# VxWorks高级程序设计

李方敏　编著

清华大学出版社
北　京

## 内 容 简 介

全书深入而系统地讲解了 VxWorks 高级程序设计的重点和难点，尤其对 POSIX 编程、I/O 系统、网络应用编程等作了详细的介绍，并给出了众多的实用编程技巧。同时，本书对于 VxWorks 中出现的新技术及其优秀特性也作了详细的介绍。

本书共 12 章，内容包括 wind 内核、任务间通信、POSIX 编程、信号、I/O 系统、文件系统、VxWorks 网络整体分析、网络应用编程、网络驱动（END）、BSP 概述、VxWorks 映像、VxWorks 启动过程等知识。本书内容详实、实例丰富、可读性强，是 VxWorks 中、高级开发人员的一本不可多得的参考书籍。

版权所有，翻印必究。
本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

图书在版编目（CIP）数据

VxWorks 高级程序设计/李方敏编著. -北京：清华大学出版社，2004

ISBN 7-302-08127-1

Ⅰ. V… Ⅱ. 李… Ⅲ. 实时操作系统，VxWorks Ⅳ. TP316.2

中国版本图书馆 CIP 数据核字（2004）第 012635 号

出 版 者：清华大学出版社
http://www.tup.com.cn
社 总 机：010-62770175
地　　址：北京清华大学学研大厦
邮　　编：100084
客户服务：010-62776969
责任编辑：曾　刚
封面设计：秦　铭
版式设计：郑铁文
印 刷 者：北京市清华园胶印厂
装 订 者：三河市化甲屯小学装订二厂
发 行 者：新华书店总店北京发行所
开　　本：185×260　印张：25.5　字数：570 千字
版　　次：2004 年 5 月第 1 版　2004 年 5 月第 1 次印刷
书　　号：7-302-08127-1/TP・5871
印　　数：1～4000
定　　价：36.00 元

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770175-3103 或(010)62795704

# 目　录

第 1 章　wind 内核 .......... 1
1.1　内核概述 .......... 1
1.1.1　实时内核 .......... 1
1.1.2　微内核 .......... 3
1.1.3　任务调度 .......... 5
1.2　任务属性 .......... 8
1.2.1　任务控制块（WIND_TCB） .......... 9
1.2.2　任务栈 .......... 10
1.2.3　出错状态 .......... 12
1.2.4　钩子函数 .......... 14
1.2.5　任务状态 .......... 16
1.2.6　系统任务 .......... 18
1.3　内核功能接口 .......... 18
1.3.1　激活内核 .......... 18
1.3.2　任务创建 .......... 19
1.3.3　任务控制 .......... 23
1.3.4　任务结束 .......... 27
1.3.5　任务重启 .......... 28
1.3.6　调度控制 .......... 31
1.3.7　其他辅助函数 .......... 32
1.4　多任务与函数重入 .......... 32

第 2 章　任务间通信 .......... 36
2.1　概述 .......... 36
2.2　共享内存 .......... 37
2.3　信号量 .......... 37
2.3.1　概述 .......... 37
2.3.2　二进制信号量 .......... 42
2.3.3　互斥信号量 .......... 42
2.3.4　计数信号量 .......... 45
2.3.5　共享内存信号量 .......... 46
2.4　消息队列 .......... 46
2.4.1　概述 .......... 46
2.4.2　普通消息队列 .......... 51
2.4.3　共享内存消息队列 .......... 53
2.4.4　信号量和消息队列实验 .......... 53

2.5 管道 ........ 58
2.5.1 概述 ........ 58
2.5.2 使用管道 ........ 60
2.5.3 管道 I/O 控制 ........ 62
2.5.4 管道示例 ........ 63
2.6 信号 ........ 67
2.7 socket ........ 70

第 3 章 POSIX 编程 ........ 71
3.1 POSIX 标准简介 ........ 71
3.2 时钟和定时器 ........ 72
3.2.1 概述 ........ 72
3.2.2 时钟 ........ 75
3.2.3 定时器 ........ 75
3.2.4 看门狗 ........ 78
3.2.5 示例 ........ 79
3.3 内存锁定 ........ 83
3.4 线程 ........ 83
3.4.1 线程创建 ........ 85
3.4.2 动态库初始化 ........ 88
3.4.3 线程私有数据 ........ 90
3.4.4 线程互斥与同步 ........ 94
3.4.5 线程结束 ........ 102
3.4.6 线程撤销 ........ 105
3.5 任务调度 ........ 109
3.5.1 概述 ........ 109
3.5.2 调度策略 ........ 110
3.5.3 调度实现 ........ 112
3.6 信号量 ........ 116
3.6.1 概述 ........ 116
3.6.2 初始化信号量 ........ 118
3.6.3 信号量基本操作 ........ 120
3.6.4 删除信号量 ........ 121
3.7 消息队列 ........ 122
3.7.1 概述 ........ 122
3.7.2 打开消息队列 ........ 126
3.7.3 传递消息 ........ 127
3.7.4 消息到达通知 ........ 129
3.7.5 消息队列示例 ........ 131

第 4 章 信号 ........ 136
4.1 信号概述 ........ 136
4.2 信号处理函数 ........ 140

4.3 BSD 信号接口 ........ 141
4.4 POSIX 信号接口 ........ 143
4.4.1 阻塞信号集 ........ 144
4.4.2 信号处理函数 ........ 145
4.4.3 同步处理 ........ 146
4.5 POSIX1003.1b 扩展信号接口 ........ 148
4.5.1 扩展信号处理函数 ........ 148
4.5.2 发送队列信号 ........ 150
4.5.3 队列信号处理 ........ 151
4.6 信号的影响 ........ 153
4.6.1 系统调用中断 ........ 153
4.6.2 函数重入影响 ........ 155

**第 5 章　I/O 系统 ........ 156**

5.1 I/O 系统概述 ........ 156
5.1.1 I/O 系统层次结构 ........ 156
5.1.2 文件、设备和驱动程序 ........ 157
5.2 基本 I/O ........ 159
5.2.1 标准 I/O ........ 160
5.2.2 打开和关闭 ........ 162
5.2.3 创建和删除 ........ 163
5.2.4 读写 ........ 164
5.2.5 文件截平 ........ 165
5.2.6 I/O 控制 ........ 166
5.3 I/O 复用（Select） ........ 166
5.4 其他 I/O ........ 170
5.4.1 缓冲 I/O（ansiStdio） ........ 170
5.4.2 格式化 I/O（fioLib） ........ 172
5.4.3 消息记录（logLib） ........ 173
5.5 异步 I/O（AIO） ........ 175
5.5.1 AIO 控制块 ........ 176
5.5.2 AIO 函数 ........ 177
5.5.3 用 AIO 的实例 ........ 182
5.6 常用的 VxWorks 设备 ........ 189
5.6.1 串行终端设备 ........ 189
5.6.2 伪内存设备 ........ 190
5.6.3 NFS 设备 ........ 195
5.6.4 非 NFS 网络文件系统设备（netDrv 设备） ........ 197
5.6.5 RAM 盘 ........ 198
5.7 I/O 系统内部结构 ........ 201
5.7.1 驱动程序 ........ 202
5.7.2 设备 ........ 204

5.7.3 文件描述符 ........ 206
5.7.4 块设备驱动 ........ 209
5.8 串口 tty 设备 ........ 212
5.8.1 串口的层次 ........ 212
5.8.2 串口初始化过程 ........ 213
5.8.3 创建 tty 设备 ........ 213
5.8.4 tty 输入输出 ........ 216
5.8.5 控制 tty ........ 216
5.9 编写 SCC 驱动 ........ 219
5.9.1 tty 数据结构 ........ 220
5.9.2 xxDrv 数据结构 ........ 223
5.9.3 xxDrv 程序结构 ........ 225
5.9.4 查询支持 ........ 236

第 6 章 文件系统 ........ 239
6.1 文件系统概述 ........ 239
6.2 CBIO ........ 239
6.2.1 基本 CBIO ........ 240
6.2.2 CBIO 磁盘缓存 ........ 241
6.2.3 CBIO 卷设备 ........ 243
6.2.4 ioctl ........ 247
6.3 dosFs 文件系统 ........ 248
6.3.1 卷结构 ........ 248
6.3.2 使用 dosFs ........ 250
6.3.3 挂装与卸载 ........ 255
6.3.4 文件和目录 ........ 255
6.3.5 ioctl ........ 259
6.3.6 连续文件 ........ 262
6.4 rawFs 文件系统 ........ 262

第 7 章 VxWorks 网络整体分析 ........ 265
7.1 概述 ........ 265
7.1.1 TCP/IP 协议简介 ........ 265
7.1.2 VxWorks 网络栈 ........ 266
7.2 网络数据流分析 ........ 269
7.2.1 网络存储组织 ........ 269
7.2.2 数据组织 ........ 271
7.2.3 接收：从驱动程序到应用程序的数据流 ........ 273
7.2.4 发送：从应用程序到驱动程序的数据流 ........ 274
7.2.5 查看函数 ........ 275
7.3 远程访问服务 ........ 276
7.3.1 远程登录 rlogin 和 TELNET ........ 276
7.3.2 NFS 服务器 ........ 277

7.3.3 FTP 服务器 ........ 277
7.3.4 NFS 客户端 ........ 278
7.3.5 FTP 客户和 RSH ........ 278
7.3.6 TFTP 客户端 ........ 278

**第 8 章 网络应用编程 ........ 281**

8.1 socket 概述 ........ 281
8.2 网络程序设计的特殊之处 ........ 283
8.3 socket 通信属性 ........ 285
8.4 socket 端点地址 ........ 287
8.4.1 数据结构表示 ........ 287
8.4.2 协议端口号 ........ 289
8.4.3 地址操作函数 ........ 289
8.5 socket 应用框架 ........ 290
8.6 面向连接的 socket 应用 ........ 292
8.6.1 创建 socket ........ 292
8.6.2 绑定端点地址 ........ 294
8.6.3 建立连接 ........ 295
8.6.4 在连接的 socket 上发送和接收 ........ 301
8.6.5 关闭连接 ........ 303
8.6.6 面向连接的 socket 示例 ........ 304
8.7 无连接的 socket 应用 ........ 309
8.7.1 sendto 和 recvfrom ........ 309
8.7.2 无连接的 socket 示例 ........ 311
8.7.3 无连接 socket 和 connect ........ 315
8.7.4 多播的实现 ........ 317
8.7.4 广播的实现 ........ 323
8.8 裸层 socket ........ 326
8.8.1 报文格式 ........ 327
8.8.2 发送和接收 ........ 331
8.8.3 示例：Traceroute ........ 333
8.9 socket 应用高级话题 ........ 340
8.9.1 I/O 控制 ........ 340
8.9.2 socket 选项 ........ 340
8.9.3 I/O 复用 ........ 347
8.9.4 超越 I/O 复用限制 ........ 349
8.9.5 深入底层处理 ........ 352

**第 9 章 网络驱动（END） ........ 354**

9.1 网络驱动层次结构 ........ 354
9.1.1 MUX 和协议层接口 ........ 354
9.1.2 END 驱动和 MUX 接口 ........ 356
9.2 装载 END 驱动 ........ 357

第 10 章 BSP 概述 ........ 360
10.1 BSP 功能 ........ 360
10.2 BSP 标准规范 ........ 361
10.3 BSP 组织结构 ........ 362
10.4 BSP 支持主机/目标系统交叉开发环境 ........ 363
10.5 BSP 允许将应用系统移植到其他体系下 ........ 364
10.6 模板和参考 ........ 365
10.7 设备驱动开发中需要考虑的问题 ........ 365

第 11 章 VxWorks 映像 ........ 368
11.1 符号表 ........ 368
11.2 目标模块格式（OMF） ........ 369
11.3 映像类型 ........ 370
11.3.1 BSP 引导映像 ........ 372
11.3.2 VxWorks 系统映像 ........ 375

第 12 章 VxWorks 启动过程 ........ 377
12.1 目的、策略与过程概述 ........ 377
12.2 引导阶段 ........ 379
12.2.1 romInit( ) ........ 380
12.2.2 romStart( ) ........ 382
12.2.3 sysInit( ) ........ 386
12.3 准备激活内核 ........ 387
12.3.1 usrInit( ) ........ 387
12.3.2 sysHwInit( ) ........ 388
12.4 激活内核 kernelInit ........ 394
12.5 根任务：tUsrRoot ........ 395

# 第1章　wind 内 核

## 1.1　内 核 概 述

VxWorks 操作系统内核称为 wind 内核，下面从实时性能、核结构、调度特点等方面初步探讨。

### 1.1.1　实时内核

“实时”表示控制系统能够及时处理系统中发生的要求控制的外部事件。从事件发生到系统产生响应的反应时间称为延迟（Latency）。对于实时系统，一个最重要的条件就是延迟有确定的上界（这样的系统属于确定性系统）。满足这个条件后，根据这个上界大小再区分不同实时系统的性能。这里，“系统”是从系统论的观点讲的一个功能完整的设计，能够独立和外部世界交互，实现预期功能，包括实时硬件系统设计、实时操作系统设计、实时多任务设计 3 部分。后两者可以概括为实时软件系统设计。实现实时系统是这 3 部分有机结合的结果。

从另外一个角度，即实时程度看，可以把系统分为硬实时系统和软实时系统。硬实时系统是这样一种系统，它的时间要求有一个确定的底线（Deadline），超出底线的响应属于错误的结果，系统将会崩溃，上面所说的实时系统属于硬实时系统。对于软实时系统来说，“实时性”是个程度概念，在提交诸如中断、计时和调度的操作系统服务时，系统定义一个时间范围内的延迟。在该范围内，越早给出响应越有价值，只要不超出范围，晚点给出的结果价值下降，但可以容忍。

#### 1．实时硬件系统设计

实时硬件系统设计是其他两部分的基础。实时硬件系统设计要求满足在软件系统充分高效的前提下，必须提供足够的处理能力。例如，硬件系统提供的中断处理逻辑能同时响应的外部事件数量、硬件反应时间、内存大小、处理器计算能力、总线能力等，以保证最坏情况下所有计算仍然得以完成。多处理的硬件系统还包括内部通信速率设计。当硬件系统不能保证达到实时要求时，可以确信整个系统不是实时的。

目前，各种硬件速度不断提高，先进技术大量涌现，硬件在大多数应用中已经不是实时系统的瓶颈。因而，实时系统的关键集中在实时软件系统设计，这方面也成了实时性研

究的主要内容，也是最复杂的部分。许多场合甚至对实时系统和实时操作系统不加区分。

### 2．实时操作系统设计

先来看实时操作系统性能评价的几个主要指标：

- 中断延迟时间：从接收中断信号操作系统做出响应，并完成进入中断服务程序的时间；
- 任务切换时间：多任务之间进行切换所花费的时间；
- 系统响应时间：系统在发出处理要求到系统给出应答信号的时间。系统响应时间从整体上评价操作系统，综合了前面两个指标。

从实时性角度看，操作系统经历了前后台系统、分时操作系统和实时操作系统 3 个阶段。

前后台系统其实没有操作系统，系统中只运行一个无限主循环，没有多任务的概念，但是通过中断服务程序响应外部事件。在前后台系统中，对外部事件的实时响应特性从两方面看。(1) 中断延迟：主循环一般保持中断开放状态，因此前后台系统中断响应非常快，并且通常允许嵌套；(2) 系统响应时间：需要经历一次主循环才能对中断服务程序中采集的外部请求进行处理，因此系统响应时间决定于主循环周期。

**分时操作**系统将系统计算能力分成时间片，按照一定的策略分配给各个任务，通常在分配过程中追求某种意义上的公平。分时操作系统不保证实时性。

**实时操作系统（Real Time OS，RTOS）**的目的是实现对外部事件的实时响应，即根据前面对实时性的定义，实时操作系统必须在确定的时间内给出响应。实时操作系统必须满足下面几个条件：

- 可抢占的内核；
- 可抢占的优先级调度；
- 中断优先级；
- 中断可嵌套；
- 系统服务的优先级由请求该服务的任务的优先级确定；
- 优先级保护（优先级翻转保护）；
- 前述实时操作系统性能评价指标具有固定上界。

满足上面的必要条件后，内核内部具体的实现机制就决定了其实时性的优劣。**VxWorks** 的 wind 是一个真正的实时微内核，满足上述条件。同时，**wind** 采取单一实时地址空间，任务切换开销非常低，相当于在 **UNIX** 这样的主机上切换到相同进程内的另一个线程，并且没有系统调用开销。高效的实时设计使 wind 在从工业现场控制到国防、航空等众多领域中表现出优秀的实时性。

### 3．实时多任务设计

具备前面两个实时条件后，实时系统的最终实现就取决于实时多任务设计。这部分是最能体现系统设计者艺术的部分，也是最有挑战性的部分。这部分设计内容涵盖了一个完

整的软件工程过程：需求分析（需求建模）、概要设计（设计建模）、模块设计、模块实现、调试、发布。和一般软件工程过程不同的是上述每一步中都需要考虑对实时性的满足，因此更加复杂、也许需要专门通过一本书的篇幅对此进行探讨。我们只简单分析设计过程中实时多任务设计需要面临的关键问题：多任务划分、多任务分配、多任务调度。“关键”是因为它们是决定系统实时性的主要因素，具体设计时还存在其他一些问题，如任务间通信机制选择，中断服务程序设计等，都对系统实时性产生重大影响，但不属于本质的问题。对多任务划分、多任务分配和多任务调度的设计对应软件工程过程的概要设计和模块设计阶段。

**多任务划分**即如何将整个系统功能设计为不同的任务来实现，任务之间采取怎样的耦合关系，划分的粒度如何等。这里，“任务”也包括中断，因此也包括将什么功能放在中断中实现，什么功能放在常规的任务中处理。在根据数据流划分任务时，影响划分的要素包括数据流之间的并行和串行关系；根据控制流划分任务时，考虑的要素是控制的因果关系。

多任务划分影响着多任务分配和多任务调度。**多任务分配**决定任务放在哪个处理器上完成（存在多个处理器时），以及网络环境下任务如何分配。

多任务划分目的的实现需要多任务调度，**多任务调度**的设计目的是关键任务得到实时响应，同时整体上所有任务的设计内容都在允许的时间内完成。多任务调度的内容包括系统调度策略的选择，任务优先级的确定，以及任务间竞争和合作的设计。在划分多任务时，已经考虑了各任务所担任职责的轻重缓急，多任务调度时需要根据这种紧急程度分配合理的优先级；调度还必须使不同优先级的协作任务有效地同步。

多任务划分、多任务分配和多任务调度三者是有机的整体，而多任务划分则是其中决定性的部分。任何一个因素设计不合理都将影响整个系统的实时性。

### 1.1.2　微内核

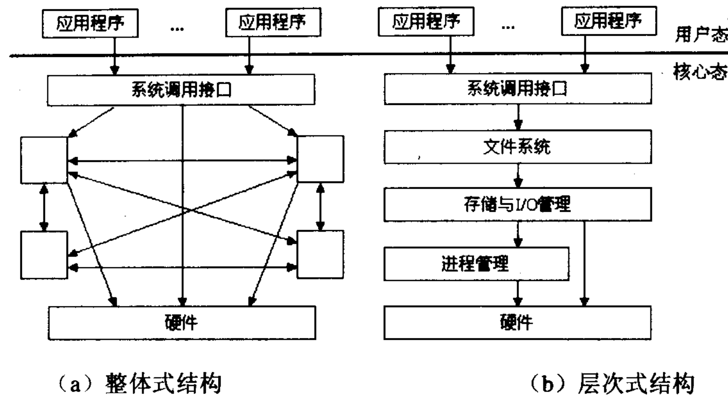
传统上，一个操作系统分为核心态和用户态。内核在核心态运行，为用户态的应用程序服务。内核是操作系统的灵魂和中心，决定了操作系统的效率和应用领域。在设计操作系统时，内核包含哪些功能以及内核功能采取何种组织结构，都是由设计者决定的。我们从内核功能和结构特点看，具有**整体式内核、层次式内核、微内核**三种不同形式。

整体式内核结构的操作系统实质上“无结构”。操作系统功能由一系列模块堆砌而成，任何模块之间可进行任意调用。整体式内核结构的操作系统不进行任何的数据封装和隐藏，在具有较高效率的同时，存在着难以扩展和升级的缺点。CP/M 和 MS-DOS 属于此类结构的操作系统。

层次式内核结构的操作系统将模块功能划分为不同层次，下层模块封装内部细节，上层模块调用下层模块提供的接口。UNIX，LINUX，VAX/VMS，MULTICS 等属于层次结构操作系统。层次化使操作系统结构简单，易于调试和扩展。两种操作系统的内核结构如图 1-1 所示。

不管整体式结构，还是层次式结构，它们的操作系统都包括了许多将其用于各种可能

领域时需要的功能，故被称为**宏内核**操作系统，以至可以认为该内核本身便是一个完整的操作系统。以 UNIX 为例，其内核包括了进程管理、文件系统、设备管理、网络通信等功能，用户层仅仅提供一个操作系统外壳和一些实用工具程序。




（a）整体式结构　　（b）层次式结构

图 1-1　内核结构

嵌入式操作系统大多采用**微内核**结构。微内核操作系统是近二十年新发展起来的技术，内核非常小但效率高，从数十 KB 到数百 KB 字节，适合于资源相对有限的嵌入式应用。微内核将很多通用操作的功能从内核中分离出来（如文件系统，设备驱动，网络协议栈等），只保留最基本的内容。

一般认为微内核操作系统具有如下优点：

- 统一的接口，在用户态和核心态之间无需进程识别；
- 可伸缩性好，易于扩充，能适应硬件更新和应用变化；
- 可移植性好，操作系统要移植到不同的硬件平台上，只需修改微内核中极少代码即可；
- 实时性好，内核响应速度快，可以方便地支持实时处理；
- 安全可靠性高，微内核将安全性作为系统内部特性来进行设计，对外仅使用少量应用编程接口；
- 适合分布式计算环境。内核为进程传递消息的方式天然适合 RPC 这一计算模式。

由于操作系统核心常驻内存，而微内核结构精简了操作系统的核心功能，内核规模比较小，一些功能都移到了外存上，所以微内核结构十分适合嵌入式的专用系统，如图 1-2 所示的 wind 微内核结构。

## 微内核的不同实现模式

从宏内核系统到微内核，操作系统结构发生了根本性的变化，导致的影响是多方面的。除了上面说明的微内核的优越性外，微内核的批评者也指出了微内核的缺陷：在微内核操作系统中，由于许多传统的操作系统功能改为用户任务实现，一般采取客户/服务器模型，即传统操作系统中许多系统功能作为微内核结构下的服务器任务，这样使任务间的数据交

换量更大，需要在内核与任务之间进行大量的数据复制，因此影响了系统性能。有研究人员指出著名的微内核操作系统 Mach 性能远不如宏内核的 BSD UNIX（当然这里与应用类型有关，例如用嵌入式微内核的系统实现一个文件服务器显然比 UNIX 效率低）。

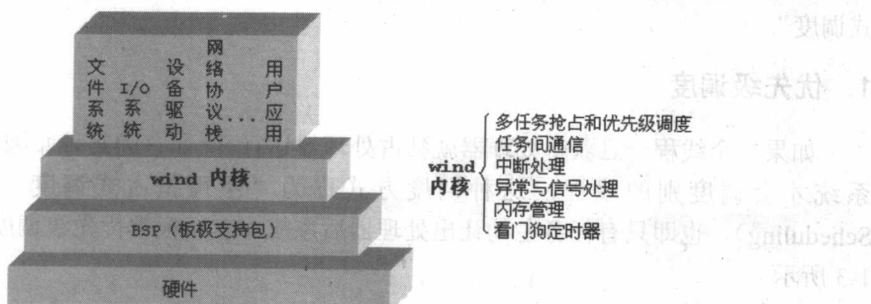



图 1-2 wind 微内核结构

为了提高微内核效率，有两种实现模式：**受保护的虚地址空间**模式和无保护的**单一实地址空间**模式。前者在宏内核操作系统（如 UNIX）和某些微内核操作系统（如 QNX）中采用。这种模式的优点是显而易见的：任务独立运行，不受其他任务错误影响，系统可靠性高。

VxWorks 的 wind 微内核采取单一实地址空间模式，所有任务在同一地址空间运行，不区分核心态和用户态。其优势在于：

- 任务切换时不需要进行虚拟地址空间切换；
- 任务间可以直接共享变量，不需要通过内核在不同的地址空间复制数据；
- 系统调用时不需要在核心态和用户态之间切换，相当于直接的函数调用。

& 系统调用时需要从用户态切换到核心态，以执行用户态下不能执行的操作，在许多处理器上这是通过一个 trap 中断处理完成的。VxWorks 中不存在这样的切换，因此系统调用和一般函数调用没有什么差别。但是我们仍然沿用一般说法。

对于两种模式孰优孰劣，各自的支持者们进行了大量的争论。比较各有所长的东西往往非常困难。我们倾向于认为，对于嵌入式实时应用，单一实地址模式要合适一些。许多实践也证明，依靠虚地址保护来提高可靠性总存在局限性，毕竟程序运行出了错误。有时虚地址保护只是使已经出现的错误经过一个延迟、积累和放大的过程，用过 Windows 就会有这种感触。而经过大量关键应用检验的 VxWorks 操作系统，则被充分证明是高度可靠的（当然，可靠的系统由可靠的操作系统和可靠的应用系统组成）。对于从“单片机＋汇编语言”成长起来的开发人员，可能更喜欢单一实地址空间的系统。

### 1.1.3 任务调度

实时系统和分时系统的一个显著差异体现在调度策略上。实时系统调度关心的是对实

时事件的相应延迟，而传统的分时系统调度时要考虑的目标是多方面的：公平、效率、利用率、吞吐量等。因此实时系统通常采用优先级调度，即操作系统总是从就绪任务队列中选择最高优先级运行。优先级调度根据调度的时机又分为“不可抢占式调度”和“可抢占式调度”。

## 1. 优先级调度

如果一个线程一旦获得处理器就独占处理器运行，除非它因某种原因决定放弃处理器，系统才会调度别的线程，这种调度方式称为“**不可抢占式调度**”（Non-Preemptive Scheduling），也即只有任务主动让出处理器后系统才重新根据优先级调度选择任务，如图1-3所示。

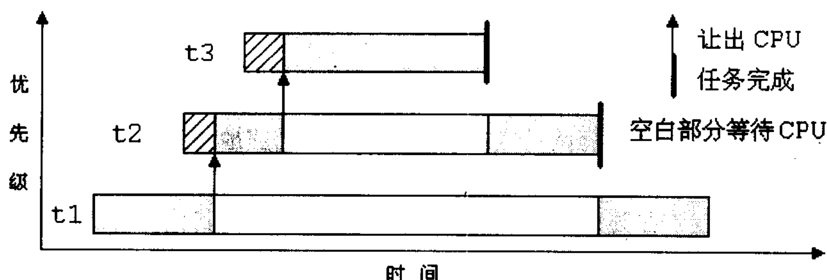



图 1-3　不可抢占优先级调度

在图1-3中，高优先级任务t2和t3就绪后（图中斜线阴影部分），并没有立即被调度，而是低优先级让出处理器时，才根据优先级高低选择任务运行。高优先级任务和低优先级任务之间是紧密合作的关系，低优先级任务必须设计为不使高优先级任务长时间等待。

当注重实时响应时，应该采用“**可抢占式调度**”（Preemptive Scheduling）。只要有更高优先级任务就绪，系统立即中断当前任务来调度高优先级任务，确保任意时刻最高优先级任务得到处理器，如图1-4所示。

在图1-4中，抢占式优先级调度使任务t2和t3就绪时总能立即得到处理器，其实时响应特性优于不可抢占调度。

对于区分核心态和用户态的系统，优先级调度还是微内核实现的基础。由于可抢占式内核中的核心态任务也可以随时让位给比其优先级高的任务，使得系统可以把一些实时设备的操作放在内核之外，通过“任务”完成，从而简化了内核设计。在传统的操作系统（如UNIX）中，必须将所有对时间要求较高的操作放在内核中实现，导致庞大的内核。

wind内核支持256级优先级：0~255。优先级0为最高优先级，优先级255为最低优先级。任务优先级在创建时确定，并允许程序运行中动态修改。但是对于内核而言，从就绪队列中选择一个任务调度时优先级是确定的，换言之，内核不会动态计算每个任务的优先级，因此这种调度策略属于**静态调度策略**；相对地，**动态调度策略**调度时需要根据某个目标（例如任务完成时间底线）动态确定任务优先级并调度。静态调度策略效率显著高于动态调度策略，并且足够满足应用需要，因而被普遍采用，包括POSIX 1003.1b标准对任

务调度的定义。

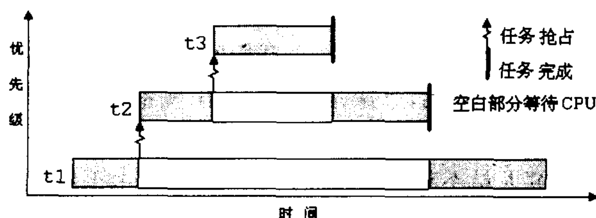



图 1-4 抢占式优先级调度

## 2. Round-Robin 调度

前面介绍的优先级调度存在这样的问题：如果没有被更高优先级任务抢占，或者因阻塞等原因让出处理器，任务将一直运行下去，在此情况下，同优先级任务将得不到运行。**Round-Robin 调度**基于这样的哲学：在更高优先级任务调度依然优先运行的前提下（这一点和前面一样），同优先级任务之间调度时追求一定意义上的公平。

Round-Robin 调度将任务运行划分为时间片，当任务运行一个时间片后，内核将其调出处理器并放在同优先级就绪任务队列尾部；调度时选择最高优先级就绪队列首部任务。Round-Robin 调度的效果是将每个任务运行一个时间片后“让出”处理器给下一个任务，如轮转一样，也称**轮转调度**。可见，Round-Robin 调度并没有改变“基于优先级”和“可抢占”这两个实时调度的特征。说在 VxWorks 中，Round-Robin 调度是基于优先级可抢占调度的一个“附加特征”，如图 1-5 所示。

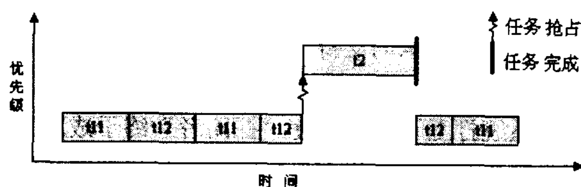



图 1-5 Round-Robin 调度

在 POSIX 1003.1b 中，基于优先级的可抢占调度定义为 **SCHED_FIFO**；Round-Robin 调度定义为 **SCHED_RR**。3.5 节“任务调度”部分深入介绍任务调度细节。

## 3. 优先级翻转问题

实时系统中，如果任务调度采用基于优先级的方式，则传统的资源共享访问机制在系统运行时很容易造成优先级翻转问题，即当一个高优先级任务访问共享资源时，该资源已被一低优先级任务占有，而这个低优先级任务在访问共享资源时可能又被其他一些低优先级的任务抢先，因此造成高优先级任务被许多较低优先级的任务阻塞，高优先级任务在低优先级任务之后运行，看起来像低优先级任务抢占了高优先级任务，即发生了**优先级翻转**（Priority Inversion）。

一个最为著名的优先级翻转问题的例子是“火星探路者”（Pathfinder），它采用了VxWorks实时内核设计。1997年7月4日，“探路者”弹出气囊在火星表明登陆，放出“漫游者”探测车，收集传回地球的大量数据。在开始的几天里，“探路者”被誉为完美无缺的作品，引起巨大轰动。但是几天后，“探路者”开始出现系统复位、数据丢失的现象。当时解释为“软件小故障”、“同时处理的事情太多”等。

为了找出故障，研究人员不断模拟“探路者”在火星上工作的条件和过程，终于在实验室中再现了系统复位的现象，并记录下来。根据分析，有如下两个任务需要互斥访问共享“信息总线”：（1）总线管理任务，具有最高优先级，运行频繁，进行总线数据I/O；（2）气象数据收集任务，优先级低，运行较少，收集数据并通过互斥信号量将数据发布到“总线”。如果数据收集任务持有信号量期间，总线管理任务就绪并且也申请获取信号量，则总线管理任务阻塞，直到收集任务释放信号量。看起来工作得很好，因为收集任务很快就会完成，高优先级的总线管理任务会很快得到运行。但是，另有一个需要较长时间运行的通信任务，其优先级比总线管理任务低，但是比数据收集任务高。在很少的情况下，如果通信被中断激活，并刚好在总线管理任务等待数据收集任务完成期间就绪，它将被系统调度，从而比它优先级低的数据收集任务得不到运行，并因此使最高优先级的总线管理任务无法运行。在经历较长时间后，看门狗观测到“总线”没有活动，将此解释为严重错误并使整个系统复位。

很显然需要防止优先级翻转以确保系统的实时响应。常见的解决办法之一是使用**优先级继承**协议（Priority Inheritance Protocol）。当高优先级任务需要低优先级任务占用资源时，将低优先级任务的优先级别提高到和高优先级同样的级别，即相当于低优先级任务继承高优先级任务的优先级级别。

VxWorks实现了对优先级继承的支持，但是默认情况下该功能被关闭，也就是说有可能发生优先级翻转问题，也就导致了“探路者”中问题的发生。不过，研究人员及时查出了问题并给在火星上的系统打上了补丁，使“探路者”终于成功完成使命。美国航空航天局JPL（喷气推进实验室）决定第二代火星探测器项目仍然与风河系统进行合作，被引为VxWorks功能强大和可靠性的例证。

防止优先级翻转的另外一种协议是**优先级天花板**（Prioceiling），其设计策略是对优先级翻转采取“预防”，而不是“补救”。也就是说：不论是否阻塞了高优先级任务，持有该协议的任务在执行期间都被赋予优先级天花板看作的优先级，以使任务尽快完成操作，因此可以把任务优先级天花板看作是更“积极”的一种保护优先级的方式。VxWorks对POSIX信号量的支持实现了该协议。

我们后面将对优先级翻转问题做更具体的描述。

## 1.2 任务属性

VxWorks任务具有两个显著不同于主机操作系统的特点：（1）VxWorks任务和内核具

有相同的权限，都能够执行处理器所支持的全部指令；（2）所有任务（包括内核）共享同一实地址空间（不进行虚拟内存管理），不同任务的数据没有任何保护机制。这两个特性一方面使 VxWorks 具有很高的效率，同时，由于没有 VxWorks 任务执行指令和没有访问内存任何约束和保护，使得某个任务的错误容易造成更严重的影响，因而对代码质量提出了更高的要求。

## 1.2.1 任务控制块（WIND_TCB）

多任务设计能随时打断正在执行着的任务，对内部和外部发生的事件在确定的时间里作出响应。VxWorks 实时 Wind 内核提供了基本的多任务环境。从表面上来看，多个任务正在同时执行，实际上，系统内核根据某一调度策略让它们交替运行。系统调度器使用任务控制块的数据结构（TCB）来管理任务调度功能。任务控制块用来描述一个任务，每一任务都与一个 TCB 关联。TCB 包括了任务的当前状态、优先级、要等待的事件或资源、任务程序码的起始地址、初始堆栈指针等信息。调度器在任务最初被激活时以及从休眠态重新被激活时，要用到这些信息，TCB 使多个任务得以独立运行，如表 1-1 所示任务控制块 TCB。

表 1-1 任务控制块 TCB

| 项 目 | 内容和含义 |
|---|---|
| 任务名称 | 指针指向表示任务名称的字符串 |
| 上下文 | 程序计数器，表示任务被中止时的位置<br>CPU 状态，包括各种处理器特定的寄存器<br>栈，用于动态变量，函数调用，信号处理等<br>标准输入，标准输出，标准错误输出定义<br>延迟定时器，用于任务延迟计数<br>时间片定时器，用于 Round-Robin 调度<br>内核控制结构（Kernel Control Structures）<br>信号处理设置信息，包括阻塞信号集，信号处理方式，信号状态等<br>错误状态，每个任务都有一个独立的全局 errno 的副本<br>调试和性能监视状态<br>任务变量（可选）<br>浮点上下文（可选） |
| 异常信息 | 因处理器体系不同而有差异，用于异常处理 |
| 退出码 | exitCode 作调试之用 |

为了便于调试，每个任务都有一个独一无二的字符串表示的名称，在任务被创建时由用户程序指定或者系统默认生成。几乎所有的任务控制函数都采用任务 ID（等于 TCB 地

址）表示一个任务。VxWorks 提供任务名称和任务 ID 之间的转换函数。

TCB 的一个重要内容就是任务上下文（ContExT），代表了任务运行状态。VxWorks 的任务切换就是将当前任务（被换出 CPU）的上下文保存到该任务的 TCB，然后从调度程序（Scheduler）选择新任务（被换入 CPU）的 TCB 中恢复上下文。

上下文切换分两种情况：

- 同步上下文切换，引起的原因是当前运行的任务执行下列操作：（1）进行阻塞、延迟、挂起的调用；（2）使更高优先级任务就绪而发生优先级抢占；（3）降低自身优先级或者退出；
- 异步上下文切换，通常由 ISR 使更高优先级任务就绪引起。

异步上下文切换比同步上下文切换需要更多时间。

调度程序开销主要取决于保存和恢复上下文需要复制的寄存器数，要求该过程非常快。对于特定体系的处理器而言，该数值是固定的。为提高调度程序效率，在默认情况下，VxWorks 上下文不包括浮点寄存器（即假设任务不使用浮点寄存器，在许多情况下的确如此）。任务创建时指定浮点寄存器是否属于上下文的一部分（标志 VX_FP_TASK）。

VxWorks 中，内存地址空间不是任务上下文的一部分。所有的代码运行在同一地址空间。如每一任务需各自的内存空间，需可选产品 VxVMI 的支持。

## 1.2.2 任务栈

每个任务都有独立的栈空间，栈用于任务的函数调用，分配自动变量和函数返回值。任务控制块 WIND_TCB 记录了位置和大小等栈信息。WIND_TCB 本身放在任务栈开始部分。

任务栈大小的设置必须合理，太大会浪费内存空间，太小时可能引起栈溢出。在 VxWorks 中，所有任务在同一地址空间运行，任务之间没有任何地址保护机制，因此栈溢出会引起连锁反应，可能导致系统崩溃，或者出现难以调试的意外结果。

栈大小设置没有可以套用的公式，一般凭经验设置一个较大的值，以存储空间换取可靠性。分析程序所有可能的分支和调用，从而计算出需要的栈大小的方法，从理论上给出了最佳的栈设置方案，但是目前好像还没有这样的自动化分析工具，靠程序员手工计算似乎不大可行。好在存储器越来越便宜，因此许多应用中，人们更趋向于使用大存储器解决问题。对可靠性要求高的应用，仍然需要充分分析和测试栈大小设置是否足够。

栈大小在 taskSpawn( )创建任务时指定。

### 1．中断

只要体系和 BSP 支持，VxWorks 支持独立的中断栈。“独立”是对任务而言，对所有的 ISR 使用相同的中断栈。中断栈在系统启动时根据配置参数设置位置、大小和填充。

如果体系不支持，中断栈属于被中断任务栈的一部分。此时任务栈必须足够大，保证

最坏情况下中断嵌套也不会溢出。

对独立中断栈的支持：MC680x0（MC68060 除外），ARM，PPC，MIPS。

VxWorks for Pentium 支持程序在任务栈和专用中断栈两种方式之间动态选择，如表 1-2 所示。

表 1-2　不同处理器的中断栈

| 处理器体系 | 中断栈 | SHELL | 根任务 | WDB |
|---|---|---|---|---|
| MC680x0 | 1000 | 10000 | 10000 | 0x1000 |
| COLDFIRE | 1000 | 10000 | 10000 | 0x1000 |
| SPARC | 10000 | 50000 | 10000 | 0x2000 |
| I960 | 1000 | 40000 | 20000 | 0x2000 |
| MIPS | 5000 | 20000 | 20000 | 0x2000 |
| PPC | 5000 | 20000 | 24000 | 0x2000 |
| I80x86 | 1000 | 10000 | 10000 | 0x1000 |
| SH | 1000 | 10000 | 10000 | 0x1000 |
| AM29xxx | 10000 | 40000 | 10000 | 0x2000 |
| ARM | 决定于BSP中断结构 | 0x10000 | 0x4000 | 0x2000 |

## 2. 栈溢出检测

当某个任务栈溢出后，系统行为将难以预料。在程序开发过程中，根据经验在可能发生较深嵌套调用的地方加入一些栈检查调用 checkStack( )来检查任务栈使用情况，例如：

```
NAME         ENTRY        TID      SIZE  CUR   HIGH  MARGIN
------------ ------------ -------- ----- ----- ----- ------
tShell       _shell       23e1c78  9208  832   3632  5576
```

checkStack( )显示了单个指定任务或者所有任务的栈使用情况，包括：

- 栈大小（SIZE）；
- 栈当前使用数（CUR）；
- 历史使用峰值（HIGH）；
- 最大可能空余数（MARGIN=SIZE-HIGH）。

如果上述结果是程序在足够长的测试运行过程中充分考虑了边界条件以及各种可能引发最深嵌套调用的情况之后得到的输出，那么可以根据该结果调整任务栈的设置，比如设置栈大小为峰值的 120%：

SIZE=HIGH*1.2

如果 ISR 使用任务栈，则还要加上一个经验常数：

SIZE=HIGH*1.2+C

C根据处理器体系特点、ISR复杂程度和内存大小确定。VxWorks要求栈大小为偶数值。

VxWorks没有直接为checkStack()记录使用峰值，而是生成任务时将栈空间以值0xee进行填充，checkStack()据此检查栈使用情况。如果在生成任务时指定VX_NO_STACK_FILL，则任务栈不会被填充，因此用checkStack()得到的峰值是没有意义的。

### 1.2.3 出错状态

ANSI C标准定义了一个全局整型变量errno，用来使应用程序知道底层函数出错的详细情况。使用errno应该注意：errno总是定义为该任务（对其他主机操作系统而言为进程）最后一次调用出错时的错误值，任何系统调用/函数执行成功都不应该清除errno，即不应该执行errno=0。

当某个函数调用返回错误的结果（如返回状态值为ERROR或者返回指针为NULL）时，程序可以立即检查errno得知错误细节，该errno一直保持，直到随后某次调用中出现新的错误将其覆盖。

在VxWorks中，每个任务TCB中都记录有一个全局errno副本，属于上下文的一部分，由调度程序在上下文切换时保存和恢复，因此各个任务的出错状态相互不会影响。与此类似，ISR也使用独立的errno，但是ISR没有TCB，内核为ISR在中断栈中保存和恢复errno。从编写程序角度来看，无论属于常规任务，还是属于ISR的代码，程序都可以使用不受其他任务或者ISR影响的errno。

程序引用errno时实际上通过一个函数得到全局errno地址（errno.h）：

```
#include "errno.h"
#define errno  ( *__errno( ))
```

有经验的程序员习惯充分利用errno，这样，程序具有很好的逻辑结构和用户接口。

当发生嵌套调用时，VxWorks定义的出错状态在错误最初发生的位置定义，例如：

```
STATUS funcA ( void )
{
   errno = S_xxx_NOT_IMPLEMENTED;
   return ERROR;
}

STATUS funcB ( void )
{
   ... if ( funcA( ) != OK ) return ERROR;
   ... return OK;
}
```

函数 funcA()功能尚未实现，简单的设置 errno 为 S_xxx_NOT_IMPLEMENTED 并返回 ERROR；函数 funcB()调用 funcA()得到 ERROR 时，funcB()返回 ERROR 给调用者但不修改 errno，因为 funcA()已经定义。当然，如果错误源于 funcB()，则 funcB()应该设置 errno。VxWorks 库函数都遵循上述约定。

VxWorks 允许用户程序中不仅使用 VxWorks 库函数中对 errno 的定义，还允许用户代码中定义新的 errno，但是应该避免和 VxWorks 的定义重复。VxWorks 定义的 errno 分两部分：

| | |
|---|---|
| [bit31~bit16] | 错误产生的模块编号 |
| [bit15~bit0] | 错误编号 |

高 16 位模块编号小于 501 的部分已经被 VxWorks 库使用，用户程序可以使用从 501 开始的模块编号，低 16 位为错误编号可以任意指定。

库 errnoLib 提供 VxWorks 库函数中定义的 errno 错误信息字符串。可以通过在 shell 执行 printErrno()打印错误信息。使用库 errnoLib 需要定义 INCLUDE_STAT_SYM_TBL。例如可以在 shell 下执行：

```
-> i            （查看任务状态，ERRNO 列出十六进制表示的错误码）
NAME     ENTRY  TID    PRI    STATUS PC     SP     ERRNO DELAY
-------- ------ -----  ------ ---    ------ -----  -------
...
t16      _pRun  7641dc 100    READY  17d1c  763de4 30    0
-> printErrno(0x30)
S_errno_EADDRINUSE
```

该输出结果可以极大地方便程序调试。但是一个缺点是库 errnoLib 以目标码形式发布，必须将其构建到执行映像中才能使用。无法直接通过文本编辑器查看。这需要消耗大约 24KB 代码空间。

实际上，不使用库 errnoLib 也可以知道错误信息细节。VxWorks 将模块号定义在头文件<install-dir>\target\h\vwModNum.h 中，可以通过得到的错误信息中的模块号找到对应的错误信息，这样可以节约代码空间。以 errno 为 OX410001 的错误为例，我们得到模块号为十进制数 65（高 16 位），然后在 vwModNum.h 找到对应模块的定义：

```
#define M_msgQLib    (65 << 16),
```

因此知道错误源于 msgQLib 模块，进而在该模块定义文件<install-dir>\target\h\msgQLib.h 中了解到：

```
#define S_msgQLib_INVALID_MSG_LENGTH    (M_msgQLib | 1)
```

从而得知 errno 为 0x410001 表示 msgQLib 模块由于发送消息超出长度而出错。

除了各个模块定义的错误号，VxWorks 还定义了一系列 ANSI C 标准错误，如 EINTR，EINVAL，EIO，EAGAIN 等，本书后面将会涉及。

## 1.2.4 钩子函数

VxWorks 允许任务创建“钩子函数”，安装钩子函数后，在规定事件发生时内核自动调用。具体来说有 3 类钩子函数：

（1）任务创建钩子：在任务被创建时调用；

（2）任务调度钩子：在任务被调度时调用；

（3）任务删除钩子：在任务被删除时调用，任务“删除”包括以任何方式调用函数 taskDelete()、taskDeleteForce()和 exit()引起的系统动作。

可以指定多个钩子函数，VxWorks 确保钩子函数具有一致的调用顺序。当多个钩子函数之间存在某种依赖时，如下两点就显得很重要：

- 任务创建钩子函数和任务调度钩子函数按照其安装的顺序调用；
- 任务删除钩子函数按照与其安装顺序相反的顺序调用。

### 1．任务创建（删除）钩子

任务创建钩子或任务删除钩子必须具有如下原型定义：

```
void xxxHook ( WIND_TCB *pTcb );
```

参数 pTcb 指向被创建（删除）任务的 TCB，xxxHook 为任务创建钩子或为任务删除钩子函数名称。

钩子函数可以随时被安装/卸载，用于安装和卸载的函数为：

```
#include "taskHookLib.h"
STATUS   taskCreateHookAdd (FUNCPTR createHook);      /* 安装任务创建钩子 */
STATUS   taskCreateHookDelete (FUNCPTR createHook); /* 卸载任务创建钩子 */
STATUS   taskDeleteHookAdd (FUNCPTR deleteHook);      /* 安装任务删除钩子 */
STATUS   taskDeleteHookDelete (FUNCPTR deleteHook); /* 卸载任务删除钩子 */
```

虽然钩子函数在某个特定任务中被创建，但是创建后的钩子却被所有任务共享：在安装钩子后创建任何任务或者删除任何任务时调用钩子函数，除非将安装的钩子函数卸载。

注意，（1）任务创建钩子在任务入口函数之前被调用；（2）任务删除钩子在任务退出后被调用。即：

任务创建钩子 → 任务入口函数 → … → 任务结束 → 任务删除钩子

如果有多个钩子，其调用顺序前面已经说明。

任务删除钩子常常用于进行任务清理工作，如释放内存和关闭打开的文件。由于每次删除任务都会引起任务删除钩子调用，因此必须注意钩子函数重入性。任务创建钩子同样需要考虑代码重入。这些考虑包括使用可扩展的数据结构定义以及避免重复释放同一块内存等。

## 2. 任务调度钩子

任务调度钩子函数原型必须定义如下：

```
void switchHook ( WIND_TCB *pOldTcb, WIND_TCB *pNewTcb );
```

参数 pOldTcb 指向被调出处理器的任务的 TCB，pNewTcb 执行被调入处理器的任务的 TCB。

任务调度钩子函数由下列函数动态安装和卸载：

```
#include "taskHookLib.h"
STATUS taskSwitchHookAdd ( FUNCPTR switchHook );
STATUS taskSwitchHookDelete ( FUNCPTR switchHook );
```

任务调度钩子函数在每次 VxWorks 调度程序选择新的任务运行时被调用。与任务创建（删除）钩子不同的是，任务调度钩子在内核上下文中运行，而任务创建（删除）钩子在入参 pTcb 所指示的任务的上下文中运行。这一差异使得任务调度钩子不能和任务创建钩子及任务删除钩子函数那样进行所有的系统调用。如表 1-3 所示列出了任务调度钩子所能调用的 VxWorks 函数。

**表 1-3 任务调度钩子可以调用的系统函数**

| 库 | 允许调用的函数 |
|---|---|
| bLib | 全部函数 |
| fppArchLib | fppSave( )，fppRestore( ) |
| intLib | intContext( )，intCount( )，intVecSet( )，intVecGet( ) |
| lstLib | 全部函数 |
| mathALib | 如果使用 fppSave( )和 fppRestore( )，则全部函数可以使用 |
| rngLib | 除 rngCreate( )外的函数 |
| vxLib | vxTas( ) |
| taskLib | taskIdVerify( )，taskIdDefault( )，taskIsReady( )，taskIsSuspended( )，taskTcb( ) |

任务调度钩子可以用来检查任务调度时两个任务的状态，有时可以利用这一特点为调试提供帮助。例如，可以在钩子函数中向某端口输出一个波形，然后通过高速的逻辑分析仪捕获信号来计算某个任务执行时间。如果系统调度策略为默认的抢占优先级调度，还可以通过比较被调入调出的任务优先级判断是否由于阻塞引起的调度，或者由于优先级抢占引起的调度。如下面这段代码，当旧任务优先级比新任务优先级高时，说明高优先级任务被阻塞（也可是挂起或延迟），这时可以通过 pOldTcb 检查任务的阻塞队列。

把任务调度钩子设计为：

```
void switchHook ( WIND_TCB *pOldTcb, WIND_TCB *pNewTcb )
```

```
{
    if( pNewTcb->priority <= pOldTcb->priority )
    {
        ...     /*阻塞，挂起等引起的调度 */
    }
}
```

另一个和任务调度钩子类似的机制是 SWAP 钩子。SWAP 钩子在 WRS 文档中没有提及，但是被实现。SWAP 钩子和任务调度钩子相比，具有不同的特点：SWAP 钩子由某个任务安装后，必须显式地将 SWAP 钩子和任务关联（taskSwapHookAttach）才会被调用。同时，也因此使 SWAP 钩子就可以在任务"换出"、"换入"、"换出＋换入"条件下被调用。而任务调度钩子只要在任何任务中被安装后，就在此后任何两个任务之间切换调度时被调用。因此可以将 SWAP 钩子看成"单向"钩子，而将任务调度钩子看成"双向"钩子。

```
#include "taskHookLib.h"
STATUS   taskSwapHookAdd (FUNCPTR swapHook);
STATUS   taskSwapHookDelete (FUNCPTR swapHook);
STATUS   taskSwapHookAttach (FUNCPTR swapHook,int tid,BOOL in,BOOL out);
STATUS   taskSwapHookDetach (FUNCPTR swapHook,int tid,BOOL in,BOOL out);
```

## 1.2.5 任务状态

实时系统的一个任务可有多种状态，其中最基本的状态有 5 种：

- 运行 —— 任务获得 CPU 运行；
- 就绪 —— 任务只等待系统分配 CPU 资源；
- 阻塞 —— PENDED，因任务需等待某些不可利用的资源而被阻塞；
- 挂起 —— SUSPENDED，如果系统不需要某一个任务工作，则这个任务处于挂起状态。调度程序忽略该状态下的任务，但是对于任务接收信号，解除阻塞，延迟时间计数等没有影响；
- 延迟 —— DELAYED，任务被延迟时所处状态。

当系统函数对某一任务进行操作时，任务从一种状态迁移到另一种状态。处于任一状态的任务都可被删除。

上述基本状态也可形成复杂组合，在表 1-4 所示中可以看到上述基本状态的组合。

表 1-4 任务状态的符号表示

| 含 义 | 符 号 表 示 |
| --- | --- |
| 延迟+挂起 | DELAY+S |
| 阻塞+挂起 | PEND+S |
| 阻塞+延迟 | PEND+T |
| 阻塞+延迟+挂起 | PEND+S+T |

习惯上常常以任务状态转换图直观表示（如图 1-6 所示）：

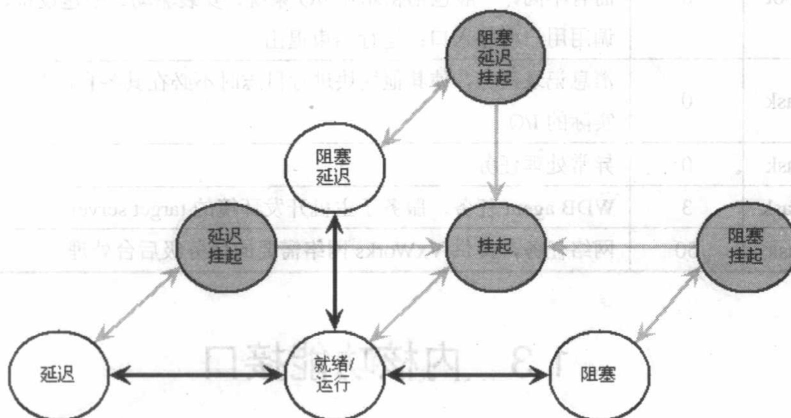



图 1-6 任务状态转换

在图 1-6 中，引起每个黑色箭头转换的典型原因是：

| | |
| --- | --- |
| 运行 → 阻塞 | semTake( ) msgQReceive( ) |
| 阻塞 → 就绪 | 等待的资源变得可用 |
| 运行 → 延迟 | taskDelay( ) |
| 延迟 → 就绪 | 延迟时间结束 |
| 运行 → 阻塞+延迟 | 设置超时调用阻塞函数 |

存在其他能使任务状态转换的情况，尤其是从“运行”到“阻塞”的转换。除了上面给出的两个函数外，还有很多调用都可能使任务阻塞。一般地，对受限系统资源以及需要互斥访问资源的读写都会引起任务阻塞。

图 1-6 中的灰色框都包含“挂起”因素。“挂起”状态可以附加在每个灰色框对应的白色框上。引起的原因都是直接或者间接的调用挂起函数 taskSuspend()。任务一旦被“挂起”，将被调度程序忽略，但是其他处理仍然不受影响，包括延迟将继续根据系统时钟计时；等待的阻塞资源有效时解除阻塞状态。解除“挂起”的原因都是调用函数 taskResume()。

任务被挂起期间如果有信号到来，则信号处理函数在挂起状态解除后调用。

### 1.2.6 系统任务

在 VxWorks 系统启动时，根据定制的组件，系统可能自动启动，如表 1-5 所示。

表 1-5 系统任务

| 任务名称 | 优先级 | 说明 |
|---|---|---|
| tUsrRoot | 0 | 内核启动的第一个任务，又称“根任务”。初始化组件/库，根据用户定制而有不同，一般包括初始化 I/O 系统、安装驱动、创建设备、设置网络等；调用用户程序入口；运行结束退出 |
| tLogTask | 0 | 消息记录任务，使其他模块进行日志时不必在其各自的任务上下文中进行实际的 I/O |
| tExcTask | 0 | 异常处理任务 |
| tWdbTask | 3 | WDB agent 任务，服务于主机开发环境的 target server |
| tNetTask | 50 | 网络任务，提供 VxWorks 网络需要的任务级后台处理 |

## 1.3 内核功能接口

VxWorks 内核功能接口包含在多个库中：

- taskLib VxWorks 任务管理：任务创建、删除、运行状态控制等；
- kernelLib VxWorks 内核：内核初始化，系统调度策略选择；
- semLib VxWorks 信号量：提供二进制，互斥，计数信号量的管理；
- tickLib VxWorks 系统时钟管理；
- wdLib VxWorks 看门狗管理：看门狗的创建、启动、撤销和删除等。

在程序中使用最多的是任务管理 taskLib 和信号量管理 semLib。

### 1.3.1 激活内核

激活内核意味着目标系统从初始单任务状态开始，完成数据结构和各种硬件初始化，开放中断，进入多任务环境。内核激活函数为 kernelInit( )，一般由 usrInit( )调用：

```
#include "kernelLib.h"
void kernelInit (
    FUNCPTR  rootRtn,         /* 根任务入口 */
```

```
    unsigned rootMemSize,     /* 根任务栈大小 */
    char *   pMemPoolStart,   /* 系统堆起始地址 */
    char *   pMemPoolEnd,     /* 系统堆结束地址 */
    unsigned intStackSize,    /* 中断栈大小 */
    int      lockOutLevel     /* 中断锁级别 */
);
```

内核激活前还要进行一系列初始化过程，将在第12章介绍从系统上电开始到根任务运行的完整初始化过程。具体来讲，kernelInit()的工作包括：

- 微内核初始化：kernelInit()最关键的功能。VxWorks 是微内核结构的操作系统，在 kernelInit()之后，系统实现了一个非常简单的微内核，具有1.1.2节讲的微内核的功能；
- 系统堆初始化：系统堆用于实现任务栈，以及 malloc()动态内存分配。参数 pMemPoolStart 和 pMemPoolEnd 指定了用于系统堆的内存范围；
- 中断栈设置；
- 中断锁级别：设置 lockOutLevel；
- 启动根任务：kernelInit()之前没有任何任务运行，该函数激活内核之后第一个运行的任务由参数 rootRtn 指定，一般为 usrRoot()，也称根任务。根任务将进一步初始化系统，使 VxWorks 具有管道、串口、文件系统等许多其他特性（取决于所选组件），但是这些设备、驱动、库等都不属于微内核，和一般的任务相似。

## 1.3.2 任务创建

### 1. taskSpawn( )

任务创建由函数 taskSpawn() 实现，实际上分两步：（1）分配任务栈空间，初始化 WIND_TCB；（2）激活任务。

taskSpawn ()定义为：

```
#include "taskLib.h"
int taskSpawn (
    char *  name,          /* 任务名称（放在任务栈起始处） */
    int     priority,      /* 任务优先级 */
    int     options,       /* 任务选项 */
    int     stackSize,     /* 任务栈大小（不含任务名称） */
    FUNCPTR entryPt,       /* 任务入口程序*/
    int arg1, int arg2, int arg3, int arg4, int arg5,  /* 10个整型参数 */
```

```
    int arg6, int arg7, int arg8, int arg9, int arg10
);
```

成功时函数返回生成任务的任务 ID；否则返回 ERROR。

参数 name 表示任务名称，可以为任意长度，但是必须惟一。当指定 NULL 时，函数自动指定为 tN，N=1，2，3，……

参数 priority 表示任务优先级，VxWorks 允许 256 级优先级（0~256），数字越小表示优先级越高。

参数 options 任务选项，可以是下列标志的组合：

- **VX_FP_TASK**　标志浮点寄存器是否属于上下文。如果任务中某个函数使用了浮点寄存器（采用浮点参数，定义了浮点变量，或者有浮点运算）则任务创建时必须带该选项，告诉调度程序将浮点寄存器环境一起保存；
- **VX_PRIVATE_ENV**　支持任务私有环境变量；
- **VX_NO_STACK_FILL**　不进行初始任务栈填充；
- **VX_UNBREAKABLE**　任务不允许断点调试（忽略断点）；
- **VX_DSP_TASK**　（VxWorks5.5 新增选项）DSP 协处理器支持；
- **VX_ALTIVEC_TASK**　（VxWorks5.5 新增选项）ALTIVEC 协处理器支持。

参数 stackSize 表示任务栈大小。taskSpawn( )在任务栈空间分配和初始化任务控制块，taskSpawn( )自动为此申请额外的空间，所以申请的任务栈大小会大于 stackSize。

函数指针 entryPt 表示任务入口程序，相当于主机操作系统环境下编程的主程序"main"。VxWorks 要求任务入口程序名称必须惟一，并定义为"main"以外的某个字符串，因为 GNU 编译器对"main"将生成一堆额外初始化代码，这些代码对不采用类似主机操作系统进程结构的 VxWorks 来说是多余的。

taskSpawn( )为任务入口程序传递 10 个整型参数 arg1~arg10，如果任务入口程序接受参数个数小于 10 个，可以将后面的参数传递 0（在 VxWorks 5.0 以前，可以将任务入口程序不需要的参数省略）。事实上，如果任务入口程序需要其他类型的入参，如字符串指针，可以进行强制转换。

taskSpawn ( )调用示例：

```
int tMyTaskId = taskSpawn ("tMyTask", 100, VX_FP_TASK, 20000, myFunc,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
```

usrLib 库另外提供一个按照默认值新建任务的简便方法，即函数 sp( )。本书给出的完整示例都通过在 shell 下执行该函数来运行。

```
#include "usrLib.h"
int sp ( FUNCPTR func,                                  /* 任务入口程序 */
    int arg1, int arg2, int arg3, int arg4,             /* 9 个整型参数 */
    int arg5, int arg6, int arg7, int arg8, int arg9
```

```
);
```

成功时函数返回生成任务的任务 ID，并输出；否则返回 ERROR。函数采用的默认值设置如下：

- 优先级：100；
- 任务栈大小：根据系统可用内存大小确定；
- 任务选项：VX_FP_TASK；
- 任务名称：tN，N=1，2，3，……

下面在 shell 下通过 sp( )生成任务。sp 创建的任务主体函数为 logMsg( )，并为该函数传递参数为字符串"say hello to logMsg"。logMsg ( )是 VxWorks 库函数，该函数在终端输出通过入参传递给它的字符串。

```
-> sp logMsg, "say hello to logMsg"
task spawned: id = 0xf7f220, name = t25
value = 16249376 = 0xf7f220
0xf7f220 (t25): say hello to logMsg
```

"→"是 shell 提示符，后面给出的命令相当于在程序中调用"sp(logMsg, "say hello to logMsg");"。我们看到 sp 生成的新任务 ID 为 0xf7f220，任务名称为"t25"，最后一行是新任务的输出。

## 2．taskInit( )/taskActivate( )

taskSpawn( )内部调用 taskInit( )/taskActivate( )来完成，后者提供了更细致的任务控制。taskInit( )也能创建任务，taskInit( )和 taskSpawn( )的差别在于：

（1）taskSpawn( )自动在系统堆上分配任务栈空间，并在任务栈上创建 TCB；而 taskInit( )由调用者分配 TCB 和任务栈空间，TCB 不属于任务栈空间。

（2）taskSpawn( )生成任务后进入就绪队列；而 taskInit( )生成任务后处于挂起状态，不能被调度，直到被 taskActivate( )激活。

由于上面提到的第 1 个差异，taskInit( )初始化的任务在删除时不会自动释放任务栈和 TCB 空间，必须由程序显式调用 free( )释放；而 taskSpawn( )生成的任务被删除时自动释放任务栈（含 TCB 块）空间。这符合"谁申请，谁释放"的程序设计原则。

taskInit( )定义为：

```
#include "taskLib.h"
STATUS taskInit (
    WIND_TCB * pTcb,          /* 任务控制块 WIND_TCB 地址 */
    char *     name,          /* 任务名称（放在任务栈起始处）*/
    int        priority,      /* 任务优先级 */
```

```
    int         options,        /* 任务选项 */
    char *      pStackBase,     /* 任务栈起始地址 */
    int         stackSize,      /* 任务栈大小 */
    FUNCPTR     entryPt,        /* 任务入口程序指针 */
    int arg1, int arg2, int arg3, int arg4, int arg5,
    int arg6, int arg7, int arg8, int arg9, int arg10
);
```

需要注意的是根据处理器体系特点，栈有向上增长和向下增长两种不同的方式。因此，pStackBase给出的任务栈起始地址可能是分配的栈空间的低地址端，也可能是高地址端。有的体系同时支持两种方式（如ARM），则pStackBase要视BSP而定。参考下面这段示例程序，该示例表示了如何使用taskInit( )初始化任务。

```
#define STACK_SIZE  20000
#define NAME_SIZE   STACK_ROUND_UP( 10 ) /* should be enough */
#define TCB_SIZE    STACK_ROUND_UP( sizeof( WIND_TCB ))

...

/* 申请任务栈空间（同时包括任务名称，TCB控制块） */
pMem = (char *)malloc(STACK_SIZE+TCB_SIZE + NAME_SIZE);

/* 根据栈方向计算栈起始地址 */
#if (_STACK_DIR == _STACK_GROWS_UP)
   pStackBase = (char *)( pMem + TCB_SIZE + NAME_SIZE );
#else
   pStackBase = (char *)( pMem + STACK_SIZE + TCB_SIZE ));
#endif

/* 初始化任务 */
taskInit ((WIND_TCB *) (pMem + NAME_SIZE), name, priority, options,
       pStackBase, STACK_SIZE,
       ... /* 其他参数 */
   );

...
```

taskActivate( )定义为：

```
#include "taskLib.h"
STATUS taskActivate ( int tid );     /* tid=(int)pTcb; */
```

其惟一的参数 tid 表示被激活的任务 ID，由 taskInit( )初始化的任务 ID 得到。

可以从 ISR 调用 taskActivate( )激活某个任务，但是 taskInit( )初始化不能在 ISR 中调用。

## 1.3.3　任务控制

### 1．挂起与解除

```
#include "taskLib.h"
STATUS taskSuspend ( int tid );    /* 挂起任务 */
STATUS taskResume ( int tid );     /* 解除任务挂起 */
```

处于挂起状态的任务被调度程序忽略，相当于“静止”下来，以便于调试，直到挂起被解除。当 taskSuspend( )的参数 tid 为 0 时将使调用任务自身挂起。

挂起是任务的一种二进制无记忆状态，任务被挂起以前不会检查以前是否被挂起或解除挂起，因此：

（1）重复挂起某个任务和一次挂起的效果是一样的；

（2）如果挂起和解除挂起由不同的任务完成，必须确保按照正确的顺序进行。

考察下面的代码，任务 t1 挂起之前自身发送消息“I am suspended”到某个队列，另一个任务收到消息后解除 t1 挂起：

```
...
mq_send ( mqPXId, "I am suspended", ... );
taskSuspend( 0 );
...
```

上面的代码存在竞争历险，t1 可能在刚好送出消息后被调出 CPU（taskSuspend( )尚未执行），另一较高优先级任务检索到 t1 发送的消息，处理后解除 t1 挂起：

```
taskResume (id_of_t1 );
```

这样，该解除将不会产生任何效果，而随后 t1 执行 taskSuspend( )后便陷入无限挂起状态。应该小心避免出现此种情况，就本例而言，可以将 t1 的代码改写为：

```
...
taskLock( );
mq_send ( mqPXId, "I am suspended", ... );
taskSuspend( 0 );
taskUnlock( );
...
```

另外要注意的是挂起时不要使系统陷入死锁，这通常需要防止任务在获得了某个互斥访问的系统资源后被挂起，尤其在异步挂起时需要小心防止这种情况。

挂起状态可以附加在延迟状态与阻塞状态上，使任务进入“延迟+挂起”或者“阻塞+挂起”状态。附加挂起状态后，与任务原来的延迟和阻塞相互没有影响。

（1）挂起期间延迟任务仍然计算延时，如果延迟到时，任务便进入只挂起状态；

（2）阻塞任务在挂起期间如果等待条件满足，则解除阻塞，进入只挂起状态；

（3）如果延迟到时或者等待条件出现之前任务被解除挂起，则任务回到原来的延迟/阻塞状态。

参考图 1-6 所示。

任务运行中出现异常情况（非法指令、总线或地址错误、被零清除等）时，VxWorks 异常处理包会将引起异常的任务挂起，保存任务在异常处的状态值。内核和其他任务继续执行。可通过 shell，查看当前任务状态，从而确定被休眠的任务。

```
-> I

NAME       ENTRY      TID     PRI  STATUS      PC       SP       ERRNO   DELAY
--------  ---------  -------  ---  ----------  -------  -------  -------  -----
tExcTask   excTask    fe7018   0    PEND        adc6c    fe6f38     0      0
...
t1         process1   f848e8  100  SUSPEND     77c00    f84848     0      0
```

上面的例子是任务 t1 中执行了一条 free( (void*)-1);语句引起异常被 VxWorks 挂起后的输出。

**taskSuspend( )和 taskResume( )**
**可以从 ISR 中调用。**
**挂起任务是无条件的（不论任务当前状态）。**
**任务中出现异常将使任务被挂起；ISR 出现异常是系统复位。**

## 2. 任务延迟

```
#include "taskLib.h"
STATUS taskDelay ( int ticks );
```

函数 taskDelay( )使调用任务睡眠一段时间，以 tick 为单位，默认情况下 1tick 等于 $\frac{1}{\text{SYS_CLK_RATE}}$ 秒，SYS_CLK_RATE 表示系统时钟溢出速率，在 configAll.h 中定义，通常为 60，此时 1tick≈17ms。VxWorks 允许运行时修改系统时钟溢出速率，因此，如果要实现任务以秒为单位的延迟，可以先通过函数 sysClkRateGet( )系统时钟溢出速率。例如使任务延迟 2 秒，可以：

```
taskDelay ( sysClkRateGet() * 2 );   /* 任务延时 2 秒 */
```

系统将在任务调用 taskDelay(n)之后的第 n 个 tick 到来时结束任务的延迟状态，如果任务没有被挂起，则将任务加入到该优先级就绪任务队列尾部。n 包括该调用所在的 tick，例如刚好在 tick 到来之前调用 taskDelay(1)，则系统马上在随后 tick 中断处理时解除延迟，这样任务实际延迟为 0（忽略函数调用和中断处理开销）。因此，如果要求至少延迟 n 个 tick，则应该调用 taskDelay(n+1)。VxWorks 允许指定延迟 n 等于 0，此时任务没有延迟，但是被重新放到就绪任务队列尾部，相当于让其他同级就绪任务运行。

成功时函数返回 OK，否则 ERROR 表示参数错误或者等待被信号中断。

函数 nanosleep( )提供同样的功能：

```
#include "timers.h"
int nanosleep ( const struct timespec * rqtp, struct timespec * rmtp );
```

该函数属于 POSIX 标准。所要求等待的时间由结构体 timespec 的指针 rqtp 表示。结构体以秒（rqtp.tv_sec）和纳秒（rqtp.tv_nsec）表示时间。和 taskDelay( )不同的是，如果 rmtp 非 NULL，则通过其返回剩余时间（如果等待被信号中断，则通常有返回时间大于 0）。

& **taskDelay( )，nanosleep( )**

不允许从 ISR 中调用。

任务延迟期间如果发生信号，则可能使等待中断，errno=EINTR。细节参考第 4 章“信号”。

## 3. 任务互斥

互斥访问是操作系统中一个经典的理论问题，用于实现对共享资源的一致性访问。VxWorks 提供多种互斥机制：

- “测试一锁” vxTas( )
- 任务锁定 taskLock( )
- 锁中断 intLock( )
- 信号量 semTake( )

“测试一锁”的方法通过处理器提供的原子操作实现互斥访问，函数 vxTas( )定义为：

```
#include "vxLib.h"
BOOL vxTas ( void * lock );
```

如果锁变量 lock 未被设置，则该函数设置 lock 并返回 TRUE；否则返回 FALSE。利用该函数实现互斥的一个示例如下：

```
#define   ENTER_CRITICAL_SECTION(lock)
          while( ! vxTas(lock) );
#define   LEAVE_CRITICAL_SECTION(lock)
          lock=0;
static char lock=0;    /* 全局锁变量 */
...
ENTER_CRITICAL_SECTION(&lock)
...    /* 访问互斥资源 */
LEAVE_CRITICAL_SECTION(&lock)
...
```

信号量方法和“测试一锁”方法相似，都不会使系统增加不必要的延迟。它们的区别在于当访问互斥资源需要较长时间时，信号量的方法更高效。考虑上面的例子，假设任务t1 已经成功进入了临界区，这时直到 t1 调用 LEAVE_CRITICAL_SECTION，另一个要求进入临界区的任务 t2 将不断查询锁变量 lock，因此浪费了 CPU 周期；而信号量方法使条件不满足时调用任务阻塞，从而节约了 CPU 周期。信号量方法将在第 2 章第 2.3 节“信号量”部分进一步介绍。

利用锁中断的方法会增加系统的中断响应延迟，并且在任务锁中断期间进行系统调用，可能使中断重新开放。锁中断可以被 ISR 利用，对一般任务而言，锁中断不是好方法。

下面主要介绍任务锁定 taskLock( )。VxWorks 允许调用 taskLock( )来使调度程序失效（抢占禁止）。当任务调用 taskLock( )后，调度程序暂时失效，即使有高优先级任务就绪，任务也不会被调出处理器，直到任务调用 taskUnlock( )解除锁定。

taskLock( )/ taskUnlock( )在 taskLib.h 中定义：

```
#include "taskLib.h"
STATUS taskLock (void);        /* 锁定任务 */
STATUS taskUnlock (void);      /* 解除锁定 */
```

在实现上，每个 VxWorks 任务 TCB 都维护一个计数器锁变量 lockCnt，taskLock( )使其加1，taskUnlock( )使其减 1；当 lockCnt 大于零时，调度程序便被禁止。因此，需要对 taskLock( )调用相同次 taskUnlock( )才能解除任务锁定，一般 taskLock( )和 taskUnlock( )成对出现，嵌套调用 taskLock( )需要更多小心。

作为一种实现互斥访问方案，taskLock( )为系统引入了优先级延迟（调度延迟）：实时性要求高的高优先级任务必须等到任务解除锁定后才被调度。这样在一定程度上牺牲了VxWorks 优越的实时性能。除非临界区内操作非常简单，或者应用对实时要求不敏感，更多的时候一般考虑采用更一般的信号量或者“测试一锁”的方法。

**taskLock( )/taskUnlock( )**

如果任务设置锁定期间被阻塞或是挂起时，调度程序会从就绪队列中取出最高优先级的任务运行；当任务的阻塞或挂起状态解除后，再次开始运行时，抢占又被禁止。在这种情况下，要仔细考虑是否会破坏对互斥资源的访问。

taskLock( )用于防止任务的切换，对中断处理不起作用，中断发生时 ISR 照常被调用。

中断服务程序中不能调用 taskLock( )/taskUnlock( )。

## 1.3.4 任务结束

任务结束意味着任务生命期终止。任务结束包括运行结束退出、任务调用 exit( )退出和任务删除 3 种情况。任务运行结束实际上也是隐式调用 exit( )实现的。

先来看任务删除，再看任务结束。

### .1. 任务删除：taskDelete( )

```
#include "taskLib.h"
STATUS taskDelete ( int tid );          /* 删除任务 */
STATUS taskDeleteForce ( int tid );   /* 删除任务 强制 */
```

调用函数 taskDelete( )使任务结束，并释放任务资源：任务栈和 TCB 控制块。需要注意系统自动释放的资源仅此两项而已。VxWorks 不会检测被删除任务是否拥有了某种共享资源，如信号量，在这种情况下删除任务导致系统出现不一致，从而运行故障甚至使系统崩溃。因此，任务删除过程中需要非常小心的是保持系统的一致性。

任务安全删除可以通过“删除保护”实现。删除保护定义了任务在何时是可以被安全删除的。需要使用函数 taskSafe( )和 taskUnsafe( )：

```
#include "taskLib.h"
STATUS taskSafe ( void );         /* 定义任务删除保护 */
STATUS taskUnsafe ( void );       /* 解除任务删除保护 */
```

任务通过 taskSafe( )申明任务将要执行一些可能破坏系统一致性的操作，不能被终止；当任务完成这些操作时，任务调用 taskUnsafe( )告诉系统不再需要保护。函数名称 taskSafe 中的“safe”表示“safe from deletion”之意（保护免受删除），而不是“safe for deletion”（删除是安全的），taskUnsafe 中的“unsafe”亦然。

taskDelete( )在删除该任务时将被阻塞，直到任务调用 taskUnsafe( )解除删除保护。

taskDeleteForce( )和 taskDelete( )相似，差别在于 taskDeleteForce( )将忽略被删除任务是否申明了删除保护；taskDeleteForce( )总是立即执行删除，被删除的目标任务结束。

**注意**

删除任务并不自动释放malloc( )分配的内存，也不会关闭打开的文件。这与主机操作系统如UNIX或者Windows环境下的编程有所不同，因此需要程序员注意在任务删除之前手动完成这些操作，防止系统资源溢出。

### 2. 任务结束exit( )

```
#include "taskLib.h"
void exit ( int code );     /* code: 任务结束码 */
```

exit( )是一种优雅的任务结束方式。调用之后使任务结束，指定的退出码保存在任务TCB中。退出码的一个用途就是给钩子程序使用。

## 1.3.5 任务重启

当任务出现问题时，可能采取的一种方法是使任务重新启动，由任务自身或其他任务调用函数taskRestart( )实现，但是不能从ISR中调用。

```
#include "taskLib.h"
STATUS taskRestart ( int tid );      /* 重新启动任务 */
```

taskRestart( )重新启动有以下特点：

- 重启后任务从创建时指定的入口开始执行；
- 支持删除保护，如果taskRestart( )要重启的任务设置了taskSafe( )删除保护，则taskRestart( )被阻塞，直到要被重启的任务调用taskUnsafe( )解除删除保护；
- 沿用重启时的任务栈和TCB，包括任务名称，任务ID，任务选项，优先级，任务栈大小等属性保持不变；
- 重启后全局变量和静态变量保持重启前的值；
- 重启不会自动关闭打开的文件和释放malloc( )分配的内存。

可以看出，任务重启相当于删除任务后重新生成任务，但是删除时不释放任务栈和TCB；而重新生成任务时也不要申请空间。

和任务删除一样，如果任务重启时没有放弃对某个互斥资源的访问权，则其他等待使用该资源的任务将会无限等待。尤其是当通过一个任务重启另一个任务时更要避免这种情况，一般情况下很少这样使用。

如果程序中需要用到任务重启，则编写代码应该考虑这种重入。以文件为例，这种重入性考虑可以以两种方式实现：

（1）重启之前检查，如果有打开的文件，则将其关闭；

（2）每次打开文件之前判断是否在上一轮任务运行时打开了。

实现第 1 种方式比较困难，可以采取第 2 种方式，这时可以把文件描述符都定义成全局变量或者静态局部变量。当程序需要打开文件时，先判断文件是否已经被打开，例如：

```
void some_func ( void )
{
    static int fd = ERROR;
    ...
    if(fd==ERROR)  {  fd = open ( ... );  ... } /* 文件尚未被打开 */
    else ... /* 任务重入，文件已经被打开 */
    ...
}
```

使用任务重启的一个例子是 target shell 因为出现除零异常而重启。

```
-> taskShow 0/0

Exception current instruction address: 0x00078cf4
Machine Status Register: 0x00009032
Data Access Register: 0x80000030
Condition Register: 0x82400040
Data storage interrupt Register: 0x00000163

 84770 vxTaskEntry    +5c : shell ()
 630cc shell         +180: 630f8 ()
 63308 shell         +3bc: execute ()
 63488 execute        +d4 : yyparse ()
 abf08 yyparse        +790: a9ecc ()
 aa03c yystart        +878: taskShow ()
 793a4 taskShow       +44 : taskInfoGet ()
 7923c taskInfoGet    +24 : taskTcb ()
shell restarted.
```

一种在众多情况下比任务重启更可取的方案是使用“环境记录点”。“环境记录点”相当于程序的上下文。和 TCB 的区别是 TCB 供调度程序使用，来选择进行调度的任务；而“环境记录点”供任务使用，用于在任务内执行“goto”语句。

使用“环境记录点”包括调用 setjmp( )设置“环境记录点”和调用 longjmp( )跳到环境记录点（“goto”）。

```
#include "setjmp.h"
int  setjmp ( jmp_buf env );
```

```
void longjmp ( jmp_buf env, int val );
```

一旦任务初始化完成后，就通过 setjmp( )创建一个环境记录点；任何时候，调用 longjmp( )恢复在环境记录点的执行。这时，一个问题是 setjmp( )必须能使程序区分是直接调用还是由于 longjmp( )跳转引起的调用，这一点和 UNIX 下编程时 fork( )必须使进程知道自己是父进程还是被创建的子进程一样。对于直接的 setjmp( )调用，函数返回 0；对于 longjmp( )跳转引起的调用，setjmp( )的返回值由 longjmp( )的第 2 个参数 val 指定，因此区别这两种情况可以通过为 longjmp( )指定非零参数 val。

我们通过下面的例子说明函数 setjmp( )和 longjmp( )的用法：

```
void testjmp( void )
{
    int     i;
    int     gogogo = 0;
    jmp_buf jbRecord;

    printf( "\ntestjmp: a demo of setjmp/longjmp usage.\n\n" );

    for ( i=0; i<5; i++ )
    {
        printf ( "test jmp: local loop, id=%d\n", i );

        /* 循环两次后设置环境记录点 */
        /* This will set the stack back as well if longjmp is called */
        if ( i==2 )
        {
            printf ( "*** call setjmp ***\n" );
            if ( setjmp(jbRecord)!=0 )
            {
                printf ( "test jmp: get re-entered into loop, id=%d\n", i );
                gogogo = 1;
            }
        }
        taskDelay( sysClkRateGet( ));    /* 等候 1 秒 */
    }

    /* 如果 gogogo 为 0 跳到环境记录点 */
    if ( gogogo==0 )
    {
```

```
        printf( "*** call longjmp ***\n" );
        longjmp ( jbRecord, 1 );
    }

    printf( "\ntestjmp done!\n" );
}
```

上面的代码输出如下结果：

```
testjmp: a demo of setjmp/longjmp usage.

test jmp: local loop, id=0
test jmp: local loop, id=1
test jmp: local loop, id=2
*** call setjmp ***
test jmp: local loop, id=3
test jmp: local loop, id=4
*** call longjmp ***
test jmp: get re-entered into loop, id=5

testjmp done!
```

setjmp( )/longjmp( )较多地应用在信号处理函数中。当任务发生错误时，可以通过发送信号给出错任务，出错任务在信号处理函数中通过 setjmp( )/longjmp( )达到重启的效果。不过，从本质上讲，setjmp( )/longjmp( )并没有消除在任务重启中可能带来的不一致，因此，任务执行中仍然需要在敏感区域阻塞信号到达。

> **setjmp( )/longjmp( )**
> C 语言的 goto 不能跨越函数，因此 setjmp( )/longjmp( )实际上使 C 语言有了跨越函数范围进行“goto”的能力。

### 1.3.6 调度控制

VxWorks 支持基于优先级的可抢占式调度和 Round-Robin 调度。默认调度策略为前者。

Round-Robin 调度可以看成对基于优先级的可抢占式调度的一种附加特征，VxWorks 允许对此进行动态选择，即在任一任务中调用 kernelTimeSlice( )来开放或禁止 Round-Robin 调度。

```
#include "kernelLib.h"
STATUS kernelTimeSlice ( int ticks );
```

参数 ticks 指定以 tick 为单位的 Round-Robin 时间片大小，如果为 0，则相当于 Round-Robin 调度被禁止。

如果在 Round-Robin 调度策略下，任务通过 taskLock( )禁止抢占，系统将不对该任务的时间片进行计数，直到 taskLock( )允许抢占。Round-Robin 时间片未用完就被抢占的任务将优先于其他同级任务被调度。

VxWorks 调度符合 POSIX 1003.1b 定义，我们在第 3 章第 3.5 节“任务调度”中将更详细介绍任务调度。

### 1.3.7 其他辅助函数

库 taskLib 还提供了下列任务操作辅助函数：

```
#include "taskLib.h"
STATUS taskPrioritySet ( int tid, int newPriority );  /* 设置任务优先级 */
STATUS taskPriorityGet ( int tid, int * pPriority );  /* 获取任务优先级 */
STATUS  askOptionsSet (int tid, int mask, int newOptions); /*设置任务选项*/
STATUS   taskOptionsGet (int tid, int *pOptions);      /*读取任务选项*/
WIND_TCB * taskTcb ( int tid );      /* 根据 ID 得到任务 TCB */
char * taskName ( int tid );         /* 由任务 ID 得到任务名 */
int  * taskNameToId ( char * name ); /* 由任务名得到任务 ID */
int     taskIdSelf ( void );         /* 得到任务自身 ID */
STATUS taskIdVerify ( int tid );     /* 检验某 ID 任务是否存在 */
BOOL     taskIsSuspended (int tid);  /* 检查任务是否被悬置 */
BOOL     taskIsReady (int tid);       /* 检查任务是否准备运行 */
```

## 1.4 多任务与函数重入

“可重入性”是指函数可以被多个任务调用，而不会破坏数据。不管有多少个任务使用，可重入函数总是为每个任务得到预期结果。允许重入的函数称为“可重入函数”，否则为“不可重入函数”。

由于代码重入性问题源于多线程（对 VxWorks 而言是多任务）并行运行，因此又称代码重入性为“多线程安全性”。以大家熟悉的 Microsoft Visual C++程序设计为例，在链接阶段要根据线程特点选择单线程版本的运行期库（如 libc.lib）或者多线程的版本（如 libcmt.lib）。多线程版本中用到的每个函数都是可重入函数。

在 VxWorks 中，可能引起代码重入的场合包括：（1）多任务调度；（2）中断服务程序调度；（3）信号处理函数调度。对于后面两种情况，在设计程序时还需要考虑除了代码重

入之外的其他问题，这将在后面相关章节涉及。

考虑下面两个函数：

```
char *ctime(const time_t *clock)
{
    static char s[SIZE];
    ...        /*将字符串放入 s*/
    return(s);
}
```

```
char *ctime_r(const time_t *clock,
     char *s, int buflen)
{
    ...        /*将字符串放入 s*/
    return(s);
}
```

两个函数都将参数 clock 表示的时间转换成字符串返回。ctime( )定义字符串为局部静态缓冲区，考虑多个任务“同时”调用 ctime( )，很显然，对于不同任务的调用都将使 time( )修改同一字符串缓冲区，因此 ctime( )是不可重入函数。相比之下，ctime_r( )的字符串缓冲区由调用者分配，不同任务运行修改其各自的缓冲区，因此对于多线程调用是安全的。

上面的例子同时给出了第 1 种实现可重入函数的方法，即使用局部动态变量。因为此类变量在当前任务栈上分配空间，因此函数的不同运行实例引用的是不同的地址。

然而，有些函数不可避免地要使用全局的或者静态的变量，如 malloc( )，该函数从全局的系统堆上分配内存，需要访问很多全局数据结构。在这种情况下，可以采取第 2 种实现可重入函数的方法，即通过互斥信号量保护对全局数据结构的访问。

```
void *malloc( size_t nBytes )
{
    ...
    semTake( heapMutex, ... );
    ...                          /*访问全局数据结构*/
    semGive( heapMutex );
    ...
}
```

互斥锁保证任意时刻只有一个任务在访问全局数据结构，因此，malloc（）也实现了可重入。

仔细比较可以发现，上面两种实现可重入的方法存在着差异。后面在介绍中断和信号时我们将看到，上述第 2 种方法限制只有一个实例运行在函数的互斥访问部分，使得此类函数不能用于中断服务程序和信号处理程序。而第 1 种方法允许在函数任意点存在任意个运行实例，因而不存在这种限制。

我们称允许在任意点存在多个运行实例的函数为**第一类可重入函数**（完全可重入）；除此以外的其他可重入的函数称为**第二类可重入函数**（部分可重入）。第一类可重入函数可以用于上述三种能引起重入的场合，而第二类可重入函数只能在第一种场合下使用。

**可重入函数**

如果一个 VxWorks 函数有 name( )和 name_r( )两种命名，则 name( )是不可重入的，而 name_r ( )是可重入的。通常前者属于 ANSI C 标准（ANSI C 没有考虑多线程定义），后者属于 POSIX 标准。

其他函数是否属于可重入函数没有定义，需要具体分析。一般情况下，如果函数需要读写全局或者静态数据结构，则函数只能为第二类可重入函数或者不可重入函数。

## 任务变量

除了上面两种实现可重入函数的一般方法外，VxWorks 还提供一种独特的“任务变量”机制来实现函数重入。

任务变量：就是任务上下文中记录的一个 4 字节数值，可以将其看成指向一个全局变量的指针。每次任务被调入处理器时，系统根据该指针自动从任务上下文装入全局变量的值；相应地，任务被调出处理器时，系统根据该指针自动将全局变量的值保存到任务上下文。

任务变量实现可重入的过程可以从下面的例子中得到说明：

```
BUF_DEF *pGlobalBuf;  /* 指向缓冲区的全局指针 */

void taskMain ( )
{
    if ( taskVarAdd (0, (int *)&pGlobalBuf ) != OK )   /* 声明任务变量 */
        ... /* 失败处理 */

    /* 分配缓冲区 */
    if ( (pGlobalBuf = (BUF_DEF *) malloc (sizeof (BUF_DEF))) == NULL )
        ... /* 失败处理 */
    ...
}
```

显然，对于每次根据 taskMain( )生成的新任务运行，所使用的缓冲区本身都是 malloc( )从系统堆上新分配的，是相互独立的，要求指向该缓冲区的指针 pGlobalBuf 也是独立的。如果不声明任务变量，则每次根据 taskMain( )生成新任务时 pGlobalBuf 就会被修改。声明了任务变量后，系统将在上下文切换时通过 TCB 来保存和装入各自任务拥有的全局变量 pGlobalBuf 的副本，如图 1-7 所示。

声明任务变量后，上述过程对各个任务而言是透明的。除了开始时调用 taskVarAdd( )声明任务变量外，函数在其他地方访问全局变量感受不到任何差异。

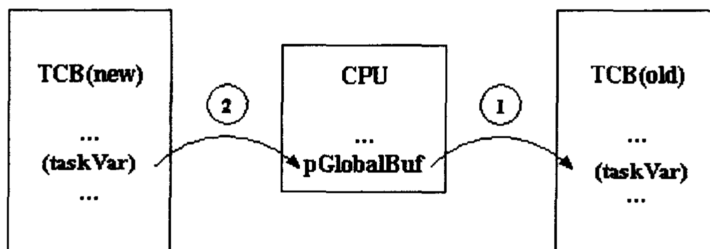



图 1-7 任务变量

**注意**

任务变量必须在对其进行任何赋值之前声明。

试分析 taskMain( )中的两条 if 语句交换秩序后运行的结果。另外，对于图 1-7 中对任务变量的保存和恢复过程，通过任务调度钩子和任务删除钩子实现，因此需要任务钩子库 taskHookLib 和任务变量库 taskVarLib（宏定义为 INCLUDE_TASK_VARS）的支持。taskVarLib 库提供任务变量的增加、删除、读取、设置的功能。

```
#include "taskVarLib.h"
STATUS taskVarAdd (int tid, int *pVar);          /*定义新任务变量*/
STATUS taskVarDelete (int tid, int *pVar);       /*删除任务变量*/
STATUS taskVarSet (int tid, int *pVar, int value); /*设置任务变量值*/
int taskVarGet (int tid, int *pVar);                 /*取任务变量的值*/
int taskVarInfo (int tid, TASK_VAR varList [],int maxVars);/*任务变量列表*/
```

函数 taskVarSet( )和 taskVarGet( )通常用于使参与某项工作的多个协作任务可以相互得到其他任务的任务变量当前值，具有任务间通信的特点。一个简化的例子是，有 n 个搜索任务 tSearch[1~n]协作进行迷宫算法求解，另外有一个分配任务 tDispatch 负责划分任务，tSearch[1~n]具有相同的主函数。任务分配数据和搜索状态定义全局变量并且各个搜索任务将其声明为各自的任务变量。tDispatch 通过 taskVarGet( )取得各个搜索任务的状态，当其完成时，tDispatch 通过 taskVarSet( )为其分配新任务。

从图 1-7 中可以看出，每增加一个任务变量，将使上下文切换增加 4 字节的内存复制开销，另外，调用钩子函数本身需要更多开销。这是任务变量的不足。相比之下，前文的基于局部动态变量的方法也存在局限性，在需要跨越函数范围时，必须使用全局变量；而在各种操作系统编程中应用都很普遍的“全局变量+互斥”的方法比任务变量的方法要繁琐一些。互斥也需要一定的运行开销。

以任务变量实现的可重入函数属于第二类可重入函数。

& 任务出错状态 errno 和任务变量有些类似，但是被固定当作每个任务上下文，因而保存和恢复没有钩子函数开销。

# 第2章　任务间通信

## 2.1 概　　述

多个任务间无可避免的存在相互协同的关系，来完成一定的系统功能。这种协同最直观的看法就是任务间互通信息，即 IPC。IPC 是操作系统经典理论之一。IPC 需要解决两个基本问题：互斥和同步。这是源于多个协同工作的多个任务之间存在竞争与合作这一永恒的主题。

“竞争”意味着多个任务对独占使用资源的争夺使用。在不对其加以控制时，这种竞争可能是没有任务能使用独占资源得到预期的结果。“同步”则意味着多个任务在一些协同点（同步点）上需要相互等待与互通消息，共同参与任务并知道其他任务工作的进展，或者使用其他任务的成果。任务间的这种竞争与合作和人类社会中的竞争与合作是类似的。

VxWorks 提供的 IPC 和 Linux 操作系统 IPC 相似，如图 2-1 所示。

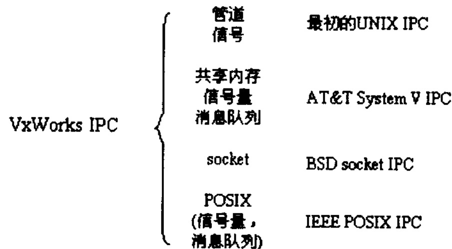



图 2-1　VxWorks 支持的 IPC

各种 IPC 的侧重点如表 2-1 所示。

表 2-1　各种 IPC 侧重点

| 应用目的 | 合适的 IPC 手段 |
|---|---|
| 数据的共享 | 共享内存 |
| 基本的互斥和任务同步 | 信号量 |
| 单 CPU 的消息传送 | 消息队列和管道 |
| 异常处理 | 信号 |
| 同一目标系统不同处理器任务通信 | 共享内存对象（共享内存信号量，共享内存消息队列） |
| 网络间任务消息传送 | Socket |

当考虑可移植性时，应该优先选用 POSIX IPC。

# 2.2 共享内存

“共享内存”就是不同的任务都可以访问同一地址空间。因为 VxWorks 采用单一线性实地址空间，任何任务的数据，包括堆栈上的变量在其生存期内，都可以被其他任务直接访问。因此，共享内存为 VxWorks 任务提供了一种非常简单而且高效的通信机制。

共享内存部分可以采用通信任务双方协商一致的数据结构，从简单的各种类型的变量定义，到采用 VxWorks 提供的双向链表，环行队列等复杂的数据结构。以下是 VxWorks 提供的两个实用程序库，其具体用法可以参考[WRS-oslib5.5]：

| | |
|---|---|
| lstLib | #include "lstLib.h"<br>双向链表管理：创建和删除链表，取出节点，添加节点，统计节点，合并链等 |
| rngLib | #include "rngLib.h"<br>环行（字符）队列管理：创建和删除队列，从队列中取出一个字符，向队列添加一个字符等 |

伴随共享内存而来的一个问题是需要对多个读者任务或者多个写者任务进行互斥访问，通常需要使用信号量来解决。

# 2.3 信号量

## 2.3.1 概述

### 1．竞争条件与互斥

[Tan99]对“竞争条件”（又称竞争历险）作了经典定义：“……两个或多个进程读写某些共享数据，而最后的结果取决于进程运行时的精确时序”。实际上，竞争条件发生在共享内存，共享文件以及其他任何共享资源的场合。而“互斥”即“以某种手段确保当一个进程在使用一个共享变量或文件时，其他进程不能做同样的操作”。其中，最常见的“某种手段”就是信号量。

在 VxWorks 中，多任务运行同样存在竞争条件，差别仅在于 VxWorks 比一般的主机操作系统更方便在任务间共享数据，同时，设计互斥方案时往往更注重对系统实时性的影响。

VxWorks 信号量提供最快速的任务间通信机制，它主要用于解决任务间的互斥和同步。针对不同类型的问题，有以下 3 种基本类别的信号量：

- 二进制信号量。

- 互斥信号量：主要用于优先级继承、安全删除和回溯。
- 计数器。

VxWorks 应用程序还可以使用兼容 POSIX 的信号量，这将在第 3 章介绍。

## 2. 内部数据结构

任何操作系统下信号量都至少包括信号量的值和阻塞任务队列两部分，VxWorks 定义的信号量还包括其他一些字段。

3 种不同类型的 VxWorks 信号量都用结构体 SEMAPHORE 表示。VxWorks 另外定义了一个数据类型为 SEM_ID，也就是指向该结构体的指针。对应用程序而言，只需要通过 SEM_ID 就可以使用信号量，如同信号量就是一个整型的 ID 标识一样。实际上，SEMAPHORE 定义为：

```
typedef struct semaphore {
    OBJ_CORE objCore;     /* 0x00: object management */
    UINT8    semType;     /* 0x04: semaphore type */
    UINT8    options;     /* 0x05: semaphore options */
    UINT16   recurse;     /* 0x06: semaphore recursive take count */
    Q_HEAD   qHead;       /* 0x08: blocked task queue head */
    union {
        UINT    count;              /* 0x18: current state */
        struct windTcb *owner;     /* 0x18: current state */
    } state;
    EVENTS_RSRC  events;  /* 0x1c: VxWorks events */
} SEMAPHORE;
typedef struct semaphore * SEM_ID;
```

可能有多个任务在一个信号量上阻塞，上面的 qHead 表示阻塞任务队列。队列中阻塞任务排列顺序分两种情况：基于优先级和先进先出，由创建信号量时指定的 options 字段值表示的信号量选项决定，如图 2-2 所示。

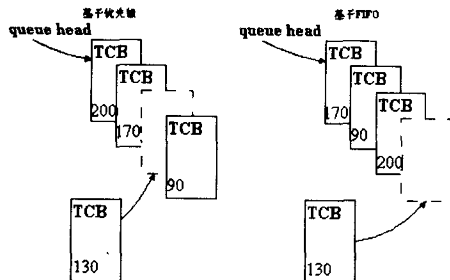



图 2-2　信号量阻塞任务队列

信号量变为有效时系统总选择队列首部的任务解除阻塞。因此队列顺序决定了获取信号量的顺序：按照基于优先级的策略，阻塞队列中具有最高优先级的任务得到信号量；按照先进先出策略，最先阻塞的任务得到信号量。很显然，优先级排序带来了额外的开销。

字段 recurse 和 state（作 windTcb*解时）用于互斥信号量。对于其他类型的信号量，联合体字段 state 表示信号量计数。

字段 events 用于实现信号量事件，表示了接收信号量事件的任务 ID，事件值以及一些控制选项。

### 3. 信号量事件

信号量事件属于 VxWorks 对信号量扩展的内容，即调用 semGive( )时，向注册了信号量事件的任务发送一个事件。信号量事件是 semEvLib 对 eventLib 的扩展，本章后面将要介绍事件这一通信方式。

一个信号量最多支持一个任务注册信号量事件。注册由函数 semEvStart( )和 semEvStop( )完成：

```
#include "semEvLib.h"
/* 注册当条件满足时发送信号量事件 */
STATUS  semEvStart (SEM_ID semId, UINT32 events, UINT8 options);
/* 取消注册 */
STATUS  semEvStop  (SEM_ID semId);
```

**注意**

（1）信号量事件仅在信号量变得有效而且没有任何任务在该信号量上阻塞时才会送出信号量事件。

（2）对于发送了信号量事件后，VxWorks 未保证随后信号量仍然有效。

由于上面的原因，信号量事件实际使用比较少。尤其是尽量避免混合使用 semTake()和信号量事件两种方式。

### 4. 信号量操作

VxWorks 信号量通过多个库提供：

| | | |
|---|---|---|
| semBLib | INCLUDE_SEM_BINARY | 二进制信号量库 |
| semCLib | INCLUDE_SEM_COUNTING | 计数信号量库 |
| semMLib | INCLUDE_SEM_MUTEX | 互斥信号量库 |
| semSmLib | INCLUDE_SM_OBJ | 共享内存信号量库 |

程序可以根据需要进行定制。如果程序不需要使用计数信号量，则可以将其去掉以节省代码空间。二进制信号量和互斥信号量是很多系统组件的基础，通常不能去掉。

使用任何信号量之前需要先创建，不同的信号量创建由各自的库实现，略有差异。下

面先介绍的各种信号量都定义了两种基本操作：TAKE（获取）信号量和 GIVE（释放）信号量，后面几节介绍各种信号量的创建及其特点。TAKE 和 GIVE 在许多经典教科书里用 P 和 V 表示。

获取信号量：

```
#include "semLib.h"
STATUS    semTake (SEM_ID semId, int timeout);
```

如果获取了信号量，函数返回 OK；否则返回 ERROR。参数 timeout 表示以时钟 tick 为单位的等待时间。两个特殊的等待时间是：

- -1：不限制等待时间，直到获取信号量才返回（OK）。定义为宏 WAIT_FOREVER；
- 0：零等待时间，如果不能获取信号量，函数立即返回（ERROR）。

如果在设置的大于 0 的等待时间内不能获取信号量，函数返回 ERROR，设置 errno 为 S_objLib_OBJ_TIMEOUT。

不允许在中断服务程序中调用此函数。

释放信号量：

```
#include "semLib.h"
STATUS   semGive (SEM_ID semId);     /*释放信号量*/
```

对于互斥信号量，要求释放信号量的任务当前拥有该信号量，否则函数返回 ERROR 并设置 errno 为 S_intLib_NOT_ISR_CALLABLE。

对于其他类型的信号量，释放信号量引起的动作需根据不同情况。

- 阻塞队列中有任务：选择一个任务解除阻塞；
- 阻塞队列为空：信号量变为 FULL（对二进制信号量）或者计数加 1（对计数信号量）。如果有任务注册信号量事件，则 semGive( )任务还向注册信号量事件的任务发送指定的事件。注意信号量创建标志 SEM_EVENTSEND_ERR_NOTIFY 对 semGive( )返回结果的影响：如果指定该标志，则 semGive( )发送事件失败后返回 ERROR 并设置 errno= S_eventLib_EVENTSEND_FAILED；否则使发送失败函数返回 OK。

另外，除了互斥信号量，VxWorks 还支持刷新（FLUSH）信号量：

```
#include "semLib.h"
STATUS   semFlush (SEM_ID semId);    /*刷新信号量*/
```

该函数将使信号量的阻塞队列上所有任务解除阻塞，这些任务的 semTake( )调用将返回 OK。注意该调用不会改变信号量状态（二进制信号量的 FULL/EMPTY 状态或者计数信号量的计数值）。

除了互斥信号量，函数 semGive( )和 semFlush( )都允许从中断服务程序中调用，即用

于ISR和任务同步的场合。例如设备驱动程序的ISR将数据写入缓冲区，然后通过这两个调用解锁阻塞的驱动程序任务。

使用任一调用解除某任务阻塞后，如果被解除阻塞的任务优先级更高，将发生优先级抢占。

### 5. 信号量删除

```
#include "semLib.h"
STATUS   semDelete (SEM_ID semId);   /*删除信号量*/
```

当信号量不再使用时可以删除以释放占用的内存。任何任务试图操作一个被删除的任务将会出错（errno=S_objLib_OBJ_ID_ERROR）。因此必须避免删除已经被其他任务得到的信号量。试想一个任务在不知道信号量已经被删除时去释放信号量的情形。一个做法就是要删除信号量的任务自己先获取信号量，然后删除。

如果有任务在该信号量上阻塞，该函数将同时使阻塞队列内的所有任务和等待信号量事件的任务都解除阻塞，相关调用将返回ERROR。

### 6. 一致性

和信号量有关的一致性问题主要包括：（1）删除信号量时不会使阻塞其上的任务陷入无限等待；（2）持有信号量的任务不会因为意外停止运行而使其他任务陷入无限等待。一致性问题可能会使部分任务无法运行，严重时使整个系统得不到期望的结果甚至崩溃。其中，第（1）点由系统保证，系统在删除信号量时自动解除信号量阻塞队列上所有阻塞的任务。

任务意外停止运行包括任务被删除，异常操作等情况。在这种情况下，任务已经持有的信号量得不到释放。如何保证此时系统的一致性是个比较复杂的问题。

对于二进制信号量和计数信号量，VxWorks的策略是不考虑该问题，因为系统无法了解程序使用信号量的意图。

对于互斥信号量，可以假定任务获取信号量后必定会释放信号量，这样似乎可以让系统自动回收信号量。但是这不能解决所有问题，毕竟任务被意外停止，它也许被期待完成许多重要的动作。因此VxWorks采取的策略是：对互斥信号量定义“任务删除保护”（SEM_DELETE_SAFE），即如果被删除任务持有“任务删除保护”的信号量，则删除延迟到任务释放信号量后进行。对于其他情况，VxWorks的策略依然是予以忽略。互斥信号量是否“任务删除保护”在创建时指定。

### 6. 实时性

前面已经介绍过，优先级调度时可能发生优先级翻转。这时，系统的实时性将无法保证。VxWorks对信号量的一个实时扩展就是实现了对信号量优先级继承协议的支持，不过仅仅限于互斥信号量，对其他信号量没有支持。

## 2.3.2 二进制信号量

二进制信号量是 VxWorks 中最高效，使用最广泛的信号量。主要用于同步或互斥。二进制信号量有满（SEM_FULL）和空（SEM_EMPTY）两种状态。“满”也称“有效”或者“空闲”，“空”也称“无效”。本书中根据上下文环境使用了不同的表述。

二进制信号量由下述函数创建：

```
#include "semLib.h"
SEM_ID semBCreate (
    int          options,         /* 信号量选项 */
    SEM_B_STATE initialState  /* 信号量初始化状态：SEM_FULL 或者 SEM_EMPTY */
);
```

参数 option 设置信号量选项：

- SEM_Q_PRIORITY　表示阻塞任务队列基于优先级排序；
- SEM_Q_FIFO　表示阻塞任务队列采用 FIFO 排序；
- SEM_EVENTSEND_ERR_NOTIFY　发送信号量事件失败后返回结果提示。

参数 initialStat 表示信号量创建时的初始状态：SEM_FULL 和 SEM_EMPTY。

在对二进制信号量执行 TAKE 操作时，如果信号量为满，则任务得到信号量，信号量变为空；如果信号量为空，则任务阻塞，并添加任务 ID 到阻塞队列。

在对二进制信号量执行 GIVE 操作时，如果阻塞队列中有任务，则唤醒一个；否则信号量变为满状态。

二进制信号量特点（比较互斥信号量）：

- 不支持嵌套；
- 不支持优先级继承协议；
- 没有任务删除保护。

因此，在只用到互斥的场合，使用 2.3.3 节中介绍的互斥信号量更可取。

## 2.3.3 互斥信号量

互斥信号量是一种特殊类型的二进制信号量，由下列函数创建：

```
#include "semLib.h"
SEM_ID semMCreate ( int options );     /* options: 互斥信号量选项 */
```

互斥信号量被创建后初始状态总是 FREE。其惟一参数 options 表示信号量选项，除了二进制信号量标志 SEM_Q_PRIORITY，SEM_Q_FIFO 和 SEM_EVENTSEND_ERR

_NOTIFY 以外，互斥信号量还扩展了如下选项：

- SEM_INVERSION_SAFE 任务优先级翻转保护；
- SEM_DELETE_SAFE 任务删除保护。

互斥信号量的一个特色就是对优先级翻转的保护（优先级继承协议）。任务在持有定义了 SEM_INVERSION_SAFE 选项的信号量期间，任务始终以该信号量阻塞队列内所有任务的最高优先级运行：每次有更高优先级的新任务增加到阻塞队列，就将持有信号量的任务优先级提高到新的优先级；任务释放信号量后恢复其自身优先级。

**注意**

和信号量队列内部实现方式有关，VxWorks 要求选项 SEM_INVERSION_SAFE 必须和选项 SEM_Q_PRIORITY 一起使用。

作为维护一致性的一种手段，“任务删除保护”是互斥信号量的另一个特色。定义了选项 SEM_DELETE_SAFE 后，任务在持有信号量期间不会被删除，直到释放信号量。

下面这个例子演示了互斥信号量的使用，注意其中任务删除保护的作用。

```
void taskSub();
SEM_ID semm;
void taskMain ( )   /*主任务：在 shell 下输入 sp taskMain*/
{
    int subId;
    semm = semMCreate ( SEM_DELETE_SAFE );
    /*生成子任务，其优先级为 90，高于主任务的优先级（100）*/
    subId = taskSpawn ( "taskSub", 90, 0, 2000, taskSub );
    printf ( "taskSub spawned\n" );
    /*删除子任务*/
    if ( taskDelete ( subId )!=OK )
        printf ("delete taskSub error: %d\n", errno);
    else
        printf ("taskSub deleted. OK\n");
}

void taskSub( )   /*子任务，由主任务创建 */
{
    semTake(semm, WAIT_FOREVER);   /*获取信号量*/
    printf("taskSub: mutex got and slept\n");
    taskSuspend(0);                /*挂起*/
    printf("taskSub: resumed and release mutex\n");
    semGive(semm);                 /*释放信号量*/
    taskSuspend(0);                /*挂起*/
```

```
}
```

创建互斥信号量时定义了任务删除保护，程序运行时包括两个任务，一个是 shell 下输入命令“sp taskMain”创建的“主任务”，一个是主任务创建的“子任务”。运行时将看到，由于定义了任务删除保护，所以主任务删除子任务时将由子任务第一个 taskSuspend(0)调用而阻塞。

```
-> sp taskMain
task spawned: id = 0xf7bff8, name = t2
value = 16236536 = 0xf7bff8
-> taskSub: mutex got and slept
taskSub spawned                    (taskMain 阻塞在 taskDelete 调用上)
-> i
...
taskSub  taskSub     ffbe80  90 SUSPEND    843d4   ffbe08    0     0
t2       taskMain    f7bff8 100 PEND       83d88   f7bf60    0     0
-> taskResume(0xffbe80)            (解除对 taskSub 的挂起)
...                                (taskMain 恢复执行，删除 taskSub)
taskSub: resumed and release mutex
taskSub deleted. OK
```

即被删除任务持有受保护的互斥信号量时，taskDelete( )将阻塞。当系统非常复杂时，要因此而防止发生死锁。

实际上，上述过程是通过在 semTake( )和 semGive( )时分别隐式地调用 taskSafe( )和 taskUnsafe( )时实现的。所以，“任务删除保护”只能对 taskDelete( )式的删除有保护作用，而对 taskDeleteForce( )式的删除没有效果。

对于任务被删除之外的其他原因引起任务停止运行而导致互斥信号量没有被释放的情况，系统没有予以考虑。但是，semMLib 提供了一个函数作为调试时的一种补救措施：

```
#include "semLib.h"
STATUS   semMGiveForce (SEM_ID semId);
```

在任何任务中被调用该函数的效果和在持有信号量的任务本身调用 semGive( )是一样的。调用该函数之后，原来的任务不再持有信号量，信号量被其他阻塞的任务得到或者变为 FULL 状态。虽然可以在运行时使用该函数，但是对于开发可靠的产品，VxWorks 不鼓励使用该函数，而是从根本上查找引起任务异常停止的原因并解决问题。

互斥信号量的获取和释放必须在同一个任务中完成，因此专门用于解决互斥类问题：

```
funcA ( )
{
```

```
    semTake (semM, WAIT_FOREVER);
    ... /* 临界区操作 */
    semGive (semM);
    /* 其他操作 */
}
```

很显然，上述限制使得中断服务程序无法使用互斥信号量。同样，任何任务调用semFlush( )都是非法的。

在内部实现上，互斥信号量有点像计数信号量。信号量内部有一个字段 recurse 表示累计 TAKE 次数。和另外联合体字段 state 一起，互斥信号量实现了对嵌套调用的支持。信号量被创建时 state 为 0，即不被任何任务拥有。信号量被某任务 TAKE 后，则 state 表示该任务 ID。任务必须用和 semTake 相同次数的 semGive( )才能释放信号量。

嵌套调用在和互斥信号量相关的操作比较多且分布在不同的函数内时是有用的，考虑在上面的函数 funcA( )的临界区操作中调用如下函数 funcB( )：

```
funcB ()
{
    semTake (semM, WAIT_FOREVER);
    ... /* 临界区操作 */
    semGive (semM);
}
```

## 2.3.4 计数信号量

创建计数信号量：

```
#include "semLib.h"
SEM_ID  semCCreate (int options, int initialCount);
```

参数 options 表示计数信号量选项，和二进制信号量选项一样，有 SEM_Q_PRIORITY，SEM_Q_FIFO 和 SEM_EVENTSEND_ERR_NOTIFY。

参数 initialCount 表示信号量的初始值。

在对计数信号量执行 TAKE 操作时，如果信号量大于 0，则任务得到信号量，信号量减 1；如果信号量为 0，则任务阻塞，并添加任务 ID 到阻塞队列。

在对计数信号量执行 GIVE 操作时，如果阻塞队列中有任务，则唤醒一个；否则信号量计数加 1。

和二进制信号量相比，计数信号量只是能够表示更多的状态。因而用来解决“多生产者一消费者”问题。虽然可以用计数信号量实现互斥，但是基本上没有人那样做。二进制

信号量“空”相当于计数信号量为0。

### 2.3.5 共享内存信号量

可选 VxWorks 组件 VxMP 提供共享内存信号量，分共享内存二进制信号量和共享内存计数信号量两种类型，用于：

- 不同处理器任务之间访问共享数据时的互斥，此时信号量数据结构和受保护的变量数据都属于处理器之间的共享内存；
- 不同处理器任务之间的同步。

相比之下，前面介绍的信号量都只能用于单处理器应用中的同步和互斥。

共享内存信号量在多处理器专用的共享内存中分配。这部分空间大小由宏 SM_OBJ_MAX_SEM 定义。而普通信号量在系统堆分配，只受系统空余内存大小限制。

VxMP 实现的共享内存信号量具有如下限制：

- 不能在中断服务程序中使用；
- 不能删除不再使用的共享内存信号量；
- 阻塞队列只能是 FIFO。

共享内存信号量可以由任意处理器的任务创建：

```
#include "semSmLib.h"
/* 创建共享内存二进制信号量 */
SEM_ID  semBSmCreate (int options, SEM_B_STATE initialState);
/* 创建共享内存计数信号量 */
SEM_ID  semCSmCreate (int options, int initialCount);
```

参数含义分别与普通二进制信号量和计数信号量相同。但是 options 不能为 SEM_Q_PRIORITY。

创建后共享内存信号量使用方式和相应的普通信号量一样。

## 2.4 消息队列

### 2.4.1 概述

信号量通信所提供的信息量是非常有限的，通常只是作为解决通信中涉及的互斥和同步问题的底层机制。相比之下，VxWorks 消息队列作为一种更高级的通信方式，能够在同一处理器的各个任务间传递任意长度（理论上只受物理内存和机器字长限制）的信息，如

图 2-3 所示。

图 2-3 中任务 A 可以看成生产者，任务 B 可以看成消费者，表示了“单生产者一单消费者”之间的一种简单单工通信模型。实际应用中往往要更复杂，可能存在多个生产者或多个消费者，但是不会出现某个任务同时既是生产者又是消费者的情况。也就是说，任务和“消息队列”之间是单向的：发送消息到消息队列或者从队列中检索消息。在图 2-3 中，实现任务 A 和任务 B 之间的双工通信，需要增加一个消息队列。

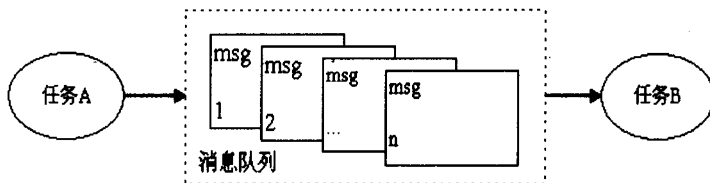



图 2-3 消息队列通信

在应用较多的一种“客户一服务器”模型中，使用多个消息队列维护通信。服务器创建请求队列，并侦听队列上的客户请求；客户将请求放入请求队列并在各自创建的响应队列上取回服务器，结果如图 2-4 所示。

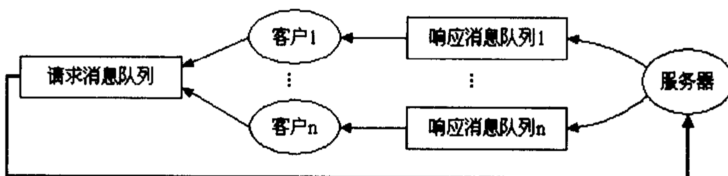



图 2-4 客户服务器消息队列通信

## 1. 为什么使用消息队列

使用“信号量+共享缓冲区”也可以实现图 2-3 中的通信。下面是一个典型的例子，如图 2-5 所示：

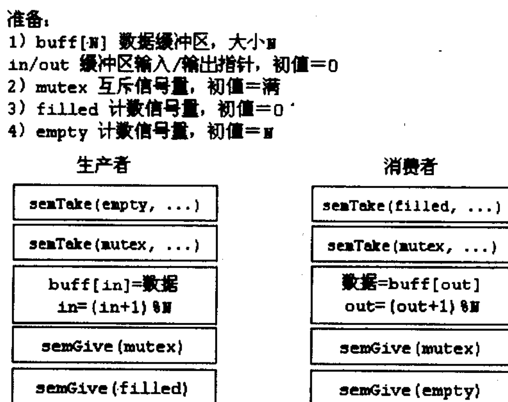



图 2-5 信号量+共享缓冲区

“信号量＋共享缓冲区”不需要在用户缓冲区（即共享缓冲区）和系统内核缓冲区之间复制数据，因此效率很高，适合数据量非常大的场合；而使用消息队列时，通信双方需要经过系统内核缓冲区交换数据，因此效率比“信号量＋共享缓冲区”低。

使用消息队列优势在于：程序简单很多，简单的系统往往意味着减少许多不必要的错误，消息队列和共享缓冲区的比较如图 2-6 所示。

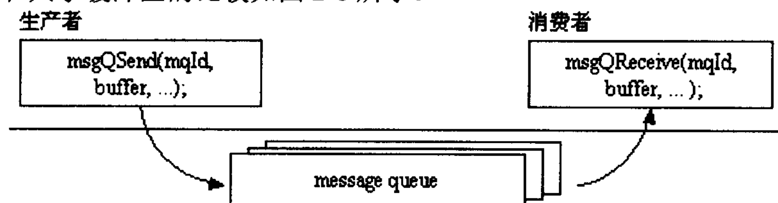



图 2-6　消息队列和共享缓冲区比较

消息队列一个简单的函数调用为应用程序封装了上述生产者和消费者在信号量保护下利用共享缓冲区通信的细节。对于数据量不太大的场合，消息队列带来的程序简化要比效率影响更有诱惑力。许多“客户－服务器”应用属于这种情况。

## 2. 消息队列特点

对于实时系统的消息队列，我们更多地关注其优先级方面的特点。在很多场合下，优先级对响应时间具有重要影响。

信号量的优先级问题只考虑一个阻塞任务队列的优先级排序。和信号量优先级不同，消息队列优先级问题从下面两个方面考虑。

（1）消息自身的优先级：决定消息队列中多条消息提交给接收任务的顺序；

（2）阻塞任务队列优先级：决定阻塞任务队列中多个发送者任务或者多个接收者任务谁先被执行。

消息自身的优先级：VxWorks 以一种很简单的方式实现了两级消息优先级，普通优先级 MSG_PRI_NORMAL 和高优先级 MSG_PRI_URGENT。简单之处在于，两级优先级的差异仅仅体现在链队列（下面将介绍）有空闲节点的情况下，不论链队列中已有多少消息及优先级，标记为 MSG_PRI_URGENT 的消息总会作为首节点插入链队列，这样将在后面被优先收取，而 MSG_PRI_NORMAL 的消息将插入到链队列尾部。考虑很多 MSG_PRI_URGENT 消息到达消息队列的情况，容易得出上述处理方法趋近 LIFO，从而可能使高优先级消息具有不确定的长延迟。消息自身的优先级对消息接收者是透明的，接收者总是接收链队列首部的消息。

阻塞任务队列优先级：优先级决定两个阻塞队列中任务排序的方式。一个消息队列包含两个阻塞任务队列（发送者阻塞队列和接收者阻塞队列），都涉及优先级问题并且对同一个消息队列而言两个优先级是一致的。和信号量阻塞任务队列一样分 FIFO 排序（MSG_Q_FIFO）和基于任务优先级排序（MSG_Q_PRIORITY）。显然，将发送消息的任务放在发送者阻塞队列尾部将使消息最后被处理，同时使发送任务最后被解除阻塞；将接

收消息任务放在接收者阻塞队列尾部将使接收任务最后被解除阻塞。反过来任务放在队列首部可以同样分析。

细心的读者会考虑发送消息的任务优先级和消息自身优先级采取不同组合时的情况，比如低优先级任务发送一个高优先级消息，是降低优先级任务。VxWorks 的做法如表 2-2 所示。

**表 2-2　发送消息处理**

| 前　　提 | 处　　理 |
| --- | --- |
| 有接收者等待消息 | 立即完成 |
| 没有接收者等待消息但是链队列有可用空间 | MSG_PRI_URGENT：将消息插入链队列首部<br>MSG_PRI_NORMAL：将消息插入链队列尾部 |
| 链队列没有可用空间 | MSG_Q_FIFO：将发送任务插入发送者阻塞队列尾部<br>MSG_Q_PRIORITY：根据发送任务优先级将其插入发送者阻塞队列合适位置 |

### 3. 内部数据结构

一条“消息”实际上就是一块地址连续的缓冲区，内容可以是字符串、二进制数据等，由使用消息队列的任务负责解释，系统只关心缓冲区长度。

在 VxWorks 中，消息队列数据类型定义为结构体 MSG_Q。应用程序较少涉及数据结构内部细节，因而通过一个定义为 MSG_Q_ID 的指向该结构体的指针来使用消息队列。

```
typedef struct msg_q
{
    OBJ_CORE      objCore;       /* object management */
    Q_JOB_HEAD    msgQ;          /* message queue head */
    Q_JOB_HEAD    freeQ;         /* free message queue head */
    int           options;       /* message queue options */
    int           maxMsgs;       /* max number of messages in queue */
    int           maxMsgLength;  /* max length of message */
    int           sendTimeouts;  /* number of send timeouts */
    int           recvTimeouts;  /* number of receive timeouts */
} MSG_Q;
typedef struct msg_q *MSG_Q_ID;
```

可以看出，一个消息队列实际上将消息缓冲区组织在两个单链表结构的链队列上：msgQ 和 freeQ。msgQ 和 freeQ 都定义为结构体 Q_JOB_HEAD，包括表示链队列的链表首节点指针和尾节点指针、链队列节点计数、阻塞任务队列以及链队列操作方法类，如图 2-7

所示。

freeQ 链队列上的节点属于空闲节点，msgQ 链队列上的节点属于未读取的消息节点(每节点表示一条消息)。它们同时还分别记录了因接收消息而阻塞的消费者队列（TCB 及接收缓冲区信息）和因发送消息而阻塞的生产者队列（TCB 及发送缓冲区信息)。

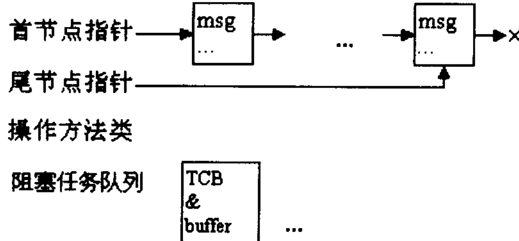



图 2-7　消息队列结构

VxWorks 在创建消息队列时就已经分配了所有 maxMsgs 缓冲区，并组织在 freeQ 上。初始状态下 msgQ 为空。

操作方法类控制链表操作（插入，删除节点)。这种做法源于面向对象的设计思想（对象＋方法)。

不采用循环队列而采用链队列的好处是通过首指针加入高优先级消息时效率很高。缺点在于每个节点增加了额外存储一个指针的开销。

## 4. 消息队列事件

消息队列事件和信号量事件类似。

在发送消息到消息队列时，如满足下列条件：

（1）没有接收者任务在等待消息；

（2）有任务注册了消息队列事件。

VxWorks 将向注册任务发送消息队列事件。

消息队列事件是对 eventLib 的扩展，由库 msgQEvLib 实现。一个消息队列最多支持一个任务注册信号量事件。注册由下述函数完成：

```
#include "msgQEvLib.h"
/* 注册当条件满足时发送信号量事件 */
STATUS   msgQEvStart  (MSG_Q_ID msgQId, UINT32 events, UINT8 options);
/* 取消注册 */
STATUS   msgQEvStop   (MSG_Q_ID msgQId);
```

消息队列事件的发送在调用发送消息函数 msgQSend( )的任务环境中完成。

## 2.4.2 普通消息队列

### 1. 创建消息队列

使用消息队列通信之前，先由任何一方调用 msgQCreate( )创建消息队列。该函数定义为：

```
#include "msgQLib.h"
MSG_Q_ID msgQCreate (int maxMsgs, int maxMsgLength, int options);
```

参数 options 表示消息队列选项：

- MSG_Q_PRIORITY 阻塞任务队列基于优先级排序；
- MSG_Q_FIFO 阻塞任务队列采用 FIFO 排序；
- MSG_EVENTSEND_ERR_NOTIFY 发送消息队列事件失败后返回结果提示。

参数 maxMsgs 和 maxMsgLength 指定消息队列大小：最多消息条数×最大单条消息长度。

调用该函数时即根据消息队列大小（maxMsgs×maxMsgLength）申请消息队列缓冲区。一个要求就是，从提供系统处理效率角度来说，不应该使消息生产者阻塞。因此确定参数 maxMsgs 和 maxMsgLength 使得在不浪费内存的情况下，具有较高的消息处理效率就有一定的艺术。参数 maxMsgLength 的选择容易根据消息特点确定。对于 maxMsgs，一个基于经验的做法是先将 maxMsgs 设置为较小的值，例如 5，然后在每次出现生产者被阻塞时增加设置。一个比较理性的做法是计算消息消费者最大消息处理延迟 maxRecvDelay 和生产者每秒最大产生消息数 maxMsgOutput，然后计算：

maxMsgs=maxRecvDelay×maxMsgOutput

确定 maxMsgs 时还需要考虑生产者和消费者的相对优先级的影响：消费者优先级越高，maxMsgs 越小。

该函数创建消息队列成功时返回 MSG_Q_ID；如果没有足够内存，函数返回 ERROR。消息队列不能在中断服务程序中创建。

### 2. 删除消息队列

消息队列不再使用时可以将其删除以减少内存使用。

```
#include "msgQLib.h"
STATUS msgQDelete (MSG_Q_ID msgQId);
```

该函数将同时使阻塞队列内的所有任务和等待消息队列事件的任务都解除阻塞，相关调用将返回 ERROR（errno= S_objLib_OBJ_DELETED）。

不能在中断服务程序中删除消息队列。

### 3. 发送消息

发送消息的任务即消息“生产者”。发送消息调用函数 msgQSend( )完成：

```
#include "msgQLib.h"
STATUS msgQSend(
   MSG_Q_ID msgQId, char * buffer, UINT nBytes, int timeout, int priority
);
```

该函数把 buffer 中 nBytes 字节数据（即“消息”）发送到消息队列 msgQId 中，nBytes 必须小于消息队列最大消息长度，否则函数会失败。具体执行分 3 种情况（参考表 2-2 所示的“发送消息处理”)：

（1）如果有队列中有接收任务阻塞，则消息直接复制到第一个阻塞的接收者任务的缓冲区（如果接收者缓冲区不够容纳整条消息，则将超出部分丢弃，但不指示错误），同时解除阻塞。

（2）如果没有接收任务等待消息但是链队列有空间，则消息添加到消息队列，函数返回 OK。此时根据参数 priority 表示的消息优先级，若是 MSG_PRI_URGENT 则将消息加入到链队列首部，若是 MSG_PRI_NORMAL 则将消息加入到链队列尾部。

（3）消息队列已满。此时任务将阻塞等待空间，参数 timeout 控制阻塞的最大时间，以 tick 为单位。当时间到时还没有空闲节点则函数返回 ERROR。两个特殊的等待时间是 WAIT_FOREVER（-1）和 NO_WAIT（0）。

在后面两种情况下，如果满足消息队列事件发送条件，函数 msgQSend( )将发送消息队列事件给注册任务。

可以看出，msgQSend( )将消息本身从发送者任务缓冲区复制到消息队列缓冲区。因此，当 msgQSend( )返回时，发送者任务可以安全地删除自己的缓冲区或者改做他用。这里并非理所当然，在第 5 章第 5.4 节“其他 I/O”介绍 logMsg( )时我们将看到，记录消息指针而不是复制消息将带来一些微妙的问题。

函数 msgQSend( )允许从中断服务程序中调用，必须设置参数 timeout 为 NO_WAIT。

### 4. 接收消息

接收消息的任务即消息“消费者”。接收消息调用 msgQReceive( )完成。

```
#include "msgQLib.h"
int msgQReceive(MSG_Q_ID msgQId, char * buffer, UINT maxNBytes, int timeout);
```

该函数从参数 msgQId 所示的消息队列中接收一条消息并复制到参数 buffer 表示的缓冲区，maxNBytes 表示缓冲区大小，maxNBytes 可以大于消息队列最大消息长度。成功时函数返回实际复制的字节数（>0），失败时返回 ERROR（-1）。具体执行情况如下：

（1）如果队列中有消息，则复制消息到buffer，最大maxNBytes。如果消息长度小于maxNBytes，则buffer多余部分未修改；如果消息长度大于maxNBytes，则超出部分丢弃，但不指示错误。

（2）如果队列中没有消息，则任务将阻塞，此时timeout表示以tick为单位的最大阻塞时间。当时间到时还没有消息就绪，则函数返回ERROR，errno=S_objLib_OBJ_TIMEOUT。两个特殊的等待时间是WAIT_FOREVER（-1）和NO_WAIT（0）。任务在接收者队列中阻塞时，根据消息队列创建时指定的优先级排序方式决定任务在阻塞队列中的位置：MSG_Q_FIFO表示将任务插入阻塞队列尾部，MSG_Q_PRIORITY表示根据任务优先级决定任务在阻塞队列中的位置。

函数msgQReceive( )不允许从中断服务程序中调用。

## 2.4.3 共享内存消息队列

与共享内存信号量类似，可选组件VxMP还提供了共享内存消息队列。允许同一目标系统上属于多个处理器的不同任务之间以消息队列的方式通信。

共享内存消息队列和普通消息队列相比，具有以下不同：

- 共享内存消息队列所需内存在多处理器专用的共享内存中分配。这部分空间大小由宏SM_OBJ_MAX_MSG_Q定义。普通消息队列所需内存在系统堆分配，只受系统空余内存大小限制；
- 共享内存消息队列不能在中断服务程序中使用，包括发送和接收；
- 系统没有提供删除共享内存消息队列的方法；
- 共享内存消息队列的阻塞任务队列只能是FIFO。

共享内存消息队列由下列函数创建：

```
#include "msgQSmLib.h"
MSG_Q_ID msgQSmCreate (int maxMsgs, int maxMsgLength, int options);
```

参数含义分别和普通消息队列相同。但是options不能为MSG_Q_PRIORITY。同时，指定maxMsgs和maxMsgLength更需要考虑节约内存。

共享内存消息队列创建后的使用方式和普通消息队列一样。

## 2.4.4 信号量和消息队列实验

下面给出一个实例，来测试发送/接收消息的顺序，同时说明信号量的基本用法。消息队列的大小为3（条）×50（字节）。考虑应用中较多出现“多生产者—单消费者”的情况，我们生成8个发送者任务和1个消费者任务。程序使用的主要数据有：

- 消息队列 msgQueue；
- 互斥信号量 semMutex：用于发送任务和接收任务互斥访问记录缓冲区；
- 计数信号量 semCounter：用于发送任何和接收任务通知主任务运行结束；
- 记录缓冲区 logBuf 及其指针 nLog：用于（1）发送任务发送消息前记录消息内容；（2）发送任务送出消息后记录结果；（3）接收任务记录接收结果。对 logBuf 和 nLog 的访问通过互斥信号量 semMutex 进行保护。

程序运行方式是在 shell 下执行命令 sp msgOrder。

```
/*
 * msgorder.c: test message processing order
 */
#include "vxworks.h"
#include "msgQLib.h"
#include "semLib.h"
#define MAX_MSG 3
#define MAX_MSG_LEN 50
#define TEST_NUM 8

MSG_Q_ID msgQueue = NULL;
SEM_ID   semMutex = NULL;
SEM_ID   semCounter = NULL;
char logBuf[TEST_NUM*3][MAX_MSG_LEN+20];
int  nLog = 0;
void sendMsg( int );
void rcvMsg( void );

void msgOrder( )
{
   /* 定义：发送任务优先级 消息自身优先级 接收任务优先级 */
   const int senderPri[TEST_NUM] = { 50, 51, 52, 49, 53, 56, 55, 54};
   const int msgPri[TEST_NUM]    = { 0, 1, 1, 1, 0, 1, 0, 1 };
   const int rcvPri = 60;
   int i, j;

   if( msgQueue || semMutex || semCounter )
   {
     printf("last running not exit properly!\n");
     exit(-1);
   }
```

```
    /* 创建：消息队列 互斥信号量 计数信号量 */
    msgQueue   = msgQCreate(MAX_MSG, MAX_MSG_LEN, MSG_Q_FIFO);
    semMutex   = semMCreate(SEM_Q_FIFO);
    semCounter = semCCreate(SEM_Q_FIFO, 0);
    if( !msgQueue || !semMutex || !semCounter )  exit(-1);

    /* 生成发送任务（发送消息） */
    for( i=0; i<TEST_NUM; i++ )
        taskSpawn( 0, senderPri[i], 0, 2000, sendMsg, msgPri[i],
               0, 0, 0, 0, 0, 0, 0, 0, 0 );

    /* 生成接收任务（接收消息） */
    taskSpawn( 0, rcvPri, 0, 2000, rcvMsg,
               0, 0, 0, 0, 0, 0, 0, 0, 0, 0 );

    /* 等待所有任务退出 */
    i = 0;
    while( i++ < TEST_NUM+1 )
        semTake(semCounter, WAIT_FOREVER);

    printf("\n*** Logged result:\n");  /* 输出结果 */
    for( i=0; i<nLog; i++ )
        printf("  <%2d> %s\n", i, logBuf[i]);

    msgQDelete(msgQueue);   msgQueue   = 0;  /* 正常退出清理 */
    semDelete(semMutex);    semMutex   = 0;
    semDelete(semCounter);  semCounter= 0;
}

void sendMsg( int msgPri )  /* 发送消息 msgPri 表示消息优先级*/
{
    int taskPri;
    char msgBuf[MAX_MSG_LEN];

    /* 生成消息内容 msgBuf：任务优先级+消息优先级  记录 */
    taskPriorityGet(taskIdSelf(), &taskPri);
    sprintf(msgBuf, "task pri: %d, msg pri: %s",
            taskPri, msgPri==MSG_PRI_NORMAL ? "LO":"HI");
    semTake(semMutex, WAIT_FOREVER);
```

```
        sprintf( logBuf[nLog], "---S: [ %s ] ready", msgBuf);
        nLog++;
    semGive(semMutex);

    /* 发送消息  记录 */
    msgQSend(msgQueue, msgBuf, MAX_MSG_LEN, WAIT_FOREVER, msgPri );
    semTake(semMutex, WAIT_FOREVER);
        sprintf( logBuf[nLog], "---S: [ %s ] sent", msgBuf);
        nLog++;
    semGive(semMutex);

    semGive(semCounter);
}

void rcvMsg( void )   /* 接收消息 */
{
    int i, j;
    char msgBuf[MAX_MSG_LEN];
    for( i=0; i<TEST_NUM; i++ )
    {
        j = msgQReceive( msgQueue, msgBuf, MAX_MSG_LEN, WAIT_FOREVER );
        semTake(semMutex, WAIT_FOREVER);
            if(j)
                sprintf( logBuf[nLog], "---R: [ %s ] received", msgBuf );
            else
                sprintf( logBuf[nLog], "---R: 0 bytes read" );
            nLog++;
        semGive(semMutex);
    }
    semGive(semCounter);
}
```

在程序中，我们设置 8 个发送任务优先级位于 49～56 之间，接收任务优先级为 60。程序运行方式为在 shell 输入“sp msgOrder”命令。这样，sp 为主任务设置的优先级为 100。数字越小代表优先级越高。程序运行得到下面的结果，从中可以看出发送任务运行的顺序，送出消息的顺序，接收消息的顺序。

```
*** Logged result:
 < 0> ---S: [ task pri: 50, msg pri: LO ] ready
```

```
< 1> ---S: [ task pri: 50, msg pri: LO ] sent
< 2> ---S: [ task pri: 51, msg pri: HI ] ready
< 3> ---S: [ task pri: 51, msg pri: HI ] sent
< 4> ---S: [ task pri: 52, msg pri: HI ] ready
< 5> ---S: [ task pri: 52, msg pri: HI ] sent
< 6> ---S: [ task pri: 49, msg pri: HI ] ready
< 7> ---S: [ task pri: 53, msg pri: LO ] ready
< 8> ---S: [ task pri: 56, msg pri: HI ] ready
< 9> ---S: [ task pri: 55, msg pri: LO ] ready
<10> ---S: [ task pri: 54, msg pri: HI ] ready
<11> ---S: [ task pri: 49, msg pri: HI ] sent
<12> ---R: [ task pri: 52, msg pri: HI ] received
<13> ---S: [ task pri: 53, msg pri: LO ] sent
<14> ---R: [ task pri: 49, msg pri: HI ] received
<15> ---S: [ task pri: 56, msg pri: HI ] sent
<16> ---R: [ task pri: 51, msg pri: HI ] received
<17> ---S: [ task pri: 55, msg pri: LO ] sent
<18> ---R: [ task pri: 56, msg pri: HI ] received
<19> ---S: [ task pri: 54, msg pri: HI ] sent
<20> ---R: [ task pri: 50, msg pri: LO ] received
<21> ---R: [ task pri: 54, msg pri: HI ] received
<22> ---R: [ task pri: 53, msg pri: LO ] received
<23> ---R: [ task pri: 55, msg pri: LO ] received
```

在上面的程序中，一条消息内容由“(任务优先级，消息自身优先级)”组成。我们将消息队列和阻塞任务队列抽象为“(…，…)”的形式，得到上述程序运行过程（如图 2-8 所示）。

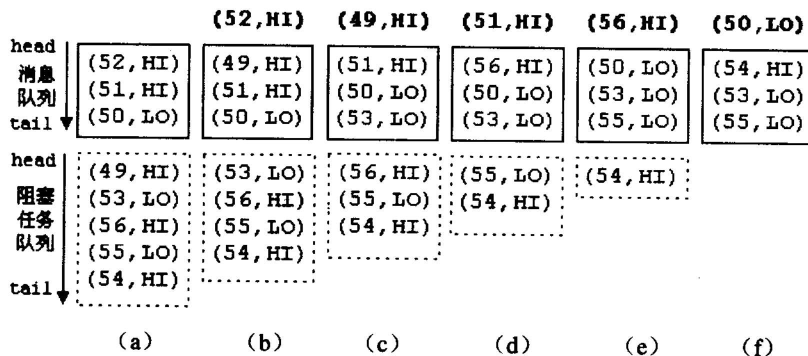



图 2-8　消息处理顺序（FIFO 任务队列）

实线框表示消息队列，虚线框表示阻塞任务队列。（a）图为初始状态，发送任务要么已经将消息复制到消息队列，要么阻塞在任务队列中。（b）~（f）为每次从队列接收一条消息后的结果，图上方黑体部分表示本次读取的消息。从（f）开始，已经没有发送任务阻塞，后面的过程将读出消息队列中最后 3 条消息，简单地从队列首读到队列尾。为节约篇幅，略去了（a）的形成过程和（f）之后的部分。

分析至少有以下结论：

（1）接收消息时总是取消息队列首部的消息，即使阻塞任务队列有更高优先级消息如图（c）。

（2）消息队列中高优先级消息总是排在普通优先级消息前。而同为高优先级的消息之间类似 LIFO；同为低优先级的消息之间类似 FIFO。

（3）阻塞任务队列中完全为 FIFO。任务优先级和消息自身优先级都不起作用。

（4）优先级抢占调度。由于发送任务优先级高于接收任务优先级，当接收任务取走一条消息同时，一个发送任务被解除阻塞并抢占接收任务进入运行状态。例如上面程序输出结果的第 11~第 12 条记录。

如果创建消息队列时指定阻塞任务队列为基于优先级的队列：

```
MsgQueue = msgQCreate(MAX_MSG, MAX_MSG_LEN, MSG_Q_PRIORITY);
```

可以运行程序得到图 2-9 中的结果。阻塞任务队列中根据任务优先级排序，消息自身优先级被忽略。其他结论和采用 FIFO 阻塞任务队列时的结论一样。

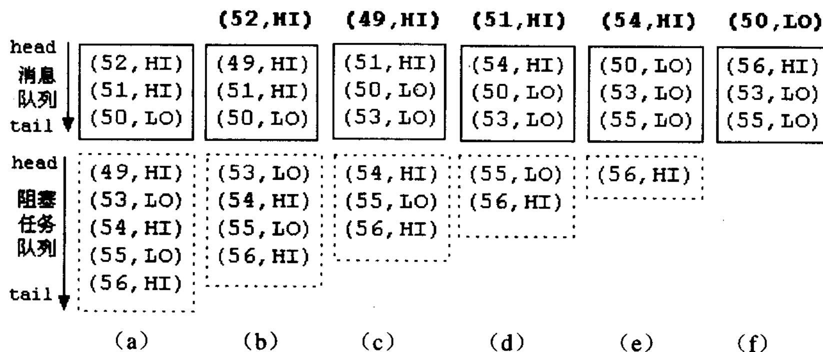



图 2-9 消息处理顺序（优先级任务队列）

# 2.5 管 道

## 2.5.1 概述

管道是 UNIX 操作系统中传统的进程通信技术，分无名管道和命名管道，以 I/O 系统

调用方式进行读写。前者限于父子进程和兄弟进程通信。具有半双工，先进先出等特点。

VxWorks 管道既是对传统 UNIX 命名管道的继承，同时又具有一定差异。UNIX 管道通过在内存中模拟文件系统实现 IPC，当存在多个生产者/消费者时，并不保证操作的原子性，用于两个进程之间的半双工通信。但是 VxWorks 管道其实是通过管道驱动 pipeDrv 对消息队列采用 I/O 系统的方式进行简单封装，虚拟成 I/O 设备，如图 2-10 所示。VxWorks 保证管道 I/O 的原子性，因此在前面介绍过的多客户－单服务器应用中也可以使用。显见 VxWorks 管道承载的信息量受内部消息队列大小限制。

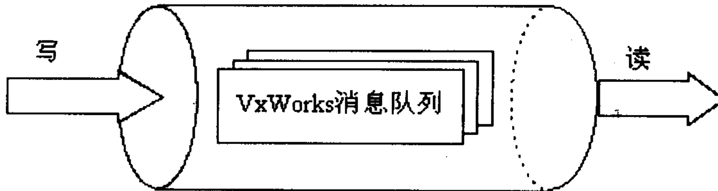



图 2-10　管道 IPC

和直接的 VxWorks 消息队列相比，VxWorks 管道具有如下一致的地方：

- 当任务试图从一个空的管道中读取数据，或向一个满的管道中写入数据时，任务被阻塞；
- 支持中断服务程序向管道中写入信息，驱动程序判断调用环境属于中断则不阻塞，即使没有可用的缓冲区，但是不能从管道中读取，如同中断服务程序不能对消息队列做接收操作一样；
- 往管道中写入数据时，一次写入的数据不能超出消息队列最大允许的消息长度(这一点是和传统 UNIX 管道不同的)。

由于管道具有上述消息队列的本质内容，本书有时称管道中的数据为消息。

另外，VxWorks 管道相比消息队列有如下不足：

- 管道的效率不如消息队列的高。需要将应用程序中标准的 I/O 调用映射到 pipeDrv 驱动程序函数，并最终由消息队列函数实现；
- 管道会稍微多占用一些内存资源和 I/O 系统资源；
- 不支持消息的优先级控制，消息队列支持两级消息优先级和阻塞任务队列优先级；
- 管道不支持超时控制；
- 较老版本可能不支持管道删除，消息队列不再使用时可以删除以释放资源。

尽管如此，管道依然有其价值，有时候用管道更简单更灵活，一个最明显的优势就是管道支持 select( )，这样任务可以同时等待包括管道在内的一系列 I/O 设备上的数据。此外，pipeDrv 提供的几个 I/O 控制命令也比较有用。

使用管道除了需要管道驱动 pipeDrv（定义为宏 INCLUDE_PIPES）支持外，还需要用到 I/O 系统库 ioLib/iosLib（定义为宏 INCLUDE_IO_SYSTEM）的支持，其他库支持视应用需要，如 selectLib。使用这些库必须初始化自动在加入相应组件后自动调用。

## 2.5.2 使用管道

就应用程序来看，其访问方式和UNIX管道相似。任务通过调用标准的I/O函数打开、读出、写入管道。大致包括这样几个步骤：（1）创建管道设备；（2）管道文件操作，包括打开管道文件，读写，关闭管道文件；（3）删除设备释放资源。此外，管道驱动还支持几个I/O控制命令。本节不可避免涉及了许多I/O系统相关内容，读者有疑问可以参考第3章输入/输出系统。

### 1. 创建管道设备

使用管道之前必须创建管道设备：

```
#include "pipeDrv.h"
STATUS pipeDevCreate ( char * name, int nMessages, int nBytes );
```

参数name表示创建的管道设备名称。该设备名称必须在所有I/O设备名称中惟一。后面两个参数指定管道容量：nMessages表示管道中最大消息条数，nBytes表示每条消息最大字节长度。

### 2. 管道文件操作

管道IPC采取I/O操作形式，需要打开管道文件进行。函数open打开一个管道文件：

```
#include "ioLib.h"
int open (const char *name, int flags, int mode );
```

参数name需要和管道设备名称一致，flags表示3种打开的读写方式之一：只读（O_RDONLY），只写（O_WRONLY），读写（O_RDWR）。对于管道文件，参数mode指定为0。

成功时，函数返回一个非负整型数标识得到的管道的文件描述符；失败时返回ERROR。

对于使用某管道进行IPC的多个任务，可以在各自任务中分别以合适的读写方式打开管道文件，也可以由任何一个任务打开管道文件然后被其他任务共同使用。两种方式的结果是一样的。即使在前一种情况下，任务通过各自得到的管道文件描述符进行读写都是对同一“管道设备”进行，任务不应该假定由其独自读（或写）管道。2.5.3节给出的例子属于后面一种情况。

通过前面的open( )得到的管道文件描述符，就可以通过下面的基本I/O调用实现管道IPC了。

```
#include "ioLib.h"
```

```
int  read (int fd, char *buffer, size_t maxbytes);  /* 从管道读数据 */
int  write (int fd, char *buffer, size_t nbytes);   /* 数据写入管道 */
```

参数 fd 为 open( )得到的文件描述符。

对于 read( )，后面的参数表示接收管道数据的缓冲区 buffer 及其大小 maxbytes，如果消息缓冲区数据大于 maxbytes，则超出部分被丢弃。函数返回非负整型数时表示实际读取的字节数，否则为 ERROR。

对于 write( )，参数 buffer 和 nbytes 表示要写入管道的数据缓冲区和缓冲区数据字节长度。注意，nbytes 必须小于创建管道时指定的单条消息最大长度。成功时函数返回实际写入管道的字节数，总等于 nbytes；出错时函数返回 ERROR。

函数 read( )和 write( )都可能阻塞。write( )允许从中断服务程序中调用，此时如果没有空闲缓冲区，函数立即返回 ERROR 而不阻塞。

管道文件不再使用时应该将其关闭，以减少 I/O 资源占用。由函数 close( )完成。一般应该有和 open( )个数相同的 close( )。

```
#include "ioLib.h"
int close (int fd);  /* 关闭管道文件 */
```

### 3. 删除管道设备

管道不再需要时可以删除以释放资源。

```
#include "pipeDrv.h"
STATUS pipeDevDelete ( char * name, BOOL force );
```

参数 name 表示管道设备名称，必须和创建时的指定一致。参数 force 表示是否强制删除。在正常情况下（设置 force 为 false）删除管道必须满足两个约束：（1）所有该管道上打开的文件都已经关闭；（2）没有任务因该管道上 select 操作而阻塞。当不满足约束时，函数亦不会阻塞，而是返回 ERROR，此时 errno 为：（1）EMFILE 管道上有未关闭的文件；（2）EBUSY 至少有一个阻塞任务的 select 对象包括该管道。如果指定了强制删除（设置 force 为 true)，则不论后果立即删除管道，但不鼓励强制删除。

需要说明可能有些用户的系统中没有实现 pipeDevDelete( )。下面的代码例示了应用程序释放删除管道的方法。该代码比较简单，释放了管道占用的资源但是没有考虑上述两个约束。

```
STATUS pipeDevDelete ( char *name )
{
   DEV_HDR * pDevHdr;
```

```
    char     * pNameTail;

    /* 查找设备 */
    if ((pDevHdr = iosDevFind (name, &pNameTail)) == NULL)
        return ERROR;
    if (strcmp(name, pDevHdr->name)!=0)
        return ERROR;
    /* 删除管道设备并释放内存 */
    iosDevDelete (pDevHdr);
    free (pDevHdr);
    return OK;
}
```

### 2.5.3 管道 I/O 控制

管道以 I/O 设备形式出现，因此相比前述其他 IPC 机制多了 I/O 控制功能。pipeDrv 实现了如表 2-3 所示的 I/O 控制命令：

表 2-3 I/O 控制命令

| 命　令 | 示　例 | 意　义 |
|---|---|---|
| FIOGETNAME | status = ioctl (fd, FIOGETNAME, &nameBuf); | 取管道文件描述符对于的管道设备名称 |
| FIONREAD | status =ioctl(fd,FIONREAD, &nBytesUnread); | 读管道中第一条消息剩余未读字节数到 nBytesUnread。如果管道没有消息，nBytesUnread=0 |
| FIONMSGS | status = ioctl (fd, FIONMSGS, &nMessages); | 取管道中剩余消息条数到 nMessages。如果管道没有消息， nMessages=0 |
| FIOFLUSH | status = ioctl (fd, FIOFLUSH, 0); | 丢弃管道中所有消息 |

#### 关于 FIONREAD 和 FIONMSGS

对于磁盘文件，可以通过 FIONREAD 分多次读取数据。对于管道，请考虑下列代码：

```
NBytesRead = read( pipeFd, buffer, n);
if(ioctl(pipeFd, FIONREAD, & nBytesUnread)!=ERROR)
    nBytesRead += read( pipeFd, buffer+ nBytesRead, nBytesUnread);
...
```

不要期望上述语句会恰好读出一条消息。在任何情况下，不论第一条语句中 n 为何值（n>0），上述第一次 read( )是针对管道中第一条消息，而 ioctl( )和第二次 read( )是针对管道中第二条消息。源于 VxWorks 管道是基于消息队列这一特性，对于一条消息只能通过一次读取，如果该条消息没有读完，则管道将剩余部分丢弃，同时 FIONREAD 只能得到下一条消息长度。

FIONREAD 最大的用途在于：读取管道的任务可以先据此判断管道中是否有消息（有则 nBytesUnread 大于 0），然后作相应处理以避免阻塞。另外一个用途就是在读取一条消息之前得到消息长度，然后分配缓冲区。如果消息都很短，后一个用途是不重要的。

与 FIONREAD 对应，命令 FIONMSGS 为消息发送者提供了避免阻塞的方法。同样是先通过 FIONMSGS 取得管道中消息条数，比较管道容量即可知道继续发送是否将被阻塞。

必须指出，在写者（生产者）或者多读者（消费者）应用环境下，不仔细考虑任务调度时，不论发送任务还是接收任务，上述避免阻塞的做法是不可靠的。当然容易利用互斥信号量实现可靠的避免阻塞。

## 2.5.4 管道示例

下面给出使用管道 IPC 的例子 vxPipe.c。主要有 3 个函数如表 2-4 所示：

表 2-4　vxPipe.c 的 3 个函数

| | |
|---|---|
| testPipe( ) | 主函数，创建管道设备，打开关闭管道文件，释放资源；生成多个写者任务 writer( )和读者任务 reader( )；输出中间结果 |
| writer( ) | 根据任务优先级生成一条消息并写到管道，不考虑阻塞；记录中间结果 |
| reader( ) | 从管道读多条消息，考虑阻塞；记录中间结果 |

writer( )和 reader( )将中间结果记录在全局的 logBuf 中，运行结束时由主函数 testPipe( )输出显示。对 logBuf 的访问用到了互斥信号量 semMutex。另外一个计数信号量 semCounter 用于等待所有读者任务和写者任务运行结束。

运行程序时，直接在 shell 下输入 sp testPipe 即可。

```
/*
 * vxPipe.c: test pipe operation
 */
#include "vxworks.h"
#include "taskLib.h"
#include "semLib.h"
#include "pipeDrv.h"
#include "ioLib.h"
```

```
#include "iosLib.h"
#include "string.h"
#include "errno.h"
#define MAX_MSGS 2         /* 管道最多记录消息条数 */
#define MAX_MSG_LEN 100    /* 最大单条消息长度 */
#define TEST_NUM 5         /* 测试次数对应写者个数 */
#define PIPE_NAME  "/mypipe"
int  pipeFd = NULL;             /* 管道文件描述符 */
SEM_ID   semMutex = NULL;       /* 互斥信号量 */
SEM_ID   semCounter = NULL;     /* 计数信号量 */
char logBuf[TEST_NUM*3][MAX_MSG_LEN];   /* 记录中间结果缓冲区 */
int  nLog = 0;                          /* 缓冲区指针*/

void fatal( char * errInfo ) { printf("%s\n",errInfo); exit(-1); }

void writer( )   /* 写 1 条消息到管道 */
{
    int  taskPri;
    char szMsgBuf[MAX_MSG_LEN];
    int  msgLen, result;

    /* 生成消息并记录 */
    taskPriorityGet(taskIdSelf(), &taskPri);
    sprintf(szMsgBuf, "msg from writer %d", taskPri);
    msgLen = strlen(szMsgBuf);
    semTake(semMutex, WAIT_FOREVER);
        sprintf( logBuf[nLog], "---S: [ %s ] ready", szMsgBuf);
        nLog++;
    semGive(semMutex);

    /* 写消息到管道并记录 */
    result = write(pipeFd, szMsgBuf, msgLen);
    semTake(semMutex, WAIT_FOREVER);
        if(result>=0)
        {
            sprintf( logBuf[nLog], "---S: [ %s ] bytes wrote: %d, total: %d",
                szMsgBuf, result, msgLen);
```

```
        }
        else
            sprintf( logBuf[nLog], "---S: [ %s ] err: %x", szMsgBuf, errno);
        nLog++;
semGive(semMutex);

    semGive(semCounter);
}
void reader( )    /* 从管道读取多条消息 */
{
    int nRead, nUnread, nMsg;
    char szMsgBuf[MAX_MSG_LEN];
    nMsg = 0;
    while( nMsg<TEST_NUM )    /*循环读取消息 */
    {
        if( ioctl(pipeFd, FIONREAD, &nUnread)!=ERROR )
        {
            if(nUnread==0)  /* 没有消息 注意要使低优先级写者任务有机会运行 */
            {
                taskDelay(2);  continue;
            }
            /* 读取消息 记录所读消息 */
            nRead = read( pipeFd, szMsgBuf, nUnread );
            szMsgBuf[nRead]=0;   /* 添加字符串结束符'\0' */
            semTake(semMutex, WAIT_FOREVER);
                sprintf( logBuf[nLog], "---R: [ %s ] bytes read: %d",
                    szMsgBuf, nRead);
                nLog++;
            semGive(semMutex);
            nMsg++;
        }
        else
            fatal("reader: ioctl on pipe failed!");
    }
    semGive(semCounter);
}
```

```
void testPipe( )  /* 主函数 */
{
    int i, j;
    const int writerPri[TEST_NUM] = { 50, 51, 52, 49, 53 };
    const int readerPri = 40;
    nLog = 0;

    /* 初始化（1）创建管道设备；（2）打开管道文件 */
    if(pipeDevCreate(PIPE_NAME, MAX_MSGS, MAX_MSG_LEN)!=OK)
        fatal("cant create pipe dev");
    if((pipeFd=open(PIPE_NAME, O_RDWR, 0))==ERROR)
        fatal("Writer: cant open pipe");

    /* 初始化，创建信号量 */
    semMutex   = semMCreate(SEM_Q_FIFO);
    semCounter = semCCreate(SEM_Q_FIFO, 0);
    if(!semMutex || !semCounter)     fatal("cant create semaphore");

    /* 生成多个写者任务 */
    for( i=0; i<TEST_NUM; i++ )
        taskSpawn( 0, writerPri[i], 0, 2000, writer,
            0, 0, 0, 0, 0, 0, 0, 0, 0, 0 );
    /* 生成一个读者任务*/
    taskSpawn( 0, readerPri, 0, 2000, reader, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 );
    /* 等待所有任务运行完毕 */
    i = 0;
    while( i++ < TEST_NUM+1 )
        semTake(semCounter, WAIT_FOREVER);

    /* 输出读者—写者运行的中间结果 */
    printf("\n*** Logged result:\n");
    for( i=0; i<nLog; i++ )
        printf("  <%2d> %s\n", i, logBuf[i]);

    /* 清理，退出 */
    semDelete(semMutex);     semMutex  = 0;
    semDelete(semCounter);   semCounter= 0;
    close(pipeFd);           pipeFd = 0;
```

```
    printf("delete pipe: %s\n", pipeDevDelete(PIPE_NAME)== K?"ok":"error");
}
```

可以得到输出结果为：

```
*** Logged result:
  < 0> ---S: [ msg from writer 50 ] ready
  < 1> ---S: [ msg from writer 50 ] bytes wrote: 18, total: 18
  < 2> ---S: [ msg from writer 51 ] ready
  < 3> ---S: [ msg from writer 51 ] bytes wrote: 18, total: 18
  < 4> ---S: [ msg from writer 52 ] ready
  < 5> ---S: [ msg from writer 49 ] ready
  < 6> ---S: [ msg from writer 53 ] ready
  < 7> ---R: [ msg from writer 50 ] bytes read: 18
  < 8> ---R: [ msg from writer 51 ] bytes read: 18
  < 9> ---S: [ msg from writer 49 ] bytes wrote: 18, total: 18
  <10> ---S: [ msg from writer 52 ] bytes wrote: 18, total: 18
  <11> ---R: [ msg from writer 49 ] bytes read: 18
  <12> ---R: [ msg from writer 52 ] bytes read: 18
  <13> ---S: [ msg from writer 53 ] bytes wrote: 18, total: 18
  <14> ---R: [ msg from writer 53 ] bytes read: 18
delete pipe: ok
```

# 2.6 信　　号

信号和管道一样，也是UNIX系统中传统的通信方式，用于通知进程已经发生的某种条件，如图2-11所示，这种通知方式如同中断一样，不同的是该“中断”源于软件——另一任务，中断服务程序，或者内核，因此也称“软中断”。

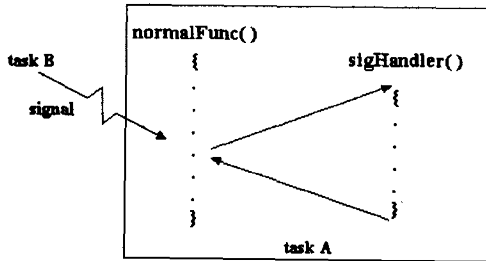



图2-11　信号IPC

在实现形式上，不同的信号具有形如 SIGXXX（“XXX”具体区别不同的信号）的不同名称，惟一对应一个按照标准定义的的整数值。这些信号有不同的来源，其中有些有特定语义，有些信号完全由应用程序定义。

实际上，信号是一种相当复杂的技术。和前面各种 IPC 机制相比，信号最本质的不同在于“异步性”。以图 2-11 中 task A 为例，这种“异步性”体现在：task A 完全不确定在执行到哪条语句时会到来，或者对于已经发生的信号，task A 完全不确定信号何时来过了（不要考虑让 task A 不断查询一个变量来判断信号是否发生，因为那是极不现实的）。相比之下，前面各种 IPC 机制总是在合适的地方加入语句去完成设计预期的事件（等待信号量，写管道，读消息队列等），也就是说这些 IPC 机制本质上具有“同步性”。

信号的独特特点使其具有许多其他 IPC 无法实现的重要用途。对复杂系统而言，灵活运用信号是非常关键的。本节只探讨将信号作为用户任务之间的一种 IPC 机制的最简单的方法。第 4 章将更详细介绍信号的使用。

下面是一段示例性的代码，并不完整，目的在于说明一种简单的信号处理过程。

```
void task1 ( void )    /* task1：发送信号 */
{
   printf ( "send signal to the other task" );
   if ( kill (task2, SIGUSR1)==ERROR )
      ...  /* 发送信号出错处理 */
}

/* 信号处理函数定义 */
void sigHandler (int signo)
{
    ...
}

task2 ( )     /* task2：接收信号 */
{
   struct sigaction sa;
   sigset_t newset;

   /* 为信号 SIGUSR1 设置信号处理函数 */
   sa.sa_handler = sigHandler;
   sigemptyset(&sa.sa_mask);
   sa.sa_flags = 0;
   if (sigaction(SIGUSR1,  &sa,  NULL))
```

```
    ... /* 设置信号处理函数出错的处理 */

  /* 设置阻塞信号集 */
  sigemptyset(&newset);
  sigaddset (&newset, SIGUSR2);
  sigprocmask(SIG_BLOCK, &newset, NULL);

  ...   /*在某点上信号将到来*/
}
```

在这个简单的例子中我们希望通过 SIGUSR1 在任务 1 和任务 2 之间进行通信。信号 SIGUSR1 的处理函数为 sigHandler。设置阻塞信号集包括 SIGUSR2，避免发生信号嵌套。内部的处理过程如图 2-12 所示。

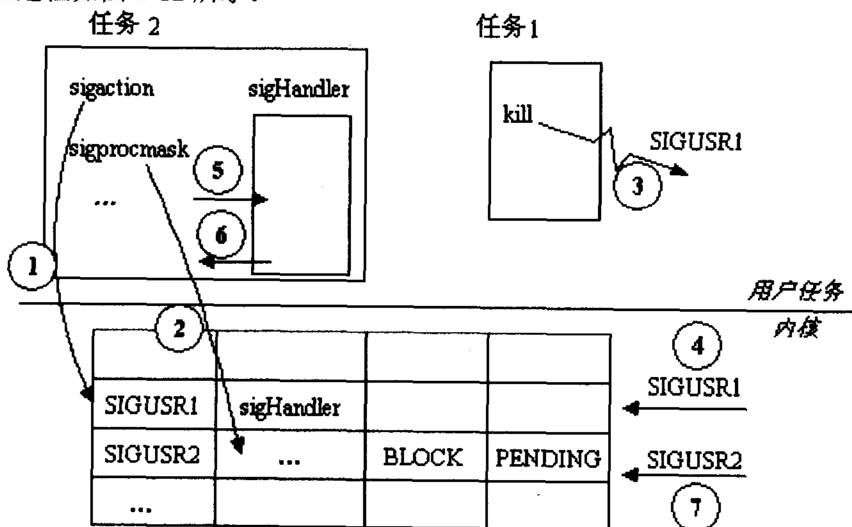



图 2-12 信号处理

（1）任务 2 调用 sigaction( )声明设置信号 SIGUSR1 的处理函数为 sigHandler( )，内核将在其内部数据结构中建立指定信号和对应信号处理函数指针的关联项目；

（2）接下来，任务 2 调用 sigprocmask( )告诉内核任务要接收的信号，这里，期待接收 SIGUSR1，SIGUSR2 将被阻塞；

（3）在某个时刻，另一个任务 1 发送信号 SIGUSR1 到任务 2；

（4）内核在发现任务任务 2 设置了信号处理函数要求捕获信号 SIGUSR1，因此立即递交；

（5）内核停止任务 2 当前正在进行的事情，转入其设置的信号处理函数 sigHandler( )；

（6）通常信号处理函数运行完毕，返回任务 2。

如果 SIGUSR2 在某时刻到达，系统发现任务 2 指定 SIGUSR2 为阻塞，因此不会递交

给任务 2，而是在内部数据结构中登记为 PENDING。即图 2-12 中“7”号箭头表示的。

从第（5）步看，信号处理函数 sigHandler( )运行时无法确知任务 2 运行到了哪一点，如同 ISR 运行一样，具有异步性。

向另一个任务发送信号，不管 BSD 接口还是 POSIX 接口，都调用 kill( )实现。发送给一个任务信号并不一定导致该任务结束（当然可以这样实现），这样命名也许是源于早期 UNIX 环境下信号较多用于杀进程（SIGKILL）的缘故。

## 2.7 socket

socket 通信又称“套接字”通信，用来通过网络实现位于不同计算机系统上任务间的双向通信。socket 通信是整个网络通信的基础，其细节和使用的网络协议有关。第 8 章将专门介绍 socket 通信。

# 第3章 POSIX编程

## 3.1 POSIX 标准简介

POSIX具有多重含义，通常指POSIX标准，该标准是一个可移植操作系统接口(Portable Operating System Interface)，由 IEEE 提出，ANSI 和 ISO 将其标准化。POSIX 的目标是使应用程序源代码可以在兼容 POSIX 的操作系统上移植。理想的目标是应用程序移植到另一个操作系统只需要重新编译就可以运行。POSIX 最后一个字母“X”表达了这种超乎操作系统差异的理想。目前并没有实现这种理想：从操作系统看，由于目标、要求、理念、条件等差异，并不是所有的操作系统都实现了 100% POSIX 兼容；从应用程序看，很多代码编写使用了特定操作系统支持的调用，没有很好地使用 POSIX 接口。但是，很显然，使用 POSIX 接口的应用程序在兼容 POSIX 的操作系统间移植将是很轻松的事情。

POSIX 标准是一个处于不断发展之中的庞大体系，包括：

- 1003.1　系统 API
- 1003.2　SHELL 及工具
- 1003.3　POSIX 符合性测试方法
- 1003.5　ADA 语言接口
- 1003.13　标准化可移植实时应用环境 AEP

其中，POSIX 1003.1 系列标准是 POSIX 最主体内容，也是我们关心的部分。该系列内容由如下主体定义以及一些扩展和增补组成：

- 1003.1　1988 年通过，基本 OS 接口
- 1003.1b　1993 年通过，实时扩展
- 1003.1c　1995 年通过，线程扩展
- 1003.1d　1999 年通过，实时扩展
- 1003.1j　2000 年通过，高级实时扩展
- 1003.1q　2000 年通过，事件数据流跟踪

VxWorks 正在不断完善对 POSIX 的支持。应用程序使用 POSIX 定义的标准，API 可以方便程序以后在兼容 POSIX 标准的操作系统上进行移植。

VxWorks 的 POSIX 组件如表 3-1 所示：

表 3-1　POSIX 组件

| 库　名　称 | 标　　准 | 说　　明 | 宏 |
| --- | --- | --- | --- |
| AioPxLib | 1003.1b | POSIX 异步 I/O 函数 | INCLUDE_POSIX_AIO |
| pthreadLib | 1003.1c | POSIX 线程函数 | INCLUDE_POSIX_PTHREADS |
| clockLib | 1003.1b | POSIX 时钟 | INCLUDE_POSIX_CLOCKS |
| mqPxLib | 1003.1b | POSIX 消息队列 | INCLUDE_POSIX_MQ |
| mmanPxLib | 1003.1b | POSIX 内存锁定 | INCLUDE_POSIX_MEM |
| schedPxLib | 1003.1b | POSIX 调度 | INCLUDE_POSIX_SCHED |
| semPxLib | 1003.1b | POSIX 信号量 | INCLUDE_POSIX_SEM |
| sigLib | 1003.1b | POSIX 信号 | INCLUDE_POSIX_SIGNALS |
| timerLib | 1003.1b | POSIX 定时器 | INCLUDE_POSIX_TIMERS |

## 3.2　时钟和定时器

### 3.2.1　概述

通过底层时钟硬件和 BSP 的周期性定时中断服务支持，VxWorks 时钟管理维护一个系统日历时钟，该功能是下列处理的基础：

- 各种超时控制：任务延迟、消息队列、信号量等操作超时控制；
- 定时处理：以一定的时间间隔或在特定的时间通过一个信号唤醒一个任务；
- 看门狗处理；
- 任务调度中的时间片轮循。

称 VxWorks 维护的系统日历时钟为**全局实时时钟**。

和许多其他 VxWorks 服务不同，VxWorks 对时钟和定时器只提供 POSIX 接口。

#### 1. 内部数据结构

**（1）时钟和定时器**

POSIX 要求定义表示时钟和定时器的数据结构：

| | |
| --- | --- |
| clockid_t | 在时钟和定时器函数中标识时钟 ID |
| timer_t | timer_create( )创建的定时器 ID |

对上述类型定义允许通过形参传递表示时钟和定时器。

POSIX 定义 clockid_t 为 CLOCK_REALTIME 的时钟有特殊意义，用于表示系统范围内的全局实时时钟。除 CLOCK_REALTIME 时钟之外，POSIX 框架允许操作系统提供多个

“虚拟”时钟。VxWorks 只实现了一个 CLOCK_REALTIME 时钟，所有的 VxWorks 任务都使用该时钟，包括各种延迟处理，以及各种需要 clockid_t 参数的函数。如果硬件支持，高级用户可以修改 BSP 实现虚拟时钟。

对于 CLOCK_REALTIME，POSIX 限制计时精度不能低于下限_POSIX_CLOCKRES_MIN（20 ms）。VxWorks 提供的更高的精度，具体由系统时钟速率决定。按照 BSP 对此的默认设置（SYS_CLK_RATE=60），VxWorks 时钟表示的最小计时精度为六十分之一秒，约 17ms。在需要精确计时的场合，可以看到计时精度是非常重要的。后文将提到时间规格 timespec，其表示的时间精度为纳秒，时钟精度影响将是显而易见的。

上面提到的几个宏常量定义如下：

```
#include "time.h"
#define CLOCK_REALTIME  0x0        /* 系统范围内实时时钟 */
#define _POSIX_CLOCKRES_MIN 20     /* 最低时钟精度，毫秒 */
```

& POSIX 定义的 timer_t 属于进程，即定时器 ID 在进程范围内是独一无二的；VxWorks 中定时器 ID 在整个系统范围内定义。

**（2）时间规格**

许多和时间有关的函数通过 timespec 数据结构作为入参或者返回值，timespec 称为**时间规格**。定义为：

```
#include "time.h"
struct timespec    /*时间规格*/
{
   time_t tv_sec;     /* 秒 */
   long tv_nsec;      /* 纳秒，必须满足 0 ≤ tv_nsec < 1,000,000,000 ! */
};
typedef long time_t;
```

在 VxWorks 和其他一些库（如 GNU C）中，time_t 定义为长整型（long）。

在不同的场合，时间规格 timespec 可以表示“绝对”时间或者“相对”时间，两者的区别是相对的。

- 绝对时间：VxWorks 系统时钟 CLOCK_REALTIME 就属于绝对时间。系统维护一个 timespec 类型的变量，时钟滴答时修改该结构变量的值。BSP 指定的时钟溢出速率决定了时钟滴答速率，从而决定了 CLOCK_REALTIME 精度；
- 相对时间：许多函数通过一个 timespec 变量表示某个时间间隔，该时间值相对于某时刻的 CLOCK_REALTIME 的系统时间。如 nanosleep( )等函数中一般以相对时间表示延迟。表示相对时间的时间间隔为（tv_sec*10*9 + tv_nsec）纳秒。

& 一个有趣的问题是：当使用绝对时间时，如何以有限的字长来表示无限延伸的

时间。以当前 32 位处理器为例，一个 timespec 表示的绝对时间最大值为 4294967295（十六进制为 0xffffffff），即 136 年多一点。假设当前系统时钟已经是 0xfffffffe，而程序需要在两秒后进行一个定时处理，很显然，这时只能采用相对时间定时器。事实上，POSIX 甚至允许系统时钟和基于该时钟的绝对时间定时器有不同的最大允许值。在 VxWorks 中，两者都是一致的，即系统时钟最大允许时间和其他任何引用绝对时间的场合都可以采用该机器字长下 time_t 变量允许的所有值。

ANSI C 定义 time_t 解释为绝对时间时，表示从 GMT 时间 1970 年 1 月 1 日零时刻起经过的秒数。

另一个数据结构定义**定时器时间规格** itimerspec，用于表示定时器溢出时间和定时器重装入时间间隔。

```
#include "time.h"
struct itimerspec      /*定时器时间规格*/
{
   struct timespec it_interval; /*时间间隔，用于定时器到期后自动重装入*/
   struct timespec it_value;         /*首次到期时间 */
};
```

启动定时器时 timer_settime( )指定首次到期时间 it_value，即从定时器启动时开始，经过 it_value 时间长度后定时器到期并发送定时器信号；此后，定时器自动根据 it_interval 重新装入，it_interval 同样由 timer_settime( )指定。可以看出，it_value 和 it_interval 都是相对时间（后面会看到，可以以相对时间或者绝对时间设置定时器）。

### 2. 定时器信号事件

POSIX 1003.1b 扩展了信号系统，增加了队列信号定义。一个定时器可以对应一个信号事件，当定时器到期时发送给定时器创建任务。又称为**定时器到期信号事件**。信号事件由结构体 sigevent 定义。

和其他 POSIX 扩展定义的信号事件，如消息队列信号事件相比，定时器信号事件独特之处在于：创建定时器的任务不一定启动了定时器，但是信号只能而且必须发送给创建定时器的任务。

系统递交定时器信号时，如果出现下面的情况，系统将通过 logMsg( )记录定时器错误情况：

- 创建定时器的任务已经结束；
- 任务没有设置信号处理函数。

## 3.2.2 时钟

VxWorks 时钟支持库为 clockLib，实现了 POSIX 定义的时钟接口，包括设置、读取当前时间和读取时钟分辨率。

```
#include "time.h"
int  clock_gettime (clockid_t clock_id, struct timespec *tp);/*取时钟时间*/
int  clock_settime (clockid_t clock_id, const struct timespec *tp);
                                                        /*设置时钟时间*/
int clock_getres (clockid_t clock_id, struct timespec *res);
                                                        /*取时钟分辨率*/
```

上面 3 个函数成功时返回 0；如果参数无效，函数返回-1，errno=EINVAL。

参数 clock_id 总是为 CLOCK_REALTIME；tp 表示取出或者设置的时间（绝对时间）；res 表示当前时钟分辨率。

## 3.2.3 定时器

定时器通过时钟和信号量来使系统在指定时刻通知任务，使任务处理某种定时事件。标准定时器函数包括创建定时器、删除定时器和设置定时器。VxWorks 定时器库为 timerLib。

本节表述定时器时使用了下列提法：

- 定时器到期：从定时器启动开始，经过定时器时间规格 itimerspec.it_value 纳秒，即“到期”或“到时”(Expiration)；
- 定时器溢出：由于信号处理机制设计为只表示信号的有与无，而不能对同一信号的递交进行计数，因此，POSIX 定义当定时器到期发送信号时，如果前次信号还在挂起状态(即未被处理)，系统将不发送本次信号，此即一次“溢出”(Overrun)；
- 启动定时器：设置定时器首次到期时间大于 0，同时使定时器启动进行计数（Arm)；
- 停止定时器：设置定时器首次到期时间等于 0，使定时器停止计数（Disarm)。

本节另外需要说明下面几个 POSIX 定义的宏常量，其具体含义将在后文涉及。

```
#include "time.h"
#define  _POSIX_TIMER_MAX     32        /*每个任务最多可以创建的定时器数 */
#define  _POSIX_DELAYTIMER_MAX   32  /*1 个定时器最多溢出次数*/
```

## 1. 创建定时器

```
#include "time.h"
int timer_create ( clockid_t clock_id, struct sigevent * evp, timer_t *
pTimer );
```

参数 clock_id 表示时钟 ID 总是 CLOCK_REALTIME。如果成功，函数将创建定时器并通过参数 pTimer 返回定时器 ID；如果失败，函数返回-1。

创建时钟同时通过参数 evp 指定了时钟信号事件 sigevent。可以指定 evp=NULL，系统将发送默认信号事件：信号编码=SIGALRM，附加信息=定时器 ID。关于信号事件，可以参考第 4 章。

新创建的定时器没有被启动。启动定时器的动作即下文的“设置定时器”。

一个任务最多允许创建_POSIX_TIMER_MAX 个定时器，否则函数失败（errno= EMTIMERS）。

## 2. 设置（启动/停止）定时器

根据传递的入参，设置定时器即启动/停止定时器，由下述函数实现：

```
#include "time.h"
int timer_settime (  timer_t timerid, int flags,
       const struct itimerspec * value, struct itimerspec * ovalue );
```

参数 clock_id 表示时钟 ID 总是 CLOCK_REALTIME；value 表示定时器时间规格；flags 表示 value 属于绝对时间还是相对时间；ovalue 非 NULL 时返回原来的定时器设置。

timer_settime( )设置定时器时间规格。前文述及，value 表示的定时器时间规格包括：首次到期时间（value→it_value）和自动重装入的时间间隔（value→it_interval）。

定时器内部采取相对时间表示时间规格（首次到期时间和重装入的时间间隔），但是为 timer_settime( )传递的 value→it_value 可能为绝对时间，因此视 flags 参数的不同，函数设置首次到期时间的细节如表 3-2 所示。

表 3-2 flags 设置与结果

| flags | 结　果 |
|---|---|
| TIMER_ABSTIME（绝对时间） | 如果 value→it_value 已经成为“过去”，立即当成定时器到期发送信号；否则设置定时器首次到期时间为与当前系统时钟的差。即定时器将在 value→it_value 时刻首次到期 |
| 其他值（相对时间） | 设置定时器首次到期时间为 value→it_value，即经过 value→it_value 纳秒后定时器首次到期 |

如上所述设置定时器首次到期时间时，定时器可能已经处于启动状态，则定时器的时间规格将被覆盖。

如果 value→it_value 等于 0，则定时器将被停止。

定时器自动重装入时间由 value→it_interval 指定：如果 value→it_interval 表示的时间间隔为 0 纳秒，则定时器只工作一次即停止；否则，定时器首次到期后自动重装入新的到期时间为 value→it_interval，这样的定时器即周期定时器。value→it_interval 总表示为相对时间。

**timer_settime( )**

（1）可以在 ISR 中设置定时器，但是不能创建或者删除定时器；

（2）注意时间圆整效果：当定时器到期时间（包括程序设置的首次到期时间和定时器自动重装入的时间）落在两个正整数倍系统时钟精度之间时，系统会将到期时间圆整到较大倍数的值。例如，如果到期时间为 3′ 150″，而系统计时精度在该值附近只提供 3′ 140″和 3′ 157″两个计数，系统会取 3′ 157″作为定时器到期时间。这种圆整策略保证定时器不会提前，但可能稍后。

### 3. 读取定时器

读定时器包括读定时器时间规格（到期剩余时间，重装入值）和溢出计数。

```
#include "time.h"
int timer_gettime ( timer_t timerid, struct itimerspec * value );
int timer_getoverrun ( timer_t timerid );
```

timer_gettime ( )读取 timerid 定时器时间规格。timerid 表示定时器 ID；value 用于返回读取的结果，包括：到期剩余时间，重装入值。显然，value 中得到的都是相对时间。函数成功时返回 0；否则返回-1（errno=EINVAL）。

timer_getoverrun ( )读取 timerid 指定的定时器溢出计数。常用在定时器到期信号处理函数中。系统最多只记录_POSIX_DELAYTIMER_MAX 次溢出，超出部分无法确定次数。函数返回值：0~_POSIX_DELAYTIMER_MAX 表示溢出次数；-1 表示函数失败。

### 4. 删除定时器

```
#include "time.h"
int timer_delete ( timer_t timerid );
```

删除 timerid 表示的定时器，定时器可以处于未启动或者已启动状态。函数返回 0 表示成功；当 timerid 无效时返回-1（errno=EINVAL）。

应该避免删除存在未决信号的定时器（例如任务设置阻塞信号量将定时器信号暂时阻塞）。

### 5. VxWorks 特定函数

VxWorks 提供了另外两个使程序简化的非 POSIX 函数：

```
#include "time.h"
int timer_cancel ( timer_t timerid );        /*停止定时器*/
int timer_connect ( timer_t timerid, VOIDFUNCPTR routine, int arg );
                                             /*连接信号处理函数*/
```

timer_cancel( )相当于调用 time_settime( )时指定参数 value=NULL。

timer_connect ( )相当于调用 sigaction( )为定时器信号建立信号处理函数。定时器信号由创建定时器时的 evp 参数指定，当该信号递交给任务时，系统将以 arg 为参数调用 routine。如果不使用 timer_connect ( )，程序需要通过 sigaction( )设置信号处理函数（参考第 4 章“4.4 POSIX 信号接口”）。

除了上面两个函数，VxWorks 5.5 的 timerLib 还实现了两个 UNIX 的定时控制函数：

```
#include "unistd.h"
unsigned int ( unsigned int secs );
unsigned int alarm ( unsigned int secs );
```

sleep( )使调用任务睡眠 secs 秒。如果 secs 不等于 0，alarm( )将在 secs 秒后向调用任务发送 SIGALRM 信号。不论 secs 是否等于 0，调用 alarm( )后都将取消之前要求 alarm( )发送的信号（如果有的话）。

## 3.2.4 看门狗

“看门狗”就是一个定时器，在定时到期时会执行一个预先定义的动作。在没有操作系统的单片机时期，看门狗通常是一个硬件定时器，如果定时器到期，硬件将自动使整个系统复位，因此程序中需要每经过一定的间隔通过几条指令使看门狗复位。因此，系统正常运行期间，看门狗是不会动作的；当程序出错时，比如陷入死循环，看门狗将在一个较短的时间后“检测”到这一情况从而复位系统。

在 VxWorks 中，看门狗由软件实现，内核在系统时钟上维护一个看门狗队列。队列上每个看门狗定义一个预定义的动作（即一个函数）；每次系统时钟滴答时定时器加 1，如果到期，则在系统时钟 ISR 中调用预定义的函数。因此，VxWorks 看门狗功能更强大，任务可以通过预定义函数实现更复杂的故障检测与处理。

看门狗的软件结构定义为结构体 wdog。任务通过 wdLib 使用看门狗时只需要一个看门狗 ID，基本上不需要涉及 wdog 结构体内部细节。看门狗 ID 定义为 WDOG_ID，即：

```
#include "wdLib.h"
```

```
typedef struct wdog * WDOG_ID;
```

使用看门狗的过程包括：创建、启动、撤销和删除。调用下列 wdLib 函数实现：

```
#include "wdLib.h"
WDOG_ID wdCreate ( void );              /*创建看门狗*/
STATUS wdDelete ( WDOG_ID wdId );       /*删除看门狗*/
STATUS wdStart ( WDOG_ID wdId, int delay, FUNCPTR pRoutine, int arg );
                                        /*启动看门狗*/
STATUS wdCancel ( WDOG_ID wdId );       /*撤销看门狗*/
```

wdCreate ( )创建看门狗只初始化了必要的内核数据，并未启动。

wdStart( )将创建的看门狗加入到系统时钟的看门狗队列同时启动看门狗计数。为 wdStart( )指定的参数 delay 表示以 tick 为单位的看门狗到期时间；pRoutine 为看门狗预定义的动作函数，arg 为系统调用 pRoutine 时传递的参数。注意 pRoutine 调用环境为中断环境，因此必须符合 ISR 的约束，即不能进行任何可能引起阻塞的调用。对于已经启动的看门狗，wdStart ( )将重新设置到期时间和预定义函数及其参数。

wdCancel ( )撤销指定的看门狗：停止计数，从系统时钟的看门狗队列中移走看门狗。该函数可以从 ISR 中调用。正常情况下总是在定时器到期之前通过该函数使看门狗停止工作。如果定时器到期还没有调用此函数，则解释为程序故障，系统将让看门狗“履行职责”，即带参数 arg 调用 pRoutine。

wdCancel ( )删除看门狗，释放内存资源。如果已启动，该调用将从系统时钟的看门狗队列中移走看门狗。

下面这段示例代码在调用函数 myFunc( )时通过看门狗进行检测，如果 myFunc( )在 100 个 tick 还没有结束，看门狗将记录一条消息。

```
WDOG_ID wid = wdCreate ( );
wdStart ( wid, 100, logMsg, "myFunc maybe dead!" );
myFunc ( );
wdCancel ( wid );
```

& 任何任务都可以创建一个或多个看门狗。看门狗定时器只作用一次，一次到期或者被取消后，还可以通过 wdStart( )重新启动。
本小节内容不是 POSIX 标准的一部分，但是看门狗是 VxWorks 内核功能之一，并且基于系统时钟实现，因此在这里讲述。

### 3.2.5 示例

作为一种让任务在某个设计间隔后唤起自身完成某些动作的手段，定时器应用非常广泛。下面是一个比较简单但却包括较多处理细节的时钟和定时器功能演示程序。除此之外，

该程序还意在说明 POSIX 1003.1b 扩展队列信号的处理过程（这部分内容的细节需要参考第 4 章）。因此，我们在设置信号处理函数时没有使用 VxWorks 提供的简便方法。

程序包括 3 个函数，如表 3-3 所示。

表 3-3　3 个函数及其作用

| | |
|---|---|
| timer_demo | 主体函数：完成各种创建和初始化；等待定时器到期；处理过程（简化为显示定时器到期时间）；任务结束清理 |
| timer_handler | 定时器到期信号处理函数：给出信号量通知主体任务接收了定时器信号；运行 TEST_NUM 次后停止定时器 |
| show_time | 显示当前时间 |

主体函数和信号处理函数的设计：信号处理函数非常简单，通常就是给出某个信号量；主要的处理都在主体任务中完成。两者之间通过信号量进行同步。许多定时器应用（或者说许多信号处理应用）中都具有这样的哲学，使信号处理函数简洁高效。

```
/*
 * timer_demo.c: POSIX 定时器功能演示
 */
#include "vxWorks.h"
#include "semLib.h"
#include "time.h"
#include "signal.h"
#include "errno.h"
#define TIMER_SIG   SIGALRM    /*定时器信号编号*/
#define TEST_NUM    8          /*测试次数*/
#define FATAL_ERROR(s) { \
       printf("%s [errno: 0x%x]\n", s, errno); \
       taskSuspend(0);  \
   }
SEM_ID semaphore;              /*主体任务与信号处理 IPC*/
void timer_handler ( int signo, siginfo_t * pInfo, void * pContext );
                               /*信号处理函数*/
void show_time ( void );       /*显示时间*/

void timer_demo ( )    /*主函数*/
{
   struct itimerspec itmsp;     /*定时器时间规格*/
   struct sigevent tmevent;     /*定时器信号事件*/
   timer_t tm;                  /*定时器 ID*/
   struct sigaction tmact;      /*定时器信号处理*/
   int  i;
```

```
    struct timespec tp;

    tp.tv_sec  = 0xfffffff0;      /*设置当前时间*/
    tp.tv_nsec = 0;
    clock_settime( 0, &tp);
    show_time ( );
    if ( (semaphore = semCCreate ( SEM_Q_FIFO, 0 )) == 0 )  /*创建信号量*/
        FATAL_ERROR("create semaphore failed")

    /*设置定时器信号处理函数*/
    sigemptyset ( &tmact.sa_mask );
    tmact.sa_u.__sa_sigaction = timer_handler;
    tmact.sa_flags = SA_SIGINFO;
    sigaction ( TIMER_SIG, &tmact, NULL );
    /*创建定时器*/
    tmevent.sigev_signo = TIMER_SIG;
    tmevent.sigev_value.sival_int = (int)&tm;
    tmevent.sigev_notify = SIGEV_SIGNAL;   /*must!*/
    if ( timer_create ( CLOCK_REALTIME, &tmevent, &tm ) != 0 )
        FATAL_ERROR("create timer failed")

    /*启动定时器*/
    itmsp.it_value.tv_sec  = 6;
    itmsp.it_value.tv_nsec = 123;
    itmsp.it_interval.tv_sec = 2;
    itmsp.it_interval.tv_nsec = 123;
    timer_settime ( tm, TIMER_ABSTIME+1, &itmsp, NULL );

    show_time ( );      /*显示开始的时间*/

    /*等待定时器到期，简单的记录定时器到期时间*/
    for ( i = 0; i<TEST_NUM; i++ )
    {
        if ( semTake( semaphore, WAIT_FOREVER) == -1 )
            FATAL_ERROR( "error wait for semaphore" )
        show_time( );
    }
    semDelete ( semaphore );
    timer_delete ( tm );
}

/*定时器信号处理函数*/
```

```
void timer_handler ( int signo, siginfo_t * pInfo, void * pContext )
{
    static test_num = 0;
    timer_t * tm = (timer_t *) pInfo->si_value.sival_int;
    struct itimerspec itmsp;

    if ( signo != TIMER_SIG )  return;

    if ( ++test_num >= TEST_NUM )  /*如果已经 TEST_NUM 次，停止定时器*/
    {
        itmsp.it_value.tv_sec  = 0;
        itmsp.it_value.tv_nsec = 0;
        timer_settime ( *tm, TIMER_ABSTIME+1, &itmsp, NULL );
    }
    semGive ( semaphore );    /*给出信号量*/
}

void show_time( )   /*格式化输出当前时间*/
{
    struct timespec tp;
    clock_gettime( 0, &tp);
    printf ( "current time: %x s %x ns\n", tp.tv_sec, tp.tv_nsec );
}
```

在 shell 下运行程序 sp timer_demo，可以得到类似下面的输出结果。说“类似”是因为当系统时钟精度不同时输出结果将会有微小变化。

```
current time: fffffff0 s 0 ns              ←开始时间
current time: fffffff6 s fe502a ns         ←第 1 次定时器到期
current time: fffffff8 s 1fca054 ns        ←第 2 次定时器到期
current time: fffffffa s 2faf07e ns        ←...
current time: fffffffc s 3f940a8 ns
current time: fffffffe s 4f790d2 ns
current time: 0 s 5f5e0fc ns
current time: 2 s 6f43126 ns
current time: 4 s 7f28150 ns
```

程序设计了 8 次定时器到期，为了演示系统时钟在达到计数边界时的表现，程序选择在 0xfffffff0 时刻开始运行，以相对时间表示，第一次定时器在 6 秒 123 纳秒后到期，以后 7 次到期由重装入时间决定，即 2 秒 123 纳秒。可以看到，定时器实际到期时间被向上圆整到系统时钟精度的整数倍。

可以看到，当系统时钟计数溢出，即由 0xffffffff 秒变为 0 秒后，以相对时间表示的定

时器到期不受影响。可以修改程序，使第一次到期时间为绝对时间的 6 秒 123 秒，看看有什么结果。

# 3.3 内存锁定

**分页**和**交换**使程序运行在比物理内存空间更大的虚拟地址空间。理论上使应用程序的访问空间只受处理器字长限制（除开操作系统限制不能让应用程序访问的部分）。但是分页和交换开销将使程序运行时间变得不确定，因此，作为实时扩展，POSIX 1003.1b 定义可以将内存锁定，即不让操作系统将进程使用的页面换出内存，来避免发生交换过程中不确定的延迟。

为保证实时性能，VxWorks 不进行分页和交换。为了体例完整，我们仍然列出 VxWorks 实现的内存锁定函数。这些函数通过库 mmanPxLib 提供。实际上，这些函数是空壳函数，直接返回 OK。

```
#include "sys/mman.h"
int mlockall ( int flags );                         /*将进程所有页面锁定在内存*/
int munlockall ( void );                            /*解锁进程所有页面*/
int mlock ( const void * addr, size_t len );        /*锁定指定页面*/
int munlock ( const void * addr, size_t len );      /*解锁指定页面*/
```

# 3.4 线　　程

VxWorks 5.5 开始支持 POSIX 1003.1c 线程实时扩展。习惯上常常将 POSIX 线程称为“pthread”。本节内容不适于 VxWorks 5.4。

VxWorks 的 POSIX 线程实现表现出与 POSIX 标准的差异。这些差异源于 VxWorks 自身多方面的特点。

## 差异 1：线程属性

VxWorks 本质上只包括系统和任务两个概念，线程以任务形式实现，线程不属于任何进程，只属于整个系统。

- POSIX 定义线程竞争 CPU 资源的范围包括

  PTHREAD_SCOPE_SYSTEM 整个系统范围内竞争

  PTHREAD_SCOPE_PROCESS 所在进程范围内竞争

  VxWorks 只实现了前者；
- POSIX 定义每个线程有其调度策略；VxWorks 只有一个调度策略——整个系统调度策略，线程调度策略默认设置源于当前系统设置。

## 差异 2：IPC

VxWorks 任务在同一实地址空间运行，没有任何保护机制。任何任务可以直接访问其他任务数据。

POSIX 定义了进程间共享信号量和条件变量等 IPC 机制。配置选项

_POSIX_THREAD_PROCESS_SHARED

和相关函数

pthread_condattr_getpshared( ) 取进程共享条件变量属性

pthread_condattr_setpshared( ) 设置进程共享条件变量属性

pthread_mutexattr_getpshared( ) 取进程共享互斥对象状态

VxWorks 没有实现。

## 差异 3：用户和组

VxWorks 没有用户和组的概念。

VxWorks 也不需要实现和用户及组管理相关的内容，因此函数

getlogin( ) / getlogin_r( ) 取和当前进程相关的用户名

getgrgid( ) / getgrgid_r( ) 检索指定的组信息

getpwnam( ) / getpwnam_r( ) 检索指定名称的用户信息

getpwuid( ) / getpwuid_r( ) 检索指定 ID 的用户信息

没有实现。

## 差异 4：进程

VxWorks 没有进程的概念，因此

fork( ) 创建新进程

pthread_atfork( ) 定义 fork( )处理函数

wait( )等待进程结束

没有实现。

## 差异 5：线程撤销

VxWorks 没有实现全部的 POSIX 撤销点，因为 VxWorks 中没有这些函数。见 3.4.6“线程撤销”一节。

在 VxWorks 中，POSIX 线程从本质上讲是一个任务。在本节，我们将看到 POSIX 线程为任务增加了许多属性：

- POSIX 线程除了任务 ID 还包括一个不同的线程 ID；
- POSIX 线程具有离合性，用于控制线程创建者在所创建线程退出前是否要阻塞；
- POSIX 线程允许撤销，线程被撤销时可以调用一个清除函数；
- POSIX 线程私有数据，允许 POSIX 线程根据一个 KEY 访问。

& POSIX 线程本质上以任务实现，因此本节介绍的许多 POSIX 线程控制都可以对应到某个 VxWorks 任务调用。但是编程时不应该对线程和任务之间的联系作任何假设，因为任何假设都是与 POSIX "源代码可移植" 这一基本思想背道而驰的。

## 3.4.1 线程创建

### 1. 属性对象

所有POSIX线程属性通过一个属性对象表示，该属性对象定义为结构体pthread_attr_t。POSIX 定义了一系列函数设置和读取各个属性值，用户不需要知道 pthread_attr_t 定义的细节。

POSIX 定义了函数 pthread_attr_init( )和 pthread_attr_destroy( )分别用于属性对象的初始化和删除。

```
#include "pthread.h"
int pthread_attr_init ( pthread_attr_t * pAttr );      /*初始化属性对象*/
int pthread_attr_destroy ( pthread_attr_t * pAttr );  /*删除属性对象*/
```

属性对象的初始化即为各个属性赋默认值。POSIX 未限制默认，VxWorks 对默认值的定义在下面给出。属性对象被初始化后，程序往往还需要为单个属性设置合适的值。

删除属性对象不是释放对象所占用的资源，而是将各个属性值修改为无效值。属性对象删除后不能用于线程创建，除非被再次初始化。

属性对象被删除前可以多次在 pthread_create( )使用。

& 事实上，完全遵循 POSIX 的应用有时也不能保证彻底的源代码级可移植性，例如与栈设置有关的属性在不同系统上进行移植时常常需要修改。因此，POSIX 将线程属性集中通过属性对象表示使修改相对集中。同时便于未来扩充属性定义。

POSIX 定义的所有线程属性如下。

**（1）栈大小**

作用：指定新线程使用栈的大小，可以圆整到页面大小，属性如表 3-4 所示。

**表 3-4 栈大小的属性**

| 默认值 | 对 taskLib 设置的默认栈大小 |
|---|---|
| 访问函数 | pthread_attr_getstacksize( )<br>pthread_attr_setstacksize( ) |

### （2）栈地址

作用：指定用户为新线程申请的栈起始地址。默认值为 NULL，此时系统将在创建线程时在系统堆上为新线程申请栈空间，属性如表 3-5 所示。

表 3-5 栈地址的属性

| 默认值 | NULL |
|---|---|
| 访问函数 | pthread_attr_getstackaddr( )<br>pthread_attr_setstackaddr( ) |

### （3）离合状态

作用：线程离合状态，合态（PTHREAD_CREATE_JOINABLE）或者离态（PTHREAD_CREATE_DETACHED）。为合态时，线程创建新线程将使其自身被阻塞直到新线程退出。

如果线程以合态创建，则线程可以调用 pthread_detach( )使其进入离态；反之不行。当线程处于离态时，以其线程 ID 调用 pthread_detach( )和 pthread_join( )将失败，属性如表 3-6 所示。

表 3-6 离合状态的属性

| 允许值 | PTHREAD_CREATE_JOINABLE<br>PTHREAD_CREATE_DETACHED |
|---|---|
| 默认值 | PTHREAD_CREATE_JOINABLE |
| 访问函数 | pthread_attr_getdetachstate( )<br>pthread_attr_setdetachstate( ) |
| 动态访问 | pthread_detach( ) |

### （4）竞争范围

作用：线程竞争 CPU 资源范围，系统范围内（PTHREAD_SCOPE_SYSTEM）竞争或者进程范围内（PTHREAD_SCOPE_PROCESS）竞争，属性如表 3-7 所示。

表 3-7 竞争范围的属性

| 允许值 | PTHREAD_SCOPE_SYSTEM<br>PTHREAD_SCOPE_PROCESS |
|---|---|
| 默认值 | PTHREAD_SCOPE_SYSTEM |
| 访问函数 | pthread_attr_getscope( )，pthread_attr_setscope( ) |
| VxWorks 限制 | 只实现了 PTHREAD_SCOPE_SYSTEM |

### （5）继承调度

作用：决定创建线程时是从父线程继承调度参数（PTHREAD_INHERIT_SCHED）还是显式地指定（PTHREAD_EXPLICIT_SCHED），属性如表 3-8 所示。

表 3-8　继承调度的属性

| | |
|---|---|
| 允许值 | PTHREAD_EXPLICIT_SCHED<br>PTHREAD_INHERIT_SCHED |
| 默认值 | PTHREAD_INHERIT_SCHED |
| 访问函数 | pthread_attr_getinheritsched( )<br>pthread_attr_setinheritsched( ) |

**（6）调度策略**

如果继承调度为 PTHREAD_EXPLICIT_SCHED，该参数指定新创建线程的调度策略，SCHED_FIFO 表示强制优先级调度，SCHED_RR 表示 round-robin 优先级调度，属性如表 3-9 所示。

表 3-9　调度策略的属性

| | |
|---|---|
| 允许值 | SCHED_FIFO<br>SCHED_RR |
| 默认值 | SCHED_RR |
| 访问函数 | pthread_attr_getschedpolicy( )<br>pthread_attr_setschedpolicy( ) |

**（7）调度参数**

如果继承调度为 PTHREAD_EXPLICIT_SCHED，该参数指定新创建线程的调度参数。目前调度参数即线程优先级。

线程创建后，调度参数允许动态修改，属性如表 3-10 所示。

表 3-10　调度参数的属性

| | |
|---|---|
| 允许值 | 0~255 |
| 默认值 | 为 taskLib 指定的默认任务优先级 |
| 访问函数 | pthread_attr_getschedparam( )<br>pthread_attr_setschedparam( ) |
| 动态访问 | pthread_getschedparam( )<br>pthread_setschedparam( )<br>sched_getparam( )<br>sched_setparam( ) |

## 2. 创建线程

pthread_create( )创建一个 POSIX 线程。线程属性通过 pthread_attr_getXXX 指定，每种属性都有其默认值。下面具体来看 pthread_attr_t 定义的线程属性。

```
#include "pthread.h"
int pthread_create (pthread_t *pThread, const pthread_attr_t *pAttr,
   void * (*start_routine)(void *), void *arg);
```

pThread 存放创建线程得到的线程 ID。pAttr 表示新线程属性，包括多项内容，下面进行介绍。POSIX 允许程序指定 pAttr 为 NULL 来使用默认属性设置。start_routine 表示新线程入口函数，arg 为其惟一入参。start_routine 返回时，新线程结束，相当于隐式地以 start_routine 返回值作入参调用 pthread_exit( )。

pthread_create( )创建线程成功时返回 0，失败时分两种情况：没有足够的资源（EAGAIN）和参数无效（EINVAL）。

线程创建时通过属性对象为新线程赋予各种属性。

```
#include "pthread.h"

pthread_attr_t attr;          /*定义一个属性对象*/
pthread_attr_init(&attr);     /*初始化属性对象（装入默认值）*/
...  /*调用相关 POSIX 函数，设置需要的属性参数*/
pthread_create(..., &attr, ..., ... ); /* 创建 POSIX 线程*/
...  /*线程创建后通过动态访问函数读取或者修改 POSIX 线程属性*/
```

pthread_create( )创建的新线程的阻塞信号集继承自创建者线程；新线程的挂起信号集为空。

### 3. 线程 ID

POSIX 定义线程属于进程内的一条执行流程，线程 ID 仅仅在所属进程内定义，在系统范围内使用线程 ID 没有定义。在 VxWorks 中，线程属于系统，因此线程 ID 使用范围包括整个系统。

线程 ID 定义为 pthread_t 型变量。线程创建者在线程创建时得到所创建的线程 ID；线程可以通过函数 pthread_idself ( )得到其自身 ID。

POSIX 还定义了 pthread_equal( )用于比较两个线程：如果 t1 和 t2 表示同一线程，该函数返回非零值；否则返回零值。

```
#include "pthread.h"
pthread_t pthread_idself ( void );
pthread_t pthread_equal ( pthread_t t1, pthread_t t2 );
```

## 3.4.2 动态库初始化

程序往往有这种需要：使用的动态库需要在第一次使用时初始化，并且初始化函数被执行一次且仅执行一次。考虑下面的一个例子：

```
static int libIsInitialized = 0;
extern int libInitialization ( );
int some_function( )
{
    if (libIsInitialized == 0) {
        libInitialization ( );
        libIsInitialized = 1;
    }
    ...
}
```

在单线程时，上面的代码只通过一个标志变量 libIsInitialized 就实现了只运行一次的要求。但是在多线程环境下，显然需要对 libIsInitialized 进行互斥访问，否则竞争历险将使程序行为不可确定。

容易想到通过信号量实现对 libIsInitialized 的互斥访问。但是问题在于：信号量本身也需要调用函数初始化，且同样需要保证仅仅一次初始化。由此带来一个没有结束条件的递归问题。

下面的 POSIX 函数 pthread_once( )提供了一个灵巧的解决办法。当某个动态库为多个线程所用时，确保该动态库的初始化函数被执行一次且仅执行一次。

动态库初始化由 pthread_once( )实现：

```
#include "pthread.h"
pthread_once_t onceControl = PTHREAD_ONCE_INIT;
int pthread_once ( pthread_once_t * onceControl, void (* initFunc)(void) );
```

pthread_once( )的逻辑很简单，任何线程使用相同的变量 onceControl 调用一初始化函数 initFunc，则仅仅有第一次调用将发生实际对 initFunc 的调用，其他时候函数直接返回。调用初始化函数时不为 initFunc 传递任何参数。pthread_once( )内部通过“测试一锁”指令对 onceControl 进行访问。

pthread_once( )只观察 onceControl 状态而决定对 initFunc 的调用，而不论 initFunc 是否和以前的初始化函数相同。

VxWorks 的 pthread_once( )实现只返回 0 表示成功。

**注意：**

onceControl 不能在某个线程栈上分配，否则 onceControl 不能维护对初始化函数的调用状态；

onceControl 必须在首次使用前初始化为 PTHREAD_ONCE_INIT。

## 线程撤销的影响

pthread_once( )本身不是撤销点，但是如果 initFunc 表示的初始化函数是一个撤销点，

并且线程在调用 initFunc 期间被撤销，则 pthread_once( )对 once_control 的影响如同没有进行该次调用。这样能够保证初始化函数完整运行一次。线程撤销和撤销点在 3.4.6 小节“线程撤销”进行介绍。

## 3.4.3 线程私有数据

简单地讲，**线程私有数据**就是对于多个线程而言具有相同的名字，但是其值却相互独立的变量，即变量值特定于不同线程，也称为**特定线程数据**（Thread-Specific Data，TSD）。从这一点讲，线程私有数据和第 2 章介绍过的任务变量相似。

TSD 使多线程环境下的每个线程可以维护一个不受其他线程影响的全局变量；否则，线程要么使用局部变量（这使得变量无法跨越函数进行访问），要么使用不同名称的全局变量，或者退回到单线程环境。

POSIX 标准没有提供对其他线程的 TSD 进行访问的能力，也就是说任何线程只能访问其自身的 TSD。相比之下，VxWorks 提供对其他任务的任务变量的访问，只要知道目标任务 ID。

TSD 通过“密钥”进行标识，或者说密钥为“TSD 密钥”，也就是前面提到的线程私有数据的“名字”。POSIX 将密钥定义为 pthread_key_t，并提供初始化函数。来自不同线程的所有对 TSD 的读写访问都根据 TSD 密钥这一惟一标识进行。对应用程序而言，系统如何根据调用线程不同而实现对对应 TSD 的访问是不透明的，线程私有数据（TSD）如图 3-1 所示。

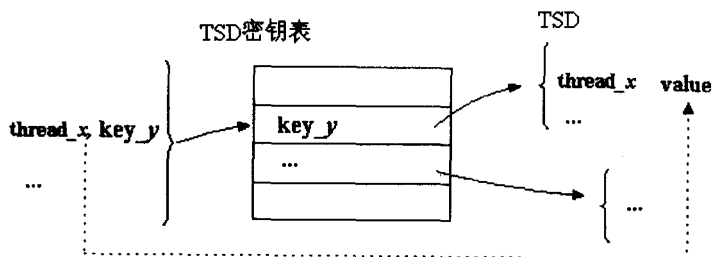



图 3-1　线程私有数据（TSD）

可以认为，一个密钥对应各个线程的 TSD 的副本；线程通过密钥对其 TSD 副本的读（写）操作将得到（设置）一个 void 型指针。通常该指针是一个整型值，或者指向一个代表更复杂的数据结构的用户分配的缓冲区。

POSIX 定义密钥可以为与初始化密钥的线程同一进程的所有线程使用；VxWorks 中密钥使用范围是整个系统。

下面先介绍密钥清除函数，然后介绍 POSIX 定义的 TSD 操作，包括：初始化密钥，密钥删除，读 TSD，写 TSD。

## 1. 清除函数

密钥清除函数（Destructor）一个典型的用途就是释放内存，尤其是 TSD 表示一个用户分配的缓冲区时。

一个密钥惟一对应一个密钥清除函数，该函数被所有线程共享。密钥清除函数必须在初始化密钥时指定，初始化密钥后无法更改。如果不需要，可以指定清除函数为 NULL。

密钥清除函数在满足下列两个条件时由系统自动进行调用：

- 设置了清除函数（清除函数非 NULL）；
- 线程结束（包括线程被取消）并且属于该线程的 TSD 非 NULL。

调用清除函数时，系统为其传递的参数即为该线程的 TSD 值。当系统需要调用多个清除函数时，程序不应该对清除函数调用顺序作任何假设。

清除函数具有如下原型定义：

```
void func_name ( void * param );
```

密钥清除函数常常用于清除被结束线程对应的 TSD（所表示的缓冲区）。密钥本身不会由系统自动删除，因为可能还有正在运行的线程需要该密钥，错误的删除会导致其他线程无法访问 TSD。

下面是一个密钥删除函数的例子，在函数中除了释放线程 TSD 对应的缓冲区，同时还判断如果所有线程都已经结束，则将密钥本身也删除。

```
static void key_destructor ( void * value )
{
    free ( value );    /*释放线程 TSD 表示的缓冲区*/

    pthread_mutex_lock ( &key_mutex );
    key_counter--;
    if (key_counter <= 0)    /*判断释放所有线程结束*/
    {                        /*是则删除密钥*/
        if (pthread_key_delete ( key )!= 0)
            ...
    }
    pthread_mutex_unlock ( &key_mutex );
};
```

key 表示 TSD 密钥。任意时刻，key_counter 表示 key 上所有线程 TSD 非 NULL 的线程计数。我们对 key_counter 通过 key_mutex 进行互斥访问。

密钥删除需要小心进行。上例中，我们定义 key_counter 为 TSD 非 NULL 的线程数，实际上存在这一问题：在某时刻 key_counter 变为零并且函数删除了密钥 key，但是另一线程（并不知道 key 被删除）可能此后会引用 key，这样程序运行结果将是不确定的。程序

员必须确保没有线程会在一个已经被删除的密钥上操作。如果将 key_counter 定义为所有可引用 key 的线程数，也存在问题。在 VxWorks 中，线程属于整个系统，因此可以认为任何密钥永远可能被某个线程访问。

## 2. 初始化密钥

```
#include "pthread.h"
int pthread_key_create ( pthread_key_t * pKey, void (* destructor) (void * ) );
```

函数 pthread_key_create( )初始化一个新密钥。参数 pKey 指向一用户分配的结构体 pthread_key_t，用于存放新密钥。如果已经达到系统允许的密钥数上限，函数失败并返回 EAGAIN；成功时返回 0。参考后面给出的例子。

对于当前系统中各个线程，根据新初始化密钥访问得到的 TSD 的值都为 NULL。

可以通过参数 destructor 为新密钥指定一个清除函数；如果不需要，为该参数传递 NULL。

一个密钥只能初始化一次。例如上面的函数成功时 pKey 表示的位置将得到一个密钥，如果以参数 pKey 再次初始化密钥，则原密钥将丢失，原密钥所对应的各个线程的 TSD 都将无法访问。保证只初始化一次的一种方法包括在一个“主线程”中初始化密钥，其他线程只进行读（写）TSD 的操作。另一个更“POSIX”的方法是使用 pthread_once( )，我们在后面例子中给出。

pKey 不应该指向一个栈上地址，否则其他线程无法访问。

## 3. 写 TSD

```
#include "pthread.h"
int pthread_setspecific ( pthread_key_t key, const void * value );
```

参数 key 为 pthread_key_create( )初始化得到的密钥；value 为要为调用线程设置的 TSD。成功时函数返回 0；否则返回 EINVAL（key 不代表一个有效密钥）或者 ENOMEM（没有内存空间）。

如果原来的 TSD 表示用户分配的缓冲区，则 pthread_setspecific( )设置新的 TSD 时应该注意将原缓冲区释放。

如果第一次调用，系统可能需要为线程分配 TSD 空间（参考图 3-1）。

## 4. 读 TSD

```
#include "pthread.h"
void * pthread_getspecific ( pthread_key_t key );
```

参数 key 是 pthread_key_create( )初始化得到的密钥。pthread_getspecific( )返回该线程通过 pthread_setspecific( )设置的 TSD。

新初始化的密钥对任何线程都返回 NULL。

& 读（写）TSD时的密钥必须是pthread_key_create( )得到的密钥，对于未初始化的密钥，读（写）TSD结果是不确定的，应注意避免。

## 5. 删除密钥

```
#include "pthread.h"
int pthread_key_delete ( pthread_key_t key );
```

pthread_key_delete( )删除密钥key。除非被再次初始化，任何线程不能继续根据key 进行TSD读写。

pthread_key_delete( )删除不会引起密钥清除函数的调用。但是密钥清除函数可以调用pthread_key_delete( )，如前面给出的例子。

如果调用pthread_key_delete( )时还有TSD指向的用户分配缓冲区，则应用程序应该自己保证将它们保证释放。

成功时pthread_key_delete( )返回0；否则返回EINVAL（参数key无效）。

下面给出一个简单的例子来说明上述各个函数用法。该例中TSD为用户程序定义的某数据结构SOME_VAR，线程结束时将对应缓冲区释放。

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;
void  key_destructor  ( void * mem )
{
    SOME_VAR * ptr = (SOME_VAR *)mem;
    ...  /*对数据结构的其他处理*/
    free( mem );
}
static void key_init( )   /*初始化密钥*/
{
    (void) pthread_key_create(&key, key_destructor );
}

thread_func( )   /*线程*/
{
    SOME_VAR * var;   /*SOME_VAR：某种数据结构定义*/
    (void) pthread_once(&key_once, key_init);
    if ( pthread_getspecific(key) == NULL )  /*对新密钥，该条件总应该成立*/
    {
        var = malloc( sizeof(SOME_VAR) );
        ...
        (void) pthread_setspecific(key, (void *)var);
    }
    ...
```

```
}
```

### 3.4.4 线程互斥与同步

VxWorks 提供和 POSIX 标准 1003.1c 兼容的互斥体和条件变量，在实现上采用互斥访问和二进制信号量。和 POSIX 线程一样，互斥体和条件变量也有其属性，并且也采取属性对象形式。

#### 1. 互斥体

互斥体的基本操作包括：锁、解锁和对 wind 互斥信号量的获取和释放有相似的语义。但是二者功能上存在一些差异。

**属性对象**

互斥体属性对象定义为结构体 pthread_mutexattr_t。VxWorks 5.5 的实现包括“协议”（Protocol）和“天花板”（Prioceiling）两个属性，都和线程在持有互斥体期间以何种优先级运行有关，因此也可将这两个属性称为“优先级协议”和“优先级天花板”。

POSIX 1003.1c 中定义的互斥体属性对象还包括“类型”，该属性决定当线程重复对一个互斥体加锁时的结果：死锁；错误检查；递归锁定。但是 VxWorks 目前没有实现。重复锁定在 VxWorks 下将导致死锁。

POSIX 定义这部分内容是为了解决优先级翻转问题，该问题已经在第 2 章中讨论过，我们这里只研究 POSIX 的特定实现。

使用属性对象之前，必须先初始化。POSIX 定义的互斥体属性对象的初始化和删除函数为：

```
#include "pthread.h"
int pthread_mutexattr_init ( pthread_mutexattr_t * pAttr );
int pthread_mutexattr_destroy ( pthread_mutexattr_t * pAttr );
```

参数 pAttr 都是指向用户分配的属性对象的指针。pthread_mutex_init( )为 pAttr 对象各个属性初始化 POSIX 定义的默认值；pthread_mutexattr_destroy( )功能相反，将各个属性设置为无效值。pthread_mutexattr_destroy( )不释放属性对象缓冲区，属性对象可以重新被初始化使用。

VxWorks 初始化的互斥体默认属性为：

- 互斥体协议　　优先级继承（PTHREAD_PRIO_INHERIT）；
- 互斥体天花板　0。

两个函数操作成功时返回 0；否则为 EINVAL（无效参数）。

对于新属性对象，应该先通过 pthread_mutex_init( )初始化，然后通过具体的各种属性定制函数设置需要的值；而不是直接定制各个属性。这样可以使程序能较好地适应未来可

能的标准升级。下面具体看对 POSIX 定义的互斥体“协议”和“天花板”两个属性。

**（1）协议**

协议属性定义互斥体如何处理优先级翻转的问题。在 VxWorks 中该属性有两种可能值，如表 3-11 所示：

**表 3-11　Vx Works 的属性**

| | |
|---|---|
| **优先级继承：**<br>PTHREAD_PRIO_INHERIT | 如果线程持有互斥体期间有比其优先级更高的线程在该互斥体上阻塞，则持有互斥体的线程将以被阻塞线程中最高的优先级运行；如果没有线程阻塞，任务将以其自身优先级运行 |
| **优先级保护：**<br>PTHREAD_PRIO_PROTECT | 线程持有互斥体期间，线程将以其自身优先级和该互斥体的优先级天花板中的较高者运行，而不论是否有其他高优先级线程阻塞在该互斥体上 |

设置协议为优先级继承时，即和 VxWorks 中对互斥信号量定义的优先级继承协议类似。

协议在互斥体属性对象初始化后通过下列函数设置和读取：

```
#include "pthread.h"
int pthread_mutexattr_setprotocol ( pthread_mutexattr_t * pAttr,
      int protocol );  /*设置互斥体对象协议*/
int pthread_mutexattr_getprotocol ( pthread_mutexattr_t * pAttr,
      int * pProtocol );  /*读取互斥体对象协议*/
```

为 pthread_mutexattr_setprotocol( )指定的参数 protocol 即为 PTHREAD_PRIO_INHERIT 或 PTHREAD_PRIO_PROTECT。

**注意：**

*互斥体协议只能在互斥体被初始化前在属性对象中确定，互斥体初始化后不能动态指定。*

**（2）天花板**

当互斥体协议为 PTHREAD_PRIO_PROTECT 时，“天花板”指定互斥体能提升持有线程的优先级到何种水平。如果持有线程优先级高于“天花板”，则线程优先级将不受影响。

天花板可以理解为互斥体所保护的代码部分执行时要求的最低优先级。该优先级的设置能控制线程持有该互斥体时完成动作的相对速度，其效果相当于控制交叉路口的宽度，但是实时系统环境也许比交通路口更复杂。

当互斥体协议为 PTHREAD_PRIO_INHERIT 时，天花板没有意义。

天花板在互斥体属性对象初始化后通过下列函数设置和读取：

```
#include "pthread.h"
int pthread_mutexattr_setprioceiling ( pthread_mutexattr_t * pAttr,
      int prioceiling );  /*设置互斥体对象天花板*/
```

```
int pthread_mutexattr_getprioceiling ( pthread_mutexattr_t * pAttr,
        int * pPrioceiling );  /*读取互斥体对象天花板*/
```

为 pthread_mutexattr_setprioceiling( )指定的参数 prioceiling 即为任意有效的 POSIX 线程优先级。要有效防止优先级翻转，应该设置天花板等于或高于所有可能使用互斥体的任务的最高优先级。

和协议属性不同，天花板可以在互斥体初始化后动态改变。

```
#include "pthread.h"
int pthread_mutex_setprioceiling ( pthread_mutex_t * pMutex,
    int prioceiling, int * pOldPrioceiling );  /*动态设置互斥体对象天花板*/
int pthread_mutex_getprioceiling ( pthread_mutex_t * pMutex,
    int *            pPrioceiling );  /*动态读取互斥体对象天花板*/
```

& 当线程持有多个互斥体时，线程实际运行的优先级将是根据各个互斥体的协议和天花板设置能够得到的最高优先级。线程释放某个互斥体后优先级又重新据此规则确定。释放了所有互斥体后，线程将恢复到其自身优先级。
优先级提示过程是传递的：如果线程 thread1 从其持有的互斥体 mutex1 提升了自身的优先级，在释放 mutex1 之前因试图获取另一互斥体 mutex2 而不阻塞，则持有 mutex2 的线程 thread2 的优先级将根据 mutex2 的属性和 thread1 提升后的优先级重新确定。

**互斥体初始化**

从数据结构讲，互斥体定义为结构体 pthread_mutex_t。POSIX 定义了两种初始化互斥体的方式：编译时用宏初始化和运行时函数初始化，两种方式初始化后得到的互斥体在使用上是一样的。

VxWorks 的互斥体通过一个信号量实现，支持第二种初始化方式，初始化函数将创建该信号量。

**（1）宏 PTHREAD_MUTEX_INITIALIZER**

静态分配存储互斥体可以在定义时通过宏 PTHREAD_MUTEX_INITIALIZER 初始化，例如：

```
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

PTHREAD_MUTEX_INITIALIZER 在“pthread.h”中定义。宏初始化有下面的限制：

- 只能初始化静态分配存储的互斥体，包括显式以关键字 static 声明，以及全局变量；
- 得到的互斥体属性为默认属性。

**（2） 函数 pthread_mutex_init( )**

```
#include "pthread.h"
```

```
int pthread_mutex_init ( pthread_mutex_t * pMutex,
       const pthread_mutexattr_t * pAttr );
```

参数 pMutex 指向用户分配的未初始化的互斥体，新初始化的互斥体的属性由属性对象 pAttr 指定，如果 pAttr 指定为 NULL，新互斥体将默认属性初始化。

如果 pMutex 指向一个已经被初始化的互斥体，函数的行为是不确定的：一种结果是 pthread_mutex_init( )返回 EBUSY，一种结果是重新初始化得到一个新互斥体。

同时要注意避免出现多线程重复初始化的情形。如果存在“主线程”，可以让主线程初始化；否则可以使用 POSIX 动态库初始化机制，即每次访问互斥锁时执行类似下面的操作：

```
pthread_once(&once, mutex_init);      /*动态初始化*/
pthread_mutex_lock(&mutex);           /*锁定互斥资源*/
    ...     /* 访问互斥资源 */
pthread_mutex_unlock(&mutex);         /*解锁互斥资源*/
```

上面的函数 mutex_init( )初始化互斥体 mutex：

```
void mutex_init( )
{
   pthread_mutex_init(&foo_mutex, NULL);
}
```

显然，编译时通过宏初始化要使程序简单许多。比较两种初始化方式的效率需要考虑和体系特点相关的底层实现细节，但一般而言编译时初始化互斥体比 POSIX 动态库初始化要更高效，避免了每次访问互斥体时 pthread_once( )函数运行开销。

在 VxWorks 中，考虑系统的支持以及效率因素，最合适采取“主线程”动态初始化方式。下面要讲到的条件变量初始化存在同样问题。

**互斥体删除**

```
#include "pthread.h"
int pthread_mutex_destroy ( pthread_mutex_t * pMutex );
```

成功删除互斥体时函数返回 0；如果互斥体还被某线程持有，函数立即返回 EBUSY。引用一个已经被删除的互斥体的结果是不确定的；被删除的互斥体重新被初始化后可以使用。

**锁互斥体**

```
#include "pthread.h"
int pthread_mutex_lock ( pthread_mutex_t *pMutex );
```

如果 pMutex 没有被任何线程锁定，函数 pthread_mutex_lock( )将得到该互斥体，即将其锁定；如果函数 pMutex 已经被其他线程锁定，线程将被阻塞，直到互斥体被当前的拥有者释放。

VxWorks 5.5 不支持线程递归锁定，也不进行此类错误检查。因此，如果调用线程当前已经是 pMutex 的拥有者，则该次重复锁定将使调用线程死锁。

如果锁定成功，函数返回 0，此时 pMutex 的拥有者为调用者线程。错误时返回 EINVAL（pMutex 不是一个有效互斥体）。

如果要避免调用者被阻塞，可以考虑下面的函数：

```
#include "pthread.h"
int ( pthread_mutex_t *pMutex );
```

函数 pthread_mutex_trylock( )和 pthread_mutex_lock( )一样，差别在于：如果 pMutex 已经被锁定（包括任何线程），该函数不阻塞，而是立即返回 EBUSY。

**解锁互斥体**

```
#include "pthread.h"
int pthread_mutex_unlock ( pthread_mutex_t *pMutex );
```

pthread_mutex_unlock( )释放 pMutex 指向的互斥体。如果有线程在互斥体上阻塞，则由调度策略决定系统选择哪个线程得到互斥体。

如果 pthread_mutex_unlock( )的调用线程不是互斥体的拥有者，函数返回 EPERM。

## 2. 条件变量

条件变量是 POSIX 定义的另一种同步机制。一个条件变量代表某种条件，允许线程阻塞直到该条件满足。

在条件变量上有两种基本操作：

- 等待条件（变量）pthread_cond_wait：一个线程因等待某种条件成立而阻塞。该线程称为等待线程；
- 触发条件（变量）pthread_cond_signal：一个线程使条件成立并给出信号将阻塞者唤醒。该线程称为触发线程。

还可以在条件变量上进行“广播”pthread_cond_broadcast，即唤醒所有在条件变量上阻塞的线程。

可以从多方面理解条件变量：

- 从“生产者－消费者”角度看，等待线程为消费者，触发线程为生产者；
- 从操作语义看，“等待条件”即等待线程在条件变量上阻塞，然后触发线程“触发条件”的动作将其解除阻塞。条件变量本身可以抽象地看成只是一个阻塞队列，之所以是“条件”就在于从“等待”到“触发”的过程中，生产者线程生产出了消费者线程等待的条件（比如将要求的数据在缓冲区中准备好）；
- 从 VxWorks 内部实现机制看，条件变量采用信号量实现。等待线程实际上阻塞在内部信号量上，触发线程一次解除信号量上一个或所有阻塞线程。POSIX 标准在表述时使用了术语“signal”，但不属于第 4 章将要讲到的“信号”。因此，这里我

们以“触发”界定“signal”。

条件变量总是和一个互斥体关联，互斥体保证所有线程等待该条件变量时的互斥访问。其过程如图 3-2 所示：

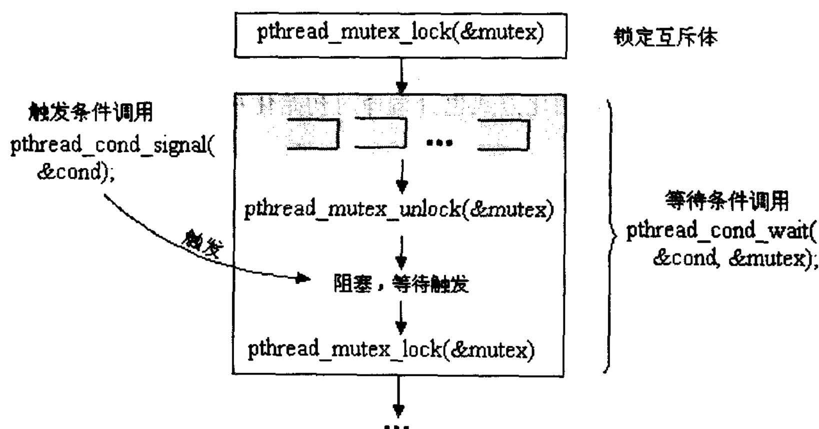



图 3-2 条件变量

（1）等待线程锁定互斥体，调用 pthread_cond_wait( );

（2）pthread_cond_wait( )将调用线程加入到条件变量的阻塞队列，解除互斥体锁定，然后阻塞；

（3）触发线程调用 pthread_cond_signal( )唤醒一个 pthread_cond_wait( );

（4）pthread_cond_wait( )重新锁定互斥体后返回。

互斥体用于使多线程同时等待一个条件变量时，确保只有一个线程能操作图 3-2 中的阴影部分（参考[Tan01]中“管程”的概念）。在线程通过阴影部分后，为了使其他线程在该线程阻塞期间能进入条件变量，pthread_cond_wait( )中解除对互斥体锁定，pthread_cond_wait( )结束时恢复对互斥体锁定。

上面过程的具体细节将在后面“等待条件”和“触发条件”部分介绍。先看条件变量的属性以及条件变量的初始化（即创建条件变量）。

### 属性对象

条件变量的属性对象比较简单，POSIX 定义了一个共享属性，控制进程间对条件变量的访问。VxWorks 中不存在进程，因此条件变量属性对其起始是一个空壳。

数据结构定义为结构体 pthread_condattr_t。VxWorks 中实现了属性对象初始化和删除函数：

```
#include "pthread.h"
int pthread_condattr_init ( pthread_condattr_t * pAttr );  /*初始化属性对象*/
int pthread_condattr_destroy ( pthread_condattr_t * pAttr );/*删除属性对象*/
```

上述函数在 VxWorks 没有实质功能，只设置了 VxWorks 特定的一些标志。如果使用了属性对象，程序应该通过上述函数初始化条件变量，以及在不需要使用时删除条件变量，以保持对未来系统升级的支持。

**初始化条件变量**

POSIX 定义条件变量的初始化方式也分编译时初始化和运行时初始化。VxWorks 支持第二种初始化方式。

**（1）宏 PTHREAD_COND_INITIALIZER**

对于静态分配存储的条件变量，定义时可以用宏 PTHREAD_COND_INITIALIZER 初始化，如：

```
static-pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

PTHREAD_MUTEX_INITIALIZER 在“pthread.h”中定义。

**（2）函数 pthread_cond_init( )**

```
#include "pthread.h"
int pthread_cond_init ( pthread_cond_t * pCond, pthread_condattr_t * pAttr );
```

参数 pCond 指向用户分配的未初始化的条件变量，新条件变量的属性由属性对象 pAttr 指定，如果 pAttr 指定为 NULL，新条件变量将已默认属性初始化。

如果 pMutex 指向一个已经被初始化的条件变量，函数的行为是不确定的：一种结果是 pthread_cond_init ( )返回 EBUSY，一种结果是重新初始化得到一个新条件变量。

**删除条件变量**

```
#include "pthread.h"
int pthread_cond_destroy ( pthread_cond_t * pCond );
```

成功时函数返回 0；如果条件变量还被某线程持有，函数立即返回 EBUSY。引用一个已经被删除的条件变量的结果是不确定的；被删除的条件变量可以重新被初始化后使用。

**等待条件**

等待线程通过下面的调用来等待某种条件满足：

```
#include "pthread.h"
int pthread_cond_wait ( pthread_cond_t * pCond,
        pthread_mutex_t * pMutex );
int pthread_cond_timedwait ( pthread_cond_t * pCond,
        pthread_mutex_t * pMutex, const struct timespec * pAbstime );
```

上面两个函数用于使调用线程在条件变量 pCond 上阻塞，调用之前线程必须先锁定互

斥体 pMutex，pMutex 用于进行多线程同时试图等待一个条件时的互斥。

当等待的条件被触发后，函数返回 0。两个函数内部都经历了一个“解锁——阻塞——重新加锁”的过程。正确的情况下，函数返回到调用线程时，调用线程仍然持有 pMutex，因此必须在上面的函数返回后解锁 pMutex。参考图 3-2。

pthread_cond_wait( )将一直等待条件直到条件被触发；pthread_cond_timedwait( )多了超时控制，如果到了指定时间但条件还没有被触发，函数将返回 ETIMEDOUT。超时参数 pAbstime 对时间的表示和 VxWorks 库函数表示不同：timespec 表示从系统时钟得到的绝对时间（秒+纳秒）；而 VxWorks 库函数中一般以 tick 为单位，而且是从调用时刻开始的相对时间。

对于相同的条件变量，所有线程都必须通过相同的互斥锁互斥，否则程序得到的结果将是不确定的。

**撤销点影响**

pthread_cond_wait( )和 pthread_cond_timedwait( )都属于撤销点调用，如果设置撤销类型为延迟撤销并且有撤销请求到来，则线程将：（1）从阻塞进入运行状态；（2）重新锁定互斥体；（3）离开 pthread_cond_wait 或 pthread_cond_timedwait；（4）执行取消动作。也就是说如果 pthread_cond_wait()被取消，mutex 是保持锁定状态的，需要线程清除函数实现解锁。

被撤销线程不会消耗一个条件。

### 触发条件

触发条件将使等待条件的线程从条件变量上解除阻塞。触发条件分两种：单一触发和广播触发。

**（1）单一触发**

```
#include "pthread.h"
int pthread_cond_signal ( pthread_cond_t * pCond );
```

pthread_cond_signal( )解除 pCond 上一个等待线程阻塞。如果没有线程在阻塞，该次调用不进行任何动作。

如果有多个等待线程阻塞，系统将根据调度策略选择一个线程解除阻塞。

**（2）广播触发**

```
#include "pthread.h"
int pthread_cond_broadcast ( pthread_cond_t * pCond );
```

和 pthread_cond_signal( )类似，差别在于一次解除 pCond 上所有阻塞的线程。

**触发条件**

触发线程进行触发调用（pthread_cond_signal 或者 pthread_cond_broadcast）之

前不需要锁定等待线程在等待条件时使用的互斥体，但是可以这样做以控制线程调度顺序。
如果广播触发解除了多个阻塞的等待线程，则这些线程回到调用者之前将竞争以重新锁定互斥体。调度策略决定其中的顺序。

## 3.4.5 线程结束

```
#include "pthread.h"
void pthread_exit ( void * status );
```

线程结束即线程显式或者隐式调用 pthread_exit( )，status 为线程结束码，通常表示一个整数，也可以指向一个更复杂的数据结构。线程结束码将被另一个与该线程进行了结束同步的线程得到（后面介绍）。

线程结束具体分 3 种情况：

（1）在线程任何地方显式调用 pthread_exit( )；

（2）线程主入口函数运行完毕，隐式调用 pthread_exit( )，退出码为函数返回值；

（3）线程被撤销，隐式调用 pthread_exit，退出码为 PTHREAD_CANCELED。

线程撤销比较复杂，专门开辟一小节介绍。在 3 种情况下，线程结束时都将先后进行下面两项动作：

① 调用线程清除函数，如果有多个，按照与压栈顺序相反的顺序调用；

② 调用 TSD 清除函数，如果有多个，不确定调用顺序。

### 1. 线程清除函数

POSIX 对每个线程规定了一个清除栈（Cleanup Stack），其中以类似堆栈的方式记录一个或者多个线程清除函数，在线程结束时由系统自动调用。线程清除函数可以用来清除线程状态，如关闭打开的文件、释放内存等，清除栈如图 3-3 所示。

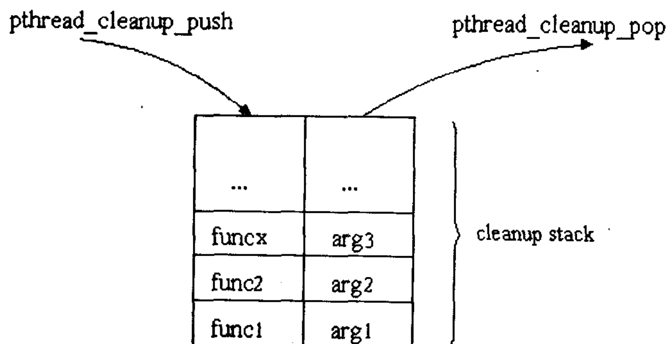



图 3-3 清除栈

POSIX 定义的压栈函数和出栈函数为：

```
#include "pthread.h"
void pthread_cleanup_push( void (* routine)(void * ),void * arg );/*压栈*/
void pthread_cleanup_pop ( int run );                           /*出栈*/
```

为压栈函数 pthread_cleanup_push( )指定的参数为线程清除函数 routine 及其惟一参数 arg。

出栈函数 pthread_cleanup_pop( )被调用一次将弹出栈顶一个清除函数及其参数。如果指定 pthread_cleanup_pop( )参数 run 非零，则被弹出的清除函数将被运行。

pthread_exit( )时自动以非零入参调用 pthread_cleanup_pop( )直到栈空。程序可以设计在线程运行期间，也加入出栈过程，以实现对线程撤销的保护，如图 3-3 所示。实际上，线程清除函数主要目的正式在线程被撤销后进行环境清理，因此，线程清除函数又称为**线程撤销清除函数**（Thread Cancellation Cleanup Handler)。参考 3.4.6 小节对线程撤销的讨论。

线程清除具有如下原型定义：

```
void func_name ( void * arg );
```

该函数具有惟一参数 arg，一般表示一个整型值或者为指向一个更复杂数据结构的指针。如图 3-3 所示，在线程清除函数栈中直接记录了该参数，在线程清除函数被弹出运行时系统将记录的参数传递给它。

线程清除函数必须返回到系统，注意避免在其内部使用 longjmp( )和 siglongjmp( )。

## 2. 线程结束同步

线程结束同步即一线程等待目标线程结束，相当于使两个线程（注意只能是两个）串行化。线程结束同步时目标线程必须处于合态，如图 3-4 所示。

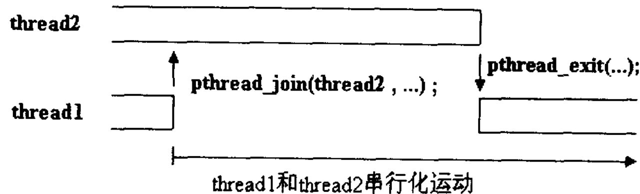



图 3-4 线程结束同步

需要同步时，线程调用 pthread_join( ):

thread1 和 thread2 串行化运动

```
#include "pthread.h"
int pthread_join ( pthread_t thread, void **ppStatus );
```

目标线程由参数 thread 表示。调用线程将被阻塞，直到目标线程结束；或者目标线程结束时立即返回。参数 ppStatus 非 NULL 时，用于接收目标线程结束时的状态，该状态来自目标线程显式或者隐式调用 pthread_exit( )的退出状态，或者退出状态由

PTHREAD_CANCELED 表示目标线程被撤销。

返回结果：

- 不允许线程等待自身，即目标线程为线程自身时，函数立即返回 EDEADLK 错误；
- 如果目标线程不存在，函数立即返回 ESRCH；
- 如果目标线程以离态创建，则函数立即返回 EINVAL。

& POSIX 未定义多个线程等待一个目标线程结束时的结果；VxWorks 对此有明确定义：一个目标线程上只能有一个线程等待，如果调用 pthread_join( )时目标线程已经有线程等待，则调用立即返回 EINVAL 表示错误。

对目标线程来说，如果要避免线程结束同步，可以将其离合状态设置离态。离态意味着线程结束后内存资源可以被系统回收。

pthread_detach( )将线程设置为离态：

```
#include "pthread.h"
int pthread_detach ( pthread_t thread );
```

使用线程结束同步一个常见的例子是一个主线程创建多个子线程协同进行计算，子线程使用通过主线程初始化的某些资源，如用于子线程间同步的信号量，所有子线程运行结束后，主线程释放资源并退出。例如：

```
void * sub_func( void* arg );   /*子线程函数*/

main_func( )            /*主线程函数*/
{
    ...
    pthread_t sub_thr[NUM_OF_SUBS];
    void * sub_status[NUM_OF_SUBS];
    ...  /*为子线程分配资源*/

    for ( i=0; i<NUM_OF_SUBS; i++ )      /*创建子线程*/
        pthread_create ( &sub_thr[i], NULL, sub_func, arg_for_sub );

    for ( i=0; i<NUM_OF_SUBS; i++ )      /*子线程结束同步*/
        pthread_join ( sub_thr[i], &sub_status[i] );

    ...  /*释放资源及其他*/
}
```

### 3.4.6　线程撤销

#### 1. 撤销概述

在编程实践中经常遇到线程完成预期的工作之前结束线程，比如多个线程在搜索一个迷宫，其中一个线程找到了出口，则该线程将结束其他线程，该动作在POSIX中被称为**撤销**（Cancellation），在Win32中被称为**终止**（Termination）。在VxWorks中，类似的概念是**任务删除**（Taskddelete）。

正确取消一个线程要确保该线程能够释放其所持有的任何锁、分配的内存，使整个系统保持一致性。在很多复杂情况下要保证这种正确性是有一定困难的。

一种简单的线程撤销：撤销线程调用一个撤销线程的函数，被撤销线程死亡。在这种情况下，被撤销线程所持有的资源得不到释放。撤销线程负责保证被撤销者处于可以安全撤销状态，在一个要求可靠性高的系统中，这种保证非常困难或者无法实现。这种撤销称为不受限制的**异步撤销**。

与异步撤销相关的一个术语是**异步撤销安全**，即一段代码执行期间可以在任意点上被撤销而不会引起不一致。满足该条件的函数称为**异步撤销安全函数**。显然，异步撤销安全函数不涉及使用互斥锁，也不动态分配内存。POSIX标准要求这些函数是异步撤销安全函数：pthread_cancel( )，pthread_setcancelstate( )，pthread_setcanceltype( )。

POSIX标准定义了一种更安全的线程撤销机制。一个线程可以以可靠的受控制的方式向进程的其他任何任务发出撤销请求，目标线程可以挂起这一请求使实际的撤销动作在此后安全的时候进行，称为**延迟撤销**（Deferred Cancellation）。目标线程还可以定义其被撤销后自动被系统调用的线程清除函数。和延迟撤销相关的一个概念是**撤销点**。

POSIX 通过为线程定义可撤销状态，撤销点函数以及一系列撤销控制函数实现延迟撤销。

#### 2. 可撤销状态

线程可撤销状态包括撤销允许状态和撤销类型选择，如表3-12所示。

**表3-12　线程可撤销状态**

| | |
|---|---|
| **撤销允许状态** | 撤销允许 PTHREAD_CANCEL_ENABLE<br>撤销禁止 PTHREAD_CANCEL_DISABLE |
| **撤销类型选择** | 异步撤销 PTHREAD_CANCEL_ASYNCHRONOUS<br>延迟撤销 PTHREAD_CANCEL_DEFERRED |

设置撤销允许状态：

```
#include "pthread.h"
```

```
int pthread_setcancelstate ( int state, int * oldstate );
```

通过为参数 state 指定 PTHREAD_CANCEL_ENABLE（PTHREAD_CANCEL_DISABLE）来允许（禁止）撤销。参数 oldstate 非 NULL 时返回原撤销允许状态。

当设置撤销允许状态为撤销禁止时，对该线程的撤销请求将被忽略。当设置线程为允许撤销时，如果收到撤销请求，则系统的动作由所选择的撤销类型决定：

- 异步撤销：撤销请求立即被执行；
- 延迟撤销：撤销请求被挂起，直到运行到一个撤销点才被执行。

线程撤销类型通过下列函数选择。

```
#include "pthread.h"
int pthread_setcanceltype ( int  type, int * oldtype );
```

为参数 type 指定的撤销类型可以为 PTHREAD_CANCEL_ASYNCHRONOUS 或者 PTHREAD_CANCEL_DEFERRED。参数 oldtype 非 NULL 时返回原撤销类型设置。

& pthread_setcancelstate( )和 pthread_setcanceltype ( )的操作具有原子性，为异步撤销安全函数。

### 3. 撤销点

在使用延迟撤销机制时，一个线程在可以被撤销的地方定义**撤销点**（Cancellation Point），当收到撤销请求时，被撤销的线程执行到撤销点时退出，或者在一个撤销点调用被阻塞时退出。通过延迟撤销，程序在进入临界区时不需要进行禁止/允许撤销操作。

由于在延迟撤销时必须在撤销点才能撤销约束，这一限制可能使撤销请求被挂起任意长的时间。因此，如果某个调用可能使线程被阻塞或者进入某个较长时间的过程，POSIX 要求这些调用属于一个撤销点，或者称这些调用为**撤销点调用**，以防止撤销请求陷入长时间等待。POSIX 要求属于撤销点调用的函数在“表撤销点调用”中列出。注意其中有些并不是调用线程阻塞。

撤销点调用列表：

```
aio_suspend        fcntl *                   sigtimedwait
close              pause                     sigwait
creat              pthread_cond_timedwait    sigwaitinfo
fsync *            pthread_cond_wait         sleep
mq_receive         pthread_join              system
mq_send            pthread_testcancel        tcdrain *
msync *            read                      wait *
nanosleep          sem_wait                  waitpid *
open *             sigsuspend                write
```

带“*”的撤销点调用在VxWorks中没有实现，因为这些VxWorks中没有这些函数。

撤销点使在延迟撤销方案下，撤销请求不致被挂起太长时间，但是没有考虑其他影响。因此，应用程序需要注意在其中一段含有撤销点调用的代码中，是否有内存申请以及锁共享资源的情况。例如，对下面这段代码，如果撤销请求在free( )之前到来，撤销点调用sem_wait( )有可能使malloc( )分配的内存泄漏。

```
mem = malloc( SOME_MEMORY );
...
sem_wait(sem);    /*该函数将创建一个撤销点! */
...
free(mem);
```

笨拙的解决办法是禁止撤销。较好的解决办法是使用撤销清除函数。下面会看到在该段代码前后加上清除函数保护使得代码中使用的资源得以释放。

下面一个简单的查询方案可以帮助理解撤销点调用如何实现延迟撤销：线程T2通过调用pthread_cancel( )让系统在T1线程数据结构中设置一个变量，线程T1运行到一个撤销点调用时查询该变量，如果变量被设置则退出。

```
T2                  T1                        撤销点调用
cancel[T1] = TRUE;
...
pthread_testcancel ( );
... pthread_testcancel (...) {
    if (cancel[self])
        pthread_exit(
            PTHREAD_CANCELED
        );
}
```

可以看出，只有在撤销点调用处才可能退出，因此，要限制延迟撤销时等待时间，程序员必须保证执行某个长时间循环期间，撤销点调用不致太少。一个有效但影响效率的做法是每轮循环调用一次pthread_testcancel ( )。

一种不使用频繁的撤销点调用的方法是在异步线程撤销和延迟线程撤销之间切换，如图3-5所示。

在图3-5中，要求阴影部分代码是异步撤销安全。

**& 线程撤销**

如果线程在撤销点因为等待某种条件而阻塞期间发生撤销请求，线程将退出阻塞状态而执行撤销，即执行pthread_eixt(PTHREAD_CANCELED)，以及该调用引起的一系列过程。但是POSIX未定义当线程等待的条件也同时满足时，线程解除阻塞后是取得条件正常运行还是完成撤销动作。

如果线程在因等待条件变量而阻塞期间被撤销，系统在调用该线程第一个

撤销清除函数前该线程将重新锁定和条件变量关联的互斥体。

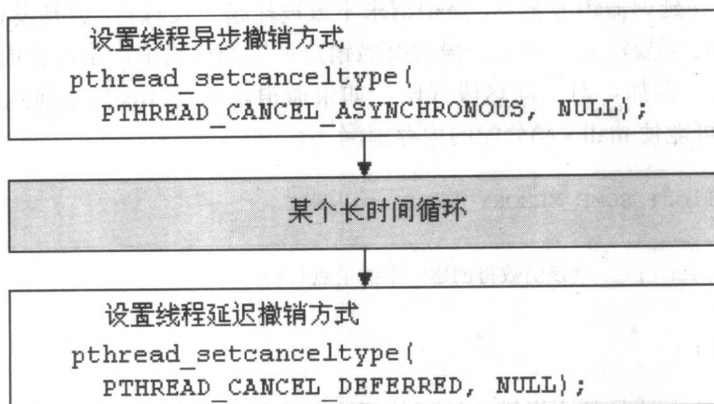



图 3-5 异步撤销

## 4. 撤销清除函数

线程被撤销时自动调用撤销清除函数（即线程清除函数），撤销清除函数实现对代码的撤销保护。如图 3-6 中，阴影部分代码需要分配和使用一些资源，如内存或锁互斥体等，在该部分代码期间需要允许延迟撤销。函数 func_free_resource( )负责释放所有使用的资源，将其作为撤销清除函数压栈和出栈实现了对阴影部分代码的撤销保护。

如果阴影代码部分被撤销，pthread_exit( )将自动弹出释放资源的函数运行；如果线程正常运行下来，则 pthread_cleanup_pop(1)将完成该动作。

撤销清除函数运行期间，系统将自动禁止线程撤销。因此撤销清除函数运行不必考虑线程撤销的影响。

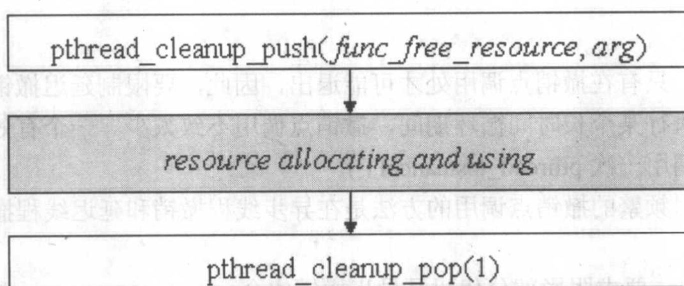



图 3-6 清除保护

## 5. 撤销实现

撤销一个线程调用 pthread_cancel( )实现：

```
#include "pthread.h"
```

```
int pthread_cancel ( pthread_t thread );    /*撤销线程*/
```

参数 thread 表示被撤销的目标线程。pthread_cancel( )和目标线程实际的撤销动作是异步的。根据目标线程设置，撤销请求可能被忽略、立即执行或者延迟处理。

VxWorks 5.5 中异步线程撤销实际上是通过一个特殊信号 SIGCANCEL 实现(参考第 4 章“信号”)。要求被撤销线程不能阻塞、忽略或者捕获该信号。

当延迟撤销时，目标线程往往通过函数 pthread_testcancel( )来检查是否需要结束自己的运行。pthread_testcancel( )和其他撤销点调用不同之处在于该函数除了创建一个撤销点以外什么也不做，即专门为响应撤销请求服务。

```
#include "pthread.h"
void pthread_testcancel (void);        /*创建一个撤销点*/
```

如果有挂起的线程撤销请求，pthread_testcancel() 不返回到调用者，而以 PTHREAD_CANCELED 为参数调用 pthread_exit( )使线程结束。否则函数返回，线程继续正常运行。

# 3.5 任务调度

## 3.5.1 概述

基于“多进程－多线程”的模型，POSIX 标准定义**调度**决定如何选择就绪的“进程”运行。在 VxWorks 中，问题变成了如何选择就绪“任务”运行，或者根据 VxWorks 5.5 对 pthread 的实现，如何选择就绪“线程”运行。和 VxWorks 习惯一致，我们使用“任务”进行表述。三个概念可以认为是一致的。

调度行为受两个因素影响：调度策略和任务优先级。概念上可以认为，每个任务都有一个优先级，系统为每个允许的优先级维护一个就绪任务列表，列表具有某种顺序，表首任务和表尾任务，如图 3-7 所示。

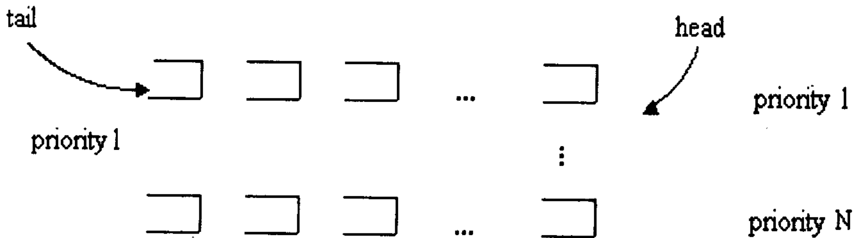



图 3-7 任务调度模型

POSIX 标准从对图 3-7 中列表的操作过程的定义来刻画调度策略，即在特定事件触发下，任务将（从运行，阻塞或者就绪等状态）被置于上述列表中何种位置。本质上，调度策略决定如何选择就绪任务进入运行。

对实时系统而言，调度策略对实时响应的影响是最重要的，通常实时事件对应于系统中的高优先级任务，因此，作为 POSIX 基本定义的实时扩展，POSIX 1003.1b 定义的调度策略都是基于优先级的。其他评价指标，如公平、有效、响应时间、周转时间，吞吐量则是次要的。

POSIX 要求兼容的操作系统至少实现 32 级优先级，VxWorks 实现了 256 级优先级，从 0～255。

& POSIX 标准规定优先级高的任务优先级数字大。VxWorks 原来使用的任务优先级表示与此相反，高优先级任务的优先级数字小，最高优先级任务优先级表示为 0。因此，schedPxLib 提供了两种优先级表示的支持，通过全局变量 posixPriorityNumbering 选择使用 POSIX 方式的优先级表示（为 TRUE）或者 VxWorks 的优先级表示（为 FALSE）。VxWorks 中其他库都惟一使用 VxWorks 的表示方法。

## 3.5.2 调度策略

VxWorks 支持两种 POSIX 调度策略：SCHED_FIFO（FIFO 调度）和 SCHED_RR（Round-Robin 调度）。不论采用哪种策略，调度程序都选择图 3-7 中非空的最高优先级列表的取表首任务运行，从这个意义上讲，高优先级任务总是优先运行。因此，VxWorks 调度属于优先级调度。前面已经说明，优先级调度是实时系统的必要条件之一。

& 同样属于优先级调度，POSIX 任务调度使用术语 **FIFO 调度**，wind 任务调度使用术语**可抢占优先级调度**，其内容是一致的，但是“FIFO 调度”强调具体对每个优先级列表的操作规则；“可抢占优先级调度”强调整体调度效果。
POSIX 的任务调度策略设置是逐步进程的，不同进程内的线程可以采取不同的调度策略；wind 的调度算法是整个系统范围内的，所有任务在 Round-Robin 算法或者可抢占优先级算法下竞争 CPU 资源。

### 1. SCHED_FIFO —— FIFO 调度

就绪列表上的任务根据未被执行的时间排序，表首任务最久未被执行。具体如何实现，POSIX 定义了下列情况下就绪列表被修改：

（1）如果任务被占先，该任务成为其优先级列表上的表首任务；

（2）如果阻塞中的任务解除阻塞，该任务加入到其优先级列表表尾。这里“阻塞”应该包括 VxWorks 中的“阻塞”，“延迟”，“挂起”3 种任务状态；

（3）如果运行中的任务调用 sched_setscheduler( )或者 sched_setparam( )，则指定的任

务（可能是调用任务自身）的调度策略和优先级根据参数指定被修改；

（4）不论什么时候修改运行任务或者就绪任务的优先级，被修改任务将成为其新优先级列表的表尾任务；

（5）如果运行中的任务调用 sched_yield( )，该任务将成为其优先级列表的表尾，即让出处理器。

实际上，根据 POSIX 定义实现的“多进程－多线程”模型将比 VxWorks 中单一任务模型复杂的多。简单给 VxWorks 带来高效调度的同时，也使有些调用被限制，例如 VxWorks 应用无法通过 sched_setscheduler( )修改调度策略。

## 2. SCHED_RR —— Round-Robin 调度

根据 FIFO 调度规则，如果不被占先，则阻塞，调用 sched_yield( )主动让出处理器或者改变其优先级和调度策略，运行的任务将一直独占处理器。当存在多个高优先级任务时，这使得其他高优先级任务得不到运行。

Round-Robin 调度和 FIFO 调度相比的差异在于 Round-Robin 调度增加了时间片控制，一个任务运行指定时间片长度后让出 CPU，调度程序选择下一个符合条件的任务运行。该时间片称为 **Round-Robin 时间片**，或者简称为**时间片**。

从队列操作角度精确定义，Round-Robin 调度在 FIFO 调度基础上增加了一种队列操作：对运行中的任务的执行时间进行计数，如果已经执行了一个时间片长的时间，该任务将被放到其所属优先级的就绪任务列表表尾并将其运行时间归零，同时取出该列表表首任务进入运行。同时队列上每个任务节点都增加了一个时间片属性：任务初始进入队列时该属性值为零；如果运行中的任务被占先，该属性值表示任务被占先时的时间片，任务恢复执行时在该时间片基础上计数。

如果采用 Round-Robin 调度策略，一个值得考虑的问题是时间片大小的确定：时间片小有利于同优先级任务公平共享处理器，但是显然增加了调度开销；较大的时间片使调度开销相对低，但时间片太大时调度效果趋向于 FIFO 调度。合理的时间片将是在公平和效率之间的折衷。[Tan01]建议将时间片设置为 100ms，但是该值是针对通用操作系统并考虑用户交互的情况下确定的。我们认为，对于实时嵌入式应用，可以根据经验并综合下面两个因素确定：

（1）上下文切换开销 $C$；

（2）FIFO 调度时上下文切换平均间隔 $V$。

$C$ 比较容易测量确定，而 $V$ 的确定则比较复杂，需要模拟系统运行加上经验分析。假定期望系统调度开销小于 $p$，则根据 $C$，时间片大小 $x$ 必须满足 $x \geq [(1-p)*C]/p$；然后根据 $V$ 调整 $x$。

& 可以看出，POSIX 定义的 SCHED_FIFO 和 SCHED_RR 都属于优先级调度，并且两种调度的差异仅仅在于后者在前者基础上增加了时间片控制。

“FIFO”在这里有些容易让人误解，它只表示任务在具体的就绪队列上以 FIFO 顺序排列，从整个系统（即所有就绪队列）上看，并非 FIFO。

### 3.5.3 调度实现

POSIX 定义结构体 sched_setparam 来表示和设置任务属性。目前任务属性仅仅表示优先级，结构 sched_setparam 的设计是为了未来扩展的需要。

```
#include "sched.h"
struct sched_param
{
   int sched_priority;     /*任务优先级 */
};
```

#### 1. 任务调度策略

POSIX 定义了函数动态选择进程的调度策略（SCHED_FIFO 和 SCHED_RR）：

```
#include "sched.h"
int sched_getscheduler ( pid_t tid );                  /*读取调度策略*/
int sched_setscheduler ( pid_t tid, int policy,
   const struct sched_param * param );                 /*设置调度策略*/
```

上述函数的参数中，tid 表示目标任务 ID，tid 为 0 时表示调用任务自身；policy 表示调度策略：SCHED_FIFO 或 SCHED_RR；param 执行任务属性结构体（含优先级定义）。

在 VxWorks 中，sched_getscheduler( )返回调度策略 SCHED_FIFO 或者 SCHED_RR，如果指定了错误的 tid 返回 ERROR。

需要注意的是 sched_setscheduler( )。由于 VxWorks 不允许为任务指定和系统不一致的调度策略。当 policy 指定的调度策略和 sched_getscheduler( )返回调度策略不同时，调用将失败。因此该函数实质上只能用于设置任务优先级。优先级设置还可以调用函数 sched_setparam( )完成。

在 VxWorks 中改变系统的优先级必须通过内核调用 kernelTimeSlice 完成：

```
#include "kernelLib.h"
STATUS kernelTimeSlice ( int ticks );                  /*设置调度策略*/
```

参数 ticks 是 VxWorks 中常用的计时单位，即时钟 tick 数。设置 ticks 为 0 将使系统调度策略设置为 SCHED_FIFO 调度；否则为一个大于零的数选择系统调度策略设置为 SCHED_RR 调度，同时该参数表示 Round-Robin 时间片长度。

一个时钟 tick 表示的时间长度决定于系统时钟溢出速率。通常情况下，时钟溢出率为 60Hz 时 1tick≈17ms。

## 2. 任务优先级

任务优先级允许动态设置和读取：

```
#include "sched.h"
/*读取优先级*/
int sched_getparam ( pid_t tid, struct sched_param * param );
/*设置优先级*/
int sched_setparam ( pid_t tid, const struct sched_param * param );
```

为上述函数指定的参数 tid 为目标任务 ID，0 表示调用者；param 表示含优先级定义的任务属性结构体。上述函数成功时返回 0，否则返回-1。

必须为 param 指定系统允许的优先级。POSIX 允许采用不同的调度策略时定义不同的优先级范围，并定义函数 sched_get_priority_max( )和 sched_get_priority_min( )得到系统支持的优先级范围。对于当前 VxWorks 的实现，两种协议支持的优先级范围都是 0~255。

```
#include "sched.h"
int sched_get_priority_max( int policy ); /*取指定调度策略允许的最高优先级值*/
int sched_get_priority_min( int policy ); /*取指定调度策略允许的最低优先级值*/
```

## 3. Round-Robin 时间片

kernelTimeSlice( )选择 SCHED_RR 调度同时指定了 Round-Robin 时间片大小。下列 POSIX 函数读取时间片大小。

```
#include "sched.h"
int sched_rr_get_interval ( pid_t tid, struct timespec * interval );
```

参数 interval 指向存储时间片的结构体 timespec。kernelTimeSlice( )指定的时间片为 tick 数，sched_rr_get_interval( )在结构体中返回以秒和纳秒表示的时间片大小。

只能在系统调度策略为 SCHED_RR 时调用 sched_rr_get_interval( )，否则函数失败。

下面这个示例程序测试 VxWorks 对调度的支持。该程序表明了如何控制系统的调度策略和优先级设置。程序先读取当前的调度设置，然后选择另一种调度策略，将优先级修改为最低优先级。

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

STATUS vxSched ( void )
{
    int i;
    int policy, pri;
    int fifo_min, fifo_max, rr_min, rr_max;
```

```
    struct sched_param param;
    struct timespec tm;

    /*判断允许的优先级范围*/
    fifo_min = sched_get_priority_min( SCHED_FIFO );
    fifo_max = sched_get_priority_max( SCHED_FIFO );
    rr_min   = sched_get_priority_min( SCHED_RR );
    rr_max   = sched_get_priority_max( SCHED_RR );
    printf("SCHED_FIFO priority range: %d ~ %d\n", fifo_min, fifo_max);
    printf("SCHED_RR priority range: %d ~ %d\n", rr_min, rr_max);

    /*读取当前调度设置*/
    if ( (policy=sched_getscheduler(0)) == -1 )
        return (ERROR);
    if ( sched_getparam ( 0, &param )==-1 )
        return (ERROR);
    printf("Current scheduling setup: %s, PRIORITY: %d\n",
      policy==SCHED_FIFO?"SCHED_FIFO":"SCHED_RR", param.sched_priority );
    if(policy==SCHED_RR)
    {    /*读时间片*/
        sched_rr_get_interval( 0, &tm );
        printf("RR time-slice: %d'%d''\n", tm.tv_sec, tm.tv_nsec );
    }

    /*修改调度设置*/
    switch ( policy )
    {
    case SCHED_FIFO:      /*选择 SCHED_RR 调度*/
        kernelTimeSlice( 10 );      /*注意不是 sched_setscheduler*/
        param.sched_priority = rr_min;
        if ( (sched_setparam( 0, &param )) == -1 )
            return (ERROR);
        break;
    case SCHED_RR:       /*disable*/
        kernelTimeSlice( 0 );
        param.sched_priority = fifo_min;
        if ( (sched_setparam( 0, &param )) == -1 )
            return (ERROR);
        break;
    default:
        break;
    }
```

```
    /*判断结果*/
    if ( (policy=sched_getscheduler(0)) == -1 )
        return (ERROR);
    if ( sched_getparam ( 0, &param )==-1 )
        return (ERROR);
printf("Scheduler set to: %s, PRIORITY: %d\n",
    policy==SCHED_FIFO?"SCHED_FIFO":"SCHED_RR", param.sched_priority );
    if(policy==SCHED_RR)
    {
        sched_rr_get_interval( 0, &tm );
        printf("RR time-slice: %d'%d''\n", tm.tv_sec, tm.tv_nsec );
    }
    return (OK);
}
```

通过在 shell 下运行“sp vxSched”可以得到类似下面的结果：

```
SCHED_FIFO priority range: 0 ~ 255
SCHED_RR priority range: 0 ~ 255
Current scheduling setup: SCHED_RR, PRIORITY: 155
RR time-slice: 0''166666660''
Scheduler set to: SCHED_FIFO, PRIORITY: 0
```

注意上面的运行结果，sp 生成的优先级为 100 的任务对应 POSIX 优先级为 155，sched_get_priority_min()得到的最低优先级为 0，恰好和 VxWorks 相反。另外可以看到，我们将 SCHED_RR 调度时间片设置为 10 ticks，结果输出 sched_rr_get_interval()返回的时间片约为 167ms（系统时钟溢出率为 60Hz）。

## 4．调度同步

当涉及同步问题时，有下面的细节需要注意。这些特点体现了这样的调度策略：在尊重任务优先级前提下，使任务尽快完成互斥部分的操作，缩短高优先级任务因互斥而等待的时间。

- 任务（或者 pthread）持有互斥体（不论属性为 PRIO_INHERIT 或者 PRIO_PROTECT）或者 VxWorks 信号量期间，不会因为 sched_setp aram( )改变其原始优先级而被移到其新优先级就绪队列尾部；
- 任务解锁互斥体时（不论属性为 PRIO_INHERIT 或者 PRIO_PROTECT），不会因为任务恢复到其原始优先级而被移到相应优先级就绪队列尾部；
- 持有多个互斥体的线程执行时的优先级根据其在各个互斥体上能得到的最高优先级允许；
- 多个线程竞争一个互斥体时，不论系统调度策略为 SCHED_FIFO 或 SCHED_RR，

线程都按照其优先级竞争互斥体。

可以参考3.4小节“线程”以及第2章“任务间通信”。

# 3.6 信 号 量

## 3.6.1 概述

信号量可以用于任务同步或者互斥访问。POSIX定义了无名信号量和命名信号量，两种信号量有相同的属性，初始化和删除时稍有区别。VxWorks 库 semPxLib 提供 POSIX 1331.1b 信号量支持。

如表3-13所示是POSIX信号量函数列表：

表3-13 POSIX信号量函数列表

| | 无名信号量 | 命名信号量 |
|---|---|---|
| 初始化（打开） | sem_init( ) | sem_open( ) |
| 删除（关闭） | sem_destroy( ) | sem_unlink( )<br>sem_close( ) |
| 锁定信号量 | sem_wait( )<br>sem_trywait( ) | sem_wait( )<br>sem_trywait( ) |
| 解锁信号量 | sem_post( ) | sem_post( ) |
| 信号量取值 | sem_getvalue( ) | sem_getvalue( ) |

### 1. 内部数据结构

不论无名信号量还是命名信号量，都由结构体 sem_t 表示。该结构体在 semaphore.h 中定义。程序初始化信号量之前，先要为信号量分配内存。

可以看出POSIX信号量是对wind信号量的简单封装。

```
typedef struct sem_des
{
    OBJ_CORE    objCore;      /* 内核对象 */
    SEM_ID      semId;        /* wind信号量（内部实现机制）*/
    int         refCnt;       /* 引用计数（sem_open次数） */
    char *      sem_name;     /* 名称 */
} sem_t;
```

refCnt 和 sem_name 对无名信号量没有意义。引用计数 refCnt 表示命名信号量被打开的次数，删除命名信号量时需要 refCnt=0。

## 2. 比较无名信号量和命名信号量

相对无名信号量，命名信号量在信号量初始化时需要搜索系统所有的POSIX信号量来检查指定名称的信号量是否存在，因此增加了运行开销。POSIX定义的命名信号量在打开和释放时具有文件系统的特点，主要表现在多个进程可以通过相同的文件名共享一个信号量，例如，所有进程都可以通过执行下面简单的调用，就可以在多个进程间共享名为name的信号量。

```
sem_open ( name, O_CREAT );
```

系统自动为第一个进行上述调用的进程分配信号量内核数据结构，对于后续调用，系统直接返回已经分配的地址。该过程如同文件系统一样，“天然”就为多个地位平等的进程共享该信号量所设计。该过程也简化了多进程使用信号量时的初始化。在上一章我们看到，要确保“一次且仅一次”初始化，实现起来并不简单。

而对于无名信号量，其内核数据结构在某个进程中分配，默认为该进程中多个线程进行同步和互斥：

```
Sem_t * semaphore = (sem_t *) malloc( sizeof(sem_t) );
sem_init ( semaphore, 1, ... );    /*多进程共享*/
...
```

如果需要多进程共享，需要在创建时设置sem_init( )的参数pshared非零；使用该信号量的进程需要知道信号量地址才能共享该信号量。

初始化时可以任意指定信号量初值（0～SEM_VALUE_MAX）；命名信号量初始化时也可以指定初值，打开已经创建的命名信号量的值维持其当前值不变。

在VxWorks中，由于其特殊的任务模型，命名信号量的优势并不明显；反而由于初始化命名信号量时的不确定延迟会影响系统实时性能，使得无名信号量更可取。VxWorks目前的命名信号量实现和文件系统无关：

- 信号量不能作为文件使用；
- 信号量名称不会出现在文件系统中，信号量名称不会和文件系统的设备名冲突，也无法通过iosDevFind( )查找命名信号量；
- 任务结束不会自动调用sem_close( )关闭信号量。

& 无名信号量和命名信号量差异仅体现在初始化（打开）和删除（关闭）时，在使用过程中，即进行同步和互斥时，两者是完全一样的。

## 3. 比较POSIX信号量和POSIX互斥体

POSIX信号量和POSIX互斥体都能用于互斥与同步，其差异如同wind计数信号量和wind互斥信号量的差异。

POSIX信号量在VxWorks 5.4已经有支持；POSIX互斥体从VxWorks 5.5开始支持。

### 4. 比较 POSIX 信号量和 wind 信号量

VxWorks 中 POSIX 信号量是在 wind 计数信号量的基础上实现的，因此效率要受一些影响。从另一个角度讲，历来“通用”、“标准”、“兼容”等词汇总是在带来的显见的优越性的同时伴随着或多或少的效率下降。现将两种信号量比较如表 3-14 所示。

表 3-14　两种信号量的比较

| | POSIX | wind |
|---|---|---|
| 优先级翻转 | POSIX 互斥体支持优先级继承和优先级保护（“天花板”） | wind 互斥信号量支持优先级继承 |
| 任务删除保护 | 通过线程撤销清除函数保护 | wind 互斥信号量通过隐式调用 taskSafe( )和 taskUnsafe( )实现任务删除保护 |
| 嵌套锁定 | VxWorks 未实现 | wind 互斥信号量支持 |
| 超时控制 | 不支持，但提供 sem_trywait | 可以指定任意超时控制 |
| 队列机制 | 未实现 | 支持 SEM_Q_PRIORITY 和 SEM_Q_FIFO |
| 信号量取值 | 支持 sem_getvalue( ) | 不支持 |
| 删除信号量 | 无名信号量：只能在没有任务阻塞时进行<br>命名信号量：系统记录删除请求，但直到引用次数为 0 时才实际执行 | 随时可删除，如果有任务阻塞，则相关任务的阻塞调用得到错误的返回值 |

上表中 POSIX 栏内容是基于 VxWorks 5.5 的实现给出的，也就是说：（1）在 VxWorks 5.4 中有些功能还不支持，如 POSIX 互斥体；（2）某些 POSIX 标准定义但是 VxWorks 未实现，如互斥体嵌套锁定在 POSIX 1003.1c 中定义但是 VxWorks 还没有支持，也许未来的版本会实现这些功能。

不加说明时，本节“信号量”表示 POSIX 信号量。

## 3.6.2 初始化信号量

初始化是使用信号量的第一步，无名信号量和命名信号量的初始化过程不同。

### 1. 初始化无名信号量

无名信号量由 sem_init( )初始化。调用者必须为无名信号量数据结构分配存储。

```
#include "semaphore.h"
int sem_init ( sem_t * sem, int pshared, unsigned int value );
```

函数将在用户分配的数据结构 sem 上初始化；pshared 对于 VxWorks 应用没有意义（是否允许其他进程共享）；value 指定信号量初值，不能大于 SEM_VALUE_MAX（100）。

当信号量用于任务同步时，信号量一般初始化为 0，相当于被锁定。从“生产者－消费者”角度看，相当于初始状态下生产者没有产品。

当信号量用于互斥访问时，通常被初始化为某个大于 0 的数（一般等于 1），表示资源可用。wind 互斥信号量创建时初值即为 1。

sem_init( )成功时返回 0。返回-1 表示出错，有两种错误情况：EINVAL（参数 value 表示的初值无效）和 ENOSPC（系统资源限制，最多允许同时 SEM_NSEMS_MAX 个 POSIX 信号量）。

## 2. 初始化命名信号量

初始化命名信号量由 sem_open( )实现。系统自动为第一个初始化信号量的调用分配存储。

```
#include "semaphore.h"
#include "fcntl.h"

sem_t * sem_open ( const char * name, int oflag, ... );
```

参数 name 表示要初始化的信号量名称，可以使用长达 NAME_MAX（99）个字符。斜杠（“/”）字符没有特殊意义，“/mysem”和“mysem”表示两个不同的合法的信号量名称。根据指定的信号量是否存在，以及参数 oflag 表示的参数，该函数实际的动作可能是打开，新创建，或者调用失败，如表 3-15 所示。

**表 3-15　函数的动作**

| 标志位 | 信号量已经存在 | 信号量不存在 |
|---|---|---|
| 0 | 打开信号量 | 调用失败（ENOENT） |
| O_CREAT | 打开信号量 | 创建信号量 |
| O_CREAT \| O_EXCL | 调用失败（EEXIST） | 创建信号量 |
| O_EXCL | 无意义 | |

省略部分表示的第 3、第 4 个参数可以用来在第一次打开命名信号量时指定初值，第 3 个参数非零表示第 4 个参数作为信号量初值。例如设置新信号量初值为 3，可以：

```
sem = sem_open ( sem_name, O_CREAT | O_EXCL, 1, 3 );
```

第 3，4 个参数在打开已经初始化的信号量时被忽略。

一般情况下，保持向后兼容，只需要指定前面两个参数，此时 sem_open( )将为新信号量设置的初值为 0。对于创建新信号量：

```
sem = sem_open ( sem_name, O_CREAT | O_EXCL );
```

```
sem = sem_open ( sem_name, O_CREAT );
```

或者，对于打开已经存在的信号量：

```
sem = sem_open ( sem_name, O_CREAT );
sem = sem_open ( sem_name, 0 );
```

信号量被初始化后，一直在系统中可用，直到被显式地删除（命名信号量可以自动被系统析构）。如果在项目中定义了 INCLUDE_POSIX_SEM_SHOW，还可以通过 shell 命令 show 显示 POSIX 信号量信息。

```
-> show 0xffe758
Semaphore name     :     /mysem
sem_open() count  :     3
Semaphore value   :     0
No. of blocked tasks :   2
```

无名信号量没有信号量名称和 sem_open( )次数的输出。

*应该注意避免重复初始化。系统不能检测出重复初始化的错误。如果有任务在信号量上阻塞，重复初始化将使阻塞的任务永远无法退出。*

## 3.6.3 信号量基本操作

POSIX 文档中用术语**等待**（Wait）或者**锁定**（Lock）表示对信号量的操作，对应于对 wind 信号量的**获取**（Take）信号量；而 wind 信号量的**释放**（Give）则对应 POSIX 信号量上的**解锁**（Unlock/Post）操作。

除了“锁定”和“解锁”，POSIX 还定义了取值操作（Sem_getvalue）。锁定和解锁类似 wind 信号量的 semTake 和 semGive。

### 1. 锁定信号量

```
#include "semaphore.h"
int sem_wait ( sem_t * sem );        /*锁定信号量（可阻塞）*/
int sem_trywait ( sem_t * sem );     /*锁定信号量（不阻塞）*/
```

sem 指向无名信号量或者命名信号量。根据信号量当前值，函数的执行情况如表 3-16 所示：

表 3-16　函数执行情况

| | 信号量当前值大于 0 | 信号量当前值等于 0 |
|---|---|---|
| sem_wait( ) | 将信号量的值减 1，然后返回 0 | 阻塞直到被一个 sem_post( )解除阻塞，然后返回 0 |
| sem_trywait( ) | 将信号量的值减 1，然后返回 0 | 立即返回-1，设置 errno=EAGAIN |

可见，"锁定"似乎有些误导，它并没有像互斥体那样防止其他任务同时得到该信号量，只是在信号量值为 1 时才"锁定"了信号量。

锁定 POSIX 信号量的操作类似对 wind 计数信号量调用 semTake( )：sem_wait(sem)类似 semTake(sem, WAIT_FOREVER)，sem_trywait(sem)类似 semTake(sem, 0)。

### 2．解锁信号量

```
#include "semaphore.h"
int sem_post ( sem_t * sem );
```

如果没有任务在信号量上阻塞，sem_post( )使信号量值加 1 然后返回；否则选择最高优先级任务解除阻塞。

### 3．信号量取值

```
#include "semaphore.h"
int sem_getvalue ( sem_t * sem, int * sval );
```

相对于 wind 信号量，信号量取值是 POSIX 增加的操作。sem_getvalue( )将 sem 当前值写到 sval 中。成功是返回 0，这时写入 sval 的值如表 3-17 所示。

**表 3-17　sval 值**

| | |
|---|---|
| 0 | 信号量被某个任务锁定，没有其他任务在阻塞 |
| <0 | 信号量被某个任务锁定，同时 sval 绝对值表示阻塞的任务数 |
| >0 | 信号量的值（没有任务任务锁定或者阻塞） |

sem_getvalue( )写入 sval 的值是在函数运行某点上的信号量状态，当函数返回时，可能由于其他任务对该信号量的操作使取出的结果过时。

## 3.6.4　删除信号量

### 1．无名信号量

无名信号量通过 sem_destroy( )删除：

```
#include "semaphore.h"
int sem_destroy ( sem_t * sem );
```

sem_destroy( )将删除系统为 sem 分配的部分存储，信号量结构体 sem 自身的存储由信号量初始化任务进行分配，因此也由其释放。

程序试图在一个已经被删除的信号量上操作将失败，errno=EINVAL。

如果信号量上有任务阻塞，sem_destroy( )将失败并返回-1，errno=EBUSY。

和删除 wind 信号量一样，建议在删除信号量之前先锁定它。

### 2. 命名信号量

删除命名信号量比删除无名信号量复杂，包括“释放”和“关闭”两个动作。当满足（1）程序发出释放信号量的请求 sem_unlink；（2）所有的打开都被关闭，即每个 sem_open 都被 sem_close 时，系统自动删除命名信号量。

```
#include "semaphore.h"
int sem_unlink ( const char * name );     /*释放命名信号量*/
int sem_close ( sem_t * sem );            /*关闭命名信号量*/
```

sem_unlink( )发出释放信号量请求，告诉系统要删除名为 name 的命名信号量。系统实际等待信号量引用次数（即被打开次数）为 0 时才进行删除。如果引用计数大于 0，sem_unlink( )只将信号量名称 name 从系统中移除，已经打开信号量的任务仍旧可以锁定或解锁信号量不受任何影响，但是任何任务将不能以相同的名称打开该信号量（可以以该名称创建新信号量，新信号量将不再是原来的信号量）。

sem_close( )解除一次对信号量的 sem 引用（即一次 sem_open 操作），VxWorks 不对调用的任务作任何检查，只将 sem 的 refCnt 字段减 1，因此任务应用关闭自己打开的信号量，防止系统删除信号量时其他任务还在使用信号量。

如果没有调用 sem_unlink( )，则即使 sem_close( )将 sem 引用计数减为 0，系统都不会删除信号量，任务仍然可以打开信号量。

# 3.7 消 息 队 列

## 3.7.1 概述

VxWorks 库 mqPxLib 提供 POSIX 1003.1b 消息队列支持。消息队列是任务间又一通信方式。消息队列允许队列两段的任务传递“任意”大小和结构的消息，从信息量讲，消息队列的能力比信号量强大。VxWorks 消息队列是在 wind 互斥信号量和 wind 消息队列基础上实现的；为了实现 POSIX 要求的功能，VxWorks 需要在 wind 基础上扩展 POSIX 消息队列的内核数据结构定义。

### 1. 比较 POSIX 消息队列和 VxWorks 消息队列

从创建消息队列，发送接收消息，删除消息队列角度看，两种消息队列接口功能大体上一样，如表 3-18 所示。

表 3-18　比较 POSIX 消息队列和 Wind 消息队列的接口功能

| | POSIX 消息队列 | Wind 消息队列 |
|---|---|---|
| 创建消息队列 | mq_open | msgQCreate |
| 删除消息队列 | mq_close<br>mq_unlink | msgQDelete |
| 发送消息 | mq_send | msgQSend |
| 接收消息 | mq_receive | msgQReceive |

POSIX 消息队列的创建和删除过程具有类似文件系统的特点，适合多个地位平等的任务使用：POSIX 消息队列具有独一无二的字符串名称（在所有 POSIX 消息队列范围内，与其他设备名称无关）。打开消息队列时 mq_open( )根据指定名称设备是否已经存在，分“新建”和“打开”两种不同处理；删除动作分 mq_unlink 请求释放和 mq_close 关闭队列两步完成。Wind 消息队列的创建和删除由一个“主动”的任务进行一次调用完成。

打开 POSIX 消息队列时必须指定该任务对此队列的读写方式：从队列接收消息或发送消息到队列；对同一消息队列，不同的任务可以以不同的读写方式打开；如果通过任务通过形参传递继承其他任务打开的消息队列，该任务同时继承该任务对此消息队列的读写属性。wind 消息队列没有类似约束，但是一个任务总是只需要对消息队列进行接收或者发送。

POSIX 消息队列是在 wind 消息队列基础上实现的，因此效率上后者更高。此外，从两种消息队列能力看，两者区别如表 3-19 所示：

表 3-19　比较 POSIX 消息队列和 Wind 消息队列的能力

| | POSIX 消息队列 | Wind 消息队列 |
|---|---|---|
| 消息优先级数 | 32 级优先级：0~31 | MSG_PRI_NORMAL<br>MSG_PRI_URGENT |
| 阻塞任务队列 | 基于优先级 | FIFO 或者基于优先级 |
| 超时控制 | 无，但可指定消息<br>队列为非阻塞方式 | 可指定任意的<br>超时控制时间 |
| 信号事件 | 可选（1 个任务） | 无 |

## 2. 内部数据结构

POSIX 定义消息队列数据结构为 mqd_t，称其为**消息队列描述符**。要求不能是数组类型，在 Free BSD 系统上实现为一个 int 型。

VxWorks 将类型 mqd_t 实现为一个结构体指针：

```
#include "mqueue.h"
struct mq_des;
typedef struct mq_des * mqd_t;
```

因此，程序可以对消息队列描述符进行复制（例如通过形参传递），通过得到的指针将访问同一个消息队列。后面将看到，在以信号事件方式接收消息时，信号处理函数正是通过这种传值的方式访问消息队列。

实际上，对同一个消息队列，任务通过 mq_open( )得到的消息队列描述符并不一定指向同一个地址。因为系统需要为访问同一消息队列的不同任务维护一些特定于任务的信息，如任务对消息队列的读写权限。如图 3-8 所示。

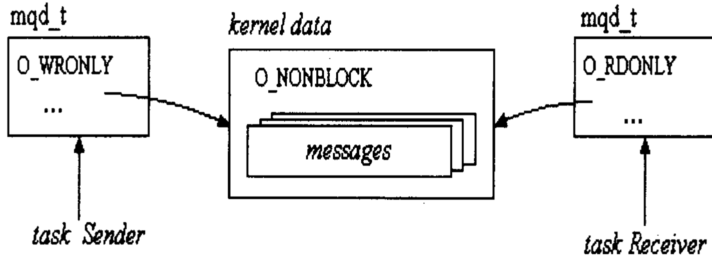



图 3-8　消息队列描述符

在任务间复制 mqd_t，得到的消息队列描述符表示同一地址，也具有相同的读写权限。

## 3．信号事件

POSIX 1003.1b 扩展了信号系统，增加了队列信号定义。扩展定义包括：当消息发送到一个空消息队列时，可以让应用程序选择是否发送一个信号给一个注册了接收此事件的任务。信号事件由结构体 sigevent 定义。消息队列和一个信号事件，以及信号事件和接收事件的任务之间是一一对应的关系，即如果有多个接收消息的任务，仅有一个任务能接收消息事件。

信号事件使任务不必常常去查询消息队列或者阻塞在消息队列上，任务可以执行其他计算，当消息到来时，系统会中断任务当前的计算过程，转入信号处理，如图 3-9 所示。

图 3-9 表示了消息队列信号事件的一般过程：

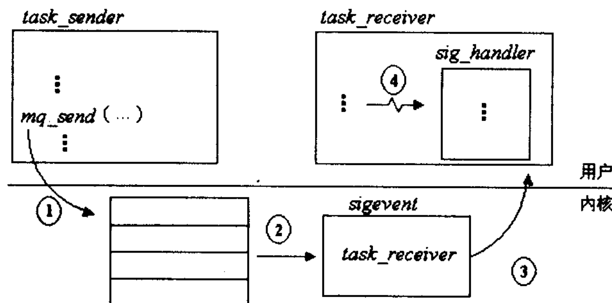



图 3-9　消息队列信号事件

（1）发送任务 task_sender 调用 mq_send( )发送一条消息；

（2）系统发现消息队列为空，检查信号事件设置；

（3）发现任务 task_receiver 注册了信号事件，发送 task_receiver 指定的信号；

（4）task_receiver 中断当前工作，进入信号处理过程 sig_handler。

图 3-9 只是一种简化了的情况。实际上信号处理过程比较复杂，主要在于接收信号方将有许多不同的选择方案。我们将在第 4 章专门介绍信号处理。

### 4. POSIX 消息队列属性

POSIX 消息队列有如下属性：是否阻塞，队列中最大消息数，最大每条消息长度，队列中当前消息数。由结构体 mq_attr 表示，包含以下内容：

```
#include "mqueue.h"
struct mq_attr        /*消息队列*/
{
    size_t      mq_maxmsg;          /*队列容量：最大消息条数*/
    size_t      mq_msgsize;         /*最大每条消息长度*/
    unsigned    mq_flags;           /*队列属性标志，如 O_NONBLOCK */
    size_t      mq_curmsgs;         /*队列中当前消息条数*/
};
```

和 POSIX 常用的哲学不同的是 POSIX 标准并没有定义上述属性操作函数，如 mq_attr_init( )或者 mq_attr_setmaxmsg( )之类（前文曾指出，定义这样的属性操作函数有利于代码向后兼容）。因此对于 POSIX 消息队列，程序直接深入结构体 mq_attr 的内部实现对属性的设置和读取，例如创建消息队列时，需要逐项初始化各个属性参数：

```
struct mq_attr attr;
attr.mq_flags   = 0;
attr.mq_maxmsg  = 16;
attr.mq_msgsize = 16;
mqPXId = mq_open (name, O_CREAT | O_RDWR | O_NONBLOCK, 0, &attr);
```

消息队列大小属性（mq_maxmsg 和 mq_msgsize）只能在创建时确定，无法改变。队列中当前消息条数（mq_curmsgs）对程序来说属于只读属性。

mq_flags 表示队列选项，表示发送消息到已满的消息队列或者从空消息队列中接收消息时是否要将调用任务阻塞。创建消息队列时可将 mq_flags 初始化为 0，而用 mq_open( ) 的第 2 个参数表示该选项：不指定 O_NONBLOCK 时，队列将在上述情况下阻塞；否则将不阻塞调用，而直接在上述情况下返回 ERROR，errno=EAGAIN。

程序可以随时调用 mq_getattr( )读取的所有属性 mq_attr.*：

```
#include "mqueue.h"
int mq_getattr ( mqd_t mqdes, struct mq_attr * pMqStat );  /*读取队列属性*/
```

函数将消息队列的属性写入 pMqStat 指向的属性结构体。常用该调用判断消息队列中

未读取的消息条数。多个任务间还通过该调用得知消息队列大小，以及阻塞属性。

如果有多个任务竞争读取队列，必须注意读出的消息条数只反映了队列在函数调用某一点时的状态。

POSIX 标准支持动态修改消息队列的阻塞属性(pMqStat->mq_flags)，调用 mq_setattr( )实现。

```
#include "mqueue.h"
int mq_setattr ( mqd_t mqdes, const struct mq_attr * pMqStat,
   struct mq_attr * pOldMqStat );              /*设置队列属性*/
```

参数 pMqStat 表示的新属性中只有 pMqStat->mq_flags 有意义，其他属性被忽略。对 pMqStat->mq _flags 的指定用于告诉系统该队列上的发送和接收是否允许任务阻塞。

如果 pOldMqStat 非 NULL，该调用同时将当前队列属性写入其中，结果将和在该点上调用 mq_getattr( )得到的结果一致。

调用 mq_setattr( )对读写方式（如 O_RDWR 等）没有影响。

下面这段程序判断队列是否会阻塞，然后将其修改为相反的状态：

```
mq_attr mqStat;
/*读属性*/
mq_getattr ( mqdes, &mqStat );
printf("block ? %s", mqStat.mq_flags & O_NONBLOCK ? " NONBLOCK" : "BLOCK" );
/*修改*/
mqStat.mq_flags ^= O_NONBLOCK;
mq_setattr( mqdes, &mqStat, NULL);
```

### 3.7.2 打开消息队列

使用消息队列的任务先必须调用 mq_open( )打开消息队列，该函数定义为：

```
#include "mqueue.h"
#include "fcntl.h"
mqd_t mq_open ( const char * mqName, int oflags,...);
```

函数成功时返回消息队列描述符，作为随后的消息接收发送函数的入参。出错时返回 ERROR（-1)。

第一个打开消息队列的任务负责创建消息队列，同时也为消息队列指定了名称，之后，其他任务就可以通过名字 mqName 来打开该消息队列并传递和接收消息。创建新消息队列或者打开已创建的消息队列通过标志 oflags 指定如表 3-20 所示。

表 3-20　标志位与消息队列

| 标志位 | 消息队列已经存在 | 消息队列不存在 |
|---|---|---|
| 无 | 打开消息队列 | 调用失败（ENOENT） |
| O_CREAT | 打开消息队列 | 创建消息队列 |
| O_CREAT \| O_EXCL | 调用失败（EEXIST） | 创建消息队列 |
| O_EXCL | 无意义 | |

mq_open( )还接受 oflags 中其他参数如表 3-21 所示。

表 3-21　其他参数功能

| 参　数 | 功　能 |
|---|---|
| O_RDONLY | 打开消息队列接收消息 |
| O_WRONLY | 打开消息队列发送消息 |
| O_RDWR | 打开消息队列进行消息接收和发送 |
| O_NONBLOCK | 发送接收消息时不阻塞任务 |

当且仅当指定了 O_CREAT 创建消息队列时，mq_open( )需要第 3、第 4 个参数：

```
mqd_t mq_open ( const char * mqName, int oflags, mode_t mode, mq_attr* pAttr );
```

其中，mode 只起形式上的作用，调用时赋 0；pAttr 指定新建消息队列的队列容量（pAttr->mq_maxmsg）和最大消息长度（pAttr->mq_msgsize），如果 pAttr 为 NULL，则 根据默认设置建立消息队列。

下面的代码创建一个名为“myqueue”的消息队列：

```
struct mq_attr mqAttr;
mqAttr.mq_maxmsg = 1;
mqAttr.mq_msgsize = 20;
mqd_t mqDes = mq_open("myqueue", O_CREAT|O_RDWR, 0, &mqAttr);
if (mqDes == (mqd_t)ERROR)   /* 出错处理 ... */
```

## 3.7.3 传递消息

### 1. 发送消息

函数 mq_send( )用于从任务环境或者 ISR 中发送消息到消息队列，该函数定义为：

```
#include "mqueue.h"
int mq_send ( mqd_t mqDes, const void * pMsg, size_t msgLen, int msgPrio );
```

参数 mqDes 指定消息发送到的消息队列，pMsg 指向要发送的消息，msgLen 表示消息

字节长度，msgPrio 指定消息的优先级，从最低优先级 0 到最高优先级 31（POSIX 标准）。优先级不是在队列时体现，而是体现在接收端被优先取出。

消息 pMsg 可以包含字符串，或者任意二进制内容，但是 msgLen 不能超过创建消息队列时指定的消息长度，否则消息无法加入消息队列，函数返回 ERROR，设置 errno 为 EMSGSIZE。

如果消息队列有空间，则 mq_send( )将消息放入队列并返回 OK。

如果消息队列已满，mq_send( )可以阻塞也可以不阻塞：

- 如果调用环境是 ISR，则 mq_send( )不阻塞，并返回 ERROR（-1），设置 errno 为 EAGAIN，这样发送者可以检查消息发送结果并采取相应的处理措施；
- 如果从任务中调用，则根据 mq_open( )打开消息队列时指定是否可选标志 O_NONBLOCK，是则不会被阻塞，如同 ISR 中调用，否则任务阻塞直到消息队列出现空闲。

## 2. 接收消息

接收消息使用 mq_receive( )，函数定义为：

```
#include "mqueue.h"
int mq_receive ( mqd_t mqDes, void * pMsg, size_t msgLen, int *  pMsgPrio );
```

该函数返回当前消息队列中最高优先级的消息；如果有多条，则按照先进先出 FIFO 顺序取出消息。

参数 mqDes 指定接收消息的消息队列，pMsg 指向要存放接收消息的缓冲区，msgLen 表示缓冲区长度，pMsgPrio 非 NULL 时用于存放接收消息的优先级。接收消息的缓冲区不能小于创建消息时指定的消息长度，否则调用失败，函数返回 ERROR。

消息队列空时，根据消息队列阻塞属性决定调用是否应该被阻塞：如果没有指定 O_NONBLOCK，调用任务被阻塞，直到另一任务发送一条消息到消息队列；如果指定了 O_NONBLOCK，调用任务不会被阻塞，mq_receive( )返回 ERROR（-1），errno=EAGAIN。

成功时返回读取的消息的实际字节数。

函数 mq_send( )可以被 ISR 调用，因此提供了一种 ISR 和任务之间的通信机制。不过，ISR 只能发送消息，它不能调用 mq_open( )创建消息队列或者打开已经创建的消息队列，ISR 可以通过全局变量使用在任务环境中已经得到的消息队列描述符，ISR 也不能从消息队列中接收 mq_receive( )。

在 ISR 和任务之间传递消息：

```
mqd_t theMqDes = (mqd_t)ERROR;
some_task ( )
{
    struct mq_attr mqAttr;
    mqAttr.mq_maxmsg = ...;
```

```
    mqAttr.mq_msgsize = ...;
    theMqDes = mq_open( qname, O_CREAT|O_RDWR, 0, &mqAttr);
    ...
}
void isr_func ( void )
{
    ...
    if ( theMqDes!= (mqd_t)ERROR)
        mq_send( theMqDes, ... ) /* 发送到消息队列 */
}
```

作为一种IPC机制，消息队列机制的优点在于简单。但是在消息队列上发送和接收时需要在内核和用户程序之间复制数据，因此对大量数据的传递效率不如“共享缓冲区+信号量”方式高。当数据量小时，程序运行开销主要在于对消息内容的处理，因此复制数据的开销是次要的。参考第2章“2.4 消息队列”对wind消息队列的讨论。

### 3. 删除消息队列

和POSIX命名信号量一样，POSIX消息队列也具有文件系统的特点。其删除过程分“关闭”和“释放”两部分：当满足（1）程序发出释放消息队列的请求mq_unlink；（2）所有的打开都被关闭，即每个mq_open都被mq_close时，系统自动删除命名信号量。

```
#include "mqueue.h"
int mq_unlink ( const char * mqName );    /*释放消息队列*/
int mq_close ( mqd_t mqdes );             /*关闭消息队列*/
```

mq_close( )关闭消息队列，即通知系统该任务不再需要使用指定的消息队列。系统为该任务使用消息队列所分配的资源将被释放，但是消息队列自身不会被删除，指定的消息队列不会从系统的消息队列名称表删除，不会释放系统资源。

任务可以调用mq_unlink( )松开一个消息队列。该调用将使所指定的消息队列从消息消息队列名称表中删除，随后其他试图打开该消息队列的调用将失败。但是系统没有立即删除该消息队列所占有的资源，已经打开该消息队列的任务可以继续使用该消息队列。当最后一个任务调用了mq_unlink( )之后，该消息队列将被删除。

## 3.7.4 消息到达通知

POSIX定义当有消息到达空队列时，系统可以向一个任务发送通知，即**信号事件**。信号事件指定了是否发送信号，以及发送的信号编码，附加信息。如果一个任务需要得到通知，必须先注册信号事件：

```
#include "mqueue.h"
```

```
int mq_notify ( mqd_t mqdes, const struct sigevent * pNotification );
```

mqdes 表示消息队列；pNotification 指向表示信号事件的结构体。mq_notify( )建立了消息队列 mqdes 和信号事件 pNotification 之间的关联。

一个消息队列只能被一个任务注册信号事件。如果 mqdes 已经被注册，mq_notify( )将出错（errno=EBUSY）。如果要更改注册，必须由原来的注册任务取消注册，即传递参数 pNotification 为 NULL。

信号事件结构体 sigevent 定义为：

```
#include "sigevent.h"
#include "signal.h"
struct sigevent      /*信号事件*/
{
    int          sigev_signo;   /*发送的信号编号*/
    union sigval sigev_value;   /*附加信息*/
    int          sigev_notify;  /*是否发送信号：SIGEV_NONE 或者 SIGEV_SIGNAL */
};
#define SIGEV_NONE     0
#define SIGEV_SIGNAL   1
```

上面，sigev_notify 表示是否要发送信号事件，SIGEV_NONE 时将不发送信号事件，SIGEV_SIGNAL 时发送信号事件。注意 SIGEV_NONE 也会阻止其他任务注册信号事件。

系统没有限制可以发送的信号编码 sigev_signo，但是一般不使用有特定含义的信号。可以使用 POSIX 1003.1 基本 OS 接口为用户应用定义的信号 SIGUSR1，SIGUSR2 以及 POSIX 1003.1b 扩展定义的 SIGRTMIN~SIGRTMAX。

sigev_value 作为信号的附加信息，将传递给信号处理函数。对消息队列信号事件，总是指定为消息队列本身，如下面这段代码：

```
struct sigevent;
/*设置消息队列信号事件*/
sigNotify.sigev_signo  = SIGUSR1;
sigNotify.sigev_notify = SIGEV_SIGNAL;
sigNotify.sigev_value.sival_int = (int) mqId;
if (mq_notify (exMqId, &sigNotify) == -1)
    ...    /*出错处理*/
```

和上面的信号事件对应，信号处理函数通常设计为从 mqId 接收消息。假定消息队列不阻塞（O_NONBLOCK），下面的信号处理函数将从消息队列中读取全部消息。

```
static void sig_handler (
        int sig,               /*信号编码，对应 sigNotify.sigev_signo */
        siginfo_t * pInfo,     /*信号信息，含信号事件中指定的编码和附加信息*/
        void  * pSigContext    /*源于 UNIX，一般不需要*/
```

```
)
{
    char msg[MSG_SIZE];
    mqd_t mqId = (mqd_t) pInfo->si_value.sival_int;

    /*循环读出队列中所有消息*/
    while (mq_receive (mqId, msg, MSG_SIZE, NULL) != -1)
        ...  /*消息已经读入msg后的处理*/

    /*正常情况下，函数读完所有消息，并得到结果errno=EAGAIN*/
    if(errno!=EAGAIN)
        ...  /*出错处理*/
}
```

上述信号处理函数使用POSIX 1003.1b扩展定义的信号接口。正确的使信号处理过程运行起来还需要进行信号处理函数安装，设置阻塞信号集等。这些细节在第4章第4.5节“POSIX 1003.1b扩展信号接口”中介绍。

**mq_notify( )**

（1）当消息到达非空队列时不给出任何通知；
（2）如果有另一任务阻塞在mq_receive()上，则被阻塞任务解除阻塞，注册mq_notify()的任务不会收到消息到达通知；
（3）一次mq_notify()注册只能使任务接收一次消息到达通知。系统发送一次信号事件后自动取消该任务的注册。如果某个任务要连续接收消息到达通知，最好在消息到达通知处理程序里面继续调用mq_notify( )。

## 3.7.5 消息队列示例

下面给出一个使用POSIX消息队列通信的示例程序pxQueue.c，该程序的目的一方面在于说明使用POSIX消息队列通信的方法；另一方面，程序还演示了消息队列通信中任务优先级和消息优先级的影响。

程序包括3个函数，如表3-22所示：

**表3-22　函数的功能**

| | |
|---|---|
| pxQueue | 初始化信号量，创建消息队列“myqueue”，生成发送和接收任务，释放创建的各种资源 |
| sender | 发送一条消息到“myqueue” |
| receiver | 循环读出所有“myqueue”中的消息 |

运行程序时，在shell输入“sp pxQueue”。pxQueue将根据sender和receiver函数创建8个发送任务和一个接收任务，并为发送任务和接收任务创建消息队列，属性为：3条消息

×50字节，阻塞方式。

许多“客户一服务器”应用都属于多发送任务（客户），一个接收任务（服务器），但是我们的例子也很容易扩展到多接收任务的情形。

发送任务优先级高于接收任务优先级，在接收任务运行之前，发送任务已经在消息队列上阻塞。可以通过观察程序的输出了解任务优先级和消息优先级对消息传递的影响。读者可以修改 pxQueue 函数中对两种优先级的定义（斜体部分），然后运行，观察有什么变化。

```
/*
 * pxQueue.c: test posix message processing order
 */
#include "vxworks.h"
#include "mqueue.h"
#include "semLib.h"
#include "fcntl.h"
#include "errno.h"

#define MQ_NAME "myqueue"
#define MAX_MSG 3
#define MAX_MSG_LEN 50
#define TEST_NUM 8
#define FATAL_ERROR(s) { \
      printf("%s [errno: 0x%x]\n", s, errno); \
      taskSuspend(0);  \
   }

SEM_ID  semMutex = NULL;
SEM_ID  semCounter = NULL;
char logBuf[TEST_NUM*3][MAX_MSG_LEN+20];
int nLog = 0;
void sender ( int, int );
void receiver ( void );

void pxQueue ( )    /*主任务函数，在shell运行：sp pxQueue*/
{
   int i;
   const int senderPri[TEST_NUM] = { 50, 51, 52, 49, 53, 56, 55, 54 };
   const int msgPri[TEST_NUM]   = { 10, 12, 11, 13, 14, 10, 16, 15 };
   const int rcvPri = 60;
   mqd_t mqDes;
   struct mq_attr mqAttr;
   semMutex  = semMCreate(SEM_Q_FIFO);     /*创建信号量*/
```

```
    semCounter = semCCreate(SEM_Q_FIFO, 0);
if( !semMutex || !semCounter )
        FATAL_ERROR("create semaphore failed")

    /*创建消息队列*/
    mqAttr.mq_maxmsg  = MAX_MSG;
    mqAttr.mq_msgsize = MAX_MSG_LEN;
    mqAttr.mq_flags   = 0;
    if ( (mqDes = mq_open( MQ_NAME, O_CREAT, 0, &mqAttr)) == (mqd_t)ERROR )
        FATAL_ERROR("mq_open failed to create mqueue")

    /*生成消息接收任务和发送任务*/
    for( i=0; i<TEST_NUM; i++ )
        if( taskSpawn( 0, senderPri[i], 0, 2000, sender, i+1, msgPri[i],
                0, 0, 0, 0, 0, 0, 0, 0 ) == ERROR )
            FATAL_ERROR("spawn sender failed")
    if( taskSpawn( 0, rcvPri, 0, 2000, receiver,
                0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ) == ERROR )
        FATAL_ERROR("spawn receiver failed")

    i = 0;                                  /*等待其他任务退出*/
    while( i++ < TEST_NUM+1 )
        semTake(semCounter, WAIT_FOREVER);

    printf("\n*** Logged result:\n");     /*输出记录的结果*/
    for( i=0; i<nLog; i++ )
        printf("  <%2d> %s\n", i, logBuf[i]);

    /*释放（删除）消息队列*/
    mq_close( mqDes );
    if( mq_unlink(MQ_NAME) == -1 )
        FATAL_ERROR("mq_unlink failed")

    semDelete(semMutex);                    /*删除信号量*/
    semDelete(semCounter);
}

void sender( int senderId, int msgPri )    /*消息发送任务函数*/
{
    int taskPri;
    char msgBuf[MAX_MSG_LEN];
    mqd_t mqDes = NULL;
    taskPriorityGet ( 0, &taskPri );
```

```
    /*以只写方式打开消息队列*/
    if ( (mqDes = mq_open( MQ_NAME, O_WRONLY)) == (mqd_t)ERROR )
        FATAL_ERROR("S: mq_open failed")

    /*记录 sender 就绪，互斥访问 logBuf*/
    semTake ( semMutex, WAIT_FOREVER );
    sprintf( logBuf[nLog], "S %d#: ready", senderId );
    nLog++;
    semGive(semMutex);

    /*生成消息 msgBuf，发送到消息队列，假定消息队列为阻塞方式*/
    sprintf ( msgBuf, "S %d#: task pri = %d, msg pri = %d",
              senderId, taskPri, msgPri );
    if ( mq_send( mqDes, msgBuf, strlen(msgBuf)+1, msgPri ) == -1 )
        FATAL_ERROR("S: mq_send failed")

    /*记录发送结果，互斥访问 logBuf*/
    semTake(semMutex, WAIT_FOREVER);
    sprintf ( logBuf[nLog], "S %d#: sent", senderId );
    nLog++;
    semGive(semMutex);

    mq_close(mqDes);            /*关闭消息队列*/
    semGive(semCounter);        /*给出信号量使主任务退出*/
}

void receiver( )        /*消息接收任务函数*/
{
    int i, pri;
    char msgBuf[MAX_MSG_LEN];
    mqd_t mqDes = NULL;

    /*以只读方式打开消息队列*/
    if ( (mqDes = mq_open( MQ_NAME, O_RDONLY)) == (mqd_t)ERROR )
        FATAL_ERROR("R: mq_open failed")

    /*依次读取所有队列消息，在 logBuf 中记录读取结果，假定队列为阻塞方式*/
    for( i=0; i<TEST_NUM; i++ )
    {
        if ( mq_receive( mqDes, msgBuf, MAX_MSG_LEN, &pri ) == -1 )
            FATAL_ERROR("R: mq_receive failed")
        semTake(semMutex, WAIT_FOREVER);    /*互斥访问 logBuf*/
        sprintf( logBuf[nLog], "R: [ %s ] received, pri=%d", msgBuf, pri );
        nLog++;
```

```
        semGive(semMutex);
    }

    mq_close(mqDes);           /*关闭消息队列*/
    semGive(semCounter);       /*给出信号量使主任务退出*/
}
```

我们略去程序运行输出结果，直接给出结论：

- 多个任务竞争发送消息时，仅仅任务优先级本身决定谁将获得消息队列发送权；
- 对于已经写入消息队列的消息，消息优先级惟一决定其被接收任务读取的顺序。

VxWorks 任务优先级 0~255 数字越小优先级越高；POSIX 消息优先级 0~31 数字越大优先级越高。
创建 POSIX 消息队列的任务可以不指定消息队列的读写方式；具体对消息队列进行读写的任务必须在打开消息队列时指定正确的读写方式。
任务可以将其打开的消息队列描述符传递给其他任务，但是不提倡。建议：各任务自己打开消息队列（mq_open），使用完后关闭（mq_close）；创建消息队列的任务关闭并释放消息队列（mq_close 和 mq_unlink）。
信号处理函数一般通过形参传递得到消息队列描述符。

# 第4章 信 号

## 4.1 信号概述

信号主要用于任务之间传递控制信号，是一个软件的概念，类似于系统中硬件中断或者异常发生事件，用于通知运行的任务，有某个感兴趣的事件发生了，有时也称信号为软中断。

任何任务或者 ISR 都可以对某个任务引发一个信号。在 VxWorks 中，收到信号的任务立即挂起当前执行的线程，并在下次任务被调度时执行该任务事先指定的信号处理程序。

使用信号的利弊：信号使程序可以专注于其主要任务，而对次要的和不经常的事件采用信号处理的方式实现，这样程序处理主要任务时不需要不时地去检查这些次要事件；但是从整体上讲，信号对任务来说属于异步事件，信号处理程序无法确知信号发生时系统正在进行的动作和存在的状态，因而在某些场合下会存在麻烦。

VxWorks 信号功能由库 sigLib 实现。分基本信号接口（宏定义 INCLUDE_SIGNALS）和 POSIX1003.1b 扩展信号接口（宏定义 INCLUDE_POSIX_SIGNALS）。基本信号接口又分 BSD 信号接口和 POSIX 信号接口。

[Stevens2000]对信号作了详尽而深入的分析。

### 1. 信号的产生

产生信号的原因大体上归于下面 3 类：

- 错误引起：程序出现错误并且无法继续运行。通常属于特定模块调用的错误只需要返回错误状态，不需要产生信号，由发出调用的程序“负责”该错误状态，如打开无效文件时返回-1 即可。对于可以在程序任意地方引起的错误，例如除零错，读写无效内存地址，则需要引起信号，此时信号处理程序对该错误“负责”。
- 外部事件引起：外部事件通常由 I/O 和其他一些事件引起，如请求的输入有效，定时器到时，子进程结束（在 VxWorks 中不存在）等。
- 显式产生：程序进行系统调用产生信号，如 kill，raise，sigqueue 等。在第 2 章第 2.6 节“信号”一节中曾给出的例子属于此种情况。

具体地，VxWorks 中定义了如表 4-1 所示的信号来源：

表 4-1 信号来源

| | |
|---|---|
| SI_SYNC | 硬件产生（非 POSIX 标准） |
| SI_USER/ SI_KILL | 来源于用户发送（调用 kill 或者 raise） |
| SI_QUEUE | 来源于 sigqueue 传递 |
| SI_TIMER | 来源于定时器到时 |
| SI_ASYNCIO | 来源于异步 I/O 完成 |
| SI_MESGQ | 消息到达通知信号 |

## 2. 信号种类

VxWorks 定义了 31 种信号。在当前主流的 32 位处理器上，可以用一个 32 位整型变量的 31 位映射 31 种信号。下面几个表给出这些信号的名称及其用途。

（1）POSIX 要求的信号如表 4-2 所示：

表 4-2 POSIX 要求的信号

| 信 号 | 用 途 |
|---|---|
| SIGABRT | 异常结束程序，调用 abort( )得到 |
| SIGALRM | alarm 时钟超时 |
| SIGFPE | 浮点异常 |
| SIGHUP | 终端挂起（连接断开） |
| SIGILL | 非法指令 |
| SIGINT | CTRL-C 中断（终端中断符） |
| SIGKILL | 终止 |
| SIGPIPE | 写无读进程的管道（VxWorks 未实现） |
| SIGQUIT | 终端异常结束 |
| SIGBUS | 总线异常 |
| SIGSEGV | 存储访问异常 |
| SIGTERM | 终止 |
| SIGUSRI | 应用程序定义 |
| SIGUSR2 | 应用程序定义 |

（2）POSIX 可选任务控制信号，在 VxWorks 中定义但未实现，如表 4-3 所示：

表 4-3 可选任务控制信号

| 信 号 | 用 途 |
|---|---|
| SIGCHLD | 子进程状态改变 |

续表

| 信　　号 | 用　　途 |
| --- | --- |
| SIGSTOP | 无法捕获的进程停止 |
| SIGTSTP | 从终端停止进程 |
| SIGCONT | 停止的进程继续运行 |
| SIGTTOU | 停止后台进程 |
| SIGTTIN | 停止后台进程 |

（3）可选 POSIX 队列实时信号如表 4-4 所示：

表 4-4　实 时 信 号

| 信　　号 | 用　　途 |
| --- | --- |
| SIGRTMIN SIGRTMAX | 应用程序定义 |

对上述信号的更详尽的定义请参考[Stevens2000]。注意 VxWorks 对 SIGKILL，SIGCONT 和 SIGSTOP 未定义 UNIX 中赋予的含义。例如 UNIX 定义 SIGKILL 为终止进程，进程不能捕获或阻塞该信号，但是在 VxWorks 中，任务可以像使用 SIGUSR1 一样使用它(发送和捕获)。另外，UNIX 中信号 SIGURG 表示 socket 收到紧急带外数据，在 VxWorks 中没有实现。

在 VxWorks 5.5 中，编号为 16 的信号 SIGCANCEL（又名 SIGCNCL）具有特殊意义。VxWorks 通过 SIGCANCEL 实现 POSIX 1003.1c 中定义的异步线程删除。

### 3. 信号处理方式

当信号“递交”给接收任务时，可以分为如下几种：

- 默认 —— 系统对不同的信号有不同的默认处理方式，包括忽略信号，挂起进程，终止进程；
- 忽略 —— 好像信号没有发生过。如果信号既没有被阻塞，也没有设置处理函数，则应将其忽略。否则系统默认动作将使任务结束；
- 捕捉 ——信号到达任务时调用预先设置好的信号的处理函数。

信号处理的简单过程如图 4-1 所示：

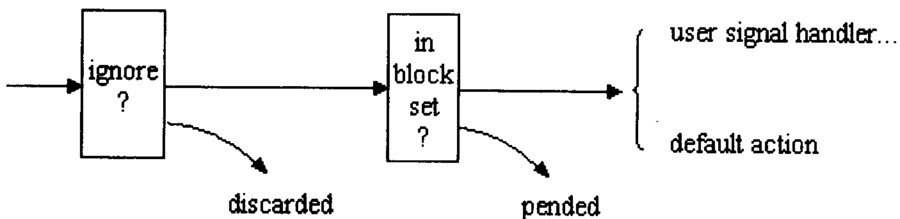



图 4-1　信号处理的简单过程

& 任务可以“阻塞”信号。每个任务有一个阻塞信号集，属于该集的信号不会立即递交给任务，而是被记录下来，等待解除阻塞后可以继续处理。阻塞使信号被推迟传递给任务。后面将要看到不同的信号接口（BSD 和 POSIX）对阻塞信号集有不同的表示和处理。

### 4. 信号与异常

异常是运行中出现的程序代码或者数据错误，如非法指令、总线错误、除数为零等。默认情况下，VxWorks 让其异常处理包来处理这些异常，而处理的方法就是将产生异常的任务挂起，其他任务不受影响。

通过捕获引起的异常信号，任务可以安装自己的异常处理函数，这样默认的异常处理将不被调用。在用户定义的异常处理函数中，常常通过 longjmp( )使任务从某个“环境记录点”重新开始运行。环境记录点必须有效，任务需要事先通过 setjmp( )设置环境记录点，并且设置环境记录点的函数没有结束。我们在第 1 章“1.3 内核功能接口”一节中曾介绍了通过环境记录点使任务重启的方法。根据 VxWorks 的实现，跳到环境记录点时任务的阻塞信号集也被重新设置。

下面表示了这样一个通过信号捕获来处理异常的方法。我们假定函数 func( )要执行某设备访问，可能会发生总线或存储访问异常，因此我们事先设置函数 excHandler( )捕获信号 SIGBUS 和 SIGSEGV，并在每次访问设备的循环开始时设置一个环境记录点。这样，如果发生异常，信号处理函数 excHandler( )将能够处理该错误，然后将执行跳到所设置的环境记录点，继续循环。

```
jmp_buf exc_restore;    /*异常的环境记录点*/

void func (void)
{
    signal (SIGBUS, excHandler );    /*为可能引起的设置信号处理函数*/
    signal (SIGSEGV, excHandler );

    FOREVER
    {
        if (setjmp (exc_restore) != 0)
        {
            ...  /*这部分表示因异常处理而进入*/
        }
        ...  /*正常访问设备*/
    }
}
```

```
void excHandler (int sigNum)
{
    ...  /*其他操作，比如将设备复位*/
    longjmp (exc_restore, 1);
}
```

# 4.2 信号处理函数

当信号处理方式为“捕捉”时，需要为信号指定一个“信号处理函数”，该函数和中断服务程序一样，当捕捉到对应的信号时，系统自动调用该函数。信号处理函数运行具有以下特点：

- 信号处理函数在被信号中断的任务上下文中运行（为任务设置堆栈大小时要考虑信号处理函数开销）；
- 被中断的任务可以处于就绪、运行、阻塞、延迟等状态及这些状态的组合下。如果不是就绪态，系统会将任务暂时提升为就绪态。信号处理函数运行时具有所属任务的优先级，调度时仍然遵从系统调度策略；
- 对于被 taskSuspend( )挂起的任务的信号处理函数不会被调用，直到挂起解除；
- 信号处理函数运行结束后，任务回到被中断前的状态；
- 如果被中断前任务为延迟状态，则信号处理函数返回后任务继续未完的延迟部分。

从上面看，信号处理过程对任务是基本透明的，但不是完全透明，在后面的第 4.6 节“信号的影响”中分析了信号处理过程对任务的影响。

信号处理函数是一个用户定义的普通C语言函数。根据信号接口的不同和程序的期望，信号处理函数有两种类型的定义：一般定义和扩展定义。

一般定义为：

```
void sigHandler ( int signo )      /* signo: 信号编号 */
{
    ...
}
```

信号处理可能发生嵌套：如果在运行一个信号处理函数系统递交了一个新的信号，则系统中断当前的信号处理函数转入新的信号处理。嵌套只发生于不同的信号，相同的信号在其信号处理函数被调用期间被自动阻塞。可以通过指定信号处理期间的阻塞信号集避免发生嵌套。

信号可能丢失，即递交一个信号的次数大于信号处理函数调用次数。因为系统对一种信号只有一个标志位记录其是否发生，因此，对于在系统调用信号处理函数之前的多次递交，只能保证一次信号处理函数调用。多见于信号在其被阻塞期间多次到达。

虽然不是强制要求，但是在信号处理程序中应避免进行可以引起阻塞的调用。

扩展定义支持 POSIX 1003.1b 实时扩展，信号处理函数提供了更多的信息。具体参考第 4.5 节“POSIX1003.1b 扩展信号接口”。

# 4.3 BSD 信号接口

VxWorks 支持 4.3BSD 传统 UNIX 风格和 POSIX 标准的信号接口。本节简单介绍 BSD 信号接口。使用 POSIX 信号接口比传统 BSD 信号接口更具有优势：（1）在很多系统上得到支持，包括 UNIX 本身；（2）接口功能更强，编程方便；（3）可以和后面介绍的 POSIX 扩展信号接口一起使用。因此，我们在 4.4 节将更详细地介绍 POSIX 信号接口。

需要注意避免在程序中混用两种接口。

## 1. 发送信号

```
#include "signal.h"
int raise ( int signo );      /* signo: 信号 */
int kill ( int tid, int signo );  /* tid: 接收信号任务 ID, signo: 信号 */
```

raise( )发送信号给调用任务自身；kill( )向任何指定任务（tid 表示）发送信号，现在看来名称“kill”并不十分恰当，早期 UNIX 系统中 kill( )较多用来发送 SIGKILL 信号使目标进程终止。

kill( )可以从中断服务程序中调用。接收任务设置的信号处理程序将被“立即”调度。信号处理程序以被中断任务优先级运行，因此“立即”调度受优先级制约。除非任务已经在 CPU 中，还需要进行上下文切换。信号处理程序运行结束后被中断任务回到被信号中断前的状态。

& kill( )同时属于 POSIX 信号接口。POSIX 定义允许传递给 kill( )的信号 signo 为 0，此时定义 kill( )执行正常的错误检查但不发送信号。这可以用来在发送前确定一个特定进程是否存在。但是 VxWorks 对这种情况将返回 ERROR，并设置 errno 为 EINVAL（无效参数）。

## 2. 设置信号处理方式

```
#include "signal.h"
void (*signal(int __sig, void (*__handler)(int)))(int);
```

该原型比较复杂。参数__sig 表示信号，__handler 表示信号的处理程序。signal( )能设置信号的处理方式为如下三者之一：

- 默认，设置__handler 为 SIG_DFL（定义为 1）；

- 忽略，设置__handler 为 SIG_IGN（定义为 0）;
- 捕捉，设置__handler 为用户定义的信号处理函数地址。

函数返回值为之前的信号处理函数地址，或者 SIG_ERR（定义为-1）表示错误。

如果觉得 signal( )函数原型过于麻烦，可以看看[Stevens2000]给出的一个等价定义：

```
typedef void SigFunc ( int );
SigFunc * signal ( int, SigFunc * );
```

signal( )同时属于 BSD 和 POSIX 接口。

函数 sigvec( )和 signal( )类似，但是只属于 BSD 接口。

```
#include "signal.h"
int  sigvec( int __sig, const struct sigvec *__vec, struct sigvec *__ovec );
```

sigvec( )有两个用途：（1）设置信号__sig 处理函数为__vec；（2）如果__vec 为 NULL，则读取当前信号处理函数到__ovec。

__vec 和__ovec 为指向结构体 sigvec 的指针，在 signal.h 中，该结构体定义为：

```
struct sigvec
{
    void (*sv_handler)(int); /* signal handler */
    int sv_mask;      /* 阻塞信号集 */
    int sv_flags;     /* 信号选项 */
};
```

信号处理函数地址只是结构体 sigvec 内容之一，即 sv_handler 字段。

在信号处理函数运行期间，引起该次调用的信号本身自动被阻塞。如果还希望信号处理不被其他信号中断，可以通过参数 sv_mask 指定信号处理期间应予阻塞的信号（信号处理函数运行完毕后恢复任务函数调用前的阻塞信号集）。在 sv_mask 中一位代表一个信号，参考下面对设置阻塞信号集的介绍。

### 3. 设置阻塞信号集

BSD 通过下列函数指明阻塞信号集：

```
#include "signal.h"
int sigblock ( int mask );    /* 添加 mask 中信号到阻塞信号集 返回原来的阻塞信号集 */
int sigsetmask ( int mask ); /* 设置任务阻塞信号集为 mask 返回原来的阻塞信号集 */
```

即 BSD 通过一个整型变量中一位（bit）表示对应信号是否被阻塞。例如要添加阻塞信号 SIGUSR1，应该：

```
sigblock ( SIGMASK(SIGUSR1) );
```

宏 SIGMASK 将信号转换到阻塞信号掩码。在 signal.h 中定义为：

```
#define SIGMASK(m)    (1 << ((m)-1))
```

显然，这种方式受机器字长限制，不过目前在 VxWorks 中信号最大编号为 31，因此在 32 位处理器上还是安全的。

sigblock( )和 sigsetmask( )常常用来对关键代码段进行保护。例如，期望下面这段代码执行期间不被信号 signo1 和 signo2 中断，可以将这两个信号临时进行阻塞：

```
oldMask = sigblock (SIGMASK(signo1) | SIGMASK(signo2));
        ...          /* 不被信号中断的代码 */
sigsetmask ( oldMask );
```

### 4. 等待信号

```
#include "signal.h"
int pause( void );
```

pause( )使任务阻塞，直到信号到来并且相应信号处理函数运行结束。该函数总是返回 ERROR 并且设置 errno 为 EINTR。

被阻塞，或者未被阻塞但是设置了以默认或者忽略方式处理的信号不会使 pause( )返回。

## 4.4 POSIX 信号接口

POSIX 信号接口包括下列函数。如表 4-5 中还给出了对应的 BSD 函数，但是功能对应上并不完全一致。

**表 4-5　POSIX 信号接口函数**

| POSIX 定义 | 函数功能 | 对应 BSD 定义 |
|---|---|---|
| sigemptyset( )<br>sigfillset( )<br>sigaddset( )<br>sigdelset( )<br>sigismember( ) | 信号集操作 | 宏 SIGMASK |
| sigprocmask( ) | 设置/检查任务阻塞信号集 | sigsetmask( )<br>sigblock( ) |
| sigsuspend( ) | 阻塞任务，直到信号到达 | pause( ) |
| kill( ) | 发送信号到指定任务 | kill( ) |
| signal( ) | 设置信号处理程序 | signal( ) |
| sigaction( ) | 设置/检查信号处理函数 | sigvec( ) |
| sigpending( ) | 取回被阻塞的信号 | |

## 4.4.1 阻塞信号集

阻塞信号集包含不能立即递交给接收者的信号。POSIX 是具体通过一个数据集结构 sigset_t 和下面 5 个信号集操作函数来实现这一抽象概念。

```
#include "signal.h"
int sigemptyset(sigset_t *__set); /* 初始化信号集__set 不包含任何信号 */
int sigfillset(sigset_t *__set);  /* 初始化信号集__set 包含所有信号 */
int sigaddset(sigset_t *__set, int __signo); /* 将__signo 添加到__set */
int sigdelset(sigset_t *__set, int __signo);  /* 将__signo 从__set 删除 */
int sigismember(const sigset_t *__set, int __signo);
    /* 判断__signo 是否属于__set 表示的信号集 */
```

前面 4 个函数返回 OK/ERROR 表示操作是否成功。

最后一个函数 sigismember()返回 1 表示__signo 属于信号集__set，0 表示不属于，-1（ERROR）表示操作错误。

> POSIX 这种实现方式比 UNIX 更一般化，因而便于未来扩充以及移植到不同的体系上。虽然 sigset_t 就是一个长整型掩码字，掩码字每 1 位对应阻塞信号集中一个信号，但是，编写应用代码时应该脱离具体实现细节，比方说设置阻塞信号集为空，不要直接赋__set 等于 0，而应该调用 sigemptyset( )实现。

上述 5 个函数得到的信号集作为 POSIX 函数 sigprocmask( )的参数：

```
#include "signal.h"
int sigprocmask ( int how, const sigset_t * pSet, sigset_t * pOset )
```

该调用用于设置或者读取任务的阻塞信号集。前面已经说明，如果某个信号在阻塞信号集中，则表示该信号被阻塞。

第 1 个参数 how 指定如何根据第 2 个参数 pSet 设置任务阻塞信号集。以图 4-2 为例说明参看表 4-6 所示：

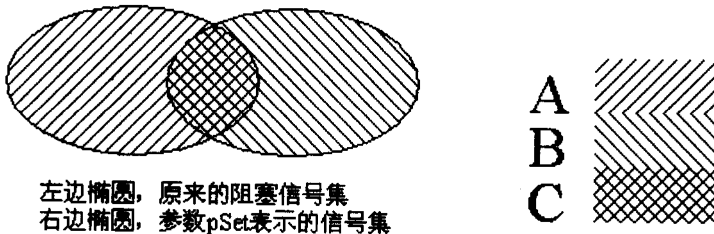



图 4-2　阻塞信号集

表 4-6 阻塞信号参数

| 参数 how | 含 义 | 图 4-2 结果 |
|---|---|---|
| SIG_BLOCK | 将 pSet 表示的信号集加入到阻塞信号集 | A+B+C |
| SIG_UNBLOCK | 从当前阻塞信号集减去 pSet 表示的信号集 | A |
| SIG_SETMASK | 设置阻塞信号集为 pSset 表示的信号集 | B+C |

第 3 个参数 pOset 如果非 NULL，则该调用将当前的阻塞信号集保存到 pOset。如果第 2 个参数 pSet 为 NULL，仅仅读取当前的阻塞信号集而不修改。

& 任务初始时的阻塞信号集为空，相当于任何信号都被立即送达任务。信号 SIGKILL 和 SIGSTOP 不能被阻塞。
POSIX 标准定义的阻塞信号集设置是基于线程的，而信号处理函数是基于进程的。也就是说一个进程的多个线程可以单独设置各自的阻塞信号集。但是 VxWorks 实际上没有多线程，因此 VxWorks 中信号都是基于任务的，包括：阻塞信号集的设置，信号处理函数的设置，调用都对任务或者以任务为对象进行。这实际上使事情变得更加简单。

## 4.4.2 信号处理函数

POSIX 使用结构体 sigaction 表示信号处理函数和一些相关信息，定义为：

```
#include "signal.h"
struct sigaction
{
    union
    {
        void (*__sa_handler)(int);
        void (*__sa_sigaction)(int, siginfo_t *, void *);
    } sa_u;
    sigset_t  sa_mask;
    int       sa_flags;
};
```

信号处理函数由指针 sa_u.__sa_handler 或者 sa_u.__sa_sigaction 指定。前面已经说明，POSIX 标准定义了两种信号处理函数：一般定义和扩展定义。信号处理函数的一般定义由上述结构中的指针 sa_u.__sa_handler 表示；POSIX 1003.1b 实时扩展定义的信号处理函数使用 sa_u.__sa_sigaction 表示。后面一种情况时必须设置 sa_flags 的 SA_SIGINFO 标志。

除了 POSIX 1003.1b 实时扩展定义的队列实时信号（SIGRTMIN 到 SIGRTMAX）外的信号处理函数都采用一般定义形式。4.4.3 节“同步处理”将介绍扩展定义。

位域 sa_mask 和前面 BSD 函数 sigvec( )中同名参数一样，表明该信号被处理期间所应自动被阻塞的信号。调用时将 sa_mask 指定的信号加入任务原先的阻塞信号集。阻塞信号集在信号处理函数返回后恢复。引起该次调用的信号本身自动被阻塞。

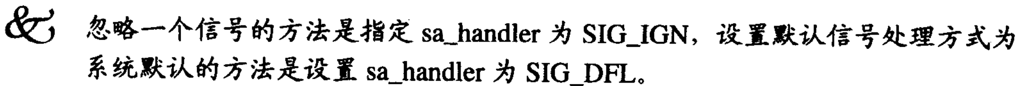
标志 sa_flags 包含两个标志位。其中一位 SA_NOCLDSTOP 用于控制信号 SIGCHLD 的传递，在 VxWorks 中没有实现。另一位 SA_SIGINFO 表示信号属于 POSIX 1003.1b 实时扩展定义的信号，这时的信号处理函数定义采用带 3 个参数的扩展定义，由指针 sa_sigaction 表示。

具体安装信号处理函数由同名的 sigaction( )实现：

```
#include "signal.h"
int sigaction(int __sig, const struct sigaction *__act, struct sigaction
*__oact);
```

参数__sig 表示信号，__act 和__oact 是指向结构体 sigaction 的指针。sigaction( )可以安装或者读取信号处理函数和其他信息。当__act 不为 NULL，函数根据__act 表示的结构体设置__sig 的信号处理函数；如果__oact 不为 NULL，则当前的信号处理通过__oact 返回；两者同时为 NULL 时函数将指示错误。

成功时返回 OK，否则返回 ERROR。


忽略一个信号的方法是指定 sa_handler 为 SIG_IGN，设置默认信号处理方式为系统默认的方法是设置 sa_handler 为 SIG_DFL。

## 4.4.3 同步处理

已经说明，信号具有异步的本质，其产生总是异步的。从处理方式讲，则有异步和同步的区分。

刚才介绍的 sigaction( )的方法具有异步处理的特点：（1）设置信号处理函数；（2）信号递交给接收任务；（3）系统中断任务当前工作；（4）设置环境转入信号处理函数；（5）完毕后清除环境继续运行被中断的工作。这种处理方式和设置中断服务程序对中断进行处理一样。如果信号处理过程和任务中的处理过程没有强耦合性，则异步处理是可以的。

具有“同步”特点的信号处理适合于信号处理过程和任务中的处理过程等具有较强的耦合性的场合，此时，任务必须在“耦合点”和信号“同步”。一般属于任务和信号需要协同工作的情况。同步时，任务通过一个等待信号的调用引起自身被阻塞。当等待的条件满足时（或者发生超时）解除任务阻塞。等待的条件有两种情况：

（1）等待非阻塞信号，该信号到来后，系统设置环境运行信号处理函数，完毕后清除环境并解除任务阻塞；

（2）等待阻塞信号到来，条件满足后，返回信号编号和其他信息（如果有的话），任务判断信号类型再作相应处理。

其中，第一种情况通过 sigaction+ sigsuspend 实现，第二种情况有几种稍有区别的实现方法（如 sigwait、sigwaitinfo、sigtimedwait），下面分别介绍。

同步信号传递的特点是阻塞信号与等待信号。要注意等待的是阻塞信号还是非阻塞信号，正确设置信号集。

## 1. 等待非阻塞信号

```
#include "signal.h"
int sigsuspend ( const sigset_t * pSet );
```

函数 sigsuspend( )根据传递的入参 pSet 设置任务的阻塞信号集，然后停止执行（调用 sigsuspend 的任务被阻塞），直到没有阻塞的信号到来。该信号必须设置了信号处理函数，则信号处理函数返回后，sigsuspend( )返回，任务解除被阻塞的状态。

sigsuspend 总是返回 ERROR（-1），并设置 errno 未 EINTR，可以忽略该结果。

注意:

（1）sigsuspend( )等待的是不在其参数表示的信号集（阻塞信号集）内的信号;

（2）如果递交的信号被指定以默认或者忽略的方式处理，并不能使 sigsuspend ( )返回。

## 2. 等待阻塞信号

```
#include "signal.h"
int sigwait ( const sigset_t * pSet, int * pSig );
```

pSet 指定要等待的阻塞信号集。当其有一个到来时，存储信号编号到 pSig 中。如果调用时已经有多个属于阻塞信号集的信号被阻塞，则其中编号最小的信号放在 pSig 中返回，同时清除对应阻塞记录。

sigwait( )只等待阻塞信号，不负责调用信号处理函数。事实上很多情况下可以利用这一点提高效率，因为信号处理函数由程序调用比让系统自动调用更具有效率。

如果条件不满足，函数将一直阻塞，直到条件满足时返回 OK。因此，如果函数返回 ERROR 时，只表示参数 pSet 无效（errno=EINVAL）或者 pSig 有错误（errno=ENOMEM）。

**sigwait( )**

（1）等待的是属于其参数表示的信号集（阻塞信号集）内的信号;

（2）如果希望避免陷入无限等待，可以考虑使用 4.5 中将介绍的两个具有超时控制的类似函数 sigtimedwait( )，除了 sigwait( )的所有功能外，sigtimedwait( )还可用于等待 POSIX1003.1b 定义的扩展队列信号。

# 4.5 POSIX1003.1b 扩展信号接口

前面介绍的信号接口由 POSIX 1003.1 基本 OS 接口定义。该信号模型存在一些问题，主要是两个问题：

（1）很多信号主要用于系统控制，用户代码使用有限制。用户自由使用的有 SIGUSR1 和 SIGUSR2，但是信息量太少，在某些复杂场合应用不便；

（2）信号传递给任务是通过设置任务数据结构中一个对应位来实现的，这样如果信号在未被处理之前再次到来，则可能因为后面的信号覆盖前面的记录而丢失信号。

POSIX 1003.1b 对 POSIX 1003.1 定义的信号模型进行了实时扩展，新定义了一组实时信号，从 SIGRTMIN 到 SIGRTMAX。原来定义的信号接口继续使用，新的接口具有如下特点：

（1）丰富了信号内容。增加了可以由应用程序定义的信号，并且新增加的信号附带额外的信息，使信号更有利于任务之间通信；

（2）信号进行队列，不会发生覆盖，使得信号更可靠；

（3）信号有了优先级，进行队列时，按照优先级顺序排队，程序可以将紧急信号表示为优先级高的信号。

VxWorks 实现了 7 个 POSIX 1003.1b 扩展的实时信号，从 SIGRTMIN 开始连续编号，供应用程序使用。这些信号发送时将按照编号从小到大的优先顺序进行队列。

下面是 POSIX 1003.1b 队列信号接口，如表 4-7 所示：

**表 4-7　队列信号接口函数**

| 函　数 | 功　能 |
| --- | --- |
| sigqueue( ) | 发送一个队列信号 queued signal |
| sigwaitinfo( ) | 等待一个信号 |
| sigtimedwait( ) | 限时等待一个信号 |

处理 POSIX 1003.1b 扩展信号和处理 POSIX 1003.1 基本信号相比，前者多了对额外信息的传递和处理，但是方法和函数基本相似。

& 因为 POSIX1003.1b 扩展定义的信号总是进行队列，因此又称为“队列信号”。

## 4.5.1 扩展信号处理函数

POSIX 为扩展信号处理函数提供了比一般定义的信号处理函数更多的信息：（1）信号来源；（2）一个整数值或者 void 型指针的附加信息。扩展信号处理函数具有如下原型定义：

```
void sigHandler ( int signo, siginfo_t * pInfo, void * pContext )
{
    ...
}
```

参数 signo 和一般信号处理函数的参数意义相同，都表示信号编号。参数 pContext 继承自 UNIX 中的函数原型定义，指向信号到来时的任务上下文。真正为应用程序控制用于表示额外的辅助信息的是第 2 个参数 pInfo，pInfo 是指向结构体 siginfo_t 的指针，POSIX 对 siginfo_t 的定义为：

```
typedef struct siginfo    /*信号信息*/
{
    int             si_signo;    /*信号编号*/
    int             si_code;     /*信号来源*/
    union sigval    si_value;    /*附加信息*/
} siginfo_t;
```

其中，域 si_code 表示信号来源。对于队列信号，si_code 可能为 SI_QUEUE、SI_TIMER、SI_ASYNCIO、SI_MESGQ 之一（见“表 VxWorks 的 POSIX 信号源定义”）。

最重要的是域 si_value，该值为信号源指定的附加信息。si_value 为联合体类型，可以表示多种类型信息，POSIX 1003.1b 要求能表示整型值和 void 型指针：

```
union sigval {    /*信号附加信息*/
    int sival_int;
    void * sival_ptr;
};
```

对于来源与 sigqueue( )发送的队列信号，si_value 由 sigqueue( )第 3 个参数指定；对于其他队列信号来源，si_value 由在为相应来源指定的信号事件 sigevent 指定。

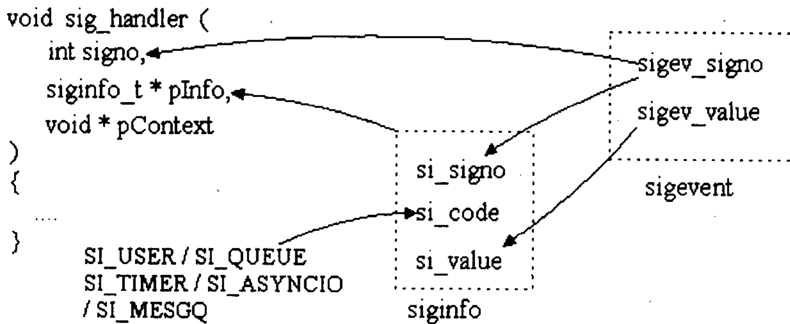



图 4-3　信号处理函数的参数传递过程

& 扩展定义的信号处理函数仍然由 POSIX 基本 OS 接口定义的 sigaction( )安装，此时必须为其指定参数 sa_flag 中的 SA_SIGINFO 标志以启用 POSIX 1003.1b 扩展的信号队列和附带信息功能。

## 4.5.2 发送队列信号

队列信号可以由程序控制一个任务显式地发送到另一个任务，也可以在其他机制下产生，下面分别介绍。

### 1. 显式发送队列信号

```
#include "signal.h"
int sigqueue ( int tid, int signo, const union sigval value );
```

该函数允许对任何任务队列多个信号。而 kill( )只能传递一个信号，即使在信号处理程序运行之前有多个信号到达。

第 1 个参数指定接收队列信号的任务 ID，第 2 个参数为所传递的对了信号编码（SIGRTMIN～SIGRTMAX），第 3 个参数 value 可以指定一个特定的额外信息，可以传递一个整型值或者 void 型指针。程序可以根据需要选择传递一个整型变量或者指向一个更复杂数据结构的指针。

函数返回 OK/ERROR 表示成功/失败。引起失败的原因包括：（1）任务 ID 无效，errno=EINVAL 或者（2）队列已经没有空间，errno=EAGAIN。

& **sigqueue( )**
POSIX 允许指定参数 signo 为 0，此时函数进行错误检查但不实际发送信号。但是 VxWorks 对这一情况返回 ERROR，errno=EINVAL。

### 2. 其他队列信号来源

POSIX 1003.1b 定义的除了通过 sigqueue( )发送的实时信号外，还有其他 3 种信号：

- POSIX 1003.1b 定时器到时；
- 异步 I/O 完成；
- 消息到达通知。

在这 3 种情况下发送信号的动作需要在各自初始化时定义一个“信号事件”的数据结构来确定，即 sigevent，VxWorks 将其定义为：

```
struct sigevent       /*信号事件*/
{
    int             sigev_signo;   /* 发送的信号 */
```

```
    union sigval  sigev_value;   /* 附加信息（si_value）*/
    int           sigev_notify;  /* 是否发送信号 */
};
```

上面，域 sigev_notify 定义是否使用信号进行通知，可以是下面两个值之一：

- SIGEV_SIGNAL —— 使用信号进行异步通知；
- SIGEV_NONE—— 不使用信号通知。

定义 sigev_notify 为 SIGEV_SIGNAL，则当定时器到时，异步 I/O 完成，消息到达空队列时将发出信号，信号编号和额外值由结构 sigevent 的另外两个域 sigev_signo 和 sigev_value 定义。如果 sigev_signo 所表示的信号属于 7 个实时队列信号（SIGRTMIN~SIGRTMAX），并且 sigaction 设置信号处理函数时指定了 SA_SIGINFO，则指定 sigev_value 为一个特定的值，该额外信息将传递给扩展定义的信号处理函数。

更多的内容参考第 3 章 3.2 节“时钟和定时器”，3.7 节“消息队列”，第 5 章 5.5 节“异步 I/O”。

### 4.5.3 队列信号处理

和 POSIX 1003.1 信号一样，队列信号也可以通过设置信号处理函数处理，或者将其阻塞、忽略等。4.5.2 中介绍的信号接口仍然可以用于队列信号处理，差别在于采用扩展信号处理函数时接收者可以得到更多的信息。

另外，POSIX 1003.1b 还引入了一种更快速处理接收信号的可选方案：

```
#include "signal.h"
int sigtimedwait ( const sigset_t * pSet, struct siginfo * pInfo, const struct
timespec * pTimeout );
int sigwaitinfo ( const sigset_t * pSet, struct siginfo * pInfo );
```

函数调用 sigwaitinfo( )sigtimedwait( )都用于同步等待信号到来。其区别在于函数 sigtimedwait( )比 sigwaitinfo( )多了超时控制功能。参数 pTimeout 指定 sigtimedwait( )的最大等待时间。如果为 NULL，其功能和 sigwaitinfo( )一样，即无限等待。

第 1 个参数 pSet 用于指定要等待的阻塞信号集。如果调用时已经有一个多个属于阻塞信号集的信号被阻塞，则编号最小的信号被返回（同时清除对应阻塞记录）。注意和前面介绍的 sigaction( )等操作不同的是，函数返回的条件是 pSet 指定的被阻塞信号到来。不在 pSet 指定的阻塞信号集内的信号将按照正常的信号处理过程被传递和处理。

第 2 个参数 pInfo 如果非空，则存储信号信息。其中 pInfo.si_signo 表示信号编号；pInfo.si_value 表示信号源指定的附加信息；pInfo.si_code 表示信号来源如表 4-8 所示：

表 4-8 PInfo.si-code 信号来源

| si_code 编码 | 信号来源 |
|---|---|
| SI_USER | 源于 kill( ) |
| SI_QUEUE | 源于 sigqueue( ) |
| SI_TIMER | 源于 timer_settime( )设置的定时器到时 |
| SI_ASYNCIO | 源于异步 I/O（AIO）完成 |
| SI_MESGQ | 源于空消息队列收到消息 |

显然，对于 kill( )发送的信号，pInfo.si_value 是没有意义的。

如果 pInfo 为 NULL，则队列信号的额外值被丢弃，但是信号编号仍然通过函数返回值得到。

两个函数在等待信号期间，如果有不属于信号集 pSet 的信号到来，并且该信号设置了用户定义的信号处理函数，则信号处理函数返回时，sigtimedwait( )和 sigwaitinfo( )将被终止并返回-1，errno=EINTR。

成功时，两个函数返回值都是信号编号，表示至少有一个属于信号集 pSet 的信号到达或者已经在调用任务中有阻塞记录。

如果失败，两个函数返回-1。对于 sigtimedwait( )，函数可能返回 errno=EAGAIN 表示超时错误。sigwaitinfo( )返回-1 时只可能有一种错误 errno=EINTR。

& sigwaitinfo( )和 sigtimedwait( )对 sigsuspend( )最大的区别在于：当信号可用时，前者只返回信号值作为结果，不需要调用已注册的信号处理函数。

sigwaitinfo( )和 sigtimedwait( )提供了“阻塞信号－检测－处理”的信号处理方法。如果期望信号以该方式被处理，应该在调用 sigwaitinfo( )，sigtimedwait( )，sigwait( )之前设置信号为阻塞。否则此前递交给任务的信号将按照一般设置信号处理函数的方式处理。也就是说，任务可以一直设置信号为阻塞，然后想要信号的时候调用函数这 3 个函数即可。

在很多情况下，“阻塞信号－检测－处理”这一处理方法具有比 sigaction+sigsuspend 提供的“捕获－处理”方法更有效率。因为系统在调用信号处理函数时需要进行上下文切换，设置信号处理环境，以及清除恢复等一系列费时的操作，这些操作甚至可能超过信号处理函数本身的运行开销。

& **sigwaitinfo( )和 sigtimedwait( )**

（1）等待的是属于其参数表示的信号集（阻塞信号集）内的信号，包括 POSIX 1003.1b 定义的扩展队列信号；

（2）要等待的信号不能是被忽略的信号（被忽略的信号轮不到判断是否阻塞就被丢弃）;

（3）如果要等待的信号设置了处理函数，则信号处理函数不会被调用，相当于被旁路。

# 4.6 信号的影响

前面已经说明，信号具有中断的特点，主要是由于信号的发生和中断一样具有异步性和随机性的特点。但是信号的发生和中断在处理上有明显不同之处。

从应用程序角度看，中断对任务是完全透明的，而信号对任务不是完全透明的。这种不透明有多方面的原因：（1）最主要的就是信号具有特定语义，在某种情况下，系统将其解释为和被其中断的任务有联系，而中断和被其中断任务没有特定联系，中断属于整个系统；（2）信号在所属任务上下文中运行，而中断在专门的中断上下文中运行。本节分析信号对被中断任务的影响。

由于信号处理程序使用被中断任务上下文，一个显而易见的影响就是 errno 可能因此被修改。VxWorks 在转向信号处理程序之前并没有保存被中断任务的 errno 状态。如果信号处理程序中涉及可能修改 errno 的调用，则信号处理程序应该在入口保存 errno，在退出时恢复。

## 4.6.1 系统调用中断

如果任务在某些低速系统调用的阻塞期间“捕捉”到一个信号，则该系统调用就被中断不再继续执行，调用返回出错，并 errno 设置为 EINTR。这其中，VxWorks 实际上将信号的发生解释为和阻塞的系统调用有联系，因此解除任务阻塞，让任务检查处理。注意信号必须是被“捕捉”的，而不是被忽略（或默认处理）的。这样做理由很简单，如果让系统猜测，那么程序要“捕捉”的信号和阻塞的系统调用之间的联系，无疑比要被忽略（或默认处理的）信号和阻塞的系统调用之间的联系强。

& “系统调用”在 UNIX 系统上用于使应用程序由用户态切换到核心态，以执行在用户态下没有权限完成的功能。在 VxWorks 系统上，所有任务和内核都运行在相同的权限下。因此，我们说 VxWorks 的“系统调用”和普通函数调用是一样的。我们这里沿用 UNIX 中的表述更多的是为了和经典说法保持一致。

我们前面介绍过的属于这一类的能被信号中断的系统调用包括：

- POSIX 消息队列调用：mq_receive( ) mq_send( )；
- POSIX AIO 调用：aio_suspend( ) ；
- 信号调用：sigsuspend( ), pause( ), sigtimedwait( ), sigwaitinfo( )；
- 任务调用：taskDelay( )；
- 定时器调用：nanosleep( ), sleep( )。

信号对上述系统调用的作用，可能正是设计所期望的，也可能是设计必须避免的。无论哪种情况，完善的系统应该充分考虑这种影响。比如，如果要避免信号对 mq_send( )的

影响，则应该：

```
try_again:
    if(mq_send()==ERROR)
    {
        if(errno==EINTR)    goto try_again;  /* 恢复被中断的消息发送 */
        else                ...              /* 其他情况 */
    }
```

在 4.3BSD 中，程序能够选择自动恢复被信号中断的系统调用，只要设置信号捕捉函数时指定相应标志。但是 POSIX 和 VxWorks 不支持，因此程序需要自己判断并恢复被中断的系统调用。

下表给出了 VxWorks 中其他能被信号中断的调用：

```
arpResolve( )
bootChange( )
cisGet( )
cpmattach( )
dcattach( )
dhcpcInit( )
dhcpcRelease( )
dhcpcShutdown( )
eexattach( )
eiattach( )
eidveattach( )
eihkattach( )
el3c90xEndLoad( )
elcattach( )
elt3c509Load( )
eltattach( )
eneattach( )
esmcattach( )
evbNs16550HrdInit( )
fdDrv( )
fei82557DumpPrint( )
fei82557EndLoad( )
fei82557ErrCounterDump( )
feiattach( )
ftpLs( )
lnattach( )
loginPrompt( )
lptDrv( )
m68332DevInit( )
muxSend( )
ncr5390CtrlInit( )
ncr710CtrlInit( )
ncr810CtrlInit( )
nicEvbattach( )
ossThreadSleep( )
rdiscTimerEventRestart( )
recv( )
recvfrom( )
recvmsg( )
ripLibInit( )
rlogin( )
rlogind( )
pccardAtaEnabler( )
pcmciaInit( )
periodRun( )
ping( )
proxyReg( )
proxyUnreg( )
rcmd( )
rdCtl( )
sendAdvertAll( )
sendmsg( )
sendto( )
shell( )
smNameFind( )
smNameFindByValue( )
smNetShow( )
smObjAttach( )
snmpIoMain( )
snmpIoTrapSend( )
snmpIoWrite( )
snmpdInitFinish( )
snmpdTrapSend( )
sntpcTimeGet( )
sym895CtrlInit( )
sym895HwInit( )
sym895Intr( )
sym895Loopback( )
telnetdExit( )
tftpCopy( )
tftpGet( )
tftpPut( )
tftpSend( )
tftpdTask( )
ultraLoad( )
```

续表

| | | |
|---|---|---|
| ftpXfer( ) | rdisc( ) | ultraattach( ) |
| gei82543EndLoad( ) | rdiscIfReset( ) | wdbNetromPktDevInit( ) |
| i8250HrdInit( ) | rdiscInit( ) | wdbSystemSuspend( ) |
| iOlicomEndLoad( ) | sched_yield( ) | wdbUserEvtPost( ) |
| ln97xEndLoad( ) | send( ) | |
| lnPciattach( ) | sendAdvert( ) | |

## 4.6.2 函数重入影响

在第 1 章第 1.4 节“多任务与函数重入”中讨论了多任务与函数重入的问题。在这里问题是：任务在运行某个函数期间被信号中断，而信号处理程序也调用了该函数，由此可能引起系统不一致。例如任务调用 malloc( )时，malloc( )在访问全局的系统堆上的数据结构期间被信号中断，信号处理程序也调用 malloc( )访问并修改了堆数据结构。这种情况下系统将出现难以预料的结果。

第一类可重入函数允许在其一个运行实例的任意点上被该函数的另一个运行实例中断而不会引起系统不一致。信号处理程序要求使用第一类可重入函数。VxWorks 文档中没有具体给出这类可重入函数。从原则上讲，应该避免调用涉及以读写方式访问静态数据结构和全局数据结构的函数，malloc( )和 free( )是此类函数的典型代表。此外，标准 I/O 函数，以及任何直接或者间接调用 malloc( )和 free( )的函数都是应该避免的。

POSIX 定义了可重入函数。下表是对应 VxWorks 中已实现的部分。

| | | | |
|---|---|---|---|
| alarm | write | read | sigfillset |
| chdir | fstat | rename | sigismember |
| close | kill | rmdir | sigpending |
| creat | lseek | sigaction | sigprocmask |
| fcntl | mkdir | sigaddset | sigsuspend |
| time | open | sigdelset | sleep |
| unlink | pause | sigemptyset | stat |
| utime | | | |

& 允许在信号处理函数中使用的可重入函数属于**异步信号安全函数**（async-signal-safe）。

另外一个问题和编译优化相关。即在信号处理函数和正常任务间共享某全局变量时，需要将变量声明为 volatile 类型。这实际上告诉编译器信号处理函数可能在任意点修改该变量，不能进行代码优化，代码优化的结果将可能使信号处理函数修改后的值和任务中的值不一致。

# 第5章 I/O系统

I/O 系统在嵌入式系统设计中占有非常重要的地位。一个嵌入式系统最终可以看成是一个与外部世界进行信息交换和控制的系统。尤其在实时系统里面，I/O 操作的可靠性和实时性成了系统设计的首要目标。在前面的章节中，我们已经讲述了 VxWorks 内核中几个重要的概念：任务、任务调度、任务间通信等。这些是 I/O 系统的基础，本章将详细介绍。

- VxWorks 中 I/O 系统层次结构；
- 文件、设备、设备驱动程序的概念；
- 基本 I/O 缓冲 I/O 格式化 I/O；
- 异步 I/O 操作（AIO）；
- 设备类型；
- 驱动程序结构。

## 5.1 I/O 系统概述

### 5.1.1 I/O 系统层次结构

VxWorks 的 I/O 系统设计采取如图 5-1 所示的层次结构。对于应用程序而言，VxWorks 的 I/O 系统是简单的，与设备无关。

应用程序访问 I/O 的接口包括：

- 基本 I/O 系统接口；
- 缓冲 I/O（ansiStdio）；
- 格式化 I/O（fioLib）；
- socket I/O（socketLib）。

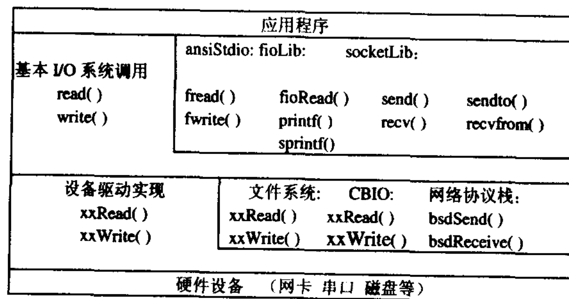



图 5-1 I/O 系统层次结构

## 5.1.2 文件、设备和驱动程序

VxWorks 中设备的分类：

- 串行终端设备；
- 管道设备；
- NFS 网络设备；
- 非 NFS 网络设备（netDrv 设备）；
- 使用 dosFs 的文件系统设备；
- 其他虚拟设备。

用户应用使用设备时，总是以一种“文件”的方式使用，即使对许多非块设备而言，一个设备恰好对应一个文件。例如对块设备，用户先创建名为“/pipe/xxx”的管道设备，然后进行 I/O 时，通过名为“/pipe/xxx”的管道设备文件进行操作。因此，对这样的设备，我们后面将根据方便用“设备”和“文件”两个概念表示。这类设备的命名也可以完全由应用程序决定，实际上，VxWorks 只是用他们的名称在设备链表中查找需要的驱动程序来完成应用程序要求的操作。

块设备通常包括目录和文件的概念，一般对应多个文件。对这类设备，用户程序能自由确定的只是设备名称，文件名称是根据文件在设备上所处的逻辑位置确定的。典型的如 DOS 文件系统设备，如表 5-1 所示。

表 5-1 设备及其命名

| 设备类型 | 创建设备 | 设备命名约定 |
| --- | --- | --- |
| 串行终端设备 | ttyDevCreate( ) | 以/tyCo/开头，后跟一个数字表示从 0 开始的串口编号 |
| 管道设备 | pipeDevCreate( ) | 以/pipe/开头，另外跟一个字符串 |
| 伪内存设备 | memDevCreate( ) | 以/开头，另外跟一个字符串 |
| NFS 网络设备 | nfsMount( ) | 以斜线加名称表示，如/usr |
| netDrv 网络设备 | netDevCreate( ) | 以远程主机名加冒号（:）表示，如 host: |
| DOS 文件系统设备 | dosFsDevCreate( ) | 以大写字母（跟数字）加冒号（:）表示，如 DEV1:<br>或者以斜线加名称表示，如/hda1<br>文件名和目录名通常以斜线间隔(/) |

使用设备的过程通常包括：（1）驱动程序初始化；（2）创建设备；（3）打开文件；（4）I/O 操作；（5）关闭文件；（6）删除设备。

每种设备都在驱动程序支持下为用户应用服务，一个驱动程序可以为多个设备服务。设备创建之前一般需要先初始化驱动程序，很多设备在系统启动时的根任务 usrRoot( )中进行这一动作。由于用户代码都在此之后，因此用户往往只涉及上面的（2）～（5）步。有

些设备在驱动程序初始化之后立即创建了设备（如 tty 设备），这样用户只感觉到（3）～（5）步。

## 1. 文件

块设备：一个设备可以实现不同的文件系统，可以有多个文件。

非块设备：一个设备通常对应一个文件。

VxWorks 应用通过打开一个命名文件访问相应的 I/O 设备。

文件名由一串字符串指定。对于无结构的设备，文件名即设备名，如串口就是无结构的设备。具有结构的设备文件名由设备名后跟随字符串表示文件名，如文件系统设备就是有结构的设备。考虑下列命名文件：

/usr/myfile —— 位于磁盘设备/usr 上的文件 myfile；

/pipe/mypipe —— 命名管道；

/tyCo/0 —— 串口。

在 VxWorks 中，它们都被称为文件，即表示完全不同的物理设备。当在 I/O 调用中指定一个文件名时，I/O 系统搜索设备，该设备名称和指定文件名至少和开始部分子串一致，然后 I/O 操作定向到该设备。

如果不能找到指定的设备名称，则 I/O 操作定向到一个默认设备。默认设备由用户程序设置。函数 ioDefPathGet( )用于读取当前默认设备路径，ioDefPathSet( )用于设置默认设备路径。可以指定任何类型的默认设备，包括空设备。

## 2. 文件类型测试

测试文件类型宏，如果属于右边的类型，则左边的宏结果为真，如表 5-2 所示。其中 mode 由取文件状态函数 fstat 或者 stat 调用得到。

表 5-2　文件类型测试

| S_ISDIR(mode) | 目录 |
|---|---|
| S_ISCHR(mode) | 字符设备 |
| S_ISBLK(mode) | 块设备 |
| S_ISREG(mode) | 常规文件 |
| S_ISFIFO(mode) | 管道 |

在下面的例子 read_file 中，我们要读取一个常规文件内容到缓冲区。在打开文件后，我们先判断文件类型是否属于常规文件，如果不是，则放弃读取。

```
#include "vxworks.h"
#include "ioLib.h"
#include "sys/stat.h"
char* read_file (const char* filename, unsigned int* length)
```

```
{
    int fd;
    struct stat file_info;
    char* buffer;
    fd = open (filename, O_RDONLY);   /* 打开文件 */
    fstat (fd, &file_info);   /* 取文件类型信息 */
    *length = file_info.st_size;
    /* 检查文件类型 */
if (!S_ISREG (file_info.st_mode))
    {
        /* 不是常规文件 关闭 */
        close (fd);
        return NULL;
    }
    /* 是常规文件 分配缓冲区 读文件 返回 */
    buffer = (char*) malloc (*length);
    read (fd, buffer, *length);
    close (fd);
    return buffer;
}
```

在上面的例子中，我们用到了一些基本 I/O 函数，这将在 5.2 节中讲述。

## 5.2　基本 I/O

VxWorks 中，I/O 分为基本 I/O 和缓冲 I/O。基本 I/O 是 VxWorks 中最底层的 I/O，提供 7 个标准 C 语言库兼容的编程接口，如表 5-3 所示。

表 5-3　基本 I/O 函数

| 函　数 | 功　能 |
| --- | --- |
| creat( ) | 创建文件 |
| delete( ) | 删除文件 |
| open( ) | 打开/创建文件 |
| close( ) | 关闭文件 |
| read( ) | 读取之前创建或者打开的文件 |
| write( ) | 写入到之前创建或者打开的文件 |
| ioctl( ) | 特殊控制 |

### 文件描述符

所有的基本 I/O 操作都引用文件描述符（fd）。文件描述符是调用 open( )或者 creat( )返回的一个整型值，其他 I/O 调用需要使用该 fd 参数。

文件描述符全局可见，两个不同的任务通过相同的文件描述符访问的将是同一个文件。VxWorks 支持有限的文件描述符，具体在 I/O 系统初始化时指定有效的 fd 数目（宏定义 NUM_FILES）。因此当文件不再使用时应该立即关闭文件，防止超出系统限制。

## 5.2.1 标准 I/O

标准 I/O 包括标准输入、标准输出、标准错误输出。在这里，标准 I/O 属于基本 I/O，因此更准确的说法应该是"标准基本 I/O"。VxWorks 为标准 I/O 保留了 3 个描述符，如表 5-4 所示：

表 5-4　标准 I/O 的文件描述符

| | |
|---|---|
| 0 | STD_IN 标准输入 |
| 1 | STD_OUT 标准输出 |
| 2 | STD_ERR 标准错误输出 |

后面会要介绍具有换出功能的标准 I/O，分别为 stdin、stdout、stderr。

程序使用标准 I/O 可以使程序简洁，并增强程序的灵活性。标准 I/O 可以被**重定向**。比如，子模块将错误信息输出到 STD_ERR，而 STD_ERR 可以根据需要重定向到一个文件或者设备，无需更改子模块。重定向分全局重定向和任务重定向。

### 1. 全局重定向

全局重定向只作用于新生成的任务。系统将自动初始化全局重定向到 console，可以在任何一个任务中随时通过调用下述函数修改全局重定向：

```
#include "ioLib.h"
void ioGlobalStdSet ( int stdFd, int newFd );  /*设置全局重定向*/
int ioGlobalStdGet ( int stdFd );            /*返回全局重定向文件描述符*/
```

上述参数中，stdFd 表示三个标准 I/O（STD_IN，STD_OUT，STD_ERR）之一；newFd 是已经通过 open 或者 create 建立的文件描述符。

系统启动时，usrRoot( )将进行下面的全局重定向初始化：

```
int     consoleFd;
```

```
#ifdef  INCLUDE_IO_SYSTEM

consoleFd = NONE;        /* 默认值：NONE*/
...
/*初始化 console：命名，打开，设置通信参数*/
#ifdef  INCLUDE_TYCODRV_5_2    /*老式 tty 驱动*/
#ifdef  INCLUDE_TTY_DEV
    ...  /*初始化 tty，以及 consoleName*/
    consoleFd = open (consoleName, O_RDWR, 0);
    (void) ioctl (consoleFd, FIOBAUDRATE, CONSOLE_BAUD_RATE);
    (void) ioctl (consoleFd, FIOSETOPTIONS, OPT_TERMINAL);
#endif
#else                          /*新 tty 驱动*/
#ifdef  INCLUDE_TTY_DEV
    ...  /*初始化 tty，以及 consoleName*/
    consoleFd = open (consoleName, O_RDWR, 0);
    (void) ioctl (consoleFd, FIOBAUDRATE, CONSOLE_BAUD_RATE);
    (void) ioctl (consoleFd, FIOSETOPTIONS, OPT_TERMINAL);
#endif /*INCLUDE_TTY_DEV*/
#endif /*INCLUDE_TYCODRV_5_2*/

/*全局重定向：STD_IN STD_OUT STD_ERR*/
ioGlobalStdSet (STD_IN,  consoleFd);
ioGlobalStdSet (STD_OUT, consoleFd);
ioGlobalStdSet (STD_ERR, consoleFd);

#endif /* INCLUDE_IO_SYSTEM */
...
#ifdef   INCLUDE_LOGGING      /*日志记录将定向输出到 console*/
logInit (consoleFd, MAX_LOG_MSGS);
# ifdef  INCLUDE_LOG_STARTUP
```

## 2. 任务重定向

在任务范围内，任务重定向覆盖全局重定向。下面是任务重定向函数定义：

```
#include "ioLib.h"
void ioTaskStdSet ( int taskId, int stdFd, int newFd );
int ioTaskStdGet ( int taskId, int stdFd );
```

参数 taskId 表示要设置重定向的任务（0 表示调用任务自己）；参数 stdFd 和 newFd 的含义和全局重定向函数中同名参数相同。

任务调用 ioTaskStdSet( )设置全局重定向之后，在此之前和之后在 ioGlobalStdSet( )中对同一个标准 I/O 的重定向设置在该任务中将不起作用。

& 标准 I/O 重定向

对标准 I/O 的重定向属于 WIND_TCB 的一部分，每当任务被调出处理器时，定义被恢复（保存），因此每个任务都拥有自己的重定向。

标准 I/O 操作和其他非标准的基本 I/O 操作相比，前者已经在系统启动时初始化，任务可以直接使用，不需要再通过 open( )得到文件描述符。后者需要通过打开得到文件描述符方可使用。

## 5.2.2 打开和关闭

进行非标准的基本 I/O 操作之前需要先打开文件创建文件描述符。文件描述符是一个 int 型变量，VxWorks 维护一个文件描述符表，记录所有打开的文件的描述符、文件名称、读写属性等信息。文件描述符表和文件描述符属于全局可以访问的资源。

打开文件调用函数 open( )实现：

```
#include "ioLib.h"
int open ( const char * name, int flags, int mode );
```

准备打开的文件或设备的名字被当作第 1 个参数 name 传递到函数中去，第 2 个参数 flags 用来定义准备对打开的文件进行的操作动作如表 5-5 所示：

表 5-5　操作动作

| O_RDONLY | 打开文件用于读取 |
|---|---|
| O_WRONLY | 打开文件用于写入 |
| O_RDWR | 打开文件进行读写 |
| O_CREAT | 创建新文件 |

函数 open 一般只用于打开已经存在的文件。仅当使用 NFS 设备时，也可以指定标志 O_CREAT 将创建新文件，此时，必须用第 3 个参数 mode 指定 UNIX 风格的文件访问权限，如：

```
fd = open ("/usr/myFile", O_CREAT | O_RDWR, 0644);
```

在 UNIX/Linux 等操作系统中，open( )的第 3 个参数 mode 属于可选参数，用于创建文

件时指定文件的访问权限的初始值。但是在 VxWorks 中，该参数必须给出，如果没有用到，则设置该参数为 0。

还可以使用 O_CREAT 将创建新文件 dosFs 和 NFS 子文件夹，这时要指定 mode 为 FSTAT_DIR。

如果 open 调用成功，则返回打开的文件描述符。如果 open 返回 ERROR，可能由于这样一些原因引起：（1）没有指定文件名 name；（2）设备不存在；（3）超出 VxWorks 打开文件数限制；（4）设备驱动程序告错。

POSIX 定义了打开文件时截平文件（O_TRUNC），VxWorks 中未获支持。

在 VxWorks 中，文件描述符属于全局的，直到被 close 关闭，都可以让所有任务访问。两个不同的任务通过相同的文件描述符访问的将是同一个文件。如果打开的文件不再使用，应该调用 close( )将其关闭，防止系统资源溢出，close( )定义为：

```
#include "ioLib.h"
STATUS close ( int fd);
```

close( )调用会终止 fd 与物理文件之间的关联，并释放在文件描述符表中占用的资源。调用 close( )后不能继续在该文件描述符上进行 I/O。释放后文件描述符和文件描述符表的空间能够重新使用。

如果 close 操作成功则返回底层设备驱动程序结果，如果出错就返回 ERROR。

**注意**

（1）调用 close 通常会使缓冲区数据刷新到设备，因此，检查 close 返回的结果有时十分重要。有的文件系统，特别是 NFS 文件系统，可能需要在关闭文件时才报告文件写操作中出现的错误。

（2）打开的文件不会因为将其打开的任务结束而被关闭，如果文件不再使用，任务必须显式地调用 close 关闭。

& 说明：UNIX 风格的文件访问权限包括读/写/执行权限（rwx），由一个八进制数（二进制为 000-111）表示。UNIX 用这样 3 个八进制数分别表示文件拥有者，同组用户，其他用户的访问权限。以上述 644 为例，6 表示文件拥有者具有读写权限，后面两个 4 分别表示同组用户和其他用户具有只读权限。

## 5.2.3 创建和删除

函数文件通过函数 creat( )完成。该函数定义为：

```
#include "ioLib.h"
int creat ( const char * name, int flag );
```

参数 name 指定新建文件名称，flag 表示读写属性，可以是 O_RDONLY/O_WRONLY/O_

RDWR 之一。

函数 creat 创建文件同时以指定标志打开文件。实际上，creat 创建文件时先确定文件所在设备，然后调用设备驱动程序提供的函数创建文件。

如果创建成功，creat 返回创建文件的文件描述符。可以通过该描述符进行文件读写控制等操作。如果返回 ERROR，可能由于这样一些原因引起：（1）没有指定文件名 name；（2）设备不存在；（3）超出 VxWorks 打开文件数限制；（4）设备驱动程序告错。

在非文件系统设备上调用 creat 相当于调用 open。

较早版本的 VxWorks 调用 delete 删除文件，在 VxWorks V5.x 中，删除文件通过 remove 实现。调用 delete 在 ioLib.h 中是一个条件宏__DELETE_FUNC 定义：

```
#if  __DELETE_FUNC
#include "stdio.h"
#define delete(filename)  remove(filename)
#endif
```

在非文件系统的设备上调用文件删除没有作用。

函数 remove 定义为：

```
#include "ioLib.h"
STATUS remove ( const char * name );
```

参数 name 指定要删除的文件名称。和 creat 一样，remove 先定位文件所在设备，然后调用设备驱动程序提供的函数删除文件。

如果文件已经打开，先关闭文件再删除。

VxWorks 另外提供一个 POSIX 兼容的删除文件的函数 unlink( )，unlink( )完成和 remove( )一样的功能。

### 5.2.4 读写

这里将要讲的读写操作为 read( )和 write( )调用，属于基本 I/O 的读写操作。有别于后面要介绍的缓冲的 I/O 和异步 AIO，随着每次对基本 I/O 的读写的调用，调用任务被阻塞，直到 I/O 完成：对于 read( )调用，直到系统将要求读的数据从设备读到指定缓冲区，或者对于 write( )，直到指定输出的数据写到设备或者驱动程序将数据加入设备队列，调用任务才返回（驱动程序返回错误时也使任务返回）。

调用 read( )和 write( )之前需要先通过 open( )或者 creat( )得到文件描述符。

函数 read( )定义为：

```
#include "ioLib.h"
```

```
int read ( int fd, char * buffer, size_t maxbytes );
```

其参数为：文件描述符 fd，读取数据缓冲区指针 buffer，最大读取字节数 maxbytes。

对于文件系统设备，如果文件剩下的长度小于 maxbytes，函数返回的实际读取字节数小于 maxbytes，如果继续调用 read( )，返回值将等于 0，表示文件结束（EOF）。对于非文件系统设备，虽然文件大小没有限制，但是函数返回的实际读取的字节数仍然可能小于请求的字节数 maxbytes，在这种情况下，随后调用 read( )的返回值可能等于 0，也可能大于 0。因此，在串口和网卡 I/O 中，可能需要多次调用 read( )来读取某个大小的数据（参考[WRS-oslib5.5]对 fioRead( )的说）。

上面的返回值（0～maxbytes）属于成功时的情况，如果出错，函数返回 ERROR（-1）。可能的原因有：文件描述符错，文件没有以只读方式打开，设备（驱动程序）不支持 read 及驱动程序返回 ERROR。对于设备不支持 read 的情况，将设置 errno 为 ENOTSUP。

函数 write( )定义为：

```
#include "ioLib.h"
int write ( int fd, char * buffer, size_t nbytes );
```

其参数为：文件描述符 fd，要输出数据所在缓冲区指针 buffer，要求写的字节数 nbytes。

函数 write( )返回时数据是否实际写到了设备属于驱动程序的细节，在不同的设备中实现有差异。VxWorks 保证 write( )返回时要输出的数据至少加入了设备的输出队列。其返回值表示所写的字节数，通常应该等于 nbytes；如果等于 0，表示没有写入任何数据；如果介于 0 和 nbytes 之间，可能是由于底层设备驱动程序对写入的数据块大小要求比较严格。

如果调用时给定的参数错误，如 fd 无效或不支持 write 操作，或者驱动程序返回错误，write( )将返回 ERROR（-1）。

### 5.2.5 文件截平

POSIX 1003.1b 定义了文件截平操作（Ftruncate）。VxWorks 的 DOS 文件系统支持库 dosFsLib 对 POSIX 1003.1b 的规定提供了有限的支持。

文件截平操作通过函数 ftruncate( )实现，定义为：

```
#include "ioLib.h"
int ftruncate ( int fildes, off_t length );
```

参数 fildes 是需要截平的文件的文件描述符，参数 length 为截平后文件长度。成功时函数返回 OK。如果失败，函数返回 ERROR，并设置 errno 表示错误类型。如表 5-6 所示参数定义：

表 5-6　参数定义

| | |
|---|---|
| EROFS | 文件位于只读文件系统 |
| EBADF | fildes 是以只读方式打开的 |
| EINVAL | 参数无效 |

使用 ftruncate 需要定义组件 INCLUDE_POSIX_FTRUNCATE。

### 5.2.6　I/O 控制

函数 ioctl( )提供了一个对设备、文件描述符、驱动程序的配置工作等方面进行控制的接口。函数定义为：

```
#include "ioLib.h"
int ioctl ( int fd, int function, int arg );
```

ioctl( )对描述符 fd 指定的对象执行在 function 中给出的操作，可以通过 arg 指定该操作的一个整数参数。实际中，不同的设备和厂商对 function 和 arg 有不同的定义，POSIX 标准亦无法对众多的设备逐一作出规范。在 VxWorks 中，ioctl 属于与特定驱动程序（tyLib、pipeDrv、nfsDrv、dosFsLib、rt11FsLib、rawFsLib）相关的细节。

## 5.3　I/O 复用（Select）

在有些情况下，程序需要等待多个 I/O，这些 I/O 完成的先后顺序不能确定。在这种情况下，先阻塞在任何一个 I/O 上都可能延迟对其他先到来的 I/O 的响应。考虑一个 client-server 的例子：两个 client 通过 I/O 发出服务请求，一个采用管道，一个采用 socket，server 必须尽快予以响应。如果 server 在管道上阻塞，等待请求到来，则如果其间有 socket 请求 server 将无法处理，直到在管道上的阻塞解除，从而延缓了 I/O 响应时间；反过来也是一样。

VxWorks 提供与 BSD 4.3 兼容的 **I/O 复用**（Select）功能，或称**等待复用**，使任务可以同时监视多个 I/O。在上面的问题中，server 任务可以同时阻塞在管道和 socket 上，当其任意一个到来时，server 立即予以响应。

I/O 复用是一个非常实用的功能，相对于其他 I/O 操作也更复杂。

#### 1. 初始化

该功能具体由库 selectLib 实现。用户程序在任务中使用 selectLib 使任务同步能等待多个设备 I/O，并指定最大等待时间。对于驱动 I/O 设备驱动而言，通过使用 selectLib，驱动

程序支持能检测在其上阻塞的任务，这样驱动程序（ISR）可以直接唤起阻塞的任务。

使用库 selectLib 需要引用头文件 selectLib.h，并定义宏 INCLUDE_SELECT。函数 selectInit( )初始化库 selectLib，一般在 usrRoot( )中调用：

```
#ifdef  INCLUDE_SELECT
   selectInit (NUM_FILES);
#endif
```

用户程序在某文件描述符上使用 select 必须有对应设备驱动提供的 select 支持。下面的介绍属于如何在用户任务中使用 selectLib 实现等待多个设备。驱动程序中使用 selectLib 实现 select 支持在 5.7 节“I/O 系统内部结构”中介绍。

## 2. 文件描述符集 fd_set

函数 select( )使用文件描述符集（结构体 fd_set）表示要在哪些文件上等待；函数结束时等待的结果也通过文件描述符集有哪些等待的文件就绪。

文件描述符集其实就是一个通过一个整型数组表示的一个“超长的”整型变量，变量的每一位表示一个文件描述符：即文件描述符集的第 n 位表示的文件描述符为 n。如图 5-2 所示，阴影部分表示 1，空白部分表示 0，则图中表示的 fd_set 包括文件描述符为 5，8，10 的文件。

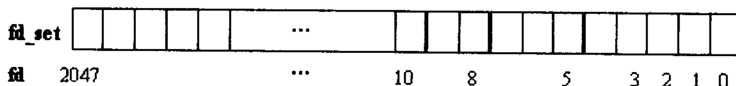



图 5-2 fd_set

很显然，文件描述符集必须有足够多的位数，使打开的文件描述符非常大时也可以表示。具体的位数由宏 FD_SETSIZE 定义，默认情况下，一个文件描述符集包括 2048 位，即允许的最大文件描述符为 2047，通常这足以满足应用需求。修改该定义要求大于 iosInit( ) 指定的允许同时打开的文件数。

所有对 fd_set 的操作都由 VxWorks 定义的宏进行，如表 5-7 所示：

表 5-7 文件描述符集操作宏

| | |
|---|---|
| FD_CLR(fd, &fdset) | 清除文件描述符集中对应 fd 的位 |
| FD_SET(fd, &fdset) | 设置文件描述符集中对应 fd 的位 |
| FD_ISSET(fd, &fdset) | 如果文件描述符集中对于 fd 的位置位，宏结果等于 true，否则 false |
| FD_ZERO(&fdset) | 清除文件描述符集中所有的位 |

下面是一个一般化过程的示例，在文件描述符集 read_set 设置一个文件描述符 fd，然后 I/O 复用，检测结果。

```
FD_ZERO( &read_set );
FD_SET ( fd, &read_set );
if ( select (fd+1, &read_set, NULL, NULL, timeout_period) )  /*等待*/
if ( FD_ISSET(fd, &read_set) )    /*检测等待结果*/
    ...
```

I/O 复用调用 select( )实现，定义为：

```
#include "selectLib.h"
int select (
    int        width,           /* 文件描述符集有效部分大小 */
    fd_set *   pReadFds,        /* 读文件描述符集 */
    fd_set *   pWriteFds,       /* 写文件描述符集 */
    fd_set *   pExceptFds,      /* VxWorks 未支持 */
    struct timeval * pTimeOut    /* 最大等待时间 */
);
```

第 2、第 3 个参数为指向文件描述符集的指针，表示 select( )要等待的文件描述符集，pReadFds 表示等待 read 的文件描述符集，pWriteFds 表示等待 write 的文件描述符集。select( )允许同时等待读和写文件。

参数 width 在数值上应该等于最大的等待集中文件描述符加 1，实际上系统借此来判断 fd_set 中要等待的位映射最大宽度。如果传递的值小于应该的值，则会出现漏等待。

pTimeOut 指定最大等待时间，为 NULL 表示不限制等待时间。

select( )不实际进行 I/O 复用。

select 返回结果分两部分，如表 5-8 所示：

表 5-8　select 返回信号结果

| | |
|---|---|
| 函数返回值 | （1）等待满足时：返回就绪的文件描述符个数，包括读和写两部分<br>（2）超时时：返回 0<br>（3）出错时：返回 ERROR（-1） |
| 入参指定的读写文件描述符集 | 当函数返回值大于 0 时，入参指定的 fd_set 被修改以反映就绪的文件描述符，即：如果某 fd 满足等待条件，则其在文件描述符集中的对应位被置位 |

上面的“就绪”对 read( )来说，通常代表设备驱动程序的数据缓冲区中有有效输入数据；对 write( )来说，通常代表设备驱动程序缓冲区数据输出缓冲区有空间。

select( )出错原因有下面几种，如表 5-9 所示：

表 5-9 出错原因

| | |
|---|---|
| S_selectLib_NO_SELECT_SUPPORT_IN_DRIVER | 至少有一个 fd 对应的底层设备驱动不支持 select |
| S_selectLib_NO_SELECT_CONTEXT | 创建任务时没有初始化 select 环境 |
| S_selectLib_WIDTH_OUT_OF_RANGE | 参数 width 超出最大可能值 |

**select( )**

（1）当 select( )返回并指示某 fd 就绪时，仍然存在 read/write 在该 fd 上阻塞的可能。这取决于驱动程序细节。但是 select( )帮助避免出现长时间等待 I/O 的现象。当同时属于 select( )的读文件描述符集和写文件描述符集的 fd 就绪时，该 fd 被计算两次；

（2）在块设备上使用 select( )总是返回 fd 就绪的结果，通常只对非块设备如 tty、socket 等进行 select；

（3）I/O 复用的实现在于驱动程序的支持：支持 select 的驱动程序为每个设备实现了一个任务唤醒列表，当调用 select( )时，系统将任务添加到对应设备的任务唤醒列表。这样，当驱动程序接收到数据或者发送缓冲区空时，就可以唤醒相应的阻塞任务。如果发生超时，则由系统解除 select( )调用的阻塞，并返回超时的结果 ETIMEOUT。

## 3. I/O 复用示例

```
...
tv.tv_sec =5000;
tv.tv_usec = 0;
while(1)
{
    error = select (socket+1, &readfds, NULL, NULL, &tv);
    switch (error)
    {
    case 0:
        printf("timeout!\n");
        break;
    case -1:
        printf ( "error!\n" );
        break;
    default:
        if ( FD_ISSET ( socket, &readfds ) )
        {
            numbytes = recv ( socket, buf, MAX_DATA_SIZE, 0 );
```

```
            if ( numbytes > 0 )
                printf("recv: %s received\n", buf);
        }
    }
}
```

# 5.4 其他 I/O

除了基本 I/O，VxWorks 还提供 ANSI 标准库 stdio 功能。但是 ANSI 标准的 stdio 通过两个组件实现：

- 缓冲的 ansiStdio，提供除了下面的函数外其他 ANSI 标准库 stdio 定义的函数。VxWorks 程序使用 ansiStdio 组件需要定义 INCLUDE_ANSI_STDIO，引用文件是 stdio.h；
- 格式化 I/O fioLib，提供 ANSI 标准 stdio 中的 printf( )，sprintf( )，vprintf( )，vsprintf( )，sscanf( )以及其他两个 VxWorks 扩展定义的 I/O 函数。使用 fioLib 需要定义 INCLUDE_FORMATTED_IO，引用文件为 fioLib.h 和 stdio.h。

## 5.4.1 缓冲 I/O（ansiStdio）

在内部实现上，库 ansiStdio 通过在内部维护一个缓冲区，以较大的单位和底层设备驱动缓冲区进行数据交换，以减少访问驱动程序缓冲区次数的方式提高 I/O 效率。在用户应用看来，ansiStdio 将不同设备类型的 I/O 封装成逻辑上一致的数据流形式来访问。对应于基本 I/O 调用的缓冲 I/O 调用接口如下表。

```
#include "stdio.h"
FILE * fopen ( const char * file, const char * mode );
int fread ( void * buf, size_t size, size_t count, FILE * fp );
int fwrite ( const void * buf, size_t size, size_t count, FILE * fp );
int fclose ( FILE * fp );
```

ansiStdio 行为受数据流的形式和缓冲区设置的影响。

### 1. 数据流的两种实现形式：

- 文本数据流：由一系列的 line 组成，每个 line 包括 0 个或多个字符，并由‘\n’结束。输入输出时 ansiStdio 需要对数据进行处理，比如将‘\n’映射成‘CR+LF’；
- 二进数据流：由未经处理的原始数据组成。

用户程序通过 ansiStdio 打开一个文件（Fopen）时，就将一个数据流连接到了该文件上，直到关闭文件（Fclose）。ansiStdio 用结构 FILE 维护所有通过该数据流进行 I/O 的信息，打开该文件 fopen( )时返回指向该结构的指针。fopen 定义为：

FILE * fopen ( const char * filename, const char * mode );

入参 mode 指定文件打开方式：

- 文本数据流方式："r"，"r+"，"w"，"w+"，"a"，"a+"。
- 二进数据流方式："rb"，"rb+"，"wb"，"wb+"，"ab"，"ab+"。

其中 r 表示读，w 表示写，a 表示在文件尾部追加，+表示更新（在连续的读写操作之前必须进行 fflush 或者文件定位操作）。

## 2. 缓冲区设置

ansiStdio 可以实现 3 种缓冲方式：

- IOFBF 全缓冲，打开文件建立数据流时的默认设置。用户程序以任意大小进行输入输出，ansiStdio 将字符缓冲直到填满一个块才在底层驱动之间传递；
- IOLBF 底层驱动和 ansiStdio 之间的输入输出以行为单位进行，一行以‘n’为标志；
- IONBF 无缓冲。ansiStdio 将用户程序输入输出的字符立即在 ansiStdio 和底层驱动之间进行传递。

函数定义：

```
#include "stdio.h"
int setvbuf ( FILE * fp, char * buf, int  mode, size_t size );
```

该函数设置数据流 fp 的缓冲区大小 size，以及缓冲模式 mode：IOFBF/IOLBF/IONBF。

调用者可以不指定缓冲区（传递指针 buf=NULL），setvbuf( )会自动申请 size 大小的缓冲区，否则 setvbuf( )使用 buf 表示的缓冲区。

函数返回 0 表示成功或者非零结果表示出错。

**注意**

*setvbuf 必须在任何读写等操作之前调用；*

*setvbuf 指定的缓冲区为 stdio 使用，在调用任务看来，缓冲区中的内容在任何时候都是不确定的。*

另外一个函数：

```
#include "stdio.h"
void setbuf ( FILE * fp, char * buf );
```

提供 setvbuf( )的部分功能。如果为调用 setbuf( )指定指针 buf 为 NULL，则 fp 的被设置为无缓冲模式，否则相当于调用 setvbuf (fp, buf, IOFBF, BUFSIZ)，因此调用之前必须 buf=malloc(BUFSIZ)或者定义 char buf[BUFSIZ]。

### 3. 缓冲基本 I/O

ansiStdio 对基本 I/O 中文件描述符为 0，1，2 的标准 I/O 进行了封装，封装后在程序中表示为 stdin，stdout，stderr。

每个任务都有一套独立的缓冲基本 I/O，用户任务使用 stdin/stdout/stderr 进行读/写操作时，系统会自动建立 FILE 结构，包括缓冲区，不需要显式地调用 fopen( )。该任务结束时 stdin/stdout/stderr 占用的系统资源自动释放。

### 4. 使用 ansiStdio 注意事项

（1）打开的数据流属于打开的任务，多个任务可以同时打开一个文件。为了提高效率，ansiStdio 不进行数据流的互斥访问。因此如果多个任务同时在同一个数据流上进行输出可能会导致错误结果；

（2）ANSI 定义当控制数据流的指向 FILE 的指针生存期结束时将自动关闭文件，并将输出缓冲区未刷新的数据进行刷新。但是 VxWorks 中这一工作必须由用户程序显式地完成（stdin/stdout/stderr 除外）。因此，程序员必须记住及时关闭文件，以避免超出系统同时打开文件的限制，要考虑的关闭文件的场合包括因为收到 kill( )发出信号而结束任务的情况，参见第 4 章“信号”。另一方面，VxWorks 这一特性使得一个任务打开的文件可以在该任务退出后被另一个任务继续使用；

（3）由于（2）中的原因，ANSI 定义的创建临时文件 tmpfile( )没有实现（VxWorks 无法关闭临时文件）；

（4）如果程序想让一个任务打开的文件被另一个任务继续使用，可以复制一个指向 FILE 结构的指针。但不要试图复制 FILE 结构本身。

## 5.4.2 格式化 I/O（fioLib）

fioLib 提供的格式化 I/O：printf( )，sprintf( )，vprintf( )，vsprintf( )，sscanf( )，和 ANSI 标准库 stdio 不同的是，fioLib 不进行缓冲。fioLib 因为不进行缓冲，因此目标码节省很多。如果只要上述几个函数就可以满足应用需要，则可以不包含 ansiStdio 组件，VxWorks 通过它来将生成的执行映像精简。

注意另有几个格式化 I/O 函数：fprintf，vfprintf，fscanf，scanf 仍然在 ansiStdio 中实现，使用它们需要 INCLUDE_ANSI_STDIO。

库 fioLib 另外提供几个扩展的格式化 I/O 函数，如表 5-10 所示：

表 5-10　扩展的格式化 I/O 函数

| | |
|---|---|
| printErr( ) | 格式化输出到标准错误输出（fd=2） |
| fioFormatV( ) | 转换格式化字符串 |
| fdprintf( ) | 相当于增加了格式化功能的基本 I/O 中的 write |

续表

| | |
|---|---|
| vfdprintf( ) | 类似 fdprintf，但是采用一个参数列表 va_list |
| fioRead( ) | 相当于 read |
| fioRdString( ) | 相当于 read，内部采用行模式 |

后面 4 个函数采用一个文件描述符 fd 指示输入输出的文件，该 fd 可以通过前面介绍的基本 I/O 中的 open( )或 create( )得到。

**关于浮点数**

如果要在 printf/scanf 等格式化 I/O 函数中输入或者输出浮点数（%e，%E，%f，%g，%G），需要用到另一个浮点 I/O 库：floatLib。该库必须在进行浮点 I/O 之前初始化。库 floatLib 提供初始化函数 floatInit( )，如果定义 floatLib 的宏 INCLUDE_FLOATING_POINT，该函数将在根任务 usrRoot( )中被调用。

### 5.4.3　消息记录（logLib）

VxWorks 为应用提供消息记录机制，属于另一种 I/O 机制，如图 5-3 消息记录所示。库 logLib 内部维护一个全局的消息队列，通过初始化时生成一个任务 logTask 监听该队列，将队列的每条消息输出到指定的文件 fd。

库 logLib 的最大特点是对中断 ISR 中的 logMsg 调用，调用不会阻塞（实际上也不能阻塞，阻塞需要任务环境，这正是 ISR 所没有的）。ISR 不能进行可能引起调用者阻塞的其他 I/O 调用。

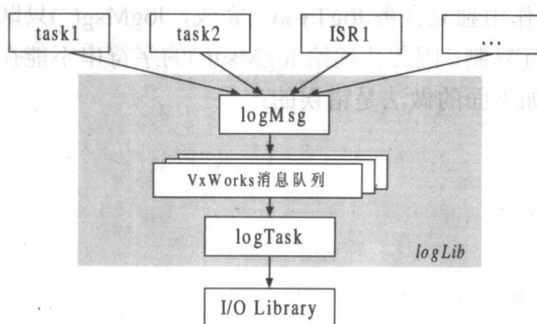



图 5-3　消息记录

使用 logLib 需要定义 I/O 系统组件 INCLUDE_LOGGING。如果定义，系统在 usrRoot( )调用 logLib 的初始化程序：

```
logInit (consoleFd, MAX_LOG_MSGS);
```

宏常数 MAX_LOG_MSGS 指定了消息队列大小，默认定义为 50。

logInit 初始化时 consoleFd 指定默认的记录文件为控制台（使用终端设备 tty 时，

consoleFd 默认定义为第 1 个串口）。用户可以修改 logInit 中对默认记录文件的定义，或者在运行时调用 logFdSet 动态的设置：

logFdSet ( fd );　/* fd 是通过基本 I/O 调用 open 或者 creat 得到 */

其中，fd 通过基本 I/O 调用 open 或者 creat 得到。上述动态设置时 logLib 会将之前使用的 fd 从记录文件列表中移除（但不关闭）。还可以调用 logFdAdd( )和 logFdDelete( )向列表中添加或者删除用于输出记录的文件 fd。

## logMsg( )

程序调用 logMsg( )实现消息记录，在 5.1 版本以前 logMsg 采取可变参数列表形式：

```
include "logLib.h"
int logMsg (char *fmt, ...);
```

其中格式参数 fmt 和 printf 中的格式化参数相似。该定义在某些体系下存在 bug，从 5.1 beta2 开始，logMsg 定义为：

```
include "logLib.h"
int logMsg (char *fmt, int arg1, int arg2, int arg3, int arg4, int arg5, int
arg6);
```

即固定采用 6 个整型参数。VxWorks 仍然提供兼容老版本的定义，在包含头文件 logLib.h 之前定义宏__PROTOTYPE_5_0 即可。

logMsg( )判断调用环境，如果是中断，函数立即返回，这样在队列已满的情况下将发生记录丢失；如果是从任务环境中调用，则 logMsg( )调用将被阻塞等待队列空闲。因为格式化和输出到 fd 的动作由独立任务 logTask( )完成，logMsg( )只队列要输出的参数信息，没有对结果字符串进行复制，因此传递给 logMsg( )的字符串不能在调用者的任务栈或者中断栈上动态分配，例如下面的做法是错误的：

```
/*错误的做法*/
    logger(which)
    {
        char string [100];
        strcpy (string, which ? "hello" : "goodbye");
        ...
        logMsg (string);
    }
```

除非 logTask( )进行格式化时，函数 logger( )栈中内容依然存在，logTask 才能正确记录。在 logger( )返回后，程序无法保证这一点。如果类似上面这段程序，可以简单地将 string 定义为 char *类型，并且将 strcpy 调用改为指针赋值，即：

```
/*正确的做法*/
    char *string;
    string = which ? "hello" : "goodbye";
```

便可以实现正确的记录。

然而，现实中程序要记录的往往并非编译时就能确定的静态字符串，这时只能定义为字符数组。很自然就能想到的方法就是：

（1）将 string 定义为全局数组；

（2）定义 string 时加上 static 修饰符，static char string[100]；

（3）用 malloc 分配（ISR 中不能这样做）。

然后将要记录的字符串复制到字符数组。

不幸的是这样并非万事大吉。因为调用 logMsg 的程序无法知道 logTask 什么时候完成记录，如果程序需要继续使用缓冲区 string，将冒一定风险。

如果 logTask 将记录输出到低速的串口，或者调用 logMsg 的任务优先级非常高，并频繁地调用 logMsg，上述情况将会很明显，终端上显示的记录看起来像是乱七八糟的。

## 5.5　异步 I/O（AIO）

同步 I/O 表示任务必须等待 I/O 操作。对于读操作，调用任务阻塞，直到驱动程序将任务要求读的数据写到缓冲区，或者出错。对于写操作，直到任务指定的数据加入到设备写队列。异步 I/O 时，程序可以在下达 I/O 操作后继续运行，当需要时检查结果或者采取其他某种机制来使自己知道 I/O 操作完成。异步 I/O 消除了同步 I/O 的一些不必要的等待。

VxWorks 的 AIO 符合 POSIX 1003.1b 标准。使用 AIO 要求定义组件 POSIX AIO 组件（INCLUDE_POSIX_AIO）。同时还要定义 POSIXAIO 驱动程序组件（INCLUDE_POSIX_AIO_SYSDRV），该组件为不具有 AIO 功能的 VxWorks 驱动程序提供 AIO 能力。

VxWorks POSIX AIO 组件对应的库是 aioPxLib。

VxWorks POSIX AIO 驱动程序组件对应库 aioSysDrv，aioSysDrv 提供了独立于任何特定设备的 AIO 请求队列，使任何设备驱动程序都可以使用 AIO。包含该组件时，将自动调用初始化函数：

```
STATUS aioSysInit ( int numTasks, int taskPrio, int taskStackSize );
```

该程序在库 aioPxLib 初始化之后初始化 aioSysDrv。该函数生成 numTasks 个系统 I/O 任务，新任务的优先级和栈大小分别由 taskPrio 和 taskStackSize 指定。3 个参数的默认值在 aioSys Drv.h 中定义，分别为 AIO_IO_TASKS_DFLT，AIO_IO_PRIO_DFLT 和 AIO_IO_STACK_DFLT。

编程时引用的 AIO 头文件为 aio.h。

### 5.5.1 AIO 控制块

系统执行异步 I/O 操作时需要一些控制信息和缓冲区，POSIX 1003.1b 通过结构 aiocb 定义这些控制信息。VxWorks 扩展了 aiocb 的定义：

VxWorks 对 aiocb 的定义为：

```
#include "aio.h"
struct aiocb    /*AIO 控制块*/
{
        int                 aio_fildes;
        off_t               aio_offset;
        volatile void * aio_buf;
        size_t              aio_nbytes;
        int                 aio_reqprio;
        struct sigevent aio_sigevent;
        int                 aio_lio_opcode;
        AIO_SYS             aio_sys;
};
```

任务提交每个异步 I/O，都必须为 aiocb 和数据缓冲区分配空间并初始化，传递给异步 I/O 调用函数。直到任务调用 aio_return，系统都需要引用 aiocb 及其所设定的缓冲区，应用程序不能修改 aiocb 数据结构。调用 aio_return 使 aiocb 被释放，释放后可以被修改或者被另一个异步 I/O 使用。因此，如果任务结束之前 I/O 不会完成，则不能在该任务的栈空间分配上述空间，也就是不能定义为局部动态变量。通过动态存储分配函数从系统内存池分配空间比较好。

对 aiocb 的说明：

- aio_fildes 通过 open 调用得到的文件描述符。异步访问一个文件之前，和其他 I/O 一样，先必须通过 open 调用打开文件；
- aio_offset 异步 I/O 操作相对于文件开始处的偏移量。VxWorks 打开文件时不支持追加方式 O_APPEND，可以通过该字段指定文件偏移量。注意，和 read/write 操作不同，异步 I/O 不会使偏移量自增，调用者必须记住文件的正确位置；
- aio_buf 异步 I/O 的缓冲区地址；
- aio_nbytes 要读/写的字节数；
- aio_reqprio 用于降低 AIO 操作的优先级。AIO 操作的优先级用于控制在对同一个文件上有多个 AIO 请求时，谁先被执行。默认时，每个 AIO 请求都被赋予了发出此 AIO 请求的任务所具有的优先级。该参数允许在此先级的基础上将 AIO 请

求优先级降低参数指定的级数。该参数可以在0到AIO_PRIO_DELTA_MAX之间。如果据此参数计算得出的AIO优先级小于最低有效任务优先级，则使用最低有效任务优先级作为AIO优先级；

- aio_sigevent 可选项，表示完成AIO信号事件（标志，信号编号和附加参数）。如果指定了一个有效的信号事件，则异步I/O完成时将发出信号；
- aio_lio_opcode 调用 lio_listio 时执行的操作。valid entries include LIO_READ, LIO_WRITE, and LIO_NOP；
- aio_sys WRS 增加定义。系统用于控制访问AIO控制块以及维护错误状态（返回信息），用户不能修改，也不必直接引用。

## 5.5.2 AIO 函数

VxWorks 通过库 aioPxLib 提供 POSIX 异步 I/O 函数。异步访问一个文件时，和同步方式一样，先调用 open 得到文件描述符，然后初始化和传递一个 aiocb 参数给下列异步 I/O 函数，如表 5-11 所示：

表 5-11 异步 I/O 函数

| | |
|---|---|
| aioPxLibInit( ) | 初始化库 aioPxLib |
| aio_read( ) | 请求一次 AIO 读操作 |
| aio_write( ) | 请求一次 AIO 写操作 |
| lio_listio( ) | 请求多个 AIO 操作 |
| aio_suspend( ) | 等待 AIO 操作结果 |
| aio_error( ) | 返回 AIO 操作错误状态 |
| aio_return( ) | 返回 AIO 操作返回状态 |

当包含 aioPxLib 时，函数 aioPxLibInit(int lioMax)将自动调用，参数 lioMax 初始化最多允许同时进行的 lio_listio( )调用。默认时为 MAX_LIO_CALLS。该宏定义为 0，此时实际上将用 AIO_CLUST_MAX 初始化。头文件 aioPxLibP.h 将其定义为：

```
#include "aio.h"
#define AIO_CLUST_MAX  100
```

**（1）aio_read**

函数定义：

```
#include "aio.h"
int aio_read ( struct aiocb * pAiocb );
```

aio_read 向系统提交一次 AIO 读操作请求，该次 AIO 操作由参数 pAiocb 指向的 AIO 控制块表示。

该函数异步读取文件 aio_fildes 中从偏移量 aio_offset 开始的 aio_nbytes 个字节，写到 aio_buf 所表示的缓冲区。aio_reqprio 表示根据调用任务的优先级减去 aio_reqprio 作为异步读取操作的优先级。aio_sigevent 定义系统完成 AIO 时向任务发送的信号，如果为 NULL，则不会发送任何信号。

如果操作请求成功的队列到设备，aio_read 就返回 OK。系统实际完成操作的情况不能立即知道。程序可以在需要知道系统完成结果时调用 aio_error( )或者 aio_return( )。后面介绍这两个函数。

如果出错，aio_read 返回 ERROR，并设置 errno 表示错误类型，如表 5-12 所示：

**表 5-12　错误类型**

| | |
|---|---|
| EAGAIN | 系统资源溢出，无法将 AIO 加入系统队列 |
| EINVAL | AIO 控制块中设置的初始信息有错误 |
| EBADF | AIO 控制块中的文件描述符 aio_fildes 不能用于读操作 |

**（2）aio_write**

函数定义：

```
#include "aio.h"
int aio_write ( struct aiocb * pAiocb );
```

aio_write 向系统提交一次 AIO 写操作请求。其中，AIO 写操作由参数 pAiocb 指向的 AIO 控制块表示。aio_fildes 表示要写入的文件，aio_fildes 表示控制文件中开始写的偏移量，aio_buf 表示要写的源数据缓冲区地址，aio_nbytes 表示字节数，aio_reqprio 表示控制从调用任务的优先级减去 aio_reqprio 作为写操作的优先级，aio_sigevent 表示定义系统完成 AIO 时向任务发送的信号，如果为 NULL，则不会发送任何信号。

和 aio_read 一样，如果操作请求成功地队列到设备，aio_write 返回 OK，如果出错，aio_write 返回 ERROR。返回 ERROR 时，将设置 errno 表示错误类型，如表 5-13 所示：

**表 5-13　错误类型**

| | |
|---|---|
| EAGAIN | 系统资源溢出，无法将 AIO 加入系统队列 |
| EINVAL | AIO 控制块中设置的初始信息有错误 |
| EBADF | AIO 控制块中的文件描述符 aio_fildes 不能用于写操作 |

**（3）lio_listio**

函数定义：

```
#include "aio.h"
```

```
int lio_listio ( int mode, struct aiocb * list[],
        int nEnt, struct sigevent * pSig );
```

调用 lio_listio( )可以一次提交多个 AIO 读写操作，能够提交的操作数受 aioPxLibInit( )初始化设置的限制。参数 list 是一个指向一组 AIO 控制块的指针，该组 AIO 控制块的个数由参数 nEnt 表示。POSIX 没有对多个操作被提交的先后次序作说明。

lio_listio 提交的多个 AIO 可以同时存在读请求和写请求。lio_listio 使用每个 AIO 控制块的 aio_lio_opcode 字段表示该 AIO 的操作类型：设定为 LIO_READ 和 LIO_WRITE 时分别表示进行读和写的 AIO 请求。允许的 aio_lio_opcode 还可以为 LIO_NOP，此时该 AIO 将被忽略。

实际上，函数 lio_listio( )工作方式分为同步方式和异步方式，通过第 1 个参数 mode 指定。

如果参数 mode 为 LIO_NOWAIT，则 lio_listio( )以异步方式工作，在系统将所有的 AIO 请求加入到队列后便返回，相当于为每个 AIO 请求调用 aio_read( )或者 aio_write( )。在异步方式下时，如果第 4 个参数非 NULL，则系统完成所有的 AIO 后将向任务发送参数 pSig 所指定的信号。

当 mode 为 LIO_WAIT 时，lio_listio( )工作在同步方式下，直到所有的 AIO 操作完成函数才返回。同步方式时忽略对参数 pSig 的指定。

在异步方式时，如果所有的操作都被队列，lio_listio 返回 OK，否则返回 ERROR。

在同步方式时，lio_listio 等待所有的 AIO 完成，然后返回 OK。如果出错则返回 ERROR，这时应该通过 aio_error( )和 aio_return( )检查每个 AIO 操作执行结果。

当 lio_listio 返回 ERROR 时，进一步的错误信息通过 errno 指示，如表 5-14 所示：

**表 5-14 错误信息**

| | |
|---|---|
| EAGAIN | 系统队列容量限制，lio_listio( )不能将所有 AIO 操作队列。可能由于有较多的未完成的 AIO 占据了队列。如果同步方式下出现这种情况，lio_listio( )将等待其他 AIO 完成。在使用异步方式时，POSIX 未明确规定该结果出现的时机，可以在调用 lio_listio( )时得到该结果，或者在系统实际处理这些 AIO 操作时得到该结果 |
| EINVAL | 一个或者多个参数指定了无效值 |
| EIO | 一个或者多个 AIO 操作失败。同异步都可能发生 |
| EINTR | （同步方式时）lio_listio( )同步等待 AIO 结果时被一个来到的信号中断。信号产生可能是某个 AIO 操作完成引起的，这时，用户程序可以使用 aio_suspend( )等待其他操作完成 |
| ECANCELED | （异步方式时）操作取消 aio_cancel( ) |

**（4）aio_suspend**

函数定义：

```
#include "aio.h"
int aio_suspend ( const struct aiocb * list[],
      int nEnt, const struct timespec * timeout );
```

函数 aio_suspend( )用于等待已经提交的一组 AIO 中任意个操作完成。参数 list 是一个指向一组 AIO 控制块的指针，该组 AIO 控制块的个数由参数 nEnt 表示。参数 timeout 非 NULL 时表示最大等待时间。

函数 aio_suspend( )将使任务挂起直到发生下列条件：至少有一个等待的 AIO 完成；超时；等待信号被中断。

如果 list 指示的一个或者多个 AIO 完成，aio_suspend( )返回 OK，但是此时 aio_suspend( )没有指示具体哪个 AIO 完成的机制，需要用户程序调用 aio_error( )或者 aio_return( )检查 list 列表中的每个 AIO 结果。

如果 aio_suspend( )返回 ERROR，可以检查 errno 判断具体情况：

| EAGAIN | 超时 |
|---|---|
| EINTR | 函数被信号中断 |

> aio_suspend( )与信号中断
> 被信号中断时函数返回-1，errno=EINTR。
> **注意：**
> （1）能中断 aio_suspend( )的信号不一定是 AIO 完成的信号，只要信号的处理方式为“捕获”，都能中断 aio_suspend( );
> （2）如果被 AIO 完成信号中断（该信号一般是程序设置发出的），则可以确定 AIO 完成，但是 aio_suspend( )返回结果仍然是-1，errno=EINTR。

**（5）aio_cancel**

函数定义：

```
#include "aio.h"
int aio_cancel (int fd, struct aiocb *pAiocb );
```

函数 aio_cancel( )用于比较优雅地取消一个或者多个在 fd 所指定的文件描述符上阻塞的 AIO。如果指针 pAiocb 非 NULL，则该函数将取消其所指向的单个 AIO 操作（该 AIO 的 aio_fildes 必须等于 fd)。如果 pAiocb 为 NULL，则函数将尝试取消所有阻塞在 fd 上的 AIO 操作。

其优雅之处在于：被成功取消的 AIO 操作以正常的方式结束，如果 AIO 控制块定义了信号 aio_sigevent，系统仍然会发送该信号，但是 aio_error( )或者 aio_return( )将对该 AIO 返回 ECANCELED，表示操作属于被提前取消。如果某个 AIO 操作不能取消，则 aio_cancel( )不会对该 AIO 的执行产生影响。

aio_cancel( )返回值如表 5-15 所示：

表 5-15　aio_cancel( )返回值

| | |
|---|---|
| AIO_CANCELED | 请求取消的操作全部被取消 |
| AIO_NOTCANCELED | 至少一个要求被取消的操作无法取消 |
| AIO_ALLDONE | 调用 aio_cancel( )之前所有要求取消的操作都已经完成 |
| ERROR | 函数出错（指定了无效参数） |

**（6）aio_error**

函数定义：

```
#include "aio.h"
int aio_error ( const struct aiocb * pAiocb );
```

在调用 aio_read( )、aio_write( )以及 lio_listio( )时，errno 的值通常只是反应了将请求的 AIO 操作进行队列的结果。实际的 I/O 完成情况通过调用 aio_error( )得到。aio_error( )返回 I/O 操作错误（或者 I/O 成功完成）状态。参数 pAiocb 指向要检查状态的操作的 AIO 控制块。

aio_error( )的返回值分 4 种情况，如表 5-16 所示：

表 5-16　aio_eror( )的返回值

| | |
|---|---|
| OK | 要检查的 AIO 已经成功执行 |
| EINPROGRESS | 要检查的 AIO 尚未完成 |
| ERROR | 参数 pAiocb 无效，此时设置 errno 为 EINVAL |
| 具体出错值 | AIO 操作失败<br>ECANCELED/ EAGAIN/ EINVAL/ECANCELED 等 |

通常程序只需要检查前面 3 种情况。

**（7）aio_return**

函数定义：

```
#include "aio.h"
size_t aio_return ( struct aiocb * pAiocb );
```

函数 aio_return 读取 pAiocb 所表示的 AIO 操作的返回状态。如果操作正在执行，或者指定了无效参数 pAiocb，函数返回 ERROR，否则函数返回 AIO 执行结果。

调用 aio_return( )后系统将释放 pAiocb 指向的 AIO 控制块，释放后的 AIO 控制块可以用于新的 AIO 操作。如果是动态内存，不用时可以调用 free( )将其释放，防止泄漏。

**aio_return( )**

*几乎总是在调用 aio_error( )确定 AIO 已经完成（成功或者失败）后，调用*

aio_return( )释放系统资源。有一个例外不需要调用 aio_error( )，那就是在信号处理函数里面。可以确信 AIO 已经完成，才有了信号处理函数的调用。
操作系统为每个 AIO 维护一个错误和返回状态信息。返回状态信息用于用户任务告诉系统是否已经结束对某个 AIO 的处理，系统可以释放该 AIO 占有的任何资源，对 aio_return()的调用就是为了实现这一目的。因此，对于每个 AIO 操作，最后都必须调用且仅调用一次 aio_return( )以释放系统资源，调用了 aio_return( )后不能再对该 AIO 调用 aio_error( )或者 aio_return( )。

## 5.5.3 用 AIO 的实例

在同步 I/O 体制下，检查 I/O 完成情况是简单而直接的，根据函数返回值和 errno 就可以指定操作完成结果：read( )和 write( )直到操作完成才返回，根据其返回结果，如果是 ERROR，查看任务的全局变量 errno 就可以知道错误细节。

对于 AIO 操作，检查完成结果就复杂得多。主要源于检查的时机不再仅仅是函数返回时。aio_read( )和 aio_write( )返回结果和当时的 errno 只能表示系统将请求的 AIO 进行队列的情况，系统实际完成 AIO 的情况要在后来通过周期性检查或者其他机制得到。检查结果也更复杂，POSIX 还引入了信号通知方式告诉任务处理 AIO 完成情况。

为了说明检查 AIO 执行结果的方法，下面我们将针对[WRS-pg5.5]中给出在一个管道上使用 AIO 的 3 个非常简洁的实例进行分析。在这些例子中，首先将创建管道，提交 AIO 请求读取管道，显然这时管道是空的，但是正常情况下 AIO 返回 OK。然后再提交 AIO 请求写数据到管道。3 种实现方式差别在于对 AIO 执行完成的判断和处理上。

实例 1 采用的是周期检查的方式，这是一种最简单的方法，但是没有充分发挥 AIO 的优越性。周期检查一般是下面这种情况：

```
...
aio_read(&theAiocb);
while ((the_errno_stat = aio_error(&theAiocb) == EINPROGRESS)
    taskDelay (sometime);
return_value = aio_return(&theAiocb);
...
```

在上面的代码中，如果延迟因子 sometime 设得太大，则影响了 I/O 的实时性，如果设得太小，程序频繁查询 aio_error( )又浪费了 CPU 周期。

实例 2 采用挂起调用者方式。在 AIO 请求被系统加入到队列后，程序调用 aio_suspend( )挂起任务等待 AIO 完成。对于多个位于 aio_suspend( )参数列表中的 AIO，需要在 aio_suspend( )返回后逐一检查 AIO 完成情况，如下：

```
...
```

```
aiocb_list[0] = &aiocb_read;
aiocb_list[1] = &aiocb_write;
while ((aio_error (&aiocb_read) == EINPROGRESS) ||
      (aio_error (&aiocb_write) == EINPROGRESS))
aio_suspend (aiocb_list, 2, NULL);
...
```

和实例 1 相比，上面的代码本质上不再是周期检查方式，而是阻塞等待方式。

实例 3 采用信号通知方式。使用异步 I/O 信号通知属于 POSIX1003.1b 定义的扩展信号接口部分。VxWorks 实现了该接口。程序先编写并安装扩展信号处理函数，然后设置 AIO 控制块中的信号事件 aio_sigevent，通常将 AIO 控制块地址作为信号附加信息参数，这样信号处理函数可以知道谁引起了该信号，类似下面的代码：

```
theAiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
theAiocb.aio_sigevent.sigev_signo = THE_SIGNAL_NO;
theAiocb.aio_sigevent.sigev_value.sival_ptr =(void *) &theAiocb;
```

提交了 AIO 请求后，程序接下来继续执行其他处理。当设置了信号事件的 AIO 完成时，系统将自动中断任务当前的处理，转入信号处理函数。我们在本例的信号处理函数中进行释放 AIO 控制块，显示信息。

为了简化讨论，实例 3 仅仅表示了 AIO 中信号的设置和处理。必须说明：

（1）因为 AIO 控制块在信号处理函数中释放，因此 AIO 控制块不能在函数 aioExampleSig( )的栈空间中分配。通过将 AIO 控制块定义为 static 型来实现这一目的。根据实际情况，许多实际应用中通过 malloc( )动态分配内存可能更可取。不过注意信号处理函数中应该避免调用 free( )释放内存以防死锁。

（2）对 AIO 读操作，程序实际上是简单处理。更正常的做法应该是和写 AIO 同样的处理。但是这样修改会使代码延迟许多。

参考信号相关章节获得关于信号更详细的介绍。

## 实例 1：周期检查 AIO

```
#include "vxWorks.h"
#include "aio.h"
#include "errno.h"
#define BUFFER_SIZE 200
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AIO控制块 */
    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;

    /* 开始读AIO */
    if (aio_read (&aiocb_read) ==ERROR)
       printf ("aioExample: aio_read failed\n");

    /* 读AIO状态应该为EINPROGRESS */
    if (aio_error (&aiocb_read) == EINPROGRESS)
       printf ("aioExample: read is still in progress\n");

    /* 写管道AIO */
    printf ("aioExample: getting ready to initiate the write\n");
    if (aio_write (&aiocb_write) ==ERROR)
       printf ("aioExample: aio_write failed\n");
```

```
    /* 周期查询等待读 AIO 和写 AIO 完成 */
    while ((aio_error (&aiocb_read) == EINPROGRESS) ||
           (aio_error (&aiocb_write) == EINPROGRESS))
        taskDelay (1);

    /* AIO 操作完成 结束操作，释放系统资源*/
    printf ("aioExample: message = %s\n", buffer);
    if (aio_return (&aiocb_read) ==ERROR)
        printf ("aioExample: aio_return for aiocb_read failed\n");
    if (aio_return (&aiocb_write) == ERROR)
        printf ("aioExample: aio_return for aiocb_write failed\n");
    close (fd);
    return (OK);
}
```

## 实例 2：阻塞等待 AIO（A 串行 io_suspend）

```
#include "vxWorks.h"
#include "aio.h"
#include "errno.h"
#define BUFFER_SIZE 200
STATUS aioExampleSus (void)
{
    int fd;
    static char exFile [] = "/pipe/1stPipe";
    struct aiocb aiocb_read; /* read aiocb */
    struct aiocb aiocb_write; /* write aiocb */
    struct aiocb * aiocb_list[2]; /* list of aiocb(s) */
    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 ("aioExampleSus: cannot open %s errno 0x%x\n", exFile, errno);
        return (ERROR);
}
    printf ("aioExampleSus: Example file = %s\tFile descriptor = %d\n",
```

```
            exFile, fd);

    /* 初始化读写 AIO 控制块 */
    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;

    /* 开始读 AIO */
    if (aio_read (&aiocb_read) ==ERROR)
       printf ("aioExampleSus: aio_read failed\n");

    /* 读 AIO 状态应该为 EINPROGRESS */
    if (aio_error (&aiocb_read) == EINPROGRESS)
       printf ("aioExampleSus: read is still in progress\n");

    /* 写管道 AIO */
    printf ("aioExampleSus: getting ready to initiate the write\n");
    if (aio_write (&aiocb_write) ==ERROR)
       printf ("aioExampleSus: aio_write failed\n");

    /* 调用 aio_suspend 挂起任务等待 AIO 结束 */
    aiocb_list[0] = &aiocb_read;
    aiocb_list[1] = &aiocb_write;
    while ((aio_error (&aiocb_read) == EINPROGRESS) ||
           (aio_error (&aiocb_write) == EINPROGRESS))
       aio_suspend (aiocb_list, 2, NULL);

    /* AIO 操作完成 结束操作，释放系统资源*/
    printf ("aioExampleSus: message = %s\n", buffer);
    if (aio_return (&aiocb_read) ==ERROR)
```

```
        printf ("aioExampleSus: aio_return for aiocb_read failed\n");
    if (aio_return (&aiocb_write) == ERROR)
        printf ("aioExampleSus: aio_return for aiocb_write failed\n");
    close (fd);
    return (OK);
}
```

## 实例3：使用信号通知AIO完成

```
#include "vxWorks.h"
#include "aio.h"
#include "errno.h"
#define BUFFER_SIZE 200
#define LIST_SIZE 1
#define EXAMPLE_SIG_NO 25   /*用于AIO通知的信号编号*/
void mySigHandler (int sig, struct siginfo * info, void * pContext);

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 ("aioExampleSig: cannot open %s errno 0x%x\n", exFile, errno);
        return (ERROR);
    }
    printf ("aioExampleSig: Example file = %s\tFile descriptor = %d\n",
              exFile, fd);

    /* 设置信号 EXAMPLE_SIG_NO 的信号处理函数 */
    action.sa_sigaction = mySigHandler;
    action.sa_flags = SA_SIGINFO;
    sigemptyset (&action.sa_mask);
```

```
sigaction (EXAMPLE_SIG_NO, &action, NULL);

/* 初始化读写 AIO 控制块 */
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;

/* 设置信号事件，AIO 控制块地址作为信号的附加信息 */
aiocb_write.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
aiocb_write.aio_sigevent.sigev_signo = EXAMPLE_SIG_NO;
aiocb_write.aio_sigevent.sigev_value.sival_ptr =(void *) &aiocb_write;

/* 开始读 AIO */
if (aio_read (&aiocb_read) == ERROR)
printf ("aioExampleSig: aio_read failed\n");

/* 读 AIO 状态应该为 EINPROGRESS */
if (aio_error (&aiocb_read) == EINPROGRESS)
printf ("aioExampleSig: read is still in progress\n");

/*开始写 AIO */
printf ("aioExampleSig: getting ready to initiate the write\n");
if (aio_write (&aiocb_write) == ERROR)
    printf ("aioExampleSig: aio_write failed\n");

/* 信号处理函数将处理 AIO 写操作完成事件 */
/* 下面对 AIO 读操作的简单处理 */
if (aio_return (&aiocb_read) == ERROR)
    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 )
{
    printf ("mySigHandler: Got signal for aio write\n");

    /*结束AIO操作，释放系统资源*/
    if (aio_return ((struct aiocb*)info->si_value.sival_ptr) == -1)
    {
        printf ("mySigHandler: aio_return for aiocb_write failed\n");
        printErrno (0);
    }
}
```

# 5.6　常用的VxWorks设备

## 5.6.1　串行终端设备

终端的种类有很多，包括字符型的“哑”终端，“灵巧的”图形终端，还有软件模拟的伪终端。从通信手段看，“哑”终端一般通过低速的串口和主机相连，不需要终端计算能力，能进行简单的配置。“灵巧的”图形终端一般通过网络和主机相连，要求终端具有计算能力。本节介绍的串行终端设备属于“哑”终端，串行终端具有如下属性：

- 串口速率：终端一般支持300、600、1200、2400、4800、9600、19200、38400、57600、115200bps，默认值一般为9600bps；
- 数据位：可能选项包括5~8位，默认一般为8位；
- 停止位：可能选项包括1位、1.5位、2位，默认一般为1位；
- 奇偶校验：可能选项包括奇校验、偶校验、无校验，默认一般为无校验；
- 流控方式：可能选项包括硬件流控、软件流控、无流控，默认值因设备而异。

VxWorks程序通过tty驱动ttyDrv/tyLib和串行终端通信。如果定义I/O系统组件INCLUDE_TTY_DEV，则系统初始化时在usrRoot( )中初始化tty驱动并创建tty设备，宏

NUM_TTY 定义了 tty 设备数目，需要等于目标板上异步通信控制器 SCC 提供的通道数目。创建 tty 设备后用户程序可以通过如同访问其他设备一样访问 tty。更多的信息在 5.8 节“串口 tty 设备”。

## 5.6.2 伪内存设备

伪内存设备驱动使程序可以访问内存像访问一个虚拟的 I/O 设备一样。当前的版本没有将伪内存设备通过 CDF（组件描述语言）包装成一个标准 VxWorks 组件，而是通过两个源文件提供伪内存设备驱动 memDrv：

- <install-dir>/target/src/usr/memDrv.c；
- <install-dir>/target/h/memDrv.h。

memDrv 提供了初始化驱动和创建/删除设备的用户应用程序接口以及必需的内部函数。4 个应用程序接口函数如表 5-17 所示：

**表 5-17 应用程序的接口函数**

| | |
|---|---|
| memDrv( ) | 初始化和安装 memDrv 驱动 |
| memDevCreate( ) | 创建单文件伪内存设备 |
| memDevCreateDir( ) | 创建多文件伪内存设备 |
| memDevDelete( ) | 删除伪内存设备 |

memDrv( )必须调用后才能使用 memDrv 的另外 3 个接口函数。一个伪内存设备可以对应一个文件/设备或多个文件/设备，通过不同的函数创建。

### 1. 单文件

函数 memDevCreate( )用于创建伪内存设备，定义为：

```
#include "memDrv.h"
STATUS memDevCreate ( char * name, char * base, int length );
```

该函数创建名为 name 的单个文件伪内存设备，base 指定设备对应的内存地址，length 规定该段内存字节长度。如果成功，memDevCreate ( )返回 OK，否则返回 ERROR。例如：

```
memDevCreate ("/mem/cpu0/", 0, sysMemTop( ));
```

创建名为“/mem/cpu0/”的设备，该设备对应所有系统内存。

memDrv 需要确定文件对应上述设备的位置（即内存中的地址）。该位置通过相对于设备起始内存地址 base 的一个偏移量表示。因此一种在该设备上打开文件的方式就是根据“设备名+文件在设备中偏移量”对文件进行命名，memDrv 将根据文件名提供的偏移量确

定文件位置。例如对上面创建的“/mem/cpu0/”设备，下面调用：

```
fd = open ("/mem/cpu0/1000", O_RDONLY, 0);
```

将在设备中偏移量为1000处建立文件。

另一种打开伪设备文件的方式：

```
fd = open ("/mem/cpu0/", O_RDONLY, 0);
```

文件指针定位在设备所在内存的开始处，随后可以调用 I/O 控制命令 ioctl( fd, FIOSEEK, posi)在设备内存范围内指定文件偏移量。

I/O 控制命令 FIOSEEK 覆盖打开文件时指定的偏移量。

## 2. 多文件

创建多文件的伪内存设备使得可以在内存中建立文件系统。通过下面的函数实现：

```
#include "memDrv.h"
STATUS memDevCreateDir ( char * name, MEM_DRV_DIRENTRY * files, int numFiles);
```

该函数创建多文件设备，相当于将数组 files 表示的文件目录结构挂装到 name 表示的根文件系统下，name 可以看成被创建设备的名称，或者是挂装点，numFiles 表示数组 files 中包含多个文件/目录信息数。

数组 files 表示文件目录结构，其中每个元素代表一个“文件/目录”项，具有如下定义：

```
#include "memDrv.h"
typedef struct mem_drv_direntry {
    char * name; /* 相对于挂装点或者所在目录的文件名称 */
    char * base; /* 文件内容起始地址（如是目录为 NULL）*/
    struct mem_drv_direntry *pDir;
  /* 目录包含的文件（如是文件为 NULL）*/
    int length;  /* 对文件表示字节长度对目录表示包含文件数 */
} MEM_DRV_DIRENTRY;
```

虽然没有限制，但是实际应用中通常是将主机开发环境下的文件/目录挂装到目标系统，创建目标系统的伪内存设备，供程序访问。

实例：主机开发环境下面有这样一个含有 3 个数据库 db1~3.dat 的目录 database，如图 5-4 所示。

我们希望创建一个多文件伪内存设备，将目录 database 及其所有子目录和文件挂装到该设备下，目标系统应用程序需要读取其中记录，和访问磁盘文件一样。

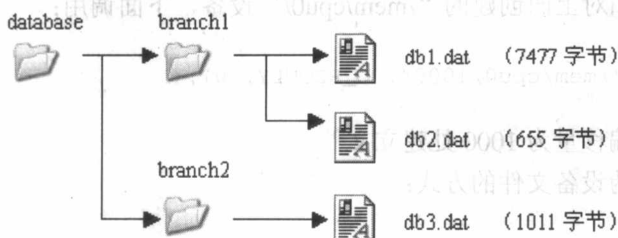



图 5-4 目录结构示例

```
#include <vxWorks.h>
#include <memDrv.h>

/* 第一步 定义 3 文件 database\branch1\db1~3.dat 的内容 */
static unsigned char datadatabase_branch1_db1_dat[] = { ... };
static unsigned char datadatabase_branch1_db2_dat[] = { ... };
static unsigned char datadatabase_branch2_db3_dat[] = { ... };

/* 第二步 定义 database 和子目录 database\branch1~2 的目录结构数组 */
static MEM_DRV_DIRENTRY dirdatabase_branch1[] = {
  { "db1.dat", datadatabase_branch1_db1_dat, NULL, 7477 },
  { "db2.dat", datadatabase_branch1_db2_dat, NULL, 655 },
};
static MEM_DRV_DIRENTRY dirdatabase_branch2[] = {
  { "db3.dat", datadatabase_branch2_db3_dat, NULL, 1101 },
};
static MEM_DRV_DIRENTRY dirdatabase[] = {
  { "branch1", NULL, dirdatabase_branch1, 2 },
  { "branch2", NULL, dirdatabase_branch2, 1 },
};


/*
 * 挂装文件目录结构的函数
 */
STATUS memDrvAddFiles_database_ (void)
{
   return memDevCreateDir ("/database/", dirdatabase, 2);
}
/*
```

```
 * 卸载挂装的文件目录结构
 */
STATUS memDrvDeleteFiles_database_ (void)
{
    return memDevDelete ("/database/");
}
```

这样，当用户程序需要挂装 database 时，就可以调用 memDrvAddFiles_database_完成。挂装后，原来的文件都可以通过相对于挂装点的文件路径访问，例如，需要打开 db2.dat，可以：

```
fd = open ("/database/branch1/db2.dat", O_RDONLY, 0);
```

伪内存设备文件并不限制只读方式。如果以写入方式打开，则写入最大长度限于创建设备时“文件/目录”项指定的文件长度，例如对上面的 db2.dat，如果写入内容超出 655 字节，memDrv 会将超出部分丢弃，剩下部分写入文件，并返回实际写入的字节数。对于单文件设备，允许写入的字节数受限于为设备指定内存大小。

多文件伪内存设备为挂装多个文件目录提供了极大的方便。但是上述定义文件内容和“文件/目录”项的工作比较繁琐，将一个几百几千字节的文件内容表示成一个字节数组不但工作量大，而且容易出错。好在 Tornado 包含了一个工具 memdrvbuild.exe，可以自动生成上述数据和函数。该工具位于<install-dir>\host\x86-win32\bin\下，在 DOS 命令行下运行即可，其语法比较简单：

```
memdrvbuild [ -o filebase ] [ -m mount ] directory
```

directory 表示要挂装的主机目录，可选参数 mount 指明挂装点，filebase 指明生成的 C 语言源文件名称。可选参数默认时和主机目录名称相同。

### 3. 通过 memDrv 实现动态模块装载

memDrv 一个方便的用途就是用来实现动态装载模块。动态模块装载尤其在目标系统程序升级时候有用，例如，可以将要装载的模块烧写到 EEPROM，在系统已经引导完成后，再将对应的内存地址虚拟成伪内存设备文件，然后通过 loadLib 提供的动态装载将其装入。

下面的函数 memLoadModule 是这样一个例子。调用之前，目标模块已经通过某种机制放到了内存（RAM 或者 EEPROM）中，object 指向其地址，模块大小 size 也已经确定，然后调用此函数实现模块装入。

```
#include <vxWorks.h>
#include <ioLib.h>
#include <loadLib.h>
```

```
#include <memDrv.h>

STATUS memLoadModule ( char * name,    /* 命名memDrv设备的字符串*/
        int * object,  /* 目标模块地址*/
        int size       /* 目标模块大小*/
)
{
    int fd;
    MODULE_ID modId;

    if ( memDrv( ) == ERROR )  return ERROR;

    if (memDevCreate(name, (char *) object, size) == ERROR)
        return ERROR;

    fd = open(name, O_RDONLY, 0);
    if (fd == ERROR)   return ERROR;

    modId = loadModule( fd, LOAD_ALL_SYMBOLS );
    close (fd);
    return modId!=NULL ? OK : ERROR;
}
```

## 4. I/O控制

memDrv支持下面的I/O控制功能。对其他I/O控制命令，函数返回ERROR，并设置errno等于S_ioLib_UNKNOWN_REQUEST。

- FIONREAD

  result = ioctl( fd, FIONREAD, &bytesleft);

如果fd表示文件，返回OK，设置bytesleft为从当前位置到文件结束剩下的字节数；如果fd表示目录，返回ERROR（-1），并设置errno为EINVAL。

- FIOSEEK

  result = ioctl( fd, FIOSEEK, newposi);

定位fd的文件指针到newposi处。如果fd表示目录，或者newposi大于文件长度，则返回ERROR（-1），并设置errno为EINVAL；否则返回OK。

- FIOWHERE

  where = ioctl( fd, FIOWHERE, NULL);

返回fd的文件指针当前位置。对于目录总返回当前位置等于0。

- FIOGETFL

  result = ioctl( fd, FIOGETFL, &mode);

读取 fd 的打开模式（O_RDONLY/O_WRONLY/O_RDWR）。

- FIOFSTATGET

  struct stat s;

  result = ioctl( fd, FIOFSTATGET, &s);

类似 fstat( )调用，用于读取 fd 状态信息。可以通过该调用来判断 fd 是一个文件（s.st_mode 中 S_IFREG 被置位）还是一个目录（s.st_mode 中 S_IFDIR 被置位）。

- FIOREADDIR

  result = ioctl( fd, FIOREADDIR, &d);

读取目录 fd 的内容。例如：

```
...
DIR d;
d.dd_cookie = 0;
printf(" read contents of the dir\n ");
while(ioctl( fd, FIOREADDIR, &d)==OK)
{
    printf(" \t%d : %s ", d.dd_cookie, d.dd_dirent.d_name);
}
```

如果 fd 是一个文件，函数返回 ERROR（-1），并设置 errno 为 EINVAL。

## 5.6.3 NFS 设备

NFS 设备使位于远程主机上的文件系统可以通过 NFS 协议进行访问。NFS 协议规定了在客户端获取文件和服务器端输出文件给远程客户的过程。对客户端应用程序而言，通过使用 VxWorks 的 NFS 驱动挂装服务器文件系统后，可以像访问本地文件一样访问 NFS 设备上的文件。VxWorks 的 NFS 客户端驱动包括应用接口 nfsDrv 及其所需要的支持库 nfsLib。VxWorks 下运行 NFS 服务器端参见第 7 章 7.3 节“远程访问服务”。

应用程序使用 NFS 客户端驱动需要定义 INCLUDE_NFS。初始化函数 nfsDrv( )在定义 INCLUDE_NFS 时自动调用。

### 1. 挂装/卸载 NFS 文件系统

NFS 客户端访问 NFS 服务器需要在本地挂装（Mount）远程文件系统。调用 nfsMount( ) 实现该函数的参数是：（1）远程 NFS 服务器名称；（2）远程主机文件系统名称；（3）本地文件系统名称。如：

```
#include "nfsDrv.h"
nfsMount ( "mars", "/usr", "/vxusr" );
```

将位于远程主机 mars 上的/usr 作为本地文件系统/vxusr 挂装。nfsMount( )相当于创建了本地文件系统/vxusr，随后程序可以像访问本地其他文件系统一样访问/vxusr。

如果指定 nfsMount( )的第 3 个参数为 NULL，则本地文件系统名称和远程文件系统名称相同。

不再使用挂装的 NFS 文件系统可以通过调用

```
#include "nfsDrv.h"
STATUS nfsUnmount ( char * localName );
```

进行卸载，该函数的惟一参数指定为挂装的本地文件系统名称。

## 2. NFS 客户端 I/O 控制

nfsDrv 支持下列 I/O 控制命令，每个命令都需要一个打开的有效 NFS 文件描述符 fd。

- FIOGETNAME

  status = ioctl (fd, FIOGETNAME, &nameBuf);

取得 fd 的名称。

- FIONREAD

  result = ioctl( fd, FIONREAD, &bytesleft;

取得文件 fd 中从当前位置到文件结束剩下的字节数。

- FIOSEEK

  status = ioctl (fd, FIOSEEK, newposi);

定位 fd 的文件指针到 newposi 处，如果 newposi 超出文件长度，该调用将增长文件长度，然后定位到 newposi 处，增加部分内容置为 0。

- FIOSYNC

  status = ioctl (fd, FIOSYNC, NULL);

刷新驱动程序缓冲区数据到所在的远程 NFS 文件。

- FIOWHERE

  where = ioctl( fd, FIOWHERE, NULL);

返回 fd 的文件指针的当前位置（等于下次读/写所在的地址）。

- FIOFSTATGET

  status = ioctl (fd, FIOFSTATGET, &st);

取得文件状态信息，填入参数 st 中（st 定义为 struct stat），相当于调用 stat( )或者 fstat( )。

- FIOFSTATFSGET

  status = ioctl (fd, FIOFSTATFSGET, &stfs);

取得文件系统参数信息，填入结构 stfs（定义为 statfs），相当于调用 stat( )或者 fstat( )。

- FIOREADDIR
  status = ioctl (fd, FIOREADDIR, &d);

读取目录 fd 的内容，相当于 readdir( )，参见前面对伪内存设备的介绍。

上述后面 3 个 I/O 控制命令一般通过对应的库 dirLib 提供的函数进行，参考第 6 章 6.3 节“dosFs 文件系统”。

## 5.6.4 非 NFS 网络文件系统设备（netDrv 设备）

对不支持 NFS 的环境提供了使用网络设备 VxWorks，提供另外一种机制使用网络文件系统，即通过 netDrv 创建网络设备进行访问。netDrv 具有如下特点：

（1）使用主机文件的本地复制，因此很需要内存空间。对于不固定缓冲区的 netDrv 设备创建，至少需要和主机文件大小相同的本地内存。

（2）创建文件时，将初始化一个空的本地缓冲区，打开文件时文件的整个内容都下载到本地，以后的读写和 I/O 控制都在本地复制上进行，文件关闭时，如果以可写方式打开，本地复制写入远程主机。

（3）远程主机上的文件通过 FTP 或者 RSH 下载。对于 FTP，要求远程主机提供 FTP 下载权限。如果要提交文件修改，还需要具有 FTP 上传权限。

（4）目录库 dirLib 的函数 opendir / readdir 不能在 netDrv 目录上操作。

（5）stat / fstat 没有完全实现，通常用来得到文件大小。

使用 netDrv 必须定义 INCLUDE_NET_REM_IO，netDrv 的驱动程序初始化函数 netDrv( )在定义了该宏之后自动调用，就可以创建 netDrv 设备。netDrv 设备创建分固定缓冲区大小和不固定缓冲区大小两种方式。

固定缓冲区大小的 netDrv 设备为本地文件复制指定固定大小的缓冲区，如果主机上的文件太大，则本地缓冲区维护的复制是一个分段完整的映像。固定缓冲区的设备具有下面两个额外限制：

- 只支持以只读（O_RDONLY）或者只写（O_WRONLY）的方式打开；
- 文件指针定位只在以 O_RDONLY 方式打开时允许，以 O_WRONLY 方式打开时不支持。并且在 O_RDONLY 方式下定位文件指针超出缓冲区缓存的内容时会很慢，在 O_WRONLY 方式下，可能需要分段刷新缓冲区更改，也会使操作费时。

比较之下，非固定缓冲区的 netDrv 没有上述限制，但是文件太大时内存使用难以控制。

### 1. 创建设备

不固定缓冲区大小的创建设备调用 netDevCreate( )实现，该函数定义为：

```
#include " netDrv.h"
STATUS netDevCreate ( char * devName, char * host, int protocol );
```

参数 devName 指定本地设备名称，host 指定主机名称（*），protocol 表示用于与主机之间文件传递协议，0 = RSH，1 = FTP。

```
result = netDevCreate ("tgt:", "host1", 0);
```

上面的调用创建了本地设备“tgt:”，风河公司建议对非 NFS 网络设备命名遵循“名称+冒号”的惯例。如果创建成功，函数返回 OK，否则返回 ERROR。

创建了“tgt:”之后，如果要打开主机 host1 上的文件，如/usr/note，可以调用：

```
fd = open("tgt: /usr/note ", O_RDONLY, 0);
```

### 2．I/O 控制

netDrv 支持 I/O 命令：FIOGETNAME / FIONREAD / FIOSEEK / FIOWHERE / FIOFSTATGET。其用法和 NFS 设备相似。注意对 FIOFSTATGET 命令：

```
struct stat statStruct;
fd = open (pFileName, O_RDONLY);
status = ioctl (fd, FIOFSTATGET, &statStruct);
```

netDrv 始终设置 statStruct.st_mode 为 S_IFREG（常规文件），另外设置 statStruct.st_dev 和 statStruct.st_size，statStruct 中其他域的信息没有意义。

## 5.6.5 RAM 盘

RAM 盘驱动使用一片内存（RAM 或者 NVRAM）模拟磁盘的行为。内存的位置和大小在创建 RAM 盘时指定。如果要在多次 VxWorks 热启动之间保存数据，则 RAM 盘比较合适。RAM 盘另一个用途就是用于多个 CPU 之间共享数据。

前面讲的伪内存设备 memDrv 具有和 RAM 盘类似的功能，但是具有不同的实现机制：RAM 盘驱动并不是一个“独立”的 I/O 设备驱动程序，不同于严格意义上的“I/O 驱动”在系统驱动程序表中一个条目（见本章 5.7 节“I/O 系统内部结构”），因此不能直接被应用程序使用，而必须在其上建立文件系统（如 dosFs、rawFs、rt11Fs 等），然后通过文件系统函数访问，由于相同的原因，创建 RAM 盘也不需要在系统启动时进行驱动程序初始化，直接创建“设备”即可；而伪内存设备 memDrv 是一个“独立”的设备驱动，需要在系统初始化时将 memDrv 安装在系统驱动程序表，应用程序可以直接通过基本 I/O 系统调用（read/write 等）访问 memDrv 设备。

根据 RAM 盘对上层程序提供的接口不同，RAM 盘可以分为两种，如表 5-18 所示：

表 5-18　RAM 盘

| | |
|---|---|
| 块设备 RAM 盘 | 适于 VxWorks 5.4 / dosFs(1)，由驱动 ramDrv 支持<br>创建方式：ramDevCreate( ) |
| CBIO RAM 盘 | 适于 VxWorks 5.5 / dosFs(1)(2)，由 ramDiskCbio 支持<br>创建方式：ramDiskDevCreate( ) |

VxWorks 5.4 只提供块设备 RAM 盘。CBIO 是我们将要在第 6 章 6.2 节“CBIO”中介绍的概念，这里简要说明一下。CBIO 是对块设备（BLK_DEV）的封装，为建立于其上的文件系统提供标准 CBIO 访问接口。CBIO 也是一种块设备，只是具有了新的访问接口。因此，这里“块设备”专门指未封装的 BLK_DEV。

## 1．块设备 RAM 盘：ramDrv

使用 ramDrv 创建 RAM 盘不需要预先初始化，直接调用下述函数：

```
#include "ramDrv.h"
BLK_DEV *ramDevCreate ( char * ramAddr, int bytesPerBlk,
    int blksPerTrack, int nBlocks, int blkOffset );
```

参数 ramAddr 表示创建 RAM 盘对应的内存地址。如果该参数为 0，ramDrv 将自动调用 malloc( )申请。随后几个参数表示 RAM 盘对应的内存大小以及逻辑结构：bytePerBlk 表示块大小，一般为 512；nBlocks 表示总块数；blkPerTrack 表示每道块数，如果指定为 0，则对应所有块数 nBlocks；blkOffset 表示以块为单位的从 ramAddr 开始的偏移量上层文件系统函数访问 RAM 盘时指定的块地址都被加上该偏移量，通常指定为 0。

对上层程序来说，“块”就相当于磁盘上的“扇区”。

可以指定 nBlocks 为 0 让 ramDrv 自动计算 RAM 盘大小。以字节为单位，ramDrv 取 51200 和最大空闲内存块大小 50%中较小值作为 RAM 盘大小。

在 dosFs(1)中，ramDrv 通常以如下方式建立文件系统。

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

## 2．CBIO RAM 盘：ramDiskCbio

如果使用 dosFs(2)，则最好使用 CBIO 接口块设备：ramDiskCbio。VxWorks 以源代码形式提供 ramDiskCbio：

<install-dir>\ target\src\usr\ramDiskCbio.c

下面的函数创建 CBIO RAM 盘：

```
#include "ramDiskCbio.h"
CBIO_DEV_ID ramDiskDevCreate ( char * pRamAddr, int bytesPerBlk,
   int blksPerTrack, int nBlocks, int blkOffset );
```

该函数的参数及用法和 ramDrv 相同。但是返回的是一个 CBIO 设备，因此可以直接用于 dosFs(2)创建 dosFs 设备。

下面这段代码在一片大小为 128KB 的 RAM 盘上创建 dosFs 系统。对于 RAM 盘不需要使用缓存 CBIO。

```
#define  /* 128KB, 128 bytes per sector */
...
STATUS usrRamDiskInit ( void )
{
   const RAM_DISK_SIZE  = 128 * 1024;
   char *ramDiskDevName = "/ram0";
   CBIO_DEV_ID cbio;
   STATUS status;

   /* 创建 RAM 盘: 128 bytes/sec, 17 secs/track, auto-allocate */
    cbio = ramDiskDevCreate(NULL, 128, 17, RAM_DISK_SIZE/128, 0) ;
    if( cbio == NULL ) return ERROR ;

   /* 格式化 生成 dosFs 文件系统 */
   status  = dosFsVolFormat( cbio, DOS_OPT_BLANK | DOS_OPT_QUIET, NULL );
   if ( status == NULL )  return ERROR;

   /*创建 RAM 盘 dosFs 设备 允许同时打开 4 个文件 不进行 CHKDSK */
   status = dosFsDevCreate( ramDiskDevName, cbio, 4, NONE );
   if ( status == NULL )  return ERROR;

   return OK;
}
```

在 shell 下运行程序，将看到 usrRamDiskInit 创建的设备：

```
-> sp ramDiskInit
...
drv name
...
  7 /ram0
```

# 5.7 I/O 系统内部结构

VxWorks 的 I/O 系统设计一个特点是系统非常简单，如图 5-5 所示。

位于图 5-5 核心部分的是 VxWorks 的 I/O 系统，从程序角度看，I/O 系统功能通过两个接口库提供，如表 5-19 所示。

表 5-19 接口库

| | |
|---|---|
| iosLib | I/O 系统的驱动程序接口：在 I/O 系统中安装驱动程序，创建设备 |
| ioLib | I/O 系统的应用程序接口：7 个基本 I/O 调用及其他一些实用功能 |

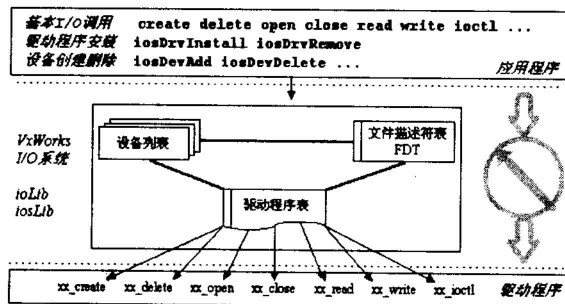



图 5-5 I/O 系统内部层次

I/O 系统主要功能就是通过 iosLib 的驱动程序接口，在 I/O 系统中建立核心数据结构：

- 驱动程序表 —— 一张表格，表格的每一项是一个设备驱动程序，内容为对应 7 个基本 I/O 调用的驱动程序函数指针。表格中每一项的位置即一个驱动程序号；
- 设备列表 —— 所有 I/O 系统设备都维护在一个双向链表上，链表的一个节点表示一个设备，表示的内容包括：设备名称、驱动程序号、设备参数（可以表示多项内容，由驱动程序函数定义、解释、处理）；
- 文件描述符表 —— 一个表格，表格中每一项对应一个打开的文件，表示的内容包括：驱动程序号、驱动程序参数（该参数由驱动程序函数解释、处理）。表格中每一项的位置即一个**文件描述符**。

在上述核心数据结构基础上，I/O 系统只是简单地将用户程序中基本 I/O 调用引导给正确的驱动程序去完成，所有的处理逻辑都设计在驱动程序中体现。驱动程序如何实现用户 I/O 请求对 VxWorks 的 I/O 系统来说是一个透明的过程，例如，I/O 系统记录的文件描述符表由许多（驱动程序号，驱动程序参数）这样的二元组组成，I/O 系统只知道根据驱动程序号找到正确的驱动程序函数，并将“驱动程序参数”传递给找到的函数，至于该函数如何完成用户的 I/O 请求，I/O 系统并不知道。

用户在进行基本 I/O 调用之前必需的过程是：

（1）I/O 系统初始化：在整个系统中只需进行一次。由 usrRoot( )中初始化整个 VxWorks 系统时自动调用 iosInit( )完成；

（2）安装驱动程序：同类驱动程序进行一次，一个驱动程序可以为多个同类设备服务。调用 iosDrvInstall( )完成，通常也发生在 usrRoot( )中，但是不限制，可以在需要时安装驱动；

（3）安装设备：调用 iosDevAdd( )完成，可以在 usrRoot( )中，也可以在任何需要的时候。

& 驱动程序和“设备”（这里指 I/O 系统中的逻辑设备，不是指物理设备）的关系：

（1）“设备”是驱动程序维护的数据结构。必须先安装驱动，然后才能创建设备;

（2)一个逻辑设备通常对应一个物理上的设备或者一个物理设备上的一个功能相对独立的单元;

（3）一个逻辑设备维护着对应的物理设备状态，如收到的数据，控制状态等;

（4）一个驱动程序通常维护多个设备，即多套数据结构。

在许多其他的系统（如 UNIX）中，I/O 处理逻辑集中在 I/O 系统内部，设备驱动通常只提供一些简单的底层 I/O 函数，比如将字符设备缓冲区内容读到 I/O 系统指定位置，或者一个相反的写字过程，只有这一部分是“设备相关的”。高层处理逻辑属于 I/O 系统的设备无关部分。

相比之下，将处理逻辑在 I/O 系统内部实现的优点是驱动程序简单以及设备无关性。缺点除了 I/O 系统复杂之外，还有更重要的一点就是设备无关性封装对系统效率产生影响。作为实时应用，VxWorks 开放的标准 I/O 系统驱动接口使得用户可以自己设计和上层应用程序之间高效通信的驱动程序，或者为驱动程序增加新特色。系统在驱动程序和设备驱动之间的封装和转换往往导致缓冲区复制开销。

由于 VxWorks 将 I/O 处理逻辑放在驱动程序中实现，因此驱动程序设计要复杂一点。不过，VxWorks 也为驱动程序开发提供了实现字符设备和块设备标准协议的几个高层程序库以简化设计。

下面分别介绍上述 I/O 系统内部数据结构的各个部分，它们体现了 VxWorks I/O 系统设计的细节。

## 5.7.1 驱动程序

### 1. 驱动程序表

I/O 系统的功能是将用户 I/O 请求定位到合适的驱动程序来实现。实现这样的功能的方

法是系统维护一个含有每个驱动程序 7 个标志调用的表格，如图 5-6 所示。驱动程序通过调用 I/O 系统内部函数 iosDrvInstall( )动态进行安装。iosDrvInstall( )定义为：

```
#include "iosLib.h"
int iosDrvInstall
(
    FUNCPTR pCreate,     /* 指向 creat( ) 函数的指针 */
    FUNCPTR pDelete,     /* 指向 delete( ) 函数的指针 */
    FUNCPTR pOpen,       /* 指向 open( ) 函数的指针 */
    FUNCPTR pClose,      /* 指向 close( ) 函数的指针 */
    FUNCPTR pRead,       /* 指向 read( ) 函数的指针 */
    FUNCPTR pWrite,      /* 指向 write( ) 函数的指针 */
    FUNCPTR pIoctl       /* 指向 ioctl( ) 函数的指针 */
);
```

函数在驱动程序表中查找一个空余的条目，将 7 个指针填入其中。函数返回值表示驱动程序在表中的位置。该返回值被系统解释为驱动程序号，系统将使用驱动程序号建立设备和驱动程序直接的连接（如图 5-6 所示）。

图 5-6 是一个驱动程序的安装过程：

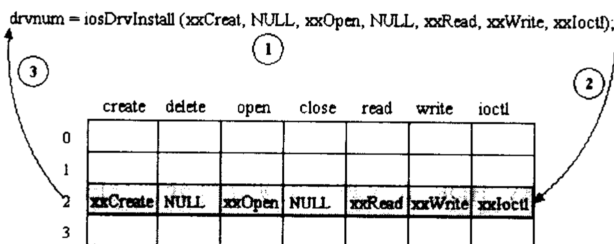



图 5-6 驱动程序表

- 调用 iosDrvInstall，参数指定服务于 7 个基本 I/O 调用的驱动程序函数；
- 系统再驱动程序表中找一个空闲条目，填写数据结构；
- 返回驱动程序号为 2。

如果驱动程序表中没有剩余的空间来安装新的驱动程序，函数返回 ERROR。上面函数调用成功，得到的驱动程序号 2 将用于随后创建设备时指定设备的驱动程序。

图 5-6 是一个典型的非文件系统设备驱动程序，通常这些设备的驱动程序对 close( )调用和 delete( )调用不做任何操作，入参中对应位置传递的值等于 NULL。

## 2. 显示安装的驱动程序

任意时候，可以在 shell 通过 iosShow 的库函数 iosDrvShow( )显示当前系统中安装的所有驱动程序：

```
-> iosDrvShow
  drv    create    delete    open    close    read    write    ioctl
   1     5baec       0      5baec    5bb50    5c310   5c1f4    5bbac
   2       0         0      559e0      0      55a34   55a94    55c10
   ...
```

## 5.7.2 设备

### 1. 系统设备列表

VxWorks 中所有设备通过一个双向链表的数据结构维护，链表中每个节点描述一个设备信息，驱动程序通过该节点控制该设备。不同的设备，不同的驱动程序，所需要的用于控制设备的信息千差万别，但是所有的设备节点始终以一个标准的结构体头部 DEV_HDR 开始：

```
#include "iosLib.h"
typedef struct          /* DEV_HDR - 标准设备头部结构 */
{
    DL_NODE  node;         /*指向前一个和后一个设备的指针*/
    short    drvNum;       /*设备驱动程序号*/
    char *   name;         /*设备名称*/
} DEV_HDR;;
```

如图 5-7 所示，DEV_HDR 定义图中的阴影部分；空白部分由设备驱动程序定义和识别。

采用双向链表结构，使设备易于被动态地安装和卸载。图 5-7 中的驱动程序号 drvNum 是 iosDrvInstall( )的返回值。VxWorks 的设备驱动结构使多个设备共享一个驱动程序，设备驱动程序的数据结构必须设计为支持这种动态重入性。设备名称 name 是一个普通字符串，一般遵循 VxWorks 对设备名称的命名规范。

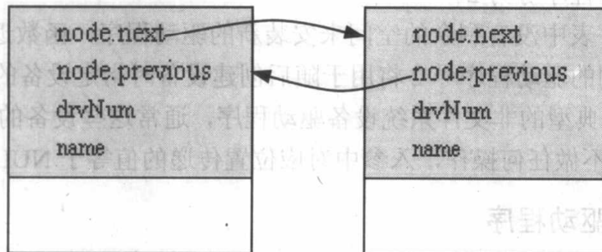



图 5-7　设备描述结构

当 open( )打开一个文件时，VxWorks 根据文件名称在上述双向链表中查找对应设备，并根据链表中的驱动程序号调用驱动程序表中的对应函数。

图 5-7 中完整的设备描述结构定义由驱动程序实现，也就是在 DEV_HDR 上扩充驱动程序需要的特定数据结构，以类似下面的方式：

```
typedef struct {  /*设备 X 的设备描述结构定义*/
   DEV_HDR devHdr;  /* 设备标准头部信息 */
        ...    /*定义所有设备需要的特定数据结构 */
} DEV_X;
```

## 2. 设备安装

驱动程序不需要填写设备标准头部信息 DEV_X.devHdr，只填写 DEV_X 需要的特定数据结构。驱动程序要安装设备时系统内部函数 iosDevAdd( )定义为：

```
#include "iosLib.h"
STATUS iosDevAdd (
   DEV_HDR * pDevHdr,    /* 指向新设备标准头部信息的指针 */
   char *    name,        /* 设备名称 */
   int       drvnum       /* 设备驱动程序号，由 iosDrvInstall( )返回 */
);
```

其中，参数 pDevHdr 通过将指向 DEV_X 的指针强制转换为 DEV_HDR*得到。iosDevAdd( )在 DEV_HDR 中填入设备名称，驱动程序号，并将该设备描述结构添加到系统设备列表。

非块设备安装直接调用上述函数 iosDevAdd( )完成。块设备安装调用该设备的文件系统的设备初始化函数（dosFsDevCreate( )或者 rawFsDevInit( )），设备初始化函数仍然通过调用 iosDevAdd( )实现设备安装，如前面的介绍。

任意时候，可以在 shell 通过 iosShow 的库函数 iosDevShow( )显示当前系统中安装的所有设备：

```
-> iosDevShow
drv name
  0 /null
  1 /tyCo/0
  4 host:
  5 /pty/telnet.S
  6 /pty/telnet.M
  7 /vio
```

### 3. 设备卸载

设备不再使用时，可以通过调用 iosDevDelete( )进行卸载。函数定义为：

```
#include "iosLib.h"
void iosDevDelete ( DEV_HDR * pDevHdr );
```

参数 pDevHdr 为指向设备描述结构的指针。通常，用户程序中通过安装设备时指定的设备名称来删除设备。驱动程序调用系统提供的函数 iosDevFind( )来定位某个名称的设备描述结构。函数 iosDevFind( )可以用来检查某个名称的设备是否被安装，定义为：

```
#include "iosLib.h"
DEV_HDR * iosDevFind(char * name, char * *pNameTail );
```

下面一个例子演示了如何通过查找来删除设备。注意彻底的删除设备应该释放内存，需要显式调用 free( )。

```
STATUS xxDevDelete( char * name )
{
    DEV_HDR * pDevHdr;
    /*查找设备*/
    if ((pDevHdr = iosDevFind (name, NULL)) == NULL)
        return (ERROR);

    /*从 I/O 系统中删除，并释放内存*/
    iosDevDelete (pDevHdr);
free ((MEM_DEV *) pDevHdr);

    return (OK);
}
```

## 5.7.3 文件描述符

用户程序中所有的基本 I/O 调用（ creat/delete/open/close/read/write/ioctl 等），都由系统引导给对应设备的驱动程序所提供的相应函数（xxCreat / xxDelete / xxOpen / xxClose / xxRead / xxWrite /xxIoctl）完成。调用 open( )打开文件时，传递给 open( )的文件名称字符串表明了该文件所属的设备。系统在已经安装的设备的双向链表中查找设备描述结构中设备名称和文件名开始部分子串一致的节点，如果找到，则根据该设备描述结构中指示的驱

动程序找到对应的驱动程序函数完成 open( )功能。

调用 open( )得到了一个全局惟一的文件描述符，为了加快随后在此文件描述符上的 I/O 操作，系统需要在文件描述符和对应的驱动程序之间维护某种关联，另外，驱动程序也需要对每个打开的文件描述符维护一些状态信息。

VxWorks 通过一个全局的文件描述符表实现上面的两个需要。文件描述符表 FDT 包含驱动程序号 drvnum 和一个标识值 value，如图 5-8 所示。

| | drvnum | value |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | 2 | xxdev |
| 4 | | |

图 5-8　文件描述符表

驱动程序号在上述系统搜索文件所属设备的过程中得到，标识值是一个由驱动程序使用的 4 字节值，使得驱动程序能够标识该文件，借此维护该文件打开状态。该值由系统调用驱动程序提供的打开文件函数 xxOpen( )时返回。在一个设备对应一个文件的场合下，value 通常就是指向设备描述结构的指针。

对于非块设备，通常一个设备对应一个文件，因此标识值通常就是指向设备描述结构的指针。此时文件状态信息可以看成设备状态信息，由设备描述结构表示。

调用 open( )建立 FDT 以后，随后对该文件的其他调用就根据 FDT 的 drvnum 和 value 完成：根据 drvnum 直接定位驱动程序位置，以参数 value 和用户调用中给定的其他参数调用该位置的驱动程序函数，例如如下用户程序作调用：

```
n = read (fd, buf, len);
```

系统根据第一个参数定位 FDT，对于有效的 fd，该位置有一条有效的（devnum, value）记录，系统根据 devnum 找到驱动程序提供的 xxRead，以参数(value, buf, len)调用 xxRead，xxRead 返回值作为 read 的返回值赋给 n，所以系统执行的调用为：

```
n = xxRead (value, buf, len);
```

任意时候，可以在 shell 通过 iosShow 的库函数 iosFdShow( )显示当前 FDT 中的项目：

```
-> iosFdShow
 fd name               drv
  3 /tyCo/0             1 in out err
  4 (socket)            3
  5 (socket)            3
  6 (socket)            3
```

### select( )的实现

在 5.3 节“I/O 复用”中提到，对于用户任务而言，select( )使任务可以等待多个设备的输入；对驱动程序而言，驱动程序需要支持这一特性，即检测阻塞在设备上的任务，并在条件满足时唤起任务运行。分两种情况：

（1）驱动程序支持多个设备，任务希望同时在其中任意个设备上等待数据；

（2）任务希望等待多个不同类型的设备，这些设备由不同的驱动程序驱动。

这些功能的实现，需要设备驱动程序设计中考虑对 select( )的支持，设备驱动程序必须维护一个阻塞任务目录：SEL_WAKEUP_LIST。该目录包括一个被阻塞任务的双向链表，以及一个控制访问该列表的信号量，其中双向链表中每个“阻塞任务节点”（SEL_WAKEUP_NODE）对应一个被阻塞的任务。通常在设备描述结构中定义阻塞任务目录，作为特定设备依赖信息，如下所示（参考图 5-7）：

```
typedef struct
{
    DEV_HDR devHdr; /* 总是以设备标准头部信息开始 */
    SEL_WAKEUP_LIST selWakeupList;  /* 在该设备上的阻塞任务目录 */
    ... /*定义其他设备需要的特定数据结构 */
} DEV_X;
```

开发驱动程序不需要了解阻塞任务目录 SEL_WAKEUP_LIST 的具体细节，selectLib 提供了函数控制对目录的访问。目录需要初始化，通常在安装设备时完成，典型情况为驱动程序的 xxDevCreate( )中。当任务调用 select( )时，selectLib 调用驱动程序的 I/O 控制函数 xxIoctl( )，并给定入参 function 为 FIOSELECT 或者 FIOUNSELECT，入参 arg 为封装的指向阻塞任务节点的指针。

对于支持 select( )的驱动程序，其 xxIoctl( )必须如下处理 function 参数为 FIOSELECT 的情况：

（1）添加阻塞任务节点到阻塞任务目录：调用 selNodeAdd( )；

（2）检查任务阻塞类型：等待设备上数据就绪（SELREAD）或者等待设备写就绪（SELWRITE）；

（3）检查如果设备就绪（对 SELREAD 为设备数据到达设备，或者对 SELWRITE 为设备写就绪），则调用 selWakeup( )唤起阻塞的任务。

当入参 function 为 FIOUNSELECT 时，驱动程序调用 selNodeDelete( )删除之前加入到阻塞任务目录的阻塞任务节点。

当数据到达设备，或者设备准备好接收驱动程序输出时，驱动程序将调用 selWakeupAll( )唤起所有因为 SELREAD 或者 SELWRITE 而阻塞的设备。通常在驱动程序的 ISR 中完成这一动作。如果驱动程序在其他情况下感知到设备就绪，也可以在当时唤起阻塞任务，比如对管道驱动或者内存虚拟设备驱动，设备就绪不需要通过中断感知。

总的来说，驱动程序对 select( )的支持包括 3 方面的工作：

（1）在 I/O 控制函数 xxIoctl( )中实现对 funtion 为 FIOSELECT 和 FIOUNSELECT 的处理；

（2）在设备就绪时调用 selWakeupAll( )唤起阻塞的任务，可能在 ISR 中，或者其他地方如 xxRead( )/xxWrite( )；

（3）实现上述支持需要定义数据结构 SEL_WAKEUP_LIST 并在创建设备时初始化。

## 5.7.4 块设备驱动

在 VxWorks 中，块设备驱动程序和其他 I/O 设备驱动有差别，如图 5-9 所示：

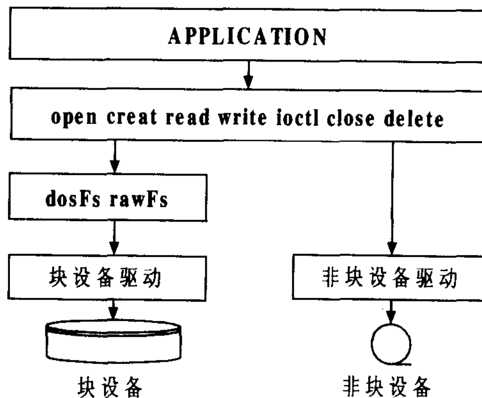



图 5-9 块设备驱动和非块设备驱动

在图 5-9 中，可以看出：（1）块设备驱动通过文件系统对 I/O 系统提供接口，块设备驱动不能直接和 I/O 系统打交道；（2）一个块设备驱动可以支持不同文件系统。

### 1. 随机块设备

VxWorks 从 SCSI-1 开始支持随机访问块设备，兼容 dosFs 和 rawFs 文件系统。

对高层程序而言，它所看到的随机块设备是一个可以通过一个线性的块地址对设备进行随机访问的设备。这种线性地址称为 LBA（Linear Block Address）地址，“块”是基本数据传送单位，块的大小在创建设备时得到的设备描述结构 BLK_DEV 中有表示（见表 BLK_DEV 定义）。

随机块设备驱动 xxBlkDrv（如 IDE 硬盘驱动 ataDrv）实现了这种抽象接口。下面是一个随机块设备驱动 xxBlkDrv 所必须提供的读写和 I/O 控制函数：

- 读块设备

```
STATUS xxBlkRd
(
```

```
    XX_DEVICE *  pDev,    /* 标识块设备的设备描述结构 */
int       startBlk,   /* 起始块编号 */
    int       numBlks,    /* 要读取的块数 */
    char *    pBuf        /* 数据存放缓存区 */
);
```

- 写块设备

```
STATUS xxBlkWrt (
    XX_DEVICE *  pDev,    /* 标识块设备的设备描述结构 */
    int       startBlk,   /* 起始块编号 */
    int       numBlks,  /* 要写的数据块数 */
    char *    pBuf      /* 源数据所在缓冲区 */
);
```

- I/O 控制

```
STATUS xxIoctl (XX_DEVICE * pDev, int funcCode, int arg );
```

在上面的 3 个函数中，都需要一个用于表示块设备的设备描述结构 XX_DEVICE。和前面介绍的其他驱动程序一样，块设备 xxBlkDrv 提供设备创建函数 xxBlkCreate( )，该函数返回一个指向所创建设备的设备描述结构 XX_DEVICE 的指针，从数据结构上代表该设备。XX_DEVICE 实现两方面功能：

（1）高层程序通过该结构访问 xxBlkDrv 提供的接口，除了上面介绍的读写和 I/O 控制等函数外，还包括许多状态变量定义；

（2）维护足够的信息使 xxBlkDrv 实现对设备的读写和 I/O 控制。

如图 5-10 所示，XX_DEVICE 总是以高层访问接口的标准结构 BLK_DEV 开始，该结构在 blkIo. h 中定义，如表 5-20 所示。

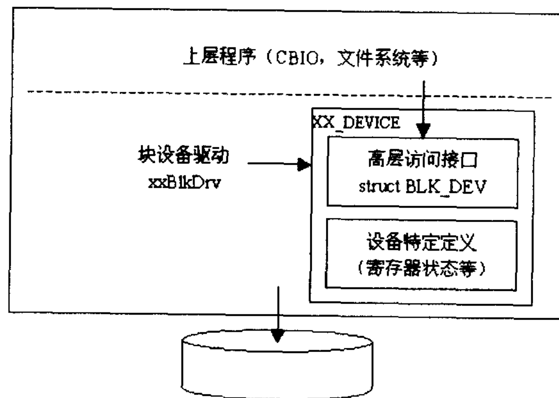



图 5-10　块设备描述结构

表 5-20 BLK_DEV 定义

| | |
|---|---|
| bd_blkRd | 读块设备函数地址 |
| bd_blkWrt | 写块设备函数地址 |
| bd_ioctl | 块设备 I/O 控制函数地址 |
| bd_reset | 复位块设备函数地址（可以为 NULL） |
| bd_statusChk | 检查设备状态函数地址（可以为 NULL） |
| bd_removable | 可移动标志。对于可移动媒介，该值为 TRUE；否则为 FALSE |
| bd_nBlocks | 块设备上块总数 |
| bd_bytesPerBlk | 每块字节数 |
| bd_blksPerTrack | 每磁道块数（扇区数） |
| bd_nHeads | 磁头数 |
| bd_retry | 读写失败时重试次数 |
| bd_mode | 写保护模式，可以为 O_RDONLY/O_WRONLY/O_RDWR<br>一般为可读写 O_RDWR，上层程序读该状态判断可以进行的操作 |
| bd_readyChanged | 设备就绪状态已改变。块设备驱动检测到可移动的设备被移除或者改变时设置该标志为 TRUE，由高层程序（CBIO 或者文件系统）负责解释 |

除了最重要的读写和 I/O 控制函数外，BLK_DEV 中提供的函数还包括设备复位和设备状态查询函数。

VxWorks 中典型的块设备驱动是 ATA/IDE 硬盘驱动 ataDrv 和 SCSI 硬盘驱动 scsiLib/scsi1Lib/scsi2Lib。

## 2. 序列块设备

VxWorks 从 SCSI-2 开始支持序列块设备。典型的序列块设备是磁带。

对高层程序而言，序列块设备以“块”为基本存取单位，块的大小是可变的，通过“块间间隙”来判断块的开始和结束。整个序列块设备上的数据可以全部以数据块的形式存在，还可以通过“文件头标”和“文件尾标”组织成文件形式。一个文件内的文件头标/尾标与数据块存在前后两个“文件间隙”。

高层程序访问序列块设备比访问随机块设备具有更多限制：(1) 只能以跳过多少个“块间间隙”或者“文件间隙”的方式进行顺序寻址；(2) 数据块只能写在媒质末尾，不能替换中间的数据。

高层程序所感觉到的上述序列块设备特性决定了设备驱动程序应该提供的接口函数，该接口在 seqIo.h 中定义为 SEQ_DEV，如表 5-21 所示。

序列块设备驱动在 VxWorks 层次化的 I/O 系统中处于相同的地位，具有类似的结构，只是图 5-10 中的 struct BLK_DEV 被换成了 struct SEQ_DEV。

表 5-21　SEQ_DEV 定义

| | |
|---|---|
| sd_seqRd | 读块设备函数地址 |
| sd_seqWrt | 写块设备函数地址 |
| sd_ioctl | 块设备 I/O 控制函数地址 |
| sd_seqWrtFileMarks | 写文件头标/尾标的函数地址 |
| sd_rewind | 控制倒带函数地址 |
| sd_reserve | 预约设备以实现独占访问 |
| sd_release | 释放预约使设备 |
| sd_readBlkLim | 读设备块大小限制 |
| sd_load | 装载或者卸载设备 |
| sd_space | 控制磁头向前/后移动指定个数的块间间隙或者文件间隙 |
| sd_erase | 擦除整个序列块设备上记录的数据 |
| sd_reset | 复位设备（可以为 NULL） |
| sd_statusChk | 检查设备状态（可以为 NULL） |
| sd_blkSize | 设备块大小（如果块大小可变，该项为 0） |
| sd_mode | 写保护模式，可以为 O_RDONLY/O_WRONLY/O_RDWR<br>一般为可读写 O_RDWR，上层程序读该状态判断可以进行的操作 |
| sd_readyChanged | 设备就绪状态已改变。块设备驱动检测到可移动的设备被移除或者改变时设置该标志为 TRUE，由高层程序（CBIO 或者文件系统）负责解释 |
| sd_maxVarBlockLimit | 允许的最大块大小 |
| sd_density | 媒介存储密度 |

# 5.8　串口 tty 设备

## 5.8.1　串口的层次

串口通信由串行通信控制器（SCC，Serial Communication Controller）控制。一个 SCC 芯片一般有 2~4 个通道（CHAN），一个通道物理上对应一个串口，每个 SCC 通道有独立的通道缓冲区（一般几个到十几个字节大小），能独立地进行配置；在控制上，软件为每个通道维护一组控制结构信息。

对用户程序而言，串口以串口终端(tty)设备方式访问。为了实现这种访问，从 VxWorks 5.3 开始，系统采用了 3 层抽象的软件结构：标准 I/O 库（ioLib）→tty 库（ttyDrv/tyLib）→底层 SCC 驱动（xxDrv），如图 5-11 所示。

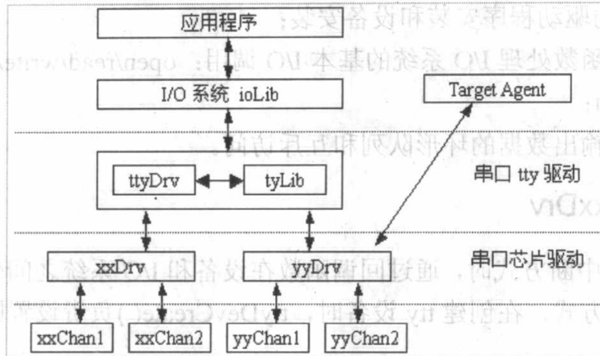



图 5-11　串口软件结构

在 VxWorks 5.2（含）以前是通过一个驱动库 tyCoDrv 实现从 ioLib 到 SCC 的接口。由于 SCC 芯片种类繁多，变化很大，这样设计导致 tyCoDrv 非常复杂，而且不利于系统移植。因此，VxWorks 5.2 以后将 tyCoDrv 中相对稳定的部分拿出来做成了 ttyDrv。

在 Tornado 安装目录<install-dir>/target/src/drv/serial/下还可以看到 tyCoDrv 驱动库的实现代码；目录<install-dir>/target/src/drv/sio/下是 SCC 驱动 xxDrv 的实现代码；tty 库以目标码形式提供。

串口 tty 驱动（ttyDrv/tyLib）使 I/O 系统独立于具体 SCC 驱动，可以同时支持多个不同类型的 SCC 驱动。SCC 驱动（xxDrv/yyDrv）实际处理不同串口芯片的细节，对串口 tty 驱动提供标准设备 I/O 接口和 target agent 两种接口。串口芯片驱动支持中断和查询两种工作模式。

## 1．tty 驱动：ttyDrv/tyLib

在“I/O 设备驱动程序结构”一节，我们介绍驱动程序安装时将在驱动程序表中设置 7 个基本 I/O 调用的驱动程序函数，系统将把用户程序的基本 I/O 调用引导给这些函数。在 tty 驱动方面，用户程序也通过基本 I/O 实现串口输入输出，但是和其他驱动程序不同的是，tty 驱动分别由两个库 ttyDrv 和 tyLib 实现，如图 5-12 所示。

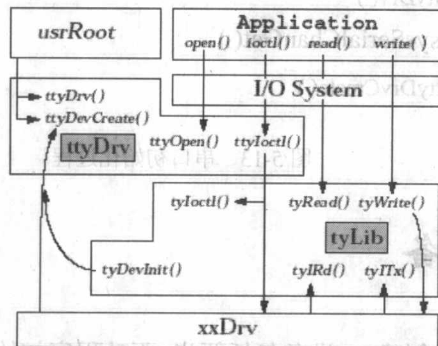



图 5-12　ttyDrv 和 tyLib

- I/O 系统的驱动程序安装和设备安装；
- 驱动程序函数处理 I/O 系统的基本 I/O 调用：open/read/write/ioctl 等；
- 支持 select；
- 维护输入输出数据的环形队列和互斥访问。

2．SCC 驱动：xxDrv

SCC 驱动采用中断方式时，通过回调函数在设备和 I/O 系统之间传送数据，tty 设备需要使用串口的中断方式，在创建 tty 设备时，ttyDevCreate( )负责设置回调函数。

## 5.8.2 串口初始化过程

串口的初始化是 I/O 设备初始化中最复杂的部分：

（1）从涉及的库看，包括串口芯片驱动 xxDrv 的初始化和 tty 库 ttyDrv/tyLib 初始化，tty 设备创建；

（2）初始化时间看，分多次完成；

- 内核启动之前：函数 usrInit( )初始化串口，使串口通过查询方式实现系统级调试。由 sysHwInit( )调用函数 sysSerial.c 中定义的 sysSerialHwInit( )实现；
- 用户根任务 usrRoot( )中；
- tty 设备创建。

如图 5-13 所示。

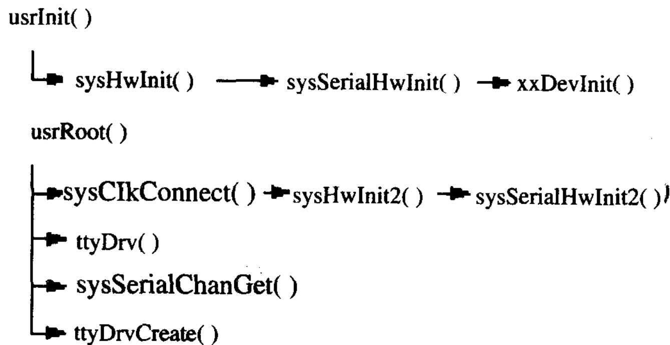



图 5-13　串口初始化过程

## 5.8.3 创建 tty 设备

和其他 I/O 设备一样，创建 tty 设备包括两步：驱动程序初始化和创建设备。在 usrRoot( )中完成。

## 1. 驱动程序初始化

初始化函数：

```
STATUS ttyDrv (void);
```

该函数将调用 iosDrvInstall( )在系统的驱动程序表中安装 tty 驱动程序。所有的用户应用的串口 I/O 请求通过 ttyDrv 和 tyLib 实现。

如图 5-12 所示，ttyDrv( )安装的函数分别来自 ttyDrv 和 tyLib：

- ttyDrv 负责 open 和 ioctl（ttyOpen( )和 ttyIoctl( )）；
- tyLib 负责 read 和 write（tyRead( )和 tyWrite( )）；
- 设备创建在系统初始化时完成，creat( )和 close( )对应的驱动程序函数为 NULL。

## 2. 创建设备

创建设备函数：

```
STATUS ttyDevCreate (devName, pSioChan, rdBufSize, wrtBufSize);
```

该函数完成的工作包括：

- 分配并初始化设备描述结构空间；
- 进行 tyLib 库初始化，tyLib 提供初始化函数 tyDevInit( )：selectLib 初始化，创建输入输出的环形队列，创建信号量；
- 调用 iosDevAdd( )将设备加入设备列表；
- 安装 tyLib 提供的输入输出回调函数，底层 SCC 驱动负责调用；
- 以中断方式启动串口通道。

上述驱动程序初始化和设备创建在系统初始化 usrRoot( )中完成。从 VxWorks 5.3 开始的 tty 初始化如下面的代码所示：

系统初始化：usrRoot( )

```
ttyDrv( );     /* 初始化 tty 驱动程序 */
for (ix = 0; ix < NUM_TTY; ix++)
{
    sprintf (tyName, "%s%d", "/tyCo/", ix);
    (void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);

    if (ix == CONSOLE_TTY)
    {   /* init the tty console */
        strcpy (consoleName, tyName);
        consoleFd = open (consoleName, O_RDWR, 0);
        (void) ioctl (consoleFd, FIOBAUDRATE, CONSOLE_BAUD_RATE);
        (void) ioctl (consoleFd, FIOSETOPTIONS, OPT_TERMINAL);
    }
}
```

其中，宏 NUM_TTY 表示串口总数，在<install-dir>/target/config/target/config.h 中定义。

## 5.8.4 tty 输入输出

用户程序的基本 I/O 读写请求由 tyLib 的 tyRead( )和 tyWrite( )实现。函数 ttyDrv( )初始化 tty 时在驱动程序表对应 read( )和 write( )调用的位置安装指向这两个函数指针，如图 5-13 所示。

### 发送过程

用户程序调用 write(fd, buf, len)，系统根据 fd 定位到 tyWrite( )，将入参用户缓冲区地址 buf 和发送数据字节数 len 传递给 tyWrite( )。tyWrite( )复制 buf 到输出环形队列，如果输出环形队列满，tyWrite 将阻塞。

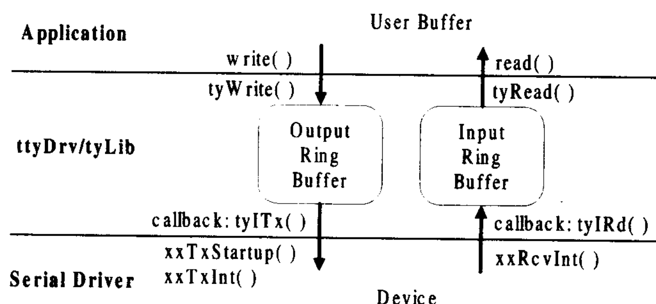



图 5-13 tty 输入输出

如果设备发送已经被挂起，tyWrite( )调用 SCC 驱动 xxDrv 的启动发送程序。

芯片细节：当串口芯片的某个通道的发送缓冲区空时，该通道将发出发送空中断。驱动程序处理该通道中断时将新数据写入通道的发送缓冲区。如果没有更多的数据要发送，该通道将挂起发送。挂起发送后将不能产生发送中断，直到 SCC 驱动写入新数据到通道的发送缓存启动该通道的新一轮发送。因此，如果设备发送已经被挂起，tyWrite( )需要提示 SCC 驱动 xxDrv 的启动发送程序。串口芯片处于发送状态时，将再次发送。

## 5.8.5 控制 tty

用户程序通过 ioctl(fd, command, arg)对 tty 设备进行 I/O 控制。用户可以使用的 command 及其意义如表 5-22 所示。

表 5-22　tty 控制命令

| 命　　令 | 含　　义 |
| --- | --- |
| FIOBAUDRATE | 设置波特率<br>arg 为一整数表示要指定的波特率 |
| FIOGETOPTIONS | 读取 tty 选项<br>arg 表示读出选项存放位置 |
| FIOSETOPTIONS | 设置 tty 选项<br>arg 表示设置的选项 |
| FIOGETNAME | 根据文件描述符 fd 取得文件名称，等于 tty 名称<br>arg 为存放读出 tty 名称的缓冲区 |
| FIONREAD | 取得输入环形队列中没有读取的字节数<br>arg 用于接收结果的整型指针 |
| FIONWRITE | 取得输出缓冲区中字节数<br>arg 用于接收结果的整型指针 |
| FIOFLUSH | 丢弃在 tyLib 输入输出环形队列中的数据 |
| FIOCANCEL | 取消一个读/写操作 |

### 1. 底层 SCC 驱动的实现

上表中设置波特率的命令 FIOBAUDRATE 通过底层 SCC 驱动 xxIoctl( )实现。调用 xxIoctl( )之前，ttyIoctl( )将命令 FIOBAUDRATE 转换为 xxIoctl ( )识别的命令 SIO_BAUD_SET。

如果 tty 对其他通信格式有要求，比如字长、停止位数、流控协议等，则需要底层 SCC 驱动支持另一组 I/O 控制命令：SIO_HW_OPTS_SET。

### 2. 设置 tty 模式

VxWorks 的 tty 驱动有两种工作模式：原始模式（RAW）和行模式（LINE）。

设置 tty 模式用命令 FIOSETOPTIONS 实现，设置原始模式：

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

或者设置行模式：

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

原始模式：底层 SCC 驱动（通过 tyLib 的回调函数）将数据从设备读到 tyLib 的输入环形队列之后，用户程序可以立即通过 read 调用将每个字符读到用户程序缓冲区；

行模式：tty 驱动在输入和输出过程中实现一个行规则，将输入的原始字节流以行为单位提交给用户程序。行规则使用换行符 NEWLINE（C 语言表示为‘\n’，ASCII 码值为 0x0a）表示行结束，行包括 NEWLINE 自身。行规则可以实现一些编辑和转换功能，通过行模式选项控制。

如图 5-14 所示行模式和原始模式。

打开 tty 时默认使用原始模式。

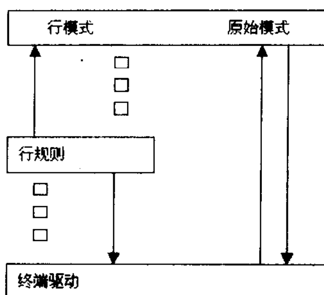



图 5-14　行模式和原始模式

### 3. 行模式选项

行模式有许多选项用于控制行规则，选择这些选项通过命令 FIOSETOPTIONS 调用 ioctl( )，指定参数 arg 为下列选项或组合，如希望 tyLib 实现软流控，可以：

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

如表 5-23 所示行模式选项：

表 5-23　行模式选项

| 选　项 | 说　明 |
|---|---|
| OPT_ECHO | 自动回送输入的字符。tyLib 简单地将写到输入环形队列的字符同时加入到该通道输出环形队列尾部，但是不会阻塞，如果输出环形队列满，tyLib 丢弃要回送的字符 |
| OPT_CRMOD | 设置回车换行模式。许多终端在按回车键时发送字符 CR（ASCII 码值 0x0d）而向他们输出 CR 时终端只将光标移到行首，还需要一个 LF（ASCII 码值 0x0a）才能实现许多程序需要的回车换行效果<br>如果指定该选项，则 tyLib 自动将终端输入的 CR 转换为 NEWLINE，将用户输出给中断的 NEWLINE 转换为 CR+LF |
| OPT_TANDEM | 启用 tyLib 的软流控支持。终端一般支持软流控，当缓冲区将要溢出时发送 XOFF，缓冲区恢复空余时发送 XON，还可以在终端键盘按下 CTRL+Q 或者 CTRL+S 来发送 XON 和 XOFF<br>如果指定该选项，tyLib 将在收到上述字符时设置相应的 tty 状态以实现软流控。同时 tyLib 也会根据缓冲区状态向终端发送 XON 和 XOFF |
| OPT_7_BIT | 指定字符位数为 7 位<br>和底层 SCC 驱动不同，tyLib 通过去掉收到字符的最高位实现，而 SCC 驱动通过 SCC 硬件实现 |
| OPT_MON_TRAP | 允许通过按 CTRL+X 进入 ROM 监控程序<br>如果启用该选项，则从 shell 按下 CTRL+X 时会自动调用 reboot( )，该程序使系统转入 ROM 中一固定地址使系统重新启动<br>默认 CTRL+X 可以通过 tyMonitorTrapSet (new MonTrapChar)修改 |

续表

| 选 项 | 说 明 |
| --- | --- |
| OPT_ABORT | 允许通过按 CTRL+C 使 shell 重启<br>默认 CTRL+C 可以通过 tyAbortSet(newAbortChar)修改 |
| OPT_TERMINAL | 相当于 OPT_ECHO\| OPT_CRMOD\| OPT_TANDEM \| OPT_7_BIT \| OPT_MON_TRAP \| OPT_ABORT\| OPT_LINE |

tty 工作在行模式时，可以在终端输入下列特殊字符进行编辑，如表 5-24 所示：

表 5-24 行模式特殊字符

| | |
| --- | --- |
| 退格（CTRL+H） | 删除当前行当前位置上一个字符。驱动程序向串口回送一个 ASCII 字符的退格+空格+退格<br>重新设置：tyBackspaceSet( newBackChar) |
| 删除行（CTRL+U） | 删除当前行所有字符<br>重新设置：tyDeleteLineSet(newDelLineChar) |
| EOF（CTRL+D） | 文件结束字符（EOF）。收到该字符使当前行提前结束，但是和 NEWLINE 不同的是 EOF 本身不是行的一部分<br>重新设置：tyEOFSet(newEOFChar) |

## 4. 防止 tty 阻塞

相对于其他 I/O 设备，如 socket 和磁盘，tty 设备的一个特点是速度慢，可靠性低，容易因为丢失数据而使 I/O 操作阻塞。因此防止 tty 阻塞就显得更加重要。一般防止读写操作在 tty 设备上阻塞的方法包括：读写之前检查，I/O 复用，使用 AIO，或者对于已经阻塞的调用，可以通过 FIOCANCEL 将其取消。

**（1）在 read/write 前检查**

读写之前检查通过 ioctl( )调用完成，最多是用在 read( )操作之前，例如：

```
int nbytes;
if ((ioctl(fd, FIONREAD, (char *)&nbytes) != ERROR) && (nbytes > 0))
{
    /* Will I block getting here? */
    read from fd;
}
```

**（2）FIOCANCEL**

命令 FIOCANCEL 用于取消 tty 读写操作。任务调用 read/write 后被阻塞，直到 I/O 完成。为了防止无限等待，用户程序做如下标准 I/O 调用：

```
ioctl ( fd, FIOCANCEL, NULL );
```

# 5.9 编写 SCC 驱动

在 5.8 节“串口 tty 驱动”中，我们介绍的 tty 驱动 ttyDrv/tyLib 属于系统提供的一般化代码，通过本节将要介绍的 SCC 驱动为 I/O 系统提供脱离设备细节的接口。下面介绍直接驱动底层串口芯片的 SCC 驱动程序的编写。这部分内容对编写新串口驱动或彻底了解串口工作过程有用，但是具体的串口驱动编程涉及不同的串口控制芯片而不同，本节给出的是一般性和示例性的做法。

## 5.9.1 tty 数据结构

### 1．tty 驱动程序表

如图 5-15 所示 tty 驱动程序表：

| 驱动程序号 | | | open | | read | write | ioctl |
|---|---|---|---|---|---|---|---|
| … | | | | | | | |
| 7 | | | ttyOpen | | tyRead | tyWrite | ttyIoctl |
| … | | | | | | | |

图 5-15　tty 驱动程序表

### 2．tty 设备描述结构 struct TY_DEV

如图 5-16 所示 tty 设备描述结构：

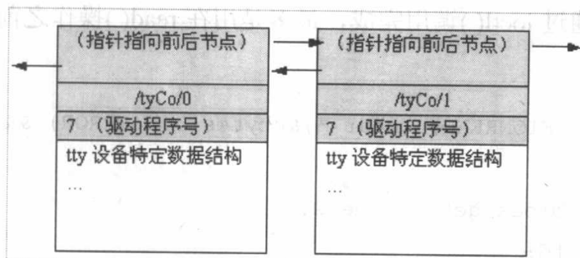



图 5-16　tty 设备描述结构

阴影部分表示标志设备头部信息（结构类型 DEV_HDR）。TY_DEV 中 tty 设备特定数据结构包括：

- 接收环形队列 rdBuf;
- 接收状态;

- 发送数据环形队列 wrtBuf；
- 发送状态；
- 同步和互斥访问信号量；
- 指向底层 SCC 驱动 xxDrv 提供的启动发送的函数指针；
- 阻塞任务目录。

系统初始化时，ttyDrv( )和 ttyDevCreate( )建立 tty 驱动程序表和 tty 设备描述结构。

在初始化过程中，ttyDevCreate( )还进行的另一个重要的操作是设置了 tty 设备对应串口通道描述符中的回调函数，正是这一点使得特定芯片的 xxDrv 能够读写一般化的 tyLib 的输入输出环形队列，这一过程在下面通道描述符部分详细介绍。

## 3. tty 文件描述符表

tty 文件描述符表记录了每个 tty 设备的驱动程序信息，如图 5-17 所示：

| fd | 驱动程序号 | 驱动程序标识 |
|---|---|---|
| 0 | | |
| 1 | | |
| 2 | 7 | pDevTyCo0 |
| 3 | 7 | pDevTyCo1 |
| 4 | | |

图 5-17 tty 文件描述符表

下面说明根据上述 3 组数据结构，在设备“/tyCo/0”上进行 I/O 的过程：

**（1）打开 tty**

如图 5-18 所示打开串口 tty 操作过程：

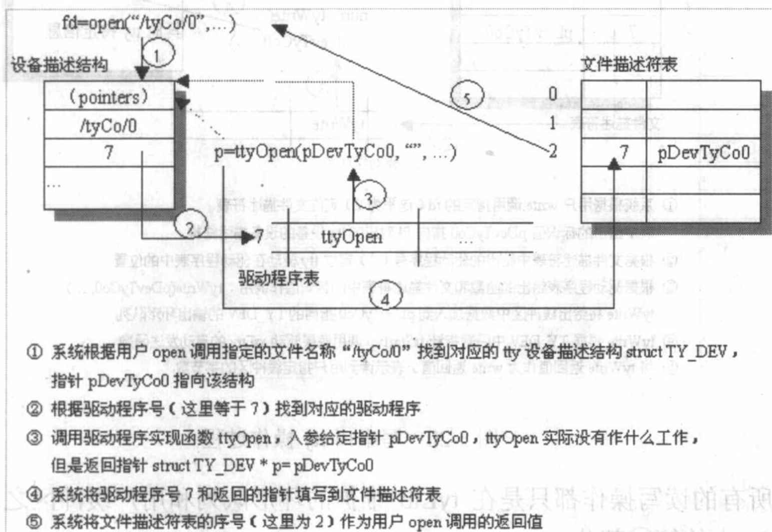



图 5-18 打开串口 tty 操作过程

**（2）tty 读**

如图 5-19 所示读串口 tty 操作过程：

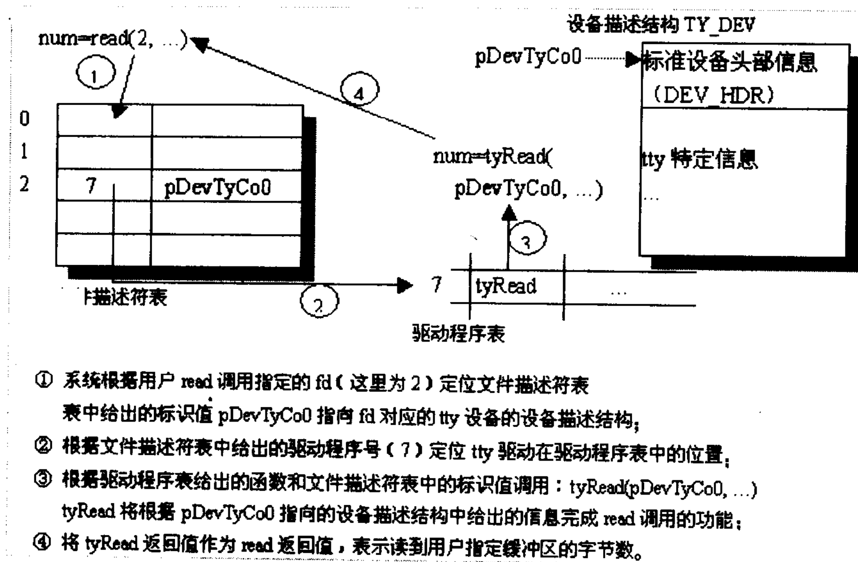



图 5-19　读串口 tty 操作过程

**（3）tty 写**

如图 5-20 所示写串口 tty 操作过程：

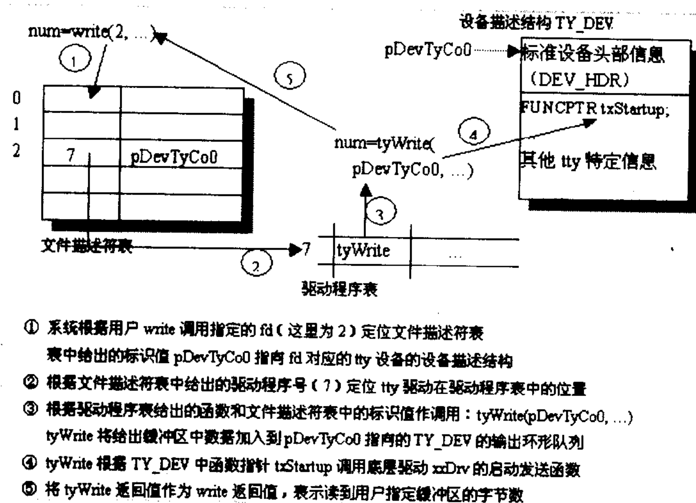



图 5-20　写串口 tty 操作过程

这里，所有的读写操作都只是在 tyLib 维护的环形队列和用户缓冲区之间进行，实际上属于图 5-21 中的第①部分。

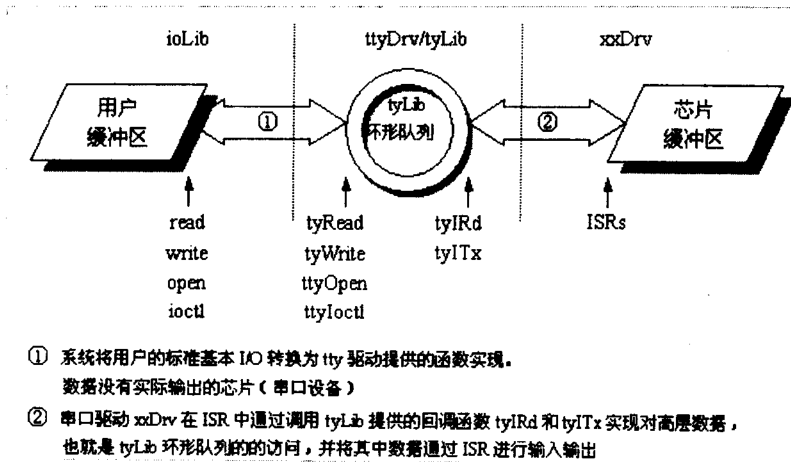



图5-21　串口输入输出的几个缓冲区

下面介绍第②部分的实现细节。

## 5.9.2　xxDrv数据结构

SCC驱动xxDrv属于BSP范围，系统已经不为xxDrv维护驱动程序表，xxDrv也没有设备描述结构和文件描述符。对xxDrv最核心的数据结构就是通道描述符。

通道描述符需要实现的两个主要目的：

（1）作为xxDrv和高层驱动之间的一种耦合手段（回调函数和启动发送的函数指针）；

（2）xxDrv本身用来维护串口通道状态。

### 通道描述符

SCC驱动xxDrv为每个串口通道维护一个通道描述符struct XX_CHAN，该结构在<install-dir>/target/h/sioLib.h中定义，包括一个指向xxDrv提供的SCC驱动函数结构struct SIO_DRV_FUNCS的指针，如图5-22所示：

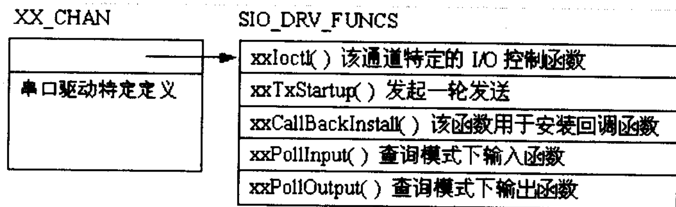



图5-22　串口通道描述符XX_CHAN

通道描述符XX_CHAN的第一个变量必须定义为指向SIO_DRV_FUNCS的指针，因

为不同驱动程序定义的各种不同类型描述符 XX_CHAN 要统一以结构指针 SIO_CHAN 的形式传递给高层驱动 ttyDrv。高层驱动只需要引用 SIO_DRV_FUNCS 中定义的函数。

XX_CHAN 的其他部分由驱动程序自行定义。一般这部分都包括对回调函数指针的定义。回调函数 tyIRd 和 tyITx 由 tyLib 提供，在系统初始化时由 ttyDevCreate( )负责安装到 XX_CHAN 的回调函数指针。SCC 驱动 xxDrv 在 ISR 处理通道发送中断时调用回调函数从 tyLib 的输出环形队列中取数据，或者在处理通道接收中断时将收到数据写到输入环形队列，如图 5-23 所示。

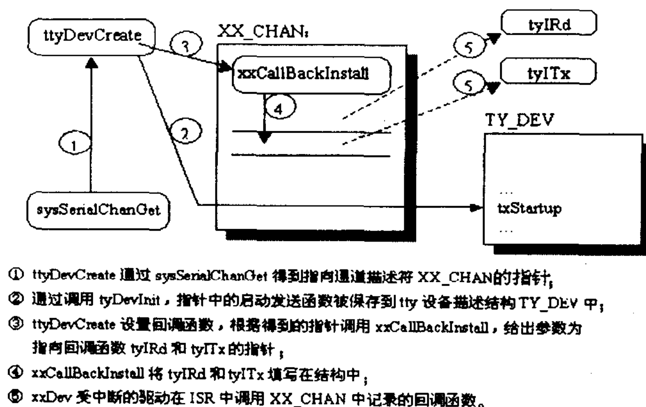



图 5-23　SCC 驱动 xxDrv 和高层 tty 驱动之间的通信

XX_CHAN 还需要定义其他信息，包括通道控制状态（如中断状态，波特率，流控状态），以及必需的硬件寄存器定义等。通道描述符建立了 SCC 驱动 xxDrv 和高层 tty 驱动之间的通信媒介。

**实例：XX_CHAN**

```
typedef struct foo_chan
{
    /* always the first var */
    SIO_DRV_FUNCS * pFooDrvFuncs;
    /* callbacks to higher level protocols */
    STATUS   (*getTxChar) ( );
    STATUS   (*putRcvChar) ( );
    void *   getTxArg;
    void *   putRcvArg;
    /* Device specific data */
    volatile char * cr;  /* control register */
    volatile char * dr;  /* data register */
    ...
```

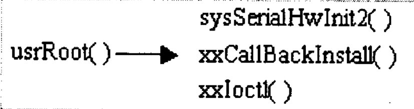
```
    /* misc. */
    int mode;     /* interrupt or poll mode */
    int baudRate;
    BOOL isrInstalled;
    ....
} FOO_CHAN;
```

## 5.9.3 xxDrv 程序结构

实际上，系统和高层驱动（ttyDrv/tyLib）希望 xxDrv 是透明的。

第一步，系统在内核启动之前，在 usrInit 中进行第一步初始化后，串口 SCC 被复位，禁止中断，串口能够通过查询方式访问，实现系统级调试。

第二步，系统激活内核之后，在根任务 usrRoot( )中进行第二步初始化，使串口可以以中断方式实现对 tty 库的底层支持。




实现上述对高层的透明，结合前面介绍的 SCC 通道描述符，xxDrv 遵循一种规范的函数结构，如图 5-24 所示。由于设计目标不同，或者不同 SCC 芯片支持的功能差异，该结构中的有些函数的功能可以不实现，但是应该具有接口。比如 xxDrv 可以不实现对查询的支持，但是应该具有查询函数接口（即函数简单的返回 ERROR）。

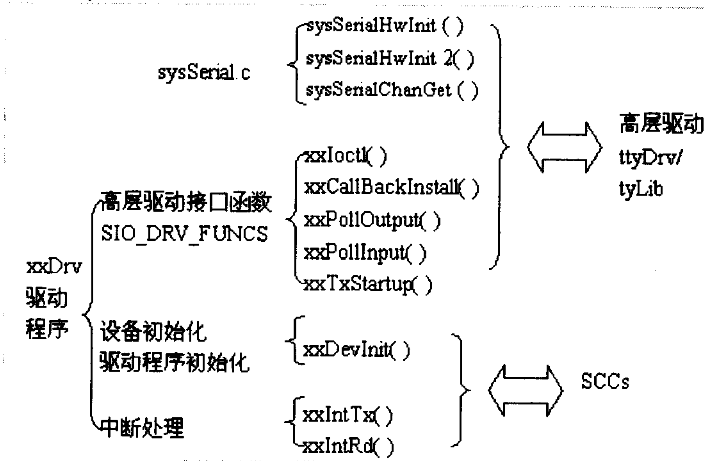



图 5-24 xxDrv 程序结构

设备初始化放在 sysSerial.c 中实现。严格来讲，sysSerial.c 也属于 SCC 驱动的一部分。只是该文件位于特定的目录<install-dir>/target/config/<bspname>/下。

编程步骤：

- 定义支持的串口通道数目；
- SCC 初始化代码：初始化数据结构 XX_CHAN，设备初始化；
- 编写高层驱动接口函数 SIO_DRV_FUNCS；
- 编写 ISR；
- 编写查询支持。

参考：

- SIO 模板：<install-dir>/target/src/drv/sio/templateSio.c；
- 其他 SIO 驱动：<install-dir>/target/src/drv/sio/*.c；
- <install-dir>/target/config/<bspname>/sysSerial.c；
- <install-dir>/target/config/<bspname>/sysLib.c。

### 1. SCC 初始化

SCC 初始化分两步。

第一步使串口复位在查询模式下。包括 SCC 硬件初始化和 SCC 驱动 xxDrv 软件初始化，分别在函数 xxDevInit( )和 sysSerialHwInit( )中实现。在 wind 内核被激活之前调用：

usrInit( ) → sysHwInit( ) → sysSerialHwInit( ) → xxDevInit( )

第二步初始化由 sysSerialHwInit2( )完成，使串口开始在中断模式下运行。

- **sysSerialHwInit( )**

（1）初始化通道描述符 XX_CHAN 的硬件依赖部分；

（2）得到一个 SCC 通道描述符 XX_CHAN 指针，作入参调用 xxDevInit( )，xxDevInit( )完成该 SCC 通道的实际的软硬件初始化。

- **xxDevInit( )**

和所有在 sysHwInit( )中被调用的初始化函数一样，xxDevInit( )复位对应的 SCC 通道到不能产生中断的静止状态。xxDevInit( )只初始化一个 SCC 通道，sysSerialHwInit( )循环调用 xxDevInit( )实现对整个 SCC 上所有通道的初始化。sysSerialHwInit( )通过另一个函数 sysSerialChanGet( )将得到一个 SCC 通道描述符 XX_CHAN 指针，作为参数传递给 xxDevInit( )，使 xxDevInit( )知道需要初始化的通道。

**注意：**

sysSerialHwInit( )和 xxDevInit( )这种分工的必要性在于 xxDevInit( )专注于和 SCC 硬件特性和 xxDev 数据结构相关的初始化；而 sysSerialHwInit( )专注于和其他 BSP 特性，如 CPU 体系中断向量结构，内存映射，总线访问等相关的初始化（通常是对 SCC 的各种寄存器地址和中断向量的初始化），如：

```
fooChan[i].cr = ADDRESS_OF_SCC_REG_ON_TARGET_BOARD;
```

有了 sysSerialHwInit( )初始化执行在前，xxDevInit( )初始化 SCC 硬件时不必考虑 SCC 上寄存器映射的地址，如要设置控制寄存器 cr，可以：

```
* (fooChan[i].cr) = some_value;
```

上面的语句用宏实现比较好（考虑 I/O 地址映射方式对上述语句的影响）。

**实例：sysSerialHwInit( )**

```
void sysSerialHwInit (void)
{
    int i;
    for (i = 0; i < N_UART_CHANNELS; i++)
    {   /* 初始化每个 CHAN 的硬件依赖部分
         * 如依赖于 BSP 硬件配置的寄存器地址等
         */
        fooChan[i].cr = ...;
        fooChan[i].dr = ...;
        ...
        /* 调用 CHAN 初始化 */
        fooDevInit (&fooChan[i]);
    }
}
```

**实例：xxDevInit( )**

```
static SIO_DRV_FUNCS fooSioDrvFuncs =
{
    fooIoctl,
    fooTxStartup,
    fooCallbackInstall,
    fooPollInput,
    fooPollOutput
};
void fooDevInit (FOO_CHAN * pFooChan)
{
    /* 初始化 SIO_DRV_FUNCS 指针 */
    pFooChan->pDrvFuncs  = &fooSioDrvFuncs;

    /* 回调函数初始时只简单返回 ERROR */
```

```
    pFooChan->getTxChar  = fooDummy;
    pFooChan->putRcvChar = fooDummy;

    /* CHAN 硬件复位，与具体的 SCC 控制方式有关 */
    fooHrdInit( pFooChan );

    /* 设置 CHAN 工作在查询模式下 */
    fooIoctl ((SIO_CHAN *)&pFooDrv->portA,
    SIO_MODE_SET, (void *) SIO_MODE_POLL);
}
static int fooDummy (void)
{
    return (ERROR);
}
```

- **sysSerialInit2( )**

该函数连接 SCC 通道中断的 ISR：

```
intConnect ( vector, fooIsr, (int)pSioChan);
```

第三个参数必须是一个指向通道描述符 XX_CHAN 的指针。

该函数具体实现细节与处理器体系及目标板设置有关，差异表现在有些配置下可以为每个 SCC 通道的每种中断类型连接一个中断向量，有些配置下每个通道或者每个 SCC 共享一个中断向量。这样在连接时和所连接的 ISR 内有不同的处理。下面是一个简单的实例。

**实例：sysSerialInit2( )**

```
void sysSerialHwInit2 (void)
{
    int i;
    for (i = 0; i < N_UART_CHANNELS; i++)
    {   intConnect (
                INUM_TO_IVEC(THE_INT_NUM),
                (VOIDFUNCPTR)the_isr,
                (int)pxxChan
            );
        ...
    }
}
```

### 2. 高层驱动接口函数 SIO_DRV_FUNCS

XX_CHAN 必须定义的第一个变量就是指向结构 SIO_DRV_FUNCS 的指针。通过该指针，高层程序可以访问 xxDrv 提供的下面 5 个接口函数。这 5 个函数都接收一个 SIO_CHAN 型指针变量。当被高层程序调用时，这 5 个函数通过该指针识别所属的 SCC 通道以及相关的通道描述符 XX_CHAN。

**回调函数安装函数 xxCallBackInstall( )**

安装函数采取如下标准形式：

```
int xxCallbackInstall (
        SIO_CHAN* pSioChan,
        int         callbackType,
        STATUS    (*callback)( ),
        void *      callbackArg
);
```

回调函数分发送回调函数和接收回调函数，分别有入参 callbackType 为 SIO_CALLBACK_GET_TX_和 CHARSIO_CALLBACK_PUT_RCV_CHAR 表示。入参 callback 为指向回调函数的指针，callbakcArg 为回调函数接收参数的地址。安装函数将第 3，第 4 个指针参数记录在 pSio Chan 指向的通道描述符 XX_CHAN 中，回调函数对每个通道单独设置，而不是对整个 SCC 设置。

**实例：xxCallbackInstall**

```
static int fooCallbackInstall
(
    SIO_CHAN * pSioChan,
    int callbackType,
    STATUS (*callback)( ),
    void * callbackArg
)
{
    FOO_CHAN * pFooChan = (FOO_CHAN *)pSioChan;
    switch (callbackType)
    {
    case SIO_CALLBACK_GET_TX_CHAR:
        pFooChan->getTxChar = callback;
        pFooChan->getTxArg = callbackArg
        return (OK);
```

```
    case SIO_CALLBACK_PUT_RCV_CHAR:
        pFooChan->putRcvChar = callback;
        pFooChan->putRcvArg = callbackArg
        return (OK);
    default:
        return (ENOSYS);
    }
}
```

**通道 I/O 功能控制：xxIoctl( )**

I/O 控制用于实现对设备的特殊控制功能，函数 xxIoctl( )标准形式为：

```
static int xxIoctl (
    SIO_CHAN * pSioChan,
    int command,
    void * arg
);
```

用户对 I/O 控制调用的实现过程是：ioctl( ) → ttyIoctl( ) → xxIoctl( )。

入参 command 为 ioctl 中指定的 I/O 控制命令。如果 xxIoctl( )不支持，则返回给 ttyIoctl( )结果 ENOSYS，则 ttyIoctl( )调用 tyIoctl( )。如表 5-25 所示是在 sioLib.h 中定义的标准串口 I/O 命令列表。

表 5-25　标准串口 I/O 命令

| 命令定义 | 意　　义 |
| --- | --- |
| SIO_BAUD_SET | 设置波特率 |
| SIO_BAUD_GET | 读取波特率 |
| SIO_MODE_SET | 设置工作模式：<br>中断 SIO_MODE_INT 或查询 SIO_MODE_POLL |
| SIO_MODE_GET | 读取当前工作模式 |
| SIO_AVAIL_MODES_GET | 返回允许的工作模式 |
| SIO_HW_OPTS_SET | 设置硬件选项（POSIX） |
| SIO_HW_OPTS_GET | 读取硬件选项（POSIX） |

表中没有 FIOSELECT 和 FIOUNSELECT 命令，因为对 select 的支持由 tyLib 实现。

**硬件选项控制**

在标准串口 I/O 命令中，最后两个命令 SIO_HW_OPTS_SET 和 SIO_HW_OPTS_GET 用于设置和读取硬件选项。

硬件选项控制串口通信格式：数据位，停止位，奇偶校验，Modem 控制信号的使用等。在 sioLib.h 中，这些选项定义如表 5-26 所示。

表 5-26　硬件选项控制

| 选项定义 | 意　义 |
| --- | --- |
| CLOCAL | 忽略 MODEM 控制信号 |
| CREAD | 启动接收器 |
| CSIZE | 指定数据位（5～8 位）：CS5，CS6，CS7，CS8 |
| HUPCL | 最后关闭时挂断 MODEM 连接 |
| STOPB | 被设置时指定 2 位停止位，否则 1 位停止位 |
| PARENB | 被设置时启用奇偶校验，否则不进行奇偶校验 |
| PARODD | 被设置时使用奇校验，否则使用偶校验<br>（当 PARENB 被设置时有意义） |

上述有些选项不一定被 SCC 支持，这时 xxDrv 返回 ENOSYS。通常高层程序设置了硬件选项后应该再读取当前的硬件选项，来判断所做的设置是否实现，如：

```
int options;
...
if (ioctl (fd, SIO_HW_OPTS_SET, PARENB | PARODD) == OK)
{
    /* Setting hardware options supported */
    ioctl (fd, SIO_HW_OPTS_GET, &options);
    if (options & PARENB)
        { /* parity supported */ }
    if (options & PARODD)
        { /* odd parity supported */ }
}
else
{
    /* driver does not support setting hardware options */
}
```

默认情况下的设置为：8/1/N（8 数据位/1 停止位/无校验）。

用位操作表示即：(CLOCAL | CREAD | CS8) & ~(HUPCL | STOPB | PARENB)。

**实例：xxIoctl( )**

```
static int fooIoctl
(
```

```
    SIO_CHAN * pChan,
    int command,
    void * arg
)
{
    FOO_CHAN * pFooChan = (SIO_CHAN *)pChan;
    switch (command)
    {
    case SIO_BAUD_SET:
        if (arg < FOO_MIN_BAUDRATE || arg > FOO_MAX_BAUDRATE)
            return (EIO);
        ...
        pFooChan->baudrate = baudrate;
        return (OK);
    case SIO_BAUD_GET:
        *(int *)arg = pFooChan->baudrate;
        return (OK);
    case SIO_MODE_SET:
        return (fooModeSet (pFooChan, (uint_t)arg) == OK ? OK : EIO);
    case SIO_MODE_GET:
        *(int *)arg = pFooChan->mode;
        return (OK);
    case SIO_AVAIL_MODES_GET:
        * (int *)arg = SIO_MODE_INT | SIO_MODE_POLL;
        return (OK);
    case SIO_HW_OPTS_SET:
    case SIO_HW_OPTS_GET:
        return (ENOSYS);
    default:
        return (ENOSYS);
    }
}

Ioctl Support Routine

static int fooModeSet
(
    FOO_CHAN *   pFooChan,
```

```
    uint_t    newMode
)
{
    switch (newMode)
    {
    case SIO_MODE_INT:
        /* fail request for interrupt mode if ISRs
        are not yet installed */
        if (pFooChan->isrInstalled == FALSE)
            return (EIO);
        *pFooChan->cr |= FOO_INT_ENABLE;
        break;
    case SIO_MODE_POLL:
        *pFooChan->cr &= ~FOO_INT_ENABLE;
        break;
    default:
        return (EIO);
    }
    pFooChan->mode = newMode;
    return (OK);
}
```

- **启动新一轮发送：xxTxStartup( )**

函数 xxTxStartup( ) 负责启动新一轮发送过程，由高层程序 tyWrite 调用如图 5-25 所示。

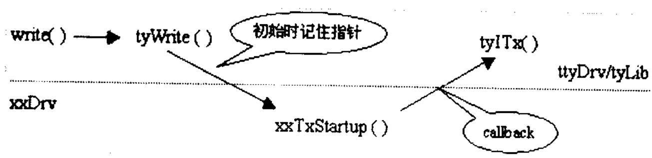



图 5-25 xxTxStartup

该函数是 SIO_DRV_FUNCS 中定义的 5 个接口函数中最复杂的一个。因为该函数和 SCC 发送特性有关。许多 SCC 在发送完一批数据后，如果 ISR 没有在发送中断中写入新数据，SCC 将挂起发送中断，这样需要驱动程序向 SCC 写入新的数据才能启动新的发送。这一过程因 SCC 差异而导致软件设计变化较大。软件需要知道当前的发送状态（发送中还是已经挂起），如果 SCC 正在发送，则向其写入数据可能引起数据丢失。

VxWorks 提供的文档称 ttyDrv 判断当前 SCC 是否处于一个进行的发送过程中，如果否，则调用 xxTxStartup( )启动新一轮发送。不管 ttyDrv 根据 xxDrv 所提供的接口能否知道

SCC是否处于发送中，这样的设计对于软件模块功能的封装都是不利的。

**实例：xxTxStartup( )**

```
static int fooTxStartup (SIO_CHAN * pSioChan)
{
    FOO_CHAN * pFooChan = (FOO_CHAN *)pSioChan;
    char outChar;
    if ((*pFooChan->getTxChar)(pFooChan->getTxArg, &outchar) != ERROR)
    {
        *pFooChan->dr = outChar;
        /* Turn on transmitter interrupts, if needed */
        *pFooChan->cr |= TX_INT_ENABLE;
    }
    return (OK);
}
```

- **查询输入 xxPollInput( )**
- **查询输出 xxPollOutput( )**

查询输入/输出在5.9.4节中介绍。

### 3. 中断处理

中断处理实际上和下列因素相关：

- 处理器体系特点 —— 中断向量；
- 中断控制器的使用；
- SCC中断特性 —— 提供的中断类型（接收/发送/状态），中断应答。

这些因素在设计过程中都必须考虑到。但是我们这里的讲述包括程序实例将忽略和处理器等其他硬件特性相关的细节，而专注于xxDrv对下面两种SCC中断的处理。

（1）接收中断 —— 通道收到数据。该中断发出时可能通道的硬件缓冲区有多个字节数据等待ISR读取；

（2）发送中断 —— 通道硬件发送缓冲区空。

对这两种类型的中断处理的介绍也是一般性的，具体在编写驱动时需要参考SCC的data sheet。

串口ISR和其他任何ISR一样，都需要遵循一般ISR的约束，比如不进行阻塞调用等。

**接收中断处理 xxIntRd( )**

SCC驱动xxDrv不设数据缓冲区，因此接收中断ISR从SCC的硬件缓冲区读入数据，通过调用回调函数将数据写到上层程序（tyLib）的缓冲区如图5-26所示。

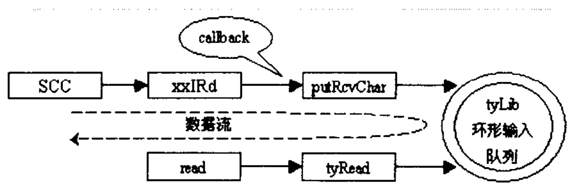



图 5-26 xxIntRd

**实例：xxIntRd( )**

```
void fooIntRcv (FOO_CHAN * pFooChan)
{
    char inChar;
    inChar = *pFooChan->dr;
    /* acknowledge interrupt if needed */
    *pFooChan->cr = FOO_RESET_INT
    (*pFooChan->putRcvChar)(pFooChan->putRcvArg, inchar);
}
```

**发送中断处理 xxIntTx( )**

通过回调函数从高层缓冲区获得要发送的数据，将数据输出到 SCC 对应的端口。如果高层缓冲区没有数据需要发送，可能需要复位发送中断(此后不再有发送中断，直到 xxDRv 将新的数据输出到设备启动发送中断)，如图 5-27 所示。

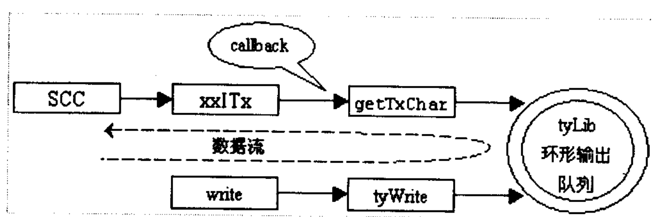



图 5-27 xxIntTx

**实例： xxIntTx( )**

```
void fooIntTx (FOO_CHAN * pFooChan)
{
    char outChar;
    if ((*pFooChan->getTxChar)(pFooChan->getTxArg, &outchar) != ERROR)
        *pFooChan->dr = outChar;
    else
        /* Reset interrupt, if needed */
        *pFooChan->cr = FOO_RESET_TX;
    /* acknowledge interrupt if needed */
```

```
    *pFooChan->cr = FOO_RESET_INT;
}
```

## 5.9.4 查询支持

查询主要为了支持调试。前面讲述的 SCC 基本工作在中断方式，在这种方式下，可以进行任务级调试。任务级调试是在任务运行环境工作正常的情况下，通过串口 I/O 输出诊断信息给主机，来帮助开发人员定位任务中出现的错误。

但是，任务运行环境本身工作不正常，即出现了系统级故障，如中断系统故障，这时需要进行更底层的调试——系统级调试。

如图 5-28 所示的内核前初始化在 sysHwInit( )被调用时完成，SCC 被复位，禁止硬件中断，然后 Target Agent 就可以通过 xxDrv 的查询输入输出函数实现串口 I/O。

查询支持属于 SCC 驱动的可选项。任务运行只需要 SCC 的中断方式。因此，SCC 驱动的实现可以包括：

- 只支持中断方式（SIO_MODE_INT），支持用户任务的 open/read/write，可以进行任务级调试；
- 支持中断方式和查询方式（SIO_MODE_POLL）。可以进行系统级调试，不动态切换；
- 支持中断方式和查询方式，可以通过 I/O 控制命令进行动态切换。

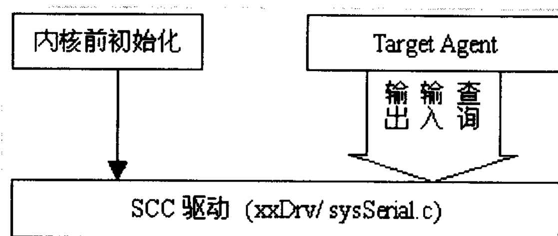



图 5-28 串口的系统级调试支持

I/O 控制函数 xxIoctl( )必须实现和 SCC 工作模式相关的命令：SIO_MODE_SET，SIO_MODE_GET，SIO_AVAIL_MODES_GET，以体现 xxDrv 对两种工作模式的支持。比如，如果不支持查询模式，则 xxIoctl( )应该有如下语句：

```
...
switch (command)
{
    ...
    case SIO_MODE_SET:
        if (arg== SIO_MODE_POLL) return EIO;  /*不支持查询，返回错误*/
```

```
        else ...
    case SIO_MODE_GET:
        *(int *)arg = SIO_MODE_INT;
        return (OK);
    case SIO_AVAIL_MODES_GET:
        * (int *)arg = SIO_MODE_INT;
        return (OK);
    ...
}
...
```

实现查询支持需要进行以下工作：

- 编写查询输入函数 **xxPollInput**；
- 编写查询输出函数 **xxPollOutput**。

如果不支持，仍然需要编写上述函数，因为它们属于 SIO_DRV_FUNCS 中定义的供高层访问的接口。此时这两个函数只需要一条语句：return ERROR。

## 1. 查询输入

函数用于在 SCC 查询模式下查询串口并读入一字节数据。定义为：

```
int xxPollInput ( SIO_CHAN * pSioChan, char * pChar );
```

函数至少应实现两种返回结果：当 SCC 通道有数据时读入数据到 pChar 并返回 OK；当通道中没有数据时返回 EAGAIN。还可能的实现包括：当通道工作在中断模式时返回 ENOSYS，当设备出错时返回 EIO 等。

**实例：xxPollInput**

```
static int fooPollInput (
    SIO_CHAN * pSioChan,    /* pointer to channel */
    char * pInputChar       /* pointer to char */
)
{
    FOO_CHAN *pFooChan = (FOO_CHAN *) pSioChan;
    if ((*pFooChan->sr & FOO_RX_AVAIL) == 0)
        return (EAGAIN);   /* no char available */

    /* char available */
    *pInputChar = pFooChan->dr;
    return (OK);
```

```
}
```

## 2. 查询输出

函数用于在 SCC 查询模式下查询串口并输出一字节数据。函数定义为：

```
int xxPollOutput ( SIO_CHAN * pSioChan, char outChar );
```

函数至少应实现两种返回结果：当 SCC 通道可以接收数据时将 outChar 输出到通道并返回 OK；当通道缓冲区满时返回 EAGAIN，outChar 没有被输出。还可能的实现包括：当通道工作在中断模式时返回 ENOSYS，当设备出错时返回 EIO 等。

**实例：xxPollOutput**

```
static int fooPollOutput (
    SIO_CHAN * pSioChan,          /* pointer to channel */
    char  outchar                 /* char to send */
)
{
    FOO_CHAN * pFooChan = (FOO_CHAN *)pSioChan;
    if ((*pFooChan->sr & FOO_TX_READY) == 0)
        return (EAGAIN); /* can't send */
    /* OK to send char */
    *pFooChan->dr = outChar;
    return (OK);
}
```

# 第 6 章　文件系统

## 6.1　文件系统概述

文件系统建立在块设备驱动之上，块设备驱动提供对设备上的块进行边界划分和读写；文件系统在此基础上建立文件结构提供应用程序访问。提供的块设备驱动有：

- ramDrv 虚拟 RAM 盘驱动；
- scsiLib SCSI 块设备驱动；
- tffsDrv 可选的 TrueFFS；
- ataDrv ATA/IDE 块设备驱动；
- cbioLib CBIO 驱动；
- 第三方块设备驱动。

VxWorks 支持的文件系统如表 6-1 所示：

**表 6-1　VxWorks 支持的文件系统**

| | |
|---|---|
| dosFs | 兼容 MS-DOS 的文件系统，但是考虑了实时应用需求 |
| rawFS | 提供将整个磁盘当成一个大文件的访问方式 |
| tapeFs | 磁带文件系统 |
| cdromFs | ISO-9660 标准格式的 CD-ROM 文件系统 |
| TSFS | 目标服务器的文件系统，通过调试代理和目标服务器之间的连接访问 |

从 VxWorks5.5 / Tornado 2.2 开始，文件系统升级到 dosFs 2.0 版本，引入了 CBIO 概念，文件系统建立在 CBIO 上。下面在介绍 dosFs 和 rawFs 之前，先介绍 CBIO。

## 6.2　CBIO

CBIO 称为缓冲块设备 I/O 库（Cached Block I/O library），在块设备以及文件系统中是个非常重要的概念。对 CBIO 可以从几方面了解：

- CBIO 为上层文件系统提供块设备的标准缓冲接口(CBIO API)。该接口包括 CBIO 设备初始化，以及文件系统在内存和 CBIO 设备缓冲区之间传递块和相关控制功能的函数；

- CBIO 封装的具有 CBIO 接口的虚拟设备，在 CBIO 设备下面是实际的块设备 BLK_DEV。在上层文件系统看来，就是一个数据结构 CBIO_DEV_ID，在其上可以进行 CBIO API 中的操作。在创建 CBIO 设备时得到一个 CBIO_DEV_ID，随后的所有 CBIO API 调用均根据该 CBIO_DEV_ID 标识 CBIO 设备；
- CBIO 家族包括许多其他库：基本的 cbioLib 本身、dcacheCbio、dpartCbio 等；
- CBIO 使文件系统在一定程度上忽略块设备 BLK_DEV 细节，并提高 I/O 效率。

如果使用文件系统，则需要使用 CBIO 库支持。

必须定义的库：

INCLUDE_CBIO 基本 CBIO 库 cbioLib。

根据需要选用的库：

- INCLUDE_DISK_CACHE 库 dcacheCbio；
- INCLUDE_DISK_PART 库 dpartCbio 和 usrFdiskPartLib。

## 6.2.1 基本 CBIO

cbioLib 通过封装块设备 BLK_DEV 得到的 CBIO 设备主要为了实现 CBIO API 接口，提供的缓存机制是最简单的。在 dosFs2.2 中，创建 dosFs 设备时将在内部调用 CBIO 的块设备封装接口。

### 创建 CBIO 设备

和许多其他库不同，cbioLib 库不需要在系统初始化阶段初始化，其初始化函数 cbioLibInit( )在第一次创建 CBIO 设备时调用，并且可以多次调用。

创建 CBIO 设备通过封装一个有效的块设备 BLK_DEV 得到，函数定义为：

```
#include "cbioLib.h"
CBIO_DEV_ID cbioWrapBlkDev ( BLK_DEV * pDevice );
```

pDevice 指向的块设备已经通过创建块设备得到。如果函数成功，则返回一个有效的 CBIO_DEV_ID；如果 pDevice 无效，或者其他错误，函数返回 NULL。

如果传递的入参本身是个有效的 CBIO_DEV_ID，则函数简单地返回本身，不做任何修改。允许这种情况是因为 CBIO 设备允许“级连”，比如下面要讲到的为 CBIO 创建缓存 dcacheDevCreate( )，其入参是 CBIO_DEV_ID，返回仍然是该 CBIO_DEV_ID，但是附加了额外的特性。

cbioLib 另外有个函数 cbioDevCreate( )，从名称上看也是创建 CBIO 设备，但实际上只是被 cbioLib 封装在内部 BLK_DEV 或者创建具有 CBIO 接口的 RAM 虚拟盘时调用（见“<install-dir>\target\src\usr\ramDiskCbio.c”），注意不要和以前讲的其他设备混淆。

## 6.2.2 CBIO 磁盘缓存

基本的 cbioLib 只实现了最简单的缓存，VxWorks 通过库 dcacheLib 在基本 CBIO 设备的基础上实现了更高效的缓存，尤其适合于磁盘这样的旋转媒介。该库实现的另外一个功能就是可移动磁盘被移除检测和处理。dcacheLib 支持同时最多为 16 个 CBIO 设备缓存。

引用库 dcacheLib 需要定义宏 INCLUDE_DISK_CACHE。

### 1. dcacheLib 的磁盘缓存算法

dcacheLib 使用的磁盘缓存算法和操作系统存储管理中的 LRU（Least Recently Used）页面替换算法相似：dcacheLib 维护一个按照 MRU（Most Recently Used）排序的磁盘缓存块链表，也就是说每当一个块被使用时，将其排到链表头部，这样当需要丢弃一块磁盘缓存块以装入新的磁盘块时，将排在链表尾部的块替换掉，该块就是最久未使用的块（LRU 块）。

dcacheLib 还通过一种预读取机制提高系统的磁盘 I/O 效率。在常规的按照磁盘扇区大小（512 字节）设立的缓存块链表之外，dcacheLib 另外留出一个更大的“留出缓存”，通常为常规缓存块链表所有节点大小的 1/4 至 64KB。当每次从磁盘读入新的块时，dcacheLib 读磁盘上该块后面的几块放入留出缓存。留出缓存会带来一定的额外开销(磁盘传输时间)，但是相比之下，访问磁盘的开销主要是另外两项：寻道时间+旋转延迟。留出缓存通过增加一定的磁盘传输时间来减少访问磁盘的次数，从而减少总计寻道时间和旋转延迟，以此提高效率。不过，留出缓存属于可调参数之一，以适应不同应用的特点。

“脏块”刷新机制：脏块是 dcacheLib 缓存块中被修改但是没有写入磁盘的缓存块。dcacheLib 内部启动一个低优先级刷新任务，用于周期性地将脏块刷新到底层磁盘驱动。对于可移除设备，该周期固定为 1 秒。另外两种使脏块被刷新的情况是：

- 用户程序显式调用刷新函数；
- 脏块总数到达最多允许的脏块数。

### 2. 参数调节

所有的可调参数都属于某种在多个利弊因素之间的权衡，dcacheLib 的默认设置适应多数应用。可以调节的参数包括：

- 最多允许的“脏块”：较大的脏块数可以减少写磁盘次数，但是太多脏块会降低缓存块命中率；
- 缓存旁路阈值：当对超出缓存旁路阈值的大块数据进行 I/O 时，dcacheLib 缓存将被旁路，这样高层（文件系统）缓冲区直接和底层块设备驱动缓冲区（BLD_DEV）交换数据，减少一个中间转换过程；
- 为留出缓存预读取的块数。该数值高时可以提高读磁盘效率，但是占用了太多缓

存空间，使缓存命中率下降；

- 缓存块刷新周期：定义刷新任务将脏块刷新到底层设备的周期。需要在数据可靠性和效率之间权衡。

调节上述参数调用函数 dcacheDevTune( )实现：

```
#include "dcacheCbio.h"
STATUS dcacheDevTune ( CBIO_DEV_ID dev, int dirtyMax,
   int bypassCount, int readAhead, int syncInterval
);
```

调用时设置需要调节的参数，除了 syncInterval 以外，都可以传递 0 表示不更改该项参数。传递 syncInterval 为 0 表示每次文件系统写磁盘时直接写到底层驱动。

调节参数后可以通过 dcacheLib 提供的函数 dcacheShow( )查看效率改进情况。主要的评价指标是缓存命中率。该函数将在标准输出上显示结果。

### 3. 可移动磁盘

对可移动磁盘，dcacheLib 需要考虑更多。要求实现：

（1）尽量减少数据丢失；

（2）保证上层系统的一致性。

对于第一点，要求尽量避免丢失脏块，dcacheLib 通过对可移动磁盘固定每隔 1 秒就刷新一次做到这一点。

要做到第二点，dcacheLib 需要及时知道磁盘的变化（被移除或者更改），并通知上层系统以维持系统的一致性。

磁盘被格式化时会被设置卷标和卷序列号，dcacheLib 根据他们计算一个校验值，以区别不同的磁盘。当磁盘空闲超过 2 秒时（假定更换磁盘需要超过 2 秒时间），dcacheLib 就会检查当前磁盘的校验值。当 dcacheLib 检测到可移动磁盘被移除时，通知上层文件系统卸载该卷，标记所有在该卷上打开的文件状态，或者当新的磁盘出现在驱动器中时挂装新的磁盘卷。

VxWorks 允许格式化卷时由程序指定卷标和卷序列号。从上述过程可以看出，更改这两项参数都相同的磁盘无法被检测出。因此格式化卷时不显示指定卷序列号，格式化程序会保持厂商设置的惟一序列号，或者当该序列号无效时根据格式化的时间设置卷序列号。

### 4. 创建缓存 CBIO 设备

创建缓存 CBIO 设备有两种方法：

- 由基本 CBIO 设备创建缓存 CBIO 设备；
- 直接在块设备 BLK_DEV 基础上创建缓存 CBIO 设备。

两种情况都调用 dcacheLib 提供的设备创建函数 dcacheDevCreate( )实现。对于第二种情况，dcacheDevCreate( )内部自动调用 cbioLib 提供的封装函数由 BLK_DEV 得到基本

CBIO 设备，如：

```
...
/*创建块设备*/
BLK_DEV* pBlkDev = xxxBlkDevCreate( ... );

/* 由块设备 BLK_DEV 创建缓存 CBIO 设备 */
CBIO_DEV_ID cBioDev = dcacheDevCreate(
        (CBIO_DEV_ID)pBlkDev, 0, 0, "Cached CBIO Device"
    );
...
```

如表 6-2 所示是 dcacheLib 提供的主要接口函数。

**表 6-2　dcacheLib 提供的主要接口函数**

| | |
|---|---|
| dcacheDevCreate( ) | 根据基本 CBIO 设备创建缓存 CBIO 设备<br>参数指定使用的内存位置、大小，以及说明设备的字符串。内存位置或大小可以指定为 0，函数采用默认处理 |
| dcacheDevDisable( ) | 禁用缓存功能（并不释放缓存空间）<br>相当于指定缓存旁路阈值为零，但是保存原值以再次启用缓存 |
| dcacheDevEnable( ) | 启用缓存功能 |
| dcacheDevTune( ) | 性能参数调节 |
| dcacheDevMemResize( ) | 重设用于磁盘缓存的内存<br>许多上层程序在底层媒介改变后调用此函数 |

## 6.2.3　CBIO 卷设备

本小节适用于磁盘这一类典型的块设备。

先来看一般化的磁盘使用过程：低级格式化、分区和格式化。然后讨论 VxWorks 的实现方式：CBIO 卷设备。

### 1．低级格式化与磁盘分区

每块硬盘在出厂时，硬盘生产商通常要进行低级格式化，这是硬盘使用的第一步。低级格式化也称为物理格式化，是针对一块物理上完整的空白磁盘划分出柱面和磁道，再将磁道划分为若干个扇区，每个扇区又划分出标识部分 ID、间隔区 GAP 和数据区 DATA 等。

低级格式化之后就可以在这一物理上完整的磁盘上建立分区，也称为卷（Volume）。分区时在磁盘的主引导扇区（0 柱面 0 磁道 1 扇区）建立磁盘分区表，如图 6-1 所示。

主引导扇区包含 4 条分区表记录，这 4 条记录表示的分区称为原始（Primary）分区或

者扩展（Extended）分区。扩展分区可以包含更多的逻辑分区，这些逻辑分区信息记录在主引导扇区之外。

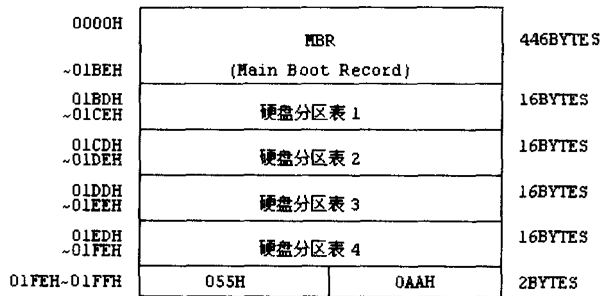



图 6-1　主引导扇区

4 条分区表记录每个表项，各字节含义如表 6-3 所示。

**表 6-3　分区表记录**

| 字　节 | 含　义 |
|---|---|
| 第 0 字节 | 是否为活动分区，是则为 80h，否则为 00h |
| 第 1 字节 | 该分区起始磁头号 |
| 第 2 字节 | 该分区起始扇区号（低 6 位）和起始柱面号（高 2 位） |
| 第 3 字节 | 该分区起始柱面号的低 8 位 |
| 第 4 字节 | 系统标志，00h 表该分区未使用，06h 表高版本 DOS 系统，05h 扩展 DOS 分区、65h 表 Netware 分区 |
| 第 5 字节 | 该分区结束磁头号 |
| 第 6 字节 | 该分区结束扇区号（低 6 位）和结束柱面号（高 2 位） |
| 第 7 字节 | 该分区结束柱面号的低 8 位 |
| 第 8～第 11 字节 | 相对扇区号、该分区起始的相对逻辑扇区号 |
| 第 12～第 15 字节 | 该分区所用扇区数 |

分区之后，就可以进行高级格式化，也称逻辑格式化，或者简称为格式化。高级格式化与所使用的上层文件系统有关，用于在卷上建立特定文件系统，如 FAT16，FAT32，Linux Ext2 等。

## 2．VxWorks 的实现：CBIO 卷设备

VxWorks 没有提供磁盘低级格式化功能。VxWorks 对磁盘分区的支持通过 CBIO 卷设备（库 dpartCbio）实现，引用宏定义为 INCLUDE_DISK_PART。

在存在磁盘分区的情况下，文件系统不是直接建立在 CBIO 设备基础上，而是建立在通过库 dpartCbio 封装的 CBIO 磁盘分区设备上。在这里，“分区”和卷的概念是一致的。分区容易让人和表示动作的“分区”联系起来，为了清晰起见，我们用“卷”的概念。这

样，我们已经涉及了3个概念，或者称设备。

- 基本CBIO设备：对块设备（BLK_DEV）的简单封装，为了对文件系统提供CBIO API接口；
- 缓存CBIO设备：为基本CBIO设备上实现复杂的缓存机制，提高系统磁盘I/O效率；
- CBIO卷设备：为文件系统提供封装的具有CBIO API接口的块设备，一个CBIO卷设备对应块设备上的一个逻辑卷。

实际上dpartCbio本身实现一般化CBIO卷设备，与具体的分区格式无关，它需要进行具体分区格式解码的库支持。当前FDISK分区是事实上的标准，库usrFdiskPartLib就实现了对FDISK分区格式的底层处理。因此，dpartCbio几乎总是和usrFdiskPartLib一起实现一个CBIO卷设备。

**注意：**

*缓存CBIO设备总是对整个磁盘进行缓存，而不是对磁盘上的每个卷进行处理。因此，在考虑磁盘分区的情况时，逻辑上CBIO卷设备必须建立在缓存CBIO设备之上。*

库dpartCbio提供了下面两个函数：

- 初始化所有CBIO卷设备；

```
#include "dpartCbio.h"
CBIO_DEV_ID dpartDevCreate (
   CBIO_DEV_ID subDev, int nPart, FUNCPTR pPartDecodeFunc);
```

该函数通过pPartDecodeFunc解码分区表，来初始化所有subDev上nPart个卷设备。该函数总是和下面的函数一起使用，得到具体每个分区对应的CBIO卷设备。

- 取得某个卷的CBIO卷设备；

```
#include "dpartCbio.h"
CBIO_DEV_ID dpartPartGet ( CBIO_DEV_ID masterHandle, int partNum );
```

该函数返回前面初始化的CBIO卷设备中对应的partNum分区的CBIO卷设备。

库usrFdiskPartLib实现了对FDISK分区格式的支持。风河公司以源文件形式提供该库，用户可以根据需要进行修改。

<install-dir>\target\src\usr\usrFdiskPartLib.c

<install-dir>\target\h\usrFdiskPartLib.h

库usrFdiskPartLib具有以下限制：

（1）只支持创建主分区表中最多4个原始分区记录（不创建扩展分区和逻辑分区）；

（2）由usrFdiskPartLib创建的分区不一定被DOS等主机操作系统兼容；

（3）可以读取DOS主机操作系统创建的分区，包括原始分区，扩展分区，逻辑分区。

如果希望磁盘在VxWorks和主机操作系统之间交换数据，则最好用主机操作系统的FDISK类工具创建分区。

- 创建 FDISK 分区表；

```
#include "usrFdiskPartLib.h"
STATUS usrFdiskPartCreate (
   CBIO_DEV_ID cDev, int nPart, int size1, int size2, int size3 );
```

该函数创建磁盘分区表，最多支持创建 4 个分区。参数 cDev 表示磁盘设备，通常是对应整个磁盘的 CBIO 设备，如缓存 CBIO 设备。参数 nPart 表示所要创建的分区数(1～4)，后面的参数分别表示从第二个卷开始的每个卷占磁盘空间的百分比，第一个卷使用剩下的空间。

- 读 FDISK 磁盘分区表；

```
#include "usrFdiskPartLib.h"
STATUS usrFdiskPartRead (
   CBIO_DEV_ID cDev, PART_TABLE_ENTRY * pPartTab, int nPart );
```

该函数读取 cDev 的分区表记录并填写在分区表记录数组 pPartTab 中，参数 nPart 表示数组维数。参数 cDev 通常是对应整个磁盘的 CBIO 设备。

每条分区表记录包含了分区在磁盘中起始位置、块数以及其他标志信息。通常由前面介绍的 dpartDevCreate( )调用此函数，以确定每个 CBIO 卷在磁盘中的位置。

- 显示分区信息。

```
#include "usrFdiskPartLib.h"
STATUS usrFdiskPartShow ( CBIO_DEV_ID cbio, block_t extPartOffset,
          block_t currentOffset, int extPartLevel );
```

函数将在控制台显示 cDev 的所有分区信息。后面 3 个参数必须传递 0。显示的信息包括分区表记录中所有字段，如：

```
Master Boot Record - Partition Table
--------------------------------------
 Partition Entry number 00    Partition Entry offset 0x1be
 Status field = 0x80         Primary (bootable) Partition
 Type 0x04: MSDOS 16-bit FAT <32M Partition
 Partition start LCHS: Cylinder 0000, Head 001, Sector 01
 Partition end  LCHS: Cylinder 0255, Head 127, Sector 01
 Sectors offset from MBR partition 0x00000001
 Number of sectors in partition 0x0000ffff
 Sectors offset from start of disk 0x00000001
 ...（其他分区记录）
```

函数 usrFdiskPartShow 定义了大量的 static 文本来支持显示上述信息，如果不需要该函数，强烈建议去掉对 INCLUDE_PART_SHOW 的宏定义，然后重新编译源文件，这样可以显著节省代码空间。默认情况下该宏被定义。

在使用上述 API 编程时，根据是否要创建分区有不同的处理。下面这段代码将在磁盘上创建 3 个逻辑分区，大小分别为：3%，50%，45%。

```
...
/* 创建缓存 CBIO 设备*/
if((cbio = dcacheDevCreate(blkDevId, NULL, dcacheSize, "/sd0")) == NULL )
    return ERROR ;

/* 创建 FDISK 分区 */
if((usrFdiskPartCreate (cbio,3,50,45,0)) == ERROR)  return ERROR;

/* 创建表示所有分区的 CBIO 卷设备 cbio1 */
if((cbio1 = dpartDevCreate( cbio, 3, usrFdiskPartRead )) == NULL)
    return ERROR;

/* 得到 cbio1 第 1 卷的表示 */
if((cbio1Vol1 = dpartPartGet(cbio1,0)) == NULL)
    return ERROR;
...
```

代码中的 blkDevId 表示底层块设备，例如 ataDevCreate( )返回的指针。最后得到的 cbio1Vol1 表示磁盘上创建的第一个卷。通常该卷还需要进行文件系统创建和高级格式化。

如果磁盘已经进行了分区，则上述代码中创建 FDISK 分区部分（即斜体部分）可以略去。

### 6.2.4 ioctl

CBIO 支持下列 I/O 控制命令：

表 6-4　CBIO 支持的 I/O 控制命令

| | |
|---|---|
| CBIO_RESET | 复位 CBIO 设备 |
| CBIO_STATUS_CHK | 检查设备状态 |
| CBIO_DEVICE_LOCK | 防止磁盘移除 |
| CBIO_DEVICE_UNLOCK | 允许磁盘移除 |

续表

| | |
|---|---|
| CBIO_DEVICE_EJECT | 卸载（Unmount）设备 |
| CBIO_CACHE_FLUSH | 刷新脏块 |
| CBIO_CACHE_INVAL | 刷新并置缓存块为无效状态 |
| CBIO_CACHE_NEWBLK | 分配新块 |

# 6.3 dosFs 文件系统

VxWorks 的 dosFs 文件系统兼容 DOS/Windows/OS2 上广泛使用的 DOS 文件系统。以 VxWorks 5.5 为界，dosFs 在提供的功能和组件封装上有了较大变化。我们用“dosFs (2)”表示从 VxWorks 5.5（含）开始的 dosFs 文件系统；用“dosFs (1)”表示 VxWorks 5.5 以前的 dosFs 文件系统；用“dosFs”表示与版本无关的 dosFs 特性。主要讲述的是 dosFs (2)：

（1）兼容 MS-DOS；

（2）支持 NFS。

## 6.3.1 卷结构

dosFs 文件系统建立在磁盘“卷”（或者称为分区）的基础上。6.2.3 节中 CBIO 卷设备为 dosFs 封装了这样一个卷，该卷在磁盘上的物理位置，是一个从 0 开始顺序寻址扇区的简单结构。图 6-2 表示了 dosFs 在卷上建立的 dosFs 文件分配表 FAT 和文件目录结构。

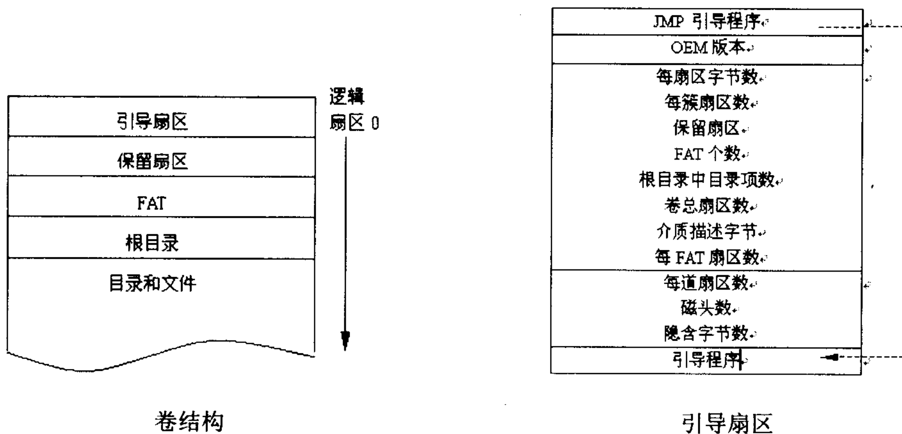



图 6-2 dosFs 卷结构

卷上的第一个扇区称为引导扇区，引导扇区记录了该卷的全部介质信息。引导扇区后为保留扇区，大小由引导扇区中保留扇区数指定。引导扇区中的保留扇区数包括引导扇区

本身，如果保留扇区数为 1，则实际上保留扇区长度等于 0。

引导扇区信息在磁盘初始化时写入：

（1）在主机操作系统上执行 MS-DOS 格式化（format.exe）；

（2）在目标系统中以命令 FIODISKINIT 调用 ioctl( )；

（3）目标系统格式化 dosFsVolFormat。

VxWorks 的 dosFs 识别主机环境下以 MS-DOS 格式化的磁盘。

### 1. 簇

簇通常是比磁盘扇区更大的单位，MS-DOS/dosFs 文件系统以簇为最小单位将磁盘空间分配给文件。扇区取决于磁盘物理属性，而簇是由 MS-DOS/dosFs 记录在引导扇区中的每簇扇区数决定的。通常软盘上 1 簇相当于 2 扇区，硬盘上 1 簇对应更多扇区。

### 2. 文件分配表

每个 MS-DOS/dosFs 卷对应一个或多个文件分配表（FAT），FAT 记录磁盘上每个簇的使用情况，也可以将 FAT 看成表示卷上各个文件所分配的簇的单链表：如果该簇分配给某文件，则记录下一个分配给该文件的簇编号（若是最后一个簇，则记录-1）；若簇未被分配，则记录 0。VxWorks 支持为文件分配连续的簇，如图 6-3 所示。

| cluster | FAT |
|---|---|
| 0 | |
| 1 | |
| 2 | 500 |
| 3 | 0 |
| ⋮ | ⋮ |
| 300 | 500 |
| ⋮ | ⋮ |
| 500 | -1 |

图 6-3　FAT

由图 6-3 可以看出，1 簇对应的扇区数目决定了在某大小的卷上 FAT 所占用的空间。增加簇对应的扇区数可以减少 FAT 大小，但是使磁盘利用率下降。

根据 FAT 中每条记录项长度，FAT 有 3 种人们熟知的格式：FAT12，FAT16 和 FAT32，如表 6-5 所示。

表 6-5　FAT 的格式

| 格　式 | 兼容性 | 说　明 |
|---|---|---|
| FAT12 | dosFs dosFs2 | 一般每簇 2 扇区，每条记录 12 位（1 卷可以包含 4085 簇，注意不是 $2^{12}$） |

续表

| 格　式 | 兼容性 | 说　明 |
|---|---|---|
| FAT16 | dosFs dosFs(2) | 每条记录 16 位 ，可以支持 2～8GB 磁盘 |
| FAT32 | 仅 dosFs(2) | 每条记录 32 位，一般 2GB 以上磁盘使用此格式 |

### 3. 根目录

在 FAT 之后一系列连续的扇区被用作根目录。根目录和子目录一样其中包含子目录和文件。根目录具有固定位置和大小，一般包含 512 项记录，由格式化程序确定。

根目录另外一个特殊之处在于记录卷标。每个 dosFs 卷可以拥有一个 11 个字符构成的卷标，该卷标在根目录中占有一个记录项。因此，根目录中可以包含的子目录/文件记录项要比根目录容量小 1；或者所有根目录空间都用于子目录/文件记录项，这样试图设置卷标将会出错。

### 4. 子目录/文件

区别于根目录，子目录/文件以簇为单位分配空间，并且不要求连续，因此子目录/文件大小只受可用卷空间限制。

子目录和文件的区别在于子目录包含记录项，每个记录项表示归属该子目录的一个下级子目录或者文件，而文件则是应用程序记录的一些数据。记录项内容包括：文件名、扩展名、生成或最新修改时间、生成或最新修改日期、开始簇、文件大小等。FAT12/FAT16/FAT32 对记录项定义各有不同，dosFs(1)只支持 FAT12/FAT16，dosFs(2)支持 FAT12/FAT16/FAT32。

每个子目录都包含两个特殊的记录项：•和••分别表示目录本身和父目录。根目录中没有•和••记录项。

## 6.3.2 使用 dosFs

使用 dosFs 包括对从内核的定制（包含必要的组件和初始化）到创建设备，到应用程序文件/目录操作的过程。典型情况是：

（1）定义底层块设备驱动和 dosFs 组件；
（2）创建底层块设备；
（3）创建基本 CBIO 设备；
（4）创建缓存 CBIO 设备；
（5）创建 CBIO 卷设备（创建分区）；
（6）创建 dosFs 设备；
（7）高级格式化 dosFs 设备；
（8）挂装 dosFs 设备（文件/目录操作）。

其中，步骤（3）和步骤（4）可以合并为一步，如果磁盘只有一个卷，步骤5也可以省略，如果卷已经被格式化（如用 MS-DOS 的 format.exe），则步骤（7）省略。如图 6-4 所示步骤。

### 1．定义底层块设备驱动和 dosFs 组件

dosFs 需要底层的块设备支持，常见的有 VxWorks 提供的 SCSI 和 ATA/IDE 块设备驱动，以及 RAM 盘驱动，通常选择下面两者之一：

- INCLUDE_SCSI；
- INCLUDE_ATA。

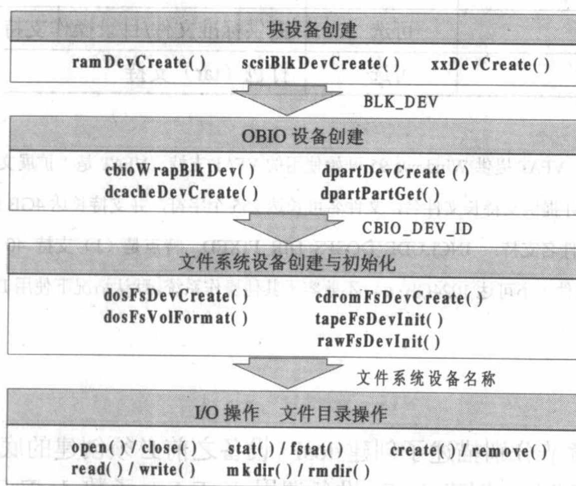



图 6-4　使用 dosFs 设备的完整步骤

### 2．dosFs 组件构成与初始化

dosFs (2)包含一系列必须或者相关的组件（“表 6-6 dosFs(2)组件构成”）。当定义相应的宏时，如表 6-6 所示的初始化函数会被自动调用。

表 6-6　dosFs(2)组件构成

| 宏 定 义 | 类　型 | 说　明 | 初 始 化 |
|---|---|---|---|
| INCLUDE_DOSFS_MAIN | 必选 | dosFsLib (2)<br>DOS 文件系统主组件，需要许多下面其他 DOS 组件支持 | dosFsLibInit( ) |
| INCLUDE_DOSFS_FAT | 必选 | FAT12/16/32 处理 | dosFsFatInit( ) |
| INCLUDE_CBIO | 必选 | 基本 CBIO API | cbioLibInit( ) |
| INCLUDE_DOSFS_DIR_VFAT | 2选1（注） | VFAT 长文件名支持 | dosVDirLibInit( ) |
| INCLUDE_DOSFS_DIR_FIXED | | 标准“8.3”和 VxWorks 长文件名支持 | dosDirOldLibInit( ) |

续表

| 宏 定 义 | 类 型 | 说 明 | 初 始 化 |
|---|---|---|---|
| INCLUDE_DISK_CACHE | 可选 | CBIO 磁盘缓存 | |
| INCLUDE_DISK_PART | 可选 | CBIO 卷封装 | |
| INCLUDE_RAM_DISK | 可选 | ramDiskCbio | |
| INCLUDE_DOSFS | 可选 | 提供 VxWorks5.5 以前 dosFs 兼容性支持（usrDosFsOld.c） | DosFsInit( ) |
| INCLUDE_DOSFS_FMT | 可选 | 提供 dosFs2 磁盘格式化功能 | dosFsFmtLibInit( ) |
| INCLUDE_DOSFS_CHKDSK | 可选 | 提供文件系统完整性检查 | dosChkLibInit( ) |
| INCLUDE_DISK_UTIL | 可选 | 提供标准文件/目录操作支持 | |
| INCLUDE_TAR | 可选 | 打包（tar）支持 | |

注：

INCLUDE_DOSFS_DIR_VFAT 提供 Windows 95 开始使用的 VFAT 支持。VFAT 是“扩展文件分配表系统”的意思，它对 FAT16 文件系统进行扩展，并提供支持长文件名，文件名可长达 255 个字符，并支持长达 4GB 的文件大小。此外 VxWorks 自己定义了一种更简单的长文件名支持：INCLUDE_DOSFS_DIR_FIXED，特点是（1）支持 40 个字符的长文件名；（2）文件名区别大小写；（3）理论上文件大小可达 1024GB；（4）不兼容于其他操作系统。默认情况下使用 INCLUDE_DOSFS_DIR_VFAT。

### 3. 设备创建

前面一些相关章节分别描述了创建dosFs设备之前必须创建的底层设备(块设备、CBIO设备等)。在这些基础上，创建 dosFs 设备调用 dosFsLib 函数 dosFsDevCreate( )完成：

```
#include "dosFsLib.h"
STATUS dosFsDevCreate (
    char * pDevName, CBIO_DEV_ID cbio, u_int maxFiles, u_int autoChkLevel
);
```

参数 cbio 表示底层 CBIO 设备，如缓存 CBIO 设备或者 CBIO 卷设备。

参数 maxFiles 表示最多可以同时打开的文件数。系统需要为此预分配空间，因此该数值一般不要太大。超出 maxFiles 限制打开文件时将会出错。

参数 autoChkLevel 表示挂装 dosFs 设备时所作的卷结构完整性检查（CHKDSK）以及信息提示方式，默认（指定 0）时为 DOS_CHK_REPAIR|DOS_CHK_VERB_1，如果不想进行 CHKDSK，则指定 autoChkLevel 为 NONE。

参数 pDevName 表示创建的 dosFs 设备名称，该名称必须是惟一的。如果函数成功，则返回 OK，随后可以根据 pDevName 进行高级格式化和其他文件操作。出错时函数返回 ERROR。

MS-DOS 环境下 DOS 设备名称由一个字母和冒号（如“A:”）构成，在 VxWorks 下，可以通过 pDevName 指定更有意义的名称，如“DOS1:”或者 UNIX 风格的“/hda1”。

### 4. 高级格式化

高级格式化：（1）调用 dosFsLib 的 ioctl 命令 FIODISKINIT 以默认方式格式化磁盘卷；（2）使用可选工具库 dosFsFmtLib 进行格式化。如果设备上还没有创建 DOS 文件系统，则只能通过第 2 种方式格式化。

格式化函数 dosFsVolFormat 定义为：

```
#include "dosFsLib.h"
STATUS dosFsVolFormat ( void * device, int opt, FUNCPTR pPromptFunc );
```

一般情况下，参数 device 是一个指向 dosFsDevCreate( )创建 dosFs 设备时指定的设备名称的指针；或者是一个未创建 dosFs 设备的 CBIO 设备。在 dosFs(1)中 device 还可以指向一个 dosFsLib 卷描述符 DOS_VOL_DESC。dosFs(2)已经不再采用 DOS_VOL_DESC，惟一的区别一个卷的标志是创建时指定的 dosFs 设备名称，这和其他设备驱动（如管道设备）是一致的。

参数 opt 指定操作选项如表 6-7 所示：

表 6-7　参数 opt 指定操作选项

| 选项 opt | 含　义 |
| --- | --- |
| DOS_OPT_DEFAULT | 使用默认选项：如果当前卷的引导区相当完整，则使用当前参数，否则根据磁盘大小重新计算参数。可能保持卷标和序列号 |
| DOS_OPT_PRESERVE | 保持当前卷参数，即使卷参数可能有错误 |
| DOS_OPT_BLANK | 放弃当前卷参数，根据磁盘大小计算新的卷参数 |
| DOS_OPT_QUIET | 使用安静模式，格式化过程中不输出任何诊断信息 |
| DOS_OPT_FAT16 | 限制使用 FAT16 格式，即使磁盘大于 2GB（超出 2GB 的磁盘默认使用 FAT32 格式） |
| DOS_OPT_FAT32 | 限制使用 FAT32 格式，即使磁盘容量小于 2GB（FAT32 不能用于小于 512MB 的磁盘） |
| DOS_OPT_VXLONGNAMES | 以 VxWorks 长文件名系统格式化 |

第 3 个参数 pPromptFunc 指向一个函数，该函数用于交互地让用户在开始格式化前编辑可修改的卷参数，函数声明如下：

```
void formatPromptFunc( DOS_VOL_CONFIG *pConfig );
```

DOS_VOL_CONFIG 是一个包含卷参数的结构体，在 dosFsLib.h 中定义。

如果不需要交互修改卷参数，指定 pPromptFunc 为 NULL。

### 5. 示例代码

下面示例函数 usrPartDiskFsInit( )表示了上述各个步骤：创建 ATA 块设备、分区、创建 dosFs 设备、创建 dosFs 文件系统（高级格式化）。

函数成功时返回 OK，这时创建了两个 dosFs 设备，名称分别为“/hda1”和“/hda2”。

```
#define CACHE_SIZE 0x30000

STATUS usrPartDiskFsInit ( void * blkDevId )
{
    const char * devNames[] = { "/hda1", "/hda2" };
    BLK_DEV * pBlkDev ;
    CBIO_DEV_ID cbio, cbio1 ;

    /* 创建 ATA 块设备 */
    pBlkDev = ataDevCreate (0,0,0,0);

    /* 创建缓存 CBIO */
    if((cbio = dcacheDevCreate(pBlkDev, NULL, CACHE_SIZE, "/sd0")) == NULL )
        return ERROR ;

    /* 如果需要，在磁盘上创建两个分区 */
    if((usrFdiskPartCreate (cbio,2,55,0,0)) == ERROR)
        return ERROR;

    /* 创建表示所有分区的 CBIO 卷设备 cbio1 */
    if((cbio1 = dpartDevCreate( cbio, 2, usrFdiskPartRead )) == NULL)
        return ERROR;

    /* 创建第 1 个卷 dosFs 设备 允许同时打开 8 个文件 使用默认 CHKDSK 方式 */
    if(dosFsDevCreate( devNames[0], dpartPartGet(cbio1,0), 8, 0 ) == ERROR)
        return ERROR;

    /* 创建第两个卷 dosFs 设备 允许同时打开 4 个文件 不进行 CHKDSK */
   if(dosFsDevCreate( devNames[1], dpartPartGet(cbio1,2), 4, NONE ) == ERROR)
        return ERROR;

    /* 依次格式化两个卷 创建 dosFs 文件系统 */
```

```
    if(dosFsVolFormat (devNames[0], 2,0) == ERROR)     return ERROR;
    if(dosFsVolFormat (devNames[1], 2,0) == ERROR)     return ERROR;

    return OK;
}
```

在其他块设备上创建和使用 dosFs 的过程与此相似，主要差别在于块设备的创建。

## 6.3.3 挂装与卸载

dosFs 允许设备动态地挂装和卸载。“挂装”意味着确认设备使之进入可以进行文件/目录操作状态；“卸载”则是使设备和文件系统脱离，设置该设备上打开的文件为无效状态。

一般地，dosFs 设备创建后在第一次在该设备上调用 open( )或者 create( )时被挂装。如果在此之前进行卷格式化，也会使 dosFs 挂装设备。

### “就绪改变”机制

当使用可移动介质时，底层介质的变化可能导致文件系统和磁盘记录的不一致。保持一致性包括两方面的努力：对用户程序而言，及时关闭不再使用的文件来刷新缓存内容到磁盘，同时还节约了系统 I/O 资源；而 VxWorks 系统则使用“就绪改变”机制来通知文件系统底层介质的改变，使文件系统标志其上打开的 fd 为“过时”，程序对过时的 fd 进行读写会出错，但是可以关闭以释放系统资源。

VxWorks 块设备驱动 BLK_DEV 实现了一个标志：bd_readyChanged，当底层介质变化时，将标志置位。每次在 dosFs 设备上进行 I/O 时，dosFs 将检查底层设备 BLK_DEV 的就绪该标志来判断是否需要挂装设备。除了这种块设备驱动通知方式外，还有两个通知 dosFs“就绪改变”的方式：

（1）调用 ioctl( fd, FIODISKCHANGE, NULL )，fd 是某个在 dosFs 设备上打开的文件描述符；

（2）调用 cbioRdyChgdSet( )在 CBIO 层设置 bd_readyChanged 标志为 TRUE。

dosFs 还支持用户程序中显式卸载 dosFs 设备，这是一种更可取的卸载磁盘的方法，在磁盘被移走之前，在非 ISR 任务中调用 ioctl 命令 FIOUNMOUNT 实现。该命令使 dosFs 刷新 dosFs 设备上所有的缓存，所有未关闭的文件被标记为“过时”。

## 6.3.4 文件和目录

创建 dosFs 设备之后，就可以进行文件/目录操作。对文件而言，包括创建/打开新文件、读写文件，删除文件。对目录而言，包括新建/删除目录，读目录内容。不管文件还是目录，

创建或打开时都需要指定路径名称。VxWorks 和 MS-DOS 在路径名上有两个不同之处：

（1）MS-DOS 设备名总是单个字母加冒号开始；VxWorks 可以自由使用更有意义的长设备名；

（2）MS-DOS 使用只能“\”分隔目录和文件；VxWorks 路径中可以同时用“/”或者“\”分割目录和文件。

**注意：**

*虽然 VxWorks 允许使用正斜杠“/”和反斜杠“\”，但是建议统一使用“/”以避免一些不必要的错误。VxWorks 提供的部分程序在处理路径时只考虑了使用“/”。*

“目录”也是一种特殊的文件，由表示文件和子目录的目录项组成，目录项中有一字节表示文件/目录属性：

- DOS_ATTR_RDONLY　只读属性，文件只能以只读（O_RDONLY）方式打开；
- DOS_ATTR_HIDDEN　隐藏属性，VxWorks 忽略该标志；
- DOS_ATTR_SYSTEM　系统属性，VxWorks 忽略该标志；
- DOS_ATTR_VOL_LABEL　卷标，仅适用于根目录；
- DOS_ATTR_DIRECTORY　目录，表明该目录项对应一个子目录；
- DOS_ATTR_ARCHIVE　文档，当文件被创建或者修改后 VxWorks 将该标志置位，但不负责清除标志位。

上述属性通过 dosFs 的 ioctl( )函数进行控制。

文件 I/O 操作和目录的创建及删除通过基本的 I/O 调用（ioLib）实现，I/O 系统将把用户程序的基本 I/O 引导给 dosFs 驱动提供的基本 I/O 实现函数。这一过程是：

（1）在 dosFs 驱动初始化时，初始化函数 dosFsLibInit( )在系统驱动程序表中安装 dosFs 的 6 个基本 I/O 操作对应的驱动程序函数（参考图 5-6），并得到全局驱动程序号；

（2）dosFsDevCreate( )创建 dosFs 设备，并将包含设备名称和驱动程序号的“设备描述结构”加入到系统的设备链表中；

（3）用户程序通过 open( )打开 dosFs 设备上的文件时，文件名给出了 dosFs 设备名称，I/O 系统通过该名称找到对应的设备描述结构，并根据其中的驱动程序号找到 dosFs 驱动提供的基本 I/O 实现函数 dosFsOpen；

（4）dosFs 的基本 I/O 实现函数通过底层 CBIO 实现对设备的读写和控制。如图 6-5 所示。

上述过程在 5.7 节“I/O 系统内部结构”中有更深入的说明。对用户程序而言，可以不关心这么多，只需要创建了 dosFs 设备之后调用基本 I/O 函数 open/close/read /write/remove/ioctl 就行了。例如：

```
/* 创建目录 */
open ( dirName, O_RDWR | O_CREAT, FSTAT_DIR | DEFAULT_DIR_PERM );

/* 删除目录 */
remove ( dirName );
```

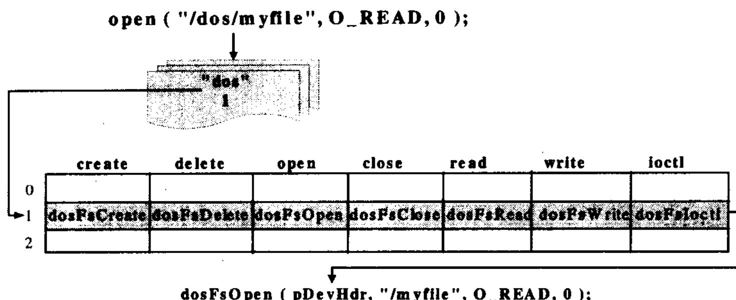



图 6-5　在 dosFs 设备上打开文件

由于目录具有特殊的内容，因此，VxWorks 通过一个目录操作库 dirLib 提供兼容 POSIX 标准的 API，用于读取/查找目录中包含的文件和子目录。dirLib 函数原型在 dirent.h 和 stat.h 中定义，如表 6-8 所示。

表 6-8　dirlib 函数

| 函　　数 | 说　　明 |
|---|---|
| opendir( ) | 打开一个目录。目录名用一个字符串指定，必须存在 |
| readdir( ) | 读取打开目录中当前记录，如果已达末尾，返回 NULL |
| rewinddir( ) | 复位打开的目录当前记录位置 |
| closedir( ) | 关闭目录 |

上述函数使用一个结构 struct DIR 表示打开的目录，DIR 中除了定义基本 open( )调用得到的文件描述符 fd 外，还包括一个目录 cookie 记录目录当前记录项位置。当目录刚刚被 opendir( )打开时，当前记录项为第 1 项；每次调用 readdir( )将读出当前记录项，然后将当前记录项后移 1 次；rewinddir( )复位当前记录项为打开时状态。

dirLib 还定义了一组兼容 POSIX 标准的 API 获取文件状态信息，如表 6-9 所示。

表 6-9　API 获取文件函数

| 函　　数 | 说　　明 |
|---|---|
| fstat( ) /fstatfs( ) | 根据文件描述符 fd 获取文件状态信息 |
| stat( )/ statfs( ) | 根据文件路径名获取文件状态信息 |
| utime( ) | 更新文件时间（非 POSIX 标准） |

实际上，dirLib 也是通过 dosFs 驱动提供的基本 I/O 调用函数完成的，只是 dirLib 为用户程序隐藏了特定 FAT 和目录格式的细节，使用户专注于文件操作。

下面这段示例代码统计指定目录下所有文件大小，函数 calcFileLen( )接受的入参为表示目录名的字符串。

```
#include "dirent.h"     /* 库 dirLib 的头文件 */
#include "sys/stat.h"

long calcFileLen( const char * dirName )
{
    DIR * pDir;
    char  pathName [MAX_FILENAME_LENGTH];
    char  fullName [MAX_FILENAME_LENGTH];
    long totalSize = 0;
    struct dirent * pDirEnt;
    struct stat      fileStat;
    if ( pDir = opendir(dirName) == NULL )  return 0;
    strcpy( pathName, dirName );
if (dirName[strlen(pathName)-1]!='/' )
        strcat( pathName, "/" );
    while (1)
    {
        errno = OK;
        pDirEnt = readdir (pDir);
        if( pDirEnt )
        {
            strcpy( fullName, pathName );
            strcat( fullName, pDirEnt-> d_name );
            if (stat(fileName, &fileStat)==OK && S_ISREG(fileStat.st_mode))
                totalSize += fileStat. st_size;
        }
        else
            break;
}
    closedir ( pDir );
    return totalSize;
}
```

VxWorks 另外通过源代码形式提供了一个库 usrFsLib 用于实现常用文件/目录操作，类似主机操作系统的 shell（如 MS-DOS 的 command.exe 和 UNIX 的 bsh），包括文件复制、删除、建立/删除目录、目录列表、磁盘检查等。详情参考：

<install-dir>\target\src\usr\usrFsLib.c

### 6.3.5 ioctl

- FIODISKINIT

```
fd = open ("/hda1", O_WRONLY);
status = ioctl (fd, FIODISKINIT, DOS_OPT_BLANK);
```

该函数内部调用 dosFsVolFormat( )进行磁盘格式化，为 ioctl( )指定的第 3 个参数传给 dosFsVolFormat( )作为格式化选项的位标志。以此方式格式化磁盘要求得到 dosFs 设备的 fd，因此只能在已经具有 dosFs 文件系统的磁盘上进行。如果磁盘未格式化，则只能通过直接调用 dosFsVolFormat( )格式化。

- FIODISKCHANGE

```
status = ioctl (fd, FIODISKCHANGE, 0);
```

通知 dosFs 底层介质发生改变。该调用允许在 ISR 中进行。因为介质改变，dosFs 不会刷新缓冲区。

- FIOUNMOUNT

```
status = ioctl (fd, FIOUNMOUNT, 0);
```

卸载磁盘，类似 dosFs(1)的函数 dosFsVolUnmount( )。该调用不能在 ISR 中发出。

- FIOGETNAME

```
status = ioctl (fd, FIOGETNAME, &nameBuf );
```

获得文件描述符 fd 对应的文件名称，并复制到 nameBuf。

- FIORENAME

```
fd = open( "oldname", O_RDONLY, 0 );
status = ioctl (fd, FIORENAME, "newname");
```

重命名文件或者目录。

- FIOMOVE

```
fd = open( "oldname", O_RDONLY, 0 );
status = ioctl (fd, FIOMOVE, "newname");
```

移动文件或者目录。

- FIOSEEK / FIOSEEK64

```
long    newOffset;
status = ioctl (fd, FIOSEEK, newOffset);
```

FIOSEEK 和 FIOSEEK64 用于设置文件偏移量。前者能表示 32 位偏移量，对多数应用而言已经足够。后者使用 64 位偏移量，仅仅在卷格式化时使用 VxWorks 长文件名结构才支持。

- FIOWHERE / FIOWHERE64

```
long    position;
position = ioctl (fd, FIOWHERE, 0);
```

获取当前偏移量，32 位或 64 位，表示下一个读/写时的位置。

- FIOFLUSH

```
status = ioctl (fd, FIOFLUSH, 0);
```

刷新缓冲区内容到磁盘。

- FIOSYNC

```
status = ioctl (fd, FIOSYNC, 0);
```

用于卷同步。参数 fd 所在卷的 CBIO 缓存都将刷新到磁盘，并相应更新文件分配表 FAT 记录。另一种较好的卷同步方式是关闭所有在卷上打开的文件，然后调用以命令 FIOUNMOUNT 调用 ioctl。关闭单个文件使该文件对应的 CBIO 缓存刷新到磁盘。

- FIOTRUNC / FIOTRUNC64

```
status = ioctl (fd, FIOTRUNC, newLength);
```

将文件 fd 截平。新长度 newLength 小于文件当前长度，截下来的磁盘空间（簇）被重新标记为空闲，同时更新文件所在目录的记录项。

- FIONREAD / FIONREAD64

```
unsigned long unreadCount;
status = ioctl (fd, FIONREAD, &unreadCount);
```

获取文件 fd 剩下未读的字节数。

- FIONFREE / FIONFREE64

```
unsigned long freeCount;
status = ioctl (fd, FIONFREE, &freeCount);
```

获取空闲字节数。

- FIOMKDIR

```
status = ioctl (fd, FIOMKDIR, dirName );
```

创建新目录 dirName。

- FIORMDIR

```
status = ioctl (fd, FIORMDIR, dirName );
```

删除目录 dirName。

- FIOLABELGET

```
char        labelBuffer [DOS_VOL_LABEL_LEN];
status = ioctl (fd, FIOLABELGET, (int)labelBuffer);
```

读取卷标。DOS_VOL_LABEL_LEN 为最大卷标字符个数（11），当卷标字符个数小于 11 时，上述缓冲区内字符串以‘\0’结尾。

- FIOLABELSET

```
status = ioctl ( fd, FIOLABELSET, (int)newLabel );
```

设置卷标。

- FIOATTRIBSET

```
int        fd;
```

```
struct stat    statStruct;
fd = open ("file", O_RDONLY, 0);
fstat (fd, &statStruct);
status = ioctl (fd, FIOATTRIBSET, statStruct.st_attrib | DOS_ATTR_RDONLY);
```

设置文件 fd 属性，原来的属性被覆盖。

- FIOCONTIG / FIOCONTIG64

```
status = ioctl (fd, FIOCONTIG, bytesRequested);
```

为 fd 申请 bytesRequested 字节大小的连续存储空间，fd 可以为常规文件或者目录。

- FIONCONTIG / FIONCONTIG64

```
status = ioctl (fd, FIONCONTIG, &maxContigBytes);
```

获取卷上可以为 fd 分配的最大连续空间大小。单位为字节。

- FIOREADDIR

```
status = ioctl (fd, FIOREADDIR, &dirStruct);
```

读目录 fd 的内容。一般调用前面介绍的 readdir( )函数实现。

- FIOFSTATGET

```
struct stat statStruct;
fd = open ("file", O_RDONLY);
status = ioctl (fd, FIOFSTATGET, (int)&statStruct);
```

获取文件状态信息。一般调用 stat( )或 fstat( )函数实现。

- FIOTIMESET

```
struct utimbuf newTimeBuf; ;
newTimeBuf.modtime = newTimeBuf.actime = fileNewTime;
fd = open ("file", O_RDONLY);
status = ioctl (fd, FIOTIMESET, (int)&newTimeBuf);
```

设置文件的访问时间（如果 newTimeBuf. actime 不为 NULL）和修改时间（如果 newTimeBuf. modtime 不为 NULL）。为 ioctl( )指定第 3 个参数为 NULL 时两个时间都设置为当前系统时间。前面介绍的 dirLib 函数 utime( )完成同样的功能。

- FIOCHKDSK

```
int fd = open (device_name, O_RDONLY, 0);
status = ioctl (fd, FIOCHKDSK, DOS_CHK_REPAIR | DOS_CHK_VERB_1);
close (fd);
```

检查磁盘文件系统的完整性和一致性。第 3 个参数控制检查方式和检查中诊断信息输出。可能的检查方式为：

DOS_CHK_ONLY　只检查，不尝试修复磁盘

DOS_CHK_REPAIR　检查并修复错误

默认时使用 DOS_CHK_ONLY 选项。

诊断信息输出有 3 种：

DOS_CHK_VERB_SILENT 只在错误时报告，不输出其他信息
DOS_CHK_VERB_1 报告错误，检查完毕输出统计信息
DOS_CHK_VERB_2 显示过程中正在检查的每个文件，错误信息及统计信息
默认时使用 DOS_CHK_VERB_1 选项。
库 usrFsLib 函数 chkdsk( )完成同样的功能，见源代码
<install-dir>\target\src\usr\usrFsLib.c

### 6.3.6 连续文件

在默认情况下，VxWorks 为大小超出 1 簇的文件分配磁盘簇采取相对连续的“簇组（Cluster Group）”方式。簇组是一组物理上连续的簇，簇组大小根据磁盘物理特性计算并且相对固定。簇组方式能够减少磁盘定位时间。

VxWorks 还允许为文件/目录指定“绝对连续”方式分配磁盘簇。dosFs 提供的 ioctl 命令 FIOCONTIG/FIONCONTIG 用于实现此种分配方式，必须在文件/目录新建时调用。指定分配给文件的绝对连续簇大小。例如：

```
fd = creat ("/dos1/myDir/myFile", O_RDWR);
status = ioctl (fd, FIOCONTIG, 0x10000);
```

dosFs 将为 fd 保留磁盘空间，该空间不能被其他文件使用。如果要释放文件部分空间，可以通过 ioctl 命令 FIOTRUNC 实现：

```
ioctl (fd, FIOTRUNC, newLength);
```

对许多应用而言，连续文件带来的效率改进并不显著，反而会使磁盘出现较严重的碎片。相比之下，默认的簇组方式能兼顾时间效率，又能显著减少磁盘碎片。

## 6.4 rawFs 文件系统

VxWorks 提供一个比 dosFs 简单得多的文件系统：rawFs，用于只需要基本磁盘 I/O 的应用场合。对比 dosFs，rawFs 具有以下两个突出特点：

（1）rawFs 直接建立在块设备 BLK_DEV 上，不需要中间 CBIO 层；

（2）没有类似 dosFs 的分区以及文件和目录结构的概念，把整个块设备当成一个很大的“文件”，“文件读写”直接转换为对磁盘（BLK_DEV）的扇区进行读写。

因此，rawFs 也可以说成是没有文件系统的“文件系统”。

rawFs 支持在一个 rawFs 设备上同时打开多个文件，虽然每个文件都表示从磁盘开始的整个磁盘。在这种情况下要注意多个文件对相同磁盘区域读写不要导致系统的不一致性。

用于实现 rawFs 的库为 rawFsLib，定义宏 INCLUDE_RAWFS。加入该组件后，系统

初始化时在 usrRoot( )函数中调用 rawFsInit( )对库 rawFsLib 进行初始化。rawFsInit( )同时设置允许打开的 rawFs 文件数 NUM_RAWFS_FILES。

## 1. 创建设备

rawFs 设备由块设备 BLK_DEV 创建，创建函数定义为：

```
#include "rawFsLib.h"
RAW_VOL_DESC *rawFsDevInit ( char * pVolName, BLK_DEV * pDevice )
```

参数 pVolName 表示创建的 rawFs 设备名称，该名称必须全局惟一；pDevice 由块设备驱动程序创建，如 ATA 磁盘驱动 ataDrv 或者虚拟 RAM 盘驱动 ramDrv 等。

上述函数成功创建 rawFs 设备之后，将在系统设备列表中添加名为 pVolName 的设备，并返回指向设备描述结构的 RAW_VOL_DESC 指针。该指针可以用于后面要介绍的其他 rawFsLib 函数。而 rawFs 的文件 I/O 则通过文件描述符 fd 进行，fd 由基本 I/O 调用 open( )得到，如：

```
pBlk = ramDevCreate ( ... );
rawFsDevInit ("/raw", pBlk );
...
fd = open ("/raw", O_RDWR, 0 );
```

可以在一个 rawFs 上打开多个 fd，此时一致性必须由用户程序保证。

## 2. 挂装和卸载

rawFs 中动态挂装和保持文件系统与磁盘一致性的机制和 dosFs 相同。rawFsLib 实现了下面这样两个函数：

```
#include "rawFsLib.h"
void rawFsReadyChange ( RAW_VOL_DESC * pVd );
STATUS rawFsVolUnmount ( RAW_VOL_DESC * pVd );
```

rawFsReadyChange( )用于通知 rawFs 底层介质变化，rawFsVolUnmount( )用于卸载 rawFs 设备，参数 pVd 是创建 rawFs 设备时得到的 RAW_VOL_DESC 指针。

## 3. ioctl

rawFs 支持下列 I/O 控制命令，其含义和 dosFs 同名命令相同，如表 6-10 所示：

表 6-10　I/O 控制命令

| | |
|---|---|
| FIODISKCHANGE | 通知底层介质变化 |
| FIODISKFORMAT | 对 rawFs 无实际意义 |

续表

| | |
|---|---|
| FIODISKINIT | 对 rawFs 无实际意义 |
| FIOFLUSH / FIOSYNC | 刷新 rawFs 缓存内容到磁盘 |
| FIOGETNAME | 根据文件描述符 fd 获取设备名称 |
| FIONREAD | 返回剩下未读字节数 |
| FIOSEEK | 设置当前文件位置 |
| FIOWHERE | 返回当前文件位置 |
| FIOUNMOUNT | 卸载设备 |

# 第 7 章　VxWorks 网络整体分析

从本章开始，我们将陆续介绍 VxWorks 网络的设计和应用。其中：

- 本章将从网络管理员的观点进行整体分析，认识 VxWorks 网络架构，网络协议栈的组织，VxWorks 提供的各种网络功能等；
- 网络应用编程：从网络应用程序开发的角度，分析如何在 VxWorks 上设计 TCP/IP 应用；
- 网络驱动（END）：从底层驱动角度，分析硬件如何驱动、如何与网络协议栈结合在一起。探讨从底层驱动到应用层的 TCP/IP 实现过程。

其中，网络应用编程是本书重点。如果需要更详细了解网络配置和网络驱动方面的细节，可以参考[WRS-npg5.5]。

## 7.1 概　　述

### 7.1.1 TCP/IP 协议简介

TCP/IP 的发展始于 1969 年美国国防部高级研究计划署（ARPA）建立的网络 ARPAnet，当时该网络只包括 4 台位于不同的大学和研究所的主机，并且不是采取 TCP/IP 协议。4 年后 ARPA 的几个研究人员提出了 TCP/IP，并在 20 世纪 70 年代末期逐渐实现出来。后来 TCP/IP 被接纳为 ARPAnet 标准，并被加入到 UNIX 成为其内核功能的一部分，ARPAnet 也发展成为今天庞大的 Internet，而 TCP/IP 则成了事实上的网络标准协议。

如图 7-1 所示是一个简单的 TCP/IP 示意图。传输层协议实现了端到端的传输，一个“端”对应一个网络应用。典型的传输层协议是 TCP 和 UDP，它们差别在于提供不同的端到端的传输质量。传输层的实现建立在网络层基础上，IP 层只实现不可靠的主机到主机的传输。中间的连接 X 可能简单到只是本网络内的一个 HUB，交换机或者路由器，也可能是复杂到跨越广域网的连接。

从上层应用程序角度看，在具体的操作系统上，TCP/IP 具有标准的接口（API），程序通过 API 使用 TCP/IP 提供的服务。存在多种这样的 API，因为 TCP/IP 标准未对此作出规定，因此不同的操作系统有不同的实现。第 8 章将深入介绍一种被广泛采用的 API，即 BSD socket，又称为**套接字**。这部分是 VxWorks 网络程序设计的主要内容。

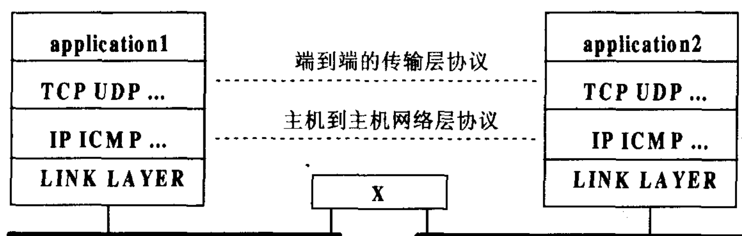



图 7-1　TCP/IP

## 7.1.2　VxWorks 网络栈

VxWorks **网络栈**（即网络层和传输层协议实现）是一个功能完整的兼容 BSD 4.4 的网络协议栈，从之前的基于 BSD 4.3 的协议栈升级而来。

VxWorks 网络栈实现了现代网络协议新特征，如 IP 多播，无类域间路由（CIDR），动态主机配置协议（DHCP），域名系统（DNS）客户端，简单网络时间协议（SNTP）。

VxWorks 的网络栈在如下几方面作了优化：

- 优化了在 TCP 协议层的数据复制（ZBUF）；
- 散列表优化了管理多 TCP 连接的服务器对连接表和多播应用对组地址表的访问；
- 改进了缓冲区管理方案。使用单一的“mBlk－clBlk－簇块”结构，并可以对 6 种不同大小的簇块进行设置。

### 1. IP 多播

IP 多播使 VxWorks 应用能通过一次发送将数据送给一个组内的所有接收者。典型地，可以利用这一特色将音频流、视频流发布给多个用户。基于 VxWorks 网络栈的客户可以动态地加入和退出组。

VxWorks 网络栈的 IP 层内实现了 IGMP 支持，能与支持多播的网关进行组员信息动态交互。同时利用散列表来访问多播组信息，充分考虑了未来单个系统加入许多组的需求。

### 2. DHCP

DHCP 是对 BOOTP 的扩展。在许多无限网络和移动计算应用中，计算机需要移动到另一个地方后快速接入 Internet。BOOTP 依赖于一个静态的配置文件，无法满足这一要求。通过为 DHCP 提供一组地址，DHCP 能在客户请求时动态地分配一个 IP 地址。DHCP 支持 3 种类型的地址分派方式，这些都在 VxWorks 网络协议栈中得到了支持。

### 3. DNS

DNS 是一个分布式的层次结构系统，提供从 IP 名称到域名的解析服务。需要进行解析的称为 DNS “resolver”。VxWorks 网络栈提供 DNS 客户端 API，用户程序通过 API 调用

来使用系统提供的 DNS 服务。

## 4. SNTP

SNTP 用于主机（客户）与 SNTP 服务器进行时间同步，SNTP 服务器本身和世界协同时间（Coordinated Universal Time）同步。SNTP 提供的同步服务在局域网环境下可以达到毫秒级精度，在广域网络环境下可以达到数十毫秒级精度。

## 5. 路由支持

VxWorks 网络栈提供了一组完整的路由功能。优化设计的 IP 层充分考虑了将 VxWorks 用于路由器应用。对新的管理路由表的路由引擎进行了优化，引擎采用称为 PATRICIA 的改进后的二叉树算法提供最快的路由查找。同时引擎提供一组 API 用于程序添加和删除路由。

VxWorks 网络协议栈的组件结构如图 7-2 所示。

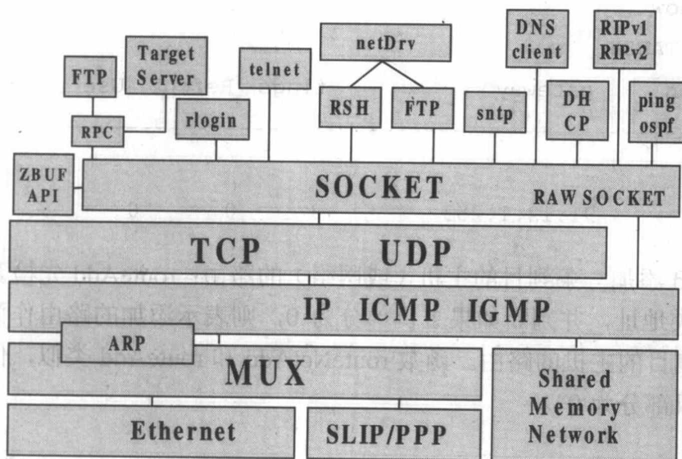



图 7-2　VxWorks 网络组件结构

## 6. 路由

系统维护着一个路由表，保存子网的标志信息，网上路由器的个数和下一个路由器地址等内容。如果只有一个网络接口，分组直接通过该接口发送；当存在多个网络接口时，系统通过查找路由表将决定由哪个接口发送。

VxWorks 允许静态设置路由表，也可以由系统根据网络状态动态修改路由表。

## 7. 设置静态路由

VxWorks 提供了一个路由表操作库 routeLib，提供添加、删除和查看路由的功能。使用 routeLib 需要定义 INCLUDE_NETWRS_ROUTELIB 如表 7-1 所示。

表 7-1　路由表操作库

| routeAdd | 添加一条到目的网络或主机的路由 |
|---|---|
| routeNetAdd | 添加一条到目的网络的路由 |
| mRouteAdd | 添加多条到同一目的地的路由 |
| mRouteEntryAdd | 添加到目的地的特定协议的路由 |
| routeDelete | 删除到一目的地的一条路由 |
| mRouteDelete | 删除到一目的地的路由 |
| mRouteEntryDelete | 删除到一目的地的特定协议的路由 |

函数 mRouteAdd 能添加无类别域间路由（CIDR），也就是“(目的地 IP) /掩码长度”型地址的路由，如下面添加到“90.12.1.*”的路由，其中网络地址掩码为 24 比特。

```
-> mRouteAdd ("90.12.1.3", "91.12.1.253", 0xffffff00, 0, 0)
-> routeShow
ROUTE NET TABLE
destination     gateway           flags  Refcnt  Use         Interface
--------------------------------------------------------------------------
...
90.12.1.0       91.12.1.253       3      0       0           cpm0
```

而 routeAdd 添加一条到目的主机（或网络）的路由：routeAdd 先检查目的地址是一个 A、B 或者 C 类地址，并判断如果主机部分为 0，则表示添加的路由作为到目的网络的路由；否则作为到目的主机的路由。函数 routeNetAdd 和 routeAdd 类似，但目的地址只能是网络地址（主机部分为 0）。

```
routeAdd ("90.12.2.3", "91.12.1.253")  /*到主机“90.12.2.3”的路由*/
routeNetAdd ("90.12.3.0", "91.12.1.253")  /*到网络“90.12.3.*”的路由*/
```

在上面的函数中的点分十进制表示的 IP 还可以用主机名称表示，但是必须在主机表中可以找到对应的 IP。

## 8. 设置动态路由

VxWorks 实现了 RIP（）和 OSPF（Open Shortest Path First）路由协议，属于内部网关协议（IGP），用于在单一自治系统内决策路由，路由协议如表 7-2 所示。

表 7-2　路由协议

| RIPv1（RFC1058） | VxWorks 标准组件（INCLUDE_RIP） |
|---|---|
| RIPv2（RFC1723） | VxWorks 标准组件（INCLUDE_RIP） |
| OSPF（RFC1583） | 可选组件（INCLUDE_OSPF） |

RIP 通过广播 UDP 报文来交换路由信息，每 30 秒发送一次路由信息更新。RIP 提供跳跃计数作为尺度来衡量路由距离，跳跃计数是一个包到达目标所必须经过的路由器的数目。如果到相同目标有两个不等速或不同带宽的路由器，但跳跃计数相同，则 RIP 认为两个路由是等距离的。RIP 最多支持的跳数为 15，即在源网和目的网间所要经过的最多路由器的数目为 15，跳数 16 表示不可达。当组织内需要超过此限制时，需要采取下面的 OSPF 协议。

OSPF 直接通过 IP 分组传输（IP 首部的协议字段为 89）。与 RIP 相对，OSPF 是链路状态路有协议，而 RIP 是距离向量路由协议。OSPF 通过路由器之间通告网络接口的状态来建立链路状态数据库，生成最短路径树，每个 OSPF 路由器使用这些最短路径构造路由表。OSPF 的优势在于：

- 对链路状态的度量比 RIP 的跳数度量更优化；
- 路由更新更及时；
- 可用于跳数超出 15 的网络环境。

OSPF 是可选组件，需要单独获取。使用 RIP 和 OSPF 还涉及一些配置的细节，具体可参考[WRS-npg5.5]。

# 7.2　网络数据流分析

## 7.2.1　网络存储组织

网络部分（TCP/IP 协议层和底层驱动）所使用的内存组织在 3 个内存池上：

- 网络协议栈系统内存池（NSSP，Network Stack System Pool）；
- 网络协议栈数据内存池（NSDP，Network Stack Data Pool）；
- 网络驱动内存池（NDP，Network Driver Pool）。

这 3 个内存池以“mBlk－clBlk－簇块”方式组织。在初始化时从系统堆上分配 3 个内存池的空间，分配后不能增加这些内存池大小。对“mBlk－clBlk－簇块”结构不熟悉的读者可以参考[WRS-npg5.5]。3 个内存池对驱动和协议栈的作用，如同系统堆对以 malloc( ) 方式动态请求内存中应用程序的作用。

### 1．网络协议栈系统内存池（NSSP）

用于系统数据结构开销，如协议控制块（PCB），socket，路由表等。默认时，NSSP 由下列簇块组成：

- NUM_SYS_64　　64
- NUM_SYS_128　　64
- NUM_SYS_256　　64

- NUM_SYS_512  64

上述配置使系统能打开 50 个 socket 并维持其他系统数据结构。它们在 netBufLib.h 中定义，用于构建执行映像。可以找到和上述相似的定义 XXX_MIN，用于构建 BOOTROM 映像，只需要比上面更小的定义。

### 2. 网络协议栈数据内存池（NSDP）

NSDP 包括许多簇块，当应用程序需要 TCP 协议或者 UDP 协议需要发送数据时（例如 send 或者 sendto 调用），协议将从 NSDP 中分配簇用于复制应用程序缓冲区中的数据。协议本身也通过簇块发送数据（ICMP，TCP，RIP 等）。簇块数据最终被驱动程序的输出函数复制到其输出缓冲区，此时是否是 NSDP 簇块视协议而定。

协议栈需要分配足够大的块来装入整个上层数据和协议头部结构开销。如果没有足够大的单个簇块，协议栈会从 NSDP 中分配多个小簇块串起来。簇块大小一般为 2 的 n 次幂。

例如，应用程序中，在一个 TCP socket 上调用 write 发送一个 1460 字节的消息时，一个 2048 字节的簇块从 NSDP 中分配，TCP 协议将数据封装后在发送队列中进行队列，并持有该簇块直到该报文被 ACK。UDP 与此不同，簇块在数据底层被驱动程序复制后即释放。

默认的 NSDP 簇块具有 6 种不同的大小组成：

- NUM_64  100
- NUM_128  100
- NUM_256  40
- NUM_512  40
- NUM_1024  25
- NUM_2048  25

它们在 netBufLib.h 中定义，用于构建执行映像。可以找到和上述相似但值更小的定义 XXX_MIN，用于构建 BOOTROM 映像。

### 3. 驱动程序内存池

Tornado 2.0 允许使用 BSD 4.4 和 END 两种驱动。多播和系统级调试需要使用 END 驱动。驱动程序设置独立于网络协议栈的驱动程序内存池（NDP，Network Driver Pools）来接收和发送数据。NDP 可以采取簇块或者“帧”的形式。注意“帧”和链路层的上的“帧”是两个概念，但是有联系，比如驱动程序。

BSD 驱动程序总是以帧形式组织 NDP。有些 END 完全采取帧形式组织 NDP。有些 END 驱动采取帧形式组织发送数据，而采取簇块形式接收数据。

对于上层协议发送数据，驱动程序将其复制到自己的 NDP。对于接收的数据，END 和 BSD 驱动有不同的做法：END 驱动总是将包含接收数据的簇块“借”给上层协议，避免复制开销；而对于 BSD 驱动，如果要支持借出缓冲区，需要先从协议栈借 NSDP 簇块，有些 BSD 不支持借出缓冲区。

缓冲区必须足够能装下整个接收或者发送的帧。对于 10 Base T 的以太网，最大的帧

为 1518 字节，如果将 IP 头部对齐 4 字节边界，还需要额外 2 字节。

## 7.2.2 数据组织

已经说明，上述 3 种内存池如同 malloc( )请求内存时使用的堆一样；从驱动程序到 socket 层实现，都不断地动态申请和释放内存池上的簇块。

具体来说，有 7 种类型的队列需要从 NSDP 和 NDP 上动态申请簇块，用于保存等待处理的数据，5 个接收队列和 2 个发送队列。

- 接收队列：IP 接收队列，IP 分片重组队列，ARP 接收队列，TCP 重组队列，SOCKET 接收队列；
- 发送队列：网络接口发送队列，SOCKET 发送队列。

这 7 种队列使用从上述 NSDP 和 NDP 分配的缓冲区。协议栈设置其中某些队列的默认大小，并且能够在初始化后改变队列大小。

### 1. IP 接收队列

队列驱动程序从各个网络接口接收的 IP 分组，IP 协议层对队列中的分组进行处理，并传递给上层协议。默认大小为 50，当 IP 分组到来而队列满时，分组被丢弃。

### 2. IP 分片重组队列

IP 允许的最大分组长度为 65535 字节，如果一个分组长度超出 MTU（链路最大传输单元大小），该分组将被重组。接收方用 IP 分片重组队列保存所有需要重组分片。

例如，对 10 Base T 以太网，最大帧长为 1518 字节，除去以太网开销 18 字节，IP 头部开销 20 字节，UDP 开销 8 字节，还剩 1472 字节可以用于应用程序消息数据。如果应用程序通过 UDP socket 发送一个 1473 字节大小的消息，底层将通过两个 IP 分组发送，接收方会进行重组。

在两种情况下，IP 分片重组队列中的分片将被丢弃：（1）在一个默认的时间内没有完成重组；（2）IP 层簇块用完。

### 3. ARP 接收队列

该队列保存 ARP 分组。默认大小为 50，队列溢出时 ARP 分组被丢弃。

### 4. TCP 重组队列

保存乱序到达的 TCP 报文，直到丢失的报文到达。

### 5. SOCKET 接收/发送队列

上述其他各种队列在系统内部都只有一个，而 SOCKET 队列（或者说 SOCKET 缓冲

区）对于每个 socket 存在一个，因而更加重要，而程序中需要调整最多的也是该缓冲区大小，因此我们稍作多一些讨论。

UDP socket 发送队列不实际持有任何数据，应用缓冲区数据直接装入驱动程序发送缓冲区。但是 UDP 会检查发送的消息大小不超出 socket 缓冲区大小，如果超出，函数返回 EMSGSIZE 错误。对于 UDP socket 接收，UDP 将数据报的源 IP 地址和源端口号保存在接收缓冲区。因此每条消息需要附加 16 字节的 UDP 头部开销。如果没有足够的剩余空间，该报文被丢弃（这一错误信息可以通过 udpstatShow 查看）。

驱动程序接收的许多数据最终都放在 socket 接收缓冲区（TCP 或 UDP socket），直到应用程序取走后，缓冲区才得到释放。UDP 接收缓冲区大小影响本地缓存的释放，而 TCP 由于其可靠流服务的特性，接收缓冲区的影响更复杂。

对于 TCP socket，接收缓冲区大小决定了该 socket 连接的接收窗口大小。如果需要修改缓冲区大小，应该在建立连接前通过 setsockopt 调用来修改，因为连接建立过程中双方即确定最大窗口尺寸（为己方空闲接收缓冲区大小的一半）。具体来说对于服务器应该在 listen 调用之前修改；对客户端应该在 connect 调用之前修改。

连接建立后可以动态修改 TCP 缓冲区大小，TCP 协议将在送出的报文中动态通告对方己方的接收缓冲区大小，但是不能超过连接建立时确立的最大窗口尺寸。

为优化 TCP 性能，有研究者指出 socket 的接收缓冲区为 MSS（最大 TCP 报文长度）的偶数倍（大于 3 倍），同时应该不小于链路的带宽（字节/秒）和 RTT（RTT 为链路往返时间）的乘积（乘积表示链路容量）。将链路看成一个管道，不同的带宽和 RTT 表示了不同类型的管道（粗细和长短）。RTT 可以通过测量报文往返时间得到。实际上，仔细的分析还必须考虑网络状态变化对 RTT 的影响，以及服务器处理开销等，因此应该在 RTT 上增加一定的裕度。

例如，对于平均 RTT 为 50ms 的 10 兆以太网，MSS 为典型的 1460 字节，计算链路容量为 10,000,000/8×0.05=62500 字节=42.8MSS，向上圆整到 MSS 偶数倍，即 44MSS=64240 字节，以此作为 socket 接收缓冲区大小将得到性能优化。RTT 在不同的网络环境下差异可能非常大，在没有过载的局域网环境下，RTT 一般小于 10ms，而一条卫星链路，RTT 可能为数百 ms。

新创建的 TCP socket 和 UDP socket 接收缓冲区大小分别由全局变量 tcp_sendspace 和 udp_sendspace 控制，默认情况下为 8192 字节和 41600 字节。新创建的 TCP socket 和 UDP socket 大小分别由 udp_recvspace 控制。默认为 8192 字节和 9216 字节。这 4 个控制变量可以在 shell 下查看和修改：

```
-> udp_recvspace
udp_recvspace = 0xeeedc: value = 41600 = 0xa280
-> udp_sendspace
udp_sendspace = 0xeeed8: value = 9216 = 0x2400
-> tcp_recvspace
tcp_recvspace = 0xeeeb4: value = 8192 = 0x2000
```

```
-> tcp_sendspace
tcp_sendspace = 0xeeeb0: value = 8192 = 0x2000
```

上面设置的初始"大小"表示最大允许大小，并在创建一个 socket 时分配实际的内存空间。可以在 shell 或者程序中修改 4 个全局变量来影响所有新建 socket 的默认缓冲区大小。

对于 socket 的接收和发送缓冲区大小，还可以通过以选项 SO_RCVBUF 或 SO_SNDBUF 调用 setsockopt 来进行修改，但只修改最大允许大小，并不立即申请实际的内存空间。最大允许大小影响了剩余空间的计算：UDP 和 TCP 的 socket 缓冲区的剩余空间为最大允许字节大小减去当前缓冲区被占有的字节大小。

### 6. 网络接口发送队列

IP 层协议对于每个配置的网络接口维护一个发送队列，队列包含需要从该网络接口输出给驱动程序的帧。如果此队列满且驱动程序发送处于忙状态，帧被丢弃，应用层调用得到 ENOBUFS 的结果，队列默认大小为 50。

## 7.2.3 接收：从驱动程序到应用程序的数据流

### tNetTask

任务 tNetTask 的用于实现驱动程序的任务级处理。默认优先级为 50。其执行过程比较复杂，包括图 7-3 中所有阴影部分。下面对该过程进行更深入的说明。

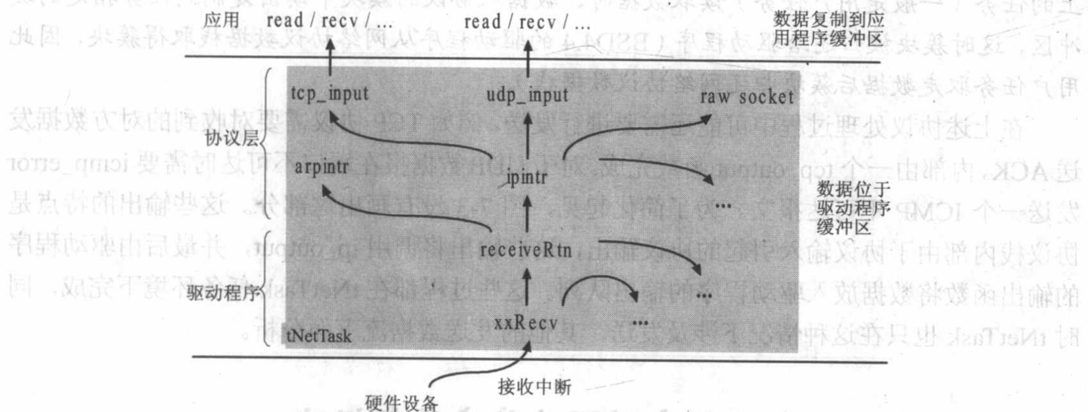



图 7-3　收数据流（阴影部分在 tNetTask 任务环境执行）

tNetTask 有一个能记录 85 个作业的作业队列，驱动程序 ISR 收到一个包后，调用 netJobAdd 添加驱动程序的**任务级接收处理函数 TLRF**（Task-Level Receive Function），在图 7-3 中 TLRF 即函数 xxRecv。同时 netJobAdd 允许 ISR 指定 5 个参数用于调用 xxRecv 时作为其入参。一个 TLRF 及其参数表示了一个**作业**。这种设计哲学是使 ISR 中的工作尽

量转移到任务环境下执行，以使ISR时间最小化。

netJobAdd添加作业后，将计数信号量netTaskSemId加1来唤起tNetTask执行该作业。如果作业队列溢出，将调用logMsg记录一条消息。

当tNetTask唤起运行后，将取出作业队列中所有作业执行。设计驱动程序的xxRecv时，xxRecv必须处理所有的接收帧的工作：和硬件交互，申请存储，组成一帧，保证帧完整无误后，将整个帧连同链路层头部信息通过协议栈函数receiveRtn提交给上层协议。协议栈将根据其中的类型字段决定将包交给谁处理。例如，如果类型为0x800，表明该帧是一个IP分组，则将去掉链路层封装后的分组加入到IP协议的接收队列（ipintr）；如果类型为0x806，则属于一个ARP分组，去掉封装后加入到ARP协议的接收队列（调用arpintr）。这两种协议的接收队列默认大小都是50。如果队列满，帧被直接丢弃。tNetTask在这一过程中（从开始执行作业到调用ipintr和arpintr）通过调用splnet锁定信号量splSemId来和其他网络协议栈任务互斥。

IP协议对接收队列收到的分组首先检查，如果没有被破坏，将调用上层协议的接收处理函数。IP分组头部结构中指明了上层协议类型，典型地，对于TCP或UDP分组，调用的上层协议处理函数是tcp_input或udp_input。

上层协议进一步将数据放入socket的接收缓冲区（缓冲区有空间的话），如果有任务在socket上因一个read调用而阻塞，阻塞任务将解除阻塞。

ARP分组用于实现地址解析，由arpintr进一步处理。

**注意：**

*上面的过程中（从驱动程序到传输层协议处理）没有直接对数据进行复制。而是以mBlk方式在各层处理时复制对簇块clBlk的引用，而不是簇块表示的数据本身。当socket上的任务（一般是用户任务）读取数据时，数据从协议的簇块中读出复制到任务指定的缓冲区，这时簇块被归还给驱动程序（BSD4.4的驱动程序从网络协议数据栈取得簇块，因此用户任务取走数据后簇块归还网络协议数据栈）。*

在上述协议处理过程中可能还需要进行发送，例如TCP协议需要对收到的对方数据发送ACK，内部由一个tcp_output函数完成。对于UDP数据报在端口不可达时需要icmp_error发送一个ICMP不可达报文。为了简化起见，图7-3没有画出这部分。这些输出的特点是协议栈内部由于协议输入引起的协议输出，这些输出将调用ip_output，并最后由驱动程序的输出函数将数据放入驱动程序的输出队列。这些过程都在tNetTask任务环境下完成，同时tNetTask也只在这种情况下涉及发送，其他的发送数据流下面分析。

### 7.2.4 发送：从应用程序到驱动程序的数据流

输出可以由应用程序或者协议栈内部的输入引起。前文说明了协议内部引起的输出在tNetTask上下文中执行，但是从应用程序中发出的输出流不涉及tNetTask。当应用程序中调用sendto send或者write等输出函数时，将发生下列情况（在调用发送的任务上下文中运行）：

- 调用 splnet 锁定信号量 splSemId，和其他协议栈任务互斥；
- 如果 socket 的发送缓冲区中有空间，将应用程序要输出的数据复制到从网络协议栈数据内存池（_pNetDpool）。这里发生第一次数据复制；
- 然后，socket 层将调用对应的协议输出函数，例如对 TCP socket 调用 tcp_output，UDP socket 调用 udp_output；
- 传输层处理后将报文传输给 IP 层，此时已经加上了传输层头部（得到传输层报文）。IP 层进一步封装，然后根据目的 IP 地址，IP 将定位合适的路由，路由即决定了输出的网络接口。然后 IP 层调用此接口上的输出函数（此时得到 IP 分组）；
- 如果需要，接口输出函数会进行地址解析以得到对应 IP 地址的硬件地址。然后，输出函数将帧（此时得到链路帧）加入到接口的输出队列。当为一个网络接口指派一个 IP 地址时，该接口输出队列大小被初始化为 50。如果输出队列中没有空间，接口输出函数返回 ENOBUFS 并将帧丢弃；
- 接口的输出队列中的帧通过 if_start 函数传递给合适的驱动程序发送函数去发送（驱动程序中将发生第 2 次数据复制）。至此，协议栈和上层的处理完毕。驱动程序如何将来自协议栈的帧发送出去在下文介绍；
- 上述过程后，splx 被调用以释放信号量 splSemId。

驱动程序输出到设备的过程：

- 获取驱动程序内部信号量，优先级保护（SEM_INVERSION_SAFE）；
- 从自己的内存池申请一块存储来复制待发送的帧，如果内存申请成功，驱动程序会释放协议栈的存储；如果内存不足，驱动程序返回 END_ERR_BLOCK（调用 if_start 的协议栈的输出函数将检测这一情况并重复将“加入帧到队列－调用 if_start 发送”的过程）；
- 释放信号量。

可以看出，发送过程比接收过程多了一次数据复制。

& VxWorks 实现了另外一种非 BSD 标准的缓冲区可以避免在应用程序和网络栈之间复制数据，即“ZBUF”。可以参考[WRS-npg5.5]。

### 7.2.5 查看函数

VxWorks 提供下列程序，可以用来查看网络的错误信息，利用率，内存使用等信息，对分析和优化网络性能非常有用，如表 7-3 所示。

表 7-3　函数功能表

| 函数名称 | 函数功能 |
|---|---|
| i | 输出当前所有任务信息 |
| muxShow | 显示当前在 MUX 注册的 END 驱动信息 |

续表

| 函 数 名 称 | 函 数 功 能 |
|---|---|
| ifShow | 显示所有网络接口信息 |
| inetstatShow | 显示所有 socket 状态信息 |
| ipstatShow | 显示 IP 协议统计信息 |
| tcpstatShow | 显示 TCP 协议统计信息 |
| udpstatShow | 显示 UDP 统计信息 |
| icmpstatShow | 显示 ICMP 统计信息 |
| mRouteShow | 显示所有路由信息（详细） |
| routeShow | 显示所有路由信息（概要） |
| ripRouteShow | 显示 RIP 所维护的路由信息 |
| iosFdShow | 显示所有文件描述符（含 socket 描述符） |
| arpShow | 显示 ARP 高速地址缓存表中的信息 |
| netStackSysPoolShow | NSSP 内存池统计查看 |
| netStackDataPoolShow | NSDP 内存池统计查看 |
| mbufShow | mbuf 统计查看 |

# 7.3 远程访问服务

VxWorks 提供了一系列远程访问服务功能，使得能够方便地在主机和客户端通过远程网络共享计算功能。VxWorks 提供的客户端和服务器远程访问服务关系如图 7-4 所示。

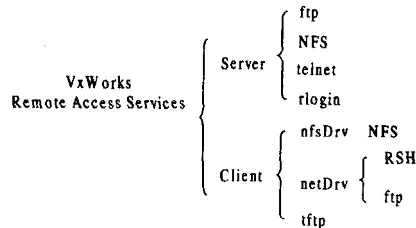



图 7-4 VxWorks 提供的远程访问服务

TSFS（目标服务器文件系统 Target Server File System）允许 VxWorks 目标系统访问主机文件系统，但是通过 Target Server 和 WDB agent 的连接实现。

## 7.3.1 远程登录 rlogin 和 TELNET

远程登录 rlogin 和 telnet 是 VxWorks 交叉开发提供 target server 之外的另外两种方式。

- rlogin——允许远程主机或其他 VxWorks 系统 login 登录到本地 VxWorks 系统。组件定义为 INCLUDE_RLOGIN；
- telnet——允许远程主机通过 telnet 登录到本地 VxWorks 系统。组件定义为 INCLUDE _TELNET。

包含 rlogin 或 telnet 组件需要增加大约 4KB 目标映像。

VxWorks 还允许对登录过程进行账号控制，这需要安装 loginLib（组件定义为 INCLUDE_SECURITY）。安装此组件后，VxWorks 将对 rlogin 和 telnet 的登录进行用户名和密码验证；否则不对远程登录进行任何验证。loginLib 提供用户名和密码的安装，验证以及账号数据库加密等功能。

以这两种方式登录后将自动重定向全局标准输入、标准输出、标准错误输出到登录的远程客户端。

## 7.3.2 NFS 服务器

NFS 服务器允许远程主机通过 NFS 协议访问本地 VxWorks 系统上的 dosFs 文件系统。提供此服务需要使用组件 nfsdLib 和 mountLib：

- mountLib——实现 RFC 1094 定义的挂装协议，宏定义为 INCLUDE_NFS _MOUNT _ALL；
- nfsdLib——实现 RFC 1094 定义的 NFS 协议，宏定义为 INCLUDE_NFS_ SERVER。

默认情况下 NFS 服务器不对客户请求进行任何验证。

在一个 VxWorks 系统上，要输出一个文件系统的步骤为：

- 创建可以挂装的 dosFs 文件系统。在 VxWorks 中对 “/” 使用没有限制，但是用于输出供客户端挂装时必须注意 “/” 的使用，如设备名不能以 “/” 结尾；
- 输出文件系统，如 nfsExport (name, fsId, rdOnly, notUsed)；
- 在远程客户端挂装文件系统。

在 VxWorks 系统上，可以看到如下相关任务：

- tNfsd 守护任务，将客户 NFS 请求进行队列；
- tNfsd0~tNfsd*n* 从队列中取出客户请求处理，最大任务数在初始化时配置；
- tMountd 守护任务，管理每一个挂装请求。

## 7.3.3 FTP 服务器

NFS 只能输出 dosFs 文件系统。VxWorks 另外提供 FTP 服务器，允许输出其他文件系统。FTP 的客户端比 NFS 客户端更普遍。

在 VxWorks 系统下配置 FTP 服务器需要以下组件：

ftpdLib —— 实现 RFC 959 定义的 FTP 服务器端，件宏定义为 INCLUDE FTP_ SERVER,

如果需要客户端验证用户名和密码，还需要定义 INCLUDE_ TPD_SECURITY。

## 7.3.4 NFS 客户端

VxWorks 的 NFS 客户端允许目标系统挂装主机上输出的文件系统，使用挂装的文件系统如同使用本地文件系统一样。使用 NFS 客户端功能需要组件 nfsLib 和 nfsDrv，宏定义为 INCLUDE_NFS。

在客户端挂装主机上的 NFS 文件系统的步骤是：

- 在主机上输出文件系统。这一步依主机而不同，如 UNIX 系统上输出一个文件系统的方式是在/etc/exports 中指定输出的文件系统和权限，而 Windows 系统需要安装第 3 方 NFS 服务器输出；
- 如果主机要求验证，还需要修改默认验证参数（NFS_GROUP_ID = 100，NFS_USER_ID = 2001）。如 nfsAuthUnixSet ( hostName, uid, gid, ngids, aup_gids)。UNIX 主机需要进行验证，Windows 上第 3 方软件一般可以指定是否需要验证；
- 客户机挂装文件系统，如 nfsMount( hostName, hostFs, localName )。

在 VxWorks 客户端还可以查看主机上输出了哪些文件系统，如 nfsExportShow ( hostName )；或者查看客户端已经挂装了哪些文件系统设备，如 nfsDevShow( )。

## 7.3.5 FTP 客户和 RSH

FTP 客户和 RSH 使用 netDrv 提供的远程网络文件 I/O 驱动功能，访问远程网络文件如同本地文件一样。因此 netDrv 和 nfsDrv 有些类似，它们的差异体现在：

- netDrv 需要在打开远程网络文件时一次将整个文件读到本地，然后在副本上进行操作，因此不适合大文件；而 nfsDrv 只需要本地缓存文件一部分，因此能访问任意大的文件，同时加快了 open( )/close( )速度；
- nfsDrv 支持 dirLib 函数；而 netDrv 由于其特殊实现方式，对目录操作有限制；
- nfsDrv 支持磁盘同步 FIOSYNC；而 netDrv 不支持，必须在 close( )时 netDrv 才将文件写入远程主机；
- 由于 netDrv 在 read( )/write( )时只访问本地缓存，因此读写比 nfsDrv 速度快。

参考第 5 章 5.6 节“常用的 VxWorks 设备”对 netDrv 的介绍。

## 7.3.6 TFTP 客户端

前面介绍的远程访问服务的客户端都是先将远程设备“映射”为本地设备，“映射”对后面的访问过程是透明的。“设备”包括文件和目录。TFTP 客户端没有“映射”概念，同

样使目标系统以基本文件系统I/O函数访问远程主机上的文件。

VxWorks的TFTP客户端通过库tftpLib提供，宏定义为INCLUDE_TFTP_CLIENT。该库提供高级函数接口和低级函数接口。一般来说使用高级函数接口tftpXfer( )和tftpCopy( )就足够了。函数tftpXfer( )提供“流”的访问方式：先得到一个文件描述符（代表一个流），然后在该文件描述符流上进行read( )或write( )，完毕后关闭流。具体是：

- 对于“put”型操作，TFTP客户端通过tftpXfer( )得到一个put型的文件描述符，该文件描述符记录了服务器名称、端口、写入TFTP服务器的文件名称以及传输方式的信息，但是真正的数据上传通过对该文件描述符作write( )调用实现，完毕后调用close( )关闭；
- 对于“get”型操作，TFTP客户端通过tftpXfer( )得到一个get型的文件描述符，该文件描述符同样记录了服务器名称，端口，从TFTP服务器读的文件名称以及传输方式的信息。当程序需要访问文件时，对该文件描述符作read( )调用即可；
- 另一个高级接口函数tf tpCopy( )不同于它在远程主机文件和本地文件之间作一次性完整复制。

下面是使用tftpLib函数tftpXfer( )的一个简单示例：

```
#include "tftpLib.h"
#define BUFFERSIZE      512
int  dataFd;
int  errorFd;
int  num;
char buf [BUFFERSIZE + 1];
if ( tftpXfer ("hostname", 0, "hostfilename", "get",
       "ascii", &dataFd, &errorFd) == ERROR )
   return (ERROR);
while ((num = read (dataFd, buf, sizeof (buf))) > 0)  /*读数据*/
{
   ...
}
close (dataFd);
num = read (errorFd, buf, BUFFERSIZE);        /*读错误信息*/
if (num > 0)
{
   buf [num] = '\0';
   printf ("An error occurred: %s\n", buf);
}
close (errorFd);
```

可以看出 TFTP 客户端和 NFS/FTP/NSH 客户端存在一些差异：

- TFTP 客户端没有映射概念；
- TFTP 客户端只能对文件操作，不能对目录操作；
- TFTP 客户端对“流”的操作只能是 read( )或 write( )，取决于是“get”还是“put”。

# 第8章　网络应用编程

网络应用编程即利用 TCP/IP 提供的端到端的传输能力在位于不同的主机上的任务间传输用户数据。操作系统将这种传输能力提供给应用程序的方式一般是套接字（Socket）。

## 8.1　socket 概述

作为一组协议，TCP/IP 没有具体规定应用程序和协议软件的接口细节，以期望使协议不限于任何特定的硬件和操作系统。因此，对于应用程序怎样使用 TCP/IP，标准制订着允许不同的系统定义有不同的定义。

socket 是 BSD UNIX 定义的一种应用程序使用 TCP/IP 协议的接口，被广泛使用，不仅仅限于 BSD UNIX，其他许多操作系统都采用 socket 接口。同时还存在着其他一些 TCP/IP 接口，如图 8-1 所示 socket 框架。

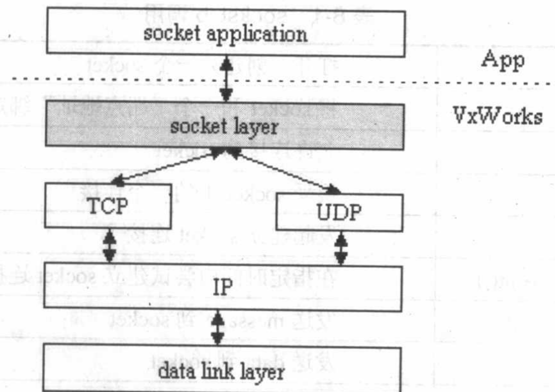



图 8-1　socket 框架

TCP/IP 标准概念性地说明了接口规格，规定接口必须提供下面这样一些操作能力。这些将在本章介绍 socket 接口时体现在各个调用中，先来看一下有助于更好地理解 BSD UNIX 设计者们为什么采取那样的 socket 设计。

- 为通信分配本地资源；
- 指明本地和远处通信断点；
- 发起连接（客户端）；
- 等待客户入连接（服务器端）；

- 发送和接收数据；
- 确定数据如何到达；
- 产生紧急数据；
- 处理流入的紧急数据；
- 文明终止连接；
- 处理远程来的连接终止；
- 异常通信终止；
- 处理差错条件或连接异常终止；
- 在通信结束时释放本地资源。

在设计 socket 时，设计人员充分考虑了复用现有基本 I/O 调用，因此，read，write 仍然在 socket 接口中被采用，但是做了扩展，使这些基本 I/O 调用能操作 TCP/IP 协议；但是 open 被另一个调用 socket（这里“socket”表示一个函数）调用代替。数据结构也增加了定义，这些将在后文中体现出来，如表 8-1 所示 socksLib 调用给出了一个概览。对于基本 I/O 调用，我们在第 5 章作了介绍，在后文将有少量涉及。

VxWorks 的 socket 接口由库 socksLib 提供，定义为宏 INCLUDE_BSD_SOCKET。VxWorks 的 socket 实现是基于 BSD 4.4 UNIX 的，但是简化了对 VxWorks 应用没什么用处的内容。对被简化掉的部分只简单提及，同时会指出 VxWorks 实现的差异。

**表 8-1 socksLib 调用**

| | |
|---|---|
| socket( ) | 打开（创建）一个 socket |
| bind( ) | 将 socket 和一个“端点地址”绑定（注） |
| listen( ) | 允许连接到 socket |
| accept( ) | 接受 socket 上的一个连接 |
| connect( ) | 发起建立 socket 连接 |
| connectWithTimeout( ) | 在指定时间内尝试建立 socket 连接 |
| sendto( ) | 发送 message 到 socket |
| send( ) | 发送 data 到 socket |
| sendmsg( ) | 发送 message 到 socket |
| recvfrom( ) | 从 socket 接收 message |
| recv( ) | 从 socket 接收 data |
| recvmsg( ) | 从 socket 接收 message |
| setsockopt( ) | 设置 socket 选项 |
| getsockopt( ) | 读取 socket 选项 |
| getsockname( ) | 读取本地 socket 端点地址 |
| getpeername( ) | 读取连接的另一端端点地址 |
| shutdown( ) | 关闭连接 |

注：有些文献在表达时不使用"端点地址"，而是用"名称"。源于 BSD-设计 socket 时，定义 socket 可以用于许多 IPC 通信场合，如一个 UNIX 主机内部，这时 socket 表示为一个字符串"名称"。我们使用"端点地址"的说法，更贴切地表示了由"IP 地址"和"协议端口号"标识的 Internet 域 socket。

## 8.2　网络程序设计的特殊之处

网络程序设计需要在不同操作系统、不同机器、不同字长等差异重大的系统上实现统一的协议，不可避免地受底层硬件特性的影响。最关键的一点就是字节顺序的问题，也就是多字节变量在内存中存放顺序的问题。存在两种字节顺序：**大端字节序**（Big-Endian）和**小端字节序**（Iittle-Endian）。

- 大端字节序："大端"（序列中的高位值）首先存放（位于较低的存储地址）；
- 小端字节序："小端"（序列中的低位值）首先存放（位于较低的存储地址）。

以 32 位变量 0x12345678 为例，在大端字节序和小端字节序中的存放方式如图 8-2 所示。

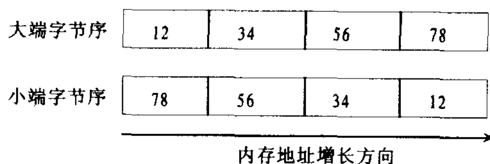



图 8-2　大端和小端

当数据在网络上传输时，采取的是大端字节序，即数据帧发送顺序为从帧的高位值到帧的低位值。

问题的产生源于不同的主机系统存在各种各样的字节序。典型地，摩托罗拉芯片采用的是大端字节序，而英特尔的芯片采用小端字节序。

解决字节序的问题就是将主机字节序转换成和网络字节序一致。转换在应用程序级完成，具体来说，只要应用程序中指定一个多字节变量，而该变量将作为协议头部结构（如 IP 头部结构，TCP 头部结构等）的一个字段时，就必须将其转换为网络字节序。例如，为 socket 绑定一个本地 IP 地址，该 IP 地址将作为每个发出去的 IP 分组头部的源 IP 地址字段，因此必须将其转换为网络字节序。

我们可能期望绑定 socket 的本地 IP 地址时让系统去执行这种转换，因为系统需要从 socket 数据结构中取出 IP 地址组成 IP 分组，因此在这一"取出一写入"的过程中让系统自动完成是最好的。其他多字节变量同样适合系统去转换。这样 socket 应用程序将简单很多，可惜不是这样。

同样地，反方向的转换过程将网络字节序转换到主机字节序。系统的各层协议处理在收到包时都执行了这样的过程。但是一般应用程序中很少需要这种转换，除非需要解码协

议头部结构时。

VxWorks 定义了 4 个宏用于字节序转换。如表 8-2 所示。有些系统上是以函数调用形式实现。在第 9 章将看到许多使用这些函数进行字节序转换的例子。

表 8-2 宏

| 宏 定 义 | 转 换 作 用 |
| --- | --- |
| htons( ) | 将短整数从主机字节序转换到网络字节序 |
| htonl( ) | 将长整数从主机字节序转换到网络字节序 |
| ntohs( ) | 将短整数从网络字节序转换到主机字节序 |
| ntohl( ) | 将长整数从网络字节序转换到主机字节序 |

VxWorks 的宏实现效率要高于函数调用的实现，当主机字节序和网络字节序相同时，开销为 0。虽然 Motorola 体系的处理器上都是如此，但是良好的程序设计实践不应该对其运行的处理器作此假设。

另外一个在网络程序设计中的问题就是地址对齐和位操作。有些体系访问存储时必须对齐某个边界，如 4 字节。在这种体系下，读取一个位于边界上的多字节变量会导致严重错误，如图 8-3 所示：

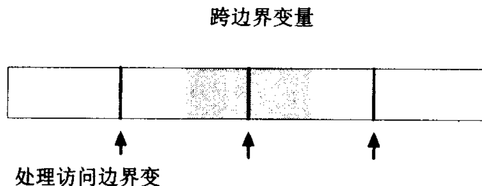



图 8-3 跨边界访问

幸运的是除非编写驱动程序，才不会遇到这个问题。因为在这样的处理器上，任何变量或者动态申请的缓存都是已经对齐了访问边界的，而 TCP/IP 各层协议定义的头部结构都是 4 字节的整数倍，这几乎使所有的处理器都能正确访问。而对于驱动程序（例如以太网卡驱动），假设处理器访问边界为 4 字节，而收到的帧的链路层头部长度为 14 字节，这样将整个帧放在一片连续缓冲区时 IP 分组起始位置将落在两个边界内部，如图 8-4 所示。在此情况下，IP 头部结构中的许多域将无法正常访问。我们对此问题不打算作更多深入，[WRS-npg5.5]对此有些讨论。

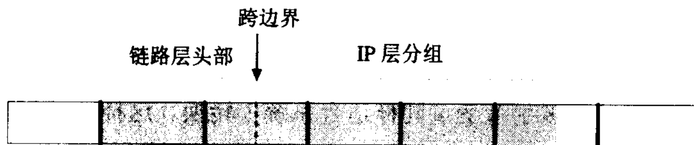



图 8-4 访问边界问题

# 8.3　socket 通信属性

下面从不同的侧面看 socket 属性：socket 抽象，socket 通信域，socket 类型。本小节内容和第 5 章 5.7 节“I/O 系统内部结构”有一定联系。

## 1．socket 抽象

因为沿用了基本 I/O 函数，因此 socket 描述符和文件描述符一样，都在系统的全局文件描述符表中记录，如图 8-5 所示。一个 socket 描述符即代表了一个 socket 的数据结构，该数据结构记录了通过该数据结构完成抽象的端到端通信必需的所有信息。其中尤其关键的是协议控制块（PCB），协议控制块包含与特定协议有关的状态信息和 socket 参数，该协议借助 PCB 得以完成该 socket 的操作。每一种协议都定义了自己的控制块结构。

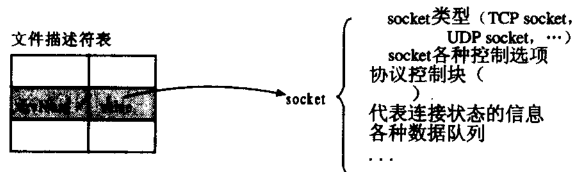



图 8-5　socket 文件描述符

具体地，socket 数据结构 struct socket 在<install-dir>/target/h/net/socketvar.h 中定义，对于一般 socket 应用，几乎不必考虑该数据结构定义的细节。回顾 7.2 节，我们曾简单介绍了 socket 的数据缓冲区。另外，在 8.9 节“socket 应用高级话题”中，我们稍微涉及了 socket 数据结构内部，来得到该结构体内部定义的 I/O 系统文件描述符。

如图 8-1，socket 建立在 TCP 和 UDP 之上。一个基于 TCP 或 UDP 的通信通道由两个 socket 组成，一个 socket 代表通信的一端（如图 8-6 所示）。

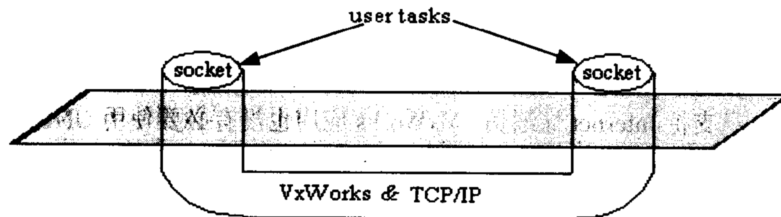



图 8-6　一对 socket 表示了连接的两端

在创建了上述 socket 之后，socket 就一直存在，直到将其关闭。任务通过标准的 I/O 调用 read( )和 write( )即可访问网络。和其他 VxWorks I/O 操作相比，socket I/O 的创建不通过 open( )完成，BSD 专门定义了 socket( )调用来创建一个 socket，socket( )调用将创建图 8-5 中的数据结构，并在系统的全局文件描述符表中记录。同时，我们称 socket( )得到的描

述符为**socket 描述符**。可见，socket 描述符和文件描述符一样，都表示在文件描述符表中的位置。

## 2．socket 通信域

通信域是 socket 的最重要的属性，定义不同软硬件系统进行通信时的语义，这些语义包括对下面几项内容的解释：

- socket 的协议簇（Protocol Family，PF）。实际上 socket 不仅仅是 TCP/IP 的 API 接口，BSD 定义 socket 支持多簇协议，即 PF_XXX， TCP/IP 协议（定义为 **PF_INET**）只是其中之一，同时也是最常用的一簇协议；
- 对名字进行操作和解释的规则；
- socket 的地址簇，每个协议簇都有其特定的地址和端口表示方式，TCP/IP 协议即采取 32 位 IP 地址＋16 位端口号的形式（定义为 **AF_INET**）。

& BSD socket 为 IPv6 定义了新的协议簇：PF_INET6 及对应的地址簇：AF_INET6，目前在 VxWorks 中未被支持。不特别指出时本章内容适于 IPv4。

通常使用的通信域有在 UNIX 主机内部进程间通信的 **UNIX 域**(PF_UNIX+AF_ UNIX）和使用 TCP/ IP 协议通信的 **Internet 域**（PF_INET＋AF_INET）。如图 8-3 所示。

表 8-3　常使用的通信域

| | UNIX 域 | Internet 域 |
|---|---|---|
| socket 类型 | SOCK_STREAM<br>SOCK_DGRAM | SOCK_STREAM<br>SOCK_DGRAM<br>SOCK_RAW |
| 命名 | 字符串名称 | IPv4 32 位 IP 地址＋16 位端口号<br>IPv6 128 位 IP 地址＋16 位端口号 |

UNIX 域并不存在严格意义上的通信协议，因此通信过程中也没有 TCP/IP 那样的交互过程，具有消息队列等 IPC 机制的特点，因此效率比较高。

VxWorks 只支持 Internet 域通信。VxWorks 应用也没有必要使用 UNIX 域通信，在第 2 章中介绍的 IPC 机制足以满足在同一主机系统内部 IPC 的要求。因此 VxWorks 中创建 socket 时总是指定 Internet 通信域。

## 3．Socket 类型

Socket 类型是使用 socket 的高层应用的 socket 属性，对于 Internet 通信域，这些属性表现为 TCP/IP 协议传递该连接的数据时的可靠性、有序性、避免消息重复性等。在创建 socket 时必须指定 socket 类型。两个任务一般通过同类型的 socket 进行通信。

我们可以使用 3 种 socket 类型：可靠的数据流 SOCK _STREAM，数据报 SOCK

_DGRAM 以及裸层 SOCK_RAW，它们的特点如表 8-4 所示。

表 8-4 socket 类型的特点

| | |
|---|---|
| SOCK_STREAM | 可靠的连接，提供有序双向字节流。带外数据传输能力。每一次完整的数据传输都要经过建立连接（虚电路），使用连接，终止连接的过程<br>UNIX 域：SOCK_STREAM 类似全双工管道<br>Internet 域：SOCK_STREAM 使用 TCP 协议实现 |
| SOCK_DGRAM | 数据报方式，无连接，每个报文的最大长度固定，可以单独寻址。不保证报文的有序到达和可靠传递<br>UNIX 域：SOCK_STREAM 类似消息队列<br>Internet 域：SOCK_STREAM 使用 UDP 协议实现 |
| SOCK_RAW | 直接构建在 IP 协议之上的 socket，只在 Internet 域存在 |

# 8.4 socket 端点地址

socket 内部数据结构中最主要的内容之一就是 **socket 端点地址**，用于区分惟一不同的系统中的不同任务（在 UNIX 中是“进程”）的通信连接。

## 8.4.1 数据结构表示

在 PF_INET Internet 域，端点地址由 32 位**主机 IP 地址**和 16 位**协议端口号**组成。如图 8-7 所示，Internet 域的 socket 端点地址由结构体 sockaddr_in 表示。

| 结构长度 | AF_INET |
|---|---|
| 16位协议端口号 | |
| 32位IP地址 | |
| (8字节未用) | |

```
#include "sys/socket.h"
struct sockaddr_in {
    u_char      sin_len;
    u_char      sin_family;
    u_short     sin_port;
    struct      in_addr sin_addr;
    char sin_zero[8];
};
```

图 8-7 Internet 域的 socket 端点地址：sockaddr_in

图 8-7 中，IP 地址由 sockaddr_in 内的结构体 in_addr 表示：

```
#include "netinet/in.h"
struct in_addr {
u_long s_addr;
```

```
};
```

我们习惯上以“.”表示的是点分十进制 IP 地址，在 in_addr 中的表示是将点分十进制表示中的 4 个整数直接转换为二进制，对应到 in_addr.s_addr 的 4 个字节上。例如 IP 地址“127.0.0.1”表示为 in_addr.s_addr=0x7f000001。这是一个用于表示环回地址的特殊 IP，文件 netinet/in.h 还定义了如表 8-5 所示的几个特殊 IP 地址方便程序使用，这些 IP 地址常常在程序设计中用到。

表 8-5 特殊 IP 地址

| | | |
|---|---|---|
| INADDR_ANY | (u_long)0x00000000 | 通配地址 |
| INADDR_LOOPBACK | (u_long)0x7f000001 | 环回地址 |
| INADDR_BROADCAST | (u_long)0xffffffff | 广播地址 |
| INADDR_NONE | 0xffffffff | 作返回值表示地址无效 |

说明：上面定义的 32 位全为 1 的广播地址 INADDR_BROADCAST 是受限广播地址，只是在系统还不知道自己的 IP 和网络掩码时的配置过程中使用。路由器不会转发该类广播。实际上用户程序中广播时指定目的地址为 INADDR_BROADCAST，系统将在发送时根据网络掩码计算得到指向网络或者指向子网的广播。

对于点分十进制字符串表示的 IP 地址到二进制 in_addr 地址的转换，有标准函数可用，见下文。

### 一般性和特殊性

前面已经说明，BSD 设计 socket 接口并不是只为了进行 Internet 域的通信，因此上面 socket 端点地址的讨论只是属于 Internet 通信域的特例。例如，在 UNIX 域 socket 用一个类似“/dev/mysock1”的字符串表示。因此，BSD 实际的做法是定义更一般的表示，即结构体 sockaddr：

```
#include "sys/socket.h"
struct sockaddr {
    u_char   sa_len;          /* total length */
    u_char   sa_family;       /* address family */
    char     sa_data[14];     /* actually longer; address value */
};
```

由于上述一般表示形式和特殊表示形式的差异，在将 sockaddr_in 表示的端点地址传递给 socket 调用接口时，程序需要将其强制转换为结构 sockaddr。具体地，需要结构体 sockaddr 表示端点地址的 socket 调用包括：bind( )，accept( )，connect( )，connectWithTimeout( )，sendto( )，recvfrom( )，getsockname( )和 getpeername( )。

作为从特殊到一般的转换示例，下面的代码段将一个通配 IP 地址（INADDR_ANY）

的端口绑定到一个 socket 上：

```
struct sockaddr_in sock_name;
sock_name.sin_family = AF_INET;
sock_name.sin_addr.s_addr = INADDR_ANY;
sock_name.sin_port = 0;
bind ( sock_fd, (struct sockaddr *) & sock_name, sizeof(sock_name) );
```

若不这样做，在编译时可能输出与指针类型不匹配的警告信息，但是实践中很多人往往没有进行强制转换。

& sockaddr_in 是 sockaddr 在 Internet 域的定义，sockaddr_in 中的“in”即“internet”前面的两个字母。很多其他 in_xxx 或者 xxx_in 定义都与此类似。

## 8.4.2　协议端口号

“端口”是系统维护的一种抽象的软件结构，可以看成任务访问传输层协议服务的访问点，传输层通过端口号来区分不同的上层，包括用户任务或者高层协议。

具体地，任务必须将 socket 绑定在一个端口上，即指定了 socket 端点地址中的协议端口号。这样，传输层传给该端口的数据都被该任务接收，相应地，该任务发送给传输层的数据都通过该端口输出。

通常一个系统（一个 IP 地址）中同时存在许多 socket 和使用 socket 的任务，这时一个显然的要求就是任务在为其 socket 通信指定端口号时必须选择一个没有被其他任务使用的端口。

事实上，有些端口已经被用于实现 TCP/IP 中的许多高层协议，这些端口称为**公认端口**或**保留端口**，如 23 号端口实现 TELNET，80 端口实现 HTTP 等。

除了避免使用公认端口外，BSD 对通信两端的任务使用端口号不做任何限制，也不要求两端的端口号一致。因此位于同一个 socket 通信两端的任务可以在本地系统内自由选择没有被其他任务使用的端口号，只要不重复。在 socket 程序设计中，常常让系统自动确定客户端端口号。

## 8.4.3　地址操作函数

在现实世界，IP 地址表示为一个点分十进制字符串，而 socket 定义的端点地址中的 IP 地址为前文介绍的结构体 in_addr，其惟一成员为一个长整数。VxWorks 的网络地址操作库 inetLib 实现了一组函数用于地址转换。常用的有下面 3 个：

```
#include "inetLib.h"
```

```
STATUS inet_aton ( char * pString, struct in_addr * inetAddress );
u_long inet_addr ( char * pString );
void inet_ntoa_b ( struct in_addr inetAddress, char *pString );
```

上述函数名称中“a”表示“ascii”，“n”表示“numeric”，这里即我们说的二进制地址。

函数 inet_addr( )将点分十进制表示的字符串 IP 地址转换到二进制地址，转换结果为 u_long 型返回值。字符串表示的 IP 地址可以为 8 进制（以 0 开始如“0178.1.2.3”）或者 16 进制（以 0x 开始，如“0x12.3.4.5”）形式。有趣的是该函数转换“255.255.255.255”地址时，得到的结果和函数出错返回结果一样（-1）。因此不能用于该地址转换。建议不使用该函数，而采用 inet_aton( )。

函数 inet_aton( )同样将点分十进制表示的字符串 IP 地址转换到二进制地址，转换结果已经采取了网络字节序，在结构体参数中返回。

本书第 9 章将有许多从字符串 IP 地址到二进制地址的转换例子。

函数 inet_ntoa_b( )将二进制地址转换成点分十进制格式，例如下面的转换后 pString 表示的字符串为“90.0.0.2”。该函数是 BSD 定义的 inet_ntoa( )的改进版，允许重入，但是调用之前必须为 pString 分配空间。

```
struct in_addr iaddr;
...
iaddr.s_addr = 0x5a000002;
...
inet_ntoa_b ( iaddr, pString );
```

除了上述两个函数，inetLib 还实现了其他函数从 IP 地址中分离出网络号和主机号，或者反过来由网络号和主机号构造 IP 地址。

提取网络号：inet_netof( ) inet_netof_string( );

提取主机号：inet_lnaof( )。

# 8.5 socket 应用框架

通过 BSD socket 常常构造两种应用类型：**面向连接的 SOCK_STREAM** 应用和**无连接的 SOCK_DGRAM** 应用。前文已经说明，在 Internet 通信域，面向连接的应用即基于 TCP 协议的 socket 应用；无连接的应用即 UDP 应用。本节给出两种类型的应用框架，通过后面几小节的介绍，读者将明白这些具体通过 socket 调用实现这两种框架时的细节问题。

在内容组织上，我们没有将框架和属于该框架的 socket 调用集中讲述，是因为 BSD 并没有限制一个 socket 调用只能用于特定的应用框架下。例如，close( )调用同时适于两种

socket，在用于 SOCK_DGRAM，它只是进行基本 I/O 系统中的语义；而对于 SOCK_STREAM，它将先进行一个关闭 TCP 连接的过程。读者在阅读本部分内容时需要结合后面几节的内容。

## 1．面向连接的应用：SOCK_STREAM

面向连接的应用中，客户和服务器端在数据传输之前必须建立连接：服务器端调用 bind 将 socket 与一个端点地址绑定，并通过 listen 申明要在该端口侦听客户连接请求（该 socket 即侦听 socket，系统会将到该端点地址的连接请求记录下来），然后调用 accept 在此阻塞，等待请求的到来；客户端通过 connect 连接到服务器的侦听端口。connect 和 accept 将使两端的系统内部进行一个 TCP 3 次握手过程建立端到端的连接。系统将为服务器端 accept 调用创建一个不同于侦听 socket 新 socket 代表和该客户的连接。如图 8-8 所示。

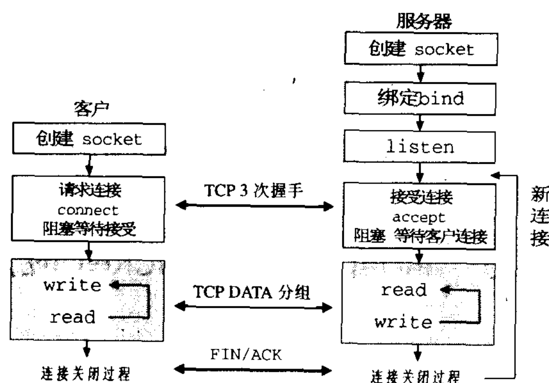



图 8-8　面向连接的 socket 应用框架

连接后的双方的数据以流的形式发送和接收，可以通过 I/O 系统基本调用 read/write 或者专门的 BSD socket 调用 send/recv。

连接关闭过程还涉及"文明地终止"连接，需要解决由谁（发起）终止以及什么时候可以终止。在许多应用中，文明终止连接需要在连接的双方分别调用 close 和 shutdown，在 8.6 节中将深入介绍。

服务器端在处理客户连接请求时可以有两种不同的设计：**并发处理和循环处理**。并发处理：服务器在与一个客户建立连接后，创建一个新任务处理随后的过程，服务器任务则继续"建立新连接－生成新任务"的过程。新生成的任务通过 accept 得到的 socket 描述符为该连接上的客户服务，在服务过程结束后关闭连接。循环处理的服务器直接和客户交互，直到服务结束并关闭连接后才接受下一个客户的连接请求。

## 2．无连接的应用：SOCK_DGRAM

无连接的应用中，在调用 socket 后双方即可开始直接发送（Sendto）和接收（Recvfrom）数据。在 sendto 调用中，同时给出了需要发送的数据和目的端点地址；而 recvfrom 调用则在读出收到的数据的同时，还得到了发送源端的端点地址信息。

在图 8-9 中，客户 socket 都可以和任意的服务器通信；而服务器 socket 收到的数据报也可能来自任意的客户。实际上，connect 调用也用于无连接的应用的客户端，但是不会真正在传输层建立连接，而是实现与特定服务器的通信。

同样地，无连接的 socket 应用中，服务器在处理客户请求时也分并发处理和循环处理两种方式。

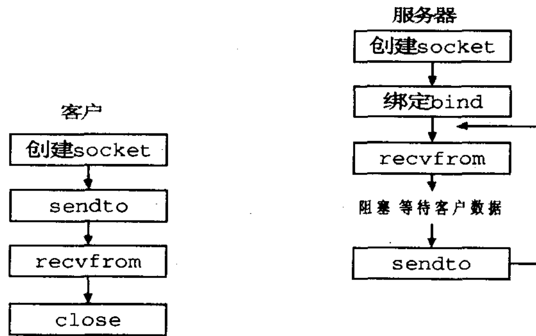



图 8-9　无连接 socket 应用框架

# 8.6　面向连接的 socket 应用

## 8.6.1　创建 socket

创建 socket 是 socket 通信的第一步。socket( )调用创建一个 socket，返回 socket 描述符。该描述符被其他 socket 接口函数和标准 I/O 使用，以惟一标识创建的 socket。

```
#include "sockLib.h"
#include "netinet/in.h"
int socket ( int domain, int type, int protocol );
```

参数 domain 表明通信域，对 VxWorks 总是 AF_INET，即 Internet 域；参数 type 指定 socket 类型：SOCK_STREAM、SOCK_DGRAM、SOCK_RAW；参数 protocol 表示该 socket 属于何种通信协议，即 IP 协议层的载荷类型。在 netinet/in.h 中定义了如下标准协议：

```
#define    IPPROTO_ICMP    1     /* control message protocol */
#define    IPPROTO_IGMP    2     /* group mgmt protocol */
#define    IPPROTO_GGP     3     /* gateway^2 (deprecated) */
#define    IPPROTO_TCP     6     /* tcp */
#define    IPPROTO_EGP     8     /* exterior gateway protocol */
```

```
#define    IPPROTO_PUP      12      /* pup */
#define    IPPROTO_UDP      17      /* user datagram protocol */
#define    IPPROTO_IDP      22      /* xns idp */
#define    IPPROTO_TP       29      /* tp-4 w/ class negotiation */
#define    IPPROTO_EON      80      /* ISO cnlp */
#define    IPPROTO_OSPF     89      /* OSPF version 2  */
#define    IPPROTO_ENCAP    98      /* encapsulation header */
```

对于 SOCK_STREAM 和 SOCK_DGRAM 类型的 socket，一般指定 protocol 为 0，系统自动选择：对于 SOCK_STREAM 类型的 socket，选择协议为 IPPROTO_TCP，对于 SOCK_DGRAM 类型的 socket，选择协议为 IPPROTO_UDP。因此，SOCK_STREAM socket 在本书中和 TCP socket 是等价的，SOCK_DGRAM socket 和 UDP socket 是等价的。在其他 BSD 文档中并不这样将它们划等号是因为在非 Internet 通信域并不一定采取 TCP/IP 协议。

对于 SOCK_RAW 类型的 socket，protocol 表示更复杂的含义，我们在 8.8 节“裸层 socket”中将专门介绍。

下面创建一个类型为 SOCK_STREAM 的 socket：

```
s = socket ( AF_INET, SOCK_STREAM, 0 );
```

如果成功，上面得到的结果 s 表示一个新的 socket；如果创建 socket 失败，函数返回值为-1。

## 阻塞模式

socket 可以是阻塞或非阻塞的。任务试图在阻塞的 socket 上调用 write( )时，如果没有更多的输出缓冲区，该调用将被阻塞；对 read( )调用也是类似的，即要读取的数据没有就绪时，调用者被阻塞。

对于非阻塞式的 socket，即使 I/O 条件不满足，调用也立即返回错误，并设置 errno 为 EWOULDBLOCK（0x46）。具体来说，程序在非阻塞式 socket 上进行下面列出的调用时必须考虑这种出错情况：

- 发起连接调用：write( )  send( ) sendto( )  sendmsg( )；
- 接受连接调用：accept( )；
- 输入型调用：read( )  recv( )  recvfrom( )  recvmsg( )；
- 输出型调用：connect( )。

对于上面的输入型和输出型调用，socket 类型将影响非阻塞调用返回结果。在如表 8-6 所示的情况下这些调用失败，errno=EWOULDBLOCK：

表 8-6 输入/输出调用

| | SOCK_STREAM | SOCK_DGRAM |
|---|---|---|
| 输入型调用 | 没有一个 TCP 字节 | 没有一个完整的 UDP 数据报 |
| 输出型调用 | 缓冲区没有空间。不要求该空间可以容纳整个要求输出的数据，如果部分数据可发送，系统送出可发送部分并返回实际送出了多少数据 | 对于发送 UDP 数据报，阻塞式 socket 上的操作和非阻塞式 socket 上的操作是一样的，都不会阻塞 |

对于接受连接的调用 accept( )，如果没有新的连接，则对于阻塞式 socket，调用被阻塞；对于非阻塞式的 socket，调用失败，errno=EWOULDBLOCK。

仅在 SOCK_STREAM 型的 socket 上才需要发起连接的调用 connect( )，该调用比较复杂，我们将在后文分析。

新创建的 socket 是阻塞的。对阻塞模式的控制通过 socket 的 I/O 控制完成。见第 8.9 节“socket 应用高级话题”。

### 8.6.2 绑定端点地址

在 socket 被创建后，还没有任何端点地址概念，我们在本章开始时介绍了 Internet 域端点地址包括 IP 地址和端口号。

一个有效的通信连接，在任意一端（客户或服务器）看来都表示为这样一个四元组：（本地 IP 地址，本地端口号，远程 IP 地址，远程端口号），这个四元组在整个通信域内都是惟一的。这里，本地和远程的概念是相对的，任意一端都把自己看成“**本地**”而另一端为“**远程**”。那么，任意一端，本地地址的确定方式分：

（1）显式通过一个调用指定“本地”的端点地址，称这一动作为“**地址绑定**”。通常面向连接的服务器应用中在侦听一个端口之前需要先绑定本地端点地址。无连接的服务器应用中，也常常在等待客户数据到达前先绑定本地端点地址；

（2）让系统自动选择本地端点地址。这样一些 socket 调用中，如果没有绑定本地端点地址，系统将自动选择一个：connect，sendto，recvfrom。

对另一端（即远程）端点地址的确定也分两种情况：

（1）面向连接的应用中，远程端点地址在连接建立过程中确定；

（2）无连接应用中，远程端点地址稍复杂一点。可以在每次发送和接收时指定，也可以通过一个类似连接的过程（一个无连接的 connect 调用）确定。

具体地，绑定端点地址的调用为 bind( )，该调用定义为：

```
#include "sockLib.h"
#include "netinet/in.h"
```

```
STATUS bind ( int s, struct sockaddr * name, int namelen );
```

参数 s 为 socket 描述符；name 为要绑定的本地端点地址；namelen 为 name 的结构体长度。

为 socket 绑定的本地端点地址可以是任意有效的网络地址。如果系统有多个 IP 地址，可以绑定到其中之一，这样到其他 IP 地址的连接将不会连接到该 socket。简化的方法可以在绑定时指定通配地址 INADDR_ANY，我们在介绍 8.4 节“socket 端点地址”时曾给出了这样一个端点地址绑定的例子。

不要求同时绑定 IP 地址和端口号。如果绑定的 IP 地址为通配地址，系统将自动选择 IP 地址；如果绑定的端口号指定为 0，系统会自动选择端口号。在客户端有时需要利用这一点来绑定 IP 地址但是让系统选择一个空闲端口。

> 绑定一个 socket 到某个特定（非通配）地址，则该 socket 上发送的数据的源 IP 即为此地址；同时，对于服务器的绑定，还限制了只接收以此 IP 地址为目的地址的连接。客户端不需要事先绑定，如果有多个输出接口，连接时系统会根据服务器 IP 确定路由，并根据以此确定的输出接口来选择源 IP。

## 8.6.3 建立连接

socket 创建后处于无连接状态。在面向连接（socket 类型为 SOCK_STREAM）的应用中，客户端通过 connect( )调用指明和远程的另一个端点地址（即服务器）建立一个 TCP 的 socket 连接。因此客户端需要指定连接的远程端点地址，即远程 IP 地址和远程端口号；服务器端的操作则分两步：（1）通过 listen( )申明要在一个 socket 上侦听，此后系统会自动将客户的连接请求在一个队列中记录；（2）在愿意的时候通过 accept( )来接受队列中一个连接。一般服务器在某个初始化的地方一次性调用 listen( )，然后通过一个循环来 accept( ) 并处理每一个客户连接。

实际上，连接的建立过程是非常复杂的，TCP 采取“3 次握手”来建立连接。在一种非常理想的情况下，连接的建立过程是这样的（假定两端的 socket 都是阻塞式的），如图 8-10 所示：

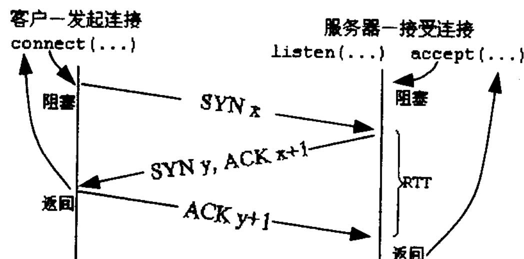



图 8-10　TCP 连接过程

图 8-10 中，RTT 表示一次往返时间（Round Trip Time）。和我们以前介绍过的许多 I/O 不同，connect( )并不会无限等待，TCP 对上述握手过程有超时控制，当连接超时时，调用失败。

连接过程同时确定双方的 TCP 传输的开始序号，在图 8-10 中为 x 和 y。连接过程的复杂性在于连接过程中的同步和超时问题。所幸的是这些都是TCP 和 socket 内部工作的细节，作为进行连接的客户和服务器任务，只需要进行简单的函数调用。但是深入了解 TCP 连接建立过程有助于理解 connect( )/accept( )可能出现的结果及其原因。

## 1．发起连接

客户端调用 connect( )发起连接，该调用定义为：

```
#include "sockLib.h"
#include "netinet/in.h"
STATUS connect ( int s, struct sockaddr * name, int namelen );
```

参数 s 为 socket 描述符；name 为远程端点地址；namelen 为 name 的结构体长度。如果 s 表示的本地 socket 还没有绑定一个本地端点地址，系统将会自动绑定一个合适的（一般在客户端程序中都让系统自动绑定）端点地址。

根据 s 的不同 socket 类型，系统将有不同的处理，如表 8-7 所示：

**表 8-7　socket 类型**

| SOCK_STREAM | 意味着使用 TCP 协议，并试图建立到远程端点的虚电路 |
|---|---|
| SOCK_DGRAM | 将远程端点地址记录在 s 里面，以后在 sendto 和 recvfrom 调用中不再需要指定远程端点地址；否则需要每次指定 |
| SOCK_RAW | 指定发送和接收数据的裸层 socket |

从上面可以看出，只有在 SOCK_STREAM 的 socket 上才会发起建立连接的动作，即 TCP 协议“3 次握手”连接过程。

如果 s 为阻塞式 socket，connect( )将试图建立连接直到 3 次握手过程完毕才返回；对于非阻塞式 socket 不会使 connect( )阻塞，如果不能立即建立连接，connect( )不等待 TCP 协议过程结束就会返回-1。如果 s 上的连接已经建立，调用 connect( )将返回-1。

任何情况下，返回 0 表示 s 上的连接已经建立，程序可以发送和接收数据。connect( )返回-1 时可以通过 errno 了解连接的状态。如表 8-8 所示是常见的与 socket 有关的出错情况，和文件系统相关的出错如 EBADF 没有列出。

**表 8-8　与 socket 有关的出错情况**

| EISCONN | 已连接 |
|---|---|
| ECONNREFUSED | 连接被拒绝。通常是由于指定远程端点地址不存在一个服务器，而不是超出了服务器连接能力 |

续表

| | |
|---|---|
| ETIMEDOUT | 连接超时 |
| ENETUNREACH | 网络不可达 |
| EADDRINUSE | 地址正在使用中（参考8.9节“socket应用高级话题”中对地址重用的讨论） |
| EINPROGRESS | 在一个非阻塞的socket上建立连接并且不能立即建立 |
| EALREADY | 在一个非阻塞的socket上建立连接并且有未完成的连接过程还在进行中 |

**connect()**

在connect建立连接失败后，应该将socket关闭，才能再使用该socket。这是由于从TCP建立连接的内部状态将新建的socket的状态变为“CLOSED”，而连接失败后的状态常常为“SYN_SENT”，必须将其关闭以回到初始状态。

在connect失败后，系统自动绑定的本地端点地址不会被解除。

下面给出一个connect的例子scan函数，该函数通过尝试和指定主机建立连接来测试指定服务器上开放的TCP端口。入参serverName指定服务器IP地址或者名称，start指定开始端口，end指定结束端口。函数的工作就是简单地“建立连接－关闭连接”的过程。

```
#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "hostLib.h"
#include "ioLib.h"
#define SA_LEN    sizeof (struct sockaddr_in)
STATUS scan ( char * serverName, int start, int end )
{
    struct sockaddr_in  serverAddr;  /* server's socket address */
    int          sockAddrSize;    /* size of socket address structure */
    int          sFd;             /* socket file descriptor */
    int          portn;

    bzero ( (char *) &serverAddr, SA_LEN ); /*填写服务器端点地址，除端口号外*/
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_len = SA_LEN;
    if ( ((serverAddr.sin_addr.s_addr=inet_addr(serverName)) == ERROR) &&
        ((serverAddr.sin_addr.s_addr=hostGetByName(serverName)) == ERROR)
        )
    {
        perror ( "scan: unknown server name.\n");
```

```
        return ( ERROR );
    };
    printf("scanning:\n");
    for( portn=start; portn<=end; portn++ ) /*循环测试和服务器每个端口的连接*/
    {
        serverAddr.sin_port = htons ( portn );  /*指定服务器端口号*/
        if ( (sFd = socket (AF_INET, SOCK_STREAM, 0)) > 0 )
          if ( connect (sFd, (struct sockaddr *) & serverAddr, SA_LEN ) == 0 )
                printf ( "[%5d]: connected.\n", portn );
        close (sFd);
    }
    printf("scan finished!\n");
}
```

在测试运行中，scan 的输出结果如下：

```
-> sp scan,"210.43.106.152",1,1024
task spawned: id = 0xf7c9a0, name = t308
value = 16239008 = 0xf7c9a0
-> scanning:
[  135]: connected.
[  139]: connected.
[  445]: connected.
scan finished!
```

上面检测出的 TCP 端口都是公认端口，例如 139 是 NETBIOS 会话服务端口。

**connect( )**

可以这样理解 connect 调用：bind 指定了本地端点地址，而 connect 则指定了远程端点地址。同时如果在 TCP 应用中，connect 还建立了实际的虚电路连接。

## 2. 侦听连接请求

在 TCP 连接的另一端，服务器任务通过一个调用来申明要在 socket 上侦听连接客户端的连接请求，称该 socket 为侦听 socket 或者被动 socket。

实际上，TCP 需要为每个侦听 socket 维护两个队列：已经连接队列和正在连接队列。参考图 8-10，当客户端调用 connect 发送一个连接请求 SYN 包时，该连接被加入到正在连接队列，完成整个连接过程后，该连接移到已连接队列。当队列满时，新的连接请求将被忽略，而不是给客户端一个拒绝信息，因为系统在其他场合（如没有服务器指定的端点地址）时会给出拒绝信息，同时这样设计节约了服务器开销。

具体地，要侦听一个 socket 时调用 listen()完成。listen( )在申明侦听一个 socket 的同时，还需要告诉系统上述连接队列的大小。显然，listen( )调用不需要阻塞。

listen( )定义为：

```
#include "sockLib.h"
STATUS listen ( int s, int backlog );
```

参数 s 表示进行侦听的 socket 描述符，一般已经将 s 和一个本地端点地址绑定；backlog 即请求队列大小。函数成功时返回 0，否则返回-1。

**对 backlog 的讨论：**

在不同的系统不同的时期，对 backlog 有不同的定义：有的系统中 backlog 表示已连接队列和连接中队列大小的总和；有的系统中自动将 backlog 增加到一定比例后表示 2 个队列总和。在 VxWorks 中，backlog 表示连接中队列大小。

如果不期望 socket 接受连接，不应该通过指定 backlog 为 0，较好的做法是将其关闭。指定 backlog 为 0 时不同的系统有不同的解释：允许 0 个连接或者 1 个连接。

允许为 backlog 指定的最大值受系统内部实现的限制，BSD4.4 要求不大于 5。对于 VxWorks，backlog 不能超出 socket.h 中定义的宏 SOMAXCONN 的定义。在 Tornado 2.0.x 中，SOMAXCONN 是不可配置的，但是在 Tornado 2.2 中可配置。

### 3. 接受连接

服务器调用 listen( )只表明愿意在一个 socket 上接受连接以及给出一个持有连接的队列大小。实际接受连接的动作调用 accept( )完成：

```
#include "sockLib.h"
#include "netinet/in.h"
int accept ( int s, struct sockaddr * addr, int * addrlen );
```

参数为已经通过 listen( )调用申明的侦听 socket 的描述符；addr 用于读出连接的另一端（即客户端）端点地址。addrlen 参数有两层意义：在调用时，调用者初始化 addrlen 为 addr 的结构体长度，在函数返回时，系统设置 addrlen 为读出的客户端端点地址结构体长度。

参数 addr 用于让服务器任务了解所连接的另一端是谁，如果不需要了解客户身份信息，可以传递参数 addr 和 addrlen 为 NULL 指针。当前在 VxWorks 中，addr 总是表示由 IP 地址和 TCP 端口号组成的 IPv4 端点地址。

accept( )是能引起阻塞的调用。如果 s 是阻塞式 socket，调用将阻塞直到建立一个 TCP 连接；如果 s 是非阻塞是 socket 并且 s 上当前没有客户连接请求，调用将立即返回“-1”，设置 errno 为 EWOULDBLOCK。

在成功建立一个连接后，accept( )返回值为代表该连接的 socket 描述符。注意区别该描述符和为入参传递的 s 描述符：s 代表的是侦听 socket，服务器需要创建一个这样的 socket 来侦听所有客户的连接请求，该 socket 的生命期通常和服务器任务的生命期一样；而对每个每个已经建立的客户连接，系统自动创建一个新 socket，即 accept( )返回值，该 socket 随着连接的关闭而关闭。

**accept( )**

socket 没有提供机制让服务器在接受连接的过程中指定接收哪个特定客户的连接请求。因此，如果要筛选特定客户，必须在先 accept( )连接后才能通过 close( )关闭不期望的客户连接。

如果服务器需要在多个 socket 上接受连接，较好的处理是使用选择等待（即 select）机制。

## 4. 查看连接的端点地址

BSD 定义了两个调用用于查看连接两端的端点地址：getsockname 查看本地端点地址；getpeername 查看连接的远程端点地址。

```
#include "sockLib.h"
#include "netinet/in.h"
STATUS getsockname ( int s, struct sockaddr * name, int * namelen );
STATUS getpeername ( int s, struct sockaddr * name, int * namelen );
```

对于客户端，getpeername 所得到的远程端点地址即 connect 连接时指定的地址；getsockname 得到的是系统自动绑定或者程序调用 bind 绑定的地址。对于服务器端，getpeername 得到客户端端点地址，和 accept 调用中得到的地址是一致的；getsockname 得到绑定的本地端点地址。

在任何情况下，如果绑定时指定了通配地址，则连接建立后的 getsockname 得到的本地地址是确定的。例如，服务器通过多个网络接口和通配地址服务于客户，则 getsockname 可以使服务器知道当前这个客户所连接的网络接口。

在无连接的 UDP 应用中，也可以有 connect 调用。并且对于已连接的 UDP socket，getsockname 和 getpeername 的上述解释仍然适用。

我们在 8.9 节“socket 应用高级话题”中对连接的建立过程进行了更深入的探讨。

## 5. 查看连接状态

如果定义了 INCLUDE_TCP_SHOW，还可以通过 inetstatShow 查看所有连接的状态。该函数的输出和 UNIX 的 netstat 命令相似。例如：

```
-> inetstatShow
Active Internet connections (including servers)
```

```
PCB     Proto Recv-Q Send-Q  Local Address      Foreign Address    (state)
------- ----- ------ ------  -----------------  -----------------  -------
f6faf8  TCP      0      0    0.0.0.0.1024       0.0.0.0.0           LISTEN
f6b5c4  TCP     0      0   168.10.10.152.80  168.10.10.252.11   TIME_WAIT
f74b30  UDP      0      0    0.0.0.0.36096      0.0.0.0.0
...
```

## 8.6.4 在连接的 socket 上发送和接收

已经连接的 socket 即 SOCK_STREAM 型 socket，如同一条虚电路，socket 描述符惟一表示了该虚电路，在其上的数据传递数据“流”时只需要该 socket 描述符即可。“流”在这里一个意思就是系统将不区分用户数据的边界，整个用户提交发送的数据按照提交顺序“流入”和“流出”虚电路，“流”的传递过程是可靠的。

可以简单地使用基本 I/O 调用：read( )和 write( )，来实现发送和接收。我们在第 5 章介绍过这两个函数，在 socket 上使用时方法也是一样的，例如：

```
read( s, buf, buflen );     /*接收 socket 数据*/
write( s, buf, buflen );    /*发送 socket 数据*/
```

参数 s 为已经连接的 socket 的描述符；buf 表示接收的数据放入的目标缓冲区（对于 read)，或者表示要发送的源数据缓冲区（对于 write)；buflen 表示期待接收多少字节（对于 read)，或者 buf 中要发送的数据字节数（对于 write)。

实际上，系统将上述 read( )和 write( )调用引导给 TCP 的发送 tcpread( )和接收 tcpwrite( )去实现，但是我们不用考虑这些。

发送和接收受 socket 的阻塞方式影响，如表 8-9 所示：

**表 8-9 阻塞式和非阻塞式的影响**

| | 阻塞式 socket | 非阻塞式 socket |
|---|---|---|
| Read( ) | 如果 socket 接收缓冲区没有数据，调用将阻塞；当数据到达时被唤醒，读取数据并返回实际读取字节数（可能小于要求的读取字节数） | 如果 socket 接收缓冲区没有数据，返回-1，errno=EWOULDBLOCK；否则读取 socket 接收缓冲区已有字节数（不超过 buflen）到 buf，返回实际读取的字节数 |
| Write( ) | 如果 socket 发送缓冲区没有空间，调用被阻塞；当缓冲区有空间可用时被唤醒，送出数据并返回实际发送字节数（可能小于要求发送的字节数） | 如果 socket 发送缓冲区没有空间，返回-1，errno=EWOULDBLOCK；如果空间不足，按照实际空间发送，返回实际发送的字节数 |

read( )和 write( )返回值表示实际读取或者发送的字节数，或者为-1 时表示错误。对于 read( )，不论阻塞方式，都存在一种情况即函数返回值等于 0 且小于 buflen，这表明连接单向或者双向被停止，见 8.6.5 节对关闭连接的讨论。

后文将会说明 read( )和 write( )可以在 SOCK_DGRAM 类型的 socket 上使用，也受因阻塞式 socket 和非阻塞式 socket 的影响，但语义和这里的稍有差异。

除了 read( )和 write( )，程序还可以使用 BSD socket 中定义的专门用于流发送和流接收的调用：recv( )和 send( )。

```
#include "sys/socket.h"
int recv ( int s, char * buf, int bufLen, int flags );  /*发送到socket*/
int send ( int s, const char * buf, int bufLen, int flags ); /*从socket接收*/
```

recv( )和 send( )比 read( )和 write( )增加了一个重要的参数 flags，可以进行一些特殊的控制。其他参数和函数返回值都是一样的。

参数 flags 表明下述特性，将被传递给底层协议。如图 8-10 所示在 sys/socket.h 中定义：

表 8-10　参数特性

| | | |
|---|---|---|
| MSG_OOB | 0x1 | 发送/接收带外数据 |
| MSG_PEEK | 0x2 | 数据预览 |
| MSG_DONTROUTE | 0x4 | 发送时不使用路由表 |

数据预览（MSG_PEEK）用于 recv( )，使调用者“偷看”数据，即：调用者读到数据，但在 TCP 内部，这部分数据仍然标志为“未读取”，下次 read( )或 recv( )将读出这部分数据。

MSG_DONTROUTE 用于 send( )，用于路由表管理，一般应用中不需涉及。

除了 flags 控制参数带来的不同之外，recv( )/send( )和 read( )/write( )在阻塞语义上是一样的。

& TCP 具有传递**带外数据**（Out-of-band Data）的能力。设计带外数据传递时应用能打破按照常规 TCP 数据流处理顺序，优先处理带外数据。正常的 TCP 数据流总是要求接收者顺序地处理所有数据。在 UNIX 系统上，socket 连接上有带外数据到来时可以给进程发送信号，也可以在 recv 调用中指定标志 MSG_OOB 来读取。带外数据可以和正常数据流一起传输，也可以单独传输，可以通过 socket 选项 SO_OOBINLINE 选择。但是带外数据缺乏具体的标准，并且当前在 VxWorks 中的实现有限制。

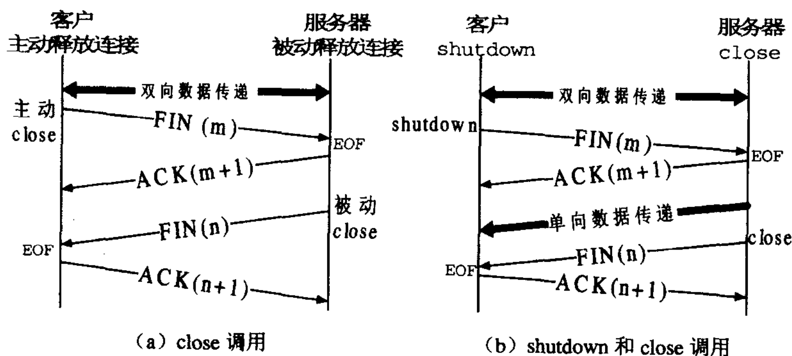
## 8.6.5 关闭连接

当客户或者服务器不再需要一个 socket 时，调用 close( )将其关闭。对 SOCK_DGRAM 和 SOCK_STREAM 类型的 socket 都通过该调用来关闭。

close( )是一个基本 I/O 系统调用，因此，除了具有基本 I/O 系统中的语义外（从 I/O 系统中关闭，即释放 I/O 资源），还有 socket 特定语义，即：如果 socket 属于 SOCK_STREAM 连接类型，系统将会尝试将系统内部缓冲区的数据送出，直到送出或者超时时将数据丢弃才正式关闭 socket。显然，close( )先执行 socket 特定语义，然后执行基本 I/O 系统中的语义。

调用 close( )也用于关闭 SOCK_DGRAM 类型的 socket，但是仅仅需要执行基本 I/O 系统中的语义。

关闭 socket 的过程和建立 TCP 连接一样，也需要进行同步（通知对方来关闭连接），但是更简单。如图 8-11 中，客户端先调用 close( )来主动释放连接，这将导致 TCP 向对方发送一个 FIN 报文表示完成，服务器收到该 FIN 后将使相对位置的 read 调用得到 0 返回值（解释为文件结束 EOF）；服务器端在收到 EOF 后应该调用 close( )关闭，因为在此语义下，对方调用 close( )后将不会对该 socket 进一步读写。




（a）close 调用　　（b）shutdown 和 close 调用

图 8-11 关闭连接

在客户－服务器应用中，以图 8-11（a）中的方式关闭 socket 存在两个值得考虑的问题：由哪一方主动关闭连接，以及什么时候关闭连接。不论客户还是服务器单方面 close( )关闭连接都有不妥之处：某些应用中，在客户端关闭时不能确定是否所有的服务器数据都已经到达；同样在服务器关闭也不能确定是否客户还有更多的服务请求。因此，在这类应用中，只通过 close( )调用来关闭连接存在歧义。

在 BSD 4.2 中首次定义了 shutdown，一个方向的传输完毕后可以单方向关闭，即**部分关闭**。该调用消除了关闭过程中的歧义。

```
#include "sys/socket.h"
STATUS shutdown ( int s, int how );
```

参数 s 为 socket 描述符，how 具体控制 shutdown( )的动作，如表 8-11 所示：

表 8-11　控 制 动 作

| | |
|---|---|
| 0 | 停止接收，任务此后不再对 socket 进行读取。当前系统的内部接收缓冲区中的数据都被丢弃；此后收到的数据直接丢弃，但是给对方发送确认 |
| 1 | 停止发送，任务此后不再对 socket 进行写操作。当前系统内部发送缓冲区数据都被发送 |
| 2 | 同时停止接收和发送 |

可以看出停止发送和停止接收的语义是不对称的：停止接收只是本地任务不再读 socket 中的数据，并且丢弃当前和以后 socket 中的数据，对远程没有影响；停止发送时，系统将依旧送出当前 socket 中的数据（这一点很重要），然后发送一个单方向停止序列 FIN 告诉远程本地已经完成数据发送。

对于收到 FIN 的一方，影响是两方面的：（1）在从 TCP 内部，系统将发送对 FIN 的确认，并设立相应标志表明已经关闭了部分连接；（2）该位置应用程序的 read 调用将得到返回值为 0 的结果，以表明结束（EOF）。

从图 8-11 可以看出：从技术上讲，close()和 shutdown( )调用都会使对端得到 read 调用返回值为 0 的结果。因此，对于对这种情况的解释完全是应用程序决定的：如果应用程序“认为”这时对方调用了 close( )，则应该立即调用 close( )，即使还有数据要发送；如果“认为”是对方调用了 shutdown()，则还可以继续发送，等数据发送完毕再调用 close 关闭连接。因为 shutdown( )只是 BSD socket 调用，并没有执行 close( )具有的 I/O 系统语义，因此 shutdown( )停止 s 的发送后还可以继续使用 s 进行接收。

**文明终止连接**

本章开始时指出了 TCP/IP 标准的概念性接口规格规定接口必须能够文明地终止连接。文明终止连接意味着两点要求：（1）不能在某一方还需要传输时关闭连接；（2）对于可靠连接，还要求关闭时高层程序已经提交给传输层的数据要“尽量”可靠地传送，因为高层程序已经假定这部分数据可靠地传递了。没有 shutdown 时文明终止连接是非常困难的。

通过 shutdown，一方在数据传输完毕后调用 shutdown 停止发送，另一方能判断出这一点并在送出最后的数据后调用 close()，即实现了文明的终止连接。可以看出，close 和 shutdown 停止发送时都仍然“尽量”送出系统发送缓冲区的数据对于保证上述第（2）点要求是非常重要的。这里，在连接终止时的传输只能是“尽量”而不是“可靠”。

### 8.6.6 面向连接的 socket 示例

我们知道，在 TCP/IP 协议簇中定义了 ICMP，该协议可以实现的一个功能就是测试网

络连通性，如著名的工具 ping 就是利用了 ICMP。作为面向连接的 socket 的一个例子，在这里给出一个在 TCP 层测试连通性的简单程序 tcpEcho.c，我们定义了两个函数如表 8-12 所示：

表 8-12　函　　数

| | |
|---|---|
| tcpES( ) | 作为服务器任务运行。指定的入参表示服务器端口号。可以在 shell 下运行，如："sp tcpES, 5000"，注意选择一个未用的端口号 |
| tcpEC( ) | 作为客户任务运行。指定的入参为服务器名称（IP 地址或者主机名称）和服务器端口号。可以在 shell 下运行，如："sp tcpEC, "localhost",5000"，端口号必须和服务器端口号一致 |

测试过程是客户向服务器发送一个预定义的字符串"1234567890"，服务器收到后直接将该字符串返回，然后客户端读出服务器返回的结果并显示。为了简化设计，服务器对客户连接请求采取循环处理的方式，即经过"接受一个连接－服务交互－关闭连接"后才服务下一个客户请求，容易将其扩展到并发处理。本例中客户和服务器端的 socket 都是阻塞式的。

```
/*
 *tcpEcho.c : TCP echo. Client/Server
 */
#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "hostLib.h"
#include "ioLib.h"
#include "strLib.h"

#define MSG "1234567890"
#define MSG_SIZE        10
#define MAX_CONNECTION  2
#define SA_LEN          sizeof ( struct sockaddr_in )

/*tcpEC: 客户端函数，servarName 为服务器名称，portn 表示服务器端口号*/
STATUS tcpEC ( char* serverName, int portn )
{
    struct sockaddr_in  serverAddr;          /*服务器端点地址*/
    char            replyBuf[MSG_SIZE+1];  /*接收回应的缓冲区*/
    int             nRead;
    int             sFd;                   /*客户 socket 描述符*/
```

```
/*初始化服务器端点地址*/
bzero ((char *) &serverAddr, SA_LEN);
serverAddr.sin_family = AF_INET;
serverAddr.sin_len = SA_LEN;
serverAddr.sin_port = htons (portn);
/*由服务器名称得到二进制地址*/
if ( ((serverAddr.sin_addr.s_addr = inet_addr (serverName)) == ERROR) &&
     ((serverAddr.sin_addr.s_addr = hostGetByName (serverName)) ==
ERROR)
   )
{
   perror ("Client: unknown server name.\n");
   return (ERROR);
};

/*创建客户的 socket*/
if ((sFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
{
   perror("Client: create socket failed.\n");
   return(ERROR);
};

/*连接服务器*/
if ( connect(sFd, (struct sockaddr *)&serverAddr, SA_LEN) == ERROR )
{
   perror ("Client: connect failed.\n");
   close (sFd);
   return (ERROR);
};

/*交互过程: "发送一接收",显示结果*/
if (write (sFd, MSG, MSG_SIZE) == ERROR)
{
   perror ("Client: write failed.");
   close (sFd);
   return (ERROR);
}
```

```
    if ( (nRead = read (sFd, replyBuf, MSG_SIZE)) == ERROR)
    {
        perror ("Client: read failed.\n");
        close (sFd);
        return (ERROR);
    }
    replyBuf[nRead] = 0;  /*字符串结尾的0*/
    printf("Client: the server replied [%s]\n", replyBuf);

    /*关闭连接，任务结束*/
    close (sFd);
    return (OK);
}

/*tcpES: 服务器函数，参数portn表示服务器侦听端口号*/
STATUS tcpES ( int portn )
{
    struct sockaddr_in  serverAddr; /*服务器端点地址*/
    struct sockaddr_in  clientAddr; /*客户端点地址*/
    int                 sockAddrSize;  /*端点地址结构体长度*/
    int                 lsnFd;    /*用于侦听的socket描述符*/
    int                 conFd;    /*代表和客户连接的socket描述符*/
    int                 nRead;    /*从客户读到的字节数*/
    static char         replyBuf[MSG_SIZE];

    /*创建socket*/
    if ((lsnFd = socket (AF_INET, SOCK_STREAM, 0)) == ERROR)
    {
        perror ("Server: socket failed.\n");
        return (ERROR);
    };

    /*初始化服务器的本地端点地址，与socket绑定*/
    bzero ((char *) &serverAddr, SA_LEN);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_len = (u_char) SA_LEN;
    serverAddr.sin_port = htons (portn);
    serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);
```

```
if (bind (lsnFd, (struct sockaddr *) &serverAddr, SA_LEN) == ERROR)
{
    perror ("server: bind failed.\n");
    close (lsnFd);
    return (ERROR);
};

/*申明要在绑定地址后的 socket 上侦听客户连接请求*/
if (listen (lsnFd, MAX_CONNECTION) == ERROR)
{
    perror ("Server: listen failed.\n");
    close (lsnFd);
    return (ERROR);
};

printf ( "Server: port %d ready ... \n", portn );

/*以循环方式处理客户连接请求*/
while(TRUE)
{
    sockAddrSize = SA_LEN;
    if ( (conFd = accept (lsnFd, (struct sockaddr *) /*接受连接*/
            &clientAddr, &sockAddrSize ) ) == ERROR)
    {
        perror ("Server: accept failed.\n");
        close (lsnFd);
        return (ERROR);
    };
    printf ( "Server: the connected client ip=[%s] port=[%hd]\n",
            inet_ntoa(clientAddr.sin_addr), clientAddr.sin_port
        );  /*显示连接的客户*/

    /*读客户数据，然后直接写回到客户*/
    if ((nRead = fioRead (conFd, replyBuf, MSG_SIZE)) > 0)
    {
        if ( write(conFd, replyBuf, nRead)!=nRead )
            perror ("Server: write\n");
    }
    else
```

```
            perror ("Server: read\n");
        close (conFd);    /*关闭与该客户的连接*/
    }
}
```

运行可以得到类似下面的结果：

```
-> sp tcpES,5000
task spawned: id = 0xf77010, name = t4
value = 16216080 = 0xf77010
-> Server: port 5000 ready ...
sp tcpEC, "vxTarget", 5000
task spawned: id = 0xf71fd8, name = t5
value = 16195544 = 0xf71fd8
-> Server: the connected client ip [210.43.106.252] port=[1042]
Client: the server replied [1234567890]
```

# 8.7　无连接的 socket 应用

无连接的 socket 应用和面向连接的 socket 应用有许多相似之处，如：创建 socket 的过程是一样的；在服务器端，一般也需要绑定端点地址；socket 通信完毕，也执行 close 释放系统资源。因此，8.6 节的许多调用都要在本节中用到。本节只介绍无连接应用中区别于面向连接应用的不同的内容。

在面向连接应用中，“客户”和“服务器”以主动和被动的方式为一次通信过程建立一条端到端的虚电路连接；在无连接的应用中，通信双方在整个过程中是平等的，也没有建立连接的过程，双方直接通过 socket 调用来发送或者接收数据报。在无连接 socket 中，每个收发的数据报都是具有目的端点地址的。

## 8.7.1　sendto 和 recvfrom

无连接的发送和接收由调用 sendto 和 recvfrom 完成：

```
#include "sys/socket.h"
#include "netinet/in.h"
int sendto ( int s, caddr_t buf, int bufLen, int flags,
      struct sockaddr * to, int tolen );
```

```
int recvfrom ( int s, char * buf, int bufLen, int flags,
    struct sockaddr * from, int * pFromLen );
```

和 send( )/read( )相比，sendto( )和 recvfrom( )的前面 3 个参数都是一样的。增加的参数：to 表明数据报发往的端点地址；tolen 为 to 表示的结构体的长度；from 用于保存收到数据报的源端点地址；参数 pFromLen 在调用 recvfrom( )时表示 from 结构体长度，在调用返回时表示收到的端点地址结构体长度。参数 flags 指明控制选项：MSG_OOB / MSG_PEEK / MSG_DONTROUTE，和 send( )/read( )中的 flags 一样。

为 sendto 指定的后面两个参数类似 connect 调用的后面两个参数。为 recvfrom 指定的后面两个参数类似 accept 的后面两个参数，同样，如果对数据来源地址不感兴趣，可以指定 recvfrom 的参数 from 和 pFromLen 都为 NULL 指针。

传输层在实现 sendto 和 recvfrom 调用时，相比面向连接的 send/recv，一个特点是对将数据看成完整的，不会进行拆分。一次 sendto 调用所发送的数据只会被一次 recvfrom 读出，如果 recvfrom 中给定的缓冲区 buf 大小小于传输层所接收的报文大小，超出部分将被丢弃。

两个函数在成功时返回值都是实际发送或者读到的字节数。失败时他们返回-1。一个应用程序可以感觉到的差异是 sendto 可以发送长度为 0 的报文，在接收端的 recvfrom 将返回读到的字节数为 0，但不像面向连接的应用那样将读到 0 字节解释为连接关闭。

sendto( )不受阻塞和非阻塞方式影响，它总是立即送出数据，即使服务器没有运行；对 recvfrom( )，在没有数据报到来时，如果 socket 为阻塞方式，调用会一直阻塞，如果为非阻塞方式，调用立即返回-1，errno=EWOULDBLOCK。

还有一组调用 sendmsg 和 recvmsg，完成和 sendto/recvfrom 类似的功能：

```
#include "sys/socket.h"
#include "netinet/in.h"
int sendmsg ( int sd, struct msghdr * mp, int flags );
int recvmsg ( int sd, struct msghdr * mp, int flags );
struct iovec { /*iovec 每一项表示一块数据（发送或接收）*/
 caddr_t iov_base;  /*数据缓冲区指针*/
 int iov_len;       /*缓冲区大小*/
};
struct msghdr {
 caddr_t msg_name;          /*端点地址*/
 u_int    msg_namelen;      /*端点地址 msg_name 长度*/
 struct  iovec *msg_iov;    /*iovec 数组*/
 u_int    msg_iovlen;       /*上述数组中元素个数*/
 caddr_t msg_control;       /*访问权限（VxWorks 不支持）*/
 u_int    msg_controllen;   /*访问权限 msg_control 长度*/
 int      msg_flags;        /*收到数据标志*/
};
```

调用 sendmsg 和 recvmsg 更复杂，它们将寻址信息以及缓冲区信息封装在一个结构体 msghdr 中，sendmsg 和 recvmsg 的数据缓冲区可以是离散的多个，并且有访问权限控制。对于 VxWorks 而言，它们和 sendto/recvfrom 是一样的。

**校验和**

VxWorks 基于 BSD 4.4 实现的 UDP 允许对其发送和接收的 UDP 数据报进行校验和控制。对某些应用而言，数据到达比数据无错更重要，因此这类应用的 UDP 报文不进行校验和检查更可取，并不只是节约了计算开销。

送出的数据报是否计算校验和由 udpcksum 控制，接收数据报是否检查校验和由 udpDoCkSumRcv 控制。这两个都是全局变量，可单独控制。默认情况下它们都是 1（表明要做校验），将其设置为 0 表示不做校验、计算和检查。

## 8.7.2　无连接的 socket 示例

这里给出一个无连接的 socket 例子 udpEcho.c，类似 8.6 节“面向连接的 socket 应用”中给出的一个面向连接的 socket 示例，差别在于这里采用无连接的 socket，系统传输层协议为 UDP。同样包括两个函数，如表 8-13 所示。

表 8-13　函　　数

| | |
|---|---|
| udpES( ) | 作为服务器任务运行。指定的入参表示服务器端口号。可以在 shell 下运行，如：“sp udpES, 5000”，注意选择一个未用的端口号 |
| udpEC( ) | 作为客户任务运行。指定的入参为服务器名称（IP 地址或者主机名称）和服务器端口号。可以在 shell 下运行，如：“sp udpEC, ‘localhost’,5000”，端口号必须和服务器端口号一致 |

测试过程是客户向服务器发送一个预定义的字符串“1234567890”，服务器收到后直接将该字符串返回，然后客户端读出服务器返回的结果并显示。本例中客户和服务器端的 socket 都是阻塞式的。

```
/*
 * udpEcho.c UDP echo. Client/Server
 */
#include "vxWorks.h"
#include "sockLib.h"
#include "inetLib.h"
#include "hostLib.h"
#include "ioLib.h"
```

```
#include "strLib.h"

#define SA_LEN     sizeof ( struct sockaddr_in )
#define MSG "1234567890"
#define MSG_SIZE       10

/*udpEC：客户端函数，servarName为服务器名称，portn表示服务器端口号*/
STATUS udpEC (char * serverName, int portn )
{
    int     s;                          /*socket描述符*/
    struct sockaddr_in serverAddr;      /*服务器端点地址*/
    char    recvBuf[ MSG_SIZE + 1 ];    /*接收缓冲区*/
    int     n;

    /*创建客户端socket*/
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s < 0)
    {
        perror("Client: opening datagram socket");
        return (ERROR);
    }

    /*初始化服务器端点地址*/
    bzero ((char *) &serverAddr, SA_LEN);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_len = SA_LEN;
    serverAddr.sin_port = htons (portn);
    /*由服务器名称得到二进制地址*/
    if ( ((serverAddr.sin_addr.s_addr = inet_addr (serverName)) == ERROR) &&
        ((serverAddr.sin_addr.s_addr = hostGetByName (serverName)) == ERROR)
      )
    {
        perror ("Client: unknown server name.\n");
        return (ERROR);
    };

    /*发送数据报到服务器*/
    if ( sendto ( s, MSG, MSG_SIZE, 0, (struct sockaddr *)
```

```
                &serverAddr, SA_LEN ) < 0 )
            perror("Client: sending datagram message");
    /*接收服务器返回结果*/
    if ( (n=recvfrom (s, recvBuf, MSG_SIZE, 0, NULL, NULL)) >= 0 )
    {
        recvBuf[n] = 0;
        printf ( "Client: %s", recvBuf);
    }
    else
        perror ( "Client: recvfrom" );

    close ( s );
    return ( OK );
}

/*udpES：服务器函数，参数portn表示服务器侦听端口号*/
STATUS udpES ( int portn )
{
    int     s;                                          /*socket描述符*/
    struct sockaddr_in clientAddr, serverAddr;  /*端点地址*/
    int     addrLen = SA_LEN;
    char    recvBuf[ MSG_SIZE ];
    int     nRecv;

    /*创建服务器socket*/
    s = socket(AF_INET, SOCK_DGRAM, 0);
    if (s < 0)
    {
        perror("Server: opening datagram socket");
        return (ERROR);
    }

    /*初始化服务器的本地端点地址，与socket绑定*/
    bzero ((char *) &serverAddr, SA_LEN);
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_len  = SA_LEN;
    serverAddr.sin_port = htons (portn);
    serverAddr.sin_addr.s_addr = htonl (INADDR_ANY);
    if (bind (s, (struct sockaddr *) &serverAddr, SA_LEN) == ERROR)
```

```
    {
        perror ("server: bind failed.\n");
        close (s);
        return (ERROR);
    };

    printf ( "Server: port %d ready ... \n", portn );

    /*以循环方式处理客户请求*/
    while(TRUE)
    {
        /*接收客户请求*/
        nRecv = recvfrom (s, recvBuf, MSG_SIZE, 0,
                    (struct sockaddr *)&clientAddr, &addrLen );
        if ( nRecv < 0 )
        {
            perror ( "Server: recvfrom" );
            continue;
        }
        /*显示连接的客户名称*/
        printf ( "Server: the connected client ip=[%s], port=[%hd]\n",
          inet_ntoa(clientAddr.sin_addr), clientAddr.sin_port );
        /*回送收到的数据*/
        if ( sendto ( s, recvBuf, nRecv, 0, (struct sockaddr *)
                    &clientAddr, addrLen ) < 0 )
            perror("Server: sending datagram message");
    }
}
```

**注意：**

由于 UDP 不保证可靠传输，在运行时，即使服务器 udpES 没有运行，客户端 udpEC 的 sendto 仍然返回调用成功的结果。因此，上面的例子要求服务器任务先运行才能得到期待的输出，如下所示。

```
-> sp udpes, 5000
task spawned: id = 0xf7c980, name = t1
value = 16238976 = 0xf7c980
-> Server: port 5000 ready ...
```

```
sp udpec, "vxTarget", 5000
task spawned: id = 0xf77948, name = t2
value = 16218440 = 0xf77948
-> Server: the connected client ip=[210.43.106.252], port=[1024]
Server: done
Client: 1234567890
Client: the connected server name: ip=[210.43.106.252], port=[5000]
```

如果客户任务先运行，客户的sendto调用发送的数据不会被服务器收到，也就得不到服务器响应，从而客户任务在recvfrom接收服务器回应时陷入无限等待。这同时也是UDP应用的必须面临的一个难题，在现实网络环境中，UDP数据报可能丢失、重复、乱序到达。就上面的例子而言，要避免客户陷入无限等待，可以通过限时等待机制，如I/O复用（Select），采取这类机制涉及的问题则包括等待时间的确定（必须大于网络往返时间RTT加服务器处理开销），重发机制的确定，重复和乱序检测机制等。真正可靠的UDP应用相当于将TCP在传输层为了保证可靠而做的工作搬到应用层中实现。

实际上，系统在传输sendto指定的数据时，由于远程主机上服务器没有运行，远程主机将通过ICMP报告“端口不可达”错误，但是这一错误很晚才得到。我们称这一错误为**异步错误**，即用户程序中的动作已经完成了，然后底层系统才发现错误。依上面的例子设计的无连接应用中，系统不将异步错误告诉应用程序。这一点和下面要介绍的“连接的UDP socket”应用不同。

## 8.7.3 无连接socket和connect

可以对无连接socket调用connect，指明一个目的地端点地址，但是connect并不建立一个传输层连接，而是将目的地端点地址立即放在该socket的系统数据结构中。但是我们仍然称connect调用之后的socket为**连接的UDP socket**（加“UDP”以区别于8.7.2节中连接的TCP socket）。我们将看到，连接的UDP socket和未连接的UDP socket存在着一些差异。

例如，s代表本地UDP socket，现在将其“连接”到remoteAddr：

```
struct sockaddr_in  remoteAddr;
...
connect ( s, &remoteAddr, addrLen );
```

最明显的差异在于程序不再需要为每次发送和接收时指定对方端点地址，可以调用read/write和send/recv完成。实际上，对于连接后的UDP socket，继续通过sendto/recvfrom将得到EISCONN的错误结果。连接的UDP socket使得与多个远程端点的通信不可能，而只能固定与连接的端点通信。

另外一个差异在于对异步错误的处理。在8.7.2节给出的程序示例中，我们指出BSD

设计并没有考虑让用户程序得到异步错误结果。在这里，“让用户程序得到错误结果”有两种解释：（1）sendto 调用直接等到实际发送结果发生时才返回，返回值包括错误信息；（2）sendto 直接立即返回，但是系统随后收到 ICMP 报告错误时标记该端点地址不可达，使此后 recvfrom 接收该端点地址数据时告知没有连接。

对于不保证可靠传输的 UDP，让用户程序在 sendto 上阻塞等待结果是不可取的。而对于上述第（2）种解释，BSD 在未连接的 UDP 中没有实现，原因是很显然的：未连接的 UDP 可能需要和许多远程端点通信，如果依第（2）种解释告知应用程序异步错误，系统将需要维护许多标志信息，同时，对于未连接的 UDP socket，在 recvfrom 时告知“没有连接”也是矛盾的。而对于连接的 UDP socket，BSD 定义依上述第（2）种解释报告异步错误既是语义上一致的，同时也是实现上可行的。

现在总结 recvfrom 和 sendto 在连接的 UDP socket 和未连接的 UDP socket 上的执行情况，如表 8-14 所示：

表 8-14 执行情况

| | 连接的 UDP socket | 未连接的 UDP socket |
|---|---|---|
| sendto( ) | 立即送出 | 立即送出 |
| recvfrom( ) | 服务器未运行时：立即返回 ECONNREFUSED（不区分阻塞方式）<br>否则：接收服务器数据（区分阻塞式 socket 和非阻塞式 socket） | 不受服务器是否运行影响。根据 socket 阻塞特性决定立即返回或是阻塞 |

注：对服务器和客户的区别在于谁先调用 sendto 发送数据（此即客户），“服务器未运行”这一条件正是由于 sendto 之后远程通过 ICMP 回应端口不可达而得到的结果。

在 8.7.2 节给出的示例程序 udpEcho.c 的基础上，容易将客户端修改为连接的 UDP socket，只需要在 sendto 之前将创建的 socket 和服务器地址进行连接，并将 sendto 和 recvfrom 分别改为 send（或 write）和 recv（或 read）即可。下面给出修改后的不同部分：

```
/****修改之前的未连接 UDPsocket****/
    /*发送数据报到服务器*/
    if ( sendto ( s, MSG, MSG_SIZE, 0, (struct sockaddr *)
            &serverAddr, SA_LEN ) < 0 )
        perror("Client: sending datagram message");
    if ( (n=recvfrom (s, recvBuf, MSG_SIZE, 0, NULL, NULL)) >= 0 )

/****修改后连接的 UDP socket****/
    connect ( s, serverAddr, SA_LEN );
    if ( send ( s, MSG, MSG_SIZE, 0 ) < 0 )
        perror("Client: sending datagram message");
    if ( ( n=recv (s, recvBuf, MSG_SIZE, 0) ) >= 0 )
```

修改后，运行程序，如果服务器没有运行，程序将在 recv 调用时立即返回并得到错误信息“S_errno_ECONNREFUSED”。

和面向连接的 TCP socket 应用不同，因为 connect 完全只对本地有影响，所以对 UDP 服务器端也可以通过 connect 得到连接的 UDP socket。实践中确实存在这种情况，即服务器只需要和特定的客户进行 UDP 通信。我们在介绍 accept 时层指出 accept 无法筛选客户；而对于连接的 UDP socket，不论是客户还是服务器，都将只收到来自 connect 中指定的端点地址的数据，系统为指定发送到其他端点地址的 UDP 数据报利用 ICMP 返回端口不可达的结果。

## 8.7.4　多播的实现

**多播**或称**组播**（Multicast）技术提供一种一个发送者向一组接收者传送数据的有效传输方式。参加多播的主机称为一个**主机组**（Host Group），这些主机有一个共同的**主机组地址**（常称为**组地址**），而每个主机还拥有一个常规意义上的惟一的 IP 地址，这里该地址为**单播地址**。多播时，分组被发送到该主机组的组地址，而不是每个接收者的单播地址。发送者只发送一个数据复制，源端到目标端路径上的中间节点复制该数据。

多播的实现机制比较复杂，主机和网关使用 IGMP 协议来管理参与多播的数据报传送的计算机。协议规定 IPv4 多播的组地址为 D 类 IP 地址，即从 224.0.0.0 到 234.255.255.255（即 IP 地址开始 4 位为二进制 1110），其中，部分地址有特殊含义，如表 8-15 所示：

表 8-15　特殊地址

| | |
|---|---|
| 224.0.0.1 | 网段中所有支持多播的主机 |
| 224.0.0.2 | 网段中所有支持多播的路由器 |
| 224.0.0.4 | 网段中所有的 DVMRP 路由器 |
| 224.0.0.5 | 所有的 OSPF 路由器 |
| 224.0.0.6 | 所有的 OSPF 指派路由器 |
| 224.0.0.9 | 所有 RIPv2 路由器 |
| 224.0.0.13 | 所有 PIM 路由器 |

这些组地址称为**永久组地址**，不需要每次都建立组（这些组一直存在），例如网络中所有支持多播的路由器都属于组 224.0.0.2，都接收此目的 IP 的多播数据包。另外的**临时组地址**需要每次使用前创建主机组。“创建”主机组有多层含义，后面我们将看到，根据多播的范围，“创建主机组”可以只是本地主机行为（在一个子网内多播时），在主机系统 IP 协议层和网络接口部分申明要接收目的地为某个组地址的多播；或者需要跨越网关进行多播时（例如在广域网上进行多播），创建主机组还同时通知多播网关“我加入了某个组，组地址

为X.X.X.X”，这样来建立从其他网段到本网段主机组的路由，使远程网络上的的报文能够通过多播网关到达该组。这种区别完全是从必要性上讲的，实际上当主机上一个新加入一个组时，主机发送一个IGMP组员报告，表明自己是组“X.X.X.X”的成员；当主机上所有的进程都离开一个组后，主机便不再属于该组，发送一个离开组的IGMP报告。这样，如果多播网关收到一个组员报告的IGMP报文时，它将知道这里有一个主机组“X.X.X.X”，然后根据一定策略决定该主机组是否应该被其他网络的主机看见（即多播　　范围）。

假设主机2上一个接收任务要加入到一个主机组，以接收所有发往该组的多播，组地址为224.1.2.3。我们忽略一些复杂的细节，先来看在以太网上一个简单的多播的实现过程：

（1）接收任务请求加入组224.1.2.3；

（2）主机2上的系统记录要加入的主机组信息表（组地址和进程）；

（3）主机2按照图8-12中的方法计算224.1.2.3对应的硬件组地址为01:00:5e:01:02:03，告诉网卡“请接收硬件组地址为‘01:00:5e:01:02:03’的帧”；

（4）当主机1上一个任务（发送任务）要多播时，简单地将目的IP地址指定为目的组地址，端口为接收任务UDP端口。系统把该目的组地址转换为硬件组地址01:00:5e:01:02:03，发送到以太网上；

（5）硬件地址为01:00:5e:01:02:03的帧被主机2的网络接口接收。因为第（3）步中已经为网卡绑定了此硬件地址；

（6）网络接口层将该帧解开传递给IP，IP层查找主机组信息表，进一步判断是否发送到属于自己的主机组；如果不是，则丢弃，否则解开传递给UDP层；

（7）UDP判断指定端口5000上有任务，将其放入接收缓冲区。

上面的过程在图8-12中得到说明：

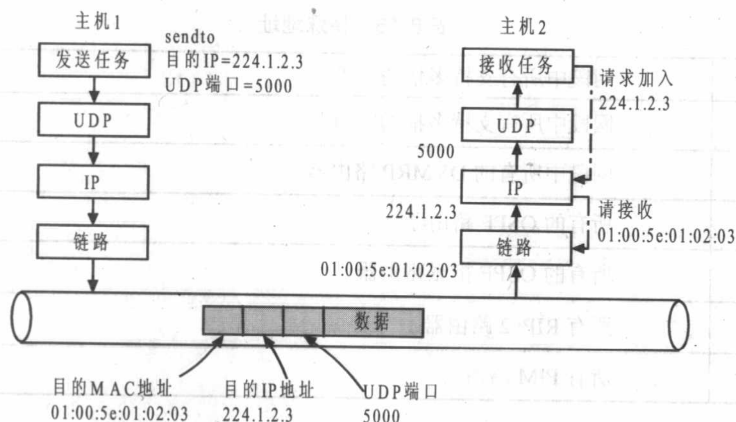



图8-12　一个简单的多播过程

可以看出，硬件支持在上述过程起了很大作用。如果硬件不支持，IP可能需要利用硬件广播来传输所有的组播数据包。在系统绑定一个硬件组地址后，网卡就会收下该地址的帧。硬件组地址的确定方法是将0x01005e000000和组地址的低23位相加得到，如图8-13所示：

上面的过程需要进行几点说明：

- 主机 2 新加入一个组时，主机通过 IGMP 向其他组成员及网关报告"我加入了组 224.1.2.3"，对网关而言，这表示收到此报告的网络接口上有一个组 224.1.2.3，会经过一个 IGMP 过程建立一个到此组的路由；

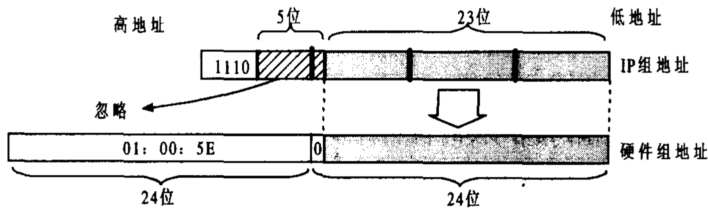



图 8-13　硬件组地址计算

- 接收时，主机进行了两级多播过滤：网卡进行的基于硬件组地址的过滤和 IP 协议层进行的基于 IP 组地址的过滤。在第 3 步，主机 2 指定了网卡接收硬件地址为 01:00:5e:01:02:03 的帧，对于网卡而言，这需要记录硬件组地址信息。由于上层协议可能要求网卡接收的硬件组地址非常多，而网卡只能记录有限的硬件组地址，因此，网卡通常将指定的硬件组地址经过散列运算后放在内部散列表中，当收到目的硬件地址为组地址的帧时，将收到的硬件地址执行同样的散列运算后与散列表中保存的结果相比较，如果相同则收下此帧。这样存在两种情况导致 IP 协议层收到不期望多播：（1）不同的硬件组地址散列运算后结果相同；（2）散列表溢出后网卡以混杂模式接收多播帧（即收下所有多播帧）。所以需要在 IP 协议层执行基于 IP 组地址的精确过滤；
- 在第 4 步，可以看出发送方的工作是非常简单的，只是发送任务指定的不是普通单播地址，而是一个组地址，另外一点差异是发送方不需要通过 ARP 确定 IP 组地址对应的硬件地址，而是计算出来。

RFC1112 对多播主机描述了 3 个级别，如表 8-16 所示：

**表 8-16　多播主机级别**

| | |
|---|---|
| 第 0 级 | 主机不能接收或发送多播 |
| 第 1 级 | 主机能发送但是不能接收多播。如上所述，发送方没有什么特殊要求，只是发送任务指定的是一个组地址并且要求系统识别该地址 |
| 第 2 级 | 主机能发送和接收多播 |

VxWorks 对多播的支持属于第 2 级，主机可以任意加入和退出一个或多个组，并且通过 IGMP 交换组员信息。同时，VxWorks 要求网卡必须支持多播，当为网卡绑定 IP 地址时，即为网卡指定了接收到所有主机组（224.0.0.1）的多播，以后每次主机新加入一个组时，又需要为网卡指定新的组地址。可以通过 ifShow( )调用来查看网卡是否支持多播，支持多

播的网卡输出信息中包括“MULTICAST”。我们不去深入讨论IGMP的细节，对于多播应用而言，使用UDP socket可以很容易地实现多播，面向连接的TCP无法实现多播。在VxWorks中，任务要进行多播只需要将UDP的目的IP地址指定为接收多播的组地址；接收多播的任务则多了加入指定组和离开指定组的调用。

### 1. 多播的范围

由于主机可以任意加入一个组，即在该子网内部创建了一个所指定组地址的主机组，所以，当该组的作用范围跨越网关时自然就涉及组地址惟一性的问题，也就是说要控制多播范围。对于只在本地子网内部的多播，不需要考虑这一问题。

IPv4没有专门定义多播的范围字段（区别于IPv6，专门定义4位表示多播范围），而事实上的做法是利用IP包首部的TTL字段。我们知道，TTL用于控制IP包在网络上的生存时间，每次经过一个网关时将TTL减1，TTL为0时IP包被丢弃。因此，TTL决定了多播能经过多少个多播网关。实际中控制多播范围是一个复杂的问题，涉及TTL，自治系统结构和网关的设置。系统配置多播网关时指定一个决定是否转发多播的TTL阈值，结合主机对多播分组TTL的设置，来控制多播的范围。当多播到达自治系统边界时，边界网关应该禁止对属于自治系统内部的多播进行转发。

系统提供接口允许指定此主机上多播分组的TTL。

### 2. 实现多播的系统调用

实现多播通过setsockopt设置多播选项来实现。在8.9节“socket应用高级话题”中将介绍该函数的用法。多播选项包括下面几个，如表8-17所示：

表8-17 多播选项

| 多播选项（optname） | 功　能 |
|---|---|
| IP_ADD_MEMBERSHIP | 加入主机组 |
| IP_DROP_MEMBERSHIP | 离开主机组 |
| IP_MULTICAST_IF | 指定多播网络接口 |
| IP_MULTICAST_TTL | 指定多播分组的TTL |
| IP_MULTICAST_LOOP | 指定是否环回 |

- 加入和离开多播组

当optname是IP_ADD_MEMBERSHIP时，表明调用任务要加入一个多播组，例如：

```
setsockopt ( s, IPPROTO_IP, IP_ADD_MEMBERSHIP, (char *)&ipMreq,
          sizeof (ipMreq));
```

参数ipMreq定义为结构体ip_mreq，该结构体表明任务要加入的组地址和主机单播地址，格式为结构体in_addr表示的二进制IP地址。结构体ip_mreq定义为：

```
struct ip_mreq {
    struct in_addr imr_multiaddr;    /*组地址*/
    struct in_addr imr_interface;    /*网络接口的单播地址*/
};
```

必须指明 imr_multiaddr 为一个有效的组地址；单播地址 imr_interface 可以显式指定，也可以让系统自动选择（指定 imr_interface=INADDR_ANY）。

如图 8-14 所示，可以在一个 socket 上多次加入组，(a）通过不同的本地网络接口加入不同的组；(b）通过不同的网络接口加入相同的主机组；(c）通过同一个网络接口加入不同的主机组。在某个网络接口上加入一个主机组意味着到达该网络接口的目的地址为该主机组的多播可以被 socket 接收。图 8-14 中不涉及 UDP 端口信息，加入组和离开组只涉及本地 IP 和组地址。可以将上述“网络接口－加入的组”看成一个特殊的 IP，“特殊”在于到达该接口的 IP 分组包括多播分组。除此以外，UDP 处理部分和单播是一致的：检查 UDP 端口号决定是否提交给任务或者将其丢弃（即端口不可达）。

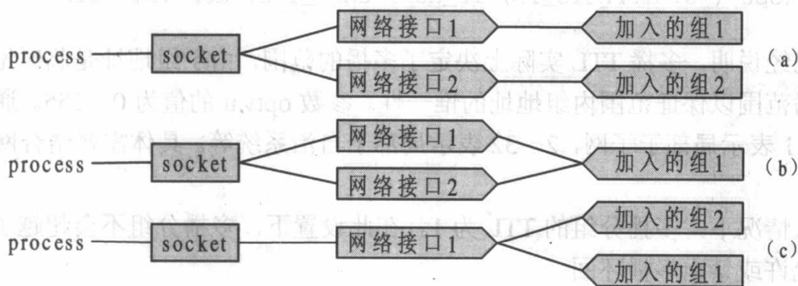



图 8-14 网络接口和主机组

当任务不需要接收多播时，可以通过 optname 为 IP_DROP_MEMBERSHIP 的调用离开一个组。如果 socket 被关闭，则该 socket 所加入的组也将自动离开这些组。

```
setsockopt ( s, IPPROTO_IP, IP_DROP_MEMBERSHIP, (char *)&ipMreq,
        sizeof (ipMreq));
```

参数 ipMreq 也表示结构体 ip_mreq，上述调用将使 socket 在 ipMreq.imr_interface 上离开加入的组 ipMreq.imr_multiaddr。如果没有指定网络接口的单播地址，则在第一个匹配的网络接口上离开组，例如图 8-14（b）将使 socket 在“网络接口 1”上离开组。

系统内部为组和加入该组的任务维护着关联信息。“组—加入该组的任务”之间是多对多的关系。当第一次有一个任务加入一个组时，主机加入该组；最后一个任务离开一个组时，主机离开该组。主机加入和离开一个组将根据 IGMP 协议执行组成员信息报告的过程。

- 指定多播的网络接口

当 optname 是 IP_MULTICAST_IF 时，传递的参数 ifAddr 包含了多播时的单播地址，

即在这个插口上发送的多播时使用该地址指定的特定接口。不指定时系统自动根据路由为送出的多播选择网络接口。

```
struct in_addr ifAddr;
...
setsockopt ( s, IPPROTO_IP, IP_MULTICAST_IF, (char *)&ifAddr,
             sizeof (ifAddr));
```

参数 ifAddr 表示本地网络接口 IP 地址。

IP_MULTICAST_IF 选项和前述 IP_ADD_MEMBERSHIP 不同在于：前者控制发送多播时选择哪个网络接口；后者控制在哪个网络接口上接收哪个组地址的多播。

- 指定多播分组 TTL（分组存活时间）

当 optname 是 IP_MULTICAST_TTL 时，传递的参数 optval 指定送出的多播分组 TTL。

```
int optval;   /*optval: 0~255*/
setsockopt ( s, IPPROTO_IP, IP_MULTICAST_TTL, &optval, sizeof(optval));
```

前面已经说明，多播 TTL 实际上决定了多播的范围，由于组地址是临时地址，因此必须控制多播范围以保证范围内组地址的惟一性。参数 optval 的值为 0～255。通常 0 表示局部于主机，1 表示局部于子网，2～32 表示局部于自治系统等。具体需要结合网络结构分析多播范围。

在默认情况下，多播分组的 TTL 为 1。在此设置下，多播分组不会超越子网范围。

- 允许或禁止多播环回

当 optname 是 IP_MULTICAST_LOOP 时，表示送出的多播分组是否环回。默认时是环回的，即主机送出的所有多播都将被接口收到，如果此时发送多播的任务还属于发送时指定的组，该任务将收到自己发送的数据。

```
int optval;   /*optval: 1=允许环回 0=禁止环回*/
setsockopt ( s, IPPROTO_IP, IP_MULTICAST_LOOP, &optval, sizeof(optval));
```

上述 5 个多播选项中，前面两个用于接收多播任务（必需的），后面 3 个选项用于发送多播的任务（非必需的）。以下给出实现多播的简单示例。在该示例中，假定默认设置适合多播发送。如果需要修改发送方多播选项，只需要在 sendto 前加上合适的设置即可。

```
/*** 发送多播代码段（主要差别在于指定接收方地址为组地址）***/

sockFd = socket ( ... ); /*创建 socket 及 bind 绑定本地端点地址和单播是一样的*/
bind ( sockFd, ... )     /*bind 绑定本地端点地址（不一定需要）*/

/*初始化接收方地址 toAddr（组地址）*/
bzero ((char *)&toAddr, sizeof (toAddr));
```

```
toAddrLen = sizeof(struct sockaddr_in);
toAddr.sin_len = (u_char) toAddrLen;
toAddr.sin_family = AF_INET;
toAddr.sin_addr.s_addr = inet_addr (mcastAddr);  /*mcastAddr 为组地址*/
toAddr.sin_port = htons(mcastPort); /*mcastPort 为 UDP 端口号，所有接收*/
                                    /*多播的任务必须在该端口接收多播*/
sendto (sockFd, ..., (struct sockaddr *)&toAddr, toAddrLen));
...

/*** 多播接收方代码段（加入组－接收－离开组）***/

sockFd = socket ( ... ); /*创建 socket 及 bind 绑定本地端点地址和单播是一样的*/
bind ( sockFd, ... )    /*bind 绑定本地端点地址（不一定需要）*/

/*加入多播的主机组*/
/*  填写 ipMreq 数据结构：组地址和本地网络接口地址 */
/*  下面设置要接收网络接口 ifAddr 上收到的 mcastAddr 为组地址的多播*/
ipMreq.imr_multiaddr.s_addr = inet_addr (mcastAddr);
ipMreq.imr_interface.s_addr = inet_addr (ifAddr);
if ( setsockopt (sockFd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
        (char *)&ipMreq, sizeof (ipMreq)) < 0)
{
    ... /*setsockopt：加入组错误！*/
}

...  /*从 sockFd 接收数据，例如 read( sockFd, ... )*/

/*不再需要接收多播时离开组*/
if (setsockopt (sockFd, IPPROTO_IP, IP_DROP_MEMBERSHIP,
        (char *)&ipMreq, sizeof (ipMreq)) < 0)
{
    ... /*setsockopt：离开组错误！*/
}
```

### 8.7.4 广播的实现

广播和多播都用于实现向多个接收者发送 UDP 数据报，但是广播不像多播那样在接收端有复杂的控制过程，因而实现比多播简单的多，只需要允许 socket 的广播选项（默认情

况下广播被禁止)，然后在发送时指定目的 IP 为广播地址即可。

具体地，允许 socket 广播通过 setsockopt 设置广播选项来实现。当 setsockopt 参数 optname 为 SO_BROADCAST 时，表示打开或禁止从该 socket 广播：

```
int  optval = 1;    /*optval: 1=允许广播 0=禁止广播*/
setsockopt ( s, SOL_SOCKET, SO_BROADCAST, &optval, sizeof (optval));
```

参数 optval 为 1 表示启用广播，或者为 0 表示不启用广播。如果没有启用广播，则任务发送到广播地址时 sendto 将报告错误 EACCES。

下面的函数 bcast( )表明了一个广播的过程，该函数将参数 msg 表示的消息广播到网络上，msglen 表示消息长度，portn 表示广播出去的数据报的 UDP 端口。

setsockopt 允许广播之后，发送广播只需要简单地将目的端点地址指定为广播地址 INADDR_BROADCAST（即 255.255.255.255)。注意系统通过一个特定的网络接口广播出去时，并不是直接指定目的 IP 地址为 255.255.255.255，而是该网络接口 IP 地址的主机 ID 部分设置为全 1，网络 ID 部分不变。广播地址“255.255.255.255”称为受限的广播地址，将不会被网关转发。在我们的例子中，没有通过 bind 调用绑定本地端点地址，系统将自动选择一个合适的网络接口。如果存在多个网络接口，则一般应该在发送广播之前先绑定一个网络接口。

```
STATUS bcast ( int portn, char* msg, int msglen )
{
    int     s;
    struct sockaddr_in bcastAddr;
    int     bcastFlag = 1;

    if ( (s = socket(AF_INET, SOCK_DGRAM, 0)) < 0 ) /*创建 UDP socket*/
    {
        perror("bcast: socket");
        close("s");
        return ERROR;
    }

    /*允许该 socket 发送广播*/
    if ( setsockopt (s, SOL_SOCKET, SO_BROADCAST, &bcastFlag,
                sizeof (bcastFlag)) == -1 )
    {
        perror("bcast: setsockopt");
        close("s");
        return ERROR;
```

```
    }

    bzero ((char *) & bcastAddr, SA_LEN);          /*初始化广播端点地址*/
    bcastAddr.sin_family = AF_INET;
    bcastAddr.sin_len = SA_LEN;
    bcastAddr.sin_port = htons (portn);
    bcastAddr.sin_addr.s_addr = htonl (INADDR_BROADCAST);
    if ( sendto ( s, msg, msglen, 0, (struct sockaddr *)
              & bcastAddr, SA_LEN) < 0 )           /*发送广播*/
       perror("bcast: broadcast ");

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

需要注意的是，取决于所使用的版本，接收方可能需要绑定本地端点地址到通配地址INADDR_ANY才能接收广播。在VxWorks系统内部，基于BSD 4.4的udp_input有类似下面的处理过程（udp_input用于IP协议层将UDP数据报提交UDP协议处理）：

```
if (in_broadcast(ip->ip_dst, m->m_pkthdr.rcvif))
{
    struct socket *last;
    /*
    * Deliver a multicast or broadcast datagram to *all
    * sockets for which the local and remote addresses
    * and ports match those of the incoming datagram.
    ...
    if (inp->inp_lport != uh->uh_dport)
        continue;
    if (inp->inp_laddr.s_addr != INADDR_ANY)
    {
        if (inp->inp_laddr.s_addr != ip->ip_dst.s_addr)
            continue;
    }
    if (inp->inp_faddr.s_addr != INADDR_ANY)
    {
        if (inp->inp_faddr.s_addr != ip->ip_src.s_addr ||
                inp->inp_fport != uh->uh_sport)
```

```
        continue;
    }
    ...
```

如果将 socket 绑定到非 INADDR_ANY 的 IP 地址，将无法收到广播，并且没有任何错误指示。

## 8.8 裸层 socket

裸层 socket 直接构建在 IP 层之上，不经过 TCP/IP 的传输层，允许发送方直接从 IP 协议层开始数据结构；接收方可以在 IP 的上层接收除部分限制处理方式以外的 IP 分组。我们将在后文中分析裸层 socket 可以得到哪些分组。

实际上，不只是一般用户程序，许多现有协议和流行工具程序都是通过裸层 socket 实现的，如路由跟踪 trouceroute 程序和 OSPF 路由协议的实现等。甚至可以在裸层 socket 上实现新的传输层协议。

创建过程和其他 socket 是一样的，也是通过 socket( )调用实现：

```
s = socket ( AF_INET, SOCK_RAW, protocol );
```

创建裸层 socket 指定类型为 SOCK_RAW，protocol 表示该 socket 的协议类型。从该 socket 接收时，系统只将 IP 分组的头部结构中的协议字段和 protocol 参数相同的分组给该 socket（0 表示接收任何协议类型的分组）；同样地，发送时，参数 protocol 将被写入发出 IP 分组头部的协议字段。

裸层 socket 的使用和 TCP socket 以及 UDP socket 大致差不多，使用相同的系统调用实现，一般包括创建 socket、发送、接收、关闭等。但是对比 TCP 和 UDP socket，存在下列明显差异：

- 最明显的差别在于裸层 socket 发送时通常涉及某种协议数据结构的构造，如 ICMP/UDP/TCP 报文，接收时需要对协议数据结构进行分析（当然也可以不涉及任何标准的 TCP/IP 协议内容，直接在裸层 socket 上传递应用程序的数据结构）；
- 裸层 socket 同样具有本地端点地址和远程端点地址属性。我们前文指出，端点地址包括端点 IP 地址和端口号。但是对于裸层 socket 端点地址只有 IP 地址有意义而没有端口号的概念。因此对裸层 socket 调用 bind( )将绑定本地 IP，调用 connect( )将指定远程 IP 地址。

和连接的 UDP socket 一样，称 connect( )调用后的裸层 socket 为连接的裸层 socket，显然裸层 socket 不具有 TCP socket 那种意义上的连接。

对裸层 socket 的发送和接收根据其是否已连接有不同，如表 8-18 所示：

表 8-18 发送/接收报文

| | 发 送 报 文 | 接 收 报 文 |
|---|---|---|
| 未连接 | sendto( ) sendmsg( ) | recvfrom( ) recvmsg( ) |
| 已连接 | write( ) send( ) | read( ) recv( ) |

可以看出裸层 socket 的输入输出和 UDP socket 输入输出相似。

在具体讨论如何通过裸层 socket 发送和接收之前，先简单介绍一下应用中可能会涉及的一些报文格式。

## 8.8.1 报文格式

裸层 socket 编程的一个特点是用户程序涉及了底层协议数据格式。具体来说，取决于所做的选择，用户程序可能需要填写 IP 分组的头部，或者对其他 IP 层协议数据（如 ICMP 和 IGMP）的报文结构进行解码。另外，系统不会将 TCP 和 UDP 的分组递交给一个裸层 socket，但是特殊场合可能需要裸层 socket 构造一个 TCP/UDP 分组发送。

### 1. IP 头部结构

IP 头部结构 struct ip { … }在头文件 netinet/ip.h 中定义，如图 8-15 所示。图中直接用 struct ip 中定义的字段名称表示，小括号内为字段位数（Bit）。在该头部结构之后即跟数据字段，可以是传输层协议数据（TCP 和 UDP 报文），或者其他协议数据（ICMP，IGMP，OSPF），或者用户程序指定的数据。

<table>
<tr><td>ip_v(4)</td><td>ip_hl(4)</td><td>ip_tos(8)</td><td>ip_len(16) 总长度</td></tr>
<tr><td colspan="3">ip_id(16) 标识</td><td>ip_off(16) 分片标志及片偏移量</td></tr>
<tr><td colspan="2">ip_ttl(8)</td><td>ip_p(8)</td><td>ip_sum(16) 校验和</td></tr>
<tr><td colspan="4">ip_src(32) 源IP地址</td></tr>
<tr><td colspan="4">ip_dst(32) 目的IP地址</td></tr>
<tr><td colspan="4">可选选项字段，32位对齐</td></tr>
</table>

图 8-15 IP 头部结构

其中：

- ip_v 版本号，对于 IPv4，该字段即等于 4（VxWorks 目前只支持该值）；
- ip_hl 表示 IP 头部结构长度，包括图 8-15 中所有字段，以 4 字节为单位；
- ip_tos 服务类型，是一个标志字段，可以是这样一些标志的组合：优化延迟 IPTOS_LOWDELAY，优化吞吐率 IPTOS_THROUGHPUT，优化可靠性

IPTOS_RELIABILITY，优化成本 IPTOS_MINCOST（这些选项在 netinet/ip.h 定义）；

- ip_len 表示含头部和头部之后的数据总长度，以字节为单位。分片时表示所有分片长度之和；
- ip_id 用于惟一标识每个分组，主要使分片后的分组能重新组装，从 Berkeley 派生的系统每送出一个 IP 分组后由 IP 协议层将此字段加 1。为裸层 socket 填写送出的 IP 分组的头部结构时可将该字段留给系统填写；
- ip_off 分片标志（高 3 位）及当前分片在分片前的数据字段内的偏移量（低 13 位）。偏移量以 8 字节为单位；
- ip_ttl 分组在网络中的生命周期（TTL），实践中往往用来控制经过多少次路由器转发。标准建议值为 32，VxWorks 默认为 64；
- ip_p 头部结构之后的数据属于何种协议的数据，netinet/in.h 给出了标准的协议定义 IPPROTO_*XXX*，如 TCP 为 IPPROTO_TCP，ICMP 为 IPPROTO_ICMP；
- ip_sum 对 IP 首部的校验和，不含数据部分（为裸层 socket 填写 IP 头部结构时可将该字段留给系统填写）；
- ip_src 和 ip_dst 为源和目的 IP 地址。

## 2. ICMP 头部结构

ICMP 报文是 IP 分组的负载。其头部结构 struct icmp{ … }在 netinet/ip_icmp.h 中定义，如图 8-16 所示。

| icmp_type(8) | icmp_code(8) | icmp_cksum(16) 校验和 |
|---|---|---|
| 依赖于icmp_type和icmp_code的内容 | | |

图 8-16 ICMP 头部结构

ICMP 头部结构中包括的字段比 IP 简单，但其用法和实现的功能比 IP 要复杂。其中：icmp_type 表示 ICMP 报文的类型；icmp_code 表示进一步区分某些报文的子类型代码；icmp_cksum 表示整个 ICMP 头部的校验和。

如表 8-19 所示 ICMP 报文类型（它们在 netinet/ip_icmp.h 中定义）。

表 8-19 ICMP 报文类型说明

| icmp_type 和 icmp_code | 报 文 说 明 |
|---|---|
| ICMP_ECHO | ECHO 请求 |
| ICMP_UNREACH | 目标不可达，有以下子类型 |
| ICMP_UNREACH_NET | 网络不可达 |
| ICMP_UNREACH_HOST | 主机不可达 |
| ICMP_UNREACH_PROTOCOL | 目的主机上协议不能用 |

续表

| icmp_type 和 icmp_code | 报 文 说 明 |
|---|---|
| ICMP_UNREACH_PORT | 目的主机上端口没有被激活 |
| ICMP_UNREACH_NEEDFRAG | 需要分片并设置 DF 比特 |
| ICMP_UNREACH_SRCFAIL | 源路由失败 |
| ICMP_UNREACH_NET_UNKNOWN | 未知的目的网络 |
| ICMP_UNREACH_HOST_UNKNOWN | 未知的目的主机 |
| ICMP_UNREACH_ISOLATED | 源主机被隔离 |
| ICMP_UNREACH_NET_PROHIB | 从管理上禁止与目的网络通信 |
| ICMP_UNREACH_HOST_PROHIB | 从管理上禁止与目的主机通信 |
| ICMP_UNREACH_TOSNET | 对服务类型，网络不可达 |
| ICMP_UNREACH_TOSHOST | 对服务类型，主机不可达 |
| ICMP_SOURCEQUENCH | 源抑制报文，要求放慢发送 |
| ICMP_REDIRECT | 有更好的路由，有以下子类型 |
| ICMP_REDIRECT_NET | 网络有更好的路由 |
| ICMP_REDIRECT_HOST | 主机有更好的路由 |
| ICMP_REDIRECT_TOSNET | TOS 和网络有更好的路由 |
| ICMP_REDIRECT_TOSHOST | TOS 和主机有更好的路由 |
| ICMP_ECHOREPLY | ECHO 应答 |
| ICMP_ROUTERADVERT | 路由器通告 |
| ICMP_ROUTERSOLICIT | 路由器请求 |
| ICMP_TIMXCEED | 超时，有以下子类型 |
| ICMP_TIMXCEED_INTRANS | 传送过程中 IP 生存期到期 |
| ICMP_TIMXCEED_REASS | 重装生存期到期 |
| ICMP_PARAMPROB | 首部的问题，有以下子类型 |
| ICMP_PARAMPROB_OPTABSENT | 丢失需要的选项 |
| ICMP_TSTAMP | 时间戳请求 |
| ICMP_TSTAMPREPLY | 时间戳应答 |
| ICMP_IREQ | 信息请求 |
| ICMP_IREQREPLY | 信息应答 |
| ICMP_MASKREQ | 地址掩码请求 |
| ICMP_MASKREPLY | 地址掩码应答 |

### 3. IGMP 头部结构

IGMP 报文头部结构 struct igmp { … }在 netinet/igmp.h 中定义，如图 8-17 所示。

| igmp_type(8) | igmp_code(8) | igmp_cksum(16) 校验和 |
| --- | --- | --- |
| igmp_group 组地址 | | |

图 8-17 IGMP 头部结构

其中：

- igmp_type 表示 IGMP 版本及报文类型。如用于组员关系维护的报文：多播网关查询报文 IGMP_MEMBERSHIP _QUERY，组员报告报文 IGMP_V1 _MEMBERSHIP_REPORT，IGMP_V2_MEMBERSHIP_REPORT 和 IGMP_V2_ LEAVE_GROUP；还有其他路由信息报文；
- igmp_code 代表子类型；
- igmp_cksum 表示校验和（和 ICMP 中校验和类似）。

## 4. TCP 头部结构

TCP 报文头部结构 struct tcphdr { … }在 netinet/tcp.h 中定义，如图 8-18 所示。

| th_sport(16) 源TCP端口 | | | th_dport(16) 目的TCP端口 |
| --- | --- | --- | --- |
| th_seq(32) 序号 | | | |
| th_ack(32) 确认序号 | | | |
| th_off(4) | th_x2(4) | th_flags(8) | th_win(16) 窗口大小 |
| th_sum(16) 校验和 | | | th_urp(16) 紧急指针 |
| 可选选项字段，32位对齐 | | | |

图 8-18 TCP 头部结构

其中：

- th_sport 和 th_dport 表示源和目的端的 TCP 端口号，用以提供端到端的 TCP 可靠数据流服务；
- th_seq 表示本报文数据字段的第 1 字节在整个 TCP 数据流中的相对位置；
- th_ack 表示确认对方数据流的序号，即下次期望接收该序号开始的数据流；
- th_off 表示数据偏移，即该报文的 TCP 头部长度，以 4 字节为单位；
- th_flags 表示这样一些控制标志： TH_FIN，TH_SYN，TH_RST，TH_PUSH，TH_ACK，TH_URG；
- th_win 表示发送方通知对方的自己的接收窗口大小；
- th_sum 表示校验和，是根据整个 TCP 头部和后面的数据字段内容并加上一个 12 字节的“伪头部”计算而来。

其中，用于计算 TCP 校验和的伪头部如图 8-19 所示，图中的 6 表示 TCP 协议号。该

头部在计算 th_sum 时加在 TCP 头部前面，但只用在发送方和接收方的计算过程中，并不被实际传输。

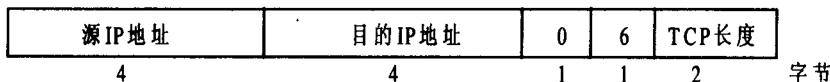
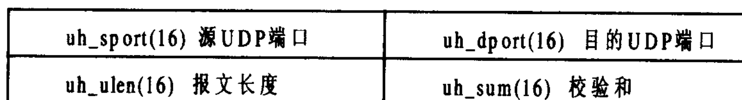



图 8-19　TCP 伪头部

### 5. UDP 头部结构

UDP 报文头部结构 struct udphdr { … }在 netinet/udp.h 中定义，如图 8-20 所示。



| uh_sport(16) 源UDP端口 | uh_dport(16) 目的UDP端口 |
|---|---|
| uh_ulen(16) 报文长度 | uh_sum(16) 校验和 |

图 8-20　UDP 头部结构

UDP 报文结构比 TCP 简单得多。UDP 报文没有流的概念，因此也不含任何序号和控制选项。UDP 通过 uh_ulen 指示报文长度，含 UDP 头部本身和数据字段。和 TCP 的校验和字段一样，uh_sum 也是根据整个 UDP 头部和后面的数据字段内容并加上一个"伪头部"计算而来。UDP 的伪头部只是将图 8-19 中的 6 改为 17 表示 UDP 协议，同时 TCP 长度改为 UDP 报文的长度。

**校验和计算**

在上面的每种协议头部结构都包含 16 位校验和字段，不同的协议下校验和计算范围不同。在不同的处理器体系下都可以找到 UNIX 下的校验和算法源程序 sys/i386/i386/in_cksum.c。生成校验和的算法过程是：将待校验缓冲区看成由 16 位的字组成的串（该缓冲区包括 16 位的校验和字段本身，被清 0），然后逐字将其反码累加，最后的结果取反后填入校验和字段。在计算前，如果待校验缓冲区不是 16 位对齐的，需要在后面补 0 对齐 16 位。验证校验和的过程是将待验证的缓冲区执行同样的逐 16 位字进行反码累加过程，最后应该得到 16 位全 1 的校验码。

## 8.8.2　发送和接收

### 1. 裸层 socket 的接收

系统是否将收到的一个 IP 分组递给某个裸层 socket，需要考虑下列不同情况：

- 类型为 TCP 和 UDP 的分组（即 IP 首部的协议类型字段为）：系统处理，不提供给任何裸层 socket;

- ICMP 分组：回射请求，时间戳请求，地址掩码请求全部由系统处理，其他分组可以被某裸层 socket 得到；
- IGMP 分组：首先由系统处理，然后可以由某裸层 socket 得到；
- 内核不能失败协议字段的 IP 分组都可以被某个裸层 socket 得到。但是这些分组对 IP 层将是有效的，即系统会检查 IP 头部这些字段是否有效：IP 版本，头部校验和，头部长度，目的 IP 地址。系统不会将无限的 IP 分组递给裸层 socket;
- 如果 IP 分组被分片，系统直到所有分片到达并整理后递给裸层 socket。

上面“可以”的意思是系统需要进一步判断谁将得到该 IP 分组，因为一个系统中可能同时有多个裸层 socket。具体来说系统根据裸层 socket 的协议号、本地地址、远程地址来判断。如果一个裸层 socket 满足下列 3 个条件，则该 socket 将得到该 IP 分组；如果同时有多个裸层 socket 满足这些条件，则每个 socket 都得到该 IP 分组的一个备份。

- 裸层 socket 的协议号和 IP 分组的协议号字段相同。但是如果裸层 socket 的协议号为 0，则直接视该条件为满足；
- 裸层 socket 绑定的本地 IP 地址和 IP 分组中的目的 IP 地址相匹配。但是如果裸层 socket 没有绑定本地 IP 地址，则直接视该条件为满足；
- 裸层 socket 通过 connect 调用连接的远程 IP 地址和 IP 分组中的源 IP 地址匹配。但是如果裸层 socket 没有进行连接，则直接视该条件为满足；

注意：

- 裸层 socket 得到的将是整个 IP 分组，包含 IP 头部信息；
- 从裸层 socket 收到的 IP 分组中，ip_len，ip_off，ip_id 字段是主机字节序，其他字段是网络字节序的。

具体地，程序从裸层 socket 接收时，根据 socket 是否已连接，可以调用 read( )/recv( ) 或者 recvfrom( )实现，例如：

```
recv ( s, buf, buflen, flags );                          /*s 为已连接的裸层 socket*/
recvfrom(s, buf, buflen, flags, fromAddr, addrLen);/*s 为未连接的裸层 socket*/
```

从上面 recvfrom 返回后，参数 fromAddr 中只有 IP 地址有意义。

## 2. 裸层 socket 输出

根据裸层 socket 是否已连接，可以调用 send( )/write( )或者 sendto( )实现裸层 socket 输出，例如：

```
Send ( s, buf, buflen, flags );                          /*s 为已连接的裸层 socket*/
sendto ( s, buf, buflen, flags, toAddr, addrLen ); /*s 为未连接的裸层 socket*/
```

对于上面 sendto 中的参数 toAddr，系统只关心其中的 IP 地址，而忽略端口号。

在上面的函数中涉及的一个问题就是参数 buf 表示的缓冲区是否包括 IP 头部数据结

构。可以让系统生成 IP 分组的头部，也可以让用户程序填写其中绝大部分字段。具体通过 socket 选项 IP_HDRINCL 控制。

```
int optval;   /*optval: 1=用户程序填写 IP 头部 0=系统自动填写 IP 头部*/
setsockopt(raw,0,IP_HDRINCL,(char *)&optval,sizeof(optval));
```

两种情况的细节如表 8-20 所示。

**表 8-20 填写 IP 头部结构的两种情况**

| | |
|---|---|
| 用户程序填写 IP 头部结构 | 用户程序指定的发送数据缓冲区 buf 包括整个 IP 头部。但是其中的标识字段 ip_id 和头部校验和字段 ip_sum 写为 0，系统自动填写。同时用户提供的数据大小 buflen 包括 IP 头部的字节数在内 |
| 系统自动填写 IP 头部结构 | 用户程序指定的发送数据缓冲区 buf 不包括 IP 头部，系统自动在前面生成 IP 头部，其中的协议字段 ip_p 为裸层 socket 创建时指定的协议 |

## 8.8.3 示例：Traceroute

Traceroute 是一个著名的网络工具，它能够检测出到某个目的 IP 地址所经过的路由。Traceoute 检测过程是建立在裸层 socket 之上的，本小节将通过分析 Traceroute 来说明裸层 socket 编程方法。

Traceroute 利用了路由器和主机的两个属性：

- 路由器属性：路由器转发分组之前，先将分组的 TTL 减 1，如果 TTL 为 0，路由器不转发该分组，而是将该分组丢弃，并给发出此分组的主机发送一个 ICMP 超时报文，该报文中，源 IP 地址为路由器本身的 IP 地址；
- 主机属性：主机的传输层协议在收到一个目的端口号上没有应用的数据报时会给发出此报文的主机发送一个 ICMP 端口不可达报文。

具体地，探测主机利用上述属性探测的过程是：探测主机发送一个 TTL 字段为 1 的 IP 分组给目的主机；收到该分组的第一个路由器将 TTL 减 1，发现 TTL 为 0，于是丢弃该分组，并给探测主机回送一个 ICMP 超时报文；探测主机收到 ICMP 超时报文之后，便知道了路由上的第一个路由器，然后向目的主机发送一个 TTL 字段为 2 的分组，该分组被第一个路由器转发到第 2 个路由器之后，第 2 个路由器同样得到 TTL 为 0 的结果，于是也给探测主机回送一个 ICMP 超时报文；这样，探测主机不断重复下去，直到分组到达目的主机，就得到了该路由上的所有路由器地址。

上述过程中，得到路由器 IP 地址利用了路由器属性。使探测过程结束，关键的一点在于探测主机如何知道分组到达了目的主机。这一点 Traceroute 利用了上述主机属性。即在探测过程中，探测主机送出的每一个 IP 分组的载荷都是一个 UDP 数据报，该数据报巧妙地选择了一个很少可能有任务在使用的端口（大于 30000），因此主机会回送一个 ICMP 端

口不可达报文。这样，收到此报文后，探测主机即知道了完整的路由，可以结束探测过程了。

总结上述过程，Traceroute 程序所要做的就是不断（以某种时间间隔）发送 TTL 从 1 开始逐步增大的 UDP 数据报，该数据报的 UDP 端口号是一个不常见的端口，然后通过区分的收到的是 ICMP 超时报文还是 ICMP 端口不可达报文，来判断路由追踪是否成功。

Traceroute 的源程序可以公开获得，不同的版本基本上都有大约 1000 行左右。下面给出移植到 VxWorks 后简化版本，约 200 行。需要使用的和 socket 编程有关的头文件包括：sockLib.h，netinet/in_systm.h，netinet/in.h，netinet/ip.h，netinet/ip_icmp.h，netinet/udp.h。程序包括两个任务：

- 主任务（traceRouteMain 函数）：创建接收任务，循环发送 UDP 探测数据报，检查接收任务以退出。

接收任务（troute_icmpMsgReceiver 函数）：在一个无限循环接收 UDP 数据报，根据收到的数据报设置到达目的主机或者不可达标志使主任务退出。

Traceroute 使用的数据结构和全局变量定义：

```
struct opacket {
    struct ip iphdr;
    struct udphdr udp;
    u_char seq;          /* sequence number of this packet */
    u_char ttl;          /* ttl packet left with */
    struct timeval tv;   /* time packet left */
};
struct opacket  *outpacket;     /* last output (udp) packet */
u_char  packet[512];           /* last inbound (icmp) packet */
#define DELTATIME(t1p, t2p) ( \
        (long) ( (t2p)->tv_sec - (t1p)->tv_sec ) * 1000 + \
        (long) ( (t2p)->tv_nsec - (t1p)->tv_nsec ) / 1000 \
    )
void ERR_QUIT(char * s, int n) { perror ( s );  exit ( n ); }

int    sndsock;                    /* send (udp) socket file descriptor */
struct sockaddr_in whereto;          /* Who to try to reach */
int    datalen;                    /* How much data */
int    nprobes = 3;
int    max_ttl = 30;
u_short ident;
u_short port = 32768+666;       /* start udp dest port # for probe packets */
int waittime = 2;               /* time to wait for response (in seconds) */
```

```
int seq;                  /*sequnece number*/
int got_there = 0;        /*if have reached destination host */
int unreachable = 0;      /*destination unreachable counter*/
```

## 1. 主任务

```
int traceRouteMain ( char *hostaddr )
{
    int   probe, ttl, on = 1, icmpTid = 0;

    seq    = 0;
    ident   = ( taskIdSelf( ) & 0xffff ) | 0x8000;
    datalen = 56 + sizeof(struct opacket);

    bzero((char *)&whereto, sizeof(struct sockaddr));
    whereto.sin_family = AF_INET;
    if ( inet_aton(hostaddr, &whereto.sin_addr) != 0)
        ERR_QUIT("invalid IP address", 1);

    outpacket = (struct opacket *)malloc((unsigned)datalen);
    if (! outpacket)
        ERR_QUIT("traceroute: malloc", 1);

    bzero((char *)outpacket, datalen);
    outpacket->iphdr.ip_dst = whereto.sin_addr;
    outpacket->iphdr.ip_tos = 0;
    outpacket->iphdr.ip_v = IPVERSION;
    outpacket->iphdr.ip_id = 0;

    if ((sndsock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)) < 0)
        ERR_QUIT ("traceroute: raw socket", 5);
    if (setsockopt(sndsock, IPPROTO_IP, IP_HDRINCL, (char *)&on,
              sizeof(on)) < 0)
        ERR_QUIT ("traceroute: IP_HDRINCL", 6);

    icmpTid = taskSpawn("tIcmpRecv", 90, 0, 50000,
          (FUNCPTR)troute_icmpMsgReceiver );
    if (icmpTid == ERROR)
      perror("Error in spawning icmpReceiver Task ");
```

```
    printf("traceroute to %s ", inet_ntoa(whereto.sin_addr));
    printf(", %d hops max, %d byte packets\n", max_ttl, datalen);

    for (ttl = 1; ttl <= max_ttl; ++ttl) {
        got_there = 0;
        unreachable = 0;

        printf("%2d ", ttl);
        for (probe = 0; probe < nprobes; ++probe) {
            send_probe(++seq, ttl);
            taskDelay(sysClkRateGet() * waittime);
        }
        putchar('\n');
        if (got_there || unreachable >= nprobes-1)
            break;
    }
    taskDelete ( icmpTid );
    close ( sndsock );
    return 0;
}
```

发送函数 send_probe( )从 IP 头部结构开始构造用于探测的 UDP 数据报并发送。

**注意：**

（1）用于发送的裸层 socket 具有 IP_HDRINCL 选项；（2）每次发送的 UDP 数据报的目的端口号根据序号生成，这样接收时可以区分不同的发送。

```
void send_probe(seq, ttl)
    int seq, ttl;
{
    struct opacket *op = outpacket;
    struct ip *iphdr = &op->iphdr;
    struct udphdr *up = &op->udp;
    struct timespec ts;
    iphdr->ip_off = 0;                  /*IP 头部结构*/
    iphdr->ip_hl = sizeof(*iphdr) >> 2;
    iphdr->ip_p = IPPROTO_UDP;
    iphdr->ip_len = datalen;
    iphdr->ip_ttl = ttl;
    iphdr->ip_v = IPVERSION;
```

```
    iphdr->ip_id = htons(ident+seq);
    up->uh_sport = htons(ident);       /*UDP 头部结构*/
    up->uh_dport = htons(port+seq);
    up->uh_ulen = htons((u_short)(datalen - sizeof(struct ip)));
    up->uh_sum = 0;
    op->seq = seq;
    op->ttl = ttl;
    clock_gettime(CLOCK_REALTIME, &ts);
    op->tv.tv_sec = htonl(ts.tv_sec);
    op->tv.tv_usec = htonl(ts.tv_nsec);

    sendto(sndsock, (char *)outpacket, datalen, 0,
           (struct sockaddr*)&whereto, sizeof(whereto));
}
```

## 2. 接收任务

troute_icmpMsgReceiver( )在一个无限循环中接收 ICMP 数据报，对收到的数据报调用 packet_ok( )检查 ICMP 类型，设置相应标志使主任务退出。

```
void troute_icmpMsgReceiver ( )
{
    int   s;
    u_long lastaddr = 0;
    if ((s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP)) < 0)
        ERR_QUIT ("traceroute: icmp socket", 5);

    for (;;) {
        struct sockaddr_in from;
        register int size;
        int   fromlen;
        struct timespec ts1, ts2;
        int   i;
        struct ip *iphdr;
        clock_gettime(CLOCK_REALTIME, &ts1);

        fromlen = sizeof(from);
        if ((size = recvfrom(s, (char *)packet, sizeof(packet), 0,
            (struct sockaddr *)&from, &fromlen)) < 0) {
            perror("ping: recvfrom");
```

```
            continue;
        }
        if ((i = packet_ok(packet, size, &from, seq))) {
            if (from.sin_addr.s_addr != lastaddr) {
                printf(" %s", inet_ntoa(from.sin_addr));
                lastaddr = from.sin_addr.s_addr;
            }
            clock_gettime(CLOCK_REALTIME, &ts2);
            printf("  %lu ms", DELTATIME(&ts1, &ts2));
            switch(i - 1) {
                case ICMP_UNREACH_PORT:
                case ICMP_UNREACH_PROTOCOL:
                    ++got_there;
                    break;
                default:
                    ++unreachable;
                    break;
            }
        }
    }
}
```

接收任务中使用了函数 packet_ok( )检测接收的结果。packet_ok( )期待接收子类型为 ICMP_TIMXCEED_INTRANS 的超时报文 ICMP_TIMXCEED，或者不可达报文 ICMP_UNREACH_XXX。在生成这些 ICMP 报文时，源 IP 分组被完整地包含在 ICMP 报文内部，如图 8-21 阴影部分所示。因此 packet_ok( )判断 ICMP 类型同时检查是否该报文是否由探测主任务引起的。packet_ok( )对探测主任务引起的超时报文返回-1；对探测主任务引起的不可达报文返回不可达类型+1；其他情况返回 0。

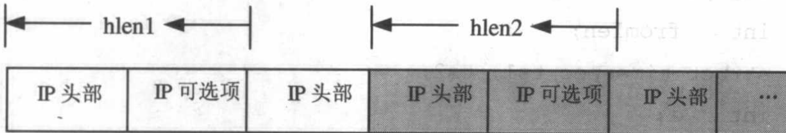



图 8-21　ICMP 出错报文

```
int packet_ok(buf, size, from, seq)
    u_char *buf;
    int size;
    struct sockaddr_in *from;
```

```
    int seq;
{
    register struct icmp * icmphdr;
    u_char type, code;
    int    hlen1, hlen2;

    struct ip *iphdr = (struct ip *) buf;
    hlen1   = iphdr->ip_hl << 2;
    icmphdr = (struct icmp *)(buf + hlen1);
    if (size < hlen1 + ICMP_MINLEN)
        return (0);

    type = icmphdr->icmp_type;
    code = icmphdr->icmp_code;
if ((type == ICMP_TIMXCEED && code == ICMP_TIMXCEED_INTRANS)
            || type == ICMP_UNREACH) {
        struct ip *hip = &icmphdr->icmp_ip;
        struct udphdr *up;
        hlen2 = hip->ip_hl << 2;
        up = (struct udphdr *)((u_char *)hip + hlen2);

        if (hlen1+hlen2 + 12 <= size && hip->ip_p == IPPROTO_UDP &&
            up->uh_sport == htons(ident) && up->uh_dport == htons(port+seq))
            return (type == ICMP_TIMXCEED? -1 : code+1);
    }
    return(0);
}
```

在运行 Traceroute 时需要注意，在某些受限网络环境只允许通过一些常用端口的数据报，例如网络中安装防火墙限制只能访问 HTTP，这种情况下 Traceroute 发送的探测报文被网络忽略。以下是 traceroute 运行的一个示例输出：

```
-> sp(traceRouteMain,"202.114.95.104");
task spawned: id = 0x1f7c700, name = t3
value = 33015552 = 0x1f7c700
-> traceroute to 202.114.95.104 (202.114.95.104), 30 hops max, 96 byte packets
 1  192.168.0.254  0 ms  2000 ms  2000 ms
 2  202.114.85.129  35333.3 ms  -563667 ms  534333 ms
 3  202.114.95.104  2000 ms  2000 ms  2000 ms
```

# 8.9 socket 应用高级话题

## 8.9.1 I/O 控制

VxWorks 支持通过基本 I/O 系统调用 ioctl 对 socket 进行一些 I/O 控制，实现了的控制命令有：

### 1. FIONBIO

前面论述中已经涉及了一些对阻塞和非阻塞方式的讨论。阻塞或非阻塞可以动态地改变，例如，下面将设置 s 为非阻塞方式：

```
int blockflag = TRUE;   /*指定为 FALSE 将设置阻塞方式*/
status = ioctl ( s, FIONBIO, &blockflag );
```

**注意：**

在 UNIX 系统以及许多 BSD 官方资料中，通过 fcntl( )调用来设置阻塞方式，VxWorks 中没有 fcntl，调用 ioctl 完成同样的功能。

### 2. FIONREAD

该命令用于报告 socket 中可以被读取的字节数。如：

```
int bytesAvailable;
status = ioctl ( s, FIONREAD, &bytesAvailable );
```

### 3. SIOCATMARK

包括 socket 中是否有带外数据可以读取。如：

```
int mark;
status = ioctl ( s, SIOCATMARK, &mark );
```

如果有带外数据，ioctl 返回后 mark 为 TRUE；否则为 FALSE。

## 8.9.2 socket 选项

对 socket 定义了一些选项，用于控制 socket 及底层协议的多方面属性。这些选项通过下面的函数进行设置和读取：

```
#include "sockLib.h"
STATUS setsockopt ( int s, int level, int optname, char * optval, int optlen );
STATUS getsockopt ( int s, int level, int optname, char * optval, int * optlen );
```

其中，各个参数表示的含义是：

- s 表示打开的 socket 描述符；
- level 是函数所使用的协议标准(Protocol Level)，分 IPPROTO_TCP，IPPROTO_IP 和 SOL_SOCKET；
- optname 是设置或者读取的选项；
- optval 表示要设置的选项值的地址（Setsockopt），或者读出的选项值存放地址（Getsockopt）；
- optlen 表示要设置的选项值 optval 的长度（Setsockopt），或者读出选项值 optval 的长度（Getsockopt），以字节为单位。

函数返回值为 0 表示成功；或者-1 表示失败。失败的原因包括：有无效参数或者无法设置指定选项。

&　**getsockopt( )**

调用之前一定要将 optlen 初始化为 optval 长度，否则 getsockopt( )内部的 bcopy 可能导致错误地修改其他存储空间。

对于 0/1 型的布尔结果，应该将 optval 定义为 int 型，不要定义为 char 型。

当 level 为 SOL_SOCKET 时，optname 可以为下列选项。如表 8-21 所示，读写属性栏“R”和“W”分别表示可以执行 getsockopt 和 setsockopt；类型栏中的“T”，“U”和“R”分别表示 SOCK_STREAM，SOCK_DGRAM，SOCK_RAW。

**表 8-21　选项及读写属性、类型与含义**

| 选项（optname） | 读写属性 | 类　型 | 含　义 |
|---|---|---|---|
| SO_ERROR | RW | TU | 读待处理错误并清除 |
| SO_KEEPALIVE | RW | T | 启用或禁用保持连接定时器 |
| SO_LINGER | RW | T | 控制文明的关闭连接 |
| SO_DEBUG | RW | T | 启用或禁止对底层协议调试 |
| SO_BROADCAST | RW | U | 允许或禁止广播 |
| SO_REUSEADDR | RW | TU | 允许重用本地地址 |
| SO_REUSEPORT | RW | TU | 允许重用本地地址和端口 |
| SO_SNDBUF | RW | TU | 发送缓冲区大小 |
| SO_RCVBUF | RW | TU | 接收缓冲区大小 |
| SO_OOBINLINE | RW | TU | 带外数据放在正常数据流中 |

当 level 为 IPPROTO_TCP 时，optname 可以为下列选项，如表 8-22 所示：

表 8-22 optname 选项说明一

| 选项（optname） | 读写属性 | 类 型 | 含 义 |
|---|---|---|---|
| TCP_NODELAY | RW | T | 禁止 Nagle 算法（立即送出） |

当 level 为 IPPROTO_IP 时，optname 可以为下列选项，如表 8-23 所示：

表 8-23 optname 选项说明二

| 选项（optname） | 读写属性 | 类 型 | 含 义 |
|---|---|---|---|
| IP_ADD_MEMBERSHIP | W | UR | 加入主机组 |
| IP_DROP_MEMBERSHIP | W | UR | 离开主机组 |
| IP_MULTICAST_IF | RW | UR | 指定多播的网络接口 |
| IP_MULTICAST_TTL | RW | UR | 指定多播分组的 TTL |
| IP_MULTICAST_LOOP | RW | UR | 指定是否环回 |
| IP_OPTIONS | RW | TUR | 设置 IP 分组选项字段 |
| IP_HDRINCL | RW | R | 发送数据是否包括 IP 头部 |
| IP_TOS | RW | TUR | 设置 IP 分组服务类型字段 |
| IP_TTL | RW | TUR | 设置 IP 分组存活时间 TTL |
| IP_RECVDSTADDR | RW | TUR | 设置队列 UDP 目的 IP 地址 |

表 8-23 中的广播选项和多播选项在前面 8.7 节“无连接的 socket 应用”中已经分析了，IP_HDRINCL 表示裸层 socket 发送数据是否包含 IP 头部信息，在 8.8 节“裸层 socket”中进行了介绍。下面对几个重要的选项进行深入一些的分析，这些特性对一些需要深入 TCP/UDP 内部的应用比较有用。

## 1. SO_KEEPALIVE（保持连接定时器）

选项 SO_KEEPALIVE 用来决定是否启用保持连接定时器。该定时器用于检测死连接。为 setsockopt 传递的参数 optval 为一个整数，指定的值表示启用定时器或者禁用定时器，如：

```
int optval = 1;  /*optval: 1=启用定时器 0=禁用定时器*/
setsockopt ( s, SOL_SOCKET, SO_KEEPALIVE, (char*)&optval, sizeof (optval));
```

TCP 必须考虑，在 TCP 连接建立后，如果连接的一端死掉或者网络中断，TCP 必须能检测到这一情况并释放连接。该过程是系统自动进行的，用户程序可以进行一些控制。

具体地，如果连接已经建立，KEEPALIVE 定时器在没有传输时间长达 TCPTV_KEEP_

IDLE 后启动，TCP 通过发送一个“保持连接”报文（即数据段长度为 0），接收该报文的另一端收到后会发送一个 ACK。如果没有收到 ACK，TCP 将在 TCPTV_KEEPINTVL 时间间隔后继续发送保持连接报文。在连续发送了 TCPTV_KEEPCNT 次后还没有收到 ACK 时，TCP 将释放该连接，所有的协议数据结构资源都将被释放（不包括 VxWorks I/O 系统资源），所有阻塞在该连接上的读写任务都解除阻塞，相关调用返回 ETIMEOUT。

除了在连接建立后检测连接断开外，在连接未建立时，保持连接定时器还用于控制连接时间。但不受 SO_KEEPALIVE 控制，因为系统总是通过该定时器来控制最大建立连接的时间：从连接过程开始，经过 TCPTV_KEEP_INIT 后保持连接定时器溢出，如果连接还未建立，则发生 ETIMEOUT（即我们介绍 connect 调用时说的超时）。

在<install-dir>/target/h/net/tcp_timer.h 文件中可以找到所有上面涉及的常量定义。默认情况下，保持连接时间大约有数小时，即在数小时内没有收到对方回应才会释放连接。可以直接修改一个全局变量 tcp_keepidle 来减少保持连接时间，例如：

```
-> tcp_keepidle=1800
```

将保持连接时间设置为 30 分钟。

在程序中可以通过检查 recv 或 read 调用是否返回 0 来得知对方是否已经关闭连接。这一点在 8.6 节“面向连接的 socket 应用”中已经讨论过了。

## 2. SO_LINGER（控制连接关闭过程）

选项 SO_LINGER 控制在连接关闭时是否按照标准的 TCP 状态转换过程关闭连接，遵循这样一个标准的关闭过程能尽量确保网络中的数据被可靠传输。

下面表示需要执行这样一个“LINGER”关闭过程：

```
struct  linger optval;
optval.l_onoff  = 1;     /*执行 linger 关闭过程*/
optval.l_linger = 0;    /*设置 linger 时间=0（用系统默认值）*/
setsockopt ( s, SOL_SOCKET, SO_LINGER, (char *)&optval, sizeof (optval));
```

结构体 linger 成员 l_onoff 为 1 表示执行 LINGER 关闭过程；为 0 表示不执行 LINGER 关闭过程。

对于服务器：从一个 socket 上调用 accept 接受连接，该 socket 的 l_onoff 为 1 而 l_linger 为 0 时，代表新连接的 socket 将使用系统默认时间（TCP_LINGERTIME）作为其 l_linger。服务器可以在 accept 后对表示新连接的 socket 设置 LINGER 时间。

对于客户端：设置 SO_LINGER 选项时指定 l_linger=0 不会修改其 LINGER 时间。

如果执行 LINGER 关闭过程（l_onoff=1）并且 LINGER 时间大于 0，关闭时 TCP 将尝试使所有没有被确认的数据被确认，然后执行 TCP 连接关闭的状态转换过程。

如果执行 LINGER 关闭过程（l_onoff=1）并且 LINGER 时间指定为 0（l_linger=0），TCP 只发送一个 RST 报文。该 socket 占用的 TCP 资源被释放，阻塞的任务解除阻塞。

### 3. TCP_NODELAY（立即发送）

该选项用于控制当用户数据以很小的单位到达TCP发送方时，TCP是否立即发送。因为一个TCP报文有固定的封装开销，当频繁发送用户数据非常小的报文时系统效率会非常低，并且可能导致网络拥塞。

默认时，VxWorks对此的做法是（Nagle算法）：当用户数据以很小的单位到达时，只发送第一个TCP报文，而将后续的数据累积在缓冲区，直到送出的报文被确认或者缓冲区积累的数据到达某个阈值后再将缓冲区内数据送出。

如果应用要求TCP立即送出到达的数据，可以指定TCP_NODELAY，如：

```
int  optval = 1;   /*optval: 1=启用立即发送 0=不启用*/
setsockopt ( s, IPPROTO_TCP, TCP_NODELAY, (char*)&optval, sizeof (optval));
```

参数optval为1表示启用立即发送，或者为0表示不启用立即发送（即使用系统默认的传输策略）。启用立即发送后，发送与否只取决于对方通知的接收窗口大小，而与用户数据大小无关。

### 4. TCP_MAXSEG（最大TCP报文）

选项TCP_MAXSEG用于设置最大TCP报文（MSS）大小，含TCP封装结构和用户数据。对设置MSS的要求是不能超出IP载荷能力（65535）和特定网络的最大传送单位（MTU），例如在以太网上，MTU为1460字节。

在socket被创建之初，MSS大小为系统定义的默认值TCP_MSS_DFLT（512字节）；TCP3次握手连接过程中，将根据网络MTU大小（自动检测）和对方的MSS重新确定新的MSS值，如果都在以太网上，MSS一般被确定为1460。

例如，下面设置MSS为1024：

```
int  optval = 1024;   /*optval: MSS大小*/
setsockopt ( s, IPPROTO_TCP, TCP_MAXSEG, (char*)&optval, sizeof (optval));
```

**MSS**

*任何时候，修改MSS只能是在当前MSS的基础上减少而不能增加。*

### 5. SO_SNDBUF和SO_RCVBUF（缓冲区设置）

选项SO_SNDBUF和SO_RCVBUF同时适用于TCP和UDP。对于每个socket，系统都为其维护一个发送缓冲区和接收缓冲区，VxWorks允许通过这两个选项动态地调整缓冲区大小：

```
setsockopt ( s, SOL_SOCKET, SO_SNDBUF, &optval, sizeof (optval));
setsockopt ( s, SOL_SOCKET, SO_RCVBUF, &optval, sizeof (optval));
```

参数 optval 为一整型变量指定发送或接收缓冲区大小。

对缓冲区的设置不会立即从系统 mbuf 链表上申请空间，而是记录一个最高允许值，只是当缓冲区不足时以上述设定为限申请新空间。但是，对接收缓冲区的设置能立即影响发送方的行为。

对于 UDP，当没有更多接收缓冲区容纳远程 UDP 数据报时，系统简单地将数据丢弃；对于 TCP，本地接收缓冲区用于实现流控：系统将本地的接收缓冲区中的可用空间大小的一半（“接收窗口”大小）通知给对方，表示允许对方在未确认情况下最多发送多少字节。通常，设置接收缓冲区大小要考虑“管道存储”效应，即将连接看成具有一定大小 S（相当于连接的带宽）和一定长度 L（相当于该连接上的一次往返时间 RTT），则接收缓冲区大小不应该小于 S×L，以充分利用连接的带宽。

当一个 TCP socket 被创建时，发送缓冲区大小为 8192 字节，接收缓冲区大小为 8192 字节；当一个 UDP socket 被创建时，发送缓冲区大小为 9216 字节，接收缓冲区大小为 41600 字节（大约可以容纳 40 个 1KB 大小的 UDP 报文）。

需要注意的是最大窗口大小在建立 TCP 连接时既已确定，setsockopt 增加己方接收缓冲区并不会使通告对方的接收缓冲区大小超出该最大值。

## 6. SO_REUSEADDR 和 SO_REUSEPORT（地址重用）

地址重用即一个或多个任务重复绑定到相同的传输层端口。有时候系统中配置了多个网络接口，这时某些应用需要重用一个传输层端口号，也就是在一个端口上接收来自不同 IP 地址的数据。如图 8-22 所示是两种非常典型的地址重用模型。

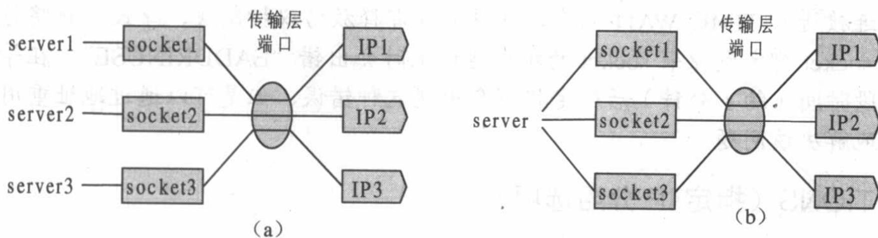



图 8-22　地址重用

在不使用端点地址重用时，在同一个传输层端口上创建第二个 socket 时将失败，错误为 EADDRINUSE。

在图 8-22 中，（a）和（b）常常代表两种不同类型应用。图 8-22（a）一般表示 TCP 端口重用，服务器 server1~server3 通过一个 TCP 端口分别和 3 个不同的网络接口绑定，例如在多网络接口的主机上同时存在 3 个侦听 80 端口的 HTTP 服务器。在图 8-22（b）中，一个服务器将 3 个 socket 绑定到同一个传输层端口但是不同的网络接口。某些 UDP 应用采取图 8-22（b）的模型，这一服务器可以根据不同的 socket 判断出客户数据指定的目标 IP 地址（即服务器本地网络接口的 IP 地址）。在两种模型中，不论从哪个网络接口看，主

机上的特定服务器都在同一端口为客户服务，此即地址重用的要旨。

如果属于上述情况，则需要指定地址重用。指定地址重用的时机是在 bind 调用之前，通过 SO_REUSEADDR 选项指定，如：

```
int optval = 1;  /*为 1 表示该启用该 socket 的地址重用，为 0 表示禁止地址重用*/
setsockopt ( s, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof (optval));
```

还有另外一种重用，将 socket 绑定到完全相同的端口和 IP 地址（**完全重复绑定**）。这时需要通过 SO_REUSEPORT 选项，该选项可以看成在 SO_REUSEADDR 基础上进一步允许绑定的 IP 地址相同。

完全重复绑定主要用在多播的场合，即多个任务绑定到相同的 UDP 端口号和组地址。如果属于这种情况，SO_REUSEADDR 和 SO_REUSEPORT 的功能是一样的，即 SO_REUSEADDR 可以用于重复绑定到相同的组地址，但是相同的普通 IP 地址只能通过 SO_REUSEPORT 来重复绑定。

SO_REUSEPORT 的设置方式和 SO_REUSEADDR 类似，但是对每一个 socket 进行 bind 调用以前必须设置此方式；而 SO_REUSEADDR 只要求后面的 socket 要绑定端口已经被绑定后设置此方式以表明这是一次重复绑定。

```
int optval = 1; /*为 1 表示该启用完全充分绑定，为 0 表示禁止完全重复绑定*/
setsockopt ( s, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof (optval));
```

& 还存在这样一种需要地址重用的情况，即在关闭一个 socket 后，该 socket 上的 TCP 连接进入 TIME_WAIT 状态，并没有立即释放协议控制块，导致重新将另一个 socket 绑定到原来 socket 的端点地址上时会出错 “EADDRINUSE”。在等待一段时间（约 1 分钟）后绑定将不会出现这种错误，但是可以通过地址重用更好地解决该问题。

### 7. IP_OPTIONS（指定 IP 分组选项）

```
setsockopt ( s, IPPROTO_IP, IP_OPTIONS, optbuf, optbuflen);
```

参数 optbuf 指定 IP 分组头部的选项字段，optbuflen 表示 optbuf 长度。IP 协议允许指定 1～40 字节的选项字段。系统自动将 optbuflen 向上圆整到 4 的倍数。选项字段可以完成排错、跟踪以及安全等功能，但是较少使用。

### 8. IP_TTL（指定 IP 分组存活时间）

选项 IP_TTL 指定 IP 头部的 TTL 字段。默认情况下，UDP 和 TCP 报文的 IP 分组头部 TTL 字段为 64。

```
int optval;   /*optval: 0~255*/
setsockopt ( s, IPPROTO_IP, IP_TTL, &optval, sizeof(optval));
```

### 9. IP_TOS（设置服务类型 Type-Of-Service）

```
int optval;   /*optval: 服务类型*/
setsockopt ( s, IPPROTO_IP, IP_TOS, &optval, sizeof(optval));
```

参数 optval 指定发出 IP 分组头部的 TOS 字段，可以是下列值，这些值在 netinet/ip.h 中定义：

- 优化延迟 IPTOS_LOWDELAY；
- 优化吞吐量 IPTOS_THROUGHPUT；
- 优化可靠性 IPTOS_RELIABILITY；
- 优化成本 IPTOS_MINCOST。

### 10. IP_RECVDSTADDR（设置是否队列目的 IP 地址）

指定系统是否为 socket 记录收到的 UDP 数据报的目的 IP 地址。

```
int optval;    /*optval: 1=接收 UDP 目的 IP 地址 0=不接收*/
setsockopt ( s, IPPROTO_IP, IP_RECVDSTADDR, &optval, sizeof(optval));
```

有时候接收 UDP 数据报之前没有绑定 socket 的本地地址，或者绑定到通配地址 INADDR_ANY，在这种情况下，可以通过 IP_RECVDSTADDR 确定数据报的目的 IP 地址。但是目前 VxWorks 中该选项只能检查出到单播的目的地址，而无法用于读出多播数据报的目的 IP 地址。

## 8.9.3 I/O 复用

I/O 复用，即 select 机制，允许同时在多个文件描述符上时等待读写就绪，使程序对 I/O 的处理具有并发的特点。我们在第 5 章 5.3 节“I/O 复用”中对此介绍过。

在 socket 应用中，不论客户还是服务器，都可以利用 I/O 复用使得 I/O 效率充分提高。同时，select 不仅仅用在 socket 的发送和接收上，还用在连接建立过程中。

socket 传输中的复用本质上和一般文件系统 I/O 复用没有差别，即一个任务（客户或服务器）通过 select 同时监视多个 socket 描述符，任意一个读或写就绪时就对其处理（如果要做到并发，可以生成新任务对其处理，而监视任务继续进行监视）。还可以对 socket I/O 和一般文件系统 I/O 一起复用，如写管道和读 socket。

### 对 TCP 连接的进一步讨论

建立连接的过程是 socket 编程中最复杂的内容之一，也最能体现程序设计的艺术。我们来看连接过程中比较深入的问题。

在客户端：通过 connect 建立连接，其等待连接建立的时间受网络延迟影响变换比较

大（可以长达75秒）。有时候希望在确定时间内连接，或者需要在多个socket上操作，则应该避免在一个socket上阻塞。大体上，实现这一目的有两种方法。

一种方法就是设置socket为非阻塞方式，这时，如果连接不能马上建立，函数会立即返回-1，errno= EINPROGRESS，系统将在幕后进行建立连接的过程。在实践中，该方式的复杂性在于什么时候以什么方式去检查连接已经建立。对此常常采取的做法是I/O复用，即select机制。也就是先在非阻塞的socket上调用connect以启动一个连接过程，然后通过select等待（或检查）连接是否已经建立，select的等待过程具有超时控制。

select本来用于等待多个读或写文件描述符就绪，将其用于判断socket连接是否已经建立需要用到BSD定义的两条特殊规则：(1) **如果连接已经建立，socket描述符编程可写**；(2) **如果连接出错，socket描述符编程可读又可写**。我们不去探讨这两条规则的来源，下面大致地给出利用这两条规则实现一个受控制的连接过程的方法：

```
...
blockflag=TRUE;
ioctl (sFd, FIONBIO, &blockflag );    /*设置socket为非阻塞方式*/
if ( connect  ( s, &serverAddr, addrLen ) == 0 )
    ... /*表明连接已经建立*/
if ( errno != EINPROGRESS)
    ... /*非阻塞的socket连接期间errno为EINPROGRESS */
FD_ZERO ( &rd_set ); FD_SET ( s, &rd_set ); /*rd_set: 读文件描述符集*/
FD_ZERO ( &wr_set ); FD_SET ( s, &wr_set ); /*wr_set: 写文件描述符集*/
if ( select(s+1, &rd_set, &wr_set, NULL, &timeout )==0 )/*timeout:超时*/
    ... /*表明连接超时*/
if ( FD_ISSET(s, &rd_set) && FD_ISSET(s, &wr_set) )
    ... /*表明连接错误，可能errno为EINPROGRESS或其他*/
if ( !FD_ISSET(s, &rd_set) && FD_ISSET(s, &wr_set) )
    ... /*连接成功！*/
```

从上面可以看出I/O复用的方法是比较复杂的。如果只是控制等待时间，而不需要进行等待复用，可以使用另外一种简便方法，即VxWorks实现的另一个发起连接的函数connectWithTimeout：

```
#include "sys/socket.h"
#include "netinet/in.h"
STATUS connectWithTimeout ( int sock, struct sockaddr * adrs, int adrsLen,
       struct timeval * timeVal );
```

该函数非常简单，除了指定一个超时参数timeVal控制最大等待连接时间外，connectWithTimeout和connect在各方面都是一样的。

在服务器端，有时也希望避免在accept上阻塞，例如，服务器可能需要同时在多个socket上侦听客户连接，或者还有处理和其他客户的交互。实际上，在阻塞式的socket上，accept可以无限等待，只要没有客户请求到来。

没有类似acceptWithTimeout这样的函数，因此控制accept的等待时间只能通过I/O复用。BSD同样定义了一条规则用于accept时的select调用：**当侦听socket中有连接请求时，该socket是可读的**。因此，对accept的I/O复用的做法大致是下面这样一个过程。这里我们只对一个侦听socket进行select操作。

```
...
bind ( s, &serverAddr, addrLen );  /*地址绑定*/
listen ( s, MAX_CONNECTIONS );      /*指定侦听*/
do {
FD_ZERO ( &ready );
    FD_SET ( s, &ready );
    if ( ( select ( s+1, &ready, 0, 0, &timeout ) > 0 )  &&
            FD_ISSET ( s, &ready ) )
    {   /*表明有连接请求*/
        newsock = accept(s, (struct sockaddr *)clientAddr, (int *)addrLen);
        ...
    }
    else
        ...   /*其他情况处理*/
} while (TRUE);
```

通常，服务器上对accept进行I/O复用和客户端对connect进行I/O复用有不同的目的，后者是为了限制3次握手时间；前者是避免在客户请求到来之前长时间阻塞在一个socket上。因此，上面的示例代码中不需要像前面对connect进行I/O复用那样将服务器的侦听端口设置为非阻塞方式。但是accept支持非阻塞方式，前文曾指出：对于非阻塞式的socket，如果没有新的连接请求，accept立即返回-1，errno=EWOULDBLOCK。

### 8.9.4 超越I/O复用限制

我们知道select能够复用等待多个文件描述符读/写就绪，但是select能复用的I/O受FD_SETSIZE限制，即select等待的文件描述符中，最大只能到FD_SETSIZE。在Tornado 2.0种，FD_SETSIZE定义为256，在Tornado 2.2中增加到了2048。对于socket以外的I/O，256个文件描述符也许足够了，但是对于socket，一个处理大量客户请求的服务器需要的文件描述符往往会对FD_SETSIZE定义。在这种情况下，select无法满足socket应用的需

求（我们无法增加 FD_SETSIZE 定义，因为这需要获得 VxWorks 的协议栈实现的源码然后重新编译）。

不使用 select，稍微深入 socket 内部可以解决上面的问题。先来回顾一下，在第 7 章“网络数据流分析”中，我们曾指出当 IP 层向传输层 TCP 和 UDP 提交一个报文时，传输输入函数 tcp_input 和 udp_input 会将数据加入到 socket 的接收缓冲区然后唤醒阻塞在 socket 上的任务（如果有的话）。实际上，在去检查唤醒阻塞任务之后，tcp_input 和 udp_input 还有一个细节：调用一个可由用户定义的全局回调钩子 sowakeupHook，这一细节体现在下面的代码段 sowakeup( )中：

```
void sowakeup(so, sb, wakeupType)
struct socket *so;
struct sockbuf *sb;
SELECT_TYPE wakeupType;
{
    /* 唤醒阻塞的任务（包括 select 调用），如果有 */
    sbwakeup(so, sb, wakeupType);
    /* 检查如果定义了回调钩子，则调用之 */
    if (sowakeupHook != NULL)
        (*sowakeupHook) (so, wakeupType);
    ...
}
```

函数 sowakeup( )在多种事件触发下将被 TCP 和 UDP 调用，包括上面说的收到数据时被 tcp_input( )和 udp_input( )调用。其中参数 so 表示 socket 数据结构；sb 表示 socket 的接收或发送缓冲区（可不理会此参数）；wakeupType 为 SELREAD 和 SELWRITE，分别表示由接收数据和发送数据引起的函数调用。可以看出，在检查唤醒阻塞任务后，sowakeup ( ) 会检查是否定义了回调钩子 sowakeupHook，如果有，则传递参数 so 和 wakeupType 调用它。因此，我们需要了解在什么时候会引起 sowakeup 调用，才能决定设计回调钩子能完成什么工作。

具体来说，TCP 和 UDP 对 sowakeup( )的调用发生在：

- 新接收数据队列到 socket 的接收缓冲区后，以参数 wakeupType = SELREAD 调用 sowakeup( )；
- socket 发送缓冲区可用，以参数 wakeupType = SELWRITE 调用 sowakeup( )；
- 完成建立 TCP 连接的 3 次握手过程，以参数 wakeupType = SELREAD 调用 sowakeup( )。

基于上面的分析，我们可以设计一个回调钩子，通过对上述 3 种情况的捕获，来实现类似 select 的效果。回调钩子是全局的，其原型应该定义为：

```
void wakeupHookName (struct socket *so, SELECT_TYPE wakeupType );
```

其中，指向 socket 结构体的 so 中包括 socket 文件描述符（so→so_fd）；wakeType 为 SELREAD 或者 SELWRITE。

上面只说明了回调钩子的机制，如何利用这一机制实现 I/O 复用具体有不同的做法。例如，对于要处理许多连接的客户，可以将 socket 设置为非阻塞方式，然后在该 socket 上执行 connect( )调用。一般情况下，connect( )会立即返回。然后在回调钩子中检测是否是一个连接请求完成，比如可以这样设计回调钩子：

```
void myWakeupHook ( struct socket *so, SELECT_TYPE wakeupType )
{
    if(wakeupType == SELREAD)
    {
        if ( sockFd == so->so_fd )
... /*这里表示完成连接过程*/
    ...
```

和 select 不同的是，回调钩子在另一任务（tNetTask）中运行，本质上是一种异步于任务的操作，而 select 是同步于任务的。以上述客户程序中的回调钩子为例，myWakeupHook( )运行时，它并不知道客户任务处于何种状态，而 select 机制下，任务从调用 select 到 select 返回，这一过程是同步的。这种异步带给我们的设计一个问题就是如何设计钩子函数和任务的同步。在我们的例子中，一个方案是通过信号量，例如，在执行 connect( )调用的客户端和钩子函数中如下设计：

```
/* 客户端设计 */
connect( sockFd );   /*注意 sockFd 为非阻塞式*/
if ( semTake(sem_for_client, timeout_period ) == OK )
    printf( "connected!" );
else
    printf( "cant connect in specified timeperiod!" );

/* 回调钩子设计 */
    if(wakeupType == SELREAD)
    {
        if ( sockFd == so->so_fd )
            semGive( sem_for_client );
```

回调钩子机制和 select 机制的另一不同还在于 select 有超时控制。

在上面给出的代码段中，我们没有考虑多个 socket 文件描述符。但增加这种考虑没有

本质不同，只是设计任务函数时，需要将每个 socket 文件描述符加入到一个链表结构，然后在回调钩子中检查链表中每个 socket 文件描述符。

### 8.9.5 深入底层处理

VxWorks 提供一种“钩子函数”机制允许任务在系统之前处理接收的 IP 分组。严格来说这不属于 socket 编程的内容，钩子函数在 socket 层下面，甚至在 IP 层下面进行处理。如果希望知道其中细节，可以参考第 7 章 7.2 节“网络数据流分析”和第 9 章“网络驱动”部分。这里我们只给出如何使用这一机制。简单地说，如果要在系统之前处理 IP 分组，只需要利用 VxWorks 钩子库 ipFilterHook 提供的接口函数即可安装钩子函数。钩子在某个任务中安装，但属于整个系统，一个系统也只能有一个钩子。

钩子函数具有固定的原型定义：

```
BOOL ipFilterHook (
    struct ifnet *pIf,         /* interface that received the packet */
    struct mbuf  **pPtrMbuf,   /* pointer to pointer to an mbuf chain */
    struct ip    **pPtrIpHdr,  /* pointer to pointer to IP header */
    int          ipHdrLen,     /* IP packet header length */
);
```

其中 pPtrIpHdr 指向待处理分组的 IP 头部指针，需要注意钩子函数的返回值：如果钩子函数返回 TRUE，表示不需要系统继续处理该分组；否则，钩子函数应该返回 FALSE，表示需要系统继续处理。一般情况下应该返回 FALSE，否则钩子函数必须负责 IP 及其上层协议栈实现的功能，包括释放 mbuf。

具体安装钩子函数由钩子库 ipFilterHook 提供的下列接口实现：

```
STATUS ipFilterHookAdd ( FUNCPTR ipFilterHook );  /*安装钩子*/
void ipFilterHookDelete ( void );   /*删除钩子*/
```

下面是一个简单的例子，该例子中钩子函数 myIpFilterHook 将判断正在接收的分组是否是一个 ICMP ECHO 请求，如果是，则输出一条指示信息。

```
#include "vxWorks.h"
#include "netinet/in.h"
#include "netinet/ip.h"
#include "netinet/in_systm.h"
#include "netinet/ip_icmp.h"

BOOL myIpFilterHook (
```

```
    struct ifnet *pIf, struct mbuf **pPtrMbuf,
    struct ip **pPtrIpHdr, int ipHdrLen
)
{
    struct ip * ipHdr;
    struct icmp * icmpHdr;
    ipHeader = *pPtrIpHdr;
    if ( ipHdr->ip_p == IPPROTO_ICMP )
    {
        icmpHdr = (struct icmp *)
                ( (void *)ipHdr + ipHdr->ip_hl * 4 );
        if ( icmpHdr->icmp_type == ICMP_ECHO )
            printf("Ping from [%s] to [%s]\n", inet_ntoa(ipHdr->ip_src),
                    inet_ntoa(ipHdr->ip_dst));
    }
    return(FALSE);
}
```

安装钩子后，将得到类似下面的结果。

```
-> ipFilterHoodAdd(myIpFilterHook)
...       (如果从其他主机 ping 目标系统)
Ping from [168.10.10.152] to [168.10.10.252]
```

# 第9章 网络驱动（END）

## 9.1 网络驱动层次结构

在 VxWorks 中，网络驱动程序和上层网络协议栈可以有两种实现方式：BSD 方式和 MUX 方式。BSD 方式下，网络驱动直接和上层协议集成到一起。MUX 方式下，驱动程序和上层协议之间不直接交互：MUX 提供上层接口和下层接口，协议栈通过上层接口和 MUX 绑定，当底层驱动通过 MUX 下层接口将数据提交给 MUX 时，MUX 根据绑定将数据转给协议栈处理。MUX 允许系统实现多个底层驱动，同时允许定制专门的上层协议。如图 9-1 所示。

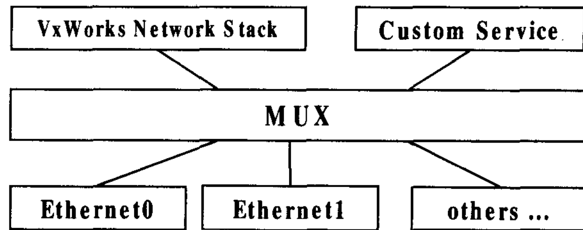



图 9-1 MUX 的层次结构

MUX 方式下，网络驱动程序基本上就是 **END 驱动**（增强型网络驱动），END 和 MUX 之间的数据交换以帧为单位。另外一种比较新的可用于 MUX 方式下的驱动为 NPT 驱动，NPT 驱动和 END 驱动有些微小的差别，但是主体结构是一样的。WRS 目前没有提供 NPT 驱动的模板，可用的 NPT 驱动也很少，因此本书以 END 驱动为基础讲述，但是容易将此论述扩展到 NPT 驱动。

### 9.1.1 MUX 和协议层接口

对于上层协议来说，MUX 必须为它提供发送数据的接口；同时，对于 MUX 来说，上层协议必须告诉 MUX：上层协议希望接收哪个网络接口的何种类型数据？如何将接收的数据提交给上层协议？当出现错误时，MUX 如何使上层协议知道？此外，还有其他接口需求，例如，多播时上层协议需要让底层驱动知道组地址；上层协议可能需要通过查询方式发送和接收等。具体地，上层协议和 MUX 的接口主要包括下面的内容（如果只需要了解 END 驱动和 MUX 接口，可以跳过本小节）。

## 1. 上层协议绑定 —— muxBind( )

函数 muxBind( )将上层协议和底层 END 驱动绑定，绑定的过程就是 MUX 知道上层协议希望从哪个 END 接收数据。

```
void * muxBind
(
    char * pName,           /*接口名称，如 ln，ei，... */
    int    unit,            /*接口单元号，0，1，...*/
    BOOL (* stackRcvRtn) (void* , long, M_BLK_ID, LL_HDR_INFO * , void* ),
                            /*接收回调函数*/
    STATUS (* stackShutdownRtn) (void* , void* ),
                             /*关闭协议栈的回调函数*/
    STATUS (* stackTxRestartRtn) (void* , void* ),
                             /*发送重新开始的回调函数*/
    void (* stackErrorRtn) (END_OBJ* , END_ERR* , void* ),
                             /*出错时的回调函数*/
    long   type,            /*协议*/
    char * pProtoName,      /*协议名称，可为 NULL*/
    void * pSpare           /*协议定义的指针，将传递给回调函数*/
);
```

在 muxBind( )时，pName 和 unit 参数指定了和网络接口设备绑定，type 期望的数据协议类型，标准的 VxWorks TCP/IP 协议栈绑定时指定的类型 type 在 RFC 1700 中定义，例如 0x800 为 IP 分组，即以太网帧头部结构的类型部分。如表 9-1 所示。

表 9-1　type 的特殊含义

| | |
|---|---|
| MUX_PROTO_SNARF | 上层协议要接收所绑定的网络接口上所有数据包 |
| MUX_PROTO_PROMISC | 其他绑定该网络接口的上层协议优先处理，并且如果被“处理掉”，该绑定不会收到数据包 |
| MUX_PROTO_OUTPUT | 收到所绑定网络接口的输出数据包而不是输入的数据包 |

在调用 muxBind( )时，最关键的是上层协议指定了 4 个回调函数：stackRcvRtn ( )，stackShutdownRtn ( )，stackTxRestartRtn ( )，stackErrorRtn ( )。例如，如果所绑定的 END 驱动收到一个数据包，MUX 就会回调 stackRcvRtn( )。关于这 4 个函数的原型定义，以及应该执行的操作，可以参考[WRS-npt5.4]。

调用 muxBind( )得到一个表示该绑定的“cookie”，在随后发送到该网络接口时需要该 cookie。

## 2. 上层协议发送 —— muxSend( )

上层协议发送数据非常简单，调用 muxSend( )即可。

```
STATUS muxSend ( void *  pCookie, M_BLK_ID pNBuff );
```

指定的参数为 muxBind( )时得到的返回值“cookie”和表示发送数据包的 mBlk 指针 pNBuff。pNBuff 包含链路层地址信息，调用 MUX 接口函数 muxAddressForm( )封装得到。

MUX 协议层接口使得应用程序可以构造专门的协议处理栈。遵循 MUX 协议层接口非常容易做到这一点，因为不需要了解 END 驱动的细节。另外，即使不实现专门的协议处理，有时也需要了解 MUX 协议层接口，比如要将 VxWorks 协议栈绑定到另外的 END 驱动（当存在多个网络接口时）。

上面介绍的只是最主要 MUX 协议层接口，用于使读者大致了解 MUX 和上层协议的数据交互过程。如果需要设计专门的协议处理，还需要用到其他 MUX 协议层接口，可以参考[WRS-npt5.4]和[WRS-oslib5.5]。图 9-2 表示了完整的协议层和 MUX 接口。

# 9.1.2 END 驱动和 MUX 接口

这部分是本节和 9.2 节的重点，因为我们要了解 END 驱动如何集成到 MUX。和 BSD 驱动不同的是，END 驱动只需要将数据包交给 MUX 就可以了，而 BSD 驱动需要将数据报交给特定的协议层接收者处理。图 9-2 表明了 MUX 和底层 END 驱动的常用接口函数。

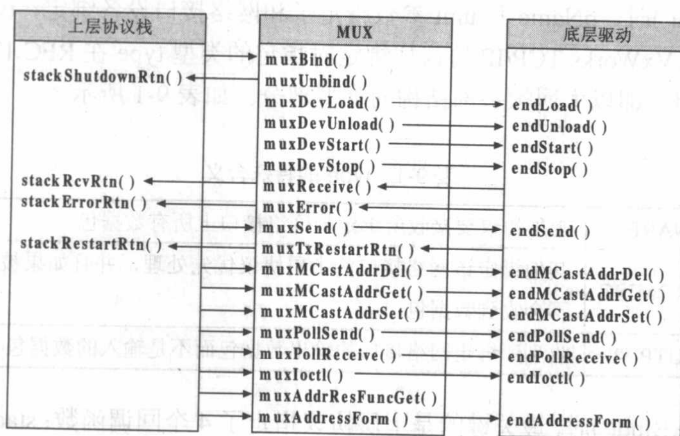



图 9-2　MUX 主要接口

总体看，MUX 的驱动接口需要解决如下问题：

- 任何一种 VxWorks I/O 驱动都有装载过程，因此 MUX 必须提供 END 的装载，以及启动接口；

- 对于 END 驱动收到的数据包，MUX 必须提供给 END 提交的方法；
- MUX 必须知道如何将数据包输出给 END 驱动；
- 其他问题：多播时加入组，离开组的支持，查询式 I/O 支持等。

解决上面的问题首先要考虑 MUX 驱动接口的设计（其他需要考虑的设计后叙）。具体来说，在 MUX 一方，要求 END 驱动提供如表 9-2 所示接口：

**表 9-2　END 驱动接口**

| | |
|---|---|
| 装载/卸载驱动程序 | endLoad( )，endUnload( ) |
| 启动/停止网络接口 | endStart( )，endStop( ) |
| 将数据输出到网络接口 | endSend( ) |
| 查询输入输出接口 | endPollSend( )，endPollReceive( ) |
| 多播接口 | endMCastAddrDel( )，endMCastAddrGet( )，endM CastAddrSet( ) |
| 设备控制接口 | endIoctl( ) |
| 链路层地址 | endAddressForm( )（即在数据包前添加链路层地址） |

注：具体不同的 END 驱动的接口函数名称通常不是以 END 开始，例如 MPC860 一个 BSP 对 cpm 网络接口的 endLoad( ) 函数名称是 sysMotCpmEndLoad( )，但是这不影响讨论。

在这里，存在一个问题：MUX 被设计为支持“任意”个 END 驱动，那么 MUX 如何知道每个 END 驱动提供的接口函数地址？和这一问题相关的还有一个细节，即 END 驱动如何将收到的数据包提交给 MUX，实际上的做法并不是图 9-2 中所表示的那样直接调用，由 END 驱动调用 muxReceive( )。当然可以这样做，因为 MUX 只有一个，END 驱动总能正确地访问 muxReceive( )。

在回答这个问题之前，我们先来看如何装载 END 设备驱动。同任何其他设备一样，装载设备驱动是使用 END 设备的第一步。

# 9.2　装载 END 驱动

每个 END 驱动程序都设计了一个函数 endLoad( )用来装载自己。装载过程由一个任务调用 MUX 的装载函数 muxDevLoad( )实现，endLoad( )就是给 muxDevLoad( )的一个参数。通常实现装载的任务就是系统启动时的根任务 tUsrRoot。根任务 tUsrRoot 执行的装载是自动进行的，只要 BSP 中进行了配置。也可以手动加载，即在普通用户任务中调用 muxDevLoad( )加载，本质上和根任务中的加载是一样的。

## 1．BSP 中的配置

决定 tUsrRoot 如何装载 END 驱动的参数在<install-dir>/target/<bspname>/configNet.h 中定义。该文件包括一个 END 设备列表“endDevTbl”，endDevTbl 记录了 END 的装载函

数 endLoad( )，不同的 END 设备驱动的装载函数定义为不同的名字，如 xxxEndLoad。下面是 MPC860 的一个 BSP 定义的 END 设备列表。

```
/*defined in <install-dir>/target/<bspname>/configNet.h */
END_TBL_ENTRY endDevTbl [] =
{
   { 0, CPM_LOAD_FUNC, CPM_LOAD_STRING, 1, NULL, FALSE},
#ifdef INCLUDE_MOT_FEC
   { 1, FEC_LOAD_FUNC, FEC_LOAD_STRING, 1, NULL, FALSE},
#endif
   { 0, END_TBL_END, NULL, 0, NULL, FALSE},
};
```

列表允许定义多个 END 设备。表中每项的第 1 列表示 END 设备单元号，例如，对于上表中第 1 项，其单元号为 0 的设备，则设备装载后的名称将是 devicename0。接下来的第 2 列和第 3 列表示装载函数 XXX_LOAD_FUNC 和装载字符串 XXX_LOAD_STRING(下面分别介绍)，最后一列控制是否让系统自动加载：如果为 FALSE，系统将加载该项，然后将 FALSE 置为 TRUE；否则为 TRUE 表示已经加载了。如果要避免系统自动加载，将表项中的 FALSE 置为 TRUE 即可，这时可以手动加载，如表 9-3 所示。

**表 9-3　定义 END 设备**

| | |
|---|---|
| XXX_LOAD_FUNC | 定义 END 设备的装载函数 endLoad( )，例如：<br>#define CPM_LOAD_FUNCsysMotCpmEndLoad<br>表示 END 设备驱动装载函数为 sysMotCpmEndLoad( ) |
| XXX_LOAD_STRING | 一个字符串，传递给 muxDevLoad( )然后直接转给 END 驱动装载函数，如 sysMotCpmEndLoad( )，因此该字符串语法完全由驱动程序实现两者定义，但一般包括物理设备号、中断向量号、寄存器地址映射等信息 |

& 上述 END 设备列表 endDevTbl 只在根据 BSP 生成可引导项目时加入项目中。因此，如果和我们的例子中一样，endDevTbl 有宏定义控制的条件编译时，必须在生成项目之前修改宏定义才有效。以上面的例子为例，如果想使用 END 设备 INCLUDE_MOT_FEC，则应该在该 BSP 的 config.h 中定义 INCLUDE_MOT_FEC。

### 2. 装载和启动 END 驱动

在 Tornado 集成环境下根据 BSP 生成了可引导项目后，新项目 usrRoot( )将会对 endDevTbl 中所有的项目进行装载。图 9-3 表明了一个完整的网络接口初始化过程：usrRoot( )，usrEndLibInit( )，再到 usrEndLibInit( )完成 END 设备的装载和启动。

图 9-3 中，为 muxDevLoad( )指定的参数即在 endDevTbl 中指定的项目，包括装载函数，

单元号等，如果装载成功，立即调用 muxDevStart( )启动网络接口。这两个函数可以参考[WRS-oslib5.5]。

```
prjConfig.c : usrRoot( )
        prjConfig.c : usrNetworkInit( )
        prjConfig.c : usrNetProtoInit( )
        muxLibInit( )
        comps/src/net/usrEndLib.c : usrEndLibInit( )
        prjConfig.c : usrNetworkBoot( )
        prjConfig.c : usrNetAppInit( )

usrEndLib.c : usrEndLibInit( )
    for pEndDev =endDevTbl 第 1 项 pEndDev 表示一个有效 END 设备; PEndDev++;
    {
      if pEndDev->processed = FALSE
        if nuxDevLoad (pEndDev...) ! = NLLL
          nuxDevStart (...) ;
    }
```

图 9-3 网络接口初始化过程

现在可以回答前面提出的问题了，即 MUX 如何知道每个 END 设备驱动的接口函数。在上面 muxDevLoad( )装载 END 设备驱动时，endLoad( )会创建一个驱动程序数据结构“END_OBJ”，该结构内部记录了所有 END 驱动和 MUX 的接口函数；endLoad( )返回指向该数据结构的指针，这样，MUX 就据此指针可以访问 END 驱动程序提供的接口函数了。同时，END_OBJ 数据结构内部还有一个函数指针字段 receiveRtn，在 muxDevLoad( )成功的情况下，MUX 已经将该字段填写为 MUX 接收 END 数据包的函数，即 muxReceive( )。END 驱动保证收到一个完整有效的数据包后调用 receiveRtn( )来提交给 MUX。

至此，我们已经说明了 END 驱动和 MUX，以及 MUX 和上层协议之间的接口。读者可以参考第 7 章 7.2 节“网络数据流分析”以获得从静态的接口到动态的数据流的完整概念。

弄清楚上述过程之后，实现多网卡驱动是非常容易的事情，如果 BSP 已经提供了多网卡的驱动，则通常只需要使能多网卡驱动即可（默认情况下 BSP 可能只会允许一个网卡驱动）。“使能”即配置 BSP 的相关条件编译来设置 endDevTbl。当然，使能多网卡驱动只是其中一步，完成这一步意味着系统中有了这样一个 END 设备了。要使协议栈通过该 END 设备来接收和发送，还需要绑定该网卡，这一步可以简单调用 ipAttach( )即可，ipAttach( )将 TCP/IP 协议栈绑定到该 END 设备，该函数内部通过 muxBind( )实现，我们前文曾对协议栈绑定进行了介绍。另外，还需要增加对宏 IP_MAX_UNITS 和 MUX_MAX_BINDS 的定义，如它们的名称所表示的，这两个宏定义了最大允许的网络接口数目和绑定数目。

同时，总结上述过程，可以看出在 MUX 机制下，实现专门的协议栈和编写自己的 END 驱动都比较简洁，这一点比 BSD 驱动具有明显的优越性。编写新的 END 驱动可以参考 END 驱动模板：<install-dir>/target/src/drv/end/templateEnd.c，以及 WRS 手册[WRS-npt5.4]。

# 第 10 章　BSP 概述

VxWorks 的一个长处是为用户应用代码提供高度的体系和硬件独立性。这得益于 VxWorks 模块化的设计，将所有硬件相关的功能都放在一系列称为**板级支持包 BSP**（Board Support Package）的库中实现，BSP 为各种开发板硬件（CPU，存储器，I/O，定时器，通信口）的功能提供相同的软件接口，包括硬件初始化，中断处理，硬件时钟和定时器管理，地址映射，内存有大小确定等。

大概来讲，VxWorks 移植化工作有如下几种：

- 主机移植 —— 移植 Tornado 和 VxWorks 到一个未被支持的开发环境下；
- 体系移植 —— 移植 VxWorks 和 Tornado 调试器到某种未被支持的类型或者体系的目标处理器；
- 板移植 —— 移植 VxWorks 到一个新的目标板。开发主机和该目标板的处理器体系已经被移植支持；
- 移植一个可选组件到某个 Tornado 发行版本。

主机移植和体系移植要求涉及整个对 VxWorks 内核的修改，终端用户和一般的 BSP 开发厂商通常不会涉及这种移植。本书讨论的是板移植。板移植只需要涉及依赖硬件的部分代码（VxWorks 和 debugger）。这些代码也就是 BSP 的组成部分。

本章从不同的角度概要地介绍 BSP 的基础知识、原理和开发技术。

## 10.1 BSP 功能

BSP 在硬件驱动程序和操作系统之间提供了一个标准接口，通过该接口，操作系统内核和应用程序可以使用各种硬件资源：设备控制器，CPU，存储器，局部总线和外部总线。图 10-1 表示了 BSP 在整个目标系统中所扮演的角色。

BSP 所提供的功能可以概括为两部分：初始化和驱动程序支持。

### 1. 初始化

初始化是指从系统上电复位开始直到内核和根任务（usrRoot）启动的这段时间的执行过程。

初始化包括：CPU 初始化、目标板初始化和内核初始化。CPU 初始化是 CPU 内部寄存器的初使化，禁止中断。目标板初始化是内存资源，各种控制器，以及各种硬件设备的初始化。内核初始化为系统的运行准备数据结构，以及各种软件库初始化，如图 10-2 所示。

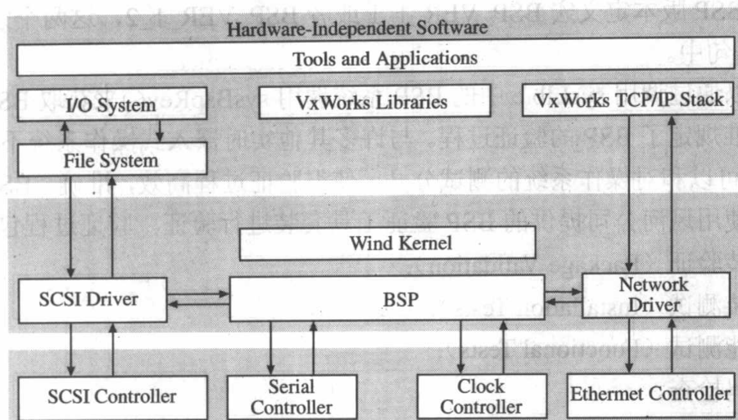



图 10-1　BSP 功能

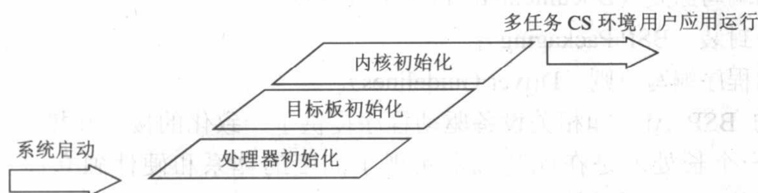



图 10-2　系统初始化

### 2．驱动程序

驱动程序负责对硬件设备的初始化，并与硬件设备交互实现系统对设备功能的调用。在 VxWorks 中，设备分“纯设备”和 BSP 控制设备。纯设备的驱动适用于多种 BSP，具有一定的通用性。WRS 已经实现的此类设备驱动程序放在目录<install-dir>/target/src/drv 和<install-dir>/target/h/drv 下。此外，特定于某种 BSP 的设备，由其 BSP 进行驱动，并且驱动程序位于该 BSP 目录下。

## 10.2　BSP 标准规范

从 1.1 版的 BSP 开始，BSP 都遵循一个标准，为 VxWorks 提供了一个稳定的编程接口 API。

BSP 对版本号定义有一定的规则。1.0 版 BSP 可以在 VxWorks5.2（含）以下版本中使用。1.1 版 BSP 可以在 Tornado1.0 和 Tornado1.1 中使用。1.2 版 BSP 可以在 Tornado2.x 中使用。1.1 版 BSP 和 1.2 版 BSP 接口上区别主要是对网络协议栈的支持和 BSP 包装差异。在每个版本号内，修订号随着每次发行所作修订从 0 开始增加。

定义 BSP 版本号包含两部分宏定义：字符串宏定义 BSP_VERSION 主要用于打印输出；

另外要根据 BSP 版本定义宏 BSP_VER_1_1 或者 BSP_VER_1_2，这两个宏主要用在#if 和#ifdef 测试语句中。

用户可以通过调用 sysLib.c 中的 BSP 系统调用 sysBspRev( )来获取 BSP 版本字符串。

BSP 标准规定了 BSP 的验证过程。与许多其他实时嵌入式操作系统不同，在 VxWorks 下验证 BSP 可以和对操作系统的测试分开，使得验证过程高效，准确。BSP 开发者和 BSP 用户都可以使用风河公司提供的 BSP 验证工具套装进行验证。验证过程包括如下测试：

- 封装验证（Package Validation）；
- 安装测试（Installation Tests）；
- 功能测试（Functional Tests）；
- 代码检查。

BSP 标准还包括如下 4 种开发人员所应遵循的惯例：

- 代码书写规范（Coding Conventions）；
- 文档编写原则（Documentation Guidelines）；
- BSP 封装（BSP Packaging）；
- 驱动程序编写原则（Driver Guidelines）。

标准化的 BSP API 和相关设备驱动程序提供了一致化的接口和模块化设计结构，VxWorks 的一个长处就是在此基础上实现了高度的体系和硬件独立性。应用代码和 VxWorks 可以从一种体系移植到另一种体系。

## 10.3 BSP 组织结构

BSP 将特定于硬件的程序分离到一组库来向上提供统一的软件接口，稍后将对常用的库进一步说明。图 10-3 是支持 Motorola MPC8260 的文件及其相对位置。BSP 包括目标系统上电时要进行的初始化和硬复位后执行的操作、中断以及其他一些硬件管理。

其他设备驱动程序和相关的支持服务可以包含在 BSP 库，来扩展嵌入式系统对定制硬件设备服务的抽象，包括：网络功能、安全、存储、图形和 I/O。BSP 硬件设备可以位于单板机 SBC 上，也可以是片上系统 SOC，或者是通过无线方式或者背板硬件总线连接的外设。

目录 target/config/all 下包括下列文件：

- configAll.h —— 通用配置文件。设置所有 VxWorks 映像的默认配置。用户在 config.h 中的相同定义能够覆盖此中的设置；
- bootInit.c —— 各种可引导映像的映像装入部分；
- bootConfig.c —— BSP 引导映像初始化部分（在 bootInit.c 的初始化之后）；
- dataSegPad.s —— 代码段保护；
- usrConfig.c —— VxWorks 系统初始化代码。

目录<install-dir>/target/config/bspname 下包含运行 VxWorks 所需要的依赖特定目标板

硬件的文件，包括：

- Makefile 和 depend.<bspname> 控制映像的生成；
- sysLib.c 系统依赖库，主要有接口设备初始化；
- sysSerial.c 串口控制器初始化；
- sysALib.s VxWorks 可装入映像入口（sysInit）；
- romInit.s 系统上电初始化入口代码；
- <bspname>.h 硬件相关的目标板寄存器，中断等的定义；
- config.h 针对目标板的系统定义和包含文。

每个发行的 VxWorks 都包含两个特定于目标系统的库文件：sysLib 和 sysALib，这两个文件是 VxWorks 可移植性的关键。这两个库文件提供和各种目标板所有硬件功能一致的软件接口，包括：硬件初始化、中断处理和硬件生成、硬件时钟和定时器管理、管理局部和总线内存空间、确定物理内存大小等。

BSP 所有的源和头文件位于如下目录<install-dir>/target/config/all 和<install-dir>/target/config/<bspname>下，如图 10-3 所示。

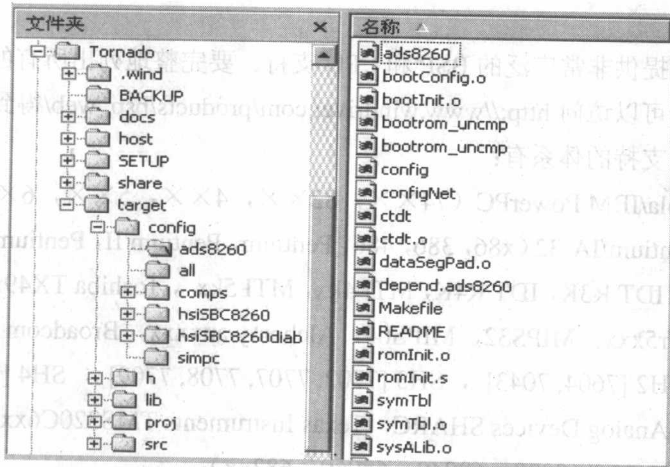



图 10-3　BSP 组织

## 10.4　BSP 支持主机/目标系统交叉开发环境

如图 10-4 展示了一个典型的嵌入式编程交叉开发模型。目标系统 BSP 通过一种 Tornado 集成开发主机环境和嵌入式目标系统之间的客户机/服务器通信协议，为开发人员提供下载，调试，测试支持，实现了跨越各种目标体系进行集成开发的能力。

BSP 为主机**目标服务器**（Target Server）提供到**调试代理**（WDB agent）的接口，主机上的 Tornado 工具集和目标服务器之间通过 WTX 协议通信；目标服务器和调试代理之间通过网口，串口，JTAG 定制驱动程序通信。当嵌入式应用开发完成，要形成产品时，调

试代理就可以从 BSP 中去掉。

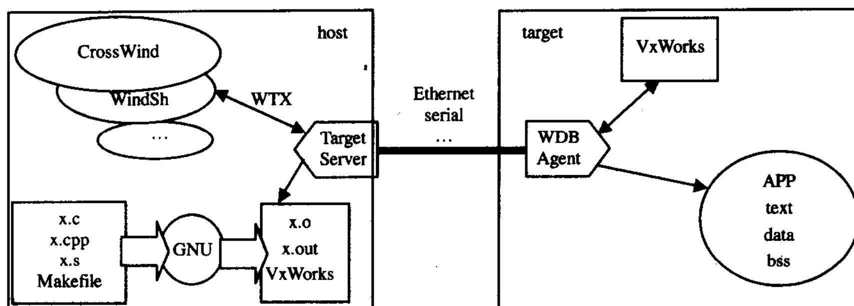



图 10-4　BSP 支持主机/目标系统交叉开发环境

## 10.5　BSP 允许将应用系统移植到其他体系下

Wind River 提供非常广泛的 BSP 和 CPU 支持。要完整地列出所有的 BSP 配置和特性需要很长篇幅。可以访问 http://www.windriver.com/products/bsp_web/得到 BSP 更详细的信息。目前已经被支持的体系有：

- Motorola/IBM PowerPC（74××，82××，4××，5××，6××，7××）；
- x86/Pentium/IA-32（x86，386，486，Pentium，Pentium II，Pentium III，Pentium IV）；
- MIPS（IDT R3K，IDT R4K，MTI 4kx，MTI 5kx ，Toshiba TX49xx，NEC Vr4xxx ，NEC Vr5xxx，MIPS32，MIPS64，Alchemy auxxxx，Broadcom BCMxxxx，）；
- SH（SH2 [7604, 7043] ，SH3 [7702, 7707, 7708, 7709] ，SH4 [7750]）；
- DSP（Analog Devices SHARC，Texas Instruments TMS320C6xxx）；
- 68K/CPU32（68030，68040，68060，683xx）；
- ARM（ARM7，ARM7 Thumb，ARM9 Thumb，ARM9 others）；
- ColdFire（MCF5xxx）；
- SPARC（Sun UltraSPARC-II）；
- M*Core（MC2xxx）；
- XScale/StrongARM；
- i960。

广泛的体系支持使用户可以选择合适应用特点的 BSP 开发目标系统，节约开发时间，提高系统效率。用户也可以自己开发 BSP。目前 VxWorks 下 BSP 已经形成了标准接口，用户自行开发 BSP 要求熟悉 BSP 标准，熟悉 BSP 对上层提供支持的细节。以下一些章节将提供这方面的知识。

# 10.6　模板和参考

用户可以开发新的BSP。BSP开发是比较复杂的过程，一般需要有一个该体系的模板BSP和一个参考BSP。

BSP模板，以及所用到的设备的驱动程序模板，是开发BSP的起点。BSP模板是一个完整但是几乎所有的可选功能都被禁用了的BSP。可以直接复制一个BSP模板目录，不需要做什么修改就可以编译，并使它开始工作，简单快速。

作参考的BSP所用的处理器和要开发的目标系统所使用的处理器相同，可以用来互相使用，以提高效率。

风河公司提供的BSP开发包含有各种体系的BSP模板，以及所有类型设备的驱动程序模板。

# 10.7　设备驱动开发中需要考虑的问题

有时需要在目标系统中增加不被WRS和BSP支持的新设备，现在许多开发板都允许用户进行这种扩展。开发驱动程序的细节根据不同的设备特点而差异很大，这里我们简单列出一些和系统结构相关的问题，以及需要考虑的策略性问题。

## 1．地址空间映射：内存空间和I/O空间

对I/O芯片的访问，不同体系的CPU有两种编址方式，即按内存空间编址（统一编址）和I/O编址。

在统一编址方式下，设计人员决定如何将芯片的寄存器映射到内存空间，程序对芯片的访问指令和访问内存的指令是一样的。对一块芯片可以有不同的映射方式，比方对一个有4字节宽度寄存器的串行芯片，可以将其映射为4个连续的按字节访问的地址，或者在16位系统上将其映射为2个半字，通过16位总线的低8位访问，还可以在32位系统上映射为长字空间。在这种情况下，驱动程序不能假定一个字节读操作可以读该芯片上的一个字节寄存器。一个解决办法是：访问芯片的操作应当尽可能简单，并且使用预处理宏定义实现，这样，在特定的系统上只要重定义宏，无需修改驱动程序代码。

C语言和各种体系的处理器都支持统一编址的映射方式。要注意高速的内存芯片和某些老式I/O设备速度上的差异。

有些体系的处理器支持单独的I/O编址，访问I/O芯片使用专门的I/O指令。这样C语言编程时代码有点特殊，必须嵌入汇编代码实现对I/O寄存器的访问，因为C语言本身不支持I/O地址空间。一般将访问I/O的这段代码用宏实现。

### 2．处理器与总线

总线实际上是一组导线，是各种公共信号线的集合，用于作为嵌入式系统中所有各组成部分传输信息共同使用的“公路”。

- 数据总线（DB）—— 数据总线用来传输数据信息，是双向总线，CPU 既可通过 DB 从内存或输入设备读入数据，又可通过 DB 将内部数据送至内存或输出设备；
- 地址总线（AB）—— 地址总线用于传送 CPU 发出的地址信息，是单向总线。目的是指明与 CPU 交换信息的内存单元或 I/O 设备；
- 控制总线（CB）—— 控制总线用来传送控制信号、时序信号和状态信息等。CB 中每一根线的方向是一定的、单向的，但作为一个整体则是双向的。在各种结构框图中，控制总线 CB 以双向线表示。

在嵌入式系统中，存在多种总线：CPU 局部总线，VMEbus、VXI、Qbus、Sbus、PCI 局部总线等。有些一个系统中存在多种总线，总线使用不同于 CPU 的端序（大端字节序和小端字节序）。驱动程序设计灵活性的目标之一是使其做最小的修改能在任何总线上工作。对不同总线，定义不同的字节交换宏。

设计驱动程序需要考虑的与总线特点相关的问题包括：

- 总线周期（同步，复用等）；
- 总线仲裁（总线锁定，优先级等）；
- 地址映射（映射为内存还是映射为 I/O）；
- 数据属性（宽度，端序等）；
- 中断策略（中断生成，应答，中断传送）。

### 3．中断

理想情况下，BSP 负责连接中断到中断向量，设备驱动程序不管其中细节，但是驱动程序提供 ISR，该函数属于全局符号，BSP 在 sysHwInit2( )中将其连接到中断向量。如果驱动程序必须涉及连接中断向量，则应该通过宏定义来实现灵活的硬件抽象，使不同中断结构的不同开发板能被该驱动程序支持。

### 4．驱动程序中的同步机制

驱动程序和应用级任务必须采取某种同步机制。最常用的是信号量。VxWorks 提供了丰富的信号量类型。

驱动程序和设备可以采取中断方式，或者查询方式，在每种方式下，都可以通过信号量进行同步。

和驱动程序的查询任务同步的操作如图 10-5 所示。

（1）任务 A 调用 semTake( )，等待某个事件：设备状态变化，收到数据，写设备就绪，出错条件等；

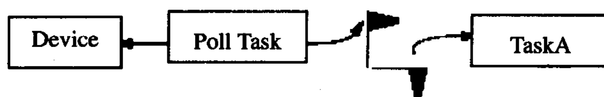



图 10-5 和驱动程序的查询任务同步

（2）查询任务检测到事件后，调用 semGive( )，使任务 A 从 semTake( )调用返回；

（3）或者驱动程序需要同步多个任务调用 semFlush( )。

和驱动程序的 ISR 同步的操作如图 10-6 所示。

图 10-6 和驱动程序 ISR 同步

（1）任务调用 semTake()，阻塞；

（2）设备产生中断；

（3）ISR 判断任务要求的条件满足，调用 semGive( )；

（4）任务从 semTake( )返回。

### 5. 数据缓冲

驱动程序在设备和应用程序传递数据，可以通过系统内核数据结构（管道和消息队列）实现，或者通过共享变量（共享缓冲区，环形队列，链表等）实现。

用系统内核数据结构提供了任务同步功能。如果使用任务间共享内存数据结构、环形队列及链表，则需要任务采用其他机制以确保任务间同步和互斥访问共享变量。

# 第11章　VxWorks映像

## 11.1　符号表

VxWorks 应用有两种运行方式：混合方式和独立（Standalone）方式。

- 混合方式 —— WindRiver 的设计哲学不是设计单一系统来完成所有的任务，而是使用 VxWorks 嵌入式实时系统来完成关键任务，实现底层和前端控制，另外使用主机操作系统提供的强大功能完成程序开发，用户控制接口，数据统计分析等非实时性问题；
- 独立方式 —— 可能在许多应用场合常常是这样：在主机的交叉开发环境中进行编辑、编译、链接、下载程序代码，然后在 VxWorks 上调试运行。运行中额外使用一台主机显得多余。如设计一个 cisco2500 那样的路由器。这时启动目标系统运行不再需要主机或者网络支持，VxWorks 映像已经被固化到 ROM/FLASH 或者磁盘上，引导程序直接从目标系统就可以引导。

符号表包含许多条目用来关联目标模块中符号名称/类型和对应的值。主机工具通过驻留主机的 Tornado 目标服务器进行动态链接和符号调试时，需要使用 VxWorks 符号表文件，该文件由随 Tornado 提供的工具 xsym 创建。Xsym 对一个目标文件进行处理会形成一个新的对象模块，该模块就是源文件符号表，但不含源文件生成的任何代码和数据。在 makefile 中，如下命令行执行上述过程：xsym < vxWorks > vxWorks.sym。通过 Tornado IDE 构建时，用户只需要指定所要求的映像类型，不需要手工维护 makefile。

符号表文件和生成的 VxWorks 映像在相同的目录下，目标服务器开始运行时装入该文件。因此，在主机运行的工具不需要驻留目标系统的符号表。例如我们可以在主机的 shell 下运行“sp mytask”，这里，主机 shell 会直接在 xsym 生成的符号表里查找符号“mytask”，并不要求在目标系统中找到符号“mytask”。

为了方便通过 Target Shell 调试，在目标机上运行的 VxWork 也可以提供符号表支持(在用到了驻留目标系统的工具时会要求符号表驻留目标系统)。这样 VxWorks 运行时符号表驻留目标系统。同样，如果目标系统有符号表，我们可以在目标系统的 shell 下运行“spmytask”。使目标系统支持符号表需要定义 INCLUDE_SYM_TBL，此时需要指定符号表来源：

- INCLUDE_NET_SYM_TBL —— 目标系统通过网络从主机下载符号表；
- INCLUDE_STANDALONE_SYM_TBL —— 目标系统使用自含的符号表。

当使用主机下载的符号表时，称“**Tornado 符号表**”；而称目标系统使用自身符号表为

“**独立符号表**”，我们后文介绍构建 VxWorks 系统映像时将需要区分这两个概念。

不论哪种符号表，都需要占用一定内存，在构建成品的发行版本时要去掉符号表支持（除非成品需要使用一个像 shell 这样的工具来执行命令）。

使用独立符号表时，不需要主机和网络支持，构建映像时，链接器会将符号表和 VxWorks 映像链接在一起。

从上文可以看出，存在着主机符号表（总存在）和目标系统符号表（需要定义）之分。有时候，还需要在两者之间进行**符号表同步**。符号表同步源于这样的需求：项目中使用了可下载应用模块，在该模块重新编译并下载到目标系统后，如果不同步，目标系统驻留工具将无法找到编译后的符号，因为符号地址都已经改变，还可能增加了新符号。让目标系统和主机同步符号表非常简单，在前面的基础上，增加定义目标系统的同步符号表组件 INCLUDE_SYM_TBL_SYNC，同时在启动主机的目标服务器（Target Server）时指定进行符号表同步（确定启动目标服务器时有“-s”选项），如图 11-1 所示。

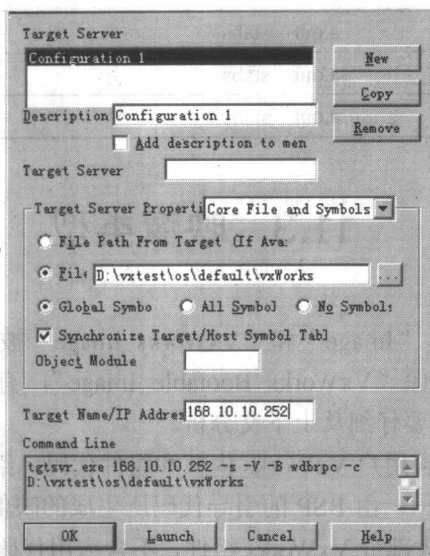



图 11-1　同步符号表

对于最终发行的成品，一般即不需要包括符号表，也谈不上符号表同步，即同时去掉对 INCLUDE_SYM_TBL 和 INCLUDE_SYM_TBL_SYNC 的定义。

## 11.2　目标模块格式（OMF）

不论何种 VxWorks 映像，都是由以下几部分构成的：

- 代码段 —— 指令；
- 数据段 —— 初始化的全局变量和静态变量；

- BSS 段 —— 未初始化的全局变量和静态变量（ANSI C/C++要求 BSS 段初始化为 0，VxWorks 对此支持）。

Tornado 2.2 支持下列目标模块格式，如表 11-1 所示。GNU 和连接器也支持其他 OMF，但是没有被支持。

表 11-1　目标模块格式

| 处理器体系 | 支持的 OMF |
| --- | --- |
| ARM | coff　stabs |
| i960 | coff　COFF |
| CPU32 | a.out　stabs |
| MIPS | elf　stabs |
| PowerPC | elf　stabs |
| 68K | a.out　stabs |
| SPARC | a.out　stabs |
| X86 | a.out　stabs |

## 11.3　映像类型

在风河公司的文档中，“Image”和“VxWorks Image”被当成“万金油”使用。有时它指称 VxWorks 引导映像“VxWorks Bootable Image”，有时指称“loadable/ROMable/ROM-resident Image”，需要仔细从上下文分析。

下面按照映像是否包含用户 VxWorks 内核和组件代码，给出更精确的分类：

- **BSP 引导映像** —— 由 BSP 的引导代码所生成的映像，只有基本的 wind 多任务微内核，不包括大部分 VxWorks 组件，不含应用代码。能引导目标系统，用于装入 VxWorks 系统映像；
- **VxWorks 系统映像** —— 用户通过定制 VxWorks 和进行应用程序开发所生成的映像，该映像也可能包括 BSP 的引导代码。VxWorks 系统映像包括需要 BSP 引导映像才能装入的可装入映像，还包括 ROM 化的可以自行引导目标系统的 VxWorks 映像。

因此，上面两种映像都可能引导系统，但是 BSP 引导映像除了引导系统没有其他功能。对于每种映像，又因其存在和运行方式不同而区分不同子类型。

两种映像都涉及两种映像属性：（1）映像运行时是否驻留 ROM；（2）映像是否压缩。第（1）个属性是为了节约使用 RAM，而第（2）个属性是为了解决 ROM 容量受限的问题。同时 VxWorks 系统映像还有其他属性（符号表特点，是否可下载等），将在后文介绍。

## 1. 驻留 ROM 的映像

驻留 ROM 的映像可以减少对 RAM 的使用，运行时只将数据段装入 RAM 中，代码段驻留 ROM。BSP 引导映像和 VxWorks 系统映像都可以驻留 ROM 或者不驻留 ROM，图 11-2 表明了两种映像在 ROM 中的格式，以及装入数据段到 RAM 时的位置。

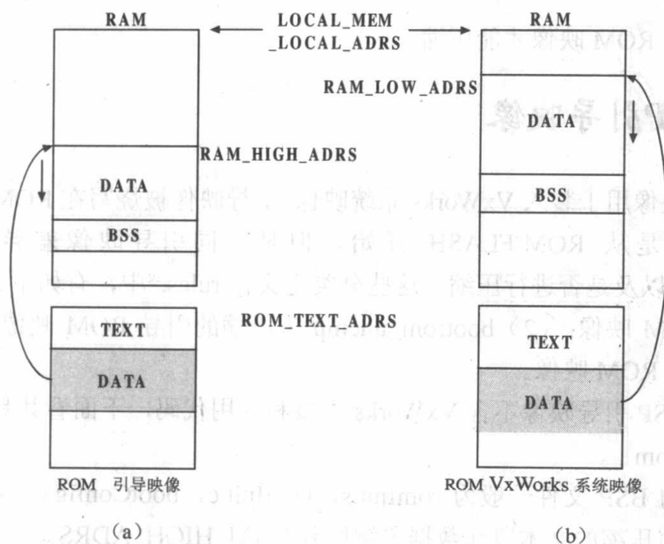



图 11-2　ROM 映像

除了节约 RAM 占用之外，映像驻留 ROM 的缺点在于降低了运行速度，同时限制了访问 RAM 的数据宽度。

对于使用独立符号表的 VxWorks 系统映像，驻留 ROM 将节约 300KB 以上 RAM 开销。

驻留 ROM 映像对 ROM 大小的建议是：VxWorks 系统映像 512KB（含独立符号表），BSP 引导映像 256KB。如果应用代码和 VxWorks 一起驻留 ROM，可能需要更大的 ROM。

驻留 ROM 的 VxWorks 系统映像装入数据段时装入位置为 RAM_LOW_ADRS，而 BSP 引导映像装入数据段的位置为 RAM_HIGH_ADRS，因为 BSP 引导映像职责在于装入 VxWorks 系统映像，RAM_LOW_ADRS 被约定为 VxWorks 系统映像位置（从 RAM_LOW_ADRS 到 RAM_HIGH_ADRS 作为 VxWorks 系统和应用程序的内存使用）。

对于有限存储系统（一般小于 1MB），需要注意 RAM_HIGH_ADRS 小于 LOCAL_MEM_SIZE，并且足够装入 BSP 引导映像的数据段。

**RAM_HIGH_ADRS**

RAM_HIGH_ADRS 在 makefile 和 config.h 中都有定义，两者必须一致。

## 2. 压缩

对于固化在 ROM 中的映像（BSP 引导映像或者固化的 VxWorks 系统映像），映像压

缩可以将其尺寸减小到未压缩的40%左右。解压缩采用标准解压缩程序 inflate( )完成，该程序压缩效率通常在55%以上。

压缩后的映像开始有一部分未压缩的引导码（约 8KB）。后文将介绍这部分未压缩的引导码为 romInit 和 romStart，它们将自身一起和其他所有压缩的部分解压缩到 RAM 后跳到解压缩后的入口执行。根据不同的处理器速度及代码大小，解压缩过程将使启动时间增加数秒。

只有非驻留 ROM 映像才能压缩。

## 11.3.1 BSP引导映像

BSP 引导映像用于装入 VxWorks 系统映像。引导映像被烧写在 ROM 或者装入 FLASH 中，执行时总是从 ROM/FLASH 开始，但是不同引导映像差异体现在是否驻留 ROM/FLASH，以及是否进行压缩。这些分类定义在 rules.*中，有如下 3 种：（1）bootrom 压缩的引导 ROM 映像；（2）bootrom_uncmp 未压缩的引导 ROM 映像；（3）bootrom_res 驻留 ROM 引导 ROM 映像。

已说明，BSP 引导映像不含 VxWorks 内核和应用代码，下面看其具体的构成。

（1）bootrom

编译使用的 BSP 文件一般为 romInit.s，bootInit.c，bootConfig.c，sysLib.c。搬移程序 bootInit.c 负责将压缩的文本段和数据段解压到 RAM_HIGH_ADRS。

（2）bootrom_uncmp

编译使用的 BSP 文件一般为 romInit.s，bootInit.c，bootConfig.c，sysLib.c。搬移程序 bootInit.c 负责将文本段和数据段搬移到 RAM_HIGH_ADRS。文本段和数据段均没有经过压缩。

（3）bootrom_res

映像驻留 ROM 运行，不进行压缩，数据段搬移到 RAM_HIGH_ADRS。编译使用的 BSP 文件一般为 romInit.s，bootInit.c，bootConfig.c，sysLib.c。

### 1. 文件构成与执行过程

生成 BSP 引导映像时使用了 bootConfig.c，该文件的作用类似 VxWorks 系统映像中的 usrConfig.c，但是后者初始化了许多 VxWorks 组件。BSP 引导映像的执行过程如图 11-3 所示。

图 11-3 给出的是一个大致的过程，还有许多细节没有体现，例如初始化的步骤，映像装入的位置等，这些将在第 12 章得到说明。

图 11-3 中，汇编模块“romInit.s”和 C 模块“bootInit.c”被称为引导代码，它们总是从 ROM 开始执行。“引导”的意思在于它们从系统上电开始，实现了将被引导部分（所有其他部分）装入内存并开始被引导部分执行的过程。引导代码要求和地址无关（PIC），而被引导部分无此要求。

BSP 的 bootConfig.c（即图 11-3 中从 usrInit 到 tBoot 部分）只为了装入系统映像，因此提供的功能主要包括取得引导参数，然后下载映像并执行。同时提供一些简单的单字符命令来完成这些功能，即图 11-3 中 tBoot 任务是其重要部分，虽然前面的初始化过程比 tBoot 更复杂。任务 tBoot 职责就是取得引导参数，装入系统映像。在 tBoot 跳到装入映像的入口开始，引导映像的工作即告结束，系统映像开始其新的"初始化—多任务内核运行"的过程。

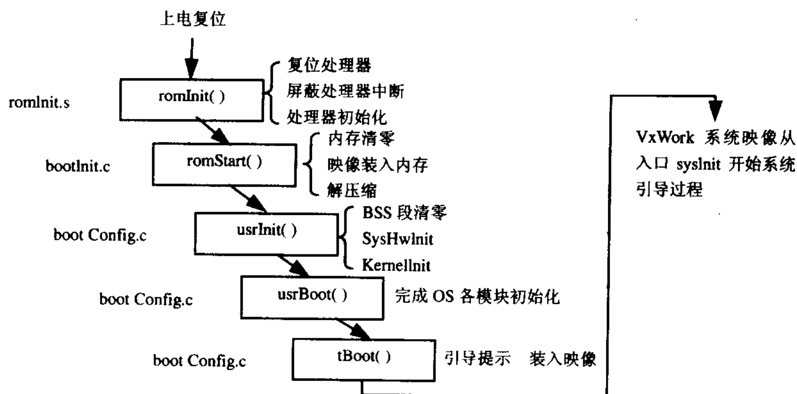



图 11-3 BSP 引导映像执行过程

实际上，也正是使用了 bootConfig.c 而不是 usrConfig.c 使其成为 BSP 引导映像而不是 VxWorks 系统映像（当然 makefile 规则有不同）。下面来看引导映像的引导参数设置。

& 通过分析 tBoot 任务的主函数 bootCmdLoop( )可以得知引导的一些细节。该函数在 bootConfig.c 中定义，但是有些 BSP 的 bootConfig.c 以目标码形式提供。

## 2. 引导参数设置

不同的引导映像支持不同的引导方式，例如网络引导、软盘引导、USB 引导等，每种引导方式都需要确定一系列参数。具体来说，引导任务 tBoot 使用一个宏定义的字符串参数 DEFAULT_BOOT_LINE 来表示这些信息，在 <install-dir>/target/config/<bspname>/config.h 中可以找到其定义。引导行参数被构建到引导映像中，使没有 NVRAM 的目标系统得到引导参数，否则必须每次输入引导参数。引导参数还可以在引导提示时改变。

引导行参数有固定的格式：

```
#define DEFAULT_BOOT_LINE  "$dev(0,procnum)host:/file h=# e=# b=# g=#
                    u=usr [pw=passwd] f=# tn=targetname s=script o=other"
```

有些项在不同的启动设备类型中不需要。各项含义为：

- $dev —— 启动设备类型；
- procnum —— 处理器序号一般从零开始；
- host —— 主机名；

- /file —— 被加载的 VxWorks 文件所在的完整路径；
- h —— 主机 IP；
- e —— 目标板 IP；
- b —— 背板 IP，用户可不定义；
- g —— 网关，用户可不定义；
- u —— 用户名；
- pw —— 登录口令；
- f —— 网络加载方式（0 表示 FTP ，0x80 表示 TFTP）。默认值为 0；
- tn —— 目标板名；
- s —— 启动描述字符串，用户可不定义；
- o —— 不从网络启动时指明网络接口。

以下是一些引导行参数的例子。

（1）引导设备为网卡 elPci。

```
#define DEFAULT_BOOT_LINE \
    "elPci(0,0)zlot:/projpentium/default/vxWorks \
     h=147.11.41.98  e=147.11.41.180:ffffff00 \
     u=mypctarget \
      s=/projpentium/script.txt  pw=myPassword"
```

引导时将会显示为：

```
boot device           : elPci
unit number           : 0
processor number      : 0
host name             : zlot
file name             : /projpentium/default/vxWorks
inet on Ethernet (e)  : 147.11.41.180:ffffff00
host inet (h)         : 147.11.41.98
user (u)              : mypctarget
ftp password (pw)     : myPassword
flags (f)             : 0x0
startup script (s)    : /projpentium/script.txt
```

（2）引导设备为软驱 Notice the use of the parameter。

```
#define DEFAULT_BOOT_LINE \
     "fd=0,0(0,0)zlot:/projpentium/default/vxWorks \
      h=147.11.41.98  e=147.11.41.180:ffffff00 u=mypctarget \
```

```
    s=/projpentium/script.txt  pw=myPassword o=elPci"
```

（3）如果主机和目标系统不在同一子网，还需要网关。

```
#define DEFAULT_BOOT_LINE \
    "elPci(0,0)zlot:/projpentium/default/vxWorks \
     h=147.11.48.28  e=147.11.41.180:ffffff00 \
     g= 147.11.41.254 u=mypctarget \
     s=/projpentium/script.txt  pw=myPassword \
     tn=PCtarget"
```

具体每种设备下如何引导，可以参考[WRS-tug2.2]。

## 11.3.2 VxWorks系统映像

VxWorks 系统映像包括 wind 多任务微内核，用户定制 VxWorks 组件，以及用户应用代码。

VxWorks 系统映像也可能包括 BSP 的引导代码（romInit.s 和 bootInit.c），这样可以将映像固化在 ROM，每次上电后即能引导目标系统，即**可引导的系统映像**，或者，根据其存在方式，称其为 **ROM 固化的系统映像**。另外一种 VxWorks 系统映像虽然也含 BSP 的引导代码，但是总是由 BSP 引导映像装入 RAM 后执行，即**可下载的系统映像**，如图 12-2（b）所示，BSP 引导映像装入可下载的系统映像后，跳到可下载系统映像的 sysInit( )开始运行。

& **BSP 引导映像和 VxWorks 系统映像**

各种不同的映像虽然简单，但是这些区分的概念非常容易让人困惑。从可引导讲，BSP 引导映像和 VxWorks 系统映像都可以引导；从 ROM 固化讲，BSP 引导映像都固化，而 VxWorks 系统映像有固化和非固化之分；从驻留 ROM 讲，BSP 引导映像和 VxWorks 系统映像都可以驻留也可以不驻留。同时，BSP 引导映像也包含多任务的 wind 内核。因此我们说，两种映像的区分更主要在于映像的目的。

### 1. 固化在 ROM 中的系统映像

固化在 ROM 中的系统映像包括根据使用符号表方式不同分为两种基本类型：

- vxWorks　　使用主机符号表；
- vxWorks.st　使用独立符号表。

进一步，上述两种基本类型是否在压缩和驻留 ROM 上存在不同。固化在 ROM 中的系统映像类型如表 11-2 所示。

表 11-2　固化在 ROM 中的系统映像类型

| 固化的系统映像类型 | 说　明 |
| --- | --- |
| vxWorks_rom | 主机符号表＋不压缩，引导时复制代码和数据段到 RAM 执行 |
| vxWorks.st | 独立符号表＋不压缩，时复制代码和数据段到 RAM 执行 |
| vxWorks.st_rom | 独立符号表＋压缩，数据段到 RAM 执行，需要较大的 ROM |
| vxWorks.res_rom | 独立符号表＋不压缩＋驻留 ROM 执行，引导时只复制数据段到 RAM |
| vxWorks.res_rom_nosym | 主机符号表＋不压缩＋驻留 ROM 执行，引导时只复制数据段到 RAM |

代码段装入：上述几种 VxWorks 系统如果不驻留 ROM 运行，则连接时实际上整个映像分为两部分：执行重定位的引导部分和映像主体部分。连接器将前者连接到地址 ROM_TEXT_ADRS，对应装入 RAM 的地址为 RAM_HIGH_ADRS，因为这两部分地址一般不一致，这部分必须被设计为地址无关的（PIC）；对于映像主体部分，连接器直接将连接到 RAM_LOW_ADRS，不存在 PIC 问题。可以参考图 11-2 得到清晰的说明。对于未压缩映像，重定位就是简单地复制代码段和数据段到 RAM 的过程；对于压缩映像，重定位就是解压缩＋复制过程。

## 2. 可下载的系统映像

可下载的系统映像不能引导目标系统，必须要 BSP 引导映像装入。如图 12-2（b）“图引导阶段”表示了一个 BSP 引导映像装入可下载映像的过程。在产品开放阶段，一般使用可下载系统映像，到产品发布时改为 ROM 化的系统映像。除了引导过程差异外，两种系统映像在操作系统运行起来之后是一样的。

上述两种 VxWorks 系统映像的执行过程，即 VxWorks 操作系统启动过程，将在第 12 章分析。

# 第12章　VxWorks启动过程

## 12.1　目的、策略与过程概述

作为一种实时操作系统，VxWorks 的启动过程需要达到的目的是：为运行用户应用初始化一个计算环境，该计算环境可能构建在各种不同的目标硬件上，根据用户定义，VxWorks 计算环境必须充分利用目标硬件资源。

那么，在启动过程面临的具体问题是，如何从目标系统上电复位开始，逐步达到这一目的。先来看，在启动之前，目标系统状态是处于不确定的混沌状态，例如各种硬件设备可能处于出错状态或者中断状态，执行映像也可能位于不同的物理介质内；在启动之后，多任务微内核精确而可靠地调度各种设备，用户任务和操作系统实时服务于外部事件。因此，启动过程中需要解决：

- 处理器初始化 —— 使处理器复位，禁止中断，内部寄存器都为确定的值；
- 代码装入 —— 将执行映像从存储介质装入 RAM 执行；
- 硬件初始化 —— 使硬件复位，并初始化为中断或查询方式为系统服务；
- 内核激活 —— 使多任务微内核开始调度任务运行；
- 操作系统组件初始化 —— 初始化用户应用需要的各种操作系统功能。

设计上面 4 个问题的解决方案时，WRS 考虑一个策略是尽量实现模块化，使 BSP 设计简化。模块化意味着：

- 处理器初始化是最初执行的，不论后面步骤的细节，处理器初始化提供一个状态确定的且中断被禁止的处理器；
- 代码装入时只关心映像源所在位置和 RAM 中目的地址；
- 初始化某目标板硬件不需要考虑处理器和其他设备细节；
- 内核激活和特定的硬件无关，不需要再去考虑硬件初始化；
- 操作系统组件初始化认为多任务微内核已经开始执行。

上面的策略体现在启动过程中就是将整个启动过程分为如图 12-1 所示几个部分。

图 12-1 是一种常见情况。以内核被激活为界，将启动过程分为两大阶段。

- 第一阶段（图中第一个阴影部分） —— 为激活多任务微内核准备了一切条件：（1）代码已经位于 RAM 中；（2）系统处于不会发生中断的静止状态（必须包括处理器中断被禁止和各个设备中断被禁止）。在该阶段，“处理器初始化”和“代码装入”即通常所说的引导代码（romInit.s 和 romStart.c）。
- 第二阶段（图中第二个阴影部分） —— 工作包括 3 部分：（1）激活多任务微内

核；（2）安装设备驱动；（3）各种组件初始化。VxWorks通过一个根任务来完成（2）和（3）的操作。对内核而言，根任务和普通的用户任务一样。由于VxWorks的wind内核是一个微内核，所有的设备驱动和常规操作系统功能都不在内核中实现，因此通过一个普通任务即根任务完成设备驱动的安装和系统组件初始化。

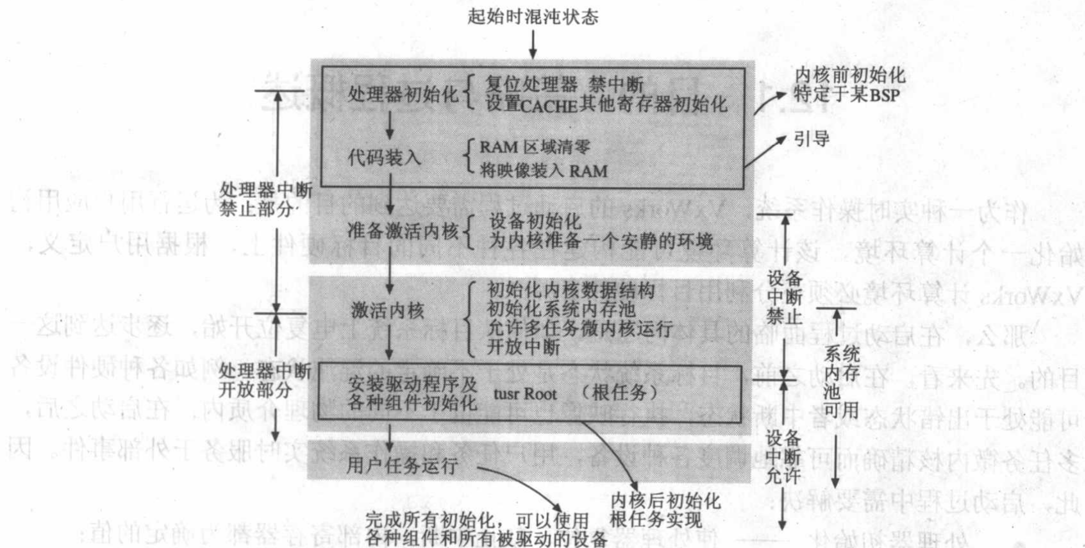



图12-1 VxWorks启动过程

在图12-1中，可以看出对设备的两次初始化，在激活内核的准备阶段，初始化主要是将设备置为静止状态避免其产生中断；在根任务中，调用设备驱动程序对设备的初始化是彻底的和面向应用的（例如如果根据应用，需要设备工作在中断方式，则为设备安装ISR，打开设备中断等）。

需要指出，以内核是否激活为界的区分并不是惟一方法。还有其他区分，如安装和目标硬件的相关性，可以将启动过程分为：处理器相关部分，特定BSP相关部分，一般化初始化部分；或者按照启动的时序，将启动过程分为：引导，激活内核，驱动和组件初始化。

WRS更多以硬件相关性区分启动过程。但是我们认为，在上述启动过程各个阶段，都直接或间接地和特定硬件配置相关。例如激活内核的初始化kernelInit()，看来是和硬件耦合程度最轻的，但是为其指定中断栈，以及系统内存大小等确是和处理器体系及BSP相关的。

下面来介绍启动过程的各个部分：引导，准备激活内核，激活内核，根任务运行。注意图12-1中的几个分界点（处理器中断禁止和开放的分界点，设备中断禁止和开放的分界点，系统内存池可用的分界点）的实现，以及它们相互之间的约束关系，将有利于更好地理解VxWorks的启动过程。同时要注意，作为对VxWorks启动过程的概括表示，图12-1是普遍适用的，即总是引导、准备激活内核、激活内核、（根）任务运行的过程。但是不同的实现会存在微小差异，尤其体现在设备初始化方面。

# 12.2 引导阶段

引导过程是上电后最先执行的部分，引导过程确保系统代码会位于正确的位置。第 11 章已经说明，在 ROM 固化的 VxWorks 系统映像和 BSP 引导映像中都含有引导代码，另外，对于不含有引导代码的 VxWorks 可装入映像，我们也认为装入它的 BSP 引导映像的执行是 VxWorks 可装入映像的引导过程。如图 12-2 所示。

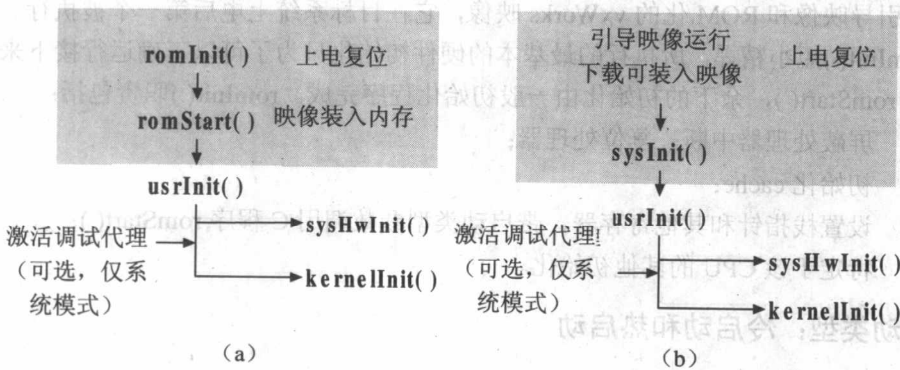



图 12-2 引导阶段

图 12-2 中阴影部分是特定于 BSP 的内核前初始化程序，包括如表 12-1 所示 3 个函数。

表 12-1 内核前初始化程序函数

| 函 数 | 执行条件 | 所在文件 |
|---|---|---|
| romInit( ) | 所有烧写到 ROM 中的映像：BSP 引导映像，ROM 化 VxWorks 系统映像 | romInit.s |
| romStart( ) | 所有烧写到 ROM 中的映像：BSP 引导映像，ROM 化 VxWorks 系统映像 | bootInit.c |
| sysInit( ) | 可装入映像 | sysALibs |

romInit( ) 始终在 ROM/FLASH 中执行，跳到 ROM/FLASH 中的 romStart( ); romStart( ) 始终在 ROM/FLASH 中开始执行。

- 驻留 ROM 映像将数据段装入 RAM，继续在 ROM 执行；
- 非驻留 ROM 映像复制启动 start-up 代码到 RAM，然后跳到 RAM 执行；
- sysInit( )对各种映像都被连接，但只对可装入 VxWorks 映像才在 RAM 中运行。

稍后还要讲到的一般化内核前初始化程序 usrInit，该程序除了对驻留 ROM 映像外都在 RAM 中执行。

**romInit( )和 romStart( )**

对各种可引导映像的启动（BSP 引导映像和固化的 VxWorks 系统映像），初始阶段的是相同的。在启动的初始阶段，要执行引导代码。引导代码包括 romInit( ) 和 romStart( )，编译连接生成可执行映像后烧写在 ROM 或者 FLASH 中。因此本章对 romInit( )和 romStart( )的介绍不作申明时适于所有可引导映像。

## 12.2.1 romInit( )

romInit( )是个和目标系统 CPU 特性直接相关的汇编程序，位于 BSP 文件 romInit.s 中。对所有引导映像和 ROM 化的 vxWorks 映像，它在目标系统上电后第一个被执行。

romInit( )短小精悍，所执行的最基本的硬件初始化只为了可以正确运行接下来的 C 语言程序 romStart( )，余下的初始化由一般初始化程序完成。romInit( )职责包括：

- 屏蔽处理器中断，复位处理器；
- 初始化 cache；
- 设置栈指针和其他寄存器，带启动类型参数调用 C 程序 romStart( )；
- 特定于该 CPU 的其他初始化。

### 1. 启动类型：冷启动和热启动

进入 romInit 时分两种情况：

- 冷启动，目标系统上电复位；
- 热启动，调用 reboot( )，^X 或者中断时异常可以进入热启动，sysLib.c 中的程序 sysToMonitor( )将控制权转到 ROM 监控程序(＝romInit( )入口处加上一个偏移量，通常为 4 字节或 8 字节。如下是一个热启动的例子：

```
STATUS sysToMonitor (
    int startType    /*该参数传递给 ROM 监控程序*/
)
{
   FUNCPTR pRom = (FUNCPTR) (ROM_TEXT_ADRS + 8);    /* Warm reboot */
   /* 其他初始化，硬件复位等 */
   (*pRom) (startType); /* 跳到 bootrom 热启动入口 */
   /* 其他操作 */
}
```

### 2. 栈指针初始化

romInit( )使用宏 STACK_ADRS 初始化栈指针 SP。随目标体系结构不同，栈的使用方式分向上增长_STACK_GROWS_UP 和向下增长_STACK_GROWS_DOWN 两种。宏 STACK_ADRS 表示栈起始地址，在头文件 configAll.h 中定义，如下是一个栈指针初始化

的例子：

```
#if  (_STACK_DIR == _STACK_GROWS_DOWN)

#ifdef   ROM_RESIDENT
#define STACK_ADRS  STACK_RESIDENT
#else
#define STACK_ADRS  _romInit
#endif   /* ROM_RESIDENT */

#else    /* _STACK_DIR == _STACK_GROWS_UP */

#ifdef   ROM_RESIDENT
#define STACK_ADRS  (STACK_RESIDENT-STACK_SAVE)
#else
#define STACK_ADRS  (_romInit-STACK_SAVE)
#endif   /*  ROM_RESIDENT */

#endif   /* _STACK_DIR == _STACK_GROWS_UP */
```

对于驻留 ROM 的 VxWorks 映像，如果栈的使用方式为向下增长，则定义为从 RAM 中数据段开始处；如果栈的使用方式为向上增长，则栈定义为 RAM 中数据段开始处减去栈大小。

对于在RAM中运行的VxWorks映像，如果栈的使用方式为向下增长，则定义为从RAM中代码段开始处（即 romInit 处）；如果栈的使用方式为向上增长，则栈定义为 RAM 中代码段开始处减去栈大小。

### 3. 地址无关性 PIC

何谓地址无关性？地址无关性代码是相对于PC的代码，PIC代码可以放在RAM另一区域也可以正确运行。romInit( )在 ROM/FLASH 中运行，必须设计为 PIC，以支持各种 VxWorks 映像的不同启动机制。要做到PIC，必须使程序中不出现对绝对符号地址的应用。所有符号地址只能是相对于程序计数器PC的相对地址或者相对于 romInit 的相对地址。通常使用如下宏计算相对于 romInit 偏移量：

```
#define ROM_OFFSET(x)   ((x) - _romInit+ ROM_TEXT_ADRS)
```

构建 VxWorks 映像时，连接器将 romInit( )静态地连接到 VxWorks 映像，连接器所指定的 romInit 地址就是驻留 ROM 映像的 ROM 地址或者非驻留 ROM 映像的 RAM 地址。因此上述宏定义能使所引用的符号地址相对于 romInit 的 PC 值，从而实现 PIC。

### 4．romInit 其他注意事项

- 不宜调用其他程序或函数。对于压缩映像，romInit 中调用其他程序或函数可能会出现问题，调用时被调用模块可能被压缩。另外，在 romInit 中调用 C 程序可能破坏 PIC 特性；
- romInit 必须是 romInit.s 中第一个出现的程序；
- 从参考 BSP 开始编写 romInit( )；
- 宏 ROM_TEXT_ADRS 和 ROM_SIZE 在 BSP 文件 Makefile 和 config.h 中都有定义，必须确保一致并且正确。

## 12.2.2 romStart( )

在 romInit( )之后执行的第一个程序是 romStart( )。这是一个 C 语言程序，定义在../all/bootInit.c 中。romStart( )在 ROM 中执行的部分必须是 PIC 的。

对 ROM 映像：romStart( )将 VxWorks 复制到内存中。

对驻留 ROM 映像：执行必要的代码重定位，解压缩，RAM 初始化。（1）复制合适的 ROM 映像段到 RAM；（2）如果是冷启动，清除没有使用的 RAM；（3）如果 ROM 映像被压缩，复制同时执行解压缩；（4）转移控制到一般化内核初始化程序 usrInit( )。

不要直接编辑 romStart( )，其功能由宏定义控制。

对于 i960 系列处理器，romStart( )结束时不是转移到 usrInit( )，而是调用 sysInitAlt( )。

### 1．映像解压缩（从 ROM 到 RAM）

为了压缩 ROM 映像能正常启动，其中的引导代码部分 romInit 和 romStart 没有被压缩。其余部分代码可能被压缩也可能未被压缩。因此本步骤实际上存在两种不同操作：

- 对于压缩映像：解压缩 ROM 中映像到 RAM；
- 对于未压缩映像：直接复制映像到目的 RAM（数据段或者代码段＋数据段），但是我们统称该步骤为映像解压缩。

解压缩分两阶段：

- 第一阶段，将未压缩部分，也就是引导代码，从 ROM 复制到 RAM；
- 第二阶段，解压缩 ROM 映像中剩下的压缩部分，并定位到 RAM 中目的位置。

第一阶段的直接复制由在 ROM 中运行的 romStart 完成，接下来转入 RAM 中刚刚复制的 romStart，并由其完成第二阶段。第一阶段是为了完成第二阶段而进行的过渡。

如果第二阶段的 RAM 目的位置是 RAM_LOW[HIGH]_ADRS，则第一阶段 RAM 目的位置为 RAM_HIGH[LOW]_ADRS，避免重叠。

对于 BSP 引导映像，上述第二阶段解压缩的代码比较少，而固化的 VxWorks 系统映像则通常包含许多内容，因此需要数秒的时间来解压缩。

本步骤后，各种映像的对应 RAM 目的地址表示如表 12-2 所示。对于未压缩映像，只

有上述解压缩过程的第一阶段，因此第二阶段目的 RAM 地址为空。

表 12-2　映像对应的 RAM 目的地址

| 映像类型 | | | 第 1 阶段 RAM 目的地址 | 第 2 阶段 RAM 目的地址 |
|---|---|---|---|---|
| B/V | 压　缩 | 驻　留 | | |
| BSP | 否 | 否 | RAM_HIGH_ADRS | |
| BSP | 是 | 否 | RAM_LOW_ADRS | RAM_HIGH_ADRS |
| BSP | 否 | 是 | RAM_HIGH_ADRS | |
| VxWorks | 否 | 否 | RAM_LOW_ADRS | |
| VxWorks | 是 | 否 | RAM_HIGH_ADRS | RAM_LOW_ADRS |
| VxWorks | 否 | 是 | RAM_LOW_ADRS | |

上表中，映像类型栏中“BSP”表示 BSP 引导映像；“VxWorks”表示 ROM 固化的 VxWorks 系统映像。如果映像驻留 ROM，则目的 RAM 地址表示数据段复制到 RAM 的地址，否则为整个映像复制到 RAM 的地址。

一个简单的确定上述各种映像的目的地址的准则只需要考虑下面 3 点。

- 如果解压缩，第 1 阶段的未压缩代码和第 2 阶段解压缩后的代码位于不同区域；
- VxWorks 系统映像在 RAM 中重定位后地址必须是 RAM_LOW_ADRS，因为连接器连接时使用该地址；
- 对于 BSP 引导映像在 RAM 中的地址必须是 RAM_LOW_ADRS，以避免装入系统映像时冲突。

## 2. 内存初始化

冷启动时，romStart 将在解压缩 ROM 映像到 RAM 之前进行内存清零。下列区域不进行清零：

- VxWorks 代码段，数据段，BSS 段；
- 保留区域 USR_RESERVED_MEM（在 config.h 中定义）；
- 保留区域 RESERVED（在 configAll.h 中定义）；
- 保留区域 STACK_SAVE（中 configAll.h 定义）。

## 3. 检查内存错误

由于多种原因，映像在内存重定位后不一定能正确访问和执行。引起错误的原因有：

- 对 RAM 的访问工作不正常；
- ROM 驻留映像的数据段在 RAM 中的重定位不正确；
- 下载过程出错；
- 程序错误。

检查上述错误，可以通过读写判断一个未初始化的全局变量。VxWorks 全局变量位于 bss 段。如果通过下述检查，则 RAM 访问基本正常。

```
int dummyVar; /* BSS segment variable */
...
dummyVar = 0xaaaa;
if (dummyVar != 0xaaaa)
somethingWrongWithRAM( );
```

### 4. 修改 romStart( )

函数的功能由宏定义控制，一般不需要修改。但是在 BSP 开发阶段，可以在 bootInit.c 中加入一些调试和诊断代码。

如果需要修改 bootInit.c，可以在其备份上进行。这时可以修改 Makefile，使编译器使用修改后的 bootInit.c 的副本，即在 Makefile 的宏 HEX_FLAGS 下面添加一行宏定义：BOOTINIT = “bootInit.c 副本名称”。默认时 BOOTINIT 的值在 defs.$(WIND_HOST_TYPE) 中定义为“<install-dir>/target/config/all/bootInit.c”。

### 5. 配置 romStart( )的宏定义

所有配置 romStart( )的宏定义在文件 config.h、Makefile、configAll.h 和 bootInit.c 中。BSP 开发人员要注意对 config.h, <bspname>.h 和 Makefile 中的宏的修改必须准确而且一致。bootInit.c 本身，包括其中宏定义不应该做修改。bootInit.c 中的宏属于：

- BSP 独立 —— 受 config.h, Makefile, configAll.h 的宏控制；
- 映像类型独立 —— 编译时 rules.bsp 控制；
- 特定于 bootInit.c。

如表 12-3 所示列出了所有配置 romStart( )的宏定义：

**表 12-3　配置的宏定义**

<table>
<tr><th>宏　定　义</th><th>含　　义</th><th>所在文件</th></tr>
<tr><td>LOCAL_MEM_LOCAL_ADRS</td><td>RAM 开始</td><td rowspan="8">config.h</td></tr>
<tr><td>LOCAL_MEM_SIZE</td><td>RAM 大小</td></tr>
<tr><td>USER_RESERVED_MEM</td><td>用户保留 RAM 区域大小，位于 RAM 顶端。该区域在冷启动时不清零</td></tr>
<tr><td>RAM_HIGH_ADRS</td><td>非驻留 ROM 的 vxworks 引导映像装入 RAM 的地址</td></tr>
<tr><td>RAM_LOW_ADRS</td><td>VxWorks 系统映像装入 RAM 的地址</td></tr>
<tr><td>ROM_TEXT_ADRS</td><td>ROM 引导代码入口地址</td></tr>
<tr><td>ROM_SIZE</td><td>ROM 大小</td></tr>
<tr><td>ROM_BASE_ADRS</td><td>ROM 起始地址</td></tr>
<tr><td>RAM_HIGH_ADRS</td><td>必须和 config.h 中同名宏定义一致</td><td>Makefile</td></tr>
</table>

续表

| 宏定义 | 含　义 | 所在文件 |
|---|---|---|
| RAM_LOW_ADRS<br>OM_TEXT_ADRS<br>ROM_SIZE | | |
| RESERVED | RAM 开始部分保留的字节数。冷启动时不被清零 | configAll.h |
| STACK_SAVE | romStart( )所能使用的最大栈空间。冷启动时不被清零 | configAll.h |
| USER_RESERVED_MEM<br>SYS_MEM_BOTTOM | 如果 config.h 中没有定义，则为 boot Init.c 定义为 0，即无用户保留区域。系统 RAM 低地址。冷启动时从该地址开始清零。宏展开后等于：LOCAL- MEM-LOCAL -ADRS+RESERVED | bootInit.c |
| SYS_MEM_TOP | 系统 RAM 高地址。冷启动时清零结束处（不含该地址本身）。宏展开后等于：LOCAL-MEM-LOCAL -ADRS +LOCAL_MEM_SIZE -USR_RESERVED_MEM | bootInit.c |
| UNCMP_RTN | 用于解压缩的程序地址 | bootInit.c |
| ROM_OFFSET | 计算绝对符合的 PIC 地址 | bootInit.c |
| RAM_DST_ADRS | 压缩映像重定位后的地址。默认值为 RAM_HIGH_ADRS，在 rules.bsp 会根据映像类型重定义 | bootInit.c |
| RESIDENT_DATA | 因处理器体系而异。在 MIPS 和 PowerPC 上定义为 RAM_DST_ADRS，其他体系定义为数据段起始位置 | bootInit.c |
| ROM_COPY_SIZE | 对未压缩和驻留 ROM 映像，定义为需要重定位的映像长度 | bootInit.c |
| ROM_BASE_ADRS | 在 config.h 中定义，如果定义 BOOTC ODE_IN_RAM 则重定义为 romInit | bootInit.c |
| binArrayStart | 压缩映像起始位置 | |
| binArrayEnd | 压缩映像结束位置 | |
| UNCOMPRESS | 编译时为生成不压缩映像而定义。不要重新定义 | rules.bsp（可选） |
| ROM_RESIDENT | 编译时为生成 ROM 驻留映像而定义。不要重新定义 | rules.bsp（可选） |
| BOOTCODE_IN_RAM | 表示引导代码已经在 RAM 中，不要进行 RAM 初始化。对 x86 体系结构的系统必须在 config.h 中定义该宏。 | config.h（可选） |

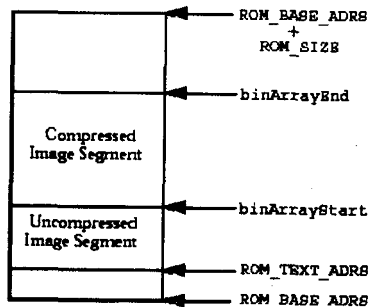



图 12-3　ROM 布局

图 12-3 和图 12-4 表示了在上述宏定义下，ROM 和 RAM 的布局。

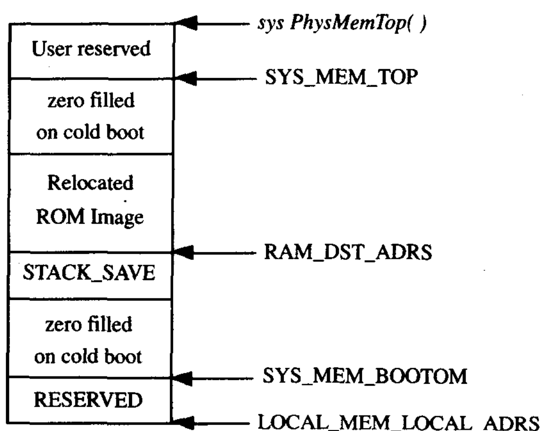



图 12-4 RAM 布局

## 12.2.3 sysInit( )

sysInit( )是可装入映像的入口程序，在装入 RAM 后的 sysInit( )位于 RAM 中 RAM_LOW_ADRS 处。该函数执行可装入映像的最初的初始化，其功能和 romInit( )基本相同，差别在于 sysInit( )不进行内存初始化，可装入映像运行时，假定系统已经对内存进行了初始化。可装入映像的 sysInit( )重复 romInit( )已经完成的初始化操作主要是出于模块化的考虑，增强映像独立性和可靠性。

可装入映像由引导程序装入内存。如果引导程序非驻留 ROM 类型，则引导程序本身必须被装入内存另一区域，前文曾指出这一区域即 RAM_HIGH_ADRS。在从引导程序转入可装入映像后，可装入映像会在随后一系列初始化操作中收回引导程序占用的内存并归入系统内存池。

和 romInit 一样，sysInit( )是汇编程序，位于模块 sysALibs。SysInit 可以简单地从 romInit 得到：（1）去掉 romInit 中对内存的初始化；（2）将结束时跳到 romStart 改为跳到 usrInit；（3）代码在 RAM 中执行，不需要 PIC。

生成各种映像时 sysInit 都被连接，但是只有对可装入映像才会被执行。

### 栈初始化

sysInit( )将为随后的 C 程序 usrInit( )初始化一个栈，该栈也称**初始栈**。sysInit( )初始化的栈从 VxWorks 可装入映像起始处 RAM_LOW_ADRS 开始向下的一段空间，栈在该空间的增长方式分向上增长和向下增长。其中，RAM_LOW_ADRS 和 LOCAL_MEM_LOCAL_ADRS 之间有些区域含有特定信息（如中断向量，异常信息，引导命令行参数等），不能作为栈使用，因此，决定 VxWorks 映像装入地址 RAM_LOW_ADRS 时必须考虑可以留给栈使用的空间。图 12-5 为初始栈在 RAM 中的位置。

usrInit( )不返回，在为其初始化的栈上工作，直到内核被激活。

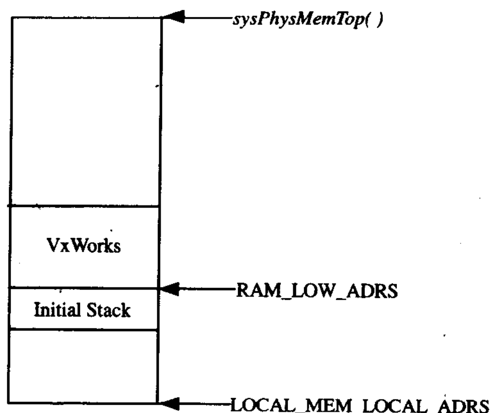



图 12-5　栈初始化

# 12.3　准备激活内核

内核激活的准备工作有 usrInit( )实现，“准备”也就是使激活内核的条件满足：（1）设备处于静止状态；（2）处理器中断处于禁止状态。在引导阶段已经使条件（2）满足，因此 usrInit( )的主要工作在于使设备处于静止状态。

## 12.3.1　usrInit( )

有两个版本的 usrInit( )：

- BSP 引导映像使用<install-dir>/config/all/bootConfig.c 中定义的 usrInit( )；
- VxWorks 系统映像使用 C 模块<install-dir>/config/all/usrConfig.c 中定义 usrInit( )。

BSP 版本的 usrInit( )同样使硬件置于静止状态，但是，差别在于通过它最终进入引导任务 tBoot；而 VxWorks 系统映像中的 usrInit( )最终使操作系统和用户应用运行起来。下面主要针对 VxWorks 系统映像使用的 usrInit( )进行分析。

romInit( )/sysInit( ) 只执行最小的初始化，这些初始化只是为了使 usrInit( )正确执行。VxWorks 通过 usrInit( )实现其余的硬件初始化。

要使内核被正确地激活，必须将硬件置于静止状态。usrInit( )主要职责之一，就是通过调用 sysHwInit( )，初始化依赖于目标板的硬件，禁止硬件中断，置于静止状态。

在正常情况下，usrInit( )一直运行，除非出现致命异常或者复位系统，usrInit 不返回。

usrInit( )调用函数 kernelInit( )，函数 kernelInit( )负责激活多任务环境，派生根任务 tUsrRoot，tUsrRoot 将：

- 创建设备，安装设备驱动程序；
- 安装中断处理程序，激活中断；
- 初始化 VxWorks 库程序；
- 调用应用程序启动代码。

### 1. 调试代理（WDB agent）

在内核激活之前，目标系统的 VxWorks 可以通过调试代理可以访问 Tornado 工具。由于中断环境没有建立，网卡也不被支持，只能通过串口以查询方式和主机通信。调用 sysHwInit( )将进行必须的串口初始化，使得 WDB agent 可以激活，并通过查询串口的方式访问，这一访问方式也称为**系统模式**（System Mode）。当设备中断被初始化并允许后，调试代理可以运行在**任务模式**。

系统模式下调试可以访问 CrossWind：

- 调试中断处理程序；
- 调试内核后设备初始化。

### 2. 栈

前已述及，对可装入映像，sysInit( )为 usrInit( )初始化栈；对于 ROM 映像，romInit( )为 usrInit( )初始化栈。如果栈的使用方式为向下增长，则定义为从映像装入地址开始处；如果栈的使用方式为向上增长，则栈定义为映像装入地址开始处减去栈大小。

栈的大小由宏 STACK_SAVE 定义。

### 3. 修改与开发 usrInit

usrInit 是一般化程序，执行的功能受支持程序和配置参数控制，这样开发新的 usrInit 可以通过某个参考 BSP 修改得到：

- 修改系统硬件初始化程序 sysHwInit( )；
- 修改系统支持程序 sys*X*( )；
- 修改 BSP 和驱动程序配置文件；
- 修改设备驱动程序代码。

除了为了激活 WDB agent 在系统模式下进行调试，usrInit( )一般不需要被修改。

## 12.3.2 sysHwInit( )

sysHwInit( )在 C 模块 sysLib.c 中定义，只能被 usrInit 调用，用户应用程序不能直接调用该程序。sysHwInit( )负责在 VxWorks 内核被激活之前，将硬件置于静止状态：

- 初始化硬件环境；
- 禁用硬件中断；
- 初始化设备控制/状态寄存器。

所有设备中断必须都被禁止：

- sysHwInit( )在锁中断条件下执行；
- 当 kernelInit( )激活内核时解锁中断；
- 在内核激活后才能安装 ISR。

各个设备中断功能将在内核激活后由各个设备初始化程序启用。

### 1．设备环境、设备和中断

设备环境和设备初始化主要用来禁止中断。禁止设备中断需要访问设备。某些特定设备要在内核被激活之前被访问，sysHwInit( )必须对设备环境做部分初始化，这些初始化可能包括：

- 初始化内存控制器；
- 配置通过总线访问设备。

### 2．环境初始化以支持设备访问

为了能够访问设备，必须：

- CPU 访问系统总线，对于只有局部总线的目标系统，CPU 对总线的访问由 romInit( )或 sysInit( )提供，如果存在其他总线与局部总线相连，则由 sysHwInit( )提供 CPU 对总线的访问支持；
- 通过系统总线能够访问设备，通过系统总线访问设备由 sysHwInit( )提供支持。x86 体系的 PCI 总线环境（Plug-N-Play）初始化比较独特：上电时 BIOS 提供许多必需的初始化来支持设备访问。

PowerPC 体系的 PCI 总线环境：

- sysHwInit( )提供 PCI 环境初始化（CPU/PCI 桥和中断控制权）；
- sysHwInit( )检测 PCI 总线上的设备，并进行配置（LAN，SCSI 和 VME）。

### 3．设备初始化

在总线可以访问设备后，sysHwInit( )可以进行初始化。sysHwInit( )进行的设备初始化主要是禁止中断。在禁止设备中断之前，必须做如下设备初始化：

- 配置设备，使得其中断控制寄存器可以被访问；
- 中断控制器初始化。设备环境还可以包括中断控制器，中断控制器本身也是设备，由其驱动程序控制，但是 BSP 负责其初始化。

余下部分的设备初始化在内核激活后完成。

例外情况：

- 由 BSP 控制的设备（如：总线桥接器中断控制器）；
- 查询模式下串口控制器（用于激活内核前系统级调试）。

### 4. BSP 控制设备

讨论设备初始化之前，要区分两类设备：纯设备和 BSP 控制设备。

纯设备驱动程序是在目录<install-dir>/target/src/drv 和<install-dir>/target/h/drv 下的代码，已经被 WRS 支持。这类驱动程序的特点是：

- 控制设备硬件，而和特定 CPU 无关；
- 在多种 BSP 环境下使用。

除了上述驱动程序的设备以外，其他的 WRS 不支持的特定设备属于 BSP 控制设备，设备驱动代码位于该 BSP 目录下。

WRS 区分纯设备和 BSP 控制设备是为了简化 BSP 设计。下面会看到在 sysHwInit( ) 中纯设备和 BSP 控制设备的初始化位置及负责初始化的函数是不同的。

### 5. 有中断控制器的 BSP

中断控制器本身由 sysHwInit( )初始化。对于连接到中断控制器的设备，中断控制器通过设置中断掩码进行中断屏蔽。

- 有驱动程序的设备，其中断和其他控制寄存器在内核激活后由对应的驱动程序进行初始化。例外：sysHwInit( )需要初始化串口控制器（即串口）；
- 对于其他设备（即 BSP 控制设备），中断和其他控制寄存器在 sysHwInit( )中初始化。

和中断没有关系的初始化：

- 检测设备的存在；
- 最小设备特定初始化使得设备可以被访问。

如下体系处理器的 BSP 通常含有中断控制器：

- Motorola PowerPC 系列；
- Intel 80X86 系列；
- Intel i960 系列的某些型号可以配置成使用中断控制器。

### 6. 无中断控制器的 BSP

没有专门的中断控制器的系统，所有设备的中断请求引脚直接连接到 CPU。因此 sysHwInit( )对这类设备中断的禁止是逐设备独立进行的。

对设备的中断控制寄存器访问方式：

- 直接通过局部总线；
- 通过一个外部内存控制器进行仲裁访问（内存控制器可能需要被初始化以支持设备访问）。

如下体系处理器的 BSP 通常不含中断控制器：

- Motorola 68k 系列；
- SPARC 系列；

- MIPS 系列；
- Intel i960 系列的某些型号可以配置成不使用中断控制器。

### 7. 串口初始化

在内核启动之前，sysHwInit( )需要初始化串口，使得能够通过查询方式实现系统级调试。串口初始化调用函数 sysSerialHwInit( )实现，该函数在 sysSerial.c 中定义。

串口由串行通信控制权 SCC 控制。一个 SCC 芯片一般有 2～4 个通道，一个通道物理上对应一个串口；在控制上，软件为每个通道维护一组控制结构信息，其中包含回调函数。调试代理通过调用回调函数实现对串口的查询访问。sysSerialHwInit( )对串口的初始化包括：

- 初始化 SCC 控制结构信息；
- 调用驱动程序初始化代码禁止中断。

如果系统有多个串口，sysSerialHwInit( )全部初始化。

### 8. 附加初始化

除了禁止所有设备中断，SCC 初始化，sysHwInit( )还进行：

- 初始化 BSP 控制的设备；
- 动态确定内存大小（Memory Autosizing）；
- 提取网卡硬件地址。网卡硬件地址是一个全球惟一的 48 位编号，通常存储在 NVRAM 或者由电池供电的 RAM 里。

### 9. 确定内存大小

从 Tornado 1.0.1 开始许多 BSP 支持通过宏 LOCAL_MEM_AUTOSIZE 来定义是否在运行时动态确定内存大小。与动态确定内存大小相对的是静态确定，即通过宏 LOCAL_MEM_SIZE 在编译时即确定内存大小。具体地，确定内存大小由 sysPhyMemTop( )实现，该函数一般具有如下形式，从中可以看出动态确定内存大小的过程。

```
char * sysPhysMemTop (void)
{
    static char * memTop = NULL;
    if (memTop == NULL)
    {
#ifdef LOCAL_MEM_AUTOSIZE
        /*
         * This is the place to do dynamic memory sizing,
         * or create an error.
         */
# error This BSP does not support LOCAL_MEM_AUTOSIZE
```

```
#else
        /* Static memory sizing */
        memTop = (char *)(LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE);
    }
#endif
    return memTop;
}
```

如果没有定义 LOCAL_MEM_AUTOSIZE 或者动态确定内存大小，则内存大小由 LOCAL_MEM_SIZE 静态确定。

**注意**

LOCAL_MEM_SIZE 表示整个物理内存大小，不需要预留空间。预留用户空间后的内存大小由 USER_RESERVED_MEM 定义。函数 sysMemTop( )返回除用户预留部分外的由系统分配使用的内存部分。

```
char * sysMemTop (void)
{
    static char * memTop = NULL;
    if (memTop == NULL)
    {
        memTop = sysPhysMemTop( ) - USER_RESERVED_MEM;
    }
    return memTop;
}
```

## 10. sysHwInit( )和 sysLib

sysHwInit( )是 sysLib 库的一部分。库 sysLib 实现功能包括：

- VxWorks 和用户代码对硬件的访问；
- 硬件环境独立接口；
- 许多程序不由用户代码调用。

模块 sysLib.c 中的程序都属于 sysLib 库，sysLib 还包括其他模块中的程序，主要是 sysX( )。

sysLib.c 通过#include 相关文件来支持：

- BSP 必须使用的驱动程序代码，位于<install-dir >/target /src/drv 和 BSP 目录下；
- 环境和驱动程序控制参数，位于<install-dir >/target/config，<install-dir >/target /h 和<install-dir >/target/h/drv 目录下。

一个属于 sysLib 但不在 sysLib.c 中定义的例子是程序 sysClkConnect( )，该程序在<install-dir>/target/src/drv/timer/xxTimer.c 中定义。此外还有 BSP 目录下的 sysNet.c 和 sysScsi.c 都属于 sysLib。

## 11. 调试 sysHwInit( )

函数 sysHwInit( )对硬件的初始化并不一定确保所有的硬件都进入了静止状态，因此在内核激活后，随着中断的开放，可能在根任务 tUsrRoot 开始运行之前有一个或多个设备产生中断。这一情况是不期望的。必须找出引起中断的设备并将其置为静止。

为找出中断源，可以采取的做法有：

（1）在 sysHwInit( )中为可疑的设备中断安装 ISR，该 ISR 是一个诊断程序；

（2）使用 ICE 在中断向量表中设置断点；

（3）使用逻辑分析仪检查访问中断向量表的指令。

第（1）种方式最简单，不需要专门的调试设备。具体来说，如果体系支持，可以通过 intVecSet( )安装 ISR。注意在 sysHwInit( )中还不能使用 intConnect( )来安装 ISR，因为系统还没有初始化该函数需要访问的系统内存池。

首先定位可疑设备的中断向量。在调用 sysHwInit( )之前，中断向量表基地址已经有 usrInit( )初始化。可以使用宏 INUM_TO_IVEC( )来计算 WRS 中断向量。

例如，假定可以设备中断的中断号为 intNum，则可以：

```
intVec = INUM_TO_IVEC(intNum);
intVecSet (intVec, debugISR);
```

函数 debugISR 为用于调试的中断服务程序，该程序用来指示谁产生了中断。它可以简单地实现为将中断向量写入某个内存地址，或者在中断向量被复用时，更深入地检查设备的控制寄存器来找出其中哪个设备引发了中断。

& 上述函数 intVecSet( )需要处理器体系支持中断向量基地址寄存器，如果处理器体系不支持，intVecSet( )可能是一个空函数。此时程序需要创建一个中断向量表，并专门写一个函数将调试 ISR 写入中断向量表。在 PowerPC 处理器即是如此。

## 12. 系统重启

在 sysHwint( )遇到不可继续执行的错误时，需要重启系统，即调用 STATUS sysToMonitor (startType)将控制跳转到 ROM 监控程序。

startType：表明重启类型的一个整数值。重启类型在../h/sysLib.h 中定义：

- BOOT_NORMAL ——正常重启；
- BOOT_NO_AUTOBOOT —— 不自动引导；
- BOOT_CLEAR —— 重启时清除内存；
- BOOT_QUICK_AUTOBOOT —— 快速自动重新引导。

# 12.4 激活内核 kernelInit

内核由函数 kernelInit( )激活，具体职责包括：

- 初始化，启动启动内核；
- 初始化系统内存池；
- 激活一个 tUsrRoot 任务完成初始化过程；
- 解锁中断（在 kernelInit 退出而 tUsrRoot 运行之前执行）。

如果体系支持，kernelInit( )还设置一个专用的中断栈。

内核数据结构由 usrKernelInit( )进行配置，该程序在 sysHwInit( )和 kernelInit( )之间调用，代码位于<install-dir>/target/src/config/usrKernel.c 中。

内核数据结构包括：二进制信号量、看门狗、内核队列等。

## 1．系统内存池

系统内存池（主要就是用于实现 malloc 动态存储分配的系统堆）的低端地址和高端地址作为参数传递给 kernelInit( )。系统内存池的初始化是 VxWorks 启动过程中的重要标志之一。因为有了系统内存池，所以很多需要使用 malloc 的初始化函数能够被调用。在下一步，根任务的许多初始化函数就需要动态存储分配功能。

系统内存分区底端值的确定：

- 如果没有定义 INCLUDE_WDB：FREE_RAM_ADRS；
- 否则 FREE_RAM_ADRS + WDB_POOL_SIZE。

系统内存高端：调用 BSP 提供的函数 sysMemTop( )得到。

## 2．中断

为了可靠执行，中断在 sysHwInit( )被关闭。现在，在 kernelInit( )执行完毕，开始运行根任务 tUsrRoot 之前，中断将被解锁。如果 sysHwInit( )中没有禁止所有中断，tUsrRoot 可能不会正确执行。

如果系统支持专用中断栈，kernelInit( )还会根据传递给它的参数初始化专用中断栈：

- 中断栈空间大小由参数列表指定；
- 中断栈位于系统内存分区开始处；
- 函数 checkStack( )将中断栈填写为 0xee。

kernelInit( )不负责设置 ISR，必须在适当的 ISR 填写到中断向量表后，中断才能被激活。

为 kernelInit( )传递的参数还包括中断锁级别。

## 12.5　根任务：tUsrRoot

根任务 tUsrRoot 是多任务内核激活并开放中断后第一个运行的任务，负责彻底初始化操作系统，使操作系统具有用户所定义的功能。根任务的函数为 usrRoot( )，在 usrConfig.c 中定义。

在根任务之前，因为没有系统内存池，所以许多直接或间接需要 malloc( )的初始化函数无法调用。例如为设备连接中断向量的调用 intConnect( )需要间接调用 malloc( )，因此此前无法进行。根任务中将调用各种定制的设备驱动程序。安装 ISR，开放设备中断都在调用设备驱动时完成。

根任务 tUsrRoot 具体会执行下列操作，并且因用户定制的不同组件而不同。

- 初始化内存分区管理库；
- 初始化 MMU（可选）；
- 初始化系统时钟；
- 初始化 I/O 系统（可选）；
- 创建设备（可选）；
- 配置网络（可选）；
- 激活 WDB agent（可选）；
- 激活应用系统。

根任务的栈空间和 TCB 从系统内存高端分配，栈大小由 ROOT_STACK_SIZE 控制，优先级为 0（最高优先级）。任务结束时 VxWorks 收回栈和 TCB 占用的内存空间。应用系统代码通过生成一个新任务实现，对于 ROM 引导映像，所激活的应用系统代码就是装入并定位 VxWorks 可装入映像。