w32.h"
#include <atlbase.h>
#include <atlimpl.cpp>

#else

#include "comLib.h"
#include "CoMathDemo.h"
#include "comObjLib.h"
#define mbstowcs comAsciiToWide

#endif

#include <stdio.h>
#include <iostream.h>
#include <math.h>

#ifdef _DCOM_CLIENT
#ifdef _WIN32
#include "dcomLib.h"
#endif
#endif
```

2. 创建、初始化客户端

代码的这个部分通过调用适当的 COM 或 DCOM 函数，CoCreateInstance( )或 CoCreateInstanceEx( )创建 COM 或 DCOM 对象（组件）。DCOM 客户端由于安全目的需要附加的初始化代码。这个创建和初始化代码是由向导自动生成的。

```
#define MAX_X 79
#define MAX_Y 25

int CoMathDemoClient (const char* serverName,const char* expression)
    {
    HRESULT     hr = S_OK;
    double      result;
    int         x;
    int         y;
    IUnknown *  pItf;
```

```
    IEvalDemo * pEvalDemo;
    IMathDemo *  pMathDemo;
```

如果客户端是一个 COM 客户端，代码如上。

```
#ifndef _DCOM_CLIENT
    // 创建 COM 对象
    hr = CoCreateInstance (CLSID_CoMathDemo,
                           0,
                           CLSCTX_INPROC_SERVER,
                           IID_IEvalDemo,
                           (void **) &pItf) ;
#else
```

如果客户端是 DCOM 客户端，代码如上。这部分代码为线程初始化 DCOM 并在目标机上创建 DCOM 对象。

```
    OLECHAR wszServerName [128];

    // 将服务器名转换为宽字符
    mbstowcs (wszServerName,serverName,strlen (serverName) + 1) ;

    // 初始化 DCOM
    hr = CoInitializeEx (0,COINIT_MULTITHREADED) ;
    if (FAILED (hr) )
        {
        cout << "Failed to initialize DCOM\n";
        return hr;
        }

    // This initializes security to none.
    hr = CoInitializeSecurity (0,-1,0,0,
                           RPC_C_AUTHN_LEVEL_NONE,
                           RPC_C_IMP_LEVEL_IDENTIFY,
                           0,EOAC_NONE,0) ;
    if (FAILED (hr) )
        {
        cout << "Failed to initialize security\n";
        return hr;
        }
```

下列代码创建了一个 MQI 结构，该结构用来访问 IID_IMathDemo 接口的 COM 服

务器对象。这是由 CoMathDemo CoClass（初始化为 DCOM 服务器）定义的两个接口中的一个。

当编写一个典型的具有多个端口的 DCOM 客户端程序时，需要将所有的接口申请包含进 MQI 并询问它们，以此作为一次操作（从而节省了带宽）。然而，出于这个例子的目的，我们需要保持代码的主体不变，所以只需要 Iunknown 用于 DCOM 对象。我们可以视其为一个 COM 对象。

```
MULTI_QI mqi [] = { {&IID_IEvalDemo,0,S_OK} };
COSERVERINFO serverInfo = { 0,wszServerName,0,0 };

hr = CoCreateInstanceEx (CLSID_CoMathDemo,
                         0,
                         CLSCTX_REMOTE_SERVER,
                         &serverInfo,
                         1,
                            mqi) ;

if (SUCCEEDED (hr) && SUCCEEDED (mqi [0].hr) )
    {
    cout << "Created CoMathDemo OK\n";
    pItf = mqi [0].pItf;
    }

else

    {
    cout << "Failed to create CoMathDemo,HRESULT=" <<
        hex << cout.width (8) << mqi [0].hr << "\n";
    return E_FAIL;
    }

#endif
```

9.10.4　询问服务器

询问 COM 对象的 Iunknown 接口来获得指向 IevalDemo 接口的接口指针。

```
if (FAILED (hr = pItf->QueryInterface (IID_IEvalDemo,
            (void**) &pEvalDemo) ) )
    {
```

```
        cout << "Failed to create IEvalDemo interface pointer,
                HRESULT=" << hex << cout.width (8) << hr << "\n";
        pItf->Release ( );
        return hr;
        }
```

询问 COM 对象的 Iunknown 接口来获得指向 IMathDemo 接口的接口指针。

```
    if (FAILED (pItf->QueryInterface (IID_IMathDemo,void**) &pMathDemo) ) )
        {
        cout << "Failed to create IMathDemo interface pointer,HRESULT="
             << hex << cout.width (8) << hr << "\n";
        pEvalDemo->Release( );
        pItf->Release ( );
        return hr;
        }

    pItf->Release ( );
```

该代码调用 IevalDemo 接口的 eval( )方式，对从命令行到客户端程序的给定表达式求值[5]。

```
    cout << "Querying IEvalDemo interface\n";
    CComBSTR str;

    str = expression;

    hr = pEvalDemo->eval (str,&result) ;
    if (SUCCEEDED (hr) )
        {
        cout << expression << "=" << result;
        }
        else
        {
        cout << "eval failed (" << hr << "," << result << ") \n";
        }
    pEvalDemo->Release ( );
```

这个代码询问 ImathDemo 接口来画出正弦和余弦函数。注意，它调用了那个接口的

[5] 输入的表达式具有 char 数组形式，但是必须将其转换为 BSTR 以便使其配置到 COM 服务器中。

pi( )，sin( )和 cos( )方式。

```
printf ("Querying IMathDemo interface\n") ;

double sinResult;
double cosResult;
double pi;

hr = pMathDemo->pi (&pi) ;
if (FAILED (hr) )
    return hr;

double step_x = (pi * 2.0) / ( (double) MAX_X) ;
double scale_y = ( (double) MAX_Y) / 2.0;

for (y = MAX_Y; y >= 0; y--)
    {
    for (x = 0; x < MAX_X; x++)
        {
        hr = pMathDemo->sin ( (double) x * step_x ,&sinResult) ;
        if (FAILED (hr) )
            return hr;
        hr = pMathDemo->cos ( (double) x * step_x ,&cosResult) ;
        if (FAILED (hr) )
            return hr;
        if ( (int) ( (double) ( (sinResult + 1.0) * scale_y) ) == y)
            {
            putchar ('*') ;
            }
        else if ( (int) ( (double) ( (cosResult + 1.0) * scale_y) ) ==y)
            {
            putchar ('+') ;
            }
        else
            {
            putchar (' ') ;
            }
        }
    putchar ('\n') ;
    }
pMathDemo->Release ( );
```

```
#ifdef _DCOM_CLIENT
    CoUninitialize ( );
#endif
    return hr;
    }
```

9.10.5　执行客户端代码

代码的这个部分用于在 Windows NT 操作系统上运行的 DCOM C++客户端。除了变量 exp，该代码几乎全部由向导生成。

```
#ifdef _WIN32
int main (int argc,char* argv [])
    {
    if (argc != 3)
    {
    puts ("usage: CoMathDemoClient <server> <exp>") ;
    exit (1) ;
    }

    return CoMathDemoClient (argv [1],argv[2]) ;
    }
```

代码的这个部分是由程序员加入的，以防 C++名称混淆。

```
#else
extern "C"
    {
    int ComTest (char * server,char * exp)
        {
        return CoMathDemoClient (server,exp) ;
        }
    }
#endif
```

9.11　比较 VxDCOM 和 ATL 执行

本节总结了 VxDCOM 和 Microsoft ATL 的 COM 接口执行之间的主要不同点。同时本节也包括了有关对于 VARIANT 的 VxDCOM 使用说明。

9.11.1 CcomObjectRoot

VxDCOM 直接执行 CcomObjectRoot，而 ATL 将它作为一个 CcomObjectRootEx 的类型定义来执行。

CcomObjectRoot 通常会执行集合指针，即使不使用该指针。

1. Constructor

类的构造器。将 ref 计数器初始化为 0。并且如果需要，将为集合外层接口提供存储空间。

```
CComObjectRoot
  (
  IUnknown *   punk = 0          //集合外层 aggregating outer
  )
```

2. InternalAddRef

使 ref 计数器加 1，并返回 ref 计数器结果。

```
ULONG InternalAddRef ( )
```

3. InternalRelease

使 ref 计数器减 1，并返回 ref 计数器结果。

```
ULONG InternalRelease ( )
```

9.11.2 CcomClassFactory

来自于 IclassFactory 和 CcomObjectRoot。

1. Constructor

类的构造器。

```
CComClassFactory ( )
```

2. AddRef

使 ref 计数器加 1，并返回 ref 计数器结果。这个过程需要调用 CcomObjectRoot 中的 InternalAddRef( )。

```
ULONG STDMETHODCALLTYPE AddRef ( )
```

3. Release

使 ref 计数器减 1，并返回 ref 计数器结果。这个过程需要调用 CcomObjectRoot 中的 InternalRelease( )。

```
ULONG STDMETHODCALLTYPE Release ( )
```

4. QueryInterface

准备 QueryInterface 机制，为 IClassFactory 提供 IID_IUnknown 和 IID_IclassFactory 接口。

```
HRESULT STDMETHODCALLTYPE QueryInterface
  (
  REFIID riid，// 需要接口的 GUID
  void ** ppv // 指向接口 riid 的指针
  )
```

5. CreateInstance

为所需的对象创建一个新的实例；调用 CComObjectRoot::CreateInstance。因此，与 ATL 不同，为 WOTL 创建的对象被初始化。

```
HRESULT STDMETHODCALLTYPE CreateInstance
  (
  IUnknown * pUnkOuter，//集合外层 aggregated outer
  REFIID riid，// 创建接口的 GUID
  void ** ppv //指向接口 riid 的指针
  )
```

6. LockServer

这是一个存根函数，通常返回 S_OK，使用它的目的在于加强兼容性。

```
HRESULT STDMETHODCALLTYPE LockServer
  (
  BOOL Lock
  )
```

9.11.3 CcomCoClass

VxDCOM 不执行 DECLARE_CLASSFACTORY 或 DECLARE_AGGREGATABLE 宏。它们的功能将隐性建立到类中。

GetObjectCLSID
返回对象的 CLSID。
static const CLSID& GetObjectCLSID( )

9.11.4 CcomObject

它来自于给定的接口类。

1. CreateInstance

CreateInstance 有两个版本。每一个版本什么时候使用以及怎样调用，这些细节请见 Microsoft COM 文档。

下面定义的 CreateInstance 创建了一个实例，该实例没有集合，而且没有指定的 COM 接口。

```
static HRESULT CreateInstance
  (
  CComObject** pp // 创建的对象  )
```

下面的 CreateInstance 版本创建了类的一个实例，并查找所需的接口。

```
static HRESULT CreateInstance
  (
  IUnknown* punkOuter, // 可累加接口
  REFIID riid, // 接口的 GUID
  void** ppv // resultant 对象
  )
```

2. AddRef

在 punkOuter 或 CcomObjectRoot（取决于对象的类型）上执行 AddRef。

```
ULONG STDMETHODCALLTYPE AddRef( )
```

3. Release

在 punkOuter 或 CComObjectRoot::AddRef（取决于对象的类型）上执行 Release。

```
ULONG STDMETHODCALLTYPE Release ( )
```

4. QueryInterface

询问 punkOuter 或对象以寻找接口。

```
HRESULT STDMETHODCALLTYPE QueryInterface
  (
  REFIID riid, // 要询问的 GUID
```

```
void ** ppv // resultant 接口
)
```

9.11.5　CComPtr

CcomPtr 是具有 COM 接口的模板类，它指定了存储的指针类型。

1. Constructors

支持构造器。

```
CComPtr ( )
CComPtr（Itf* p）
CComPtr（const CComPtr& sp）
```

2. Release

释放一个对象的实例。

```
void Release ( )
```

3. Operators

支持操作符。

```
operator Itf* ( ) const
Itf** operator& ( )
Itf* operator-> ( )
const Itf* operator-> ( ) const
Itf* operator=（Itf* p）
Itf* operator=（const CComPtr& sp）
bool operator! ( ) const
```

4. Attach

将一个对象赋给指针，该过程不会使 ref 计数器加 1。

```
void Attach
  (
  Itf * p2 // 赋给指针的对象
  )
```

5. Detach

解除一个对象与其指针的联系，该过程不会使 ref 计数器减 1。

```
Itf *Detach( )
```

6. CopyTo

将一个对象复制到指针并使 ref 计数器加 1。

```
HRESULT CopyTo
  (
  Itf ** ppT //复制到指针的对象)
```

9.11.6 CComBSTR

BSTR 类型的类封装。CComBSTR 为安全创建、分配、转换以及解构一个 BST
方式。

1. Constructors

CComBSTR 支持的构造器。

```
CComBSTR ( )
explicit CComBSTR（int nSize，LPCOLESTR sz = 0）
explicit CComBSTR（LPCOLESTR psz）
explicit CComBSTR（const CComBSTR& src）
```

2. Operators

CComBSTR 支持的操作符。

```
CComBSTR& operator=（const CComBSTR& cbs）
CComBSTR& operator=（LPCOLESTR pSrc）
operator BSTR ( ) const
BSTR * operator& ( )
bool operator! ( ) const
CComBSTR& operator+=（const CComBSTR& cbs）
```

3. Length

获得 BSTR 的长度。

```
unsigned int Length ( ) const
```

4. Copy

在封装类中对 BSTR 进行复制并将其返回。

```
BSTR Copy( ) const
```

5. Append

Append 到 BSTR。

void Append（const CComBSTR& cbs）
void Append （LPCOLESTR lpsz）
void AppendBSTR（BSTR bs）
void Append（LPCOLESTR lpsz，int nLen）

6. Empty

从封装类中删除现有的任何 BSTR。
void Empty ( )

7. Attach

将 BSTR 加入封装类中。
void Attach（BSTR src）

8. Detach

从封装类中解除 BSTR 并将其返回。
BSTR Detach ( )

9.11.7 VxComBSTR

ComObjLibExt 文件是定义在 comObjLib.h 中的类似于 ATL 的类，该文件提供 VxWorks 的特定扩展。VxComBSTR 来自于 CComBSTR 并使其扩展。

1. Constructors

这个类的构造器来自于 CComBSTR 构造器。
VxComBSTR ( )
explicit VxComBSTR（int nSize，LPCOLESTR sz = 0）
explicit VxComBSTR（const char * pstr）
explicit VxComBSTR（LPCOLESTR psz）
explicit VxComBSTR（const CComBSTR& src）
explicit VxComBSTR（DWORD src）
explicit VxComBSTR（DOUBLE src）

2. Operators

被支持的操作符列于下方，并且这些操作符被声明。

将 BSTR 转化为一个 char 数组，并返回给临时复制的指针。这个复制确保在 VxComBSTR 对象下一次调用之前有效。它可以代替宏 OLE2T。

operator char * ( )

返回十进制数值并将其作为 DWORD 数值存储于 BSTR。这种方式遵循了与标准库函

数 atoi 相同的流规则。

operator DWORD ( )

VxComBSTR& operator = （const DWORD& src）

将双精度数值转换为十进制串表示，并将其存为 BSTR 类型。

VxComBSTR& operator = （const DOUBLE& src）

将一个字符数组转换为 BSTR 形式。它能够替换 T2OLE：

VxComBSTR& operator = （const char * src）

如果给定的 VxComBSTR 数值等于已存储的 BSTR，则返回 TRUE，否则返回 FALSE。

bool const operator == （const VxComBSTR& src）

如果给定的 VxComBSTR 数值不等于已存储的 BSTR，则返回 TRUE，否则返回 FALSE。

bool const operator != （const VxComBSTR& src）

3. 设置为十六进制

将一个 DWORD 数值转换为十六进制串表示，并将其存储为 BSTR。

void SetHex （const DWORD src）

9.11.8 CcomVariant

它来自于 tagVARIANT。

1. Constructors

CcomVariant 支持的构造器。

CComVariant( )

CComVariant （const VARIANT& varSrc）

CComVariant （const CComVariant& varSrc）

CComVariant （BSTR bstrSrc）

CComVariant （LPCOLESTR lpszSrc）

CComVariant （bool bSrc）

CComVariant （int nSrc）

CComVariant （BYTE nSrc）

CComVariant （short nSrc）

CComVariant （long nSrc， VARTYPE vtSrc = VT_I4）

CComVariant （float fltSrc）

CComVariant （double dblSrc）

CComVariant （CY cySrc）

CComVariant （IUnknown* pSrc）

2. Operators

CcomVariant 支持的操作符。

CComVariant& operator=（const CComVariant& varSrc）
CComVariant& operator=（const VARIANT& varSrc）
CComVariant& operator=（BSTR bstrSrc）
CComVariant& operator=（LPCOLESTR lpszSrc）
CComVariant& operator=（bool bSrc）
CComVariant& operator=（int nSrc）
CComVariant& operator=（BYTE nSrc）
CComVariant& operator=（short nSrc）
CComVariant& operator=（long nSrc）
CComVariant& operator=（float fltSrc）
CComVariant& operator=（double dblSrc）
CComVariant& operator=（CY cySrc）
CComVariant& operator=（IUnknown* pSrc）
bool operator==（const VARIANT& varSrc）
bool operator!=（const VARIANT& varSrc）

3. Clear

在封装类中清除变量。

HRESULT Clear( )

4. Copy

将给定的变量复制到封装类中。

HRESULT Copy（const VARIANT* pSrc）

5. Attach

将给定的 VARIANT 链接到类中，过程中并不对 VARIANT 进行复制。

HRESULT Attach（VARIANT* pSrc）

6. Detach

从类中释放 VARIANT 并返回。

HRESULT Detach（VARIANT* pDest）

7. ChangeType

改变 VARIANT 的类型为 vtNew。这种封装函数所支持的类型与 comLib 函数

VariantChangeType 所支持的类型相同。

```
HRESULT ChangeType
  (
  VARTYPE vtNew,
  const VARIANT* pSrc = NULL
  )
```

第 10 章　分布式信息队列

10.1　简　　介

VxFusion 是一个轻量级的、独立于介质的机制，它基于 VxWorks 信息队列，是为开发分布式应用程序服务的。

在 VxWorks 中有好几种分布式多处理的选择。风河公司选择的 VxMP 仅仅通过共享内存而实现允许目标的共享。TCP/IP 可以通过网络通信，但它是低层的、而不是用来做实时应用的。对于分布式计算来说可以使用标准的各种高层通信机制，但是它们按照一般不被实时系统接受的内存应用和计算时间来说存在很高的企业管理费用。大量的其他方法也被开发出来，但是它们越来越多地面临维护、端口移植和升级的问题。然而，VxFusion 是一个标准的 VxWorks 组件。它能：

- 提供基于 VxWorks 信息队列的轻量级的分布式机制。
- 提供独立于介质的服务，允许分布式系统通过任何传送设备有效交换数据，为通信硬件解决常用的要求。
- 提供解决由一点对多点的主从依赖导致的单点错误的安全措施，这通过在多节点系统中复制一个每个节点上的已知对象的数据库来实现。
- 支持在系统中通过单点和多点方式为目标发送信息。
- 展示位置透明，也就是说，对象在系统中可以在不重写应用程序代码的情况下无缝地移动。在多节点系统中，为目标对象写入信息时并不考虑它们的位置。

VxFusion 与 VxMP 共享内存对象选项相似。为了通过共享内存而共享信息队列、信号量和存储区，VxMP 增加了基本的 VxWorks 信息队列的功能性支持；为了通过任一传送器共享信息队列以及对信息队列组的多点传送，VxFusion 增加了对 VxWorks 支持。不象 VxMP 共享内存选项那样，VxFusion 并不支持分布式信号量或分布式存储区的分配。

警告：在某一主机上创建一个分布式队列时，此信息就通过名称数据库被广播至别的主机。如果创建队列的主机发生了冲突，那么对于其他主机来说就没有简单的方法查找此信息了。这样，别的主机可能在接受过程中永远处于挂起状态，这依赖于用户提供检测远端节点冲突和更新数据库的方法。

10.2 用 VxFusion 配置 VxWorks

用组件 INCLUDE_VXFUSION 配置 VxWorks 可以提供基本的 VxFusion 功能。为了配置 VxFusion 的显示程序，在内核区也包括以下的组件：

INCLUDE_VXFUSION_DIST_MSG_Q_SHOW
INCLUDE_VXFUSION_GRP_MSG_Q_SHOW
INCLUDE_VXFUSION_DIST_NAME_DB_SHOW
INCLUDE_VXFUSION_IF_SHOW

⚠ **注意**：VxFusion 不能被不支持以太网广播的目标机如 VxSim 目标机仿真器来配置。

10.3 使用 VxFusion

VxFusion 为标准的 VxWorks 信息队列增加了两种类型的分布式目标：

- 分布式信息队列
- 组信息队列

分布式信息队列可以通过任一通信介质来共享。组信息队列是一个虚拟信息队列，它接收一个信息然后把它发送给此组中的所有其他信息队列成员。VxFusion 是提供调谐控制的一组程序，它是通过分布式信息队列控制的定时而完成此功能的。然而，很多被用来操作标准信息队列的 API 调用也操作分布式目标对象，使用户的关于 VxFusion 的基于信息队列的应用程序的端口化问题变得容易一些。

VxFusion 也提供了分布式名称数据库，这个分布式名称数据库用来在程序间共享数据。此分布式名称数据库的 API 与 VxMP 的共享名称数据库的 API 相似。请参照"*VxWorks API Reference.*"中的 distNameLib 的相关条目。

这部分讨论了系统的结构和初始化、配置 VxFusion、操作不同组件以及如何写适配器的问题。

10.3.1 VxFusion 的系统结构

一个典型的 VxFusion 系统由通过一个通信路径连接的两个或多个节点组成，如图 10-1

示。通信路径被认为是传送器，它可以是通信硬件，像以太网网卡或总线，或者可以是到通信硬件的软件接口，像驱动程序或协议栈。通常情况下传送器是软件接口，对于同通信硬件直接通信来讲、它只在为数不多的情况下是有用的或是必需的。

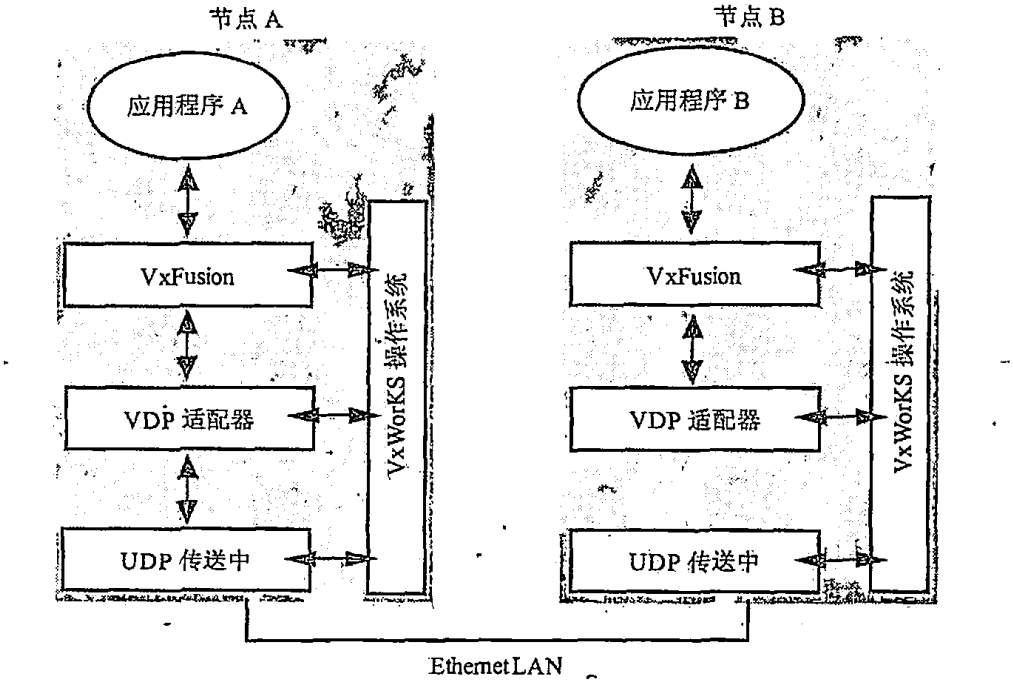


图 10-1　VxFusion 系统举例

有很多可能的传送方式，这样，就有很多可能的不同 API，寻址机制等；因此，VxFusion 要求在它自己和传送器之间存在一个叫做适配器的软件。适配器提供了到 VxFusion 的统一接口，而不考虑下层的传送方式。在这种情况下，适配器与驱动程序相似。对于驱动程序，为了支持每一种类型的 VxFusion 传送方式，要为它写一个新的适配器。VxFusion 组件为写新的适配器提供了一个 UDP 适配器作为例子或向导（参看“10.3.7 操作适配器”）。

图 10-1 举了两个节点 VxFusion 系统的例子。VxFusion 已经被安装到与同一个以太网子网相连的两个节点的任　节点上。因为这两个节点通过以太网相连，TCP、UDP、IP 以及自然以太网都是可能的通信传送方式。

在图 10-1 中，UDP 协议按传送方式工作并作为适配器而提供 UDP 的适配器服务。

1. 服务和数据库

VxFusion 实际上由很多服务和数据库组成，如图 10-2 所示。VxFusion 数据库和服务在表 10-1 和 10-2 中单独列出（这些表中使用了报文一词，请参看“10.6 报文与消息”）。任　服务作为　个独立任务运行，当用户列出安装并运行在 VxFusion 的节点上的任务时会看到这些确定的任务。注意在使用 VxFusion 时不必意识到这些服务和数据库的存在。

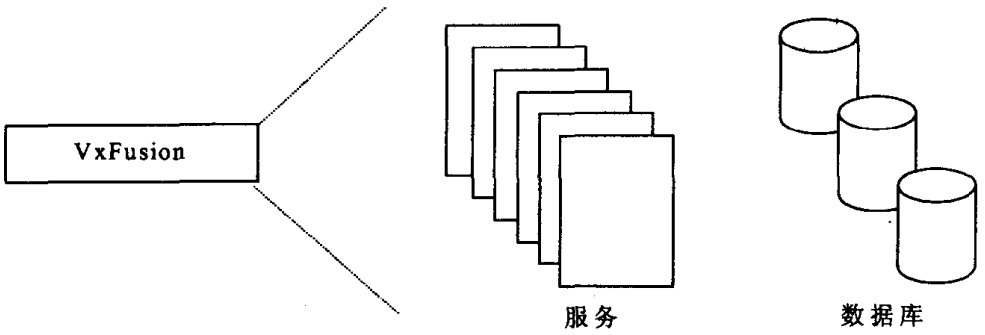


图 10-2　VxFusion 组件

表 10-1　VxFusion 数据库

数　据　库	描　　述
分布式名称数据库	分布名字数据库由名字-数值-类型等组成（参看 10.3.4　分布式名称数据库）。它在系统中的每个节点上都有备份，使这些条目对节点上每个任务都可得
分布式组数据库	分布式组数据库包括分布式信息队列组和它们的局部附加成员的列表
分布式节点数据库	分布式节点数据库包括系统中所有其他节点和它们的状态列表

表 10-2　VxFusion 服务

服　　务	描　　述
分布式信息队列服务	处理远端节点的分布式信息队列报文
组信息队列服务	处理远端节点的组信息队列报文
分布式名称数据库服务	处理远端节点的分布式名称数据库报文
分布式组数据库服务	处理远端节点的分布式信息队列组数据库报文
组合服务	处理远端节点的组合信息。组合信息使用于信号以及确认数据库更新的开始和结束
组一致协议服务（GAP）	处理 GAP 信息。GAP 信息为了给组选择一个单独的 ID 而在节点间被传送

2. 软件库

VxFusion 提供了以下的软件库：

分布式名称数据库（distNameLib，distNameShow）

分布式信息队列（msgQDistLib，msgQDistShow）

组信息队列（msgQDistGrpLib，msgQDistGrpShow）

报文缓冲（distTBufLib）

分布式对象统计（distStatLib）

VxFusion 适配器接口（distIfLib，distIfShow）

VxFusion 网络层（distNetLib）

10.3.2　VxFusion 的初始化

当用户启动存在激活的 VxFusion 的 VxWorks 映像时，即可以用在 usrVxFusion.c 中查找的函数 usrVxFusionInit( )来调用 VxFusion 的初始化程序 distInit( )以启动 VxFusion，而用函数 distInit( )在当前节点上初始化 VxFusion。VxFusion 一定要被安装在系统的每一个节点上以使这些节点上的程序之间可以进行通信，distInit( )的参数被设置为默认值。函数 distInit( )完成以下基本的操作：

- 初始化局部服务及数据库
- 确定系统中别的 VxFusion 节点并决定它们的状态
- 如果系统中有其他节点，使用其他节点中的一个数据来更新局部数据库

当目标启动的时候由于自动调用 distInit( )，如果用户的 VxWorks 映像包括 VxFusion，那么用户不应该从用户程序中直接调用此程序。

10.3.3　配置 VxFusion

用户可以用以下一种方法来配置 VxFusion 的各个部分，包括初始化、运行时间以及适配器接口。

- 用 usrVxFusionInit( )来改变初始化时设置的特征
- 用 distCtl( )来控制运行时间
- 用结构 DIST_IF 来调整适配器接口

用户初始化的每一种方法都在这部分中有详细描述

1. 用 usrVxFusionInit( )来进行个性化设置

这个程序调用了 VxFusion 的初始化函数 distInit( )，它可以进行用户化设置并且可用来改变的配置参数 distInit( )，此参数能够控制 VxFusion 环境的初始化。表 10-3 列出了大量的配置参数。想了解更多关于 distInit( )的更多信息，请参看 *VxWorks API Reference.*的相关条目。

表 10-3　用 usrVxFusionInit( ) 来改变的配置参数

参数/定义	默 认 值	描　　述
MyNodeId 节点 ID	启动接口的 IP 地址	为节点指定 Id 号。VxFusion 系统中的每一个节点都有一个单独的 ID。通过默认值，usrVxFusionInit( )代码启动的接口的 IP 地址作为节点 ID 号。函数 UsrVxFusionInit( )提供其作为函数 distInit( )的第一个参数
IfInitRtn 具体适配器初始化程序	distUdpInit( )	指定将被使用的接口适配器的初始化程序。通过默认值，函数 usrVxFusionInit( )指定了 UDP 适配器的初始化程序

续表

参数/定义	默 认 值	描　述
pIfInitConf 具体适配器配置结构	启动接口	为适配器提供了任一附加具体适配器的配置信息。比如，函数 usrVxFusionInit( )给 UDP 适配器提供了节点从其上启动的接口
maxTBufsLog2 Tbufs 的最大数值	9（512 log 2）	指定将被创建的报文缓冲区的最大值。函数 usrVxFusionInit( ) 一定要以 log 2 的形式提供此参数，也就是说，如果最大值是 512，参数就是 9
maxNodesLog2 在分布式节点数据库中的最大节点数	5（32 log 2）	指定分布式节点数据库中的最大节点值。函数 usrVxFusionInit( )一定要以 log 2 的形式提供此参数
maxQueuesLog2 节点上队列的最大数	7（128 log 2）	指定能够在一个单独节点上创建的分布式信息队列的最大数。函数 usrVxFusionInit( )一定要以 log 2 的形式提供此参数
maxGroupsLog2 分布式组数据库中的组最大数	6（64 log 2）	指定可以在分布式组数据库中被创建的组信息队列的最大数。函数 usrVxFusionInit( )一定要以 log 2 的形式提供此参数
maxNamesLog2 分布式名字数据库中的最大条目	8（256 log 2）	指定可以被存储在分布式名称数据库中的最大条目数。函数 The usrVxFusionInit( )一定要以 log 2 的形式提供此参数
WaitNTicks 等待时钟的最大数值	240	指定启动时等待其他节点的响应时间的时钟数

1：4 *sysClkRateGet( )的典型值是 240，但也不总是这样。默认值在 vxfusion/distLib.h 中有定义，并且如果用户需要可以修改它。

⚠ **注意**：如果调用函数 distInit( )失败，就返回 ERROR，那么 VxFusion 就不在主机上启动了。

2. 用 distCtl( ) 进行个性化设置

此程序完成分布式目标对象的控制功能 。可以使用 distCtl( )的配置表 10-4 中列出的与运行时间相关的参数和钩子函数。关于使用 distCtl( )的可用控制函数的更多信息，请参看“*VxWorks API Reference*”。

表 10-4　用 distCtl( )修正的配置参数

参数或钩子函数	默 认 值	描　述
DIST_CTL_LOG_HOOK	NULL	每产生一个日志信息就设置一个将被调用的程序。如果没有设置日志钩子函数，那么输出日志将被打印到标准输出设备上

续表

参数或钩子函数	默 认 值	描　述
DIST_CTL_PANIC_HOOK	NULL	设置一个当系统遇到没有碰到过的错误时的调用程序。如果没有设置紧急钩子函数，那么输出将被打印到标准输出设备上
DIST_CTL_RETRY_TIMEOUT	200ms	设置初始的重发时限。尽管默认超时设定以毫秒形式表示，重试时限实际上以时钟设置。超时设定值最少可以到 200ms
DIST_CTL_MAX_RETRIES	5	设置发送失败时再试的极限次数
DIST_CTL_NACK_SUPPORT	TRUE	允许或禁止发送否定确认（NACK）
DIST_CTL_PGGYBAK_UNICST_SUPPORT	FALSE	允许或禁止单播机载
DIST_CTL_PGGYBAK_BRDCST_SUPPORT	FALSE	允许或禁止广播机载
DIST_CTL_OPERATIONAL_HOOK	NULL	每次节点改变到就绪状态，则增加一个将被调用程序的列表，最多可以增加 8 个程序
DIST_CTL_CRASHED_HOOK	NULL	每次节点改变到冲突状态，则增加一个调用程序的列表。此列表最多可容纳 8 个程序，然而，一块空间被 VxFusion 使用，仅留出够 7 个用户指定增加的程序空间
DIST_CTL_SERVICE_HOOK	NULL	为远端节点调用的服务设置每次节点上的服务程序失败后的调用程序
DIST_CTL_SERVICE_CONF	具体服务	为具体服务设置任务和网络优先级

⚠ **注意**：DIST_CTL_CRASHED_HOOK 应该总是用 distCtl( )来调用，因为它是 VxFusion 能提供节点冲突提示的惟一方式。

3. 用 dist_if 结构进行个性化设置

表 10-5 列出了用 DIST_IF 结构配置的内容，此结构用来给 VxFusion 传送关于适配器的数据。DIST_IF 结构使 VxFusion 独立传送。关于 DIST_IF 结构的更多信息，请参看“10.7.2 写一个初始化程序”。

表 10-5　用接口适配器改变的配置参数

DIST_IF 内容/定义	DIST_IF 内容/定义	描　述
DistIfName 接口适配器名字	具体适配器即 UDP 适配器：“UDP 适配器”	指定接口适配器的名字
DistIfMTU MTU 大小	具体适配器即 UDP 适配器：1500 字节	指定接口适配器协议的 MTU 大小
distIfHdrSize 网络报头大小	具体适配器即 UDP 适配器：NET_HDR 大小	指定网络报头大小

续表

DIST_IF 内容/定义	DIST_IF 内容/定义	描　　述
DistIfBroadcastAddr 广播地址	具体适配器即 UDP 适配器：子网的 IP 广播地址	为将被传送的接口和传送器指定广播地址
distIfRngBufSz 环形缓冲器大小	具体适配器即 UDP 适配器：256	指定在变化窗口协议中用到的元件数
distIfMaxFrags 最大片断数	具体适配器即 UDP 适配器：10	指定　个消息可以被分解成的是最大片断数

10.3.4　分布式名称数据库

分布式名称数据库允许任何数数值与任一名字匹配起来，比如用一个单独的名字表示一个分布式信息队列的 ID 号。分布式名称数据库提供了名字到数值和类型以及数值和类型到名字的两种转换方式，允许以名字或数值和类型的形式访问数据库。

系统中的每一个节点上都有　个分布式名称数据库的备份。任何对于数据库的局部备份的修改都被立即传送到系统中所有其他节点的其他备份上。

典型的情况是，想要共享数值的任务往分布式名称数据库中增加　个名字-类型-数值的记录。当往数据库中增加记录时，此任务用一个单独的、具体的名字与此数值相联系。不同节点上的任务都使用这个名字来得到相关数值。

请看　下图 10-3 的例子，本例说明了不同节点上的两个任务如何共享　个通用分布式信息队列的 ID。

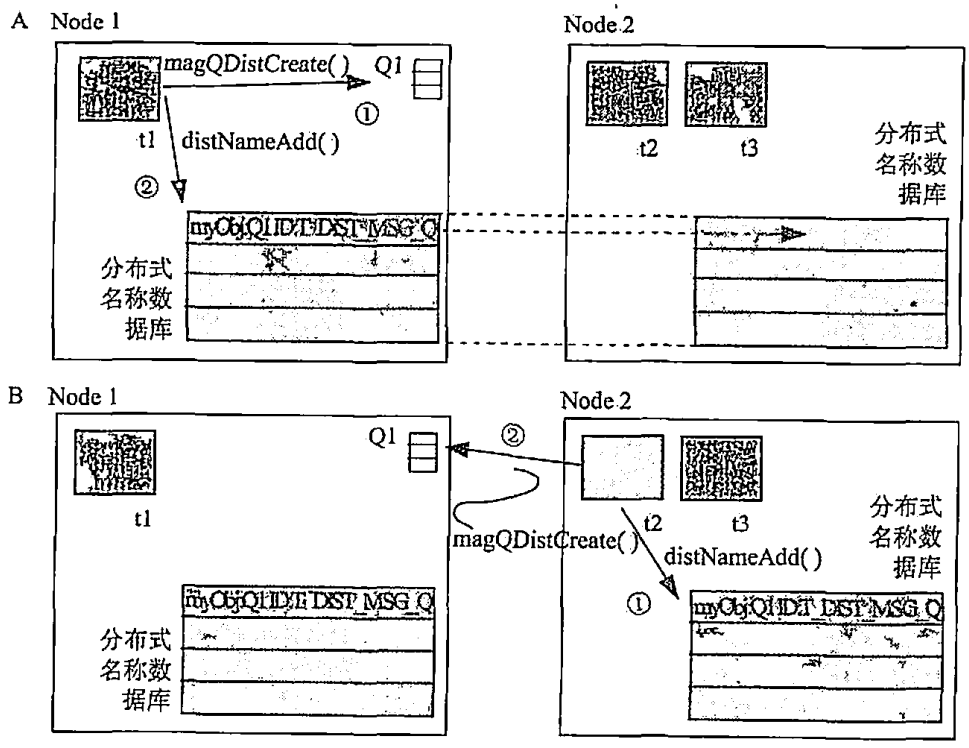


图 10-3　使用分布式数据库

节点 1 上的任务 t1 创建了一个消息队列。分布式信息队列的 ID 通过创建程序被返回。

任务 t1 把它的 ID 号和与其相关联的名字 myObj 增加到分布式名称数据库中。然后这个数据库记录被传播至系统中的所有其他节点上。如果节点 2 上任务 t2 要给此分布式信息队列发送一个消息，那么它首先要在节点 2 的分布式名称数据库的局部备份中查找名字 myObj 来找到其 ID 号。

然而，如果节点 2 没有收到广播（比如因为网络不通）两个节点上的信息就会不匹配，节点 2 就不会觉察到 Q1。（这是一个可以用函数 distCtl( )来通知函数 DIST_CTL_CRASHED_HOOK 的例子；请看表 10-4。）

表 10-6 列出了分布式名称数据库的服务程序，分布式名称数据库因为要调用 printf( ) 来打印信息，所以它包含浮点值。任何调用函数 distNameShow( )的任务都应该设置任务选项 VX_FP_TASK。

表 10-6　分步式名称数据库服务函数

程　序	功　能
distNameAdd( )	往分布式名称数据库中增加一个名字
distNameFind( )	通过名字查找分布式对象
DistNameFindByValue( )	通过数值和类型查找分布式对象
distNameRemove( )	从分布式名称数据库中移走一个对象
distNameShow( )	把分布式名称数据库的所有内容显示到标准输出设备上
distNameFilterShow( )	显示一个指定类型的数据库中的所有记录

此部分提供了关于往分布式名称数据库中增加名字以及相关显示程序的补充信息。关于所有这些函数的详细信息，请参看“*VxWorks API Reference*”。

1. 给分布式名称数据库增加名字

用 diatNameAdd( )往分布式名称数据库增加一个名字-数值-类型的记录。这个类型可以被用户定义或者在库 distNameLib.h 中提前定义。

> ⚠ **注意**：分布式名称数据库仅为预定义的类型提供了与网络字节命令之间的互相转换。不要试图调用 htnol( )或 ntohl( )从分布式名称数据库中得到预定义类型的数值。

表 10-7 列出了预定义的类型。

表 10-7　分布式名称数据库类型

常　数	十 进 制 数	目　的
T_DIST_MSG_Q	0	分布式信息队列标识符（也就是组 ID）
T_DIST_NODE	16	节点标识符

续表

常　数	十 进 制 数	目　的
T_DIST_UINT8	64	8 比特无符号整数
T_DIST_UINT16	65	16 比特无符号整数
T_DIST_UINT32	66	32 比特无符号整数
T_DIST_UINT64	67	64 比特无符号整数
T_DIST_FLOAT	68	单精度浮点数（32 比特）
T_DIST_DOUBLE	69	双精度浮点数（64 比特）
用户定义的类型	4096 以上	用户定义的类型

属于一个特别名字的数值仅通过再一次调用 distNameAdd( )就可更新。

2. 显示分布式名称数据库的信息

可以使用两个函数来显示分布式名称数据库的数据：distNameShow( )和 distNameFilterShow( )。

⚠ **注意**：由 VxFusion 提供的分布式名称数据库可以包含浮点值。函数 distNameShow( )调用 print( )来打印它们。任何调用 distNameShow( )的任务应该设置 VX_FP_TASK 任务选项，目标 shell 也要设置此选项。

函数 distNameShow( )把分布式名称数据库的全部内容显示到标准输出设备上。以下示例说明了对 distNameShow( )的应用，输出被传送到标准输出设备上：

```
[VxKernel]-> distNameShow()
        NAME                TYPE                VALUE
-------------------- -------------- ------------------------
nile                    T_DIST_NODE 0x930b2617(2466981399)
columbia                T_DIST_NODE 0x930b2616(2466981398)
dmq-01                  T_DIST_MSG_Q 0x3ff9fb
dmq-02                  T_DIST_MSG_Q 0x3ff98b
dmq-03                  T_DIST_MSG_Q 0x3ff94b
dmq-04                  T_DIST_MSG_Q 0x3ff8db
dmq-05                  T_DIST_MSG_Q 0x3ff89b
gData                   4096 0x48 0x65 0x6c 0x6c 0x6f 0x00
gCount                  T_DIST_UINT32 0x2d(45)
grp1                    T_DIST_MSG_Q 0x3ff9bb
grp2                    T_DIST_MSG_Q 0x3ff90b
value = 0 = 0x0
```

函数 distNameFilterShow( )显示了按类型选择的分布式名称数据库的内容，也就是说，它仅显示了数据库中与指定类型相匹配的记录。以下输出说明了使用 distNameFilterShow( )仅显示信息队列的 ID：

```
[VxKernel]-> distNameFilterShow(0)
        NAME                TYPE                VALUE
-------------------- -------------- -------------------------
dmq-01                  T_DIST_MSG_Q 0x3ff9fb
dmq-02                  T_DIST_MSG_Q 0x3ff98b
dmq-03                  T_DIST_MSG_Q 0x3ff94b
dmq-04                  T_DIST_MSG_Q 0x3ff8db
dmq-05                  T_DIST_MSG_Q 0x3ff89b
grp1                    T_DIST_MSG_Q 0x3ff9bb
grp2                    T_DIST_MSG_Q 0x3ff90b
value = 0 = 0x0
```

10.3.5　操作分布式信息队列

分布式信息队列是可以被本地和远程任务透明操作的信息队列。表 10-8 列出了用来控制分布式信息队列的程序。

表 10-8　分布式信息队列函数

函　数	功　能
sgQDistCreate( )	创建一个分布式信息队列
msgQDistSend( )	给分布式信息队列发送一个消息
msgQDistReceive( )	从分布式信息队列接收一个消息
msgQDistNumMsgs( )	得到分布式信息队列的消息数

分布式信息队列一定要用函数 msgQDistCreate( )来创建，在物理上它位于发起创建访问的节点上。

一旦创建，分布式信息队列就可以通过由 msgQLib 提供的标准信息队列函数来操作，它们是 msgQSend( )，msgQReceive( )，msgQNumMsgs( )，msgQDelete( )和 msgQShow( )。直到局部节点用 distNameAdd( )往分布式名称数据库中增加其 ID 号后，新创建的分布式信息队列才可以被远程节点所使用。

在分布式信息队列上使用标准信息队列函数的时候，超时参数仅指定了远端信息队列的等待时间，而没有指定节点间传送时间的机制。当使用专门为分布式信息队列设定的发送、接收和关于信息数量的函数（msgQDistSend( )，msgQDistReceive( )，及 msgQDistNumMsgs( )）

时，用户也可以利用一个附加的超时参数 *overallTimeout* 来说明传送时间。

> ⚠ **注意**：对于这个释放过程，用户不能删除一个分布式信息队列。没有函数 msgQDistDelete( )，用分布式信息队列 ID 号对 msgQDelete( )的调用总是返回一个错误信息。

图 10-4 说明了发送和接收操作。

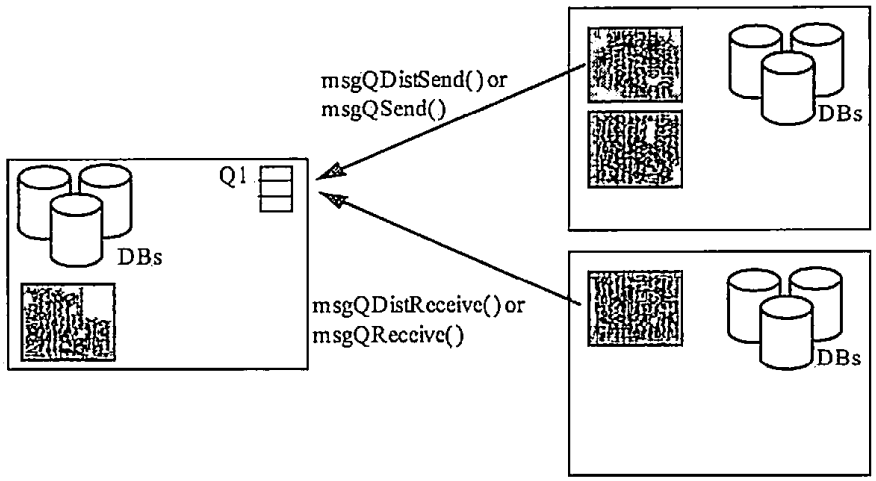


图 10-4　远端分布式信息队列发送和接收消息

然而，发送和接收操作发生之前，节点 1 上的任务一定要创建一个分布式信息队列。远端信息队列 Q1 已经通过调用 msgQDistCreate( )创建了节点 1 上的任务 1。然后通过调用 distNameAdd( )，Q1 被增加到分布式名称数据库中。任务 t2 和 t4 也已经调用函数 distNameFind( )从分布式名称数据库中得到了 Q1 的远端信息队列的 ID。有了这个数据，节点 2 上的任务 2 可以调用标准函数 msgQSend( )或指定的 VxFusion 的函数 msgQDistSend( )来给 Q1 发送消息。相似的，节点 3 上的任务 4 也可以调用标准函数 msgQReceive( )或指定的 VxFusion 函数 msgQDistReceive( )来接收一个消息。

关于分布式信息队列函数的详细信息，请看“*VxWorks API*”的相关条目。

1. 发送限制

本地发送——也就是在一个单独节点上的发送行为——在分布式信息队列中以与标准信息队列相同的方式发生，因此在本书中没有详细讨论。然而，使用函数 msgQDistSend( )往远程信息队列发送信息时可以导致不同的结果，这取决于为超时参数指定的值。图 10-5 列出了正被传送到远程节点的三个消息的例子。它要执行两个线程：局部节点等待发送动作状态，远端节点等待将此数据装入到分布式信息队列中。如果使用 msgQDistSend( )，两个线程均用超时设定来控制。具体的发送过程见图 10-5 所示。

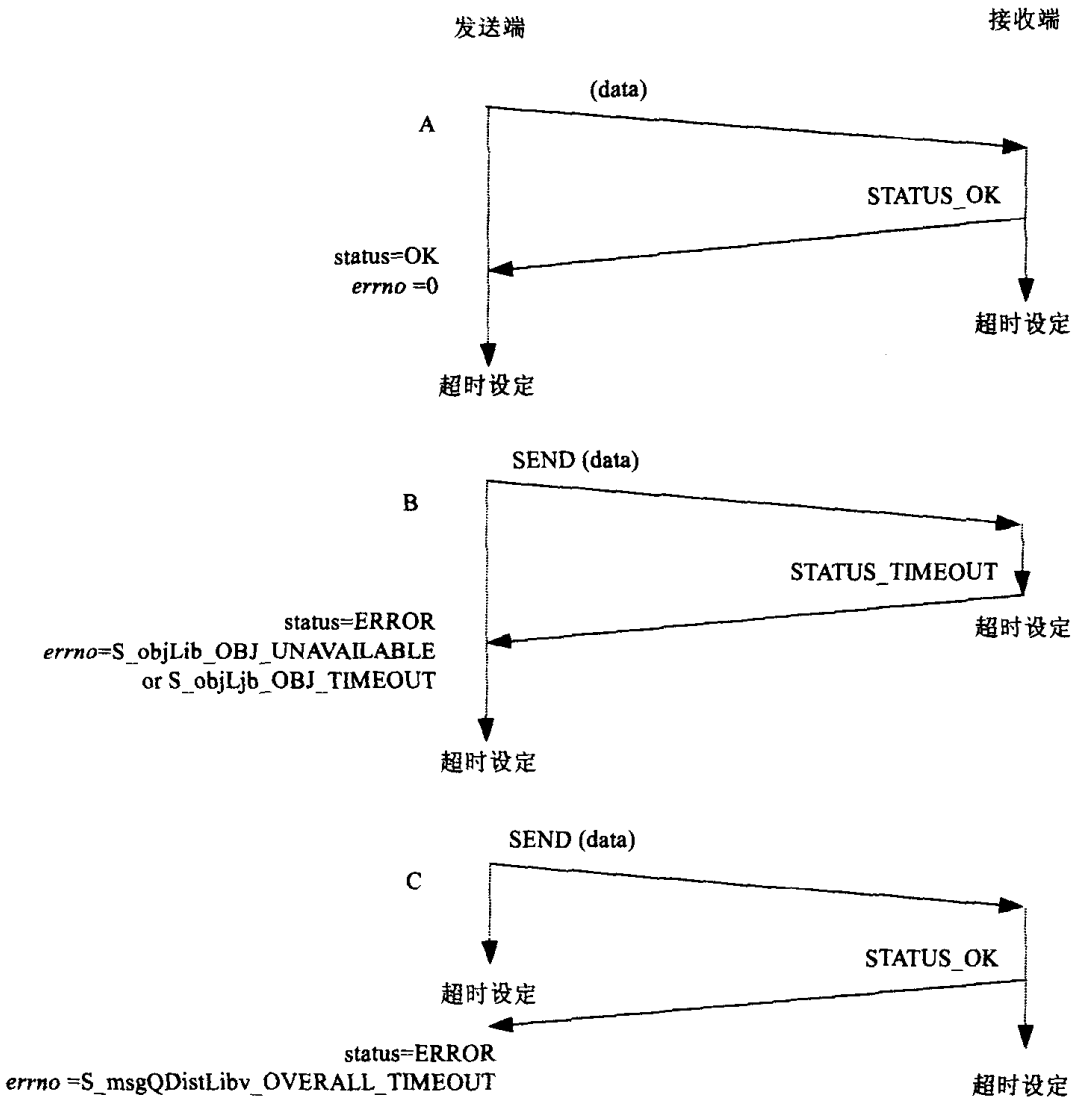


图 10-5　具体发送过程

远端超时设定指的是参数 *msgQTimeout*，也就是远端信息队列等待的脉冲数。局部超时设定指的是参数 *overallTimeout*，也就是等待的全部脉冲数，其中包括传送时间。

在例子 A 中，在传送行为完成以及状态返回之前没有发生超时，因此本地节点收到 OK 后就知道信息已经被收到。在 B 和 C 中，一个超时设定到达后，发送程序就返回 ERROR，error 变量就被设置用来指示已经超时。

在例子 B 中，本地节点意识到超时；然而，例子 C 中在状态接收到之前本地节点就已经超时。在这种情况下，局部节点不知道发送是否已经完成。在例子 C 中，即使本地操作失败，此信息也被增加到远端节点上，这样两个节点就有了关于系统状态的两个不同结果。为了避免这个问题，可以把 *overallTimeout* 的值设定到足够大，这样此状态就会总被接收。

因为 msgQSend( )永远等待远端的回应，所以可以用 msgQSend( )防止像例子 C 中出现的情况的发生。

关于往不可到达节点上发送信息的局限性，请参看“检测缺少的接收节点”。

2. 接收限制

对于信息的本地发送来说，分布式信息队列的本地接收与标准信息队列的本地接收以同样的方式进行。

正如发送限制中的细节部分所示，当使用函数 msgQDistReceive( )从远程信息队列中接收消息时可以出现不同结果，这取决于指定的超时参数的值。图 10-6 列出了正被一个节点接收的三个消息的例子。其中存在两个执行线程：局部节点等待将被接收的数据，远程节点等待数据到达分布式信息队列，两个线程都用超时设定来控制。

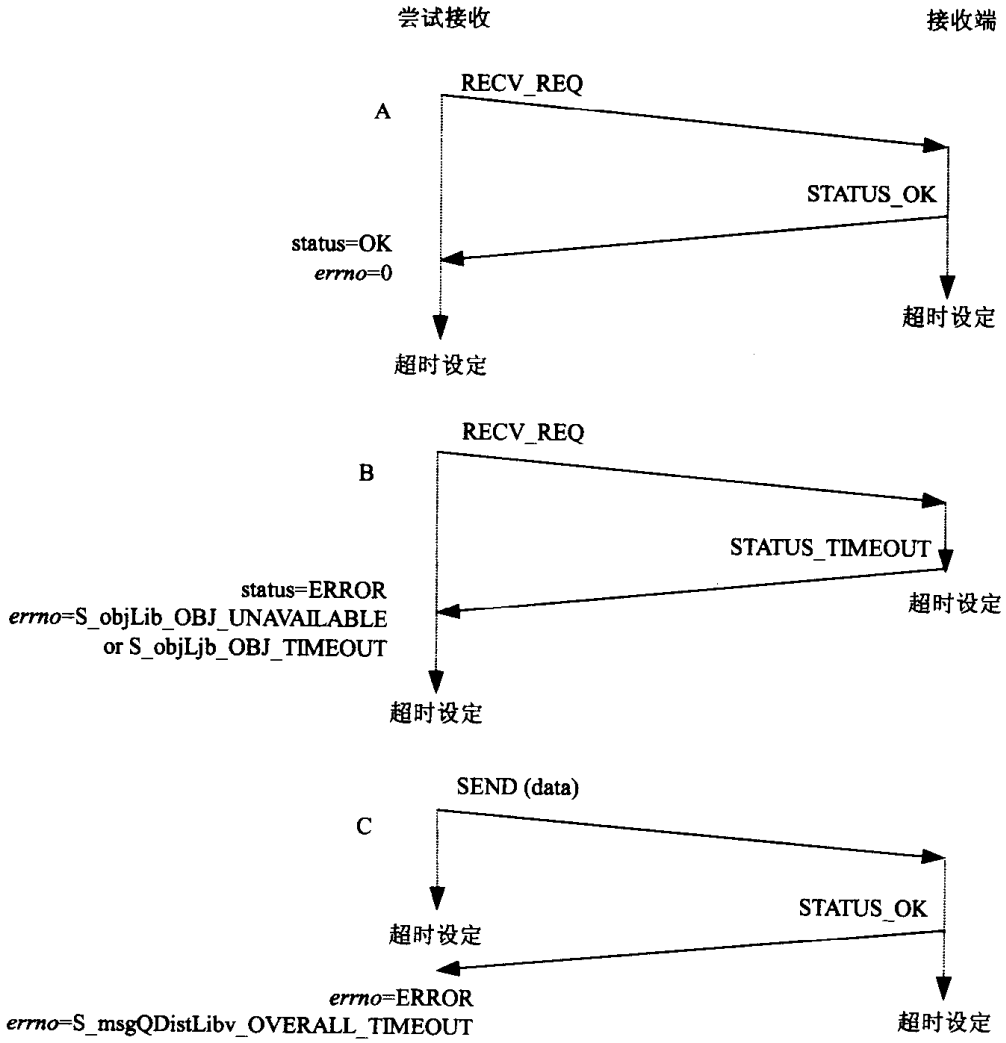


图 10-6　接收的具体内容

远端超时设定用参数 *msgQTimeout* 来指定，也就是远端信息队列等待的时钟数。局部超时设定用参数 *overallTimeout* 指定，即等待的全部时钟数，其中包括传送时间。

例子 A 说明了不带超时设定的成功的接收过程。从信息队列中接收消息的请求被传送

到远端节点，其结果在其中任一线程超时之前被返回。

例子 B 中远端存在超时，但是状态回应在到达超时之前被返回到本地节点。接收程序返回一个错误值，并且设定变量 *errno* 来指示超时。这样两端都意识到接收失败并且拥有同样的远端信息队列的状态结果。

例子 C 中本地节点想要从远端节点接收一个消息。但是在一个回应到达之前，参数 *overallTimeout* 达到最大值。因为尽管从远端队列中成功移走消息，本地端却认为操作失败，所以本地和远端就以关于远端信息队列状态的不同结果而结束。

为了避免这个问题，把参数 *overallTimeout* 的值设定的比通常从远端接收的回应值足够的大，或者使用 msgQReceive( )，因为它永远等待远端的回应。

3. 显示分布式信息队列的内容

可以使用标准信息队列函数 msgQShow( )来显示分布式信息队列的内容。

以下例子显示了一个局部分布式信息队列的调用 msgQShow( )的输出结果。

```
[VxKernel]-> msgQShow 0xffe47f
Message Queue Id     : 0xffe47f
Global unique Id     : 0x930b267b:fe
Type                 : queue
Home Node            : 0x930b267b
Mapped to            : 0xea74d0
Message Queue Id     : 0xea74d0
Task Queueing        : FIFO
Message Byte Len     : 1024
Messages Max         : 100
Messages Queued      : 0
Receivers Blocked    : 0
Send Timeouts        : 0
Receive Timeouts     : 0
value = 0 = 0x0
```

以下例子显示了不同主机上相同队列的输出结果。

```
[VxKernel]-> msgQShow 0x3ff9b7
Message Queue Id     : 0x3ff9b7
Global unique Id     : 0x930b267b:fe
Type                 : remote queue
Home Node            : 0x930b267b
value = 0 = 0x0
```

10.3.6 操作组信息队列

VxFusion 使用组信息队列支持到一组分布式信息队列的信息的多点广播。组信息队列是一个虚拟信息队列，它接收发送给它的信息并将信息传送给所有成员队列。

表 10-9 列出了可以用来处理分布式信息队列的程序。

表 10-9 分布式组信息队列

函　数	功　能
msgQDistGrpAdd( )	往一个组中增加一个分布式信息队列
msgQDistGrpDelete( )	从一个组中删除一个信息队列
msgQDistGrpShow( )	显示组信息队列的信息

可以用函数 msgQDistGrpAdd( )创建并增加组信息队列。当调用 msgQDistGrpAdd( )时，如果组信息队列不存在，那么就要创建一个信息队列，并且这个指定的队列就变成此组中的第一个成员。如果组信息队列已经存在，那么这个指定的分布式信息队列仅被简单地增为一个成员。函数 msgQDistGrpAdd( )总是返回组信息队列的 ID 号。

如果用户希望一个分布式信息队列由多个组共享，那么一定要调用 msgQDistGrpAdd( )来声明每一个附加的成员关系。

这里仅支持发送和显示组信息队列。尝试从组信息队列中接收它或查询它以得到信息数目都是错误的。

⚠ **注意**：尽管有一个 msgQDistGrpDelete( )函数，但不能从组信息队列中删除分布式信息队列。函数 msgQDistGrpDelete( )总是返回 ERROR 值。

系统中已经创建的所有组信息队列的信息和它们的本地附加成员都被存储在分布式组数据库的本地备份中。函数 msgQDistGrpShow( )显示了分布式组数据库中的所有组以及它们的局部附加成员或者是一个指定的组和它的局部附加成员。关于使用 msgQDistGrpShow( )的更多信息，请看“显示关于分布式组信息队列的信息”。

请看图 10-7 的例子，通过调用 msgQDistCreate( )已经创建了分布式信息队列 Q1、Q3、Q4、Q5 和 Q6，并且它们已经通过调用 distNameAdd( )而增加到分布式名称数据库中。通过调用 msgQDistGrpAdd( )创建信息队列的同样的任务也为每一个成员把 Q3、Q4 和 Q5 增加到组信息队列 Q2 中。信息队列 Q1 和 Q6 没有被加到 Q2 组中。第一次对 msgQDistGrpAdd( )的调用在每一个节点的分布式组数据库中都创建了 Q2 作为入口。（在图 10-7 中，三个数据库－分布式节点、名字和组－通过三个柱面来象征，前面的那一个代表了组数据库并且显示了 Q2 入口。

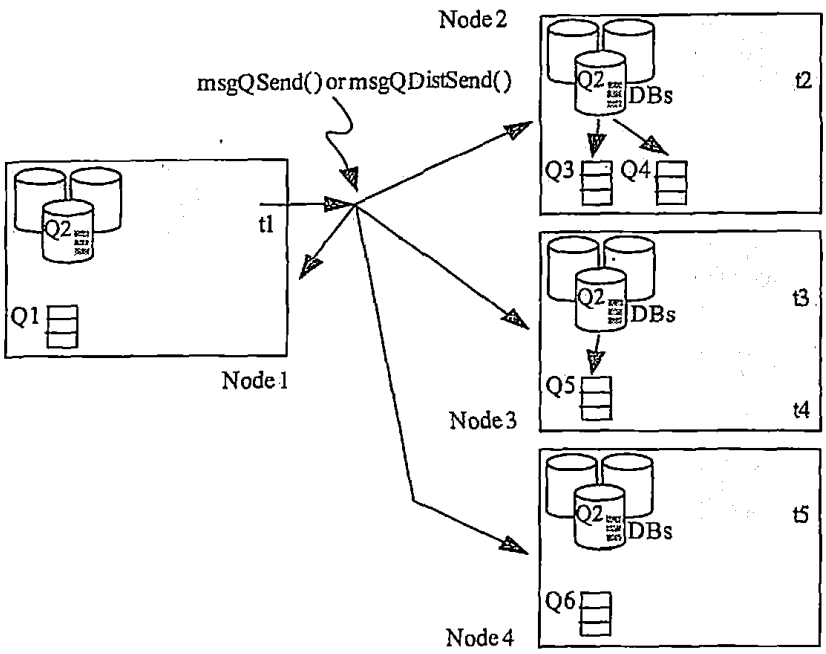


图 10-7　组信息队列

⚠ **注意**：分布式信息队列成员 Q2 不必通过创建它们的任务而增加到组中。实际上，只要那个队列的 ID 号局部可知或可以从分布式名称数据库中得到，任何一个任务都能把任一分布式信息队列增加到组中。

图 10-7 中，在组信息队列 Q2 被建立的情况下，当任务 t1 给组 Q2 发送一个消息时，此消息被传送到系统中的其他所有节点上。每一个节点都用分布式组数据库来识别组中的成员并把消息传送给它们。在这个例子中，此信息被传送给成员 Q3、Q4 和 Q5，但是不给非成员 Q1 和 Q6。

关于分布式组信息队列的详细信息，请看“*VxWorks API Reference*”的相关条目。

1. 显示关于分布式组信息队列的信息

函数 msgQDistGrpShow( )显示了组数据库中的所有组及它们的局部附加成员或者一个指定的组及其局部附加成员。

以下输出说明了使用不带参数的 msgQDistGrpShow( )的输出结果。

```
[VxKernel]-> msgQDistGrpShow(0)
NAME OF GROUP          GROUP ID    STATE   MEMBER ID TYPE OF MEMBER
-------------------- ---------- ------- ---------- ----------------------
grp1                   0x3ff9e3  global 0x3ff98b distributed msg queue
                                        0x3ff9fb distributed msg queue
grp2                   0x3ff933  global 0x3ff89b distributed msg queue
```

```
                                        0x3ff8db distributed msg queue
                                        0x3ff94b distributed msg queue
value = 0 = 0x0
```

以下调用说明了使用带有字符串参数 grp1 的函数 msgQDistGrpShow( )。

```
[VxKernel]-> msgQDistGrpShow("grp1")
NAME OF GROUP          GROUP ID   STATE  MEMBER ID TYPE OF MEMBER
------------------- ---------- ------- ---------- ----------------------
grp1                   0x3ff9e3  global  0x3ff98b distributed msg queue
                                         0x3ff9fb distributed msg queue
value = 0 = 0x0
```

10.3.7　操作适配器

不管是不是使用了特殊的传送方式，适配器给 VxFusion 提供了一个统一的接口。表 10-10 列出了关于适配器的惟一的 API 调用，即 distIfShow( )。

表 10-10　适配器函数

函　数	功　能
distIfShow( )	列出关于已安装的接口适配器的信息

关于如何写用户自己的 VxFusion 适配器的信息，请参看“10.7　设计适配器”。关于适配器的详细信息，请看“*VxWorks API Reference*”中的相关内容。

以下例子示范了使用 distIfShow( )

```
[VxKernel]-> distIfShow
Interface Name                 : "UDP adapter"
MTU                            : 1500
Network Header Size            : 14
SWP Buffer                     : 32
Maximum Number of Fragments    : 10
Maximum Length of Packet       : 14860
Broadcast Address              : 0x930b26ff
Telegrams received             : 23
Telegrams received for sending : 62
Incoming Telegrams discarded   : 0
Outgoing Telegrams discarded   : 0
```

关于如何更换已安装的接口适配器或是改变它的值，请看“10.3.3　配置 VxFusion”。

10.4　系统局限性

1. 中断服务程序的局限性

与标准信息队列不同，分布式目标对象不能在中断级使用。从 ISR 中不能调用使用分布式对象的程序。可以用 ISR 来处理与外部事件相关的实时性过程，因此在中断时使用分布式目标不合适。在多处理器系统中，应该在发生实时中断的 CPU 上运行与事件相关的实时性处理过程。

2. 检测缺少的接收节点

当在节点上创建分布式信息队列时，它的创建也被通知到其他节点。然而，如果信息队列创建于其上的节点或者发生冲突或者被重启，那么对于其他节点来说就没有检测到信息队列丢失的简单方法了。即使重启时系统重建了此队列，名称数据库的入口也没有被改变。结果别的节点可能仍使用无效的信息队列的 ID 并且挂起在一个永远不会被发送的接收通知的状态。因此，当希望更多的数据时，接收节点上的应用程序要设定超时值。

10.5　节 点 启 动

运行时 VxFusion 系统可以用来支持新节上点的补充内容，同时可以承受一个甚至多个节点的错误。这样就可以通过节点结合过程在运行时增加新节点的处理能力，而且通过复制节点启动过程中移入到其中的数据库，同时也使它可以具有承受一个甚至多个节点的错误的能力。此部分详细讨论了节点启动过程。

表 10-11　节点启动状态

状　态	行　为
启动的	定义系统中别的节点以及确定“父节点”
网络的	从“父节点”更新数据库
可操作的	通知别的节点，此节点已经启动并运行

表 10-11 列出了它的三个状态，其中的每一状态此部分都有详细说明。图 10-8 图示说明了节点启动过程。（为了简化表 10-8，它并不显示确认发送信息。）

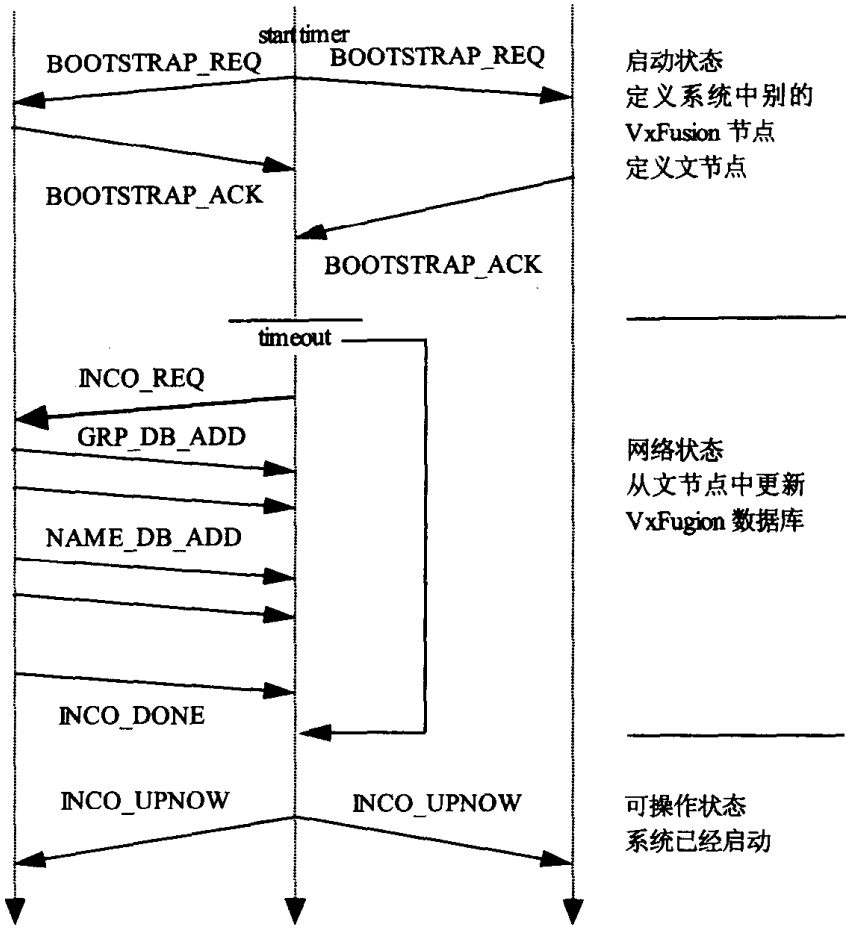


图 10-8　启动系统

1. 启动状态

当一个节点第一次初始化 VxFusion 时，它就广播一个自举请求信息（BOOTSTRAP_REQ），用来把别的激活的 VxFusion 节点安装在网络上。收到自举请求信息的节点用一个自举确认信息（BOOTSTRAP_ACK）来回应。

这样有一个节点要发送给另一节点，第一个自举请求确认回应，它就变为本地节点的父节点。在图 10-8 中，节点 1 是父节点，因为它的自举确认信息被最先接收。父节点的目的是帮助本地节点更新其数据库。如果在一段指定的时间中没有收到回应信息，节点就假定它是在网络上出现的第一个节点。

父节点一经定位或是一旦认为某一个节点是网络中的第一个节点，此节点就从启动状态变为网络状态。

2. 网络状态

某个父节点被定位后，局部节点就通过发送一个组合请求信息（INCO_REQ）来请求父节点更新其数据库，其实质是父节点改变局部节点的名字和组数据库，这些更新在图 10-8 中用箭头 GRP_DB_ADD 和 NAME_DB_ADD 来指出。完成以后，父节点要告知接收节点：

通过发送组合完成信息（INCO_DONE）更新数据库的过程已经完成。

数据库更新完成以后，节点就切换到可操作状态。如果没有父节点，节点就从启动状态直接切换到可操作状态。

3. 可操作状态

一个节点移动到可操作状态时，VxFusion 已被完全初始化并在其上运行。这样节点就可以广播“up now”组合信息（INCO_UPNOW）来告知系统中的其他节点当前它已被激活。

10.6　报文和消息

10.6.1　报文与消息比较

由于 VxFusion 通过网络传送数据，有可能单个消息的总的字节数过大而不能作为一个单独的单元来传送，这样它必须被分成几个小的部分，这些 VxFusion 消息段被称为是报文。报文是可以在节点间传送的最大数据包。

报文是在传送过程中由一些字节组成的很小的要传送的数据块。传送器的 MTU 大小（初始化时由适配器提供给 VxFusion）定义了报文的大小。

用户发送　个消息时，系统把它分成　个或几个报文。图 10-9 说明了一个报文及其组成。

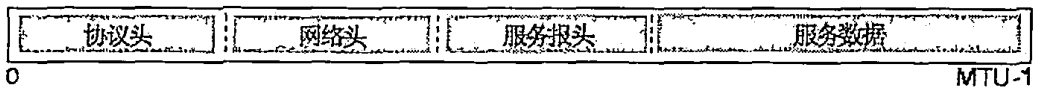


图 10-9　报文缓冲器

1. 协议报头

传送方式定义了协议头，它是由适配器提供的数据来构成的。协议头的内容因协议不同而有所区别，但是如果传送方式支持优先级，它也可以包括如源地址、目的地址和优先级等内容。

2. 网络报头

适配器定义并创建了网络头。关于网络头和它的详细内容请看“10.7.1　设计网络报头”。

3. 服务报头

VxFusion 定义并创建了服务报头。服务报头是一个定义内部服务，即发送信息及信息

类型的小的报头。

4. 服务数据

服务数据是正在被发送的数据。当往远端信息队列或信息队列组发送消息时，此数据可以是将被发送的全部信息或部分信息。当信息大小超过了报文中为服务数据分配的空间大小时，它就变为部分信息。

用拥有小的 MTU 的传送机工作的时候，由于可能需要大量的报文，VxFusion 并不认识单独的数据报，相反只认识整个信息。当传送出现错误或报文丢失时，一定要重新传送整个信息。

10.6.2 报文缓冲器

当 VxFusion 需要给远端节点发送消息时，它首先把消息分解成报文并存储在预先分配的报文缓冲区中。图 10-10 显示了报文缓冲区和它的各部分内容。

VxFusion 每次给适配器发送一个报文缓冲，它由网络报头、服务报头和服务数据而组成相应的报文。最后，适配器将报文发出。

在接收节点上，当适配器接收到一个数据报时，它就要在数据报中重建一个报文缓冲。重建报文缓冲区之后，适配器把它发送给 VxFusion。

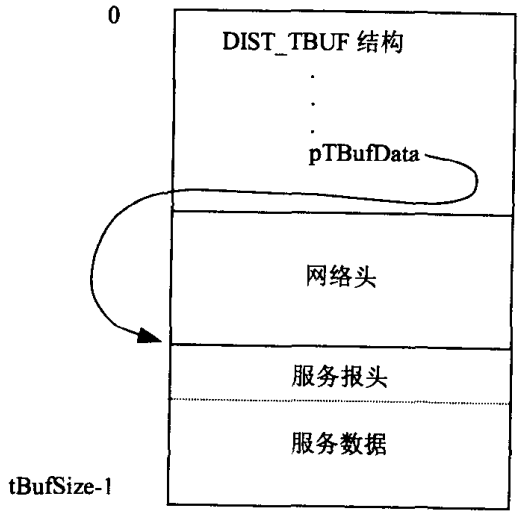


图 10-10 报文缓冲

DIST_TBUF 的结构成员 pTBufData 包含了服务报头的开始地址。如果要访问网络报头，要从这个地址中减去网络头的大小。

为了在远端节点重建一个报文缓冲，发送此报文以前，某些报文缓冲的内容一定要通过适配器被复制到网络报头中。关于一定要被复制的内容，请看“10.7.1 设计网络报头”。

10.7　设计适配器

这部分描述了如何为 VxFusion 组件写适配器。

适配器是能提供 VxFusion 节点间通信的软件机制。适配器位于 VxFusion 和单个通信传送方式之间，如图 10-1 所示。传送方式可以是高层的通信协议（如 UDP）或低层的通信或总线驱动程序，如以太网驱动程序。当一个消息被传送到远端节点时，VxFusion 给适配器传送一个报文缓冲，此适配器就通过支持的传送方式给远端节点传送数据。相似地，当从远端节点接收一个消息时，适配器要由输入数据重建一个报文缓冲，并把此缓冲区发送给 VxFusion。

一个适配器一定包括以下内容：

- 一个传送报文缓冲内容的网络报头
- 初始化功能
- 启动程序
- 发送功能
- 输入功能
- I/O 控制功能

以上内容在这部分中都有详细描述。

注意：在此版本中 VxFusion 只支持一个适配器，所以一次也只支持一种传送方式。

10.7.1　设计网络报头

报文缓冲区本身并没有被传送到远端节点，它仅从局部节点传送到适配器的位置。为了将数据从适配器移动到远端节点，VxFusion 使用适配器的网络报头。网络报头是一个存储来自某些报文缓冲区并要求在远端节点上重建报文缓冲区的数据的结构。实际上由适配器到远端节点传送的是由网络头和由报文缓冲区指向的数据。

其性能和信息大小与网络头的大小直接相关。因为网络头由用户设计，所以用户可以通过指定规定网络头报文的部分来影响其性能，同时用户也可以交替地使用信息。

比如，对于一个仅有小的传输单元的传送器，用户可能想要通过一个小的网络头来使传送数量最大化。然而，给一个区域减少比特数（像片断序号数）同时也减小了可以被发送的消息的大小。另一方面，对于有大的传输单元的传送器来说，网络报头仅组成了整个数据报的很小的一部分；适配器可以在不明显影响吞吐量的情况下在报头部分使用一个比

较大的片段序号数。

报文缓冲结构 DIST_TBUF 如下所示：

```
typedef struct  /* DIST_TBUF */
    {
    DIST_TBUF_GEN          tBufGen;     /* TbufGen 结构*/
    void                   *pTBufData; /*指向数据的指针*/
/* 要求在远端创建报文缓冲的内容 */
    uint16_t               tBufId;      /*包的 ID 号*/
    uint16_t               tBufAck;     /*最近准确无误接收并确认的包的 ID*/
    uint16_t               tBufSeq;     /*段的序列号*/
    uint16_t               tBufNBytes; /*非网络报头的数据字节的数目*/
    uint16_t               tBufType;    /*报文类型*/
    uint16_t               tBufFlags;   /*报文标志*/
    } DIST_TBUF;
```

用户一定要在与以下 DIST_TBUF 的内容相应的指定适配器的网络头中创建内容。除了开始的两个内容 *tBufGen* 和 *pTBufData* 以外，其他的都要求在远端节点上重建报文缓冲区。

用户可以按照传送器的需要给网络报头创建和增加内容。比如，信息优先级是函数 distIfXxxSend( ) 和 distNetInput( )的一个参数，但并不是所有的传送器都支持优先级。如果传送器支持优先级，优先级就在协议头中传送并且可在远端获得。如果传送器不支持优先级并且用户想保存它，那么用户应该在网络头中增加一项用来给远端节点传送优先级的值。

10.7.2　写一个初始化程序

适配器通过对初始化程序的自动调用来实现初始化，用户可以写一些代码来完成这个工作。当用装有 VxFusion 的 VxWorks 映像来启动目标机时，初始化整个 VxFusion 的函数 distInit( )被自动调用。

```
STATUS distInit
    (
    DIST_NODE_ID    myNodeId,        /*此节点的节点 ID*/
    FUNCPTR         ifInitRtn,       /*接口初始化程序*/
    void            *pIfInitConf,    /*指向接口配置的指针*/
    int             maxTBufsLog2,    /*报文缓冲区的最大数目*/
    int             maxNodesLog2,    /*节点数据库中的节点的最大数*/
    int             maxQueuesLog2,   /*节点上队列的最大数目*/
```

```
    int             maxGroupsLog2,   /*数据库中组的最大数目*/
    int             maxNamesLog2,    /*命名数据库中最大的附加数目*/
    int             waitNTicks       /*引导程序执行时等待的时钟数目*/
    )
    {
    }
```

在用户指定的名字为 Xxx 的适配器上，参数 *ifInitRtn* 指定了适配器初始化函数 distIfXxxInit( )。

⚠ **注意**：不要直接调用 distInit( ) 或 distIfXxxInit( )。

用户可以将 distIfXxxInit( )的形式基于以下源代码：

```
STATUS distIfXxxInit
    (
    void      *pConf,      /*指向配置数据的指针*/
    FUNCPTR   *pStartup    /*指向启动程序的指针*/
    );
```

调用 distIfXxxInit( )时应该带有以下参数：

pConf

指向由 distInit( )的参数 *pIfInitConf* 指向的接口配置数据的指针。

PStartup

指向适配器初始化程序返回之后设置的启动程序的指针。

适配器初始化程序应该完成以下操作：

- 使启动程序指针指向适配器启动程序
- 设置 DIST_IF 结构的内容

结构 DIST_IF 是一个给 VxFusion 提供传送独立性的机制：不管适配器是否正在使用，此结构都包含它的形式。结构 DIST_IF 由将被使用的、定义的适配器和传送信息的操作环境配置两部分内容组成。关于使用 DIST_IF 的详细信息，请参看“使用 DIST_IF 结构”。

尽管用户不能直接调用 distInit( ) 和 distIfXxxInit( )，但可以在 usrVxFusion.c 中修改 VxFusion 的启动代码以改变 VxFusion 的初始化过程。

如果用户需要指定附加信息来初始化适配器，那么可以改变 distInit( )的 *pIfInitConf* 参数来提供那个信息。其参数 *pIfInitConf* 被作为 distIfXxxInit( )的 *pConf* 而传递。要想保护由 *pConf* 指向的信息，用户应该在适配器中将它的值复制到一个更加稳定的结构中。如果适配器不需要额外的配置信息，那么 *pConf* 应该被忽略。

如果初始化成功，适配器初始化程序就返回一个 OK 值；如果失败，则返回 ERROR。

1. 使用 DIST_IF 结构

可以用结构 DIST_IF 把关于适配器的详细信息传送给 VxFusion，这样 VxFusion 可以把消息分成大小合适的报文，以便在传送器上传送。

DIST_IF 结构有以下声明：

```
typedef struct   /* DIST_IF */
    {
    char         *distIfName;          /*接口的名字*/
    int          distIfMTU;            /*接口的传送部分的 MTU 大小*/
    int          distIfHdrSize;        /*网络头大小*/
    DIST_NODE_ID distIfBroadcastAddr; /*接口的广播地址*/
    short        distIfRngBufSz;       /*变化的窗口协议中的缓冲区的数目*/
    short        distIfMaxFrags;       /*信息可以被分成的最大片断数目*/
    int     (*distIfIoctl)(int fnc, ...); /*适配器 IOCTL 函数*/
    STATUS (*distIfSend)(DIST_NODE_ID destId, DIST_TBUF *pTBuf, int prio
);                                    /*适配器的发送函数*/
    } DIST_IF;
```

关于 DIST_IF 的内容定义如下：

distIfName

此内容是接口或适配器的名字。调用 distIfShow( )时它就被显示出来。

DistIfMTU

此内容指定了传送器的 MTU 大小。

DistIfHdrSize

此内容说明了指定适配器的网络报头大小。如果它们的 MTU 比较小，某些适配器可以使用小的报头使每个报文的数据数量最大化。拥有较大的 MTU 的传送器的适配器可以使用较大的报头来确定较大的信息。

DistIfBroadcastAddr

此内容指定了传送器的广播地址。如果传送器不支持广播操作，那么一定要提供一个虚拟广播地址，并且不管什么时间，当它接收到一个目的地址为虚拟广播地址的信息时，传送器一定要模拟广播操作。模拟广播操作的一种方式是用点对点地址方式把消息发送给系统中的其他节点。

distIfRngBufSz

此内容指定了在变化的窗口协议中用到的成分数量大小。较大的数目意味着为等待确认的消息而保持在环形缓冲器的报文有较大数目。

distIfMaxFrags

此内容指定了一个消息可以被分块的最大数目。可以被传送的消息的最大大小是：

最大大小=（分块数）*（MTU 大小-网络报头大小-服务头大小）

服务头大小取决于正被传送的 VxFusion 消息的类型。比如 BOOTSTRAP_REQ 或 BOOTSTRAP_ACK。

DistIfIoctl

这是适配器接口的 ioctl 函数。

DistIfSend

这是适配器接口的发送程序。

10.7.3　写一个启动程序

适配器启动程序应该由适配器初始化程序返回给 distInit( )。用户可用下列源代码为适配器接口 Xxx 写一个启动程序。

```
STATUS distXxxStart
    (
     void *    pConf      /*指向配置函数的指针*/
    );
```

这个启动程序在 VxFusion 的网络层被初始化后由 distInit( )调用。启动程序应该包括输入任务，还包括用来通过在需要的传送器上传送和接收报文的任何初始化或启动操作。比如，此时 UDP 适配器应该创建一个用来通信的套接字。

如果操作成功，启动程序应该返回 OK，失败则返回 ERROR。

10.7.4　写一个发送程序

用户可以用下列源代码为写适配器接口 Xxx 写一个发送函数 distIfXxxSend( )：

```
STATUS distIfXxxSend
    (
     DIST_NODE_ID    nodeIdDest,   /*目的节点*/
     DIST_TBUF       pTBuf,        /*要发送的 TBUF*/
     int             priority      /*包优先级*/
    );
```

disIfXxxSend( )的参数按以下定义：

nodeIdDest

此参数是目的节点或广播节点的惟一标识。

PTBuf

此参数是获得发送的缓冲区。

Priority

此参数说明了消息的优先级，传送器对它可以支持，也可以不支持。

发送程序的目的是接收从 VxFusion 传送的报文缓冲区，并通过传送器把相关报文发送出去。

发送程序应该完成以下任务：

- 增加统计表（distStat.ifOutReceived++）。
- 通过作为参数而传递的报文缓冲区的数值来确定并填充预分配的网络报头。网络报头应该用网络字节按顺序填充，这样即使远端使用了不同的字节顺序，它们在远端也可以被正确解码。（关于哪些报文缓冲区的内容一定要被复制的信息，请看“10.7.1　设计网络报头”。）
- 填满任一可能需要填充的附加网络头的内容（比如优先级）。
- 发送报文。

如果操作成功，发送程序应该返回 OK 值，失败就返回 ERROR。

10.7.5　写一个输入程序

用户可以用下列源代码为适配器接口 Xxx 写一个输入函数 disIfXxxInputTask( )：

```
void distIfXxxInputTask( );
```

输入程序的目的是通过调用 distNetInput( )读取一个报文并把它发送给 VxFusion。关于它的详细信息，请看“the *VxWorks API Reference*”中的关于 distNetInput 相关内容。

输入程序应该监听或等待来自传送器的输入报文。接收一个报文时，应该增加一个统计报表 distStat.ifInReceived，然后应该检测报文以确定它是否比网络报头长。如果不是，则报文太小可以被忽略，在这种情况下，distStat.ifInLength 和 distStat.ifInDiscarded 的内容也应该被加进去。

如果用户的传送器没有丢弃它本身发送的广播信息包，那可以用输入程序将基于传送方式的广播信息包选出来。

在输入程序放弃了任一基于传送器的备份或残缺的报文之后，输入报文就被认为是正确的（尽管以后还要检查）。这样就可以分配一个报文缓冲区，并且报文的内容可以被复制进去。没有被传送的报文缓冲区中的非数据部分的内容如下：

- tBufId
- tBufAck
- tBufSeq
- tBufNBytes
- tBufType
- tBufFlags

这些内容可以使用网络报头来重新构建。在重建过程中，它们应该从网络命令转换回

主机命令。

在报文缓冲区被重建并且报文的非报头部分期望的字节数已知的情况下（由 tBufNBytes 得来），增加到网络报头的报文长度与 tBufNByte 的大小差不多。如果长度不匹配，报文就应该被放弃，并且增加统计表 distStat.ifInLength 和 distStat.ifInDiscarded。

如果长度匹配，报文应该通过调用 distNetInput( )继续向前传送。

10.7.6　写一个 I/O 控制程序

用户可以用以下源代码来给适配器接口 Xxx 写一个 I/O 控制函数 disIfXxxIoctl( )。

int distIfXxxIoctl（int func,）;

用户可以定义一定要由适配器支持的控制函数。

如果操作成功，此函数应该返回正在执行的 I/O 控制函数的返回值，如果操作失败，就返回 ERROR。如果没有提供控制函数，函数 distIfXxxIoctl( )应该返回 ERROR。

第 11 章　共享内存对象

可选的 VxWorks 组件

11.1　简　　介

VxMP 是一个可选的 VxWorks 组件，它提供共享内存对象，这些对象有助于高速同步和运行在分布 CPU 上的任务间通信。关于如何安装 VxMP，请参看“*Tornado Getting Started*”。

共享内存对象是一类可以被运行在不同处理器上的任务访问的系统对象。它们之所以被称为共享内存对象是因为对象的数据结构一定要位于可以由所有的处理器访问的存储区中。共享内存对象是局部 VxWorks 对象的扩展，而局部对象仅可以由单个处理器上的任务访问。VxMP 提供了三种共享内存对象：

- 共享信号量（二进制计数）
- 共享消息队列
- 共享内存的分区（系统及用户创建的分区）

共享内存对象有以下优点：

- 提供允许共享内存对象可以由操作局部对象的、同样的程序操作的透明接口。
- 高速内部处理器间通信——不需传递无用的包。
- 共享内存可以位于双端口的 RAM 上或一个独立的存储板上。

组件 VxMP 由以下部分组成：名称数据库（smNameLib）、共享信号量（semSmLib）、共享消息队列（msgQSmLib）、共享内存分配器（smMemLib）。

本章详细说明了每一共享内存对象以及其内部需要考虑的问题，然后提供了配置和解决问题的方法。

11.2　使用共享内存对象

VxMP 提供了一个透明接口，可以在多处理器系统和单处理器系统上使用共享内存对象来使执行代码变得更容易。对象被创建后，可以用任务在相应的本地对象上执行相同程序来操作共享对象。比如，共享信号量，共享消息队列等，并且共享存储分区与它们的本地对应部分有相同的语法和接口。程序如 semGive( )，semTake( )，msgQSend( )，msgQReceive( )，memPartAlloc( )和 memPartFree( ) 都可以操作局部和共享对象，仅仅是创

建程序不同而已。在系统配置、初始化和创建对象仅有微小变化的情况下，它允许程序运行在单处理器或多处理器环境中。

单处理器系统上可以运行所有的共享内存程序。这对于在多处理器配置端口化之前测试应用程序是很有用的。然而，对于仅在局部使用的对象来说，局部对象总是提供最好的性能。

在共享内存工具初始化以后（关于初始化区别参见“11.4 配置”），所有的处理器都被一致对待。任一 CPU 上的任务可以创建和使用共享内存对象。从共享内存对象的角度来说，处理器的优先级都是一样的[①]。

使用共享内存的系统可以包括由软件支持的结构组合。这允许程序使用不同类型的处理器并且仍使它们通信。然而，在处理器存在不同的按字节排序的系统上，用户一定要调用宏 ntohl 和 htonl 来交换应用程序的共享数据（参见“VxWorks Network Programmer's Guide: TCP/IP Under VxWorks”）。

一个对象被创建时，被返回对象的 ID 以标识它。为了使不同 CPU 上的任务能够访问共享内存对象，它们首先一定要得到这个 ID。不管 CPU 如何，一个对象的 ID 总是相同的，这就允许 ID 可以通过共享消息队列、共享内存中的数据结构或名称数据库来传送。

此章的其余部分中，除非特别指明外，要讨论的系统对象均指的是共享对象。

11.2.1　名称数据库

名称数据库允许把任何数值跟任何名字联系起来，比如用一个特殊的名字表示一个共享内存对象的 ID。它可以传送或广播一个共享内存块的地址和对象类型。名称数据库提供了名字到数值以及数值到名字的转换，允许通过数值或名字来访问数据库中的对象。尽管广播一个对象的 ID 还有别的方法，但名称数据库的确是一个很便捷的途径。

典型的，创建对象的任务也可以通过名称数据库的方法来广播对象的 ID。通过给数据库增加新的对象，此任务用一个名字来连结对象的 ID。其他处理器上的任务可以在数据库中查找其名字以得到对象的 ID。任务拥有了 ID 后，可以用它来访问对象。比如，CPU 1 上的任务 1 创建了一个对象。对象的 ID 就通过创建程序被返回，并且以名字 myObj 进入名称数据库。为了使 CPU 0 上的任务 t2 能够操作此对象，它首先通过在名称数据库中查找名字 myObj 来找到其 ID。

表 11-1　名称服务程序

程　序	功　能
smNameAdd( )	向名称数据库中增加一个名字
smNameRemove( )	从名称数据库中删除一个名字
smNameFind( )	通过名字查找一个共享字符

[①] 不要把此优先级类型和与 VME 总线入口相关的 CPU 优先级搞混淆了。

续表

程　序	功　能
smNameFindByValue( )	通过数值查找一个共享字符
smNameShow( )	在标准输出设备上输出名称数据库的内容[②]

可以用同样的方法来广播一个共享内存的地址。比如，CPU 0 上的任务 t1 分配了一块存储区，并且以名字 mySharedMem 向数据库中增加了一个地址。通过用 mySharedMem 在名称数据库中查找地址 CPU 1 上的任务 t2 可以找到共享内存的地址。

不同处理器上的任务可以用一个统一的名字得到一个新创建的对象的值，表 11-1 是名称服务程序的列表。注意对每一任务来说，从名称数据库中重新得到 ID 的操作只需要一次，而这通常发生在应用程序的初始化过程中。

名称数据库的服务程序自动转换成或从网络字节顺序转换；并不调用 htonl( )或 ntohl( ) 从名称数据库中得到其数值。

表 11-2 中的对象类型在 smNameLib.h 中被定义。

表 11-2　共享内存对象类型

常　量	16 进制值
T_SM_SEM_B	0
T_SM_SEM_C	1
T_SM_MSG_Q	2
T_SM_PART_ID	3
T_SM_BLOCK	4

下例列出了由调用 smNameShow( )显示的名称数据库，如果 INCLUDE_SM_OBJ 被选择在 VxWorks 的工程工具中，它就被自动包括进去。参数 smNameShow( )指定了列出的信息的层次；在这种情况下，1 就说明所有的信息都被显示出来。关于 smNameShow( )的附加信息，请看其参考条目。

```
-> smNameShow 1
value = 0 = 0x0
```

输出被送至标准输出设备，看起来如下所示：

```
Name in Database Max : 100 Current : 5 Free : 95
Name                    Value           Type
----------------- ------------ -------------
myMemory              0x3835a0       SM_BLOCK
```

[②] 在 INCLUDE_SM_OBJ 被选择的情况下自动包括进去。

```
myMemPart          0x3659f9        SM_PART_ID
myBuff             0x383564        SM_BLOCK
mySmSemaphore      0x36431d        SM_SEM_B
myMsgQ             0x365899        SM_MSG_Q
```

11.2.2　共享信号量

像局部信号量一样，共享信号量通过更新信号量状态信息而提供同步信息。关于信号量的详细讨论，请看本书中的第 2 章“基本操作系统”以及“semLib”的相关内容。执行任一可以访问共享内存的 CPU 上的任务都可以发出和接收共享信号量，它们可以被作为运行在不同 CPU 上的任务的同步信号或作为共享资源的互斥信号。

为了使用一个共享信号量，任务要首先创建它并广播它的 ID 号。这可以通过把它加进名称数据库中来完成。系统中的任一 CPU 上的任务可以通过首先获得 ID（比如，从名称数据库中）而使用此信号量，得到了 ID 后，它就可以获得或释放信号量。

在为互斥现象而使用信号量的情况下，典型的情况是存在一个由不同 CPU 上任务间共享的系统资源，并且用信号量来阻止并发访问。当任务要求独占资源时，它就获得信号量；任务完成后，它就释放该信号量。

比如，有两个任务：CPU 0 上的 t1 和 CPU 1 上的 t2。任务 t1 通过将其加进数据库并分配名字 myMutexSem 来创建信号量并广播此信号量的 ID；任务 t2 通过在数据库中查找名字 myMutexSem 来获得信号量的 ID。不管何时，如果任务要访问资源，它首先用该信号量的 ID 来获得信号量；当任务完成时，它就释放该信号量。

在为同步情况而使用共享信号量的情况下，假定某个 CPU 上的任务一定要通知另一个 CPU 上的任务某些事件已经发生，被同步的任务挂起在信号量处以等待事件的发生。事件发生时，使之同步的任务就释放该信号量。

比如，有两个任务：CPU 0 上的 t1 和 CPU 1 上的 t2，t1 和 t2 都管理遥控武器。由 t1 控制的遥控武器正传送一个物理对象给由 t2 控制的遥控武器。任务 t2 把武器移动到指定地点，但是一定要等到 t1 指示它已经准备好为 t2 接收此对象了。任务 t1 通过将其加入到数据库中并指定名字 objReadySem 来创建一个共享信号量并广播该信号量的 ID。任务 t2 通过在数据库中查找名字 objReadySem 来得到信号量的 ID，然后它通过此信号量的 ID 来获取信号量。如果信号量不可得，t2 挂起，等待 t1 指示对象已经为 t2 做好准备。当 t1 准备好给 t2 传送控制对象时，它就释放信号量，并在 CPU 1 上准备好 t2。

表 11-3　共享信号量创建程序

创 建 程 序	描　　述
semBSmCreate( )	创建一个共享二进制信号量
semCSmCreate( )	创建一个计数型信号量

有二进制和计数型两种类型的共享信号量。共享信号量有它们自己的创建程序，并且返回一个 SEM_ID，表 11-3 列出了创建程序。所有其他信号量程序，除了 semDelete( )以外，在创建的共享信号量上都是透明操作的。

共享信号量和局部信号量的用法有以下不同：

- 信号量创建时共享信号量的指定队列顺序一定要是 FIFO 的。图 11-1 说明了在不同 CPU 上执行的两个任务都想得到相同的信号量。任务 1 首先执行并且由于信号量不可得（空）而被放在队列的前端。任务 2（执行在不同的 CPU 上）在任务 1 的尝试后也想得到该信号量，这样就被放在队列任务 1 的后面。

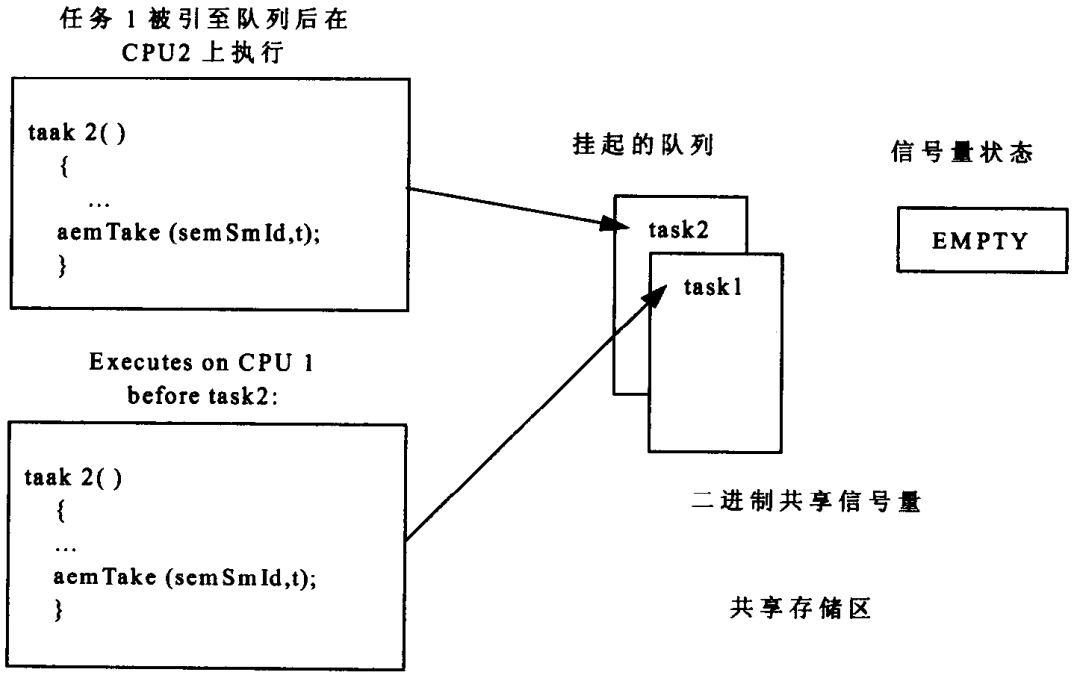


图 11-1 共享信号量队列

- 共享信号量不能在中断时被释放。
- 共享信号量不能被删除。尝试删除一个共享信号量时会返回一个 ERROR，并把 S_smObjLib_NO_OBJECT_DESTROYs 设置为 errno。

可以用 semInfo( )得到挂起在共享信号量上的任务的共享任务控制块。如果 INCLUDE_SEM_SHOW 包含在 VxWorks 的工程工具中，可以用 semShow( )来显示共享信号量和挂起任务列表的状态。以下例子显示了共享信号量 mySmSemaphoreId 的详细信息，正如由第二个参数（0 = summary，1 = details）所示：

```
-> semShow mySmSemaphoreId,1
value = 0 = 0x0
```

输出被传送至标准输出设备，如以下内容所示：

```
Semaphore Id     : 0x36431d
Semaphore Type   : SHARED BINARY
Task Queuing     : FIFO
Pended Tasks     : 2
State            : EMPTY
TID              CPU Number      Shared TCB
-------------- ------------- --------------
0xd0618             1           0x364204
0x3be924            0           0x36421c
```

例 11-1：共享信号量

以下的代码例子描述了在不同 CPU 上执行并且使用共享信号量的的两个任务。程序 semTask1( )创建了共享信号量，将其初始化为满的状态。它把信号量加到名称数据库中（使别的 CPU 上的任务能够访问它），取得信号量，做一些处理再释放信号量。程序 semTask2( ) 从数据库中得到该信号量的 ID，获得信号量，做某些处理后也释放信号量。

```
/* semExample.h - 共享信号量头文件举例*/

#define SEM_NAME "mySmSemaphore"

/* semTask1.c - 共享信号量举例*/
 /* 此代码由 CPU1 上的任务执行 */
#include "VxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "stdio.h"
#include "taskLib.h"
#include "semExample.h"
 /*
 * semTask1 - 共享信号量用户
 */
 STATUS semTask1 (void)
    {
    SEM_ID semSmId;
     /* 创建共享信号量*/
     if ((semSmId = semBSmCreate (SEM_Q_FIFO,SEM_FULL)) == NULL)
        return (ERROR);
     /* 向命名数据库中增加对象 */
     if (smNameAdd (SEM_NAME,semSmId,T_SM_SEM_B) == ERROR).
        return (ERROR);
```

```
    /* 获得共享信号量并保持一段时间 */
    semTake (semSmId,WAIT_FOREVER);
     printf ("Task1 has the shared semaphore\n");
   taskDelay (sysClkRateGet () * 5);
   printf ("Task1 is releasing the shared semaphore\n");
    /* 释放共享信号量 */
    semGive (semSmId);
    return (OK);
    }
/* semTask2.c - 共享信号量举例 */
 /* 此代码由 CPU2 上的任务执行 */
 #include "VxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "stdio.h"
#include "semExample.h"
/*
 * semTask2 - 共享信号量用户
 */
 STATUS semTask2 (void)
    {
    SEM_ID semSmId;
    int    objType;
    /* 在命名数据库中查找对象 */
    if (smNameFind (SEM_NAME,(void **) &semSmId,&objType,WAIT_FOREVER)
       == ERROR)
       return (ERROR);
    /* 获得共享信号量 */
    printf ("semTask2 is now going to take the shared semaphore\n");
semTake (semSmId,WAIT_FOREVER);
    printf ("Task2 got the shared semaphore!!\n");
    /* 释放共享信号量 */
    semGive (semSmId);
    printf ("Task2 has released the shared semaphore\n");
    return (OK);
   }
```

11.2.3 共享消息队列

共享消息队列是一个 FIFO 队列，它可以用任务来发送和接收任何一个可以访问共享

内存的 CPU 上的变长信息。它们可以用来使任务同步或者在运行于不同 CPU 的任务间交换数据。关于消息队列的详细讨论，请看本书中的“第 2 章基本操作系统”中的关于 msgQlib 的参考内容。

为了使用消息队列，任务要创建一个消息队列并广播其 ID 号。想利用此消息队列发送或接收消息的任务首先要得到消息队列的 ID 号，然后用此 ID 号访问消息队列。

比如，假定一个典型的服务器/客户的例子。CPU 上的服务器任务 t1 从一个消息队列中读出请求并用另一个的消息队列应答这些请求。任务 t1 通过将其加入到名称数据库，并指定名字 requestQue 来创建一个队列并广播其 ID 号。如果 CPU 0 上的任务 t2 要给 t1 发送一个请求，它首先通过在名称数据库中查找名字 requestQue 来得到消息队列的 ID。在发送第一个请求之前，任务 t2 已经创建了一个应答消息队列。它通过将其作为发送的请求信息的一部分来广播其 ID，而不是把它的 ID 加到数据库中。当 t1 从用户端接收到请求时，在应答此用户前，它首先从消息中找到队列的 ID 号以使用它，然后任务 t1 通过使用此 ID 给用户发送应答信息。

为了在不同 CPU 的任务间传送消息，首先要通过调用 msgQSmCreate( )来创建一个消息队列，此程序返回一个 MSG_Q_ID。此 ID 用来在共享消息队列上发送和接收消息。

象它们的局部的对应部分一样，共享消息队列可以发送优先级紧急或一般的消息。

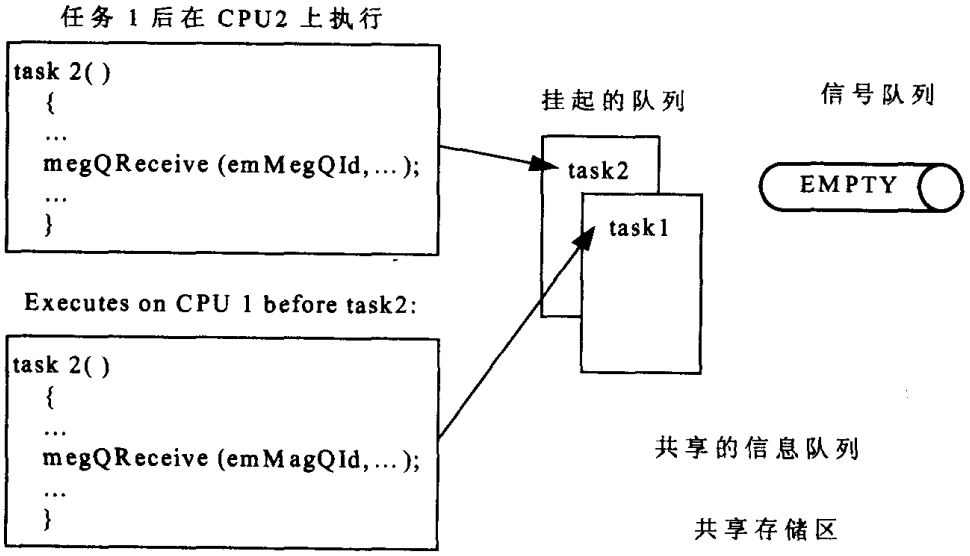


图 11-2　共享消息队列

使用共享消息队列和局部消息队列有以下不同：

- 消息队列创建时，共享消息队列的任务队列的顺序一定要是 FIFO 的。图 11-2 显示了执行在不同 CPU 上的两个任务都想从同一个共享消息队列中接收消息。任务 1 首先执行，因为消息队列中没有消息所以被放在队列的前端。任务 2（执行在另一 CPU 上）在任务 1 的尝试之后也想从消息队列中接收一个消息，就被放在队列中任务 1 的后面。

- 消息不能在中断时从共享消息队列上发送（即使在模式 NO_WAIT 下也一样）。
- 共享消息队列不能被删除。删除一个共享消息队列时返回 ERROR，并把 errno 设置到 S_smObjLib_NO_OBJECT_DESTROY 中。

为了达到共享消息队列的最佳性能，把发送和接收缓冲区用 4 字节的分界排列起来。为了显示共享消息队列以及挂起在队列上的任务列表的状态，选择将 CLUDE_MSG_Q_SHOW 包括在 VxWorks 的工程工具中并且调用 msgQShow( )。以下例子显示了共享消息队列 x7f8c21 上的详细信息，如第二个参数（0=概要说明，1=详细说明）所示：

```
-> msgQShow 0x7f8c21,1
value = 0 = 0x0
```

输出被送至标准输出设备，如以下所示：

```
Message Queue Id   : 0x7f8c21
Task Queuing       : FIFO
Message Byte Len   : 128
Messages Max       : 10
Messages Queued    : 0
Receivers Blocked : 1
Send timeouts      : 0
Receive timeouts   : 0
Receivers blocked :
TID            CPU Number            Shared TCB
---------- -------------------- --------------
0xd0618             1                0x1364204
```

例 11-2：共享消息队列

在以下的代码例子中，执行在不同 CPU 上的两个任务用共享消息队列互相传送数据。服务器任务创建了请求消息队列，把它加进名称数据库中，并从消息队列中读取消息。用户任务从名称数据库中得到 smRequestQId，创建一个应答消息队列，把应答队列的 ID 号增加作为消息的一部分，并把消息发送至服务器。服务器得到应答队列的 ID，并用它给用户返回消息。因为大量的队列 ID 在数据部分通过网络传送，所以此技术要求使用网络字节命令转换宏 htonl( )和 ntohl( )。

```
/* msgExample.h - 共享消息队列头文件举例 */

#define MAX_MSG      (10)
#define MAX_MSG_LEN (100)
#define REQUEST_Q    "requestQue"
```

```
typedef struct message
    {
    MSG_Q_ID replyQId;
    char     clientRequest[MAX_MSG_LEN];
    } REQUEST_MSG;
/* server.c - 共享消息队列服务举例 */

/* 此文件包含此消息队列服务任务的代码 */

#include "VxWorks.h"
#include "msgQLib.h"
#include "msgQSmLib.h"
#include "stdio.h"
#include "smNameLib.h"
#include "msgExample.h"
#include "netinet/in.h"

#define REPLY_TEXT "Server received your request"

/*
 * serverTask - 接收并处理来自共享消息队列的请求
 */

STATUS serverTask (void)
    {
    MSG_Q_ID    smRequestQId;   /* 请求共享消息队列*/
    REQUEST_MSG request;        /* 请求文本 */

    /* 创建共享消息队列以处理请求 */

    if ((smRequestQId = msgQSmCreate (MAX_MSG,sizeof (REQUEST_MSG),
        MSG_Q_FIFO)) == NULL)
        return (ERROR);

    /* 向命名数据库中增加新创建的请求消息队列 */

    if (smNameAdd (REQUEST_Q,smRequestQId,T_SM_MSG_Q) == ERROR)
        return (ERROR);

    /* 从请求队列中读信息 */
```

```
    FOREVER
        {
        if (msgQReceive (smRequestQId, (char *) &request, sizeof (REQUEST_MSG)
            WAIT_FOREVER) == ERROR)
            return (ERROR);

        /* 处理请求 - 在这种情况下仅打印它 */

        printf ("Server received the following message:\n%s\n",
            request.clientRequest);

        /* 用在用户的请求消息中指定的 ID 号发送应答 */

        if (msgQSend ((MSG_Q_ID) ntohl ((int) request.replyQId),
            REPLY_TEXT, sizeof (REPLY_TEXT),
            WAIT_FOREVER, MSG_PRI_NORMAL) == ERROR)
            return (ERROR);
        }
    }
/* client.c - 共享消息队列用户举例 */

/* 此文件包含了消息队列用户任务的代码 */

#include "VxWorks.h"
#include "msgQLib.h"
#include "msgQSmLib.h"
#include "smNameLib.h"
#include "stdio.h"
#include "msgExample.h"
#include "netinet/in.h"

/*
 * clientTask - 给服务器发送请求并接收应答
 */

STATUS clientTask
    (
    char * pRequestToServer  /* 给服务器的发送请求 */
                             /* 限制到 100 个字符 */
    )
    {
```

```
MSG_Q_ID     smRequestQId; /* 请求消息队列 */
MSG_Q_ID  smReplyQId;      /* 应答消息队列 */
REQUEST_MSG request;       /* 请求文本*/
int          objType;      /* smNameFind 的虚拟变量*/
char         serverReply[MAX_MSG_LEN]; /*服务器应答的缓冲区*/
/* 用它的名字得到请求消息队列的 ID 号 */

if (smNameFind (REQUEST_Q,(void **) &smRequestQId,&objType,
    WAIT_FOREVER) == ERROR)
    return (ERROR);

/* 创建应答队列,建立请求并发送给服务器 */

if ((smReplyQId = msgQSmCreate (MAX_MSG,MAX_MSG_LEN,
    MSG_Q_FIFO)) == NULL)
    return (ERROR);

request.replyQId = (MSG_Q_ID) htonl ((int) smReplyQId);

strcpy (request.clientRequest,pRequestToServer);

if (msgQSend (smRequestQId,(char *) &request,sizeof (REQUEST_MSG),
    WAIT_FOREVER,MSG_PRI_NORMAL) == ERROR)
    return (ERROR);
/* 读应答并打印它 */

if (msgQReceive (request.replyQId,serverReply,MAX_MSG_LEN,
    WAIT_FOREVER) == ERROR)
    return (ERROR);

printf ("Client received the following message:\n%s\n",serverReply);

return (OK);
}
```

11.2.4　共享内存分配器

共享内存分配器允许不同 CPU 上的任务分配和释放存储区变量块，这些变量块可以由可访问共享内存系统的所有 CPU 访问。它提供了两组程序：操作用户创建部分的共享内存的底层程序和操作有助于共享内存系统部分的共享存储分区的高层程序（这种组织方法与

由局部存储区管理器使用的 memPartLib 很相似)。

共享内存块可以从不同的分区中分配，共享内存的系统分区和用户创建的分区两者都可使用。可以创建一个用户分区并用来分配指定大小的数据块。从用来表示指定的块大小的用户创建的分区中分配固定大小的块，这样就避免了存储区碎片问题。

1. 共享内存系统分区

为了使用共享内存分区，任务要分配一个共享内存块并广播其地址。广播 ID 的一种方式是把地址加到名称数据库中。用来在共享内存系统分区中分配块的程序，返回一个局部地址。在此地址被广播到其他 CPU 上的任务之前，此局部地址必须被转换为全局地址，而且如果要使用此共享内存上的任务，那么首先要得到存储块的地址，并把其全局地址转换为局部地址。任务拥有了此地址以后就可以使用此存储区了。

然而，为了解决互斥问题，典型情况是用共享信号量来保护共享内存中的数据。这样在一个更为普遍的假设情况中，创建共享内存（把它加进数据库）的任务也创建了一个共享信号量。共享信号量的 ID 通过把它存储在位于共享内存块中的共享数据结构的部分中来广播它。任务第一次访问共享数据结构时，它首先在数据库中查找存储区的地址，并从共享数据结构中的相应部分得到信号量的 ID。不管什么时间，任务要想访问共享数据时，它首先一定要获得信号量；同时当任务完成对共享数据的操作时，它就释放该信号量。

比如，假定执行在两个不同 CPU 上的任务一定要共享某数据。执行在 CPU 1 上的任务 t1 从共享内存系统分区中分配了一个存储块并把其局部地址转换为全局地址。然后用名字 mySharedData 把共享数据的全局地址加进名称数据库中。任务 t1 也创建了一个共享信号量，并且把其 ID 号存在位于共享内存的数据结构第一部分。执行在 CPU 2 上的任务 t2 通过在名称数据库中查找名字 mySharedData 来得到共享内存的地址，然后把此地址转换为局部地址。在共享内存中访问此数据以前，t2 先从位于共享存储块的数据结构的第一部分中获得共享信号量的 ID。它在使用数据之前先取得信号量，并且当完成了使用数据的操作后就释放该信号量。

2. 用户创建的分区

为了利用用户创建的共享内存，任务首先创建一个共享存储器分区，并把它加进名称数据库中。在任务能使用共享内存之前，它首先一定要从名称数据库中查找分区的 ID。当任务拥有了分区的 ID 后，它就能访问位于共享内存中的存储部分了。

比如，任务 t1 用名字 myMemPartition 创建了一个共享存储器分区，并把它加进名称数据库中。执行在另外一个 CPU 上的任务 t2 要从新的分区中分配存储空间。任务 t2 首先在名称数据库中查找 myMemPartition 以得到其分区的 ID 号，然后就可以用此 ID 号在它内部分配存储空间了。

3. 使用共享内存系统分区

共享内存的系统分区与局部存储区的系统分区很相似。表 11-4 列出了操作共享内存系

统分区的函数。

表 11-4　共享内存系统分区函数

函　数	功　能
smMemMalloc( )	分配一个共享系统存储区块
smMemMalloc( )	分配一组共享系统存储区块
smMemRealloc( )	调整一个共享系统存储区块的大小
smMemFree( )	释放一个共享系统存储区块
smMemShow( )	在标准输出设备上显示共享内存系统分区的用法统计。如果 INCLUDE_SM_OBJ 被选择包括在 VxWorks 的工程工具中，此函数就被自动包括进去
smMemOptionsSet( )	为共享内存系统设置调试选项
smMemAddToPool( )	给共享内存系统增加存储区
smMemFindMax( )	在共享内存系统分区中找到最大的空闲块的大小

给分配的存储区返回一个指针的函数返回一个局部地址（也就是适合在局部 CPU 上使用的地址）。为了通过处理器共享此存储区，此地址一定要在广播至其他 CPU 上的任务之前转换成一个全局地址。另一个 CPU 上的任务使用此存储区之前，必须把这个全局地址转换为局部地址。系统中提供了宏和程序模块以在局部地址和全局地址之间转换；参见头文件 smObjLib.h 以及 smObjLib.h 中的参考内容。

例 11-3：共享内存系统分区

以下代码的例子使用共享内存系统分区的存储区，以在不同 CPU 上的任务间共享数据。数据结构的第一个成员是用来处理互斥问题的信号量。发送任务创建并初始化此结构，然后接收任务，再访问此数据并显示它。

```
/* buffProtocol.h - 简单缓冲区交换协议头文件*/

#define BUFFER_SIZE    200            /* 共享数据缓冲区大小 */
#define BUFF_NAME      "myMemory"     /* 数据库中数据缓冲的名字 */

typedef struct shared_buff
    {
    SEM_ID semSmId;
    char   buff [BUFFER_SIZE];
    } SHARED_BUFF;
/* buffSend.c - 简单缓冲区交换协议发送端 */

/* 此文件用来写共享内存 */

#include "VxWorks.h"
```

```
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "stdio.h"
#include "buffProtocol.h"

/*
 * buffSend - 写共享信号量保护缓冲区
 */

STATUS buffSend (void)
    {
    SHARED_BUFF * pSharedBuff;
    SEM_ID        mySemSmId;

    /* 获得共享系统存储器 */

    pSharedBuff = (SHARED_BUFF *) smMemMalloc (sizeof (SHARED_BUFF));

    /*
     * 在增加到数据库中之前初始化共享缓冲器结构。保护信号量最初不可得,并作为接收
块  */。

    if ((mySemSmId = semBSmCreate (SEM_Q_FIFO,SEM_EMPTY)) == NULL)
        return (ERROR);
    pSharedBuff->semSmId = (SEM_ID) htonl ((int) mySemSmId);

    /*
     * 把共享缓冲器的地址转换为全局地址并增加到数据库中 */

    if (smNameAdd (BUFF_NAME,(void *) smObjLocalToGlobal (pSharedBuff),
                   T_SM_BLOCK) == ERROR)
        return (ERROR);

    /* 向共享缓冲器中放置数据 */

    sprintf (pSharedBuff->buff,"Hello from sender\n");

    /* 允许接收器通过释放信号量读数据 */

    if (semGive (mySemSmId) != OK)
```

```
        return (ERROR);

    return (OK);
    }
/* buffReceive.c - 简单缓冲器交换协议接收端 */

/* 此文件读共享缓冲器 */

#include "VxWorks.h"
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "stdio.h"
#include "buffProtocol.h"

/*
 * buffReceive - 接收共享信号量保护的缓冲器*/

STATUS buffReceive (void)
    {
    SHARED_BUFF * pSharedBuff;
    SEM_ID        mySemSmId;
    int           objType;

    /* 从命名数据库中取得共享缓冲器地址 */

    if (smNameFind (BUFF_NAME,(void **) &pSharedBuff,
                    &objType,WAIT_FOREVER) == ERROR)
        return (ERROR);

    /* 把缓冲器的全局地址转换为其本地值 */

    pSharedBuff = (SHARED_BUFF *) smObjGlobalToLocal (pSharedBuff);

    /* 把共享信号量的 ID 号转换为主机(本地端)字节顺序 */

    mySemSmId = (SEM_ID) ntohl ((int) pSharedBuff->semSmId);

    /* 在读数据缓冲区以前取得共享信号量 */

    if (semTake (mySemSmId,WAIT_FOREVER) != OK)
```

```
        return (ERROR);

    /* 读数据缓冲区并打印它*/

    printf ("Receiver reading from shared memory: %s\n",pSharedBuff->buff);

    /* 返回数据缓冲区信号量 */

    if (semGive (mySemSmId) != OK)
        return (ERROR);

    return (OK);
    }
```

4. 使用用户创建的分区

共享内存分区有一个独立的创建函数 memPartSmCreate( )，它返回一个 MEM_PART_ID。在用户定义的共享内存分区被创建以后，memPartLib 中的函数在它上面透明运行。注意传递给 memPartSmCreate( )（或 memPartAddToPool( )）的共享内存的地址必须是全局地址。

例 11-4：用户创建的分区

这个例子很像例 11-3，在例 11-3 中使用了共享内存系统分区。这个例子创建了一个用户定义的分区，并把共享的数据存储在此新的分区中。共享信号量用来保护此数据。

```
/* memPartExample.h - 共享内存分区头文件举例 */

#define CHUNK_SIZE      (2400)
#define MEM_PART_NAME   "myMemPart"
#define PART_BUFF_NAME  "myBuff"
#define BUFFER_SIZE     (40)

typedef struct shared_buff
    {
    SEM_ID semSmId;
    char   buff [BUFFER_SIZE];
    } SHARED_BUFF;
/* memPartSend.c - 共享内存分区举例发送端 */

/* 此文件写用户定义的共享内存分区 */

#include "VxWorks.h"
#include "memLib.h"
```

```
#include "semLib.h"
#include "semSmLib.h"
#include "smNameLib.h"
#include "smObjLib.h"
#include "smMemLib.h"
#include "stdio.h"
#include "memPartExample.h"

/*
 * memPartSend - 发送共享内存分区缓冲部分
 */

STATUS memPartSend (void)
    {
    char *          pMem;
    PART_ID         smMemPartId;
    SEM_ID          mySemSmId;
    SHARED_BUFF *   pSharedBuff;

    /* 为分区分配要用到的共享系统存储器 */

    pMem = smMemMalloc (CHUNK_SIZE);

    /* 用以前分配的内存块创建用户定义的分区.
     * 注意: memPartSmCreate 使用内存池的全局地址作为第一个参数 */

    if ((smMemPartId = memPartSmCreate (smObjLocalToGlobal (pMem),CHUNK_SIZE))
          == NULL)
        return (ERROR);

    /* 从分区中分配存储区 */

    pSharedBuff = (SHARED_BUFF *) memPartAlloc ( smMemPartId,
                sizeof (SHARED_BUFF));
    if (pSharedBuff == 0)
        return (ERROR);
    /* 在增加到数据库中之前初始化结构 */

    if ((mySemSmId = semBSmCreate (SEM_Q_FIFO,SEM_EMPTY)) == NULL)
        return (ERROR);
```

```
    pSharedBuff->semSmId = (SEM_ID) htonl ((int) mySemSmId);

    /* 进入命名数据库中的共享分区 ID*/

    if (smNameAdd (MEM_PART_NAME,(void *) smMemPartId,T_SM_PART_ID) ==
ERROR)
        return (ERROR);

    /* 将共享缓冲器地址转换为全局地址并加到数据库中 */

    if (smNameAdd (PART_BUFF_NAME,(void*) smObjLocalToGlobal(pSharedBuff),
                T_SM_BLOCK) == ERROR)
        return (ERROR);

    /* 用共享缓冲区发送数据 */

    sprintf (pSharedBuff->buff,"Hello from sender\n");

    if (semGive (mySemSmId) != OK)
        return (ERROR);

    return (OK);
    }
/* memPartReceive.c - 共享内存分区接收端的例子 */

/* 此文件从用户定义的共享内存分区中读取数据 */

#include "VxWorks.h"
#include "memLib.h"
#include "stdio.h"
#include "semLib.h"
#include "semSmLib.h"
#include "stdio.h"
#include "memPartExample.h"

/*
 * memPartReceive - 接收执行在 CPU1 上的共享内存分区缓冲部分-使用共享信号量以保护共
享内存*/
STATUS memPartReceive (void)
    {
    SHARED_BUFF * pBuff;
```

```
SEM_ID          mySemSmId;
int             objType;

/* 从命名数据库中取得共享缓冲区地址 */

if (smNameFind (PART_BUFF_NAME,(void **) &pBuff,&objType,
                WAIT_FOREVER) == ERROR)
    return (ERROR);

/* 将缓冲区的全局地址转换为它的本地值 */

pBuff = (SHARED_BUFF *) smObjGlobalToLocal (pBuff);

/* 在使用共享内存之前取得共享信号量*/

mySemSmId = (SEM_ID) ntohl ((int) pBuff->semSmId);
semTake (mySemSmId,WAIT_FOREVER);
printf ("Receiver reading from shared memory: %s\n",pBuff->buff);
semGive (mySemSmId);

return (OK);
}
```

5. 共享内存分区选项部分的边缘效应

就像其本地对应部分一样，共享内存分区（系统以及用户创建的）对于错误的处理可以有不同的选项设置；参见关于 memPartOptionsSet( )和 smMemOptionsSet( )的参考条目。

如果选项 MEM_BLOCK_CHECK 在以下情况中使用，则系统可以进入内存分区中不可使用的状态。如果一个任务想释放一个出现问题的块并且总线上发生了错误，此任务就被挂起。由于共享信号量在内部是为互斥问题而使用的，所以挂起的任务仍占有此信号量，并且没有别的任务可以访问此内存分区。通过默认值，共享内存分区可以在不设置选项 MEM_BLOCK_CHECK 的情况下被创建。

11.3　内部需注意的事项

11.3.1　系统要求

系统中所有的 CPU 都应该能够操作由共享内存对象使用的共享内存。在主 CPU

（CPU0）上，双端口存储器或单独的存储器都可以被使用。共享内存对象一定要与共享内存位于同一块地址空间。注意，对所有的 CPU 来说，存储区并不是要都占用同样的局部地址。

> ⚠ **注意**：利用 VxMP 板一定要支持硬件测试与设置（不可分割的读-改-写操作环）。PowerPC 是个例外，参见当前的 PowerPC 结构补充。

系统中所有的 CPU 一定要通过 VME 总线支持不可分割的读-改-写操作环。可以由旋转上锁机制使用不可分割的 RMW，以得到内部共享数据结构的惟一入口地址；参见 11.3.2 “旋转上锁机制”小节的详细介绍。因为所有的板一定要支持硬件测试和设置，常量 SM_TAS_TYPE 一定要设置到工程工具 VxWorks 参数表中的 SM_TAS_HARD 中。

作用于其上的任一事件一定要通知对象 CPU。对于初始化 CPU 的事件来说首选的方法是中断与之相关的 CPU。中断的使用独立于硬件的处理能力。如果不能使用中断，那么可以使用查询方案，尽管这通常导致重大的错误的执行结果。

可以使用共享内存对象的 CPU 的最大数目是 20（CPU 标号从 0 到 19）。实际的最大数目通常是一个依赖于 CPU、总线带宽以及应用程序的较小的数目。

11.3.2 旋转上锁机制

内部共享内存对象的数据结构通常要被保护起来，以防止旋转上锁机制的并发访问。旋转上锁机制是一个试图获取资源的外部入口的环路（这种情况下指的是内部数据结构）。通常用一个不可分割的硬件读-改-写操作环（硬件测试和设置）来解决互斥的问题。如果第一次取得上锁的尝试失败了，接下来还会有多次重试，每一次都与下一次之间有个逐步减小的随机延迟。在 20MHz 的 MC68030 上第一次尝试获取上锁和第一次重试之间花费的平均时间是 70 微秒。操作旋转上锁循环在时间上之所以存在很大差异是由于它受处理器的 cache、访问共享内存的时间以及总线流量的影响。如果在经历了由 SM_OBJ_MAX_TRIES（在 VxWorks 的共享内存对象的特性窗口的参数表中定义）指定的最大的尝试数目之后上锁仍没有取得，就把 errno 设置给 S_smObjLib_LOCK_TIMEOUT。如果发生了此类错误，就得把尝试的最大数目设置成一个更大的值。注意任何取得旋转上锁的失败都会阻止共享内存对象的某些特有功能。大多数情况下，这是由共享内存配置的问题而引起的；参见“11.5.2 发现并解决故障的技巧”。

11.3.3 中断延迟

旋转上锁期间，中断要禁止以减少任务在占有上锁期间被抢占的可能性。结果系统中

每一个处理器的中断延迟都会增加。然而，对于一个特定的 CPU 来说，由共享内存对象增加的中断延迟是一个常数。

11.3.4　约束

与局部信号量和消息队列不同，共享内存对象不能在中断级被使用。不使用共享内存对象的函数可以从 ISR 中调用。ISR 是用来处理与外部事件相关的关键时间的过程；因此在中断时使用共享内存对象不合适。在一个多处理器系统中，应该在与时间相关的中断发生的 CPU 上运行与事件相关的关键时间处理过程。

注意，由于共享内存对象是从共享内存中分配的，所以不能被删除。

当使用共享内存对象的时候，各个对象类型的最大数目一定要在属性窗口的参数表中指定好。参见“11.4.3　初始化共享内存对象包”。如果程序创建了多于指定数目的对象，它就有可能溢出。如果发生此情况，共享对象创建函数就返回一个错误，并且 error 被设置给 S_memLib_NOT_ENOUGH_MEM。为了解决这个问题，应该首先增加相应类型的共享内存对象的最大数目，参见表 11-5 中的可使用的配置常量的列表。因为共享内存池使用了共享内存的剩余部分，所以它减少了共享内存系统区的大小。如果这不是所希望的，那么可以增加相应的共享内存对象的数目以及整个共享内存 SM_OBJ_MEM_SIZE 的大小。参见 11.4 “配置”中关于用来配置的常量部分的讨论。

11.3.5　高速缓存一致性

在没有 MMU 或总线监听机制的情况下，当双端口存储器在板上被使用时，一定要为主 CPU 上的共享内存的操作而禁止运行高速缓存区。如果用户看到以下的错误信息，请确认常量 INCLUDE_CACHE_ENABLE 是否没有被选择包括进 VxWorks 中：

```
usrSmObjInit - cache coherent buffer not available. Giving up.
```

11.4　配　　置

为了在 VxWorks 中包含共享内存对象，要选择把 INCLUDE_SM_OBJ 包括进 VxWorks 的工程工具中。大多数配置已经在 usrConfig.c 中的 usrSmObjInit( )当中自动完成。然而，用户也需要调整属性窗口的参数表中的某些值以反映用户的配置情况，这些内容在此节中都有描述。

11.4.1 共享内存对象和共享内存网络驱动

共享内存对象和共享内存网络[③]使用相同的存储区、地址锁定和中断机制。用共享内存对象配置系统与配置共享内存网络驱动很相似。关于配置和使用共享内存网络的详细说明，请参见“*VxWorks Network Programmer's Guide: Data Link Layer Network Components.*”如果共享内存定位点地址的默认值被修改，那么修改后一定不要超过 256 字节。

配置共享内存对象的一个最重要的方面是计算共享内存定位点后定位的地址。共享内存位置定位是指一个系统中所有的 CPU 都可以访问的位置，并且可以由 VxMP 或共享内存网络驱动所共用。此定位给共享内存头位置、共享内存包头（由共享内存网络驱动使用）、以及共享内存对象头各存储了一个指针。

在属性窗口的参数表中，此定位点的地址是用常量 SM_ANCHOR_ADRS 来定义的。如果处理器用共享内存网络的驱动程序启动，那么定位点地址就与启动设备（sm=*anchorAddress*）的值一样。共享内存对象的初始化代码使用启动行的值，而不是使用常量。如果没有使用共享内存网络驱动程序，那么就得修改 SM_ANCHOR_ADRS 的定义以恰当地反映用户的系统情况。

M_INT_TYPE 支持及定义两种类型的中断：邮箱中断和总线中断（请参见 *VxWorks Network Programmer's Guide: Data LinkLayer Network Components*）。邮箱中断（SM_INT_MAILBOX）是首选中断，总线中断（SM_INT_BUS）是次选中断。如果不能用中断，那么可以使用查询方案（SM_INT_NONE），但这样做效率很低。

当一个 CPU 初始化它的共享内存对象的时候，它要定义中断类型以及三个中断参数，它们说明了如何通知 CPU 事件的发生。通过调用 smCpuInfoGet( )这些值可以为任一附加的 CPU 获得。

对象的默认中断方法由参数表中的 SM_INT_TYPE，SM_INT_ARG1，SM_INT_ARG2 和 SM_INT_ARG3 来定义。

11.4.2 共享内存区

共享内存对象位于对所有的处理器都可见的共享内存区。可以用这个区域来存储内部共享内存对象的数据结构和共享内存系统分区。

共享内存区总是位于主 CPU 上的双端口 RAM 上，但是它也可以被置于单独的存储条上。当把系统配置为共享内存定位点地址 SM_INT_ARG3 的偏移量时，这样共享内存区的地址就被定义了，如图 11-3 所示。

[③] 已知为插板网络。

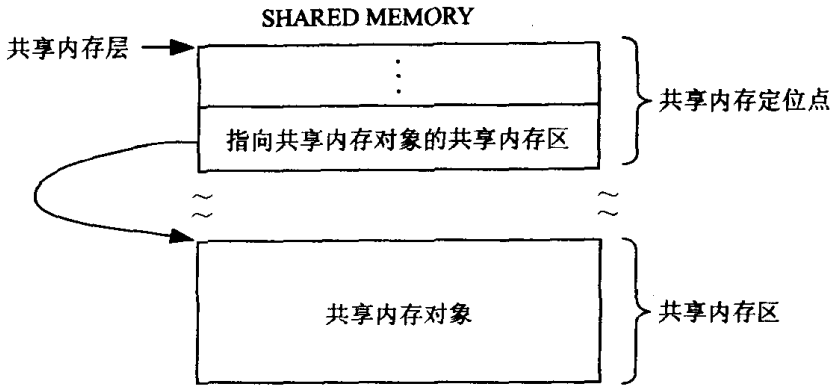


图 11-3　共享内存布局图

11.4.3　初始化共享内存对象包

共享内存对象由 *installDir*/target/src/config/usrSmObj.c 中的函数 usrSmObjInit( )的默认值来初始化。主 CPU 采取的配置步骤与从 CPU 采取的步骤稍微有所不同。

一定要定义共享内存区的地址。如果存储区在板子外部，那么它的值一定要计算出来（参见图 11-5）。

图 11-4 中的例子的配置使用了主 CPU 的双端口 RAM 的共享内存区。

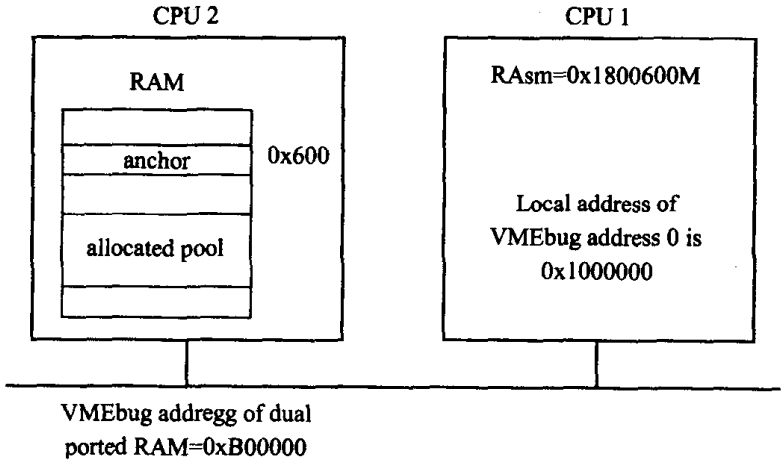


图 11-4　配置例子：双端口存储器

主 CPU 的属性窗口的参数表中 SM_OFF_BOARD 定义为 FALSE，SM_ANCHOR_ADRS 为 0x600。因为要使用板上的存储器，所以 SM_OBJ_MEM_ADRS 被设置为 NONE，SM_OBJ_MEM_SIZE 被设为 0x20000。对于从 CPU 来说，把 VME 总线的基地址映射为 0x1000000，SM_OFF_BOARD 被设置为 TURE，定位点地址为 0x1800600，这通过取得 VME 总线地址（0x800000）并把它加到定位点地址（0x600）上计算出来。很多板要求深层的地

址转换，这取决于板映射到 VME 存储区的位置。在这个例子中，因为板子把 VME 总线的基地址映射为 0x1000000，所以从 CPU 的定位点地址就是 0x1800600。

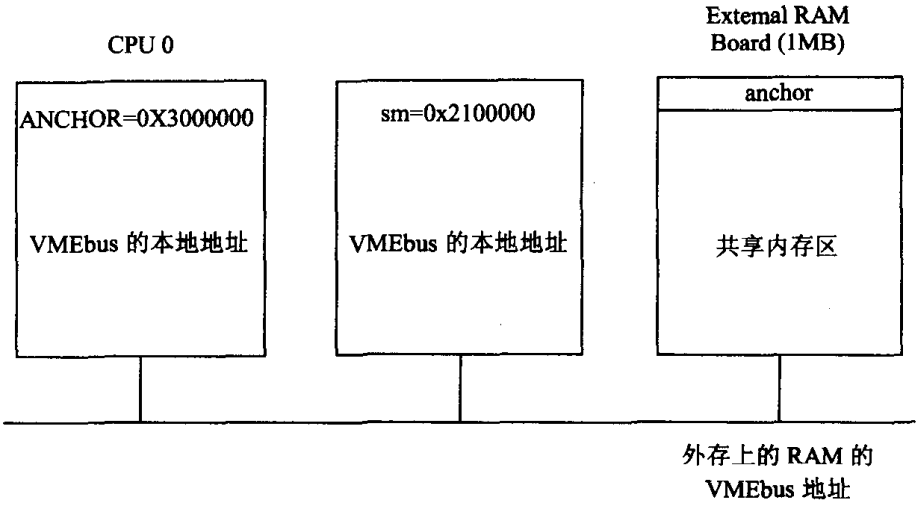


图 11-5　配置例子：外存板

在图 11-5 例子的配置中，共享内存位于一个单独的存储条上。主机的参数表上，SM_OFF_BOARD 为 TRUE，SM_ANCHOR_ADRS 为 0x3000000，SM_OBJ_MEM_ADRS 被设置给 SM_ANCHOR_ADRS，SM_OBJ_MEM_SIZE 被设置为 0x100000。对于从板来说，M_OFF_BOARD 是 TRUE，定位点地址是 0x2100000。这通过取得存储条的 VME 总线地址（0x2000000），并把它加到局部 VME 的总线地址（0x100000）上计算出来。

因为共享内存池可由底板上所有的处理器访问，所以一些附加的配置有时需要使共享内存变为非缓存区。带有 MMU 的板通过默认值将 MMU 激活。在 MMU 激活的情况下，不在板上的存储器一定要作为非缓存的，这通过使用 sysLib.c 中的数据结构 sysPhysMemDesc 来完成。这个数据结构一定要包含作为共享内存池的 VME 地址空间的虚拟地址到物理地址的地址映射，并把此存储区标注为非缓存区（大多数 BSP 通过默认值包括此映射）。关于附加信息参见“12.3　虚拟内存配置”一节。

⚠ *注意：对于通常的 MC68K 来说，如果 MMU 被关闭，那么一定要全局地关闭数据缓冲部分，参见关于 cacheLib 的参考条目。*

初始化共享内存对象时，一定要将存储区大小以及每一对象类型的最大数目指定下来。主处理器使用常量 SM_OBJ_MEM_SIZE 指定存储区的大小。符号常量被用来设置不同对象的最大数目，这些常量都在属性窗口的参数表中被指定。关于这些常量的列表参见表 11-5。

表 11-5　共享内存对象的配置常量

符号常量	默认值	描　述
SM_OBJ_MAX_TASK	40	使用共享内存对象的任务的最大数目
SM_OBJ_MAX_SEM	30	共享信号量（计数型和二进制）的最大数目
SM_OBJ_MAX_NAME	100	名称数据库中的名字的最大数目
SM_OBJ_MAX_MSG_Q	10	共享消息队列的最大数目
SM_OBJ_MAX_MEM_PART	4	用户在共享存储区中创建的分区的最大数目

如果创建的目标对象的大小超过了共享内存区，那么在初始化过程中 CPU 0 上就会显示一个错误消息。当共享内存为共享目标对象配置好了以后，共享内存的剩余部分就被用来作为共享内存的系统分区了。

函数 smObjShow( )显示了正被使用的共享内存对象的当前数目以及其他的统计信息，如下所示：

```
-> smObjShow
value = 0 = 0x0
```

VxWorks 的工程工具中如果选择 INCLUDE_SM_OBJ，那么函数 smObjShow( )就被自动包括进去。smObjShow( )的输出被送至标准输出设备，如以下所示：

```
Shared Mem Anchor Local Addr : 0x600
Shared Mem Hdr Local Addr    : 0x363ed0
Attached CPU                 : 2
Max Tries to Take Lock       : 0
Shared Object Type      Current      Maximum      Available
------------------      -------      -------      ---------
Tasks                         1           40             39
Binary Semaphores             3           30             27
Counting Semaphores           0           30             27
Messages Queues               1           10              9
Memory Partitions             1            4              3
Names in Database             5          100             95
```

⚠ **注意**：如果重启主 CPU，则要重启所有的从 CPU。如果要重启一个从 CPU，那么它一定不能存在挂起在共享内存的对象上的任务。

11.4.4 配置举例

下例显示了包括三个 CPU 的多处理器系统的配置。主 CPU 是 CPU 0，应该从它的双端口存储区中配置共享内存部分。这个应用程序包含 20 个使用共享内存对象的任务，并且使用了 12 个消息队列和 20 个信号量。名称数据库的最大值是默认值（100），它仅需要一个用户定义的存储区分区。CPU 0 上，共享内存池被配置为在板级上，此存储区从处理器的系统存储区中分配。在 CPU 1 和 CPU 2 上，共享内存池被配置为不在板级上。表 11-6 显示了在特性窗口的参数表上为工程工具中 INCLUDE_SM_OBJECTS 设置的值。

表 11-6　三个 CPU 系统的配置设置池

CPU	符 号 常 量	值
主（cpu0）	SM_OBJ_MAX_TASK	20
	SM_OBJ_MAX_SEM	20
	SM_OBJ_MAX_NAME	100
	SM_OBJ_MAX_MSG_Q	12
	SM_OBJ_MAX_MEM_PART	1
	SM_OFF_BOARD	FALSE
	SM_MEM_ADRS	NONE
	SM_MEM_SIZE	0x10000
	SM_OBJ_MEM_ADRS	NONE
	SM_OBJ_MEM_SIZE	0x10000
从（cpu1，cpu2）	SM_OBJ_MAX_TASK	20
	SM_OBJ_MAX_SEM	20
	SM_OBJ_MAX_NAME	100
	SM_OBJ_MAX_MSG_Q	12
	SM_OBJ_MAX_MEM_PART	1
	SM_OFF_BOARD	FALSE
	SM_ANCHOR_ADRS	（char *） 0xfb800000
	SM_MEM_ADRS	SM_ANCHOR_ADRS
	SM_MEM_SIZE	0x10000
	SM_OBJ_MEM_ADRS	NONE
	SM_OBJ_MEM_SIZE	0x10000

注意对于从 CPU 来说，SM_OBJ_MEM_SIZE 的值实际上并没有被使用。

11.4.5　初始化步骤

初始化通过库 *installDir*/target/src/config/usrSmObj.c 中的 usrSmObjInit( )的默认值来完成。在主 CPU 上，共享内存对象的初始化过程包括以下步骤：

1．用函数 smObjSetup( )在共享存储定位点部分中建立共享内存对象头和它的指针。

2．用函数 smObjInit( )为此 CPU 初始化共享内存的对象参数。

3．用函数 smObjAttach( )将此 CPU 附给共享内存对象工具。

对于从 CPU 来说仅需要 2、3 两步。

函数 smObjAttach( )检查共享内存对象的设置。它需要查找共享内存 *Heartbeat* 以确定此工具已经运行。共享内存 *Heartbeat* 是一个无符号整数，由主 CPU 每秒钟增加一次。它向从 CPU 说明了共享内存对象已经初始化，可以用来调试了。*Heartbeat* 是共享内存对象头中的第一个内容；参见“11.5　发现故障及解决措施”。

11.5　发现故障及解决措施

共享内存对象的问题可以由很多原因造成。这部分讨论了多数情况下通常的问题以及一些解决问题的方法。通常情况下，用户可以通过再检查硬件及软件的配置来查找问题。

11.5.1　配置问题

按以下所示的步骤可以确信用户的系统已被正确配置：

- 确信对于每一个使用 VxMP 的处理器而言，已经将常量 INCLUDE_SM_OBJ 选择包括进了 VxWorks 的工程工具中。
- 确信指定的定位点地址是可以由 CPU 得到的地址，这可以由常量 SM_ANCHOR_ADRS 在特性窗口的属性表中或启动时（如果对象机用共享内存的网络启动的话）定义。
- 如果存在与共享内存对象相关的很大的总线通信量，就有可能发生总线错误。可以通过改变总线仲裁模式或总线上相关的 CPU 的优先级来避免这个问题。
- 如果 memAddToPool( )，memPartSmCreate( )或 smMemAddToPool( )操作失败，那么就应检查一下用户给这些程序传递的地址是不是实际的全局地址。

11.5.2 发现并解决故障的技巧

用以下技巧来解决用户遇到的任一问题：

- 对指示已经达到尝试取得旋转上锁的极限数目的错误信息的函数 smObjTimeoutLogEnable( )允许或禁止打印。
- 函数 smObjShow( )把共享内存对象工具的状态显示在标准输出设备上。它显示了一个任务要在特定的 CPU 上取得旋转上锁的尝试的最大数目。一个大的值说明程序遇到了竞争共享内存资源的问题。
- 可以检查共享内存的 *heartbeat* 以证明主 CPU 已经将共享内存对象初始化了。共享内存 *heartbeat* 位于共享内存对象头的前 4 个字节的字中，到其头的偏移量位于共享内存定位点位置的第六个 4 字节的字中。（参见 "*VxWorks Network Programmer's Guide: Data Link Layer Network Components.*"）

这样，如果共享内存定位点地址位于 0x800000:

```
VxWorks Boot]: d 0x800000
800000: 8765 4321 0000 0001 0000 0000 0000 002c *.eC!............*
800010: 0000 0000 0000 0170 0000 0000 0000 0000 *...p............*
800020: 0000 0000 0000 0000 0000 0000 0000 0000 *................*
```

到共享内存对象头的偏移量就是 0x170。为了察看共享内存对象头，可以显示 0x800170:

```
[VxWorks Boot]: d 0x800170
800170: 0000 0050 0000 0000 0000 0bfc 0000 0350 *...P...........P*
```

在前述的例子中，共享内存的 heartbeat 值是 0x50。再次显示此位置以确信 heartbeat 处于激活状态；如果它的值被改变了，说明共享内存对象已经被初始化了。

- 当设置为 1（TRUE）时，全局变量 smIfVerbose 就会将共享内存接口的错误信息连同附加的共享内存操作的细节信息打印在控制台上。这个变量使用户能够从在调试层不可得的设备的驱动程序层中得到运行时间的相关信息。设置给 smIfVerbose 的默认设置是 0（FALSE），它可以由程序或在 shell 中重新设定。

第 12 章　虚拟内存接口

12.1　简　　介

VxWorks 提供了两层的虚拟内存支持。基本层在 VxWorks 中，并给每一页面提供了高速缓存；完全层没有做在 VxWorks 中，而且要求选择可选组件 VxVMI。VxVMI 提供了文本段和 VxWorks 外部矢量表的写保护，还提供了 CPU 内存管理单元（MMU）的与结构无关的接口。关于如何安装 VxVMI 的详细情况，请参见“*Tornado Getting Started Guide*”。

本章包括以下部分：

- 关于支持基本层的详细说明。
- 适用于两个支持层的配置向导。
- 仅适用于可选组件 VxVMI 的两部分内容。

一个是总体应用，讨论了由 VxVMI 实现的写保护。

另一个描述了用来操作 MMU 的一组程序。VxVMI 以与结构无关的方式提供了与 MMU 接口的底层程序，从而允许用户实现自己的虚拟内存系统。

12.2　基本虚拟内存支持

对于拥有 MMU 的系统来说，VxWorks 通过把相关缓冲区更改为非高速缓存而允许用户更有效地完成 DMA 操作和内部处理器间的通信。当别的处理器或 DMA 设备访问同样的内存位置时，有必要确认一下数据是不是没有在原位置缓冲。在不能把某部分存储区更改为非高速缓存区的情况下，一定要全局地禁止高速缓冲（会导致性能降低），或者人工禁止缓冲，或使之变为无效。

在 VxWorks 工程工具中通过选择 INCLUDE_MMU_BASIC 而将基本虚拟内存支持包括进去，请参见“12.3　虚拟内存配置”。同时也可以使用 cacheDmaMalloc( )分配非高速缓冲区，请参见 cacheLib 的相关参考条目。

12.3 虚拟内存配置

以下配置的讨论运用了打包和非打包的虚拟内存支持两种方式。在工程工具中，定义了如表 12-1 中的常量来反应用户系统的情况。

表 12-1 MMU 配置常量

常 量	描 述
INCLUDE_MMU_BASIC	不带 VxVMI 选项的基本 MMU 支持
INCLUDE_MMU_FULL	带有 VxVMI 选项的完全 MMU 支持
INCLUDE_PROTECT_TEXT	文本段保护（要求完全 MMU 支持）
INCLUDE_PROTECT_VEC_TABLE	附加的矢量表保护（要求完全 MMU 支持）

相应与用户的处理器的默认页面大小（4K 或 8K 字节）在用户的 BSP 中是由 VM_PAGE_SIZE 定义的。如果用户因为某些原因一定要改变这些值，那么可以在 config.h 中重新定义 VM_PAGE_SIZE。（请参见 *"Tornado User's Guide: Configuration and Build"*）

为了使存储区非缓存化，一定要有一个虚拟地址到物理地址的地址映射表。vmLib.h 中的数据结构 PHYS_MEM_DESC 中定义了用来反映物理内存情况的参数。在 sysLib.c 中每个板的存储区使用表都用 sysPhysMemDesc(声明作为 PHYS_MEM_DESC 的数组)定义。除了定义存储区页面的初始状态以外，结构 sysPhysMemDesc 还定义了用来映射虚拟到物理内存位置的虚拟地址。关于页面状态的进一步讨论，请参见"页面状态"。

可以通过修改 sysPhysMemDesc 的结构来反应用户的系统配置情况。比如，用户可能需要增加一些处理器间的通信缓冲区的地址，而它们可能不包括在此结构中；或者，用户需要将共享内存数据结构的 VME 总线地址做一映像，并作为非高速缓存部分。大多数板级支持包拥有一个在 sysPhysMemDesc 中定义的 VME 部分；然而，这可能不包括能满足用户的系统配置要求的全部空间。

在此结构中没有被包括的 I/O 设备和存储区也要被映像，并作为非缓存部分使用。总之，板以外的存储区被指定为非高速缓存；请参见"VxWorks Network Programmer's Guide: Data Link Layer Network Components."

⚠ **注意**：sysPhysMemDesc 中定义的存储区一定要按照页面排列并要求占满全部页面。换言之，PHYS_MEM_DESC 结构的前三个内容（虚拟地址，物理地址和长度）一定都要是 VM_PAGE_SIZE 的偶数倍。若指定一些没有按照页面排列的 sysPhysMemDesc 的内容会导致 VxWorks 初始化的冲突。

以下配置的例子是由使用共享存储区网络层的多个 CPU 组成，共享内存要使用一个单独的内存条。由于此存储区没有做映射，所以为了使网络中所有的板卡都能使用，它要增加到 sysPhysMemDesc 上。此存储区从地址 0x4000000 开始，并且一定要作为非缓冲区，如下面摘录的代码所示：

```
/* 共享内存 */
{
(void *) 0x4000000,        /* 虚拟地址*/
(void *) 0x4000000,        /* 物理地址*/
0x20000,              /* 长度*/
/* 初始状态 */
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |VM_STATE_MASK_CACHEABLE,
/* 初始状态 */
VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}
```

对于 MC680x0 的板子来说，虚拟地址一定要跟物理地址相同；对于其他板，虚拟地址和物理地址可以按情况而定。

12.4 普 通 应 用

这部分为写保护文本段的配置和附加矢量表的使用描述了 VxVMI 的通常应用和配置情况。

VxVMI 使用 MMU 以防止存储区部分被覆盖，这是由存储区的写保护页完成的。不是所有的对象硬件都支持写保护；关于详细信息请参见此手册中的结构附录部分。对于大多数结构而言，页面大小是 8K。试图往已经写保护的存储区位置写东西会导致发生总线错误。

当已经安装了 VxWorks 时，所有的文本段都被写保护起来，请参见“12.3　虚拟内存配置”。用 ld( )安装的附加目标模块的文本段自动被标识为只读。安装目标模块时，将需要被写保护的存储区按照页面大小的增量形式分配。对于应用代码的写保护来说不需要额外的步骤。在系统初始化过程中，VxWorks 写保护了附加矢量表。改变中断矢量的惟一方法是调用函数 intConnect( )，在调用过程中它允许对附加矢量表的写操作。

为了把写保护包括进去，需要在 VxWorks 工程工具中选择以下内容：

INCLUDE_MMU_FULL

INCLUDE_PROTECT_TEXT

INCLUDE_PROTECT_VEC_TABLE

12.5　使用程序化的 MMU

这部分提供了一些用 vmLib 中的底层程序来操作 MMU 的工具，并对它作了说明。用户可以将某些数据作为任务或代码段的私有数据，使部分存储区非缓存化，或者写保护部分存储区。其中用来实现虚拟内存的基本结构是虚拟内存上下文（VMC）。

关于 VxVMI 程序的概要情况，请参见 vmLib 的参考条目。

12.5.1　虚拟内存上下文

虚拟内存上下文（VM_CONTEXT，在 vmLib 中定义）由用来把虚拟地址映射到物理地址的翻译表和其他信息组成。可以创建多个虚拟内存上下文并可以在它们之间随意切换。

1. 全局虚拟存储区

一些系统对象（比如文本段和信号量）一定要能被系统中所有的任务访问，而不管其中的哪一个是当前的虚拟存储区上下文。这些对象可以通过全局虚拟存储区的方式更改为可访问的。全局虚拟存储区通过把系统中所有的物理内存（此映射在 sysPhysMemDesc 中有定义）映射到虚拟存储区空间的同样的地址而创建起来。在默认的系统配置中，它最初赋予物理存储区和虚拟存储区一一对应的关系：比如，虚拟地址 0x5000 映射到物理地址 0x5000。在某些结构中，也可以使用 sysPhysMemDesc 来设置虚拟存储区，这样虚拟地址到物理地址的映射就不是一对一的了，关于附加信息请参见“12.3　虚拟内存配置”。

全局虚拟存储区可以从所有的虚拟存储区上下文中访问，而且对一个虚拟存储区上下文中全局映射所做的修改会出现在所有的虚拟存储区上下文中。虚拟存储区上下文创建以前，可以用 vmGlobalMap( )加上所有的全局存储区；虚拟存储区上下文创建以后，被增加的全局存储区可能不能被现存的上下文所利用。

2. 初始化

全局虚拟存储区是通过从 usrRoot( )当中调用的 usrMmuInit( )中的 vmGlobalMapInit( )来初始化的。函数 usrMmuInit( )位于 *installDir*/target/src/config/ usrMmuInit.c 中，并使用 sysPhysMemDesc 来创建全局虚拟存储区；然后再创建一个默认的虚拟存储区上下文，并把它作为当前值。另外它还可以选择启动 MMU。

3. 页面状态

每一个虚拟存储区页面（典型大小是 8KB）都有一个与它相联系的状态。页面可以是有效或无效的、可写或非可写的、高速缓冲或非高速缓冲的。请参见表 12-2 中相关的常量

内容。

表 12-2　状态标志

常　量	描　述
VM_STATE_VALID	有效翻译
VM_STATE_VALID_NOT	无效翻译
VM_STATE_WRITABLE	可写存储区
VM_STATE_WRITABLE_NOT	只读存储区
VM_STATE_CACHEABLE	高速缓存区
VM_STATE_CACHEABLE_NOT	非高速缓存区

有效性

有效状态指明了虚拟地址到物理地址的翻译是正确的。初始化翻译表时，全局虚拟存储区被标为有效状态，同时将所有别的虚拟存储区初始化为无效。

可写性

通过把状态设置为非可写，可以把页面定为只读，并且通过 VxWorks 来写保护所有的文本段。

高速缓冲性

通过设置状态标志位为非缓冲状态可以禁止缓存区页面的缓存性，这对于由多个处理（包括 DMA 设备）共享的存储器来说是很有用的。

可以用函数 vmStateSet( )来改变页面状态。除了指定的状态标志位，状态掩码一定要说明哪个标志位正在被改变，请参见表 12-3。另外的一些与结构无关的状态在 vmLib.h 中都有说明。

表 12-3　状态掩码

常　量	描　述
VM_STATE_MASK_VALID	改变有效标志
VM_STATE_MASK_WRITABLE	改变写标志
VM_STATE_MASK_CACHEABLE	改变高速缓存标志

12.5.2　私有虚拟内存

可以通过建立一个新的虚拟内存上下文来创建私有虚拟内存，通过使其他任务不能进入或限制具体程序的访问，这种方法对于保护数据来说是很有用的。虚拟内存上下文没有为任务的执行自动创建，但是可以被用户创建并且用具体的应用程序的方式进行内外切换。

初始化过程中，系统要创建一个默认的上下文，所有的任务都应该使用这个默认的上下文。为了创建一块私有虚拟存储区，任务应该用 vmContextCreate( )创建一个新的虚拟内

存上下文，并把它作为当前值。所有的虚拟内存上下文共享系统初始化时创建的全局映射，请参见图 12-1。仅仅当前虚拟内存上下文（包括全局虚拟内存）中的有效虚拟内存是可以访问的。不能访问在其他虚拟内存上下文中定义的虚拟内存。为了使另一个内存上下文变成当前值，可以调用函数 vmCurrentSet( )。

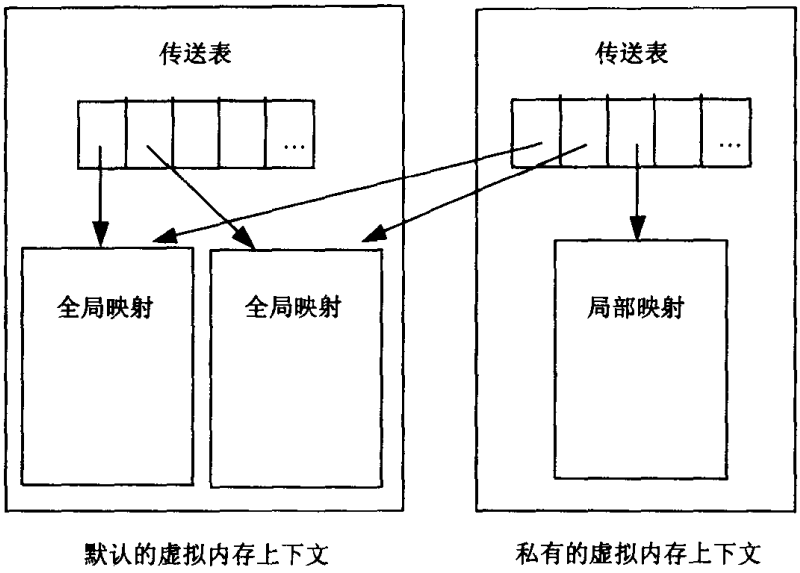


图 12-1　虚拟内存的全局映射表

可以使用函数 vmMap( )来创建一个新的虚拟地址到物理地址的映射表；物理和虚拟地址一定要提前指定。物理内存（一定要按页面排列）可以使用函数 valloc( )获得。获得虚拟地址的最简单的方法是通过调用函数 vmGlobalInfoGet( )来查找一个不是全局映射表的虚拟页面。按照这个方案，如果需要多个映射表，任务一定要跟踪它自己的私有虚拟内存页面，以保证它没有两次将其映射为相同的非全局地址。

当物理页面被映射到虚拟空间的新的位置时，物理页面可以从两个不同的虚拟地址访问（一种情况为 *aliasing*）：新映射的虚拟地址和在全局虚拟内存中等价于物理地址的虚拟地址。这可能给某些结构带来一些问题，因为高速缓存区可以给相同的内存地址保持两个不同的值。为了避免这个问题，可以把全局虚拟内存中的虚拟页面（使用 vmStateSet( )）变为无效，这也保证了仅仅当包含新的映射的虚拟内存上下文是当前值时，此数据才是可以访问的。

图 12-2 描述了两个私有虚拟内存上下文。新的上下文（pvmc2）把虚拟地址 0x6000000 映射为物理地址 0x10000。为了避免从此虚拟上下文外部（pvmc1）访问此地址，相应的物理地址一定要被设置为无效状态。如果要用地址 0x10000 来访问此存储空间，因为那个地址当前为无效状态，这样就会发生总线错误。

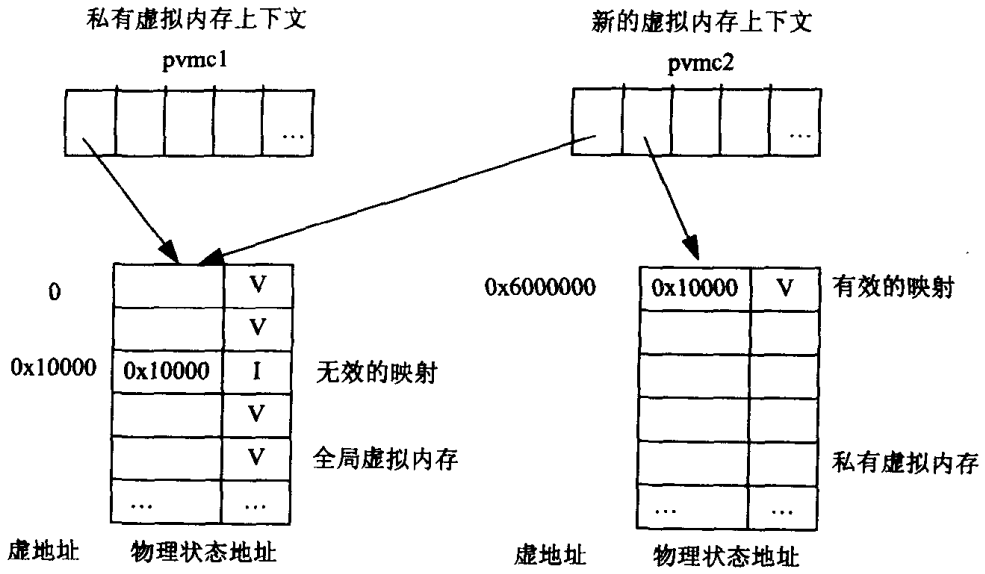


图 12-2　映射私有虚拟内存

例 12-1：私有虚拟内存上下文

在以下的代码例子中，是用私有虚拟内存上下文从任务的私有存储区分区当中分配存储区空间。在文本切换的过程中安装函数 contextSetup( )创建了一个作为当前值的私有虚拟内存上下文，虚拟内存上下文被存储在任务的 TCB 中的 spare1 区域中。这里使用切换钩子函数来保存原来的上下文并建立任务的私有上下文。请注意使用切换钩子函数会增加上下文的切换时间。用户定义的内存分区要用私有虚拟内存上下文来创建，分区的 ID 号被存在任务 TCB 的 spare2 中。任一需要私有虚拟内存上下文的任务都必须调用函数 contextSetup( )。以下为包括测试代码的示例的任务。

```
/* contextExample.h - 由钩子函数使用的 vm 上下文头文件 */

#define NUM_PAGES (3)
/* context.c - 使用文本切换钩子函数把任务私有上下文作为当前值 */

#include "vxWorks.h"
#include "vmLib.h"
#include "semLib.h"
#include "taskLib.h"
#include "taskHookLib.h"
#include "memLib.h"
#include "contextExample.h"

void privContextSwitch (WIND_TCB *pOldTask,WIND_TCB *pNewTask);
```

```
/*
 * initContextSetup - 安装上下文切换的钩子函数
 */

STATUS initContextSetup ( )
    {
    /* 安装切换钩子函数 */

    if (taskSwitchHookAdd ((FUNCPTR) privContextSwitch) == ERROR)
        return (ERROR);
    return (OK);
    }

/*
 * contextSetup - 初始化上下文并创建独立的内存分区 *
 * 对需要私有上下文的每一个任务仅调用一次 *
 * 如果系统上的每个任务都需要一个私有上下文,这需要增加创建钩子函数程序。为了作为创建钩
子函数而使用它,安装新的虚拟内存上下文的代码应该通过在任务的 TCB 中简单地保存新的上下文
来替换。 */
STATUS contextSetup (void)
    {
    VM_CONTEXT_ID pNewContext;
    int pageSize;
    int pageBlkSize;
    char * pPhysAddr;
    char * pVirtAddr;
    UINT8 * globalPgBlkArray;
    int newMemSize;
    int index;
    WIND_TCB * pTcb;

    /* 创建上下文 */

    pNewContext = vmContextCreate( );

    /* 得到页面及页面的块大小*/

    pageSize = vmPageSizeGet ( );
    pageBlkSize = vmPageBlockSizeGet ( );
    newMemSize = pageSize * NUM_PAGES;
```

```
/* 分配按页面排列的物理内存*/

if ((pPhysAddr = (char *) valloc (newMemSize)) == NULL)
    return (ERROR);
```

/* 选择虚拟地址以映射它。在这个例子中,既然每个任务中仅有一个页面块被使用,就可以仅使用不是全局映射的首地址。VmGlobalInfoGet()返回一个每一元素都与一虚拟内存块对应的布尔阵列。 */

```
globalPgBlkArray = vmGlobalInfoGet( );
for (index = 0; globalPgBlkArray[index] == TRUE; index++)
    ;
pVirtAddr = (char *) (index * pageBlkSize);

/* 把物理存储区映射到新的上下文 */

if (vmMap (pNewContext,pVirtAddr,pPhysAddr,newMemSize) == ERROR)
    {
    free (pPhysAddr);
    return (ERROR);
    }

/*
 * 把全局虚拟存储区的状态设为无效—任何对此存储区的访问都必须通过新的上下文进行*/
if (vmStateSet(pNewContext,pPhysAddr,newMemSize,VM_STATE_MASK_VALID,
                VM_STATE_VALID_NOT) == ERROR)
    return (ERROR);

/* get tasks TCB */

pTcb = taskTcb (taskIdSelf( ));

/* 改变虚拟内存上下文 */

/*
 * 中断空闲的 TCB 中的当前 vm 上下文—当此任务要交换时，切换钩子函数将安装它。*/

pTcb->spare1 = (int) vmCurrentGet( );

/* 安装新的任务上下文*/

vmCurrentSet (pNewContext);
```

```
    /* 创建新的内存分区并将其 ID 存入任务的 TCB 中。 */

    if ((pTcb->spare2 = (int) memPartCreate (pVirtAddr,newMemSize)) == NULL)
        return (ERROR);

    return (OK);
    }
/*
 * privContextSwitch - routine to be executed on a context switch
 *
 * If old task had private context,save it. If new task has private
 * context,install it.
 */

void privContextSwitch
    (
    WIND_TCB *pOldTcb,
    WIND_TCB *pNewTcb
    )

    {
    VM_CONTEXT_ID pContext = NULL;

    /* 如果以前的任务拥有私有上下文,就保存它一并重启以前的上下文。 */
    if (pOldTcb->spare1)
        {
        pContext = (VM_CONTEXT_ID) pOldTcb->spare1;
        pOldTcb->spare1 = (int) vmCurrentGet ( );

        /* 重存以前的上下文*/

        vmCurrentSet (pContext);
        }

    /*
     * 如果下一任务拥有私有上下文,映射新的上下文并把以前的上下文保存在任务的 TCB 中。
*/

    if (pNewTcb->spare1)
        {
        pContext = (VM_CONTEXT_ID) pNewTcb->spare1;
```

```
    pNewTcb->spare1 = (int) vmCurrentGet( );

        /* 安装新的任务上下文*/

        vmCurrentSet (pContext);
        }
    }
/* taskExample.h - 用切换钩子函数来测试 VM 上下文的头文件 */

/* 此代码由任务例子来使用 */

#define MAX (10000000)

typedef struct myStuff
    {
    int stuff;
    int myStuff;
    } MY_DATA;
/* testTask.c - 测试切换钩子函数的任务代码 */

#include "vxWorks.h"
#include "memLib.h"
#include "taskLib.h"
#include "stdio.h"
#include "vmLib.h"
#include "taskExample.h"
IMPORT char *string = "test\n";

MY_DATA *pMem;

/*
 * testTask - 分配私有存储区并使用它 *
 * 永远循环,改变存储区并打印到全局环当中,使用它从 shell 的测试中来联结。既然 pMem 指向
私有存储区,当试图读取它的时候 shell 应该产生一个总线错误。
 * 比如:
 *      -> sp testTask
 *      -> d pMem
 */

STATUS testTask (void)
    {
    int val;
```

```
    WIND_TCB *myTcb;

    /* 安装私有上下文 */
     if (contextSetup ( ) == ERROR)
        return (ERROR);
     /* 获得 TCB */
     myTcb = taskTcb (taskIdSelf ( ));

    /* 分配私有存储区*/

    if ((pMem = (MY_DATA *) memPartAlloc((PART_ID) myTcb->spare2,
         sizeof (MY_DATA))) == NULL)
        return (ERROR);

    /*
     * 修改私有存储区中的数据并在全局存储区中显示它。*/

    FOREVER
    {
        for (val = 0; val <= MAX; val++)
            {
            /* 修改结构*/

            pMem->stuff = val;
            pMem->myStuff = val / 2;
            /* 确认可以访问全局虚拟存储区 */

            printf (string);

            taskDelay (sysClkRateGet( ) * 10);
            }
        }
    return (OK);
    }

/*
 * testVmContextGet - 用 vmContextShow( )[①]返回任务的存储在 TCB 中的虚拟内存上下文
以显示任务的虚拟内存上下文 *
 * 比如,from the shell,type:
 *    -> tid = sp (testTask)
```

[①] 此函数没有被包括在 Tornado 的 shell 中。为了在 Tornado shell 中使用它，用户一定要在 VxWorks 的配置中定义 INCLUDE_MMU_FULL_SHOW；请参见《Tornado User's Guide》Projects，调用时此程序的输出被送至标准输出设备上。

```
 *    -> vmContextShow (testVmContextGet (tid))
 */

VM_CONTEXT_ID testVmContextGet
    (
    UINT tid
    )
    {
    return ((VM_CONTEXT_ID) ((taskTcb (tid))->spare1));
    }
```

12.5.3　非高速缓存存储区

不支持总线监听的结构一定要禁止应用于处理器间通信（或由 DMA 设备）的高速缓存区。如果多个处理器从存储区位置读或写数据，用户一定要保证当 CPU 访问数据时，它使用的是当前的值。如果缓存被系统中的一个或多个 CPU 所使用，也可以在其中一个 CPU 的数据缓存区中存有一个此数据的局部的备份。

在图 12-3 的例子中，拥有多个 CPU 的系统共享数据，其中系统上的一个 CPU（CPU 0）将共享数据作了缓存。CPU 0 上的任务读数据[1]然后改变值[2]；然而，新的数值可能仍在高速缓存区中，并且当另一个 CPU（CPU 1）上的任务访问它的时候不会被冲掉。这样由 CPU 1 上的任务使用的数据的值是以前的数值，它并不反应由 CPU 0 上的任务所做的改动，那个值仍存在于 CPU 0 的数据缓存区[2]当中。

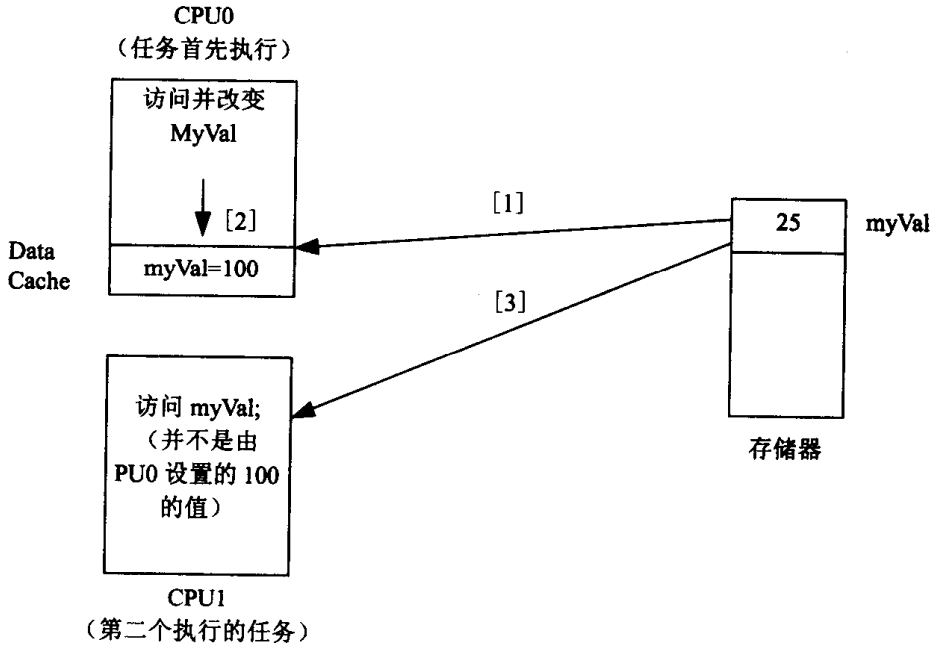


图 12-3　数据缓存可能出现的问题的例子

可以用函数 vmStateSet( )来禁止页面的高速缓冲，比如：vmStateSet (pContext, pSData, len, VM_STATE_MASK_CACHEABLE, VM_STATE_CACHEABLE_NOT)。为了分配非缓存存储区，请参见关于 cacheDmaMalloc( )的相关条目。

12.5.4 非可写存储器

存储器可以被标为非可写性的，部分存储器可以用 vmStateSet( )来写保护以防止意外操作。它的一个应用是限制对于某些特定程序所做的针对数据对象的修改。如果数据对象是全局但只读的，任务就可以读此对象但是不修改它。任务如果一定要修改此对象，那么一定要调用相关程序。在程序内部，数据在此程序的整个过程中是可写的；退出时，存储器被设置为 VM_STATE_WRITABLE_NOT。

例 12-2：非可写存储器

在此代码的例子中，为了改变由 pData 指向的数据结构，任务一定要调用函数 dataModify( )。此程序先将存储区作为可写区，然后修改数据，并把存储区设置回非可写状态。任务可以成功读取数据；但是，如果它要试图改变 dataModify( )外部的数据，就会发生总线错误。

```
/* privateCode.h - 使数据仅从程序中可写的头文件 */

#define MAX 1024

typedef struct myData
    {
    char stuff[MAX];
    int moreStuff;
    } MY_DATA;
/* privateCode.c - 用 VM 的上下文使数据对代码段私有。*/
#include "vxWorks.h"
#include "vmLib.h"
#include "semLib.h"
#include "privateCode.h"

MY_DATA * pData;
SEM_ID dataSemId;
int pageSize;

/*
 * initData - 分配存储区并使它处于非可写状态 *
 * 此函数初始化数据并且仅应该被调用一次。 *
```

```
 */

STATUS initData (void)
    {
    pageSize = vmPageSizeGet( );

    /* 创建信号量以保护数据 */

    dataSemId = semBCreate (SEM_Q_PRIORITY,SEM_EMPTY);
    /* 给页面分配存储区 */

    pData = (MY_DATA *) valloc (pageSize);

    /* 初始化数据并使它处于仅可读状态 */

    bzero (pData,pageSize);
    if (vmStateSet (NULL,pData,pageSize,VM_STATE_MASK_WRITABLE,
            VM_STATE_WRITABLE_NOT) == ERROR)
        {
        semGive (dataSemId);
        return (ERROR);
        }

    /* 释放信号量 */

    semGive (dataSemId);
    return (OK);
    }

/*
 * dataModify - modify data
 *
 * 为了修改数据,任务应该调用函数,给数据传送一个指针。 *
可以从 shell 的应用中测试:
 *      -> initData
 *      -> sp dataModify
 *      -> d pData
 *      -> bfill (pdata,1024,'X')
 */

STATUS dataModify
    (
```

```
MY_DATA * pNewData
)
{

/* 从数据的外部入口取得信号量*/

semTake (dataSemId,WAIT_FOREVER);

/* 使存储区可写 */

if (vmStateSet (NULL,pData,pageSize,VM_STATE_MASK_WRITABLE,
        VM_STATE_WRITABLE) == ERROR)
            {
            semGive (dataSemId);
            return (ERROR);
            }

/* 更新数据*/

bcopy (pNewData,pData,sizeof(MY_DATA));

/* 使存储区不可写 */

if (vmStateSet (NULL,pData,pageSize,VM_STATE_MASK_WRITABLE,
        VM_STATE_WRITABLE_NOT) == ERROR)
            {
            semGive (dataSemId);
            return (ERROR);
            }

semGive (dataSemId);

return (OK);
}
```

12.5.5 故障检验

如果 INCLUDE_MMU_FULL_SHOW 被包括在 VxWorks 的工程工具中，用户可以用函数 vmContextShow( )来把虚拟内存上下文输出到标准输出设备上。在以下例子中，当前虚拟内存上下文被列了出来。从 0x0 到 0x59fff 之间的虚拟地址是写保护的；0xff800000 到

0xffbfffff之间的是非缓存的；0x2000000 到 0x2005fff之间的是私有的。所有有效的地址入口都被列了出来，并标注了一个 *V*+，无效的地址入口没有被列出来。

```
-> vmContextShow 0
value = 0 = 0x0
```

输出被送至标准输出设备上，如以下所示：

```
VIRTUAL ADDR    BLOCK LENGTH    PHYSICAL ADDR   STATE
0x0             0x5a000         0x0             W- C+ V+ (global)
0x5a000         0x1f3c000       0x5a000         W+ C+ V+ (global)
0x1f9c000       0x2000          0x1f9c000       W+ C+ V+ (global)
0x1f9e000       0x2000          0x1f9e000       W- C+ V+ (global)
0x1fa0000       0x2000          0x1fa0000       W+ C+ V+ (global)
0x1fa2000       0x2000          0x1fa2000       W- C+ V+ (global)
0x1fa4000       0x6000          0x1fa4000       W+ C+ V+ (global)
0x1faa000       0x2000          0x1faa000       W- C+ V+ (global)
0x1fac000       0xa000          0x1fac000       W+ C+ V+ (global)
0x1fb6000       0x2000          0x1fb6000       W- C+ V+ (global)
0x1fb8000       0x36000         0x1fb8000       W+ C+ V+ (global)
0x1fee000       0x2000          0x1fee000       W- C+ V+ (global)
0x1ff0000       0x2000          0x1ff0000       W+ C+ V+ (global)
0x1ff2000       0x2000          0x1ff2000       W- C+ V+ (global)
0x1ff4000       0x2000          0x1ff4000       W+ C+ V+ (global)
0x1ff6000       0x2000          0x1ff6000       W- C+ V+ (global)
0x1ff8000       0x2000          0x1ff8000       W+ C+ V+ (global)
0x1ffa000       0x2000          0x1ffa000       W- C+ V+ (global)
0x1ffc000       0x4000          0x1ffc000       W+ C+ V+ (global)
0x2000000       0x6000          0x1f96000       W+ C+ V+ (global)
0xff800000      0x400000        0xff800000      W- C- V+ (global)
0xffe00000      0x20000         0xffe00000      W+ C+ V+ (global)
0xfff00000      0xf0000         0xfff00000      W+ C- V+ (global)
```

12.5.6　需警惕的问题

被标为全局的存储区不能用 vmMap( )来重新映射。可以使用函数 vmGlobalMap( )来增加全局虚拟存储区。关于增加全局虚拟内存的进一步信息，请参见“12.5.2　私有虚拟内存”。

MMU 的性能按结构而改变；实际上，某些结构可以增加系统的不确定性。关于其他信息，请参见关于用户的硬件的具体结构文件。