

前 言

VxWorks 是美国 Wind River System 公司的产品，在通信、网络、工业、消费电子、军事等领域都有广泛的应用。本书介绍了嵌入式实时多任务操作系统 VxWorks 及其主机环境 Tornado，并实践性地描述了其上的程序开发过程。

本书主要面向有一定 VxWorks 经验或嵌入式经验的软件开发人员，但是部分章节对初学者也非常有帮助，如一些基本概念和一般开发过程。希望能帮助 VxWorks 软件开发人员更好地使用平台，并理解其内部工作机制，扩展平台功能。

本书主要针对 VxWorks 基本平台进行描述。而 Wind River 的其他扩展产品都依赖于该平台，并和特定应用领域有关，对此，书中只做简单介绍，如 WFC、VxMP、VxFusion 和 VxDCOM 等。另外还有丰富的第三方产品，书中也不做描述。

本书不特别针对某个硬件平台，主要进行硬件无关性描述。如果有硬件平台假设，则为 PC104/486 或 MFC5272。软件平台介绍的是 VxWorks 5.4 版本和 Tornado 2.0 版本。

本书相对其他 VxWorks 书籍和参考资料的优点如下。

1. 最明显的特点就是中文文化，关于 VxWorks 的参考资料，很多都是英文的，不利于开发人员快速阅读，建立基本概念。
2. 本书不是简单的手册翻译，而是作者数年宝贵开发经验的总结，有价值的东西很明显。虽然手册是主要的参考资源，但对于学习 VxWorks 来说，却太过生硬，和实际的开发过程不对应。
3. 本书是 VxWorks 相关主题的有机综合，避免了同一个主题分散在多本书籍，不便参考。
4. 书中各处都有对 VxWorks 相关资源的参考指导，可作为读者查阅各种资料的总索引。
5. 书中的所有内容都融合了作者的经验和理解，对读者的实践过程有很强的指导作用。
6. 常见问题（FAQ）的收集，可以帮助读者及时地解决开发过程中遇到的问题。

本书共包括 10 章，各章节的排列是相对 VxWorks 开发的自然分类，与 VxWorks 的组件结构对应，部分章节按库条目组织，而且只对常用库函数做描述，注重基本原理的介绍。

第 1 章“WindRiver”，介绍 Wind River 公司及其产品。让读者了解 VxWorks 的背景、应用领域和成功范例等，以及 VxWorks 在 WindRiver 产品系列中的基础地位和可能配合其应用的扩展产品。

第 2 章“Tornado”，介绍 VxWorks 的主机开发工具 Tornado，让读者建立工作环境的第一印象。

第 3 章“VxWorks”，介绍 VxWorks 操作系统，描述一些基本组件。



第 4~8 章，详细介绍 VxWorks 各个重要的组件，包括多任务环境、内存管理、IO 系统（包括字符设备）、文件系统和网络通信等。

第 9 章“建立开发环境”，指导读者建立基于硬件的嵌入式交叉开发环境，讲述了 BSP 的基本概念和定制步骤，说明了 BootRom 的建立，并列举了典型实例。

第 10 章“程序开发实践”，描述 VxWorks 程序开发实践中需要特别注意的内容，包括

Tornado 扩展、调试实践、编程实践和移植实践等，带领读者进入真正的 VxWorks 程序开发。

书中符号和格式说明。

1. 每章最后一节为常见问题解答，即 FAQ，问题描述用黑体字。
2. “[]” 内为英文或中文的同义词。
3. “()” 中为相关的扩展解释。
4. “ ” 给出读者可以进一步查阅的资料。
5. “ ” 表示要点或注意。
6. 文中的代码部分用阴影表示，DOS 窗口与 Shell 中的文字用边框表示。

本书由陈智育主编，主要的编写人员有温彦军（完成第 6 章、第 8 章、第 9.7 节和第 10.4.3 小节的写作）和陈琪（完成第 1 章和第 5 章的写作）等。在本书的写作过程中，得到了“电子产品世界”论坛（<http://bbs.edw.com.cn>）网友的鼓励，任建福、于青松、姜海涛、李庆华、郝刚和陈磊等同事的大力支持，以及孙运权老师的教学实践建议，在此一并表示感谢。在编写过程中，我们力求精益求精，但难免存在不足之处，恳请读者予以指正。如果在使用本书时遇到任何问题，可以发 E-mail 到 amine@263.net，我们将及时给您回复。或者直接与本书的责任编辑联系，本书责任编辑的 E-mail 是 tangqian@ptpress.com.cn。

编 者

2004 年 3 月

第 1 章 WindRiver

1.1 风河系统公司简介

说到嵌入式技术，就不能不提及美国风河系统公司（简称风河公司）Wind River System，及其开发的著名嵌入式实时操作系统 VxWorks。风河系统公司于 1981 创立，总部设在美国加利福尼亚的 Alameda。它是全球领先的嵌入式软件与服务提供商，也是业界唯一提供面向行业市场的嵌入式软件平台的厂商。它所提供的嵌入式软件平台包括集成化的实时操作系统、开发工具和技术。风河公司的产品和服务已经在许多市场领域得到认可，主要包括空间技术、国防、汽车、消费电子、工业制品和网络基础设施领域。世界各地的电子设备制造商普遍把风河公司的嵌入式软件产品作为行业标准。风河公司的软件能够支持现在市场上出售的超过 90% 的微处理器，其嵌入系统已被应用在大量的硬件和软件中，包括从激光打印机到控制战斗机发动机的自动刹车系统等。风河公司的产品可以帮助客户快速、低风险地创建高可靠性、复杂的实时应用系统。据资料表明，风河系统公司拥有 30% 以上的商业化嵌入式软件市场。

1983 年，风河公司推出了业界第一个实时操作系统 VxWorks；1995 年，推出了嵌入式软件领域的第一个集成开发环境[Integrated Development Environment] Tornado。如今，风河公司又率先推出了面向行业市场的集成化嵌入式平台[IEP]，即 Wind River Platforms。

1.2 实时操作系统 VxWorks 简介

VxWorks 操作系统是风河公司设计开发的一种嵌入式实时操作系统[RTOS]，是嵌入式开发环境的关键组成部分。良好的持续发展能力、高性能的内核以及友好的用户开发环境，使 VxWorks 在嵌入式实时操作系统领域占据一席之地。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求极高的领域中，如卫星通信、军事演习、弹道制导和飞机导航等。在美国的 F-16、FA-18 战斗机，B-2 隐形轰炸机和爱国者导弹上，甚至连 1997 年 4 月在火星表面登陆的火星探测器上也使用到了 VxWorks 操作系统。

VxWorks 是一个具有可伸缩、可裁剪、高可靠性，同时适用于所有流行目标 CPU 平台的实时操作系统。所谓可伸缩性指 VxWorks 提供了超过 1800 个应用编程接口[API]供用户自行选择使用；可裁剪性指用户可以根据自己的应用需求对 VxWorks 进行配置，产生具有各种不同功能集的操作系统映像；可靠性指能够胜任一些诸如飞行控制这样的关键性任务。VxWorks 包括一个微内核、强大的网络支持、文件系统、I/O 系统、C++ 支持的各种模块。与此同时，VxWorks 还支持超过 320 家的合作伙伴公司的第三方产品。

VxWorks 的主要特点如下。

- 高性能的微内核设计

VxWorks 的微内核具有全部实时特性，包括迅速的多任务调度、中断支持以及同时支持抢占式调度和时间片轮转调度。与此同时，该微内核还具有系统负担小、对外部事件的响应时间确定等特点。VxWorks 提供了广泛的任务间通信机制，包括共享内存、消息队列、Sockets、

远程过程调用[RPC]和信号量[Semaphore]等。提供的信号量有 3 种：二进制（也称作二值）信号量、计数信号量和互斥信号量。

- **可裁剪的运行软件**

VxWorks 在设计之初就具有可裁剪特性，使得开发者可以对操作系统的功能、大小进行增减，从而为自己应用程序保留了更多的系统资源。例如，在深层嵌入式应用中，操作系统可能只有几十 KB 的存储空间；而对于一些高端的通信应用，几乎所有的操作系统功能都可能需要。这就要求开发者能够从 100 多个不同的功能选项中生成适用于自己应用的操作系统配置。这些独立的模块既可以用于产品中，也可省去。利用 Tornado 的工程项目管理工具，可以十分轻松地对 VxWorks 的各种功能选项进行增减。

- **丰富的网络支持**

VxWorks 是第一个集成标准 TCP/IP 网络功能的实时操作系统。到目前为止，VxWorks 的 TCP/IP 协议支持最新的 Berkeley 网络协议。

- **POSIX1003.1b 兼容**

VxWorks 支持 POSIX 1003.1 规范的基本系统调用。包括进程原语、文件目录、I/O 原语、语言服务和目录管理等。另外，VxWorks 还遵循 POSIX 1003.1b 实时扩展标准，包括异步 I/O、计数信号量、消息队列、信号、内存管理（页面锁定）和调度控制等。

- **BSP 移植**

风河公司提供了大量预制的、支持许多商业主板及评估板的 BSP。同时，VxWorks 的开放式设计以及高度的可移植性使得用户在使用不同的目标板进行开发时，所做的移植工作量非常小。到目前为止，风河公司提供了超过 200 个的 BSP，当用户在自己的目标板开发 BSP 时，可以从风河公司的标准 BSP 中选一个最接近的来加以修改。

- **操作系统可选附件**

为了扩展 VxWorks 的功能，Wind River 公司还提供了一些可选附件，包括 BSP 开发工具包、支持 Flash 文件系统的 TrueFFS 组件、用于虚拟存储管理的 VxVMI 组件、用于支持多处理器的 VxMP 组件和 VxFusion 组件，以及各种图形、网络方面的组件。

- **可靠性**

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

- **实时性**

实时性是指系统能够在限定时间内，执行完规定的功能并对外部的异步事件作出响应的能力。实时性的强弱是以完成规定功能和做出响应时间的长短来衡量的。

VxWorks 的实时性做得非常好，其系统本身的开销很小，任务调度、任务间通信和中断处理等系统公用程序精练而高效，它们造成的延迟很短。VxWorks 提供的多任务机制对任务的控制采用了优先级抢占调度[Preemptive Priority Scheduling]和轮转调度[Round-Robin Scheduling]机制，也充分保证了可靠的实时性，使同样的硬件配置能满足更强的实时性要求，为应用的开发留下更大的余地。

- **多任务**

由于真实世界事件的异步性，嵌入式系统能够运行许多并发进程或任务是很重要的。

VxWorks 的多任务环境提供了一个较好的对真实世界的匹配，因为它允许对应于许多外部事件的多任务执行。系统内核通过给这些任务分配 CPU 时间来获得并发性。

- **抢占调度**

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

- **任务间的通信与同步机制**

在一个实时系统中，可能需要多个任务协同完成某个功能。系统必须提供这些任务间的快速且功能强大的通信机制。内核也要提供为了有效地共享不可抢占资源或临界区所需的同步机制。VxWorks 中提供了信号量、消息队列和事件等机制来实现任务间的通信与同步。

- **任务与中断之间的通信机制**

尽管真实世界的事件通常作为中断方式到来，但为了提供有效的排队、优先化和减少中断延时，通常希望在中断服务程序[ISR]中仅作一些必要的处理，其他的处理工作尽可能交给某个特定的上层任务来处理。所以，任务和 ISR 之间需要存在通信机制。信号量和消息队列同样可以实现该功能。

1.3 Wind River 的产品系列

风河公司提供了嵌入式系统开发的一系列产品，包括操作系统、软件开发调试工具、各种软件模块、BSP、参考板、硬件开发调试工具等。

- **集成开发环境 Tornado**

Tornado 是风河公司推出的图形化的开发工具，包括调试器 CrossWind、命令行界面 Wind Shell、资源查看器 Browser、模拟器 Simulator 等工具。使用 Tornado 及其开发调试工具，用户可以轻松地编译生成 BootRom，创建并配置 VxWorks，编辑、编译、下载和调试代码，随时查看目标机的系统资源，帮助用户缩短交叉式开发的周期。

- **单独的开发工具**

- ✧ **visionWARE**: 是一个高级的软件开发工具，主要用于开发硬件控制、操作部分的代码。使用 visionWARE，用户可以加快并简化对目标机的诊断和初始化部分代码的编写。
- ✧ **diab C/C++编译器**: 为了实现更好的代码优化和支持更多的 CPU，风河公司提供了 diab 编译器，可以使得嵌入式程序更具可靠性和灵活性，提供更快速、紧凑的目标码。
- ✧ **TurboJ**: TurboJ 是一个先进的 Java 编译器，能把 Java 的字节码转换成原码。在 VxWorks 上运行 Java 程序，传统的方式是使用 Java 虚拟机 JVM 进行字节码的解释执行，而对于使用 TurboJ 编译过的程序，其执行速度可以提高约 10 倍。
- ✧ **SingleStep**: SingleStep 是个源码级的调试器。对于一个开发小组，SingleStep 可以帮助用户进行硬件调试、设备驱动程序和 BSP 的开发、硬件和软件的集成、嵌入式应用程序的开发调试。
- ✧ **visionClick**: 通过与 visionICE 或 visionProbe 等硬件的结合使用，visionClick 可以

方便开发者对目标机硬件环境进行调试，包括 BootRom、VxWorks 和应用程序的调试，Flash 的擦写等，特别适用于设备驱动程序和 BSP 的开发。visionClick 是一个在主机上运行的软件，需要特定硬件设备的支持。

- ✧ visionProbe: 是一个硬件仿真器，它通过并口连接开发主机，通过 JTAG 或 BDM 接口连接目标机，从而实现在主机上对目标机的 BootRom、VxWorks 和应用程序的调试，Flash 的擦写等。
- ✧ visionICE: 是一个硬件仿真器，它通过网口连接开发主机，通过 JTAG 或 BDM 接口连接目标机，从而实现在主机上对目标机的 BootRom、VxWorks 和应用程序的调试，Flash 的擦写等。比 visionProbe 功能更强大。
- ✧ Wind Power IDE: 是一个功能强大的开发环境，可用于硬件、固件和应用程序的开发。通过提供更高程度的可视化和自动化，Wind Power IDE 可以帮助不同的开发人员快速简单地启动硬件、开发 BSP 和创建独立的应用程序。
- ✧ Wind Power ISS: 这是一个指令集的模拟工具，通过使用 Wind Power ISS，开发者可以在硬件环境还不具备的情况下运行代码，在硬件资源有限的情况下运行和测试代码。Wind Power ISS 是基于 Wind Power IDE 的。
- ✧ Wind Power ICE: 这是一个基于 JTAG 接口的硬件仿真器，通过多个 JTAG 设备的串行连接，可以实现多内核、多会话的硬件调试，适用于从硬件到应用程序整个开发阶段的调试。
- ✧ SNIFF+: 对于在 Unix 或 Windows 平台上开发应用程序的软件工程师来说，当代码量很大（100KB 到 5MB 的代码行），或使用了多种语言（如 C、C++、Java 和 ADA 等）时，SNIFF+ 是一个理想的集成开发环境，非常适合用于电信、自动化、国防和航天工业的软件开发。SNIFF+ 比 Tornado 具备更强大的代码开发、调试和管理功能。

此外，Wind River 还提供大量的、适用于各种硬件目标板的 BSP。

● 操作系统

- ✧ VxWorks: 这是一个在嵌入式行业被广泛采用的实时操作系统，具备实时性、灵活性、兼容性、可裁减性。
- ✧ VxWorks AE: 针对国防、航天、测量、控制系统所要求的高可靠性，风河公司推出了 VxWorks AE。VxWorks AE 是嵌入工业界最先进的实时操作系统，它提供了无与伦比的可靠性、可用性和可服务性[RAS]。为了高可用性的要求，VxWorks AE 提供了一些专门的特性：高性能的 Wind 微内核、保护域管理、隔离错误的应用、资源继承和管理、系统溢出保护、强大的装载工具、完整的网络工具。
- ✧ OSEKWorks: 是基于 OSEK-VDX 标准、可裁减、高可靠性的实时操作系统，包含多任务操作系统内核、通信机制和网络管理功能，尤其适用于汽车工业，因此被称为引擎盖下的操作系统，也可用于工业自动化行业。在使用了较多 8 位或 16 位 CPU 的网络中，OSEKWorks 可以使控制器良好地工作，其集成的通信机制使设备和上一级控制器能够更好地协调工作。
- ✧ pSOSystem: 是一个模块化、高性能的实时操作系统，能够提供一个多任务环境。主要用于数据通信、电信、机顶盒、消费电子、掌上电脑、测试设备和国防有关的装备。pSOStem 首先由印度 ISI 公司推出，在国内拥有很多的用户。20 世纪 90 年

代, ISI 被风河公司收购。

- ✧ **BSD/OS:** 使用 Wind River 的 BSD/OS 可以构建因特网服务器, 提供诸如 HTTP、FTP、电子邮件和拨号接入等服务。BSD/OS 具有了更高的可靠性和性能, 它是一个软件包, 包含操作系统、工具和因特网服务提供商[ISP]所需要的网络服务功能。支持 Intel 的体系平台。
- ✧ **VSPWorks:** 这也是一个高性能的实时操作系统, 尤其适用于多处理器的 DSP 系统。VSPWorks 通过提供一个虚拟单一处理器[Virtual Single Processor, VSP]结构, 使得用户易于开发分布式应用程序。

● 网络组件

- ✧ **USB:** 风河公司提供的 USB 开发包, 可以使用户方便地将某些 USB 设备挂接到嵌入式系统上, 包括键盘、鼠标、打印机、扬声器、数码相机、调制解调器和海量存储设备等。
- ✧ **WindNet 802.11b:** 风河公司提供的 WindNet 802.11b 设备驱动程序开发包, 支持 Intersil 的 PRISM 系列无线以太网芯片。使用 WindNet 802.11b 可以实现 IEEE802.11b 有关无线局域网的标准。
- ✧ **WindNet 802.1x:** IEEE802.1x 是针对网络安全和接入控制的一种解决方案, 风河公司推出的 WindNet 802.1x 包含 AUTHENTICATOR 和 SUPPLICANT 两部分, 实现了 IEEE802.1x 中关于无线客户和接入点的基于端口的接入控制, 可用于解决有线或无线局域网的安全问题。
- ✧ **Wind Manage CLI:** 是网络管理模块中命令行接口的开发软件, 可以使用户在很短的时间内编写出层次化、具备一定语义、适用于嵌入式系统的命令行接口。
- ✧ **Wind Manage WEB:** 是网络管理模块中 Web 服务器的开发软件, 可以使用户迅速、方便地开发出嵌入式系统中基于 Web 的设备管理软件。
- ✧ **Wind Manage MIBWay:** 在网络管理模块中, Wind Manage MIBWay 通过提供一种接口, 可以使用户在 CLI 或 Web 中轻松操作 SNMP 管理信息库中的对象。
- ✧ **Wind Manage SNMP:** 简单网络管理协议[SNMP]用于管理和配置网络设备, Wind Manage SNMP 是该协议的一个完整的实现, 包括源代码、工具和 MIB, 支持 SNMP v1、v2、v3、AgentX 和 IPv6 等特性。
- ✧ **WindNet Radius Client:** Radius 协议用于解决拨号用户的远程认证。WindNet Radius Client 实现了该协议的客户端功能, 包括完整的认证、计费和安全, 可以和 VxWorks 及其他 WindNet 产品良好的结合。
- ✧ **WindNet NAT:** 网络地址转换[NAT]协议用于路由器、防火墙、DSL、Cable Modem 和家庭网关中进行网络地址和端口号的转换。WindNet NAT 是 NAT 协议的完整实现, 通过地址映射和转换, 可以将只有一个全局 IP 地址的内部网络连接到外部因特网, 具备基本的网络安全性能。
- ✧ **WindNet IPSec & IKE:** 是 IETF 关于 IPSec 和 IKE 的实现, 通过使用因特网密钥交换、数据加密、认证和 IP 层网络流量的完整性功能, 为嵌入式设备提供了增强的网络安全性能。
- ✧ **WindNet PPP:** 实现了点对点协议[PPP], 可以为不同的平台提供远程接入方案, 并

为高级的应用添加扩展的功能。支持同步和异步的 HDLC 组帧。WindNet PPP 引入了远程接入框架的概念，在一个框架内可以实现多个 PPP 协议的实例，也可以轻松加入其他远程接入协议组件（如 PPPoE 和 Multilink 等）。

- ✧ WindNet PPPoE: PPP Over Ethernet[PPPoE]可以在以太网上建立 PPP 会话，广泛应用于宽带接入服务中。WindNet PPPoE 作为 WindNet PPP 的一个插件，实现了标准的 PPPoE 协议。
- ✧ WindNet Multilink: 是 WindNet PPP 远程接入框架中的一个可选插件，为了解决 PPP 连接中带宽的动态管理，WindNet Multilink 可以把多个并行的 PPP 连接聚合成一个虚拟链路，根据需要分配带宽的应用程序。WindNet Multilink 使用了带宽分配控制协议[BACP]，可以动态地从一个虚拟链路中添加或去掉某些 PPP 连接，从而实现带宽的动态分配。
- ✧ WindNet OSPF: 是开放式最短路径优先（OSPF，参见 RFC2328）路由协议的源码级实现，可以读、写并创建遵循 RFC1850 的 SNMP 管理信息库，用于路由器设备上的路由信息管理。
- ✧ WindNet ISDN: 是综合业务数字网[ISDN]协议的源码级实现，可以广泛应用于软交换、程控交换机、网关和其他的集成接入设备上。
- ✧ WindNet DCOM: 风河公司对微软的 DCOM 做了一些裁减，使其适用于嵌入式系统，这样就形成了 WindNet DCOM。它可以使嵌入式设备无缝地与远程 PC 机进行交互。
- ✧ WindNet OPC: 是适用于 VxWorks 嵌入式系统下的 OPC 实现，包括 OPC 数据访问服务器和交互式客户端工具，与 Tornado、VxWorks 和 WindNet DCOM 紧密结合。
- ✧ VxWorks 网络协议栈: 是 VxWorks 中标准 TCP/IP 协议的实现，基于 BSD4.4 的网络协议栈，并根据最新的协议标准做了相应的更新，尤其针对嵌入式系统进行了优化，适用于大量的网络端节点。
- ✧ WindNet Router Stack: 在 VxWorks 网络协议栈的基础上，针对第三层路由设备做了更多的优化和增强，WindNet Router Stack 提供了一系列额外的特性和能力，可以满足路由器、接入集中器[Access Concentrator]、网关和无线基站等交换设备的更高性能要求。
- ✧ WindNet IPv6: 是高度优化、性能增强的 IPv4/IPv6 的双协议栈，可以替换掉 VxWorks 的网络协议栈，完全符合相关标准，适用于高性能、高带宽要求的网络端点，如数据中心设备、消费类设备、SOHO 类设备和无线网络端点等。
- ✧ Tornado for Intelligent Network Acceleration[TINA]: 通过把一个系统中 TCP/IP 协议栈的处理工作完全交给一个专用的嵌入式处理器进行处理，TINA 可以加速 TCP/IP 的处理工作，降低 CPU 的利用率，提高 I/O 系统的吞吐量，极大地提高服务器的性能。

此外，Wind River 还曾经推出过 WindNet BGP-4、WindNet MPLS、WindNet ATM、WindNet L2TP 等网络协议，现在由于策略的调整，部分产品已经取消了。

● 图形软件

- ✧ Wind Media Library (WindML): 这是一个适用于 VxWorks 下开发图形用户界面的媒体库，在 2.0 版本之前称作 UGL。WindML 通过提供一系列的图形 API 函数，以

及多种键盘、鼠标、字体和显示芯片的驱动，使得用户可以很容易在嵌入式系统设备上实现图形用户界面。

- ✧ **Zinc:** 这是一个用于创建图形界面的软件，运行于主机上。使用 Zinc 软件，通过鼠标的点击和拖动，用户可以快速创建出内容丰富的图形界面，包括窗口、对话框、菜单和一些基本控件。更为重要的是，Zinc 能够自动生成 C++ 代码。自动产生的代码需要在 WindML 的支持下才能运行。

1.4 Wind River Platform 系列

2002 年下半年，风河公司推出第一个面向行业市场的集成化嵌入式平台 Wind River Platforms [风河平台]，同时在全球范围实行基于授权（licensing model）的租用定价模式，实现了嵌入式软件领域产品定价模式的一个创举。Wind River Platforms 面向几个主要的行业市场，提供高度集成化，并且经过了长期验证的技术，使电子产品开发商大幅度提高产品开发效率，加快产品上市时间，确保产品可靠性。

推出最可靠的集成化嵌入式平台，表明风河公司已经远远不只是一个操作系统提供商。有了 Wind River Platforms，电子产品制造商可以在不降低可靠性的前提下快速推出更为复杂、更加高效的产品，而且在智能化、网络化方面也将得到极大的改善。

所有这些面向不同行业领域的 Platform 都是一个集成化的整体，其中包括风河公司最知名的操作系统、开发工具、互连软件和管理工具，也包括相应的参考硬件和服务承诺，足以满足特定行业进行产品开发的特殊要求。同时，Wind River Platforms 也包括扩展的源代码。所有这些周密的安排都将在很大程度上降低产品开发项目的风险，并使产品开发人员可以集中精力去研究功能独特、性能出众的产品。

风河公司已经推出的面向特定市场的集成化嵌入式平台 [IEP, Integrated Embedded Platform] 包括：

- ✧ Wind River Platform for Consumer Devices
- ✧ Wind River Platform for Car Infotainment
- ✧ Wind River Platform for Industrial Devices
- ✧ Wind River Platform for Network Equipment
- ✧ Wind River Platform for Safety Critical
- ✧ Wind River Platform for Industrial Automation

● 面向消费电子产品市场的 Platform CD

Wind River Platform for Consumer Devices 是面向消费类电子产品开发的企业级嵌入式平台。终端应用包括数字影像设备、宽带接入设备、交互式数字电视和多媒体信息终端等。Platform CD 包括消费类电子产品通常需要的所有特性，例如图形、Web 服务、无线、因特网安全访问和管理。

Platform CD 包含的开发工具有：

- ✧ Tornado IDE
- ✧ Diab Utilities
- ✧ WindView
- ✧ SNiFF+
- ✧ Tornado Full Simulator

- ✧ Tornado BSP Developer's Kit

Platform CD 包含的组件有:

- ✧ VxWorks RTOS
- ✧ TrueFFS
- ✧ Wind Media Library (Wind ML)
- ✧ USB
- ✧ WindNet IPv6
- ✧ WindNet 802.11b Station
- ✧ WindNet 802.1x Supplicant
- ✧ WindNet PPP/PPPoE
- ✧ WindNet IPSec and IKE
- ✧ Wind Manage SNMP
- ✧ Wind Manage Web

- 面向工业电子产品市场的 Platform ID

Wind River Platform for Industrial Devices 支持众多的工业应用。终端应用包括机器人、医疗器械和测试仪器等。Platform ID 集成了先进的工业产品互连工具, 包括 DCOM、CAN、图形、管理功能和工业网络访问功能。

Platform ID 包含的开发工具有:

- ✧ Tornado IDE
- ✧ Diab Utilities
- ✧ WindView
- ✧ SNiFF+
- ✧ Tornado Full Simulator
- ✧ Tornado BSP Developer's Kit

Platform ID 包含的组件有:

- ✧ VxWorks RTOS
- ✧ TrueFFS
- ✧ Wind Media Library (Wind ML)
- ✧ USB
- ✧ WindNet CAN
- ✧ VxDCOM

- 面向汽车电子市场的 Platform CI

Wind River Platform for Car Infotainment 是个集成的嵌入式开发平台, 集成了必要的基本软件, 可快速设计和研制性能可靠的汽车信息设备和远程信息处理设备。Platform CI 中包含了适用于汽车信息系统的网络协议, 如 CAN、MOST、802.11b 和 PPP。

Platform CI 包含的开发工具有:

- ✧ Tornado IDE
- ✧ Diab Utilities
- ✧ WindView
- ✧ Tornado Full Simulator

- ✧ Tornado BSP Developer's Kit

Platform CI 包含的组件有：

- ✧ VxWorks RTOS
- ✧ TrueFFS
- ✧ Wind Media Library (Wind ML)
- ✧ USB
- ✧ WindNet 802.11b Station
- ✧ WindNet CAN
- ✧ WindNet PPP/PPPOE
- ✧ WindNet MOST
- ✧ JWorks
- ✧ Scope Tools

- 面向国防军工产品的 Platform for Safety Critical

Wind River Platform for Safety Critical 主要用来支持国防和空间技术等具有高度安全可靠性的项目。终端应用包括航空管理与控制系统，核能及核武器控制系统。对于必须满足 RTCA/DO-178B 和 ARINC 653 认证标准的工程项目，该平台是理想的应用开发工具。

- 面向网络设备市场的 Platform NE

Wind River Platform for Network Equipment 支持高可靠性网络基础设施的快速开发。终端应用包括应用集中器、DSLAM、多服务交换机和无线基站等。Platform NE 包括经过长期应用验证的路由协议栈、网管组件、三层技术和远程访问等。

Platform NE 包含的开发工具有：

- ✧ Tornado IDE
- ✧ Diab Utilities
- ✧ WindView
- ✧ SNiFF+
- ✧ Tornado BSP Developer's Kit
- ✧ Wind Manage Integration Tool

Platform NE 包含的组件有：

- ✧ VxWorks RTOS
- ✧ USB
- ✧ WindNet OSPF
- ✧ WindNet 802.11b Access Point
- ✧ WindNet 802.11b Station
- ✧ WindNet 802.1x Authenticator
- ✧ WindNet PPP/PPPOE
- ✧ WindNet IPSec and IKE
- ✧ WindNet Router Stack
- ✧ WindNet NAT
- ✧ WindNet Radius Client
- ✧ WindNet Learning Bridge

- ✧ Wind Manage SNMP
- ✧ Wind Manage SNMP AgentX
- ✧ Wind Manage CLI, Web, MIBway

● 面向工业自动化领域的 Platform IA

Wind River Platform for Industrial Automation 可用于开发可靠的、可管理的工业自动化设备，这些设备可连接到现场总线和基于以太网的网络上。

Platform IA 包含的开发工具有：

- ✧ Tornado IDE
- ✧ Diab Utilities
- ✧ WindView
- ✧ SNIFF+
- ✧ Tornado Full Simulator
- ✧ Tornado BSP Developer's Kit

Platform IA 包含的组件有：

- ✧ VxWorks RTOS
- ✧ TrueFFS
- ✧ Wind Media Library (Wind ML)
- ✧ USB
- ✧ WindNet CAN
- ✧ WindNet DeviceNet
- ✧ WindNet Ethernet/IP
- ✧ WindNet OPC
- ✧ VxDCOM

1.5 Wind River 产品的成功范例

Wind River 产品的成功范例如表 1-1 所示。

表 1-1 成功范例

客户名称	典型产品	采用的 Wind River 产品
ABB	IRB 6600 型工业机器人	VxWorks 操作系统及网络协议
Apple	AirPort 802.11 无线网关	Tornado for Home Gateway, WindNet Radius Client。
Applied Biosystems	ABI PRISM 3700 DNA 分析仪	VxWorks, pSOSytem
Avaya	基于 Java 的 4630 型 VOIP 电话	JWorks
BAE Systems	化学药品检测系统	VxWorks, SNMP
Bang & Olufsen	BeoVision Avant 电视	pSOSytem
eRemote	无线遥控器	VxWorks, JWorks, TrueFFS
欧洲空间局	PROBA 卫星	VxWorks
goReader	eReader 数字课本	VxWorks, JWorks
Honda	Asimo 机器人	VxWorks

InnoMedia	VoIP 可视电话	VxWorks
Kenwood	家庭音频服务器	操作系统, 网络协议
MBARI	水下远程操作运载工具	VxWorks
Mitsubishi	钻石/白金系列高清电视	VxWorks, JWorks, WindML
Nixvue	Vista FS-1 数字相册	VxWorks, Zinc
Performance Technologies	IPnexus CPC4400 嵌入式以太网交换机	Tornado for Managed Switches
Pingtel	xpressa VoIP 电话	JWorks
Redlake	MotionXtra HG-100K 高速、高分辨率相机	VxWorks, TrueFFS, 网络协议
Ricoh	RDC 系列数字相机	VxWorks, 网络协议
上海贝尔	Matix2000 网络接入服务器	VxWorks, 网络协议
Siemens	GGSN @dvanage CPG-3000 移动网络设备	网络协议
SONICblue	ReplayTV 4000 数字录像机	VxWorks
Sony	HAR-D1000 数字视音设备	VxWorks
Sony	NW 系列网络随身听	VxWorks
Telestream	ClipExpress 视频传输系统	VxWorks, 网络协议, Zinc
Xanavi Informatics	CARWINGS 信息通信业务	VxWorks
Xerox	DocuPrint M700 喷墨打印机	VxWorks

1.6 Wind River 服务支持途径

目前, 在国内有几家公司代理风河公司的产品。通过这些代理商购买了风河公司产品的客户, 可以向代理商要求培训和技术支持, 也可以通过各种方式直接与 Wind River 公司联系。Wind River 公司总部的地址及联系方式:

500 Wind River Way

Alameda, CA94051

(800) 545-WIND(9463)

Tel: (510)748-4100

Fax: (510)749-2010

Wind River 公司北京代表处的地址及联系方式:

北京市朝阳区西坝河东里 18 号

三元大厦 2101 室

邮编: 100028

电话: 010-84603640/41/42/43

用户也可以通过电子邮件获得技术支持:

support@windriver.com

support-china@windriver.com

此外, 用户还可以登录风河公司的网站 <http://www.windriver.com>, 下载相关资料和补丁程序, 浏览该公司的最新产品信息。有些资料的下载需要合法的用户名和密码, 可通过 <http://www.windriver.com/windsurf> 登录。

还有, 用户可以从 Tornado 软件的 Tools 菜单中选择 Support 菜单选项, 填写技术支持申请表。该申请表将会自动以电子邮件的方式发送给 support@windriver.com。

第 2 章 Tornado

Tornado 是主机的集成开发环境[IDE]，用于 VxWorks 嵌入式系统的交叉开发。交叉开发环境一般存在很多问题，如有限的调试通信通道，有限的目标机资源等。为了解决这些问题，Tornado 使用一种独特的结构，让用户界面尽量在主机上实现，减轻目标机负荷，以快速进行实时运行。

Tornado 使用了 Target Server-Agent 模式来建立主机和目标机的交叉开发环境。这种模式使所有主机工具可以用于各种目标机，而不必考虑目标机的资源和通信机制，这种模式还提供了 Tornado 图形界面的统一性和广泛适用性。

Tornado 是一个开放的可扩展环境，很容易集成第三方的开发工具，用户也可以按自己习惯定制开发环境。

为了提高用户的开发速度，Tornado 提供了很多方便的工具。

- ✧ 组件配置工具：能自动检查组件依赖关系，辅助用户完成 VxWorks 组件库配置。
- ✧ 主机 VxWorks 仿真器：可以在硬件完成前进行部分代码调试工作。
- ✧ 各种程序分析工具：超出普通调试器的概念，帮助开发者从各方面分析程序。
- ✧ 集成了编辑器、编译器、调试器等。



WindRiver, “WhitePapers-Tornado II for VxWorks”。

2.1 初识环境

在使用一个开发工具前，首先需要完成安装工作。Tornado 的安装和普通软件类似，但需要注意一些细节问题。对 Tornado 目录和文件的了解可以帮助读者建立开发环境的初步印象。本节还会指导读者如何有效地阅读随软件安装的帮助文档。

2.1.1 安装

安装时，输入安装 key，该 key 会确定可安装的组件，如 BSP 类型、可选组件等。

如果是初次安装，选择“Full Install”；若在另一主机操作系统或远程服务器已安装，可以选择“Program Group”安装程序快捷图标，实际使用已安装的副本。

输入工程名（自定义）和许可用户数（根据购买的许可形式，本机使用设为“1”即可）。“License Number”由 key 确定，可用来访问官方技术支持网站 WindSurf。

选择安装目录，如果同台主机上有多个平台或多个版本的 Tornado 安装，建议目录带平台和版本信息，以防止多个安装之间互相干扰，如 t20x86、t22cf 等。注意目录名不要含空格。

选择适合自己平台的组件。一般有多个平台的组件供选择，如 Windows95/98/NT 等。注意 Tornado 2.2 不支持 Windows95/98/Me。

选择 Registry 的启动方式，一般选择手动方式即可。

选择是否选用 Tornado1.0 的工具菜单，这主要为了保持向后兼容性。该菜单可用来生成 BootRom 和 VxWorks 映象，以代替命令行下的操作；而 VxWorks 映象最好用工程生成，不要用该菜单。不过是否选用兼容菜单对整个环境没有影响，所以建议选用。

Tornado 2.1 在 Windows 2000 Server 上安装会出现警告信息，但不影响实际使用。Tornado 2.0 在 Windows 2000 上启动时会出现 WindView 相关 TCL 文件无法读取的错误，这需要安装

Windows 2000 补丁，最好使用 T2CP4 升级到 Tornado 2.02，该补丁还修正其他一些错误，如对于 x86 版本，使用 tor202x86.tar.gz 这个补丁包。补丁包可以从 WindSurf 上或供应商处获得。T2CP4 还包含了针对 Tornado 2.0 所有平台版本的文档补丁包 patch-TSRT2CP4_Docs_Windows.tar.gz。

当安装完成后，最好到 WindRiver 的技术支持网站下载与安装版本相关的最新补丁，尽量避免在以后开发中可能出现的麻烦。WindRiver 的补丁更新很快，根据用户提交的 TSR 和 WindRiver 自己发现的 Bug，按 SPR 号推出补丁，到一个阶段还会推出一个综合的补丁包，如针对 Tornado 2.0 的 T2CP4，以及针对 Tornado 2.2 的 t22-cp1。

WindSurf 是 WindRiver 的技术支持网站。登录 WindSurf 需要用户的“License Number”（由供应商提供）。用户如果在开发中发现问题，可以在网站上提交 TSR 或直接给技术支持打电话。WindSurf 上提供了丰富开发资源，包括各种补丁、例程、AppNotes 和 TechTips 等。还有一个论坛，用于问题交流，不过人气不旺。

● 重安装

经过一段开发过程后，可能需要重新安装 Tornado，如病毒干扰、TornadoRegistry 无法启动、主机操作系统损坏、硬盘损坏、另一台机器安装和多操作系统共享等。再安装可能会遇到些问题，因为在开发过程中，用户会对 Tornado 目录下内容做很多修改，如补丁安装、源代码文件修改、系统库修改和工程配置等，重新安装可能丢失这些改动，进而耽误开发进程。

如果 Tornado 目录完好，在新操作系统中可以使用原来的目录，选择“Program Group”方式安装，这是最快的恢复方法，能省去诸多麻烦。

为了使开发环境容易恢复，这里提几点建议。对 BSP 的修改最好只改动 ALL 和 BSP 目录，而不要改动 comps 等目录；如果可能，最好使用自己命名的 ALL 和 BSP 目录，保留原始安装目录不动，以方便开发过程中进行对照；将 BSP 相关的修改文件都集中到 BSP 目录中，如网卡驱动、TFFS 驱动等；平时注意备份自己的 ALL 和 BSP 目录；需要备份工程目录，一般在“\target\proj”目录下；将自己源代码放在 Tornado 目录之外；备份系统库，因为可能包括许多修改。如果采用了上述建议，即使出现灾难性硬盘损坏，也会很容易进行恢复安装。

● 删除

Tornado 文件只集中在安装目录，不向 Windows 的系统目录添加文件，对注册表的修改也集中在“HKEY_LOCAL_MACHINE[HKEY_CURRENT_USER] > Software > Wind River Systems”下。所以，删除时可使用标准的 uninstall，也可以手动删除目录和注册表信息。

如果主机上存在多个 Tornado 安装，不同 Tornado 版本使用不同的注册表分类夹，可以删除任意版本而不影响其他版本。但如果是同一版本的 Tornado，对应不同目标的 VxWorks，它们会共用注册表信息，若用标准的 uninstall 移除一个 Tornado，会对另外的 Tornado 安装产生影响，这时最好手动删除 Tornado 安装目录，而保留注册表中的信息。

2.1.2 目录与文件

对开发环境中的目录与文件的了解，是认知新环境的第一步。本节主要描述 Tornado 相关的目录，如表 2-1 所示。VxWorks 相关内容在“WIND_BASE\target\”目录下，将在下章详


细介绍。

表 2-1

Tornado 目录列表

WIND_BASE（指向 Tornado 安装目录的环境变量）		
.wind\		存放个人定制文件和状态信息文件等
Docs\		存放各种帮助文档
Host\		存放 Tornado 主机环境相关文件
	include\	存放 Tornado API 相关的头文件
	resource\	存放 Windows 资源文件，如图标等
	resource\tcl\	存放用于实现用户界面的 TCL 文件及各种工具实现
	Tcl\	存放 TCL 相关源代码
	x86-win32\	存放主机工具及各种动态库
	x86-win32\lib\	Tornado 应用支持库
Setup \		安装相关目录（与用户无关）
Share\	Src\	WDB、WTX、Target Server 源代码，由主机和目标机共享
Target\		存放目标机 VxWorks 相关文件
Setup.log		包含安装信息，如组件列表

 WindRiver, “Tornado 2.0 API Programmer’s Guide” 的 1.6 章节。

 WindRiver, “Tornado 2.0 User’s Guide” 的附录 A。

2.1.3 帮助文档

在 Docs 目录下存放各种帮助手册，主要是与 Tornado 和 VxWorks 相关的，也有与特殊组件相关的，如 TrueFFS 等。帮助文档有 HTML 和 PDF 格式，建议使用 HTML 格式，方便交叉关联查询。

如果使用 HTML 格式的手册，“books.html”为手册总入口，进入后能看见各种手册列表，选择就能进入相关手册。Docs 根目录下还包含其他几个入口文件，如表 2-2 所示。

表 2-2

Tornado 手册列表

books.html	手册总入口
Tornado_Reference.html	分项描述各主机工具，如 WindSh、elfToBin 等
Tornado_API_Reference.html	描述 Tornado API，一般用户不用，按结构层次分类
libIndex.html	库参考入口，按库名分类，包括 Vxworks 库和 Tornado 库
BSP_Reference.html	一般与自己硬板不符，可以看看 sysLib
VxWorks_Reference_Manual.html	VxWorks 库手册，以库名排序描述，开发者经常参考
rtnIndex.html	VxWorks 库手册，以函数名排序描述，开发者经常参考

在开发中遇到问题需要帮助时，应选择合适的文件入口进入，可以加快资料查询速度。比如想查一个已知名称函数的原型参数或归属库，可以选择进入 rtnIndex.html，直接找到该函数，具体帮助文档在相应的库描述中。

WindRiver 的帮助文档很多，且归类不是太清晰，有些文档是直接从源代码中提取的。所以阅读帮助时需要讲究一定的顺序和方法。

对于初学者，可以先看“Tornado 2.0 Getting Started Guide”，该手册在有些 Tornado 版本

中没有，可以到网上查找。通过该手册，用户可以建立对 Tornado 开发环境的基本印象，熟悉基本工具的操作等。该手册用 VxSim 进行例程讲解，不需要具体的硬件目标板，比较适合初学者的前期学习。

用户也可以看看 WindRiver 的培训资料 Workshop 系列，共包括 4 部分，是 PowerPoint 讲稿形式，内容比较简洁，快速阅读后可以建立对 Tornado 和 VxWorks 更深层次理解。

如果读者觉得对 VxWorks 这个 RTOS 的概念还比较模糊，或没有嵌入式系统开发的从业经验，可以阅读 VxWorks 编程指南，其中对 VxWorks 基本概念做了全面的描述。若该手册和 VxWorks 的最新版本有不符的地方，应该以 VxWorks 的库参考手册为准。

如果读者不知道如何使用 Tornado 的功能，或在使用中遇到问题，可以参考 Tornado 用户指南，它对 Tornado 的直接用户有很大的帮助。

阅读完上面提到的各种手册后，用户能够对基本概念有较完整的了解，可以进行实际系统开发。在实际开发过程中，用户最常用的手册为 VxWorks 库参考手册，不用仔细通读该手册，只在遇到问题时查询即可。VxWorks 库参考手册中，文档直接从代码中提取，与 VxWorks 实际库情况最为吻合。

若用户在开发中使用某些专用组件，如 TureFFS、WindView、DosFs 2.0、VxDCOM 和 VxFusion 等，可以查阅专门的帮助手册。

在开发过程中查询文档与初学时阅读文档不同，不必追求全面的概貌，只需要掌握局部的细节。这时要善于利用各手册的 Index 部分，直接查询关键词，可以快速找到需要的资料。这也是为什么推荐使用 HTML 格式帮助的原因。

如果已能熟练使用 Tornado 和 VxWorks，再想深层次求解细节，源代码文件将是最好的文档，它不但可以帮助用户明白工作机制，还可以学到一些编程技巧和规范。

除了 WindRiver 提供的正式文档，还有很多其他参考资源，如 WindSurf、邮件组、论坛和中文书籍等。具体索引可以参考本书最后的附录部分。

2.2 基本结构

Tornado 结构的设计原则如下。

- ✧ 尽量减少目标机的负担，充分利用主机的能力。
- ✧ 主机工具共享单一通道和目标机通信，不必占用多个通信通道。
- ✧ 使 Tornado 易于定制和功能扩展。

Tornado 的结构示意如图 2-1 所示。

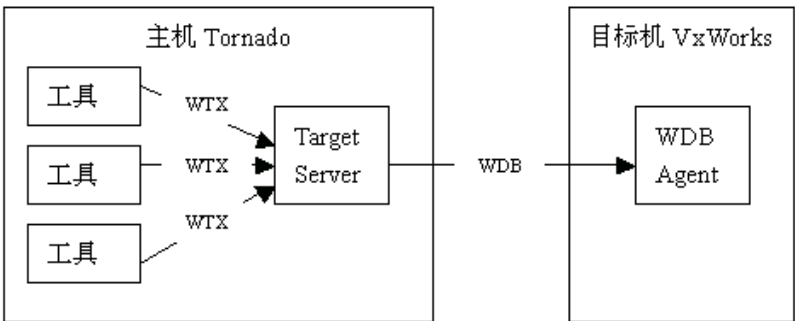


图 2-1 Tornado 结构示意图

图 2-1 所示的 WTX 和 WDB 两个协议最为重要，负责主机各工具和目标机的通信。其中 WTX（Windriver Tools eXchange）负责主机开发工具和 Target Server 之间的通信。而 WDB（Wind DeBug）负责主机的 Target Server 和目标机的 WDB Agent（Target Agent）之间的通信，该协议包括了 Gopher 编程语言。

WDB Agent 实现 Target Server 的请求，并返回结果。请求包括内存操作、断点设置、虚拟 IO 支持和任务控制等。WDB 使用 Sun 公司的 XDR 规范进行数据传输。WDB 能支持任务级和系统级两种调试模式。WDB Agent 能解释 Gopher 语言，进行目标机数据结构的传输，而不用目标机专门进行信息收集。WDB 的通信通道不经过 VxWorks 的 IO 系统，直接使用底层驱动。WDB Agent 独立于操作系统，可于操作系统之前运行或独自运行。

Target Server 在主机上运行。一个 Target Server 代表一个目标机，所有主机工具都通过它来访问目标机。Target Server 的结构如图 2-2 所示。

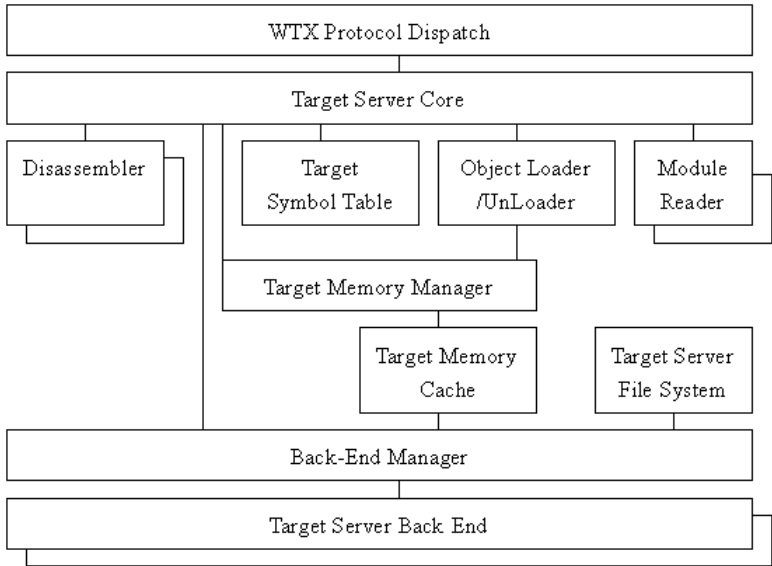


图 2-2 Target Server 结构示意图

图 2-2 所示中涉及概念的描述如表 2-3 所示。

表 2-3 Target Server 构件列表

WTX Protocol Dispatch	Target Server 的前端，负责和各主机工具接口
Target Server Core	解释 WTX，确定哪些服务在主机执行，哪些需目标机完成
Target Symbol Table	主机符号表
Object Loader/Unloader	Loader 加载目标模块到目标机中（使用主机符号表进行引用解析，并将新模块的符号添加到该符号表中），Unloader 用于模块卸载
Module Reader	用以解释目标模块格式，各种不同格式对应不同的 DLL
Target Memory Manager	目标机分配一块内存由主机使用，由 Manager 管理
Target Memory Cache	缓存部分目标机内存，如代码段，用于提供性能，如断点设置
Disassembler	反编译目标机内存，不同 CPU 对应不同 DLL
Target Server File System	在主机上为 VxWorks 虚拟一个文件系统
Back-End Manager	独立于具体 Back-End 的抽象层

Target Server Back End	与目标机 WDB Agent 通信，不同类型通道对应不同 DLL
------------------------	----------------------------------

Target Server 使用多个线程为 WTX 请求、目标机事件、Load 请求和 VIO 事件等提供高效服务，如图 2-3 所示。

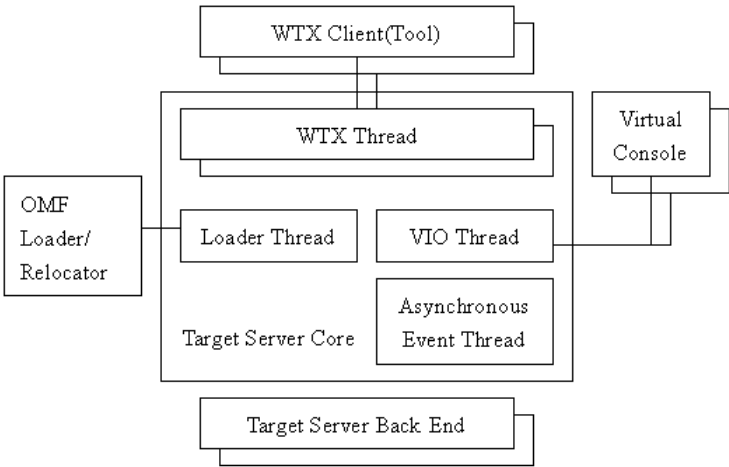


图 2-3 Target Server 多线程示意

Tornado 提供了丰富的 API，开发者能访问各层结构，用于功能扩展，如图 2-4 所示。严格来说，其中 Agent Interface 不属于 Tornado API，只是对目标机中设备驱动的接口进行了规定，要求通信设备驱动必须提供给 Target Agent 的函数接口。在主机上，不可能利用它来定制 Tornado 行为。

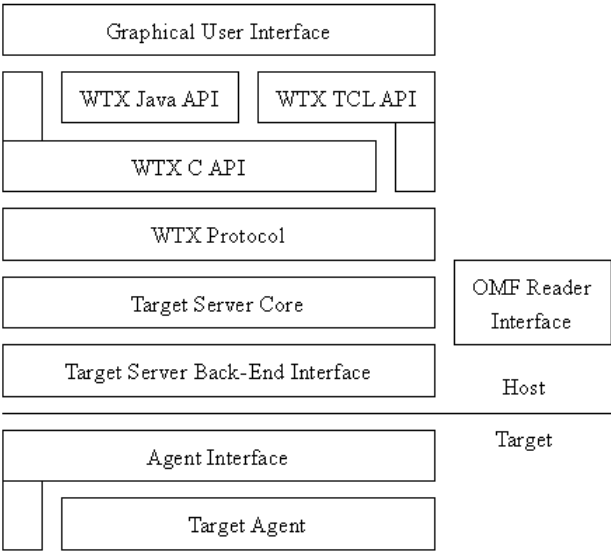


图 2-4 Tornado 各层 API 示意

Tornado 各层 API 的说明如表 2-4 所示。

表 2-4	Tornado 各层 API 说明
Graphical User Interface	提供 TCL 命令，用于 GUI 的创建和配置
WTX C API	WTX 的 C 函数接口，如 CrossWind 实现

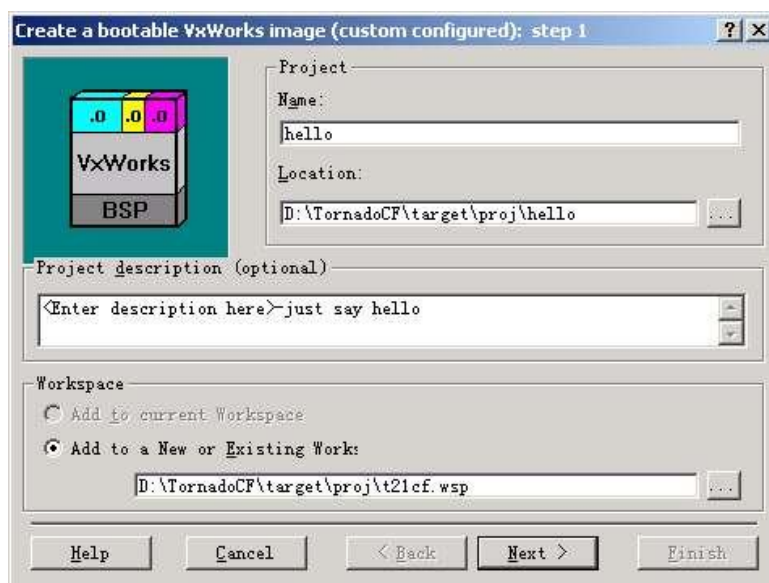


图 2-6 工程创建 2

下一步出现“step2”对话框，两种工程类型有区别，Bootable 工程需选择 BSP 基础，而 Downloadable 工程需选择工具链[toolschain]，分别如图 2-7 和图 2-8 所示。

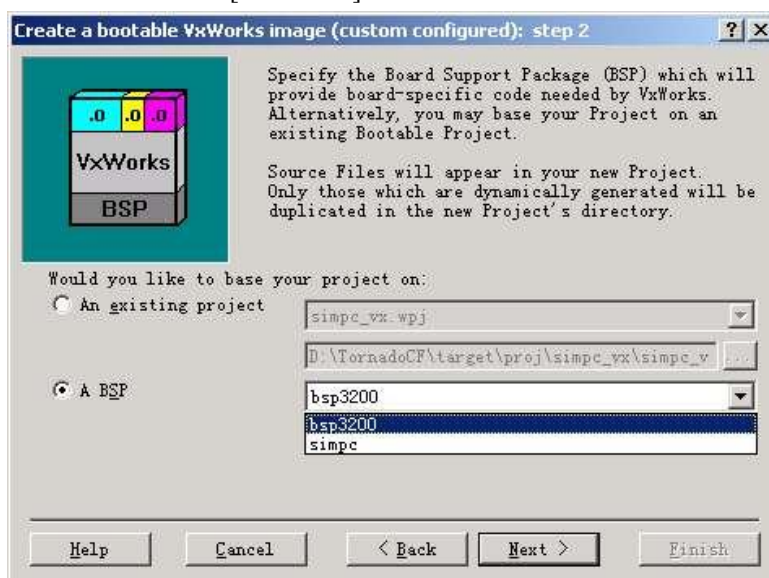


图 2-7 工程创建 3a

这时有两个选择。可以在已有工程的基础上创建新工程，类似工程复制，原工程中配置和源代码引用都会到新工程中。若想创建一个初始的空工程，应选择 BSP 基础，指向自己定制的 BSP 目录，工程创建时会搜索 BSP 目录，根据目录下的 Makefile 文件，将需要的 BSP 源代码文件自动加入新工程中。注意源代码文件名和 Makefile 中的目标名需大小写一致，否则不会自动加入。如果需重新生成主机仿真器 VxWorks.exe，比如添加一些组件，需选择 simpc。工程生成时会自动完成依赖关系，生成相关文件。

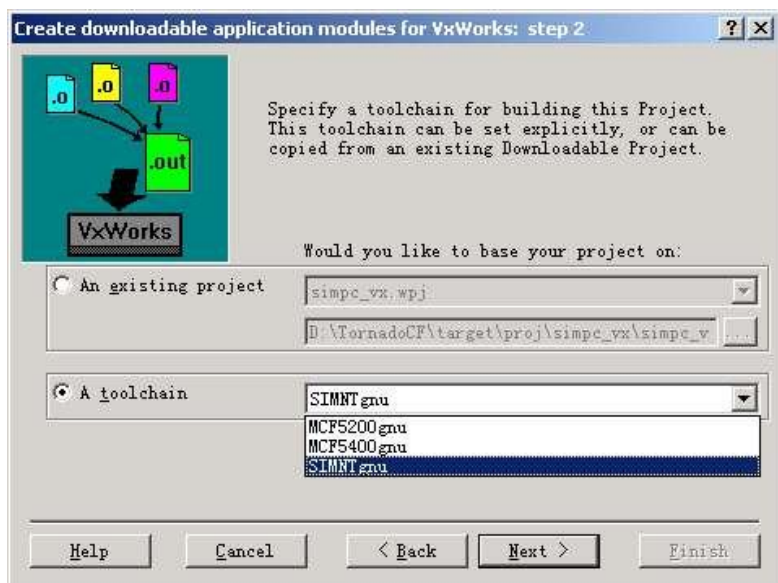


图 2-8 工程创建 3b

对于 Downloadable 工程，需要选择编译工具链，与 CPU 类型和编译器相关，如主机仿真选择 SIMNTgnu。工具链决定了编译器、系统库和目标格式等。工程生成时会自动完成依赖关系，生成相关文件。

生成的工作空间和工程如图 2-9 所示。

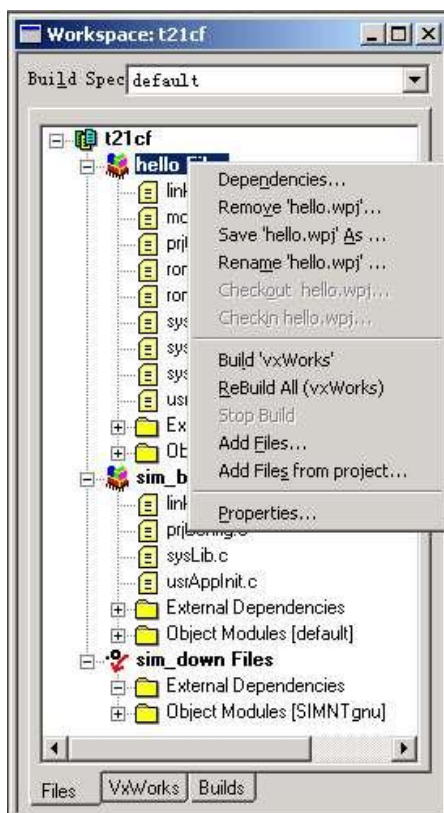


图 2-9 工程空间

图 2-9 所示的工作空间名称为“t21cf”，包含了 3 个初始创建的新工程，都还没加入相应的应用代码。“External Dependencies”中包含源代码的头文件依赖。“Object Modules”中包含生成的目标文件名和程序映像名，与工程目录下的子目录（archtool 或 default）对应。

图 2-9 所示的 Hello 示例是建立在自己定制 BSP 基础上的 Bootable 工程，引用了 BSP 目录的源代码，也包含了工程自动生成源代码。Sim_boot 为建立在 simpc 基础上的 Bootable 工程，用于生成自己定制组件的主机仿真程序 VxWorks.exe。Sim_down 为建立在 SIMNTgnu 工具链上的 Downloadable 工程，初始新工程为空，不包含任何源代码引用和组件配置。

2.3.2 源文件引用

创建的新工程只是一个初始化框架，只有和用户自己的应用代码文件关联后，才能进行实际的项目开发。工程中原文件的添加和删除可以在如图 2-9 所示窗口中完成，一般用鼠标右键进行菜单操作。

在工程名上点击右键，选择“Add Files...”来添加文件，一次可以添加多个，但不允许添加整个目录。且多文件添加时不记忆代码目录，再次添加时还需要查找目录。也可以选择“Add Files From Project...”，从同一个工作空间的其他工程中加入文件引用。新文件加入后，再次编译时会为新文件自动依赖头文件，或者用户自己选择“Dependencies”进行设置。

在文件名上点击右键，选择“Remove 文件名...”来删除文件引用，该操作并不实际删除硬盘上的文件。

需要注意加入的文件名称的大小写，Tornado 对大写名称的文件支持不好，可能会出现错误：“Warning: system doesn’t know how to build ‘XXXX.C’”。

有时需要修改安装在“\target\config\”下的模板源文件，但重新安装 Tornado 或补丁后，可能会丢失这些修改，所以最好将修改的源代码文件保存在自己的 BSP 目录下，而不使用缺省目录的源代码文件。首先需要创建自己的 BSP 目录，并复制近似的原 BSP 目录的文件，在自己的 BSP 目录下进行定制修改，这样新安装就不会影响自己的 BSP 目录下的修改。不过新安装补丁后，需要手动检查补丁修改的源文件，并将修改合并到自己 BSP 目录下相应文件中。

可以利用 Tornado 提供的文件宏来重新指定“\config\all\”的目录和文件名，如表 2-5 所示。

表 2-5文件宏列表

宏	缺省值	说明
CONFIG_ALL	all\	指定自己的定制 ALL 目录，在该目录下修改相关文件
BOOTCONFIG	bootConfig.c	指定 BootRom 的配置文件
USRCONFIG	usrConfig.c	指定命令行编译 VxWorks 的配置文件
BOOTINIT	bootInit.c	指定 BootRom 的初始化文件
DATASEGPAD	dataSegPad.c	指定数据段填充文件（工程中不用）
CONFIG_ALL_H	configAll.h	指定总配置文件

这些宏的缺省值在 defs.x86-win32 中定义，用户可以在自己的 Makefile 中重新定义，将宏指向自己定制的目录和文件。

2.3.3 组件配置

VxWorks 以静态函数库形式向应用提供服务，系统库由许多目标模块构成，各模块对应不同的功能组件。在连接生成映象时，VxWorks 库和普通的静态库一样，只有有函数引用的目标模块才会连接入最终映象。VxWorks 的组件配置机制主要用于程序映象裁减，当然还有另一个更重要的用途，支持 WindRiver 的零售策略。

组件配置窗口如图 2-10 所示（只有 Bootable 工程才有组件配置）。各组件按类别、分层次组织，组件用字体和颜色表示状态。正常字体名称表示已安装组件，但是未包含在工程中，例如图中的 spy；黑体名称表示工程已包含的组件，例如 timex；斜体名称表示未安装组件，也许需要单独购买，例如 CodeTEST；红色名称表示该分支出现组件配置冲突，例如符号表配置，不能同时配置 built-in 和 downloaded 符号表，去掉其中一个就可解决冲突。

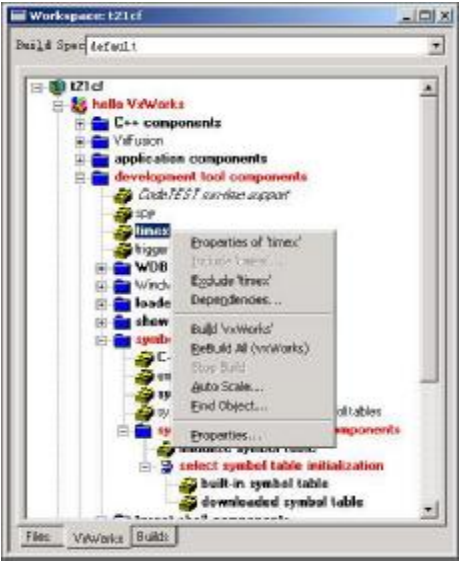


图 2-10 组件配置

组件的加入和删除操作很简单，在组件名上用右键菜单即可，如图 2-10 所示。

加入和删除组件时，需考虑相互之间的依赖关系。Tornado 会自动检查这种依赖，但删除组件时，依赖检查不太完全，可能会在工程中遗留无用的垃圾组件，如图 2-11 所示。

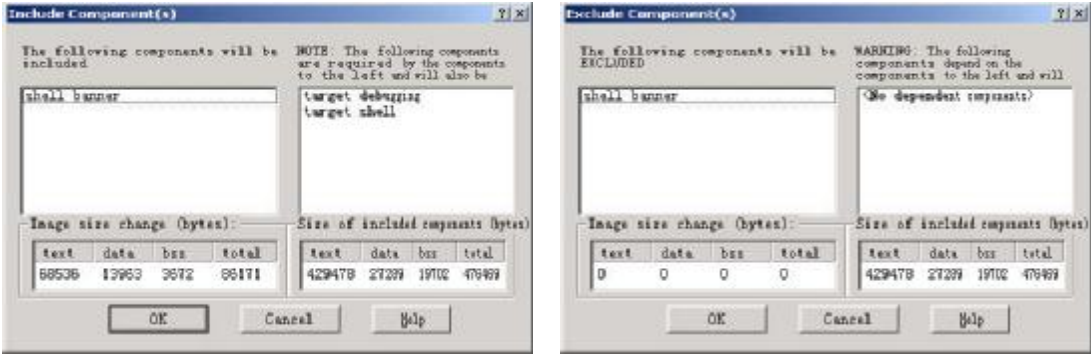


图 2-11 组件依赖

图 2-11 所示表示 target shell 组件夹中，组件 shell banner 的添加和删除操作。banner 组件

用于在启动时向目标机 Shell 输出 VxWorks 商标，没有什么实际的用处，加入后会增加映像大小和减慢启动速度，实际映像中多不用。不过此处假设需要它。

图 2-11 左图所示表示加入 banner 组件。图中右列表为 Tornado 自动依赖的组件，如 target debugging 和 shell，加入 banner 的同时，也需自动加入这两个依赖组件。图左下的 4 个数表示新加入组件（包括依赖组件）对映像大小的影响；右下的 4 个数表示加入新组件后，新程序映像的大小。

删除一个组件时，若有其他组件依赖它，这些依赖组件也会被删除。图 2-11 右图所示表示删除 banner 组件，工程中所有组件都不依赖于它，所以右列表为空。但有个问题，本例关注的是 banner 组件，banner 删除后，target debugging 和 shell 无用了，也应该删除，但 Tornado 不能自动探测到这点，因为它不知道用户实际需要哪个组件，只能保证加入组件能正常运行，而不能保证工程中现在包含的组件是必要的。

所以说，在某些情况下，同一组件的加入和删除操作是不可逆的。如果用户想保证工程中组件配置的清洁，需要自己做一些工作，比如在加入组件时，记下相关依赖组件，以便再次删除该组件时，能彻底清除依赖组件。

在定制 BSP 时，可以在 config.h 中包含或去掉某些组件定义，以辅助其上工程的组件配置。还是以上面的 banner 组件为例，可在 config.h 的尾部添加如下定义，和图形界面中配置效果相同。在修改 config.h 之前，需要先知道 banner 组件的依赖组件，以及他们的宏定义，这可通过组件配置窗口和组件属性得知。

```
/*需要 banner 组件*/
#define INCLUDE_DEBUG
#define INCLUDE_SHELL
#define INCLUDE_SHELL_BANNER
/*或者，不需要 banner 组件*/
#undef INCLUDE_DEBUG
#undef INCLUDE_SHELL
#undef INCLUDE_SHELL_BANNER
/*需要在这句之前*/
#endif /* __INCconfig */
```

对其他组件配置也可进行类似修改，当在该 BSP 上创建工程时，组件配置会自动完成，并且该 BSP 上多个工程都可使用同样的组件配置，不用再分别配置。

在工程创建时，Tornado 会根据 “\comps\src\configAll.h” 和 BSP 目录的 config.h 中的组件宏定义，自动进行组件依赖关系检查，以保证加入组件可以正常工作。工程创建后，也可在组件窗口选择右键菜单中的 “Auto scale...” 命令，进行组件依赖关系检查。

每个组件都有属性描述，可在组件窗口选择右键菜单中的 “Properties...” 命令查看。属性窗口包括 3 个方面的描述：General、Definition 和 Params。以 shell 组件为例，如图 2-12 所示。

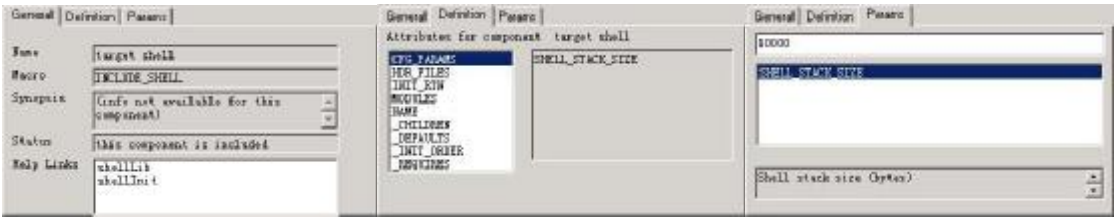


图 2-12 组件属性

- ✧ General 页中描述了组件的名称、宏定义、概要[Synopsis]、状态和帮助连接等。
- ✧ Definition 页中描述了组件的属性定义，如表 2-6 所示（斜体属性为上层的属性）。组件属性存放在“\target\config\comps\vxWorks\”目录下的 CDF 文件中，读者可参考本书 10.1.3 节的介绍。
- ✧ Params 页用于定制参数设置，结果保存在 prjParams.h 文件中。

表 2-6 组件属性说明

	target shell 组件值	说明
CFG_PARAMS	SHELL_STACK_SIZE	组件的配置参数（可能有多组）
HDR_FILES	shellLib.h	组件相关的头文件
INIT_RTN	shellInit();	初始化组件的入口函数
MODULES	shellLib.o	组件相关的目标模块（包含在系统库中）
NAME	target shell	组件名称
CONFIGLETES	无	若 INIT_RTN 不由库提供，则定义提供的源文件，如 usrBanner.c（在“\target\config\comps\”目录下）
SYNOPSIS	无	组件概要描述
REQUIRES	无	组件需依赖的组件
HELP	无	组件使用帮助
_CHILDREN	FOLDER_SHELL	归属组件夹
_DEFAULTS	FOLDER_SHELL	在组件夹中缺省包含该组件
_INIT_ORDER	usrShellInit	在该函数中调用 INIT_RTN，用来确定初始化顺序
_REQUIRES	INCLUDE_SHELL_BANNER	其他需要该组件的组件

2.4 编译器

编译器是嵌入式系统交叉开发环境的基础。编译器也是交叉的，在主机上事先完成程序代码的编译连接，而生成的程序映像则在目标机上运行。编译器如此设计适用于有限资源的嵌入式系统，能充分利用主机能力完成丰富功能，不用与应用程序竞争目标机资源。

大多 Tornado 版本使用 GNU 的编译器，新版的 Tornado 2.2 也使用 diab 编译器，而原 diab 编译器是作为单独产品发售的。本节描述主要针对 GNU 编译器。

GNU[GNU Not Unix]是自由软件基金组织[FSF]的一个开发软件集，实现类似 Unix 功能，由 Richard Stallman 首先倡导，gcc 和 emacs 是最著名的两个软件。FSF 崇尚自由软件精神，提出 GPL 和 LGPL 许可，支持自由软件开发，如 GNU 遵循 GPL，Linux 遵循 LGPL。WindRiver 在商业软件 Tornado 中使用了 GNU 的编译器(gcc)和调试器(gdb)，以支持 VxWorks

开发。这种商业软件和自由软件的融合，极不可思议。用户在购买 Tornado 时，无形中也为这部分自由软件支付了钱。网上也常见 Tornado 与 GPL 版权的争论。正因为 GPL 的要求，用户可以免费从 WindRiver 网站上获得编译器和调试器的源代码，进行定制修改和编译。

WindRiver 对 GNU 编译器和调试器的代码进行了修改，如多平台支持、性能优化等，以适用于嵌入式系统开发。

 WindRiver, “WhitePapers-Advanced Compiler Optimization Techniques”。

Tornado 2.0 使用 gcc 2.7.2 版，Tornado 2.1 和 Tornado 2.2 使用 gcc 2.96 版。而 Tornado 2.1 的随机文档却是 gcc 2.7.2 版的，需要使用 Tornado 2.2 中的文档。想要知道自己使用的 gcc 的版本，可在命令行窗口运行“ccarch -version”命令。

编译器是一个工具集，或说是工具链，用于完成程序映象的交叉编译和连接。其中 cc 是其中最基本的工具，另外还包括 cpp、make、ld、as 和一些映象文件工具。主机命令名随目标平台不同，一般为标准名加 CPU 名，如 cc386、as386 和 cccf 等。

2.4.1 编译配置

完成工程组件配置和应用源代码添加后，还需要完成编译器配置，才能得到用户需要的程序映象。编译配置在“Builds”窗口中进行，可以选择右键菜单打开编译属性配置对话框，如图 2-13 所示。

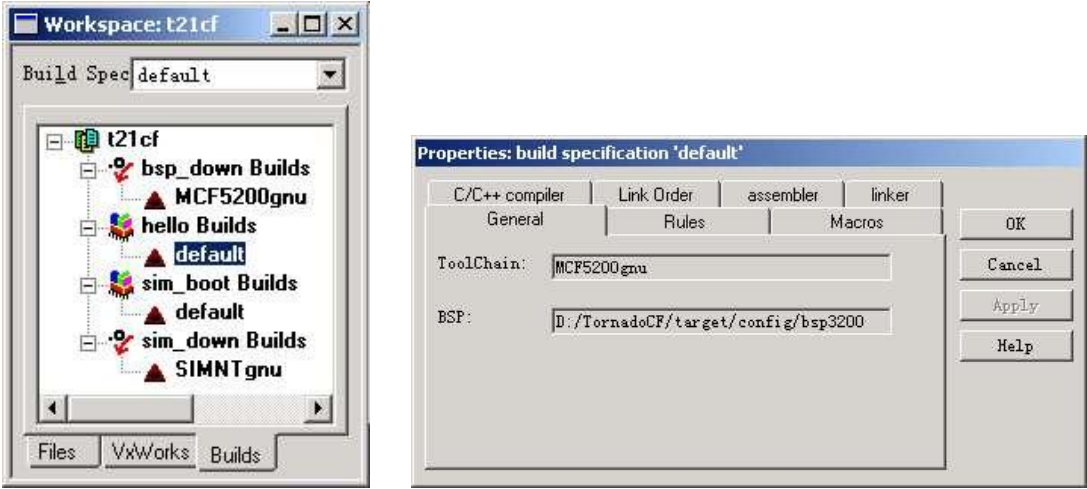


图 2-13 编译配置

对于 4 种典型工程类型，编译配置项略有不同，主要区别如表 2-7 和表 2-8 所示。图 2-13 所示“Rules”选型卡用于选择生成映象的类型，各映象类型如表 2-7 和表 2-8 所示。

表 2-7 工程类型和编译选项 1

	主机工具链工程	目标机工具链工程
工具链	SIMNTgnu	ARCHgnu
映象	VxWorks.exe	VxWorks
编译器	ccarch -mpentium -DCPU=SIMNT	ccsimpc -march -DCPU=ARCH
连接器	ldsimpc --subsystem=windows	ldarch -X -N

表 2-8 工程类型和编译选项 2

	Bootable	Downloadable
BSP	目标机 BSP 或 simpc	无
映象	VxWorks: 由 bootrom 加载, 在 RAM 中执行 VxWorks_rom: 写入 rom, 自启动, RAM 中运行 VxWorks_romcompress: 同上, 但采用了压缩算法 VxWorks_romResident: 代码在 ROM 中运行	xxxx.out: 部分连接映象 archive: 合成静态库 objects: 只生成目标文件
编译选项	-DPRJ_BUILD	
连接选项		-r
宏定义		PRJ_LIBS: 包括库 ARCHIVE: 库文件名称

图 2-13 所示“C/C++ compiler”和“assembler”选型卡用于设置编译器和汇编器的命令和选项, 常用选项如表 2-9 所示。其他选项或不同 CPU 特有的选项, 可参考 GNU 用户指南。

表 2-9 编译选项说明

-g	映象中是否包含调试信息, 会影响映象的大小	
-ansi	支持标准 C 语法, 若想在 C 中用 C++注释则去掉	
-D	编译器宏定义, 可用于代码条件编译, 用户可定义自己的条件编译宏	-DRW_MULTI_THREAD -D_REENTRANT
-O	指定编译器优化等级, 建议不使用优化	-O2
-fvolatile	防止编译器优化 volatile 变量相关的代码	
-I	包含头文件目录, 用户一般需要添加自己的	-ID:/TornadoCF/target/h
-P	预处理器不产生“#line”指令	
-Wall	报告所有警告, 建议添加	

图 2-13 所示“linker”选型卡用于设置连接器的命令和选项, 常用选项如表 2-10 所示。其他选项可参考 GNU 用户指南。

表 2-10 连接选项说明

-X	删除所有的临时局部符号
-N	设置 text 区和 data 区可读可写, data 段不页对齐
-r	产生可重定位输出, 可由连接器使用

Tornado 并不支持所有 GNU 选项, Tornado 的支持选项表可参考 GNU 用户指南的文档“GNU Options Supported by Wind River”。


图 2-13 所示“Macros”选型卡用于定义编译宏, Bootable 工程使用的宏如表 2-11 所示。

表 2-11 编译宏说明

ADDED_CFLAGS	附加编译选项	-g
CONFIG_ALL	指向 BSP 的 ALL 目录	..\allxxxx
EXTRA_MODULES	添加外部引用目标文件	app.o、app.out
LD_PARTIAL	部分连接命令	ldcf -X -r
LD_PARTIAL_FLAGS	部分连接选项	-X -r
LIBS	引用外部库文件	libARCHgnuvx.a、app.a
NO_VMA_FLAGS	CPU 相关参数, 如启动地址	

POST_BUILD_RULE	编译完调用命令，如映象文件改名	copy vxworks myvxworks
RAM_HIGH_ADRS	RAM 高地址，BootRom 加载地址	00200000
RAM_LOW_ADRS	RAM 低地址，VxWorks 加载地址	00001000
RES_LDFLAGS	用于生成 VxWorks_romResident	-m m68kelf_res_rom
VMA_FLAGS	CPU 相关参数，如启动地址	

虽然编译配置项比较多，一般用缺省选项即可，需要用户自己定制的并不多。大多数情况下，用户只需要做如下工作：选择自己的映象类型，确定是否包含调试信息（“-g”选项），选择优化等级（“-O”选项），设置警告级别（“-W”选项），包含自己的头文件路径（“-I”选项），引用自己的目标模块或库（用“Macros”选型卡）等。

 WindRiver, “Tornado 2.0 User’s Guide” 的 4.5 章节。

 WindRiver, “GNU Toolkit User’s Guide”。

2.4.2 Makefile 规则

Tornado 使用 GNU make 工具来管理工程源代码文件的编译和连接。make 能帮助用户完成工程编译的批处理过程，而不需要用户对文件一一操作。make 能有效完成重编译过程，根据文件和依赖更新编译，不用每次都编译所有文件。make 支持多编译目标，用户可选择使用。

make 的使用很简单。一般使用“make”命令就可完成缺省目标的生成。也可使用“make target”来完成用户需要的目标。还可使用“-D”选项对代码进行条件编译。

make 运行时缺省搜索当前目录下文件名为 Makefile 的编译规则文件，按 Makefile 中描述的规则进行工程编译，也可使用“-f”选项指定特定的编译规则文件。

Makefile 按一定格式书写，定义编译规则，规则的基本格式如下：

```
target ... : dependencies ...
[tab]     command
[tab]     ...
```

其中 target 为该规则的目标，一般为目标或映象文件名，也可以为动作代号，用于执行一系列命令。dependencies 为目标的依赖文件，可以包含多个文件，文件名之间用空格分隔，必须在一行写完，可以用“\”符号续行。依赖文件一般为源代码文件、目标文件和头文件等，它们的修改对目标有影响，依赖文件更新后，需要重新执行该规则以生成更新目标。command 为该规则需执行的命令，用来完成实际的编译或连接操作。行开头必须用“tab”，而不能使用普通的空格符。为了完成一个目标，允许执行多条命令。

如下所示规则在工程目录下的 Makefile 中定义，用来将 usrAppInit.c 源代码文件编译生成 usrAppInit.o 目标文件。

```
usrAppInit.o: $(PRJ_DIR)/usrAppInit.c \
    $(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/.../gcc-2.96/include/stddef.h \
    $(PRJ_DIR)/prjComps.h \
    $(PRJ_DIR)/prjParams.h
usrAppInit.o:
    $(CC) -g -m5200 -ansi ... -g -c $(PRJ_DIR)/usrAppInit.c
```

如下所示规则在“\target\make\rules.project”中定义，目标 clean 为动作代号，用来删除工程的中间文件，以进行整个工程的重编译（rebuild all）。

```
clean :
- $(RM) *.o *.rpo ctdt.c symTbl.c $(PRJ_TYPE)*
- $(RM) $(PRJ_DIR)/prjComps.h $(PRJ_DIR)/prjParams.h $(PRJ_DIR)/prjConfig.c...
```



```
- $(RM) $(PRJ_DIR)/libs.nm $(PRJ_DIR)/libs.size
```

命令前面的“-”符号用以忽略命令错误，即使没有需要删除的文件，也能保证后续命令的继续执行。

Makefile 中可以使用编译变量来简化其编写，变量用“\$()”形式进行引用。

Makefile 也支持条件分支，可使用 ifeq、ifneq、ifdef 和 ifndef 等，如下示例。


```
# top of vxWorks tree, 区分主机是 Unix 或 Windows, “#” 开头行为注释
ifeq ($(WIND_HOST_TYPE), x86-win32)
BIN_DIR=$(WIND_BASE)/host/$(WIND_HOST_TYPE)/bin
DIRCHAR=/
else
BIN_DIR=$(WIND_BASE)\host\$(WIND_HOST_TYPE)\bin
DIRCHAR=\
endif
```

Makefile 中可使用“include”指令来包含其他的编译规则文件，Tornado 中使用该指令来划分编译规则文件。工程目录下的 Makefile 由 Tornado 根据编译配置自动生成，并根据主机类型、工程类型、CPU 类型和工具链类型等，引用“/target/h/make/”目录下的编译规则模板，模板分为 3 类：defs.*、make.*和 ruels.*。工程编译规则文件层次如表 2-12 所示。

表 2-12

工程编译规则文件层次

Makefile	defs.project	defs. \$(WIND_HOST_TYPE)
		defs. \$(PRJ_TYPE)
		Make. \$(CPU)\$(TOOL)
	rules.project	rules. \$(PRJ_TYPE)

 WindRiver, “GNU Make Version 3.74”。

2.4.3 库使用

工程中需要引用外部二进制库时，可以使用 LIBS 和 EXTRA_MODULES 两个宏来指定。其实 VxWorks 本身就是作为库引入工程的，由 LIBS 指定。LIBS 和 EXTRA_MODULES 有区别：LIBS 用于引用静态库 (*.a)，库中包含多个独立的目标模块，模块之间的符号无引用解析；EXTRA_MODULES 用于引用目标模块 (*.o)，或者 Downloadable 工程生成的部分连接映象 (*.out，注意不要 munch 处理)。

如果用户有附加库的源代码，也可以将其加入系统库。在 Tornado 下建立一个 Downloadable 工程，将库源代码加入，设置输出为 archive 类型，在后面填上系统库 libARCHgnuvx.a 的绝对路径，编译后，新的库就会加入系统库中。或者填写一个不存在的库文件名，以生成一个独立的新库。

用户获得更新库或者其源代码时，可能需要替换原来系统库中的相应部分。如果更新改动不大，一般可以直接使用，而不会和原来的库发生冲突，因为连接是顺序进行的，若新库能解释所有相关引用符号，则旧库就不会连接到映象中。若改动较大，应该将旧模块从系统库中删除后，再引用新库，以避免符号重复定义。

2.4.4 命令行编译

在 Tornado 1.0 版中，一般都用命令行编译生成程序映象。而在 Tornado 2.0 版本中，集

成开发环境功能增强，可在 IDE 中方便完成整个编译过程。命令行编译对于工程开发已经过时，因而很少使用。不过工程管理对 BootRom 没有涉及，所以 BootRom 的编译一般还需在命令行下完成。

在进行命令行编译时，首先需要执行 `torvars.bat` 来配置环境。主要是设置了几个环境变量，如 `PATH`、`WIND_HOST_TYPE` 和 `WIND_BASE` 等。`torvars.bat` 在 “\host\x86-win32\bin\” 目录下，可以复制其他目录以方便执行。

没有了 Tornado 的图形界面，许多配置工作需要通过手动修改 Makefile 来完成。比如源代码的添加、依赖关系处理等。命令行编译的 Makefile 在 BSP 目录下，使用该文件的不同目标，可以选择生成 BootRom 或 VxWorks。

在命令行下生成的 VxWorks 和工程生成的 VxWorks 有所不同，区别如表 2-13 所示。

表 2-13 命令行和工程 VxWorks 区别

命令行 VxWorks	工程 VxWorks
usrConfig.c: 根据组件宏定义包含组件	prjConfig.c: 根据组件配置生成代码
\config\all\configAll.h	\config\comps\src\configAll.h
\config\all\bootInit.c	\config\comps\src\romStart.c
\target\src\config	\target\config\comps\src

2.5 调试器

Tornado 使用 GNU 的 gdb 作为底层调试工具，上层采用 CrossWind 作为图形界面，能同时支持图形界面和命令行调试。使用图形界面，可以方便进行源代码跟踪、断点设置、数据查看和修改等。命令行接口提供了丰富的特殊调试命令，可以满足复杂的调试需求。利用命令行命令，也可以使用 TCL 扩展自己定制的调试命令。与其他嵌入式调试器不同，Tornado 支持系统级和任务级调试，任务调试时只挂起任务，不会影响整个系统的运行。

在进行调试前，需要先启动 Target Server，代表被调试的目标板，Target server 的配置读者参考本书的 2.8 节。同一个 Target Server 可以支持多个 Tornado 调试，比如同时需要调试两个任务，可以启动两个 Tornado 通过同一 Target Server 分别调试两个任务。Target Sever 启动后，再启动调试器与目标机连接。

系统调试主要用于中断代码的调试，也可以用于任务代码的调试。在使用系统调试前，需要选择菜单 “Debug→Attach” 命令将调试器连接在系统上，此时系统会全挂起，在需调试代码处设置断点后，再用按钮 “Continue” 使系统继续运行。当程序运行到断点时，系统会再次挂起，此时可单步调试或查看变量。当调试完成后，可选择菜单 “Debug→Detach” 命令让调试器退出系统模式。大多数嵌入式系统调试器都只支持这种模式。

任务调试时，可以调试任何任务上下文中运行的代码。调试只对单个任务进行，不会对系统造成影响，其他任务还正常运行。这是 Tornado 所特有的调试模式，与系统模式相比，有明显的优点。建议任务级代码都用这种模式进行调试。

因为 Tornado 交叉调试环境需通信通道支持，如网络、串口等。要进行调试，必须保证通信通道的畅通。这对任务级调试来说，不是问题。而对于系统级调试来说，就比较麻烦。

因为系统模式中遇到断点时，整个系统都停止运行，包括支持通信通道的代码。VxWorks 中使用支持查询模式的通道驱动来完成这点。所以，要具有系统级调试能力，必须要选择支持查询模式的通信硬件，并选择 WDB 支持系统模式。

由于 Tornado 中新的任务级调试模式的存在，断点也有多种类型：全局断点、任务断点和临时任务断点。全局断点和任务上下文无关，只和调试的代码有关，任何任务运行该代码到断点，都会挂起该任务等待调试。而任务断点不但和代码相关，也和运行该代码的任务上下文相关。而同一段代码可以由多个任务共享，任务断点只针对指定任务。所以只有绑定 [Attach] 某个任务后，任务断点才有意义。而临时任务断点和任务断点相关，只起一次作用。在系统模式调试时，各断点作用相同。平时调试时，笔者多用全局断点，并在“Debugger”选项卡（选择菜单“Tools→Options”命令）中设置断点与任务自动连接，如图 2-14 所示，而较少使用任务断点和绑定单个任务，这是受以前的调试习惯的影响。



图 2-14 调试器配置

2.5.1 图形界面

源代码调试窗口即为编辑器窗口。需要调试某代码文件时，可通过工程管理的文件窗口打开该文件。再利用编辑器窗口的右键菜单来设置断点和查看变量。

Watch 窗口可以查看和修改全局符号，包括变量、数组、函数名等。在源代码窗口可以直接选择符号名加入 Watch 窗口。也可以先激活“Add to Watch”对话框，再手工输入符号名加入 Watch 窗口。特别是对大数组的查看，数据量过大会导致 Tornado 死掉，最好手动输入分项查看。有时加入到 Watch 窗口的符号不能被删除，会影响进一步调试，这时可以切换到其他的 Watch 子窗口，一共有 4 个子窗口。有时窗口的摆放也有问题，“dock view”处理得并不好，最好维持安装时的初始设置。笔者感觉，Tornado 并不是一个稳定的开发环境。

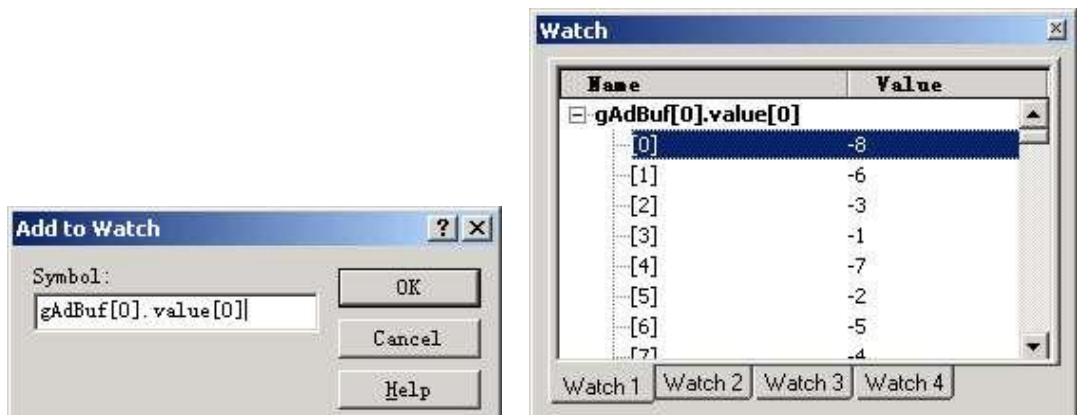


图 2-15 Watch 窗口

对于局部变量查看和修改，可用“Variables”按钮激活窗口。变量值的显示格式可以选择，如图 2-16 所示。

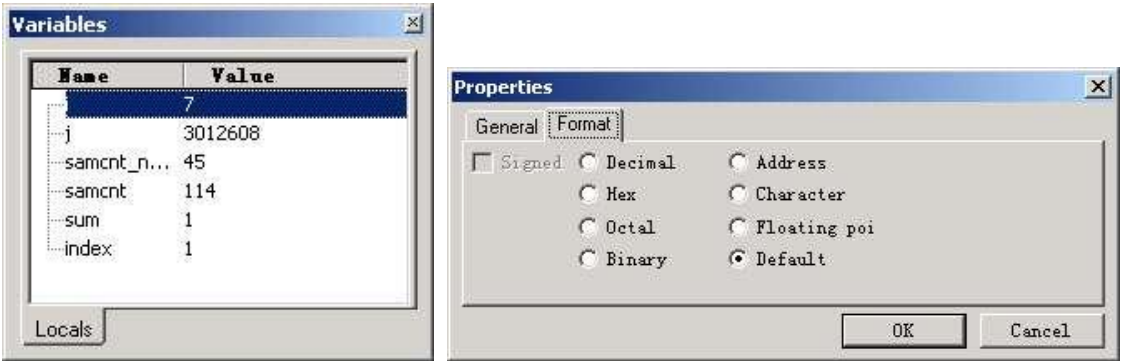


图 2-16 局部变量窗口

函数调用轨迹的查看，可用“Back Trace”窗口，如图 2-17 所示。最下层为根函数，一般为任务入口函数；最上层为断点设置函数。窗口显示函数的地址、入口参数和所处的文件等。点击函数条会进入相应的源代码文件。这些对调试函数异常时特别有用。

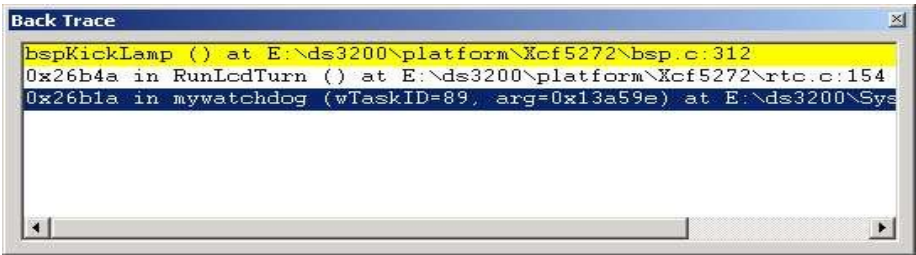


图 2-17 栈轨迹窗口

对于 CPU 寄存器的查看和修改，可使用“Registers”窗口；内存查看和修改，可使用“Memory”窗口，如图 2-18 所示。

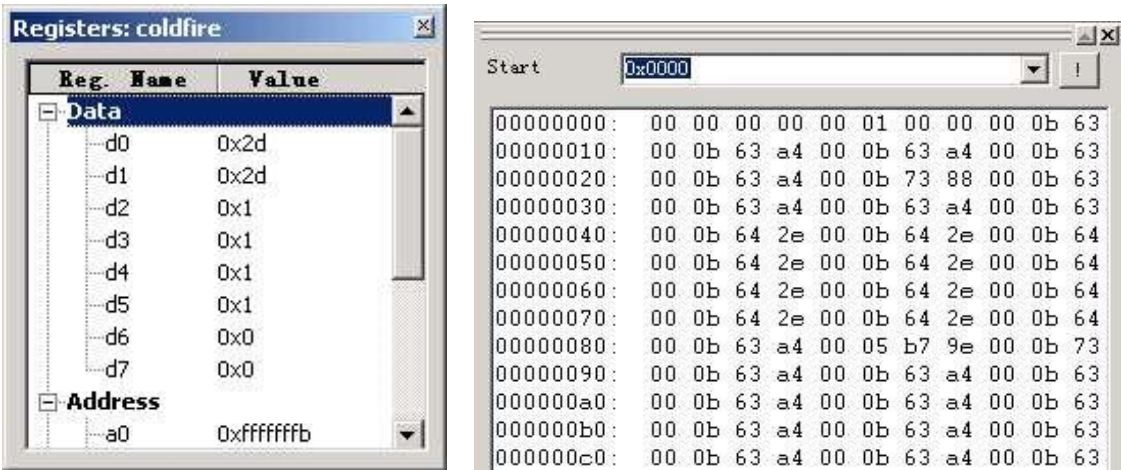


图 2-18 寄存器和内存窗口

应用代码入口函数通过 `usrAppInit` 函数启动，如果想调试应用的启动过程，可以去掉应

用初始化组件，让 VxWorks 运行时不自动启动应用代码。等 VxWorks 启动运行后，启动调试器，先在需要调试的启动应用代码处设置断点，再选择菜单“Debug→Run”命令启动对话框，手动运行应用入口函数（也可在 Shell 中直接运行），进行后续的调试工作，如图 2-19 所示。

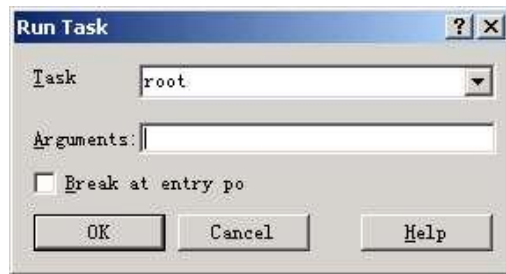


图 2-19 调试器运行函数任务

需要注意，这样做和系统自启动有点区别。系统自启动时，应用代码在 usrRoot 任务上下文中运行，任务优先级为 0；而手动启动，创建了一个调试任务运行应用代码，任务优先级为 100。这种区别可能导致一些高级别的应用任务在整个启动过程完成前先运行，调试需考虑这种影响。一个解决方法就是，一进入应用入口就使用 taskPrioritySet 函数提升调试任务优先级为最高 0 级。

2.5.2 命令行

对于一些复杂的调试操作，可使用 gdb 的命令行窗口，选择菜单“Debug→Debug Windows”命令启动命令行窗口。还可通过 help 命令获得调试命令的具体帮助，如图 2-20 所示，或参考 gdb 的用户手册。

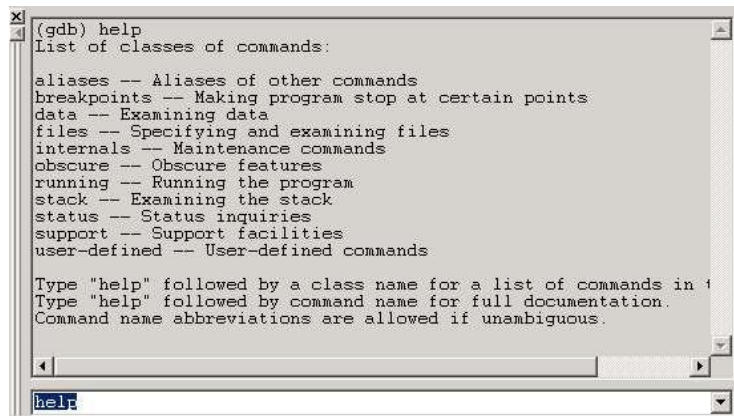


图 2-20 gdb 命令行窗口

2.5.3 目标模块调试

Tornado 调试器的另一个特点是使用 Downloadable 工程支持独立目标模块调试。工程生成的目标文件或部分连接的 out 文件，可以通过调试器下载到运行 VxWorks 的目标机，完成局部代码的开发调试工作。

下载的目标模块代码不能自启动任务，需要通过运行系统中代码，或调试器来启动其运行上下文。在调试器中，可选择菜单“Debug→Run”命令来运行调试代码函数；或者通过 WindSh 执行函数名运行；或通过 Target Shell 执行。不过这些方式都会让调试代码运行在父任务的

上下文中，如调试任务、tWdbTask 和 tShell 等，从而使调试遇到些问题。例如在 tShell 中运行函数，就不能设置断点，因为 tShell 具有 UNBREAKABLE 属性。所以，调试代码的入口函数应该创建新任务用于调试。

目标模块调试由底层的动态加载连接机制支撑。目标模块加载时，系统从目标机内存分配空间用于存放，并从 VxWorks 符号表中解析外部应用，并将模块本身的输出符号添加到符号表中，以便后续加载模块引用。不过需要注意两点，一是符号不能后向引用；二是主机加载和目标机加载有区别。目标模块的动态加载和连接机制在后面章节会详细描述。

Tornado 的目标模块调试机制允许用户进行系统的分工和增量开发。当基本的 VxWorks 系统构建起来以后，各开发者可以在此基础进行自己的开发工作，互相之间没有干扰，自己的模块调试开发完成后，可生成 out 文件（注意 munch）并加入基本的 VxWorks 中，作为下次开发的基础。这种方式大大地缩短了每个“编辑—测试—调试”的周期。当开发的 VxWorks 系统发布后，新功能的增加也可使用该机制，新功能对应一个目标模块，提供给最终用户，并修改启动脚本文件来完成新功能添加。这样做的一个额外好处是，系统可以组态构成，各功能模块可以单独发布和升级。

2.6 WindSh

WindSh(Tornado Shell、Wind Shell 或 wShell)允许用户下载应用模块，可以执行 VxWorks 和应用模块中的函数。可用于探索 VxWorks 操作系统、设计原型、交互开发和测试。通过调试通道向开发用户提供命令行用户界面。和 tShell 的功能类似，例如可使用 C 解释。wShell 与 tShell 有很多区别，例如对于同一目标机，可以同时启动多个 wShell，而 tShell 不允许存在多个实例；wShell 中运行的命令输出定向在 wShell 上，而 tShell 中运行的命令和程序代码缺省定向在 tShell 上。

wShell 可选择菜单“Tools→Shell...”命令启动；或按钮“launch shell”启动；或 Windows 命令行中用 WindSh 命令启动，例如“windsh 192.166.0.4@amine”，后面是 Target Server 的名称。启动后如图 2-21 所示。

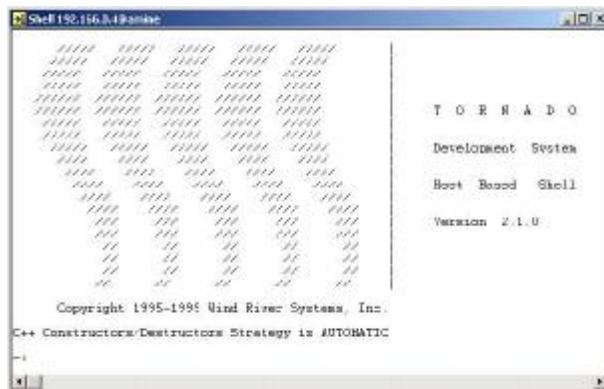


图 2-21 WindSh 窗口

退出 wShell 时，可执行 exit、quit 或<CTRL+D>。如果 wShell 和 Target Server 连接中断时，wShell 停止响应输入，可用<CTRL+BREAK>强制退出。

2.6.1 WindSh 结构

WindSh 由 4 部分组成，层次结构如图 2-22 所示。

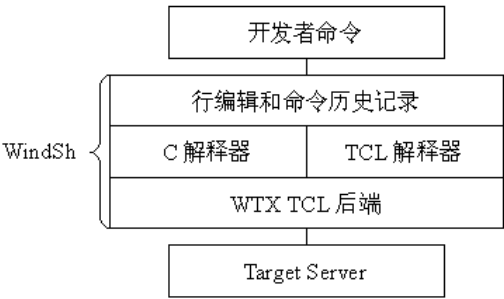


图 2-22 WindSh 结构图

上层与用户接口，处理行编辑和命令历史记录。底层为 WTX TCL 后端，与 Target Server 接口。中层为两个解释器：C 和 TCL，缺省使用 C 解释器。

平时使用时，WindSh 好像是一个无缝环境，Tornado 很好地将主机和目标机资源集成在一起，看起来 WindSh 中执行的命令好像完全是在目标机上完成的。而事实上，WindSh 中键入的命令需要经过好几层解释，一般需要由主机和目标机共同完成。和上面描述的层次结构对应，如图 2-23 所示。

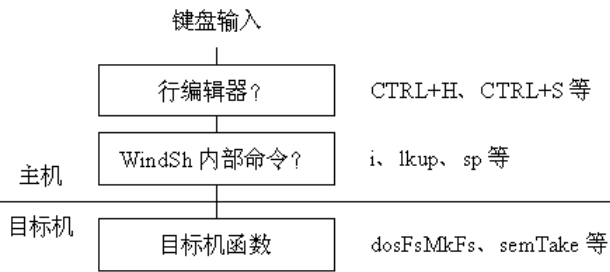


图 2-23 WindSh 命令分层解释

首先，检查输入是否是特殊的编辑组合键，若不是传下层；再检查输入是否是 WindSh 的内部命令，若不是传下层；再检查输入是否能由目标机处理。

2.6.2 WindSh 特点

WindSh 提供很多特性用于简化用户的开发调试工作。包括命令和路径补齐、命令函数的概要帮助打印、自动数据转换和 C 表达式计算等功能。

● 环境变量

WindSh 允许用户通过 TCL 函数 shConfig 修改和显示环境变量，以定制其行为。包括 IO 重定向、C++构造和析构函数、加载路径等的定义和处理。环境变量列表如表 2-14 所示。环境变量初值可以修改 shellUtilCmd.c 的 shellConfigInit 函数中的设定。

表 2-14 WindSh 环境变量

SH_GET_TASK_IO	用于设置调用函数的 IO 重定向模式
----------------	--------------------

	on: 将 WindSh 中函数执行输出定向在 WindSh 上 off: 将 WindSh 中函数执行输出定向在 Console 上
LD_CALL_XTORS	设置与构造、析构函数相关的 C++ 策略 target: WindSh 使用目标机上 cplusXtorSet 函数设置的值 on: 自动 off: 手动
LD_SEND_MODULES	设置加载模式 on: 模块传送到 target server, WindSh 能看见的模块都能加载 off: target server 必须看见模块, 才能加载
LD_PATH	设置加载模块的搜索路径
LD_COMMON_MATCH_ALL	设置加载器对公共符号的处理行为 on: 查找已存在符号, 若没有再创建 off: 直接创建
DSM_HEX_MOD	设置反汇编的格式 off: “符号+偏移”表示地址 on: 数值表示地址

● IO 重定向

环境变量 SH_GET_TASK_IO 的缺省值为 on, WindSh 中函数输出定向在 WindSh 上; 可以通过 shConfig 修改其值为 off, WindSh 中函数输出定向在 Console 上, 如下所示。

```

-> ?shConfig /*查看环境变量的缺省值*/
DSM_HEX_MODE = off
LD_CALL_XTORS = target
LD_COMMON_MATCH_ALL = on
LD_PATH = .
LD_SEND_MODULES = on
SH_GET_TASK_IO = on
-> printf "hello\n" /*WindSh 中有输出*/
hello
value = 6 = 0x6 = OVERFLOW + 0x2
-> ?shConfig SH_GET_TASK_IO off /*修改环境变量*/
-> ?shConfig SH_GET_TASK_IO /*查看环境变量*/
SH_GET_TASK_IO = off
-> printf "hello\n" /*WindSh 中无输出*/
value = 6 = 0x6 = OVERFLOW + 0x2

```

● 符号名和路径的自动补齐

若用户对符号或路径记忆不全, 可以键入部分字符, 再使用<CTRL+D>让系统自动补全, 其功能类似 Linux 中的 tab 键。如果键入字符不能惟一确定, 系统会列出所有的符合条件的候选项。

● 概要帮助显示

函数名称输入完成后, 再使用<CTRL+D>, 可以显示该函数的概要帮助。如果该函数在主机和目标机都存在, 主机的函数帮助被显示。若要显示目标机函数的概要说明, 需要在函数名前加 “@” 字符。若自己编写的函数想使用该功能, 需要遵循特定编码格式, 并进行一些操作。

● 数据转换

WindSh 按多种格式显示数据和字符,支持十进制和十六进制等,能将某些数据转换为“符号+偏移”格式,这对调试特别有用,如下所示。

```
-> 123
value = 123 = 0x7b = ' {' = INVALID_OPERATION + 0x6b
-> 'a'
value = 97 = 0x61 = 'a' = INVALID_OPERATION + 0x51
-> 0x88888
value = 559240 = 0x88888 = iosInit + 0x4
```

● 数据计算
在 WindSh 中, 支持 C 表达式运算, 如下面的例子所示。

```
-> (100+25)/5
value = 25 = 0x19
-> 4.3*5
value = 21.5
```

● 变量
在 WindSh 中可以引入变量, 如下面例子所示。

```
-> var1=1
new symbol "var1" added to symbol table.
var1 = 0x1ca160: value = 1 = 0x1 = INEXACT_RESULT
-> var2=2
new symbol "var2" added to symbol table.
var2 = 0x1ca158: value = 2 = 0x2 = UNDERFLOW
-> var3=var1+var2
new symbol "var3" added to symbol table.
var3 = 0x1ca150: value = 3 = 0x3 = UNDERFLOW + 0x1
```

● 主机和目标机名称冲突解决
若键入的命令为主机的 WindSh 命令,则执行主机命令,而不管目标机上是否有同名函数。而有时想要运行目标机的同名函数,则需要在名称前面加 “@” 符号,比如,“i” 执行主机命令,而 “@i” 执行目标机中的函数。

● 应用模块加载
对于开发, WindSh 最有用的特点是支持动态连接。使用命令 “ld”, 用户可以下载和连接应用的更新部分。

```
-> ld < /moduleDir/module.o
```

因为连接是动态完成的, 所以用户只需要一次处理一部分代码, 而不是整个应用。

2.6.3 WindSh 内部命令

WindSh 提供一些内部命令, 与目标机上一些函数功能类似, 但直接在主机上执行, 以减少对目标机性能的影响。注意区分近似的主机命令和目标机函数, 他们参考的文档不同, 一个属于 Tornado, 一个属于 VxWorks。如表 2-15~表 2-20 所示描述 WindSh 命令列表, 在 WindSh 中可以用 help 命令获得简要的帮助, 注意与@help 的输出不同。表中斜体表示 WindSh 特有命令, 目标机上没有对应函数。

表 2-15 WindSh 命令-任务管理

sp adr,args...	以缺省参数创建任务 (pri=100, opt=0, stk=20000)
sps adr,args...	以缺省参数创建任务, 使其处于挂起 (suspended) 状态
tr task	恢复挂起任务
ts task	挂起任务
td task	删除任务
period secs, adr,args...	创建任务来周期性调用函数

repeat n,adr,args...	创建任务来重复调用函数，可指定次数(0=forever)
----------------------	------------------------------

其中参数“task”，可以为任务 ID 或名称，或可省略针对当前缺省任务操作。

表 2-16

WindSh 命令-任务信息

i [task]	显示任务 TCB 信息概要
ti task	显示任务的详细 TCB 信息
w	显示任务挂起信息概要
tw task	显示任务挂起对象的详细信息
tt [task]	显示任务的调用栈轨迹

表 2-17

WindSh 命令-系统信息

devs	显示设备列表
lkup ["substr"]	从符号表中查找符号
lkAddr address	查找地址附近符号，有时直接输入地址值，也可查找符号
d [adr[,nunits[,width]]]	显示内存，可以指定地址、数目和宽度
l [adr[,nInst]]	反汇编内存，可以指定地址和数目
printErno value	显示状态值的名称描述
version	显示 VxWorks 映象的版本信息
cd "path"	改变主机的工作目录，对目标机没有影响
pwd	显示主机工作路径
ls ["path"[,long]]	显示目录内容，缺省为工作目录，还可选择显示格式
help	显示主机命令帮助列表
h	显示主机命令历史记录

表 2-18

WindSh 命令-调试命令

ld [syms[,noAbort][,"name"]]	将文件或 stdin 加载到内存， syms 表示符号添加方式（-1 = none，0 = globals，1 = all）
m adr[,width]	内存修改
mRegs [reg[,task]]	修改任务的寄存器值
b	显示断点
b addr[,task[,count]]	设置断点
bd addr[,task]	删除断点
bdall [task]	删除所有断点
c [task[,addr[,addr1]]]	从断点继续执行
s [task[,addr[,addr1]]]	单步执行程序
reboot	复位目标机，并复位 Target Server，重连 WindSh
sysSuspend	进入系统模式，系统挂起
sysResume	从系统模式回到任务模式，系统恢复

表 2-19

WindSh 命令-对象显示

show	在 Shell 中显示指定对象信息
browse	在主机工具 Browser 中显示指定对象信息
classShow	显示内核对象信息
taskShow	显示任务的 TCB 信息
taskCreateHookShow	显示任务创建挂入的 Hook 函数
taskDeleteHookShow	显示任务删除挂入的 Hook 函数
taskSwitchHookShow	显示任务切换挂入的 Hook 函数
taskRegsShow	显示任务寄存器
taskWaitShow	显示使任务挂起的对象的信息
semShow	显示信号量信息
semPxShow	显示 POSIX 信号量信息
wdShow	Watchdog 定时器信息显示
msgQShow	显示消息队列信息
mqPxShow	显示 POSIX 消息队列信息
iosDrvShow	显示驱动函数入口表
iosDevShow	显示设备列表，类似 devs 命令
iosFdShow	显示系统中文件描述符列表
memPartShow	显示内存分区信息
memShow	显示内存使用信息
moduleShow	显示模块信息，主机能看见 VxWorks 映象

表 2-20

WindSh 命令-网络信息显示

hostShow	显示主机表
icmpstatShow	显示 ICMP 协议状态信息
ifShow	显示网络接口信息
inetstatShow	显示所有连接信息
ipstatShow	显示 IP 协议信息
routestatShow	显示路由信息
tcpstatShow	显示 TCP 协议信息
udpstatShow	显示 UDP 协议信息
tftpInfoShow	显示 TFTP 信息

2.6.4 解释器 TCL

WindSh 提供 TCL 解释器接口，与 C 解释器之间用“？”字符切换，或者用“？命令”格式直接在 C 解释模式下执行 TCL 命令。命令提示符“tcl>”表示 WindSh 处于 TCL 解释模式下。

所有的 C 命令，都可在 TCL 模式下调用，加上双下划线前缀，如 help 命令，在 TCL 解释中为__help。不过有时直接用 wtxtcl 接口函数更方便。

TCL 模式下可以使用 shParse 执行 C 命令和函数，如下所示：

```
tcl> shParse {logMsg("Greetings from Tcl!\n")}  
tcl> shParse {sp appTaskBegin}
```

2.7 辅助调试工具

Tornado 除了提供 CrossWind、WindSh 等基本调试工具外，还提供一些辅助调试工具，如 Browser、VxSim 和 WindView 等。WindView 在本书 10.2.3 节中介绍。

2.7.1 Browser

Tornado 提供 Browser 来辅助调试。Browser 可以浏览目标机信息、内存信息、模块信息、对象信息、栈和任务信息等。Browser 的图形界面操作很方便，Shell 中也提供类似命令用于目标机信息查看。可以启动多个 Browser 分别查看信息。Browser 信息可以手动刷新，或自动刷新，刷新的间隔可以设定，如图 2-24 所示。



图 2-24 Browser

目标机信息显示 Target Server 名称、Tornado 版本、VxWorks 版本、WDB Agent 版本、CPU 类型、BSP 名称、内存大小、用户名、系统启动和当前时间、与目标机连接的主机工具。

可以使用如图 2-25 所示的窗口查看内存信息。“Tools”信息条表示 WDB 内存池大小，初始值为系统池的 1/16。WDB 池由主机工具使用，如目标模块加载等，如果大小不够，会再从系统池分配空间。“Application”信息条表示系统内存池大小，由目标机使用。信息条上的数据和阴影表示当前已使用的空间。信息条后数字表示内存池总大小。窗口下方为各模块内存使用信息的列表，点击模块条目可进入模块信息窗口，显示该模块更详细的信息。Shell 中也可使用 memShow 函数查看内存使用情况。需要注意的是，应该经常检查内存信息，防止出现内存泄漏。

ID	NAME	text	data	bss
0x38ac28	vxWorks	129390	937356	371536
0x38ae90	spyLib.o	2500	24	20
Total:		131890	937380	371556

图 2-25 Browser 内存查看

可以使用如图 2-26 所示的窗口查看 VxWorks 和加载目标模块信息。信息包括名称、符号表加载标志、格式、组号、段信息和符号列表等。



图 2-26 Browser 模块查看

可以使用如图 2-27 所示的窗口查看 VxWorks 系统中的任务信息，包括系统任务和用户任务。每个任务条目包括 TID、任务名和状态。点击任务条目会进入对象信息窗口，显示该任务对象的详细信息。如图 2-28 所示中 tShell 任务的信息显示，包括属性、栈和寄存器等。在 Shell 中可以使用“i”和“ti”命令查看任务信息。

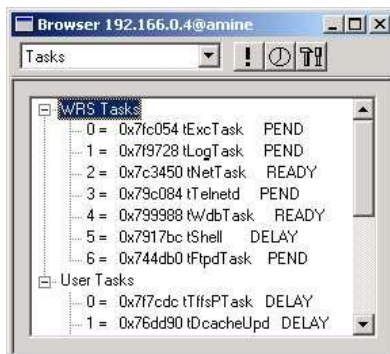


图 2-27 Browser 任务查看

如图 2-28 所示窗口用于显示对象信息，如任务、信号量等。要获得对象信息，需先获得对象的地址表示，如任务的 TID，再点击“Show”按钮。

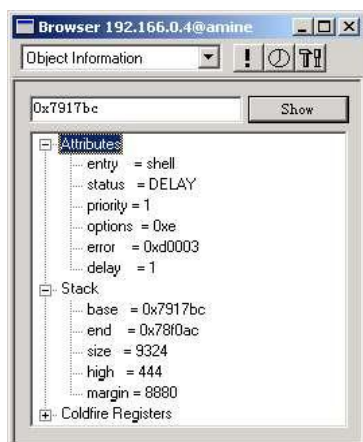


图 2-28 Browser 对象查看

如图 2-29 所示窗口用于显示各任务的栈使用情况。每个任务对应一个信息条，信息条的表示和内存信息窗口略有不同。阴影部分不表示当前使用大小，而是历史过程中使用的最大值。阴影上数值表示当前使用值，信息条尾部数值表示总的栈大小。应该经常检查各任务的栈使用情况，保证栈不会溢出。在 Shell 中也可以使用 `checkStack` 命令查看各任务的栈情况。



图 2-29 Browser 栈查看

如图 2-30 所示窗口用于显示各任务（包括中断，内核）对 CPU 占用率的情况。这种估计比较粗略，但也能检查某任务是否正常运行。正常情况下，IDLE 任务的 CPU 占有率最高，如果出现异常任务，异常任务的 CPU 占用率会急剧升高。IDLE 任务的 CPU 占有率越高，表示系统的负荷越轻，系统越高效。

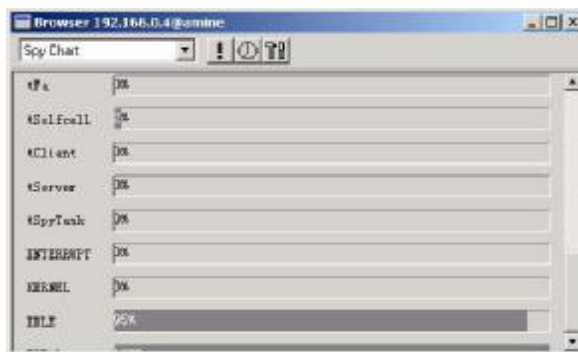


图 2-30 Browser Spy

使用“Spy Chart”时，若目标机上没有包括 spyLib，会动态加载 spyLib.o（如图 2-25 所示），并且启动 tSpyTask 任务。spyLib 使用定时器中断测量代码运行时间，所以比定时器中断高级的中断不能被监测。使用 spyLib 中的函数，可以在 Shell 中显示同样信息，而且包括任务运行的 tick 数，比百分比更为精准。

WindRiver, “VxWorks 5.4 Reference Manual” 的 “spyLib”。

2.7.2 VxSim

VxSim 是移植到主机上的 **VxWorks** 仿真程序。作为一个仿真目标机，无需目标机硬件，可用于原型开发和测试环境，形成完整的 **Tornado** 开发环境。**VxSim** 与运行在目标机硬件上的 **VxWorks** 类似，能进行应用下载或自己定制。通过通信机制，**VxSim** 可以获得自己的 IP 地址，能与同主机的 **Tornado** 工具通信，或与网络中其他节点通信。

VxSim 不使用真实目标机硬件，大多数 VxWorks 驱动都不可用，所以不适宜进行设备驱动程序开发。但软件环境仿真比较完善，高层代码可以在 VxSim 和 VxWorks 之间移植。VxSim 缺省使用“pass-through”文件系统（passFs）以直接访问主机上的文件。而大多数 VxWorks 目标机缺省使用 netDrv 来访问主机上的文件。

Tornado 中包含基本功能的 VxSim (VxSim-Lite)，初学者可以直接使用，而不需要目标机硬件和 BSP 定制，能快速熟悉 Tornado IDE 环境的使用。基本 VxSim 不包括网络，不支持多实例。完整版本的 VxSim 是作为单独产品销售的。完整 VxSim 为了仿真 VxWorks 目标机的网络 IP 连接，包括了使用 IP 操作的特殊驱动，如 ULIP 网络接口。它们作为 IP 网络的 IO 接口，允许 VxSim 进程在 IP 层寻址。当多个程序运行时，它们之间可以直接传送网络报文，因为报文都是在主机中处理，而主机可看作一个多接口的路由器。

● 基本 VxSim 使用

VxSim 一般会自动启动。例如用户请求下载 `simpc` 工具链的应用模块时，若还没启动 VxSim，VxSim 会自动启动，相应的 Target Server 也启动。也可以使用 VxSim 按钮，或通过命令行启动，如图 2-31 所示。

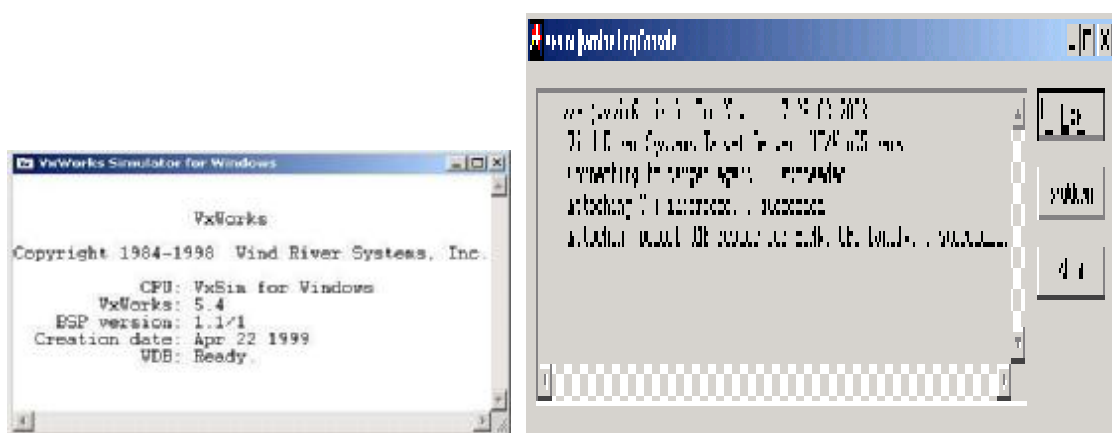


图 2-31 VxSim

VxSim 启动其实是运行 VxWorks.exe 主机程序，可以在启动时指明其路径。也可以在命令行窗口运行，并指定参数，“-p”用来设置处理器号，缺省为 0；“-r”用来设置 VxSim 仿真内存的大小，缺省为 2MB。注意数值和选项字母间不要有空格。

可以在 VxSim 窗口用<CTRL+X>键重启 VxSim，或者关闭窗口退出 VxSim。VxSim 的 WDB 使用 Pipe 与 Target Server 连接，同时支持系统和任务两种调试模式。

VxSim 可使用任何 VxWorks 文件系统，缺省使用 ntPassFs 来访问主机文件。open、read 等 VxWorks 函数，会调用主机库 libc.a 中相应的主机函数。

2.8 Target Server

Target Server 运行在主机上，管理主机工具之间以及主机和目标机之间的通信。Target Server 和其他 Tornado 工具可以运行在不同的主机上，主机间有网络连接即可。Target Registry 服务负责管理所有的 Target Server，用户都通过 Registry 来访问各 Server。

Target Server 与目标机的 Target Agent（WDB Agent）通信，该 Agent 以任务的形式运行在目标机上。Tornado 工具向 Target Server 发送请求，根据请求类型，有的请求由 Target Server 自己处理，或者转发到 Target Agent 处理。

Target Server 由多个单元组成，包括主机工具通信单元、与 Target Agent 通信单元、目标模块管理单元（加载/卸载）、符号表管理单元、目标机内存管理单元和虚拟 IO 管理单元等。

2.8.1 配置和启动

如图 2-32 所示为 Target Server 的配置对话框，可选择菜单“Tools→Target Server→Configure...”命令打开。在该对话框中，可以新建、复制和删除 Target Server 配置。描述为该配置的标识，由用户自己填写。描述下的选择项用于将该配置添加到菜单上，方便直接启动。选择项下方为目标机的别名填写区，对应命令选项“-n”。再下方为属性设置区，有多个属性，与 Target Server 的多个组成单元对应，下面会分别介绍。属性设置区下方紧接着目标机名称或 IP 地址的填写，如果是网络连接方式，名称和目标机有对应关系，其他方式可以随意填写名称。



图 2-32 Target Server 配置

配置对话框最下方为 **tgtsvr** 的配置命令行显示，不能进行修改。各种在图形界面中的配置都最终反映在各命令选项上（后面在括号中表示对应的命令选项），Target Server 的启动其实就是执行该命令行，可以在 Windows 的命令行窗口复制执行该命令行，或者写成批处理文件，同样能启动 Target Server。

● 授权和访问限制

tgtsvr 能限制用户访问以增加系统的网络安全性，授权配置如图 2-33 所示。如果选择了“Lock On Startup”（-L），则只有启动 tgtsvr 的用户的进程可以访问该 Target Server。若不选择“Lock”选项，tgtsvr 缺省查找“\$WIND_BASE/.wind/userlock”文件来获得授权用户 ID；若没有该文件，则无访问限制，或者用户自己设置指定“User ID File”（-u）。

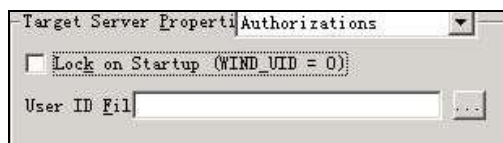


图 2-33 Target Server 授权配置

实际中，一般选择简单的“Lock”项，使得只有本机用户可以访问 Target Server，而拒绝所有其他用户的访问，最大限度地保证主机的安全性。这在使用可读可写的 TSFS 文件系统时特别重要，因为主机文件系统通过 Target Server 共享了，容易受到破坏。

● 通信后端[Back End]

根据不同的目标机，可以选择不同的 Target Server 通信后端与目标机的 Agent 通信。其中，wdbrpc 用于网络连接（-B wdbrpc）；wdbserial 用于串行通道连接（-B wdbserial）；wdbpipe 用于 VxSim 仿真目标机连接（-B wdbpipe），如图 2-34 所示。各后端支持的动态库

位于“\host\x86-win32\lib\backend”目录下。



图 2-34 Target Server Back End 配置

这里选择的后端类型，需要和 VxWorks 组件配置的“select WDB connection”类型相一致，才能顺利建立主机和目标机之间的 WDB 连接。

“Timeout”参数用于设置连接断开的确认时间（-Bt），缺省为 1S，有的后端类型忽略该参数。当连接超时，会进行连接重试，最大重试次数由“Re-try”确定（-Br），缺省为 3 次，一般可以用缺省设置。

如果选用“wdbserial”，还要根据需要选择主机串行端口（-d COM2），缺省使用第 2 串口，选择波特率（-b 9600），缺省为 9600。这两个参数和目标机的连接设置相关，对应参数为 WDB_TTY_CHANNEL 和 WDB_TTY_BAUD，可以在 VxWorks 的组件配置窗口设置。

● 核心文件和符号表

Target Server 依赖主机上存放的 VxWorks 映象文件来完成调试工作，需要指定映象核心 [Core] 文件的存放路径（-c）。

Target Server 在主机上为目标机 VxWorks 映象维护了一张符号表，该表根据核心文件建立，从文件中取得符号名称和内存地址，加载其他模块时进行引用符号解析，计算重定位地址。如图 2-35 所示，若选择“Global Symbol”，则只有 Core 文件中的全局符号加入主机符号表中，缺省为该选择，没有对应的命令选项。若选择“ALL Symbol”（-A），全局和局部符号都将加入符号表。若选择“No Symbol”（-N），则不建立主机符号表。

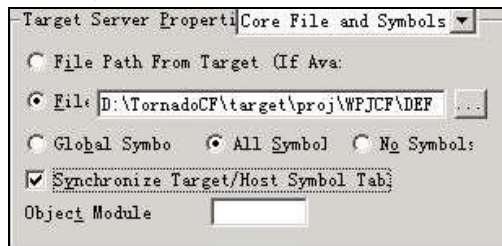


图 2-35 Target Server 符号表配置

为了实现动态加载和其他功能，目标机上也存在一张符号表。没有动态加载模块之前，主机符号表和目标机符号表是一致的，但当某一方加载了模块，新模块的符号只添加到该方符号表中，除非选择了“Synchronize Target/Host Symbol Table”（-s）。

Target Server 可以处理多种目标模块格式，如 AOUT、COFF 和 ELF 等。通常启动时，会分析 Core 文件的格式，确定要加载目标模块的格式。或由“Object Module”参数（-f）明确指定，支持的格式名称可在“\$WIND_BASE/host/resource/target/architecture.db”文件中查到，

各 CPU 类型对应的格式不同。可以对 Target Server 进行扩展，以支持新的目标模块格式。

- 目标机内存管理

Target Server 负责管理目标机上 WDB Agent 内存池。该内存区主要由主机加载目标模块时使用。若内存区不够，系统会自动增加，若设置了“-noGrowth”选项，则不进行自动增加，而报告错误。

Target Server 在主机上为该内存开辟了缓存，用于提高主机工具访问该内存的效率。缓存大小缺省为 1MB，也可以指定大小（-m），如图 2-36 所示。



图 2-36 Target Server 缓存配置

- 记录文件

Tornado 工具和 Target Server 之间通过 RPC/XDR 机制进行通信。工具的请求、Target Server 的回答或事件，都遵循 WTX 消息规约格式。这就是为什么主机工具和 Target Server 可不在同一主机上，而能在网络上分布使用的原因，类似软总线机制。所有的 WTX 请求都可以记录在一个文件中，由“WTX Log File”（-Wd）选项指定具体文件，也可设置记录文件的最大长度（-Wm），以及用“Filter”选择记录 WTX 请求（-Wf，缺省设置为 WTX_EVENT_SET），如图 2-37 所示。

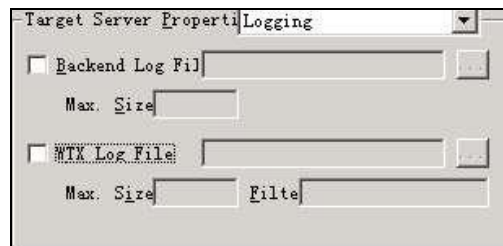


图 2-37 Target Server 记录文件配置

同样，Target Server 和 Target Agent 之间的 WDB 请求也可以记录下来，记录文件由“Backend log File”（-Bd）指定，并可设置记录文件的最大长度（-Bm）。

- 其他选项

其他未在前面图形界面出现的选项，可以在“Other Options”中补充设定，这样可充分利用 tgtsvr 的功能，减少对图形界面的依赖，如图 2-38 所示。

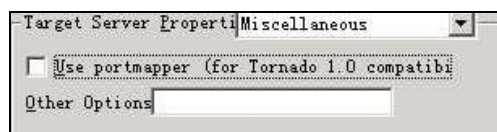



图 2-38 Target Server 其他选项配置

Target Server 属性中，关于“Console and Redirection”和“Target Server File System”的配置在后续的两小节中介绍。

 WindRiver, “Tornado Reference” 中 “Tornado Tools” 下的 “tgtsvr”。

2.8.2 虚拟 IO

虚拟 IO[Virtual IO]是在主机和目标机之间的 WDB 连接上，使用多路复用机制，分离出来的通信通道，用于传输主机和目标机之间的 IO 数据。虚拟 IO 可以提供多个逻辑通道，与其他主机工具共享调试通道。Virtual Console 和 WindSh 等主机工具会使用虚拟 IO 通道，例如 Virtual Console 使用了通道 0 (/vio/0)。

从 VxWorks 应用的角度来看，虚拟 I/O 通道只是通常的字符设备，名称如“/vio/0”、“/vio/1”等。使用与其他字符设备相同的 VxWorks 调用来操作虚拟 I/O。在逻辑结构上与串口设备对等，算是一种虚拟设备，用 `devs` 命令就可以清楚地看出这点。

要想使用 VIO，目标机的 VxWorks 需要包含“VIO driver”和“WDB virtual I/O library”两个组件。包含 VIO 驱动后，系统中会增加“/vio”设备，目标机各任务可根据需要打开逻辑通道进行读写，与主机进行数据交换。

下面显示来自 Virtual Console，说明目标机 VIO 通道和主机工具的对应关系。

-> iosDrvShow							
drv	create	delete	open	close	read	write	ioctl
1	8f842	0	8f842	8f870	8fea0	8fdda	8f8aa
2	0	0	8c05c	0	8c08e	8c0da	8c1da
3	9f5ea	9f96c	9f2d6	9f68c	a00e2	a0104	a0952
4	0	0	0	79292	79eb8	79a42	bf664
5	7ea94	7eba4	7ed26	7eddc	7f7e8	7f928	7fa7a
6	638fa	0	638fa	639c2	8fea0	8fdda	63a2c
7	63c26	63e22	63c50	63f12	63fba	6405a	640fa
value = 608 = 0x260							

第 6 号 Driver 为 VIO 驱动 ()，提供接口函数，直接挂入 IO 系统。

iosDevShow 显示 IO 系统中的实际设备列表，其中“vio”对应虚拟 IO 设备。iosFdShow 显示系统中正在使用的文件描述符，“/vio/0”对应一个虚拟 IO 逻辑通道，用于支持正在显示这些结果的主机 Virtual Console。可以看出，VxWorks 的 3 个标准 IO 描述符都定向在该通道上，也就是 Virtual Console 上，通过该 Console 窗口，可以使用 Target Shell 功能，这特别适合于缺乏通信通道的目标机系统，如单串口系统。

-> iosDevShow (同 devs)		-> iosFdShow	
drv	name	fd	name drv
0	/null	3	(socket) 4
1	/tyCo/0	4	(socket) 4
1	/tyCo/1	5	(socket) 4
5	host:	6	(socket) 4
6	/vio	7	(socket) 4
7	/tgtsvr	8	/vio/0 6 in out err
3	/ds3200/	9	(socket) 4
3	/data/	value = 1568 = 0x620	
value = 25 = 0x19			

虚拟 I/O 支持多逻辑通道，如主机连续启动了两个 WindSh，系统会为第 2 个 WindSh 分配新的通道，从下面的 iosFdshow 显示可看出，第 1 个 WindSh 对应 “/vio/1”，第 2 个 WindSh 对应 “/vio/2”。

-> iosFdShow		-> iosFdShow	
fd name	drv	fd name	drv
3 (socket)	4	3 (socket)	4
4 (socket)	4	4 (socket)	4
5 (socket)	4	5 (socket)	4
6 (socket)	4	6 (socket)	4
7 (socket)	4	7 (socket)	4
8 /vio/0	6 in out err	8 /vio/0	6 in out err
9 (socket)	4	9 (socket)	4
10 /vio/1	6	10 /vio/1	6
value = 1568 = 0x620		11 /vio/2	6
		value = 1568 = 0x620	

窗口 Virtual Console 通过 Target Server 带 “-C” 选项启动，连接在虚拟通道 0 上。目标机上的任务可以使用该通道来向 Console 窗口发送字符，或读取字符。如果 tgtsvr 使用了 “-redirectIO” 选项，会重定向目标机的标准 IO。如果只定义 “-redirectIO”，未定义 “-C”，目标机的标准 IO 会重定向到 tgtsvr，但没有 Console 窗口用于显示信息，只传送事件给连接的 WTX 工具。

tgtsvr 使用了 “-redirectShell” 选项可以将 Target Shell 的标准 IO 定向在虚拟 Console 上，该选项必须和 “-C” 一起使用。该 Shell Console 可以完成 WindSh 无法进行的操作，比如从目标机的本地文件系统加载目标模块，WindSh 中不能读取目标机的本地文件系统。

虚拟 I/O 在主机和目标机的配置如图 2-39 所示。事实上，即使主机没有明确配置，WindSh 也一直在使用虚拟 I/O 通道。图 2-39 所示的左图配置 Virtual Console 的使用，需要设置上面描述的 3 个 tgtsvr 启动选项，即图中命令窗口的阴影选项，一般只用配置后面两个选项。图 2-39 所示的右图为目标机 VxWorks 组件配置，一般需要虚拟 I/O 相关的两个组件，即图中红圈标注处。

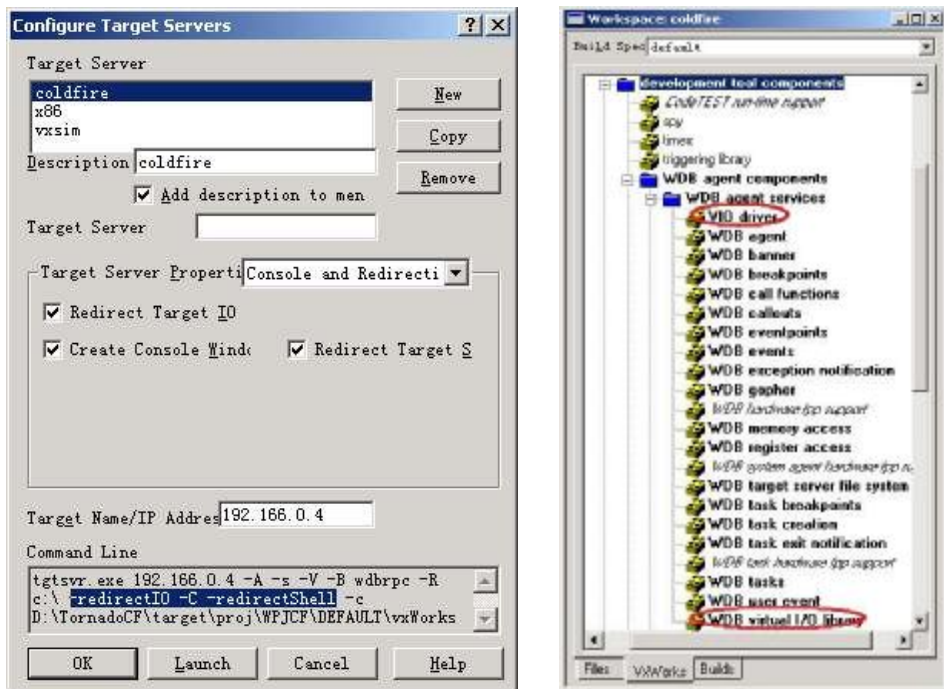


图 2-39 Target Server VIO 配置

WindRiver, “Tornado 2.0 User's Guide” 的附录 C 下 Target Server 相关条目。

2.8.3 TSFS

Target Server 文件系统用于向目标机提供虚拟文件系统，目标机对 TSFS 的操作和本地文件系统一样。TSFS 也是利用类似虚拟 IO 通道的机制，只不过用该通道来访问主机的文件系统。

要想在目标机上使用 TSFS，必须要运行 Target Server，并需进行一些配置，如主机的共享目录、读写权限等，如图 2-40 所示。共享目录在“Root”（-R）中设定，目标机只能访问该目录，目标机中对应路径为“/tgtsvr”。读写权限由下面两选项确定，缺省为读模式，读写模式对应命令选项为“-RW”。如果想使用 WindView，必须设置可写。

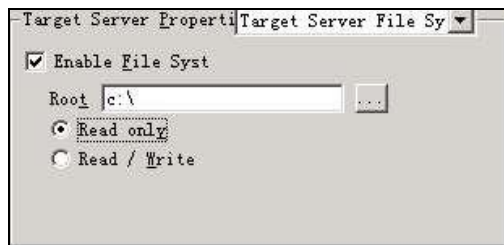


图 2-40 Target Server TSFS 配置

为了使用 TSFS，目标机 VxWorks 映象中需要包括“WDB target server file system”组件。该组件被包括后，目标机中会出现“/tgtsvr”设备，可以用“devs”命令看到，对应 Target Server 共享的主机目录。目标机中通过“/tgtsvr”来使用主机文件系统。


串行连接多使用 TSFS 来下载 VxWorks 启动。TSFS 使目标机可以直接访问主机文件系统，比配置使用 PPP 或 SLIP 更为简单。

如果通过串行通道使用 TSFS 加载 VxWorks，而没有额外的通信通道用于 BootRom 的 Shell，可以复用 TSFS 通道来使用主机上的虚拟 Console，使其作为 BootRom 的 Shell 使用，如进行启动参数修改等。若想使用该功能，需要在 config.h 中定义 INCLUDE_TSFS_BOOT_VIO_CONSOLE。该功能只用于 BootRom，对 VxWorks 没有影响，在 VxWorks 中使用标准的虚拟 IO 机制来使用 Target Server Console。该功能是否包含，和 CONSOLE_TTY 的定义有关，值为 NONE 或 WDB_TTY_CHANNEL，bootConfig.c 会自动包含该功能定义。包含该功能后，自启动过程会中断，等待用户介入修改启动参数，手动加载运行 VxWorks。

Target Server 的 Log Console 上能看见 TSFS 目录操作信息。

 WindRiver, “Tornado 2.0 User's Guide” 的 2.5.7 和 5.2.2 章节。

 WindRiver, “Tornado 2.0 User's Guide” 的附录 C 中的 tgtsvr 条目。

 WindRiver, “Tornado 2.0 User's Guide” 的 4.7 章节的 “TSFS Boot Configuration”。

2.8.4 Tornado Registry

Tornado Registry 用于维护一个数据库，存储 Target Server、目标机标识符和 RPC 端口号等，如土 2-41 所示。Target Server 的标识符是惟一的，由目标机名和主机名共同构成，形如 “targetName@serverHost”。Registry 管理所有 Target Server，防止两个 Target Server 连接到同一目标机；允许各个 Tornado 工具和 Target Server 建立连接。

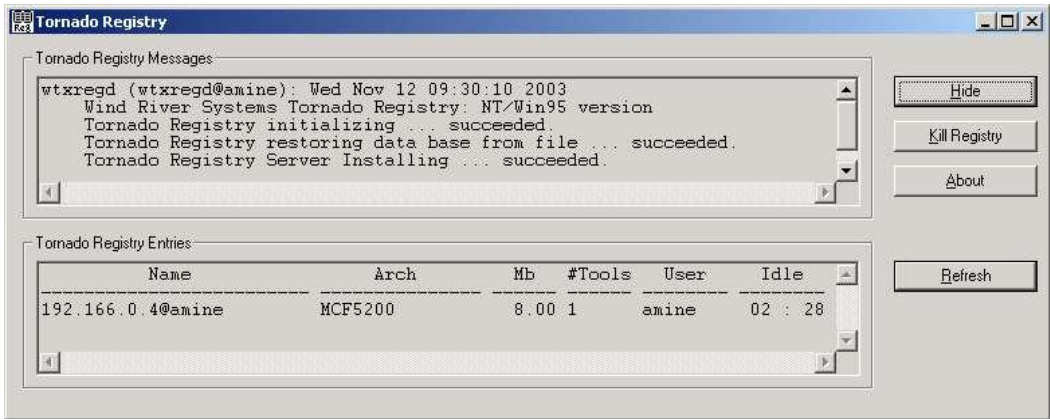



图 2-41 Tornado Registry

Registry 可以作为 Windows 服务运行，在系统初始化时启动。或者 Tornado 启动时运行 Registry。Registry 不接收命令行参数，缺省参数在 Windows 注册表中定义，用户可以修改这些参数以进行定制。Tornado 工具需要知道 Registry 运行的主机，该主机名称或 IP 为 Windows 注册表中定义的环境变量 “WIND_REGISTRY”，一般值为 “localhost”，使用 Tornado 安装的本机。

Registry 自动监测 Target Server 是否活着，认为已经死掉的 Target Server 将被从 Registry 队列中移除。Registry 使用 Ping 操作来探测 Target Server 是否正常。Ping 延迟参数用来控制平时 Ping 操作的频率，缺省为 120S。Ping 失败次数参数用来判断 Target Server 失效，缺省为 5 次。

Registry 异常关闭时，会将数据库保存在备份文件中，文件路径由 “-d” 选项指定，缺省

文件为“\$WINDBASE/.wind/wtxregd.hostname”。Registry 再次启动时，会从该备份文件中恢复数据库。若在恢复时，和 Registry 在同一主机上的 Target Server 没有反应，这些 Target Server 会从 Registry 队列中被移除。

 WindRiver, “Tornado Reference” 中 “Tornado Tools” 下的 “wtxregd”。

2.9 辅助小工具

一些主机小工具位于“\host\x86-win32\bin”目录下，用于辅助开发过程。

Tornado 中，除了编译器和调试器是 GNU 的以外，还提供了一些 GNU 小工具，用于对目标模块和映象的处理。命令名称为标准名加目标 CPU 名，如用于 Coldfire 的 arcf。如表 2-21 所示只列出命令的标准名，实际命令名需要用户根据自己平台确定。

表 2-21 GNU 小工具

ar	管理目标模块静态库[archive](*.a)，如模块加入、删除等
nm	显示目标模块符号表
objcopy	进行目标模块文件的格式转换
objdump	显示目标模块文件的信息
ranlib	为静态库产生索引，并将索引存储在库中，类似“ar -s”，用于加快连接
size	显示目标文件或静态库中各模块的段大小
strings	显示文件中的串
strip	删除目标文件或静态库中的所有符号
c++filt	将修饰过的 C++函数符号名转换为函数名加参数格式

 WindRiver, “GNU Toolkit User’s Guide” 中的 “The GNU Binary Utilities”。

如表 2-22 所示为映象格式转换工具，这些工具存在和目标机 CPU 有关，一般一种 CPU 支持一种映象格式，多为 ELF。

表 2-22 映象格式转换工具

aoutToBin	将 AOUT 格式转换为二进制格式
coffHex	将 COFF 格式转换为 HEX 格式
coffToBin	将 COFF 格式转换为二进制格式
elfHex	将 ELF 格式转换为 HEX 格式
elfToBin	将 ELF 格式转换为二进制格式（某些平台的该工具有 Bug，如 PPC）

如表 2-23 所示为符号处理工具，后续章节还会详细介绍这两种符号表的机制。

表 2-23 符号处理工具

makeStatTbl	产生 Errno 值符号表文件
makeSymTbl	产生目标机符号表文件

如表 2-24 所示为文档处理工具，Tornado 的文档就是由这些工具生成。若是开发 Tornado 或 VxWorks 第三方产品的用户，可能需要用这些工具来生成自己产品的文档。

表 2-24 文档处理工具

htmlBook	为 HTML 文档产生层次索引
htmlIndex	为 HTML 文档产生交叉引用
refgen	从源代码中提取文档

syngen	从源代码中提取概要[Synopsis]文档
--------	-----------------------

如表 2-25 所示为与引导相关的工具。

表 2-25 引导相关工具


mkboot	做 VxWorks 启动盘，综合 vxsys 和 vxcopy 操作
vxsys	写启动盘的引导扇区，使 BootRom 能得以加载
vxcopy	将 BootRom 复制到启动盘上，要求连续存放，需 chkdisk 检查
vxload	从 Dos 环境直接启动 VxWorks

如表 2-26 所示是其他的辅助工具。

表 2-26 其他辅助工具

binToAsm	将二进制文件转为汇编文件，定义 binArrayStart 和 binArrayEnd 符号
deflate	压缩映像文件，使用 zlib 算法，与 VxWorks 的 inflateLib 对应
memdrvBuild	产生 memDrv 使用的文件系统源代码文件，参考本书的第 7 章
munch	和 nm 工具合用，产生 ctdt.c，支持 C++静态对象的构造和析构
romsize	计算映像占用 ROM 的大小，用户可指定空间大小检查映像是否合适
vxsize	计算 VxWorks 映像占用 RAM 的大小，可指定大小检查映像是否合适
vxencrypt	计算用户密码的密文，与 VxWorks 的 loginLib 配合使用

Tornado 中还提供 Unix 仿真命令，如 cat、cp 和 rm 等。

 WindRiver, “Tornado Reference” 的 “Tornado Utilities”。

2.10 常见问题解答

● 如何根据工程文件(*.wpj)手动生成 prjParams.h 和 prjComps.h 等文件

解答：工程文件中包括足够的信息来重新生成工程编译所需的各种文件。首先将工程文件复制到新工程目录中，进入命令行窗口，运行 torvars.bat，并切换到新工程目录下，运行如下命令，则会生成 prjConfig.c、linkSyms.c、prjComps.h 和 prjParams.h。

```
>wtxtcl g:\tornadocf/host/src/hutils/configGen.tcl hello.wpj
```


根据工程文件也可生成 Makefile，不过需要自己编写一个 TCL 文件，如下所示：

```
# ===== BEGIN genmake.tcl =====
# Project Library
source [wtxPath]/host/resource/tcl/app-config/Project/prjLib.tcl
# Utility functions
source [wtxPath]/host/resource/tcl/app-config/Project/Utils.tcl
# Bootable vxWorks image
source [wtxPath]/host/resource/tcl/app-config/Project/prj_vxWorks.tcl
# Downloadable vxWorks Module
source [wtxPath]/host/resource/tcl/app-config/Project/prj_vxApp.tcl
set hProj [prjHandleGet $argv]
set prjDir [file dir [prjInfoGet $hProj fileName]]
if { [prjTypeGet $hProj] == "::prj_vxWorks" } {
    ::prj_vxWorks_hidden::makeGen $hProj $prjDir/Makefile
} else {
    ::prj_vxApp_hidden::makeGen $hProj $prjDir/Makefile
}
# ===== END genmake.tcl =====
```

可以将此 TCL 文件直接放到新工程目录中使用，如下所示。

```
>wtxtcl genmake.tcl hello.wpj
```

这种机制可用于工程配置而命令行编译。另外进行版本控制时，可只对工程文件进行，而不用管理生成文件。

 WindRiver, “TechTips-How to re-generate prjComps.h, projParams.h...”。

● 库连接顺序问题。

有两个库，liba.a 和 libb.a。liba.a 中有 a 函数，libb.a 中有 b 函数。而 liba.a 会使用 b 函数，libb.a 会使用 a 函数。在创建可下载目标模块时，出现如下问题：若按 liba.a、libb.a 顺序连入库，下载时出现未解析符号 a 错误；若按 libb.a、liba.a 顺序连入库，下载时则 b 错误。

解答：从根本上来说，这两个库设计欠妥。库最好能自包含，向外界提供服务。即使要依赖别的库，也是单向的，不应出现交叉引用，这样将最基本的库最后引用，就不会出现上面的问题。出现该问题的另一个原因是，目标模块只进行部分连接，不要求解析所有的符号，连接中即使出现未解析符号，也不报错，认为会在动态加载时解析，加载时若 VxWorks 中没有该符号，则出现未解析符号错误。

这个问题有很多解决方法：

- ✧ 可使用 GNU 工具 ar，将 liba.a 和 libb.a 合成一个大库；
- ✧ 将 liba.a 和 libb.a 连接入 VxWorks 中；
- ✧ 包含库时，重复写入库名，如 “liba.a libb.a liba.a”，可重复多次，使符号能完全解析；
- ✧ 使用 “--start-group liba.a libb.a --end-group”，让 GNU 连接器重复解析符号。

● 如何生成 srec 格式的映象文件

解答：主要是利用 GNU 工具 objcopy 进行格式转换。可利用工程编译配置中的宏定义 POST_BUILD_RULE，填入 “objcopy -O srec vxWorks vxWorks.srec” 类似值，使转换操作在编译后自动完成。

● 编译时 cpp.exe 出错

```
C:\Tornado\host\x86-win32\lib\gcc-lib\powerpc-wrs-vxworks\cygnus-2.7.2-960126\cpp.exe  
[switches] input output  
make: *** [AnchorData.o] Error 0x1
```


解答：这个问题和环境变量“TEMP”的设置有关，要设置为不带空格的路径名，因为 Tornado 不能正确读取带空格的路径，如 “C:\Program Files”，注意 Tornado 的安装目录，即 WIND_BASE。

● 编译出现错误 “undefined reference to ‘_eabi’”

解答：在应用代码中不要使用名称为 “main” 的函数。

● 连接时，如何将程序的 text 和 data 段放到指定位置

解答：可以使用“-Ttext”和“-Tdata”标志来指定段位置。如果只指定了“-Ttext”，data 段会紧接着 text 段存放，但一般会在两者之间插入 4KB 的空白页，以保证两段的分界，因为一个页不能同时具备两段的属性。

 WindRiver, “GNU Toolkit User’s Guide”。

- 在 WindSh 或调试器中创建任务时，如何指定栈大小，而不用缺省值

解答：WindSh 中，有 3 种方法可以创建具有指定大小栈的任务。

✧ 在 WindSh 中使用 sp 命令创建任务时，任务栈大小由 host/resource/tcl/shelltaskcmd.tcl 中的 spTaskStackSize 变量来确定，缺省值为 20000。可以在 WindSh 中修改该变量。

```
-> ?
tcl> set spTaskStackSize 70000
70000
tcl> ?
->
```

✧ 在 shelltaskcmd.tcl 中添加自己的任务创建命令，接收栈大小为参数，如下所示：


```
shellproc sp_new {arg_spTaskStackSize args} {
    global spTaskStackSize
    set spTaskStackSize $arg_spTaskStackSize
    #后面部分复制 sp 的实现
}
```

重启 WindSh 后，就可使用新加的 sp_new 命令，可以更方便地指定栈大小。

✧ 在 WindSh 中直接调用 VxWorks 库函数 taskSpawn，可定制任务的各种参数，包括栈大小。

✧ 通过设置扩展的调试器变量 wtx-task-stack-size，来指定调试器 CrossWind 中运行 [Run]任务的栈大小，变量缺省值也为 20000，如下所示：

```
(gdb) show wtx-task-stack-size
Stack size (in bytes) of tasks created using "run" command is 20000.
(gdb) set wtx-task-stack-size 15000
(gdb) show wtx-task-stack-size
Stack size (in bytes) of tasks created using "run" command is 15000.
```

 WindRiver, “Tornado 2.0 User’s Guide” 的 8.5.8。

第3章 VxWorks

3.1 初识系统

VxWorks 是个广泛使用的成功的工业 RTOS，被应用于各种嵌入式领域。包括过程控制（化工、食品加工），机器人（制造、自动操作控制），航空电子（飞行管理、GPS、喷气发动机控制），军事（武器管理、密码），数据处理（采集、信号处理），通信（税收机、数字电话），家电（微波炉、洗衣机、空调），计算机外围设备（打印机、Modem）等。

VxWorks 是嵌入式系统应用中的软件平台，是包括各种硬件驱动和内核组件的集合，提供了各种适应嵌入式系统和实时应用的特性，以及一些帮助系统开发和调试的辅助机制。

VxWorks 提供了定制硬件的板级支持包[BSP]，实现硬件抽象。硬件抽象层可以提高应用代码的移植性，使应用在各种目标硬板上的移植变得容易进行。VxWorks 还提供各种流行硬件设备的驱动，如网卡、串口和存储器驱动等，使得开发者可以快速建立应用系统。

VxWorks 本身和其应用程序都具有极好的可移植性。VxWorks 库组件几乎都用 C 语言编写，可容易移植到不同 CPU 上实现。VxWorks 的发行版本直接对多种 CPU 提供支持。最新的 VxWorks 5.5 有 8 类 CPU 版本，包括 PowerPC、Pentium、MIPS、SH、Xscale、ARM、68K 和 ColdFire 等。虽然众多版本对特定 CPU 有优化和定制，但是 Wind River 公司的主要目的可能是出于商业和安全的考虑。而 Wind River 公司内部开发肯定在不遗余力地维持 VxWorks 内核代码的可移植性。虽然有各种不同 CPU 版本的 VxWorks，但提供的函数接口是一致的，加上 BSP 硬件抽象层的支持，而且 VxWorks 支持 POSIX、ANSI 和 BSD Socket 等通用标准，使得建立在 VxWorks 之上的应用代码也有较好的可移植性。

VxWorks 提供高效的内核机制。系统调用采用普通的函数调用形式，而未采用操作系统通用的软件陷入机制，减少了系统调用花销，使得任务上下文切换速度加快，并使切换延迟确定。

VxWorks 可根据应用需求灵活配置，这主要是因为 VxWorks 软件平台具备组件式结构。VxWorks 包括核心内核功能和依赖内核的外围功能，各种功能按组件形式提供。VxWorks 可看作各种功能库的集合。各种功能库的选择加入，就能产生适应定制需求的各种不同功能集合的操作系统映象。

VxWorks 内核具有开放特性。内核函数的 Hook 机制，可以帮助用户定制内核。与此同时 VxWorks 还支持超过 320 家的合作伙伴公司的第三方产品。

VxWorks 适用于紧急任务应用，具有极高的可靠性。VxWorks 经过近 10 年的开发和应用，各内核组件都经过实践检验，已经相当稳定和成熟，包括军事、航天等在内的各种重要行业都有其应用。

3.1.1 VxWorks 特性

VxWorks 的主要特性如下。

✧ 可裁减的微内核体系结构。

- ✧ 有效任务管理和真实时支持，256个多任务优先级，抢占和轮循调度，时间确定的快速上下文切换。
- ✧ 灵活高效的任务间通信机制。
- ✧ 毫秒级中断处理响应。
- ✧ 支持POSIX 1003.1b实时扩展标准。
- ✧ 支持多种通信介质的标准TCP/IP网络，提供各种成熟的标准网络协议。
- ✧ 灵活的启动方式：ROM、本地盘、网络等。
- ✧ 多处理器支持。
- ✧ 灵活、快速的I/O系统。
- ✧ 支持dosFs和Flash文件系统。
- ✧ 1100多个兼容标准C的辅助函数。
- ✧ C++支持。

3.1.2 VxWorks 组件介绍

VxWorks 由很多可裁减的组件组成，包括 Wind[内核]、IO 系统、文件系统、网络系统、C++支持、共享内存对象[VxMP]和虚拟内存接口[VxVMI]等。

VxWorks 的组件结构是其突出的特点。VxWorks 库由 400 多个相对独立的、短小的目标模块组成，用户可根据需要选择适当模块来裁剪和配置系统，编译时链接器只链接配置的必要模块，保证生成映象的最小适用。这种组件结构和 pSOS、Nucleus 相比，是种折中方案。pSOS 也是组件配置的，但是功能划分比较粗略，组件模块的尺寸较大，一个小功能的需求就可能引入一个大的组件模块，不能量身裁减程序映象。Nucleus 最为灵活，一般应用的定制可由编译器链接时完成，最小裁减单位为源文件的目标模块，这和 VxWorks 一样，而它提供的源代码，可给用户更为随意的定制空间。

● Wind内核

Wind 内核是整个平台的核心，是实现实时多任务操作系统基本功能的微内核，其他实现外围功能的大量软件组件依赖于它。Wind 内核只关心 CPU，而对外设不做假设，不关心总线类型、内存大小和 IO 设备等。板级支持包[BSP]会驱动基本硬件，客户定制硬件可由应用程序驱动。Wind 内核实现功能包括任务调度、任务通信、内存管理、定时器和中断处理等，还提供符合实时系统标准 P.1003.1b 的 POSIX 接口，以提高应用程序代码的移植性。

● I/O系统

提供类似 Unix 的直接 I/O，与标准 C 库接口的带缓存 I/O，多种格式化 I/O，POSIX 接口的异步 I/O，对设备和文件进行操作。

● 文件系统


VxWorks 提供多种格式的文件系统，包括 dosFs、rt11Fs、rawFs、tapeFs、cdromFs 和 TSFS 等，适用于不同应用环境。通常使用 dosFs 作为目标机的文件系统。TSFS 是主机的文件系统在目标机上的映射，串口调试时常使用。

● 网络系统

使用 MUX 统一驱动接口，保证上层网络协议的可移植性。提供多种网络协议，包括 ICMP、IGMP、IP、UDP、TCP、OSPF 和 RIP 等；以及多种网络服务，包括 FTP，Telnet，HTTP 和 DNS 等；Socket 接口兼容 BSD。强大的网络功能是 VxWorks 的另一大特色。


● C++支持

VxWorks 提供异常处理、运行时类型信息[RTTI]、标准模板库[STL]和 Wind 基类等 C++ 特性支持。

 WindRiver，“VxWorks 5.4 Programmer's Guide”的第 5 章。

● 共享内存对象

VxMP 用于多 CPU 任务之间的高速同步和通信，为紧耦合多处理器支持附件。而 VxFusion 为松耦合配置和 IP 网络通信，扩展 VxWorks 消息队列为分布式消息队列。

 WindRiver，“VxWorks 5.4 Programmer's Guide”的第 6 章。

● 虚拟内存接口

VxVMI 提供第 2 级虚拟内存支持，包括代码段保护、异常向量表和 MMU 接口等。

 WindRiver，“VxWorks 5.4 Programmer's Guide”的第 7 章。

本书只关注 VxWorks 平台的最基本组件，对 Wind 内核、I/O 系统、文件系统和网络通信等进行详细描述，而略过 VxMP、VxVMI 等组件。还有其他一些小组件未在此列出，如加载组件、目标机 Shell 和标准 C 库等，后面会有相应章节介绍。

3.1.3 主机上 VxWorks 相关目录和文件

VxWorks 文件都存放在“WIND_BASE\target\”目录下，各子目录的分类描述如表 3-1 所示。

表 3-1 主机上 VxWorks 相关目录和文件

WIND_BASE\target\		
Config\		包含用来配置、建立 VxWorks 系统的文件
	Comps\	VxWorks 组件配置（Tornado 环境下的工程配置使用）
	All\	通用启动、配置
	BspName\	定制硬板的启动、配置
H\		VxWorks 库头文件
	Make\	为各 CPU 和工具集描述 Makefile 规则
Lib\	Objcputoolvx	VxWorks 目标模块文件和库文件 (cputool 为 CPU 名加编译工具名，如 libI80486gnuvx)
Proj\		VxWorks 工程目录

Src\	Config\	VxWorks 组件配置（命令行编译时使用）
	Drv\	设备驱动代码
	Demo\	演示应用
	Usr\	包含用户可修改代码
Unsupported\		非正式源代码



WindRiver, “Tornado 2.0 User's Guide” 的 A.3 章节。

下面对表 3-1 所示的相关目录做简单介绍。

✧ config\comps\src\

该目录下的文件用于 Tornado 工程管理组件配置，配置由 prjConfig.c 选择包含使用。各源代码文件与“config\comps\vxworks\”中的组件描述文件有对应关系。

✧ config\comps\vxworks\

该目录下包括 CDF 组件描述文件，由 Tornado 配置工具使用，参见本书 10.1.3 节的介绍。



配置工具会在 4 个目录下搜索 CDF 文件：本目录、arch/<arch>、config/<bsp>和工程目录。根据各 CDF 文件描述，工程管理窗口显示 VxWorks 组件列表。当安装新的组件后，新的 CDF 文件会添加在该目录下，而不会修改原来的 CDF 文件。例如，安装 dosFs 2.0 后，会出现新文件 10dosfs2.cdf。

✧ config\all\

bootInit.c 完成从 ROM 引导时的系统初始化。



需要注意的是，随生成最终程序映象的形式不同，bootInit.c 会修改文件名，再编译进入映象，所以有可能找不到 bootInit.o 这个目标模块。例如生成非压缩格式的 BootRom 时，bootInit.c 会更名为 bootInit_uncmp.c，定制 BSP 目录下会生成 bootInit_uncmp.o 文件。另外，在进行 BootRom 跟踪调试源代码时，调试器会找不到 bootInit_uncmp.c 这个文件，需要根据 bootInit.c 复制一个 bootInit_uncmp.c 存放在该目录下，才能继续调试，添加这个文件不会影响程序的编译。

configAll.h 适用于所用 BSP 的组件参数配置文件。



一般不要修改该文件，特殊的 BSP 定制应该修改 BSP 目录下的 config.h，这样各定制 BSP 之间才没有交叉影响。上层应用的进一步定制由工程管理工具进行，所做的修改体现在 prjParams.h 中。这 3 个组件参数配置文件的调用顺序为：configAll.h、config.h 和 prjParams.h，后面文件的定义会屏蔽前面文件的定义。当工程初始创建时，prjParams.h 中的配置和 configAll.h、config.h 两者的组合配置是一致的，这就是说，在定制 BSP 时，应考虑上层应用对系统组件的需求，将这些需求反映在 config.h 中，这样以后在该 BSP 创建上层应用时，不用每次都重复配置系统组件的过程，保证各应用间组件配置的一致性，减少工作量。

prjComps.h 为组件配置，包含在/comps/src/configAll.h 中。

bootConfig.c 为 VxWorks BootRom 的启动源代码文件。

usrConfig.c 为 VxWorks 应用程序的启动源代码文件。



bootConfig.c、usrConfig.c 和 prjConfig.c 的区别经常让人迷惑。从一般意义上说，它们是对等体。bootConfig.c 是 BootRom 的配置代码文件，而后两者都是 VxWorks 的配置代码文件。usrConfig.c 有点过时，用于在 BSP 目录下手动编译生成 VxWorks，也就是可以手动修改 BSP

的 Makefile 文件，在命令行下编译生成程序映像，只参考 configAll.h 和 config.h 的配置，引用“target\src\config”下的组件，包括代码文件。prjConfig.c 由工程管理工具配置生成，Makefile 等其他文件也自动生成，不能手动修改，在集成环境下编译生成程序映像，需要参考 3 个配置参数头文件，引用“target\config\comps\”下的组件包括代码文件。

✧ config\bspname\

- romInit.s: 汇编语言写的初始化代码，是 BootRom 和 ROM 引导 VxWorks 的启动入口。
- sysALib.s: 系统相关代码，用汇编语言编写。
- syslib.c: 系统相关代码，用 C 语言编写。
- config.h: 硬件参数和组件参数配置文件。
- bspname.h: 定制目标硬板的参数配置文件，会包括 CPU 和其他外围硬件的描述头文件。

✧ src\config\

该目录下存放组件包括代码文件，与“target\config\comps\”目录功能类似，只是这些代码文件由 usrConfig.c 根据配置引用，已经过时，一般不再使用。

3.1.4 VxWorks 库模块


VxWorks 子程序都组织成库（包含代码和数据的目标模块，许多库是可选的，有没有它们都可以编译建立 VxWorks。每个库都有一个或多个头文件，如表 3-2 所示。

表 3-2 库模块示例

库模块	函数	头文件
taskLib	taskSpawn	taskLib.h
semLib	semTake	semLib.h

“VxWorks Reference Manual”按字母顺序列出每个库模块的描述和函数帮助，是实际编程中程序员最常参考的手册。

VxWorks 的库和函数都使用一致的标准命名规则。比如处理 foo 的库，会称为 fooLib，库的头文件则是 fooLib.h，而这个模块中所有可见的全局变量和函数名称前缀都是 foo，如 fooParamSet 函数。这样可以很容易知道库模块实现了什么样的功能，函数该归属于哪个库，源文件中应该包含什么头文件等，这些对理解和编写源代码都很有用。但是 VxWorks 还支持一些已存在的标准，如 POSIX、BSD Socket 等，这些标准对函数、宏、头文件等的命名有自己的规范，而不能套用 VxWorks 的命名规则。

 Wind River 系统编码规则的细节信息可参考“Tornado 2.0 User's Guide”的 G 章节。

3.1.5 VxWorks 与其他 RTOS 的比较

VxWorks 与其他 RTOS 的比较如表 3-3 所示。

表 3-3 RTOS 比较

	VxWorks	pSOS	Ecos
任务	Y	Y	只有线程
调度	抢占，轮循	抢占	抢占
同步机制	无条件变量	Y	Y
POSIX 支持	Y	Y	Linux

可裁减	Y	Y	Y
定制硬件支持	BSP	BSP	HAL, IO 包
内核大小	—	16KB	—
多处理器支持	VxMP/VxFusion	pSOS+m 内核	N

VxWorks 不像 Unix，两者有根本的区别。VxWorks 没有独立程序，独立地址空间，独立全局变量等概念，只有惟一的地址空间。当加载模块时，模块的全局变量会添加到该地址空间。但有些方面类似 Unix，如 POSIX 编程接口、网络编程接口等。

3.2 VxWorks 与目标机

VxWorks 提供定制硬件的板级支持包[BSP]，实现硬件抽象层。BSP 由代码、文档和规范组成，用做 VxWorks 程序和特定硬件之间的接口。理想的 BSP 实现是上层软件只看得见 BSP 接口，而看不见实际的硬件，通过 BSP 就可以完全使用和运行硬板。BSP 代码用于完成系统初始化，设备驱动和其他一些硬件相关的操作。

BSP 的代码位于 “/target/config/” 目录下硬板相关子目录。这些代码是针对硬件厂商的 Demo 板定制的，在对应 demo 板上，这些代码可以直接使用，而对于用户自己的硬板，可能需要参考近似 Demo 板的代码进行修改。BSP 代码只适用于特定的硬件。

两个目标机（CPU）相关的库 sysALib 和 sysLib，对应的实现代码文件为 BSP 目录下的 sysALib.s 和 sysLib.c。这两个库是硬件抽象实现可移植性的关键。它们为不同硬板的类似功能提供相同的软件接口。主要工作包括：硬件初始化，中断处理和产生，硬件时钟和定时器管理，本地和总线内存空间映射，确定内存大小等。

BSP 中还包含一些硬件设备的驱，如 SysSerial.c 完成硬板上的串口的初始化。

BootRom 和 BSP 的关系

读者经常会混淆这两个概念之间的关系。BSP 是软件抽象层概念，而 BootRom 是一个实在的程序映象类型，是建立在 BSP 之上的简单的 VxWorks 映象。而 VxWorks 应用映象也要使用 BSP 代码，对于可直接启动的 VxWorks 来说，BootRom 就没有存在的必要。BootRom 的主要作用将硬板的最小系统启动起来，以完成加载和执行 VxWorks 应用映象的目的。BootRom 完成硬件初始化后，跳转到 VxWorks 应用入口时，硬件又重新被初始化。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 sysLib 条目。

3.3 VxWorks 与主机

目标机的 Agent 经过通信通道，使用 WDB 协议，与主机 Target Server 相连，用以建立交叉开发环境。主机工具与 Target Server 的连接见前面章节的描述。所有主机工具与目标机的联系都通过 Agent。

主机和目标机的 WDB 连接如图 3-1 所示。目标机 Agent 以 tWdbTask 任务形式运行，通过通信驱动和通道与主机的 Target Server 相连，接受主机的调试请求（包括内存事务、断点和其他事件服务、虚 I/O 支持和任务控制等），控制目标机中其他的运行任务完成调试并返回结果。图中只示意了网络连接，其实有多种通信方式供选择。

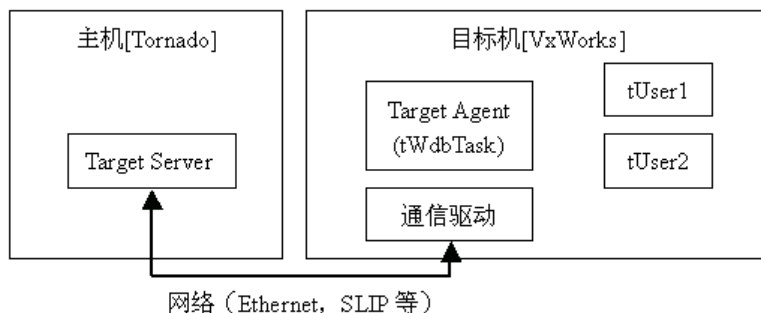


图 3-1 主机和目标机 WDB 连接

Agent 的一个重要功能是响应主机目标模块加载器的请求。为了让 Tornado 具备增量加载的能力，目标机程序映象一般包括 Agent 组件。启动时，Target Server 从主机上的程序映象初始化符号表，这些符号在增量加载时可用，极大减少了下载的数据量和时间。

Agent 不依赖运行时操作系统，不直接和运行时服务接口，这样 Agent 可利用已提供的内核特性。Agent 的通信驱动也独立于 VxWorks 的 I/O 系统。这种驱动为低级[raw]驱动，可以按查询或中断方式工作。运行时独立性意味着 Agent 可在内核运行之前执行，这个特性在 VxWorks 移植向新硬件平台的早期特别有用。

WDB 内存池由 Target Server 使用，以支持主机工具。其大小由 WDB_POOL_SIZE 决定，缺省大小为系统内存池大小的 1/16。在 Tornado 工程配置窗口不能修改该参数。如果在系统运行中，目标服务器需要更多的内存，可以从系统堆中分配。内存池在内存中紧接在 VxWorks 映象后存放。

目标机 Agent 对应“WDB agent components”组件，在主机工程配置窗口有显示。缺省创建的工程中，该组件夹下多个组件被选中。开发者可以自己选择包括哪些特性。在最终程序发布时，也可以包括这些组件，以允许现场调试，它们对内存的需求量很少。

其中“WDB agent services”组件夹下最主要组件为“WDB agent”（选中时定义 INCLUDE_WDB 宏），该组件的初始化函数为 wdbConfig（由 INIT_RTN 指定），在“/target/config/comps/usrWdbcore.c”中定义（由 CONFIGLETTES 指定）。WdbConfig 函数由 prjConfig.c 中的 usrWdbInit 函数调用，完成 Agent 通信接口的初始化，并启动 Agent 任务。该组件夹下还有许多其他的相关组件。其他组件的参数设置和用途，可以参看各组件的属性，获得相关信息，也可以查看 prjConfig.c 的 usrWdbInit 函数体中对各组件的注释描述。

“select WDB connection”组件夹用于选择 WDB 的通信方式，包括 END、netROM、网络、串口、tyCoDrv 和自定义等。只能单选其中之一，缺省配置为网络连接。Target Server 的后端[Back End]的设置应该和目标机 Agent 的通信方式一致，网络连接时后端设为 wdbrpc。各通信方式的选择和建立参见第 9 章的介绍。

“select WDB mode”组件夹用于选择 WDB 的工作模式。必须从系统模式和任务模式中选择至少一种模式。Agent 支持两种目标机控制模式：任务模式和系统模式。Agent 可以在两模式任意之一下执行，并可根据要求在两模式之间切换。这样极大地简化了嵌入式应用中各种代码（任务、中断）的调试工作。两种模式对通信驱动有要求，任务模式可使用中断驱动，系统模式必须用查询驱动（因为 VxWorks 系统停止时，中断被闭锁，只能通过查询方式与主机通信），有些通信驱动可能不同时支持这两种模式。

在任务模式下，Agent 作为 VxWorks 任务运行。调试在任务级执行。可以单独调试某些任务，而不影响系统中其他运行的任务和中断。

在系统模式下，Agent 在 VxWorks 外运行，类似 ROM 监视器。允许用户将 VxWorks 应用程序当单线程来调试。当进入断点时，整个系统运行停止，中断闭锁。这种模式最大的优点就是能单步调试中断处理函数，但难以操作单独的任务。这种模式对系统干扰较大，断点时中断闭锁会导致中断不能及时响应。


在同时使用任务模式和系统模式的情况下，系统中同时包含两个 Agent：任务模式 Agent 和系统模式 Agent。同一时刻只有一个模式被激活，可以通过调试器和 Shell 来切换模式。


 wShell 提供（只在 wShell 可用）agentModeShow 命令用以显示 Agent 的当前模式。

```
-> agentModeShow
Agent is running in task mode
value = 0 = 0x0
```

目标机的 Agent 上下文管理库（wdbLib）提供控制权转移函数。将控制权从运行系统转到外部模式运行的 Agent（外部模式即系统模式）。细节可参考 VxWorks 库手册。

WDB Agent 的相关内容在第 2 章和第 9 章中还有介绍。

 WindRiver, “Tornado 2.0 User's Guide” 的 1.5 章节，并查看索引 “target agent”。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的索引 “target agent” 下相关条目。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 “wdbLib” 条目。

3.4 符号表

符号表用于建立符号名称、类型和值之间的关系。其中，名称为 null 结尾的任意字符串；类型为标识各种符号的整数；值为一个字符指针。符号表主要用来作为目标模块加载的基础，但在需要名称和值关联的任何时候都可使用。

动态支持符号表是 VxWorks 系统的突出特点。这极大地增加了系统的灵活性，对调试、动态加载等有很大的帮助作用。

运行系统中一般存在两个符号表结构 sysSymTbl 和 statSymTbl（使用 lkup 可看到）。这两个标识为头指针，在内存 bss 段中定义，新加表项需要的存储空间从系统内存池中分配。sysSymTbl 为目标机的系统符号表，通过程序或 tShell 动态加载的目标模块的符号都添加到该符号表中，sysSymTbl 和 statSymTbl 两个标识本身也包含在该系统符号表中。符号表 statSymTbl 中包含 errno 的信息，若想在 tShell 中使用 printErrno 命令，就必须包含该符号表（对应 “error status table” 组件，可参考本书的 3.4.4 小节）。

主机的 Target Server 只使用一个符号表，可通过全局变量 tgtSymTbl 访问。

SysSymTbl 和 statSymTbl 两个符号表指针的结构在 symLib.h 中有定义。

```
typedef struct symtab          /* SYMTAB - symbol table */
{
    OBJ_CORE  objCore;         /* object maintenance */
    HASH_ID   nameHashId;      /* hash table for names */
    SEMAPHORE symMutex;        /* symbol table mutual exclusion sem */
    PART_ID   symPartId;       /* memory partition id for symbols */
    BOOL      sameNameOk;      /* symbol table name clash policy */
    int       nsymbols;        /* current number of symbols in table */
}
```

```
    } SYMTAB;  
typedef SYMTAB *SYMTAB_ID;
```

所有对符号表的操作都用互斥信号量 `symMutex` 加以保护。`nameHashId` 用于名称 hash 处理，加快按名访问的速度，hash 表的大小在符号表创建时指定，必须为 2 的幂。`symPartId` 表示新加入符号的存储内存，一般都使用系统内存 `memSysPartId`（在 `memLib.h` 中声明）。`sameNameOk` 用于选择是否允许向符号表中加入同名符号。

这两个符号表指针在 `sysSymTbl.h` 中声明。

```
extern SYMTAB_ID sysSymTbl; /* system symbol table */  
extern SYMBOL standTbl[]; /* standalone symbol table array */  
extern ULONG standTblSize; /* symbols in standalone table */  
extern SYMTAB_ID statSymTbl; /* system error code symbol table */  
extern SYMBOL statTbl[]; /* status string symbol table array */  
extern ULONG statTblSize; /* status strings in status table */
```

`sysSymTbl` 中实际的符号项存放在 `standTbl[]` 数组中，`standTblSize` 表示其中符号的个数。`StatSymTbl` 的情况类似。

符号表项 `SYMBOL` 在 `symbol.h` 中定义。

```
typedef struct /* SYMBOL - entry in symbol table */  
{  
    SL_NODE nameHNode; /* hash node (must come first) */  
    char *name; /* pointer to symbol name */  
    char *value; /* symbol value */  
    UINT16 group; /* symbol group */  
    SYM_TYPE type; /* symbol type */  
} SYMBOL;
```

该结构中包含符号的名称、值、目标模块组和类型等。系统相关的符号类型在该头文件中有定义列表。`group` 值同模块 ID，`VxWorks` 模块的 `group` 值为 1。

除了系统维护的符号表外，用户可以创建自己的符号表，并可以动态填充。这时需要使用 `symLib` 库提供的函数接口。符号类型 `SYM_TYPE` 其实为单字节整数，用户可以使用自己定义的类型，而不一定使用系统定义的符号类型。当然应用中的符号表，不一定用于程序符号解释，可以用于任何建立名称和值对应关系的地方，比如存放学生的成绩表。`symLib` 会自动有效管理该表，如加入 hash 算法以加快符号查找，互斥信号量保护访问等。

`VxWorks` 的符号表与执行文件中的符号表不同。执行文件的符号表，是文件格式要求的（如 `elf`），由编译器直接生成，可以帮助完成源代码级调试。而 `VxWorks` 符号表是一种独特的机制，只是系统运行时维护的一个表结构。

3.4.1 symLib 和 symlib

`VxWorks` 中符号表函数库 `symLib`，提供用于符号表操作的相关函数，如表 3-4 所示。

表 3-4 symLib 库函数

	symLib 库函数
<code>symLibInit</code>	初始化 <code>symLib</code> 库
<code>symTblCreate</code>	创建符号表
<code>symTblDelete</code>	删除符号表
<code>symAdd</code>	向符号表中添加符号
<code>symRemove</code>	从符号表中删除符号

symFindByName	通过名称查找符号（hash 查找）
symFindByNameAndType	通过名称、类型查找符号
symFindByValue	通过值查找符号（线性查找，值最接近）
symFindByValueAndType	通过值、类型查找符号
symEach	调用自定义函数遍历符号表

上面介绍了，用户自定义符号表可以使用任意符号类型。如果用 `symLib` 库函数来操作 `VxWorks` 系统符号表（`sysSymTbl`），符号类型值在 `symbol.h` 中定义，包括 `SYM_GLOBAL`、`SYM_ABS`、`SYM_TEXT`、`SYM_DATA` 和 `SYM_BSS` 等，各表示意思很明显，类型可能会是组合表示，如“`SYM_TEXT|SYM_GLOBAL`”。这些类型是 `VxWorks` 自己的，和具体的目标模块格式（`AOUT` 或 `ELF`）没有直接的关系。


```
sysSymTbl = symTblCreate (SYM_TBL_HASH_SIZE_LOG2, TRUE, memSysPartId);
```

系统符号表允许同名冲突（`sameNameOk`），同名符号会加入符号表而不报错。当引用符号时，系统缺省使用 `symFindByName` 函数查找，所以最新加入的符号总会先找到，代替老符号完成引用。同名符号的存在还带来另一个问题，不能确保查找到的符号是自己加载模块的，即不能保证 `group` 号。`symLib` 库缺少 `symFindByNameAndGroup` 这样的函数，用户要完成自己的条件查找符号，需要用 `symEach` 函数遍历实现。当然符号同名冲突也可以通过开发人员良好的编程风格来避免。

在“`Tornado API Reference`”中还存在一个近似库 `symlib` 的描述，注意名称中“`l`”字母的大小写。在“`/host/include/`”目录中还存在与“`/target/h/`”目录中对等的头文件，如 `symlib.h` 和 `symbol.h` 等。还有 `loadLib` 库也存在这种情况。这很让人困惑。其实这样做的主要目的，是支持目标模块在主机的动态加载。该主机库的函数在各目标模块加载动态库中定义，动态库位于“`\host\x86-win32\lib\loader`”目录下，按目标格式区分，如 `loadaout.dll` 和 `loadelf.dll` 等。主机 `symlib` 库中函数用于维护主机符号表 `tgtSymTbl`。

`VxWorks` 中还定义了符号表类 `VXWSymTab`，封装了 `symLib` 中的函数。该类属于 `WFC` 基类，需要额外购买 `WFC` 组件才可使用。

 WindRiver，“`VxWorks 5.4 Reference Manual`”的“`symLib`”条目。

 WindRiver，“`Tornado API Reference`”中“`Target Server Internal Routines`”的“`symlib`”。

3.4.2 符号表初始化

`VxWorks` 的符号表与编译器的符号表不同。`VxWorks` 的符号表是动态存在系统中的，由 `symLib` 提供的函数进行维护，在运行过程中可以增加和删除符号，主要用于目标模块的动态加载，与目标模块的格式无关。而编译器中的符号表是静态的，由编译器和连接器在建立程序映象时生成并使用，主要用于符号的静态解析，帮助建立映象。它们的格式和目标类型有关。交叉调试器中源代码调试，也是由静态符号表支持的。这种符号表占用很大的程序映象空间，当程序发布时，可以去掉（不用“`-g`”选项）。

`VxWorks` 系统符号表的建立，需要主机工具在映象生成时完成一些准备工作。前面已经描述，`VxWorks` 中一般有两张符号表 `sysSymTbl` 和 `statSymTbl`。它们对应两个主机工具 `makeSymTbl.exe` 和 `makeStatTbl.exe`，用于生成符号数组以编译进入程序映象。

工具 `makeSymTbl.exe` 用于创建包含 `SYMBOL` 类型数组的 `symTbl.c` 文件，包含目标模块中所有全局符号的名称、地址和类型。该全局数组名称为 `standTbl`，其大小为 `standTblSize`，

存放在程序映象的 `data` 段中。浏览 `symTbl.c` 源代码有助于对符号表的理解。

在程序映象编译连接过程中，“build out”窗口会出现如下类似调用。

```
E:\Tornado\host\x86-win32\bin\makeSymTbl cf tmp.o > symTbl.c
```

工具 `makeStatTbl.exe` 用于创建状态代码 `SYMBOL` 数组，包含一些头文件中定义的所有状态代码的名称和值。所有状态代码以“`S_`”开头，一般库头文件中都有这样的定义。生成过程中还引用 `vwModNum.h`，以解释各库的编号。在后面 `errnolib` 章节会进一步解释状态代码的组成。该全局数组名称为 `statTbl`，其大小为 `statTblSize`。`statSymTbl` 主要由 `printErrno` 函数使用，也可以由应用程序使用，输出确定意义的状态信息。`statSymTbl` 数组存放在程序映象的 `data` 段中。

当系统启动时，由 `usrStandaloneInit` 函数完成初始符号表的填充，该函数在 `usrStandalone.c` 中定义，主要代码如下。

```
sysSymTbl = symTblCreate (SYM_TBL_HASH_SIZE_LOG2, TRUE, memSysPartId);
for (ix = 0; ix < standTblSize; ix++) symTblAdd (sysSymTbl, &(standTbl[ix]));
```

这里可清楚地看到系统符号表 `sysSymTbl` 允许同名符号 (`TRUE`)，新加的符号在系统内存中分配空间存储 (`memSysPartId`)。

前面讲的两个工具主要用于建立 `Standalone` 系统。如果不将符号信息编译入映象，可以通过主机下载符号信息。这是两种符号表初始化方式 `build-in` 和 `downloaded`。使用内建符号表时，符号是 `VxWorks` 程序映象的一部分。下载符号表则和 `VxWorks` 映象分离，由目标机单独下载 `Sym` 文件获得符号。

如果使用下载符号表，会使用另一个主机工具 `xsym`，生成符号模块文件 `VxWorks.sym`。该文件和普通目标模块一样，只是没有 `data` 和 `bss` 段。下载到目标机后，和普通目标模块一样动态加载，模块中的符号会添加到系统符号表。当然 `sysSymTbl` 先在 `usrLoadSyms` 中初始化，再下载符号文件。该函数在 `usrLoadSym.c` 中定义。

```
sysSymTbl = symTblCreate (SYM_TBL_HASH_SIZE_LOG2, TRUE, memSysPartId);
netLoadSymTbl ();
```

`VxWorks.sym` 也可以作为独立映象的程序模块存放在目标机的盘上，从当地盘上加载。这也算 `downloaded` 模式。不过这种方式用得很少，因为这种方式主要用于调试，直接通过网络下载更方便。

目标机符号表是完成动态加载、连接，以及调试的辅助机制，映象有没有符号表都不影响程序正常运行。调试时，只要不从目标机上动态加载目标模块，程序映象符号都静态连接解析，目标机上也可以没有符号表。主机开发工具都使用主机符号表 `tgtSymTbl` 来完成交叉调试。

 WindRiver, “Tornado API Referencel” 中 “Tornado Utilities”。

3.4.3 符号表同步

主机和目标机使用不同的符号表。在主机上使用的各种工具并不能访问目标机上所有的内容。工具都是通过主机上 `Target Server` 和目标机上 `Target Agent` 进行通信的。主机和目标机都不能直接访问对方的符号信息。

主机和目标机都维护着自己的符号表。当添加符号时，只是将符号加入到其中一个表中。比如，在 `wShell` 上添加的符号到了主机符号表，而 `tShell` 上添加的符号就到了目标机符号表中。缺省时这两个表互不共享。这意味着在目标机上加载的模块，主机不知道其中的符号。

当然，也可以添加符号表同步组件[symbol synchronization]（对应库为 symSyncLib），这样符号会在主机和目标机之间复制，两符号表会同步更新。用户可以在 Target Server 的输出中看到这种过程的记录。

程序代码中对符号的引用会使用目标机的上符号表，而不管程序代码是从主机还是从目标机上加载的。

主机和目标机使用不同的符号表，系统启动初始时两个符号表的内容是一样的，当动态加载目标模块后，两个符号表就不一致了。可能出现这样的情况，从 wShell 中加载目标模块后，却想从 tShell 引用该模块符号；或者从 tShell 加载后，想用主机工具调试该模块。因为 wShell 或 tShell 都看不见对方加载的模块，所以可能出现如下的错误。

```
Error: module contains undefined symbol
Unresolved symbol
Fatal Error: unresolvable symbol
```

要解决这样的问题，就需要使用 Tornado/VxWorks 提供的符号表同步机制。VxWorks 提供了 symSyncLib 库来完成符号表同步。系统运行中，从主机或目标机添加的符号都可以被对方看见。symSyncLib 会在目标机创建同步任务 tSymSync。该任务被当作 WTX 工具与主机的 Target Server 相连。任务启动时，会立即同步符号表，避免了原来存在的一个问题：主机看不见 Target Server 启动前目标机加载的模块。这样，目标机启动脚本自动加载模块的符号，主机也可见。

不过这种机制对硬件有要求。因为同步任务使用 WTX 协议与 tgtsvr 通信，所以目标机必须能支持网络。两者之间有较大的数据流量，若有大模块加载，建议使用 wdbrpc。

符号表同步对目标机性能有影响。如果用户觉得不再需要符号表同步，可以手动将 tSymSync 悬挂起来，以提高目标机性能。

要使用符号表同步机制需要配置 Tornado 和 VxWorks。在 VxWorks 程序映象中加入符号表同步组件（对应宏定义为 INCLUDE_SYM_TBL_SYNC），如图 3-2 所示。

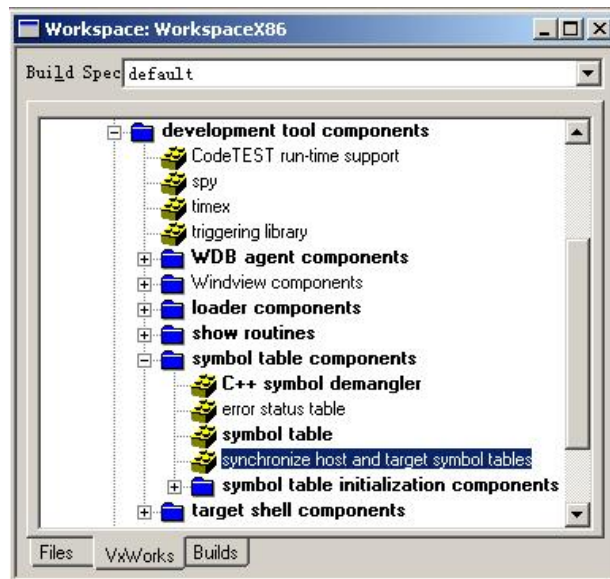


图 3-2 符号表同步组件

主机工具都通过 Target Server 访问符号表，因此还需配置主机的 Target Server。在

Tornado 中选择菜单“Tools→Target Server→Configure”命令来激活配置窗口，在窗口选中符号表同步项，如图 3-3 所示。配置完成后，tgtsvr.exe 添加了“-s”启动项。需要重新启动 Target Server，配置才生效。

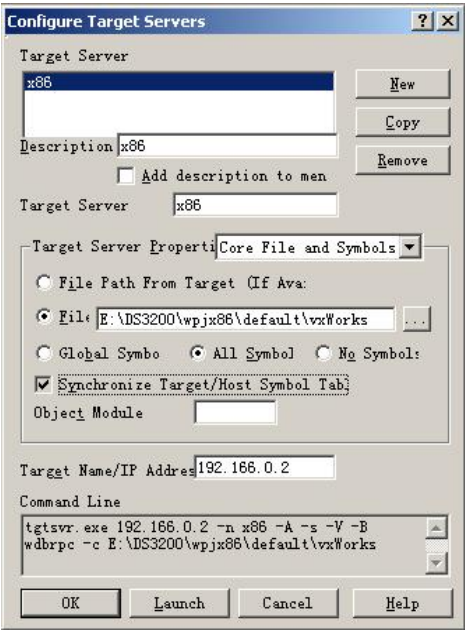


图 3-3 符号表同步 Target Server 配置

在系统运行和交互过程中，可以通过 Tornado Registry 的消息窗口看到符号表同步过程。

Added isis00000.o to target-server....done

WindRiver, “VxWorks Reference Manual” 的 symSyncLib 条目描述。

3.4.4 错误状态

若 VxWorks 库函数执行出现问题，函数会返回 ERROR 值，并设置错误状态表示具体的原因和位置。

● 错误状态库[errnoLib]

errnoLib 用于获取和设置任务和中断的错误状态值[errno]。当 VxWorks 库函数出错时，一般返回 ERROR 或 NULL，具体错误信息则用 errno 表示。这和 Unix 标准的错误表示机制类似。不同的是，Unix 用一个全局量来存储 errno，而 VxWorks 每个任务和中断都有自己的 errno。任务的 errno 存放在 TCB 中，为任务私有。中断的 errno 存放在中断栈中，只要在中断处理函数中 errno 都有效。

errnoLib 提供函数用于获取和设置 errno，如表 3-5 所示。

表 3-5 errnoLib 库函数

errnoGet	从调用任务取得 errno
errnoOfTaskGet	从指定任务取得 errno
errnoSet	设置调用任务的 errno
errnoOfTaskSet	设置指定任务的 errno

● 错误状态值[errno]

VxWorks 的 errno 值由 4Byte 构成。高字表明发生错误的库，各库对应的值在“target/h/vwModNum.h”中定义。低字表示该库发生的具体错误，在相应库的头文件中定义。

Unix 标准错误代码只使用低字，高字值为 0，Unix 的 errno.h 头文件在 VxWorks 中没有修改。另外，约定 VxWorks 系统模块使用高字值，值范围为 1~500，其他值可以由应用模块使用，对低字值没有规定。

● 获取 errno

当错误发生时，通常在 Shell 或程序中能获得 errno 的值。

例如，命令 i 显示任务列表中，ERRNO 项就表示任务最近调用库发生的错误的 errno 值。

-> i								
NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	_excTask	4ca9170	0	PEND	4276be	4ca908c	d0003	0

程序中可以用 errnoGet 函数或 errnoOfTaskGet 函数来取得任务的 errno 值，如下所示。

```
if (bootLoadModule (fd, pEntry) != OK) {
printErr ("nerror loading file: status = 0x%x.\n", errnoGet ());
printErrno(errnoGet ());
}
```

● 解释 errno

得到 errno 值，需要用户自己来确定具体哪个库发生了什么错误。usrLib 提供 printErrno 函数来显示具体错误信息。printErrno 根据输入的 errno 值，在 statSymTbl 中查找到对应的错误信息，并显示在标准输出上。

statSymTbl 为特殊的符号表，专门用于支持 printErrno。由主机工具 makeStatTbl 扫描头文件辅助生成（有自己的搜索规则）。用户也可以添加自己定义的 errno 到该表中。

可以在 Shell 或程序中调用 printErrno。例如，在 Shell 中查找“d0003”错误信息。

```
-> printErrno 0xd0003
0xd0003 = S_iosLib_INVALID_FILE_DESCRIPTOR
value = 0 = 0x0
-> printErrno 0xd0004
0xd0004 = S_iosLib_TOO_MANY_OPEN_FILES
value = 0 = 0x0
```

注意，“d0003”为 16 进制值，命令中应加“0x”前缀。信息串的格式和 errno 值的结构对应，“S_”前缀表示状态[State]，iosLib 为发生错误的库名称，INVALID_FILE_DESCRIPTOR 表示该库具体的错误。第 2 个命令显示相同库的其他错误，只有表示具体错误的信息串尾部变化。iosLib.h 中 errno 值宏定义名和错误信息串相同。

若想在 tShell 或程序中调用 printErrno 来显示错误信息，程序映象中需包含错误状态表组件（如图 3-4 所示），以在目标机建立 statSymTbl。而 printErrno 来自 usrLib，缺省被包含。

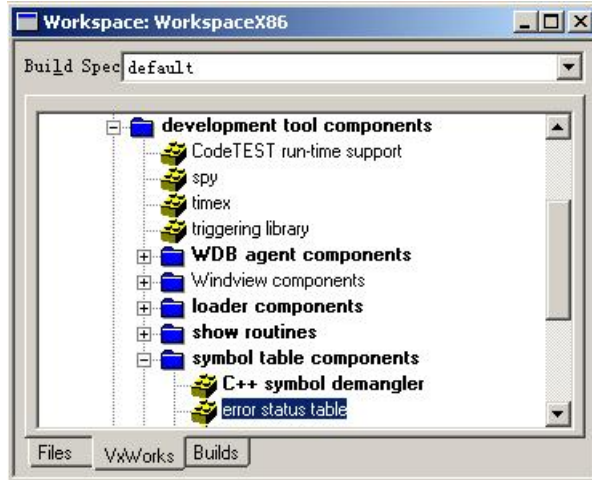


图 3-4 错误状态符号表组件

如果 Shell 中不能用 `printErrno`，也可以手动查找头文件得到错误信息。比如“d0003”，先在 `vwModNum.h` 查找“d”对应的库，“d”应换算为十进制数“13”，查到 `M_iosLib`：

```
#define M_iosLib (13 << 16)
```

再到 `iosLib` 库自己的头文件 `iosLib.h` 中查找具体错误，其中“0003”对应的宏定义为：

```
#define S_iosLib_INVALID_FILE_DESCRIPTOR (M_iosLib | 3)
```

这个宏定义名即为用户要知道的错误信息。

3.5 Linker 和 Loader

在刚开始学习 VxWorks 的时候，最让笔者迷惑和感兴趣的的就是单独目标模块的加载运行。当时手里只有 Tornado 的原型教学版本，不能进行实际目标硬件板的交叉调试开发，只能使用主机环境下的 VxSim 进行程序仿真。记得 Tornado 入门手册的第一个例子 `cobble`，相信很多人都运行过。好像是先创建一个 Downloadable 工程，再将 `cobble.c` 加入工程，编译得到 `cobble.o`，将其下载到 VxSim 中运行，连续输出 HOT、OK…。当时觉得如此不可思议，没有连接的程序代码怎么能够执行。基于原来的开发经验，程序都是经过编译连接后生成程序映象，可能再将程序映象进行格式转换，写入 Flash 或磁盘中，程序直接运行或由引导程序加载、跳转执行。虽然 pSOS 系统中也有动态加载，但是远没有 VxWorks 中使用的这般广泛和灵活。随着使用 VxWorks 进行系统开发，这个困惑都一直存在。而后接触到目标机 Shell 和 WindSh 交叉使用，还有符号表，这些概念交织在一起，使笔者更难理解这背后所发生的一切。但是，隐隐觉得这里面肯定存在某种灵活的机制，只是暂时没有理解。而这个机制却是笔者在使用 VxWorks 过程中最想弄明白的一件事。这一章节希望能解开你心中的困惑。

在现代的操作系统中，用户应用程序可以共享代码，并可以动态扩展新功能。比如 Windows 下的动态库机制。许多操作系统还允许内核功能扩展。这些都需要 Linker 和 Loader 来完成。运行时 Linker 和 Loader 是一个操作系统的基本部分，用来加载和运行应用程序。它们负责将程序从辅助存储器中加载到系统的主内存空间。并需要对应用程序做一些调整，以适应程序在内存中的新位置，也使得应用程序能对外部代码进行引用。Linker 和 Loader 功能上有些重叠，但概念上却各自独立执行操作。

无论是静态连接还是动态加载，都要进行重定位操作。依据段起始地址调整程序地址的

过程，解析对外部符号引用的过程，统称为重定位。重定位有两种类型：连接时重定位和加载时重定位。前者比后者更为复杂。在加载、重定位时，整个程序被看作一个大段，Loader 根据名义[nominal]加载地址和实际加载地址的不同，只需要调整程序地址即可。而在连接时，根据段的地址和大小，不同的地址有不同的偏移量。

笔者实在不想将 Linker 和 Loader 翻译为中文，怕读者初次看见时会把他们当成某种小型家电，用英文显得比较和谐和对称。

Linker 按常理是应该和编译器一起介绍的，但是在 VxWorks 中，它和操作系统的 Loader 结合如此紧密，实在难以分开来说，放在这里比较合适。当然这里讲的主要指运行 Linker。

Linker 和 Loader 都支持标准的目标模块格式，如 AOUT、ELF 等。VxWorks 中特定的体系结构不完全支持所有的格式，只支持指定格式。

本节主要对目标模块、Linker、Loader、符号表这些概念之间的内部联系机制做描述，有的地方和调试器也有关联。

3.5.1 静态连接

先看看静态连接的过程，以便更好地理解 VxWorks 动态加载连接。

当 Linker 运行时，需先扫描输入文件，确定段[segment]的大小，收集所有符号的定义和引用。创建段表和符号表，段表列出输入文件定义的所有段，符号表包括所有输入输出符号。Linker 再利用第一次扫描所得数据，在输出地址空间为符号指定数字地址，确定段的大小和位置，并计算出该放在输出文件的什么位置。

第二次扫描中，Linker 利用前面的信息来控制真正的连接过程。读取并重定位代码，替换符号引用的数字地址，按重定位的段地址来调整代码和数据的内存地址。

看下面的汇编代码，这是一个简单的重定位和符号解析的例子。使用“eax”寄存器，将变量“a”内容传到变量“b”中。假设“a”的地址为 0x1234，“b”为引入变量。

```
mov a, %eax ; A1 34 12 00 00
mov %eax, b ; A3 00 00 00 00
```

右边为目标代码，命令 1 个字节，地址 4 个字节。因为“b”的地址未知，所以就填为 0。假设经过重定位，在输出文件中，模块位于“0x1000”，“b”位于“0x9A12”，代码就变为：

```
A1 34 12 01 00
A3 12 9A 00 00
```

可以看到，因为重定位和地址解析，两个地址都被更新了。

这个例子取自于 John R. Levine 写的《Linkers and Loaders》，是一本非常好的书籍，有很多其他复杂的例子，以及连接和加载过程的细节信息。

Linker 会读取输入模块中所有的符号表，建立连结时[link time]符号表，在连结过程中使用。有 3 种不同的连结时符号表：一种列出输入文件、库模块，并保存每个文件的信息；第二种有全局符号，也就是 Linker 需要在输入文件间解析的符号；第三种符号表处理模块内部的调试符号。

如前面所述，连接分为两步完成。第一步，Linker 从每个输入文件读进符号表，并维护一个独立的全局量表，包括所有输入文件中定义或引用的每个符号。每次 Linker 读输入文件，都将该文件所有的全局符号添加到符号表中，并保留符号定义或引用的位置信息。第一步完成后，每个全局符号都有了精确的定义。第二步，Linker 边创建输出文件，边解析符号。解

析和重定位互相影响，因为在大多目标模块格式中，各重定位项确定程序对符号的引用。

对含有构造函数和析构函数的程序，多数 Linker 创建指针表，指针指向来自各输入文件的函数。Linker 创建类似“__CTOR_LIST__”的符号，作为语言启动辅助[stub]，用来查找指针表和调用所有的函数。

3.5.2 动态加载

当加载目标模块时，一般有如下几个主要步骤。

- ✧ 程序加载：目标模块被复制到主内存。
- ✧ 重定位：目标模块创建时，起始地址为 0。然而所有目标模块都加载到自己的空间，地址不能出现重叠。重定位就是为模块的各部分指定加载地址，按指定地址调整模块的代码和数据的过程。
- ✧ 符号解析：当存在多个模块时，需通过符号完成模块之间的相互引用。模块中的弱符号（未初始化全局量）和未定义符号，就试图用前面加载模块的符号来完成解析。
- ✧ 符号添加：将自己的输出符号添加到符号表中，用于后续加载模块引用，或用于启动模块程序等。

系统需要为模块加载分配内存，一般由系统自动从系统内存中申请，用户可以得到模块的加载地址。用户也可以自己申请内存，然后指定模块加载到该地址，这样对调试有些帮助。不过要小心处理，如仔细考虑分配内存大小等。

动态加载必须依赖符号表。如果加载的模块带有未解析符号，就需要查询符号表完成地址的自动解析。模块的输出符号，应加入默认符号表，用来被以后引用。不过新符号也可对加入哪个符号表做出选择。

模块向符号表中新添加的符号，对已有的程序代码是不认识的，也不能直接加以引用，必须显式使用名称，通过查找符号表取得地址，再通过地址引用。当然这种引用缺乏灵活性，程序中很少使用，所以动态加载的模块符号引用一般是前向的。

从上面的步骤可以看出，动态加载和静态连接过程有类似之处，如重定位，符号解析等。重定位都更新了模块内各符号地址，完成模块内自己符号的引用。对于外部符号引用解析，都依赖过程中积累建立的符号表。

VxWorks 的目标模块加载只是动态加载时的静态连接，与通常操作系统中的动态库或共享库概念不同，与 C++ 的动态绑定也有区别。

一旦应用模块加载到目标机内存，就可以使用模块输出的函数接口。如从 shell 中直接调用，创建为任务，连接到中断等。

C++ 模块加载时，应注意名称修饰问题。有时会出现函数名虽然可以 lkup，但是不能 sp，在 Shell 中可以使用 <Ctrl+D> 来补全修饰的 C++ 名称或未完全的名称，如下所示：

```
->sp ace_main(Ctrl+D)
```

可以设置 sysCplusplusEnable 为 1，来自动反修饰名称，在 Shell 和程序中都可使用。

在模块程序中还可以使用“extern “C””来修饰输出函数名，按 C 规则输出。

3.5.3 loadLib 与 loadlib

目标机的 loadLib 库提供加载目标模块的函数，如表 3-6 所示。模块可以从多种 I/O 流加载，包括 netDrv、NFS 和本地文件系统等，但不支持 Socket。

表 3-6 loadLib 库函数

LoadModule	加载目标模块
loadModuleAt	加载目标模块，可以指定地址和返回地址

库函数缺省使用系统符号表 `sysSymTbl`。加载时外部引用符号从该符号表中获取解析信息。如果不能完成解析，会输出错误信息，加载继续，符号添加到 `sysSymTbl` 中，而返回的 `MODULE_ID` 为 `NULL`。

模块中定义的新符号可选择加入系统符号表中，依赖加载函数中 `symFlag` 参数值，可选项如表 3-7 所示。

表 3-7 symFlag 选项

<code>LOAD_NO_SYMBOLS</code>	不添加符号
<code>LOAD_LOCAL_SYMBOLS</code>	只添加局部符号
<code>LOAD_GLOBAL_SYMBOLS</code>	值添加全局符号
<code>LOAD_ALL_SYMBOLS</code>	全添加局部、全局符号
<code>HIDDEN_MODULE</code>	隐藏模块， <code>moduleShow</code> 不能显示

除了模块中符号外，还添加一些辅助符号，用于定位各段的起始位置。形如 `xxx_text`、`xxx_data` 和 `xxx_bbs` 等，其中 `xxx` 为模块文件名。

`LoadModule` 函数等同于用 `text`、`data`、`bss` 为 `NULL` 调用 `loadModuleAt` 函数，其实 `loadModule` 函数最终用 `loadModuleAt` 函数实现。`LoadModuleAt` 函数比 `loadModule` 函数有更大的灵活性，可以指定和返回目标模块各段的地址，返回的段地址可帮助调试。

将模块加入系统内存，由系统自动分配加载地址，但不返回加载地址：

```
module_id = loadModule(fd, LOAD_GLOBAL_SYMBOLS);
module_id = loadModuleAt (fd, LOAD_GLOBAL_SYMBOLS, NULL, NULL, NULL);
```


同上，但返回模块各段的加载地址：

```
pText = pData = pBss = LD_NO_ADDRESS;
module_id = loadModuleAt (fd, LOAD_GLOBAL_SYMBOLS, &pText, &pData, &pBss);
```

指定 `text` 段的加载地址，返回 `data` 和 `bss` 段的自动地址：

```
pText = 0x800000; /* address of text segment */
pData = pBss = LD_NO_ADDRESS /* other segments follow by default */
module_id = loadModuleAt (fd, LOAD_GLOBAL_SYMBOLS, &pText, &pData, &pBss);
```

使用 `loadModuleAt` 函数，可以将接口类似模块加载到同一地址，模块都依赖先前加载的符号表引用，在运行时能进行动态切换，实现不同功能。

 WindRiver，“VxWorks 5.4 Reference Manual”的“loadLib”条目。


如同 `symLib` 和 `symlib` 一样，`loadLib` 在主机也有个对等库 `loadlib`。主机的 `loadlib` 使用 `Target Server` 符号表 `tgtSymTbl`。库函数由动态连接库实现，如 `loadlib.dll`、`loadaout.dll` 和 `aout68k.dll` 等。

除以上两个函数外，主机库提供其他的更多的函数。一般在编程中不会用主机库函数。

主机 Loader 按 3 层结构组织：通用层、目标模块格式层和目标机相关层。由主机目录下各 DLL 库的名称就可以看出这个结构。

从主机上加载的所有模块都包括在 WDB pool 内存中，这是从目标机内存中分配的，紧接在 VxWorks 映象之后。而模块调试信息保留在主机上，不占用内存。当加载新模块时，若 WDB pool 中没有足够的空闲空间，会从系统内存中分配额外的内存并入 WDB pool 中，完成模块的加载。

从 wShell 调用 loadModule 时，实际调用的是 loadModuleAt 函数，text、data 和 bss 的地址都为 NULL，这样目标模块可能加载到 WDB 内存池中的任何位置。LoadModuleAt 再调用 asyncLoadModuleAt 函数。asyncLoadModuleAt 用 ioRead 将目标模块读入并存储在主机内存中，并用 sllPutAtTail 将加载请求添加到 Loader 线程的队列中。

 WindRiver, “Tornado API Reference” 中 “Target Server Internal Routines” 的 “loadlib”。

3.5.4 目标模块管理

VxWorks 中的 moduleLib 库用于加载模块的信息管理。信息包括系统中已加载的模块，各模块的段信息，符号和模块的对应关系等。利用这些信息可以列出当前加载的模块，可以卸载不需要的模块等。模块对象包括如下信息：名称、段链表、组号和 symFlag 等。同名的多个模块可以同时加载，查找函数会找到最近加载的模块。

当模块被加载后，moduleCreate 给它分配惟一的模块 ID 和 group 号。模块 ID 和 group 号都可以用来引用该模块。模块加载函数和加载命令 ld 被调用后会返回模块 ID 值。通过 moduleShow 或主机 Browser 也可以查到模块 ID。group 号用于符号项中，用来识别符号表中本模块的符号。因为 group 号为 2 字节整数，而模块 ID 为 4 字节的地址，由于符号表大小的限制，group 号比模块 ID 更适合用作符号的模块索引。相同 group 号的符号都来自同一加载模块。当模块卸载时，用 group 号查找属于该模块的符号，并从符号表中删除这些符号。

一般情况下，开发者不会直接使用该库中的函数，除了显示当前加载模块信息的 moduleShow 函数。大部分函数都由 loadLib 和 unloadLib 来调用。如模块加载时调用 moduleCreate 函数，模块卸载时调用 moduleDelete 函数。

moduleShow 函数在主机有对等的实现。wShell 命令显示主机加载模块的情况，而 tShell 命令显示目标机中加载情况，wShell 中的显示如下所示：

```
-> moduleShow
MODULE NAME          MODULE ID  GROUP #    TEXT START DATA START  BSS START
-----
VxWorks              0x38abe8    1      0xb41b0    0x1000    0xe9210
value = 0 = 0x0
-> @moduleShow
MODULE NAME          MODULE ID  GROUP #    TEXT START DATA START  BSS START
-----
value = 0 = 0x0
```

目标机的 moduleShow 命令不能看见 VxWorks 模块，因为 VxWorks 启动时 moduleLib 还未工作。可以看见 VxWorks 的组号为 1，以及各段的起始地址。

再看一个区别。启动主机 Browser 中的 Spy Chart，用于查看各任务对 CPU 的使用情况。主机会自动加载 spyLib.o 模块到目标机，以支持主机工具完成工作。

```
-> moduleShow
```

MODULE NAME	MODULE ID	GROUP #	TEXT START	DATA START	BSS START
VxWorks	0x38abe8	1	0xb41b0	0x1000	0xe9210
spyLib.o	0x38ae50	2	0x16a7e0	0x16b1a4	0x16b1bc
value = 0 = 0x0					
-> @moduleShow					
MODULE NAME	MODULE ID	GROUP #	TEXT START	DATA START	BSS START
value = 0 = 0x0					


这里可以清楚地看到目标机的模块管理库看不到主机动态加载的模块。

另外主机工具 Browser 也可以查看模块的信息，其底层实现和 moduleShow 一样。Browser 的具体使用可参考上一章。

上面的显示有个错误，text 段和 data 段的信息颠倒了。这好像是 VxWorks 的 Bug，在主机工具 Browser 中也有颠倒问题。

每个加载模块都有惟一的组号（同符号项中的 group），用来识别符号表中本模块的符号。模块加载时由 moduleCreate 分配。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 moduleLib 条目。

 WindRiver, “VxWorks 5.4 User’s Guide” 的 8.4.5 节

3. 5. 5 加载方式

利用 Tornado 和 VxWorks 的加载库，可以通过多种途径完成目标模块的动态加载。

● 主机加载

主机动态加载模块的主要目的是进行应用程序的交互开发和调试。使用模块下载，用户可以专注新代码的调试。因为连接是动态完成的，只需要编译正在调试的部分代码，而不用重新建立整个程序映象。这种方式能进行应用代码的增量调试，减少应用代码的编译时间，减少映象下载量，避免不同模块之间的调试干扰。

一般创建 Downloadable 工程来调试代码模块。工程中使用 ldarch 编译命令，将多个源代码文件生成可重定位目标文件，用以动态加载。Bootable 工程则不同，生成的程序映象是绝对定址的，不能用于加载。当然也可在 Bootable 工程中加载单个目标模块，但没有意义，不如 Downloadable 工程来得直观方便。可以创建多个 Downloadable 工程来开发各自独立的代码模块。

从主机的多个界面可以完成模块加载操作，包括目标模块上右键菜单项、wShell 和 GDB 命令行窗口等。虽然形式各异，其底层实现机制都是 loadlib。主机通过动态连接库实现的 loadlib，提供模块加载辅助函数。

其中最简单直接的方法是右键下载。直接在目标模块名上单击鼠标右键，选择菜单“Download 文件名”命令，如图 3-5 所示。

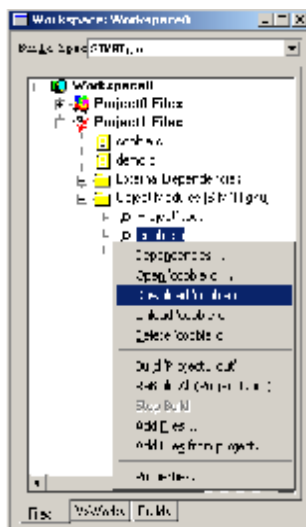


图 3-5 主机加载目标模块

wShell 中使用 ld 命令完成模块下载。命令 ld 从标准输入或文件加载模块。主机 ld 是用 Tcl 脚本实现的。使用<CTRL+C>可终止模块的加载过程。命令格式和加载信息如下。

```
ld [syms[,noAbort][, "name"]] (syms: -1 = none, 0 = globals, 1 = all)
```

其中 syms 参数用于确定模块符号添加到符号表中的原则。noAbort 没用，兼容 tShell 中 ld。

```
-> ld<e:\demo.o
Loading e:\demo.o |
value = 19221880 = 0x1254d78
-> moduleShow
```

MODULE NAME	MODULE ID	GROUP #	TEXT START	DATA START	BSS START
VxWorks.exe	0x387dc0	1	0x401000	0x43f000	0x442000
cobble.o	0x11d4460	2	0x4ddeb4c	0x4ddefec	
demo.o	0x1254d78	3	0x4ddeac4		

上面使用的是简洁的输入重定向命令格式，等同于如下形式：

```
-> ld 1,0," e:\\demo.o"
-> ld(1)<e:/demo.o
```

即添加所有符号，主要用于调试。注意文件路径的区别。

在 GDB 命令行窗口，可以使用 load 命令加载模块。不过要显示 GDB 窗口，必须启动调试器。命令 load 属于 files 命令组，可以用“help files”和“help load”来得到帮助。相关显示如下：

```
(gdb) help load
Dynamically load FILE into the running program, and record its symbols
for access from GDB.
(gdb) load e:\demo.o
Downloading e:\demo.o ...done.
Reading symbols from e:\demo.o...expanding to full symbols...done.
(gdb) help unload
Unload and close an object module that is currently being debugged.
(gdb) unload demo.o
```

```
Unloaded file e:\demo.o
```


加载模块文件的目录有多种表示方式。VxWorks 中使用 “/” 来分隔目录，“/” 在 C 串中没有特殊的意义。加载主机文件时，也可以使用 “\” 来分隔目录。但 “\” 在 C 串表示 escape 字符，当用串表示文件路径时，必须使用 “\\”。而由 Shell 直接解释的路径名没有特殊的语法要求。看看下面举例便能更好地明白上面的描述。下面的各命令都是等同的。

```
-> ld<e:\demo.o
-> ld<e:/demo.o
-> ld 1,0," e:\\demo.o"
-> ld 1,0," e:/demo.o"
```


Shell 有缺省的工作目录，也可以用相对路径来引用模块文件。可以用 pwd 查看当前工作目录。可以用 cd 命令改变工作目录。相关显示信息如下：

```
-> pwd
E:/Tornado/target/src/demo/1
value = 0 = 0x0
-> cd "e:/"
value = 0 = 0x0
-> ld<demo.o
Loading e://demo.o |
value = 19042328 = 0x1229018
-> ld 1,0,"demo.o"
Loading e://demo.o |
value = 19042328 = 0x1229018
```

主机相关命令的具体描述，请参考上一章的相关内容。

 WindRiver, “Tornado 2.0 User's Guide” 的 6.3.8 章节，并查看索引 “ld” 下相关条目。

 WindRiver, “Tornado 2.0 Tools Routines” 的 “ld” 条目。

 WindRiver, “VxWorks 5.4 User's Guide” 的 8.4.4 节。

● tShell 加载

与 wShell 类似，tShell 中也使用 ld 命令加载模块，用法和上一节讲述的几乎一样。

目标机加载模块，需要符号表 sysSymTbl 支持。需包括 “symbol table” 组件，并用下载符号文件或映象内符号完成初始化。若没有符号表支持，模块加载时不能解析其外部引用，其输出符号也不能被其他程序引用。

利用 tShell 的 C 解释特性，可以通过脚本加载模块。在系统启动时，可以让 Target Shell 读取脚本文件，由脚本控制多个应用模块的动态加载。至于脚本文件的编写和使用在后面的 “目标机 Shell” 章节细讲。由于外部符号解析的前向依赖，需注意模块加载顺序。前面介绍说主机 ld 命令的 noAbort 参数没有实际用途，而在目标机中脚本加载时有用。脚本加载模块出现错误时，若 noAbort 为 0，会调用 shellScriptAbort() 终止脚本执行；若 noAbort 为 1，脚本继续执行。

目标机模块加载与主机目的不同，主要为了提高软件配置的灵活性。用户不必将所有模块和内核静态连接在一起，而可以在运行时用 ld 函数加载、连接这些模块。在现场升级程序时，只需简单更换目标模块文件就行了。这样也能提高软件功能配置的灵活性，功能增加或删除都容易完成，只需要更换模块和改编脚本就行了。一旦用户习惯了这种方式，就会觉得非常有用。

目标机加载与主机加载有些区别，在前面的“loadLib 与 loadlib”小节有所描述。底层实现机制不同，目标机使用 loadLib，而主机使用 loadlib；实现使用的符号表不同，目标机使用 sysSymTbl，而主机使用 tgtSymTbl；加载的内存不同，目标机使用系统内存 memSysPartId，而主机使用 WDB_POOL；目的也不相同，目标机为了最终软件的灵活性，而主机为了进行代码增量调试；加载方式不同，目标机可以独立完成加载，可以使用自动脚本，而主机必须依赖 Target Server。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 usrLib 条目。

● 程序加载

在程序中可以使用 loadModule 和 loadModuleAt 函数加载模块。这两个函数由 loadLib 实现，在前面已有详细描述。

由于 Shell 的 C 解释特性，也可以在 Shell 中直接使用 loadModule 和 loadModuleAt。但是比较麻烦，因为需要自己获取文件描述符 fd，参数 symFlag 的值也与 ld 命令的定义不同，需要自己查找头文件定义。wShell 和 tShell 中都提供简洁形式的 ld 命令，直接调用函数加载很少使用。

模块加载到目标机内存后，就可以使用模块输出的函数接口。但是模块不能自启动，需要外部机制的辅助，如 Shell 或其他程序调用。实际应用中，一般用模块的某函数接口创建任务，再由任务执行线程调用模块的其他函数，使用加载模块。

当然也可以使用特殊技巧让模块能自启动。模块加载时，Loader 会检查加载模块的 C++ 静态对象，并自动初始化静态对象。C++ 对象的构造函数中可以进行模块入口的启动。但这样做会增加开发过程的复杂性，因为生成模块时需要使用 munch 工具。

下面两段代码用于模块启动，也许对读者实际应用有帮助。

```
/*用 loadModule() 来加载模块，然后用 moduleFindByName() 来找到符号表中固定入口函数地址，并启动*/
#include <VxWorks.h>
#include <stdio.h>
#include <ioLib.h>
#include <loadLib.h>
#include <symLib.h>
#include <taskLib.h>
#include <nfsLib.h>
#include <nfsDrv.h>

extern SYMTAB_ID sysSymTbl ;

int loadTestModuleAndRun()
{
    int fd = ERROR ; int status = ERROR ;
    MODULE_ID hModule ; FUNCPTR taskEntry = NULL ; SYM_TYPE Type ;

    fd = open("/sd0/test.out", O_RDONLY, 0) ;
    if (fd==ERROR) {
        printf("can not open binary file.\n") ; return ERROR ;
    } else
        printf("binary file opened.\n") ;
```



```

if ((hModule=loadModule(fd, LOAD_ALL_SYMBOLS))==NULL) {
printf("loadModule error = 0x%x.\n",errno) ; return ERROR;
}
close(fd) ;

status = symFindByName(sysSymTbl,"test", (char **)&taskEntry, &Type ) ;
if (status==ERROR) {
printf("symFindByName error=%d\n", errno) ; return ERROR;
} else
/* 符号类型定义 N_ABS=2,N_TEXT=4,N_DATA=6,N_BSS=8;N_EXT=1 */
printf("taskEntryr=0x%x, type=%d\n.", (int)taskEntry, (int)Type);

status = taskSpawn("test", 100, 0, 30000, taskEntry, 0, 0, 0, 0, 0, 0, 0, 0, 0) ;
if (status==ERROR) {
printf("taskSpawn error=%d\n",errno) ; return ERROR;
}
return OK ;
}
/*指定函数名和参数创建任务，不加载模块，未考虑 group 号， 有重复函数名时有问题*/
#include <VxWorks.h>
#include <symLib.h>
#include <sysSymTbl.h>
#include <time.h>
#include <errno.h>

#define STACK_SIZE 500000

extern char *strdup (char *);

int system (char *string)
{
char *args[10];  char *p;  int i;
char *fname;  char *pValue;  SYM_TYPE pType;

p = fname = strdup (string);
for (i=0; i<10; i++) {
while (*p && (*p != ' ' && *p != '\t')) p++;
while (*p && (*p == ' ' || *p == '\t')) *p++ = 0;

if (*p && *p != '&')    args[i] = p;
else                  args[i] = NULL;
}

if (symFindByName (sysSymTbl, fname, &pValue, &pType)) {
printf("vxwork_system() cannot find symbol %s\n", string);  return ERROR;
}

taskSpawn (fname, 150, 0, STACK_SIZE, pValue, args[0], args[1], args[2], args[3],
args[4],args[5], args[6], args[7], args[8], args[9]);

taskDelay (200);    /* 给新任务时间完成初始化*/
return OK;
}

```

用 `ld` 函数加载目标模块的顺序很重要。若一个模块在加载时有未解析符号，后面再加载的其他模块对解析这些符号没有帮助。加载模块只能和先于它加载的模块连接。例如，有两个模块 `app1.o` 和 `app2.o`，`app1.o` 可以独立运行，而 `app2.o` 对 `app1.o` 中定义的符号有依赖。这种情况下，`ld` 会执行必要的连接操作，但是 `app1.o` 必须在 `app2.o` 之前加载。

但是有时不能保证这种顺序，加载可能导致连接错误 “unresolved reference”。这种情况一般可通过主机工具 `ldarch` 解决，将这些紧张加载顺序的模块连接在一起。

另外还可以用 `jump` 表来解决。这种技巧在高级文件和低级文件之间添加抽象层，低级文件提供函数实现，高级文件通过全局函数指针调用。高级文件加载时不要求调用函数实际存在，只要全局的 `jump` 表结构存在就行了。这样高级文件可先于低级文件加载。但是要求，在使用 `jump` 表的模块加载前，需先声明 `jump` 表全局指针。

当高级模块需要根据环境调用不同的低级函数实现时，该技巧也有用。单个高级文件可以和多个低级文件配合使用。

使用这种技巧需要 3 个步骤：在内存中创建空全局 `jump` 表；填充 `jump` 表中的函数指针；调用 `jump` 表函数指针。代码如下所示：

```
/*=====
* jumpTbl.h - Jump table header file
* Declares the jump table structure and necessary functions.
*/

#ifndef __INjumpTblh
#define __INjumpTblh

typedef struct /* Jump table */
{
    FUNCPTR addRoutine;
    FUNCPTR multRoutine;
    FUNCPTR divRoutine;
} JUMP_TBL;

STATUS setupJumpTable(void);

#endif /* __INjumpTblh */

/*=====
* setupJumpTbl.c - 该文件建立全局访问 jump 表
* 这是惟一需先加载的文件。应该连接入 VxWorks 映象。保证后续加载模块可找到 jump 表
*/

#include "VxWorks.h"
#include "jumpTbl.h"
#include "memLib.h"
#include "stdlib.h"

JUMP_TBL *jumpTable;

/*****
* setupJumpTable() - Allocates memory for the jump table.
* 该函数只能运行一次，可以在初始化函数中调用
*****/
```

```

*/
STATUS setupJumpTable(void)
{
    static int alreadySetup = 0; /* Only setup once */

    if(alreadySetup == 0) {
        if((jumpTable = (JUMP_TBL *)malloc(sizeof(JUMP_TBL)))==NULL)
            return(ERROR);
        else {
            alreadySetup = 1;
            return(OK);
        }
    }
    return(OK);
}

/*=====
* initJumpTbl.c - A low level jump table initialization file.
* 该文件示例 jump 表初始化, 如何使用 jump 表.
*/

#include "VxWorks.h"
#include "jumpTbl.h"
#include "stdio.h"

IMPORT JUMP_TBL    *jumpTable; /* Globally accessible jump table */

/* Forward declarations */
LOCAL int    localAdditionRtn(int inputValue1, int inputValue2);
LOCAL int    localMultiplyRtn(int inputValue1, int inputValue2);
LOCAL uint_t localDivisionRtn(int inputValue1, int inputValue2);

/*****
* initJumpTable() - Fill the jump table with low level function pointers
*/
void initJumpTable(void)
{
    if(jumpTable == NULL) /* Check that table exists already. */
        setupJumpTable(); /* If table doesn't exist make one. */

    /* Fill in the function pointers */
    jumpTable->addRoutine = (FUNCPTR)localAdditionRtn;
    jumpTable->multRoutine = (FUNCPTR)localMultiplyRtn;
    jumpTable->divRoutine = (FUNCPTR)localDivisionRtn;
}

/*****
* localAdditionRtn() - Low level self explanatory static function.
*/
LOCAL int localAdditionRtn(int inputValue1, int inputValue2)
{
    return(inputValue1 + inputValue2);
}

```

```

/*****
* localMultiplyRtn() - Low level self explanatory static function.
*/
LOCAL int localMultiplyRtn(int inputValue1, int inputValue2)
{
    return(inputValue1 * inputValue2);
}

/*****
* localDivisionRtn() - Low level self explanatory static function.
*/
LOCAL uint_t localDivisionRtn(int inputValue1, int inputValue2)
{
    if(inputValue2 > 0)
        return(inputValue1/inputValue2);
    else{
        printf("You can\'t divide by ZERO\n");
        return(ERROR);
    }
}


/*=====
* callJumpTbl.c - A high level library using jump table function calls
*/
#include "VxWorks.h"
#include "stdio.h"
#include "jumpTbl.h"

IMPORT JUMP_TBL *jumpTable;

void callJumpTable(int inputArg1, int inputArg2)
{
    if(jumpTable != NULL) { /* Does jump table exist? */
        if(jumpTable->addRoutine != NULL) { /* Anything there? */
            printf("Calling jumpTable->addRoutine\n");
            printf("The returned value = %d\n", jumpTable->addRoutine(inputArg1, inputArg2));
        }
        else printf("jumpTable->addRoutine not initialized\n");
        if(jumpTable->multRoutine != NULL) { /* Anything there? */

            printf("Calling jumpTable->multRoutine\n");
            printf("The returned value = %d\n",
                jumpTable->multRoutine(inputArg1, inputArg2));
        }
        else printf("jumpTable->multRoutine not initialized\n");
        if(jumpTable->divRoutine != NULL) { /* Anything there? */
            printf("Calling jumpTable->divRoutine\n");
            printf("The returned value = %d\n", jumpTable->divRoutine(inputArg1, inputArg2));
        }
        else printf("jumpTable->divRoutine not initialized\n");
    }
    else printf("jumpTable struct does not exist. Run setupJumpTable()\n");
}

```

 WindRiver, “TechTips-How to use a jump table”。

另外还有一种更奇异的用法，模块自己解析外部引用，而不依赖 Loader 的解析过程。Loader 总是从系统符号表中解析符号地址，模块需要自己实现这部分功能。写程序时就知道要引用函数或变量的名称和类型，模块程序中使用 `symFindByName` 从系统符号表中查找引用符号地址，再加以使用。该模块对符号表中的符号的引用，对 Loader 来说是不可见的，根本不是显示的外部引用，而只是一些取符号地址传给指针的操作。这个用法笔者没实际试过，好像也没有什么实际用途，只是觉得有趣，让大家看看，用以加深对概念的理解。

3.5.6 模块卸载

当通过主机 Target Server 再次加载同一目标模块时，老版本会自动卸载。也可以手动卸载模块，如 `wShell` 的 `unld` 命令，模块名上的右键菜单，GDB 窗口的 `unload` 命令。

- ✧ 模块不能重复加载，`tShell` 中必须使用 `unld`，`wShell` 中自动调用 `unld`。
- ✧ 目标机重复加载，不会自动卸载老版本，需要显式卸载。
- ✧ 主机和目标机不能看见对方加载的模块，加载的模块只能自己卸载。

当模块卸载后，对该模块函数的调用会失败，结果不确定。用户负责保证卸载模块的函数或变量不再被其他模块使用。可以用 `ldarch` 将互相关联的文件连接为单个目标模块，这样它们可以同时加载和卸载，以避免出现这个问题。`unld` 命令会自动检查部分 `hook` 调用中是否使用卸载模块的函数。

卸载时，一般系统会释放所用的内存。如果用户通过 `loadModuleAt` 自己分配内存加载模块，卸载时则需要自己负责释放内存。符号表中属于该模块的符号也会被自动删除。模块管理库函数将模块描述符从模块列表中删除。

模块在卸载前，系统中所有的断点会被删除。如果需要保留断点，要设置 `UNLD_KEEP_BREAKPOINTS` 参数。被卸载模块代码中的断点不再存在。

Shell 中提供 `unld` 命令来从目标机内存卸载模块，命令格式如下：

```
STATUS unld(void * nameOrId); /*主机*/
STATUS unld(void * nameOrId, int options); /*目标机*/
```


其中 `nameOrId` 参数可以是模块文件的名称或模块 ID，参数 `options` 未使用。

在程序中使用 `unldLib` 库提供的函数卸载模块。`unldLib` 库函数如表 3-8 所示。

表 3-8 unldLib 库函数

<code>unld</code>	用文件名或模块 ID 卸载模块
<code>unldByModuleId</code>	用模块 ID 卸载模块
<code>unldByNameAndPath</code>	用文件名和路径卸载模块
<code>unldByGroup</code>	用 group 号卸载模块
<code>reld</code>	重加载模块

其中 `reld` 函数是一个组合调用，先卸载系统的加载模块，再调用 `ld` 函数加载模块的新版本。`reld` 函数使用的模块文件名称应该和原加载时使用的一样。

 WindRiver, “VxWorks 5.4 User’s Guide” 的 8.4.6 节

 WindRiver, “VxWorks 5.4 Reference Manual” 的 `unldLib` 条目。

 WindRiver, “Tornado 2.0 Tools Routines” 的 “`unld`” 条目。

3.6 目标机 Shell

VxWorks 可以为目标机提供一个 Shell，以方便应用系统的开发和使用，简称为 tShell，区别于主机提供的 WindSh（wShell）。目标机 tShell 也具有 C 解释特性。

3.6.1 tShell 创建

为了创建 tShell，需要在 VxWorks 配置中包含“Target Shell”（INCLUDE_SHELL）组件，对应库为 shellLib。若想使用功能齐全的 tShell，还需要包含其他组件，如前面描述的符号表和 Loader 相关的组件。为了使用 tShell 的调试功能，还需要包含“show”组件，以及“target debugging”组件。若想支持启动脚本功能，需要包含“shell startup script”组件。为了 tShell 能够远程登录和保证安全性，需要包含 RLOGIN 相关组件。


包含“target shell”组件后，prjConfig.c 中 usrRoot 函数会调用初始化函数：

```
shellInit (SHELL_STACK_SIZE, TRUE); /* target shell */
```

该函数会创建 tShell 任务以实现 Shell 功能。函数的第 1 个参数用于设置任务栈的大小，可在工程组件窗口设置。任务创建时使用了 VX_UNBREAKABLE 选项，所以在 tShell 任务中不能设置断点。tShell 中调用的函数都在 tShell 任务的上下文中运行，也不能设置断点。

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tShell	_shell	4b62320	1	READY	42cfb4	4b615cc	3d0005	0

VxWorks 系统只支持一个 tShell 实例。例如，Telnet 登录使用 tShell 后，串口上的 Shell 就不能使用；Telnet 退出后，串口 Shell 又可恢复使用。这主要是因为 Shell 的语法解析器是不可重入的，是用 yacc 实现的。


 WindRiver，“VxWorks 5.4 User’s Guide”的 9.2.1 节。

3.6.2 tShell 使用

目标机 Shell 的使用与 wShell 很类似，很多都可以参考前一章对 wShell 的描述。与 wShell 不同，tShell 有自己的一套终端控制字符，如表 3-9 所示。

表 3-9 tShell 终端控制字符

CTRL+H	删除一个字符（backspace）
CTRL+U	删除整行
CTRL+C	终止并重启（tShell 调用函数阻塞时可用）
CTRL+X	类似 reboot 命令，进入 BootRom，完成系统重新启动
CTRL+S	暂停输出
CTRL+Q	恢复输出
ESC	切换输入模式和编辑模式

 WindRiver，“VxWorks 5.4 User’s Guide”的 9.2.2 节。

● 终止和重启

有时候有可能需要终止和重启 tShell。例如调用函数陷入死循环，被阻塞等。这可能因为函数调用参数出错，或函数实现本来就有问题。这时可以使用<CTRL+C>终端控制字符，让 tShell 任务从原入口重新执行。tShell 重启后，标准输入输出又恢复原来的分配，所以先前

的重定向无效，正在执行的脚本也会被终止。要使用<CTRL+C>，需要包含“target debugging”组件。


有时在使用 tShell 时会出现致命错误，如总线/地址错误，违权指令等。一般任务在出现这种错误时会被挂起，以便于进一步调试。而这时 tShell 会被 VxWorks 自动重启，或者不能继续调试。因为这个原因，以及为了可以设置断点和单步调试，调试时最好用函数为入口创建任务，而不要直接从 tShell 中调用该函数。

tShell 被终止后，会自动显示 tShell 任务的栈轨迹，利用该信息可知道 tShell 任务在何处执行时出错。

需要注意，tShell 被终止重启后，导致错误的函数可能使系统的一部分资源处于无法清除的状态。例如，tShell 可能取得了一个信号量，终止操作不会自动释放该信号量。

远程登录或退出，也会引起 tShell 任务重启。如使用 Telnet 登录后，Console 输出信息：

```
telnetd: This system *IN USE* via telnet.  
shell restarted.
```

 WindRiver，“VxWorks 5.4 User’s Guide”的 9.2.4 节。

● Shell 返回值

Shell 执行完命令后，会按“value = ...”形式显示返回值，例如：

```
value = 68 = 0x44 = 'D'
```

有时，返回值显得完全随机：

```
value = 3925522 = 0x3be612 = buf1 + 0x2
```

返回值按十进制和十六进制显示，有时也显示字符常量，或符号地址和偏移。对于变量赋值和查看，或简单表达式计算，返回值的意思相当直观，例如：

```
-> buf1 = 0x01020304  
buf1 = 0x3be610: value = 16909060 = 0x1020304  
-> buf1  
buf1 = 0x3be610: value = 16909060 = 0x1020304  
-> 34 + 34  
value = 68 = 0x44 = 'D'
```

同样，函数的返回值也在意料之中。例如，malloc 函数返回分配内存块的地址，ld 函数返回加载模块的 ID，如下所示：

```
-> buf = malloc(100)  
buf = 0x3be620: value = 33549552 = 0x1ffecf0  
->ld < foo.out  
value = 1380512 = 0x1510a0 = _sigaction + 0x124
```

如果 Shell 认为返回值是指针，第 3 个返回值为符号表中最近似的地址解释，可以忽略。而对于没有返回值的函数，Shell 显示的值就没有意义了，例如：

```
-> buf1 = 0x01020304  
buf1 = 0x3be610: value = 16909060 = 0x1020304  
-> buf2 = 0x05060708  
buf2 = 0x3be600: value = 84281096 = 0x5060708  
-> bswap &buf1, &buf2, 2  
value = 3925522 = 0x3be612 = buf1 + 0x2
```

在这个例子中，显示的值没有实际的意义，和目标机体系结构有关。比如，PPC 将函数返回值存放在处理器的寄存器 3 (r3) 中。当命令执行完，不管发生什么，Shell 总显示寄存

器 3 中的值。这里的返回值描述也适用于 WindSh。

此外，Shell 能计算表达式的值，可以当作计算器来使用，比 Windows 下的计算器好用了。以前都安装 Matlab 来使用其 Shell 的计算器功能，现在有了 VxWorks，可以省去 Matlab 安装了。

● 行编辑库

在 tty 设备之上，行编辑库[ledLib]作为行编辑[line-editing]层，为 tShell 提供历史命令行编辑功能。tShell 输入命令的历史机制类似 Unix 的 Korn-Shell；内建的行编辑器类似 Unix 的 vi (VxWorks AE 还支持 emacs 方式)，可以编辑前面输入的命令。部分功能也类似 Dos 环境下的 Doskey。

usrLib 提供了 h 命令用来显示最近输入 tShell 的命令。命令 h 带参数可设置历史命令的缓存个数。

为了编辑命令，使用 ESC 键从输入模式进入编辑模式，可使用 ledLib 提供的命令。无论处于输入还是编辑模式，RETURN 键都向 tShell 提供输入行。

行编辑命令为 tShell 使用提供了便利。如使用 Tab 键或<CTRL+D>补全命令或符号名，与 Linux 下 Shell 的 Tab 键功能类似。具体命令参考前一章中对 wShell 的描述。

ledLib 除了在 tShell 中提供了行编辑命令，还提供 4 个程序用函数接口，如表 3-10 所示。

表 3-10

ledLib 库函数


ledOpen	创建新行编辑器 ID
ledClose	关闭行编辑器
ledRead	行编辑器读操作（阻塞式调用）
ledControl	修改行编辑器参数

这些函数接口的示例代码如下：

```
int led_id =ledOpen(STD_IN, STD_OUT, 100);
ledControl(led_id, NONE, NONE, 200);
ioctl(STD_IN, FIOSETOPTIONS, OPT_TERMINAL);
char line[255];
while (TRUE){
    ledRead(led_id, line, sizeof(line));
    cout << line << endl;
}
ledClose(led_id);
```

其中，led_id 为指向结构的指针，结构前面几个元素为 ledOpen 函数传入的参数。

如果想自己定制 ledLib，定制库代码中需提供前面描述的 4 个接口函数，并替换系统库中的 ledLib.o。可以参考其他人在这方面的工作，如 BGSB 为 VxWorks 提供带命令行编辑功能的 Shell，以及 GNU 的 Readline 库。

 WindRiver, “VxWorks Reference Manual” 的 ledLib 条目。

 BGSB, <http://www.xmission.com/~bgeer/bgsh.html>。

 Readline, <http://cnswww.cns.cwru.edu/~chet/readline/rltop.html>。

● 远程使用

VxWorks 初始启动时，tShell 一般定向在显示器或串口上。如果 VxWorks 包含了网络组件中的“telnet server”组件（INCLUDE_TELNET），系统会创建 tTelnetd 任务。就可以使用主机的 Telnet 通过网络远程访问 tShell。对于 Unix 主机，还可以使用 rlogin 访问 tShell，VxWorks 需要包含“RLOGIN server”组件（INCLUDE_RLOGIN），系统创建 tRlogind 任务。

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tTelnetd	telnetd	39eb18	55	PEND	3ed06	39e9f4	0	0

当通过网络远程使用 tShell 时，Console 上的 Shell 不再可用。因为 tShell 代码不可重入，只能支持一个任务实例。Console 上输出如下信息：

```
telnetd: This system *IN USE* via telnet.  
shell restarted.
```

当在 Console 上使用 tShell 时，可以用 shellLib 中的 shellLock 函数来禁止或允许远程登录，如下所示：

```
-> shellLock 1          /*禁止远程登录*/  
-> shellLock 0          /*允许远程登录*/
```

要终止远程登录的 tShell 会话，可以在其中调用 logout 函数，或直接关闭 Telnet 窗口。logout 在 usrLib 中定义。


由于 tShell 功能强大，需考虑远程登录的安全性问题。可包含远程登录密码保护组件 [INCLUDE_SECURITY]。使用该组件，远程登录时会提示输入用户名和密码才可访问 tShell。缺省的用户名和密码为：target 和 password。

可以使用库函数 loginUserAdd 函数来修改用户名和密码。

```
-> loginUserAdd "amine", "encrypted_password"
```

其中 encrypted_password 由主机工具 vxencrypt 获得。该工具使用加密算法，根据自己的明文密码，生成加密密码。

在程序和脚本中也可以使用 loginUserAdd 函数添加用户。

 WindRiver, “VxWorks 5.4 User’s Guide” 的 9.2.5 节

 WindRiver, “VxWorks Reference Manual” 的 loginLib 条目。

3. 6. 3 tShell 辅助调试

因为 tShell 具有 C 解释功能，能直接调用程序函数运行，所以能辅助 Tornado 完成部分调试工作。

● dbgLib

如果工程中包括了“target debugging”组件，tShell 将提供与 wShell 同样的调试命令，可以脱离主机环境，直接在目标机上进行部分调试工作，如断点跟踪调试，栈轨迹查看等。


“target debugging”组件位于“/development tool components/target shell components/”文件夹下，与 dbgLib 函数库对应，能完成断点、单步、反汇编和栈跟踪等调试工作。

tShell 中调试命令的使用与 wShell 中类似，具体细节可参考上一章的描述。dbgLib 提供的命令列表帮助可以用 dbgHelp 获得，如下所示。


```
-> dbgHelp
```

dbgHelp		Print this list
dbgInit		Install debug facilities
b		Display breakpoints and eventpoints
b	addr[, task[, count]]	Set breakpoint
e	addr[, eventNo[, task[, func[, arg]]]]	Set eventpoint
bd	addr[, task]	Delete breakpoint
bdall	[task]	Delete all breakpoints and eventpoints
c	[task[, addr[, addr1]]]	Continue from breakpoint
cret	[task]	Continue to subroutine return
s	[task[, addr[, addr1]]]	Single step
so	[task]	Single step/step over subroutine
l	[adr[, nInst]]	List disassembled memory
tt	[task]	Do stack trace on task
value = 0 = 0x0		

另外 dbgLib 有一个支持库 dbgArchLib。该模块为 dbgLib 提供与硬件体系结构相关的函数。还包括一些用户可调用的函数，用以访问任务 TCB 结构的寄存器成员。

 WindRiver, “Tornado 2.0 User's Guide” 的 6.2.6 章节。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 9.2.3 章节。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 dbgArchLib, dbgLib 条目。

● Show 函数

VxWorks 中包括一些系统信息显示函数，在 tShell 上输出目标模块或服务的相关信息。它们只显示函数调用瞬间的信息，而不能反映最新的当前信息。

要使用这些 Show 函数，必须包含 “/development tool components/show routines/” 组件夹下的相应的组件，当然还要包括符号表组件。另外 “/network components/network protocols/network debugging/” 组件夹中还包含网络调试用 Show 组件。

大多数 Show 函数都是在 VxWorks 目标机上实现的，主机 wShell 上也有几个 Show 函数，如 agentModeShow、hostShow 等。从 “/docs/rtnindex.html” 入口，可以找到所有系统公布的 Show 函数，包括主机的和目标机的。

“VxWorks 5.4 Programmer's Guide” 的 9.3.2 章节中有常用 Show 函数的列表。除了文档中描述的 Show 函数，VxWorks 库中还隐含着其他的 Show 函数，如 motFecEnd.o 中就包含着几个 Show 函数，可以帮助网络驱动的调试，但是需要源代码的支持。

开发者也可以编写自己的 Show 函数，来辅助程序的调试。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 9.3.2 章节。

● 其他

除了上面描述的 dbgLib 和 Show 函数外，tShell 中提供的其他命令对调试工作也有帮助。这些命令列表可以用系列 help 命令获得。如 ioHelp 显示文件系统操作命令，netHelp 显示网络相关命令。其他的任务相关，内存相关，符号表相关的命令也十分有用，后面将陆续讲到。将各 help 命令的输出信息保存为文本文件是个好主意，可以在整个开发过程中经常查看。

其实这一切的关键是目标机符号表的存在，利用符号表，可以直接调用程序中的输出函数名。这是 VxWorks 系统提供的一个独特便利，调试过程中应善于加以利用。比如在调试芯片驱动的时候，芯片出现问题，开发者可以从 tShell 中直接调用芯片的初始化函数，重新复位、初始化芯片，而对运行系统的其他方面没有影响，可继续调试工作或判断问题所在。这

就好像是，除 VxWorks 系统中各执行线程外，系统外部还存在一个灵活的线程（开发者的活动），可随时异步触发所需要的代码执行。这可减少测试程序的编写，完成自动化测试，可加快开发进程。

3.6.4 脚本支持

Shell 能解释脚本[Script]文件，执行文件罗列的命令。VxWorks 还支持启动时脚本文件执行，提高应用软件配置的灵活性。

脚本文件的编写很简单，每行写一个 C 表达式，Shell 能解释 C 表达式。不过 Shell 的 C 解释功能有些限制，如不支持程序流控制语句，不支持 struct、union 和 typedef，不支持宏等。

```
/*myscript*/
moduleShow
ld(0,0,"/tffs/foo.o") /*最好不要用 ld</tffs/foo.o*/
moduleShow
x = 1
y = 2
z = x + y
```

在 Shell 中用重定向执行，如下所示：


```
-><myscript
```

tShell 支持启动脚本。需要包含 Shell 启动脚本组件（INCLUDE_STARTUP_SCRIPT）。在 usrScript.c 中定义 usrStartupScript 函数用于执行启动脚本。在 usrRoot 函数中按下面的格式调用 usrStartupScript 函数。

```
usrStartupScript (sysBootParams.startupScript); /* shell startup script */
shellInit (SHELL_STACK_SIZE, TRUE); /* target shell */
```

usrStartupScript 已调用过一次 shellInit 执行脚本，这里再调用一次重启 shell。启动脚本文件由缺省 Bootline 指定，也可以在 BootRom 的 Shell 中修改。这有些麻烦，需要重新编译创建 BootRom。如果启动参数未指定脚本文件，就不执行脚本。

VxWorks 库 shellLib 中提供 shellScriptAbort 函数，用于通知 Shell 终止脚本处理，可以在脚本出错时调用。

 WindRiver, “Tornado 2.0 User's Guide” 的 6.3.14 章节。

VxWorks 暂时还不支持从 VxWorks 应用程序中执行脚本。但是，可以用已有的程序代码很容易实现从程序中调用的脚本解释器。下面列出实现的代码，该函数可以从 Shell 或程序中调用。

```
#define INPUT_LINE_MAX 100
void executeScript(char *scriptName)
{
    int ifd;    char inputLine[INPUT_LINE_MAX];
    if ((ifd = open(scriptName, 0, 0)) != 0) {
        printf ("Executing script %s ...\n", scriptName);
        while (fioRdString (ifd, inputLine, INPUT_LINE_MAX) != EOF)
            execute(inputLine);
        close(ifd);
        printf ("\nDone executing script %s\n", scriptName);
    }
    else printf("Failed to open script: %s\n", scriptName);
}
```

3.6.5 tShell 与 wShell 的区别

两个 Shell 都包含 C 解释器，支持 C Shell 和 vi 编辑。而 wShell 还提供 TCL 解释器。一个目标机可以同时激活多个 wShell，但是只能激活一个 tShell。wShell 支持虚拟 I/O，而 tShell 不支持。

tWdbTask 的 I/O 定向到 wShell。wShell 通过主机 Target Server 和 tWdbTask 来访问目标机。其他任务（包括 tShell）的 I/O 都定向到 Console 上。

tShell 编译进内核，实际在目标机上运行。而 wShell 在主机上运行，能减少目标机运行资源消耗。如果想减小程序映象的大小，可以去掉 tShell 组件。但是系统在现场运行出问题时，若没有主机调试工具，tShell 能辅助调试定位错误。

wShell 支持 WTX 协议，所以在 wShell 中可以执行主机的 TCL 命令。许多在 wShell 中可以调用的函数都是 TCL 命令或动态库函数，这些命令实际并不在目标机上存在。wShell 中执行的命令参考“Tornado API Reference”的“Target Server Internal Routines”；而 tShell 中执行的命令参考“VxWorks Lib Reference”。命令一般有对等体，描述却不一样。

在 wShell 也可执行 tShell 中的命令。命令前加“@”符号就是调用目标机的符号。例如，“@i”执行 tShell 中的命令，也就是目标机上的函数。

主机工具和目标机程序使用不同的符号表。wShell 使用主机上的符号表 tgtSymTbl，由 Target Server 维护；而 tShell 使用目标机上的符号表 sysSymTbl。例如，在 wShell 种使用“lkup”，会在主机符号表中查找符号；若在 wShell 中使用“@lkup”，就会查找目标机的符号表。当从 wShell 加载应用模块时，模块输出符号被添加到主机的符号表中，在 tShell 不能看见这些符号。启动时 tShell 加载的目标模块的符号主机也看不见。如果想保证两个符号表的一致性，需要使用符号表同步机制。

tShell 在使用过程中可能出现内存泄漏，而 wShell 没有这个问题。

两种 Shell 的内存分配是不同的。wShell 使用 WDB Agent 内存池，而 tShell 使用系统的堆。因为 tShell 使用系统堆，目标模块将加载到分配出的内存区中。内存分配使用最先适合[first fit]算法，一般位于内存池的尾部。而主机加载模块到 WDB 内存中。可参考前面 WDB Agent 和 Loader 的介绍。

 WindRiver, “VxWorks 5.4 Programmer's Guide”的 9.2.6 章节。

3.6.6 Console 和虚拟 Console


注意 Console 与 tShell 的区别。Console 算是一种终端设备(不管是实际的还是虚拟的)，如显示器、串口终端、Telnet 窗口和虚拟 Console 等。而 tShell 是 VxWorks 提供一种软件机制。Console 和 tShell 通过某种通信链路连接，如串口、Telnet 和 Target Server 等。可以将 tShell 的标准描述符定向到不同的连接，以在不同 Console 上使用 tShell 功能。

虚拟 Console 和 wShell 都是通过 Target Server 与目标机交互，但是虚拟 Console 背后实现是目标机函数库，而 wShell 依赖主机函数库。两者之间的区别还是集中在 tShell 与 wShell 的区别上。

目标机上的 Console 可以用显示器或者串口实现。串口 Console 需要 tty 库的支持，参考后面“IO 系统”一章的描述，IO 系统的配置会影响 Console 的实现。

虚拟 Console 是 Console 的一种，只是其通信连接是 Target Server 提供的虚拟 I/O 通

道0。主要解决在开发时目标机没有多余硬件通信端口做 Console 的问题。由于 wShell 和 Telnet Console 的存在，虚拟 Console 一般使用较少。通过 Target Server 配置，创建虚拟 Console，将 tShell 的 I/O 定向到虚拟 Console，或者将目标机的所有标准 I/O 都定向到 Console。必须在目标机中配置 tShell 后，虚拟 Console 中才能使用 tShell 功能。

 WindRiver, “Tornado 2.0 User's Guide” 的 1.5 章节。

3.6.7 usrLib 和 usrFsLib

tShell 的许多命令都是由 usrLib 和 usrFsLib 实现的。这两个库都提供源代码文件，位于“\$WIND_BASE/target/src/usr”目录下。用户可以修改代码文件实现自己的定制。usrFsLib 是 dosFs 2.0 的一部分。

命令 help 列出的大部分命令由 usrLib 实现。

```
-> help
help                Print this list
ioHelp              Print I/O utilities help info
dbgHelp             Print debugger help info
nfsHelp             Print nfs help info
netHelp             Print network help info
spyHelp             Print task histogrammer help info
timexHelp           Print execution timer help info
h      [n]          Print (or set) shell history
i      [task]        Summary of tasks' TCBs
ti     task          Complete info on TCB for task
sp     adr,args...   Spawn a task, pri=100, opt=0, stk=20000
taskSpawn name,pri,opt,stk,adr,args... Spawn a task
td     task          Delete a task
ts     task          Suspend a task
tr     task          Resume a task
d      [adr[,nunits[,width]]] Display memory
m      adr[,width]   Modify memory
mRegs  [reg[,task]]  Modify a task's registers interactively
pc     [task]        Return task's program counter
iam     "user"[, "passwd"] Set user name and passwd
whoami             Print user name
devs              List devices
ld      [syms[,noAbort][, "name"]] Load stdin, or file, into memory
                                   (syms = add symbols to table:
                                   -1 = none, 0 = globals, 1 = all)
lkup     ["substr"]   List symbols in system symbol table
lkAddr   address      List symbol table entries near address
checkStack [task]     List task stack sizes and usage
printErrno value      Print the name of a status value
period   secs,adr,args... Spawn task to call function periodically
repeat   n,adr,args...  Spawn task to call function n times (0=forever)
version                        Print VxWorks version info, and boot line
NOTE: Arguments specifying 'task' can be either task ID or name.
```

上面列出的命令大多为命令转向接口，命令函数还会进一步调用 VxWorks 其他库函数。用户可以利用该库源代码文件，进行功能扩展和修改。或者参考该代码文件格式，创建自己的 Shell 命令文件，再将该文件加入工程。

有些命令有可选参数。如果这些参数为 0，会使用缺省值；如果没参数输入，Shell 自动置参数为 0。

有些命令使用任务名或任务 ID 作为参数。如果省略该参数，或赋值为 0，则使用最近引用任务。usrLib 库使用 taskIdDefault 函数来设置和获取最近引用的任务 ID。TaskIdDefault 函数由 taskInfo 库提供。

命令 ioHelp 列出的文件系统操作命令由 usrFsLib 实现。

```
-> ioHelp
cd      "path"          Set current working path
pwd      Print working path
ls      ["wpat"[, long]] List contents of directory
ll      ["wpat"]         List contents of directory - long format
lsr     ["wpat"[, long]] Recursive list of directory contents
llr     ["wpat"]         Recursive detailed list of directory
rename  "old", "new"     Change name of file
copy    ["in"][, "out"] Copy in file to out file (0 = std in/out)
cp      "wpat", "dst"    Copy many files to another dir
xcopy   "wpat", "dst"    Recursively copy files
mv      "wpat", "dst"    Move files into another directory
xdelete "wpat"          Delete a file, wildcard list or tree
attrib  "path", "attr"   Modify file attributes
xattrib "wpat", "attr"   Recursively modify file attributes
chkdsk  "device", L, V   Consistency check of file system
diskInit "device"        Initialize file system on disk
diskFormat "device"      Low level and file system disk format


"attr" contains one or more of: " + - A H S R" characters
"wpat" may be name of a file, directory or wildcard pattern
in which case "dst" must be a directory name
chkdsk() params: L=0, check only, L=2, check and fix, V=0x200 verbose
```

要使用 ioHelp 列出的 usrFsLib 函数，VxWorks 需要 “File System and Disk Utilities” 组件 (INCLUDE_DISK_UTIL)。

usrFsLib 中函数 cp 和 rm 没有原型，没有对应的头文件 usrFsLib.h。usrLib.h 中有部分函数声明，如 cd。该库中函数主要用作 tShell 命令，而不用于编程。该源代码可作为学习文件系统使用的编程参考，最好不要直接调用其函数。

库 usrFsLib 中有些命令在 wShell 有对等体，完成类似功能，只是由主机实现。在 wShell 执行这些命令时应该注意，目标机操作命令应加 “@”，如下所示：

```
-> pwd /*显示 wShell 当前目录，目录处于主机文件系统中*/
-> @pwd /*显示 tShell 当前目录*/
```

 WindRiver, “VxWorks 5.4 Reference Manual” 的 usrLib, usrFsLib 条目。

3.6.8 最终产品 tShell 问题

若用户在最终产品中包含了 Target Shell 组件，需要注意以下一些问题。

- 安全性

发布软件是否包含 tShell 是个问题，因为存在很多安全性问题。

✧ 你的竞争对手有机会反向解析你的软件。

- ◇ 如果你的产品处于不安全环境，容易导致软件运行被破坏。
- ◇ tShell 并不是一个理想的用户界面，主要由有经验的开发人员使用。

最终软件包含 tShell，最大的用处就是作为现场维护时的调试工具。用户可以编写自己的用户接口，这样更安全，更符合自己的需要。若允许远程登录 tShell，最好加上密码保护组件“INCLUDE_SECURITY”。

● 内存泄漏

在某些情况下，目标机 Shell 会出现内存泄漏。

当 tShell 中表达式使用字符串，tShell 会为字符串分配内存，例如：

```
-> x = "hello there"
```

该表达式从系统内存分配 12 个字节用于串存储，地址赋值给“x”。

```
-> free (x)
```

上面的表达式释放串存储内存。如果省略该操作就会导致内存泄漏。

有时，即使串没赋值给一个符号，也会为串分配内存，却不能释放，如下所示：

```
-> printf ("hello there")
```

因为如果为串分配临时空间，串作为参数再传给任务入口函数，当任务执行时，该串的存储空间已经被 tShell 释放了，而不能被新任务引用。

tShell 中使用异常表达式也会导致内存泄漏。如被 0 除会损失大约 3KB 内存。tShell 异常后会重启，所以会导致内存泄漏。

直接用 logout 退出远程登录会有内存泄漏。因为在执行 logout 命令中间 tShell 会重启。如果用“sp logout”命令退出，tShell 不会在命令执行中重启，就不会损失内存。

当 tShell 使用很久后，消耗的内存就会积累很多，可能影响系统工作，就需要重新启动系统。

要知道，tShell 也是个任务，和其他任务一样都会有资源回收的问题。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 9.2.6 章节。

3.6.9 用户定制 Shell

由于 tShell 的限制，如安全性、不可重入和不适用等，因此用户都想定制自己的 Shell 或 CLI。本节介绍用户定制 Shell 相关的问题。

简单的命令添加可以修改 usrLib.c 和 usrFsLib.c，自己命令的添加可模仿文件中已有的函数。或者参考两文件形式，编写自己的文件，添加到工程中，并将帮助命令添加到 usrLib.c 的 help 函数中。

如果想实现多个 Shell，则需要创建自己的任务。Telnet 能支持多个会话，可以创建多个定制 Shell 实例，只要保证自己编写的代码是可重入的就行。系统的 tShell 也可以保留，用其中一个连接即可。可以免费得到 telnetd 源代码，再加上自己的代码。

可以创建一个任务用于监听 Telnet 端口的连接，并为每个连接创建定制 Shell 任务。在 Shell 中，可以用 ioTaskStdSet 将标准描述符定向到 Socket 上，如下所示：

```
ioTaskStdSet(0, 0, fd); /*重定向 stdin*/  
ioTaskStdSet(0, 1, fd); /*重定向 stdout*/  
ioTaskStdSet(0, 2, fd); /*重定向 stderr*/
```

其实 Telnet 只是一个 Socket 连接而已，也可以自己创建连接，重定向 Shell 的 I/O 到 Socket，

而不用动 telnetd 的代码，也可以创建多个连接。

系统 tShell 和用户 Shell 可以共享同一个 Console。在两者间切换，最简单的方法就是在 Console 上启动 myShell，将 Shell 作为函数调用，用 exit 推出 tShell 回到 myShell。不过还需要处理<CTRL+C>控制字符，以免太容易进入 tShell。

下面是 myShell 与 tShell 共用 Console 的代码实例。该例子的关键是，当 tShell 挂起和 I/O 重定向时，如何清空[flush]标准描述符，以及如何用 scanf 函数实现 Shell 的基本功能。该代码在串口、虚拟 IO 和 Telnet 上测试过。

```
/* consoleio.c - 从 tShell 重定向 console I/O 的例程，略去头文件包含*/
extern SYMTAB_ID sysSymTbl;
extern int errno;
int consoleAlloc = FALSE; /* already allocated */
int restartShell = FALSE; /* shell needs to be restarted*/
int tShellId =0; SEM_ID shellSemId=0;
/* store the I/O file descriptors */
int stdInFd=0; int stdOutFd=0; int stdErrFd=0;
/*****
* allocConsoleIO -
* 该函数用来为 myShell 分配 console I/O，有没有 tShell 都行。
* 如果有 tShell，将被挂起， myShell 释放 console I/O 后恢复运行
*/
int allocConsoleIO(void)
{
    int status = OK;
    WIND_TCB *pShellTcb=NULL;

    /* determine if we are running from shell and exit */
    if (taskIdSelf() == taskNameToId("tShell")) {
        printf("can't be run from the shell; use taskSpawn\n"); status = ERROR;
    }
    /* determine if console already allocated */
    else if (consoleAlloc == TRUE) {
        printf("console already allocated\n"); status = ERROR;
    } else {
        if (shellSemId == NULL) {
            if ((shellSemId = semBCreate(SEM_Q_FIFO, SEM_EMPTY)) == NULL) {
                printf("semBCreate failed\n"); status = ERROR;
            }
        } else {
            if(semTake(shellSemId,NO_WAIT) == ERROR) {
                printf("shellSemId busy\n"); status = ERROR;
            }
        }
        /* if shell exists, then save I/O and kill it */
        if ((status == OK) && ((tShellId = taskNameToId("tShell")) != ERROR) ) {
            /* get pointer to tShell TCB */
            pShellTcb = taskTcb(tShellId);
            stdInFd = pShellTcb->taskStd[STD_IN];
            stdOutFd = pShellTcb->taskStd[STD_OUT];
            stdErrFd = pShellTcb->taskStd[STD_ERR];
            /* suspend tShell */
        }
    }
}
```



```

        if (taskSuspend(tShellId) == OK) {
            /* target shell found to restart later */
            restartShell = TRUE; consoleAlloc=TRUE;
        } else {
            printf("tShell can't be suspended\n"); status = ERROR;
        }
    } else {
        /* shell doesn't exist, get console I/O directly*/
        stdInFd = ioGlobalStdGet(STD_IN);
        stdOutFd = ioGlobalStdGet(STD_OUT);
        stdErrFd = ioGlobalStdGet(STD_ERR);
        consoleAlloc=TRUE;
    }
    if (status == OK) {
        /* set allocated I/O for calling task */
        ioTaskStdSet(0, STD_IN, stdInFd);
        ioTaskStdSet(0, STD_OUT, stdOutFd);
        ioTaskStdSet(0, STD_ERR, stdErrFd);
        /* flush file descriptors */
        ioctl (stdInFd, FIOOPTIONS, OPT_TERMINAL);
        ioctl (stdInFd, FIOFLUSH, 0);
        ioctl (stdOutFd, FIOFLUSH, 0);
        ioctl (stdErrFd, FIOFLUSH, 0);
    }
    if (semGive(shellSemId) != OK) {
        printf("semGive failed\n"); status = ERROR;
    }
}
return status;
}
/*****
* freeConsoleIO -
* 释放分配的 console I/O, 并恢复前面挂起的 tShell
*/
int freeConsoleIO(void)
{
    int status = OK; FUNCPTR funcptr; SYM_TYPE type;
    /* determine if we are running from shell and exit */
    if (taskIdSelf() == taskNameToId("tShell")) {
        printf("can't be run from the shell\n"); status = ERROR;
    }
    else if (semTake(shellSemId, NO_WAIT) == ERROR) {
        printf("shellSemId busy\n"); status = ERROR;
    }
    else if (consoleAlloc != TRUE) {
        printf("console not allocated\n"); status = ERROR;
    } else {
        /* check if shell needs restarted*/
        if (restartShell == TRUE) {
            if (taskResume(tShellId) == OK) {
                /* flush file descriptors */
                ioctl (stdInFd, FIOOPTIONS, OPT_TERMINAL);
                ioctl (stdInFd, FIOFLUSH, 0);
            }
        }
    }
}

```

```

        ioctl (stdOutFd, FIOFLUSH, 0);
        ioctl (stdErrFd, FIOFLUSH, 0);
    }else {
        /* tShell exists but couldn't be resumed */
        if(taskNameToId("tShell") != ERROR) {
            taskDelete(tShellId); /* delete damaged tShell */
            tShellId = 0;
        }
        /* not much else to do except create a new tShell*/
        /* symFindByName() prevents link error if no target shell*/
        if ((status = symFindByName(sysSymTbl, "_shellRestart",
            (char **) &funcptr, &type)) != OK) {
            printf("symFindByName failed to find shellRestart()\n");
            printf("tShell couldn't be restarted\n");
        }
        funcptr(FALSE); /* call to shellRestart(FALSE) */
    }
    printf("tShell resumed\n");
    restartShell = FALSE;
} /* restartShell */
if (status == OK) consoleAlloc=FALSE;
if(semGive(shellSemId) == ERROR) {
    printf("semGive failed\n"); status = ERROR;
}
}
return status;
}
/*****
* myShell
* 使用 allocConsoleIO()/freeConsoleIO() 的定制 shell
*/
int myShell(void)
{
    int status =OK; int fEnabled=FALSE; char str[80];

    if (allocConsoleIO() != OK) {
        printf("allocConsoleIO failed\n"); status = ERROR;
    }else {
        printf("Started shell, enter or enter to exit\n");
        fEnabled = TRUE; printf("shell: ");
        while (fEnabled) {
            scanf("%s", str);
            if (strncmp(str, "quit", 4) == 0) {
                fEnabled = FALSE; continue;
            }
            printf("echo: %s\n", str);
            printf("shell: ");
        }
        printf("\nshell: done.\n\n");
        if (freeConsoleIO() != OK) {
            printf("freeConsoleIO failed\n"); status = ERROR;
        }
    }
}

```


```

return status;
}

```

要使用该定制 Shell，需要在 tShell 中运行“sp myShell”命令来创建新任务，因为需要挂起 tShell，该 myShell 不能使用 tShell 任务的上下文。该代码只是一个框架，需要补充自己的功能代码。下面是代码的测试显示：

```
-> sp myShell
Started shell, enter to exit
shell: test1
echo: test1
shell: test2
echo: test2
shell: quit
shell: done.
tShell resumed
```

 参考 WindRiver 网站 WindSurf 上的 myShell 代码。

3.7 其他组件

除了上面介绍的基本组件外，VxWorks 中还存在其他一些组件，一般在实际应用中不用详细了解。下面各小节对它们分别作简要介绍。

3.7.1 ANSI C库

几乎所有 C 开发环境都有标准 C 库的实现，VxWorks 也不例外。VxWorks 的标准 C 库包括一些小组件，都位于“os comps/ANSI C comps[libc]”组件夹下，如表 3-11 所示。

表 3-11 ANSI C 库

ANSI C 库

ANSI assert	INCLUDE_ANSI_ASSERT	ansiAssert	断言诊断功能
ANSI ctype	INCLUDE_ANSI_CTYPE	ansiCtype	字符类型判断和操作
ANSI locale	INCLUDE_ANSI_LOCALE	ansiLocale	本地化功能
ANSI math	INCLUDE_ANSI_MATH	ansiMath.	数学函数库
ANSI stdio	INCLUDE_ANSI_STDIO	ansiStdio	标准输入输出库
ANSI stdio ext	INCLUDE_ANSI_STDIO_EXTRA	ansiStdio	实现 getw、putw 和 setbuffer 等 WRS 自己的函数
ANSI stdlib	INCLUDE_ANSI_STDLIB	ansiStdlib	标准库
ANSI string	INCLUDE_ANSI_STRING	ansiString	串和内存操作函数
ANSI time	INCLUDE_ANSI_TIME	ansiTime	时间操作库
		ansiSetjmp	强制长跳转实现
		ansiStdarg	变数参数实现

表中最后两种库的实现，在组件中未列出，但在库用户参考文档中有相应描述。用户在某些特殊情况下才会使用这两个库，比如实现自己的 `printf` 函数。


标准 C 库[libc]对 C 开发人员来说是非常重要的,属于最基本的常识。如果对这些还不熟悉的读者,建议参考相关文档做详细了解。

ANSI C 库的名称都以 `ansi` 开头，和一般的库命名一样。在库参考中容易找到其相关帮助。函数名加“`r`”后缀，如 `div_r` 函数，表示该函数为标准 C 库中对应体的可重入实现。`libc`

对函数的可重入性没有规定，不能保证可重入，这不太适应多任务环境。这些可重入函数不属于标准 C 库，算是 WRS 自己的扩展实现。

WRS 扩展实现还包括 `getw`、`putw` 和 `setbuffer` 等。而且 WRS 也没有全实现标准 C 库，如 `mblen` 函数等，这在库帮助文档中都明确指出。

标准 C 库实现和 VxWorks 其他库有密切关联，如标准输入输出库和 I/O 系统，内存操作函数和 `memLib` 等。

VxWorks 工程创建时，这些组件都被选中有效，由其他一些 VxWorks 组件使用。 WindRiver，“VxWorks 5.4 Reference Manual”的下相关条目，按表 3-11 第 3 列名称。

3. 7. 2 数据结构组件

数据结构组件位于“/os comps/utility comps/”组件夹下，如表 3-12 所示。

表 3-12 数据结构组件库

Pool library	INCLUDE_POOL_LIB	poolLib	内存池支持函数
Set library	INCLUDE_SET_LIB	setLib	对象集支持函数
Buffer manager	INCLUDE_BUF_MGR	bufLib	缓冲区管理
Doubly linked lists	INCLUDE_DLL	dllLib[lstLib]	双向链表
Hash library	INCLUDE_HASH	hashLib	哈希表
Ring buffers	INCLUDE_RNG_BUF	rngLib	环行缓冲区

数据结构也是软件开发人员的基本知识，具体概念请参考相关教材。

这些数据结构组件几乎都是独立实现组件，对 VxWorks 的其他库没有依赖，而是提供服务。

VxWorks 工程创建时，这些组件都被选中有效。因为其他很多 VxWorks 组件都依赖它们的实现。

这几个组件在库参考中几乎未作描述，除了 `rngLib`，而 `lstLib` 的库描述和实际工程中组件属性不一致，不知道是否可用，笔者没实际试过。看来这些库并不是面向用户提供的，而只是用于支持其他库的实现。用户要使用这些类型的数据结构，可能还是自己编码来得可靠。

 WindRiver，“VxWorks 5.4 Reference Manual”的下相关条目，按表 3-12 第 3 列名称。

3. 7. 3 复位支持库

复位支持库[`rebootLib`]提供复位支持函数，用于复位目标机，将控制权交给 `BootRom`。

可以在 `Shell` 中显式调用 `reboot`，或使用<CTRL-X>控制字符隐含调用，来复位 VxWorks。可以选择使用 `Hook` 函数，一般用来复位和同步硬件。例如，`netLib` 就添加了一个 `Hook`，用来复位所有网络接口。由于可以添加多个 `Hook`，需要注意 `Hook` 之间的顺序。`reboot` 执行完 `Hook` 后，闭锁中断，调用 `sysToMonitor` 函数将控制权交给 `BootRom`。`sysToMonitor` 在 `sysLib.c` 中实现，和硬件体系相关。


有多种复位形式可以选择，这由 `reboot` 函数的参数指定，主要是关于 `BootRom` 进入，内存清除，复位速度等。

 WindRiver，“VxWorks 5.4 Reference Manual”的下 `rebootLib` 条目。

3.7.4 可选组件


- VxDCOM

VxDCOM 是参照微软 DCOM 为嵌入式应用领域定制的具体实现。制造商使用 VxDCOM 可以方便地生成快速、紧凑的嵌入式应用，与远端的计算机实现无缝隙的交互连接。

 WindRiver, “VxWorks 5.5 User’s Guide” 的第 9 章。


- VxFusion

VxFusion 是运行在目标系统上的 VxWorks 系统的扩展组件，专为松耦合的分布式处理系统而设计，基于消息传送来实现分布式系统之间的交互。VxFusion 能使基于 VxWorks 之上的应用程序无缝地扩展（跨平台、跨处理器），而不必考虑应用程序的地域性限制和分布式系统的连接方式（介质、结构）。VxFusion 非常适合一般的分布式系统之间的交互和特殊的应用环境。当今的网络通信设备都有对软件 HA（High Availability）的要求。VxFusion 能实现诸如 Heartbeat 消息的传送，节点状态的监控以及 Checkpoint 状态信息传送的功能，它还具有容错功能。当分布式系统中某一结点失效。VxFusion 能自动重新路由，传送数据信息到备份的节点上。VxFusion 特别适用于数字通信、国防、工业控制等领域中多 CPU 应用，发挥它的强大功能。

 WindRiver, “VxWorks 5.5 User’s Guide” 的第 10 章。


- VxMP

VxMP 是 VxWorks 多处理器支持扩展包，它允许将任务分布在多个 CPU 上执行以提高效率。它透明的、高性能的设计使得在不同 CPU 上运行的任务可以通过现有的通信机制，如信号量、消息队列等进行同步和数据交换。

 WindRiver, “VxWorks 5.4 User’s Guide” 的第 6 章。


- VxVMI


VxVMI 为带有 MMU 的目标板提供了虚拟内存机制。在调试阶段和软件运行时都能提供强大的内存管理功能，它包括代码段和数据段保护功能，并包含对不同 CPU 结构的标准编程接口。可参考本书第 5 章的介绍。

 WindRiver, “VxWorks 5.4 User’s Guide” 的第 7 章。

- WFC

WFC 为 Wind Foundation Classes 缩写。该 C++库主要包括两部分：VxWorks 包装类库和 “tools.h++” 库。前者为几个标准 VxWorks 模块提供 C++接口。后者来自 Rogue Wave 软件公司，提供一些 C++特性。

 WindRiver, “VxWorks 5.4 User’s Guide” 的第 5 章。

 tools.h++库文档 “<http://robotics.jpl.nasa.gov/people/jack/ndoc/rwdocs/tlsug/booktoc.htm>”。

3.8 常见问题解答

- 出现错误 ‘Loading symbol table error: status = 0xe0001’ ?

解答：“0xe0001” 错误代码的具体意思是 S_loadLib_ROUTINE_NOT_INSTALLED。很明显表示目标机上程序映像未包含 Loader 组件。在目标机上动态加载模块时，就会出现该错误。加入 “target loader” 组件（INCLUDE_LOADER），重新编译创建 VxWorks 程序映像。

- 如何用 `symFindByName` 取得函数地址？

解答：如果想查找地址，可以参照下面的做法。下面的命令在 Shell 中执行，也可以将这些步骤组合为 C 代码。`SymFindByName` 的入口参数可以参考相关文档。

```
-> lkup "SymT"
statSymTbl          0x001483c6 bss
sysSymTbl           0x00145236 bss
```

`sysSymTbl` 为系统符号表，可以用它来作为 `symFindByName` 函数的第一个入口参数。

创建 `symFindByName` 函数的指针参数，用来接受函数符号的地址和类型（赋值任意，主要是为了申请内存和添加新符号）。

```
-> addr=0
new symbol "addr" added to symbol table.
addr = 0x2550e0: value = 0 = 0x0
-> type=0
new symbol "type" added to symbol table.
type = 0x255110: value = 0 = 0x0
```

例如，查找 `rootTaskId` 变量，结果如下所示：

```
-> lkup "root"
rootTaskId          0x00146176 bss
-> symFindByName ( sysSymTbl, "rootTaskId", &addr, &type)
-> addr
addr = 0x2550e0: value = 1335670 = 0x146176 = rootTaskId
```

地址为 `addr` 的值，和用 `lkup` 命令查找的结果一样。

```
-> type
type = 0x255110: value = 150994944 = 0x9000000
```

- 如何用新加载模块的符号替换原符号表的符号？

解答：VxWorks 符号表中可以包含多个同名符号。当加载引用时，符号表查找返回最近的一个。可以用 `group` 号来判断该符号属于哪个模块。因此，如果加载的模块符号与原 VxWorks 中符号相同，后续对该符号的引用就指向加载模块的符号。这只对在该模块后加载的模块有效，而对之前已加载模块无效，因为连接已经完成，不会引用新符号。

- 如何删除加载模块中的所有符号？

解答：可以使用 `symEach` 来遍历符号表，用模块 ID 查找符号，下面是示例代码。

```
LOCAL BOOL unldSymbols
(
    char *      name,          /* not used */
    int         val,           /* not used */
    SYM_TYPE    type,          /* not used */
    int         deleteGroup,   /* group number to delete */
    UINT16      group,         /* group number of current symbol*/
    SYMBOL *    pSymbol        /* a pointer to the symbol itself */
)
{
    if (group == deleteGroup){
        if (symTblRemove (sysSymTbl, pSymbol) != OK) {
            printf ("got bad symbol, %#x\n", (UINT) pSymbol);
            return (FALSE);
        }
    }
}
```

```

    }
    /* free the space allocated for the symbol itself */
    symFree (sysSymTbl, pSymbol);
}
return (TRUE);
}
/* 从系统符号表中删除加载模块的符号 */
void *rmAllSym
(
    MODULE_ID moduleId /* module whose symbols will be removed */
)
{
    return symEach (sysSymTbl, (FUNCPTR) unldSymbols, (int) moduleId->group);
}

```

- 动态加载模块时出现“unresolved symbols”错误是什么原因？

解答：一种可能是 VxWorks 中没有包含符号表组件。或者该模块依赖的模块没有先加载。或者确实有未解析符号。这在编译目标模块时不会报错，因为编译器假设目标模块的未解析符号会在连接时解析。弱符号（未初始化全局量）或未定义符号都试图从前面加载模块获得解析。

- 为什么加载模块时出现“undefined symbols”信息，而且程序出现异常？

解答：当加载模块时，Loader 会解析模块的外部符号。如果这些符号在符号表中不存在，不能被解析，就会出现“undefined symbols”出错信息。

虽然出错，但 Loader 继续完成加载，以便让开发人员调试。模块虽然被加载，但是不能执行，否则会产生程序异常。

当加载 C++ 模块时，因为 ctors 和 dtors 处理被设置为 AUTOMATIC，构造函数会自动执行，这就会引起异常情况。为了解决这个问题，可以用下面命令禁止自动执行：

```
-> cplusXtorSet 0
```

这样就只会看见出错信息，而不会有程序异常。

- 当从 tShell 中用 loadModule 加载模块时，出现“Relocation value does not fit in 24 bits”出错信息，但同一模块却可通过主机顺利加载。为什么有两种不同结果？

解答：通过主机 Target Server 加载模块，模块将放在 WDB_POOL 中，该内存区紧接着 VxWorks 映象区。对 VxWorks 库函数的调用，地址跨度会在 24 位容量内。然而，通过 tShell 加载的模块位于系统内存最后，如果内存很大，地址跨度可能超出了 24 位容量，就会产生如上错误。为了解决这个问题，需添加“-mlongcall”编译选项来编译目标模块。

- 加载目标模块时出现错误“__eabi”，这是什么原因？

解答：可能使用了“main”应用函数名称。“main”函数名应作为关键词，避免使用。

- app.o 在 wShell 中加载正常，而在 tShell 加载时却出现内存不够，这是为什么？

解答：两个 Shell 加载模块时有很多不同，比如 wShell 使用 WDB_POOL，而 tShell 使用目标

机系统内存。

在 tShell 中用 memShow 查看空闲内存还有多少,再用 objdump 命令查看 app.o 需要从目标机分配多少内存。如果用 TFTP 加载,可能需要 2~3 倍内存来保存模块映象,直到加载完成。

wShell 能加载成功可能是因为只有部分 app.o 需要占用目标机内存,所有调试信息保留在主机上。WDB_POOL 缺省占用 1/16 的系统内存,由主机工具使用。

用主机命令“objdump<arch> --headers app.o”查看输出。要知道 Loader 并不知道这些,直到映象完全下载到目标机内存。加载后才释放掉不需要的内存,而且还要分配内存用于向符号表中添加符号。C++模块可能还有额外的要求。

● 有 wShell, tShell 还有用吗?

解答:除了 tWdbTask 的 IO 定向在 wShell 上,其他任务 IO 都缺省定向在 tShell 上。如果用户不想在主机 Shell 中反复重定义标准 I/O 方向,目标 Shell 还是有用的。

VxWorks 程序映象可以包括目标 Shell 和 telnetd 服务,这样就可以 Telnet 登录 VxWorks 系统,在 Tornado 外使用 I/O 控制台。用户可继续在主机 Shell 中输入指令,就可以在 Telnet 控制台观察调试输出信息。当使用启动脚本时,则需要目标机 Shell 支持。在工程现场,用户也可以利用 tShell 提供一些调试功能,解决现场问题。

● 如何将 Shell 命令输出定向到文件中?

解答:用下面的函数,可以将命令的输出写入文件中。

```
extern "C" int shellToFile(char * shellCmd, char * outputFile)
{
    int rtn; int STDFd; int outFileFd;
    outFileFd = creat( outputFile, 0_RDWR);
    printf("creat returned %x as a file desc\n",outFileFd);
    if (outFileFd != -1) {
        STDFd=ioGlobalStdGet(STD_OUT);
        ioGlobalStdSet(STD_OUT,outFileFd);
        rtn=execute(shellCmd);
        if (rtn !=0) printf("execute returned %d \n",outFileFd);
        ioGlobalStdSet(STD_OUT,STDFd);
    }
    close(outFileFd);
    return (rtn);
}
```

在 tShell 中执行该命令,如下所示:

```
shellToFile("ifShow","ifShow.out")。
```

● 在 Shell 中如何使用 lkup 查找符号?

解答:lkup 调用在主机和目标机上有不同的实现,wShell 中使用 lkup 会查找主机的符号表,而 tShell 中使用 lkup 会查找目标机的符号表。如果主机和目标机符号表不同步,同样的 lkup 命令在两个 Shell 上会有不同的结果。

->lkup “”	/*显示符号表中符号*/
->lkup “dd”	/*显示所有含“dd”的符号名*/

lkup 命令传入串为正则表达式，可以简单，也可以很复杂

- `loadModule` 函数调用出现的 0xE0001 错误是什么原因？

解答：Tornado 工程配置存在问题。配置 `INCLUDE_LOADER` 后，自动生成的 `prjConfig.c` 没有调用实际初始化函数。

```
void usrToolsInit (void)
{
    moduleLibInit ();
    /* support library for the target-based loader. */
    /*? target loader */

    .....
}
```

而命令行编译使用的 `usrConfig.c` 没有这问题。

```
#ifdef INCLUDE_LOADER
    moduleLibInit ();          /* initialize module manager */
#if defined(INCLUDE_ELF)
    loadElfInit ();           /* use elf format */
#endif
#endif /* INCLUDE_LOADER */
```

如果目标模块为 ELF 格式，`prjConfig.c` 中代码应为如下所示：

```
void usrToolsInit (void)
{
    moduleLibInit ();
    /* support library for the target-based loader. */
    loadElfInit();            /* target loader */
    .....
}
```

但 `prjConfig.c` 是工程工具自动生成的，所以不能手动修改。用户可以自己写组件补丁，或者在 `usrAppInit` 函数中添加实际的初始化函数调用（如 `loadElfInit` 函数）。

第 4 章 多任务环境

经过系统启动过程，硬件和上层软件组件的初始化完成后，系统进入多任务运行环境。系统中最基本的执行线程单位为任务和中断。任务由操作系统内核调度执行。中断由硬件异步触发，执行中断处理代码，中断处理结果会影响系统中任务的运行状态。

系统存在多个任务，包括 VxWorks 基本的系统任务和用户自己创建的任务。用户任务是实现应用目的的执行线程，在本质上和系统任务没有区别，所有任务位于统一地址空间，都运行在 CPU 最高级别的系统模式（具有更好的系统性能）。

任务一般都是无限循环执行的线程。操作系统协调各任务公平分享 CPU。所谓的操作系统隐藏在各任务和中断背后，在系统运行时，并不以独立的形式存在，没有自己的执行线程。就像赛场外的裁判，只是协调整个比赛过程，而不亲自参加比赛，由运动员自己进行比赛。什么时候这个裁判才起作用呢？当系统时钟中断或系统调用时，操作系统代码才会出面协调任务的运行。而中断处理在某些方面超出了操作系统软件的管理范围，而由 CPU 硬件触发，比所有任务都优先执行。中断处理以外剩余的 CPU 时间，由各任务依据优先级设置、资源分配和执行流程来共同分享。

在系统运行时，除了任务、中断这样的动态运行对象外，还存在一些静态对象，如调度机制、任务间通信机制、内存管理机制和输入输出系统等。这里所讲的静态是相对的，指这些对象代码没有自主的执行线程，总处于被动的被调用状态。调用时对内存中分配的静态数据结构进行操作，有时也改变任务的运行状态。

4.1 任务概念

任务是一个动态的抽象对象，可在运行时创建和删除，是多任务环境中最基本的执行单元。任务是多方面的综合，包括状态、堆栈、上下文和执行代码等。每个创建的任务都对应一个内核数据结构，即任务控制块[TCB]，用于保存关键的动态信息。

4.1.1 任务控制块

任务控制块[TCB]为一个数据结构，内核用来表示和控制任务。每个创建的任务，内核都会为它分配一个 TCB。TCB 中包括硬件相关的寄存器（包括 PC 和 SP）、异常信息等。另外还包括与硬件无关的一些参数，如名称、选项、调度相关、函数、Hook、栈、任务变量、标准 IO 和环境变量等。TCB 是 VxWorks 中最重要的数据结构，若想深入理解 VxWorks，应该仔细阅读 taskLib.h 中的定义。后续小节相关的内容都和 TCB 有关。

- 上下文[context]

上下文是任务运行的环境。当系统调度新任务运行时，需要进行上下文切换（保存老任务的上下文信息，并恢复新任务的上下文）。任务的上下文存放在 TCB 中，各种运行环境信息在 TCB 结构中有相应体现。上下文中最重要信息是程序计数器（PC）和栈指针寄存器（SP），PC 表示当前任务切换时正运行的代码位置，SP 表示当前任务切换时任务栈的指针位置，PC 和 SP 是多任务环境实现的基础。

- 标准 IO

像大多数编程环境一样，VxWorks 为每个任务提供了 3 个标准 IO 文件符：输入（stdin）、输出（stdout）和出错（stderr）。各任务的文件符值可以不同，指向不同的文件或设备。

任务的标准 IO 文件符存放在 TCB 中，如下所示：

```
int taskStd[3]; /*文件描述符*/
```

在任务创建时，taskStd 数组初始化为 0、1、2，缺省使用全局的标准 IO 文件描述符。各任务可以使用 ioLib 中的 ioTaskStdSe 函数来操作 TCB 中的 taskStd，以定制自己的标准 IO。

● 错误状态

每个任务都有自己的最近的错误状态表示，在 TCB 的 errorStatus 中存储。错误状态的具体细节可以参考本书 3.4.5 节中关于 errnoLib 的描述。

4.1.2 任务变量

有些被多个任务共享的函数，可能需要全局或静态变量来为各任务提供不同的值。比如，任务想通过同样的全局变量引用自己的私有缓冲区，这种情况下，可以将这样的全局变量添加为任务变量。任务变量可以为任务保存全局变量的备份值。当任务切换时，任务变量机制会切换全局变量的值，如图 4-1 所示。

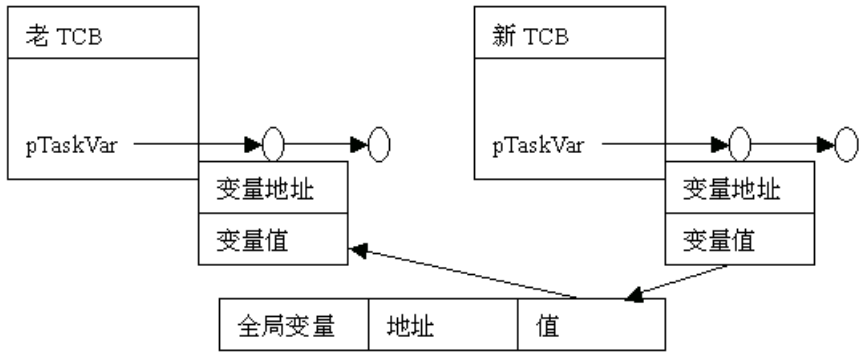


图 4-1 任务变量

每个任务变量允许存放一个 4 字节的值，添加为任务变量的全局变量应为 4 字节类型。任务变量为如下结构，在 taskVarLib.h 中定义。

```
typedef struct taskVar { /* TASK_VAR */
    struct taskVar * next; /*指向下一个任务变量*/
    int * address; /*全局变量的地址*/
    int value; /*全局变量的值*/
} TASK_VAR;
```

任务变量以单向链表的形式存在，链表头为 TCB 中的 pTaskVar。每添加一个任务变量，就会分配这个结构连接入链表中。当任务不运行时，value 用于保存该任务相关的变量值；任务运行时，value 保存任务进入时变量的原始值。因此，多个任务分享同一个全局变量时，可以有一个任务不使用任务变量机制，只要其他的任务用任务变量机制来使用该全局变量即可。

VxWorks 使用任务切换 Hook 函数来实现任务变量机制，任务变量库初始化时会添加 Hook 函数，任务切换时调用 Hook 函数遍历任务变量链表，进行变量值的恢复和保存。所以任务变量机制和任务变量个数都会降低任务切换效率，在编程中应该谨慎使用。建议将一个模块的所有任务变量组合为一个结构，将结构指针作为任务变量。如果不使用任务变量，最好在组件配置中不要包含 INCLUDE_TASK_VARS。

任务变量机制由 taskVarLib 库提供，有如下函数接口。

```
STATUS taskVarInit (void);
```

taskVarInit 函数初始化任务变量机制，添加 Hook 函数，只执行一次。可以不显式调用，添加第一个任务变量时会自动执行。

```
STATUS taskVarAdd(int tid, int *pvar);
```

taskVarAdd 函数添加任务变量。参数 tid 为任务标识符，参数 pvar 为全局变量地址。

```
STATUS taskVarDelete(int tid, int *pvar);
```

taskVarDelete 函数以全局变量地址搜索任务变量链表，删除第一个地址相同的任务变量。

```
int taskVarGet(int tid, int *pvar);
```

```
STATUS taskVarSet(int tid, int *pvar, int value);
```

以上两个函数用于读取和写入任务变量值，也是以全局变量地址为索引。

```
int taskVarInfo(int tid, TASK_VAR varList[], int maxVars);
```

taskVarInfo 函数用于读取多个任务变量。参数 varList 为接收数组，参数 maxVars 为接收数组的长度，返回值为实际读取的任务变量个数。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 2.3.9 章节。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 “taskVarLib” 条目。

4.1.3 环境变量

任务环境变量用来定制代码的行为，以保证代码的广泛适用性。环境变量类似代码中的宏定义，都可以用来定制代码行为，而宏定义是硬编码的，环境变量则更灵活，如在启动时可读取配置文件来设定环境变量值。有些库函数中使用了环境变量，如 ansiTime 库中 TIMEZONE 环境变量，应用程序使用该库前需设定 TIMEZONE，ansiTime 中读取该变量来输出正确的时间值。

VxWorks 中环境变量包括两个层次：全局环境和任务环境。如果任务没有创建自己私有的环境，该任务上下文中的环境变量操作都在全局环境中进行。环境变量都是串格式，在使用前要进行串解析。

任务私有环境变量的参数存放在任务 TCB 中。ppEnviron 指向环境变量串指针的存储空间，在私有环境创建时从堆中分配，如果不够用会扩大分配。envTblSize 表示该空间当前最大可容纳环境变量指针的个数。nEnvVarEntries 表示私有环境中已经存在的环境变量个数。

envLib 库用来实现环境变量机制，提供如下函数接口：

```
STATUS envLibInit(BOOL installHooks);
```

创建全局环境变量存储，选择安装任务环境变量的 Hook 函数。如果选择安装 Hook 函数，或任务创建使用参数选项 VX_PRIVATE_ENV，Hook 函数会自动建立该任务的私有环境，并继承父任务的环境变量，因为 Hook 在父任务的上下文中运行。如果不选择 Hook 函数或 VX_PRIVATE_ENV 选项，可在任务运行后创建私有环境或使用全局环境。

```
STATUS envPrivateCreate(int taskId, int envSource);
```

如果任务创建时没有建立私有环境，可用 envPrivateCreate 函数显式创建。参数 envSource 用于确定是否继承全局环境：创建空的私有环境（-1），继承全局环境（0），继承指定任务的环境（taskId）。该函数还会设置任务选项 VX_PRIVATE_ENV。

```
STATUS envPrivateDestroy(int taskId);
```

```
STATUS putenv(char *pEnvString);
```

putenv 函数用环境变量添加或修改。环境变量串本身从堆中分配内存存放，串指针存放在全局或任务环境分配存储中。

```
char *getenv(const char *name);
STATUS envShow(int taskId);
```

 WindRiver, “VxWorks 5.4 Reference Manual” 的下 envLib 条目。

4. 1. 4 任务栈

通过任务上下文中栈指针的切换，各任务使用独立的栈来保存局部变量和函数调用轨迹，栈的切换是多任务建立的基础。栈是刚创建任务的惟一资源，由应用代码指定栈大小，从内存池中分配，栈的大小应该为偶数。TCB、任务调试结构 WDB_INFO 以及任务名都存储在栈中，剩余的空间作为实际的栈使用。栈初始时由 0xEE 填充，用于 checkStack 检查栈情况。任务栈的内存示意如图 4-2 所示。

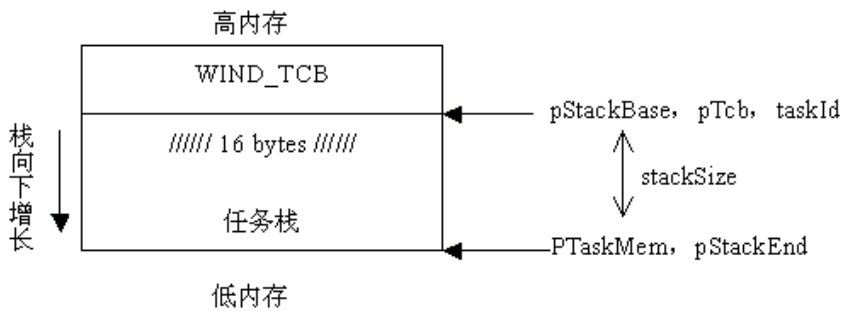


图 4-2 任务栈


TCB 中提供 pStackBase 来存储栈基址，pStackEnd 指向栈结束，初始时 pStackLimit 和 pStackEnd 相同，用于实现 taskStackAllot 函数从任务栈尾部分配内存。TaskStackAllot 函数为非阻塞调用，主要用在任务创建 Hook 函数中分配内存。

4. 1. 5 任务选项

任务选项用于任务行为的定制。各任务选项在 taskLib.h 中定义，如表 4-1 所示。

表 4-1 任务选项

选项	说明
VX_UNBREAKABLE	不允许断点调试，比如 tShell 任务，在其上运行的函数不可设置断点
VX_DEALLOC_STACK	可释放任务栈，用于任务删除
VX_FP_TASK	用于支持浮点操作
VX_PRIVATE_ENV	创建任务的私有环境变量
VX_NO_STACK_FILL	不用 0xEE 填充栈，任务重新启动（taskRestart）时使用

 WindRiver, “VxWorks 5.4 Reference Manual” 的下 taskLib 条目。

4. 1. 6 任务状态

任务在运行过程中，状态会不停的变化，而任务的状态影响内核调度的决策。任务有多种状态来表示当前任务的情况，有些还是组合状态。VxWorks 中的某些系统调用会改变任务的状态，使各任务协调运行。各种可能的任务状态如表 4-2 所示。

表 4-2 任务状态

状态	说明
READY	等待 CPU 执行，参与任务调度
PEND	阻塞等待某种资源，如信号量
DELAY	睡眠状态
SUSPEND	挂起状态，如任务出现异常
DELAY+S	睡眠+挂起状态
PEND+S	阻塞+挂起状态
PEND+T	延时阻塞状态，可能超时退出该状态
PEND+S+T	延时阻塞+挂起状态
...+I	继承优先级状态，优先级被暂时提升，+I 可以和上面所有状态组合
DEAD	任务不再存在

TCB 中用 `status` 来存储任务状态，各状态的值在 `taskLibP.h` 中定义。继承优先级状态通过 TCB 中的 `priority` 和 `priNormal` 是否相等来判断。可用 `taskStatusString` 函数来获取表 4-2 所列的任务状态串。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 2.3.2 章节。

4.1.7 任务调度

VxWorks 中多任务共享一个 CPU。操作系统的调度算法将 CPU 时间适当分配给各任务，以使多任务协调运行。调度算法在系统调用和系统时钟中断后，可能被激活，选择最合适的任务调入运行，占用 CPU。当所有任务都不可用时，系统 `Idle` 任务被调入用来消耗 CPU 时间。当内核将 CPU 从一个任务交给另一任务时，会发生任务上下文切换[context switch]。

多任务操作系统使用多种调度算法，如优先级、优先级抢占、循环、FIFO 和均匀分时等，各有自己的特点，适用不同的环境。实时系统一般使用优先级抢占调度算法。

VxWorks 缺省使用优先级抢占调度算法，还可以同时选用同级时间片循环[round-robin]调度算法。

● 优先级抢占调度

优先级抢占调度[priority preemptive]算法原则是高级 `ready` 状态的任务可以抢占正在运行的低级任务，中断可以抢占任何任务。算法使高优先级任务和中断处理可以得到及时处理，以保证系统的实时性。

下面看个示例，来理解这种调度算法的基本过程，如图 4-3 所示，其中 `INT6` 为系统时钟中断，`INT3` 为网络中断。图中出现的几次任务调度如下。

- ✧ 系统任务 `idle` 被中断 `INT6` 抢占，`INT6` 执行完成后 `idle` 恢复运行。
- ✧ `tNetTask` 抢占 `idle` 运行。
- ✧ `tEvtTask` 抢占 `tNetTask` 运行；`tEvtTask` 执行完后恢复 `tNetTask` 运行。
- ✧ `tEvtTask` 再次抢占 `tNetTask` 运行。
- ✧ `tEvtTask` 被中断 `INT3` 抢占，`INT3` 执行完成后 `tEvtTask` 恢复运行。
- ✧ `tEvtTask` 执行完后恢复 `tNetTask` 运行。
- ✧ `tNetTask` 执行完后恢复 `idle` 运行。

上面每一句话都包含一次调度、一次任务上下文切换。

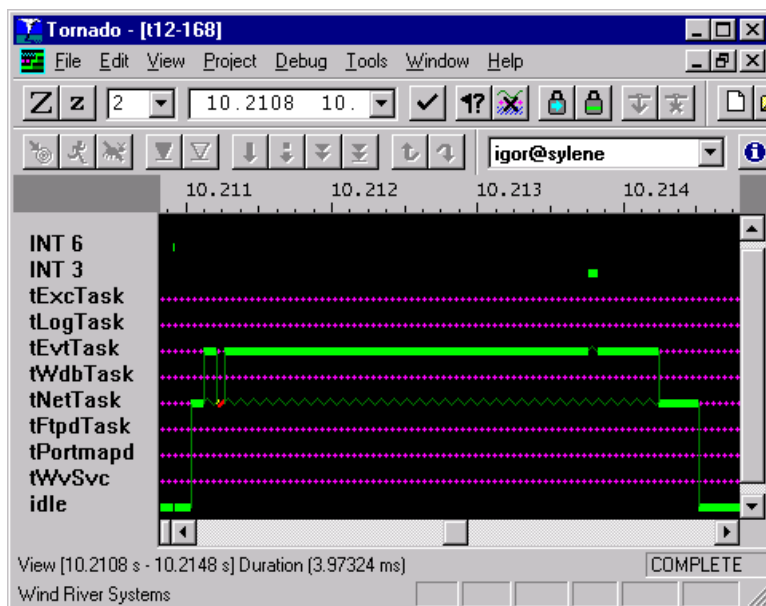



图 4-3 Windview 任务调度显示

VxWorks 中每个任务都有优先级。一共有 256 个优先级，编号从 0~255。0 为最高优先级，255 为最低优先级。任务在创建时就必须指定优先级，但是在系统运行过程中，任务可以调用 `taskPriority` 函数来改变自己的优先级，甚至可以调用 `taskLock` 函数来独占 CPU。这样系统可以随环境的变化自主动态调整任务优先级。

当然，这种调度算法在带来实时性的同时，也增加了系统的复杂性。系统任务的执行流程是不可控的，随环境而变化。任务之间的共享资源需要加以保护，否则会出现访问冲突。必须小心使用任务间的同步机制，防止出现系统死锁。也许读者听说过使用 VxWorks 的美国火星探路者在火星上反复复位的故事，就是因为系统调度复杂性导致的。

高优先级任务可以尽快完成自己的工作，然后阻塞等待下次 ready 时再被调度。如果没有足够的 CPU 时间来处理所有高优先级任务（包括中断服务程序），低优先级任务就得不到运行，而处于饥饿状态[starve]。这样的情况下，要么减轻任务的工作量，要么换一个更快的处理器，否则再好的调度算法也是无用的。

可能还存在这样一个问题：低级任务正在运行，高级任务突然变为 ready 状态，高级任务要过多少时间才能获得 CPU 运行。其实这个问题在于“突然”两个字，阻塞的高级任务是不可能无缘无故地突然变为 ready 状态，必须通过其他任务或中断程序来激活，而激活的手段就是系统调用，比如任务间通信原语等，而这些会引起任务状态变化的系统函数会自动调用调度算法，在最短的时间内或者说从系统内核返回后将高级任务投入运行，可以说是马上执行，至少这段时间不可能有其他低级任务运行的机会。

 Mitch Neilsen, “CIS 721, Lecture 13, Priority inheritance protocols and vxworks”。

● 同级时间片循环调度

优先级抢占式是 VxWorks 的基础调度算法，根据它的原则，同级任务之间是不允许抢占的。这时只遵循 FIFO 的基本原则进行调度：同级任务不可抢占，必须等正执行的同级任务完成后，等待的任务才能运行；如果不断有同级任务转为 ready 状态，这些任务按状态变化的时

间顺序排入等待队列，先进入队列的先运行。

而为了保证系统的实时性，对同级任务也应该有相应的调度算法。VxWorks 的同级时间片循环调度[round-robin]算法就是为了解决这问题而提供的。循环算法与上面描述的原则不冲突，可以和优先级算法一起用，但是优先级算法是最基础的，循环算法应该在它描述的原则下作用。循环调度算法让所有同级的 ready 任务共享 CPU，调度时间单位为时间片。一个任务执行一个时间片后，另一个任务再调入运行一个时间片，如此循环保证同级任务平等获得 CPU 时间。

同级时间片循环调度缺省是禁止的，kernelTimeSlice 函数用来控制是否使用它。若函数参数 ticks 为 0，则禁止时间片循环调度，同级任务实行 FIFO 调度，同级任务之间不能抢占，必须等前面任务运行完成后，后面的任务才会被调度。若 ticks 为 n，同级任务之间实现时间片循环调度。一个任务运行一个时间片（n 个 ticks，tick 为系统时钟单位）后，相同优先级 ready 队列首任务会被调入运行，而不管前一任务是否执行完成。前一任务退出运行后，再排入相同优先级 ready 任务队列尾部，等待下一次调入。这样反复，同级 ready 任务队列中的任务都有机会及时运行，提高了系统实时响应速度。

下面介绍一下时间片控制的细节。任务在执行时，tickAnnounce 函数将增加时间片的计数（由 usrClock 调用 tickAnnounce，usrClock 为用户定义的系统时钟中断处理函数）。当计数值达到 kernelTimeSlice 的设定值时，任务退出运行，该计数器清零。当任务刚转为 ready 状态时，该计数器初始值为 0。如果正在时间片内运行的任务被高级任务抢占，该计数器不会改变，当其再调入运行时，计算器在保存值上增加。如果禁止时间片循环调度，正执行任务的计数器则不会增加，也不起作用。

● 任务上下文切换

当一个任务停止运行，另一个任务开始运行时出现上下文切换。当更高优先级任务处于 ready 状态时，上下文切换让高级任务运行。任务可以由执行任务、中断和时钟超时激活处于 ready 状态。当执行任务执行系统调用阻塞时，也会出现上下文切换。悬挂任务的 CPU 寄存器会保存在 TCB 中，准备运行任务的 CPU 寄存器会从 TCB 中恢复。

● 调度控制

VxWorks 任务调度控制函数如表 4-3 所示。

表 4-3 任务调度控制函数

kernelTimeSlice	设置内核调度的时间片大小，控制同级时间片循环调度
taskPrioritySet	改变任务的优先级
taskLock	禁止任务调度。调用任务独占 CPU，不能在中断处理中调用
taskUnlock	允许任务调度

调用 taskLock 的任务会一直执行，而不会被其他任务（除了中断，当然也可以锁中断）抢占，除非任务自己调用系统函数被阻塞，另一任务才能被调入运行。而该任务再恢复运行，仍不可抢占，禁止任务调度。taskLock 可以用来完成互斥操作，但是抢占闭锁的时间应该尽量短，就和中断处理的原则一样，以保证系统实时性。

除了这几个与任务调度直接相关的控制函数，其他的系统调用也会影响系统的调度。如信号量获取调用可能会阻塞当前任务，信号量释放调用可能会激活高级任务抢占。

4.1.8 任务函数库

函数库 taskLib 包含了任务创建、删除、启动和停止等功能的内核函数。

● 创建和启动任务

```
int taskSpawn (name, priority, options, stackSize, entryPt, arg0, ... , arg9)
```

taskSpawn 的参数说明如表 4-4 所示。

表 4-4 任务创建函数参数

name	字符串任务名称
priority	任务优先级，从 0（最高）到 255（最低）
options	任务选项指定任务是否使用浮点，是否断点有效等
stackSize	任务栈大小，创建后就不再变化，可以用 Browser 监测栈的使用
entryPt	任务将执行的函数
arg0...arg9	传给 entryPt 函数 10 个 4 字节参数，将参数强制转换为整数类型避免编译错误

taskSpawn 函数返回任务标识 ID，其他 taskLib 函数使用标识 ID 来引用任务。任务 ID 其实是指向该任务控制块[TCB]的指针。

entryPt 是创建任务的主函数调用。任务在主函数中一般都有一个无限循环，因为大多数应用产品都是上电后一直永远运行。主函数的形式如下：

```
int main_loop_1( int arg1, int arg2 )
{
    FOREVER /* vxWorks.h 宏定义为 for(;;)* /
    {
        /* 等待数据到来 */
        taskDelay() or semTake() or msgQReceive()
        process_data();
    }
    return( ERROR ); /* 循环结束后才会到这儿 */
}
```


主函数名不能使用“main”，否则会 and 系统冲突。任务使用主函数指针为入口指针，同一个函数可以作为多个任务的入口函数，但是要注意函数必须是可以重入的。

主函数从 taskSpawn 可以接收最多 10 个 4 字节参数，不过不用完全使用，可根据需要选择参数个数。

● 任务延时

```
STATUS taskDelay (ticks)
```

任务调用 taskDelay 函数使自己退出运行，进入延时状态。延时长度由 ticks 参数指定。系统时钟单位[tick]的具体时间间隔由 sysClkRateSet 函数设定。

 WindRiver, “BSP Reference” 的 sysLib 的 sysClkRateSet () 帮助。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 taskLib 条目


4.1.9 任务功能扩展

支持不可预见的内核扩展与功能的可配置性是同样重要的。简单的内核接口和互斥方法使内核功能扩展相当容易。在某些情况下，应用可以利用内核钩子[Hook]函数来实现特定的扩展。

为了不修改内核而能够向系统增加额外的任务相关的功能，Wind 提供了任务创建、切换和删除的钩子函数。在任务创建、切换和删除时，可以调用扩展功能函数。任务控制块[TCB]有空闲区保留，可用于实现应用扩展。

要注意用户定义的任务切换钩子函数是在内核上下文中执行，因此不能调用与 VxWorks 内核有关的库函数。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 taskHookLib 条目。

 WindRiver, “VxWorks 5.4 Programmer’s Guide” 的 2.3.5 章节。

4. 1. 10 系统任务

VxWorks 有几个系统任务，用于实现最基本的功能，包括系统初始化、信息记录、任务异常处理和调试等，具体如表 4-5 所示。

表 4-5 VxWorks 基本系统任务

任务名称	说明
tUsrRoot	根任务，内核执行的第一个任务，任务函数为 usrRoot，在 xxxConfig.c 中定义，创建其他系统任务，调用应用初始化等，执行完被删除
tLogTask	系统信息记录任务，避免进行 IO 调用
tExcTask	异常处理任务，拥有最高优先级，不能被悬挂、删除和更改优先级
tWdbTask	目标代理任务，与 Target Server 一起实现交叉调试
Idle	系统空闲任务，优先级最低，没有其他任务准备好时，消耗 CPU 时间

还有几个可选组件的任务，由参数配置激活，具体如表 4-6 所示。

表 4-6 VxWorks 可选组件任务

任务名称	说明
tNetTask	网络任务
tShell	目标机 Shell 任务，Shell 调用命令一般在 tShell 上下文中执行
tTelnetd	Telnet 服务任务，用户登录后还会创建 tTelnetInTask 和 tTelnetOutTask
tFtpdTask	支持 FTP 服务器的任务
tTffsPTask	支持 TFFS 文件系统的任务

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 2.3.10 章节。

4. 2 中断处理

除了任务外，中断处理是另外的一个执行线程单元。中断处理不归属于操作系统内核调度，而由 CPU 硬件相关机制异步触发。中断处理是整个运行系统中优先级最高的代码，可以抢占任何任务级代码运行。某些 CPU 还支持中断分级，高级中断可以抢占低级中断运行。中断机制是多任务环境运行的基础，是系统实时性的保证。几乎所有的实时多任务操作系统都需要一个周期性系统时钟中断支持，用以完成时间片调度和延时处理，如 VxWorks 提供的 tickAnnounce 函数，由系统时钟中断调用，给予内核周期单位时间触发。

4. 2. 1 中断封装



硬件中断发生时，代码运行的上下文会发生切换，在进入中断处理前，需要保存当前运行的上下文，如程序计数器（PC）等。一些无 RTOS 的单片机系统，这些工作由硬件和编译

器共同完成。向量表在编译完成后就填充完成，再写入存储器中，系统运行时不能修改向量表来重新绑定中断入口函数。在 VxWorks 系统中，除了需要保存通常的寄存器环境外，还需要完成栈切换等；另外还要求中断入口运行时绑定，平台移植性，中断嵌套等，所以 VxWorks 本身也参与中断封装的管理。

VxWorks 提供 intLib 和 intArchLib，excLib 和 excArchLib 库用于中断的上层管理。用户常用的函数如表 4-7 所示。

表 4-7 中断相关函数	
函数	说明
intConnect	将 C 函数绑定到中断向量上
intLock	禁止中断
intUnlock	重新允许中断

完整的函数列表参考如下文档说明。

-  WindRiver, “VxWorks 5.4 Programmer's Guide” 的 2.5 章节。
-  WindRiver, “VxWorks 5.4 Reference Manual” 的 “intLib”, “intArchLib” 条目。

VxWorks 封装中断的伪代码如下所示：

```
* 00 e8 kk kk kk kk    call _intEnt          * 告诉内核
* 05 50                pushl    %eax          * 保存寄存器
* 06 52                pushl    %edx
* 07 51                pushl    %ecx
* 08 68 pp pp pp pp    pushl    $_parameterBoi * push BOI param
* 13 e8 rr rr rr rr    call _routineBoi      * call BOI routine
* 18 68 pp pp pp pp    pushl    $_parameter   * 传中断入口参数
* 23 e8 rr rr rr rr    call _routine         * 调用中断处理 C 函数
* 28 68 pp pp pp pp    pushl    $_parameterEoi * push EOI param
* 33 e8 rr rr rr rr    call _routineEoi      * call EOI routine
* 38 83 c4 0c          addl $12, %esp        * pop param
* 41 59                popl %ecx             * 恢复寄存器
* 42 5a                popl %edx
* 43 58                popl %eax
* 44 e9 kk kk kk kk    jmp      _intExit     * 通过内核退出
```

用 intConnect 函数关联一个新中断函数时，VxWorks 会从系统堆中分配一段内存，不经过编译器，硬编码将这段封装代码填入该内存区，更换新函数参数和入口，并将该内存起始地址和中断向量表关联。硬件中断发生时，就会调转执行这段代码。

VxWorks 将向量表定位在 RAM 中，基地址为 VEC_BASE_ADRS，在 configAll.h 中定义，一般为 RAM 存储器的起始地址。有的 CPU 向量表不能重定位，只能位于指定地址。在系统启动初始化时，由 intVecBaseSet 函数设置向量表的基地址（BootRom 在 usrInit 中调用；VxWorks 在 usrInit→sysStart 中调用）。向量单项的格式和具体 CPU 有关，包含封装代码内存的起始地址，中断发生时由硬件机制跳转执行。

向量表基地址设定后，需要进行初始化，来指定中断的缺省处理函数，如除 0 错误等。这个工作由 excArchLib 库中的 excVecInit 函数来完成，在 usrInit 中调用。VxWorks 在映象的代码段建立了一张中断缺省函数表，将各中断指向 excStub 或 excIntStub。excStub 为缺省异常处理，excIntStub 为缺省中断处理。函数 excVecInit 根据这张表来设置向量表的各中断向量

值。需初始化的向量表范围由 LOW_VEC 和 HIGH_VEC 确定，在 excArchLib.h 中定义。

为了中断的快速响应，VxWorks 的中断处理在一个特殊上下文中执行，没有任务控制块，在所有任务上下文之外。所以中断处理时不会进行任务上下文切换。

如果硬件允许，所有中断处理都共享同一个中断栈。中断栈应该足够大，以应付最坏情况的中断嵌套。有些硬件不允许使用独立的中断栈，中断处理只能使用被中断的任务的栈空间，这种情况下应该增加各任务的栈大小。

为了保护临界代码不被中断处理打断，可以使用 intLock 和 intUnLock 函数来实现。用 intLock 闭锁中断后，当前的执行线程可一直继续，任何任务和中断都不会插入执行，直到该线程主动用 intUnLock 解锁中断，才恢复允许中断处理和任务调度。注意不要在中断闭锁期间调用 VxWorks 系统函数，否则有可能意外打开中断闭锁，违反临界代码的设计意图。有些 CPU 中断是分级，可以用 intLockLevelSet 和 intLockLevelGet 函数来操作中断闭锁的级别，缺省闭锁所有等级的中断。中断闭锁对系统的实时性有很大的影响，在解决执行代码和中断处理互斥问题时才应使用，并且应使闭锁时间尽量短。如果只是任务间的互斥问题，可以使用 taskLock 和 taskUnLock 来解决。还要注意，intLock 不能屏蔽调度，如果在中断闭锁代码区使用系统调用，可能出现任务调度。

4.2.2 系统时钟

VxWorks 系统中必须激活一个时钟中断，来调用 tickAnnounce 函数，作为操作系统内核的时基。VxWorks 中已经为用户做了这部分工作，提供 usrClock 函数作为系统时钟中断处理函数，来调用 tickAnnounce，在 bootConfig.c 或 sysClkInit.c 中定义。并提供了中断连接、频率设置和中断允许等初始化函数，在 “\target\src\drv\timer\xxxxTimer.c” 中定义，具体文件和硬件情况相关。该文件包含在 sysLib.c 中编译。如果需要选择另外的硬件定时器作为系统时钟，需要修改该文件，一般用缺省的就行。usrClock 函数地址不直接填入中断封装代码，而是经由 sysClkInt 调用。在 sysLib.c 的 sysHwInit2 中调用 intConnect 将 sysClkInt 绑定到相应的时钟中断上。系统时钟中断函数也可传入参数，但是不经过中断封装代码，而由系统时钟实现完成。

时钟中断频率由 SYS_CLK_RATE 指定，在 configAll.h 中定义，可以在 config.h 或工程中重新指定，缺省值为 60，笔者一般使用 100。VxWorks 使用的时间单位为 tick，如 taskDelay 中参数的单位，一个时钟中断一个 tick，具体时间值由前面设置的频率确定。在编程时，一般不直接使用 tick 单位，而是经过如下转换：

```
#define TICK2MSE (1000/SYS_CLK_RATE) /*每 tick 的毫秒数*/
taskDelay (1000/ TICK2MSE);          /*延时 1 秒*/
```

系统时钟的频率不能太高。高频率会使内核调度用时比率偏高，而实际任务可用的 CPU 时间下降，使整个系统运行效率下降或不可用。如果用户需要高分辨率的定时，可以使用时间戳[Timestamp]、辅助时钟，或自己维护一个高频率时钟中断。

● 时间戳

时间戳[Timestamp]依附于系统定时器，在 “\target\src\drv\timer\xxxxTimer.c” 中实现。Timestamp 机制用查询方式取得当前定时器的硬件计数值，去除了中断处理负荷，一般能获得

得比系统时钟高几十倍的分辨率，这取决于一个 tick 包含多少个硬件计数。Timestamp 的函数接口一般在“BSP Reference”中描述，不过有些 Tornado 版本中没有，更直接的方法就是阅读其实现源代码。用户常用的函数接口如表 4-8 所示。

表 4-8 时间戳函数

函数	说明
sysTimestamp	取得定时器当前的硬件计数值
sysTimestampFreq	取得硬件计数的频率

Timestamp 机制在取得高精度时刻方面特别有用，比如评估一段代码的执行时间。

```
old_count = sysTimestamp();
/*被评估代码段*/
new_count = sysTimestamp();
used_time = (new_count-old_count)/ sysTimestampFreq();
```

上面只是个简单示例，没考虑硬件计数回绕的问题。由于取得的是硬件原始计数，累加溢出后，又会从零开始计数，所以需要用户根据硬件的实际情况处理计数值回绕的问题。

VxWorks 系统组件中，只有 WindView 使用了 Timestamp 来获取高分辨率时标。

● 定时器

定时器[Watchdog]是系统时钟中断处理线程的延伸，或者说是实现了系统时钟中断的 Hook 功能。Watchdog 机制允许延时执行一个指定函数。该函数一般在系统时钟中断上下文中进行，而和创建或启动该定时器的任务无关。如果被打断的是中断或内核代码，该函数不会立即执行，而放入 tExcTask 工作队列，会在 tExcTask（任务优先级为最高 0 级）上下文中执行。

wdLib 库中提供如下函数接口供用户使用，如表 4-9 所示。

表 4-9 WatchDog 函数

函数	说明
wdCreate	创建并初始化 Watchdog 定时器，但未启动
wdStart	启动一个已创建的 Watchdog 定时器，时间单位为 tick
wdDelete	停止并释放 Watchdog 定时器
wdCancel	停止正运行的 Watchdog 定时器

定时器由 wdStart 函数启动，每启动一次，都执行一次指定函数，而不会多次执行。如果需要循环触发，需要在延时执行函数中用 wdStart 再次启动定时器。用这种方法可以实现 pSOS 中 tmEvEvery 类似功能，Watchdog 定时器也类似 pSOS 中 tmEvAfter 的功能，只不过 Watchdog 延时触发的是函数，而 tmEvAfter 则是延时发送事件标志。

延时执行函数的代码应该遵循中断处理代码的编写规则。可以使用 wdShow 库中提供的 wdShow 函数查看 Watchdog 的活动和关联的函数。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 2.6 章节。

 WindRiver, “VxWorks 5.4 Reference Manual” 的“wdLib”，“wdShow” 条目。

4.2.3 辅助时钟

辅助时钟是另外一种获得更高分辨率时钟的机制。它和 Timestamp 的实现思想不太相同，启用一个与系统时钟不同的硬件定时器，挂接用户自己的中断处理函数,去掉内核驱动的负

荷，时钟分辨率的高低取决于硬件定时器的精度和用户中断函数的长短。

辅助时钟在“\target\src\drv\timer\xxxxTimer.c”中实现，结构和系统时钟实现类似。如果需要选择另外的硬件定时器作为辅助时钟，需要修改该文件，一般用缺省的即可。提供与系统时钟类似接口函数供用户使用，可以参考系统时钟相关函数的使用，如表 4-10 所示。

表 4-10 辅助时钟函数

函数	说明
SysAuxClkConnect	将用户处理函数连接到辅助时钟中断上
SysAuxClkRateSet	设置辅助时钟中断频率
SysAuxClkEnable	使能启动辅助时钟

其他的函数接口，可以参考实现源代码文件。


有些 Tornado 版本中配置选项可能有些问题。工程中有配置选项 INCLUDE_AUX_CLK，与实现文件中编译条件 INCLUDE_XXXX_AUX_CLK 不同，虽然在 config.h 中定义了 INCLUDE_XXXX_AUX_CLK，将辅助时钟组件包含在工程中，但在工程组件配置窗口却显示没有包含该组件。在窗口中是否配置该组件，对辅助时钟实现代码的包含都没有影响。但只要明白下层实现原理，实际工作应该没有问题。

4.2.4 代码限制

中断处理函数（包括 Watchdog 触发函数）在代码编写方面有些限制。由于中断处理函数不是在通常的任务上下文中运行，没有任务控制块，所有中断处理共享同一栈空间，所以中断处理函数不能调用会引起阻塞的系统函数，如取信号量，内存操作，IO 操作，硬浮点处理等。可使用的函数调用列表，可参考手册的相关章节。

最可能遇到的问题就是在中断处理函数中错误地使用 printf。因为 printf 涉及到 IO 操作。VxWorks 为解决这种情况，专门提供了 Logging 机制，将输出信息相关的 IO 操作转移到 Logging 任务上下文中执行。

前面讲过中断闭锁可以分级别进行，因此可能出现不可闭锁的高级中断。它们的代码编写有更严格的限制。除了上面的限制外，还必须使用 intVectSet 绑定中断处理函数，不能使用 VxWorks 中需要中断闭锁的函数调用。

 WindRiver，“VxWorks 5.4 Programmer's Guide”的 2.5.3 和 2.5.6 章节。

4.2.5 中断通知

为了系统的实时性，要求中断处理函数尽量短小，中断事件的进一步处理可通过任务间通信机制延迟到任务上下文中执行，类似 Linux 中的底半处理机制。这样也可避开中断代码编写的种种限制。

- ✧ 共享数据结构：驱动程序最常用循环缓冲区共享。
- ✧ 信号量：ISR 中可以释放信号量。
- ✧ 消息队列：ISR 中可以向消息队列发送消息，如果队列满，消息被丢弃。
- ✧ 管道：ISR 中可以向管道中写入消息，如果管道满，消息被丢弃。
- ✧ 信号：ISR 中可以向任务发送信号，异步调度信号处理函数。

4.2.6 用户中断

只要遵循上面描述的中断代码编写规范，任何 C 函数都可以作为用户中断处理函数，而不用添加编译器相关的预处理指令。C 函数编写完成后，通过 `intConnect` 将其与中断关联。

```
STATUS intConnect(VOIDFUNCPTR * vector, VOIDFUNCPTR routine, int parameter)
```

其中 `vector` 为相关中断向量在向量表中的偏移地址，`routine` 为 C 函数指针，`parameter` 为 C 函数的传入参数。中断连接后，该参数保存在中断封装代码中。中断函数只有一个入口参数，一般传入变量指针。或传入中断号以区分进入该函数的中断和索引各自的共享内存，使该中断函数可以被多个同类中断复用。这种方法在设计某些驱动程序时有用，比如支持多个通信通道的串口芯片 16C55x 的驱动，代码如下所示：

```
static void v16C55x_int(UINT8 int_no){.....}
#if ( (TYPE_CPU&TYPE_MASK) == CPU_X86 ) /*PC104\486 硬件平台*/
    intConnect( INUM_TO_IVEC(INT_VEC_GET(g_Chan[phy_no].int_no)),
               v16C55x_int, g_Chan[phy_no].int_no);
#endif
#if ( (TYPE_CPU&TYPE_MASK) == CPU_5272) /*MCF5272 硬件平台*/
    intConnect( INUM_TO_IVEC(g_Chan[phy_no].int_no),
               v16C55x_int, g_Chan[phy_no].int_no);
#endif
```

中断处理函数 `v16C55x_int` 传入中断向量号 `int_no`，以区分各串行通道的中断处理。一般用户可见的都是中断号，而 `intConnect` 要求传入中断向量偏移，这需要进行转换操作，VxWorks 提供相关的几个宏，如 `ivColdfire.h` 中定义的宏：

```
/*将中断向量转换为中断号*/
#define IVEC_TO_INUM(intVec)      ((int) (intVec) >> 2)
/*将中断号转换为中断向量*/
#define INUM_TO_IVEC(intNum)      ((VOIDFUNCPTR *) ((intNum) << 2))
/*将 trap 号转换为中断向量*/
#define TRAPNUM_TO_IVEC(trapNum) INUM_TO_IVEC (32 + trapNum)
```

4.3 异常处理

程序中的代码和数据错误会引起硬件异常，如非法指令、总线或地址错误、除零错误等。VxWorks 提供几个库来实现异常处理相关机制，包括异常处理库、中断处理库、`logLib`、`sigLib` 和 `dbgLib` 等。

- ✧ 异常处理库提供向量表初始化，异常处理任务创建，异常 Hook 添加等函数，为所有异常提供缺省的处理函数。
- ✧ 中断处理库能为外部中断绑定用户自定义处理函数，替代缺省的异常处理函数。
- ✧ `logLib` 提供一种非阻塞式输出方式，用于异常信息输出。
- ✧ `sigLib` 用于特定任务的异常处理。
- ✧ `dbgLib` 使用 `excLib` 中的机制，以支持断点、单步等异常处理功能。

4.3.1 异常处理库

异常处理库提供异常处理机制，和中断处理密切相关。异常和中断属于同一概念，但也有些细微差别。异常和 CPU 相关，如非法指令、总线错误、地址错误和除零错误等；中断和外部设备相关，完成 CPU 和设备间异步操作。有的 CPU 还有软件陷阱的概念，陷阱可以由

软件指令有意识地触发，有些操作系统通过陷阱来实现系统调用，如 pSOS，Linux 等，而 VxWorks 的系统调用采用直接函数调用形式。在 VxWorks 中，无论是异常、中断还是陷阱，都由同一张向量表指引处理。在向量表建立时，异常处理库将所有的向量都指向缺省的处理函数，异常类向量指向 exeStub，中断和陷阱向量指向 exeIntStub。随着用户指定处理函数的挂入，缺省向量处理由各中断封装代码替换（参照前面中断处理的描述）。异常处理库还提供异常处理任务 tExcTask 用于支持异常处理。

异常处理库包括两部分：excArchLib 和 excLib。excArchLib 实现和体系结构相关的异常处理操作，如向量表初始化等。excLib 实现上层异常处理操作，如创建异常处理任务，异常 Hook 处理等。

excArchLib 提供 excVecInit 函数来初始化向量表，使各向量指向缺省的处理函数，用来安全捕捉和报告由程序错误和意外硬件中断引起的异常。在中断允许前，由 usrInit 调用。

excLib 提供 excInit 函数来创建异常支持任务 tExcTask，任务入口为 excTask。系统初始化调用 excInit 后，由程序错误引起的硬件异常可被安全地捕捉和报告，未初始化硬件中断则会报告后被抛弃。tExcTask 任务用于处理任务级异常，能执行在中断上下文无法进行的函数，如某些情况下的 Watchdog 关联函数。在系统中，该任务拥有最高优先级，不能被挂起、删除和改变优先级。

excLib 中 excHookAdd 函数指定异常的 Hook 函数。当硬件异常发生时，先执行正常的异常处理，再调用该 Hook 函数。该函数通常由 dbgLib 使用来激活自己的异常处理机制。用户也可以用它来替换自己的 Hook 函数，如 BootRom 的 bootConfig.c 中的使用。

```
excHookAdd ((FUNCPTR) bootExcHandler);
/*****
* bootExcHandler - bootrom 异常处理函数，传入异常任务 ID
*/
LOCAL void bootExcHandler ( int tid )
{
    REG_SET regSet;                /*存放任务寄存器*/
    /*取被跟踪任务的寄存器*/
    if (taskRegsGet (tid, &regSet) != ERROR) {
        trcStack (&regSet, (FUNCPTR) NULL, tid); /*显示任务栈中函数的调用轨迹*/
        taskRegsShow (tid);                /*显示任务寄存器*/
    }
    else    printf ("bootExcHandler: exception caught but no valid task.\n");

    taskDelay (sysClkRateGet ()); /*延时 1 秒*/
    reboot (BOOT_NO_AUTOBOOT); /*重新启动后不自启动，停留在 bootRom shell*/
}
```

BootRom 下不使用 dbgLib，用 bootExcHandler 作为 Hook 函数，显示异常任务寄存器后重新启动系统。

当程序发生错误时，发生错误的执行任务被挂起，保存任务异常点的状态，并在标准输出上显示该异常的描述。而 VxWorks 内核和其他系统任务可以继续执行。开发人员可以介入，调试被挂起任务，如用 ti 查看任务信息；用 tt 查看任务栈轨迹；用 tr 尝试恢复任务运行；用 td 删除任务等。

在中断处理函数出现异常时，没有上下文用于挂起。VxWorks 将异常描述存储在内存中

的特定位置并重新启动系统。BootRom 启动时会探测该位置内存中是否有异常描述，如果有则输出描述信息。


当未初始化中断发生时，仅会显示中断描述，而不会改变系统的运行状态。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 “excArchLib”, “excLib” 条目。

4.3.2 异常信号

通过信号[signal]机制，任务也可以为硬件异常绑定自己的处理函数。如果任务为异常提供了信号处理函数，缺省的异常处理不再被执行。信号也可以用于软件异常通知。接受到信号的异常任务被立即挂起，当再次被调度时，会运行相应的信号处理函数。函数返回后，该任务会被挂起，所以在函数结束处可以执行 `exit` 结束任务，或用 `taskRestart` 重启任务，或用 `longjmp` 执行函数跳转。

`sigLib` 提供 `sigvec` 函数来为硬件异常绑定处理函数。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 “sigLib” 条目。

4.4 线程间通信

VxWorks 提供灵活丰富的线程间通信机制，用于协调任务间，中断和任务间的行为，包括共享内存、信号量、消息队列和管道等。大部分机制类似 Unix，也提供 POSIX 兼容的接口函数。本节只介绍常用的几种通信方法。

4.4.1 共享数据结构

VxWorks 使用扁平[flat]内存模式，整个系统共享统一地址空间。地址空间对所有线程都是可见的，使得这种共享数据结构或内存的通信方式自然存在。这种方式非常方便快捷，但和多任务概念是冲突的，会使任务间的耦合过紧，而增加系统的复杂程度。另外因为多任务调度，各线程对共享内存的访问可能会出现冲突，所以实际编程中多用信号量来保护共享内存资源。对于临界资源，应使用互斥型信号量来保护。原则上，应该尽量少使用这种通信方式，除非设计必要并且实现方法清晰。

4.4.2 信号量

信号量机制是许多操作系统的标准通信机制，用于实现资源互斥和任务同步。

VxWorks 提供 3 种信号量：二进制[Binary]（同步）信号量、计数信号量和互斥[Mutex]信号量。多种信号量设计是为了解决相关的编程问题。除了这些 VxWorks 专有信号量外，还提供 POSIX 兼容信号量接口。信号量操作对应的目标库如表 4-11 所示。

表 4-11 信号量函数库

库	说明
<code>semLib</code>	通用信号量库
<code>semBLib</code>	二进制信号量库
<code>semCLib</code>	计数信号量库
<code>semMLib</code>	互斥信号量库
<code>semOlib</code>	VxWorks 4.x 二进制信号库，用于向后兼容
<code>semPxLib</code>	POSIX 1003.1b 信号量库

semPxShow	提供 POSIX 信号量查看函数
semShow	提供信号量查看函数
semSmLib	共享内存信号量库 (VxMP 可选产品)

本节主要讲述 3 种基本的 VxWorks 信号量。它们有类似的操作接口，但 3 种类型的信号量的创建函数略有不同，由各自对应的函数库提供。

```
STATUS semBCreate (int options, SEM_B_STATE initialState);
STATUS semMCreate (int options);
STATUS semCCreate (int options, int initialCount);
```

创建函数都返回一个指针类型 SEM_ID，指向信号量结构，作为信号量操作的标示符，在 semLib.h 中定义。

```
typedef struct semaphore *SEM_ID;
```

创建函数的 options 参数类似，有如下位选项在 semLib.h 定义，最后两项由互斥信号量专用。

```
#define SEM_Q_MASK      0x3 /* 等待任务队列方式屏蔽码 */
#define SEM_Q_FIFO      0x0 /* 等待任务按先进先出方式排列 */
#define SEM_Q_PRIORITY  0x1 /* 等待任务按优先级方式排列 */
#define SEM_DELETE_SAFE 0x4 /* 避免删除信号量占用任务（互斥信号量用）*/
/*允许优先级继承，必须配合选项 SEM_Q_PRIORITY 使用*/
#define SEM_INVERSION_SAFE 0x8 /* 避免优先级逆转问题（互斥信号量用）*/
```

创建函数的第 2 参数用于指定信号量的初始值。

二进制信号量的初始值 (initialState) 只能为 0 或 1，SEM_B_STATE 为两值枚举类型，在 semLib.h 中定义。

```
typedef enum {SEM_EMPTY, SEM_FULL} SEM_B_STATE;
```

互斥信号量用于控制临界资源访问，初始时都假设资源可用，初始值缺省都为 1，所以省略参数指定。

计数信号量用于控制资源的多实例访问，初始值 (initialCount) 为资源实例的个数。

创建了某种信号量后，不再区分信号量类型，都针对信号量标示符进行操作。统一操作接口由 semLib 库提供。

```
STATUS semTake (SEM_ID semId, int timeout); /*获取信号量*/
STATUS semGive (SEM_ID semId); /*释放信号量*/
STATUS semFlush (SEM_ID semId); /*解锁阻塞在该信号量上的所有任务*/
STATUS semDelete (SEM_ID semId); /*删除信号量*/
```

timeout 指定等待时间，以 tick 为单位。另外还有两个特殊值：WAIT_FOREVER (-1) 和 NO_WAIT (0)，在 vxWorks.h 中定义，通用于类似的 timeout 参数。

 WindRiver, “VxWorks 5.4 Reference Manual” 的相关库条目。

4.4.2.1 二进制信号量

二进制信号量也称为二位信号量，或开关信号量，可看作计数值为 1 的特殊计数信号量。可用于共享资源互斥和事件同步。由于 VxWorks 有专门的互斥信号量用于资源保护，二进制信号量多用于线程同步。

使用二进制信号量，任务在等待事件时处于阻塞状态，不用消耗 CPU 时间来查询事件状态。事件由中断或任务异步触发。使用时，先用 semBCreate 创建信号量，初始值设为 SEM_EMPTY (0)，以等待事件发生；用 semTake 来判断事件状态；用 semGive 来通知事件发生。

当调用 `semGive` 时，如果有任务在等待这个信号量，则只有该信号量任务等待队列的第一个任务恢复到 `ready` 状态，信号量不再可用；如果没有任务等待这个信号量，则信号量转为可用状态。如果信号量已处于可用状态，再调用 `semGive` 则不起作用。

当调用 `semFlush` 时，所有等待该信号量的任务都恢复到 `ready` 状态，但信号量的状态不变化。

调用 `semTake` 时，若信号量可用，调用立即返回，任务不会阻塞而继续运行，信号量变为不可用；若信号量不可用，任务就会阻塞在调用处等待，当信号量可用，任务变为 `ready`，`semTake` 调用返回 `OK`，任务继续运行（任务本身感觉不到这个停顿）。也可能信号量一直不可用，直到超时，`semTake` 调用则返回 `ERROR`。

下面代码段来自 VxWorks 库参考手册，说明中断和任务同步中二进制信号量的使用。

```
SEM_ID semSync;                /*信号量的标识符*/
init ()
{
    intConnect (... , eventInterruptSvcRout, ...);
    /*创建二进制信号量，等待任务按 FIFO 排列，初始值为 0 以等待事件发生*/
    semSync = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    taskSpawn (... , task1);
}
task1 ()                        /*事件处理任务*/
{
    ...
    semTake (semSync, WAIT_FOREVER); /*等待事件发生*/
    ...                               /*处理事件*/
}
eventInterruptSvcRout ()        /*事件通知中断函数*/
{
    ...
    semGive (semSync);          /*通知事件发生*/
    ...
}
```

二进制信号量在事件处理方面有些缺点，比如不能等待多事件处理。另外信号量作为一个内核对象独立于任务存在，相比于直接与任务 TCB 关联的事件，效率有所降低。大多数 RTOS 都提供事件处理机制，如 pSOS, Nucleus 等，pSOS 的事件直接归属于任务 TCB，而 Nucleus 中则以独立事件对象存在。由这些操作系统向 VxWorks 移植时，事件处理机制的变化是一个常见问题。如果想遵循自己习惯的编程范型，可能需要封装信号量为自己熟悉的事件处理接口。不过幸运的是，新版的 VxWorks 5.5 开始提供专门的事件处理机制。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 “semBLib” 条目。

4.4.2.2 互斥信号量

当多个任务分享一个公共资源（如数据结构、文件和硬件等）时，可以使用互斥信号量，以防止多个任务同时访问临界资源。

使用时，先用 `semMCreate` 为资源创建互斥信号量，初始值缺省为 `SEM_FULL (1)`，表示资源可用；任务使用资源前，先调用 `semTake` 以获得使用权；任务使用完资源后，应调用 `semGive` 放弃使用权。

二进制信号量和计数信号量也可以用于互斥操作，而互斥信号量是专门设计用来解决互斥中出现的 3 个常见问题。

- ✧ 互斥信号量可以被嵌套获取。也就是拥有该信号量的任务可以多次调用 `semTake`，而不会被阻塞。但放弃使用权时，任务必须调用同样数目的 `semGive`。
- ✧ 当任务 `tA` 拥有删除安全[`delete-safe`]互斥信号量时，任务 `tB` 想删除信号量时会被阻塞，直到 `tA` 放弃该信号量。因此 `tA` 正在操作资源时是安全的。用 `SEM_DELETE_SAFE` 选项创建这种信号量。
- ✧ 反转安全[`inversion-safe`]互斥信号量提供优先级继承属性，能防止所谓的优先级反转问题。拥有该信号量的任务会临时提升优先级，级别与等待该信号量任务中的最高级别相同，放弃信号量后优先级则复原。这可以防止如下情况出现：高优先级任务等待低优先级任务拥有的信号量；而低任务又被中优先级任务抢占，不能执行信号量释放。用“`SEM_Q_PRIORITY | SEM_INVERSION_SAFE`”组合选项创建这种信号量。

互斥信号量是一种特殊的二进制信号量，两者基本行为相同，但有如下限制：只能用于互斥操作，而不能作为同步机制；只能由获取信号量的任务来释放它；不能在中断上下文中使用；不能使用 `semFlush` 函数。

 WindRiver，“VxWorks 5.4 Reference Manual”的“`semMLib`”条目。

4.2.2.3 计数信号量

计数信号量是信号量的一般形式，用于控制多实例资源的访问，比如车辆中乘客总人数的控制。对于事件处理，二进制信号量知道事件是否发生，但是不知道事件发生的次数。而计数信号量可以记录事件发生的次数。

当用 `semGive` 操作计数信号量时，如果信号量上没有等待任务，计数加 1；如果有等待任务，队列头上的任务转为 `ready` 状态，而信号量计数维持为 0。

当用 `semTake` 操作计数信号量时，如果计数大于 0，`semTake` 立即返回 `OK`，任务不会阻塞而继续运行，计数值减 1；如果计数为 0，任务会阻塞在 `semTake` 调用上。

 WindRiver，“VxWorks 5.4 Reference Manual”的“`semCLib`”条目。

4.4.3 消息队列

便捷的信号量能解决很多任务间协调问题，但是交换的信息有限，而共享内存方式又不标准，消息队列作为一种折中方式用于线程间信息交换。消息队列可以由多个任务读写，交换信息的基本单位为消息，消息队列中可以缓冲多个消息单元，各消息单元可以有不同长度。消息本身为无格式的纯数据，不带目的任务指向和触发，因此应用一般需要规定消息格式，或者绑定额外信号量来使用。

VxWorks 提供两种消息队列：Wind 型和 POSIX 型。Wind 消息队列专门为 VxWorks 设计。POSIX 消息队列用于兼容 POSIX 实时扩展标准 1003.1b，可以方便移植到遵循 POSIX 标准的操作系统上。两种类型的消息队列有很多不同，如表 4-12 所示。

表 4-12 消息队列比较

特性	Wind 消息队列	POSIX 消息队列
消息优先级数目	1	32
阻塞任务队列	FIFO 或基于优先级	基于优先级
接受超时	可选	无

任务通知	无	可选（一个任务）
Close/Unlink Semantics	无	有

消息队列相关的函数库如表 4-13 所示。

表 4-13 消息队列函数库

库	说明
msgQLib	Wind 消息队列库
msgQShow	Wind 消息队列查看函数库
mqPxLib	POSIX 消息队列库
mqPxShow	POSIX 消息队列查看函数库
msgQSmLib	共享内存消息队列库（VxMP 可选产品）

本节只介绍 Wind 消息队列，POSIX 消息队列读者可以参考相关手册的说明。

库 msgQLib 为消息队列提供如下函数接口：

`MSG_Q_ID msgQCreate (int maxMsgs, int maxMsgLength, int options)`

msgQCreate 函数用于创建消息队列。函数返回消息队列标识符，为一个指向结构的指针，MSG_Q_ID 在 msgQLib.h 中定义。maxMags 指定队列中允许的最大消息个数，maxMsgLength 为最长消息的字节数，这两个参数确定分配内存的大小。options 指明等待任务的排列方式，可为 MSG_Q_FIFO 或 MSG_Q_PRIORITY。

`STATUS msgQDelete (MSG_Q_ID msgQId)`

`STATUS msgQSend (MSG_Q_ID msgQId, char *buffer, UINT nBytes, int timeout, int priority)`

msgQSend 函数用于消息队列传送。参数 buffer 为消息数据缓冲区地址。nBytes 为传送消息的长度。参数 timeout 指定消息队列满情况下任务等待延时，与其他函数的该参数类似。priority 指定消息的级别为 MSG_PRI_NORMAL 或 MSG_PRI_URGENT。前者表示为平常消息，新消息添加到队列的尾部；后者表示紧急消息，新消息添加到队列的头部，被优先处理。

`int msgQReceive (MSG_Q_ID msgQId, char *buffer, UINT maxNBytes, int timeout)`

msgQReceive 函数用于从消息队列中读取消息。函数返回读取消息的长度。参数 buffer 为接收缓冲区指针。参数 maxNBytes 为接收缓冲区的大小。参数 timeout 为消息队列空情况下任务等待延时。

`int msgQNumMsgs (MSG_Q_ID msgQId)`

msgQNumMsgs 函数读取消息队列中消息个数。函数返回消息的个数。

下面代码段来自 VxWorks 库参考手册，说明消息队列的使用。

```
#include "vxWorks.h"
#include "msgQLib.h"

#define MAX_MSGS    (10)    /*最大消息个数*/
#define MAX_MSG_LEN (100)  /*最长消息字节数*/
#define MESSAGE "Greetings from Task 1" /*消息内容*/

MSG_Q_ID myMsgQId;          /*消息队列标识符*/
/*任务 1: 创建消息队列 myMsgQId, 并向任务 t2 发送一个消息*/
task1 (void)
{
    /*创建消息队列，等待任务按优先级排列*/
    if((myMsgQId = msgQCreate(MAX_MSGS, MAX_MSG_LEN, MSG_Q_PRIORITY)) == NULL)
```

```

        return (ERROR);
    /*发送一个普通级消息；如果消息队列满则阻塞*/
    if (msgQSend (myMsgQId, MESSAGE, sizeof (MESSAGE), WAIT_FOREVER,
        MSG_PRI_NORMAL) == ERROR)
        return (ERROR);
}
/*任务 2: 从消息队列 myMsgQId 中读取消息，并显示消息内容*/
task2 (void)
{
    char msgBuf[MAX_MSG_LEN];    /*消息接受缓冲区*/
    /*等待从消息队列中读取消息，直到消息可用*/
    if (msgQReceive(myMsgQId, msgBuf, MAX_MSG_LEN, WAIT_FOREVER) == ERROR)
        return (ERROR);
    /*显示消息内容*/
    printf ("Message from task 1:\n%s\n", msgBuf);
}

```

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 2.4.4 章节。

 WindRiver, “VxWorks 5.4 Reference Manual” 的相关库条目。

4.5 常见问题解答

● Wind 和 POSIX 调度有哪些区别？

解答：Wind 的调度是基于任务的，而 POSIX 的调度是基于进程的。

Wind 调度用优先级抢占调度，而 POSIX 使用 FIFO。

Wind 在整个系统范围应用调度算法；而 POSIX 只在进程间应用调度算法。

Wind 优先级编号方案和 POSIX 的相反。

● 使用 taskSpawn 创建任务时，栈设置多大合适？

解答：这要看自己的任务需要多大的栈。

每次函数调用时会使用栈，将栈帧[frame]压入栈中。栈帧包括传给函数的参数、函数的返回值、函数返回地址等。所以要确定该任务中函数调用链的最大深度和传递参数的个数。

有时还需要考虑中断。某些处理器（如 x86）没有中断栈指针。因此，所有中断使用当前运行任务的栈。中断会随机在任意任务中发生，所以需要保证每个任务有足够的栈空间来满足自己的需要，以及最多中断嵌套的需要。

另外还可以使用 VxWorks 提供的辅助程序来确定任务栈的大小，比如 Tornado Browser。以及 Shell 中使用 checkStack 命令。它们会显示任务运行中使用栈的最大值。用户可以根据最大值来确定任务栈的大小，需要考虑冗余。另外需要注意，taskSpawn 创建时确定的栈大小不能在运行时再更改。

● 函数 kernelTimeSlice 使用什么值合适？

解答：主要看系统的需求，合适的值能提高实时性，满足各任务的时限[deadline]要求。若值为 0，会禁止同级任务间的时间片轮转调度。时间片的单位为 tick，所以提高系统时钟频率，可以使时间片更精准细微。但不能一味提高频率，需要考虑硬件的实际情况，太高的频率会导致系统调度占用时间[overhead]过多，而任务实际可用时间下降，降低系统效率。需要注意，

时间片只对同级任务调度起作用，不同级别的任务仍遵循优先级抢占调度。

● 为什么下面的 tWdbTask 任务的入口函数显示数值？

tShell	shell	294244	1	READY	9eab8	293e84	0	0
tWdbTask	95fba	29c410	3	PEND	4a44a	29c368	0	0

解答：这是因为 tWdbTask 任务的入口函数是静态的，不是全局的，所以符号表中不包含该函数的符号，只能显示函数地址。在 Shell 中输入该数值，只能获得临近该地址的符号。

类似的情况还出现在其他的显示函数中，如 iosDrvShow 显示的驱动函数入口。

● 如何改变 ISR 栈大小？

解答：如果用 Tornado 的工程配置 VxWorks 组件，需要修改 kernel 组件的 ISR_STACK_SIZE 参数。该参数的定义在 prjParams.h 中，会替换 configAll.h 中的相应定义。

有的硬件体系结构没有独立 ISR 栈存在，而是用任务栈完成中断处理，这时任务栈的设置就需要综合考虑任务和中断两者的需求。如 MC680[016]0、I960 和 I80X86 等。不过 I80X86 分配了一个假的中断栈，用于 checkStack，实际上并未使用。

● 如何知道 ISR 栈是否溢出？

解答：Tornado 的 Browser 中不显示中断栈信息，需要使用 checkStack。不过 checkStack 在任务级运行，显示当前中断栈的使用值总为 0，不过能看到历史最大使用值，可以用来判断中断栈大小是否设置合适。

● 知道信号量的 semid，如何取得占用[take]该信号量的任务的栈轨迹？

解答：如果不是计数类型信号量，可以取得相关信息。在“target/h/private/semLibP.h”中有信号量结构定义，其中有个联合[union]成员 state。对于计数信号量，state 用来保存信号量计数。而对于二进制和互斥信号量，state 用于保存占用任务的 ID，如果该信号量没有被占用，则值为 NULL。

所以可以先用 semShow 查看信号量 semid 的信息，获得任务 ID，再用 tt 查看该任务的栈轨迹。

如果想在程序中获得类似信息，就需要利用 semLibP.h 中定义的结构。信号量的 semid 实际上是指向该结构变量的指针，加上结构定义，可以方便取得占用任务的 ID。不过需要注意的是，private 目录中头文件定义的函数或结构，都不是面向用户使用的，不保证将来的兼容性，所以最好只用于调试程序，而不要用于正式程序。

● 如何知道当前任务阻塞在哪些信号量上？

解答：VxWorks 中没有提供相应命令，用户可以参考下面代码编写自己的实现函数，注意包括相关的头文件。

```
void waitList(void)
{
    static int taskIds[64];
    int numTasks, current, i;
```



```

WIND_TCB *tcb;

numTasks = taskIdListGet(taskIds, NELEMENTS(taskIds));
printf("  NAME          TID    STATUS OBJ_TYPE\n");
printf("-----  -----  -----  -----\n");

for(i = 0; i < numTasks; i++) {
    tcb = taskTcb(taskIds[i]);
    current = taskIds[i];
    if(tcb->status & TASK_STATUS_PEND)
        printf ("%10s %8x %d    %d\n", tcb->name, current, tcb->status,
                (SEM_ID)((char*)(tcb->pPendQ) - 0x8)->semType);
}
}

```

第 5 章 内存管理

本章重点介绍了 Flash、NVRAM 和 RAM 的在嵌入式系统中的作用、管理及操作。在 VxWorks 中，内存分配采用了最先适配算法，系统内部使用了内存分区、内存池和内存块等概念来管理内存，对外向用户提供了一些内存操作 API 函数，方便用户创建内存分区和申请内存块。由于嵌入式系统的实时性和可靠性要求，在程序设计中应尽量避免使用动态内存分配。VxWorks 系统中提供了两个虚拟内存接口：基于基本 MMU 的虚拟内存和由 VxVMI 提供的全功能 MMU 的虚拟内存。通过虚拟内存，可以实现对物理内存的操作，并在 MMU 的参与之下，设置内存区域的写保护属性，防止设计不良的应用程序影响系统内核的正常运行。最后介绍了几种手段来验证嵌入式系统的内存使用，帮助用户迅速开发稳定、可靠的应用程序。

5.1 基本概念

嵌入式系统中的存储器主要包括随机存取存储器（RAM，简称随机存储器）、只读存储器（ROM）和非易失存储器（包括 Flash 和 NVRAM）。这几种存储器根据不同的存取特性、性能和成本，适用于不同的场合。ROM、Flash 和 NVRAM 是嵌入式系统中的外部存储器，其内容在系统掉电的情况下依然存在，因此适合于存放启动代码、配置信息和实时时钟信息等。绝大部分指令的执行和所有数据的修改则是在 RAM 中完成，RAM 就是通常所说的内部存储器，简称内存。在任何一个计算机系统中，内存是必不可少的。

ROM 主要用于存放 VxWorks 的引导映像 BootImage（有时也称 BootRom）。由于 ROM 中代码的调试不便，更新引导映像时需要更换 ROM 芯片，导致成本增高，因此在目前的嵌入式系统中已经很少使用。

Flash 也称闪存，同样可以存放 VxWorks 的引导映像。通过使用仿真器（如 Vision ICE 或 TRACE32），可以反复、快速擦写 Flash 中的内容，也可以调试 Flash 中代码的运行。在 VxWorks 操作系统的支撑之下，用户还可以使用自己的程序对 Flash 进行读写访问。Flash 的访问速度比 DRAM 稍慢一些，但比 ROM 快，因此目前的嵌入式系统中广泛使用了 Flash 来作为一种存储介质。尤其值得一提的是，VxWorks 中提供了 Flash 文件系统，即 TrueFFS（简称 TFFS）的驱动。通过 TFFS，用户可以把 Flash 的全部或部分空间创建一个 TFFS 设备，通过加载合适的文件系统（如 Dos 文件系统 DosFs）实现在该设备上目录和文件的操作。这意味着，用户可以在 Flash 中以文件的形式保存配置信息和日志信息，存放运行时需要动态加载的应用程序模块，通过文件传输协议（FTP 或 TFTP）方便地在线升级 BootImage、VxWorks 映像和应用程序模块。由于大部分嵌入式系统没有类似计算机的硬盘，因此 Flash 的存在就显得尤为重要。

NVRAM 由电池供电，其初始化和读写更为方便。但是由于价格因素，嵌入式系统中即使包括 NVRAM，其容量也是较小的。NVRAM 主要用于存放系统引导参数、配置信息（如以太网的 MAC 地址、IP 地址）和实时时钟[RTC]等信息。

内存 RAM 包括动态存储器 DRAM 和静态存储器 SRAM，SRAM 主要用作高速缓存 Cache。本章中提到的内存，除非特别指明，都是指 DRAM，也称主内存。

在嵌入式 VxWorks 系统中，内存管理是由操作系统来完成的。内存管理的工作主要是跟踪哪些内存区域已被使用，哪些是空闲区域，在任务需要时为其分配内存空间，使用完毕之后释放空间。对于实时操作系统来说，内存管理必须高效率，开销必须是可预见的。一种解决方法是预先分配内存，即在系统构造或编译时为每个任务指定其使用的内存空间。

但是，嵌入式系统开发者不得不参与系统的内存管理。编译内核时，开发者必须告诉系统这块开发板到底拥有多大的内存；开发应用程序时，必须考虑内存的分配情况并关注应用程序需要运行的内存空间大小。另外，由于 VxWorks 所采用的内存管理策略，用户程序、内核和其他用户程序处于同一个地址空间，开发程序时必须保证不侵犯其他程序和内核的地址空间，以免破坏系统的正常工作，或导致其他程序异常运行。因此嵌入式系统的开发者对程序中内存的操作要格外小心。

其实，实时操作系统的内核仅仅需要负责为程序分配内存、动态地分配内存和回收内存。为程序分配内存是指当嵌入式系统从主机下载程序或从外存获得程序代码时，操作系统内核必须为程序代码和数据分配内存。这种分配规模比较大，在系统运行时较少发生，一般只有在系统启动或复位时发生。第二种内存管理要求就是在应用程序运行时，动态地分配和回收内存空间。一般的实时应用总是尽量避免动态分配内存和释放内存，因为这样会增加系统的不确定性，降低系统的稳定性。

VxWorks 为用户提供了两种内存区域：内存域 region 和内存分区 partition。region 是可变长的内存区，可以从创建的 region 中再分配段 segment（比如代码段[text segment]，数据段[data segment]和未赋初值的数据段[bss segment]），其特点是容易产生碎片，但灵活、不浪费。partition 是定长的内存区，用户可以从创建的 partition 中分配内存块（buffer 或 block），也可以在某个内存分区中再创建一个内存分区。内存分区的特点是：无碎片、效率高，但浪费。通常情况下，VxWorks 内核和应用程序对内存的操作都是基于内存分区进行的。

内存能够在被使用之前，首先要进行初始化。VxWorks 对内存的初始化是在 romInit.s 文件的 romInit 函数中进行，这是一段汇编代码，主要根据启动类型（冷启动或热启动），对内存进行不同的初始化，并定义 DRAM 的刷新表。这一部分代码由 BSP 开发者完成。

5.2 VxWorks 中的内存布局

VxWorks 中主要涉及到的内存单元的概念有内存分区[memory partition]、内存池[memory pool]和内存块[memory block]。内存池是一块连续的内存区域，包含一个或多个内存块，这些内存块通过 memPartAlloc、memPartFree 来申请和释放。内存分区包含分区自身的描述信息（一个结构体）和一个或多个内存池，描述信息保存在系统内存分区中（即这个结构体是在系统内存分区中用 malloc 分配），而内存池就是该分区实际拥有的内存空间。内存分区在刚创建完毕时，只有一个内存池，用户程序可在稍后往该分区中添加别的内存池。在一个内存分区中，内存池之间的地址不一定是连续的。VxWorks 在启动过程中会创建一个包含系统内存池[system memory pool]的系统内存分区[system memory partition]。操作系统和通常的大部分用户应用程序对内存的操作，都发生在系统内存池中。

如果 VxWorks 系统的启动是 BootRom+VxWorks 方式，那么系统中会先后存在两种内存布局方式，即 BootRom 运行时的内存布局和 VxWorks 运行时的内存布局。

5.2.1 BootRom 运行时的内存布局

BootRom 运行时的内存布局如图 5-1 所示。

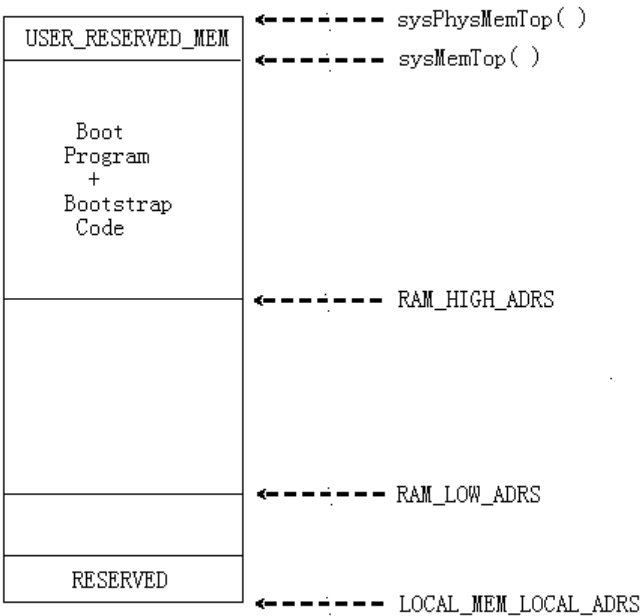


图 5-1 BootRom 的内存布局

嵌入式系统加电时，首先运行 BootRom 引导程序。BootRom 通常存在于 Flash、ROM、软盘或硬盘中。BootRom 完成必要的、最小硬件初始化，并把自身的一部分代码拷贝到某一特定的内存空间，该内存空间的起始地址通常是 RAM_HIGH_ADRS。但是对于压缩的映像，则为 RAM_LOW_ADRS，拷贝的同时还要进行解压缩，然后在进行代码的第二次重定位时，将解压缩后的部分拷贝到 RAM_HIGH_ADRS 处。拷贝完毕之后，根据启动参数从合适的启动设备加载 VxWorks，并把 VxWorks 的映像拷贝到以 RAM_LOW_ADRS 为开始地址的内存空间。图中的宏 RAM_LOW_ADRS 和 RAM_HIGH_ADRS 定义在 BSP 的 Makefile 和 config.h 文件中，两个文件中的定义应该完全一致，它们分别表示 VxWorks 和 BootRom 映像的加载起始地址。USER_RESERVED_MEM 和 LOCAL_MEM_LOCAL_ADRS 定义在 BSP 的 config.h 文件中，分别表示用户保留内存区域的大小和内存起始地址。这些宏定义应该以十六进制形式表示，在 config.h 中的表示形如 0Xxxxxxxx，而在 makefile 中则不需要“0X”前缀，形如 xxxxxxxx。函数 sysPhysMemTop 和 sysMemTop 的实现在 BSP 目录下的 sysLib.c 文件中，分别返回系统物理内存的结束地址和系统可用内存的结束地址。下面是这两个函数的实现（与目标板的 CPU 体系有关）。

```
char * sysPhysMemTop (void)
{
    PHYS_MEM_DESC * pMmu;          /* points to memory desc. table entries */
    char          gdt[6];          /* stores a copy of the GDT */
    BOOL          found = FALSE;

    if (memTopPhys != NULL)
    {
```

```

        return (memTopPhys);
    }
#ifdef LOCAL_MEM_AUTOSIZE
    /*计算内存的大小，赋给 memTopPhys，并置 found 为 TRUE*/
#endif /* LOCAL_MEM_AUTOSIZE */
    if (!found)
    {
        memTopPhys=(char *) (LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE);
    }
    .....
    return (memTopPhys);
}
char * sysMemTop (void)
{
    static char * memTop = NULL;

    if (memTop == NULL)
    {
        memTop = sysPhysMemTop () - USER_RESERVED_MEM;
    }
    return memTop;
}

```

可以看出,这两个函数的目的就是在内存顶端保留出大小为 USER_RESERVED_MEM 的内存区域。其目的何在? 系统冷启动(加电)时将会清除所有的内存空间,以防止内存校验错,而热启动(reboot)则不会清除 USER_RESERVED_MEM 和 RESERVED 这两块内存区域。因此,如果应用程序在运行时出现致命性的错误,会导致系统死机或某些任务被挂起,应用程序可以在出现这些错误之前,把一些合适的调试信息写入到 USER_RESERVED_MEM 的内存区域,待系统热启动之后,根据这块内存中保留的调试信息,即可大致判断问题的原因所在。此外,还可以在这一块内存空间中,为某些应用程序模块保留一块专用的内存分区,使得这个模块中的所有内存的申请和释放操作仅限于该内存分区中,从而减少内存碎片,降低系统内存管理的负担,使得对内存的操作更加快速、简便和安全。

如果嵌入式计算机系统能够自动计算出内存的大小(比如 x86 系列目标机),那么只需在 BSP 的 config.h 中定义 LOCAL_MEM_AUTOSIZE。对于不能自动计算内存大小的系统(比如 PowerPC 系列目标机),则需要通过定义 LOCAL_MEM_SIZE 来表示内存的大小。

5.2.2 VxWorks 运行时的内存布局

VxWorks 运行时的内存布局如图 5-2 所示。

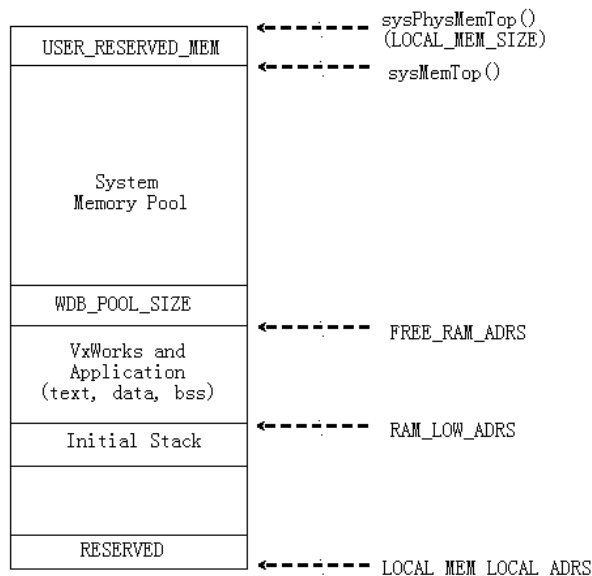


图 5-2 VxWorks 的内存布局

图 5-2 所示中，WDB_POOL_SIZE 内存区域用于 VxWorks 中调试代理任务进行模块的动态加载。当然，如果 VxWorks 的配置中没有包含 WDB Agent，这块内存则并入系统内存池。系统内存池是 VxWorks 和应用程序运行时可操作、使用的内存空间。应用程序可以根据需要，调用函数 `memPartCreate` 从系统内存池中创建新的内存分区。

Initial Stack 这部分空间是由两个宏定义形成 `STACK_ADRS` 和 `STACK_SAVE`，分别表示 Initial Stack 的起始地址和大小。在 VxWorks 内核启动时，`kernelInit` 函数将生成第一个任务 `tUsrRoot`（其入口函数为 `usrRoot`），它有自己的内存空间，其起始地址为 `MEM_POOL_START_ADRS`，大小为 `ROOT_STACK_SIZE`（在 `configAll.h` 中定义），因此它可以在这块内存空间中进行内存的申请和释放。但是，在 `kernelInit` 之前的内存申请应该在哪里实现呢？应该在 Initial Stack 中。在基于 ROM 的 VxWorks 映像中，这块内存区域由 `romInit` 初始化，交给 `romStart` 使用，而在可加载的 VxWorks 映像中，则是由 `sysInit` 初始化，由 `usrInit` 使用。

Initial Stack 和 RESERVED 之间的内存区域通常用于存放中断向量表。

5.3 内存分配算法

在 VxWorks 编程中，对于内存的使用应该遵循一个原则，即尽量使用静态的内存分配，比如变量、数组的事先申明。因为使用静态内存分配能够减少对内存的操作，降低内存碎片、提高系统运行速度。所谓内存碎片是指内存中存在一些不连续的空闲内存块。由于这些空闲内存块太小，在随后的每一次内存申请中，它们都不可能被使用。内存碎片的存在不仅使得系统的可用内存空间减少，而且会增加内存管理单元的计算负担，降低系统的实时性。但是，应用程序对内存的动态申请是不可避免的，尤其是在开发、调试阶段。因此有必要了解 VxWorks 下的内存分配算法，尽量减小内存操作对系统性能的影响。

系统在经过多次内存的分配和释放操作之后，可能存在多个空闲的内存块，后续的内存

操作必须能够检索到这些内存块，从而根据一定的算法选择一个合适的内存块。因此需要对空闲内存块进行合适的管理，这可以通过内存可用表或自由链两种方法进行。

可用表是一张二维表格，每个表项记录了一个空闲内存块，其主要参数包括块号、长度和起始地址。可用表采用的是表格管理，管理过程比较简单，但表的大小难以确定，表本身需要占用一部分内存。

自由链则是利用每个空闲内存块的开始几个单元存放本空闲块的大小及下个空闲块的开始地址。这样，管理程序通过链首指针可以检索到所有的空闲块。采用自由链法管理空闲块，空闲内存块的查询工作量较大，但由于自由链指针利用的是空闲块自身的单元，所以不必占用额外的内存块。此外，链表没有大小的限制，容易添加或删除节点。VxWorks 中采用的是自由链管理内存空闲块。

通常有 3 种内存分配算法：最先适应法[first fit]、最佳适应法[best fit]和最坏适应法[worst fit]。

✧ 最先适应法

最先适应法要求可用表或自由链按起始地址递增的次序排列。该算法的最大特点是一旦找到大于或等于所要求内存长度的内存块，则结束搜寻。然后，该算法从所找到的内存块中划分出所要求大小的内存空间分配给用户，并把余下的部分进行合并（如果相邻内存块是空闲的），合并后的内存块留在可用表或自由链中，并修改其相应的表项。

✧ 最佳适应法

最佳适应法要求空闲内存块按照从小到大的次序组成空闲内存块可用表或自由链。当系统申请一个空闲内存块时，内存管理程序从表头开始查找，当找到第一个满足要求的空闲块时就停止。如果该空闲块大于申请的长度，则与最先适应法相同，将减去请求长度后的剩余空闲块部分留在可用表或自由链中。

✧ 最坏适应法

最坏适应法要求空闲内存块按大小递减的顺序组成空闲块可用表或自由链。当系统申请一个空闲块时，先检查空闲块可用表或自由链的第一个空闲可用块的大小是否大于或等于所要求的内存大小。若可用表或自由链的第一项所示空闲块长度小于所要求的，则分配失败。否则从这个空闲块中分配相应的存储空间给用户，然后修改和调整空闲块可用表或自由链。

从搜索速度上看，最先适应法具有最佳性能。尽管最佳适应法和最坏适应法看上去能很快找到一个最适合的或最大的空闲内存块，但这两种算法都要求首先把不同大小的空闲块按其大小进行排队，这实际上是对所有空闲块进行了一次搜索。再者，从内存的回收方面来看，最先适应法也是最好的。因为使用最先适应法回收某一空闲块时，无论被释放块是否与空闲块相邻，都不用改变该内存块在可用表或自由链中的位置，只需修改其大小和起始地址。而最佳适应法和最坏适应法都必须重新调整该内存块的位置。

最先适应法的另一个优点是尽可能地利用低地址空间，从而保证高地址空间有较大的空闲区来放置要求内存较多的进程或作业。

反过来，最佳适应法找到的空闲内存块是最佳的，也就是说，用最佳适应法找到的空闲块或者是正好等于用户请求的大小，或者是能够满足用户要求的最小空闲块。不过，尽管最佳适应法能选出最适合用户要求的可用空闲块，但在某些情况下并不一定能够提高内存的利用率。例如，当用户请求的内存块比最小的空闲块小，但小得不多时，分配程序将其分配

后的剩余部分作为一个新的小空闲块留在可用表或自由链中。这种小空闲块有可能永远得不到使用（除非与别的空闲区合并），形成碎片。另外，最佳适应法也会增加内存分配和回收时的查找负担。

最坏适应法正是基于不留下碎片空闲块这一出发点设计的。它选择最大的空闲块来满足用户要求，以期分配后的剩余部分仍能够被再分配。

有关内存分配算法的描述，参见《计算机操作系统教程》（张尧学、史美林著，清华大学出版社出版）的 6.3 节。

VxWorks 正是基于以上原因及嵌入式系统对实时性的要求，采用了最先适应法来动态分配内存。VxWorks 中内存释放时，只采用了上下空闲区融合的方法，即把相邻的空闲内存块融合成一个空闲块。

VxWorks 没有清除碎片的功能，因为嵌入式实时系统找不到一个合适的时间把正在运行的程序暂停下来清理内存。例如 Windows 操作系统有时会自己响个不停，就是在清理内存碎片，但是在实时系统的设计中，只能尽量避免内存碎片的产生。

5.4 VxWorks 中内存操作 API 函数

为了保证应用程序对数据的正确访问，在一个系统中分配的内存必须是字节对齐的，即分配的内存块的起始地址和大小应该是对齐值的整数倍。如果用户申请的内存块大小不满足该要求，VxWorks 会自动调整。此外，在 VxWorks 中分配内存块时，块本身有额外的开销。对于不同的目标机体系，分配内存时的对齐值和内存块的开销是不同的。了解这一点，就能够知道分配特定大小内存空间时，实际被消耗的内存大小。如表 5-1 所示列出了各种不同体系的字节对齐值和内存块额外的开销。

表 5-1 各种体系的字节对齐值和开销

体系	对齐值（字节）	块开销（字节）
ARM	4	8
COLDFIRE	4	8
I86	4	8
M68K	4	8
MCORE	8	8
MIPS	16	16
PPC *	8/16	8/16
SH	4	8
SIMNT	8	8
SIMSOLARIS	8	8
SPARC	8	8

PowerPC 体系的 PPC604 CPU 类型（包括 ALTIVEC）的字节对齐值和内存块开销是 16 字节，而其他 CPU 类型则是 8 字节。

在 VxWorks 中内存分区只能创建，不能删除。主要的操作函数有 memPartCreate、memPartAddToPool、memPartAlignedAlloc、memPartAlloc、memPartFree、memAddToPool、malloc、free、calloc 和 cfree 等。下面对各函数做一个较为详细的说明。

✧ memPartCreate

```
PART_ID memPartCreate
(
    char *pPool,                /* pointer to memory area */
    unsigned poolSize           /* size in bytes */
)
```

该函数用于创建一个从地址 **pPool** 开始、大小为 **poolSize** 的内存分区，该分区包含了一个内存池。返回值是分区的 ID 值，如果没有 **poolSize** 大小的内存空间，则返回空（NULL）。该分区 ID 值可以传递给其他函数，以便对该内存分区进行管理、分配和释放内存块。每一个创建的分区都有一个描述符，以便操作系统对其进行管理。这个描述符所占用的空间是从系统内存分区分配出来的。有了内存分区，就可以方便地管理多个内存池。

✧ memPartAddToPool

```
STATUS memPartAddToPool
(
    FAST PART_ID partId,        /* partition to initialize */
    FAST char *pPool,           /* pointer to memory block */
    FAST unsigned poolSize      /* block size in bytes */
)
```

该函数用于把一块从地址 **pPool** 开始、大小为 **poolSize** 的内存池添加到内存分区 **partId** 中，该内存分区应该已经使用 **memPartCreate** 创建完毕。其实，在 **memPartCreate** 函数的最后一步就调用了 **memPartAddToPool** 函数，把指定的内存池添加到该分区中。本章的常见问题解答部分举例说明了如何把一个额外单独的内存池添加到某个内存分区中。每调用一次 **memPartAddToPool**，分区中的内存池数量就加一。一个分区中的相邻内存池之间地址空间不一定连续。

操作成功返回 OK，失败则返回 ERROR。

✧ memPartAlignedAlloc

```
void *memPartAlignedAlloc
(
    FAST PART_ID partId,        /* memory partition to allocate from */
    unsigned nBytes,            /* number of bytes to allocate */
    unsigned alignment          /* boundary to align to */
)
```

该函数用于从指定内存分区的内存池中分配一个大小为 **nBytes** 的内存块。此外，它保证了所分配的内存块起始地址能够被对齐值整除。对齐值必须是 2 的整数次幂。返回值是该内存块的指针，如果不能分配，则为空指针。

计算实际消耗的内存大小时，首先需要进行对齐运算，再加上内存块头信息的大小，然后和所允许的最小值进行比较。所以实际消耗的内存肯定会大于所申请的值，可使用的内存大小可能大于或等于申请值。

✧ memPartAlloc

```
void *memPartAlloc
(
    FAST PART_ID partId,        /* memory partition to allocate from */
    unsigned nBytes             /* number of bytes to allocate */
)
```

该函数除了对共享内存时的情况做了一个判断，就直接调用了 **memPartAlignedAlloc** 函

数，因此在没有使用共享内存网络的情况下，这两个函数功能相同。

✧ **memPartFree**

```
STATUS memPartFree
(
    PART_ID partId,           /* memory partition to add block to */
    char *pBlock              /* pointer to block of memory to free */
)
```

该函数把先前已分配的内存块(通过调用 **memPartAlloc** 获得)释放回同一个内存分区中。释放后的内存块被添加到内存自由链中。如果自由链上该内存块的相邻内存块也是空闲的，则进行合并，形成一个新的空闲内存块。

操作成功返回 **OK**，失败则返回 **ERROR**。

✧ **memAddToPool**

```
void memAddToPool
(
    FAST char *pPool,         /* pointer to memory block */
    FAST unsigned poolSize    /* block size in bytes */
)
{
    (void) memPartAddToPool (&memSysPartition, pPool, poolSize);
}
```

该函数用于把某一个内存池添加到系统内存分区中。**memSysPartition** 是一个全局变量，是系统内存分区的 **ID**。可以看出，该函数只能用于系统内存分区，其实质就是调用了 **memPartAddToPool** 函数。

✧ **malloc**

```
void *malloc
(
    size_t nBytes             /* number of bytes to allocate */
)
{
    return (memPartAlloc (&memSysPartition, (unsigned) nBytes));
}
```

该函数用于从系统内存分区中分配一个内存块。同样，这个函数只能用于系统内存分区。其实质就是调用了 **memPartAlloc** 函数。

如果调用成功，返回被分配的内存块地址，否则返回空指针。

✧ **free**

```
void free
(
    void *ptr                 /* pointer to block of memory to free */
)
{
    (void) memPartFree (&memSysPartition, (char *) ptr);
}
```

该函数把从系统内存分区中分配的一个内存块(通过调用 **malloc** 获得)释放回该内存分区。同样，这个函数只能用于系统内存分区，其实质就是调用了 **memPartFree** 函数。

✧ **calloc**

```
void *calloc
(
```

```

size_t elemNum,          /* number of elements */
size_t elemSize          /* size of elements */
)
{
FAST void *pMem;
FAST size_t nBytes = elemNum * elemSize;

if ((pMem = memPartAlloc (memSysPartId, (unsigned) nBytes)) != NULL)
    bzero ((char *) pMem, (int) nBytes);

return (pMem);
}

```

该函数用于从系统内存分区中划分出一定大小（ $\text{elemNum} \times \text{elemSize}$ ）的内存块，这个内存块包含 `elemNum` 个元素，每个元素的大小是 `elemSize`。`memSysPartId` 就是系统内存分区的 ID 值。除此之外，还把分配的内存块清零（`malloc` 函数则没有清零的操作过程）。

如果调用成功，返回被分配的内存块地址，否则返回空指针。

✧ **cfree**

```

STATUS cfree
(
    char *pBlock          /* pointer to block of memory to free */
)
{
    return (memPartFree (memSysPartId, pBlock));
}

```

该函数把从系统内存分区中分配的一个内存块（通过调用 `calloc` 获得）释放回该内存分区。同样，这个函数只能用于系统内存分区，其实质调用了 `memPartFree`。

操作成功返回 `OK`，失败则返回 `ERROR`。

通过上述的 API 函数，用户可以完成在 VxWorks 下对内存分区、内存池和内存块的操作，为应用程序提供合适的内存空间。

5.5 Flash 存储器

Flash 芯片具有较快的存取速度，其内容的保存无需电池供电，可多次反复擦写其中的信息，擦写时所需的电压较低，且芯片无需从目标板上取下，还可通过应用程序修改其内容。Flash 可用于存放 VxWorks 的启动代码（`BootRom` 或 `VxWorks_Rom`）。写入启动代码时通常需要某种仿真器硬件或应用程序的支持。例如，如果实现了 VxWorks 中的 TFFS 组件的底层驱动，就可以在应用程序中调用 `tffsBootImagePut` 函数把 `BootImage` 写入 Flash。

如图 5-3 所示显示了 Flash 在 VxWorks 系统中的一种典型应用。系统中共有 4MB 的 Flash，其地址空间为 `0XFFC00000` 至 `0XFFFFFFF`。有两个 Flash 芯片，第一个的地址从 `0XFFC00000` 到 `0XFFDFFFFF`，第二个的地址从 `0XFFE00000` 到 `0XFFFFFFF`。

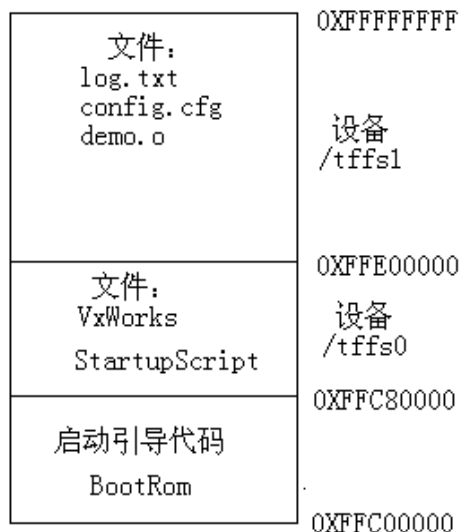


图 5-3 一种典型的 Flash 应用

在 Flash 上创建 TFFS 设备之前，需要参考芯片的详细资料和“VxWorks Programmer’s Guide”的第 8 章，实现 Flash 的 TFFS 底层驱动。TFFS 设备的创建可以使用 `usrTffsConfig` 函数，其原型如下。

```
STATUS usrTffsConfig
(
    int      drive,          /* drive number of TFFS */
    int      removable,     /* 0 - nonremovable flash media */
    char *   fileName       /* mount point */
)
```

例如，可以通过 `usrTffsConfig` 来创建一个设备“/tffs0”，并用 Dos 文件系统对该设备进行初始化。

```
usrTffsConfig (0,0,"/tffs0");
```

设备在可使用之前，需要进行格式化，可通过调用 `sysTffsFormat` 函数完成。下面是该函数的一个实例：

```
STATUS sysTffsFormat (void)
{
    STATUS status;
    tffsDevFormatParams params =
    {
#define HALF_FORMAT /* lower 0.5MB for bootimage, upper 1.5MB for TFFS */
#ifdef HALF_FORMAT
        {0x800001, 99, 1, 0x100001, NULL, {0,0,0,0}, NULL, 2, 0, NULL},
#else
        {0x0000001, 99, 1, 0x100001, NULL, {0,0,0,0}, NULL, 2, 0, NULL},
#endif /* HALF_FORMAT */
        FTL_FORMAT_IF_NEEDED
    };
    status = tffsDevFormat (0, (int)&params);
    return (status);
}
```

```
}
```

可以看出, 由于定义了 HALF_FORMAT, 那么对于 “/tffs0” 设备的格式化实际上是从 0.5MB (由参数 0x800001 指定, 注意最后一位是英文字母 “1”, 而不是数字 “1”) 后开始, 因此前面 0.5MB 空间可用于存放 VxWorks 的引导映像, 即 BootRom。当然该空间的大小可以根据实际需要增减。

设备 “/tffs0” 创建之后, 就可以在其上进行文件的操作, 不过首先需要将当前路径设置为 “/tffs0”。设置路径可以调用 ioDefPathSet 或 cd。此后, 就可以通过调用 I/O 系统的函数 create、open、read、write 和磁盘操作函数 mkdir、remove、ls 等。对于图 5-3 所示的 “/tffs0” 设备的一个典型用法是, 以文件的形式保存 VxWorks 映像, 文件是通过 FTP 或 TFTP 传送并保存于 “/tffs0” 设备上。VxWorks 保存于 “/tffs0” 设备上之后, 在系统引导过程中就可以把启动设备改为 “tffs=0,0”, 启动文件名改为 “/tffs0/VxWorks”, 从而告诉引导程序从 TFFS 设备上加加载映像文件。在 VxWorks 启动完毕之后, 可以执行用户指定的启动脚本文件[startup script], 可以进行一系列的设置或函数调用, 完成后续初始化, 因此也可以把这个脚本文件存放于 “/tffs0” 设备上, 并修改启动参数的 “startup script (s)” 项。

“/tffs1” 设备同样调用 usrTffsConfig 来初始化, 不过由于它不再需要存放引导代码, 格式化函数 sysTffsFormat 不需要定义 HALF_FORMAT。用户可以在 “/tffs1” 设备上存放日志文件、应用程序配置文件和应用程序模块文件等。

使用 Flash 也有缺点。首先, 在对 Flash 进行写操作之前必须要进行擦除操作, 擦除时不能以字节为单位进行, 只能以扇区或块为单位, 或者整片擦除。Flash 的写和擦除操作都需要较复杂的步骤才能完成, 降低了它的易用性。另外, Flash 存储器的寿命有限, 可擦写的次数与 Flash 的型号有关, 一般在 1~10 万次左右。

5.6 内存管理单元和 VxVMI

VxWorks 中提供了两种内存管理单元[MMU]的支持: 基本 MMU 和 VxVMI。基本 MMU 绑定于 VxWorks 中, 其实现要求在 BSP 的 config.h 文件中定义宏 INCLUDE_MMU_BASIC, 或在 Tornado 的工程配置中包含基本 MMU 组件。VxVMI 是 Wind River 公司推出的一个可选软件模块, 实现了 MMU 的全部功能, 稍后将对其进行介绍。VxWorks 中有关 MMU 的配置包括以下内容。

- ✧ INCLUDE_MMU_BASIC: 基本 MMU 的支持。
- ✧ INCLUDE_MMU_FULL: 完整 MMU 支持 (需要 VxVMI)。
- ✧ INCLUDE_PROTECT_TEXT: 代码段写保护支持 (需要 VxVMI)。
- ✧ INCLUDE_PROTECT_VEC_TABLE: 中断向量表写保护支持 (需要 VxVMI)。

5.6.1 虚拟内存

VxWorks 中提供了两个层次的虚拟内存管理接口, 第一个是 VxWorks 绑定的虚拟内存接口, 即基本的 MMU 支持, 能提供基于页面的缓存; 第二个是由可选组件 VxVMI 提供的虚拟内存接口, 可以实现对代码段和中断向量表的写保护, 并为 CPU 的 MMU 提供与体系无关的接口。

在诸如 Windows 和 UNIX 这样的分时操作系统中, 很可能会先后执行很多的进程, 但由于受计算机系统的实际内存容量的限制, 不可能把这些进程全部同时调入内存。此外, 运行

一些较大的程序时，很可能出现所需要的内存空间大于系统所能提供的，即实际内存的大小总是有限的。为了解决这样的问题，可以把某一部分暂时不运行的程序或某个程序的一部分放入外存，因此有可能出现一部分程序段和数据在内存，而另外一部分在外存。为了使这样的程序能够被正常执行，就必须使用虚拟内存。虚拟内存不考虑物理内存的大小和信息存放的实际位置。每个进程拥有其独立的虚拟内存空间，其大小不受实际物理内存容量的限制。在分时操作系统中，编译连接程序把用户的源程序编译后连接到以 0 为起始地址的线性或多维虚拟地址空间，而这些程序实际是运行在一个物理的内存空间中，因此需要进行虚拟地址到物理地址的转换。

在 VxWorks 中，虚拟内存地址和物理内存地址是一一对应的关系，因此如果不考虑对特定内存区域设置写保护或高速缓存机制，则完全可以不使用虚拟内存。静态编译连接的程序，它们在内存中的地址是在代码连接时就被指定的。此外，在 VxWorks 中的虚地址到实地址的转换也是非常简单的。

5.6.2 基本 MMU 的支持

在 VxWorks 系统中，内存管理单元 MMU 可以指定一些特定的内存区域为不可缓存，该特性能够使直接内存访问 [DMA] 和处理器内部通信更为有效、迅速。当其他处理器或 DMA 设备访问同一段地址空间时，可以保证这些数据不被缓存。如果系统不具备这样的不可缓存特性，那么在访问这些地址空间时，必须关闭高速缓存 Cache（这将导致性能降低），或者清洗缓冲区（flush）和使缓冲区无效（invalidate）。如果要使特定的缓冲区为不可缓存，需要建立虚地址到实地址的映射，并设置其属性。

MMU 的功能主要是由 VxWorks 中的一些库函数来控制 and 实现，这些库函数根据 CPU 的体系不同而不同。BSP 负责提供物理内存描述的支持，主要是通过 BSP 目录下 sysLib.c 文件中的结构体 sysPhysMemDesc 来实现，该结构体的类型是 PHYS_MEM_DESC（其定义于 vmLib.h 文件中），实际上这是一张虚地址和实地址之间的转换表。

```
typedef struct phys_mem_desc
{
    void *virtualAddr;
    void *physicalAddr;
    UINT len;
    UINT initialStateMask;    /* mask parameter to vmStateSet */
    UINT initialState;       /* state parameter to vmStateSet */
} PHYS_MEM_DESC;
```

其中，virtualAddr 表示被映射后的虚拟内存空间的起始地址；physicalAddr 表示将被映射的物理内存空间的起始地址。除了 MC680x0 系列外的目标机，physicalAddr 和 virtualAddr 可以不相等（但通常是相等的）。initialState 表示的是虚拟内存页面的状态，分为是否有效、是否可写、是否可缓存。len 表明这一段被映射内存的长度。由于内存映射是基于页面的，所映射的内存空间应该是页面对齐的，因此 physicalAddr、virtualAddr 和 len 都应该是页面大小的偶数倍。页面大小是由 configAll.h 中的宏 WM_PAGE_SIZE 来定义的。有些目标机的 WM_PAGE_SIZE 是 4KB，而另外一些是 8KB，与 CPU 体系有关。sysPhysMemDesc 可能包含的映射空间有内存、Flash、ROM、NVRAM、I/O 设备和 VME 总线地址空间等。下面是多 CPU 嵌入式系统中使用共享内存网络时，各目标板的 MMU 配置示例。

```
{
```

```
(void *) 0x4000000,          /* virtual address */
(void *) 0x4000000,          /* physical address */
0x20000,                    /* length */
/* initial state mask */
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE | VM_STATE_MASK_CACHEABLE,
/* initial state */
VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}
```

这一段内存起始于 0x4000000，可写，不可缓存。

MMU 使用物理内存描述来建立一种地址映射关系，这种映射方式是平面型的，实地址和虚地址之间是一一映射的关系。

有两种方法可以修改虚地址和实地址之间的映射关系：静态方式，即通过修改结构体 sysPhysMemDesc 的内容；动态方式，通过调用 vmBaseStateSet 函数来修改一块虚拟内存空间的状态。vmBaseStateSet 函数的使用可参见 VxWorks 在线手册。一些常见的存储空间的状态如表 5-2 所示。

表 5-2 MMU 中一些常用的存储空间的状态

存储器类型	是否有效	是否可写	是否可缓存
本地内存	是	是	是
ROM	是	否	通常是
Flash	是	是	否
I/O 设备	是	是	否
板外存储器	是	是	否

所有可能会在应用程序中被访问的物理内存（包括内存映射设备，比如以太网设备、SCSI 设备等）都应该被映射到不同的虚拟内存空间。在 MMU 使能的情况下，如果某一段物理地址没有被映射，而试图对这样的地址进行写操作的时候，将会引起总线错误。

MMU 的初始化，是在 VxWorks 启动时由任务 tUsrRoot 通过调用 usrMmuInit 函数来完成的。VxWorks 首先完成系统内存池的初始化，再初始化 MMU，后续的初始化过程就可以使用 MMU 功能了。

5.6.3 VxVMI

VxVMI 是 WindRiver 公司推出的 MMU 功能模块，它实现了更多的 MMU 功能。VxVMI 能够通过设置某一块内存区域的写保护属性，从而为 VxWorks 中的代码段和中断处理向量表提供了写保护，防止这部分内存的内容被改写。然而，并不是所有的 CPU 体系都能实现这种写保护。当试图对写保护的内存地址进行写操作时，将出现总线错误。如果希望修改中断处理向量表的内容，用户只能通过 intConnect 函数进行。intConnect 函数把用户的中断服务程序挂接到中断向量表中，它会暂时解除中断向量表内存区域的写保护。

用户可以调用 vmStateSet 函数来设置某一内存区域的写保护属性，以保护特定的数据块。例如，设置内存映射策略（填写 sysPhysMemDesc 结构体）时，可以使某块内存数据区的状态为 VM_STATE_WRITABLE_NOT，然后编写一个专门的函数来对这一块内存数据进行修改。这样的函数应首先调用 vmStateSet 将内存的状态置为 VM_STATE_WRITABLE，待数据修改完毕之后，再将其状态设置为 VM_STATE_WRITABLE_NOT。如前所述，VxWorks 中所有的代码和数据都位于同一地址空间，一条非法访问指令可能导致系统出现严重的错误。但

是，如果使用了 VxVMI 提供的虚拟内存写保护机制，就可以很好地保护一些关键内存数据区。

VxVMI 中用于实现虚拟内存的基本结构是虚拟内存上下文[Virtual Memory Context (VMC)]。VMC 包括地址转换表和其他一些用于实现虚地址到实地址映射的信息（比如内存区域的状态等）。用户可以同时创建多个 VMC，并根据需要在它们之间进行切换。这些 VMC 中物理内存和虚拟内存的映射关系有所不同，一个正在运行的任务能够进行什么样的内存访问，取决于系统当前的 VMC 映射关系。VxWorks 中的一些系统对象，比如代码段、信号量等，必须能够被系统中的所有任务访问，而不管当前的 VMC 是什么，而另外一些系统对象则只能被某些特定任务访问。前者可以通过全局虚拟内存来实现，后者通过私有虚拟内存实现。

5.6.3.1 全局虚拟内存

全局虚拟内存包含了系统中多个物理地址空间到虚拟地址空间的映射关系。当系统把物理内存映射到相同地址的虚拟内存（映射关系由结构体 `sysPhysMemDesc` 确定）时，即创建了全局虚拟内存，也就是说全局虚拟内存实际上是 `sysPhysMemDesc` 描述的所有虚拟内存空间。在系统默认的配置中，物理内存和全局虚拟内存是一一对应的关系，但一些目标机系统可能会使用 `sysPhysMemDesc` 来设置虚地址，这样虚地址到实地址的映射就不一定是一一对应的了，比如两段虚拟内存空间映射到同一段物理内存空间。应用程序可以通过所有的 VMC 来访问全局虚拟内存，即全局虚拟内存呈现在所有 VMC 中。在一个 VMC 中对全局映射关系的修改，将影响所有的 VMC。应用程序可以通过调用 `vmGlobalMap` 函数来动态地添加全局虚拟内存的映射关系项，但该操作应该在创建所有的 VMC 之前。否则，修改后的全局内存映射关系有可能无法在当前的 VMC 中使用。

在 VxWorks 启动时，`usrRoot` 通过 `usrMmuInit` 函数中的 `vmGlobalMapInit` 调用来初始化全局虚拟内存，利用 `sysPhysMemDesc` 结构体来创建全局虚拟内存，然后创建一个默认的 VMC，并使其为当前的 VMC。

5.6.3.2 私有虚拟内存

通过创建一个 VMC，可以产生一个私有虚拟内存。每个 VMC 有自己的内存映射关系，因此就有自己的私有虚拟内存空间。私有虚拟内存能保护数据不被其他任务或特定的函数所访问，其内存映射关系体现在 VMC 中。对于系统中正在运行的一个任务而言，它所能“看”到的内存映射关系体现在当前的 VMC 中，而别的 VMC 可能只对另外的某些任务可见。因此，当前 VMC 中的虚拟内存空间对该任务来说就是私有的。在 VxWorks 中生成任务时，并不自动创建 VMC，但可以在应用程序中创建多个 VMC，并在它们之间进行切换。如前所述，VxWorks 初始化过程中会创建一个默认的 VMC，所有的任务都会使用它。私有虚拟内存的创建是调用 `vmContextCreate` 函数来实现的，而调用 `vmCurrentSet` 函数来设置系统当前的 VMC。所有的 VMC 共享系统初始化过程中实现的全局内存映射。只有在当前的 VMC 中，且状态为有效的虚拟内存才可以被访问，其他 VMC 中定义的虚拟内存不能被访问。

为了在私有虚拟内存中建立新的虚拟内存和物理内存之间的映射关系，首先要确定实地址和虚地址，然后调用 `vmMap` 函数来实现。这里的物理地址应该是页面对齐的，可以使用 `valloc` 来分配。确定虚地址时，可以调用 `vmGlobalInfoGet` 来找到一个合适的虚拟内存页面。

当存在多个映射关系时，必须保证虚拟内存页面不会两次映射到同一全局地址。对于已经标明为全局的内存，不能使用 `vmMap` 函数来再次映射。

当物理内存页面映射到新的虚拟内存空间中后，可以通过两种虚拟地址对其访问，即新映射的虚地址和全局虚拟内存中与实际物理地址相等的虚地址。但是在一些 CPU 体系中，这样做可能会引起问题。这是因为针对某一实地址有两个不同的虚地址，那么 Cache 中就可能存在两个不同的缓存值。为解决这个问题，应该在全局虚拟内存中使这一虚拟内存页面无效（调用 `vmStateSet` 函数来设置），保证只能通过新映射的虚地址来访问某一实地址。

简单的说，VxVMI 就是通过使用虚拟内存上下文 VMC、全局虚拟内存和私有虚拟内存实现了对特定内存区域的保护。全局虚拟内存是一种全局的内存映射关系，影响所有的 VMC 和私有虚拟内存，用户程序可以在当前的 VMC 中添加、修改内存映射关系，创建一个私有虚拟内存，任务只能使用当前 VMC 所确定的私有虚拟内存空间。

5.7 高速缓存[Cache]

大家都知道，CPU 的指令处理速度与内存中指令的访问速度存在较大的差异。如果 CPU 每处理完一条指令都必须到内存中去取下一条指令，那么 CPU 的大多数时间都是在等待内存指令的访问，其速度不能得到充分有效的利用。为此，在一般的 CPU 中都使用了高速缓存 [Cache]。Cache 存在于 CPU 和主内存之间，是静态随机存储器 SRAM，因此没有刷新周期，存取速度更快，可以匹配 CPU 的处理速率与内存的访问速度。

5.7.1 Cache 的结构

如图 5-4 所示说明了 Cache 的位置及其组成。它由 3 部分组成：缓冲存储器、缓冲目录和缓存控制器。缓冲存储器分为若干块，用于缓存内存中的数据；缓冲目录用于描述各缓冲存储块的状态；缓存控制器负责缓存目录的维护，以及利用缓存淘汰算法更新缓存内容。

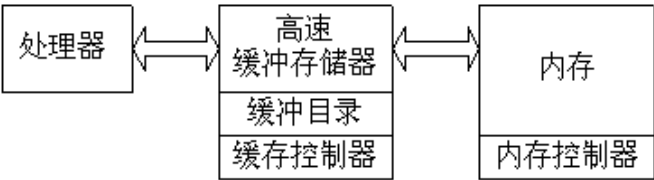


图 5-4 高速缓存的组织结构

5.7.2 Cache 的操作

VxWorks 系统加电时，Cache 通常是被禁止的，在随后的 VxWorks 内核初始化过程中将其使能。Cache 由 VxWorks 操作系统进行管理。

Cache 和 MMU 的配置可能是高度集成在一起的，也可能是相互独立的，这与 CPU 体系有关。但如果使能 MMU，则 Cache 由 MMU 控制。VxWorks 中基本的 Cache 管理函数由 `cacheLib` 库提供。BSP 对 Cache 的操作部分负责：针对不同的 Cache 实现选择合适的操作库函数及操作模式；如果 MMU 使能，内存的映射可以设置为可缓存或不可缓存；在编写 BSP 设备驱动程序时，应考虑采取合适的缓存策略。

Cache 的操作主要包括清洗缓冲区 (flush) 和使缓冲区无效 (invalidate)。flush 用于把 Cache

里的数据强制性的写到 RAM 中。当向设备写数据时，数据首先被写入 Cache，flush 操作之后，就可以保证 RAM 和 Cache 中的数据一致，即数据真正写入了设备。invalidate 用于接收处理时使 Cache 内容无效，当从设备里读取数据时，最初的数据在 RAM 中，invalidate 之后，Cache 的数据得到更新，与 RAM 中的内容一致。VxWorks 中，Cache 的 flush 和 invalidate 操作函数分别是 cacheFlush 和 cacheInvalidate。对于不可缓存内存的申请，可使用函数 cacheDmaMalloc。

5.7.3 Cache 的工作过程

在进行不同类型的内存操作时，Cache 会有不同的工作过程，具体如下。

- CPU 读数据：缓存控制器自动查找缓冲目录，确定相应内存数据是否在缓存中。

查找过程：依据读操作的地址确定查找缓冲目录的列号；比较缓冲目录相应列的各区行号与地址行号；判断有效位是否为 1（1 表示有效）。

依据查找结果，有两种可能的操作：如果在缓存，则从缓存中读数据，并修改访问标志；如果不在缓存，则从内存中读数据，同时该块内容被送到缓存相应列的某块。

- CPU 写数据：查找缓冲目录，确定相应地址是否在缓存中，有两种可能的操作。

如果在缓存，则修改缓存内容，并把缓存目录中相应修改位置 1。这时有两种做法，立即写，内存与缓存的相应块同时写；惰性写，数据只写入缓存，不马上写入内存；当该缓存块被清洗时，才写回内存。

如果不在缓存，则先把内存中的内容读入缓存，再在缓存中修改。

- 通道从内存读数据：如果在缓存中，则从缓存中读数据到通道；如果不在缓存中，则从内存中读数据到通道，但并不同时送到缓存。

- 通道向内存写数据：数据被写入内存，缓存控制器同时查找缓冲目录，如果有，则修改相应表项的有效位为 1。

5.8 NVRAM

NVRAM 全称是非易失随机存储器[Non-Volatile RAM]，它通常需要电池供电，可以保存时钟信号、配置参数和 VxWorks 的引导参数等。如果系统中不存在 NVRAM，启动参数是静态连接到启动代码中的。反之，由 configAll.h 配置文件可以知道，默认情况下启动参数保存于 NVRAM 前面 255 个字节的单元中，这样可以避免经常性地修改 BootRom 的启动参数。如果要改变默认值，须在 config.h 中取消宏定义 NV_BOOT_OFFSET、BOOT_LINE_SIZE 的定义，然后重新定义它们。NV_BOOT_OFFSET 表示启动参数存放位置相对于 NVRAM 基地址的偏移值，BOOT_LINE_SIZE 表示启动参数占用 NVRAM 存储单元的最大数目。

NVRAM 的操作接口函数是 sysNvRamGet 和 sysNvRamSet，分别用于 NVRAM 的读和写。函数原型如下：

```
STATUS sysNvRamGet
(
    char *string,    /* where to copy non-volatile RAM */
    int strLen,      /* maximum number of bytes to copy */
    int offset       /* byte offset into non-volatile RAM */
)
```

```

    )
STATUS sysNvRamSet
(
    char *string,      /* string to be copied into non-volatile RAM */
    int  strLen,       /* maximum number of bytes to copy          */
    int  offset        /* byte offset into non-volatile RAM        */
)

```

需要注意，这两个函数对 NVRAM 操作时，实际操作的地址是“offset+NV_BOOT_OFFSET”。在 VxWorks 中，要求 BSP 必须实现这两个函数。如果系统中不存在 NVRAM，应该在 BSP 的 config.h 中做如下定义。

```
# define NV_RAM_SIZE      (NONE)
```

如果存在 NVRAM，则把 NV_RAM_SIZE 定义为 NVRAM 的实际大小。

```
# define NV_RAM_SIZE      (0x1000)
```

5.9 内存的检验

由于在任何一个计算机系统中，内存都有着极为重要的功能，因此保证其正常工作后才能进行应用程序的开发。BSP 开发过程中，首先应该通过内存读写验证内存是否被正确初始化，可以采用合适的硬件仿真器来完成。在 VxWorks 操作系统运行起来之后，可使用操作系统的部分命令来查看具体内存单元的内容、已分配的内存空间和自由内存块，便于调试程序，检测内存泄漏，有效掌握内存的使用情况。Tornado 开发工具中提供了 Browser 这个强大的开发调试工具，它可以实时查看嵌入式系统中对象、模块和任务的内存使用情况，以数字和图形的方式显示内存统计信息，方便开发者合理调整内存的使用。

5.9.1 利用内存读写验证内存

VxWorks 映像通常包括 3 个部分：代码[text]段、数据[data]段和未赋初值数据[bss]段。因此可以利用未赋初值的数据来验证内存的读写是否正常。

```

int testVar;          /* bss 段的变量 */
.....
testVar = 15;
if (testVar != 15)
    somethingWrongWithRAM();

```

同样，当基于 ROM 的 VxWorks 中代码段和数据段在内存中完成重定位后，可以利用数据段的变量来验证数据段是否被正常初始化。

```

static int testVal = 13; /* data 段的变量 */
....
if (testVal != 13)
    somethingWrongWithData();

```

由此可以验证内存是否能够正常访问、数据段重定位是否正确等。

5.9.2 利用 d 和 memShow 命令查看内存

d 命令是 VxWorks 自带的内部命令，其函数原型为：

```

void d
(
    void * adrs,      /* address to display (if 0, display next block */
    int  nunits,      /* number of units to print (if 0, use default) */
    int  width        /* width of displaying unit (1, 2, 4, 8) */
)

```

)

该命令通常在 Shell 中使用，其 3 个参数可以部分忽略，或全部忽略。adrs 表示从什么地址开始显示，可以是实际地址，也可以是指针。如果 adrs 没有指定，将接着从上次 d 命令显示的结束地址开始显示。nunits 表示此次 d 命令显示多少个字节，默认为 256。width 表示一次连续显示的字节数，默认为 4。例如：

```
-> d 0x01ffe280
01ffe280: 3078 3164 3462 6361 3820 2874 5368 656c *0x1d4bca8 (tShel*
01ffe290: 6c29 3a20 6865 6c6c 6fee eeee eeee eeee *l): hello.....*
01ffe2a0: eeee eeee eeee eeee eeee eeee eeee eeee *.....*
01ffe2b0: eeee eeee eeee eeee eeee eeee eeee eeee *.....*
.....
01ffe370: eeee eeee eeee eeee eeee eeee eeee eeee *.....*
value = 21 = 0x15
->
```

还可以用 d 命令来显示 Flash 或 NVRAM 中的内容，便于应用程序的调试。

此外，也可以使用 VxWorks 中的一个组件“memory show routine”(INCLUDE_MEM_SHOW)。在 VxWorks 配置中包含该组件后，就可以在 Shell 中使用 memShow 命令来显示系统中内存的使用情况，包括当前已分配的和空闲的内存块数、字节数；累计分配的内存块数和字节数，例如：

```
-> memShow
status      bytes      blocks  avg block  max block
-----
current
  free  14812720         3  4937573  14795896
  alloc   137240        54    2541      -
cumulative
  alloc   137328        55    2496      -
value = 0 = 0x0
```

5.9.3 使用 Browser 查看内存

Browser 是 Tornado 环境中的一个开发调试工具。它是一个基于主机（即运行于主机上）的工具，因此和其他基于主机的开发调试软件一样，使用的前提是主机和目标机之间应该事先建立了 Target Server 的联系。Browser 可以用来查看内存的使用情况，每个任务分配的和实际使用的内存栈[stack]大小，以及系统对象（比如内存分区）的相关信息。Browser 的显示内容可以随时手动更新，或自动周期性更新。如图 5-5 所示显示了利用 Browser 工具查看系统内存分区的详细情况。

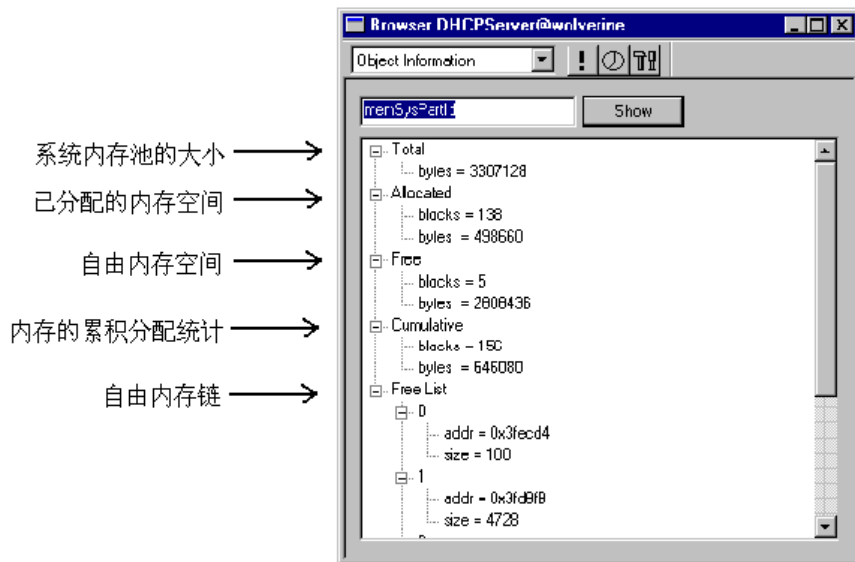


图 5-5 用 Browser 工具查看系统内存分区

如图 5-6 所示显示了利用 Browser 工具查看系统中每个模块使用内存的情况。可以看到每个模块及所有模块的代码段、数据段和未赋初值数据段的大小。

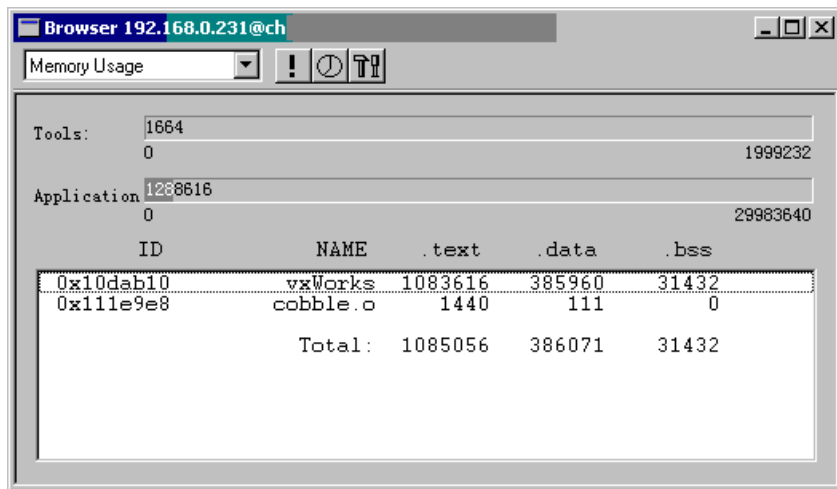


图 5-6 用 Browser 工具查看每个模块的内存使用

如图 5-7 所示显示了利用 Browser 工具查看系统中每个任务使用内存的情况。其中包括为每个任务分配的内存栈大小和当前实际使用大小。图中每个条形图最右边的数字表明指定给该任务的内存栈大小，最左边的数字表明该任务当前实际使用的内存大小，深色的进度条表示该任务运行过程中曾使用过的最大内存。

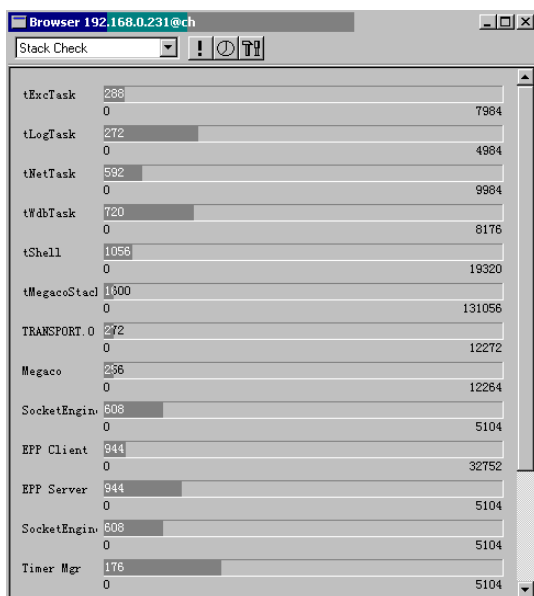


图 5-7 用 Browser 工具查看系统中每个任务的内存使用

VxWorks 程序员在调用 `taskSpawn` 函数生成任务时，往往不清楚应该为该任务分配多大的内存栈。如果通过 Browser 查看相应任务的内存栈信息，就能做到心中有数。如果某个任务实际使用的内存远小于为其指定的内存，则有必要减少指定内存的大小；反之，如果实际使用的内存接近指定内存的大小，则有必要增加指定内存。任务内存栈的大小是由 `taskSpawn` 函数的第 4 个参数决定。

5.10 常见问题解答

- 在内存中，系统内存池的起始地址是什么？

解答：系统内存池的起始地址 = `FREE_RAM_ADRS` + `WDB_POOL_SIZE` + `ISR_STACK_SIZE`

上述的宏定义见 `configAll.h`，该等式可在程序中使用，这些宏可在编译时可得到解析。

- 如何获得系统中空闲内存的总和？

解答：可以使用 `memPartInfoGet` 来获得一个内存分区的信息。该函数带两个参数，一个内存分区的 ID 值和一个内存分区的描述结构体指针（其定义参见 `memLib.h`）。结构体 `MEM_PART_STATS` 的成员 `memBytesFree` 表明了该内存分区的空闲字节数。系统内存分区的 ID 是 `memSysPartId`，是一个全局变量。

- 能否使用 `cacheDmaMalloc` 函数从指定地址来分配内存？

解答：`cacheDmaMalloc` 函数不具备这个功能。但是，可以通过一些方法来实现这个目的。

假设希望从地址 `0x0E80000` 开始分配内存，首先调用 `memalign` 从系统内存区中分配一块内存，其大小为 `size`，起始地址为 `0x0E80000`。

```
ptr = memalign (0x0E80000, size);
```

确保 `ptr` 的返回值是 `0x0E80000`。然后，把 `ptr` 作为参数传递给 `memPartCreate`，从这个地址开始创建一个内存分区。

```
PartID = memPartCreate(ptr, size of partition);
```

cacheDmaMalloc 函数的原型为:

```
cacheDmaMalloc (size_t);
```

用户可以对其进行修改, 使之带两个参数。

```
cacheDmaMalloc (PART_ID, size_t);
```

函数 cacheDmaMalloc 最终将调用函数 cacheArchDmaMalloc, 它也需要带两个参数。cacheArchDmaMalloc 函数中实际上是通过 _func_memalign 完成内存的分配, 而 _func_memalign 调用了 memPartAlignedAlloc, 因此可以在函数 cacheDmaMalloc 中直接调用 memPartAlignedAlloc, 而不再使用 _func_memalign。

● 如何将日志信息重定向到内存的某一个区域?

解答: 可以通过 memDrv 和 memDevCreate 将某块内存区域作为一个设备进行初始化和访问, 然后打开这个设备获得文件描述符, 调用 logFdSet 实现重定向, 例如在 Shell 中:

```
-> buf = malloc(1000)
-> memDevCreate "/mem1", buf, 1000
-> devs
drv name
  0 /null
  1 /tyCo/0
  4 sshenoypc:
  5 /vio
  6 /mem1
value = 25 = 0x19
-> fd = open("/mem1", 2, 0)
-> logFdSet fd
-> logMsg "hello"
-> d buf
01ffe280: 3078 3164 3462 6361 3820 2874 5368 656c *0x1d4bca8 (tShel*
01ffe290: 6c29 3a20 6865 6c6c 6fee eeee eeee eeee *1): hello.....*
01ffe2a0: eeee eeee eeee eeee eeee eeee eeee eeee *.....*
01ffe2b0: eeee eeee eeee eeee eeee eeee eeee eeee *.....*
```

● 如何创建一个内存分区, 并从这个内存分区中进行内存的分配? 函数 memPartCreate 的第一个参数是一个内存区域的指针, 如何获得这个指针? 除了调用 memPartAlloc 函数外, 还有什么需要注意的地方?

解答: 下面的例子中, pMemory 是内存池中的某个地址, 比如 0x300000。

```
-> partId=memPartCreate(pMemory, 100000)
-> ptr=memPartAlloc(partId, 200)
```

memPartAlloc 将会从这块内存块中分配出一定大小的内存区域。系统内存池的分区 ID 是一个全局变量 memSysPartId, malloc 从系统内存池中分配内存时, 实际上是调用了 memPartAlloc, 只不过其参数 partId 是 memSysPartId 而已。因此 malloc 只能用于从系统内存池中分配内存。

● 在 VxWorks 系统中, 如何“隐藏”一定大小的内存空间?

解答: 在 usrConfig.c 文件中的 usrInit 函数调用了 kernelInit 函数, VxWorks 系统内存总

和将完全由系统进行控制，而这个总和是由 kernelInit 函数的第 3 和第 4 个参数决定的。第 3 个参数是 VxWorks 映像的结束地址，第 4 个参数是 sysMemTop 的返回值。因此，可以以如下方式“隐藏”一块内存区域。

✧ 在 BSP 的 config.h 中定义 USER_RESERVED_MEM，例如：

```
#define USER_RESERVED_MEM 0x200000
```

✧ 在 usrConfig.c 中，把 kernelInit 由原来的：

```
kernelInit ((FUNCPTR) usrRoot, ROOT_STACK_SIZE, (char *) FREE_RAM_ADRS,  
sysMemTop (), ISR_STACK_SIZE, INT_LOCK_LEVEL);
```

改为：

```
kernelInit ((FUNCPTR) usrRoot, ROOT_STACK_SIZE, (char *) FREE_RAM_ADRS,  
(sysMemTop () - USER_RESERVED_MEM), ISR_STACK_SIZE, INT_LOCK_LEVEL);
```

- 是否能够把 x86 目标机上低端的 1MB 内存空间添加到系统内存池中？

解答：在 VxWorks 操作系统启动之后，x86 目标机上低端的 1MB 内存空间不会被使用，因此可以把它添加到系统内存池中，以供应用程序使用。但是需要注意的是，在这块内存区域中，某些内存是不能够使用的，有关这些内存使用的详细信息可参考“VxWorks Programmer's Guide”。首先需要忽略 IDT、GDT、意外消息、启动参数字符串和软盘的 DMA 区域，因此起始地址应该为 0x5000。结束地址应该忽略 ROM、显示 RAM，应该为 0xA0000。

调用 memAddToPool 把 0x5000~0xA0000 之间的内存区域添加到系统内存池中。

```
memAddToPool(0x5000, 0xA0000 - 0x5000);
```

用 memPartAddToPool 把它添加到某个指定的内存分区中。

```
memPartAddToPool(partId, 0x5000, 0xA0000 - 0x5000);
```

- BSP 的 sysLib.c 中，sysPhysMemDesc 数组的大小是否有最大限制？

解答：没有限制，取决于系统的内存。

- 内存分区的描述信息（比如已分配的内存块）存放于何处？是在该分区的开始位置，还是专为此分配了一个单独的内存区域来存放？

解答：每个内存分区包含了一个分区描述符和一块内存池。当使用 memPartCreate 创建内存分区时，分区描述符的存放位置是从系统内存池中进行分配的，即调用了 malloc。而系统内存分区的描述符则是在 VxWorks 内核建立过程中进行分配的。在系统内存池中有一个双向链表，保存了这些内存分区的内存块信息。

- 在某个内存分区中，如何获得已经分配的内存块的列表清单？

解答：可以使用 memPartShow 获得某个内存分区中未分配内存块的列表清单，结果在系统控制台显示，但是无法获得已经分配的内存块的列表清单，程序中也无法实现这一点。强烈建议，在任何情况下都不要操作系统内存池中的内存分区双向链表，即便是仅仅为了查看数据而无意对其进行修改，尽量使用 memPartLib 中的函数。对于一个内存分区，使用 memPartInfoGet 应该可以获得满意的结果。

- 调用 malloc 和 memPartAlloc () 函数时，有多大的额外开销？

解答：从内存池中分配内存块的时候，额外的开销有：内存块的头信息（包含双向链表，从系统内存池中分配）、对齐值、可分配的最小内存限制等。

● 开发工作结束之后，如何修改内存分区的选项，如何重新设置默认选项？

解答，默认情况下设置的选项有：MEM_ALLOC_ERROR_LOG_FLAG、MEM_BLOCK_CHECK、MEM_BLOCK_ERROR_LOG_FLAG 和 MEM_BLOCK_ERROR_SUSPEND_FLAG。

可以使用 memPartOptionsSet 函数修改内存分区的选项，例如：

```
memPartOptionsSet (myPart, MEM_BLOCK_CHECK | MEM_BLOCK_ERROR_FLAG);
```

第 2 个参数应该是逻辑与，如果不需要任何选项，它可以为 0。需要注意以下几点：

✧ memPartOptionsSet 的第 2 个参数不要使用逻辑非（~）的表达式，否则会设置一些不希望的选项。

✧ 设置 MEM_BLOCK_ERROR_LOG_FLAG 或 MEM_BLOCK_ERROR_SUSPEND_FLAG 选项时，MEM_BLOCK_CHECK 选项也会被隐含地设置。

✧ memOptionsSet (options) 等价于 memPartOptionsSet (memSysPartId, options)。

● 我想创建一个内存分区，使其包含 12 个 128 字节的内存块。但是创建 12 * 128 大小的分区是不行的，因为创建分区时有一些额外的开销。分配每个内存块和创建内存分区时各有多大开销？如果调用 memPartAlloc 函数来完成，那么传递给 memPartCreate 函数的参数应该是什么？

解答：首先要保证内存分区从字节对齐边界开始创建。对于 PowerPC 系列，可以根据文件“target/h/arch/ppc/toolPpc.h”来检验对齐。

```
#define _STACK_ALIGN_SIZE      8      /* stack alignment */
#define _ALLOC_ALIGN_SIZE      8      /* allocation alignment */
```

内存分区和内存块都使用相应的结构体来进行描述，这些结构体的大小就是分区和块的额外开销。分区开销是 16 个字节，内存块头的开销是 8 个字节。此外，如果所要分配的内存块不是 _ALLOC_ALIGN_SIZE 的偶数倍，那么需要一个额外的内存块来实现对齐。因此，为了在一个分区内创建 12 个 128 字节的内存块，需要的内存大小为 $((128+8) \times 12) + 16 = 1648$ ，这样分配的内存就没有任何浪费了。

```
-> memPartCreate (0x800000, 1648)
value = 33547848 = 0x1ffe648
-> memPartAlloc (0x1ffe648, 128)
value = 8390120 = 0x8005e8
-> memPartAlloc (0x1ffe648, 128)
value = 8389984 = 0x800560
-> memPartAlloc (0x1ffe648, 128)
value = 8389848 = 0x8004d8
..... (总共 12 次调用 memPartAlloc () 函数)
```

● 有的应用程序在运行的时候，需要一块较大的内存。但是当进行内存请求时，由于已经存在内存碎片，导致不能分配到所需的一块连续内存区域，返回的错误可能是“memPartAlloc: block too big”，如何解决这个问题呢？

解答：解决这个问题的方法是在 VxWorks 启动过程中，尽快获得所需要的内存并把它作为一

个私有的内存分区，这样可以避免这块内存由于碎片的存在而变得不连续。进行该操作比较合理的位置是 `usrRoot` 函数的末尾。如果 `VxWorks` 是在命令行下编译的，则修改“`target/config/all/usrConfig.c`”文件。如果是在 `Tornado` 集成开发环境中，通过建立 `Bootable` 工程而生成的 `VxWorks`，那么应该在该工程 `usrAppInit.c` 文件的 `usrAppInit` 函数的开始处。因为系统运行至此时，`VxWorks` 的内核正常运行，系统的各种资源可以正常使用，但用户应用程序还没有开始使用系统资源，此时的内存碎片最少。操作步骤如下：

✧ 在 `usrConfig.c` 或 `usrAppInit.c` 进行全局申明，注意包含头文件 `vxworks.h` 和 `memLib.h`。

```
#define MY_PART_SIZE 0x200000
void * memPtr; /* pointer to where to place my memory partition */
PART_ID myMemPart; /* memory partition id for my partition */
```

✧ 在 `usrRoot` 函数结束处，或 `usrAppInit` 函数开始处添加代码：

```
memPtr = malloc(MY_PART_SIZE);
myMemPart = memPartCreate(memPtr, MY_PART_SIZE);
```

`malloc` 函数从系统内存池中分配一块 2MB 空间的连续内存，应该对其返回值进行判断。`memPartCreate` 函数把这块内存空间创建为一个内存分区，此后其他应用程序对内存的申请或释放不会影响该内存分区。用户可以使用 `memPartLib` 中的函数对这个内存分区进行操作。

✧ `VxWorks` 启动后，可以在 `Shell` 中进行操作和验证。

```
-> memPartShow(myMemPart)
status  bytes    blocks  avg block  max block
-----
current
  free   2097136      1    2097136    2097136
  no allocated blocks
cumulative
  no allocated blocks
value = 0 = 0x0
-> myMemPtr = memPartAlloc(myMemPart, 2000000)
new symbol "myMemPtr" added to symbol table.
myMemPtr = 0xffe7d0: value = 13902724 = 0xd42384
-> memPartShow(myMemPart)
status  bytes    blocks  avg block  max block
-----
current
  free     97128      1     97128     97128
  alloc   2000008      1    2000008      -
cumulative
  alloc   2000008      1    2000008      -
value = 0 = 0x0
-> memPartFree(myMemPart, myMemPtr)
value = 0 = 0x0
-> memPartShow(myMemPart)
status  bytes    blocks  avg block  max block
-----
current
  free   2097136      1    2097136    2097136
  no allocated blocks
cumulative
```

```
no allocated blocks  
value = 0 = 0x0
```

第 6 章 I/O 系统

本章介绍了 I/O 系统的总体结构、操作接口和实现机制，并对文件、设备、驱动等基本概念做了解释，描述了它们和 I/O 系统之间的关系。本章最后以串行驱动为例，介绍了字符设备驱动的结构、接口和实现机制。块设备驱动将在本书的第 7 章中做介绍。

6.1 概述

本节介绍了 I/O 系统的基本结构，并解释了一些文件相关的基本概念。

6.1.1 系统结构

VxWorks 的 I/O 系统的基本结构如图 6-1 所示。

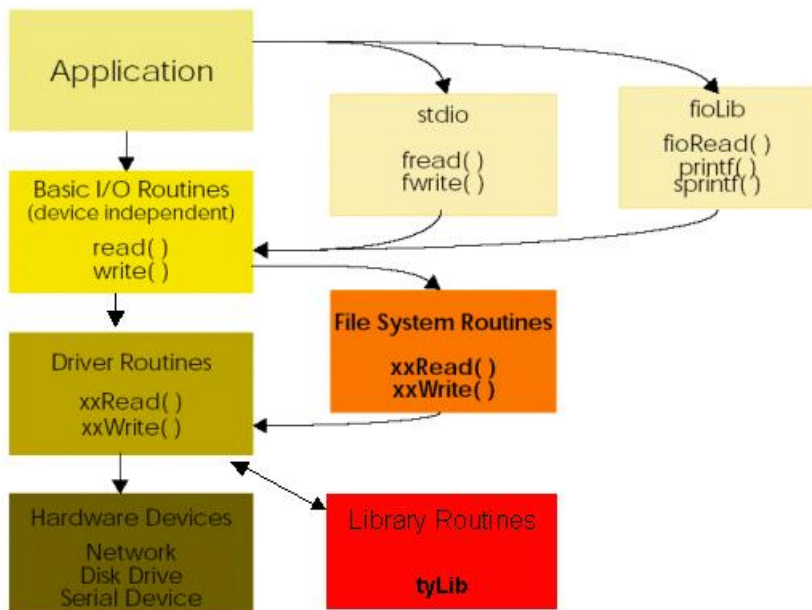


图 6-1 I/O 系统结构

如图 6-1 所示左列为函数调用主线，应用程序下面是 I/O 系统中的基本 I/O 库，此例程库接口与 Unix 兼容，为各种设备提供了简单、统一，与设备无关的访问接口。基于基本 I/O，VxWorks 同时提供了与 ANSI C 相兼容的带缓存 I/O 接口 (stdio) 和格式 I/O 接口 (fioLib)，如图中上部所示。

在基本 I/O 库下面是驱动层，驱动层在具体的设备上实现实际的 I/O 功能(如 read 和 write 等)。VxWorks 的 I/O 设备和 Unix 的 I/O 设备很相似，它们都分为字符设备和块设备。

✧ 字符设备是指以字符为单位进行数据传输的设备，如串口等。

✧ 块设备是指以“块”为单位对数据进行存取的设备，如存储设备等。

字符设备驱动可以直接通过基本 I/O 接口挂接到 I/O 系统中。而块设备驱动则不同，在块设备驱动和 I/O 系统之间存在文件系统层，如图中部所示。文件系统本身是“字符设备”，它按照字符设备驱动程序接口与 I/O 系统挂接，同时它建立在块设备驱动上，I/O 系统通过文

件系统调用块设备驱动，最终访问块设备。块设备的介绍可参见本书的第 7 章。

6.1.2 文件

与 Unix 相同，VxWorks 所有的 I/O 设备都被当做文件来存取。这儿所说的文件和文件系统没有必然的联系，只是 I/O 系统中的操作对象。文件可能指一个物理设备，或一个任务管道，或文件系统中的文件。例如，“/user/myfile”表示一个名为“/user”的磁盘设备上的文件 myfile；“/pipe/mypipe”表示一个名为 mypipe 的管道（VxWorks 中管道名都以“/pipe”开始）；“/tyCo/0”表示一个物理串口。

6.1.3 文件描述符

在基本的 I/O 调用中，文件通过文件描述符来引用，文件描述符是 open 或 creat 函数返回的小整数。其他的 I/O 操作以文件描述符做为参数来指定对应的文件。一个文件描述符并不属于某个特殊任务，而是属于 I/O 系统的全局句柄。

当一个文件被打开后，一个文件描述符被分配并返回。当文件被关闭后，文件描述符被释放。在 VxWorks 中，文件描述符的数目是有限制的，所以为了避免打开的文件数超过系统的限制，当一个打开的文件不再使用后，应该及时关闭它。事实上，VxWorks 在内部维护了一张文件描述符表，文件描述符是这张表的索引。

6.1.4 标准文件描述符

在 I/O 系统中，值 0、1、2 被保留为特殊描述符，并且不可能成为 open 或 creat 的返回，这就是分配得到的描述符永远都大于 2 的原因。其中，0 表示标准输入；1 表示标准输出；2 表示标准错误输出。

但是，VxWorks 支持 I/O 重定向，也就是说用户可以将上述 3 个描述符重新定向到任何一个自己喜欢的 I/O 设备上，如串口、Socket 和文件等，这样的好处是便于跟踪调试。

这里有两个层次的重定向。首先，对于这 3 个描述符有全局的变量，在缺省情况下，新任务都是使用这些全局变量。可以通过函数 ioGlobalStdSet 和 ioGlobalStdGet 对其进行存取。ioGlobalStdSet 将全局的标准输入、输出和错误输出重定向到其他描述符上。ioGlobalStdGet 返回系统当前全局的标准输入、输出和错误的描述符，两个函数原型如下。

```
void ioGlobalStdSet
(
    int stdFd, /* std input (0), output (1), or error (2) */
    int newFd /* new underlying file descriptor */
)
int ioGlobalStdGet
(
    int stdFd /* std input (0), output (1), or error (2) */
)
```

其次是任务级的，也就是说对于每个任务都可以单独指定自己的标准输入、输出和错误输出的描述符，更改它只对本任务有效。可以通过函数 ioTaskStdSet 和 ioTaskStdGet 对其进行存取，两个函数原型如下。

```
void ioTaskStdSet
(
    int taskId, /* task whose std fd is to be set (0 = self) */
)
```

```
int stdFd, /* std input (0), output (1), or error (2) */
int newFd /* new underlying file descriptor */
)
int ioTaskStdGet
(
    int taskId, /* ID of desired task (0 = self) */
    int stdFd /* std input (0), output (1), or error (2) */
)
```

系统启动后全局的标准输、输出和错误输出缺省被重定向到控制台，读者可参考 `usrConfig.c` 中 `usrRoot` 函数的代码实现。

6.2 I/O 接口

VxWorks I/O 系统提供了多种标准接口供开发者使用，包括基本 I/O（`ioLib`）、缓存 I/O 和格式 I/O。

6.2.1 基本 I/O

基本 I/O 接口共有 7 个，应用程序通过这些接口来访问设备。设备驱动也遵循该接口规范，提供具体的实现，如图 6-2 所示。

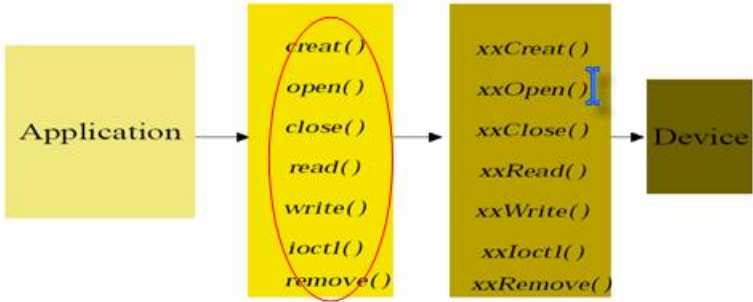


图 6-2 基本 I/O 接口

- `open`

`open` 打开一个文件，得到文件描述符（`fd`）。在对一个文件操作之前，必须首先打开它，类似的函数还有 `creat`。其中 `flags` 为文件存取标志，如表 6-1 所示。

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

表 6-1 `open` 函数 `flags` 值

Flag	值	说明
<code>O_RDONLY</code>	0	只读打开
<code>O_WRONLY</code>	1	只写打开
<code>O_RDWR</code>	2	读写打开
<code>O_CREAT</code>	200	建立新文件
<code>O_TRUNC</code>	400	合并文件

在通常情况下，`open` 只能用来打开一个已经存在的设备或文件，但是对于 NFS 网络设备或者 `dosFs` 和 `rt11Fs` 的块设备来说，也可以用 `O_CREAT` 标志来调用 `open`，建立一个新文件。对于 NFS 网络设备，当用 `open` 建立新文件时，要求填写参数 `mode`，如下所示。

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

对于 NFS 或者 dosFs 设备,也可以通过 O_CREAT 标志来建立目录,mode 需设为 FSTAT_DIR。

- **close**

close 用于关闭文件,释放文件描述符。需要注意的是,当任务退出或被删除后,被它打开的文件并不会自动关闭,所以在任务退出或被删除前,一般需要调用 close 关闭已打开的文件。

```
close(fd);
```

- **creat**

creat 在文件系统上创建一个新文件并且用指定的标志打开它。

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

这里 flag 可以设置为 O_RDONLY (0)、O_WRONLY (1) 或 O_RDWR (2)。对于 NFS 网络设备,建立新文件要使用带 mode 参数的 open 调用。对于非文件系统的设备,creat 与 open 相同。

- **remove**

remove 用于删除文件系统中的文件。当文件打开后,不允许被删除。对于非文件系统的设备来说,remove 无效。

```
remove("name");
```

- **read**

用 open 或 creat 得到文件描述符后,可以通过 read 读取数据。

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

read 返回实际所读取的字节数。对于基于文件系统的设备,如果返回的字节数 nByte 小于请求的最大个数 maxBytes,随后的 read 将返回 0 表示文件结束。对于其他设备(比如串口或 TCP 的 Socket),即使可读取的数据量大于请求的个数,实际读取的数据量也有可能小于请求的个数,所以需要循环调用 read 来读取数据。

- **write()**

与 read 对应,write 对文件进行写操作,函数返回实际写入的字节数,如下所示:

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

- **ioctl**

ioctl 对文件(设备)执行各种 I/O 控制操作。所有的操作命令在头文件 ioLib.h 中定义。实际上它调用驱动中的 xxIoctl,因此 ioctl 的完成依赖于下一级驱动层的实现,具体的操作分别在 tyLib、pipeDrv、nfsDrv、dosFsLib、rt11FsLib 和 rawFsLib 中描述。例如设置串口波特率:

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

6.2.2 带缓存 I/O

在基本 I/O 接口的基础上,VxWorks 提供了符合 ANSI C stdio 定义的带缓存 I/O 包。要使用 VxWorks 的 stdio,需要在工程中包括 INCLUDE_ANSI_STDIO 组件。

带缓存 I/O 就是在基本 I/O 的基础上增加了一层缓存机制,应用程序的数据读写通过此缓存[Buffer]来中转,如图 6-3 所示。

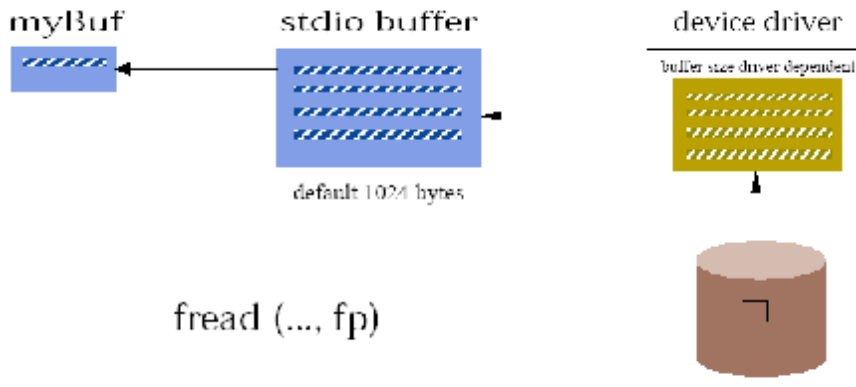


图 6-3 Buffered I/O

缓存对应用程序是透明的，属于 `stdio` 库私有，完全由 `stdio` 库函数自己使用。在 `stdio` 包中，对文件的存取使用文件指针（指向 `FILE` 结构）来代替文件描述符。`FILE` 结构在 `stdio.h` 中定义，包括文件描述符和指向缓存的指针，例如读文件如下所示：

```
FILE *fp = fopen("name", "r");
```

对应一个已经打开的文件描述符，也可以与文件指针相关联。

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

得到文件指针后，就可以使用 `fread` 读取数据块或者用 `getc` 读取字符，使用 `fwrite` 写数据块或者用 `putc` 写字符。在使用 `fclose` 关闭文件后，文件指针相关的缓存被释放。文件指针相关的缓存并不被信号量保护，所以不同的任务不能够使用相同的文件指针。

使用缓存机制主要是基于提高系统效率的考虑。对于基本 I/O 系统来说，每一次调用必然会有一定的系统开销。首先，每一次都需要传递调用到下层驱动的相应例程中；其次，大多数驱动层对于多任务的访问都增加了互斥或者排队的机制来保护驱动层缓冲区。这样，如果在应用层频繁地直接进行基本的 I/O 调用，则系统开销就会变得较大。而在 `stdio` 包中，增加缓存以后，应用程序每次只是高效率地与缓存打交道，直到读缓存空后才进行 `read` 调用，或者直到写缓存满后才进行 `write` 调用。

6.2.3 格式 I/O

VxWorks 提供了 `fioLib` 例程库用于格式 I/O。需要注意的是，`printf`、`sprintf` 和 `sscanf` 等在 ANSI C 中通常作为 `stdio` 库的函数，而 VxWorks 将它们定义到 `fioLib` 库中，不需要定义 `INCLUDE_ANSI_STDIO` 就可以使用这些函数。VxWorks 的 `fioLib` 库使用了一个格式化的、无缓存的接口。`fioLib` 库还包括一些特别的函数，如表 6-2 所示。

表 6-2

`fioLib` 函数

函数	描述
<code>printErr</code>	输出格式化信息到标准错误流，语法与 <code>printf</code> 相同
<code>fioRead</code>	读数据，实际上是循环调用 <code>read</code> ，直到最大的请求数据或文件结束
<code>fioRdString</code>	从文件中读取字符串
<code>fdprintf</code>	输出格式化的字串到指定的文件描述符中

6.3 I/O 内部管理

如图 6-4 所示为 I/O 系统、设备和驱动间的管理关系，可以看出，I/O 系统可以管理若干个设备驱动程序；I/O 系统和设备驱动程序之间的接口模型就是字符设备驱动程序接口；一个驱动程序处理所有同类型的设备，这些设备可能只是参数不同，例如 I/O 地址和中断号等。

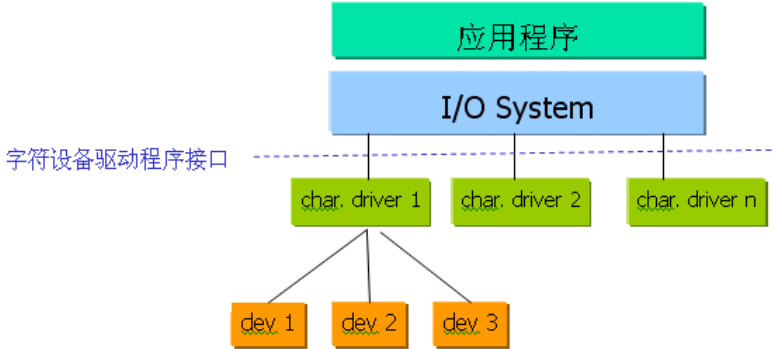


图 6-4 I/O 系统管理层次

如图 6-5 所示来自 WindRiver，表示的是各模块结构，启动流程以及彼此间的调用关系。

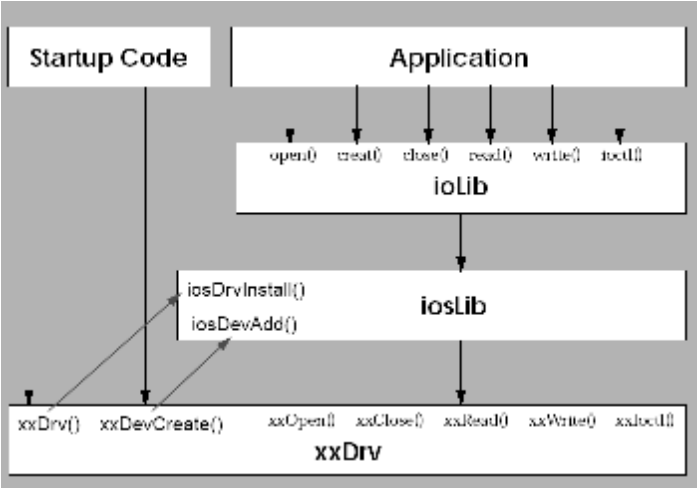


图 6-5 I/O 系统调用关系

本节主要以字符设备驱动为主，详细说明 I/O 系统的内部管理机制。块设备相关部分参见本书的第 7 章。

6.3.1 系统启动

在系统的启动过程中，VxWorks 按如下步骤初始化 I/O 系统和加载设备。

- ✧ 调用 iosInit 初始化 I/O 系统。
- ✧ 调用 xxDrv 初始化设备驱动程序。
- ✧ 调用 xxDevCreate 创建设备（可能有若干个）。

上述调用在 bootConfig.c（对于 BootRom）或 prjConfig.c（对于 VxWorks）中实现。

6.3.2 I/O 系统初始化

在 usrRoot 中，VxWorks 首先调用 iosInit 初始化 I/O 系统。

```
iosInit(NUM_DRIVERS, NUM_FILES, "/null");
```

函数中参数解释如下，参数可以在 Tornado 组件配置中修改。

- ✧ NUM_DRIVERS 是最多能安装的设备驱动程序的个数(在 configAll.h 中定义为 20)。
- ✧ NUM_FILES 是同时能够打开的文件(设备)的最大数(在 configAll.h 中定义为 50)。
- ✧ “/null” 是空设备名称。

6.3.3 驱动程序初始化

驱动程序的初始化由 xxDrv 完成。它首先调用 iosDrvInstall 来向 I/O 系统注册自己，iosDrvInstall 的参数为 7 个基本 I/O 接口的指针，在成功注册后，iosDrvInstall 返回一个非 0 的驱动号[driver number]，然后完成一些数据结构的分配，中断连接等。

```
static int xxDrvNum = 0;
STATUS xxDrv(void)
{
    if (xxDrvNum > 0)    return OK;
    if ((xxDrvNum = iosDrvInstall(xxCreat, 0, xxOpen,
                                0, xxRead, xxWrite, xxIoctl)) == ERROR)
        return ERROR;
    intConnect (intvec, xxInterrupt, ...);
    ...
    return OK;
}
```

事实上，在 I/O 系统内部，VxWorks 通过一张表来管理已注册的驱动，称之为驱动程序表，驱动程序表的大小是固定的，有 NUM_DRIVERS 项，如图 6-6 所示。

DRIVER CALL:

```
drvnum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl);
```

[1] Driver's install routine specifies driver routines for seven I/O functions.

[2] I/O system locates next available slot in driver table.

[4] I/O system returns driver number (drvnum = 2).

DRIVER TABLE:

	create	remove	open	close	read	write	ioctl
0							
1							
2	xxCreat	0	xxOpen	0	xxRead	xxWrite	xxIoctl
3							
4							

图 6-6 驱动程序初始化示例

- ✧ xxDrv 调用 iosDrvInstall 注册自己。
- ✧ I/O 系统在驱动程序表中寻找空闲的位置，得到表项 2。
- ✧ I/O 系统在表项 2 中保存驱动程序的各个 I/O 接口指针，指针为 0 表示无此项操作。

✧ 返回表项的索引 2，也就是驱动程序号。

 在 Shell 下，通过 iosDrvShow 可以显示系统驱动程序表。

```
-> iosDrvShow
drv      create      delete      open      close      read      write      ioctl
1      1f2da0          0      1f2da0      1f2dd0      1bc688      1bc5a8      1f2e00
2      108518          0      108518          0      1bc688      1bc5a8      108538
3      1f5bc0      1f5f80      1f5880      1f5c50      1f6860      1f6880      1f7020
4          0          0          0      1c9f94      1cadc4      1ca824      224978
5      1d3b58      1d3cb8      1d3ec8      1d3f98      1d4ca8      1d4e28      1d4f98
6      239d88          0      239d88      239d98      239de8      239e78      239ed8
7      239d78          0      239d78      239dc8      239e08      239e98      239ef8
8      1e84e0          0      1e84e0      1e85b0      1bc688      1bc5a8      1e8630
```

6.3.4 创建设备

在完成驱动程序初始化后，系统调用 xxDevCreate 函数来创建此驱动程序服务的设备（可能有若干个）。xxDevCreate 初始化该设备，填充设备信息，其设备信息由函数参数传入，然后向 I/O 系统注册该设备。

```
STATUS xxDevCreate(char *devName, ...)
{
    XX_DEV *pXxDev;
    if (xxDrvNum == 0) {
        errno = S_ioLib_NO_DRIVER;
        return ERROR;
    }
    if ((pXxDev = (XX_DEV *)malloc(sizeof(XX_DEV))) == NULL)
        return ERROR;
    /* todo: 将设备信息保存到 pXxDev 结构中 */
    /* todo: 设备特定的初始化代码 */
    if (iosDevAdd((DEV_HDR *)pXxDev, devName, xxDrvNum) == ERROR) {
        /* un-initialize code here */
        free(pXxDev);
        return ERROR;
    }
    return OK;
}
```

在系统内部，VxWorks 使用一个设备列表[device list]来维护已经注册的设备。此设备列表的数据结构为 XX_DEV，一般在驱动程序的头文件中定义，例如 xxDrv.h 中。XX_DEV 结构头部是一个 DEV_HDR 结构（类似 OO 中继承），DEV_HDR 结构是双向链表的节点，I/O 系统使用它来维护设备，XX_DEV 结构的剩下部分被驱动程序用来保存设备信息。XX_DEV 结构定义如图 6-7 所示。

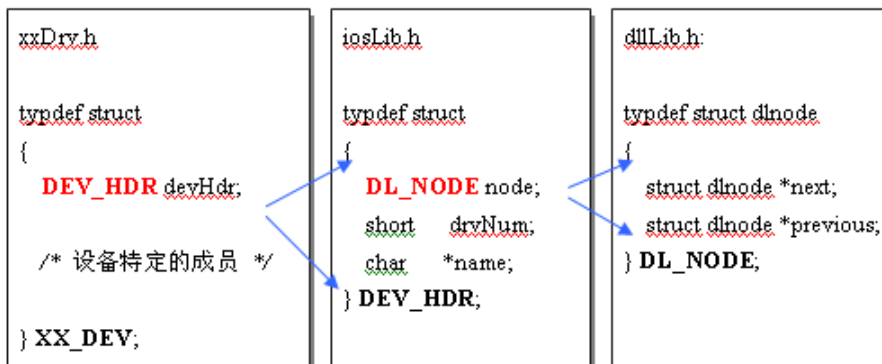


图 6-7 设备列表数据结构

如图 6-8 所示是 xxDevCreate 调用 iosDevAdd 增加设备的例子。

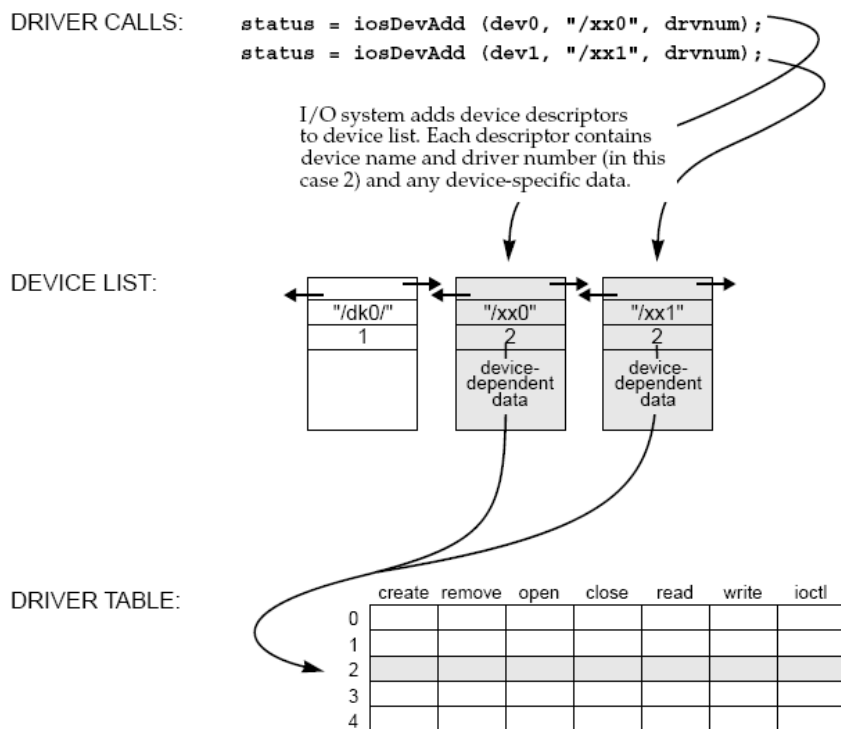


图 6-8 增加设备示例

在 Shell 下，通过 devs 或 iosDevShow 可以显示系统中的设备列表及对应的驱动程序号。

```

-> devs
drv  name
0   /null
2   /pcConsole/0
2   /pcConsole/1
5   host:
6   /pty/telnet.S
7   /pty/telnet.M
8   /vio
3   /dsxxxx/

```

6.3.5 设备的访问

用户可通过标准文件描述符来访问归入 I/O 系统的设备，与文件的操作类似。VxWorks 中的字符设备驱动支持 select 函数，详细的描述参考“Vxworks 5.4 Programmer's Guide”的 3.4.1 章节。

● 文件描述符表

要访问设备，首先需要调用 open 或 creat 打开设备。系统根据参数中的设备名查找设备列表，找到匹配的设备节点后，根据节点信息中的驱动程序号查找驱动程序表，从而取得驱动程序中相对应的入口例程。在第一次打开后，系统通过一张表来保存设备与驱动的这种关联，供后面的 I/O 调用使用，称之为文件描述符表（FD_table），如图 6-9 所示。文件描述符表的每一项记录了与设备对应的驱动程序号，而表项的索引就是函数调用后所返回的文件描述符。

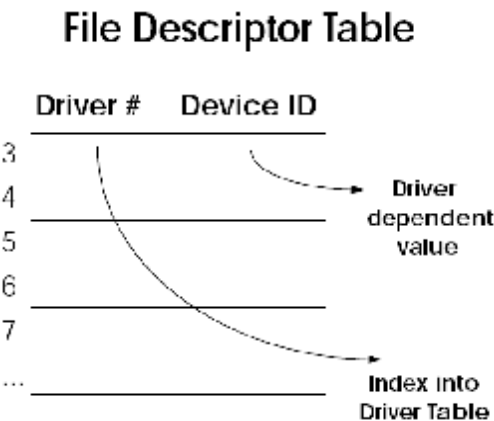


图 6-9 文件描述符表

文件描述符表大小是固定的，总共有 NUM_FILES 项，也就是同时能够打开的文件（设备）的最大个数。在 configAll.h 中，NUM_FILES 缺省定义为 50。在工程中，可以通过“IO system”组件来修改 NUM_FILES 定义。文件描述符从 3 开始分配，这是因为 0、1、2 已经被保留定义为标准输入、输出和错误输出。

 在 Shell 下可通过 iosFdShow 来显示文件描述符表。

```
-> iosFdShow
fd name          drv
3 /pcConsole/0   2
4 (socket)       4
5 (socket)       4
6 (socket)       4
7 (socket)       4
8 /pty/telnet.M  7
9 /pty/telnet.S  6
10 (socket)      4
11 (socket)      4
```

● 打开设备

调用 open 打开设备 “/xx0” 的流程如图 6-10 所示。

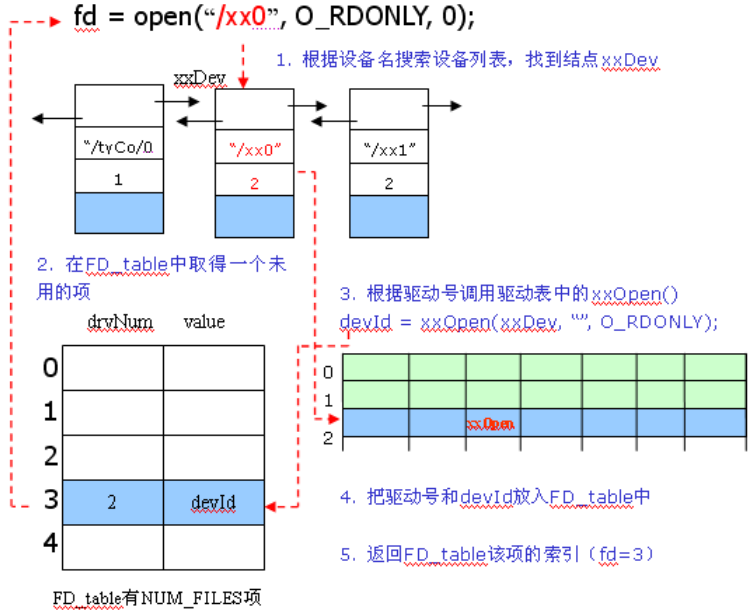


图 6-10 open 调用示例

根据设备名搜索设备列表，找到结点 `xxDev`；在 `FD_table` 中取得一个未用的项；根据驱动号调用驱动表中的 `xxOpen`；把驱动号和 `devId` 放入 `FD_table` 中；最后返回 `FD_table` 该项的索引 (`fd=3`)。

● 从设备读数据

如图 6-11 所示表示了调用 `read` 从一个已打开的设备（文件描述符为 `fd`）读取数据的流程。

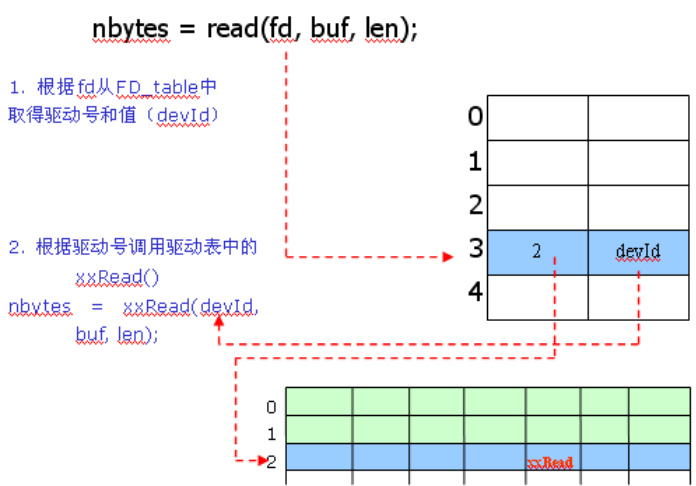


图 6-11 read 调用示例

根据 `fd` 从 `FD_table` 中取得驱动号和价值 (`devId`)，调用驱动表中的驱动号对应的 `xxRead`。

```
nbytes = xxRead(devId, buf, len);
```

需要注意的是，I/O系统只负责调用相应的xxRead或xxWrite，具体操作由驱动来完成。

读写操作完成后，用户需要调用 close 来关闭设备。类似于 read，系统首先根据 fd 调用 xxClose，之后释放文件描述符表中 fd 对应的表项，使得随后的 open 调用可用。

● 卸载设备与驱动

VxWorks 支持设备和驱动的卸载，使用 iosLib 库函数 iosDevDelete 来卸载设备，使用 iosLib 库函数 iosDrvRemove 来卸载驱动，两个函数的原型如下所示：

```
void iosDevDelete
(
    DEV_HDR * pDevHdr /* pointer to device's structure */
)
STATUS iosDrvRemove
(
    int  drvnum,      /* no. of driver to remove, returned by iosDrvInstall() */
    BOOL forceClose /* if TRUE, force closure of open files */
)
```

● I/O 系统库

I/O 系统库[iosLib]包含的函数如表 6-3 所示。

表 6-3 iosLib 函数

函数	描述
iosInit	初始化 I/O 系统
iosDrvInstall	安装 I/O 驱动
iosDrvRemove	卸载 I/O 驱动
iosDevAdd	增加设备到 I/O 系统
iosDevDelete	卸载设备
iosDevFind	在设备列表中查找指定设备
iosFdValue	验证一个 fd 是否合法并返回其对应的驱动号

● 完整的字符设备驱动程序模板

根据前几节对 I/O 系统内部的描述，很容易得出一个标准的字符设备驱动模板。

```
/******
* xxDrv - driver initialization routine
*
* xxDrv() initializes the driver. It installs the driver via iosDrvInstall.
* It may allocate data structures, connect ISRs, and initialize hardware.
*/
STATUS xxDrv ()
{
    xxDrvNum = iosDrvInstall (xxCreat, 0, xxOpen, 0, xxRead, xxWrite, xxIoctl);
    (void) intConnect (intvec, xxInterrupt, ...);
    ...
}

* xxDevCreate - device creation routine
```

```

*
* Called to add a device called <name> to be serviced by this driver. Other
* driver-dependent arguments may include buffer sizes, device addresses...
* The routine adds the device to the I/O system by calling iosDevAdd.
* It may also allocate and initialize data structures for the device,
* initialize semaphores, initialize device hardware, and so on.
*/
STATUS xxDevCreate (name, ...)
    char * name;
    ...
    {
        status = iosDevAdd (xxDev, name, xxDrvNum);
        ...
    }
/*****
* The following routines implement the basic I/O functions. The xxOpen()
* return value is meaningful only to this driver, and is passed back as an
* argument to the other I/O routines.
*/
int xxOpen (xxDev, remainder, mode)
    XXDEV * xxDev;
    char * remainder;
    int mode;
    {
        /* serial devices should have no file name part */

        if (remainder[0] != 0)
            return (ERROR);
        else
            return ((int) xxDev);
    }
int xxRead (xxDev, buffer, nBytes)
    XXDEV * xxDev;
    char * buffer;
    int nBytes;
    ...
int xxWrite (xxDev, buffer, nBytes)
    ...
int xxIoctl (xxDev, requestCode, arg)
    ...
/*****
* xxInterrupt - interrupt service routine
*
* Most drivers have routines that handle interrupts from the devices
* serviced by the driver. These routines are connected to the interrupts
* by calling intConnect (usually in xxDrv above). They can receive a
* single argument, specified in the call to intConnect (see intLib).
*/
VOID xxInterrupt (arg)
    ...

```


6.4 VxWorks 的设备和驱动

VxWorks 提供了丰富的设备和驱动，如表 6-4 所示。

表 6-4

VxWorks 的设备和驱动

设备	驱动	设备创建	设备名	描述
Tty	ttyDrv	ttyDevCreat	“/tyCo/”	终端设备
Pty	ptyDrv	ptyDevCreate	“/pty/”	伪终端设备（终端仿真）
Pipe	pipeDrv	pipeDevCreate	“/pipe/”	管道
Mem	memDrv	memDevCreate	“/mem/”	伪内存设备
Nfs	nfsDrv	nfsMount	“/”	网络文件系统设备
Remote	netDrv	netDevCreate	以 “:” 结尾	通过 RSH 或 FTP 存取远程主机文件
Ram disk	ramDrv	ramDevCreate	大写字母后面跟数字再跟 “:”，例如 “DEV1:”	内存磁盘
scsiLib				SCSI 接口库
—	xxDrv	xxDevCreate	“/”	其他的设备与驱动

其中与文件系统相关的设备和驱动的具体介绍可参见本书的第 7 章。

6.4.1 串行 I/O 设备

在 VxWorks 中，串行 I/O 设备驱动并不直接接入 I/O 系统中，而是通过在其上构建的 tty 驱动层接入。tty 驱动管理所有的串行 I/O 设备，提供标准的 I/O 接口接入 I/O 系统，并统一处理相关的终端功能，如命令行编辑、字符回显等。

pty 类似于 tty，只不过它对应的是一个虚拟的终端设备（终端仿真），比如 remote login，而 tty 对应的是一个真实的终端设备。

通过 ioctl 的功能码 FIOSETOPTIONS，可以设置 tty(pty)的各种模式，如表 6-5 所示。

表 6-5

tty 模式设置

模式	描述
OPT_LINE	选择行模式
OPT_ECHO	回显输入的字符
OPT_CRMOD	转换输入的字符 RETURN 为 NEWLINE(\n);转换输出的 NEWLINE 为 RETURN—LINEFEED
OPT_TANDEM	支持 X-on/X-off 协议(CTRL+Q 和 CTRL+S)
OPT_7_BIT	将输入的字符转换为 7 位 ASCII 码
OPT_MON_TRAP	特别的 ROM 驻留程序字符有效，缺省 CTRL+X
OPT_ABORT	支持 Shell 停止字符，缺省为 CTRL+C
OPT_TERMINAL	设置上述所有的终端属性
OPT_RAW	取消上述所有的终端属性

tty (pty) 支持的 ioctl 功能码如表 6-6 所示（在 tyLib 中定义）。

表 6-6

tty ioctl 功能码

功能	描述
FIOBAUDRATE	设置波特率
FIOCANCEL	取消读或写
FIOFLUSH	扔弃所有输入/输出缓冲区里的字符
FIOGETNAME	取得文件描述符对应的文件名
FIOGETOPTIONS	取得当前设备的控制字
FIONREAD	取得输入缓冲区内未读取字符的个数
FIONWRITE	取得输出缓冲区内字符个数
FIOSETOPTIONS	设置设备控制字

tty (pty) 设备通常工作在两种模式下，raw 模式（无缓冲）或 line 模式（行模式）。raw 模式是缺省的模式，line 模式可由上面的 OPT_LINE 选项来设置。在 raw 模式下每个输入的字符在用户输入时就立即生效，一个工作在 raw 模式的 tty (pty) 设备，除非控制可选项，否则用户无法对正在进行的输入做修改。而对于 line 模式，所有的输入字符都被保存在缓冲区中直到一个 NEWLINE 字符被输入，并且在输入的过程中可以使用特殊字符 (CTRL+字符) 来修改输入。在 Tornado 的 Shell 下的输入就是一个很好的 line 模式的例子。

6.4.2 管道设备

管道[Pipe]是一个任务间通过 I/O 系统进行通信的虚设备。任务写消息到管道，消息可以被其他任务读取。

用函数 pipeDevCreate 建立管道设备，原型如下：

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

其中参数为设备名称、最大消息个数和每个消息的最大长度。

使用管道，中断程序[ISR]可以和任务进行通信。在 VxWorks 的设备中，除了管道外，都不允许在中断中进行访问，中断只能对管道进行 write 调用。当管道满以后，消息将被丢弃，因为中断不允许被阻塞。管道的 ioctl 功能码定义如表 6-7 所示。

表 6-7 pipe ioctl 功能码

功能	描述
FIOFLUSH	丢弃管道中所有的消息
FIOGETNAME	取得 fd 对应的管道名
FIONMSGS	取得管道中当前的消息个数
FIONREAD	取得管道中第一个消息的字节个数

6.4.3 伪内存设备

伪内存[Pseudo Memory]设备使得 VxWorks 可以像读取一个伪 I/O 设备一样读取内存。内存的位置和大小在设备建立时被指定。伪内存设备对于两个或多个 CPU 之间共享内存是非常方便的。与 ramDrv 不同，伪内存设备之上并没有文件系统，其驱动 memDrv 只是提供了一个通过 I/O 调用读取内存的高层方法。通过 memDrv 初始化驱动后，调用 memDevCreate 建立设备。memDrv 对应的 ioctl 功能码定义如表 6-8 所示。

表 6-8 memDrv ioctl 功能码

模式	描述
----	----

FIOSEEK	设置伪内存设备当前的字节偏移
FIOWHERE	返回伪内存设备当前的字节偏移

6.5 串口驱动

一个完整的串口驱动包括具体的硬件驱动以及与硬件无关的终端功能等，并同时能够提供主机 Target Server 与目标机 Target Agent 之间特别的通信机制接口。在 VxWorks 中，上述工作由 ttyDrv、tyLib 和 xxDrv（硬件驱动）共同完成，如图 6-12 所示。

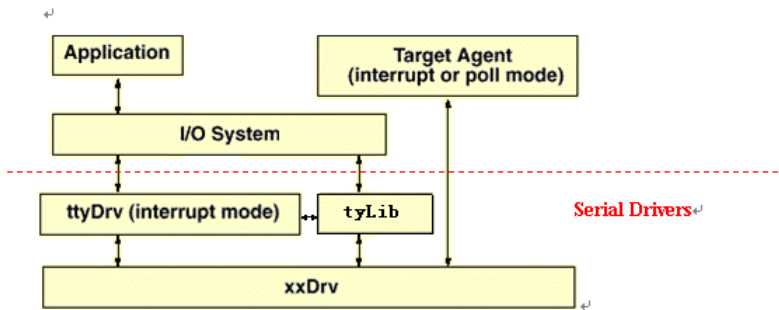


图 6-12 串口驱动结构示意图

ttyDrv 是一个虚拟的驱动，在 tyLib 库的支持下，统一管理 xxDrv（硬件驱动），按照字符设备驱动接入到 I/O 系统中，并提供读写缓冲区，完成各种与硬件无关的功能。xxDrv 作为具体的硬件驱动，除了完成硬件相关的操作外，还要对上一层（ttyDrv 和 Target Agent）提供操作接口。

xxDrv 采用回调[Callback]的方法来适应不同的上层协议，如图 6-13 所示。回调就是允许上层提供具体的操作接口函数，并在初始化时将函数指针填入到 xxDrv 的具体驱动结构中。xxDrv 在处理数据时，用回调填入的函数来完成相应的功能。这样 xxDrv 只是例行地回调填入的函数，而不用关心具体的实现细节，从而适应不同的上层实现。

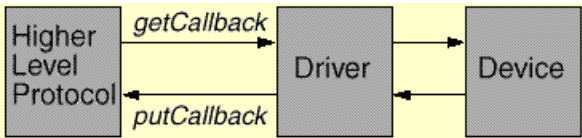


图 6-13 驱动函数回调示意

主要有两个回调函数 put 和 get。put 回调函数将从硬件设备读到的字符传送到高层协议中。get 回调函数从高层协议中读取字符写到硬件设备中。

具体到了 ttyDrv 与 xxDrv 之间，ttyDrv 将支持库 tyLib 的函数 tyIRd 和 tyITx 作为回调函数供 xxDrv 在中断时处理数据，如图 6-14 所示。

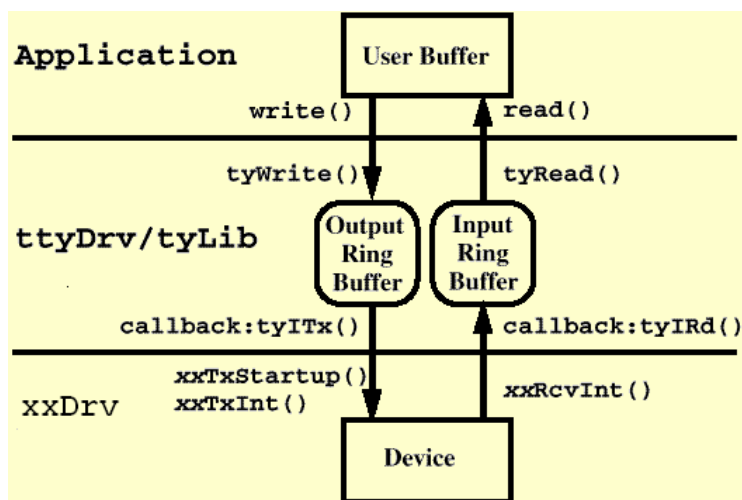


图 6-14 ttyDrv 与 xxDrv 之间的函数回调

那么，在具体的数据结构定义中，它们三者之间是如何关联的呢？ttyDrv 数据结构 TYCO_DEV 由两部分组成，TY_DEV 和 SIO_CHAN。TY_DEV 是 tty 设备的与硬件无关部分的具体描述，也是 tyLib 库函数的操作对象，同时它的头部是 DEV_HDR，这样 tty 设备可以加入到 I/O 系统的设备列表中，而 SIO_CHAN 指向了 xxDrv 的驱动函数入口，使得 tty 设备与具体的串口通道相关联，如图 6-15 所示。

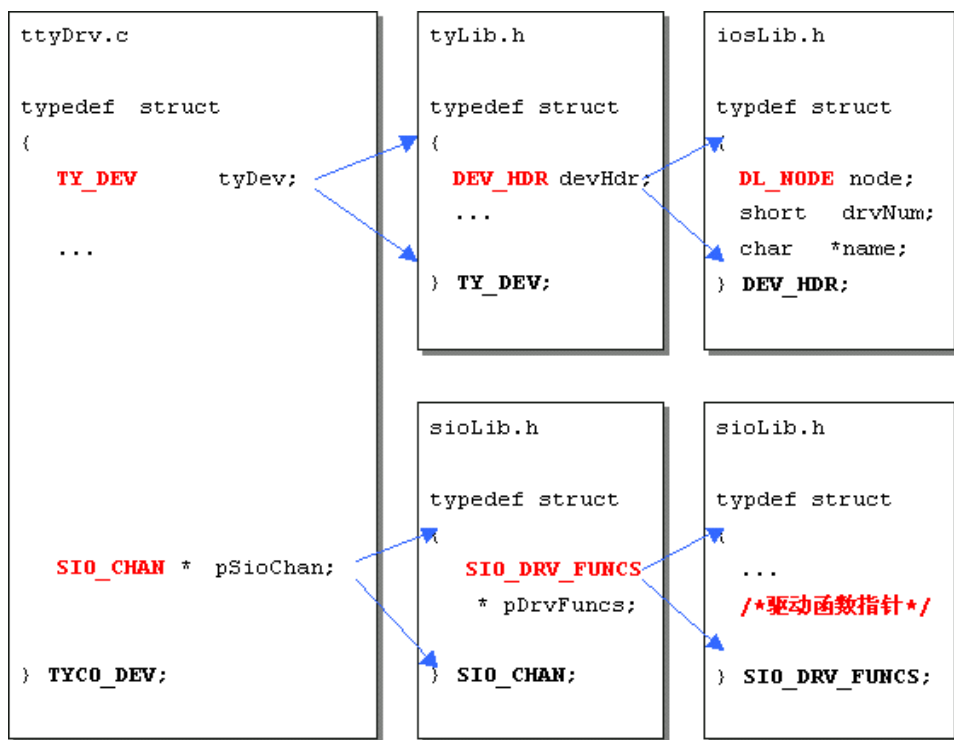


图 6-15 ttyDrv 数据结构

6.5.1 ttyDrv 和 tyLib

ttyDrv 及其支持库 tyLib 作为硬件驱动与 I/O 系统之间的接口层，主要完成以下工作。

- ✧ I/O 系统请求(安装驱动表中的入口以及创建设备并加入到 I/O 系统的设备列表中)。
- ✧ 基本 I/O 接口: ttyOpen、ttyIoctl、tyRead 和 tyWrite 等。
- ✧ 调用 selectLib。
- ✧ 命令行编辑。
- ✧ 提供数据缓冲区。
- ✧ 根据缓冲区数据的空或满同步任务。
- ✧ 缓冲区访问互斥。

下面分别描述相关的函数。

- **ttyDrv:** 完成驱动初始化。它使用 ttyDrv 和 tyLib 的函数入口调用 iosDrvInstall 来安装驱动到 I/O 系统驱动表中。

```
ttyDrvNum = iosDrvInstall (ttyOpen, (FUNCPTR) NULL, ttyOpen,
                          ttyClose, tyRead, tyWrite, ttyIoctl);
```

- **ttyDevCreate:** 用于创建 tty 设备, 如下所示:

```
ttyDevCreate ( "/tyCo/0", sysSerialChanGet(0), 512, 512);
```

该函数完成如下工作: 分配和初始化设备描述; 调用 tyDevInit 初始化 tyLib, 初始化 selectLib, 建立缓冲区, 建立信号量等; 调用 iosDevAdd()增加设备到设备列表; 安装 tyLib 库函数 tyITx 和 tyIRd 作为 xxDrv 的输入和输出回调函数; 启动设备中断模式。

- **write:** 根据驱动表调用 tyWrite, tyWrite 将数据写入到输出缓冲区中, xxDrv 调用回调函数从缓冲区取数据写入到硬件设备中。
- **read:** 根据驱动表调用 tyRead, tyRead 从输入缓冲区中读取数据。输入缓冲区中的数据是 xxDrv 调用回调函数写入的。
- **ioctl:** ioctl 的命令通过 ttyIoctl 传递到 xxIoctl, 如果 xxIoctl 失败, 则调用 tyIoctl。

ttyDrv 和 tyLib 之间的关系如图 6-16 所示。

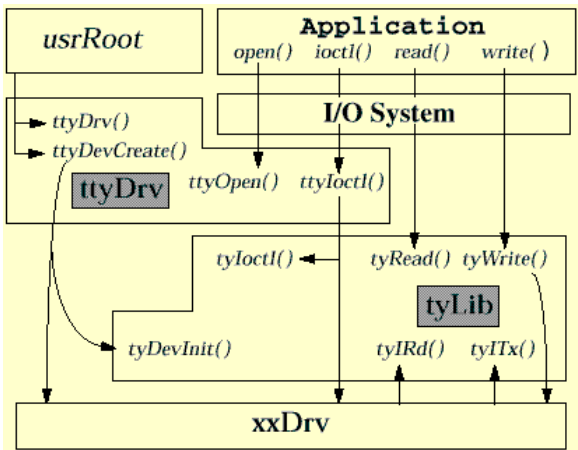


图 6-16 ttyDrv 和 tyLib

6.5.2 xxDrv

xxDrv 主要实现了硬件初始化、收发中断例程以及 SIO_CHAN 中 SIO_DRV_FUNCS 所

包括的各个驱动函数。实际上，一个指向 SIO_CHAN 的指针就是 xxDrv 与上层之间的关联，如图 6-17 所示。

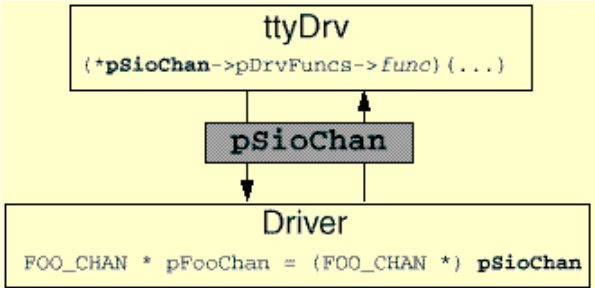


图 6-17 xxDrv

xxDrv 具体的例子参见“target/src/drv/sio”目录下的源代码，开发新的 SIO 驱动可以将“target/src/drv/sio/templateSio.c”做为模板。xxDrv 中需要实现的函数如下。

- **xxDevInit: 硬件设备初始化。**

```
void xxDevInit (pChan);
```

其中，pChan 为指向具体硬件通道的指针，因为 XX_CHAN 结构的头就是 SIO_CHAN，因此也就是指向 SIO_CHAN 的指针。

xxDevInit 填充自己的 SIO_DRV_FUNCS 结构，将回调函数设为空例程，以及完成具体的硬件初始化。

- **xxIntTx: 发送中断函数。**

```
void xxIntTx (pChan);
```

xxIntTx 先清中断（如果需要），再调用回调函数从高层协议缓冲区取得字符写到硬件设备中，当没有可发送的字符后，复位传输中断（如果需要）。

- **xxIntRcv: 接受中断函数。**

```
void xxIntRcv (pChan);
```

xxIntRcv 从硬件设备读取字符，调用回调函数写入到从高层协议缓冲区中，供 tyRead 读取。

SIO_DRV_FUNCS 包括的驱动函数如表 6-9 所示。

表 6-9 SIO_DRV_FUNCS

函数	描述
xxCallbackInstall	安装高层协议的回调函数
xxTxStartup	启动一次传输循环
xxIoctl	支持设备 ioctl 命令
xxPollOutput	查询模式输出函数
xxPollInput	查询模式输入函数

- **xxCallbackInstall: 安装高层协议的回调函数。**

```
int xxCallbackInstall (pSioChan, callbackType, callback, callbackArg);
```

其中，pSioChan 为指向 SIO_CHAN 的指针，callbackType 为回调函数类型，callback 为回调函数入口，callbackArg 为回调函数参数。

- **xxTxStartup: 启动一次传输循环。**

```
void xxTxStartup (pSioChan);
```

当 write 调用 tyWrite 后，tyWrite 拷贝数据到输出缓冲区中，并调用 xxTxStartup 来启动

一次传输，发送中断函数 `xxTxInt` 将缓冲区中的数据输出。

`xxTxStartup` 调用回调函数从输出缓冲区中取一个字符写入硬件设备从而启动传输循环, 必要的时候打开发送中断。

- `xxioctl`: 完成 `ioctl` 的控制命令。

串口驱动相关的控制命令如表 6-10 所示。如果有未识别的命令，xxIoctl 返回 ENOSYS。

表 6-10

xx|oct| 功能码

命令	描述
SIO_BAUD_SET	设置波特率
SIO_BAUD_GET	返回波特率
SIO_MODE_SET	设置中断或轮询模式
SIO_MODE_GET	返回当前模式
SIO_AVAIL_MODES_GET	返回一个可用的模式
SIO_HW_OPTS_SET	设置硬件关键字
SIO_HW_OPTS_GET	返回硬件关键字

- `xxPollOutput` 和 `xxPollInput`: 轮询模式的输出和输入函数。

主要用来支持 WDB，使得可以在系统模式下调试。

6.5.3 加载流程

串口驱动是在 VxWorks 系统启动时被加载的。VxWorks 中串口驱动相关文件的层次如图 6-18 所示。

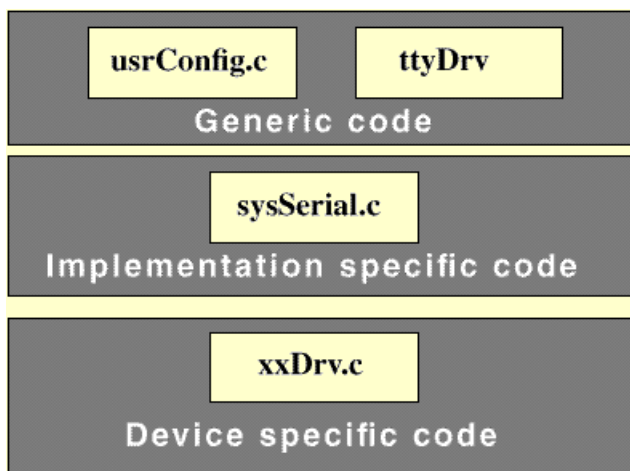


图 6-18 串口驱动相关文件的层次图

可以看出，硬件驱动 `xxDrv` 相关的加载是在 `sysSerail.c` 完成的。在 `sysSerail.c` 中实现函数的调用情况如下。

- ✧ `sysSerialHwInit`: 由 `sysHwInit` 调用, 执行设备初始化。
- ✧ `sysSerialHwInit2`: 由 `sysClkConnect` 调用, 设置中断函数入口地址。
- ✧ `sysSerialChanGet`: 由 `usrRoot` 调用 (返回的指针由 `ttyDevCreate` 使用), 根据设备通道号得到指向通道描述的指针。`Target Agent` 中也会调用它。

影响串口驱动加载相关的宏定义有：要使 VxWorks 自动加载 tty 驱动，则 INCLUDE_TYCODRV_5_2 不能被定义，可在工程配置的“obsolete component/5.2 serial drivers”组件中更改。config.h 中的 NUM_TTY 定义了 tty 设备的个数，也可以在工程配置的“hardware/serial/SIO”组件中更改。

串口驱动在 VxWorks 中的全貌如图 6-19 所示。

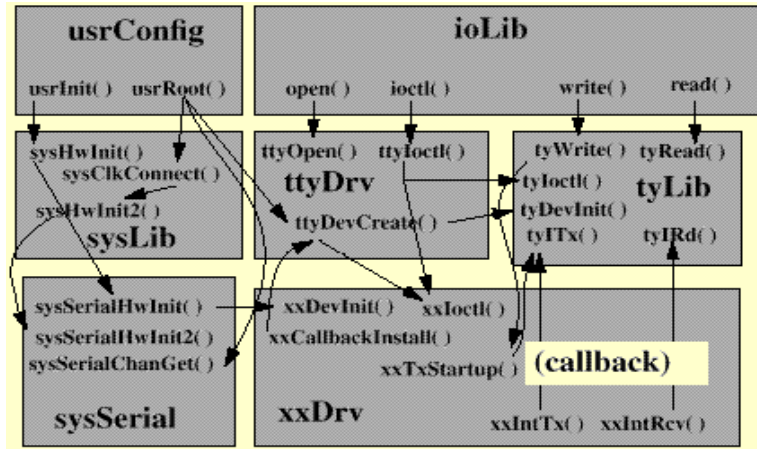


图 6-19 串口驱动在 VxWorks 中的全貌

6.7 常见问题解答

● VxWorks 的 I/O 系统和 Unix、Windows 的有何不同？

解答：根据本章的叙述可知，VxWorks I/O 系统在平常的使用中与 Unix 或 Windows 的主机 I/O 系统是类似的，但也有下面的不同。

- ✧ VxWorks 的设备驱动可以动态的安装和卸载。
- ✧ 在 Unix 和 Windows 中，文件描述符是与进程相关的，而 VxWorks 中除了标准输入、输出和错误输出外，其他的文件描述符是全局的。
- ✧ VxWorks 和 Unix 的 ioctl 控制命令不同。
- ✧ 在 Unix 中，设备驱动运行在系统模式并且没有优先级。而在 VxWorks 中，设备驱动有优先级，因为它是运行在调用它的任务的上下文中，任务的优先级就是被调用的设备驱动的优先级。

● VxWorks 中 COM1 串口名称是什么，怎么打开串口并读写，如何设置波特率和校验等？

解答：在 VxWorks 中 COM1 名称为“tyCo/0”，COM2 名称为“tyCo/1”，以此类推。使用 open 函数来打开串口。

```
fd=open("tyCo/0", 0_RDWR, 0);
```

用 read 函数从串口读数据。

```
num=read(fd, buffer, 1024);
```

用 ioctl 函数设置串口波特率和校验等，例如设置波特率为 9600。

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

用 ioctl 函数设置串口属性为 8 位数据位、1 位停止位、奇校验（宏在“target/h/sioLib.h”中定义）。


```
ioctl (fd, SIO_HW_OPTS_SET, CS8 | PARENB | PARODD);
```

- 当调用 `read` 读取串口时，如果没有数据可读，任务是否被阻塞。如何避免它不被阻塞，如何设置超时？

解答：如果没有数据可读，任务将被阻塞。可以用 `ioctl` 函数查看是否有可读数据以避免任务被阻塞。要设置超时，可以将串口的文件描述符加入到 `select` 中查询。

```
ioctl(fd, FIONREAD, (int*)&iNum);  
if (iNum > 0) {read();...}
```

 WindRiver, “Vxworks 5.4 Programmer's Guide” 的 3.3.8 章节。

- 我调用 `ioTaskStdSet` 将标准输出重定向，之后调用 `ioGlobalStdSet` 设定新的系统标准输出，会不会影响任务的标准输出？

解答：不影响。调用 `ioGlobalStdSet` 是将系统的标准输出重定向，但此时任务已经指定了特别的标准输出，所以对其无效。如果要想将任务的输出也定向到目前系统的标准输出，则应该使用如下函数。

```
ioTaskStdSet(0, STD_OUT, 1)。
```

- “`target/src/drv/sio`”目录下的驱动与“`target/src/drv/serial`”目录下的有何不同？

解答：`serial` 目录下的驱动是 VxWorks 5.2 之前的驱动，按照字符设备驱动标准直接挂到 I/O 系统中，目前一般不使用；`sio` 目录下的驱动为目前使用的驱动，它通过 `ttyDrv` 挂到 I/O 系统中，并且支持查询模式，使得在串口下可以进行系统模式调试。

- 如何开发自己的驱动并挂到 VxWorks 的 I/O 系统中？

解答：可以按照字符设备驱动模型来开发，完成 `xxOpen`、`xxRead`、`xxWrite`、`xxCtrl()` 等基本 I/O 接口，在驱动初始化时调用 `iosDrvInstall` 将上述函数入口写入到系统驱动表中。

第 7 章 文件系统

7.1 系统结构

从某种意义上说，文件系统是 IO 系统的一部分，或者说是 IO 系统的扩展。如果将 IO 系统作为一个独立的高层抽象，文件系统和字符设备驱动一样是一个驱动程序，用于辅助块设备的管理。只不过文件系统与具体的设备关系不大。如图 7-1 所示内容来自 WindRiver 的一份文档，表明应用通过 IO 系统的标准接口访问文件系统；文件系统和字符设备一样挂在 IO 系统上；块设备通过文件系统被访问，而不是直接归 IO 系统管理；同样的块设备可以选择不同文件系统来管理，如 dosFs 或 rawFs。文件系统和 IO 系统、块设备有密切的关系。

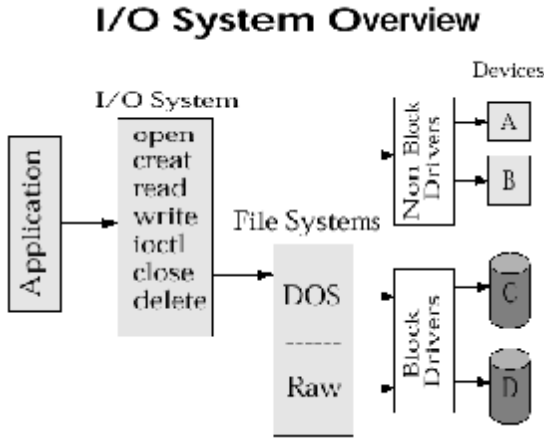


图 7-1 IO 系统概览

如图 7-2 所示内容也来自 WindRiver 文档，表明了文件系统在系统中位置，更清晰地显示了应用代码访问块设备的一个函数流程。另外读者可能注意到硬件设备中出现了两个不同寻常的块设备：网络和串行设备，它们用于支持远程文件系统。图中的 `stdio` 库和 `fioLib` 库为应用提供了丰富的操作接口，其实也是经由 IO 系统访问的。

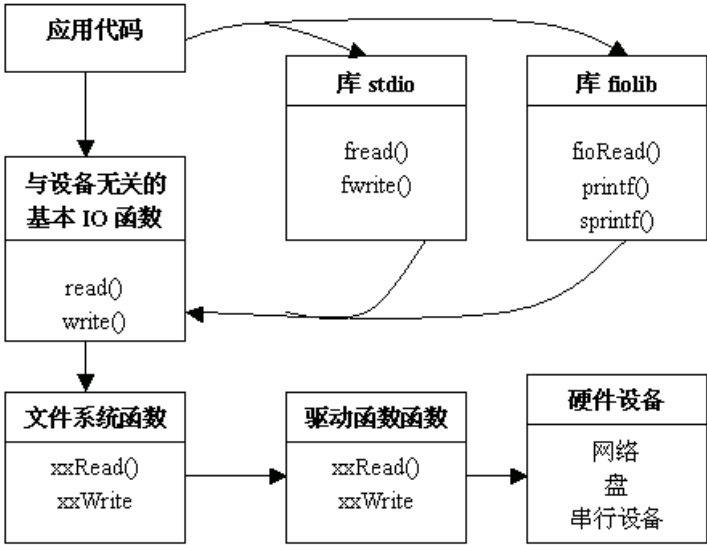


图 7-2 文件系统概览

图 7-1 和图 7-2 只是个示意，表明了大概的结构组织。而表 7-1 和表 7-2 所示列出了 VxWorks 中实际的文件系统相关函数库的层次关系。

表 7-1 简单 RAM 文件系统相关库层次表


层次	向下接口
应用模块	read、write 和 ioctl 等
VxWorks IO 系统	挂接入 IO 系统的文件系统接口 (如 dosFsRead、dosFsWrite 和 dosFsIoctl 等)
文件系统 (dosFs/rawFs)	CBIO API (cbioBlkRW、cbioIoctl 等)
CBIO API 设备驱动模块 (ramDiskCbio.c)	
RAM 硬件	

表 7-2 本地存储盘上文件系统相关库层次表

层次	向下接口
应用模块	read、write、ioctl 等
VxWorks IO 系统	挂接入 IO 系统的文件系统接口 (如 dosFsRead、dosFsWrite 和 dosFsIoctl 等)
文件系统 (dosFs/rawFs)	CBIO API 接口 (cbioLib.h)
CBIO 到 CBIO 设备 (dpartCbio)	CBIO API 接口
CBIO 到 CBIO 设备 (dcacheCbio)	CBIO API 接口
基本 CBIO 到 BLK_DEV 封装设备 (cbioLib)	BLK_DEV (blkIo.h)
BLK_DEV 设备驱动 (fdDrv、ataDrv 和 tffsDrv 等)	
存储设备硬件	

新版的 dosFs 和 rawFs 都需要一个新抽象层 CBIO 的支持，通过 CBIO 再同块设备连接。CBIO[Cache Block Input Output]用于块设备缓冲管理。对于不同的情况，VxWorks 提供多种 CBIO 库支持，但都遵循类似的接口标准。对于 RAM 设备，缓存没有必要，ramDiskCbio 用

于仿真实现 CBIO 接口，所以和普通存储设备的实现层次有所区别。如果设备需要多分区管理，可以使用 dpartCbio 库，对下层的 CBIO 再次抽象。。

 WindRiver, “Tornado Training Workshop” 的第 11 章节。

7.1.1 应用接口

在实现层次的最上层是应用对文件系统的操作接口，包括标准的 IO 系统接口，带缓存接口和格式化接口。这些在前面的“IO 系统”一章已经有了详细的介绍。为了完整性，这儿再做简单说明，强调与文件系统相关的特殊方面。

● ioLib

IO 系统提供 7 个标准接口用于文件系统操作，如表 7-3 所示。

表 7-3 IO 系统的 7 个标准接口

接口	说明
creat	创建一个文件
remove	删除一个文件
open	打开一个文件（或创建新文件）
close	关闭一个文件
read	读文件
write	写文件
ioctl	对文件系统执行特殊的控制功能

各函数的原型可参考前面“IO 系统”一章的描述，或 ioLib 的说明。实际中，这些函数多用文件描述符来对文件进行操作。除了标准的输入、输出、错误输出，VxWorks 的文件描述符是全局的，可以被任何一个任务存取。由于系统保留了 3 个描述符，具体文件得到的描述符都是大于 2 的。

ioctl 是文件系统的灵活接口，用于标准 IO 接口规定外的功能实现。ioctl 的功能代码和参数说明，与底层的具体文件系统、驱动相关，将在本章后面介绍。ioctl 的功能实现是分层的，如 FIOGETNAME 在 IO 系统实现，而 FIORENAME 在文件系统实现。上层没有实现再向下层传递，若各层都没有该功能的实现，则返回错误。功能代码统一在 ioLib.h 中定义。

除了这 7 个标准函数，ioLib 还提供与文件系统相关的特殊接口，如表 7-4 所示。

表 7-4 IO 系统的文件系统相关接口

接口	说明
unlink	删除一个文件（POSIX），同 remove（ANSI）
rename	修改一个文件的名称
lseek	设置一个文件的读写指针
ioDefPathSet	设当前的缺省路径
ioDefPathGet	取当前的缺省路径
chdir	设当前的缺省路径，同 ioDefPathSet
getcwd()	取当前的缺省路径（POSIX），类似 ioDefPathGet
getwd	取当前的缺省路径（Unix），类似 ioDefPathGet

当用名称访问文件时，可以使用全路径，也可设置当前路径后，用文件名访问。VxWorks 中使用全局变量 `ioDefPath` 来存放当前路径，由所有任务共用。

文件系统和普通设备最大的不同就是目录的相关操作，如目录创建、删除等。另外一些相同函数接口也可能有细节的区别。这主要是靠一些标志（`flag`），模式（`mode`）来区分。

标志在 `fcntlcom.h` 中定义，常用的一些如表 7-5 所示，主要由 `creat` 和 `open` 使用。

表 7-5 文件操作标志

flag	值	说明
O_RDONLY	0	只读
O_WRONLY	1	只写
O_RDWR	2	可读可写
O_CREAT	0x0200	指定 <code>open</code> 创建新文件

`open` 使用第 3 参数 `mode` 来进行目录的创建，该参数遵循 Unix 风格，如下所示。

```
int fd = open (name, O_RDWR | O_CREAT, FSTAT_DIR | DEFAULT_DIR_PERM);
```

`mode` 参数值在 `ioLib.h` 中定义，如表 7-6 所示。

表 7-6 文件操作模式

mode	值	说明
FSTAT_DIR	0040000	目录
DEFAULT_DIR_PERM	0000750	目录的操作权限（Unix 风格，缺省为 <code>rwxr-x---</code> ）

目录的删除同普通文件一样，如下所示。

```
remove(name);
```

● `dirLib`

除了目录创建和删除操作外，还需要目录文件列表读取的功能，VxWorks 提供 `dirLib` 库来实现。`dirLib` 建立在 `ioLib` 之上。库 `dirLib` 提供如下函数接口用于目录读取，如表 7-7 所示。

表 7-7 `dirLib` 库目录操作函数

接口	说明
<code>DIR *opendir(char *name)</code>	打开目录
<code>struct dirent *readdir(DIR *pDir)</code>	读取目录
<code>void rewinddir(DIR *pDir)</code>	使目录读取位置从头开始，和刚打开目录时一样
<code>STATUS closedir(DIR *pDir)</code>	关闭目录

函数相关的两个结构在 `dirent.h` 中定义。

```
struct dirent /*目录条目 dirent */
{
    /*NAME_MAX= _PARM_NAME_MAX=99, 在 limits.h 和 vxParams.h 中定义*/
    char d_name [NAME_MAX + 1]; /*文件名, null 结尾*/
};
typedef struct { /*目录描述符 DIR */
    int dd_fd; /*打开目录的文件描述符*/
    int dd_cookie; /*具体文件系统相关的目录位置标志*/
    struct dirent dd_dirent; /*取得的目录条目*/
} DIR;
```

库 `dirLib` 还提供用于获取文件或文件系统状态信息的函数接口，如表 7-8 所示。

表 7-8 `dirLib` 库状态信息函数

接口	说明
<code>fstat</code>	读取文件状态信息（用文件描述符）
<code>stat</code>	读取文件状态信息（用文件名）
<code>fstatfs</code>	读取文件系统状态信息（用文件描述符）
<code>statfs</code>	读取文件系统状态信息（用文件名）
<code>utime</code>	更新文件时标（用文件名）

函数相关的结构 `stat` 和 `statfs` 在 `stat.h` 中定义，`utimbuf` 在 `utime.h` 中定义。

```
typedef unsigned long time_t;
struct utimbuf {
    time_t    actime;    /*设置访问时间*/
    time_t    modtime;   /*设置修改时间*/
};
```

上面几个函数的用法可以参考 “`\target\src\usr\usrFsLib.c`” 中的代码。

● `usrFsLib`

为了用户更方便地操作文件系统，`VxWorks` 还提供了 `usrFsLib` 库，在 `ioLib` 和 `dirLib` 之上做了更实用的抽象。`usrFsLib` 库提供源代码，在 “`\target\src\usr\`” 目录下，也可作为使用文件系统的代码参考。`usrFsLib` 中的接口多为用户熟悉的命令，可直接在 `Shell` 中使用。各函数也可在程序中按接口规范调用。用 `ioHelp` 命令可以显示详细的命令帮助列表，如表 7-9 所示。

表 7-9 `usrFsLib` 函数接口

接口	说明
<code>ioHelp</code>	显示详细的命令帮助列表
<code>cd</code>	切换当前的工作目录
<code>pwd</code>	显示当前的工作目录
<code>mkdir</code>	建立目录
<code>rmdir</code>	删除目录
<code>rm</code>	删除文件
<code>chkdsk</code>	文件系统一致性检查
<code>ls</code>	显示文件列表
<code>ll</code>	显示详细文件列表
<code>lsr</code>	显示文件列表，包括所有子目录
<code>llr</code>	显示详细文件列表，包括所有子目录
<code>copy</code>	文件复制
<code>cp</code>	文件复制到其他文件或目录
<code>rename</code>	文件更名
<code>mv</code>	文件更名或移到其他目录
<code>xcopy</code>	文件复制，包括子目录

xdelete	文件删除，包括子目录
attrib	修改文件或目录的属性
xattrib	修改文件或目录的属性，包括子目录，支持通配符
diskFormat	格式化，ioctl 的 FIODISKFORMAT 和 FIODISKINIT 调用

● ansiStdio

ansiStdio 库提供与 ANSI C 兼容的标准 IO 函数接口。各函数使用缓存机制，可提高大块数据操作性能。

ansiStdio 中函数大多以字母“f”开头，与 IO 系统接口名称差别也多在这个首字母，如 fopen 对 open。各函数不再用文件描述符索引文件或设备，而用 FILE 指针，或称做流[stream]。FILE 指针指向 FILE 结构，该结构封装了一些与缓存操作相关的成员，在 stdio.h 中定义。FILE 结构成员“_file”指向 IO 系统中的文件描述符，FILE 指针可以直接用 fopen 得到，或者用 fdopen 关联已打开文件的描述符。与文件描述符不同，FILE 指针不是全局的，而归属于某个任务。VxWorks 允许同一文件可以同时被打开多次，不同任务访问同一文件时，需要用户自己负责保证同步。

具体的函数细节请参考库手册的描述。



WindRiver, “Vxworks 5.4 Programmer’s Guide” 的 3.4.1 章节。

● fioLib

fioLib 提供一些与 ANSI C 兼容的串格式处理函数，如 printf 等。有些函数是 ANSI C 库的一部分，按理应该在 ansiStdio 中实现，但为了减少 IO 缓存的额外负荷，则不使用缓存，在 fioLib 库中单独实现。

库中一些函数输入输出定向在标准 IO 文件描述符上，可以用 ioGlobalStdSet 或 ioTaskStdSet 重定向标准 IO 文件描述符到具体的文件描述符上，则可用 printf 等向文件格式化写入；或者使用 fdprintf 等直接文件描述符传入函数；或者使用变参数支持函数封装自己的 IO 函数；或者用 sprintf 等先在缓存格式化完成后，再用 IO 系统接口操作文件。

具体的函数细节请参考库手册的描述。

7.1.2 IO 系统

文件系统代表块设备和 IO 系统接口，而块设备（包括 CBIO 设备）都不直接挂入 IO 系统。文件系统和 IO 系统的挂接和字符设备类似。下面以 dosFs 为例作简单介绍。

在 dosFsLib 库中首先实现了 IO 系统规定的 7 个标准函数。

在 dosFsLibInit 中调用 iosDrvInstall 将 7 个函数指针填入驱动表中，并得到一个驱动号。驱动表可以用 iosDrvShow 命令在 Shell 中查看，如下所示：

-> iosDrvShow							
drv	create	delete	open	close	read	write	ioctl
1	8affe	0	8affe	8b02c	8b65c	8b596	8b066
2	0	0	87818	0	8784a	87896	87996
3	9ada6	9b128	9aa92	9ae48	9b89e	9b8c0	9c10e
4	0	0	0	72e6e	73a94	7361e	bae20
5	78670	78780	78902	789b8	793c4	79504	79656
6	5e0ca	0	5e0ca	5e192	8b65c	8b596	5elfc

```
value = 608 = 0x260
-> 0x9aa92
value = 633490 = 0x9aa92 = dosFsChkDsk + 0xf74
```

其中的 3 号驱动就表示 dosFs，后面跟的 7 个数值为函数指针。因为各实现函数都是静态局部的，所以用指针值只能取得大概的符号解释。而后面的 devs 命令显示会更清楚地表示 3 号驱动为 dosFs。

文件系统接口在驱动表注册后，dosFsDevCreate 中调用 iosDevAdd，传入设备结构指针、设备名称和驱动号为参数，将设备结构添加到设备链表中。这样 dosFs 就加入了 IO 系统中。可以在 Shell 上使用 devs 或 iosDevShow 查看设备链表，如下所示：

```
-> iosDevShow
drv name
 0 /null
 1 /tyCo/0
 1 /tyCo/1
 5 host:
 6 /vio
 3 /tffs0/
 3 /tffs1/
```

结果显示，在两个 Flash 存储区分别创建两个 dosFs 设备，都对应驱动表中第 3 表项的接口函数。这和上面 iosDrvShow 的显示是一致的。

文件系统安装到 IO 系统中后，就可以用上层库提供的接口函数进行操作了。有的函数使用设备名称作为设备链表索引，如 open、remove 等。而有的函数使用文件描述符为索引，如 read、write 等。设备名称的命名规则，以及文件描述符与设备链表的关系读者可参考本书第 6 章的介绍。

7.1.3 CBIO

这是文件系统的向下接口，原来的块设备接口都必须转换为 CBIO 接口，才能和文件系统挂接。老版本的 Tornado 2.0 中没有这层，在新版本的 Tornado 或 dosFs 2.0 中支持。

● cbioLib

块设备缓存 IO 库[Cached Block I/O]用于提供缓存块输入输出编程接口。dosFsLib、rawFsLib 和 usrFdiskPartLib 等库使用 CBIO 接口对底层设备进行 IO 操作。cbioLib 也为其他的 CBIO 模块提供基础支持，如 dpartCbio、dcacheCbio 和 ramDiskCbio。cbioLib 还提供一个基本的 CBIO 模块，为块设备提供接口封装，封装设备可以直接挂接文件系统，而只有极小的内存消耗。

“CBIO-CBIO 设备”上下接口都遵循 CBIO API，如 dcacheCbio 和 dpartCbio，这种设备可以堆叠使用。“CBIO API 设备驱动”只对上提供 CBIO 接口，对下驱动硬件，如 ramDiskCbio。如果自己设计 CBIO 驱动可参考 ramDiskCbio.c。“CBIO 封装设备”为普通块设备提供对上 CBIO 兼容接口，由 cbioLib 提供。以上 3 种类型的设备对上接口都与 CBIO 兼容的，可统称为 CBIO 设备。CBIO 设备的使用者一般为文件系统，也可能是其他可堆叠使用的 CBIO 设备。

CBIO API 为操作 CBIO 设备的接口规范，由 CBIO_DEV 结构规定，所有 CBIO 设备都

用该结构表示，用结构指针 CBIO_DEV_ID 作为标识，该结构在 cbioLibP.h 中定义，如表 7-10 所示。

表 7-10 CBIO 设备结构

成员	说明
objCore	用于类对象管理
cbio_mutex	CBIO 设备访问的互斥信号量
pFuncs	设备自己的 4 个函数接口，由各 CBIO 设备创建函数填写
params	CBIO 物理参数，由块设备向 CBIO 各层设备传递
cbio_memBase cbio_memSize	各 CBIO 设备的私有的内存区，ramDiskCbio 用来作为存储区，dcacheCbio 用来作为 cache 区
pDc	设备相关指针，建立 CBIO 各层之间的关联，如 CBIO 封装设备的 pDc 指向块设备结构；ramDiskCbio 的 pDc 无用；dcacheCbio 的 pDc 指向 CBIO 封装设备

对于文件系统来说，只有 CBIO 模块几个函数接口可见。文件系统与下层设备的交流就通过这几个接口函数进行。CBIO 设备必须提供它们自己的实现，如表 7-11 所示。

表 7-11 CBIO 设备函数接口

函数	说明
cbioBlkRW	按块单位读写
cbioBytesRW	按字节单位读写
cbioBlkCopy	块复制
cbioIoctl	多用途控制操作

当文件系统调用其中一个函数，CBIO 各层传递执行。比如 cbioIoctl 会调用 dcacheIoctl。dcacheIoctl 执行完能解释的控制命令后，会调用 blkWrapIoctl，下传不能解释的命令。同样，blkWrapIoctl 会调用块设备的 xxxIoctl，直到最终的硬件控制，执行结果再层层返回。

cbioLib 为其他的 CBIO 模块提供基础支持。如上面描述的 CBIO 的通用接口由该库提供。另外将 CBIO_DEV 归于对象管理，提供设备创建，创建互斥信号量，分配内存等。

cbioLib 提供了一个基本的 CBIO 模块实现。块设备可直接用 cbioDevVerify 或 cbioWrapBlkDev 封装，由文件系统使用。这比较适合电子盘，而对于有机机械寻道和旋转机构的存储盘，最好在 CBIO 封装设备上加 dcacheCbio 层。

● dcacheCbio

dcacheCbio 通过 CBIO 接口实现存储盘 Cache 机制，将经常使用的块存储在内存中以提高效率，主要针对 Dos 文件系统使用。Cache 不知道具体文件格式，只按块为对象操作。

dcacheCbio 可用于大多数块设备，下层可以挂接 CBIO 接口或块设备接口。当直接和块设备连接时，会用 cbioLib 提供的基本 CBIO wrapper 封装块设备 API。因为该模块上下层都遵循 CBIO 接口，在 CBIO 堆叠使用时，是可选的。比如在 TFFS 上挂接文件系统时，通常使用 dcacheDevCreate，而用户也可替换使用 cbioLib 中 cbioDevVerify 或 cbioWrapBlkDev 来

创建 CBIO 设备，只对块设备接口做基本的封装。

dcacheCbio 使用各种技术来提供块设备的性能。如最远使用块重用（LRU），提前读（Read-ahead），整理后写（Write-behind），大块请求跳过 Cache（Bypass），后台周期更新等。

Cache 按块为单位组成，按最近使用（MRU）顺序组织为链表。当需要空间保存新块时，删除链表尾部的 LRU 块。除了这些通常的 Cache 块外，Cache 分出部分空间用于“大缓存”操作（Burst），一般为 Cache 大小的 1/4，最大可以为 64KB。

dcacheCbio 会专门创建一个任务 tDcacheUpd 用于写入 Cache 中被修改的块，以保持存储盘和 Cache 的同步，系统中所有的 Cache 设备都使用这一个任务工作。该任务周期性运行，周期间隔 0.25S。各设备 Cache 的更新时机由各自的变量控制。可以由 syncInterval 参数调节，判断直接取系统时钟的 tick 数，与任务的循环周期无关。任务优先级缺省为 250（可修改）。设备的访问由 CBIO 互斥信号量保护，在某些时候任务优先级会随其他任务暂时提升，如 tFfsPTask。任务栈大小缺省为 5000B。dcacheCbio 提供 3 个全局变量用于设置任务优先级、栈和选项，如下所示。

```
int dcacheUpdTaskPriority = 250;
int dcacheUpdTaskStack = 5000;
int dcacheUpdTaskOptions = VX_FP_TASK | VX_SUPERVISOR_MODE | VX_UNBREAKABLE;
```

在 dcacheDevCreate 调用前，可以修改它们来定制 tDcacheUpd 任务。优先级也可在运行时使用 taskPrioritySet 修改，tDcacheUpd 任务每循环一次都会按 dcacheUpdTaskPriority 变量重新设定优先级。

系统运行异常终止时，延迟写入会导致 Cache 中修改数据的丢失。为了减少数据损失，Cache 会周期性和存储盘同步，一般可能最大延迟为“0.25+syncInterval”s，其中 0.25 由 tDcacheUpd 任务循环周期决定。syncInterval 缺省由 Cache 块数确定，也可由用户指定。小 syncInterval 可以减少异常时数据损失，但会降低文件系统性能。若 syncInterval 小至 0，数据会直接写入存储盘，而不延迟。或者去掉 dcacheCbio 层，直接使用 cbioLib 接口封装块设备，也会直接写入。

dcacheCbio 提供很多可调参数来优化文件系统性能，如表 7-12 所示。

表 7-12 dcacheCbio 库可调参数

可调参数	说明
Cache 大小	用 dcacheDevCreate 创建 Cache 设备时指定，或用 dcacheDevMemResize 重新设定
dirtyMax	用 dcacheDevTune 设定，允许缓存的最大修改块数，否则需立即更新，太大的值会降低命中率，或可能数据丢失
bypassCount	用 dcacheDevTune 设定，定义大数据区的块数，大数据块操作跳过 Cache，直接 在应用和设备间进行
readAhead	用 dcacheDevTune 设定，提前读入 Cache 的块数，能提供读性能，太大的值会降低 命中率。
syncInterval	用 dcacheDevTune 设定，调节 Cache 和存储盘同步的周期（s），若为 0，同 bypass 效果，直接写入设备，太大的值导致数据丢失可能性增加

参数的当前值可以使用 dcacheShow 查看。dcacheShow 能显示当前存储盘参数、Cache 大小、可调参数和性能统计等。越多的内存，越高的命中率[hit ratio]，就表示越好的性能。跳过 Cache 直接读写，加上使用连续模式文件，dosFs 能达到和 rawFs 同样的性能。

● dpartCbio

dpartCbio 库利用 CBIO 接口实现多分区的管理，支持各个分区创建独立文件系统。该分区管理库需要一个外部库支持，如 usrFdiskPartLib，用于解析特殊的存储盘分区表。

当和 dcacheCbio 库堆叠使用时，建议将其放在 dcacheCbio 上面，作为主 CBIO 设备。调用 dpartDevCreate 应先调用 dcacheDevCreate。这样 dcacheCbio 层就可以为整个存储盘服务，而不是只对单分区服务。因为该模块上下层都遵循 CBIO 接口，在 CBIO 堆叠使用时，是可选的。比如在 TFFS 建立文件系统时，可以直接挂接 dcacheDevCreate 创建的 CBIO 设备。

支持用 MSDOS 的 FDISK 创建的分区表格式，这需要 usrFdiskPartLib 库的支持，该库在“\target\src\usr\usrFdiskPartLib.c”中实现，可以用于计算机的多分区硬盘的挂接。下面代码示例 4 分区存储盘的挂接。

```
usrPartDiskFsInit( BLK_DEV * blkDevId )
{
    const char * devNames[] = { "/sd0a", "/sd0b", "/sd0c", "/sd0d" };
    CBIO_DEV_ID cbioCache;
    CBIO_DEV_ID cbioParts;

    /* 在整个 BLK_DEV 上创建 cache */
    cbioCache = dcacheDevCreate ( blkDevId, NULL, 0, "/sd0" );
    if (NULL == cbioCache) return (ERROR);
    /*创建分区管理设备，使用 FDISK 格式解析库 */
    cbioParts = dpartDevCreate( cbioCache, 4, usrFdiskPartRead );
    if (NULL == cbioParts) return (ERROR);
    /* 在各分区上建立文件系统 */
    dosFsDevCreate( devNames[0], dpartPartGet(cbioParts,0), 0x10, NONE);
    dosFsDevCreate( devNames[1], dpartPartGet(cbioParts,1), 0x10, NONE);
    dosFsDevCreate( devNames[2], dpartPartGet(cbioParts,2), 0x10, NONE);
    dosFsDevCreate( devNames[3], dpartPartGet(cbioParts,3), 0x10, NONE);
}
```

● ramDiskCbio

VxWorks 提供该模块用于实现 CBIO 接口的 RAM 盘驱动。和通常的块设备不同，该实现将 CBIO 接口融入其中，没有独立的块设备实现。这可能是由于 RAM 比较简单、统一，本不需要缓存。ramDiskCbio 替代 dcacheCbio 的位置，创建的设备可以直接用文件系统加载，这样可以减少内存的使用。该模块的实现会使用 CBIO 基本接口。

该模块以源代码形式发布，对应“\target\src\usr\ramDiskCbio.c”文件，可以作为基本 CBIO 设备实现的范例，加深读者对 CBIO 的理解。

ramDiskCbio 为实现 CBIO 接口提供如下函数，如表 7-13 所示。

表 7-13 ramDiskCbio 库函数接口

函数	说明
ramDiskBlkRW	按块单位读写
ramDiskBytesRW	按字节单位读写
ramDiskBlkCopy	块复制

ramDiskIoctl	多用途控制操作，只实现了 CBIO_CACHE_NEWBLK，用于块清零
--------------	--------------------------------------

用户不直接使用上面这些接口，只需要调用 `ramDiskDevCreate` 创建 RAM 设备。该函数将接口指针填入 `CBIO_DEV` 结构，再将该结构的指针传给上层文件系统使用。

```
CBIO_DEV_ID ramDiskDevCreate
(
    char *ramAddr,          /* RAM 盘起始内存地址。若为 0, 由 CBIO 类分配 */
    int  bytesPerBlk,       /* 块字节数，必须为 2 幂，最小为 32。若为 0, 缺省 64 */
    int  blksPerTrack,      /* 每 track 的块数。若为 0, 缺省 16 */
    int  nBlocks,           /* 设备上的总块数。若为 0, 缺省 2048 */
    int  blkOffset          /* 从设备起始跳过的块数 */
)
```

RAM 盘的大小由 `bytesPerBlk` 和 `nBlocks` 乘积决定。返回的 `CBIO` 设备指针可以直接由文件系统创建函数使用。

`ramDiskDevCreate` 实现中先调用 `cbioLibInit`，将 `CBIO` 设备归入类对象管理（`classLib`），全局变量 `cbioClassId` 用于该类对象标识，再通过类对象接口 `createRtn` 调用 `cbioLib` 中的 `cbioDevCreate` 创建一个 `CBIO` 设备。`ramDiskCbio` 直接利用 `CBIO_DEV` 中的 `cbioMemBase` 和 `cbioMemSize` 来定位 RAM 盘的地址范围，所有的读写操作都在该地址空间中进行。`ramDiskDevCreate` 再对创建的 `CBIO_DEV` 进行自己的填充，接入自己的回调函数，并将 `CBIO_DEV` 的指针返回上层文件系统使用。

`ramDiskDevCreate` 的具体使用读者可参考本书 7.3.3 中的介绍。

7.2 本机文件系统

7.2.1 dosFs

`dosFs` 提供极大的灵活性以满足实时应用的各种要求，主要特性包括以下方面。

- ✧ 文件和目录高效层次组织，允许创建任意数目文件。
- ✧ 可以使用连续文件模式，提高操作性能。
- ✧ 支持多种存储盘，兼容各种操作系统的 `dosFs`，支持 PC 风格的分区表。
- ✧ 支持长文件名，也兼容 `MSDOS` 的命名规则。
- ✧ 支持 `NFS`。

● 版本 2.0

`Tornado 2.0` 版本中包含老版本的 `dosFs`，`Tornado 2.1` 以上版本包含新版的 `dosFs 2.0`。`dosFs 2.0` 也作为单独产品销售，可以将 `Tornado 2.0` 中的 `dosFs` 升级到 `dosFs 2.0`。两个版本之间有很多不同，如下面所列。

（1）新加的特性

- ✧ 支持 `Microsoft` 风格的长文件名（`VFAT`）。
- ✧ 支持 `FAT32` 格式。
- ✧ 支持存储盘缓存。
- ✧ 文件系统的内部缓存不再使用。
- ✧ 支持存储盘大小和文件偏移的 64 位计算。新版 API 只对初始化有影响，实际应用不用

修改。但是，如果支持4GB以上的大存储盘，需要调用ioctl函数。

- ✧ 增强的VxLong文件名，支持超过4GB的大文件。
- ✧ 支持PC风格的分区管理，使用分区创建函数usrFdiskPartCreate。
- ✧ 新用户级函数库usrFsLib包括递归复制、删除、属性设置、详细列表和通配符支持等。
- ✧ 归档和备份功能，使用Unix兼容的tar格式，参考tarLib。

(2) 过时的特性

- ✧ 用dosFsLib访问raw存储盘。
- ✧ NFS服务器私有接口（DOS_OPT_EXPORT和DOS_OPT_LOWERCASE）。
- ✧ DOS_OPT_CHANGEWARN，由存储盘cache自动执行可移动盘检测。
- ✧ DOS_OPT_AUTOSYNC，由存储盘cache的syncInterval参数代替。
- ✧ DOS_OPT_LONGNAMES由DOS_OPT_VXLONGNAMES代替。
- ✧ DOSFS_MAX_PATH_LEN，由dosFsVolFormat的新格式化选项代替。
- ✧ 文件系统上层函数从usrLib分离到usrFslib中，避免对tShell的依赖。

(3) 兼容性：

新版本dosFs主要支持文件系统初始化兼容性，而文件实际操作仍遵循IO系统标准接口而自然兼容。VxWorks组件“DOS filesystem backward-compatibility”提供初始化兼容性，该组件以源代码文件提供（usrDosFsOld.c），以仿真老dosFs接口，如dosFsDevInit和dosFsMkfs等。但并不支持所有老选项，也不能使用dosFs 2.0的新特性，兼容性只意味着系统初始化代码可以不修改而直接使用。为了使用新特性，需要修改usrDosFsOld.c，或者使用新API重写初始化过程。

 WindRiver，“dosFs for Tornado 2.0 Release Notes and Supplement 2.0”的3和4章。

● 文件时标

文件系统中使用ansiTime库中time函数来取系统时间，确定文件或目录的时标。若取回的时间值比1998.1.1早，则认为是无效时间，时标缺省使用1980.1.1。time用clockLib中的clock_gettime实现，取系统软时钟“_clockRealtime”的值，软时钟由系统周期时钟中断按tick单位进位。在系统初始化时需调用clock_settime确定软时钟初值，初值来源可为硬件RTC。另外，如果系统中有比系统周期时钟中断更精确的时钟源，也可调用clock_settime不断同步系统软时钟。

● 连续文件

性能对实时系统至关重要，为了提高文件操作性能，可以使用连续文件。连续文件由连续的存储块组成，也可为目录分配连续存储区。当文件打开时，系统会检查它是否为连续文件。若是，系统会使用更高效技术定位文件存储区，而不使用FAT中的族链表。

为了给文件分配连续的存储区，需要先按正常方式创建文件，再使用ioctl的FIOCONTIG功能为新文件分配要求大小的连续存储区，若存储盘无足够大的连续区，ioctl返回错误，代码示例如下。

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

也可以请求分配存储盘可用的最大连续空间。

也可以先取得存储盘可用的最大连续区大小，再判断是否能创建想要的连续文件。


为了释放空闲分配区,可以使用 POSIX 兼容的 `ftruncate` 或 `ioctl` 的 `FIOTRUNC` 功能来剪件。


ioctl 中各功能由各层结构分别实现。dosFs 支持的功能码在 ioLib.h 中定义, 如表 7-14 所

dosFs 的 ioctl 选项

功能	说明	其他
FIOATTRIBSET	设置文件属性	
FIOCONTIG	为连续文件或目录分配存储区	FIOCONTIG64
FIONCONTIG	取得存储盘最大连续区大小	FIONCONTIG64
FIODISKCHANGE	声明存储盘改变	由底层 CBIO 实现
FIODISKINIT	dosFs 卷格式化，类似 dosFsVolFormat	
FIOFLUSH	写文件缓存到存储盘上	
FIOSYNC	同上，并重读缓存的文件数据	

FIOFSTATGET	取文件状态信息和目录条目数据	
FIOFSTATFSGET	取文件系统信息	
FIOGETNAME	根据文件描述符取文件名	
FIOLABELSET	设置卷标	
FIOLABELGET	读取卷标	
FIOMKDIR	创建目录	
FIORMDIR	删除目录	
FIONFREE	读取空闲字节数	FIONFREE64
FIONREAD	取文件未读字节数	FIONREAD64
FIOREADDIR	读下一个目录条目	
FIORENAME	为文件或目录更名	
FIOSEEK	设置文件的字节偏移	FIOSEEK64
FIOWHERE	读取文件的字节偏移	FIOWHERE64
FIOTRUNC	剪裁文件	FIOTRUNC64
FIOUNMOUNT	卸载文件系统	
FIOTIMESET	设置文件时标	
FIOCHKDSK	文件系统一致性检查	

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 4.2 章节。

 WindRiver, “dosFs for Tornado 2.0 Release Notes and Supplement 2.0”。

7.2.2 其他

rawFs 为一个小文件系统，为系统实现最基本的存储盘 IO 操作。VxWorks 提供 rawFsLib 库来实现 rawFs。rawFs 将整个存储盘作为一个大文件来管理，可以为简单文件管理提供较好的性能。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 4.4 章节。

VxWorks 提供文件系统 rt11Fs，与 RT-11 文件系统兼容。VxWorks 提供 rt11Fs 的主要目的是兼容老版本的 VxWorks。现在一般选用 dosFs 文件系统，它具有更丰富的功能，如支持连续文件分配，灵活的文件命名等。rt11Fs 已经过时了，将来新版本的 VxWorks 可能不再支持。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 4.3 章节。

tapeFs 用于支持磁带存储设备。这种设备由于物理特性的限制，不能使用标准的文件目录结构。tapeFs 和 rawFs 一样，将整个存储作为一个大文件，该文件上的具体数据组织由上层应用负责。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 4.5 章节。

cdromFs 用于支持 ISO 9660 格式的光盘访问。只能进行读操作，写操作会返回错误。cdromFs 支持多设备，多文件打开，并行文件访问。



7.3 块设备

块存储设备是支持文件系统的基础，本节将介绍各种块设备，重点介绍 TFFS。

7.3.1 概述

VxWorks 的块设备与字符设备相区别，最本质的差别是数据传输的单位，块设备以块为单位传输数据，字符设备以字节单位传输数据。另一个较大的区别是块设备不能与 I/O 系统直接打交道，在其与 I/O 系统之间必须有文件系统，如 dosFs、rt11Fs、rawFs 或 tapeFs 等，这种层次关系允许同一个块设备上使用不同的文件系统。块设备的函数入口不在驱动表中注册（上层文件系统的函数入口在驱动表中注册），块设备的描述符也不进入设备链表（文件系统的描述符进入设备链表）。因为设备本质的区别，块设备的实现不同于字符设备的实现，对上层抽象接口的规范也不相同。

为了实现上层软件的设备无关性，所有块设备驱动都遵循统一接口规范。该接口不以单独的软件层存在，而是体现在一个统一的数据结构上（BLK_DEV）。该结构由具体的设备实例化，用来代表该设备。对于上层软件来说，与块设备的接口就是这个统一结构，通过具体的结构实例来操作对应的设备。该结构中包含了统一的函数接口，函数具体的实现由各块设备驱动程序完成。

另外为了方便块设备的创建，各块设备都遵循类似的接口（如表 7-15 所示）和类似的初始化过程。如 xxxDrv 用于设备相关初始化，应先于设备实际操作前调用，一般由系统帮助完成。xxxDrv 只执行一次，影响整个设备控制器，而其他后继的操作只影响特殊的设备。xxxDrv 完成的功能包括初始化硬件，分配和初始化设备数据结构，创建互斥信号量，初始化中断等。xxxDevCreate 用来创建设备实例，当创建块设备实例时，设备没有名称与其相联。只有建立文件系统后才有名称。块设备再经过 CPIO 封装由文件系统使用。

表 7-15 块设备的类似初始化函数

	xxxDrv	xxxDevCreate
ramDrv	STATUS ramDrv	BLK_DEV *ramDevCreate
nec765Fd	STATUS fdDrv	BLK_DEV *fdDevCreate
ideDrv	STATUS ideDrv	BLK_DEV *ideDevCreate
ataDrv	STATUS ataDrv	BLK_DEV *ataDevCreate
tffsDrv	STATUS tffsDrv	BLK_DEV *tffsDevCreate
memDrv	STATUS memDrv	STATUS memDevCreate
ramDiskCbio	无	CBIO_DEV_ID ramDiskDevCreate

后面两种设备不算是严格意义上的标准块设备。虽然各设备有类似的函数名，但要注意具体的传入参数却不相同，和具体的设备类型有关。

● 数据结构

作为一个块设备必须提供一个设备结构 BLK_DEV 来存取该设备。在该结构中定义一些与块设备相关的变量，如块大小和块数目等。还定义了统一函数接口，用于实现本设备读写操作等。统一函数接口的细节在下面一节介绍。BLK_DEV 对块设备的作用，与驱动表对字符

设备的作用相同，都向上层提供本设备的规范接口。

BLK_DEV 结构在 blkIo.h 中定义，如下所示：

```
typedef struct    {    /* BLK_DEV */
    FUNCPTR    bd_blkRd;        /* 读函数指针 */
    FUNCPTR    bd_blkWrt;       /* 写函数指针 */
    FUNCPTR    bd_ioctl;       /* ioctl 函数指针 */
    FUNCPTR    bd_reset;       /* 复位函数指针 */
    FUNCPTR    bd_statusChk;    /* 状态检查函数指针 */
    BOOL       bd_removable;    /* 是否为移动存储设备 */
    ULONG      bd_nBlocks;      /* 设备的块数 */
    ULONG      bd_bytesPerBlk;  /* 每块字节数 */
    ULONG      bd_blksPerTrack; /* 每磁道块数 */
    ULONG      bd_nHeads;       /* 磁头数 */
    int         bd_retry;       /* IO 错误重试次数 */
    int         bd_mode;        /* O_RDONLY | O_WRONLY | O_RDWR */
    BOOL        bd_readyChanged; /* 设备 ready 状态改变 */
} BLK_DEV;
```

● 函数接口

块设备有 5 个标准函数接口，函数原型都遵循一定的规范，包括返回值、函数名和参数。具体的函数实现由块设备的驱动程序提供。下面描述这几个函数的原型和责任。

```
STATUS xxBlkRd (DEVICE *pDev, int startBlk, int nBlks, char *pBuf);
```

该函数用于读取存储器上的数据，以块为单位。

pDev 是指向具体块设备结构的指针，如 ATA_DEV 结构。具体设备结构的第一个成员肯定是 BLK_DEV，上层传入的是 BLK_DEV 的指针，驱动实现中转为自己的结构指针使用。

startBlk 是开始读的块的起始位置。

nBlks 指需要读的块的数目。

pBuf 是存储接收数据的缓冲区的指针。

```
STATUS xxBlkWrt (DEVICE *pDev, int startBlk, int nBlks, char *pBuf);
```

该函数用于向存储器写入数据，以块为单位。

参数和读函数类似，只是 pBuf 是写入数据缓冲区的指针。

```
STATUS xxIoctl (DEVICE *pDev, int function, int arg);
```

该函数为多用途控制函数，一些不能归入标准函数中的功能在该函数中实现，如格式化等。它相当于函数接口的扩展“xxIoctl_function(arg)”。ioctl 函数是传递执行的，上层未解释的功能码才传递到驱动层，若驱动层也不能解释，就返回错误，并设置错误码为“S_ioLib_UNKNOWN_REQUEST”。

function 为功能码，如 FIODISKFORMAT。系统功能码在 ioLib.h 中定义，如果是设备自己的功能码，最好定义在 0x1000 以上，以避免和系统的功能码冲突。

arg 为传入参数，与具体功能有关。


```
STATUS xxReset (DEVICE *pDev);
```

该函数用于块设备硬件复位。当文件系统在设备上加载，或读写失败时，由文件系统调用。

```
STATUS xxStatusChk (DEVICE *pDev);
```

该函数用于块设备状态检查，由 open 或 creat 调用。若 xxStatusChk 失败，会设置 errno 表明原因，open 或 creat 也失败。这对可插拔的设备来说很重要。当设备出错或被拔出时，

这个函数返回错误，这时文件系统就不会继续往下操作。而当一个新的设备插上时，它设置 BLK_DEV 中的 bd_readyChanged 为 TRUE，然后返回 OK，这时 open 及 create 函数可以继续操作。新的设备可以自动地加入到系统中。设置 bd_readyChanged 和 dosFsReadyChanged、ioctl(FIODISKCHANGE)两函数执行效果一样，重新加载了文件系统。

 WindRiver, “Tornado Device Driver Workshop” 的第 6 章节。

7.3.2 ramDrv

RAM 存储器是最容易使用的块设备，在物理上没有块的分界，可以使用任意大小的块。对 RAM 的操作是很方便的，RAM 也是最快的存储设备，但数据的组织却比较麻烦。一般借助编译器帮助定义变量、数据结构来使用 RAM，或者通过内存管理机制来使用 RAM。而有时候需要一种更标准、高效和抽象的数据组织形式，如文件系统或数据库，以完成更高层的功能，避免原始的存储器使用。就像不可能对几十 G 的硬盘进行扇区操作一样，上层抽象软件的加入，可将开发者从繁琐的底层操作中解放出来。

VxWorks 灵活的文件系统分层结构，让用户可以在任意能抽象成块设备的存储器上加载需要的文件系统。ramDrv 库就是用来完成 RAM 存储器的这种抽象工作。上层文件系统将 RAM 的纯地址空间抽象为名称空间。就可以用文件的标准操作来访问 RAM 空间，而不用管内存的分配和释放，不用管数据实际存放在何处。

RAM 块设备的创建和其他块设备类似，ramDrv 中提供这样的接口。

```
BLK_DEV *ramDevCreate(char * ramAddr, int bytesPerBlk, int blksPerTrack,
                      int nBlocks, int blkOffset);
```

其中 ramAddr 为 RAM 设备的起始地址，可以传入自己分配内存的地址，也可传入 0，由系统帮助分配。bytesPerBlk 指明逻辑块的大小，传入 0 用缺省值 512。blksPerTrack 表示每磁道的数据块数，没什么实际用处，传入 0 则一个磁道包括所有的块。nBlocks 表明 RAM 设备的块数，传入 0 使用缺省值，最大为 51200 字节或最大内存区的一半。blkOffset 为偏移块数，一般为 0。下面为代码示例，创建一个大小为“512×100”的 RAM 块设备，并加载 dosFs 文件系统。

```
BLK_DEV *pBlkDev;
CBIO_DEV_ID pCbio;
pBlkDev = ramDevCreate (0, 0, 0, 100, 0);
/*使用 dcacheCbio*/
pCbio = dcacheDevCreate( (void *) pBlkDev, NULL, 1024, “/ram/” );
/*或者，使用 cbioLib 的基本封装*/
/*pCbio = cbioDevVerify( “/ram/”, FALSE);*/
dosFsDevCreate ( “/ram/”, pCbio, 0, NONE);
dosFsVolFormat ( “/ram/”, DOS_OPT_DEFAULT, NULL);
```

由于新的 dosFs 2.0 加入了 CBIO 层，并提供了 ramDiskCbio.c 将 RAM 封装为 CBIO 设备，可直接挂载文件系统，ramDrv 已经过时。由于 RAM 的快速访问特性，Cache 没有存在的必要，如果使用上面的创建过程，多余的 dcacheCbio 和 Cache 会隔在文件系统和 ramDrv 之间。最好使用 ramDiskCbio.c 提供的设备创建函数 ramDiskDevCreate，参数和 ramDevCreate 函数完全相同，只是返回的指针类型不同。同样创建一个“512×100”的 RAM 块设备。

```
CBIO_DEV_ID pCbio;
pCbio = ramDiskDevCreate (0, 0, 0, 100, 0);
dosFsDevCreate ( “/ram/”, pCbio, 0, NONE);
```

```
dosFsVolFormat ("/ram/", DOS_OPT_DEFAULT, NULL);
```

RAM 最大的缺点就是数据是暂存，系统掉电后，RAM 中数据会丢失，所以前面的初始化序列都包含文件系统格式化操作。如果使用 NVRAM，系统掉电后数据能保存，加载 RAM 文件系统时就不要调用格式化操作。

7.3.3 软盘

软盘驱动以源代码文件提供（“\target\src\drv\fdisk\nec765Fd.c”），提供了两个常用函数接口。

```
STATUS fdDrv(int vector, int level);
```

其中 `vector` 为中断向量，`level` 为中断等级，系统缺省使用 `FD_INT_VEC`（在 `config.h` 定义）和 `FD_INT_LVL`（在硬板相关的头文件中定义，如 `pc.h`）。该函数初始化软盘驱动，设置中断向量，并进行硬件初始化。在实际操作软盘前，需先调用该函数。


```
BLK_DEV *fdDevCreate(int drive, int fdType, int nBlocks, int blkOffset);
```

其中 `drive` 为软驱编号。`fdType` 为软盘类型，0 为 1.44MB，1 为 1.2MB，对于未支持类型，可以在 `fdTypes` 数组中增加。`nBlocks` 指明存储盘的块数，传入 0 使用整个盘。`blkOffset` 为偏移块数，一般用 0。该函数为软盘创建标准的块设备。

在 `usrFd.c` 中封装了一个标准的初始化函数 `usrFdConfig`，用于在软盘块设备上建立 `dosFs` 文件系统。注意该函数中没包括 `fdDrv` 调用，在使用该函数前，先需要由自己或系统调用 `fdDrv`。相关的参数可以在 BSP 的配置头文件中修改，或在组件配置窗口修改。

在加载软盘前，最好将软盘在 MSDOS 下进行格式化，以保证 Dos 下和 VxWorks 下的兼容性。

软盘是移动存储设备，使用中可能需要从系统中移除。在移除前应该先卸载文件系统。卸载时不要使用 `dosFsVolUnmount`，因为 `usrFdConfig` 未返回 `dosFsVolUnmount` 需要的 `DOS_VOL_CONFIG` 结构。应该使用文件描述符，通过 `ioctl` 的 `FIOUNMOUNT` 功能来卸载软盘文件系统。

 WindRiver, “Vxworks 5.4 Programmer’s Guide” 的 D.5 的 “Diskette Driver” 章节。

7.3.4 硬盘

硬盘有两种接口类型 IDE 和 ATA，分别对应 “\target\src\drv\hdisk\” 目录下的 `ideDrv.c` 和 `ataDrv.c`。不过现在多为 ATA 硬盘，且 `ataDrv` 也支持 IDE 硬盘。同软盘驱动一样，提供两常用函数接口来加载文件系统：`ataDrv` 和 `ataDevCreate`。具体的参数说明，请参考库手册。

在 `usrAta.c` 中封装了一个标准的初始化函数 `usrAtaConfig`，用于在硬盘第 1 分区上建立 `dosFs` 文件系统。注意该函数中没包括 `ataDrv` 调用，在使用该函数前，先需要由自己或系统调用 `ataDrv`。相关的参数可以在 BSP 的配置头文件中修改，或在组件配置窗口修改。

在加载硬盘前，最好先在 MSDOS 下进行分区和格式化，以保证 Dos 下和 VxWorks 下的兼容性。

至于多分区的加载，参考前面关于 “`dpartCbio`” 中的描述。另外可以阅读 “\target\src\usr\” 目录下 `usrFdiskPartLib.c` 文件中的说明和代码。`usrFdiskPartLib` 支持 MSDOS 的 FDISK 的分区表格式，和 `dpartCbio` 一起完成硬盘分区的管理。

```
pBlkDev = ataDevCreate(0, 0, 0, 0);
```

```
usrPartDiskFsInit(pBlkDev);
```

 WindRiver, “VxWorks 5.4 Programmer’s Guide” 的 D.5 章节。

7.3.5 TureFFS

TureFFS 为各种 Flash 存储器提供通常的块设备接口，是 M-Systems 公司为 VxWorks 操作系统所做的定制实现。

● Flash 存储器

Flash 存储器和普通存储盘有区别。Flash 存储器为固态[solid-state]设备，没有运动的机械部件，具有寿命长、可靠性高、耗能少和体积小等优点，很适合用于嵌入式系统。但是 Flash 也有其缺点，如写入数据前需要擦除操作，并且只能进行块擦除；有限的擦除和写入次数，一般约为 10 万次；擦除和写入操作比较耗时，且不能同时读取；对 Flash 不能像普通 RAM 一样直接写入，需要执行系列指令。


NOR 和 NAND 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 NOR Flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发布了 NAND Flash 结构。

通常所说的 Flash 存储器多指 NOR 型存储器，一般用于程序映象的存储和运行。NOR 的特点是芯片内执行(XIP, eXecute In Place)，这样应用程序可以直接在闪存内运行，不必再把代码读到系统 RAM 中。而 NAND 更适合大容量的数据存储。两者的主要区别如表 7-16 所示。

表 7-16 NOR 和 NAND

	NOR	NAND
容量	1MB~32MB	16MB~512MB
XIP	是	否
性能	擦除很慢(5s)，写较慢	快速擦除(3ms)、写和读
可靠性	一般	低（需要 EDC/ECC 和坏块管理）
擦除次数	1 万~10 万	10 万~100 万
擦除块	大	小
接口	标准内存接口	仅 IO，CLE、OLE 和 ALE 信号必须变化
访问方法	随机	顺序
易用性	简单	复杂
用途	代码或小量数据	大量数据存储
价格	高	低

TFFS 对 NOR 和 NAND 的 Flash 存储器都提供支持。NOR 型如 28F008、28F016、CFI 接口 Flash 和 AMD Flash 等；NAND 型如 NFDC2048 和 DOC 等。用户自己特殊的 Flash 芯片驱动可根据上面类似型号的代码修改。

 寄存器, “NAND 和 NOR flash 详解”，C51BBS 论坛。

 M-Systems, “Two Technologies Compared: NOR vs. NAND White Paper”。

● 算法分析

Flash 存储器的特性使得其对上层软件有特别的要求。

✧ 平均使用[wear-leveling]: 如上所说, Flash 扇区的擦除次数都有限制, 对 Flash 的使用必须充分地考虑该特性, 最好能均匀使用 Flash 的每个扇区, 以延长 Flash 的使用寿命。

✧ 高效垃圾回收[garbage collection]: 任何存储器在分配使用一段时间之后, 都会出现空区和碎片数据, 这就需要进行垃圾回收, 以保证存储器空间的高效使用。而 Flash 擦除以扇区为单位, 垃圾的回收也应以扇区为单位, 先移动扇区数据, 再擦除整个扇区。

✧ 掉电安全[crash/powerdown-safe]: 嵌入式系统一般运行环境比较恶劣, 但要求较高的可靠性。这要求无论程序崩溃或系统掉电, 都不能影响数据的一致性和完整性。因为要求块擦除, 块中原来数据的保存就比较麻烦, 因此数据写入, 垃圾回收等操作对系统异常中止都非常敏感, 极易造成数据丢失和数据垃圾。

✧ 低空间消耗[Low OverHead]: Overhead 指上层软件管理结构在 Flash 存储器上的空间消耗, 这部分空间用于上层软件抽象, 而不能用于实际数据的存储。而一般嵌入系统中存储器空间有限, 低 Overhead 可以提高有用数据存储空间。

TrueFFS 为了将 Flash 存储器抽象为普通的块设备, 将 Flash 存储器映射为一系列连续的块。TrueFFS 使用 block-to-flash 转换系统, 基于动态维护的映射图。当块被修改、删除和垃圾回收时, 映射图动态调整。从块中读取数据比较简单, TrueFFS 将块编号直接转换为 Flash 存储器地址, 从该地址读取数据返回。对于写操作, 如果目标块从未写入过, 同读操作一样简单; 如果目标块已写入过, TrueFFS 找另外的空闲区写入, 数据安全写入后, TrueFFS 更新映射图, 将该块指向写入的新 Flash 地址。

修改块映射到新的 Flash 地址的这种机制保证了 Flash 存储区的均匀使用。但是, 有些块从未修改过, 这些存储区就是静态的, 而 TrueFFS 的均匀使用特性需通过块修改机制来达到, 所以 TrueFFS 的均匀使用算法不是完备的。完备的算法要求所有 Flash 存储区都参与擦除写入循环, 需要所有块移动, 但使用性能上有所降低。TrueFFS 算法能保证各扇区的平均擦除, 从实用的角度没有问题, 而性能有较大的改善。另外, TrueFFS 算法还避免了其他算法易出现的死锁问题。

虚拟块映射也简化了垃圾回收。随着不断的写入, Flash 存储器中出现越来越多的脏块, 在擦除前不能再使用, 上层软件必须支持垃圾回收算法, 否则存储空间会最终耗尽。而 Flash 的擦除单元较大, 包含很多块, 脏块回收的同时需注意有效块的保存。一般将回收单元的有效块先复制到别的单元中, TFFS 更新映射图, 再擦除回收单元。TFFS 的上层软件不会感觉到数据块实际存储位置的变化。垃圾回收不能频繁随意进行, 否则会导致存储器寿命缩短和操作性能下降。在 TFFS 中, 垃圾回收操作由块分配算法触发。块分配算法总是在一个擦除单元里维持一片连续可用存储区, 若存储区太小时, 触发垃圾回收。回收时按一定标准搜寻最需回收的单元, 执行回收算法。

由于 Flash 的特性, 在写入、垃圾回收、格式化时可能出现数据错误。TFFS 不能避免正写入数据的丢失, 但能保证已存在 Flash 上数据的安全性, 不会破坏文件和目录结构。TFFS 采用“写后擦除”算法来保证这种可靠性。当更新 Flash 扇区时, 以前的数据先不擦除, 直到新数据被确认写入。若新数据写入中途失败, 旧数据仍然有效。该算法也保证了映射图的一致性, 虽然 TFFS 平时使用 RAM 中的映射图, 但在 Flash 中也保存了备份, 若新映射图写入

失败，备份仍然有效，系统恢复时会利用该备份重构 RAM 中映射图。虽然备份映射图可能存在于 Flash 的任何位置，TFFS 利用确知位置的擦除单元头信息来重构 RAM 中映射图。TFFS 对写入数据采用读回校验算法，若第一次失败，TFFS 不报告用户，而采用动态映射机制在另一不同位置尝试写入，这种错误自动恢复机制在 Flash 快要损坏时特别有用。TFFS 在垃圾回收时使用两个数据暂存扇区，当一个失败时用另一个，若两都失败则转为只读状态，但原来的数据不会损失。在 TFFS 格式化时能标志排除 Flash 中损坏扇区，保证余下的空间能可靠使用。

● 结构分析

TrueFFS 由 3 层实现：翻译层（FTL）、MTD 层和 Socket 层，如图 7-3 所示。

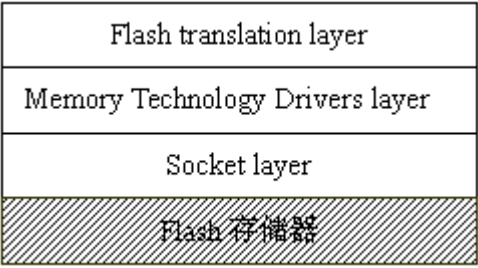


图 7-3 TrueFFS 层次结构

FTL 为上层软件提供标准块设备接口，实现上面描述的各种算法，如块映射、平均使用、垃圾回收和数据保护等。根据 Flash 存储器的类型，有 3 种类型的 FTL：NOR、NAND 和 SSFDC。

MTD 层和具体的芯片相关，用于实现 Flash 存储器相关的操作指令序列。

Socket 层为具体硬板提供接口，实现电压控制，基地址设置，写保护控制等。Socket 这名称来源于 Flash 存储器的插座，

在 VxWorks 中，FTL 以库型式提供，包括 `tffsDrv`、`tffsLib`、以及 `ftl` 和 `f1` 开头的目标模块。MTD 以源代码形式提供，在“`\target\src\drv\tffs`”目录下，以 Flash 芯片型号命名，如 `i28f008.c`。如果没有特殊芯片的代码，可以参考近似代码。Socket 层也以源代码提供，在“`\target\src\drv\tffs\sockets`”目录下，按硬板的具体型号命名，自己的硬板一般找不到对应的代码，也需参考写一个，所幸修改较少。`sysTffs.c` 中还包含编译 `tffsConfig.c`，在“`\target\src\drv\tffs`”目录下，只要在 `mtdTable` 数组中添加自己芯片的标识函数入口即可。

在为自己的硬件定制 TFFS 时，可以将 `xxxxmtd.c`、`tffsConfig.c` 和 `sysTffs.c` 直接添加到自己的 BSP 目录中，方便代码管理和 Tornado 重装。

● 共享 Flash

如果程序映象实际在 RAM 中运行，程序映象和 TFFS 可以共享同一 Flash 存储器。比如只有一片 Flash 存储器，可以将 BootRom 和 TFFS 同时存于其中，VxWorks 可以存放在 TFFS 中。或者 VxWorks_rom 和 TFFS 共存。TFFS 可以指定管理空间范围，不会破坏共存的程序映象。但如果程序映象是从 Flash 存储器上运行的（XIP），则 TFFS 只能读，因为 Flash 在擦除过程中不能读，而写会引起垃圾回收的擦除操作。

TFFS 库提供一种共享机制，在 `tffsDevFormat` 中可以指定保留一段存储区，用于启动程序映象的存放。可以使用 `tffsBootImagePut` 函数来操作这块保留存储区。或者在 Socket 注册

时就保留需要空间，TFFS 就不会对该保留区进行管理。

● 初始化

在使用 TFFS 前，首先需要调用 `tffsDrv`。该函数用于建立 TFFS 管理需要的互斥信号量、全局变量和数据结构，并完成 Socket 注册和启动 Socket 查询任务 `tTffsPTask`。`tffsDrv` 一般都能调用成功，主要需要 Socket 层的接口函数，如图 7-4 所示。

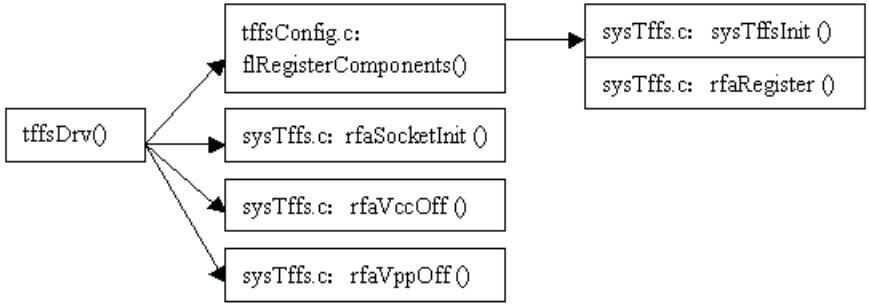


图 7-4 tffsDrv 函数调用

`tTffsPTask` 任务是周期性循环任务，不是必须的，可以通过设置 `flcustom.h` 中的 `POLLING_INTERVAL` 为 0 来禁止，缺省为 100ms。`tTffsPTask` 任务主要用于探测 Flash 存储器插拔，Vpp 和 Vcc 电源延时关闭等。

`tffsDrv` 调用成功后，在 `tffsDevCreate` 之前，需要先调用 `tffsDevFormat` 格式化 Flash。

`STATUS tffsDevFormat(int tffsDriveNo, int arg)`

其中 `arg` 为指向 `tffsDevFormatParams` 结构的指针，若用 0，则使用缺省值。结构在 `tffsDrv.h` 中定义，其中 `bootImageLen` 就是上面讲的保留区长度。`TFFS_STD_FORMAT_PARAMS` 中定义了缺省值。注意格式化分为 FAT 格式化和 FTL 格式化，FTL 格式化比较慢，缺省选项为 `FTL_FORMAT_IF_NEEDED`，若在干净的 Flash 上格式化，会同时调用 FTL 和 FAT 格式化；若在已建立 TFFS 的 Flash 上进行，则只调用 FAT 格式化，可能连 Flash 擦除操作都不进行。对于已崩溃的 TFFS 盘，最好使用 `FTL_FORMAT` 标志进行格式化。在定制自己的 TFFS 过程中，格式化是否成功是最关键的一步，能充分验证自己的 MTD 是否正确。`tffsDevFormat` 调用的 MTD 相关函数如图 7-5 所示。

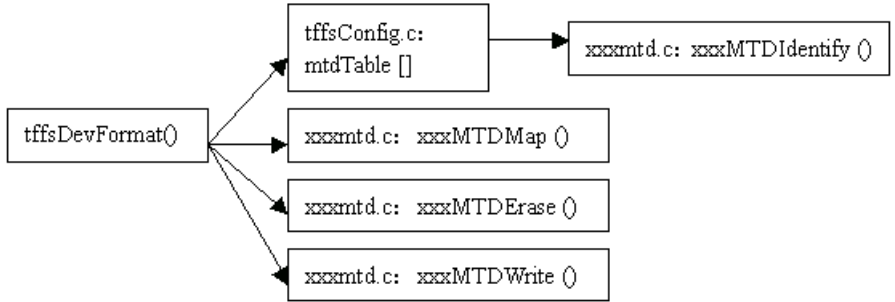


图 7-5 tffsDevFormat 函数调用

在调试 TFFS 时，`tffsDevFormat` 不容易通过。如果出现错误，函数的返回信息太简单，不能准确定位错误，最好在上面的 MTD 函数中添加调试信息，包括函数名、传入参数和返回结果。格式化完成的 Flash 存储器的每擦除单元的起始地址内容如下：


```
d 0xffc80000
ffc80000: 1303 4349 5346 ff00 4654 4c31 3030 0001 *.CISF..FTL100.*
ffc80010: 0100 0000 0000 0910 0000 1800 003a 1600 *.*****:.*
ffc80020: 0000 0100 1700 00ff 0000 0000 0000 0000 *.*****.*
ffc80030: 4400 0000 ffff ffff ffff ffff ffff ffff *D.....*
ffc80040: ffff ffff 3000 0000 3000 0000 40d2 ffff *....0...0...@...*
ffc80050: 40d4 ffff 40d6 ffff 40d8 ffff 40da ffff *@...@...@...@...*
ffc80060: 40dc ffff 40de ffff 40e0 ffff 40e2 ffff *@...@...@...@...*
ffc80070: 40e4 ffff 40e6 ffff 40e8 ffff 40ea ffff *@...@...@...@...*
ffc80080: 40ec ffff 40ee ffff 40f0 ffff 40f2 ffff *@...@...@...@...*
ffc80090: 40f4 ffff 40f6 ffff 40f8 ffff 40fa ffff *@...@...@...@...*
ffc800a0: 40fc ffff 40fe ffff 4000 0000 4002 0000 *@...@...@...@...*
ffc800b0: 0000 0000 4006 0000 4008 0000 0000 0000 *....@...@.....*
ffc800c0: 400c 0000 400e 0000 0000 0000 4012 0000 *@...@.....@...*
ffc800d0: 4014 0000 4016 0000 4018 0000 401a 0000 *@...@...@...@...*
ffc800e0: 401c 0000 401e 0000 4020 0000 4022 0000 *@...@...@...@...*
ffc800f0: 4024 0000 4026 0000 4028 0000 402a 0000 *@$.@&...@(.@*.*
```

最前面 68 个字节为擦除单元头,后面的每 4 字节对应本擦除单元的一个扇区(512 字节),按“little-endian”序,第 1 字节表示扇区状态,如 40 表示数据扇区,30 表示格式化扇区,0 表示垃圾扇区等。紧接单元头的两个扇区索引为格式化扇区,用于存放本擦除单元所有扇区的索引。后 3 字节表示映射图中的块号。未归入映射图的扇区,后两字节为 0xFFFF;从映射图中抛弃的旧数据扇区 4 个字节全为 0,标为垃圾扇区。

Flash 格式化成功后,FTL 和 FAT 结构已经形成,可调用 `tfFsDevCreate` 创建标准块设备,一般都能成功。加载 `dosFs` 文件系统时就和其他块设备一样使用。

在 `usrTffs.c` 中封装了一个标准的初始化函数 `usrTffsConfig`，用于在 TFFS 块设备上建立 `dosFs` 文件系统。注意该函数中没包括 `tffsDrv` 调用，在使用该函数前，先需要由自己或系统调用 `tffsDrv`。

 WindRiver, “TrueFFS for Tornado Programmer's Guide 1.0 Edition 1”.

 WindRiver, “PCMCIA for x86 Release Notes and Supplement”.

7.3.6 memDrv

memDrv 驱动和普通的块设备驱动不同,可不经由文件系统挂载在 IO 系统中,作为伪 IO 设备,用户通过 IO 标准接口直接访问内存空间。内存空间的地址和大小在设备创建时指定。

memDrv 与 RAM 盘有区别。RAM 盘上必须建立文件系统，才能由标准 IO 接口操作；而 memDrv 可直接接入 IO 系统，可以看作简单的文件系统。RAM 盘上文件的存储地址是不定的，而 memDrv 上的文件是和绝对地址、大小关联的。

memDrv 设备可用于 VxWorks 多次启动间保持数据，或多 CPU 之间数据共享。另外，可以将一些只读文件或目录编译连接进 VxWorks 映象，运行时再用 memDrv 加载使用。先将需加入文件的目录通过 memdrvbuild 主机工具转化为一个 C 代码格式的数组文件。比如将一些只读的 HTML 文件编译入映象，支持 WEB 服务器的网页浏览。

```
memdrvbuild web /*web 目录中包含网页文件*/
```

这样会生成 web.c 和 web.h 文件，将它们加入工程就可编译入映像。web.c 中，每个文

件对应一个大数组，包含文件内容，还包括文件表信息和两个使用函数。MEM_DRV_DIRENTRY 结构在 memDrv.h 中定义。

```
/* File table for directory web.
web 目录下包含子目录 websub 和一个文件 index.html */
static MEM_DRV_DIRENTRY dirweb[] = {
    { "index.html", dataweb_index_html, NULL, 16277 },
    { "websub", NULL, dirweb_websub, 6 },
};
/* File table for directory web\websub.
websub 是 web 目录下的子目录，包含 6 个 htm 文件 */
static MEM_DRV_DIRENTRY dirweb_websub[] = {
    { "NOTE.HTM", dataweb_websub_NOTE_HTM, NULL, 8392 },
    { "LINKS.HTM", dataweb_websub_LINKS_HTM, NULL, 10470 },
    { "DOWN.HTM", dataweb_websub_DOWN_HTM, NULL, 9026 },
    { "DOCS.HTM", dataweb_websub_DOCS_HTM, NULL, 12712 },
    { "CODE.HTM", dataweb_websub_CODE_HTM, NULL, 10503 },
    { "AMINE.HTM", dataweb_websub_AMINE_HTM, NULL, 10783 },
};
/* Setup function for directory tree /web.
在 VxWorks 运行时调用该函数加载 memDrv 设备 “/web” */
STATUS memDrvAddFiles_web (void)
{
    return memDevCreateDir ("/web", dirweb, 2);
}
/* Shutdown function for directory tree /web. */
STATUS memDrvDeleteFiles_web (void)
{
    return memDevDelete ("/web");
}
```

memDrv 以源代码文件提供 (“\target\src\usr\memDrv.c”), 用户通过它可以了解 memDrv 的底层实现机制, 以便灵活使用它。关于多 CPU 数据共享的 memDrv 使用, 可参考库手册中 memDrv 条目。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 3.7.3 章节。

7.4 常见问题解答

● 如何在 VxWorks 下挂载文件系统？

解答：本章前面已经介绍了相关概念，不同设备或不同文件系统的挂接略有不同。但基本步骤类似，就是在物理块设备上创建抽象块设备或 CBIO 设备，在抽象设备上挂接文件系统，再将文件系统挂接入 IO 系统。

不过用户在具体应用过程中，可以不关心这些细节，直接使用 “\target\config\comps\src” 目录下的代码封装接口即可。不同类型设备对应不同源代码和函数接口，如对于 TFFS 设备，对应 usrTffs.c 和 usrTffsConfig。另外还需要执行初始化函数，一般定义相应的宏就行了，如 tffsDrv 对应 INCLUDE_TFFS。

如果需要进行一些定制操作，可参考封装接口编写自己的函数。

● 如何查看文件系统的空间使用情况？

解答：使用 `chkdsk` 函数，用户可以参考 `chkdsk` 的源代码，编程实现空间查询。

- 如何从文件描述符知道所对应的设备？

解答：使用 `iosFdDevFind` 库函数。

第 8 章 网络通信

8.1 概览

VxWorks 具有强大的网络功能。标准的 VxWorks 采用了与 4.4 BSD TCP/IP 兼容的实时网络协议栈,使得网络开发变得易于进行和方便移植。由于采取了诸如在 TCP 层的数据拷贝、使用 Hash 表、缓冲管理方法等的优化,使得其在各类应用中都有非常好的性能。通过丰富的网络协议和组件, VxWorks 提供了绝大多数的网络服务功能。网络已成为 VxWorks 系统与其他系统间连接的主要途径。VxWorks 网络通信使用的传输媒介包括松耦合的串行线路(使用 SLIP、CSIP 和 PPP)、标准的以太网连接以及紧耦合的利用共享内存的背板总线。传输媒介的上层通过 TCP/IP 与 UDP/IP 协议进行数据通信。网络通信是 VxWorks 系统主机与目标机调试的主要连接, VxWorks 也提供了 FTP、Telnet、BOOTP 等各种网络服务。

VxWorks 网络协议的特性如下。

- ✧ 支持最新的协议,如 IP Multicast、CIDR 和 RFC-1323 等。
- ✧ 可配置成 IP、IP+UDP、IP+UDP+TCP。
- ✧ 可作为 DHCP 服务器、DHCP 客户端和中继代理等。
- ✧ 可作为 DNS 客户端。
- ✧ 可作为 SNTP 服务器、SNTP 客户端。
- ✧ 支持 IP 各类服务,并为 IP 转发做过优化。
- ✧ 支持 RIPv1 和 RIPv2。
- ✧ 可选支持 OSPF。
- ✧ 具有路由策略。
- ✧ 支持 IP/ICMP/IGMP。
- ✧ 支持 ARP/代理 ARP。
- ✧ 支持 TCP、UDP。
- ✧ 有 BSD 4.4 兼容的 Socket 库。
- ✧ 可作为 BOOTP 客户端。
- ✧ 可作为 RPC/NFS 服务器及客户端。
- ✧ 可作为 FTP 服务器。
- ✧ 可作为 RSH 客户端和 Telnet 服务器。
- ✧ 可作为 RLOGIN 客户端和服务器。
- ✧ 支持 PPP/SLIP/CSLIP。
- ✧ 对 TCP 连接和路由表查询做过优化。
- ✧ 在 TCP、UDP 层使用了零拷贝技术。
- ✧ 新的驱动结构,支持在同一网络设备上运行多种协议。
- ✧ 集成 MIB-II 支持。

VxWorks 所支持的网络协议组件结构如图 8-1 所示。

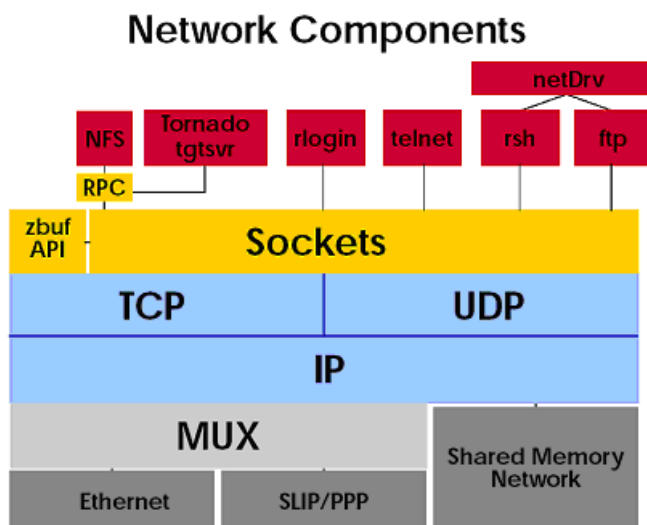


图 8-1 VxWorks 的网络协议和组件

下面对网络通信中的两个重要概念做进一步介绍。

● MUX 接口

为了支持象多播 (Multicasting)、轮询模式的以太网 (Polled-mode Ethernet) 以及零拷贝 (Zero-copy) 等特性, VxWorks 在下面网络驱动层 (Ethernet、SLIP 等) 和网络协议层 (TCP/IP) 之间增加了中间层, 叫做 MUX (Multiplex) 的接口层。MUX 负责管理网络协议层与底层硬件接口间的交互, 使得网络协议层细节与硬件无关, 从而避免使用输入输出 Hook 函数来过滤所传送的数据包。

● Socket

VxWorks 有两种 Socket 调用: 与 BSD 4.4 源码兼容的 Socket 调用和 Zbuf 的 Socket 调用。Zbuf 就是指利用零拷贝技术在应用程序和网络接口间建立的共享缓冲区 (Zerobuf), 这样应用程序就可以直接的读写 Socket 而不用在网络缓冲区和自己的缓冲区来回拷贝数据, 提高了程序效率。但 Zbuf 并不符合标准, VxWorks 为此提供了特别的 zbufSockLib。

 WindRiver, “VxWorks Network Protocol Toolkit Programmer’s Guide”。

8.2 网络驱动

VxWorks 支持两种形式的网络驱动, 一种是 BSD 驱动, 支持通用的 BSD 4.4 网络、API 结构等, 和大多数 BSD 网络的驱动类似。另一种是 END 网络驱动, 是 VxWorks 独有的, 叫做增强型的网络模型。它主要特点就是加了一个 MUX 模块, 管理所有 END 设备, 使得协议与硬件无关, 并且通过 MUX 支持数据包轮询发送和接收, 这对 WDB 的系统级调试是关键。因为 WDB 工作在系统级模式时要关闭中断, 这时通过网口的通信只能靠轮询。基本上也只有 WDB 需要轮询支持, 上层协议不需要。END 驱动在底层也要转换成 BSD 的形式。

8.2.1 网络驱动加载配置

本节介绍 BSD 驱动和 END 驱动的加载。网络驱动的调用主要在 “\target\src\config\usrNetwork.c” 文件中。

● 加载 BSD 驱动

首先来看一下 BSD 驱动的调用过程。VxWorks 系统执行的第一个任务 (“target/config/all/usrConfig.c” 文件中的 usrRoot 函数) 调用 “target/src/config/usrNetwork.c” 文件中的 usrNetInit 函数，通过数组表 usrNetIfTbl 初始化相应的 BSD 网卡驱动。

在 usrNetwork.c 中的调用过程如下。

usrNetInit 函数中调用 usrNetIfAttach。

```
#ifdef INCLUDE_BSD
if (!attached){
    if ( (usrNetIfAttach (pNetDev, params.unitNum, pBootString) !=OK))
        return (ERROR);
    attached = TRUE;
}
#endif /*INCLUDE_BSD*/
```

usrNetIfAttach 函数中查找数组表 usrNetIfTbl。

```
for (pNetIf = usrNetIfTbl; pNetIf->ifName != 0; pNetIf++)
{
    If (strcmp (buf, pNetIf->ifName) == 0)
        break;
}
```

网络 BSD 驱动数组表 usrNetIfTbl 在文件 “target/src/config/usrNetwork.c” 中定义。

```
NETIF usrNetIfTbl [] = /* network interfaces */
{
    /* 下面是定义包含的各种网络驱动 */
#ifdef NETIF_USR_ENTRIES    /* 用户自定义*/
    NETIF_USR_ENTRIES
#endif
    .....
#ifdef INCLUDE_DC          /* 从 DEC 芯片启动,即系统有 NVRAM 存在,现在已过时*/
{    "dc", dcattach, (char*)IO_ADRS_DC, INT_VEC_DC, INT_LVL_DC,
    DC_POOL_ADRS, DC_POOL_SIZE, DC_DATA_WIDTH, DC_RAM_PCI_ADRS,
    DC_MODE },
#endif /* INCLUDE_DC */
#ifdef INCLUDE_FEI          /* 如果定义了 INCLUDE_FEI,初始化 Intel 网卡 */
{    "fei", feiattach, (char*)FEI_POOL_ADRS, 0, 0, 0, 0},
#endif /* INCLUDE_FEI */
#ifdef INCLUDE_EX           /* Excelan 网卡 */
{    "ex", exattach, (char*)IO_ADRS_EX, INT_VEC_EX, INT_LVL_EX,
    IO_AM_EX_MASTER, IO_AM_EX },
#endif /* INCLUDE_EX */
#ifdef INCLUDE_ENP          /* CMC 网卡 */
{    "enp", enpattach, (char*)IO_ADRS_ENP, INT_VEC_ENP, INT_LVL_ENP,
    IO_AM_ENP },
#endif /* INCLUDE_ENP */
#ifdef INCLUDE_ENE          /* ENE 网卡 */
{    "ene", eneattach, (char*)IO_ADRS_ENE, INT_VEC_ENE, INT_LVL_ENE},
#endif /* INCLUDE_ENE */
    .....
}
```

从上面可以看出，BSD 网络驱动只需在 BSP 配置文件 config.h 中将网络

(INCLUDE_NETWORK)、BSD 网卡宏定义(INCLUDE_xxx)和一些 I/O 参数(一般不需要)加入或改动, 则文件 usrNetwork.c 中代码就会进行相应的初始化操作。这里以 NE2000 网卡为例, IO 地址为 0x300, 中断为 5。

打开 BSP 下 config.h, 配置网卡型号。

```
#define INCLUDE_ENE /* 定义 Eagle/Novell NE2000 驱动*/
/*undef 其他型号的网卡*/
#undef INCLUDE_ELT /* include 3COM EtherLink III interface */
#undef INCLUDE_ESMC /* include SMC 91c9x Ethernet interface */
#undef INCLUDE_FEI /* include Intel Ether Express PRO100B PCI */
```

配置中断, 找到如下语句, 根据目标板修改。

```
#define IO_ADRS_ENE 0x300 /*网卡 I/O 基址*/
#define INT_LVL_ENE 0x5 /*网卡中断号*/
```

设置好 DEFAULT_BOOT_LINE 中的目标机地址, 如 “e=192.166.0.2”。如果系统不是从网络引导的, 则还需添加 “o=ene”。一切配置完毕后生成 VxWorks 下载启动, 此时 ping 目标机应该成功。

● 加载 END 驱动

END 驱动的启动过程为: usrRoot 函数调用 “target\src\config\usrNetwork.c” 文件中 usrNetInit 函数。usrNetInit 中通过 END 设备配置表 endDevTbl 添加 MUX END。

```
#ifdef INCLUDE_END
.....
/* Add in mux ENDs. */
for (count = 0, pDevTbl = endDevTbl;
pDevTbl->endLoadFunc != END_TBL_END;pDevTbl++, count++)
{
    /* Make sure that WDB has not already installed the device. */
    if (!pDevTbl->processed)
    {
        pCookie = muxDevLoad(pDevTbl->unit,
                             pDevTbl->endLoadFunc,
                             pDevTbl->endLoadString,
                             pDevTbl->endLoan, pDevTbl->pBSP);

        if (pCookie == NULL)
        {
            printf("muxDevLoad failed for device entry %d!\n", count);
        }
    }
    else
    {
        pDevTbl->processed = TRUE;
        if (muxDevStart(pCookie) == ERROR)
        {
            printf("muxDevStart failed for device entry %d!\n", count);
        }
    }
}
}
#endif /* INCLUDE_END */
```

表 endDevTbl 包含了网络设备的具体参数, 在 configNet.h 中定义。

```
END_TBL_ENTRY endDevTbl [] =
```

```
{
    .....
#ifdef INCLUDE_ENE_END
{ 0, END_ENE_LOAD_FUNC,
  END_ENE_LOAD_STRING, END_ENE_BUFF_LOAD, NULL},
#endif /* INCLUDE_ENE_END */
{ 0, END_TBL_END, NULL, 0, NULL, FALSE},
};
```

相关的网络驱动程序装载函数入口和硬件的物理设置数据串在 configNet.h 中定义，一般都有缺省定义，不需要修改。

```
#ifdef INCLUDE_ENE_END
#define END_ENE_LOAD_FUNC    sysNe2000EndLoad
#define END_ENE_BUFF_LOAD    1
#define END_ENE_LOAD_STRING "" /* created in sysNE2000EndLoad */
IMPORT END_OBJ * END_ENE_LOAD_FUNC (char *, void*);
#endif /* INCLUDE_ENE_END */
```

NE2000 网络驱动装载函数 sysNe2000EndLoad 在 sysNe2000End.c 中定义，它根据 I/O 基址、中断号等形成最终的字符串来装载 NE2000 的 END 驱动。

根据上面的分析来加载 VxWorks 的 END 网络驱动，仍以 NE2000 网卡为例，步骤如下。

在文件 config.h 中添加如下定义。

```
#define INCLUDE_NETWORK
#define INCLUDE_END
```

同 BSD 网卡驱动的配置，定义网卡型号。

```
#define INCLUDE_ENE /* 定义 Eagle/Novell NE2000 驱动*/
```

定义 IO 地址和中断。

```
#define IO_ADRS_ENE    0x300 /*网卡 I/O 基址*/
#define INT_LVL_ENE    0x05 /*网卡中断号*/
```

在 configNet.h 中加入 END 驱动的入口函数和一些相关的初始化字符串（ENE 网卡相关信息程序中已包括）。

```
#define xxx_LOAD_FUNC    xxxxxEndLoad
```

这样在定制的 BSP 中包含了 END/MUX, 系统网络初始化函数 muxDevLoad 会根据这个表初始化 END 网络。

8.2.2 增加第二块网卡

在实际应用中经常需要有双网卡或更多，根据上面的分析可以很容易增加新网卡。这里以 END 网卡驱动为例，BSD 网卡类似。

● 增加不同型号的网卡

在原 NE2000 网卡上（I/O 基址 0x300，中断 5）增加 3Com EtherLink III 网卡（I/O 基址 0x320，中断 9）。

在 config.h 增加 3Com 网卡相关定义

```
#define INCLUDE_ENE
#define INCLUDE_ELT
#define IO_ADRS_ENE    0x300
#define INT_LVL_ENE    0x5
#define IO_ADRS_ELT    0x320
#define INT_LVL_ELT    0x09
```

```
#define NRF_ELT          0x00
#define CONFIG_ELT 0    /* 0=EEPROM 1=AUI 2=BNC 3=RJ45 */
```


在组件配置窗口下将“basic network initialization components”中的 IP_MAX_UNITS 参数设成 2。生成并下载 VxWorks 映像。假设启动网卡为 ene，则该网卡在 VxWorks 启动后即自动生成一个 END 设备，并且自动完成 MUX 和 IP 挂接。若不是自动加载，可以在 Shell 中手动启动网卡。

```
-> usrNetEndDevStart "elPci",0
```

注意参数中的“0”为单元号[unitNum]，表示同一型号设备在系统中的序号，这里配置的网卡为系统中的第一块 3Com 网卡，所以 unitNum 为“0”。

```
-> usrNetIfConfig "elPci",0,"138.203.175.6","192.168.175.12",0xffffffff00
```

注意两个网卡要在不同的子网下。

 可以使用 muxShow 显示已经加载的网卡驱动列表。

● 增加相同型号的网卡

增加相同型号的网卡除了更改 config.h 外还要根据“加载 END 驱动”中的分析在 endDevTbl 数组中增加相关的设备信息，以及更改 sysNe2000EndLoad 来根据参数绑定不同地址和参数。以 NE2000 (I/O 基址 0x300、中断 5) 和 NE2000 (I/O 基址 0x320、中断 9) 为例。

在 config.h 增加新网卡定义。

```
#define INCLUDE_ENE
#define INCLUDE_ENE1 /*新加网卡*/
#define IO_ADRS_ENE 0x300
#define INT_LVL_ENE 0x5
/*新加网卡*/
#define IO_ADRS_ENE1 0x320
#define INT_LVL_ENE1 0x9
#define INT_VEC_ENE1 (INT_VEC_GET (INT_LVL_ENE1))
```

在 configNet.h 的 endDevTbl 中增加新网卡。

```
{
.....
#ifdef INCLUDE_ENE_END
    {0, END_ENE_LOAD_FUNC,
END_ENE_LOAD_STRING, END_ENE_BUFF_LOAN, NULL},
    {1, END_ENE_LOAD_FUNC,
END_ENE_LOAD_STRING, END_ENE_BUFF_LOAN, NULL},
/*新加的网卡, unitNum=1*/
#endif /* INCLUDE_ENE_END */
}
```

修改 sysNe2000End.c 中的 sysNe2000EndLoad，根据不同的 unitNum 绑定不同的 I/O 基址、中断号等。

```
if (pParamStr[0] == '0') /*原网卡, unitNum=0*/
    sprintf (cp, ne2000ParamTemplate,
            IO_ADRS_ENE,
            INT_VEC_ENE,
            INT_LVL_ENE,
            ENE_BYTE_ACCESS,
            ENE_USE_ENET_PROM,
            ENE_OFFSET);
```



```

else if (pParamStr[0] == '1') /*新加的网卡, unitNum=1*/
    sprintf (cp, ne2000ParamTemplate,
            IO_ADRS_ENE1,
            INT_VEC_ENE1,
            INT_LVL_ENE1,
            ENE_BYTE_ACCESS,
            ENE_USE_ENET_PROM,
            ENE_OFFSET);
printf ("ne2000EndLoad: %s.\n", paramStr);
if ((pEnd = ne2000EndLoad (paramStr)) == (END_OBJ *)ERROR) {
    printf ("Error: NE2000 device failed ne2000EndLoad routine.\n");
}

```

在组件配置窗口下将 IP_MAX_UNITS 参数设成“2”。生成并下载 VxWorks 映象，启动新网卡。在 Shell 下，将新网卡绑上去，注意单元数[unitNum]为“1”。

```

-> usrNetEndDevStart "ene", 1
-> usrNetIfConfig "ene", 1, "138. 203. 175. 6", "192. 168. 175. 12", 0xffffffff00

```

8.2.3 添加新的网卡驱动程序

VxWorks 本身包含了常用的一些网卡驱动，但也有一些并没有带。如常用的 Realtek81X9 系列的驱动。这里以 Realtek81X9 为例来讲述新驱动添加的一种简单方法。

首先要找到网卡的驱动程序。可以到产品商的主页下载或自己开发，在 Realtek 的主页上可下载到 81X9 的驱动程序包。

解压驱动程序包，有 3 个目录 config.h 和 src。将“config\pcPentium”目录下的文件 config.h、configNet.h、sysLib.c 拷贝到相应的 BSP 目录下覆盖老文件。将新文件 sysRtl81x9End.c 也拷到 BSP 目录下。这 4 个文件是相关的配置文件以及 Rtl81X9 的 END 驱动程序。在覆盖老文件前可以先做备份，然后通过比较新老文件的不同，对理解网卡驱动配置很有帮助。

将“src\drv\end\unsupported”目录下的 rtl81x9.c 和“h\drv\end\unsupported”目录下的 rtl81x9.h 也拷贝到相应的 BSP 目录下，这两个文件是 Rtl81X9 的驱动实现和头文件。

更改 rtl81x9.c 和 sysRtl81x9End.c 中引用头文件 rtl81x9.h 的路径。这样做是为了把头文件和源文件都放在 BSP 目录下，当然也可以按照原来系统设定的路径放置。

```

#include "drv/end/unsupported/rtl81x9.h" /*原来的*/
#include "rtl81x9.h" /*修改后*/

```

修改 Makefile，将 rtl81x9.c 文件编译进映象。

```

MACH_EXTRA = rtl81x9.o

```

现在可以在修改好的 BSP 上建立工程了，此时 Rtl81X9 的网卡驱动已经加载了。

8.3 Socket 程序设计

上层网络编程主要需要关注套接字[Socket]的使用，本节将介绍与此相关的内容。

8.3.1 概念的引入

网络编程有两种接口：Unix BSD 的套接字和 Unix System V 的 TLI。VxWorks 实现了与 BSD 4.4 TCP/IP 兼容的编程接口。在开始使用套接字编程之前，首先需要建立以下概念。

- 地址和端口

网络通信中通信的两个进程分别运行在不同的机器上。在互联网中，这两台机器可能位于不同的网络，通过网络互联设备连接。因此需要三级寻址。

- ✧ 某一主机与多个网络连接，必须指定一特定的网络地址。

- ✧ 网络上每一台主机应该具有其惟一的地址。

- ✧ 每一主机上的每一进程应有在该主机上的惟一表示符。

所以主机的地址由网络 ID 和主机 ID 组成，在 TCP/IP 协议中用 32 位整数值表示，而使用 16 位的端口号来标识通信进程。

端口号类似于文件描述符，用来标识不同的通信进程。由于 TCP/IP 传输层的两个协议 TCP 和 UDP 是完全独立的软件模块，因此各自的端口号相互独立，如 TCP 有一个 255 的端口，UDP 也可以有一个 255 的端口，二者并不冲突。

TCP/IP 的端口号分为两部分，小于 256 的端口作为保留端口，以全局的方式分配给服务进程。每一个标准的服务都拥有一个公认的端口，即使在不同的机器上，如 Telnet 为 23，FTP 为 21，HTTP 为 80，其余的端口为自由端口，以本地方式分配。即进程需要访问传输层服务时，向本地操作系统提出申请，操作系统返回一个本地惟一的端口号，进程再通过系统调用，将自己与该端口绑定。

● 网络字节顺序

不同的计算机存放多字节值的顺序不同。有的计算机的起始地址存放低字节，高字节在后（_LITTLE_ENDIAN）；有的则起始存放高字节（_BIG_ENDIAN）。为保证协议的正确性，在网络协议中需要指定网络字节顺序。TCP/IP 使用高字节先存。

● 连接

两个进程间的通信链路称为连接。连接在内部表现为一些缓冲区和一组协议机制，在外部表现为可靠的透明通道。

● 半相关

网络中用一个三元组可以在全局惟一标识一个进程：协议、本地端口和本地端口号。这样一个三元组，叫做一个半相关，它指定连接的每半部分。

● 全相关

一个完整的网间进程通信需要由两个进程组成，并且只能使用同一种高层协议。因此一个完整的网间通信需要一个五元组来标识：协议、本地地址、本地端口号、远地地址和远地端口号。这样一个五元组，叫做一个相关，即由两个协议相同的半相关才能组合成一个合适的全相关，或完全指定一个连接。

● 面向连接（虚电路）或无连接

面向连接是电话系统服务模式的抽象，即每一次完整的数据通信都要经过建立连接、使用连接、终止连接的过程。在数据传输的过程中，各数据分组不携带目的地址，而使用连接号。TCP 协议提供面向连接的虚电路。在应用中比如 FTP、Telnet 都使用面向连接的传输。

无连接是邮政系统服务的抽象，每个分组携带完整的目的地址，各分组在系统中独立传送。无连接不能保证分组的先后次序，不进行分组出错的恢复与重发，不保证传输的可靠性。UDP 协议提供无连接的服务，在应用中如 E-mail 使用无连接。

8.3.2 客户/服务器模式

在 TCP/IP 网络应用中，通信的两个任务间主要模式是客户机/服务器模式，即客户先向服务器提出服务请求，服务器接收到请求后，提供相应的服务。

面向连接的协议（如 TCP）服务器端首先调用 `socket` 函数建立流式套接字，然后用 `bind` 将此套接字和本地地址绑定；调用 `listen` 准备接收客户端的连接；然后调用 `accept` 接收连接，当接收到客户端的请求后，则连接建立，`accept` 返回新的套接字，就可以在这新套接字上读写数据；原来的套接字则可以继续通过 `accept` 调用等待另一个连接。

客户端也首先调用 `socket` 建立流式套接字，然后调用 `connect` 向远地主机发起连接请求，连接建立后就可以在此套接字上进行数据读写了。上述调用过程如图 8-2 所示。

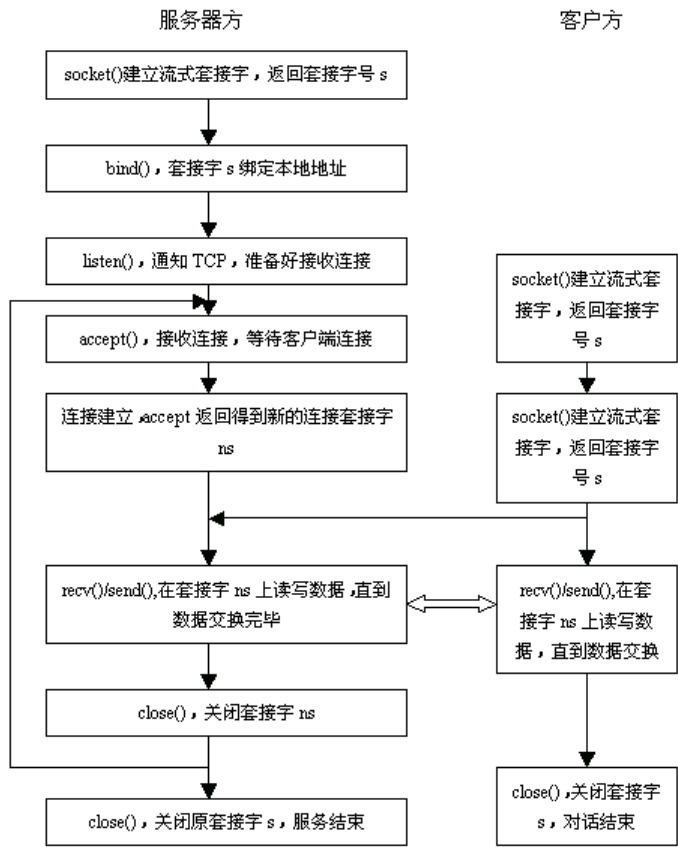


图 8-2 面向连接的协议调用

无连接服务器也首先需要调用 `socket` 建立套接字，然后用 `bind` 绑定本地地址。与面向连接的不同是它不需要侦听和建立连接，此时通过调用 `sendto` 和 `recvfrom` 就可以读写数据，客户端与此相同。无连接的调用过程如图 8-3 所示。

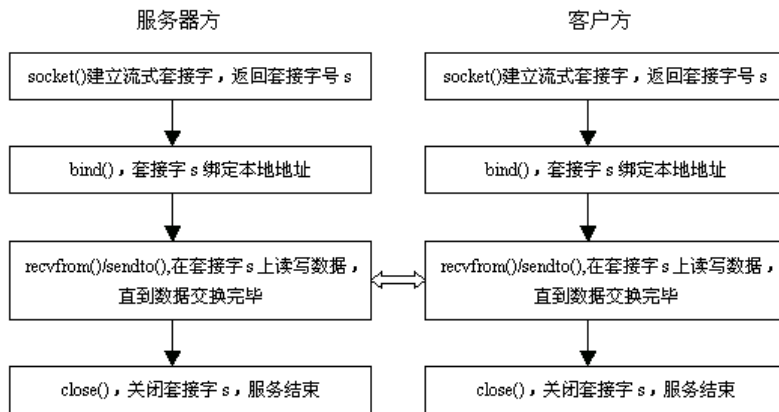


图 8-3 无连接的协议调用

下面是程序示例。

```

/***** 服务器程序 (server.c) *****/
int main(int argc, char *argv[])
{
    int sockfd, new_fd;
    struct sockaddr_in server_addr;
    struct sockaddr_in client_addr;
    int sin_size, portnumber;
    char hello[]="Hello! Are You Fine?\n";

    if(argc!=2)
    {
        fprintf(stderr, "Usage:%s portnumber\n", argv[0]);
        exit(1);
    }
    if((portnumber=atoi(argv[1]))<0)
    {
        fprintf(stderr, "Usage:%s portnumber\n", argv[0]);
        exit(1);
    }
    /* 服务器端开始建立 socket 描述符 */
    if((sockfd=socket(AF_INET, SOCK_STREAM, 0))==-1)
    {
        fprintf(stderr, "Socket error:%s\n", strerror(errno));
        exit(1);
    }
    /* 服务器端填充 sockaddr 结构 */
    bzero(&server_addr, sizeof(struct sockaddr_in));
    server_addr.sin_family=AF_INET;
    server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
    server_addr.sin_port=htons(portnumber);
    /* 捆绑 sockfd 描述符 */
    if(bind(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr))==-1)
    {
        fprintf(stderr, "Bind error:%s\n", strerror(errno));
        exit(1);
    }
    /* 监听 sockfd 描述符 */

```

```

if(listen(sockfd,5)==-1)
{
    fprintf(stderr, "Listen error:%s\n\a", strerror(errno));
    exit(1);
}
while(1)
{
    /* 服务器阻塞,直到客户程序建立连接 */
    sin_size=sizeof(struct sockaddr_in);
    if((new_fd=accept(sockfd, (struct sockaddr *)&client_addr,&sin_size))==-1)
    {
        fprintf(stderr, "Accept error:%s\n\a", strerror(errno));
        exit(1);
    }
    fprintf(stderr, "Server get connection from %s\n",
        inet_ntoa(client_addr.sin_addr));
    if(send(new_fd, hello, strlen(hello), 0)==-1)
    {
        fprintf(stderr, "Write Error:%s\n", strerror(errno));
        exit(1);
    }
    /* 这个通信已经结束 */
    close(new_fd);
    /* 循环下一个 */
}
close(sockfd);
exit(0);
}
/***** 客户端程序 client.c *****/
int main(int argc, char *argv[])
{
    int sockfd;
    char buffer[1024];
    struct sockaddr_in server_addr;
    struct hostent *host;
    int portnumber, nbytes;

    if(argc!=3)
    {
        fprintf(stderr, "Usage:%s hostname portnumber\n", argv[0]);
        exit(1);
    }
    if((host=gethostbyname(argv[1]))==NULL)
    {
        fprintf(stderr, "Gethostname error\n");
        exit(1);
    }
    if((portnumber=atoi(argv[2]))<0)
    {
        fprintf(stderr, "Usage:%s hostname portnumber\n", argv[0]);
        exit(1);
    }
}
/* 客户程序开始建立 sockfd 描述符 */

```

```

if((sockfd=socket(AF_INET, SOCK_STREAM, 0))==-1)
{
    fprintf(stderr, "Socket Error:%s\n", strerror(errno));
    exit(1);
}
/* 客户程序填充服务端的资料 */
bzero(&server_addr, sizeof(server_addr));
server_addr.sin_family=AF_INET;
server_addr.sin_port=htons(portnumber);
server_addr.sin_addr=((struct in_addr *)host->h_addr);
/* 客户程序发起连接请求 */
if(connect(sockfd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr))==-1)
{
    fprintf(stderr, "Connect Error:%s\n", strerror(errno));
    exit(1);
}
/* 连接成功了 */
if((nbytes=recv(sockfd, buffer, 1024, 0))==-1)
{
    fprintf(stderr, "Read Error:%s\n", strerror(errno));
    exit(1);
}
buffer[nbytes]='\0';
printf("I have received:%s\n", buffer);
/* 结束通信 */
close(sockfd);
exit(0);
}

```

8.3.3 Socket 函数

上节介绍了一些客户/服务器模式的基本函数调用，本节来详细介绍一下相关的函数。

- **bzero**

```
void bzero((char *) dest, size_t nbytes)
```

bzero 是源自 BSD 函数，VxWorks 也支持，其功能等同于 memset，但比 memset 少了一个参数，好记忆，并且易于程序移植，推荐使用。举例如下：

```
bzero((char *)&myaddr, sizeof(myaddr));
```

- **htonl**

```
unsigned long htonl(unsigned long hostlong)
```

此函数将一个 32 位的无符号整数从主机字节顺序转换成网络字节顺序。函数返回网络字节顺序的 32 位无符号整数。举例如下：

```
myaddr.sin_addr.s_addr=htonl(INADDR_ANY);
```

- **htons**

```
unsigned short htons(unsigned short hostshort)
```

此函数将一个 16 位的无符号整数从主机字节顺序转换成网络字节顺序。函数返回网络字节顺序的 16 位无符号整数。举例如下：

```
myaddr.sin_port = htons(pnet->pComm->wNetPortNo);
```

- **inet_aton 和 inet_addr**

```
STATUS inet_aton
(
```

```
char *      pString, /* string containing address, dot notation */
struct in_addr * inetAddress /* struct in which to store address */
)
u_long inet_addr
(
char * inetString /* string inet address */
)
```

这两个函数均是将所指的 C 字符串转换为 32 位的网络字节顺序的二进制值。inet_pton 将转换的值存在 inetAddress 所指的结构中，成功返回 OK，失败返回 ERROR。而 inet_addr 是直接返回转换的结果，或 ERROR。这样就存在一个问题，当点分字符串为“255.255.255.255”时，其返回值会解释为-1（ERROR），使得它不能被此函数处理，所以推荐使用 inet_pton。

● **inet_ntoa**

```
char *inet_ntoa
(
struct in_addr inetAddress /* inet address */
)
```

此函数将 32 位的网络字节顺序的二进制值转换为相应的点分的十进制字符串。因为函数的返回值所指的串驻留在静态内存中，所以此函数不可重入，使用时需要注意保护。另外需要注意的是它以结构为参数，而不是指向结构的指针。

● **socket**

```
int socket
(
int domain, /* address family (for example, AF_INET) */
int type, /* SOCK_STREAM, SOCK_DGRAM, or SOCK_RAW */
int protocol /* socket protocol (usually 0) */
)
```

返回：socket 描述符，或 ERROR

此函数建立一个套接字，它给指定的地址族、数据类型和协议分配一个套接字描述符和相关的资源。函数返回 Socket 描述符或 ERROR。其中 domain 为地址簇，VxWorks 只支持 AF_INET。type 为套接字的类型（如表 8-1 所示）。Protocol 为指定协议，通常设为 0。

表 8-1 套接字的类型

SOCK_STREAM	流式套接字	提供顺序的、可靠的、双向的、基于连接的字节流，并有带外数据传输机制，使用 TCP 作为网际地址族
SOCK_DGRAM	数据报式套接字	支持无连接的、不可靠的数据报，包的最大长度是固定的，使用 UDP 作为网际地址族
SOCK_RAW	原始式套接字	允许对较底层协议直接访问

在 VxWorks 中返回的 Socket 描述符是一个标准的 I/O 系统文件描述符，因此可以被 close、read、write 和 ioctl 等函数操作，也可以传给其他应用任务来操作此 Socket。

Socket 有阻塞和非阻塞的属性。初始建立的 Socket 默认为阻塞的，这意味着当一个套接口调用不能立即完成时，任务进入睡眠状态，等待操作完成。可以通过 ioctl 来设置 Socket 的非阻塞属性。Socket 使用示例如下。

```
/*建立一个 TCP 的 socket*/
if ((nSocket=socket(AF_INET, SOCK_STREAM, 0)) == ERROR) {
    return(ERROR);
}
```

```
/*建立一个 UDP 的 socket*/
if ((nSocket=socket(AF_INET, SOCK_DGRAM, 0)) == ERROR) {
    return(ERROR);
}
```

● bind

```
STATUS bind
(
    int          s,          /* socket descriptor */
    struct sockaddr * name,  /* name to be bound */
    int          namelen /* length of name */
)
```


此函数将本地地址与 **socket** 函数创建的未命名的套接字绑定,建立起套接字的本地连接。在 **AF_INET** 地址域中, **name** 的格式只能是 **sockaddr_in** 结构。

```
struct in_addr{
    u_long    s_addr;
}
struct sockaddr_in{
    u_char    sin_len;
    u_char    sin_family;
    u_short   sin_port;
    struct in_addr sin_addr;
    char      sin_zero[8];
}
```

其中 **sin_family** 置为 **AF_INET**。**sin_port** 为连接的端口号。**sin_addr** 为主机地址。如果应用程序不关心地址,即可以使用主机的任一有效地址,则可以用常数 **INADDR_ANY** 来设定。**sin_zero** 为保留字段,需全部置为 0,所以通常用 **bzero** 将整个结构首先置为 0。

函数 **bind** 的使用示例如下:

```
bzero((char *)&myaddr, sizeof(myaddr));
myaddr.sin_family = AF_INET;
myaddr.sin_addr.s_addr=htonl(INADDR_ANY);
myaddr.sin_port = htons(pnet->pComm->wNetPortNo);
if (bind(nSocket, (struct sockaddr *)&myaddr, sizeof(myaddr)) == ERROR)
    return(ERROR);
```

 **sin_port** 和 **sin_addr** 均要以网络字节顺序表示。

● listen

```
STATUS listen
(
    int s,          /* socket descriptor */
    int backlog /* number of connections to queue */
)
```

listen 用于服务器端套接字的侦听连接。当 **socket** 创建一个套接口时,该套接口被假设为一个主动套接口,也就是说,它是一个将调用 **connect** 发起连接请求的套接口;通过调用 **listen**,此套接口即被指定为被动模式,服务器端即可通过此套接口接受客户请求。所以它一般用在函数 **socket** 和 **bind** 之后, **accept** 之前。

第二个参数 **backlog** 指定了在此套接口上排队等待处理的连接请求的最大个数。**backlog** 不允许为 0,如果不想让任何客户连接到此监听套接口,则最好关掉此套接口。**BSD** 规定的 **backlog** 最大值为 5。

函数 `listen` 的使用示例如下：

```
if (listen(nSocket, 5) == ERROR) {  
    return(ERROR);  
}
```

● `select`

```
int select  
(  
    int          width,          /* number of bits to examine from 0 */  
    fd_set *     pReadFds,       /* read fds */  
    fd_set *     pWriteFds,      /* write fds */  
    fd_set *     pExceptFds,     /* exception fds (unsupported) */  
    struct timeval * pTimeout     /* max time to wait, NULL = forever */  
)
```

`select` 用于指示一个或多个套接字的状态。函数返回所有描述字集合已准备好的总位数。如果超时，则返回 0。返回 `ERROR` 表示发生错误。`select` 不只限于网络编程，还可以用于其他 I/O 操作。有时使用 `select` 函数可以帮助用户方便地解决问题。

- ✧ 进行读写操作的时候，会由于条件不满足而阻塞。比如进行读调用，可能此时缓冲区里面没有数据可读(通信对方还没有发送数据过来)，这个时候读调用就会阻塞直到有数据可读为止。可是如果并不希望阻塞，而是希望得到关于可读写的指示使得操作顺利进行，那么这个时候就可以采用 `select` 函数，由它来通知连接的一个或多个 I/O 条件是否得到满足；
- ✧ 当一个任务处理多个连接时，为了得到某一时刻哪些连接可以进行读写，也需使用 `select`。
- ✧ 服务器端如果同时处理多个监听套接口，可以通过 `select` 得知哪些连接请求被接受。

函数 `select` 中间 3 个参数的数据类型是 `fd_set`，它的意思是描述字符集合，而 `pReadFds`、`pWriteFds` 和 `pExceptFds` 则分别是指向描述字集合的指针，它们分别描述了开发者所关心的可读、可写以及状态异常的各个描述字。在 `VxWorks` 中 `pExceptFds` 不被支持。如果用户对某项集合并不感兴趣，那么可以将指针设为 `NULL`。在调用 `select` 函数实现多路 I/O 复用时，首先要声明一个新的描述字集合。`fd_set` 结构在实现上一般是一个整数数组，每个数中的每一位对应了一个描述字。`VxWorks` 提供了几个宏定义（如表 8-2 所示），用以屏蔽具体的实现细节，完成设置、测试集合中每一位的功能。

表 8-2 `fd_set` 相关的宏定义

FD_ZERO	清空此描述字集合的所有位，以免下面检测描述字位的时候返回错误结果
FD_SET	在描述字集合中设置用户关心的位
FD_ISSET	测试描述字 <code>fd</code> 是否属于集合。当 <code>select</code> 成功后，描述字集合中任何没准备好的描述字相对应的位返回时都清为 0。通过调用 <code>FD_ISSET</code> ，判断 <code>fd</code> 是否仍然属于描述字集合即可得知此描述字是否准备好
FD_CLR	将描述字 <code>fd</code> 从集合中清除

`select` 函数的第一个参数 `width` 表示需要测试的宽度，即需要遍历多少个描述字位。`width` 的值一般是这样设置的，从用户关心的所有描述字中选出最大值再加 1。例如设置的所有描

述字中最大的为 6，那么将 width 设置为 7，则系统在检测描述字状态的时候，就只用遍历前 7 位 (fd0~fd6) 的状态。不过如果不想这样麻烦的话，可以直接设置为 `FD_SETSIZE`。这是系统中设定的最大描述符字数，在 VxWorks 中此值为 **256**，定义在 `vxTypesold.h` 中。这样在检测描述字状态的时候，函数将遍历所有的描述符位，但也增加了系统开销。

`select` 函数的参数 `pTimeOut` 用于设置函数调用的超时值，时间格式如下。

```
struct timeval{
    long tv_sec;    /* seconds */
    long tv_usec;   /* microseconds */
};
```

共有如下 3 种设置。

- ✧ `select` 永远等待：此时 `pTimeOut` 设为 `NULL`。
- ✧ 等待固定时间：此时在 `pTimeOut` 所指的结构中设置等待时间值。
- ✧ 不等待，立即返回：此时在 `pTimeOut` 所指的结构中 `tv_sec` 与 `tv_usec` 均设置为 0。

使用 `select` 函数的示例代码如下：

```
socket[0]=socket1;
socket[1]=socket2;

FD_ZERO(&lfd);
FD_SET(socket[0], &lfd);
FD_SET(socket[1], &lfd);

if (socket[0]> socket[1])
    maxfd= socket[0]+1;
else
    maxfd= socket[1]+1;
for (;;) {
    memcpy(&readfds, &lfd, sizeof(fd_set));
    if (select(smax, &readfds, (struct fd_set *)0,
        (struct fd_set *)0, (struct timeval *)0)<=0)
        /*if (select(FD_SETSIZE, &readfds, (struct fd_set *)0,
            (struct fd_set *)0, (struct timeval *)0)<=0)*/
            continue;
    for (i=0; i<2; i++) {
        if (FD_ISSET(socket[i], &readfds))    /*fd readable*/
            /*do something*/;
    }
}
```

● `accept`

```
int accept
(
    int          s,          /* socket descriptor */
    struct sockaddr * addr,   /* peer address */
    int *        addrlen /* peer address length */
)
```


函数 `accept` 从已完成的连接队列头中获得一个已完成的连接。这个连接队列是调用 `listen` 建立的。参数 `s` 称为监听套接口描述字，它是由 `socket` 调用生成，用作 `bind` 和 `listen` 的第一个参数。`accept` 的返回值称为已连接套接口描述字，它与参数 `s` 具有相同的属性，但它不

能再接受其他连接请求。而原先的监听套接口描述字 `s` 仍然保持打开状态来继续侦听客户端的连接，直到该服务器关闭。

当 `Socket` 设为阻塞方式且队列中没有新的连接，那么 `accept` 调用将被阻塞。为了侦听不同端口的客户端连接，服务器端可能使用 `select` 调用，当一个已完成的连接准备好时，`select` 会设置为可读。此时接着调用 `accept`，因为 `select` 告诉用户连接已就绪，`accept` 应该不会阻塞。不幸的是，这种假设并不总成立，这里存在一个时间问题。如果一个繁忙的服务器不能做到在 `select` 一返回监听套接口可读时就调用 `accept`，而客户端在服务器端调用 `accept` 之前放弃连接，则当服务器端调用 `accept` 时，因为没有已完成的连接，就会被阻塞在此描述字上，再不能处理其他任何已就绪的描述字。解决这个问题的办法是用 `select` 探测套接字是否可读，可以在 `accept` 之前，把监听套接字置为非阻塞的。当非阻塞 `accept` 调用出现错误时，`EWOULDBLOCK` 错误可以被忽略。

使用 `accept` 函数的示例代码如下：

```
for(;;){
    if (select(smax,&readfds,(struct fd_set *)0,
        (struct fd_set *)0,(struct timeval *)0)<=0)
        continue;          /*select error*/
    for (fd=0; fd<smax; fd++){ /*have readable fd*/
        if (FD_ISSET(fd, &readfds)){
            optval=1;        /*设为非阻塞模式*/
            if (ioctl(sindex, FIONBIO, (int)&optval)<0)
                continue;
            peeraddrlen=sizeof(peeraddr);
            nSocket=accept(sindex, (struct sockaddr *)&peeraddr, &peeraddrlen);
            if (nSocket<=0){ /*Handle non-blocking*/
                if (errno==EWOULDBLOCK)
                    continue
                else
                    /*An error has occurred, see errno for details*/
            }
            else
                /*accept success*/
        }
    }
}
```

 `accept` 继承了原套接字 `setsockopt` 所设置的属性，但并没有继承 `ioctl` 的设置，所以要使新的套接字为非阻塞的，需要重新调用 `ioctl` 函数来设置。

● connect

```
STATUS connect
(
    int          s,          /* socket descriptor */
    struct sockaddr * name,   /* addr of socket to connect */
    int          namelen /* length of name, in bytes */
)
```

此函数用来与对方建立一个连接。`name` 指向一个含有对方套接字地址的结构。

对于阻塞的 `Socket`，`connect` 函数阻塞调用者，直到建立起连接（返回 `OK`）或有错误发生（返回 `ERROR`）；对于非阻塞的 `Socket`，连接不一定马上完成；此时可以通过 `select` 来判

断连接是否完成。BSD 的实现有两条与 `select` 和非阻塞 `Socket` 相关的规则：当连接成功建立时，套接字变为可写；当连接建立错误时，套接字变成既可写又可读。根据上述规则，下面为典型的非阻塞 `Socket` 的 `connect` 编程。

✧ `connect` 函数返回 `OK` 表示连接建立成功，`ERROR` 表示连接建立出错，可以检查 `errno` 确定原因。如果 `errno` 为 `EINPROGRES` 或者 `EWOULDBLOCK`，表示连接正在处理，此时需要利用 `select` 函数，设置超时时间，通过检查套接字是否可读或可写来判断 `connect` 是否完成；否则连接失败。

```
if (connect (fd, ....) == -1)
{
    if (errno == EINPROGRESS)
        // Handle non-blocking connects (see below).
    else
        // An error has occurred, see errno for details.
}
else
    // Connected immediately.
```

✧ 如果 `select` 超时（返回 0）或者发生错误（返回 `ERROR`），则连接失败。

✧ 如果 `socket` 可读或可写，则有 3 种方法可以判断连接是否成功。第 1 种，调用函数 `getsockopt`，如果函数执行失败或者 `errno` 非 0，则连接失败。第 2 种，调用 `getpeername`，如果函数成功则连接成功，否则失败，可以调用 `read` 来得到 `errno`。第 3 种，再次调用函数 `connect`，如果成功或者函数失败且返回为 `EISCONN`（已连接），则表示前面的 `connect` 连接成功。推荐使用前两种，最后一种对于某些系统可能不适用。前面两种方法的示例代码如下：

```
// Check to see if connection has been established: use getsockopt
{
    fd_set rd_fds;
    fd_set wr_fds;
    struct timeval tv = ...

    FD_ZERO (&rd_fds);
    FD_ZERO (&wr_fds);
    FD_SET (fd, &wr_fds);
    FD_SET (fd, &rd_fds);

    if (select (fd + 1, &rd_fds, &wr_fds, 0, &tv) <= 0)
        // problems, bail out...
    else if (FD_ISSET (fd, &wr_fds) || FD_ISSET (fd, &rd_fds))
    {
        int flag=0, flaglen=sizeof(flag);
        if (getsockopt (nSocket, SOL_SOCKET, SO_ERROR, (char*)&flag, &flaglen)==0)
        {
            if (flag==0)
            {
                // We've successfully connected on fd.
            }
        }
    }
    else
    {
        // Error, fd is not connected, errno contains reason.
```

```

        // ...
        close (fd);
    return -1;
    }
}

else
{
    // Error, fd is not connected, errno contains reason.
    // ...
    close (fd);
    return -1;
}
}
}

```

```

// Check to see if connection has been established: use getpeername
{
    fd_set rd_fds;
    fd_set wr_fds;
    struct timeval tv = ...

    FD_ZERO (&rd_fds);
    FD_ZERO (&wr_fds);
    FD_SET (fd, &wr_fds);
    FD_SET (fd, &rd_fds);

    if (select (fd + 1, &rd_fds, &wr_fds, 0, &tv) <= 0)
        // problems, bail out...
    else if (FD_ISSET (fd, &wr_fds) || FD_ISSET (fd, &rd_fds))
    {
        sockaddr_in addr;
        int len = sizeof addr;

        // Check to see if we can determine our peer's address.
        if (getpeername (fd, (sockaddr *) &addr, &len) == -1)
        {
            // Error, fd is not connected, errno contains reason.
            // ...
            close (fd);
            return -1;
        }
        else // We've successfully connected on fd.
        {
            // ...
        }
    }
}
}

```



Google tcp-ip 讨论组, <http://groups.google.com/groups?q=comp.protocols.tcp-ip>。

- **connectWithTimeout**

STATUS connectWithTimeout

```

(
    int                sock,    /* socket descriptor */

```

```

struct sockaddr * adrs,      /* addr of the socket to connect */
int             adrsLen,    /* length of the socket, in bytes*/
struct timeval * timeVal    /* time-out value */
)

```

前面讲述了为了非阻塞 Socket 的 connect 而所做的工作，但是在 VxWorks 中，上述工作就变的简单了，因为 VxWorks 在 socklib 中提供了一个函数 connectWithTimeout，上述的功能已经被它完全实现。connectWithTimeout 与 connect 相比增加了第 4 个参数 timeVal，通过它设置连接的超时值。使用 connectWithTimeout 函数的示例代码如下：

```

struct timeval tv = ...
if (connectWithTimeout (fd, ..., & tv) ==ERROR) {
    /*An error has occurred, see errno for details.*/
}
else
    /*connected immediately.*/

```

● recv 和 recvfrom

```

int recv
(
    int     s,           /* socket to receive data from */
    char * buf,          /* buffer to write data to */
    int     bufLen,      /* length of buffer */
    int     flags        /* flags to underlying protocols */
)
int recvfrom
(
    int     s,           /* socket to receive from */
    char * buf,          /* pointer to data buffer */
    int     bufLen,      /* length of buffer */
    int     flags,       /* flags to underlying protocols */
    struct sockaddr * from, /* where to copy sender's addr */
    int * pFromLen       /* value/result length of from */
)

```

recvfrom 用来从一个套接字 s 上读取数据，不管这个套接字是否连接。recv 一般用来从已连接的流套接字 s 上读取数据，它等同于 recvfrom 中 from 为 NULL 的情况。如果 recvfrom 的参数 from 非空，且套接字为 SOCK_DGRAM 类型，则对方的地址在返回时被拷入到 from 指向的地址。pFromLen 调用前设置为 from 所指缓冲区的大小，调用后为返回的实际大小。

如果 recv 和 recvfrom 没有错误发生，返回成功接收的字节数；如果连接被关闭，返回 0；否则返回 ERROR，通过 errno 可得知错误代码。当套接字上没有数据时，如果是阻塞模式，则调用挂起一直到数据到来；否则返回 ERROR，错误码为 EWOULDBLOCK。如果接收了 0 字节的数据，则表示连接已经被对方“雅致”地关闭。

recv 和 recvfrom 的前 4 个参数相同。buf 为接收输入数据的缓冲区。buflen 为缓冲区长度。flag 为 0 或下列值的一个或多个逻辑或：MSG_OOB（读取带外数据）和 MSG_PEEK（查看输入数据，数据被拷入缓冲区中，但不从输入队列清除）。

使用 recv 和 recvfrom 的示例代码如下：

```

nRecNum=recv (nSocket, (char *)tempBuf, NET_BUF_NUM, 0);
if (nRecNum<0) {
    if (errno==EWOULDBLOCK)
        /*非阻塞模式返回*/

```

```

        else
            /*错误发生*/
    }
    else if (nRecNum==0)
        /*连接被关闭*/
    else
        /*成功接收数据*/
    nRecNum=recvfrom(nSocket, (char *)tempBuf, NET_BUF_NUM, 0,
                    (struct sockaddr *)peerAddr, &peerAddrSize);

```

● send 和 sendto

```

int send
(
    int          s,          /* socket to send to */
    const char * buf,        /* pointer to buffer to transmit */
    int          buflen,     /* length of buffer */
    int          flags       /* flags to underlying protocols */
)
int sendto
(
    int          s,          /* socket to send data to */
    caddr_t      buf,        /* pointer to data buffer */
    int          buflen,     /* length of buffer */
    int          flags,      /* flags to underlying protocols */
    struct sockaddr * to,    /* recipient's address */
    int          tolen       /* length of to sockaddr */
)

```

sendto 一般用在 SOCK_DGRAM 格式的套接字上，向指定的目的地址发送数据，参数 to 指定对方的地址。为了发送广播报文，需要使用特殊的广播地址 INADDR_BROADCAST。send 用于已连接的套接字 s 上发送数据。

send 和 sendto 的前 4 个参数相同。buf 为存有发送数据的缓冲区指针。buflen 为缓冲区长度。flag 为 0 或下列值的一个或多个逻辑或：MSG_OOB（发送带外数据）和 MSG_DONTROUTE（发送数据不提交给路由选择）。

使用 send 和 sendto 的示例代码如下：

```

nSendNum=send(nSocket, (char *)tempBuf, countw, 0,);
if (nSendNum<0) {
    /*发生错误*/
}
else
    /*成功发送 nSendNum 个字节数据*/
nSendNum=sendto(nSocket, (char *)tempBuf, countw, 0, (struct sockaddr *)peerAddr, addLen);

```



send 和 sendto 函数只是将数据传送到输出缓冲区，它执行成功并不意味着数据成功的发送出去了。当系统中没有可用的缓冲区来发送数据时，函数调用将被阻塞，除非套接字是非阻塞模式的。

● shutdown


```

STATUS shutdown
(
    int s, /* socket to shut down */
    int how /* 0 = receives disallowed 1 = sends disallowed 2 = sends and receivers disallowed */
)

```

)

此函数用来切断一个 Socket 连接的接收、发送或全部。how 为 0，则套接字后续接收被禁止；为 1，后续发送被禁止；为 2，发送和接收同时被禁止。

 shutdown 只是切断套接字的发送或接收，并不关闭套接字，与套接字相关的资源并没有被释放。如果要关闭套接字，需要调用 close 函数。

● setsockopt 和 getsockopt

函数声明：

```
STATUS setsockopt
(
    int     s,          /* target socket */
    int     level,      /* protocol level of option */
    int     optname,    /* option name */
    char * optval,      /* pointer to option value */
    int     optlen      /* option length */
)
STATUS getsockopt
(
    int     s,          /* socket */
    int     level,      /* protocol level for options */
    int     optname,    /* name of option */
    char * optval,      /* where to put option */
    int * optlen        /* where to put option length */
)
```

setsockopt 用来设置套接字相关选项的当前值，getsockopt 用来得到套接字选项的当前值。

setsockopt 和 getsockopt 的参数类似。level 为选项定义的层次，只支持 SOL_SOCKET 和 IPPROTO_TCP。optname 指定套接字选项的名字。optval 为指向选项数据缓冲区的指针。optlen 在初始时为缓冲区 optval 的长度，在 getsockopt 调用后被置为实际值的长度。

套接字选项有两种类型：设置或禁止特征/行为的 BOOL 选项和要求整数值或结构的选项。为了设置一个 BOOL 选项，optval 应指向一个非零整数，如果禁止此选项，则指向一个等于零的整数。对于 BOOL 选项来说，optlen 应该等于整数型的长度。对于其他选项，optval 指向了一个包含了选项值的整数或结构，并且 optlen 为此整数或结构的长度。

一些常用的选项如表 8-3、8-4 和 8-5 所示。

表 8-3 SOL_SOCKET 对应的选项

选项名称	说明	数据类型
SO_BROADCAST	允许发送广播数据	int
SO_DEBUG	允许调试	int
SO_DONTROUTE	不查找路由	int
SO_ERROR	获得套接字错误	int
SO_KEEPALIVE	保持连接	int
SO_LINGER	延迟关闭连接	struct linger
SO_OOBINLINE	带外数据放入正常数据流	int
SO_RCVBUF	接收缓冲区大小	int
SO_SNDBUF	发送缓冲区大小	int

SO_RCVLOWAT	接收缓冲区下限	int
SO_SNDBLOWAT	发送缓冲区下限	int
SO_RCVTIMEO	接收超时	struct timeval
SO_SNDTIMEO	发送超时	struct timeval
SO_REUSEADDR	允许重用本地地址和端口	int
SO_TYPE	获得套接字类型	int
SO_BSDCOMPAT	与 BSD 系统兼容	int

表 8-4 IPPROTO_IP 对应的选项

选项名称	说明	数据类型
IP_HDRINCL	在数据包中包含 IP 首部	int
IP_OPTIONS	IP 首部选项	int
IP_TOS	服务类型	int
IP_TTL	生存时间	int

表 8-5 IPPROTO_TCP 对应的选项

选项名称	说明	数据类型
TCP_MAXSEG	TCP 最大数据段的大小	int
TCP_NODELAY	不使用 Nagle 算法	int

下面只介绍几个常用的选项，其他选项详细用法读者可以查阅相关书籍。

(1) 适用流格式 socket 的选项如下。

✧ SO_KEEPLIVE

应用程序通过打开 SO_KEEPLIVE 套接字选项开关来请求在 TCP 上使用保持允许包 (keep_alive)。当设置此选项后，如果在 2h 内在此套接口的任一方向都没有数据交换，TCP 就自动给对方发一个保持存活探测分节 (keepalive probe)。这是一个对方必须响应的分节，会有下面 3 种情况。

对方以期望的 ACK 响应。应用程序不会感觉到（因为一直正常）。又过 2h 后如果仍无数据交换，则 TCP 将再次发送另一个探测分节。

对方以 RST 响应，告知本地 TCP 对方已经崩溃并已重新启动。套接口的待处理错误码被置为 ECONNRESET，套接口本身被关闭。

对方对保持存活探测分节无任何反应。TCP 将发送另外 8 个探测分节，相隔 75s 一个，试图得到一个响应。这样 TCP 再发出第一探测分节的 11m15s 后仍无响应就放弃。此时错误码被置为 ETIMEOUT，套接口本身则被关闭。但如果套接口收到一个 ICMP 错误作为某个探测分节的响应，则返回响应的错误，套接口也被关闭。此情形中最常见的错误是 “host unreachable（主机不可达）”，说明对方主机并没关闭，但是不可达，此时错误码为 EHOSTUNREACH。

SO_KEEPLIVE 选项一般由服务器使用，尽管客户也可使用此选项。服务器用此选项来关闭那些已经消失的连接，这是用户未关闭连接就离开所造成的后果，比如当客户主机崩溃，

服务器将永远不会知道。此选项只对流套接字有意义。

✧ TCP_NODELAY

此选项取消 Nagle 算法。Nagle 算法用来减少主机的小包发送数量，它通过缓冲未确认的发送数据直到能够发送一个全长度的包来做到这一点。然后，对于某些应用程序来说，此算法可能不适用，就可以用 TCP_NODELAY 来关闭此算法。应用程序只有在确实需要的情况下，才设置 TCP_NODELAY 选项，因为设置后对网络性能有明显的负面影响。TCP_NODELAY 是惟一使用 IPPROTO_TCP 层的选项。

(2) 流和数据报格式均适用的选项如下。

✧ SO_REUSEADDR

套接字一般默认不能绑定 (bind) 到一个已经使用的地址。然而，有时会要求“重用”地址。既然每一个连接是通过本地地址和远程地址来惟一识别的，那么在远程地址不同时，两个套接字绑定到同一地址应该是可行的。为了通知 TCP 实现不必因为 bind 调用所要求的地址已被其他套接字使用而禁止，应用程序可以在 bind 调用之前设置 SO_REUSEADDR 选项。此选项仅在 bind 调用时解释。

✧ SO_ERROR

当套接口上发生错误时，可以通过此选项来查看错误号并将错误号复位为 0。此选项只能用于获取，不能用来设置。举例如下：

```
int optval=0;
int optvallen=sizeof(optval);
if (getsockopt(nSocket, SOL_SOCKET, SO_ERROR, (char*)&optval, &optvallen)==ERROR) {
    return(ERROR);
}
if (optval) return(ERROR);
```

● ioctl

```
int ioctl
(
    int fd,          /* file descriptor */
    int function,    /* function code */
    int arg           /* arbitrary argument */
)
```

ioctl可以控制所有的文件描述符的情况，当然也包括套接字描述符，这里介绍几个控制套接字的选项，如表8-6所示。

表 8-6 ioctl 的控制选项

选项名称	说明	数据类型
SIOCATMARK	是否到达带外标记	int
FIONBIO	设置/清除非阻塞标志	int
FIONREAD	缓冲区可读的字节数	int

关于ioctl的使用，举例如下：

```
/* 查看缓冲区可读数据*/
status = ioctl (sFd, FIONREAD, &bytesAvailable);
/*查看是否有带外数据*/
status = ioctl (sFd, SIOCATMARK, &atMark);
/*设置非阻塞套接字*/
```

```
on = 1;
status = ioctl (sFd, FIONBIO, &on);
```

8.3.4 服务器程序结构

客户端向服务器发起服务请求，其编程结构相对简单；服务器端由于要及时响应处理多个客户端的请求，所以其编程要相对复杂些。从结构来说，服务器端基本上使用两种模式：循环模式和并发模式。

循环模式就是服务器端 `de` 程序在整体上是一个循环，一次处理一个客户请求。这样当有多个客户请求时，就放入队列，依次等待处理。但是，显然前面的处理时间过长会影响后续请求而使它得不到响应，算法如下。

```
socket(...);
bind(...);
listen(...);
while(1)
{
    accept(...);
    while(1)
    {
        send(...);
        process(...);
        recv(...);
    }
    close(...);
}
```

并发模式又有两种情况：一种是单任务模式，一种是多任务模式。

多任务模式结构上一般是父任务在接收请求后生成一个新的子任务，并将新产生的数据套接口传给它，子任务负责在这个套接口数据上通信，比如 `FTP Server` 任务，当然接收客户端请求后，会产生许多子任务处理请求，这种模式架构如下。

```
socket(...);
bind(...);
listen(...);
while(1)
{
    accept(...);
    if(taskSpawn(...) != ERROR)
    {
        while(1)
        {
            send(...);
            process(...);
            recv(...);
        }
        close(...);
        exit(...);
    }
    close(...);
}
```

多任务的并发模式解决了循环服务器客户机独占服务器的情况。不过也同时带来了一个不小的问题，为了响应客户机的请求，服务器要创建子任务来处理。而创建子任务是一种非常消

耗资源的操作。

为了解决上述创建子任务带来的系统资源消耗,人们又想出了单任务下多路复用 I/O 模式,即通过 `select` 调用完成。首先将需要监听的套接字加入 `select` 的可读队列中监听,当此套接口变为可读后调用 `accept` 产生新的数据套接字,然后将此数据套接字也加入 `select` 可读监听队列中,继续监听。这样当新套接字上变为可读,则表示有新的数据待处理,当最初的套接字变为可读,则表示有新的请求,模型如下。

```
初始化(socket, bind, listen);
while(1)
{
    设置监听读写文件描述符(FD_*);

    调用 select ();

    如果是倾听套接字就绪,说明一个新的连接请求建立
    {
        建立连接(accept);
        加入到监听文件描述符中去;
    }
    否则说明是一个已经连接过的描述符
    {
        进行操作(send 或者 recv);
    }
}
```

8.4 网络服务

VxWorks 提供多种标准网络服务,本节介绍最常用的两种:FTP 服务器和 Telnet 服务器。

8.4.1 FTP 服务器

通过目标机提供的 FTP 服务器,用户可以管理目标机文件系统,能方便地升级程序映像。

● 加载 FTP 服务器

在工程中选择“Ftp server”组件即可加载 FTP 服务。此选项定义 `INCLUDE_FTP_SERVER` 使得系统启动时调用 `ftpdInit` 函数来启动 FTP 任务。

加载 FTP 服务器后,可以通过 `ioDefPathSet` 函数来设定缺省的 IO 路径,例如目标机的 TFFS 设备名为“tffs0”,设定 `ioDefPathSet (“/tffs0/”)`。这样当登录到 FTP 服务器后就可以查看目标机“tffs0”上的内容了。

● 权限设定

如果希望进行权限设定,则需要选择“Ftp server securiy”组件和“rlogin/telnet password protection”组件,然后在组件中设定用户登录名和密码即可。也可以在程序中通过 `loginUserAdd` 函数来增加新的用户。

注意 `loginUserAdd` 使用的密码不是明码,而是经过加密的密码。用户可以调用 `loginDefaultEncrypt` 来得到换算后的密码,例如要添加用户 `guest`,密码为 123456789。

```
char pw[256];
loginDefaultEncrypt("123456789",pw); /*得到 pw = “SRSQQeQccc”*/
```

```
loginUserAdd("guest", pw);
```

登录时使用如下。

```
user:guest
pw:123456789
```

在“host\hostOS\bin”目录下还提供了一个工具 vxencrypt，用来计算加密后的密码。

```
vxencrypt
please enter password: flintstone
encrypted password is ScebRez9c
```

用户也可以调用如下函数来安装自己的密码生成程序。

```
void loginEncryptInstall
(
    FUNCPTR rtn, /* function pointer to encryption routine */
    int      var /* argument to the encryption routine (unused) */
)
```

其中用户自定义密码生成程序（由参数 rtn 指定）必须符合如下格式。

```
STATUS encryptRoutine
(
    char *password,          /* string to encrypt */
    char *encryptedPassword /* resulting encryption */
)
```

● 使用一个更好的 FTP 服务器

Tornado 2.0.2 中的 FTP 服务器只提供了很少的功能，通过安装 SPR79795 DOSFS2 包，在“target/unsupported/src/netwrs”目录下提供了一个 ftpdLib.c 源文件，用户可以加载该文件来生成一个功能更强大的 FTP 服务器。并且通过修改源码可以更方便地定制自己的 FTP 服务。

首先需要从原来的库文件中卸载老的 ftpdLib.o，注意模块名称的大小写，以 ColdFire 5272 为例。


```
arcf -d <tornado>/target/lib/libMCF5200gnuvox.a ftpdLib.o
```

在成功移除 ftpdLib.o 后，通过修改 Makefile，将 ftpdLib.c 引入到工程中。

```
MACH_EXTRA = ftpdLib.o
```

也可以不修改 Makefile，编译新的 ftpdLib.o 后，把它重新加入到原来的库文件中。

```
arcf -r <tornado>/target/lib/libMCF5200gnuvox.a ftpdLib.o
```

 ar 的命令参见文档 GNU Toolkit User's Guide 中的 The GNU Binary Utilities。

ftpdLib.c 文件中关于 guest 的权限设置还有几处错误需要修改。

(1) ftpdSessionAdd 中（989 行）。

```
if(defaultHomeDir[0] == EOS )
    ioDefPathGet (pSlot->curDirName);
else
    strcpy( defaultHomeDir, pSlot->curDirName);
```

应改为：

```
if(defaultHomeDir[0] == EOS )
    ioDefPathGet (pSlot->curDirName);
else
    strcpy( pSlot->curDirName, defaultHomeDir);
```

并增加 guestHomeDir 初始设置。

```
if (guestHomeDir[0]==EOS)
```

```
strcpy(guestHomeDir, pSlot->curDirName);
```

(2) ftpPathAccessVerify 中（1091 行）应增加对 whrer 和 path 是否为空的判断。

```
if (*where==EOS || *path==EOS) goto deny;
```

(3) ftpdWorkTask 中（1411 行）如果有权限设定则处理，否则按照 guest 处理，程序中缺少“else”，应改为：

```
if ( pLoginVrfyFunc != (FUNCPTR)NULL ){
    .....
}
else if( guestHomeDir[0] != EOS ){
    .....
}
```

8.4.2 Telnet 服务器

Telnet 服务器的加载类似于 FTP 服务器。在选择“telnet server”组件即可。其权限设定和 FTP 服务器相同。

用 Telnet 登录到目标机后，目标机的标准输入输出就被重定位在 Telnet 上，比如 logMsg 的输出，tShell 也从 Console 转到了 Telnet，这样很方便开发者的调试和使用。

8.5 常见问题解答

● VxWorks 支持哪些常用网卡？

解答：VxWorks 支持的网卡如表 8-7 所示。

表 8-7 VxWorks 支持的网卡

INCLUDE_XXX	网卡	备注
INCLUDE_ENE	Eagle/Novel NE2000 网卡	包括兼容卡，不支持即插即用
INCLUDE_ELT	3COM EtherLink III 网卡	包括兼容卡，不支持即插即用
INCLUDE_FEI	Intel Ether Express PRO100B PCI 网卡	
INCLUDE_EL_3C90X_END	3com fast etherLink XL PCI 网卡	
INCLUDE_LN_97X_END	AMD 79C972 网卡	

● 如何动态更改 IP 地址？

解答：系统启动时 IP 地址设定为 Bootline 中的地址，在系统启动后可以通过下面的调用动态改变 IP 地址。

```
usrNetEndDevStart( "ene", 0 );
usrNetIfConfig( "enel", 0, "192.166.0.2", 0, 0xffffffff00);
```

● VxWorks 支持绑定多个 IP 地址吗，如何添加？

解答：调用 ifAddrSet 可以给每个网络接口添加多个不同的 IP 地址。

● 如何得到网卡的 MAC 地址？

解答：程序示例如下：

```
#include "vxWorks.h"
#include "ifLib.h"
#include "etherLib.h"
```

```

#include "netinet/if_ether.h"
void GetEtherAddress ( char * ifn )
{
    u_char eaddr[6];
    struct ifnet *pIf;
    int i;
    if (ifn == NULL)
    {
        printf ("Interface name specified is NULL.\n");
        return ;
    }
    pIf = ifunit (ifn);
    If (pIf == NULL)
    {
        printf ("Interface with name %s doesnt exist.\n", ifn);
        return ;
    }
    for (i = 0; i < 6; i++)
    {
        eaddr[i] = ((struct arpcom *) pIf)->ac_enaddr[i];
        if (i < 5)
            printf ("%02x:", eaddr[i]);
        else
            printf ("%02x\n", eaddr[i] );
    }
    return;
}

```

- 如何使得系统动态改变 Bootline 缺省的内容，比如更改其中目标机的地址？

解答：要使得系统可以动态改变 Bootline 值，则需要有 NVRam 来存放设定的值。在系统启动时调用 sysNvRamGet 从 NVRam 中指定地址处取得 Bootline 的内容，系统启动后也可以将新的 Bootline 值回写到 NVRam 中。

- 何时使用非阻塞套接字？

解答：套接字分为阻塞和非阻塞两种模式。阻塞模式意味着当一个套接字调用不能立即完成时，任务处于睡眠状态。可能引起阻塞的套接字调用包括以下几个方面。

- (1) 输入操作：read recv recvfrom recvmsg
- (2) 输出操作：write send sendto sendmsg
- (3) 接收外来连接：accept
- (4) 初始化连接：connect

套接字缺省是阻塞型的，但是当在一个任务管理多个套接字，任务并不允许阻塞在某一套接字上，而只是希望对套接字进行轮询处理，此时应该使用非阻塞型的套接字与 select 配合使用。

- 如何设置非阻塞型套接字？

解答：可以通过 ioctl 设置非阻塞套接字。

```
// Set fd into non-blocking mode via ioctl().
optval=1;
if (ioctl(fd, FIONBIO, (int*)&optval)<0)
{
    // An error has occurred, see errno for details.
}
```

● 套接字在什么条件下准备好？

解答：可能出现的情况如下所列。

(1) 读准备好

如果套接字有新数据，则此时套接字可读，读操作不阻塞并返回大于 0 的值。

如果接收到对方的关闭请求，则此时套接字可读，读操作不阻塞并返回值为 0。

如果套接字发生错误，则此时套接字可读，读操作不阻塞并返回-1，`errno` 为错误号。

也可以通过套接字选项 `SO_ERROR` 调用 `getsockopt` 来取得并清除错误。

如果套接字是监听套接字并且有新的连接请求，此时套接字可读，调用 `accept` 可以接收此请求。

(2) 写准备好

套接字发送缓冲区有可用的空间，此时可以进行 `send`、`sendto` 等调用。如果套接字为非阻塞的，则返回总共发送的字节数；如果套接字为阻塞的而缓冲区长度小于所需发送的字节数长度，此时可能引起调用阻塞。

如果一个非阻塞套接字正在连接，写准备好意味着成功建立了连接。

有一个套接字错误等待处理。此时套接字可写，写操作不阻塞并返回-1，`errno` 为错误号。也可以通过套接字选项 `SO_ERROR` 调用 `getsockopt` 来取得并清除错误。

注意当一个套接字发生错误时，它由 `select` 置为既可读又可写。但套接字既可读又可写并不一定表示发生了错误，此时要以通过套接字选项 `SO_ERROR` 调用 `getsockopt` 来确认。

● VxWorks 中如何缩短 `keepalive` 的探测时间？

解答：在 VxWorks 中有如下 3 个全局变量，在 “`target/h/netinet/tcp_timer.h`” 中定义。

`tcp_keepidle`：发送保持探测分节前的等待时间，缺省是 2h。


`tcp_keepintvl`：发送保持探测分节的间隔时间，缺省是 75s。

`tcp_keepcnt`：保持探测分节的个数，缺省是 8。

当对方主机崩溃时，TCP 的探测周期为 “`tcp_keepidle+(tcp_keepintvl*tcp_keepcnt)`”。所以用户在用户程序中创建 `socket` 之前通过修改这 3 个全局变量就可以更改 `keepalive` 的探测时间。

需要特别指出的是 TCP 是以整个内核为基础维护这些参数的，而不是以每个套接口为基础来维护的，因此更改此时间参数将影响系统中所有打开 `SO_KEEPALIVE` 选项的套接口。

在 WindSh 下，可以通过 `tcpstatShow` 命令来查看探测分节的发送情况。

 时间变量的值是以 0.5s 为单位的，如缺省值 `tcp_keepintvl=150` (75s)。

举例如下：

```
extern int tcp_keepidle;           /* time before keepalive probes begin */
extern int tcp_keepintvl;         /* time between keepalive probes */
extern int tcp_keepcnt;           /* probes count */
tcp_keepidle = 60;                /*空闲等待 30s*/
```



```
tcp_keepintvl = 30;          /*探测分节发送间隔 15s*/
tcp_keepcnt = 3;             /*探测分节发送个数 3 次*/
```

● 用 FTP 登录目标机后，当前路径显示“host:/c:”，并且告诉没有权限查看下载 VxWorks 的路径，怎样才能查看目标机上的文件？

解答：系统启动后的 IO 缺省路径是“host:/c:”，所以出现了上述情况，通过 ioDefPathSet 来更改缺省的 IO 路径即可，例如更改到“tffs0”。

```
ioDefPathSet(“/tffs0”);
```

● WindSh 下有哪些可用的命令来查看网络信息？

解答：如表 8-8 所示即为 WindSh 下用于查看网络信息的命令。

表 8-8 WindShell 下查看网络信息的命令

调用	描述
hostShow	显示 host 表
icmpstatShow	显示 ICMP 的统计信息
ifShow	显示已加载的网络接口信息
inetstatShow	显示所有活动的 Socket 连接信息
ipstatShow	显示 IP 相关的统计信息
routestatShow	显示路由信息
tcpstatShow	显示 TCP 相关信息，比如探测分节的统计信息
tftpInfoShow	显示 TFTP 相关信息
udpstatShow	显示 UDP 相关信息

第 9 章 建立开发环境

本章和下一章将介绍与实际开发工作相关的一些话题。本章主要介绍主机与目标机连接，建立交叉开发环境，BSP 定制和 BootRom 的建立，建立 VxWorks 应用，VxWorks 的启动和运行，程序映像，VxWorks 定制等。阅读本章后，读者就可以着手开始自己的应用程序开发了。

9.1 主机和目标机

嵌入式系统本身资源很有限，如内存小和 CPU 速度低等，不足以单独支撑开发环境的建立。而且用户接口缺乏，如没有显示器和键盘等，即使开发环境在嵌入式系统上建立，也很难使用，不能进行高效开发工作。另外，开发环境一般只适用于主流的硬件平台，不可能普遍适用于种类繁杂的嵌入式 CPU，所以大多数嵌入式系统开发采用交叉开发方式，即开发工具放在主机上，通过某种方式连接目标机，主机和目标机交换信息完成调试工作。

Tornado 和 VxWorks 两个名称就体现了这种开发模式。本书的描述中也处处体现出这个概念。简单的说，Tornado 位于主机上，VxWorks 运行于目标机，两者通过网络、串口等通信方式连接，共同完成整个开发过程。应用程序和装置完成后，VxWorks 脱离 Tornado 独立实现装置功能。

其实在前面的第 2、3 章中已经介绍了相关的一些内容。为了加深印象，再换个角度来看主机和目标机的关系。就像笔者对 UML 最深的一点体会，事物不可能从单一角度来描述清楚，而需要多个角度或视图配合完成整个描述。多个描述不可避免的重叠，是为了更好地看清事物的本质。下面这几个示意图也体现了这种多视图思想，便于读者进一步了解主机和目标机的关系。

如图 9-1 所示按 UML 术语可以称为物理视图，是主机和目标机关系最直观的表现。主机为功能强大的计算机或工作站，带有通常的显示器和键盘，看起来很适合在上面进行写文档、编程序、编译连接和版本控制等开发工作。而目标机就显得有些简陋，该没有的都没有，隐约能看见芯片和总线槽。看起来绝对不适合进行上面提到的工作。但目标机却是开发人员服务的对象，而主机只是一个工具而已。主机和目标机通过网络和串口相连，两者可选其一，也可同时使用，如网络用于建立调试器连接，而串口用作 Console。该图就是需要建立的开发环境的示意，是不是显得很简，当读者看了后面的两幅图也许就不会这么想了。

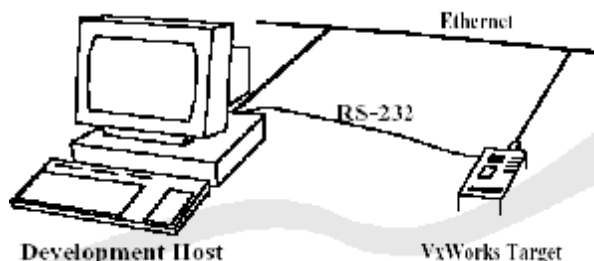


图 9-1 主机和目标机连接 1

如图 9-2 所示就是主机和目标机的另一个视图。硬件表面的简单性实在无法和软件深层

的复杂性相比。希望搞硬件的朋友不要生气，其实硬件深层也是很复杂的，只是国情所限，一般不管芯片级的复杂而已。主机上驻留了 Tornado 集成环境和众多工具，主要用于两个目的：编译和调试。GNU 工具应用 Makefile 规则将各种源文件生成目标模块、目标模块集或执行映像。各种映像通过连接下载到目标机中，按代码段、数据段、未初始化段存放在目标机内存中。再用 CrossWind、WindSh 和 Browser 等工具配合进行程序映像的调试。这个过程无限往复就构成了开发人员的生活。主机上还有一部分内容未在图上标出，就是 VxWorks 库和源代码，考虑到它们并不是真正属于主机，而只是个过客而已。各种主机工具都通过 Target Server 与目标机连接。Target Server 在目标机侧的接应者为 WDB Agent。由 Agent 作为目标机代表与主机交流。VxWorks 在目标机上运行。图中 VxWorks 独立于各段内存显示，并不说明 VxWorks 不存在内存中。可能是为了突出其基础作用，它为各应用程序提供服务。其实和别的映像一样，如小框中分段存在内存中。小框部分的表示更多是体现一个加载应用目标模块在内存中的分配。

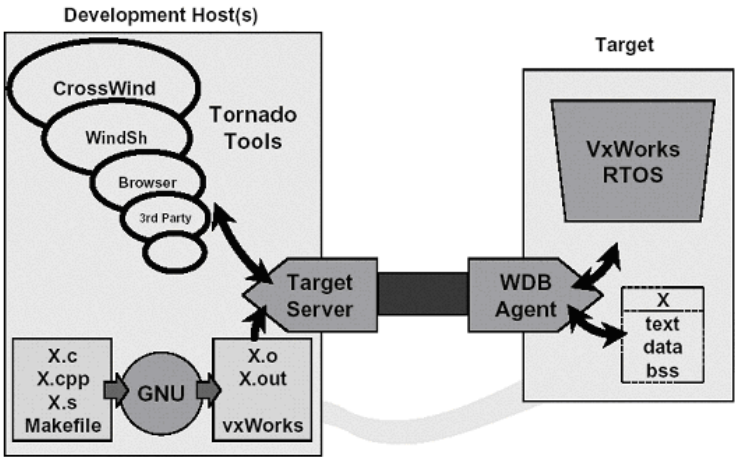


图 9-2 主机和目标机连接 2

图 9-3 所示和图 9-2 所示比较接近。最主要的不同在于体现了主机工具和 Target Server 之间，Target Server 和 WDB Agent 之间所用的软件协议，WTX 和 WDB。另外物理连接上出现了另外一种方式 netrom。

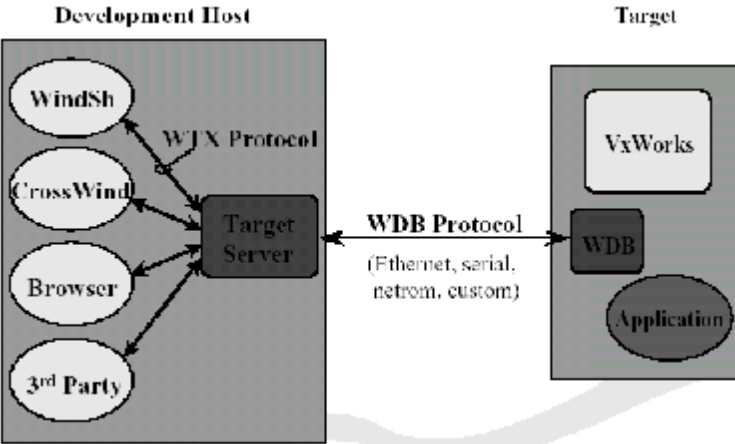


图 9-3 主机和目标机连接 3

如图 9-4 所示额外表示了编辑器和工程管理主机工具，在编写代码和工程管理中使⤵用。图 9-4 所示中另外一个不同点是显示了 VxWorks 目标机模拟器[Simulator]和 Target Server 之间虚拟连接，模拟器运行在主机上，用于原型建立和部分代码测试。

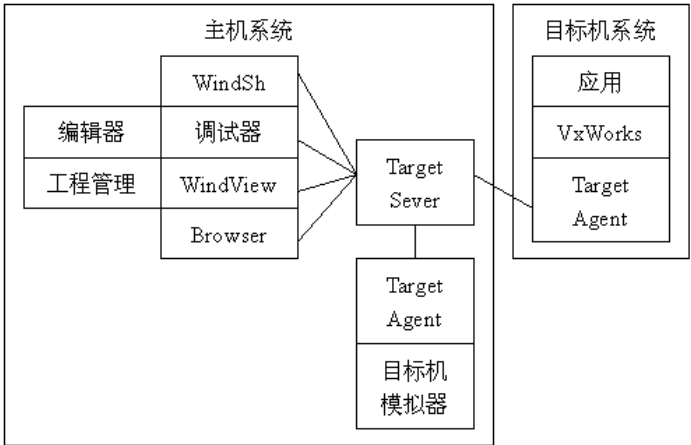


图 9-4 主机和目标机连接 4

9.2 板级支持包[BSP]

可以从 3 个独立的视角来定义 BSP。

BSP 是个软件抽象层，位于硬件和上层代码之间，为上层软件提供与特殊硬件无关的统一操作接口，如图 9-5 所示。这保证了上层代码的移植性，也使软件结构清晰。理论上任何和硬件相关的代码都属于 BSP 层，但实际中并不能做得如此完美，如用户应用代码直接访问硬件寄存器，这可能是为了效率或代码简洁，此时 BSP 和其他代码之间的界限就变得模糊。由于 BSP 的存在，VxWorks 大部分库与所使用的特殊目标板无关。但由于 VxWorks 的组件结构和库机制，BSP 层概念已经不是很清晰。BSP 目录中并没有完整地包含 BSP 层代码，有些驱动代码存在库中或其他源代码目录下，而一些 CPU 相关操作（如 intLock）也由 VxWorks 库提供。BSP 支持硬件初始化、中断处理、硬件时钟和内存映射等。

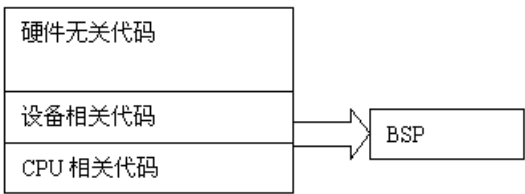


图 9-5 板级支持包示意

BSP 是设备驱动的内核接口，有自己的编写和接口规范。如网卡驱动、串口驱动和块设备驱动等都需遵循该规范，否则就不能和上层系统顺利衔接。遵循规范，可利用现成代码，或参考现成代码，并能提高代码移植性和开发效率。定制 BSP 的修改可参考近似 BSP 或 BSP 模板进行。自己的驱动或特殊硬件的代码，也可以不遵循规范，而采用自己开发规范，只是需要多做些工作。从功能角度讲，这些代码也属于 BSP 层。若从移植性看，又不属于 BSP 层，因为不能在各硬件上普遍适用，不能为上层软件的通用性服务。

BSP 支持主机&目标机交叉开发环境。VxWorks 中很容易将 BSP 和 BootRom 混淆，准确

地说是 BootRom 支持了交叉开发环境的建立。

在前面的第 3 章对 BSP 做了概略的描述，本节将从源代码的基础上做进一步详细描述，以便完成后面的 BSP 定制和 BootRom 建立工作。

前面的第 3 章中也描述了 VxWorks 相关库和代码在主机上的目录结构。和本章密切相关的目录为 “/target/config/”。一般意义上，特定 BSP 只和 “/target/config/” 下对应目录相关，目录按硬板命名，称为 BSP 目录。其他一些和硬件相关的驱动其实也是属于 BSP 这个概念层，但是通用于该系列体系结构的各硬板，并且在实际编程中一般不会涉及这些驱动，所以一般不将它们作为标准 BSP 的一部分。

要理解 BSP，需要明白 VxWorks 的启动顺序。阅读该目录下的源代码有助于了解底层工作机制，增强对系统的控制能力。有些源代码的注释在正式文档中没有，但是却很重要。下面以 MCF5272 为例，浏览一遍 BSP 目录下的源代码。其中和实际工作紧密相关的配置文件（如 config.h）放在 “BootRom 建立” 一节描述。

● romInit.s

首先是 romInit.s，定义了系统入口 romInit 函数，它是系统上电后最先执行的代码。文件开头大段的注释应该详细阅读，但注释和实际的代码有出入，需要读者自己分辨。

romInit 中完成禁止 CPU 中断，设置初始栈指针，配置 DRAM 控制器等少量 CPU 设置之后调用第一个 C 函数 romStart。进一步的硬件初始化由 sysLib.c 中定义的 sysHwInit 函数完成。在 BSP 开发中，romInit.s 只要使 RAM 工作起来就行，不用进行太多的硬件初始化。

romInit.s 中的硬件初始化会在 sysALib.s 或 sysLib.c 重复一遍。因为 VxWorks 程序只需要 BootRom 的参数串和启动类型码，而不依赖 BootRom 的硬件初始化（除了 RAM 初始化）。VxWorks 启动时会重新初始化 CPU 和所有其他硬件。RAM 也可在 VxWorks 中再初始化一遍，对程序运行不会有影响。BootRom 和 VxWorks 的无关性是个基本原则，若不遵循这个原则，就会出现 VxWorks 和 BootRom 的关联修改。

由于该文件代码在 ROM 中执行，而一般程序映像整体（包括该段代码）定位在 RAM 中。所以要求是位置无关代码，即不包含任何绝对地址引用。如果想使用绝对地址，必须将该地址转换为 ROM 中地址，这样就和映像连接时的定位地址无关。该代码中惟一的这种引用就是对 romStart 函数的调用。

```
/******
* romInit (startType) - entry point for VxWorks in ROM
*   int startType;    /* 只由热复位使用，从栈中取参数，冷复位直接指定 @/
*/
_romInit:
movew    #0x3700, sr    /* 禁止中断 */
braw cold
movew    #0x3700, sr    /* 热复位入口，见 sysLib.c 中 sysToMonitor() 定义 */
braw warm
cold:
movel     #BOOT_COLD, d7 /* 直接指定启动类型 */
bra start    /* skip over next instruction */
warm:
movel     a7@(0x4), d7   /* put startType in d7 */
start:
```

```

movel    #0, d0          /* clear both ACRs */
movec    d0, acr0
movec    d0, acr1
movel    #(INTERNAL_SRAM_BASE + 1), d0      /* Map internal SRAM */
movec    d0, rambar
movel    #(SIM_BASE + 1), d0                /* Map SIM module */
movec    d0, mbar
nop
/* 先将栈定位在 CPU 内部 RAM 中， 其实没什么用，因为在下次栈重定位前没有任何函数调用 */
movel    #(INTERNAL_SRAM_BASE+M5272_SRAM_SIZE-4), a7
nop
/* 片选设置： 2M Flash */
movel    #(ROM_BASE_ADRS + M5272_CS_CSBR_EBI_1632 + \
          M5272_CS_CSBR_BW_WORD + M5272_CS_CSBR_ENABLE), d0
movel    d0, M5272_CS_CSBR0(SIM_BASE)
movel    #(M5272_CS_CSOR_BAM_2M + M5272_CS_CSOR_WS(5)), d0
movel    d0, M5272_CS_CSOR0(SIM_BASE)
/* CPU 内看门狗初始化为最大值，以免在启动中被复位 */
movel    #(M5272_SIM_SCR_SoftRST + M5272_SIM_SCR_HWR_16384), d0
movel    d0, M5272_SIM_SCR(SIM_BASE)
/* program PMR - enable clocks to all modules */
clrl d0
movel    d0, M5272_SIM_PMR(SIM_BASE)
/* 禁止所有中断源 */
movel    #(0x88888888), d0
movel    d0, M5272_SIM_ICR1(SIM_BASE)
movel    d0, M5272_SIM_ICR2(SIM_BASE)
movel    d0, M5272_SIM_ICR3(SIM_BASE)
movel    d0, M5272_SIM_ICR4(SIM_BASE)
/* 如果不是冷复位，就跳过 SDRAM 初始化 */
cmpl #BOOT_COLD, d7
bne stackToSdram
sdramInit:
/* 初始化 CS7 控制 SDRAM */
movel    #(SDRAM_BASE + M5272_CS_CSBR_EBI_SDRAM + \
          M5272_CS_CSBR_BW_LINE + M5272_CS_CSBR_ENABLE), d0
movel    d0, M5272_CS_CSBR7(SIM_BASE)
movel    #(M5272_CS_CSOR_BAM_16M + M5272_CS_CSOR_WS(0x1f)), d0
movel    d0, M5272_CS_CSOR7(SIM_BASE)
/* 根据时钟频率设置 SDRAM 相关的 SDTR 和 SDCR */
#if (MASTER_CLOCK == 66000000)
movel    #(M5272_SDRAM_SDTR_RTP_66 + M5272_SDRAM_SDTR_RC_6 + \
          M5272_SDRAM_SDTR_RP_4 + M5272_SDRAM_SDTR_RCD_3 + M5272_SDRAM_SDTR_CLT_2), d0
movel    #6600, d1
#define REG_BIT M5272_SDRAM_SDCR_REG
/*省略其他频率的选择代码*/
#else
#error "Unsupported master clock frequency"
#endif
dly:
nop /* 延时循环，大约 100us，让 SDRAM 启动 */
subl #1, d1
bne dly

```

```

movel    d0, M5272_SDRAM_SDTR(SIM_BASE)    /* Set the timing register */
movel    #(M5272_SDRAM_SDCR_MCAS_A9 + M5272_SDRAM_SDCR_BALOC_A21 + \
REG_BIT + M5272_SDRAM_SDCR_INIT), d0
movel    d0, M5272_SDRAM_SDCR(SIM_BASE)    /* Set the mode register */
movel    #0, d0
movel    d0, SDRAM_BASE                    /* Dummy write to start */
nop
stackToSdram:
/* 将栈定位在 SDRAM 中 */
movel    #STACK_ADRS, a7
nop
/* 计算 C 函数 romStart() 的入口地址(routine - entry point + ROM base) ,
即两函数之间相当偏移 + ROM 的基地址, 因为 romInit 位于 ROM_TEXT_ADRS 处 */
movel    #_romStart, a0
/* 自己在 config.h 添加宏定义 RAM_SIM, 用于 VisionProbe 仿真 */
/* 仿真时, 代码由 Probe 直接下载到 RAM 中, 该段代码在 RAM 中运行, 所以可以直接引用 romStart */
#ifdef RAM_SIM
    subl #_romInit, a0
    addl #ROM_TEXT_ADRS, a0
#else
    /* 将启动类型码压入栈中, 传给 romStart() 函数 */
    movel    d7, a7@-
    jsr    a0@

```

● sysALib.s

这是 VxWorks 中另外一个汇编代码文件, 实现了 RAM 中 VxWorks 的入口函数 sysInit。该函数代码位于 VxWorks 程序映像的开始, 保证实现 BootRom 到 VxWorks 的成功跳转。重复实现 romInit 的部分功能, 如禁止中断和初始化栈等, 然后跳转到 usrInit。


```

/*****
* sysInit - RAM 中 VxWorks 入口
*/
_sysInit:
movew    #0x3700, sr    /* 禁止中断 */
movel    #C_CACR_CINVA, d0    /* 关闭 cache */
movec    d0, cacr
movel    #0, d0    /* setup both ACRs */
movec    d0, acr0
movec    d0, acr1
movel    #_sysInit, a7    /* 设置栈指针, 栈从代码位置向下增长 */
movel    #BOOT_WARM_AUTOBOOT, a7@-    /* 直接设置为热启动类型: WARM_BOOT */
jsr    _usrInit    /* 永不返回 - 启动内核 */

```

该栈的设置是临时的, 紧接在程序映像的下面, 只用来传递一次启动类型码给 usrInit, 后续代码会再重新定位栈, 该临时栈不再使用。由于该栈位置的特殊性, 在连接确定 VxWorks 加载地址时要考虑保留出栈空间。比如实际使用的加载地址为 0x1000, 前 0x400 空间用来存放 CPU 向量表, 0x400~0x1000 空间就可用做临时栈。其实栈只使用了紧接着 0x1000 的几个字节, 大部分空间没用。

上面的两文件为系统中仅有的两个汇编代码文件。

 其他硬件参考 vxFree 的 “MPC860 上电初始化流程分析” 和 “X86 romInit.s 分析”。

● sysLib.c

该文件是 BSP 的主要源代码文件，不像上两个系统初始化文件，sysLib.c 没有明显执行流程，而为上层代码提供硬件相关的服务接口，主要函数如表 9-1 所示。

表 9-1 sysLib 函数

函数	说明
sysHwInit	目标机核心硬件初始化，如片选、中断、系统时钟等
sysHwInit2	目标机附加硬件初始化，如辅助时钟等
sysPhysMemTop	取物理 RAM 内存最大地址
sysMemTop	取 VxWorks 可用内存的最大地址，允许用户在物理 RAM 顶保留空间
sysToMonitor	用于系统热复位

还有一些其他服务函数，如网络相关的，都比较简单，开发者一看就能明白。这些函数一般不让用户调用，而是在系统初始化时，由初始化序列调用。应用程序中可能要使用系统热复位功能，但也是不直接调用 sysToMonitor，而是调用 reboot 库函数，由 reboot 再调用 sysToMonitor。由于 sysLib.c 会加入 VxWorks 工程，且这些函数未声明为静态，应用程序必要时也可直接调用这些函数。

除了函数定义，sysLib.c 中还有一些重要的全局变量定义。

```
char sysBootHost [BOOT_FIELD_LEN]; /*启动主机名称*/
char sysBootFile [BOOT_FIELD_LEN]; /*启动文件名称*/
/*boot flags, 各标志在 sysLib.h 中定义，启动类型也在该文件中定义*/
int sysFlags;
/*系统 CPU 类型, 取值 Makefile 中 CPU 定义，具体值参考 vxCpu.h*/
int sysCpu = CPU;
char *sysBootLine = BOOT_LINE_ADRS; /*Bootline 存放地址*/
char *sysExcMsg = EXC_MSG_ADRS; /*指向异常消息区的指针*/
```

 Ron Fredericks, “FAQ: What is a Board Support Package?”, Wind River, 2002. 12.

9.3 系统启动

VxWorks 应用系统按一定的顺序完成系统启动，而在各种不同的应用环境，启动顺序也有不同。

从宏观的角度来看，一般是先加载 BootRom；BootRom 再加载 VxWorks 应用，并跳转到 VxWorks 的入口 sysInit 执行；VxWorks 有可能根据启动脚本文件加载应用模块。整个启动过程完成后，系统进入多任务环境运行。

根据硬件体系结构的不同，启动方式有所不同。对于 ROM 启动系统，程序映像存放在非易失存储器中。而存储器位于 CPU 运行空间内，CPU 复位后可以直接从存储器执行程序映像，不需要额外的辅助机制。对于 PC 兼容体系结构，程序映像存放在辅助存储器（如软盘或硬盘）中，CPU 不能直接从这样的存储器中取指令执行，而需要 BIOS 的辅助。BIOS 将启动盘的引导扇区复制到主内存空间，再跳转内存的给定地址执行。BIOS 本身所处的存储器是 CPU 可直接访问执行的。

BIOS 在加载程序映像时有自己的规范，比如引导扇区的位置和大小等。BootRom 自己并不能满足这样的要求，比如说大小，BootRom 比一个扇区大了许多。BIOS 不能直接加载 BootRom，而只能先加载辅助小程序 Vxld。

Vxld 很小，可以在引导扇区中完全存储，符合 BIOS 的加载规范。Vxld 被 BIOS 加载到内存中执行，Vxld 再加载 BootRom 来继续启动。由于大小的限制，Vxld 不支持完善的文件系统，所以要求启动盘上 BootRom 文件必须连续存放，否则会导致加载失败。

主机工具 mkboot 和 vxcopy 就是用来帮助制作符合这些限制的启动盘。mkboot 将 Vxld

写入启动盘的引导扇区，vxcopy 将 BootRom 连续地存放在启动盘上。当 BootRom 加载成功后，完善的文件系统被支持，VxWorks 应用程序映像存放在启动盘或其他盘中的存放就比较自由了，可用普通的文件操作复制应用映像文件，只要符合文件系统规范就行。

BootRom 加 VxWorks 的启动形式有其独特优点，如适应硬件、方便调试和现场升级等。对于 PC 兼容体系结构，由于硬件环境的限制，Vxld 必须做得很小，不能启动保护模式和支持文件系统，而限制了所能加载程序映像的大小和内存加载位置。当然启动盘（如软盘）的大小也会限制程序映像的大小。

BootRom 相对于 VxWorks 应用来说，功能比较简单，大小易于控制，适合作为先行加载。BootRom 的存在也维持了各种硬件平台之间的兼容性。BootRom 成功运行后，启动了保护模式和文件系统支持，驱动了网络、串口等辅助设备，消除了硬件环境对程序映像的限制，使得 VxWorks 应用程序的制作和加载变得自由随意。BootRom 的功能较简单，代码量少，没有应用代码的干扰，容易保证其正确。而初始基础系统的正确是后续开发调试工作的保证。BootRom 启动了文件系统和网络，方便了调试映像的加载，可以通过文件系统或网络加载调试映像，使开发迭代工作更为方便和快捷。

当开发工作结束后，系统到现场运行时，这种启动形式也方便了现场的程序升级，只需要替换 VxWorks 映像文件和目标模块文件，或添加、删除一些目标模块文件，修改一下启动脚本文件就可以了。其实早先在开发别的非 VxWorks 系统时，笔者就开始使用 BootRom 这个概念软件，当时主要考虑现场程序的更换，主要解决 Flash 存储器在线烧录的问题。当然没有 BootRom 完善，不支持文件系统和标准执行文件加载功能，启动后只是简单的跳转执行。

BootRom 加 VxWorks 的启动形式随应用环境不同而有所变化。在有些应用环境中，特别是 ROM 启动硬件环境，由于 ROM 特性的支持和存储器空间的限制。可能出现 BootRom 不单独存在，部分功能与 VxWorks 应用映像融在一起的情况，这种映像可以直接从 ROM 启动，而不用借助 BootRom，但是会损失 BootRom 所具有的一些灵活性，比如说现场程序升级。

程序映像启动时是否加载到 RAM 中执行也是可以选择的。同样是针对 ROM 启动硬件环境，ROM 特性支持从存储器位置直接执行指令。当 RAM 空间有限时，可以保留程序映像在 ROM 中（笔者先前做的大多数系统都是如此）。在 VxWorks 中，这种概念称为 ROM 驻留 [rom-resident]。这种情况下，程序映像的代码段留在 ROM 内，只有 data、bss 段进入 RAM 中，减少了对系统 RAM 的需求。

VxWorks 的启动情况随硬件环境和用户选择而变化。不同的启动方式也影响了对程序映像形式的选择。“程序映像”一节会对这点再做详细介绍。

虽然存在多种不同的启动方式，但这中间也存在一些共同的地方。下面再从几个角度进行描述。

先从文件的角度来看启动过程。下面按启动顺序列出相关文件，如表 9-2～表 9-4 所示按程序映像类型分类，这里只列出了几种典型程序映像的启动，对某些特殊生成的程序类型未做描述，如在命令行下手动生成的 VxWorks 映像。

表 9-2

BootRom

romInit.s	实现系统初始化入口 romInit，完成最基本的 CPU 初始化，如禁止中断和片选初始化等，为后续启动作好准备。完成后跳转到 romStart 执行
bootInit.c	实现系统第一个 C 函数入口 romStart。主要完成程序映像中各段到 RAM 中

	的复制，各种程序映像在此分支，完成后调用 usrInit 函数
bootConfig.c	实现 usrInit 函数，完成进入多任务环境前所有的初始化工作，完成后进入多任务环境，usrRoot 为第一个系统任务，创建 tBoot 任务，tBoot 任务完成 VxWorks 映像的加载

表 9-3 VxWorks

sysALib.s	实现 VxWorks 的专用入口 sysInit。假设 CPU 基础硬件初始化完成，简单初始化 CPU 后跳转到 usrInit
prjConfig.c	实现 usrInit 函数，完成进入多任务环境前所有的初始化工作。
usrKernel.c	实现 usrKernelInit 函数，使系统进入多任务环境。
prjConfig.c	实现 usrRoot 任务函数，完成操作系统一些上层初始化后，调用 usrAppInit，完成应用代码必要的初始化，应用代码可能会在这里创建多个任务

表 9-4 ROM 启动 VxWorks

romInit.s	实现系统初始化入口 romInit，完成最基本的 CPU 初始化，如禁止中断和片选初始化等，为后续启动作好准备，完成后跳转到 romStart 执行。
romStart.c	实现系统第一个 C 函数入口 romStart。主要完成程序映像中各段到 RAM 中的复制，各种程序映像在此分支，完成后跳转 sysInit() 执行
	后续步骤同前面的 VxWorks 顺序

想要详细了解启动过程和其他映像类型的启动。需要阅读相关的源代码，因为各种体系结构的差异都在代码中体现，任何文档都不能完全涵盖代码所表达的意思。而且还要了解程序映像的连接过程，所连接的目标文件，以及各函数在内存中的定位。这些可以从 build 的输出窗口得到一些信息。

了解启动过程顺序对 BootRom 制作和开发调试环境的建立有很重要的指导意义，后面章节对启动的关键步骤还会做一些实践性的描述。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 8.3、8.6.2、8.6.3 章节。

9.4 程序映像

Tornado 可以为 VxWorks 和应用代码生成多种类型的映像，以灵活地适应多样的目标机应用环境。

9.4.1 映像类型

对于初学者而言，VxWorks 系统提供的众多的程序映像类型最让人困惑，不知道这些映像有什么用，怎么使用，如何生成。而 VxWorks 的参考手册对这些映像没有集中描述，而且有些描述已经过时。手册的这个问题在 WindSurf 上 TechTips 和 SPR28252 中有描述。TechTips 指出用户了解这些类型的信息源，有兴趣的可以看看。

其实要了解这些映像类型，关键是把握映像的分类标准。各分类不是单一的，而是互相重叠，如表 9-5 所示。

表 9-5 映像类型

分类标准	类型
功能	BootRom, VxWorks, 目标模块, vxworks_rom
运行位置	ROM 运行[rom-resident], RAM 运行的
压缩	压缩的[compressed], 未压缩的[uncompressed]
文件格式	标准执行格式 (如 ELF), Hex 格式 (只对 ROM 型映像而言), 二进制格式
符号表	Standalone, 包含符号表和 tShell 等

ROM 驻留类型映像运行时代码段保留在 ROM 上, 其他段 (如 data 和 bss 等) 在 RAM 中分配, 可以节省 RAM 空间。

Standalone 映像类型现在看来已经没有实际的意义, 只能作为一种概念, 表示不需要其他计算机系统就可以运行, 需要包括一些组件, 如 tShell、内建符号表和 Loader 等。

“VxWorks Programmer's Guide 5.4”中的一些相关描述已经过时了, 比如, 在 8.9.1 章节中有这样一句话: “按 vxWorks.st 的编译规则, 定义 STANDALONE 标志编译 usrConfig.c 生成 usrConfig_st.o”。使用 Tornado 工程管理就没必要了, 只要包含需要的组件, 或删除不要的组件 (如 WDB 相关主机调试组件), 就得到用户想要的程序映像。如果自己认为该程序映像是 Standalone 的, 那它就是。

明白程序映像文件名称中一些缩写含意, 也有助于类型的理解, 如表 9-6 所示。

表 9-6 映像相关缩写含义

缩写	含意	举例
hex	文件后缀, 表示为 Hex 文件格式, 多用于 ROM 烧录; 未带后缀的映像, 多为标准执行文件格式, 用于加载和调试; 对于 ROM 启动映像都有上两种格式	vxWorks_rom vxWorks_rom.hex
uncmp	uncompressed, 未压缩映像	bootrom_uncmp.hex
Res	rom_resident, ROM 驻留执行映像	bootrom_res vxWorks_romResident
nosym	映像不带 build-in 符号表	vxWorks.res_rom_nosym
rom	BootRom 和 VxWorks 合二为一的映像	vxWorks_rom
st	Standalone 映像	vxWorks.st

下面按功能标准列出常用的映像类型 BootRom、VxWorks 和目标模块, 分别对应 3 个编译规则文件 rules.bsp、rules.vxWorks 和 rules.vxApp.vxWorks_rom 型映像一并在 rules.vxWorks 中描述。读者可以阅读 rules.* 文件中注释和编译规则, 有助于理解各种映像类型。

文件 rules.bsp 描述了建立 BootRom 映像的编译规则, 主要的几种类型如表 9-7 所示。

表 9-7 BootRom 类型

文件名	说明
bootrom	压缩、执行格式的 BootRom 映像
bootrom.hex	压缩、Hex 格式的 BootRom 映像

bootrom_uncmp	执行格式的 BootRom 映像（未压缩）（visionClick 仿真调试用）
bootrom_uncmp.hex	Hex 格式的 BootRom 映像（未压缩）（写入启动存储芯片）

BootRom 映像其实是一个最小化、专用的 VxWorks 映像，保存在固定位置，主要负责加载、运行系统最终使用的 VxWorks 映像。BootRom 运行时也建立起多任务环境，包括 usrRoot 任务、网络任务、TFFS 任务和 FTP 任务等。上面这几种映像都调入 RAM 中运行。

文件 rules.vxWorks 描述建立 VxWorks 映像的编译规则，主要的几种类型如表 9-8 所示。

表 9-8 VxWorks 类型

文件名	说明
vxWorks	RAM 运行的 VxWorks 映像
vxWorks_rom	ROM 启动 RAM 运行的 VxWorks 映像，不需要 BootRom 辅助
vxWorks_romResident	ROM 启动&运行的 VxWorks 映像，不需要 BootRom 辅助
vxWorks_romCompress	vxWorks_rom 的压缩型式
vxWorks.sym	VxWorks 的符号表文件，格式与普通目标模块同，主要用于调试

vxWorks_rom 和 vxWorks_romCompress 的重定位辅助代码（stub）连接定位到 ROM_TEXT_ADRS 和 RAM_HIGH_ADRS 地址。主映像连接定位在 RAM_LOW_ADRS。重定位过程由 stub 代码直接复制或解压缩完成。这两种 ROM 型映像由最基本的两部分组成：将映像从 ROM 重定位到 RAM 的代码；在 RAM 中运行的主程序。当 ROM 映像启动时，将加载代码从 ROM 复制到 RAM_HIGH_ADDR，并跳转到该地址执行。加载代码再复制从 sysInit 开始的剩余映像到 RAM_LOW_ADDR，然后跳转到该地址继续执行。vxWorks_romResident 的情况与它们类似，只是代码段直接定位在 RAM 中。

而 VxWorks 只包含上面提到的主程序部分，入口位置为 sysInit，一般由 BootRom 跳转到该位置执行。注意 vxWorks_romXXX 类型的映像不使用 sysInit。


加载代码可参考 romInit.s 和 bootInit.c 代码文件，sysInit 在 sysALib.s 文件中定义。阅读相关的源代码，有助于读者的理解。

文件 rules.vxApp 描述建立目标模块的编译规则，主要有 3 种类型，如表 9-9 所示。

表 9-9 Downloadable 工程映像类型

objects	由单一源代码文件生成的目标模块
archive	将多个目标模块合并为静态库，各模块之间不做符号解析，只能用于静态连接，不能用于动态加载
project_out	将多个目标模块连接为大目标模块，解析模块间符号引用，能用于动态加载

并不是每个 BSP 都支持所有这些程序映像。例如有些 BSP，甚至整个体系结构都不支持压缩映像。本书的第 5 章讲述了各种程序映像的内存分配，可参考以理解本节的内容。

gem2000，“VxWorks 压缩技术”，<http://bbs.edw.com.cn>，2003.04。

9.4.2 映像格式

在上节映像分类标准中有一项是文件格式。文件格式是标准的，不是 VxWorks 系统开发

所特有的。在所有嵌入式系统开发，以及其他计算机系统中都有执行映像文件格式的问题，只是平时容易忽略它们。比如，在用烧录器向 Flash 存储器写入程序映像时，烧录器程序就会将其他格式转换为纯二进制格式写入存储器。本节将就标准文件格式和 VxWorks 的实际情况来讲述映像的文件格式。

从 VxWorks 开发系统看，出现 3 种文件格式：标准执行文件、Hex 文件和 Bin 文件。大部分映像（包括目标模块，out 模块）都是标准执行文件格式。Hex 文件为 Motorola 格式，主要用于存储器烧录。Bin 文件没有直接出现，但可利用主机的一些小工具（如 elftobin 等）来生成，是映像存储器上的真正表现。Bin 文件在存储器中一般定地址存放，因为它不再包含地址定位信息。Bin 文件也可以用于存储器烧录，不过用户要明确知道相关的地址。

在 PC 结构系统中，由于文件系统的普遍存在，一般只使用标准格式映像。而对于一般嵌入式系统，Flash 作为主要的存储器，文件格式转换可能不可避免。

主机存在各种对映像操作的工具，可能对开发者有用，可参考第 2 章的相关描述。

下面对标准文件格式做进一步介绍，这里所讲的知识有普遍的适用性，在以后其他类型系统的开发中可能也用得上。

目标模块包含 5 种基本信息：头信息（文件的整体信息，包括代码大小、源文件名和生成日期等）、目标代码（包括 text、data 和 bss 等）、重定位信息和符号调试信息。各种格式在大的框架上都类似。要辨识一个映像属于什么类型，简单的方法就是二进制编辑器（如 Ultraedit）打开映像文件，头几个字节即显示出是什么格式。

目标模块有 3 种类型。

- ✧ 可连接的：包含 Linker 需要的丰富的符号和重定位信息，如普通的静态库。
- ✧ 可执行的：不需要任何符号，也几乎没有重定位信息。一般页对齐以允许文件被映射到地址空间，如连接完成的最终执行文件。
- ✧ 可加载的：由代码组成，可能包含完整的符号和重定位信息，以便运行时符号连接，如普通的目标模块、合并目标模块等。

基本的目标模块格式类型包括以下几种。

- ✧ AOUT：68K, sparc, x86。
- ✧ COFF：arm, i960。
- ✧ ELF：ppc, mips, sparc。
- ✧ PECOFF：x86。
- ✧ SOMCOFF：hp-pa (pa-risc)。

对于特定体系结构，VxWorks 开发环境只支持某种标准执行文件格式，不支持所有的格式。下面描述两种最常见的格式 AOUT 和 ELF，至于其他格式，读者可以参考相关的专业文档。

● AOUT 目标模块格式

典型的可重定位 AOUT 格式，如表 9-10 所示。

表 9-10

AOUT 映像格式

AOOUT 头	TEXT	DATA	Text Reloc	Data Reloc	符号表	串表
---------	------	------	------------	------------	-----	----

当代码被重定位到不同的基地址，代码中需要修改的地方被标识出来。在可连接文件中，对未定义符号的引用也被标识。这样，当符号最终被定义时，Linker 才知道要将符号的值补丁到哪些地方。

符号表中每项有 12 字节，遵循如下格式：

名称偏移(4 字节)-类型(1 字节)-保留(1 字节)-调试信息(2 字节)-值(4 字节)

text、data、bss、abs（绝对的不可重定位符号）和 undefined 是符号表中最通常的几种符号类型。这些符号用于连接器和调试器，不同于 VxWorks 中支持动态加载连接的符号表。

● ELF 目标模块格式

典型可重定位 ELF 格式，如表 9-11 所示。

表 9-11 ELF 映像格式

ELF 头	段表	.text	.data	.rodata	.bss	.sym
.rel.text	.rel.data	.rel.rodata	.line	.debug	.strtab	区表

ELF 头、段表和区表不作为一个区[section]。上面每个区都包括一种类型的信息，比如程序代码、初始化数据和重定位条目等。各区的说明如表 9-12 所示。

表 9-12 ELF 各区说明

区名	说明
.text	程序代码
.data	初始化数据
.rodata	只读初始化数据
.bss	未初始化数据
.rel.text	重定位信息
.rel.data	重定位信息
.rel.rodata	重定位信息
.init	程序启动时执行的代码
.fini	程序结束时执行的代码
.rel.init	重定位信息
.rel.fini	重定位信息
.symtab	常规符号表
.dynsym	动态符号表
.debug	调试器用符号
.line	源代码行号和目标代码位置的对应
.comment	文档串
.got	全局偏移表[global offset table]
.plt	过程连接表[procedure linkage table]

下面这小段程序实现从 VxWorks ELF 程序映像中提取区信息。程序运行显示 VxWorks 文件信息如下。

```
-> readelf "vxWorks"
```

```
Elf file is Executable
```

```
Entry point 0x100000
```

```
There are 1 program headers, starting at offset 34:
```

```
There are 17 section headers, starting at offset 1fff008:
```

```
Section Headers:
```

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	00000000	00000000	00		0	0	0
[1]	.text	PROGBITS	00100000	00010000	000c666c	00	AX	0	0	100
[2]	.rodata	PROGBITS	001c6670	000d6670	0000bbe8	00	A	0	0	8
[3]	.sdata2	PROGBITS	001d2258	000e2258	00000000	00	A	0	0	4
[4]	.data	PROGBITS	001d2260	000e2260	0000b4b0	00	WA	0	0	10
[5]	.ctors	PROGBITS	001dd710	000ed710	0000000c	00	WA	0	0	1
[6]	.dtors	PROGBITS	001dd71c	000ed71c	0000000c	00	WA	0	0	1
[7]	.got	PROGBITS	001dd728	000ed728	00000010	04	WA	0	0	4
[8]	.sdata	PROGBITS	001dd738	000ed738	00000000	00	WA	0	0	4
[9]	.sbss	NOBITS	001dd738	000ed738	00000398	00	WA	0	0	4
[a]	.bss	NOBITS	001ddad0	000ed738	00007914	00	WA	0	0	8
[b]	.stab	PROGBITS	00000000	000ed738	000576e4	0c		c	0	4
[c]	.stabstr	STRTAB	00000000	00144e1c	000b662f	00		0	0	1
[d]	.comment	PROGBITS	00000000	001fb44b	00003b48	00		0	0	1
[e]	.shstrtab	STRTAB	00000000	001fef93	00000074	00		0	0	1
[f]	.symtab	SYMTAB	00000000	001ff2b0	00017ea0	10		10	9c2	4
[10]	.strtab	STRTAB	00000000	00217150	0001276d	00		0	0	1

```
value = 0 = 0x0
```

具体代码如下。

```
#include <vxWorks.h>
#include <iolib.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <elf.h>

LOCAL char * filetype[] = {
    "None", "Rel", "Executable", "Dynamic", "Core", "Num"};

LOCAL char * sectiontype[] = {
    "NULL", "PROGBITS", "SYMTAB", "STRTAB", "RELA", "HASH", "DYNAMIC",
    "NOTE", "NOBITS", "REL", "SHLIB", "DYNsym", "NUM"};

int readelf(char *image) {
    Elf32_Ehdr * epnt;
    Elf32_Shdr * spnt;
    char *header=NULL;
    int  header_size=0;
    int lastmapped;
    int i, j;
    FILE *infile;
    struct stat statbuf;

    infile = fopen (image, "r");

    stat(image,&statbuf);
```

```

if (statbuf.st_size > header_size) {
    header = (char*)realloc(header, statbuf.st_size);
    header_size = statbuf.st_size;
}

memset(header, 0, 4096);
fread(header, statbuf.st_size, 1, infile);
epnt = (Elf32_Ehdr *) header;
if (epnt->e_ident[0] != 0x7f || strncmp(&epnt->e_ident[1], "ELF", 3)) {
    printf("image is not an ELF file\n");
    fclose (infile);
    return -1;
};

printf("Elf file is %s\n", filetype[epnt->e_type]);
printf("Entry point 0x%x\n", epnt->e_entry);
printf("There are %d program headers, starting at offset %x:\n",
    epnt->e_phnum, epnt->e_phoff);

spnt = (Elf32_Shdr *) &header[epnt->e_shoff];
spnt += epnt->e_shstrndx;
lastmapped = spnt->sh_offset;
printf("There are %d section headers, starting at offset %x:\n",
    epnt->e_shnum, epnt->e_shoff);
spnt = (Elf32_Shdr *) &header[epnt->e_shoff];
printf("Section Headers:\n");
printf("[Nr]   Name           Type           Addr       Off       Size   ES Flg   Lk   Inf Al\n");
for(i=0; i < epnt->e_shnum; i++) {
    printf("[%2x] %-10s", i, &header[lastmapped+spnt->sh_name]);
    if(spnt->sh_type < 12)
        printf(" %-12s ", sectiontype[spnt->sh_type]);
    else
        printf(" %8x ", spnt->sh_type);
    printf("%8.8x %8.8x %8.8x %2.2x",
        spnt->sh_addr,
        spnt->sh_offset,
        spnt->sh_size,
        spnt->sh_entsize);

    printf(" %c%c%c %4x %4x %x ",
        (spnt->sh_flags & 1 ? 'W' : ' '),
        (spnt->sh_flags & 2 ? 'A' : ' '),
        (spnt->sh_flags & 4 ? 'X' : ' '),
        spnt->sh_link,
        spnt->sh_info,
        spnt->sh_addralign);
    printf("\n");
    spnt++;
};
fclose (infile);
return 0;
}

```


9.4.3 映像组织


根据前面介绍的各种程序映像的特点和用途, 结合自己系统的硬件配置情况和应用方式, 用户可自由选择目标机程序映像的组织方式。不同映像组织方式, 有不同的启动速度和运行效率。程序映像的组织在程序开发、发布各阶段可能不同。开发时使用的方式应该尽量和发布时近似, 仿真系统在实际运行时环境, 以避免两种不同方式可能带来的问题。所以在系统计划开始, 就应该确定映像组织方式, 这对硬件、软件选择也有影响。

两种基本类型的硬件配置影响映像组织的选择: 存储器形式和文件系统形式。这两种形式不单纯是硬件设计, 也包含了文件系统软件组件的选择。大多数嵌入式系统使用 ROM 芯片 (包括 ROM 扩展类型, 如 Flash 存储器) 来存放存储映像, 映像存储在存储器中以二进制格式存储, 不存在文件的概念。例如对只带一片存储器的系统, 映像方式的选择就有限制, 可以选用 `vxworks_rom` 类的映像, 没有 BootRom 的支持, 系统应用和维护缺少灵活性。系统若有两片存储器, 就有更大的灵活性, 有多种方式可以选择。在第 1 片存储器写入 `vxworks_rom` 类映像, 第 2 片存储器可用于建立 TFFS 文件系统, 存放配置参数、启动脚本和应用目标模块等, 系统运行配置较灵活。或者, 存储器 1 中写入 BootRom 映像, 存储器 2 中写入 `vxworks` 或 `vxworks_rom` 类映像 (一般不必要), BootRom 启动后跳转执行 VxWorks, 这需要对 BootRom 做一些定制修改, 以支持跳转执行和 VxWorks 写入。相对于原始的存储器形式, 文件系统的应用可以带来更大的灵活性和易用性。两者最大的区别是后者一定要 BootRom 支持, VxWorks 映像以文件形式存放。当然后者需要更高的硬件配置。例如 2 片存储器系统, 存储器 1 存放 BootRom, 存储器 2 建立 TFFS 文件系统, 存放 VxWorks (这儿不能使用 `vxworks_rom` 类映像, 因为 VxWorks 文件形式不能保证存储位置和存储连续性, 而 `vxworks_rom` 中的代码要求从存储位置直接执行)、配置参数、启动脚本和应用目标模块等, 这种方式具有高度灵活性, 而且有现成 FTP Server 和 TFFS 组件支持, 减少了 VxWorks 写入存储器代码的修改工作。对于 PC 结构, 这种方式更显得理所当然, 情况与上面类似, 只是 BootRom 位于通常意义的盘上, 由 Vxld 加载运行。要使用 `vxworks.z` 压缩映像, 也需要文件系统支撑。

对上面描述映像类型的选择, 如 ROM 驻留或压缩, 读者可以参考上节的描述。基于存储器的系统最好使用两片 Flash 存储器, 存储器 2 相对存储器 1 要大, 最好有足够的 RAM 支撑。当然上面关于硬件的影响描述不是完全的, 比如 TFFS 文件系统和 BootRom 可共存在同一 ROM 存储器中, 网络支持 VxWorks 映像远程加载等。可以参考相关文档。

即使对于同样硬件配置系统, 也可以选择不同的应用方式。

上面所讲的各种影响因素确定了程序映像组织的选择。可选择的方式多种多样, 不可能一一描述。用户关键要明白映像的格式、类型, 了解自己应用的需求, 根据实际情况选择可用的方式。

 郑更生等, “基于 VxWorks 的产品映像设计”, 电子设计应用, 2003. 04。

9.5 BootRom 建立

要建立 BootRom 首先需要定制 BSP (即修改 BSP 目录的配置头文件和 C 文件), 编译连接, 再写入存储器。BootRom 也称为 bootLoader 或 bootImage, 是 VxWorks 加载、调试、存储的桥梁。

9.5.1 BSP 定制

如前面描述, BSP 是系统的基础, 无论是 BootRom, 还是 VxWorks 都要使用 BSP 代码。BSP 定制需根据用户的实际情况进行, 如硬板配置, 软件组件, 系统设计需求等。

● Bootline

当 VxWorks 系统启动时, 必须指定一些启动参数, 如网络地址、启动设备、主机名和启动脚本等。这些信息包含在称为 Bootline 的字符串中。要定制 BootRom, 首先需了解启动参数的配置。也就是能根据实际系统编写 Bootline, 而经常修改的 config.h 中 DEFAULT_BOOT_LINE 宏就是用来配置启动参数的。

Bootline 为一字符串, 由 bootlib 库中函数 bootStringToStruct 解析, 取得启动参数, 以控制启动过程。Bootline 在系统中有 3 处表现: DEFAULT_BOOT_LINE, NVRAM 中, BOOT_LINE_ADRS。

DEFAULT_BOOT_LINE 为缺省的启动参数串。在 usrBootLineInit 初始化函数中, 若冷启动时从 NVRAM 中未取得有效参数串, 就引用该缺省串, 复制到 RAM 的 BOOT_LINE_ADRS 处, 后续使用。除了缺省串, 还有其他缺省参数, 形如 HOST_NAME_DEFAULT, 在 configall.h 中定义, 由解析函数 usrBootLineParse 在判断相应参数无效时使用。

NVRAM 中 Bootline 为参数串备份, 用于在线修改保存启动参数。如对程序映像中带的缺省串不满意, 可对 Bootline 进行修改, 修改结果存放在 NVRAM 中, 掉电不丢失。当 NVRAM 中串有效时, 优先于缺省串使用。NVRAM 中该串的使用和维护由 sysNvRamGet 和 sysNvRamSet 实现, 这两个函数在 sysLib.c 中定义。一般提供的 BSP 模板中, 这两函数为哑函数, 无具体实现。其实际实现很简单, 就是从 NVRAM 给定地址读取到 BOOT_LINE_ADRS 处, 或将 BOOT_LINE_ADRS 串写入该地址。

程序中实际引用 BOOT_LINE_ADRS 处参数串。该串由 usrBootLineInit 初始化, 来源 NVRAM 串或缺省串。BOOT_LINE_ADRS 在 configAll.h 先定义, 也可在 config.h 和 prjParams.h 中重定义。

bootLib 库提供启动参数相关函数, 如参数串和结构转换, 启动参数显示和修改等。bootConfig.c 和 usrBootLine.c 中就利用这些库函数对 Bootline 进行操作。如 BootRom 的 Shell 中, 命令 p 调用 bootParamsShow 显示 BOOT_LINE_ADRS 处的参数配置, 命令 c 调用 bootParamsPrompt 修改 BOOT_LINE_ADRS 处值并向 NVRAM 中备份。为了替换 Bootline 中目标机 IP 地址, 可使用参数串和结构转换函数。

```
bootStringToStruct(bootString, &myBootParams);
strcpy(myBootParams.ead, sMyIP);
bootStructToString(bootString, &myBootParams);
```

Bootline 串经过解析后, 取得各参数值, 形成启动参数结构。VxWorks 映像的启动参数结构地址存于 sysBootParams 全局变量中, 可由所有程序引用。BootRom 中该参数结构被直接使用。该结构在 bootLib.h 中定义, 如下所示:

```
typedef struct                                /* BOOT_PARAMS */
{
    char bootDev [BOOT_DEV_LEN];              /* boot device code */
    char hostName [BOOT_HOST_LEN];            /* name of host */
}
```

```
char targetName [BOOT_HOST_LEN]; /* name of target */
char ead [BOOT_TARGET_ADDR_LEN]; /* ethernet internet addr */
char bad [BOOT_TARGET_ADDR_LEN]; /* backplane internet addr */
char had [BOOT_ADDR_LEN]; /* host internet addr */
char gad [BOOT_ADDR_LEN]; /* gateway internet addr */
char bootFile [BOOT_FILE_LEN]; /* name of boot file */
char startupScript [BOOT_FILE_LEN]; /* name of startup script file */
char usr [BOOT_USR_LEN]; /* user name */
char passwd [BOOT_PASSWORD_LEN]; /* password */
char other [BOOT_OTHER_LEN]; /* available for applications */
int procNum; /* processor number */
int flags; /* configuration flags */
int unitNum; /* network device unit number */
} BOOT_PARAMS;
```

大多参数有长度限制，用户定制参数时应加以注意。

完整的 Bootline 格式如下：

```
bootdev(unitnum,procnum)hostname:filename e=# b=# h=# g=# u=userid pw=passwd f=#
tn=targetname s=startupscript o=other
```

如表 9-12 所示显示了 Bootline 串实际使用的缩写符号，以及与上述结构的对应关系和描述。

表 9-12

Bootline 参数说明

名称	符号	说明
启动设备	N/A	表示驱动的缩写名，如 fd，ene，motfec，tffs 等
设备号	N/A	启动设备驱动具体的设备号，一般为 0
处理器号	N/A	在 backplane 上的处理器号
主机名	N/A	用于网络启动
文件名	N/A	用于启动的 VxWorks 映像文件名（包含路径）
Ethernet IP	E	目标机上启动 ethernet 接口的 IP 地址（如果不是从网络启动， 但需初始化网络，该 IP 地址则用于 o 项指定的设备）。 地址可以包含掩码，例如 192.168.1.1:ffffff00
Backplane IP	b	backplane 网络上的 IP 地址
主机 IP	h	用于网络启动
网关 IP	g	通往主机的网关
用户名	u	用于网络启动登录主机 FTP 服务器。
FTP 密码	pw	用于网络启动登录主机 FTP 服务器
标志	f	指定启动选项，可参考 sysLib.h 和 BootRom Shell 中的帮助
目标机名	tn	
启动脚本	s	如果内核中包含 tShell，并设置为自动运行脚本，tShell 将执行该启动脚本
其他	o	用来向主内核映像传送其他杂项参数

定制 BSP 最首要的一个工作就是编写正确的缺省参数串，如下所示：

```
#define DEFAULT_BOOT_LINE \
    "motfec(0,0)host:/usr/wpwr/target/config/template/VxWorks " \
    "h=90.0.0.3 e=90.0.0.50 u=username tn=targetname"
```

串定义可以分行写，用“\”续行。

包含等号的赋值项的排列顺序和是否省略都无关紧要。串中各项详细的说明可以参考 bootLib 库帮助。其中启动设备名比较多样，网卡设备更是如此，常见的设备名如表 9-13 所示。


表 9-13


Bootline 启动设备


设备	说明
fd	软盘
ide	IDE 硬盘
ata	ATA 硬盘
ene	NE2000 网卡
fex	Intel 网卡
fei	Intel 82559 EtherExpress 网卡
elt	3COM 以太网卡
elpci	3COM EtherLink XL PCI 网卡
EeV	
motfec	Motorala 芯片 fec 网络接口设备
tffs	Tffs 支持的 Slash 存储器或电子盘设备
tsfs	主机 Target Server 文件系统，使用串口调试时常用

其他设备名可以阅读 bootConfig.c 的 bootLoad 函数中的代码。

 WindRiver, “VxWorks 5.4 Reference Manual” 的 bootLib 条目。

 WindRiver, “Tornado 2.0 User's Guide” 的 2.5.4 章节。

 Drew, “关于 BSP 配置精华”。

 JohnGordan, “VxWorks Cook book”。

● 配置文件

由于 VxWorks 的组件结构以及众多的定制选项，需要通过文件配置。配置文件包括 configAll.h、config.h 和 Makefile 等。

✧ configAll.h

该头文件中包含很多的宏定义，为系统的最基本配置，包括各组件配置，常数定义等。该文件适用于同一 CPU 结构的系列 BSP 版本。另外 VxWorks 工程中使用/comps/ src/ 目录下的 configAll.h。所以对 BSP 定制时，最好不要修改该文件，而应该修改 config.h，用 config.h 中的定义重载 configAll.h 中的定义。

✧ config.h

config.h 为具体 BSP 的配置文件。包含很多与 configAll.h 类似或相同的定义。其中最重要的是 DEFAULT_BOOT_LINE 的定义，各内存地址的定义，以及其他的一些硬件配置定义。该文件由 BootRom 和 VxWorks 共同使用。该文件是最重要的配置文件，对系统影响很大，所以在进行配置修改时要特别小心。

✧ Makefile

位于 BSP 目录下，为 BootRom 映像的编译规则文件，严格说不属于配置文件，但里面有几个内存地址相关的宏定义，如 ROM_TEXT_ADRS、ROM_SIZE、RAM_LOW_ADRS 和

RAM_HIGH_ADRS 等，它们的定义应该与 config.h 文件中一致，否则会出现意外的错误。另外常常需要向 BSP 中加入自己的驱动代码，如网卡驱动和 TFFS 驱动等，这时也需要修改 Makefile，一般是添加 EXTRA_MODULE 定义。如果需要调试 BootRom，还要生成调试信息，需要在 Makefile 中添加编译选项“-g”。

9.5.2 选择 BootRom 的启动方式

本节介绍的前 3 种启动盘制作都是 PC 体系结构相关的，它们在概念和实现上都有共同处，这里先做个总体描述，为后续的具体描述做准备。

为了制作启动盘，开发系统提供很多辅助工具，如表 9-14 所示。

表 9-14 启动盘辅助工具

vxsys.com	将 Vxld 写入启动盘的引导扇区	mkboot.c 中 vxsys 函数
vxcopy.exe	将 BootRom 复制到启动盘	
Vxld	位于盘启动扇区的 bootstrap 加载代码	vxload.com
mkboot.bat	创建启动盘的批处理，使用 vxsys 和 vxcopy	mkboot.c

这些工具大多在 Dos 环境中执行，但 mkboot.c 中定义的函数，如 mkbootFd、mkbootAta 和 mkbootTffs 等，则需要在 VxWorks 系统中执行。


而 Vxld 为纯机器码，不需要 Dos 或 VxWorks 环境的支持，直接由 CPU 执行。在 mkboot.c 中有 Vxld 内容的数组定义（bootstrap），带标志“0x55 0xAA”，一共 450 字节。在制作启动盘不成功时，可以用 Winhex 等工具查看引导扇区的内容是否是 Vxld。vxsys 将 Vxld 写入引导扇区，会破坏原有操作系统的启动，所以操作中应小心。Vxld 搜索根目录下的 bootrom.sys，将其加载到 0x8000 内存，并跳转 0x8000 执行该处的 romInit 函数。但由于大小限制，Vxld 并不是完备的执行文件加载程序，不能解释标准执行文件，没有完善的文件系统支持，只能用于启动初期 BootRom 的加载。Vxld 是专门针对 PC 结构的，别的体系结构则不需要 Vxld 的辅助。Vxld 用 16 位 8088 汇编语言编写。


vxload.com 作为 Vxld 一种替代，用来从 Dos 环境（必须是纯 Dos 环境）加载和执行 VxWorks 或 BootRom。Vxload 可以更方便快捷地加载映像。Vxload 可以传映像文件名作为可选参数，无参数时缺省加载当前目录下的 vxworks.st 映像。纯 Dos 环境下不能使用保护模式程序，如 RAM 盘驱动 vdisk.sys，扩展内存管理器 emm386.exe 等。为了使用 vxload，必须移除或禁止这样的程序。vxload 将映像文件读到 0x8000 开始的内存前，会检查该内存区是否被 Dos 使用，否则会报错。出错时，需要重新配置 PC 让出内存，比如将 Dos 加载到高端内存，减少设备驱动，或者用 vxload 代替 Dos 的命令解释器 command.com。

vxcopy 将 AOUT 执行格式的 BootRom 映像复制到启动盘。vxcopy 会将映像的 32B 的 AOUT 头去掉，并将映像的后面部分连续写入。与 Vxld 不同，BootRom 映像以文件形式存在，只是启动盘上的 bootrom.sys 没有原映像文件的头 32B，其余相同。bootrom.sys 还必须连续写入启动盘中，vxcopy 并不能保证这点，最好制作前先格式化启动盘，写入后再用 chkdsk 检查确认。

mkboot.bat 是启动盘制作过程的批处理文件，综合使用 vxsys、vxcopy 和 chkdsk 等，可

以很方便地制作启动盘。mkboot.bat 只能在 Dos 环境下使用，为了在 VxWorks 系统中也可制作自己的启动盘，mkboot.c 中实现了类似功能的一些函数。阅读 mkboot.bat 和 mkboot.c 对启动盘制作过程的掌握很有帮助。

 WindRiver, “参考 “VxWorks 5.4 Programmer's Guide” 的 D.5 章节。

 Amine, “VxWorks 新手探路系列”, <http://bbs.edw.com.cn>, 2002. 04。

● 软盘启动

完成 BSP 修改后生成 BootRom 映像文件 bootrom_uncmp, 更名为 bootrom.dat。在 Dos 命令行窗口按如下步骤制作启动软盘。

```
mkboot bootrom.dat a:\bootrom.sys
```

或者分步执行：

```
vxsys a:
vxcopy bootrom.dat a:\bootrom.sys
chkdsk a:\bootrom.sys
```

● 硬盘启动

与上面软盘启动制作类似，比如把启动做到 C 盘上，步骤如下：

```
mkboot bootrom.dat c:\bootrom.sys
```

或者分步执行：

```
vxsys c:
vxcopy bootrom.dat c:\bootrom.sys
chkdsk c:\bootrom.sys
```

应将 C 盘格式化为 FAT16 格式，若为 FAT32 格式会出现 GPF 错误。

● 电子盘启动

一般专用于嵌入式系统的 PC104 模块都使用 IDE 接口的电子盘，如 DiskOnChip 等。电子盘的使用和软盘、硬盘类似。制作 BootRom 过程也和上述相同，一般在开发工装上完成。

不过 BootRom 的修改定制需要加入电子盘的 TFFS 驱动。这会在下面的关于 PC104/486 实例中详细介绍。

● ROM 启动

大多数非 PC 结构系统以及一些 PC 都使用 ROM 启动。和前面的启动方式不同，BootRom 直接由编程器写入启动存储芯片，代码直接位于程序空间。启动时，不用从二级存储器加载 BootRom 映像，使启动更加快速。

BootRom 的制作方式随系统结构不同而不同。

● 其他

上面所讲的各种启动方式，其根本就是如何将 BootRom 调入执行空间并启动运行。从这种意义上说，使用 visionProbe、visionIce、BDM 等调试工具，将 BootRom 下载到目标机 RAM 中运行，也算是一种方式。只是没什么实际价值，除了仿真调试 BootRom 自身。

也可以用其他具有 BootRom 类似功能的启动代码来完成 VxWorks 的执行。例如一种最简单的情况，VxWorks 映像以二进制形式烧录在 ROM 的固定位置，启动代码要作的工作就是简单的硬件初始化和跳转执行。VxWorks 接受控制权后几乎会将硬件重新初始化一遍，从

运行角度来看，BootRom 和 VxWorks 是互相独立的。在 Internet 上可以找到很多这种启动平台代码，如 dbug、ppcboot 和 uMon 等，自己需要做些修改，或者干脆自己写一个。

虽然有这些灵活供选择，按通常方式使用 BootRom 启动一般更为简单快捷，选一种最适合的方式也许这才是开发技术人员最想要的。

9.5.3 选择 VxWorks 的加载方式

选好启动方式后，需要确定 VxWorks 的加载方式，主要是修改 Bootline 选择启动设备。参考前面相关的 Bootline 描述。VxWorks 映像由 Bootable 工程生成。前面 3 种加载方式都要在目标机相应存储器建立 DosFs 文件系统。

● 软盘加载

在 config.h 中包含定义。

```
#define INCLUDE_FD
```

缺省 Bootline 的修改类似：

```
#define DEFAULT_BOOT_LINE \  
"fd(0,0)host:/fd0/vxWorks h=90.0.0.3 e=90.0.0.50 u=username tn=targetname"
```

其中，“fd(0,0)”中前一个 0 表示软驱编号：0 表示“A:”；1 表示“B:”。后一个 0 表示该软盘类型：0 表示小盘；1 表示大盘。“fd(0,0)”中“fd”是必须的，不能使用加载根名，bootConfig.c 中 bootLoad 函数根据“fd”串来判断是否要加载 FD 设备和文件系统。

“/fd0/vxWorks”为需要 BootRom 加载的映像文件，其中“fd0”不是必须的，只是选择的设备名称，可以根据喜好任意设定，如“/diska/vxWorks”。

有了前面的 3 个参数，fdLoad 函数就可以调用 usrFdConfig 初始化设备，搜索设备根目录下的 VxWorks 文件，加载映像到 RAM，取得程序入口。BootRom 就可以跳转执行 VxWorks 了。

● 硬盘加载

在 config.h 中包含定义。

```
#define INCLUDE_ATA
```

缺省 Bootline 的修改类似：

```
#define DEFAULT_BOOT_LINE \  
"ata(0,0)host:/ata0/vxWorks h=90.0.0.3 e=90.0.0.50 u=username tn=targetname"
```

“ata(0,0)”中前一个 0 表示硬盘编号，对应 ataResources 表中描述条目。后一个 0 表示该硬盘的分区编号。“ata(0,0)”中“ata”是必须的，不能使用加载根名，bootConfig.c 中 bootLoad 函数根据“ata”串来判断是否要加载 ATA 设备和文件系统。

“/ata0/vxWorks”为需要 BootRom 加载的映像文件，其中“ata0”不是必须的，只是选择的设备名称，可以根据喜好任意设定，如“/diskc/vxWorks”。

有了前面的 3 个参数，ataLoad 就可以调用 usrAtaConfig 初始化设备，搜索设备根目录下的 VxWorks 文件，加载映像到 RAM，取得程序入口。BootRom 就可以跳转执行 VxWorks。

● Tffs 盘加载

TFFS 盘包括电子盘和 Flash 存储器。在 config.h 中包含定义。

```
#define INCLUDE_TFFS
```

缺省 Bootline 的修改类似：

```
#define DEFAULT_BOOT_LINE \
"tffs(0,0)host:/tffs0/vxWorks h=90.0.0.3 e=90.0.0.50 u=username tn=targetname"
```

“tffs(0,0)”中前一个 0 表示驱动编号，从 0 到 “noOfDrives-1”，全局变量 noOfDrives 在库中定义，在 xxxRegister 调用中加 1，表示已注册 TFFS 的个数。后一个 0 表示 TFFS 设备不可移除。“tffs(0,0)”中 “tffs” 是必须的，不能使用加载根名，bootConfig.c 中 bootLoad 函数根据 “tffs” 串来判断是否要加载 TFFS 设备和文件系统。

“/tffs0/vxWorks”为需要 BootRom 加载的映像文件，其中 “tffs0” 不是必须的，只是选择的设备名称，可以根据喜好任意设定，如 “/flash/vxWorks”。

有了前面的 3 个参数，tffsLoad 就可以调用 usrTffsConfig 初始化设备，搜索设备根目录下的 VxWorks 文件，加载映像到 RAM，取得程序入口。BootRom 就可以跳转执行 VxWorks。

● 串口加载

BootRom 通过串口线可以使用 TSFS 来访问主机文件系统，所以 VxWorks 映像可以存放在主机上，启动时可通过串口加载该映像。配置和使用 TSFS 比 PPP 或 SLIP 更简单。

在启动目标前，应该先在主机运行带 TSFS 选项的 Target Server。

✧ 配置 config.h。

```
#define DEFAULT_BOOT_LINE \
"tsfs(0,0)host: vxWorks u=username tn=targetname"
```

最好使用相对路径，方便多工程调试和工程目录更换，绝对路径可以在 Target Server 配置时指定。

```
#undef WDB_TTY_DEV_NAME
#define WDB_TTY_DEV_NAME "/tyCo/0"
#undef CONSOLE_TTY
#define CONSOLE_TTY NONE
/*如果目标机有多个串口，另外串口可作为 console 使用 shell*/
/*#define CONSOLE_TTY 1*/
#undef WDB_TTY_CHANNEL
#define WDB_TTY_CHANNEL 0
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_SERIAL
#define WDB_TTY_BAUD 38400
#define INCLUDE_TSFS_BOOT
/*提供 Target Sever Console，对单串口目标机特别有用，可以修改启动参数*/
#define INCLUDE_TSFS_BOOT_VIO_CONSOLE
#undef INCLUDE_WDB_TSFS
#define INCLUDE_WDB_TSFS
```

✧ 配置 Target Server

选择 Backend 为 wdbserial，选择主机串行端口，波特率设置应该和 WDB_TTY_BAUD 设置相同。“Target Server File System”下使能文件系统，选择主机目录为 TSFS 根目录。课余参考第 2 章中关于 TSFS 的描述。

串口加载还有一种方法：配置串口为 SLIP 接口，就可用网络加载了。不过使用 SLIP 比 TSFS 更麻烦。

 WindRiver, “Tornado 2.0 User's Guide” 的 2.5.7, 4.7 章节。

 WindRiver, “VxWorks 5.4 Programmer's Guide” 的 4.7 章节。

● 网络加载

主机上需要启动适当的服务程序,如 FTP、TFTP 或 RSH(Unix),目标机用来加载 VxWorks 映像。

网络加载比前面几种方式复杂点,因为有太多的网络接口类型。需要加载一些非标准网口驱动,启动设备名也多种多样。不过配置过程都是类似的。

最好使用相对路径,方便多工程调试和工程目录更换,绝对路径可以由 FTP 服务器等来设定。

9.5.4 BootRom 建立

建立成功的 BootRom 是建立开发环境的最为关键的一步。有两种标准方法来完成这个工作,通过“build boot rom”图形对话框或命令行方式。

如图 9-6 所示对话框由“build”中菜单激活。只有安装时选择了与 Tornado 1.0 兼容项,该菜单才会出现,参考第 2 章中安装部分的描述。在左框选择所用的 BSP,右面选择生成 BootRom 的具体类型。点击“OK”就会得到需要的映像文件。

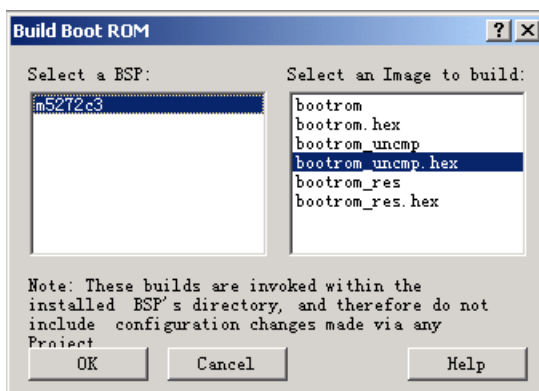


图 9-6 BootRom 建立

命令行编译相比不太直观,不过笔者更喜欢。进入 Dos 命令行,运行环境变量批处理文件 torvars.bat (一般在“\Tornado\host\x86-win32\bin\”目录下,也可复制到别处运行),建立所需要的编译环境;切换到对应的 BSP 目录,输入如下命令,即可得到与上面方法相同的映像文件。

```
make bootrom_uncmp.hex
```

映像格式的选择和用户实际系统的需要相关,也和是否用于调试相关。

生成的 BootRom 映像文件位于相应 BSP 目录下,然后使用适当的工具和设备将该映像写入启动存储设备,中间可能需要进行映像文件的格式转换。如果顺利的话,就得到第 1 个 VxWorks 运行系统,开发环境的建立工作就完成了大部分。

9.5.5 BootRom 运行

当目标机上电或复位,系统都从 BootRom 开始运行。在启动过程中,主机通过 Console 与目标机交流。BootRom 先在 console 上显示启动信息[banner],然后延时等待 7s (缺省值,可以修改 bootConfig.c 中的 TIMEOUT 宏定义,而快速启动复位固定等待 1s),如果这期间用户没有击键干预,BootRom 就立即自动加载 VxWorks 映像;若用户干预或自动加载失败,

BootRom 就进入命令行处理状态，等待执行相关用户命令，直到用户明确命令 BootRom 再次加载执行 VxWorks。

判断 BootRom 是否成功建立的第一个标志，就是能否在 console 上看见 BootRom 的启动信息。

```
DSXXXX System Boot

Copyright 2002-2005 SAC, Inc.
Copyright 1984-1999 Wind River Systems, Inc.

CPU: SAC MFC5272
Version: 5.4
BSP version: 1.2/0
Creation date: Aug 25 2003, 16:49:02

Attaching to TFFS... done.
Attached TCP/IP interface to motfec0.
Ok setting inet address of motfec0 to 192.166.0.4
Attaching network interface lo0... done.
Ok setting inet address of lo0 to 127.0.0.1

Press any key to stop auto-boot...
2
[VxWorks Boot]:
```

显示信息用户可以自己定制， 根据自己需要输出信息。修改 `printBootLogo` 函数，可改变版权信息、平台类型和 BSP 版本等。也可在 BootRom 代码中添加自己需要的信息输出，如上面的 IP 地址显示。映像创建时间值存放在程序映像中，可以作为 BootRom 程序版本的重要标志。时间串 `creationDate` 在 `version.c` 中定义，也可以在 VxWorks 程序中使用，以标示映像生成时间。这些显示可用来确定 BootRom 是否有问题，如网络驱动或 TFFS 驱动是否有问题。

在 BootRom 的 Shell 中可以执行命令完成一些基础功能。可以在 Shell 中用 “?” 或 “h” 命令获取各命令的帮助。用户也可定制添加自己的命令，如选择其他的加载方式，TFFS 格式化和 MAC 地址读写等。在 `bootCmdLoop` 中添加命令入口，并在 `bootHelp` 的 `helpMsg` 数组中添加相应命令项的帮助描述。需要注意命令都是单字符的，影响了添加命令的直观性。

若 BootRom 运行正常，要想继续加载 VxWorks 映像就必须保证启动参数正确。这些参数由前面描述的 Bootline 确定，也可在 BootRom 命令行对这些参数作修改调整。可以用 “c” 命令逐项修改参数，也可以用 “\$” 整行替换。BootRom 在加载 VxWorks 时或使用 p 命令时会显示这些参数，如下所示：

```
boot device      : motfec
unit number     : 0
processor number : 0
host name       : host
file name       : vxWorks
inet on ethernet (e) : 192.166.0.7
host inet (h)   : 192.166.0.254
user (u)        : dsxxxx
ftp password (pw) : dsxxxx
```

flags (f)	: 0x0
target name (tn)	: mcf5272

9.6 MCF5272 BootRom 实例

下面两节介绍两个 BootRom 建立的实例，分别针对 MFC5272 和 PC104/486 平台。为了保持平台建立过程的完整性，描述可能和前面章节有所重复。

本节详细描述基于 MFC5272 系统上的 BootRom 建立过程。该实例同样可以用做类似 CPU 系统的参考，如 PPC、MC68K 和 ARM 等。

9.6.1 目标系统

该硬件平台采用 MFC5272（66M）。除了 CPU 外，系统核心硬件还包括 RAM 储存器，Flash 储存器等。芯片型号和地址分配如表 9-15 所示，各芯片的基址是人为设定的。

表 9-15 MFC5272 平台配置

片选	芯片	地址范围
CS0（启动片选）	Flash 存储器 1（Am29lv160DB，2MB）	0xFFC00000~0xFFE00000
CS1	Flash 存储器 2（Am29lv160DB，2MB）	0xFFE00000~0xFFFFFFFF
CS7	RAM 储存器（Hy57V641620HG，16MB）	0x00000000~0x01000000

9.6.2 主机环境

主机操作系统为 Windows 2000 Professional。使用 Tornado 中的 GNU C/C++编译器，以及 WindRiver 额外出售的硬件调试器 visionProbe for Coldfire 和软件环境 VisionClick。

● VisionClick 使用

安装 VisionClick 后，首先需要建立“Active Project”，如图 9-7 所示，该窗口可选择菜单“File→Open Projects Files...”命令激活。

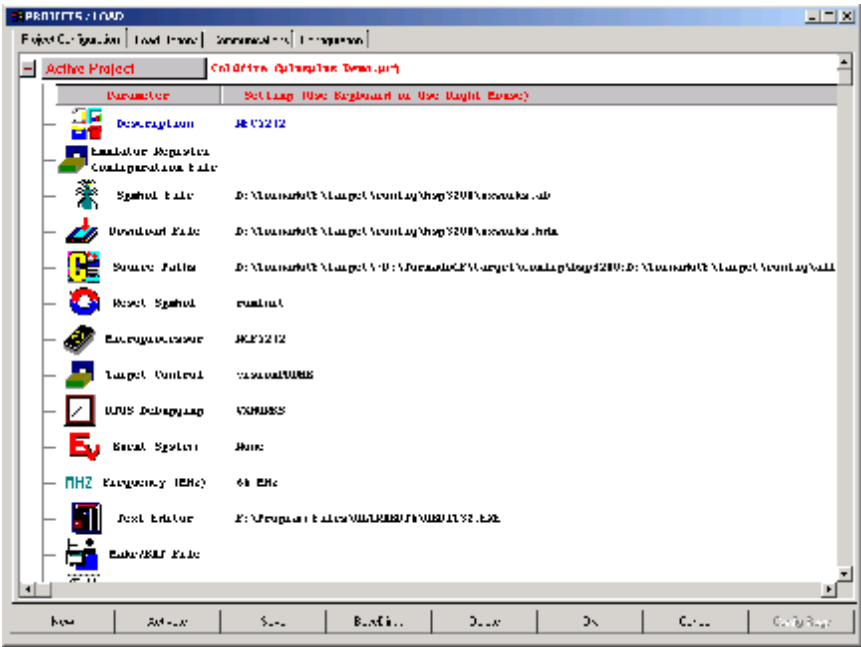


图 9-7 VisionClick 工程配置

如图 9-7 所示的具体设置选项如表 9-16 所示。

表 9-16

VisionClick 工程参数

参数	说明	值
Description	工程描述	MCF5272
Symbol File	符号表文件，用于源代码调试	xxxxx.ab
Download File	下载映像文件	xxxxx.bdx
Source Paths	源代码文件目录	加入 ALL 目录和 BSP 目录
Reset Symbol	复位符号，确定起始 PC 地址	romInit
Microprocessor	选择目标 CPU	MCF5272
Target Control	选择 BDM 硬件头	visionProbe
RTOS Debugging	选择使用的 RTOS	VXWORKS
Frequency	选择 CPU 工作频率	66MHz
Text Editor	连接外部编辑器	
Sim Register File	配置 CPU 初始状态的寄存器文件	xxxxx.reg
Starting Stack Address(highest)	栈高地址	0x200000
Ending Stack Address(lowest)	栈低地址	0

符号文件和下载文件由 convert 工具转换 bootrom_uncmp 得来，sim 寄存器文件需要根据硬板配置手动编写。栈高地址的配置和 BSP 目录中 config.h 的 RAM_HIGH_ADRS 一致，栈低地址可以简单地设置为 0，因为整个 BootRom 在 RAM_HIGH_ADRS 之上。

从图 9-7 所示窗口切换到如图 9-8 所示窗口，设置符号文件和下载文件的路径，与前面工程属性中的配置相同。

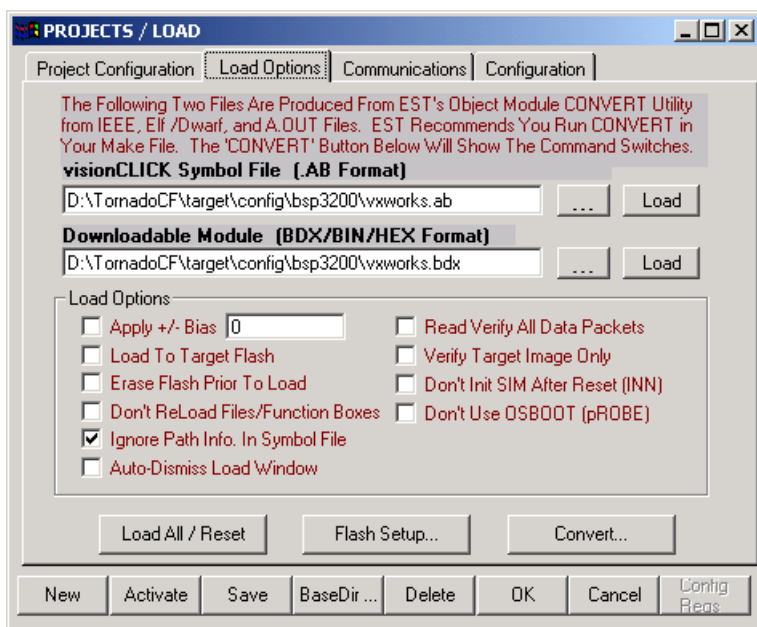


图 9-8 VisionProbe 端口配置

VisionProbe 硬件头连接在组件并口上，配置端口为 LPT1，如图 9-9 所示。主机并口自身也需要设置，以和硬件头配合。配置并口为 ECP 模式，并接受所有中断。如图 9-10 所示，该窗口从“控制面板/系统/硬件/设备管理器”中激活。

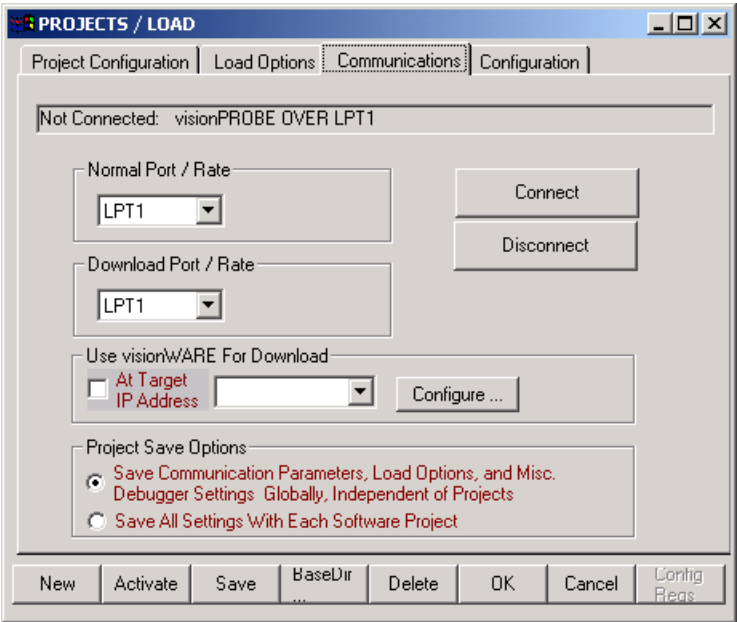


图 9-9 VisionProbe 端口配置

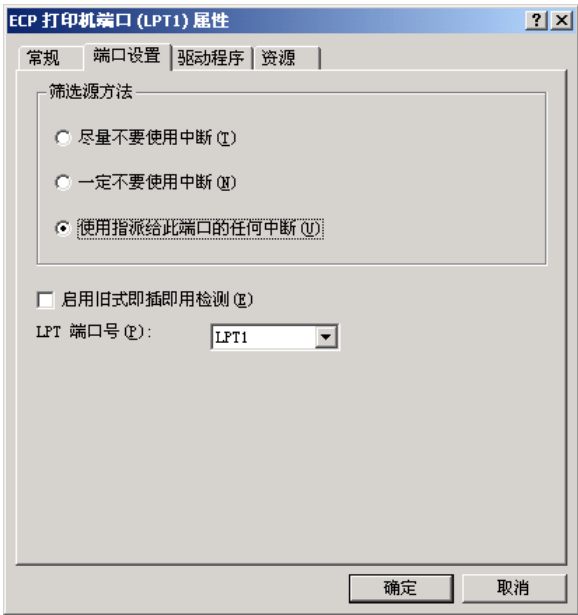


图 9-10 主机并口配置

另外一个重要的工作是配置 sim 寄存器文件。VisionProbe 和目标机连接时，使用该文件来初始化目标机 CPU 寄存器。寄存器配置需根据硬板情况（主要是内存）和 CPU 手册来进行。下面为该例子中使用的寄存器文件，手动编写完成，主要包含 CPU 核心寄存器、片选寄存器和 DRAM 寄存器的配置值，和前面的硬件描述一致。

SC GRP 000F			
SC MBAR	00000C0F	10000001	SIMR
SC RAMBAR	00000C04	20000001	SIMR
SC ROMBAR	00000C00	28000001	SIMR
SC ICR1	10000020	00000000	SIMR
SC ICR2	10000024	00000000	SIMR
SC ICR3	10000028	00000000	SIMR
SC ICR4	1000002C	00000000	SIMR
SC PITR	10000034	00000000	SIMR
SC PIWR	10000038	00000000	SIMR
SC PIVR	1000003F	0F	SIMR
SC CSBR0	10000040	FFC00201	CHIP SELECT 0
SC CSOR0	10000044	FFE00014	CHIP SELECT 0
SC CSBR1	10000048	FFE00201	CHIP SELECT 1
SC CSOR1	1000004C	FFE00014	CHIP SELECT 1
SC CSBR2	10000050	00000000	CHIP SELECT 2
SC CSOR2	10000054	00000000	CHIP SELECT 2
SC CSBR3	10000058	00000000	CHIP SELECT 3
SC CSOR3	1000005C	00000000	CHIP SELECT 3
SC CSBR4	10000060	00000000	CHIP SELECT 4
SC CSOR4	10000064	00000000	CHIP SELECT 4
SC CSBR5	10000068	00000000	CHIP SELECT 5
SC CSOR5	1000006C	00000000	CHIP SELECT 5
SC CSBR6	10000070	00000000	CHIP SELECT 6
SC CSOR6	10000074	00000000	CHIP SELECT 6
SC CSBR7	10000078	00000701	CHIP SELECT 7
SC CSOR7	1000007C	FFC0007C	CHIP SELECT 7
SC SDCTR	10000184	0000F539	SDRAM
SC SDCCR	10000180	00004213	SDRAM

这些寄存器配置在硬件头 VisionProbe[Emulator]中也存在备份，在修改寄存器文件后，需要选择菜单“Tools→Upload&download register files→Download register value from file to emulator”命令将参数重新下载到 VisionProbe 中。

VisionClick 安装配置完成后，最好备份工程文件 Active.prj 和寄存器文件，以便重新安装。当然，最好备份整个安装目录，重装系统后再安装 VisionClick，使用老目录覆盖新安装目录即可。另外，别忘了重新设置并口属性。VisionClick 的功能很丰富，如硬件故障探测等，需要用户自己在实际使用中摸索。

9.6.3 设计目标

BootRom 设计目标如下。

- ✧ 在第 1 片 Flash 存储器中存放 BootRom 程序映像，由硬件调试器写入 Flash 存储器。由于 visionProbe 价格昂贵，在加工生成线上批量生产时，只能使用华恒的硬件调试器，这要求额外的软硬件工作。
- ✧ 在第 2 片 Flash 存储器上建立 TFFS+DosFs 文件系统，用于存储应用程序映像文件，模块目标文件，参数文件等。需要添加 TFFS 格式化命令。
- ✧ 在第 1 片 Flash 的后半部分加载 TFFS。
- ✧ 启动 FTP 服务器，通过它可操作 TFFS 文件系统

- ✧ 简化应用程序工作，辅助配置系统组件。基于该 BootRom 的 BSP 创建 VxWorks 工程时，会自动引用 BootRom 中的组件配置，所以最好在 BootRom 中完成与主应用一致的配置。
- ✧ 可选程序加载方式，方便调试和程序发布。由于 TFFS 文件系统容量有限，不能存放带调试信息的应用程序映像，所以调试时，一般从网络加载。而程序发布时，将去掉调试信息的程序映像存放在 TFFS 中，从 TFFS 加载。两个加载命令，“#” 直接从网络加载，“@” 从 TFFS 上加载。
- ✧ 使用网络调试。
- ✧ RAM 存储器分配。BootRom 加载到地址 0x200000, SP 在 0x1000；应用程序加载到地址 0x1000。
- ✧ 只有空格键才能在启动过程中进入 BootRom Shell。因为系统启动时，串口可能有数据进入，导致启动异常中断。
- ✧ 减少 BootRom 自启动等待时间，加速启动过程。
- ✧ 读取和设置 MAC 地址。
- ✧ 读取应用参数文件的 IP 地址，保证 BootRom 和 VxWorks 两者 IP 一致，以防止多目标系统中启动冲突。
- ✧ BootRom 要具有自更新能力，能脱离 VisionClick 环境升级 Flash 程序。

9.6.4 编辑

下面按逻辑顺序进行说明，对修改代码不做精确的描述。因为读者使用不同的硬件平台或软件版本，讲得太过精细反而不好，主要着重步骤和原理的描述。如果需要更精细的描述，可参考笔者在网上发布的一篇文章“MFC5272_BOOTROM 开发笔记”。

● Makefile

针对特定的目标系统，首先需要修改的文件就是 BSP 目录下的 Makefile。

为了与原来 BSP 目录区分，备份原安装系统，避免和其他的 BSP 混淆，建立与自己项目相关的 BSP 和 ALL 目录，如 bspxxxx 和 allxxxx。再将原 BSP 和 ALL 目录下的文件复制到对应新目录，后续对 BSP 的修改就只在新目录中进行。

对 Makefile 进行如下修改。

```
TARGET_DIR    = bspxxxx
VENDOR        = SAC
BOARD         = dsxxxx

CONFIG_ALL    = ..\allxxxx #指向自己的 ALL 目录

#为了在 VisionClick 中看见源代码，添加调试编译选项-g
ADDED_CFLAGS  = -g

#将新加入 BSP 目录下的源文件，编译入 BootRom 影像
#加入相应的目标文件名，make 会自动识别依赖关系，并编译入程序
#根据该 BSP 创建工程时，这几个附加模块会自动加入（注意文件名大小写一致）
# sysTffs.o lv160mtd.o 用于 TFFS 文件系统建立，ftplib.o 用于 ftp 服务器建立
MACH_EXTRA    = sysTffs.o lv160mtd.o ftpdlib.o

#根据 ROM 地址设置入口地址，为 ROM 地址加 8，前 8 字节由 SP，PC 两指针占据
```

```
ROM_TEXT_ADRS = ffc00008 # ROM entry address

#根据 Flash 存储器 1 的基地址设置 set-start, 以生成绝对地址映像
NO_VMA_FLAGS  = --ignore-vma --set-start=0xffc00000 \
VMA_FLAGS= --ignore-vma --set-start=0xffc00000
```

● config.h

对于 BSP 的定制配置最好在 config.h 中进行, 不要修改 configAll.h。

config.h 中最重要的就是对缺省 Bootline 的配置, 关于 Bootline 组成在前面已做描述, 这里就只针对项目特殊处加以说明。

```
#define DEFAULT_BOOT_LINE \
    "motfec(0,0)host: vxWorks " \
    "h=192.166.0.254 e=192.166.0.2 u=dsxxxx pw=dsxxxx "
```

- ✧ Bootline 中缺省使用 motfec 作为启动设备, 这主要是利用 BootRom 中对启动网络设备初始化的代码。而实际上笔者对 bootConfig.c 中代码做了修改, 启动时先查找 TFFS 上是否有 VxWorks 映像, 如果没有才从网络加载。也许这样可能不是很好, 因为对网络初试化代码做了较多修改。有另一种方法, 使用 TFFS 为启动设备, 在 Bootline 中添加 “o=motfec” 来初始化网络设备。
- ✧ 从主机加载 VxWorks 映像, 需要指明主机地址 “h=192.166.0.254”, 以使目标机连接指定主机上的 FTP 服务器。登录主机 FTP 服务器的用户名和密码由 “u=dsxxxx” 和 “pw=dsxxxx” 指定。
- ✧ 目标机 IP 地址 “e=192.166.0.2”, 在系统正常启动时已经不再有用, 笔者使用应用参数文件中的 IP 替换, 以方便 IP 修改和避免启动冲突。该参数只在裸机启动时有用。

config.h 中内存相关配置如下。

```
#根据 SDRAM 容量设置 LOCAL_MEM_SIZE
#define LOCAL_MEM_SIZE      0x01000000    /* amine:16M, 4Meg memory is default */

#根据片选参数设置 ROM 基地址
#define ROM_BASE_ADRS      0xffc00000    /* amine:ffe, base address of ROM */

#根据 Flash 存储器大小配置 ROM 容量, 只是 BootRom 程序调入 RAM 用
#应用程序从文件系统加载 elf 文件, 在 RAM 中重定位
#define ROM_SIZE            0x00200000    /* amine:2M, 1Meg ROM space */

#设置 CPU 片上 RAM 基址, 防止与 SDRAM 地址范围重叠
#define INTERNAL_SRAM_BASE  0x20000000    /*amine:0x400000, base of internal SRAM */

#定义区分 Probe 仿真和 Flash 写入, 在 romInit.s 中使用
#undef RAM_SIM              /*用于 Probe 仿真调试*/
```

为了加载 TFFS, 需要包括相关的系统组件。INCLUDE_MTD_LV160 为自定义, 包括笔者自己编写的 Flash 存储器驱动。依赖组件可以从工程组件配置时得到。

```
#define INCLUDE_TFFS
#define INCLUDE_MTD_LV160

#define INCLUDE_SHOW_ROUTINES    /* optional */
#define INCLUDE_DOSFS            /*DOS filesystem backward-compatibility*/
```



```

#define INCLUDE_CBIO          /*CBIO (Cached Block I/O) Support, cbioLib*/
#define INCLUDE_DOSFS_CHKDSK /*DOS File System Consistency Checker*/
#define INCLUDE_DOSFS_FAT    /*DOS File System FAT12/16/32 Handler*/
#define INCLUDE_DOSFS_FMT    /*DOS File System Volume Formatter*/
#define INCLUDE_DOSFS_MAIN   /*DOSFS2 File System Primary Module*/
#define INCLUDE_DISK_CACHE   /*Disk Cache Handler*/
#define SELECT_DOSFS_DIR     /*DOS File System Directory Handlers*/
#define INCLUDE_DOSFS_DIR_FIXED /*DOS File System Old Directory Format Handler*/
#define INCLUDE_DOSFS_DIR_VFAT /*DOS File System VFAT Directory Handler*/
#define INCLUDE_POSIX_CLOCKS /*POSIX clocks*/

```

加载 FTP 服务器的组件。

```
#define INCLUDE_FTP_SERVER
```

指定网络调试方式。

```

#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE WDB_COMM_END /* default path is the network */
#define INCLUDE_WDB_SYS /*支持应用程序的系统级调试*/

```

为应用程序配置组件，方便该 BSP 上不同应用工程的创建。

#为应用程序加载 target debugging, 使 target shell 具备部分调试功能

```

#define INCLUDE_DEBUG
#define INCLUDE_SHELL
#define INCLUDE_CPLUS_DEMANGLER
#define INCLUDE_LOADER
#define INCLUDE_HW_FP_SHOW
#define INCLUDE_TASK_SHOW

```

#为应用程序加载 built-in 符号表，方便 target shell 调试

#会增加应用程序的大小，可只在开发中使用，发布程序时可去掉

```

#define INCLUDE_STANDALONE_SYM_TBL
#define INCLUDE_SYM_TBL_INIT
#define SELECT_SYM_TBL_INIT

```

#使应用程序可以加载 telnet 服务器

```
#define INCLUDE_TELNET
```

#去掉应用程序一些无用组件，以减少程序大小。有些组件不能去除，创建工程时会自动添加

```

#undef INCLUDE_BOOTP /* bootp */
#undef INCLUDE_PROXY_CLIENT /* proxy arp client (Slave Board) */
#undef INCLUDE_TFTP_CLIENT /* tftp client */
#undef INCLUDE_NET_HOST_SETUP
#undef INCLUDE_NET_REM_IO
#undef INCLUDE_ARP_API
#undef INCLUDE_HOST_TBL

```

● romInit.s

在该文件中，需根据目标硬板的内存配置做相应修改，为系统启动做好基础准备。其中最重要的是 RAM 的配置。第 2 片 Flash 作为辅助存储器，用于建立 TFFS 文件系统，它的初始化也可延后进行。

文件最后的 romStart 跳转语句，需考虑调试和实际运行的区别。实际运行时，计算出来的跳转地址应该位于第 1 片 Flash 存储空间中。而仿真调试时，BootRom 经过 visionProbe 直接放在 RAM 空间中，所以跳转地址应该在 RAM 中。

文件 `romInit.s` 中相关修改如下。

```
/*添加 FLASH2 的片选 CS1 配置, 参照 CS0 的配置,
只需修改变换寄存器名称, 和 Flash2 的起始地址就行*/
/* program /CS1 */
movel    #(ROM_BASE_ADRS + 0x200000 + \
          M5272_CS_CSBR_EBI_1632 + \
          M5272_CS_CSBR_BW_WORD + \
          M5272_CS_CSBR_ENABLE), d0
movel    d0, M5272_CS_CSBR1(SIM_BASE)

movel    #(M5272_CS_CSOR_BAM_2M + \
          M5272_CS_CSOR_WS(5)), d0
movel    d0, M5272_CS_CSOR1(SIM_BASE)

/*扩展 RAM 片选容量, 由缺省的 4M 改为 16M*/
movel    #(M5272_CS_CSOR_BAM_16M + \
          M5272_CS_CSOR_WS(0x1f)), d0
movel    d0, M5272_CS_CSOR7(SIM_BASE)

/*直接使用 romStart 的 RAM 地址跳转, 而不用 ROM_TEXT_ADRS 相对偏移*/
movel    #_romStart, a0
#ifdef RAM_SIM
    subl #_romInit, a0
    addl #ROM_TEXT_ADRS, a0
#endif
#endif
```

● TFFS

TFFS相关的一些概念性说明, 如分层, 和dosFs的关系等, 读者可以参考“文件系统”一章。下面主要介绍该实例中的实践操作。

首先需要实现Socket层, 即`sysTffs.c`。根据硬板的实际情况, 在“`\drv\tffs\sockets`”目录下选择合适模板文件, 笔者选择`mv177-sysTffs.c`。将其复制到BSP目录下, 更名为`sysTffs.c`。并在`Makefile`中的`MACH_EXTRA`下添加新目标`sysTffs.o`。`sysTffs.c`需要做的修改很少, 如下所示:

```
#define INCLUDE_MTD_LV160
    包括自己的 MTD 代码, undef 其他类型 MTD

#undef INCLUDE_TFFS_BOOT_IMAGE /* include tffsBootImagePut() */
    boot image 与 TFFS 使用不同 Flash 芯片, 此功能不需要

#define FLASH_BASE_ADRS      (0xFFE10000)
#define FLASH_SIZE           (0x001F0000)
    根据 Flash 地址和容量设置, flash 地址为 0xFFE00000,
    跳过前面 64K 不规则扇区, 简化 mtd 代码编写

rfaWriteProtect(), rfaWriteEnable()
    屏蔽这两函数的操作代码, 使其为空函数
```

第2步需要实现MTD层, 和具体的Flash存储器型号密切相关。同样需要选择近似的模板文件, 笔者选择`amdmt.c`, 复制到BSP目录下, 更名为自己的文件名, 如`lv160mtd.c`, 并

修改 Makefile 文件。在这个模板文件之上，需要参考 Flash 芯片手册进行修改，如修改命令码、命令地址（16 位端口的命令地址应该按手册命令地址×2）等。再就是修改各实现函数。读操作使用系统缺省提供的函数。写操作函数比较简单，参照手册很容易实现。擦除函数和虚拟块设置的大小有关，一般为 64KB，一次擦除一个物理扇区，小扇区 Flash 会连续擦除多个块。

在驱动的调试过程中，取地址的 map 函数给笔者带来最大困难，tffsDevFormat 总是失败。系统缺省提供的 map 函数，取地址时会间隔跳过扇区而出错。笔者重写了 map 函数，并将函数指针传回 Socket 层，问题解决。在调试出现问题时，最好将该文件的函数入口参数打印出来，可以帮助解决问题。

为了联系 Socket 层和 MTD 层，还需要简单修改 tffsConfig.c。将 “\target\src\drv\tffs\” 目录下的 tffsConfig.c 复制到 BSP 目录，并修改 sysTffs.c 中包含的路径。在 tffsConfig.c 中的 mtdTable 添加芯片辨识函数，以在初始化时注册 MTD。

剩下的工作就是启动时加载 TFFS 文件系统。因为启动设备为网口 motfec，所以 TFFS 初始化就直接在 bootConfig.c 的 bootCmdLoop 函数开始处添加代码完成，如下所示：

```
/*初始化加载 TFFS*/
#ifdef INCLUDE_TFFS
    if (tffsDrv () != OK)
        printf ("tffsDrv fail!\n");
    printf ("Attaching to TFFS... ");
    dosFsInit (NUM_DOSFS_FILES);          /* initialize DOS-FS */
    if (usrTffsConfig (0, 0, "/dsxxxx/vxworks") == ERROR)
        printf ("usrTffsConfig failed.\n");
    printf ("done.\n");
#endif
```

初始化中，首先要保证 tffsDrv 调用正确，而 usrTffsConfig 调用前必须先完成格式化。tffsDrv 成功后，就可以调用 tffsDevFormat 来完成格式化，而不用先创建块设备和加载 Dos 文件系统。一般格式化成功后，TFFS 驱动修改就基本完成了。最好在 BootRom Shell 中添加 TFFS 格式化的命令，以用于文件系统异常恢复。注意，tffsDevFormat 对已经格式化的 TFFS 操作时，不是很彻底，有时连 Flash 擦除都不进行，所以最好再加个低级格式化 Flash 命令，将 Flash 各扇区硬性擦除。成功调用 dosFSInit 和 usrTffsConfig 之后，TFFS 文件系统就可用了。操作路径在 usrTffsConfig 中指定，如前面的 “/dsxxxx/”。

✧ 在第 1 片 Flash (bootFlash) 后面空闲部分上加载 TFFS 文件系统 “/data/”

bootFlash 与第 2 片 Flash 型号相同，两者地址不连续。bootflash 起始地址为 0xFFC00000，保留 512KB 用于 BootRom，剩余 512X3KB 为文件系统，起始地址为 0xFFC80000，大小为 0x00180000，目录名称为 “/data/”。FTP 登录时指定使用路径 “/data/”，缺省路径登录还是原 TFFS 盘。

在 sysTffs.c 中进行如下修改。

```
/*添加起始地址和大小宏定义*/
#define FLASH_BASE_ADRS          (0xFFE10000)
#define FLASH_SIZE                (0x001F0000)
```

```

#define FLASH_BOOT_ADRS          (0xFFC80000)
#define FLASH_BOOT_SIZE          (0x00180000)
/*在 sysTffsInit() 中增加一个 socket 注册*/
LOCAL void sysTffsInit (void)
{
    rfaRegister ();
    rfaRegister ();
}
/*根据注册先后区分 socket，利用 noOfDrives 和 FLSocket 的 serialNo，修改 rfaRegister()*/
vol.serialNo = noOfDrives;
if (noOfDrives == 0)
    vol.window.baseAddress = FLASH_BASE_ADRS >> 12;
else if(noOfDrives == 1)
    vol.window.baseAddress = FLASH_BOOT_ADRS >> 12;
noOfDrives++;

/*修改 rfaSetWindow() 函数*/
if (vol.serialNo == 0){
    vol.window.baseAddress = FLASH_BASE_ADRS >> 12;
    flSetWindowSize (&vol, FLASH_SIZE >> 12);
}
else if(vol.serialNo == 1){
    vol.window.baseAddress = FLASH_BOOT_ADRS >> 12;
    flSetWindowSize (&vol, FLASH_BOOT_SIZE >> 12);
}

```

在 lv160mtd.c 中进行如下修改。

```

/*添加同 sysTffs.c 中的起始地址和大小宏定义*/
/*增加一个地址映射回调函数，因为不能根据 vol->socket->serialNo 来分支判断*/
static void FAR0 *lv160MTDMap1(FLFlash *vol, CardAddress addr, int length)
{
    UINT32 ret;
    ret = FLASH_BOOT_ADRS + addr;
    return (void FAR0 *)ret;
}
/*修改 lv160MTDIdentify 函数，根据 vol.socket->serialNo 分支*/
if (vol.socket->serialNo == 0){
    flSetWindowSize(vol.socket, FLASH_SIZE>>12);
    vol.chipSize = FLASH_SIZE;
    vol.map = lv160MTDMap0;
}
else if(vol.socket->serialNo == 1){
    flSetWindowSize(vol.socket, FLASH_BOOT_SIZE>>12);
    vol.chipSize = FLASH_BOOT_SIZE;
    vol.map = lv160MTDMap1;
}

```

在 bootConfig.c 中添加如下代码，用于加载 “/data/” 文件系统。VxWorks 应用代码也应添加类似代码。

```

/*修改 bootCmdLoop(), 添加类似的 usrTffsConfig 调用*/
printf (“Attaching to /data/... ”);

```

```

if (usrTffsConfig (1, 0, "/data/vxworks") == ERROR)
    printf ("usrTffsConfig failed.\n");
else
    printf ("done.\n");

```

添加该文件系统格式化命令，调用 `tffsDevFormat`，代码如下。

```

/*修改 bootCmdLoop()*/
case 'z':      /* /dsxxxx/ format */
if( tffsDevFormat(0,0) != OK )
    printf ("tffsDevFormat /dsxxxx/ fail!\n");
break;
case 'x':      /* /data/ format */
if( tffsDevFormat(1,0) != OK )
    printf ("tffsDevFormat /data/ fail!\n");
break;

```

● FTP 服务器

为了实现设计目标，目标机需启动 FTP 服务来操作 TFFS 文件系统，完成程序映像和参数文件维护。系统库中本来自带有 FTP 服务器的实现库，但版本太老，在实际应用中会有一些问题，如用 FTP 工具连接后，不能看见文件列表，但是可上传文件；而用 Dos Shell 登录后，则能看见文件列表。因此需要使用新版本的 FTP Server 代码，将其复制到 BSP 目录，并修改 Makefile。为了避免编译冲突，最好用 `arcf` 命令将 `ftpdLib.o` 从系统库中删除，注意模块名称大小写。

```
arcf -d libMCF5200gnuwx.a ftpdLib.o
```

在 `bootconfig.c` 的 `bootCmdLoop` 函数中，用缺省端口号启动 FTP 服务，代码如下：

```

#ifdef INCLUDE_FTP_SERVER
    if( ftpdInit(0,0) == ERROR)
        printf ("ftpdInit failed.\n");
#endif

```

在 `ftpdLib.c` 中指定缺省目录，不使用 `ioDefPathSet`，以操作 TFFS 文件系统，应用程序中也可用。缺省为匿名登录，没有密码。在 `ftpdLib.c` 中进行如下修改：

```

/*ioDefPathGet (pSlot->curDirName);*//*将这行去掉，用下行替代*/
strcpy(pSlot->curDirName, "/dsxxxx/");/*必须为/dsxxxx/，不能为/dsxxxx，与 TFFS 一致*/

```

● 其他

✧ 修改系统时钟的频率

修改 `bootConfig.c` 的 `usrRoot` 中时钟频率设定：`sysClkRateSet (100)`。

✧ 减少启动时间等待

修改 `bootConfig.c` 中的 `TIMEOUT` 宏。

✧ 只有空格键才能在启动过程进入 BootRom Shell

修改 `bootConfig.c` 中 `autoboot` 函数。

✧ 网卡驱动

将新的网卡驱动代码文件加入 BSP 目录，解决网络问题。修改 Makefile 和 `motFecEnd.c`，库中的 `motFecEnd.o` 不用删除。

✧ 网卡 MAC 地址操作

利用第 2 片 Flash 的空闲扇区存放 MAC 地址。在 lv160mtd.c 中添加相应的读写函数。

✧ 硬件看门狗和运行灯触发

在 bootConfig.c 中的 usrRoot 后添加无限循环触发看门狗和灯, 用 taskDelay 作等待延时, 该段代码在 tBoot 任务上下文中运行。

✧ 加载方式选择

缺省加载 TFFS 上映像, 不成功再从网络加载。在 bootConfig.c 的 autoboot 函数中进行如下修改:

```
printf ("\nauto-booting...\n\n");
/*从 TFFS 加载*/
if (tffsLoad (0, 0, "/dsxxxx/vxworks", &entry) == OK)
    go (entry);                /* ... and never return */
else
{
    printf ("Can't load boot file from TFFS!!\n");
    printErr ("\nError loading file: errno = 0x%x.\n", errno);
    taskDelay (sysClkRateGet ()); /* pause a second */
}
/*若不成功就从网络加载*/
if (bootLoad (BOOT_LINE_ADRS, &entry) == OK)
    go (entry);                /* ... and never return */
else
{
    printf ("Can't load boot file!!\n");
    taskDelay (sysClkRateGet ()); /* pause a second */
    reboot (BOOT_NO_AUTOBOOT); /* something is awry */
}
}
.....
```

✧ 读取应用参数文件的 IP 地址

在 bootConfig.c 中定义 myIpSet 函数, 用于替换 Bootline 中缺省 IP 地址。在 bootCmdLoop 中网络初始化前调用 “myIpSet(BOOT_LINE_ADRS)”。

```
BOOT_PARAMS myBootParams;
int myIpSet(char * bootString)
{
    /*TODO: 读出参数文件中 IP, 无有效参数文件则返回*/
    bootStringToStruct(bootString, &myBootParams);
    strcpy(myBootParams.ead, newip);
    bootStructToString(bootString, &myBootParams);
    return OK;
}
```

9.6.5 编译

参考前面 “BootRom 建立” 一节的描述, 简略步骤如下。

✧ 进入 Dos Shell。

✧ 运行 torvars.bat 配置 Dos Shell 的环境变量。

✧ 进入相应的 BSP 目录生成映像。

make bootrom_uncmp	(用于仿真调试, #define RAM_SIM)
--------------------	---------------------------

```
make bootrom_uncmp.hex    (用于 Flash 写入, #undef RAM_SIM)
```

9.6.6 调试

BootRom 定制过程, 不可避免会遇到问题, 有时需要进行源代码调试。VisionClick 支持源代码调试, 不过需要注意几个问题。

- ✧ visionClick 中不能看见汇编文件的源代码, 如 romInit.s。
- ✧ 代码中不要使用中文注释, 否则不能看全 C 文件的源代码, 只能看见前面部分。
- ✧ 源代码调试, 需要编译器符号表支持, 在 Makefile 中添加“-g”选项。
- ✧ bootInit.c 编译时更名, 产生复制文件 bootInit_uncmp.c 才能看见代码。

调试的简略步骤如下。

- ✧ 启动 visionClick, 关闭“Welcome to visionClick”对话框, 选择菜单“File→open project file[F12]”命令激活“project/load”对话框, 点击 OK 进入连接模式。该步骤在 Flash 写入时也需要执行。
- ✧ 使用 convert 将 bootrom_uncmp 转换生成仿真调试用的 ab、bdx 文件, 配置参数如下。

```
input module = D:\TornadoCF\target\config\bspxxxx\bootrom_uncmp
symbol file = D:\TornadoCF\target\config\bspxxxx\vxworks.ab
obj module = D:\TornadoCF\target\config\bspxxxx\vxworks.bdx
```
- ✧ 选择菜单“File→reset, load target&symbol, set pc[F11]”命令下载程序, 进入调试。

9.6.7 发布

在产品正式发布时, BootRom 需要写入第 1 片 Flash, 简略步骤如下。

- ✧ 利用 VisionClick 的 convert 工具生成 Flash 写入的 bin 文件, 配置参数如下。

```
input module = D:\TornadoCF\target\config\bspxxxx\bootrom_uncmp.hex
bin file      = D:\TornadoCF\target\config\bspxxxx\bootrom.bin
from          = 0xFFC00000    [CS0-Flash 的基地址]
to            = 0xFFFFFFFF    [足够大就行]
```
- ✧ 选择菜单“Tools→Program flash devices”命令打开“Flash Program”对话框, 进行 Flash 写入, 框配置参数如下。注意 Flash 的算法在 visionProbe 上, 需要连接 visionProbe 才能看见 Flash 列表。

```
Flash file name = D:\TornadoCF\target\config\bspxxxx\bootrom.bin
Programming algorithm = AMD      29F160xB ( 1024 x 16 )  1 Device
base addr = 0xFFC00000, erase all checked
ram space: start=0, size=9820
```

9.7 PC104/486 BootRom 实例

本节详细介绍基于 PC104/486 的系统上 BootRom 的建立过程, 可以用做类似 CPU 系统的参考。

9.7.1 目标系统

采用盛博 SCM6231 PC104/486 模块。该模块与 PC/AT 兼容, 带有 16M RAM, 4 个 16550 兼容的 RS232 端口, 对应 COM1~COM4; NE2000 兼容的网卡两块, 以及采用 TFFS 技术的 DiskOnChip 2000 电子盘一片。

9.7.2 主机环境

主机操作系统为 Windows 2000 Professional。使用了 Tornado 中的 GNU C/C++ 编译器和 PC104 工装。工装包括键盘和监视器，用来设置 BIOS、网卡地址和中断号等，以及制作 VxWorks 引导电子盘等。未使用硬件调试器，采用 printf 等方法盲调。

9.7.3 设计目标

BootRom 设计目标如下。

- ✧ 加载 COM1~COM4。
- ✧ 加载 DiskOnChip 2000 电子盘。
- ✧ 加载两块 NE2000 兼容网卡。
- ✧ 启动 FTP 服务器，通过它可以操作 TFFS 文件系统。
- ✧ 可选程序加载方式，方便调试和程序发布。在调试时，可以通过 FTP 加载 VxWorks。而程序发布时，VxWorks 存放在 DiskOnChip 中，从 DiskOnChip 加载。在 BootRom Shell 中增加两个加载命令，“#”直接从网络加载，“@”从 DiskOnChip 上加载。
- ✧ RAM 存储器分配。BootRom 加载到地址 0x8000；VxWorks 加载到地址 0x108000。
- ✧ BootRom 启动后 Console 定向到 COM1。
- ✧ 只有空格键才能在启动过程进入 BootRom Shell。
- ✧ 读取应用参数文件的 IP 地址，保证 BootRom 和 VxWorks 两者 IP 一致，以防止多目标系统中 IP 地址冲突。

9.7.4 编辑

下面按逻辑顺序进行说明，修改代码不做精确的描述。因为读者使用不同的硬件平台或软件版本，太过精细反而不好，主要着重步骤和原理的讲解。

● Makefile

针对特定的目标系统，首先需要修改的文件就是 BSP 目录下的 Makefile。

如同前面所讲，为了与原来 BSP 目录区分，需要备份原 BSP 目录夹。在这里先建立自己项目相关的 BSP 和 ALL 目录，如 bspxxxx 和 allxxxx，然后将原 BSP 目录夹 pc486 和 ALL 下的文件复制到相应新目录中，后续对 BSP 的修改就只在新目录中进行。

对 Makefile 进行如下修改。

```
TARGET_DIR    = bspxxxx
VENDOR        = SAC
BOARD         = dsxxxx
CONFIG_ALL    = ..\allxxxx      #指向自己的 ALL 目录
#将新加入 BSP 目录下的源文件，编译入 BootRom 影像
#加入相应的目标文件名，make 会自动识别依赖关系，并编译入程序
#根据该 BSP 创建工程时，这几个附加模块会自动加入
#ftplib.o 用于 FTP 服务器建立
MACH_EXTRA    = ftpLib.o
#设置 BootRom 和 VxWorks 的入口地址
#和 config.h 中保持一致
RAM_LOW_ADDR  = 00108000        # RAM text/data address
RAM_HIGH_ADDR = 00008000        # RAM text/data address
```


● config.h

对于 BSP 的定制配置最好在 config.h 中进行，不要修改 configAll.h。

与 TFFS、串口和网卡驱动相关的修改将在后面分别做具体描述。

config.h 中最重要的就是对缺省 Bootline 的配置，关于 Bootline 组成在前面已做介绍，这里就只针对项目特殊处加以说明。

```
#define DEFAULT_BOOT_LINE \  
"ene(0,0)host:vxWorks h=192.166.0.254 e=192.166.0.2 u=dsxxxx pw=dsxxxx tn=vxtarget"
```

- ✧ Bootline 中缺省使用 ene 作为启动设备，这主要是利用 BootRom 中对启动网络设备初始化的代码。而实际上笔者对 bootConfig.c 中代码作了修改，启动时先查找 TFFS 上是否有 VxWorks 映像，如果没有才从网络加载。也许这样可能不是很好，因为对网络初始化代码做了较多修改。有另一种方法，使用 TFFS 为启动设备，在 Bootline 中添加“o=ene”来初始化网络设备。
- ✧ 从主机加载 VxWorks 映像，需要指明主机地址“h=192.166.0.254”，以使目标机连接指定主机上的 FTP 服务器。登录主机 FTP 服务器的用户名和密码由“u=dsxxxx”和“pw=dsxxxx”指定。
- ✧ 目标机 IP 地址“e=192.166.0.2”。在系统正常启动时已经不再有用，笔者使用应用参数文件中的 IP 替换，以方便 IP 修改和避免启动冲突。该参数只在裸机启动时有用。

config.h 中内存相关配置如下。

```
/*无 NVRAM*/  
#define NV_RAM_SIZE NONE  
/*用户保留 RAM，设为 0*/  
#define USER_RESERVED_MEM 0  
  
#define LOCAL_MEM_LOCAL_ADRS 0x00000000 /* fixed */  
#define LOCAL_MEM_BUS_ADRS 0x00000000 /* fixed */  
  
/*RAM 大小设为 8M，通过定义 LOCAL_MEM_AUTOSIZE 让程序自动探测 RAM 大小*/  
#define LOCAL_MEM_SIZE 0x00800000 /* 8MB w lower mem */  
#define LOCAL_MEM_AUTOSIZE  
  
/*BootRom 和 VxWorks 加载地址，和 Makefile 中保持一致*/  
#define RAM_LOW_ADRS 0x00108000 /* VxWorks image entry point */  
#define RAM_HIGH_ADRS 0x00008000 /* Boot image entry point */
```

加载 FTP 服务器的组件：

```
#define INCLUDE_FTP_SERVER
```

指定网络调试方式：

```
#undef WDB_COMM_TYPE  
#define WDB_COMM_TYPE WDB_COMM_END /* END is preferred choice */
```

将 Console 定向到 COM1：

```
#undef INCLUDE_PC_CONSOLE /* KBD and VGA are included */  
#undef NUM_TTY  
#define NUM_TTY (N_UART_CHANNELS)  
#undef CONSOLE_TTY  
#define CONSOLE_TTY 0 /* console 定向到 COM1*/
```

● 串口加载

标准的 PC486 BSP 中已经有了加载 COM1 和 COM2 的代码，这里需要“照猫画虎”增加 COM3 和 COM4。

首先在 `pc.h` 中增加 COM3 和 COM4 的中断和地址定义：

```
#define COM3_BASE_ADR    0x3e8
#define COM4_BASE_ADR    0x2e8
#define COM3_INT_LVL     0x07
#define COM4_INT_LVL     0x05
```

/*串口个数定义为 4 个*/

```
#define N_UART_CHANNELS    4
```

`config.h` 中定义中断向量，注意这里在不同的宏定义中要定义两次：

```
#define COM3_INT_VEC      (INT_VEC_GET (COM3_INT_LVL))
#define COM4_INT_VEC      (INT_VEC_GET (COM4_INT_LVL))
```

在 `sysSerail.c` 中增加 COM3 和 COM4 的设备描述：

```
static I8250_CHAN_PARAS devParas[] =
{
    {COM1_INT_VEC, COM1_BASE_ADR, UART_REG_ADDR_INTERVAL, COM1_INT_LVL},
    {COM2_INT_VEC, COM2_BASE_ADR, UART_REG_ADDR_INTERVAL, COM2_INT_LVL},
    {COM3_INT_VEC, COM3_BASE_ADR, UART_REG_ADDR_INTERVAL, COM3_INT_LVL},
    {COM4_INT_VEC, COM4_BASE_ADR, UART_REG_ADDR_INTERVAL, COM4_INT_LVL}
};
```

在 `sysLib.c` 中的 `sysHwInit2` 函数中，屏蔽 LPT1 的中断绑定，因为 LPT1 中断的缺省值和 COM3 的相同。

```
/*change for com3*/
/*(void)intConnect (INUM_TO_IVEC (LPT_INT_VEC), sysStrayInt, 0);*/
```

● TFFS

✧ 到“<http://www.m-sys.com/>”下载 DiskOnChip 2000 的 VxWorks 驱动包，这里以 4.2.1 版本为例。解压软件包，得到驱动 `MSYSVXW.o` 和 `.h` 头文件。

✧ 将所有的头文件拷贝到“`target/h`”目录下。

✧ 使用 `ar` 命令列出 BSP 库文件 `libi80486gnuvx.a` 中是否包含下面的目标文件

```
ar386 -tv libI80486gnuvx.a
tffsDrv.o  tffsLib.o  dosformat.o  fatlite.o  fltl.o  nftllite.o  flflash.o
nfdc2148.o  reedsol.o  flsocket.o  flbase.o
```

然后使用 `ar` 命令从库中删除上述目标文件，例如，删除 `tffsDrv.o`。此处需要注意模块名称的大小写。

```
ar386 -dv libI80486gnuvx.a tffsDrv.o
```

✧ 将驱动文件 `MSYSVXW.o` 加入到系统库文件中：

```
ar386 -cru libI80486gnuvx.a MSYSVXW.o
```

✧ 在 `config.h` 中增加宏定义：

```
#define INCLUDE_DISKONCHIP /* M-SYSTEMS DiskOnChip */
#undef INCLUDE_TFFS /* include TrueFFS driver for Flash */
```

✧ 在 `bootConfig.c` 中的函数 `usrInit` 之前增加下面的代码：

```
#ifdef INCLUDE_DISKONCHIP
```

```

#include "fliocctl.h"
#include "fldrvvxw.h"

#ifdef __STDC__
void devSplit (char *fullFileName, char *devName);
#else
void devSplit ();
#endif /* __STDC__ */
unsigned long tffsAddresses[] = {0xd0000L, 0xd0000L};

STATUS usrTffsConfig
(
    int      drive, /* TFFS handle (usually zero) */
    int      removable, /* 0 - nonremovable flash media */
    char *   fileName /* mount point */
)
{
    BLK_DEV * pBootDev;
    char bootDir [BOOT_FILE_LEN];
    /* create block device spanning entire disk (non-destructive!) */
    if ((pBootDev = tffsDevCreate (drive, 0)) == NULL)
    {
        printErr ("tffsDevCreate(%d,0) failed.\n", drive);
        return (ERROR);
    }
    /* split off boot device from boot file */
    devSplit (fileName, bootDir);
    /* initialize boot block device as dosFs device named <bootDir> */
    if (dosFsDevInit (bootDir, pBootDev, NULL) == NULL)
    {
        printErr ("dosFsDevInit failed.\n");
        return (ERROR);
    }
    ioDefPathSet(bootDir);
    return (OK);
}

LOCAL STATUS tffsInit(void)
{
    /* tell driver to detect single DiskOnChip */
    tffsSetup (1, tffsAddresses);

    if (tffsDrv () != OK){
        printErr ("Could not initialize.\n");
        return (ERROR);
    }

    printf ("Attaching to TFFS... ");
    dosFsInit (NUM_DOSFS_FILES); /* initialize DOS-FS */
    if (usrTffsConfig (0, 0, "/tffs0") == ERROR){
        printErr ("usrTffsConfig failed.\n");
        return (ERROR);
    }
}

```

```

    printErr ("done.\n");
    return(OK);
}
LOCAL STATUS tffsLoad
(
    int      drive,      /* TFFS handle (normally zero) */
    int      removable, /* 0 - nonremovable flash media */
    char     * fileName, /* file name to download */
    FUNCPTR * pEntry
)
{
    int fd;

    if ((fd = open (fileName, O_RDONLY, 0)) == ERROR) {
        printErr ("\nCannot open \"%s\".\n", fileName);
        return (ERROR);
    }
    if (bootLoadModule (fd, pEntry) != OK)
        goto tffsLoadErr;
    close (fd);
    return (OK);

tffsLoadErr:
    printErr ("\nerror loading file: status = 0x%x.\n", errnoGet ());
    close (fd);
    return (ERROR);
}
#endif /* INCLUDE_DISKONCHIP */

```

这里要注意 `tffsAddresses` 数组定义了搜索 `DiskOnChip` 的地址空间，然后程序在 `tffsInit` 调用 `tffsSetup` 来通知驱动检测 `DiskOnChip`。`tffsSetup` 函数原型在 `fldrvvxw.h` 中定义。

```
void tffsSetup (int diskonchips, long *addressRange);
```

例如，有三片 `DiskOnChip`：

```

#include "fldrvvxw.h"
/* address ranges to search for the DiskOnChip */
long tffsAddresses[] =
{
    0xd0000, 0xd1fff, /* 1st DiskOnChip */
    0xd2000, 0xd3fff, /* 2nd */
    0xd4000, 0xd5fff, /* 3rd */
};
/* configure the driver to detect up to three DiskOnChips */
tffsSetup(3, tffsAddresses);

```

如果系统中只有一片 `DiskOnChip` 安装在已知的地址，那么可以将 `addressRange` 中起始和结束的值都设为 `DiskOnChip` 的地址值来通知驱动不必扫描地址空间检测 `DiskOnChip`：

```

#include "fldrvvxw.h"
/* single address to look for a DiskOnChip */
long tffsAddresses[] = { 0xd0000, 0xd0000 };
/* configure the driver to detect single DiskOnChip */
tffsSetup(1, tffsAddresses);

```

✧ 在 `bootConfig.c` 函数 `bootHelp` 的 `helpMsg` 数组中增加下面 `INCLUDE_DISKONCHIP` 定义相关的代码段：

```

#ifdef INCLUDE_ATA
    "boot device: ata=ctrl,drive          file name: /ata0/vxWorks","",
#endif /* INCLUDE_ATA */
#ifdef INCLUDE_DISKONCHIP
    "boot device: tffs=drive,removable    file name: /tffs0/vxWorks","",
#endif /* INCLUDE_DISKONCHIP */
#ifdef INCLUDE_PCMCIA
    "boot device: pcmcia=sock            file name: /pcmcia0/vxWorks","",
#endif /* INCLUDE_PCMCIA */

```

✧ 在 bootConfig.c 的函数 bootHelp 中增加下面 INCLUDE_DISKONCHIP 定义相关的代码段:

```

#ifdef INCLUDE_ATA
    printf (" ata");
#endif /* INCLUDE_ATA */
#ifdef INCLUDE_DISKONCHIP
    printf (" tffs");
#endif /* INCLUDE_DISKONCHIP */
#ifdef INCLUDE_TFFS
    printf (" tffs");
#endif /* INCLUDE_TFFS */

```

✧ 在 bootConfig.c 的函数 bootLoad 的最后增加下面 INCLUDE_DISKONCHIP 定义相关的代码段:

```

#ifdef INCLUDE_DISKONCHIP
    if (strncmp (params.bootDev, "tffs", 4) == 0) {
        int drive = 0;
        int removable = 0;

        if (strlen (params.bootDev) == 4)
            return (ERROR);
        else
            sscanf (params.bootDev, "%*4s%c%d%c%d", &drive, &removable);

        if (tffsLoad (drive, 0, params.bootFile, pEntry) != OK) {
            printErr ("\nError loading file: errno = 0x%x.\n", errno);
            return (ERROR);
        }
        return (OK);
    }
#endif /* INCLUDE_DISKONCHIP */

```

✧ 在 bootConfig.c 中增加宏定义 INCLUDE_DISKONCHIP 判断:

```

#if (defined (INCLUDE_SCSI_BOOT) || defined (INCLUDE_FD) || \
    defined (INCLUDE_IDE) || defined (INCLUDE_ATA) || \
    defined (INCLUDE_DISKONCHIP) || defined (INCLUDE_TFFS))
#define SPIN_UP_TIMEOUT 45 /* max # of seconds to wait for spinup */

```

✧ 在 bootConfig.c 的 bootCmdLoop 函数中调用 tffsInit。


```

#ifdef INCLUDE_WINDRIVER_DISKONCHIP
    tffsInit();
#endif

```

这样，在 BootRom 中 DiskOnChip 就加载起来了，就可以调用 tffsLoad 从 DiskOnChip 加载 VxWorks，也可以增加 FTP Server 后通过 FTP 访问 DiskOnChip。在 VxWorks 中加载

DispOnchip 与 BootRom 中类似，只是相关的改动在 usrTffs.c 中进行。

 详细的增加驱动文档，可以参考软件包中的 readme。

● NE2000 兼容的双网卡

BootRom 中驱动网卡以及增加第二块网卡与 VxWorks 中操作基本相同，具体可参见“网络通信”一章中的“网络驱动加载配置”和“增加第二块网卡”两小节。这里需要注意的是在 BootRom 中 IP_MAX_UNITS 定义在 bootConfig.c 中，需要定义为 2。

```
#ifndef IP_MAX_UNITS
#define IP_MAX_UNITS 2
#endif
```

● FTP 服务器

根据设计目标，目标机可以启动 FTP 服务来操作 TFFS 文件系统，完成程序映像和参数文件维护。系统库中本来自带有 FTP 服务器的实现库，但版本太老，可以使用 dosFs 2.0 软件包新版本的 FTP Server 代码，将其复制到 BSP 目录，并修改 Makefile。为了避免编译冲突，需要将 ftpdLib.o 从系统库中删除。

在 bootconfig.c 的函数 bootCmdLoop 中，在网络和 DispOnChip 加载完成后加载 FTP 服务。

```
#if defined(INCLUDE_FTP_SERVER)
#if defined(INCLUDE_FTPD_SECURITY)
    loginInit();
    ftpdInit((FUNCPTR) loginUserVerify, 0);
#else
    if (ftpdInit (0,0)==OK)
        printf("ftpd running...\n");
    else
        printf("ftpd failed!...\n");
#endif
#endif
```

● 其他

✧ 修改系统时钟的频率

修改 bootConfig.c 的 usrRoo 中时钟频率设定：sysClkRateSet (100)。

✧ 减少启动时间等待

修改 bootConfig.c 中的 TIMEOUT 宏。

✧ 硬件看门狗和运行灯触发

在 bootConfig.c 中的 usrRoot 后添加无限循环触发看门狗和灯，用 taskDelay 做等待延时，该段代码在 tBoot 任务上下文中运行。

✧ 只有空格键才能在启动过程中进入 BootRom Shell 以及选择加载方式。修改 bootConfig.c 中 autoboot 函数。当有空格键入则进入 BootRom Shell，否则自动加载 VxWorks。缺省加载 TFFS 上的映像，不成功则再从网络加载。

```
.....
if (timeout > 0){
    printf ("\nPress <space> to stop auto-boot...\n");
    .....
```

```

}
/*add */
if (bytesRead == 0)
    key=0;
else{
    read (consoleFd, &key, 1);
    key &= 0x7f;          /* mask off parity in raw mode */
}
/*change */
/*if (bytesRead == 0)/*      /* nothing typed so auto-boot */
if (key!=32) { /*space*/
    /* put the console back in line mode so it echoes (so's you can bang
    * on it to see if it's still alive) */
    (void) ioctl (consoleFd, FIOSETOPTIONS, OPT_TERMINAL);
    printf ("\nauto-booting...\n\n");
    if (tffsLoad (0, 0, "/dsxxx/vxWorks", &entry) == OK)
        go (entry);          /* ... and never return */
    else{
        printf ("Can't load boot file!!\n");
        if (bootLoad (BOOT_LINE_ADRS, &entry) == OK)
            go (entry);      /* ... and never return */
        /*taskDelay (sysClkRateGet ());*//* pause a second */
        /*reboot (BOOT_NO_AUTOBOOT);*//* something is awry */
    }
}
}
else{
    /* read the key that stopped autoboot */
    /*change by wyj*/
    /*read (consoleFd, &key, 1);
    return (key & 0x7f);*/      /* mask off parity in raw mode */
    return(key);
}

```

同时在函数中 `bootCmdLoop` 增加对命令 “@” 和 “#” 的处理，分别对应从 TFFS 和网络加载 VxWorks，具体不再赘述。

✧ 读取应用参数文件的 IP 地址，与 9.6.4 小节的介绍类似。

9.7.5 编译

读者可以参考前面 “BootRom 建立” 一节的介绍。简略步骤如下。

- ✧ 进入 Dos Shell。
- ✧ 运行 `torvars.bat` 配置 Dos Shell 的环境变量。
- ✧ 进入相应的 BSP 目录生成映像。

```

make bootrom_uncmp
vxcopy bootrom_uncmp bootrom.sys

```

9.7.6 调试

因为无硬件调试器，一般采用 `printf` 或 `printErr` 打印信息来盲调。

9.7.7 发布

这里采用二级引导法，PC104/486 加电后主引导扇区为 Vxld，Vxld 加载 DiskOnChip 上

的 BootRom，BootRom 加载 VxWorks。


- ✧ 准备一张空白的可引导软盘 A，将工具 vxSys.com 和生成的 bootrom.sys 拷贝至 A 盘。
- ✧ 通过工装设置 PC104/486 软驱盘为第一启动器，DiskOnChip 为 C 盘。用软盘引导进入 Dos。
- ✧ 确认 DiskOnChip 为空白，可使用“format c:”命令格式化 DiskOnChip。必要时使用 dformat 实行低级格式化，完成后需要重新启动。
- ✧ 运行厂商程序设置网卡的中断和地址参数。
- ✧ 将 DiskOnChip 作为 VxWorks 引导。

```
vxsys c:
```

- ✧ 将 bootrom.sys 拷贝至 DiskOnChip。

```
copy bootrom.sys c:\
```

- ✧ 复位，以 DiskOnChip 为启动盘，则 bootrom.sys 应该成功启动。之后可以通过 BootRom 的 FTP 服务器下装 VxWorks 到 DiskOnChip 中。

 如果系统复位后停在 Vxld 后无反应，则可能是 DiskOnChip 驱动使用高版本导致编译后 bootrom.sys 过大的缘故，建议尝试低版本的驱动。

 DISKONCHIP 及网卡的操作，参见具体产品的资料。

9.8 建立开发环境

目标机 BootRom 建立完成后，可以和主机 Tornado 建立起交叉开发环境，以进行后续的应用开发。

9.8.1 调试方式

从前面内容可以知道主机和目标机之间存在多种连接方式，如网络、串口等。还有模拟器和主机工具之间的连接。而其中最常使用的方式是网络。下面分别介绍这些连接方式的建立。

● 主机模拟

VxWorks 提供模拟器 VxSim。该程序在主机上运行，用于模拟 VxWorks 的目标机。主要用于原型建立和代码段测试。

Tornado 自带的 VxSim 不支持网络。作为单独产品的 VxSim 完全版本可以支持网络。

要选择模拟器调试程序需在工程创建时确定。选用 simNTgnu 工具链。一般创建 Downloadable 工程来生成模块映像完成调试。下载模块进行调试时，IDE 会自动提示启动模拟器和相应的 Target Server。开发者可以方便的继续调试。

当随着调试的进行，会用到越来越多的系统组件。而有些组件并没有包含在缺省的 VxWorks 映像中。这时就需要定制自己的 vxworks.exe，在启动调试器时会出现窗口让用户选择。创建 vxworks.exe 和一般的 Bootable 工程一样，只是选用 simNTgnu 工具链。

● 网卡调试

大多数 VxWorks 系统选择网络进行调试。高速通信是其最大的优点。

首先需要创建 Bootable 工程。在 VxWorks 组件窗口中选择 WDB 连接为 END 驱动。生

成的 VxWorks 映像通过前面介绍的加载方式到内存运行，等待调试。

主机侧需要配置相应的 Target Server 和目标机相连。配置中选择 BackEnd 为 wdbrpc，填写目标机 IP 地址。启动 Target Server 和目标机建立通道后，主机各工具就可与目标机交互完成调试工作。

需调试代码可以编译入 VxWorks 中整体调试，也可以创建新的 Downloadable 工程作为模块动态加载单独调试。

● 串口调试


如果目标机不提供网络接口，或需要调试网络驱动时，可以选择串口调试。

首先需要创建 Bootable 工程。在 VxWorks 组件窗口中选择 WDB 连接为 seirial。并对该连接的属性进行配置，如选择串口通道和波特率等。生成的 VxWorks 映像通过 TSFS 的方式到内存运行，等待调试。

主机侧需要配置相应的 Target Server 和目标机相连。配置中选择 BackEnd 为 wdbserial，选择主机串口和波特率（注意和前面的配置一致），配置 Target Server 文件系统。启动 Target Server 和目标机建立通道后，主机各工具就可与目标机交互完成调试工作。

● 其他方式

除了上面的介绍的方式外，还存在其他的调试方式。如 netrom、visionProbe 等。理论上，任何主机和目标机的通信通道，只要 WDB 支持，都可以用做调试。

 longyong, “VxWorks 的启动方式”。

9.8.2 Hello World!

对于任何开发环境，初学者总想运行自己的第一个“Hello World”程序。只有运行了这个程序后，学习者才会觉得开始进入其中，才有兴趣继续探索下去。然而，如果读者能建立 VxWorks 的 Hello 程序，就不是初学者了，因为需要完成了硬件调试、BSP 定制、硬件驱动等前期高难工作。如果这些前期工作都是别人替你做的，再加上本节描述，算是有福了，省去大量读英文的辛苦，轻松成为 VxWorks 开发者。

Tornado 入门指南中提供了一个 cobble 例程，算是一个经典“Hello World”的例子，先前出版的 VxWorks 中文书中都有引用。例程涉及 Downloadable 工程创建，源代码文件加入，工程编译，映像下载，程序运行，代码调试等各方面，描述了 Simulator、WindSh、Browser 和 WindView 等主机工具的使用。该例程对应用程序开发过程作了完整的概要描述，初学者都应该看看。

 WindRiver, “Tornado 2.0 Getting Started Guide” 的第 3 章。

（Tonado2 的一些版本中没有该文档，Tornado Prototyper 版中有，Tornado 2.2 中都有）

本节的内容也想实现同样的目的，对应用程序的开发过程做一个概要描述。为了节省篇幅，不提供与上例中一些重复的贴图和描述，但步骤是完整的。

这里提供的例子和 cobble 例程有所不同，与实际的开发更为近似。使用实际硬件作为目标机，建立可以脱离主机运行的 Bootable 映像，以及 TFFS、FTP Server、Target Shell 等实用

组件的加载。而应用程序本身是很简单的，不涉及复杂的多任务协调，只建立一简单的周期性任务。下面按实际工作顺序进行描述。

- 在前面工作的 BSP 上创建 Bootable 工程

工程创建时，BSP 下的源文件会自动添加到工程中。需要注意的是，工程创建产生依赖时，会使用 BSP 目录下的 Makefile 文件，其中用 MACH_EXTRA 添加的目标模块名称会区分大小写，只有与源代码文件名一致，才会被自动加入工程。否则只能等工程创建完成后，自己手动添加。

- 在任意位置创建 hello.c，并将其加入工程中，源代码如下。

```
#include "vxWorks.h"
/*Tick 任务的函数体*/
void tTick(void)
{
    for(;;){ /*嵌入式应用的特点，无限循环*/
        taskDelay(50); /*睡眠 50 个 ticks，具体时间由 SYS_CLK_RATE 值确定*/
        printf ("Hello...");
        taskDelay(50);
        printf ("World!!!\n");
    }
}
/*应用程序入口函数，可在 usrAppInit (), shell, debugger 中调用运行*/
void root(void)
{
    int tid;
    /*进入应用程序后，TFFS 加载点消失，重新加载 TFFS*/
    /* ftp 和 telnet 服务器会由包含的系统组件自己启动*/
    printf ("Attaching to TFFS... ");
    if (usrTffsConfig (0, 0, "/dsxxxx/vxworks") == ERROR)
        printf ("usrTffsConfig failed.\n");
    printf ("done.\n");
    /*创建并激活 tTick 任务，优先级 150，任务栈 4096 字节，无入口参数*/
    tid = taskCreat("tTick", 150, 0, 4096, (FUNCPTR)tTick, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
    taskActivate(tid);
}
```

- 修改工程下 usrAppInit.c，加入 root 调用。确保组件中包含了“application initialization”。

```
/******
 *
 * usrAppInit - initialize the users application
 */
void root(void); /*函数原型引用*/
void usrAppInit (void)
{
#ifdef USER_APPL_INIT
    USER_APPL_INIT; /* for backwards compatibility */
#endif

    /* add application specific code here */
    root(); /*入口函数调用*/
}
```

- 编译连接生成 VxWorks 映像，上电目标硬板，启动进入 BootRom Shell。利用 BootRom 中的 FTP 服务器，在主机用 FTP 客户程序将 VxWorks 文件下载到 TFFS 文件系统中。用 Shell 中的 “@” 命令加载 VxWorks 映像到 RAM 中运行。现在得到的就是真实的 VxWorks 应用系统，重新上电时系统会自启动运行。
- 在主机启动超级终端，连接 Console 串口或 Telnet 端口，就可以看到喋喋不休的 “Hello...World!!!”。要想让它停下来，最简单的方法就是关掉硬板的电源。
- 如果这样的 VxWorks 能让客户满意，那就恭喜你，程序可以发布了。不过一般来说，都会有一些问题。需要再修改和编译。若遇到的问题超出开发者的想像，就需要进行调试了。对于正在运行的 hello 程序，只要包含了相关的开发组件和调试符号表，它总是在等待调试。现在首先要做的就是主机配置 Target Server。假设使用网络调试，选择 BackEnd 为 wdbrpc，设置目标机的 IP 地址，核心文件指向主机上 hello 工程下的 VxWorks 映像，选择 “all symbol”。参考前面对 Target Server 配置的描述。
- 启动 Target Server 后，就可以使用丰富的主机工具来辅助开发。当然最常用的就是调试器，可以跟踪源代码的执行。为了简化断点的设置，最好先使用全局型断点。
- 有多种方式可以实现这个入门程序。比如将 hello.c 编译为可下载模块，通过主机下载执行，或者将这个模块放到目标机存储盘上，通过启动脚本加载或手动加载。随着对开发环境的熟悉，任何一种方式读者都会觉得很简单，因为这仅仅是个 “Hello World”。
- 到这里，本章的描述就算大功告成，而你就该开始真正的工作了，而不再是休闲的 “Hello World” 程序。另外需要提醒下，前面的介绍有些和硬件平台相关，也许不适合读者的应用环境，关键是注重整个过程。

9.10 常见问题解答

- **BSP 与 BootRom 的区别？**

解答：BSP 是个软件抽象层，包含硬板相关的代码，为上层代码提供抽象接口，提高可移植性。BootRom 是个应用执行程序，由 BSP 代码和 VxWorks 库代码等生成，是个简略版的 VxWorks 映像。

- **不通过网络设备加载 VxWorks, 启动时如何初始化网络设备？**

解答：一般可以直接通过 Bootline 中的 other 选项来指定初始化网络设备，如 “o=ene”。如果不是标准设备，需要修改 “\comps\src\net\usrNetBoot.c” 文件中 usrNetDevNameGet 函数，添加相应的设备名比较。

- **PC console 只显示 Vxld 的启动过程后就没有反应，这是什么原因？**

解答：Tornado 2.2 版容易出现这个问题，因为 Console 缺省不是定向在显示器上。如果 Console 定向没有问题，BootRom 映像就有问题，需要进一步调试。

- **“Attaching network interface ... ” 后未显示成功信息 “done”，该如何解决？**

解答：这表示网络接口初始化出现问题。首先检查硬件配置和网线连接是否正确，再检查 config.h 中配置的网卡类型、网卡中断和 IO 地址是否正确。检查 configNet.h 中加载串设

置是否正确，网卡驱动是否适合选择的网卡。换块网卡或目标板再试试。

● 显示“Loading ...”后系统没有反应，该如何解决？

解答：如果是网络加载，可能网络通信存在问题，如网线连接问题，主机或目标机的 IP 地址设置问题等。查看主机 FTP 服务器是否启动，是否配置目标机用户，登录目录是否和目标机中的设置一致，可以查看 FTP 服务器的 log 信息。映像文件名称是否正确，主机指定目录是否存在该文件。如果用其他方式加载，与上面类似，检查存储设备是否加载成功，指定的映像文件路径名是否正确，存储设备上是否有该文件存在。

● 显示“Starting at...”后系统没有反应，该如何解决？

解答：检查硬件配置是否正确，检查 config.h 中 LOCAL_MEM_SIZE 是否与硬件实际的内存大小一致；关闭 Cache（USER_D_CACHE_ENABLE）和 MMU（INCLUDE_MMU_BASIC）功能试试，这两个宏在 config.h 中定义。

● 显示“Core file checksum does not match”是什么意思？

解答：加载运行的 VxWorks 映像文件和 Target Server 中指定核心文件（core file）不一样。有可能是编译后未重新下载，或者指定的路径不对。

第 10 章 程序开发实践

10.1 Tornado 扩展

Tornado 是一个开放的环境，可以由用户根据需要定制。但 Tornado 实在不是一个完善的集成开发环境，不能完全覆盖软件开发的整个过程，组合利用其他工具软件，可帮助开发者提高工作效率。


10.1.1 工程组织

用户可以按 3 种方式组织工程来进行 VxWorks 程序开发。

第 1 种，建立单一 Bootable 工程，将所有应用代码加入工程，与 BSP 代码和 VxWorks 库一起编译生成独立可运行映像。这种方法最简单，也最常用。

第 2 种，用应用代码建立一个或多个 Downloadable 工程，完成软件的分块调试后，将应用代码生成静态库或部分连接映像。再建立一个基本的 Bootable 的工程，连入应用静态库生成最终映像。或者生成基本 VxWorks 映像，启动时利用动态加载和脚本机制，连入应用部分连接映像或目标模块。这种方法能提高工作组并行开发效率，增强系统配置灵活性。

第 3 种，建立有层次结构的工程组织，将各独立的工程联系在一起。这种方式与应用源代码文件的层次组织相一致，适合开发中某些实际情况，如可以对不同源代码文件使用不同的编译规则等。而 Tornado 的工程本身没有层次概念，各工程相互独立，所以需要用户自己完成一些工作，如定制宏和规则等。这种方式的应用可以参考下面所列的文档。

 WindRiver, “Tornado 2.2 User’s Guide” 的 4.2.3。

 WindRiver, “TechTips-Nested Tornado Projects”。

 WindRiver, “TechTips-How to configure a downloadable project...”。

10.1.2 Tornado API

Tornado 提供了丰富的 API 用于环境定制，用户可以按需要进行功能扩展，方便连接第三方产品。但这些 API 使用的复杂程度和 VxWorks 的使用不相上下，有几个帮助文档专门描述。一般用户很少用 API 对 Tornado 进行定制，但也能顺利完成程序开发，API 定制对一般用户来说并不是必要的，除非用户开发的是 Tornado 辅助工具。在 Tornado API 中，TCL API 最容易使用，主要因为 TCL 本身所具有的交互特性，也因为 Tornado 中提供了许多例子，如信号量检查、内存操作和任务创建等。本节举一个简单的 TCL API 例子，让读者了解这种灵活机制的存在。

- 用 TCL 为 Tornado 增加 DOS Shell 按钮

用户经常需要使用命令行窗口（先运行 torvars.bat 建立环境，再切换到工作目录），如果所有一切通过 Tornado 工具按钮来完成将非常方便。下面例子来自 WindRiver 网站的 TechTips 文章，讲述如何利用 TCL 完成这一简单的 Tornado 定制工作。

定制工作需要制作两个文件，一个为 TCL 脚本描述文件，另一个为按钮图标文件。两文件需要放到 WIND_BASE 下指定目录。

\$WIND_BASE/host/resource/tcl/app-config/tornado/03msdos.win32.tcl

\$WIND_BASE/host/resource/bitmaps/Launch/controls/msdos.bmp

重启 Tornado 后, 会看到 DOS Shell 按钮和菜单项。激活按钮, 启动 DOS 窗口, torvars.bat 已经运行, 并自动切换到相应的工程目录, 可立即继续用户的操作, 如图 10-1 所示。

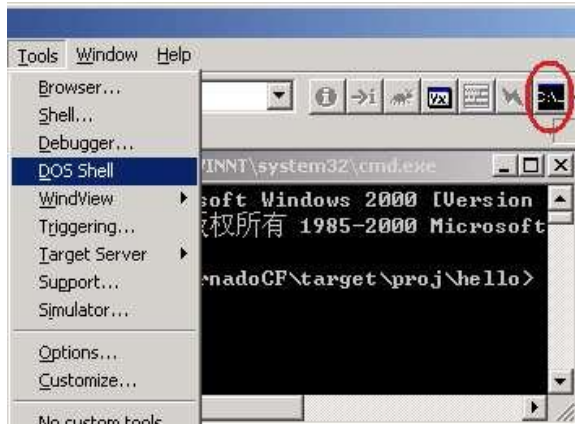


图 10-1 添加 DOS Shell 按钮

文件 03msdos.win32.tcl 的内容如下。

```
proc dosShellInitUI {} {
    controlCreate launch [list toolbarbutton \
        -callback "LaunchDosShell" \
        -name dosshell \
        -tooltip "Launch DOS shell" \
        -bitmap "[wtxPath host resource bitmaps launch controls]msdos.bmp"]

    menuItemInsert \
        -callback {LaunchDosShell} \
        -after -bypath \
        {&Tools &Debugger...} "&DOS Shell"
}

proc LaunchDosShell {} {
    global tcl_platform

    # get the directory for current project
    set projDir [::Workspace::projectDirGet]
    regsub -all {/} $projDir {\\} projDir

    # determine which OS we are running under and
    # store the appropriate command interpreter name

    if { $tcl_platform(os) == "Windows NT" } {
        # set to use NT command interpreter
        set cmd_interp cmd.exe
    } else {
        # set to use 95 command interpreter
        set cmd_interpl "[wtxPath host x86-win32 bin]torvars.bat"
        regsub -all {/} $cmd_interpl {\\} cmd_interpl
    }
}
```

```

        set cmd_interp "command.com /K $cmd_interp1"
    }

    # start the command interpreter
    toolLaunch "DosShell" "$cmd_interp" "$projDir" 0 0 0 0 0 0
}
dosShellInitUI

```

图标文件 msdos.bmp 的制作很简单，只不过要注意大小，一般 20×20 即可。

10.1.3 CDF 组件定制

Tornado 通过 CDF[Components Description File]文件来管理 VxWorks 的组件配置。本节先介绍 CDF 文件的基本结构，并举例说明如何利用 CDF 文件来添加用户定制组件。

● 组件描述文件[CDF]

Tornado 利用组件描述文件来辅助组件配置。这些文件存放在 “\target\config\comps\vxworks\” 目录下，如 00vxworks.cdf。Tornado 启动时会扫描这些文件，以获得组件列表，并提取当前配置，所以第一次打开组件窗口时有些缓慢。

以 “00” 开头的 CDF 文件为基本组件描述，如 00vxworks.cdf、00bsp.cdf 等。安装其他组件产品或补丁后，会添加相应的组件描述文件，如 10dosfs2.cdf、01zinc.cdf 等。

CDF 文件由 5 种基本部件组成：组件夹[Folder]、组件[Component]、初始化组[InitGroup]、参数[Parameter]和选项[Selection]。各部件都有自己的基本属性，属性的名称很直观，直接查看 CDF 文件就能明白。这里以 Target Shell 组件为例来讲述 CDF 文件的结构。

✧ **Folder** 为组件的归类组织单位，使组件在图形界面上显示层次结构。组件夹下面可以包含组件，也可以包含子组件夹，形成多层结构。

```

Folder FOLDER_SHELL {
    NAME      target shell components
    CHILDREN  INCLUDE_DEBUG      \    /*组件夹包含的组件*/
              INCLUDE_SHELL_BANNER \
              INCLUDE_STARTUP_SCRIPT \
              INCLUDE_SHELL
    DEFAULTS  INCLUDE_DEBUG      \    /*需缺省包含的组件*/
              INCLUDE_SHELL_BANNER \
              INCLUDE_SHELL
}

```

✧ **Component** 为最基本的组成元素，描述组件的属性，组件不允许嵌套。

```

Component INCLUDE_SHELL {                /*由库模块提供初始化函数*/
    NAME      target shell
    MODULES   shellLib.o
    INIT_RTN  shellInit (SHELL_STACK_SIZE, TRUE);
    HDR_FILES shellLib.h
    CFG_PARAMS SHELL_STACK_SIZE
}
Component INCLUDE_SHELL_BANNER {         /*由源文件提供初始化函数*/
    NAME      shell banner
    SYNOPSIS  display the WRS banner on startup
    CONFIGLETTES usrBanner.c             /*prjConfig.c 中#include "usrBanner.c"*/
    INIT_RTN  usrBanner ();               /*usrBanner() 在 usrBanner.c 中定义*/
}

```

```

    REQUIRES INCLUDE_SHELL
    HDR_FILES sysLib.h stdio.h shellLib.h usrLib.h
}

```

还有一些其他的组件属性如表 10-1 所示。

表 10-1 组件其他属性

属性	说明
INIT_BEFORE	指定在某个组件前初始化，调用 INIT_RTN
INIT_AFTER	指定在某个组件后初始化，调用 INIT_RTN
EXCLUDES	排斥组件，如 INCLUDE_SIO 和 INCLUDE_TYCODRV_5_2 互相排斥
LINK_SYMS	连接符号，用于将相应的组件编译入映像，即使应用未使用该组件
MACRO_NEST	宏嵌套，如 INCLUDE_WDB 之于 INCLUDE_WDB_COMM_END
INCLUDE_WHEN	包含时机，如 INCLUDE_CACHE_SUPPORT 时，包含 INCLUDE_CACHE_ENABLE
_CHILDREN	作为某个组件夹的子组件（组件夹也可用）
_INIT_ORDER	指定将 INIT_RTN 调用加入某个 InitGroup 函数

- ✧ **InitGroup** 为组件初始化调用函数，按一定顺序调用各组件的 INIT_RTN。初始化组可以嵌套使用，形成多层次调用。初始化组函数由 Tornado 自动生成，存放在 prjConfig.c 中。例如下面的 usrShellInit，按 INIT_ORDER 顺序调用指定组件的初始化函数，再由 usrToolsInit 调用，usrToolsInit 再由 usrRoot 调用，usrRoot 为最高级的初始化组函数。

```

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

```

```

/*****
* usrShellInit - the target shell.
*在 prjConfig.c 中由 Tornado 自动生成，注释为组件的 SYNOPSIS 或 NAME */
void usrShellInit (void)
{
    dbgInit ();          /* breakpoints and stack tracer on target. ...*/
    usrBanner ();        /* display the WRS banner on startup */
    shellInit (SHELL_STACK_SIZE, TRUE); /* target shell */
}

```

- ✧ **Parameter** 部件用来为组件定义参数，如 LOCAL_MEM_AUTOSIZE。

```

Parameter LOCAL_MEM_AUTOSIZE {
    NAME        local memory Autosize        /*名称*/
    SYNOPSIS    Run-time (dynamic) sizing    /*概要描述*/
    TYPE        exists                        /*类型*/
    DEFAULT     FALSE                        /*缺省值*/
}

```

- ✧ **Selection** 用于同类组件的选择配置。


```

Selection SELECT_SYM_TBL_INIT { /*符号表初始化配置*/
    NAME      select symbol table initialization
    SYNOPSIS  method for initializing the system symbol table
    COUNT     1-1                      /*至少选 1 个，最多选 1 个*/
    CHILDREN  INCLUDE_STANDALONE_SYM_TBL INCLUDE_NET_SYM_TBL
    DEFAULTS  INCLUDE_STANDALONE_SYM_TBL
}
Selection SELECT_WDB_MODE { /*WDB 调试模式选择*/
    NAME      select WDB mode
    COUNT     1-2                      /*至少选 1 个，最多选 2 个*/
    CHILDREN  INCLUDE_WDB_TASK INCLUDE_WDB_SYS
    DEFAULTS  INCLUDE_WDB_TASK
    HELP      tgtsvr WDB
}

```

● 定制组件举例

由于 CDF 文件的开放结构，用户可以开发自己的组件发布，并集成到 Tornado 的工程管理中。新的组件添加 CDF 文件，并不修改原来组件的 CDF 文件。CDF 文件名的前两位数字用于确定 Tornado 扫描 CDF 文件的顺序，定制组件的 CDF 名称也应该遵循该规则。

下面举个简单的例子来说明添加定制组件的过程。

添加一个“**Amine components**”组件夹。组件夹中添加几个“**Component**”，用于启动时输出一些信息。还添加一个“**Selection**”，用于选择中英文输出。添加一个“**Parameter**”用于确定信息的输出次数。

✧ 在“\target\conifg\comps\vxworks\”目录下添加“99amine.cdf”文件，内容如下。

```

Folder FOLDER_AMINE {
    NAME      Amine components
    CHILDREN  INCLUDE_AMINE_BANNER SELECT_HELLO_MODE
    DEFAULTS  INCLUDE_AMINE_BANNER SELECT_HELLO_MODE
    _CHILDREN FOLDER_ROOT
}
Component INCLUDE_AMINE_BANNER {
    NAME      amine banner
    SYNOPSIS  display the amine banner on startup
    CONFIGLETTES usrAmine.c
    INIT_RTN  usrAmineBanner();
}
Selection SELECT_HELLO_MODE {
    NAME      select hello language
    COUNT     1-1
    CHILDREN  INCLUDE_HELLO_CHINA INCLUDE_HELLO_ENG
    DEFAULTS  INCLUDE_HELLO_CHINA
}
Component INCLUDE_HELLO_CHINA {
    NAME      hello in chinese
    CONFIGLETTES usrAmine.c
    INIT_RTN  usrHelloChina(AMINE_HELLO_NUM);
    CFG_PARAMS AMINE_HELLO_NUM
}
Component INCLUDE_HELLO_ENG {
    NAME      hello in english

```

```

CONFIGLETTES  usrAmine.c
INIT_RTN  usrHelloEng(AMINE_HELLO_NUM);
CFG_PARAMS  AMINE_HELLO_NUM
}
Parameter AMINE_HELLO_NUM {
    NAME      amine hello number
    TYPE      int
    DEFAULT   1
}
InitGroup usrAmineInit {
    INIT_RTN  usrAmineInit ();
    SYNOPSIS  the amine components
    INIT_ORDER INCLUDE_AMINE_BANNER INCLUDE_HELLO_CHINA INCLUDE_HELLO_ENG
    INIT_AFTER INCLUDE_USER_APPL
    _INIT_ORDER usrRoot
}

```

✧ 在 “\target\config\comps\src\” 目录下添加 “usrAmine.c” 文件，内容如下。

```

void usrAmineBanner(void)
{
    printf("Amine Banner!\n");
}
void usrHelloChina(int num)
{
    int i;
    for(i=0;i<num;i++) printf("你好世界!");
    printf("\n");
}
void usrHelloEng(int num)
{
    int i;
    for(i=0;i<num;i++) printf("Hello World!");
    printf("\n");
}

```

重新启动 Tornado 后，打开组件列表可看见加入的新组件（如图 10-2 所示），可以像其他 VxWorks 系统组件一样使用。加入新组件再编译时，prjConfig.c 中会自动加入 usrAmineInit 函数调用。

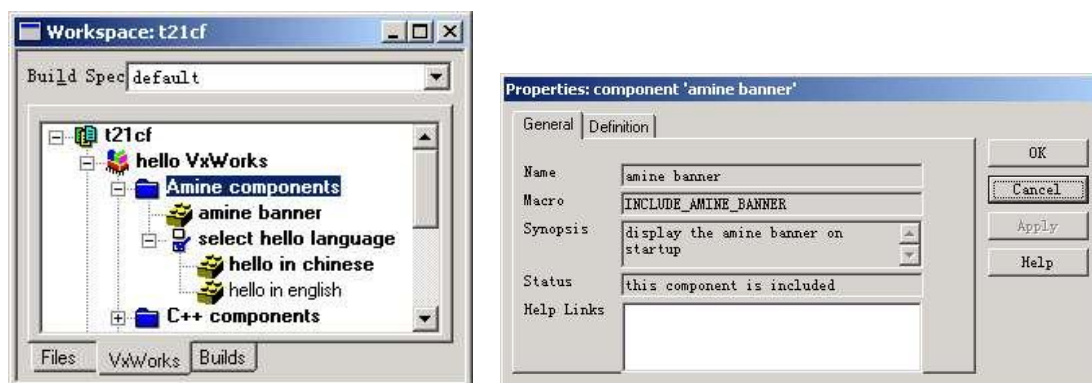


图 10-2 自定义组件

这是一个通常的 “Hello World” 示例，就是复杂了一些，不太适合初学者。确实，即使

是简单的“Hello World”程序，在嵌入式系统开发中都不太容易实现，比如在定制硬件目标机上，要说“Hello World”，得先完成硬件调试、BSP 定制等。当然 Tornado 也提供一个容易的实现方法，可以用主机仿真器脱离目标机运行程序。

二进制库组件加入和上面的过程类似，只是组件的初始化函数在目标模块中定义，需要使用 MODULES 参数，而不是 CONFIGLETES。

如果用户发布了自己的组件，帮助文档就是必需的，前面的示例略过了文档的添加。可以学习 VxWorks 方式，从代码中提取文档，这里先略过，本书的 10.1.7 节会介绍代码中文档的提取。

10.1.4 编辑器

代码编辑器对于程序员来说是一个必备工具，直接关系到工作的效率和质量。而 Tornado 集成环境的编辑器不是太完善，缺少很多方便的编辑器功能，最大缺陷是不支持中文输入和源代码解析。

Tornado 提供了扩展接口用于连接外部编辑器，缺省支持 vi、CodeWright 和 Visual SlickEdit，如图 10-3 所示，该窗口可通过选择菜单“Tools→Options”命令激活。命令行支持两个参数文件名（\$filename）和行号（\$lineno），点击命令行后的按钮可以看到。Tornado 将参数按一定格式传送给外部编辑器，以跳转到指定文件和行号，如编译出错时进行的关联修改。

● Ultraedit

大多数程序员都用过通用编辑器 Ultraedit，可以将它与 Tornado 环境集成，如图 10-3 所示，具体步骤如下。

- ✧ 指定命令路径，可以用命令行后的按钮来查找。
 - ✧ 传入参数，格式为“\$filename/\$lineno”。
 - ✧ 选择“Invoke from”选项，确定 Ultraedit 的激活时机。
- 设置完成后，就可以像 Tornado 内部编辑器一样使用。

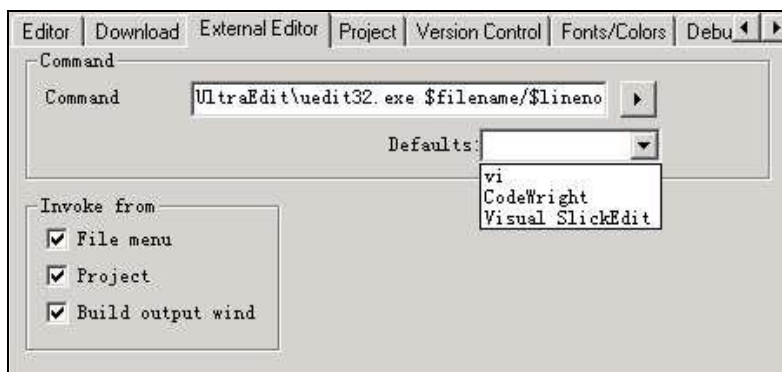


图 10-3 集成编辑器

● Source Insight

虽然 Ultraedit 是一个不错的编辑器，但并不是专门用于代码编写的，相比之下，Source Insight 更适合程序员。Source Insight 提供更合适的功能用于软件开发，如支持工程管理、头文件函数原型、源代码解析、语法补齐等，使用它可以快速进行代码编辑和浏览。另外，它本身也是个 IDE 环境，可以集成外部工具，如 Pclint 等。Source Insight 还有许多其他方便功

能，有兴趣的用户可以自己探索。不过，Source Insight 对中文输入支持不是太好。

Source Insight 为共享软件，能试用 30 天，可以到 <http://www.sourceinsight.com> 下载，最新版本为 Version 3.50.0034 - September 8, 2003。

Source Insight 也能集成到 Tornado 中，与 Ultraedit 的加入类似，只是传入参数的格式略有不同，为“-i +\$lineno \$filename”。其中参数选项“-i”用来指明如果有 Source Insight 运行，不启动新程序实例。如果代码文件归属于某个工程，该工程会被自动打开。

Source Insight 有很多好用的功能，下面介绍一个与 VxWorks 关联的例子，让开发者的应用代码工程能识别 VxWorks 系统函数的原型。

首先需要将 VxWorks 的头文件建立为一个基础工程(vxworks_x86)，再将这个工程加入工程符号搜索路径，如图 10-4 所示。符号搜索路径由所有 Source Insight 工程共用，如果有多个平台或版本的 VxWorks 头文件，这就不太适合。

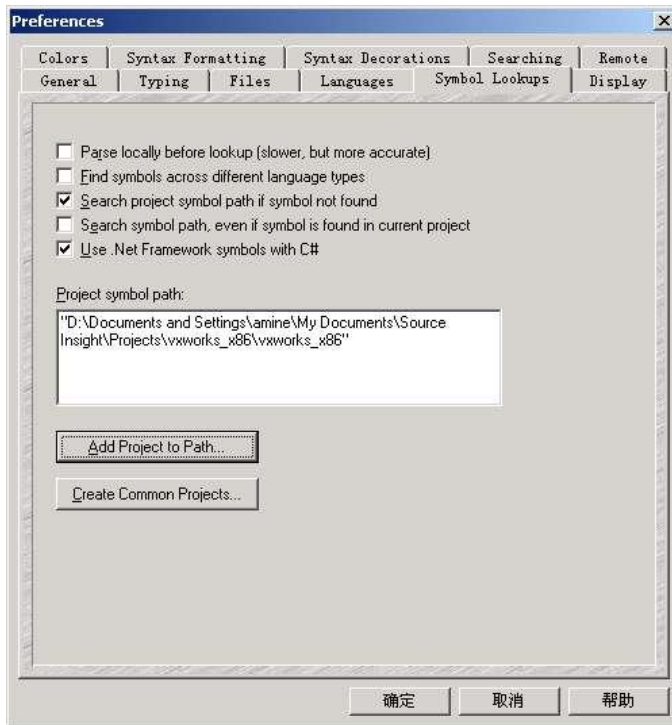


图 10-4 将头文件工程加入符号搜索路径

头文件符号加入搜索路径后，就可方便查看 VxWorks 系统函数的原型，这在填写函数参数时特别有用，可省去库手册查找。如图 10-5 所示，“Context”窗口显示了 taskDelay 的函数原型，不过有两个，因为 Source Insight 不能很好地解释 VxWorks 头文件中的__STDC__宏和__cplusplus 宏。

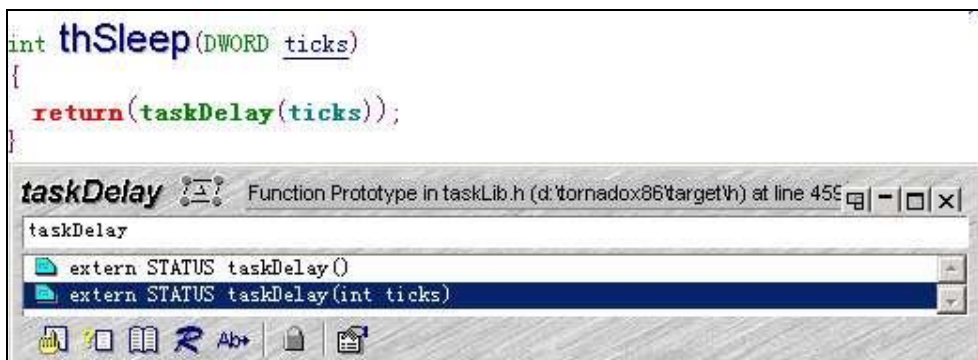


图 10-5 函数原型自动显示

有了系统函数符号，还能使用语法自动补齐功能，如图 10-6 所示，输入“task”后，编辑器就自动列出了所有以“task”开头的符号供选择，且“Context”窗口还显示了具体的符号信息。

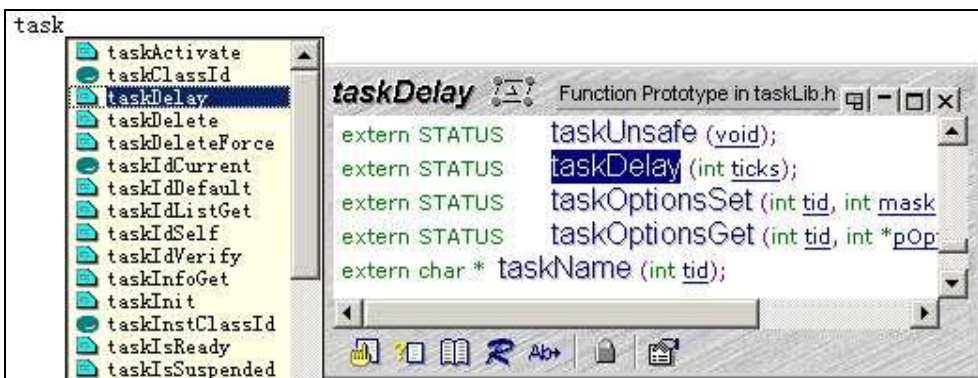


图 10-6 函数语法自动补齐

10.1.5 版本控制

版本控制是软件工程管理的的重要组成部分，负责管理源代码文件、配置文件、设计文档的升级。Tornado 环境可以集成版本控制软件，缺省支持 ClearCase、Visual Source Safe 和 PCVS，如图 10-7 所示，该窗口可以选择菜单“Tools→Options”命令打开。Tornado 提供两个参数：\$filename 和 \$comment，用于传入归档文件名和归档描述。

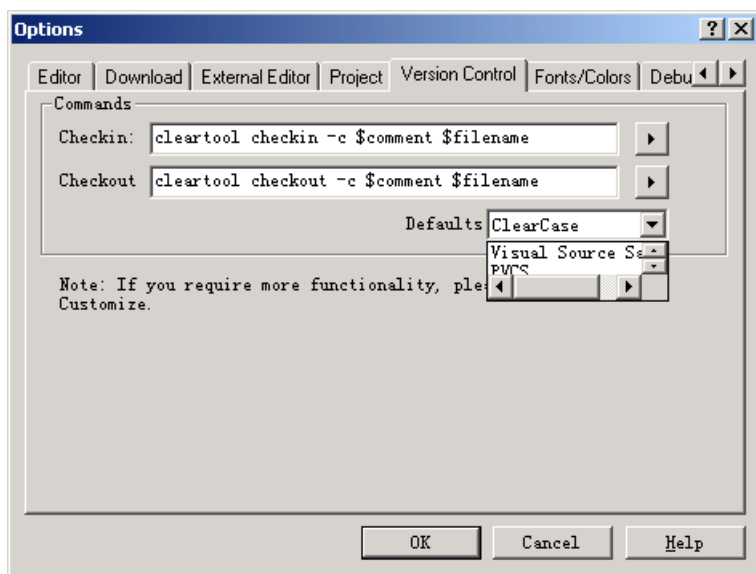


图 10-7 集成版本控制软件集成

除了上面提到的商业版本控制软件,还有共享软件和自由软件可选择,如 CS-RCS 和 CVS 等。

● CS-RCS

CS-RCS 为共享软件,可以从“<http://www.componentsoftware.com>”下载,个人版使用是免费的。CS-RCS 遵循老的 RCS 规范,提供了方便的图形操作界面,用户可以快速上手使用。CS-RCS 特别适合个人软件开发,也支持小规模开发组。CS-RCS 可以在 Windows 98 平台上建立服务器端,使用 Windows 98 的共享文件夹机制,不过安全性存在问题,只能在独立局域网中使用。该公司除了提供 CS-RCS 相关产品外,还提供文档比较工具 (CS-DIFF) 和 CVS 客户端工具 (CS-CVS)。

● CVS

早几年开发中,笔者使用 CS-RCS 作为版本控制工具,现在已转向 CVS。CVS 提供了更丰富的功能,特别适用于分布式开发,能极大地提高开发组协同工作效率。不过使用比 RCS 略复杂一些,下面简略描述其使用过程,更详细的用法可以参考专门文档。

- ✧ 下载服务器端 CVSNT, 最新稳定版为 2.0.10, 网站为 <http://www.cvsnt.org/wiki/>。
- ✧ 下载客户端 WinCvs, 最新版为 13b13-2, 网站为 <http://sourceforge.net/projects/cvsgui/>。还有其他的客户端可选择, 如前面提到的 CS-CVS。
- ✧ 下载 WinCvs 需要的 python, 最新版为 2.3, 网站为 <http://www.python.org/>。
- ✧ 在服务器端使用管理员账号安装 CVSNT。安装后, CVSNT 会自动启动两个服务, cvsservice 和 cvslock。可运行“service control panel”来启动和停止服务,以及配置 CVSNT, 如图 10-8 所示。修改 CVSNT 配置时, 应该停止服务; 修改完成后, 再重新启动服务。

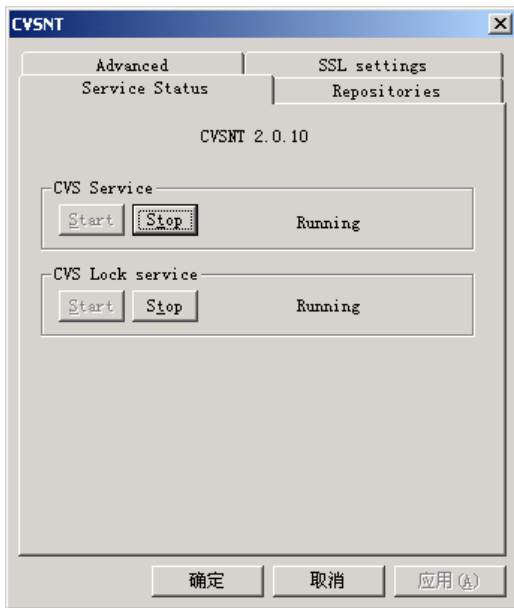


图 10-8 cvsnt 服务控制

- ✧ 设置 “Repository” 目录，用于存储归档文件。一般先设目录前缀（如 e:/），再添加各分目录（如/cvsrepo），如图 10-9 所示。存储目录建立时会自动创建 cvsroot 子目录。

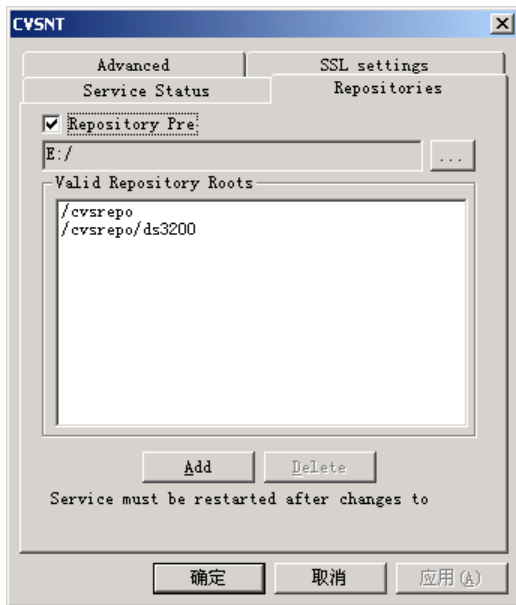


图 10-9 cvsnt 存储目录设置

- ✧ 在 Advanced 页面，选择 “Use local users for pserver...”; 设置 “Temporary” 栏中临时目录（如 E:\cvstemp），如图 10-10 所示。

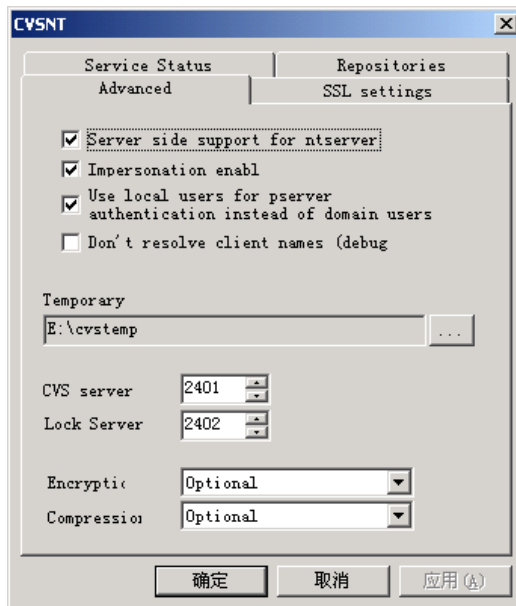


图 10-10 cvsnt 高级选项设置

- ✧ 在服务器中添加客户端用户。先在 Windows 2000 中添加各用户帐号，归入 guest 组；再进入 Dos 窗口中，为每个用户设置密码，密码可以和 Windows 2000 中配置的用户密码不同。administrator 可以使用 Windows 2000 中密码，不用特别添加。Dos 窗口的操作如下。

```
D:\>set cvsroot=:ntserver:192.168.0.112:/cvsrepo
D:\>cvs passwd -a amine
Adding user amine@192.168.0.112
New password: ****
Verify password: ****
```

其中 cvsroot 的 IP 地址为服务器的，“/cvsrepo”为“Valid Repository Roots”（见前面的 cvsnt 设置），而不是存储目录的绝对路径。

- ✧ 在客户机器上安装 python 和 WinCvs。注意不要在工作盘上使用“cvs”作为目录名，WinCvs 使用“cvs”目录来建立和服务器的联系。
- ✧ 配置 WinCvs。选择菜单“Admin→Preferences”命令激活配置窗口，如图 10-11 所示。主要需要配置授权方式、存储目录、服务器地址和用户名。

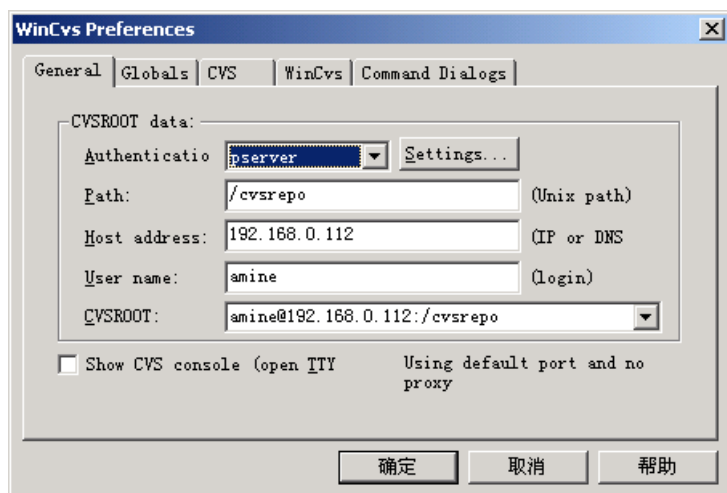


图 10-11 WinCvs 一般设置

- ✧ 在 WinCvs 中可挂接外部文档比较工具和编辑器，如图 10-12 所示。在使用 WinCvs 比较功能“diff selection”时，注意选择“use the extern diff”。

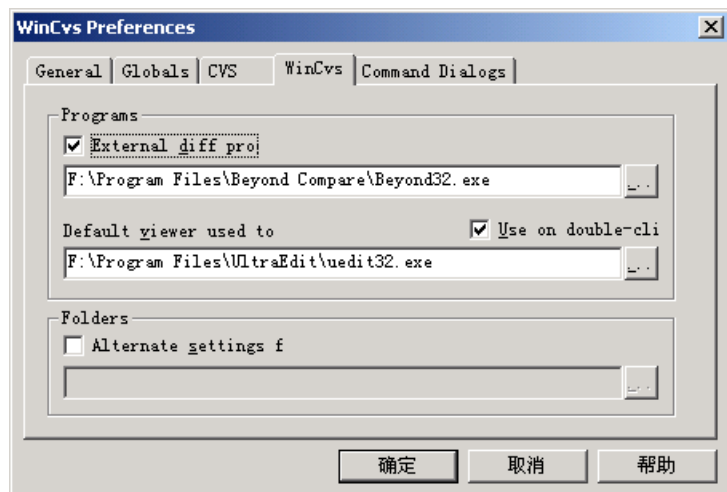


图 10-12 wincvs 外部工具设置

- ✧ 在 WinCvs 中选择菜单“Admin→Login...”命令登录服务器，密码只需要输入一次，再次启动会自动登录。登录成功显示：“***** CVS exited normally with code 0 *****”。选择菜单“View→Browser loc→Change”命令切换到自己的工作目录，导出服务器代码的工作副本。
- ✧ 管理员平时应该用普通用户登录；实现管理功能时才用 administrator 登录。administrator 和普通用户最好使用不同的工作盘。
- ✧ 读者可以参考相关的文档，掌握 WinCvs 的常用操作。

为了更好地使用 CVS 的归档记录功能，源代码文件头应该加入 CVS 关键字引用。笔者常用的文件头如下（归档前）。“rcsid”用于标示单个目标模块的版本，头文件中不使用。

```
/*-----
$Log:$
```

```
-----*/
static const char rcsid[] =
"$Id:$";
```

归档后，文件头由 CVS 扩展如下。注意不要使用中文归档描述，否则生成的 ChangeLog 会显示异常。

```
/*-----
$Log: HELLO.C,v $
Revision 1.2  2003/11/19 09:35:40  amine
Hello again

Revision 1.1.1.1  2003/09/24 09:14:08  administrator
cvs used for the first time

-----*/
static const char rcsid[] =
"$Id: HELLO.C,v 1.2 2003/11/19 09:35:40 amine Exp $";
```

10.1.6 静态检查

嵌入式系统特别强调可靠性，为了编写正确可靠的代码，程序员总小心翼翼，编码时反复检查，空闲时也常为代码中可能的 Bug 担忧。大脑的模糊逻辑与代码的精确性是个矛盾，开发者不可能直接写出无错的代码，通常需要借助编译器和调试器来排除代码的错误。但他们也不能完全解除程序员的忧虑，大多数软件都包含 Bug 就证明了这点。

代码中大多难查找的 Bug 一般不是语法错误，而是逻辑错误，不能用编译器排除，而需要程序员自己分析排除。但大量的代码，若不能有的放矢，则工作量太大而不能实际完成。代码静态检查工具能完成比编译器更严格的检查，提供可能的 Bug 线索，是程序员减少忧虑的另一个帮助。

PcLint 是最常用的静态检查工具，笔者见到的最高版本为 8.0h。PcLint 能检查多种常见的隐含错误，能按代码工程进行整体检查，能适用于多种编译器和开发环境，包括 Tornado 使用的 gnu 和 diab 编译器。

要使用 PCLint，主要需要根据编译器和 VxWorks 库头文件来修改 lnt 文件，其为文本文件，可以包含使用。如下面的“std.lnt”内容。

```
//Standard lint options
co-gnu.lnt
options.lnt -si4 -sp4
```

PcLint 为命令行工具，基本命令格式如下，lnt 文件和源代码的路径需要指定。

```
>lint-nt xxxx.lnt xxxx.c xxxx.c
```

PcLint 可以和编辑器 Source Insight 集成，具体步骤可以参考 PcLint 中“Env-si.lnt”文件中的描述。PcLint 具体用法和选项细节参考其附带的文档。


10.1.7 代码文档


编写代码时，注释和说明应遵循一定的格式，以方便将来提取代码文档。这样做有较多好处，如代码完成时，文档也同时完成，保证了文档的及时性；又如代码修改时，文档也同

时修改，保证了文档的准确性。VxWorks 库参考等文档就是从代码中直接提取的，所以说编程指南可能过时，而库参考还是及时准确的，实际编程时应该多查阅库参考。

WindRiver 使用 refgen.bat 来从代码中提取文档，存为 HTML 格式。例如，在命令行下，进入 “\target\unsupported\tools\mangen\” 目录，用命令 “refgen sample.c” 来提取 sample.c 中的文档，存为 sample.html。

VxWorks 代码的编写规则，可以参考相关文档。如果用户的代码文档也想遵循这种规则，可以参考 sample.c 中示例。

 WindRiver, “VxWorks 5.4 Programmer’s Guide” 的 I.3.8 章节。

 WindRiver, “BSP Developer’s Kit User’s Guide” 的第 9 章。

● Lccwin32 文档格式

早在使用 VxWorks 之前，笔者就使用 Lccwin32 工具来生成代码文档。

可以从 “<http://www.q-software-solutions.com/lccwin32/>” 下载 Lccwin32，最新版本为 “3.3”。Lccwin32 是一个免费工具，只有源代码和附件需要付钱。其实 Lccwin32 是一个不错的 Windows 集成开发环境，包括编译器、编辑器、版本控制和一些库等，并且有强大的 C 源代码分析功能，能形成程序结构的图形显示，比 Source Insight 的源代码分析功能更好，只不过不支持 C++ 代码，代码编辑功能也不如 Source Insight 好。在没有使用 Source Insight 前，笔者一直使用 Lccwin32 作为编码环境。

Lccwin32 的文档格式比 VxWorks 的简单，主要是文件头和函数头，如下所示：

```
/*-----
Module:      os.c
Author:      amine
Project:     dsxxxx
State:       创建
Creation Date: 2002-07-03
Description:  OS 抽象层通用接口
-----*/

/*-----
Procedure:   ThreadInit ID:1
Purpose:     初始化抽象层环境
Input:
Output:      OK : ERROR
Errors:
-----*/
```

Lccwin32 生成的代码文档为标准 Windows 帮助文件格式，老版本的生成结果还需要经过 Help WorkShop 编译，才能生成帮助文件。

10.1.8 UML

UML[统一建模语言]是一种定义良好、易于表达、功能强大且普遍适用的建模语言。1997 年 11 月 17 日，OMG 采纳 UML 1.1 作为基于面向对象技术的标准建模语言，最新版本为 UML1.3 版本。UML 采用图形化语言描述了系统，分析了系统，也实现了系统。具体的说，它采用 9 种不同的模型（图形），即用例图、类图(含包图)、对象图、组件图和配置图等五个图形实现静态建模机制，用状态图、活动图、顺序图和合作图实现动态建模机制。特别是状态图，适合实时系统，方便描述各种复杂的状态。

开发一个具有一定规模和复杂性的软件系统和编写一个简单的程序大不一样。大型的、复杂的软件系统的开发是一项工程，必须按工程学的方法组织软件的生产与管理，必须经过分析、设计、实现、测试、维护等一系列的软件生命周期阶段。编程仍然是重要的，但是更具有决定意义的是系统建模。只有在分析和设计阶段建立了良好的系统模型，才有可能保证软件工程的正确实施。随着嵌入式系统日趋复杂，软件规模和开发团队的日益扩大，基于 UML 的软件开发已经广泛应用。

Tornado 环境主要辅助用户完成编码阶段的工作，不能覆盖整个软件开发过程，如需求分析、系统设计、模式设计等阶段。而一些面向嵌入式实时系统的 UML 工具，如 Rhapsody、Rational Rose、Matlab 等，能覆盖这些阶段，可以作为 Tornado 环境的补充，完成整个软件的开发过程的支持。Douglass 提出了用于嵌入式系统的快速面向过程[ROPES]，可指导项目开发过程，其公司软件为 Rhapsody。

● Rhapsody

Rhapsody 是一个基于标准 UML 的实时应用软件开发环境，由 I-Logix 公司出品，最新版本为 5.0。Rhapsody 主要面向嵌入式实时系统开发，在众多领域有广泛应用，如军事、通信、工业控制、消费电器等。据说，NASA 的火星探路者航天器就是运用 Rhapsody 和 VxWorks 进行应用程序开发的。Rhapsody 使需求分析与跟踪、软件设计、配置管理、代码实现和功能测试融为一体，提供了可视化的开发环境，贯穿了工程化的设计思想，使用了自动化的开发模式，并支持团队化的协作开发。

Rhapsody 提供如下主要特性。

- ✧ 图形编辑器支持用例图、顺序图、协作图、对象模型图、活动图、状态图和组件图。
- ✧ 可以设计出非常复杂的状态图，适合实时系统的多状态机处理。
- ✧ 浏览器能概观整个模型。
- ✧ 内在地支持迭代式的开发过程。
- ✧ 能为多种实时平台（包括 VxWorks）产生高质量代码，如图 10-13 所示。
- ✧ 面向对象技术可直接采用 C 语言来实现，保留了 C 语言的灵活高效性。
- ✧ 自动生成格式可定制的中文文档。
- ✧ 提供操作系统抽象层，方便各种实时平台上的代码移植。
- ✧ 能保证设计模型、产生代码和文档的同步。
- ✧ 基于模型的软件测试，支持设计测试、单元测试和回归测试。
- ✧ 支持设计阶段的调试，而不用等到编译阶段。调试过程能用动态图形显示。
- ✧ 集成了 VBA。

安装 Rhapsody 需要选择面向的平台，如图 10-13 所示。

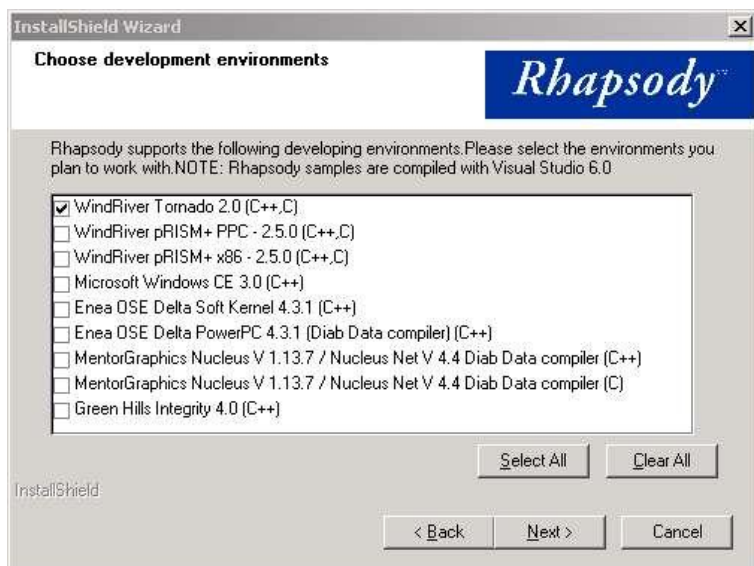


图 10-13 Rhapsody 支持的多种实时平台

Rhapsody 采用以下分层结构,如图 10-14 所示,使之非常方便地支持各种实时操作系统,包括 VxWorks、pSOS 等,而且具有很好的可扩展性。其中 OS Abstraction Layer[操作系统抽象层]对所有实时操作系统进行了抽象和封装,提供包括任务、信号量、消息队列和定时器等基本服务。OXF[对象执行框架]对这些资源进一步封装,为上层应用程序提供更高级的统一的编程接口。这样,当底层硬件修改时,上层应用程序不需要做任何修改。另外,用户可以自己定制修改 OXF 适用于自己的项目。

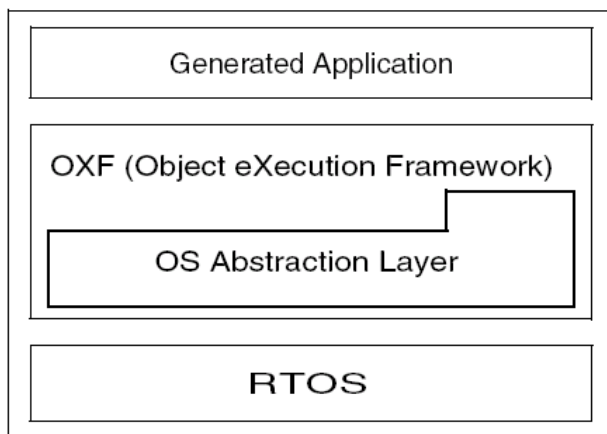






图 10-14 Rhapsody 代码分层结构

-  I-Logix, Rhapsody 的“RTOS Adapter Guide”文档 (rtos.pdf)。
-  秦遵明、黄峰, CASE 工具 Rhapsody 在综合接入服务器中的应用。
-  孙强、张振华, 使用 Rhapsody 软件框架和 UML 的实时系统开发。
-  Santa Clara 大学 COEN 课程, <http://cseserv.engr.scu.edu/NQuinn/>。

10.1.9 其他功能扩展

本节介绍一些小工具,以进一步补足 Tornado 功能,完成整个软件开发过程。

● 代码搜索

在软件开发过程中，不可避免地要阅读大量源代码，包括自己的和别人的。通过阅读好的源代码，可以学习编程技巧和理解系统深层机制。而源代码包一般都比较庞大，如 Linux 或 VxWorks 源代码，因此开发者不可能通读所有代码，只能查阅感兴趣的部分。

代码搜索工具是代码查阅的必要工具，笔者一般使用 Agent Ransack 软件来搜索代码。与 Windows 本身提供的文档搜索功能相比，该工具适合用于代码搜索，比如消耗的资源少，可以显示包含关键字的行文本和行号，更多的搜索选项，支持正则表达式，能保存搜索结果等。并且该工具也集成到右键菜单中，可以方便调用。

在搜索和编辑 VxWorks 代码时，要注意 VxWorks 代码文件的行结束是 Unix 方式的，只有换行符 (LF)，没有回车符 (CR)。选择 Agent Ransack 的菜单 “Search→Configuration” 命令，设置行结束符为 Unix 方式的，这样才能看见包含关键字的多行。使用 UltraEdit 时，也要注意该问题，需禁止其 “Unix/Mac file detection” 功能。

● 代码比较

在代码升级和归档过程中，经常需要对照新老文件，就需要代码比较工具。比较工具可以单独使用，也可集成到版本控制软件中。这里推荐几个比较工具。

免费软件有 Diffutils、Winmerge 和 CS-DIFF。Diffutils 是 GnuWin32 工程中的一部分，可以从 Sourceforge 免费下载，还提供补丁功能。Winmerge 提供图形对比窗口，很容易找到两文件的不同之处，是比较好的选择。CS_DIFF 一般随前面提到的 CS-RCS 一起发行，也有单独发行版本，可以免费使用。使用单一窗口显示比较结果，用起来还算方便。

商业软件比较好的有 Beyond Compare 和 Examdiff Pro，都提供图形对比窗口，使用比较方便，都可以和 WinCvs 集成。

这些工具一般用于比较纯文本文件，其他格式的开发文档的对比需要使用专门的工具。如 Rhapsody 就提供专门的比较工具和补丁合成工具来管理其专用文档。

● 代码补丁

如果产品以源代码形式发布，发布升级版本可能需要使用代码补丁工具。使用补丁工具，用户只需要发布代码的升级部分，而不用重新发布整个代码，程序补丁也可以安全的在网上发布，Linux 就采用这种补丁方式。

一般比较工具都提供补丁生成功能。如 CVS Diff 和 Examdiff Pro 等。有的还提供补丁合成功能，如 Diffutils。也有专门的补丁工具软件，如 WinPatch 和 Cygwin 的 patch 等。

注意有些比较工具生成的补丁文件的行结束为 Windows 风格的，而不能由某些补丁合成工具使用，需要先转换为 Unix 风格行结束。

● Bug 跟踪

代码在开发过程中，不可避免地会出现各种 Bug。Bug 需要跟踪处理，以保证 Bug 能得到及时的修正，各涉及分支版本都能同时修正，发布版本和 Bug 修正才能一致。WindRiver 的 SPR 和补丁发布就是 Bug 管理的一个最好例子。Bug 管理工具帮助开发者完成这个工作。

Mozilla 公司提供了一个共享的免费工具 Buzilla。作为一个产品缺陷的记录及跟踪工具，它能够为用户建立一个完善的 Bug 跟踪体系，包括报告 Bug、查询 Bug 记录并产生报表、处

理解解决、管理员系统初始化和设置 4 部分。

Buzilla 对 Bug 的处理流程如下，Bug 的生存周期如图 10-15 所示。

- ✧ 发现 Bug 后，判断归属模块，填写 Bug 报告，Email 通知项目组长或直接通知开发者。
- ✧ 项目组长根据具体情况，重新分配[Reassigned]给 Bug 所属的开发者。
- ✧ 开发者收到 Email 信息后，判断是否为自己的修改范围。
- ✧ 若不是，重新分配给项目组长或应该分配的开发者。
- ✧ 若是，进行处理，查找原因并给出解决方法[Resolved]（可创建补丁附件及补充说明）。
- ✧ 测试人员查询开发者已修改的 Bug，进行重新测试（可创建 test case 附件）。
- ✧ 经验证无误后，修改状态为 Verified。待整个产品发布后，修改为 Closed。
- ✧ 若还有问题，状态重新变为 New，并发邮件通知责任人员。
- ✧ 若 Bug 一周内没被处理，Bugzilla 就会一直用 Email 告知它的属主，直到 Bug 被处理。

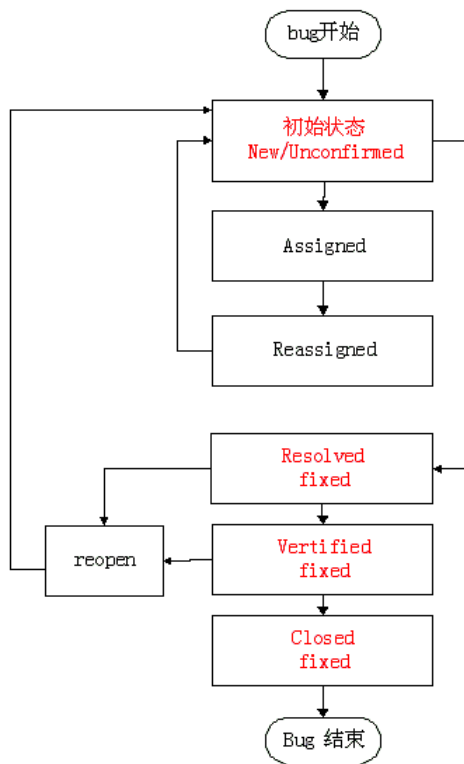



图 10-15 Bug 的处理流程

Elementool 是另一个基于 Web 的 Bug 跟踪工具，能跟踪最新 Bug，对 Bug 分级，将 Bug 处理分配给指定的开发人员和产生 Bug 报告等。另外，Elementool 还提供客户信息管理和工程时间表管理功能。

其他的工具还有 Rational ClearQuest 和 Red-gate 的 Aardvark 等。

 徐异婕，测试跟踪工具 Bugzilla 介绍，UML 软件工程组织，2002.05。

10.2 调试实践

嵌入式系统调试和普通主机程序的调试不同。主机一般具有丰富的资源，调试器和被调试程序可以同时主机上运行。而嵌入式系统的资源有限，一般不能满足图形界面的调试器

的资源需求，并且调试器运行还会影响嵌入式系统本身程序的运行，所以嵌入式系统调试器一般分前端和后端两部分，图形界面前端在主机上运行，简洁的后端在目标机上运行，两者用某种协议通信，采用交叉调试方式。

VxWorks 系统的调试和有些嵌入式系统的调试还有所不同。VxWorks 创建的多任务环境，代码有自己的运行上下文，支持任务断点和全局断点，而一般系统只支持全局断点。VxWorks 使用标准通信通道（如网络和串口等）作为调试途径，支持任务模式和系统模式（和前面的断点类型对应），而一般系统使用硬件仿真器，只支持系统模式。VxWorks 提供的目标机 Shell 是个 C 解释器，能脱离通常意义的调试器，完成某些基本调试功能。VxWorks 还提供 WindView 组件，监视系统运行过程，能完成某些错误的事后诊断工作。

10.2.1 多任务调试

在 VxWorks 的多任务调试环境中，要注意区分断点类型和调试模式，与其他的嵌入式开发系统中的调试不同。

在任务调试模式下，全局断点会在任何任务上下文中都起作用，任务断点只在调试器绑定[attach]的任务上下文中起作用。如下面的代码中所示，任务 tPlow 和 tPhigh 都调用 useResource 函数，在该函数中若设置全局断点，两个任务都会停下；若设任务断点，只会停下正调试的任务。任务模式的断点，只挂起相关的任务，而不影响整体系统的运行。下面将举例描述 VxWorks 这种独特的任务断点调试方式。

在任务调试模式下只能调试任务上下文中的代码，不能调试中断处理函数中的代码。这和调试通信通道有关，任务调试时通信通道使用高效的中断方式，而中断调试时，这种方式不再可用，必须使用查询通信方式，即进入系统调试模式（attach system）。在系统调试模式中，所有代码（包括任务代码和中断代码）为一个整体，断点和其他嵌入式开发系统类似，不区分类型。任务代码不区分运行上下文，作为一个整体调试，若遇到断点，整个系统都会停止运行而等待调试，这与大多数嵌入式开发系统类似。

● 多任务调试示例

任务模式下多任务调试是 Tornado 的一大特色，可以在主机上启动多个 Tornado，同时调试多个任务上下文中的代码。下面的例子说明该过程。

✧ 配置调试器，选择菜单“Tools→Options...”命令激活配置对话框，如图 10-16 所示。

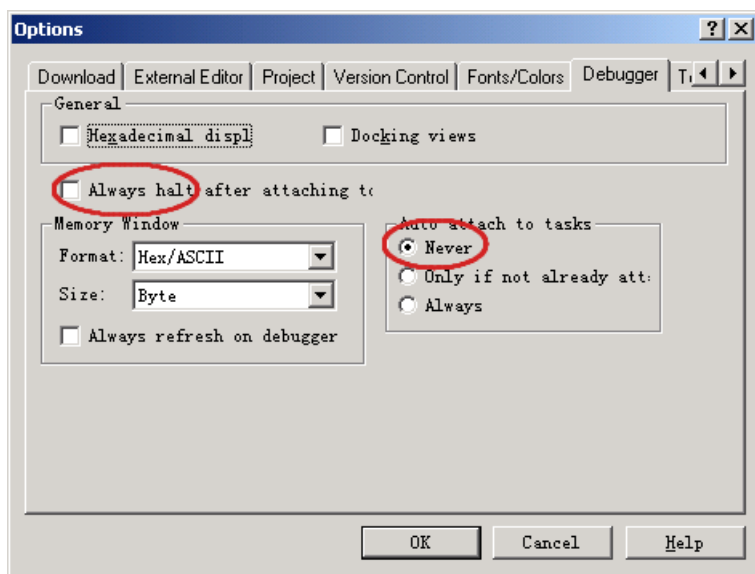


图 10-16 调试器配置

主要需要配置两项，连接任务时不挂起任务和遇到断点时不自动连接任务，如图 10-16 所示的红圈标记。如果用户习惯原来的调试方式（不区分任务），可设置“Auto attach to tasks”项为“Always”，Tornado 就会根据断点自动选择任务连接调试。但这种方式不适合多任务调试，会混乱多个调试器的独立调试。

✧ 代码文件 `inversion.c`，也用于后面的 Shell 调试和 WindView 调试描述。

```
/*优先级逆转代码示例*/
#include "vxWorks.h"
#include "semLib.h"
#include "taskLib.h"
#include "kernelLib.h"
#include "stdio.h"

/*条件编译决定互斥信号量是否是优先级逆转安全的*/
#define PRINV

int resource; /*共享资源*/
SEM_ID semId; /*保护共享资源的信号量*/
int debug; /*调试标志,用来终止各任务*/

/*初始化共享资源*/
int initResource(void)
{
    /*保护共享资源的互斥信号量创建*/
#ifdef PRINV
    semId = semMCreate (SEM_Q_PRIORITY | SEM_INVERSION_SAFE);
#else
    semId = semMCreate (SEM_Q_PRIORITY);
#endif
    if (semId == NULL) return ERROR;
    /*共享资源初始化*/
    resource = 0;
}
```

```

    return OK;
}
/*使用共享资源,time-占用时间, todo-共享资源操作*/
void useResource(unsigned int time, int todo)
{
    int i;
    semTake (semId, WAIT_FOREVER);
    resource = todo;
    for (i = 0; i < time; i++ ) {}
    resource = todo + 1;
    semGive (semId);
}
/*最低级任务和最高级任务共用函数*/
void t_low_high (int id)
{
    while(debug){
        useResource(10000, id);
    }
}
/*中级任务函数*/
void t_mid (void)
{
    int i;
    while(debug){
        for (i = 0; i < 20000; i++ ) {}
    }
}
/*入口函数, 启动 3 个任务*/
void start (void)
{
    int    t1Id, t2Id, t3Id;
    char *pstr; /*未初始化指针*/

    debug = 1;
    if (initResource() == ERROR) exit(0);
    t3Id  = taskSpawn ("tPlow", 203, 0, 1000, (FUNCPTR)t_low_high, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0);
    taskDelay(1); /*让低优先级任务先运行*/
    t2Id  = taskSpawn ("tPmid", 202, 0, 1000, (FUNCPTR)t_mid, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
    t1Id  = taskSpawn ("tPhigh", 201, 0, 1000, (FUNCPTR)t_low_high, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0);
    /*这句话会引起任务异常, 用来演示 Shell 调试功能*/
    /*strcmp(pstr, "amine");*/
}
/*停止 3 个任务*/
void stop (void)
{
    debug =0;
}

```

- ✧ 编译 inversion.c 生成 inversion.o。
- ✧ 启动 Terget Server 和目标机连接，下载 inversion.o。
- ✧ 选择 Terget Server 启动调试器，在 start 函数入口处设置任务断点。
- ✧ 选择菜单 “Debugger→Run...” 命令执行 start 函数。函数在 tDbgTask 任务上下文中

运行，任务优先级为 100。

- ✧ start 函数会在入口断点处挂起，单步运行初始化函数和创建任务“tPhigh”。
- ✧ 在主机为“tPhigh”任务启动另一个 Tornado (2)，连接 target server，启动调试器，在 useResource 函数开始处设置断点，选择菜单“Debug→Attach...”命令连接“tPhigh”任务，任务会在断点处停止。
- ✧ 在原 Tornado (1) 窗口中继续单步运行 start 函数，创建任务 tPmid 和 tPlow。
- ✧ 为任务 tPmid 和 tPlow 分别启动 Tornado (3/4) 程序，连接 Target Server，启动调试器，连接任务。
- ✧ 现在一共启动了 4 个 Tornado 程序，对应 4 个任务 tDbgTask、tPhigh、tPmid 和 tPlow，在 Tornado 的右下脚状态条有连接的任务显示，如“tDbgTask(0x78eebc)192.166.0.4@amine:Stopped”。
- ✧ tDbgTask 和 tPhigh 任务正处于断点等待调试，任务状态为 SUSPEND。tPmid 和 tPlow 任务正常运行。
- ✧ tPhigh 和 tPlow 共用函数 useResource，虽然函数中已经有断点，但只在 tPhigh 任务上下文有效，不影响 tPlow 运行。若想调试 tPlow，需要在 Tornado (4) (与 tPlow 任务上下文关联) 中设置断点，比如在 useResource 中的 semGive 处设置断点，tPlow 任务会挂起等待调试。
- ✧ 若想结束调试，先去掉各断点，使相应的任务运行，调用 stop 函数设置 debug 变量为 0，各任务会退出系统。

通过这种方式，可以同时调试多个任务，进行设置断点、单步、变量查看等操作，相互之间没有影响。

通过上面的调试，代码看起来工作很正常，但实际上却隐含着一个著名的 Bug (曾在火星探路者上出现)，这需要另外的调试方式才能排除，详见后面的介绍。

10.2.2 Shell 调试功能

除了使用标准的调试器功能来排除程序错误，VxWorks 的 Shell 也有简单的调试功能 (WindSh 也有类似功能)，如各种信息显示、任务操作、内存操作、寄存器操作、断点设置等。在 Shell 中使用 help 和 dbgHelp 等命令可以显示相关调试命令的帮助，有些命令需要库支持，如符号表和一些 show 函数。另外，Shell 是一个 C 解释器，具有丰富的功能，如函数调用、变量赋值、符号地址操作等，对调试工作也有很大的帮助。

在某些情况下，Shell 调试功能显得必不可少。比如，一些运行时错误，一闪而过或很久才出现，调试器不能捕捉住，而 Shell 可以将错误信息送到主机显示，用户可以根据信息判断出错任务和出错代码位置，再进一步分析代码和调试排错。又如，发布到现场的设备出现错误时，一般没有可用的开发主机，设备上的 Shell 可以帮助用户排除简单的错误。再如，设备测试时，需要长期的烤机运行，可用 Shell 方便记录运行信息和捕捉异常；也可以编写一些测试脚本文件，加入测试命令和数据，利用 Shell 的解释功能执行，并将结果输出到 Shell，再由主机程序检查，完成一些自动测试功能。

下面举一个简单的例子来说明如何使用 Shell 来排除运行时错误。仍然使用上节的代码段，start 函数中最后的 strcmp 调用可能产生异常。错误很简单，因为串指针局部变量 pstr 未初始化，也许读者一看就知道。不过这只是简单的示例，实际中代码更复杂庞大，简单的

错误也会被隐蔽而不好发现。这个错误的另一个复杂性在于 pstr 的值是不定的，有时 pstr 指向一个数据串有合法结束符，则 strcmp 函数不会出错。

这个错误笔者在实际中遇到过。strcmp 在系统启动时由 tRoot 任务调用。pstr 有时初始化，有时未初始化，strcmp 产生异常情况很少，且只会在系统启动时出现。即使产生了异常，大部分系统功能也不受影响，开发人员察觉不到错误。所以，直到系统开发完成半年后，才由 Shell 捕捉到这个错误，并由 Shell 信息准确定位了错误而解决。

同前节描述的过程一样，下载目标模块，但不启动调试器，而启动 WindSh。

在 WindSh 中直接运行 start 函数，出现如下错误。

```
-> start
Exception number 2: Task: 0x78eebc (t1)
Access Fault/Bus Error
Format          : 0x4c08
Status Register : 0x3000
Program Counter : 0x6c0a2
5be7a   vxTaskEntry   +10 : funcCallWrapper (7fedcc, 0, 0, 0, 0, 0, 0, 0, 0, 0)
aabd4   funcCallWrapper+76 : start ([0, 0, 0, 0, 0])
lca5f6   start         +d8 : strcmp (eeeeeeee, lca623)
value = 0 = 0x0
```

从上面信息可以看出，在 t1 任务上下文，start 函数中运行到 strcmp 出错。信息中提供了出错任务名、PC 程序指针、函数轨迹和函数参数，这些信息对错误的定位很有帮助。若用户在开发中看到这样的错误信息，应该及时保存。

若错过了上面的信息，也可在 Shell 用 i 命令看到 t1 任务处于异常状态。

```
-> i
NAME      ENTRY      TID    PRI    STATUS    PC      SP      ERRNO  DELAY
-----
tExcTask  excTask          7fc054  0  PEND      cba2a   7fbfa0      0      0
tLogTask  logTask          7f9728  0  PEND      cba2a   7f9674      0      0
tShell    shell            7917bc  1  READY     b2004   791410      0      0
tWdbTask  a9506            799988  3  PEND      5b61e   7998e0      0      0
tNetTask  netTask          7c3450  50  PEND      5b61e   7c33e4  450001    0
tTelnetd  telnetd          79c084  55  PEND      5b61e   79bf60      0      0
tTffsPTask flPollTask      7f7cdc  100  DELAY     b18ac   7f7c90      0      4
tPhigh    lca4c8           7febec  201  READY     1ca4a8   7feba8      0      0
tPmid     lca4ee           7fe648  202  READY     5be6a    7fe620      0      0
tPlow     lca4c8           7fe0a4  203  READY     5be6a    7fe07c      0      0
```

可再用 tt 命令追出部分信息。

```
-> tt t1
5be7a vxTaskEntry   +10 : aab5e (7fedcc, 0, 0, 0, 0, 0, 0, 0, 0, 0)
aabd4 wdbFuncCallLibInit+e4 : lca51e ([0, 0, 0, 0, 0])
lca5f6 _func_taskRegsShowRtn+69e6a: strcmp (eeeeeeee, lca623)
```

如果出错处是自己编写的函数，还可用 PC 值准确定位出错代码位置，如下面 value 显示。

```
-> 0x6c0a2
value = 442530 = 0x6c0a2 = strcmp + 0xe
```

系统函数调用出错一般是由于传入不正确的参数导致的，用户可使用内存查看命令来检查参数。先检查 strcmp 的后一个参数，内容显示正常。

```
-> d 0x1ca623
```

```
001ca620:      0061 6d69 6e65 0000 0000 0000 0010  *   amine.....*
001ca630: 0000 0000 0100 017c 1a0c 0f04 9a01 0000  *.....|.....*
```

其实 strcmp 的第一个参数的错误是很明显的。如果查看该内存，WindSh 会提示错误，而 tShell 中会产生异常。

```
-> d 0xe0000000
WTX Error 0x100ca (AGENT_MEM_ACCESS_ERROR)
value = -1 = 0xffffffff
```

调试到这里，可以清楚地知道了出错的任务、代码和原因。再分析相关代码，并在调试器做些跟踪，就可以解决问题。

上面示例中只描述了很少的 Shell 功能，其他丰富的功能需要用户自己探索。要想用好 VxWorks 开发系统，Shell 的熟练使用是必需的。

10.2.3 WindView 调试

WindView 为事后[post-mortem]运行过程分析工具。WindView 就像 VxWorks 内核的虚拟仪器，可以记录发生的系统或用户事件的信息。对程序调试有极大的帮助。想当年火星探路器躺下后，就是借助 WindView 查找病因来救活的。

当事件发生时，相关事件信息记录在 RAM 中的环型缓冲区内 (rBuff)。在连续上传模式时，若一半的缓冲区被填满，这一半信息就通过网络传送到主机，另一半缓冲区继续被填写。而在事后分析模式，当缓冲区满时，新信息会覆盖旧信息。

缓冲区中信息上传和存储都由 WindView rBuff 管理任务 (tWvRBuffMgr) 控制。该任务会由主机 WindView 工具激活运行。任务缺省优先级为 100，可以用 rBuffLib 库函数 wvRBuffMgrPrioritySet 来改变。

上传运行记录时，WindView 还会在目标机启动任务 tWVUpload，属于用户任务，缺省优先级为 150。注意如果出错的任务比这优先级高，记录上传可能不成功。

● 目标机组件

要想在主机上使用 WindView 辅助分析，目标机必须包括 WindView 组件。WindView 组件位于“开发工具组件”目录下。具体包括的组件如图 10-17 所示。目标机需包括“class instrumentation”用于低级内核分析，WindView 库用于初始化和控制事件记录，还需要选择时标格式，记录信息上传途径和缓冲区管理库。图中的选择是包括 WindView 组件时的缺省选择，如图 10-18 所示。利用 TSFS 上传，还需要包括 TSFS 组件，工程管理器会自动依赖包括，不用手动添加。

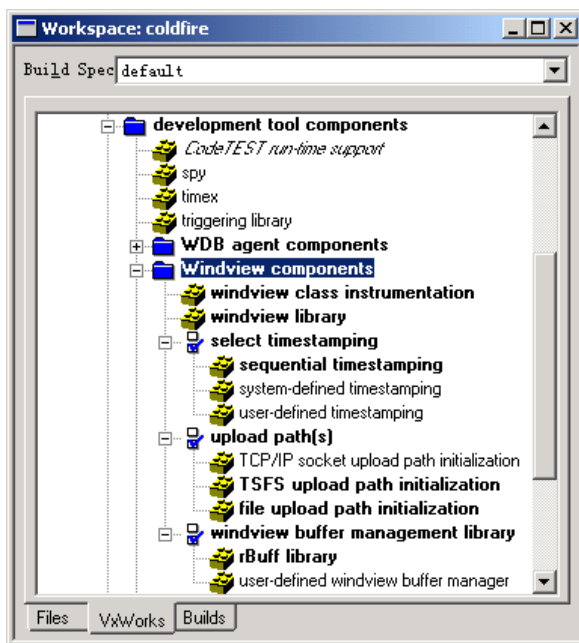


图 10-17 WindView 组件列表

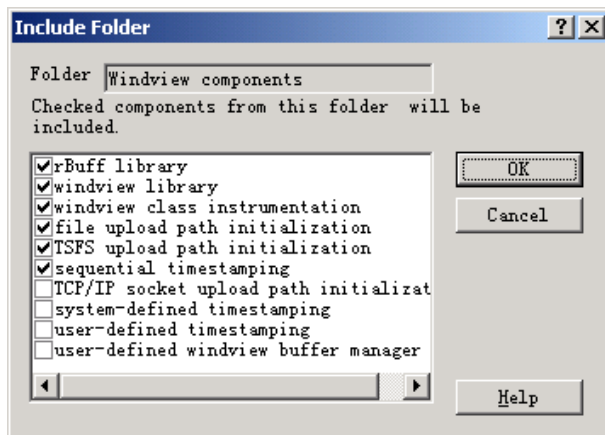


图 10-18 WindView 组件选择

● 主机工具

要想使用 WindView，首先需要配置和启动 Target Server。若选用 TSFS 上传事件记录信息，需要配置主机相应的目录。选择菜单“Tools→Target Sever→Configure”启动 Target Server 的配置对话框，在对话框 Target Server Propertyty 下拉列表选择 TSFS，如图 10-19 所示。

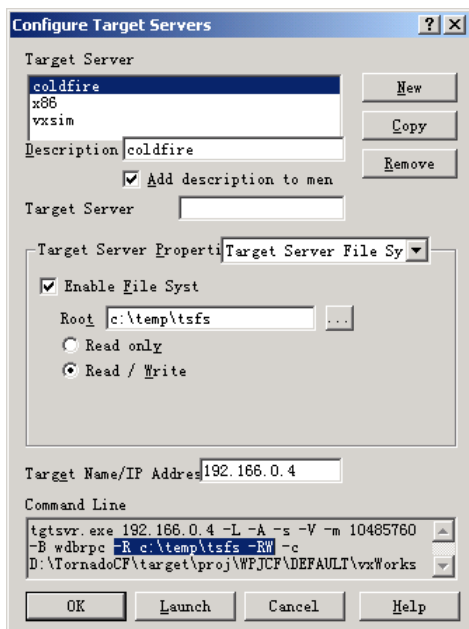


图 10-19 WindView 使用 TSFS 配置

在图 10-19 所示对话框中，使能文件系统，并指定一主机目录为其根目录。注意指定目标机对 TSFS 的访问权限为“Read/Write”，若配置为只读，上传信息时会出现错误提示。

Target Server 运行后，可选择菜单“Tools→WindView→Launch...”命令或图标按钮启动 WindView 工作窗口，如图 10-20 所示。

通过图 10-20 所示窗口的第 1 个按钮启动 WindView 配置窗口，如图 10-21 所示。首先需要选择信息记录等级，这确定了记录数据的大小和对应用系统的影响程度。有 3 个等级可供选择，CSE 级数据量最小；TST 级包括 CSE 级的所有信息，另外还收集任务状态变化的信息；AIL 级包括 TST 级的所有信息，还收集某些库信息，如 taskLib、semLib 等。

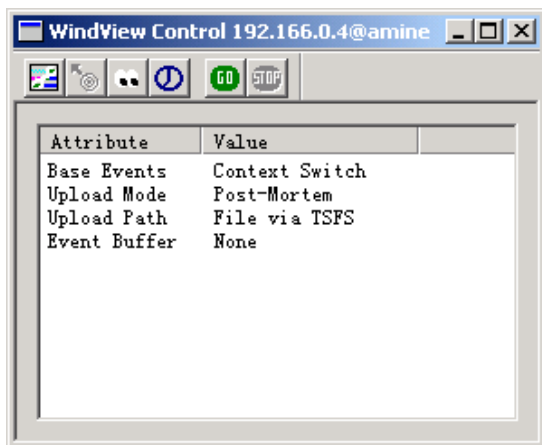


图 10-20 WindView 工作窗口

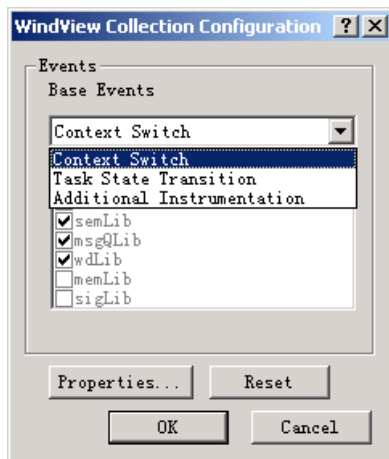


图 10-21 WindView 采集信息配置

通过图 10-21 所示窗口的“Properties...”按钮可激活属性配置窗口，如图 10-22 和图 10-23 所示。

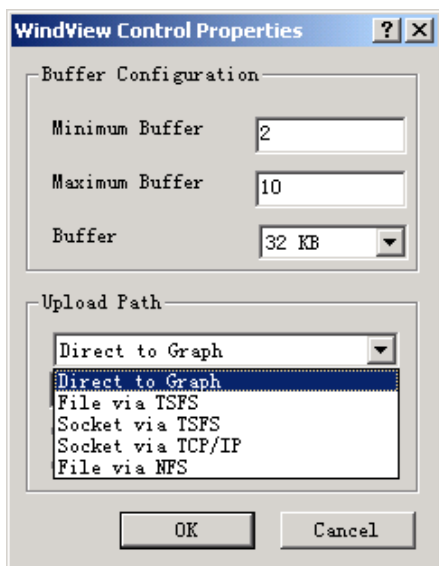


图 10-22 WindView 上传途径

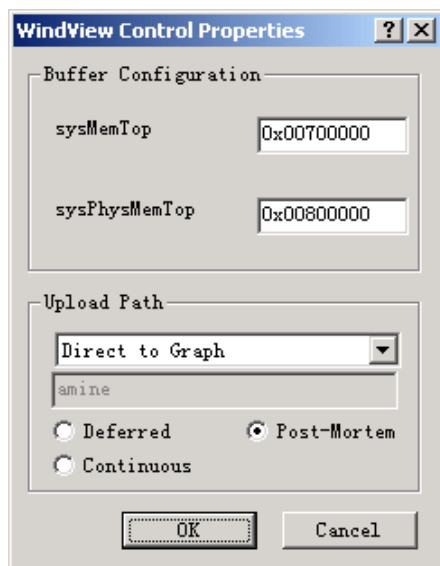


图 10-23 WindView 上传方式

其中图 10-22 所示显示记录信息上传的可用途径，一共有 5 种途径可选择。图 10-23 所示显示 3 种记录上传方式，“Deferred”方式信息在动态内存中存放，用户需要手动召唤，但系统不能复位；“Post-Motem”方式信息在用户保留内存中存放，即使系统复位后，信息仍可用；“Continuous”方式信息在动态内存中存放，信息不断主动地传到主机。

配置完成后，就可以使用图 10-20 所示工作窗口的按钮使用 WindView 功能，启动和停止信息记录，再上传信息供事后分析。

● 实例分析

本节使用前面的 inversion.c 代码（注意注释掉 strcmp 语句），演示如何使用 WindView 的事后分析功能来查找代码中的优先级逆转错误。

在前面的组件配置中，时标格式选择系统定义时标（需依赖添加高分辨时标组件），可以看见程序执行所消耗的时间。注意只能选择一种时标。

上传途径采用“Direct To Gragh”，使数据直接图形显示。分析后可选择是否保存数据为文件（*.wvr），该文件可再用 Tornado 打开（选择菜单“File→Open”命令）。

这里使用“Post-Motem”方式，该方式要求用户保留内存，以便系统复位后能保留信息。当系统出现崩溃后，事后可以利用存储的事件信息来诊断出错原因。系统复位后，重新启动 WindView 工作窗口连接新 Target Server。这时的上传按钮是不可用的，需要先激活配置窗口，点击“OK”确认后搜索是否有可用的记录，如果存在有效信息，会显示如图 10-24 所示对话框。上传按钮变为可用，能对上传系统复位前信息进行分析。

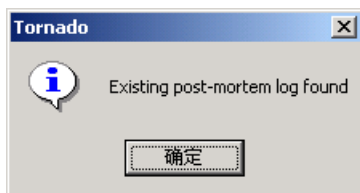


图 10-24 Post-Motem 信息可用

先演示优先级逆转错误现象。不定义 inversion.c 中的条件编译宏“PRINV”，也就是不使用带 SEM_INVERSION_SAFE 属性的互斥信号量。上传的程序运行信息 WindView 显示如图 10-25 所示。tPlow 最先被创建运行，运行 1 个时钟单位后，被 tDbgTask 抢占；tDbgTask 创建 tPhigh 和 tPmid 任务后退出；tPhigh 任务运行，寻求获取信号量，但信号量被 tPlow 占用，tPhigh 任务阻塞在信号量上；tPmid 任务运行，由于 tPmid 比 tPlow 优先级高，且是一个无限忙循环，所以 tPlow 不再有机会运行，也不能释放占用信号量。这里出现的错误就是，虽然 tPhigh 比 tPmid 优先级高，但被低优先级任务占用信号量阻塞而不能运行，事实上 tPhigh 的优先级被降低了，这是与实时系统的设计原则相违背。

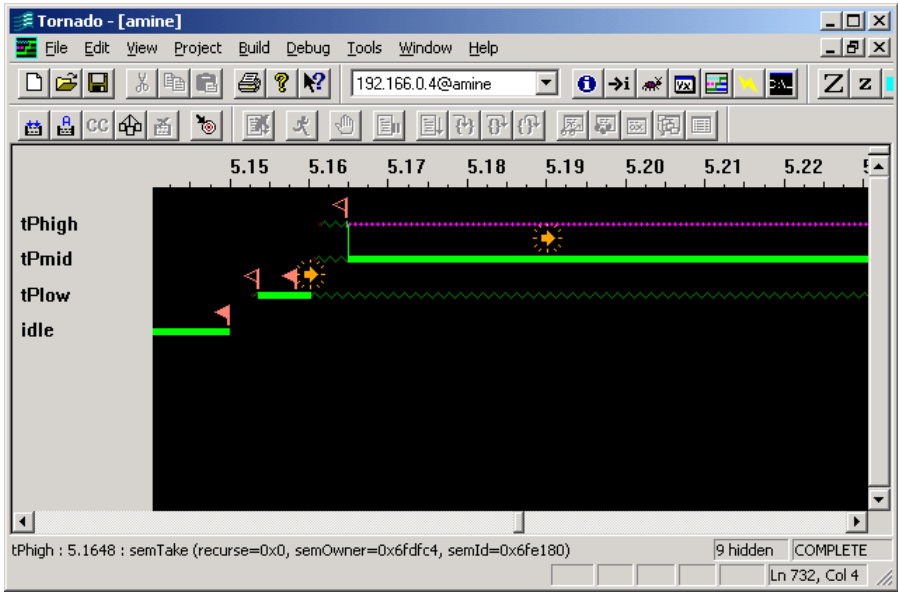


图 10-25 优先级逆转错误

当年，火星探路者也出现类似问题，首先提出优先级逆转概念。开发人员也是使用 WindView 分析运行记录，发现了该问题。VxWorks 为了解决这问题，专门提供了优先级逆转安全的互斥信号量。

定义 inversion.c 中的条件编译宏“PRINV”，使用优先级逆转安全的互斥信号量，可以显示这种信号量的作用，如图 10-26 所示。tPhigh 任务运行，寻求获取信号量，但信号量被 tPlow 占用，tPhigh 任务阻塞在信号量上，并提升 tPlow 的优先级和自己相等；tPlow 临时优先级比 tPmid 高，被调度运行，一段时间后释放信号量，优先级恢复；tPhigh 获得信号量，优先级比 tPmid 高，被调度运行。这样就保证了高优先级任务的优先权，避免了上面出现的优先级逆转错误。

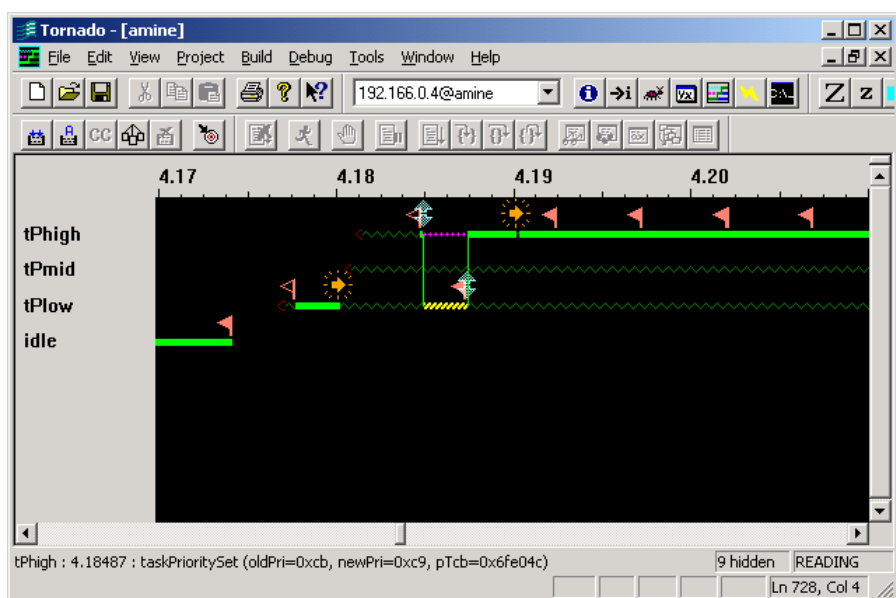


图 10-26 优先级逆转安全

从 Shell 中也可以看到 tPLOW 任务优先级被提升。不过本例优先级提升只是瞬间的事，所以需要调试器配合才能看见这一现象。先暂时挂起 tPhigh 和 tPmid 任务；在 tPLOW 中设置断点，让其在获取信号量后挂起；再恢复 tPhigh 和 tPmid 任务运行。会在 Shell 中看到 tPLOW 任务状态为“SUSPEND+I”，其中“+I”表示该任务优先级被暂时提升。

```
-> i
```

NAME	ENTRY	TID	PRI	STATUS	PC	SP	ERRNO	DELAY
tExcTask	excTask	6fc054	0	PEND	cfla6	6fbfa0	0	0
tLogTask	logTask	6f9728	0	PEND	cfla6	6f9674	0	0
tShell	shell	6917bc	1	READY	b5138	691410	0	0
tWdbTask	ac63a	699988	3	PEND	5c792	6998e0	0	0
tNetTask	netTask	6c3450	50	PEND	5c792	6c33e4	450001	0
tTelnetd	telnetd	69c084	55	PEND	5c792	69bf60	0	0
tTffsPTask	fIPollTask	6f7cdc	100	DELAY	b49e0	6f7c90	0	9
tWvRBufMgr	wvRBufMgr	68ec30	100	PEND	5c792	68ebdc	0	0
tPLOW	1be8f4	688f84	201	SUSPEND+I	1be8d6	688f40	0	0
tPhigh	1be8f4	687e58	201	PEND	badee	687e08	0	0
tPmid	1be91a	6889e0	202	READY	1be942	6889ac	0	0

WindRiver, “WindView User’s Guide”。

10.3 编程实践

本节举两个实例来介绍 VxWorks 平台上代码的编写。

10.3.1 OSAL 的 VxWorks 实现实例

为了保证程序的移植性和保留原编程风格，笔者在实际应用中使用了一个软件层（OSAL）来抽象 VxWorks 核心接口函数。

抽象是软件设计中最重要概念，其实软件就是由各种抽象设计组成的。例如，高级语

言抽象汇编，数据结构抽象原始数据单元，函数抽象实现代码，文件系统抽象原始存储等。随着软件工业的发展，抽象层次不断提高，如 UML 设计和可视开发。设计良好的抽象层次，可以极大地提高编程效率（或代码自动生成），也能保证软件的质量。

设计操作系统接口抽象层和笔者的应用环境有很大的关系。产品系列有多个硬件平台，不同硬件可能有不同的操作系统，而产品有很多功能是类似的，如果分平台实现代码，工作量巨大，代码维护困难，且开发组不能协同工作，所以需要为多个产品提高同一个代码基。实现抽象层的另一个原因是为了保持编程风格。每个开发组都有自己的统一习惯的编程风格，以方便工作交流。笔者原来是 pSOS 用户，习惯了 pSOS 的任务编程风格，特别是事件和时钟机制。而 VxWorks 没有这些系统服务（VxWorks 5.5 提供了事件机制），为了保持大家熟悉的编程风格，抽象层实现了 pSOS 类似接口。

下面这段代码是抽象层的 VxWorks 实现，后面还会描述 NuCleos 和 uC/OS 实现。该抽象层只实现了常用的系统服务，主要是事件和时钟机制。

代码文件 os.h 和 os.c 负责抽象层的一般实现，任务抽象为线程。

```
/*-----
Module:      os.h
Author:      amine
Project:     dsxxxx
State:
Creation Date: 2002-07-10
Description:  操作系统抽象层相关声明
-----*/

/*-----
$Log: OS.H,v $
Revision 1.2  2003/10/13 07:47:37  wyj
routine update
-----*/

#ifndef _OS_H
#define _OS_H

#include "mytypes.h"
#include "dsxxxxcfg.h"
/*-----
OS 抽象层参数宏定义
-----*/

#define THREAD_MAX_NUM    20      /*可创建线程的最大数目*/
#define SEM_MAX_NUM      20      /*可创建信号量的最大数目*/
#define TIMER_MAX_NUM    3       /*每线程的可行的时钟数目*/
#define QUEUE_MSG_SIZE   4096    /*队列消息最大长度*/
#define OS_OBJ_NAME_SIZE  30      /*OS 对象名称最大长度*/
#define TICK2MSE         10      /*每 tick 毫秒数*/
#define SECTOTICK(sec)   (sec*1000/TICK2MSE)
/*-----
事件标志定义(定义各任务事项, 全局定义)
-----*/

#define EV_MSG            0x00010000 /*线程收到消息*/
#define EV_TIMER1        0x00020000 /*时钟 1 事件*/
#define EV_TIMER2        0x00040000 /*时钟 2 事件*/
```

```

#define EV_TIMER3      0x00080000    /*时钟 3 事件*/
/*-----*/
OS 抽象层私有对象结构定义
/*-----*/

/*时钟实现结构定义*/
typedef struct {
    char*      name;
    void*      pevg;        /*绑定线程下事件组 ID*/
    void*      pid;         /*时钟 ID*/
    DWORD      event;       /*时钟事件*/
    DWORD      intval;      /*时钟定时周期*/
    BYTE       thid;        /*关联线程号*/
    BYTE       once;        /*是否为一次性*/
    BYTE       tmno;        /*关联线程中序号*/
    BYTE       padl;        /*字节对齐填充字节*/
} VTimer;

/*线程实现结构定义*/
typedef struct {
    BYTE       used;
    BYTE       padl[3];     /*填充对齐*/
    char*      name;
    void*      ptcb;        /*TCB 指针*/
    void*      pEv_group;   /*Nu: 存放事件组标志*/
    void*      pQu;         /*Nu: 存放消息队列标志*/
    VTimer     tm[TIMER_MAX_NUM]; /*Nu: 该线程的系列时钟*/
} VThread;
/*-----*/
OS 抽象层对外接口函数(接口细节可查看函数头说明)
/*-----*/

int thCreate(char *name, ENTRYPTR entryPtr, int argc, void *argv,
             DWORD stack_size, int priority, DWORD queue_len);
int thStart(BYTE thid);
int thPrioritySet(BYTE thid, int newPriority);
int thSleep(DWORD ticks);
int thSuspend(BYTE thid);
int evSend(BYTE thid, DWORD event);
int evReceive(BYTE thid, DWORD ev_mask, DWORD *pevent);
int msgSend(BYTE thid, void *pbuf, WORD len, BYTE send_ev);
int msgReceive(BYTE thid, void *pbuf, WORD max_len, int wait);
int tmEvEvery(BYTE thid, DWORD intval, DWORD event);
int tmEvAfter(BYTE thid, DWORD intval, DWORD event);
int tmCancel(BYTE thid, BYTE tmno);
int tmDelete(BYTE thid, BYTE tmno);

int ThreadInit(void);
void *ThreadPTcb(BYTE thid);
int ThreadReq(char *name);
#endif /*_OS_H*/
/*-----*/

Module:      os.c
Author:      amine
Project:     dsxxxx
State:

```

Creation Date: 2002-07-03
Description: OS 抽象层通用接口

```
-----*/
/*-----
$Log: OS.C,v $
Revision 1.1.1.1 2003/09/24 09:14:10 administrator
cvs used for the first time

-----*/

static const char rcsid[] =
"$Id: OS.C,v 1.1.1.1 2003/09/24 09:14:10 administrator Exp $";

#include "../include/dsxxxxcfg.h"
#if (TYPE_OS==OS_VXWORKS)
#include <vxworks.h>
#endif
#if (TYPE_OS==OS_NUCLEUS)
#include <string.h>
#endif
#include "../include/os.h"

VThread g_Thread[THREAD_MAX_NUM]; /*OS 抽象层线程信息*/
/*-----
Procedure: ThreadInit ID:1
Purpose: 初始化抽象层环境
Input:
Output: >=0: thid; <0: ERROR
Errors:

-----*/

int ThreadInit(void)
{
    memset(g_Thread, 0, sizeof(VThread)*THREAD_MAX_NUM);
#if (TYPE_OS == OS_VXWORKS)
    /*确定是否使用时间片*/
    kernelTimeSlice(100/TICK2MSE);
#endif
    return 0;
}
/*-----
Procedure: ThreadReq ID:1
Purpose: 向抽象层申请线程号
Input: 线程名
Output: 线程号或 ERROR
Errors:

-----*/

int ThreadReq(char *name)
{
    int i, len;
    char *ps;
    /*取得未使用线程位置*/
    for(i=0; i<THREAD_MAX_NUM; i++)
        if(g_Thread[i].used == 0) break;
```

```

    if(i >= THREAD_MAX_NUM) return ERROR;

    len = strlen(name);
    if(len <= 0 ) return ERROR;

    ps = (char *)malloc(len+1);
    if(ps == NULL) return ERROR;
    memset(ps, 0, len+1);

    strcpy(ps, name);

    g_Thread[i].name = ps;
    g_Thread[i].used = 1;

    return i;
}
/*-----*/
Procedure:      ThreadPtcB ID:1
Purpose:        取线程对应的 TCB
Input:          thid: 线程号
Output:         TCB 指针
Errors:
/*-----*/

void *ThreadPtcB(BYTE thid)
{
    return g_Thread[thid].ptcb;
}
代码文件 os_v.c 为抽象接口的 VxWorks 具体实现。
/*-----*/
Module:         os_v.c
Author:         rover2
Project:        dsxxxx
State:
Creation Date:  2002-07-08
Description:     用 VxWorks 实现 OS 抽象接口
/*-----*/

/*-----*/
$Log: OS_V.C,v $
Revision 1.2  2003/10/13 07:51:28  wyj
routine update
/*-----*/

static const char rcsid[] =
"$Id: OS_V.C,v 1.2 2003/10/13 07:51:28 wyj Exp $";

#include "../include/dsxxxxcfg.h"
#if (TYPE_OS == OS_VXWORKS)

#include "../include/mytypes.h"
#include "../include/os.h"
#include <vxworks.h>
#include <tasklib.h>
#include <semlib.h>

```

```

#include <msgqlib.h>
#include <wdlib.h>

#define TASKOPTION    0
/*事件实现结构*/
typedef struct{
    SEM_ID evSem;
    DWORD evFlag;
} VX_EVENT;

extern VThread g_Thread[THREAD_MAX_NUM];
/*-----
Procedure:      ThCreate ID:1
Purpose:        创建线程
Input:
Output:         >=0: thid; <0: ERROR
Errors:
-----*/
int thCreate(char *name, ENTRYPTR entryPtr, int argc, void *argv,
             DWORD stack_size, int priority, DWORD queue_len)
{
    int thid, VxWorksTid;
    MSG_Q_ID qID;
    SEM_ID semID;
    VX_EVENT *pEv_group;

    VxWorksTid=taskCreat(name,priority, TASKOPTION, stack_size, (FUNCPTR)entryPtr,
argc, (int )argv, 0, 0, 0, 0, 0, 0, 0);
    if (VxWorksTid==ERROR) return(ERROR);

    /*创建一个事件对象, 归属与该线程*/
    semID=semBCreate(SEM_Q_FIFO, SEM_EMPTY);
    if (semID==NULL) return(ERROR);

    /*创建一个消息队列对象, 归属与该线程*/
    if (queue_len){
        qID=msgQCreate(queue_len, QUEUE_MSG_SIZE, MSG_Q_FIFO);
        if (qID==NULL) return(ERROR);
    }
    else qID=NULL;

    thid = ThreadReq(name);
    if (thid==ERROR) return(ERROR);

    g_Thread[thid].pEv_group=malloc(sizeof(VX_EVENT));
    if (g_Thread[thid].pEv_group)
    {
        pEv_group=(VX_EVENT *)g_Thread[thid].pEv_group;
        pEv_group->evFlag = 0;
        pEv_group->evSem = semID;
    }
    else return(ERROR);
}

```

```

    g_Thread[thid].pQu = (void *)qID;
    g_Thread[thid].ptcb = (void *)VxWorksTid;

    return thid;
}
/*-----*/
Procedure:    thStart ID:1
Purpose:      启动线程
Input:
Output:       OK: ERROR
Errors:
/*-----*/

int thStart(BYTE thid)
{
    return(taskActivate((int)ThreadPTcb(thid)));
}
/*-----*/
Procedure:    thPrioritySet ID:1
Purpose:      线程优先级改变
Input:
Output:       old priority
Errors:
/*-----*/

int thPrioritySet(BYTE thid,int newPriority)
{
    int oldPriority;
    if (taskPriorityGet((int)ThreadPTcb(thid),&oldPriority)==ERROR) return(ERROR);
    if (taskPrioritySet((int)ThreadPTcb(thid),newPriority)==ERROR) return(ERROR);

    return(oldPriority);
}
/*-----*/
Procedure:    thSleep ID:1
Purpose:      使当前线程睡眠
Input:       ticks: 睡眠时间
Output:       >=0: OK; -1: ERROR
Errors:
/*-----*/

int thSleep(DWORD ticks)
{
    return(taskDelay(ticks));
}
/*-----*/
Procedure:    thSuspend ID:1
Purpose:      挂起自身
Input:
Output:       >=0: OK; -1: ERROR
Errors:
/*-----*/

int thSuspend(BYTE thid)
{
    exit(0);
}

```



```

/*-----*/
Procedure:      evSend ID:1
Purpose:        向线程发送事件
Input:          thid: 线程号; event: 发送事件
Output:         >=0: OK; <0: ERROR
Errors:

/*-----*/
int evSend(BYTE thid, DWORD event)
{
    int intlvl;
    VX_EVENT *pEV;

    /*判断 thid 的合法性*/
    if (thid >= THREAD_MAX_NUM) return(ERROR);
    if (g_Thread[thid].used==0) return(ERROR);

    pEV=g_Thread[thid].pEv_group;

    intlvl = intLock();          /* LOCK INTERRUPTS */

    /* OR in the events */
    pEV->evFlag |= event;

    intUnlock (intlvl);          /* UNLOCK INTERRUPTS */

    /* and signal the task that an event has been logged */

    return(semGive (pEV->evSem));
}
/*-----*/
Procedure:      evReceive ID:1
Purpose:        接收线程的事件
Input:          thid: 线程号; ev_mask: 事件屏蔽码; pevent: 事件接收
Output:         >=0: OK; <0: ERROR
Errors:

/*-----*/
int evReceive(BYTE thid, DWORD ev_mask, DWORD *pevent)
{
    int intlvl,eventMatch;
    VX_EVENT *pEV;

    /*判断 thid 的合法性*/
    if (thid >= THREAD_MAX_NUM) return(ERROR);
    if (g_Thread[thid].used==0) return(ERROR);

    pEV=g_Thread[thid].pEv_group;

    do {
        /* see which ones we are looking for */
        eventMatch = pEV->evFlag & ev_mask;

        /* see if we've matched any */
        if (eventMatch){

```

```

/* we need to unpend the events to acknowledge we've received them
lock interrupts while messing with the extension*/

intlvl = intLock();          /* LOCK INTERRUPTS */

/* clear any events we've matched */

pEV->evFlag &= ~eventMatch;

intUnlock (intlvl);          /* UNLOCK INTERRUPTS */

break;
}

semTake (pEV->evSem, WAIT_FOREVER);

} while (TRUE);

*pevent=eventMatch;

return(OK);
}
/*-----*/
Procedure:    msgSend ID:1
Purpose:      向线程发送消息
Input:        thid: 线程号; pbuf: 消息缓冲
               len: 消息大小; send_ev: 是否发送事件
Output:       >=0: OK; <0: ERROR
Errors:
/*-----*/
int msgSend(BYTE thid, void *pbuf, WORD len, BYTE send_ev)
{
    /*判断 thid 的合法性*/
    if (thid >= THREAD_MAX_NUM) return(ERROR);
    if (g_Thread[thid].used==0) return(ERROR);

    if (msgQSend((MSG_Q_ID)g_Thread[thid].pQu, (char *)pbuf, len,
NO_WAIT, MSG_PRI_NORMAL)==ERROR)
        return(ERROR);

    if(send_ev) return(evSend(thid, EV_MSG));

    return(OK);
}
/*-----*/
Procedure:    msgReceive ID:1
Purpose:      从线程接收消息
Input:        thid: 线程号; pbuf: 消息缓冲
               max_len: 最大消息大小;
               wait: 等待选项(0:不等待, -1:永久等待, x:等待时间)
Output:       >=0: 实际接收的字节数; -1: ERROR
Errors:
/*-----*/

```

```

int msgReceive(BYTE thid, void *pbuf, WORD max_len, int wait)
{
    /*判断 thid 的合法性*/
    if (thid >= THREAD_MAX_NUM) return(ERROR);
    if (g_Thread[thid].used==0) return(ERROR);

    return(msgQReceive((MSG_Q_ID)g_Thread[thid].pQu, (char *)pbuf, max_len, wait));
}
/*-----*/
Procedure:      tmExpire ID:1
Purpose:        时钟超时处理函数, 负责发送事件, 且重新启动定时器
Input:
Output:
Errors:
/*-----*/

static void tmExpire(int arg)
{
    int thid, i;
    VTimer *ptm;

    thid=arg>>16;
    i=arg&0xFFFF;

    ptm =g_Thread[thid].tm+i;

    evSend(thid, ptm->event);

    wdStart((WDOG_ID)ptm->pid, ptm->intval, (FUNCPTR)tmExpire, arg);
}
/*-----*/
Procedure:      tmAfter ID:1
Purpose:        时钟超时处理函数, 负责发送事件, 但不再重新启动定时器
Input:
Output:
Errors:
/*-----*/

static void tmAfter(int arg)
{
    int thid, i;
    VTimer *ptm;

    thid=arg>>16;
    i=arg&0xFFFF;

    ptm =g_Thread[thid].tm+i;

    evSend(thid, ptm->event);
}
/*-----*/
Procedure:      tmEvEvery ID:1
Purpose:        为线程创建周期定时器
Input:          thid: 线程号; intval: 时钟周期
                event: 触发事件;

```

Output: >=0: 时钟号; -1: ERROR

Errors:

```
-----*/
int tmEvEvery(BYTE thid, DWORD intval, DWORD event)
{
    int i,arg,intlvl;
    WDOG_ID wdID;

    if (thid >= THREAD_MAX_NUM) return(ERROR);
    if (g_Thread[thid].used==0) return(ERROR);

    wdID = wdCreate ();
    if (wdID==NULL) return(ERROR);

    intlvl = intLock();          /* LOCK INTERRUPTS */
    for(i=0; i<TIMER_MAX_NUM; i++)
    {
        if( g_Thread[thid].tm[i].pid == NULL) break;
    }
    if(i >= TIMER_MAX_NUM)
    {
        intUnlock (intlvl);      /* UNLOCK INTERRUPTS */
        return ERROR;
    }
    g_Thread[thid].tm[i].pid = (void *)wdID;
    intUnlock (intlvl);          /* UNLOCK INTERRUPTS */

    g_Thread[thid].tm[i].intval = intval;
    g_Thread[thid].tm[i].event = event;

    arg=(thid<<16)|(i&0xFFFF); /*高字节 thid, 低字节时钟索引*/
    if ((wdStart(wdID,intval, (FUNCPTR)tmExpire,arg))==ERROR) return(ERROR);

    return(i);
}
/*-----*/
```

Procedure: tmEvAfter ID:1
Purpose: 周期定时后线程触发事件
Input: thid: 线程号; intval: 时钟周期
 event: 触发事件;
Output: >=0: 时钟号; -1: ERROR
Errors:

```
-----*/
int tmEvAfter(BYTE thid, DWORD intval, DWORD event)
{
    int i,arg,intlvl;
    WDOG_ID wdID;

    if (thid >= THREAD_MAX_NUM) return(ERROR);
    if (g_Thread[thid].used==0) return(ERROR);

    wdID = wdCreate ();
    if (wdID==NULL) return(ERROR);
```

```

    intlvl = intLock();          /* LOCK INTERRUPTS */
    for(i=0; i<TIMER_MAX_NUM; i++)
    {
        if( g_Thread[thid].tm[i].pid == NULL) break;
    }
    if(i >= TIMER_MAX_NUM)
    {
        intUnlock (intlvl);      /* UNLOCK INTERRUPTS */
        return ERROR;
    }
    g_Thread[thid].tm[i].pid = (void *)wdID;
    intUnlock (intlvl);          /* UNLOCK INTERRUPTS */

    g_Thread[thid].tm[i].intval = intval;
    g_Thread[thid].tm[i].event = event;

    arg=(thid<<16)|(i&0xFFFF); /*高字节 thid, 低字节时钟索引*/
    if ((wdStart(wdID, intval, (FUNCPTR) tmAfter, arg))==ERROR) return(ERROR);

    return(i);
}
/*-----*/
Procedure:      tmCancel ID:1
Purpose:        取消线程周期定时器
Input:          thid: 线程号; tmno:定时器号
Output:         >=0: OK; -1: ERROR
Errors:
/*-----*/
int tmCancel(BYTE thid, BYTE tmno)
{
    if (thid >= THREAD_MAX_NUM) return(ERROR);
    if (g_Thread[thid].used==0) return(ERROR);
    if (tmno>=TIMER_MAX_NUM) return(ERROR);

    return(wdCancel((WDOG_ID)g_Thread[thid].tm[tmno].pid));
}
/*-----*/
Procedure:      tmDelete ID:1
Purpose:        取消线程周期定时器
Input:          thid: 线程号; tmno:定时器号
Output:         >=0: OK; -1: ERROR
Errors:
/*-----*/
int tmDelete(BYTE thid, BYTE tmno)
{
    int i;
    if (thid >= THREAD_MAX_NUM) return(ERROR);
    if (g_Thread[thid].used==0) return(ERROR);
    if (tmno>=TIMER_MAX_NUM) return(ERROR);

    /*??pid==NULL*/

```

```

i=wdDelete((WDOG_ID)g_Thread[thid].tm[tmno].pid);
if (i!=ERROR) g_Thread[thid].tm[tmno].pid=NULL;

return(i);
}
#endif /*(TYPE_OS == OS_VXWORKS)*/

```

10.3.2 通用通信层实现实例

在一个综合的控制系统中，设备的互联性是至关重要的。首先设备要提供某种形式的通信硬件，如串口、现场总线、网络等。在各种数据通道上，还要加载行业专用的规约协议，才能实现设备间的互联。

为了产品广泛的适用性，一个硬件平台上可能同时包含多种通信硬件。软件需要为它们提供统一的抽象的接口，以方便上层专用规约的实现，与底层驱动实现分离。

大多数操作系统都提供 IO 接口规范来统一设备的访问接口，如 pSOS 的 DISI 和 VxWorks 的 IO 系统。用户可以遵循操作系统规范，来方便实现各设备的统一抽象接口。但如果存在多个软件平台，而想实现跨平台代码设计，为各软件平台提供特殊的驱动实现是不可取的。笔者需要进行自己的通用通信层设计，并将设备驱动实现提升到应用层，保持与操作系统的无关性。

为了将各种不同设备驱动归入统一接口，必须规定驱动的规范接口。不同操作系统有不同的规定，但比较类似，一般有 init、open、close、read、write 和 ioctl 等。这些接口通常先在操作系统的抽象层中注册，上层应用再通过函数指针回调[callback]被动的驱动实现函数。大多数设备驱动都使用函数回调方式来分离抽象接口和具体实现，VxWorks 中也是如此。VxWorks 中，设备驱动实现函数指针填写入抽象数据结构，结构指针代表具体设备，在设备上各软件层传递。各软件层有类似的兼容结构定义，设备指针在各层会得到不同的解释，读者可参考块设备上的 CBIO 实现。

下面代码为笔者实际开发中使用的，实现了一个抽象的通信层，统一了多种通信通道的操作接口，包括网络、串口和 CAN 等。该软件层称为“comm”，采用全局统一编号区分底层实现，而没采用函数注册和回调方式。通信层提供 comm 函数作为执行线程，是个很简单的定时循环，扫描各通信通道数据缓冲区，根据缓冲区的状态产生各种事件，通知上层应用进行相应的处理，如接收数据等。上层应用和具体通道通过全局通道编号联系在一起，驱动实现中可通过通道号取得相应任务的线程号（见前面介绍的操作系统抽象层），来完成对应任务的事件通知。

```

/*-----
Module:      comm.c
Author:      amine
Project:     dsxxxx
State:       创建
Creation Date: 2002-08-26
Description: 统一通信接口实现
-----*/

/*-----
$Log: COMM.C,v $
Revision 1.1.1.1 2003/09/24 09:14:08 administrator
cvs used for the first time

```

```

-----*/
static const char rcsid[] =
"$Id: COMM.C,v 1.1.1.1 2003/09/24 09:14:08 administrator Exp $";

#include <mytypes.h>
#include <os.h>
#include <sys.h>
#include "comm_cfg.h" /*通信端口硬件选择配置*/
#include "comm.h" /*通信接口函数原型, 以及 CommCtrl() 函数命令定义*/
#include "../net/net.h"
#include "../serial/serial.h"
#include "../canbus/canbus.h"
/*-----
Procedure: comm ID:1
Purpose: 通信口管理任务, 由系统统一启动
Input: wTaskID: 高级应用号; VTaskArg: 统一函数入口
Output:
Errors:
-----*/

void comm(WORD wTaskID, struct VTaskArg *argv)
{
    int thid; DWORD events;

    thid = GetThid((int)wTaskID);
    CommInit(wTaskID);
    tmEvEvery(thid, 10/TICK2MSE, EV_TIMER1);

while (1) {
evReceive(thid, EV_TIMER1, &events);
if(events & EV_TIMER1) {
#ifdef INCLUDE_NET
NetScanOnTimer();
#endif
SerScanOnTimer();
#ifdef INCLUDE_CAN
CanScanOnTimer();
#endif
}
}/*while(1)*/
}
/*-----
Procedure: CommInit ID:1
Purpose: 初始化通信口
Input:
Output: 0:OK; -1:ERROR.
Errors:
-----*/

int CommInit (WORD wTaskID)
{
SerInit(wTaskID);
#ifdef INCLUDE_CAN
CanInit(wTaskID);

```

```

#endif
#ifdef INCLUDE_NET
    NetInit(wTaskID);
#endif
    return OK;
}
/*-----
Procedure:    CommRead ID:1
Purpose:      读通信口
Input:        num: 通信口号; pbuf: 缓冲区
               buflen: 指定长度; flags: 标志(参考具体接口)
Output:       读入字节个数; <0: 出错
Errors:
*/-----

int CommRead (int num, char* pbuf, int buflen, DWORD flags)
{
    int ret;
    if( (num >= COMM_NO_SERIAL_B) && (num <= COMM_NO_SERIAL_E) ){
        ret = SerRead( (num-COMM_NO_SERIAL_B), pbuf, buflen, flags);
        if (ret>0) BufPrint (num, 0, ret, pbuf);
        return ret;
    }
#ifdef INCLUDE_CAN
    else if( (num >= COMM_NO_CANBUS_B) && (num <= COMM_NO_CANBUS_E) ){
        ret = CanRead( (num-COMM_NO_CANBUS_B), pbuf, buflen, flags);
        if (ret>0) BufPrint (num, 0, ret, pbuf);
        return ret;
    }
#endif
#ifdef INCLUDE_NET
    else if( (num >= COMM_NO_NET_B) && (num <= COMM_NO_NET_E) ){
        ret = NetRead( (num-COMM_NO_NET_B), pbuf, buflen, flags);
        if (ret>0) BufPrint (num, 0, ret, pbuf);
        return ret;
    }
#endif
    else
        return ERROR;
}
/*-----
Procedure:    CommWrite ID:1
Purpose:      写通信口
Input:        num: 通信口号; pbuf: 缓冲区
               buflen: 指定长度; flags: 标志(参考具体接口)
Output:       写入字节个数; <0: 出错
Errors:
*/-----

int CommWrite(int num, char* pbuf, int buflen, DWORD flags)
{
    int ret;
    if( (num >= COMM_NO_SERIAL_B) && (num <= COMM_NO_SERIAL_E) ){
        ret = SerWrite( (num-COMM_NO_SERIAL_B), pbuf, buflen, flags);
        if (ret>0) BufPrint (num, 1, ret, pbuf);
    }
}

```



```

        return ret;
    }
#ifdef INCLUDE_CAN
    else if( (num >= COMM_NO_CANBUS_B) && (num <= COMM_NO_CANBUS_E) ){
        ret = CanWrite( (num-COMM_NO_CANBUS_B), pbuf, buflen, flags);
        if (ret>0) BufPrint(num,1,ret,pbuf);
        return ret;
    }
#endif
#ifdef INCLUDE_NET
    else if( (num >= COMM_NO_NET_B) && (num <= COMM_NO_NET_E) ){
        ret = NetWrite( (num-COMM_NO_NET_B), pbuf, buflen, flags);
        if (ret>0) BufPrint(num,1,ret,pbuf);
        return ret;
    }
#endif
    else
        return ERROR;
}
/*-----
Procedure:      CommCtrl ID:1
Purpose:        控制通信口
Input:          num: 通信口号; flags: 标志(参考具体接口)
Output:         0:OK; -1:ERROR.
Errors:
-----*/

int CommCtrl (int num, WORD command, DWORD *para )
{
    int ret;
    if( (num >= COMM_NO_SERIAL_B) && (num <= COMM_NO_SERIAL_E) ){
        ret = SerCtrl( (num-COMM_NO_SERIAL_B), command, para);
        return ret;
    }
#ifdef INCLUDE_CAN
    else if( (num >= COMM_NO_CANBUS_B) && (num <= COMM_NO_CANBUS_E) ){
        ret = CanCtrl( (num-COMM_NO_CANBUS_B), command, para);
        return ret;
    }
#endif
#ifdef INCLUDE_NET
    else if( (num >= COMM_NO_NET_B) && (num <= COMM_NO_NET_E) ){
        ret = NetCtrl( (num-COMM_NO_NET_B), command, para);
        return ret;
    }
#endif
    else
        return ERROR;
}

```

● 通用串口层实现

有了抽象的通信层支持，则可以独立实现各具体的通信通道，本节描述了串口的实现。在笔者的应用中存在多种串口硬件，如 CPU 串口，扩展串口芯片 16C552 等。为了屏蔽

串口的差异，又设计了通用串口层，负责实现接口抽象、缓冲区读写和定时扫描等功能，如下面代码文件 serial.h 和 serial.c 所示。

串口层为每个串口通道分配两个循环缓冲区（rx_buf 和 tx_buf）用于接收和发送数据，串口层的读写函数和扫描函数都在缓冲区上进行操作，和具体的硬件细节没有关系。底层驱动也可见该缓冲区，把接收到的数据写入 rx_buf，取 tx_buf 中的数据发送出去。这种方式和 VxWorks 中串口驱动类似，不同的是底层不能直接看见缓冲区，而通过上层提供的读写回调函数指针来操作缓冲区。

串口层和驱动主要通过缓冲区来交流，还保留了 ioctl 接口，用于实现一些额外的功能，如波特率设置，Modem 控制等。ioctl 接口由回调函数指针实现，在驱动向串口层注册时（SerRegister）填写。

```
/*-----
Module:      serial.h
Author:      amine
Project:     dsxxxx
State:
Creation Date: 2002-08-26
Description:  串行通信接口声明
-----*/

#ifndef _SERIAL_H
#define _SERIAL_H
/*-----

                                宏定义
-----*/

enum {
    SERIAL_MAX_NUM = (COMM_NO_SERIAL_E-COMM_NO_SERIAL_B+1),
    SERIAL_BUF_NUM = 2048 /*!!!必须是 2 幂*/
};
/*-----

                                类型定义
-----*/

/*循环缓冲区*/
typedef struct {
    BYTE *addr;          /*缓冲区地址*/
    WORD size;           /*缓冲区大小*/
    WORD rp;             /*读偏移*/
    WORD wp;             /*写偏移*/
} VSerBuf;
/*逻辑通道相关*/
typedef struct {
    WORD used;           /*是否有对应的物理口*/
    WORD thid;           /*挂结的线程号*/
    VSerBuf rx_buf;
    VSerBuf tx_buf;
    int (*ctrl) (int minor, WORD command, DWORD *para);
    WORD tx_empty_num;   /*发送空字符数*/
    WORD rx_avail_num;   /*接收可用字符数*/
    DWORD ev_send;       /*向上层任务通知通道可用，保持任务访问与事件发送同步*/
    WORD refer;          /*是否有上层任务引用*/
} VSerial;
```

```

/*-----
                                变量声明
-----*/

extern VSerial gaSerial[]; /*需要由下层驱动实现引用*/
/*-----
                                函数声明
-----*/

/*串行通信接口*/
int SerInit(WORD wTaskID);
int SerRead (int minor, char* pbuf, int buflen, DWORD flags);
int SerWrite(int minor, char* pbuf, int buflen, DWORD flags);
int SerCtrl (int minor, WORD command, DWORD *para);
int SerRegister(void *ctrl);
void SerScanOnTimer (void);
#endif /*_SERIAL_H*/
/*-----
Module:          serial.c
Author:          amine
Project:         dsxxxx
State:           创建
Creation Date:   2002-08-26
Description:     所有串行通信抽象接口实现
-----*/

#include <string.h>
#include <stdlib.h>
#include <os.h>
#include "../comm.h"
#include "../comm_cfg.h"
#include "serial.h"

VSerial gaSerial[SERIAL_MAX_NUM]; /*为各串口定义抽象数据结构*/
/*初始化系列串行通信口*/
int SerInit (WORD wTaskID)
{
    memset(gaSerial, 0, sizeof(gaSerial[0])*SERIAL_MAX_NUM);
#ifdef INCLUDE_SCI_332
    SciInit();
#endif
#ifdef INCLUDE_UART_5272
    uartInit();
#endif
#ifdef INCLUDE_16C55X
    V16C55xInit();
#endif
    /*根据参数初始化 gaSerial 结构，并调整硬件配置（如波特率）*/
    /*代码省略.....*/
    return OK;
}

/*读串行口*/
int SerRead (int minor, char* pbuf, int buflen, DWORD flags)
{
    int i; VSerBuf *pser;
    if(minor >= SERIAL_MAX_NUM) return ERROR;

```

```

    pser = &(gaSerial[minor].rx_buf);

    for(i=0; i<buflen; i++){
        if(pser->rp == pser->wp) break;
        *(pbuf+i) = *(pser->addr + pser->rp);
        pser->rp = (pser->rp+1)&(SERIAL_BUF_NUM-1);
    }
    gaSerial[minor].ev_send &= ~EV_RX_AVAIL;
    return i;
}
/*写串行口*/
int SerWrite(int minor, char* pbuf, int buflen, DWORD flags)
{
    int i; VSerBuf *pser; WORD wp;
    if(minor >= SERIAL_MAX_NUM) return ERROR;

    if(buflen <= 0) return ERROR;

    pser = &(gaSerial[minor].tx_buf);
    /*避免临界字符重复和缓冲区满误判*/
    for(i=0; i<buflen; i++){
        wp = (pser->wp+1)&(SERIAL_BUF_NUM-1);
        if(pser->rp == wp) break;
        *(pser->addr + pser->wp) = *(pbuf+i);
        pser->wp = wp;
    }
    /*打开发送中断*/
    if(i > 0) SerCtrl(minor, CCC_INT_CTRL|TXINT_ON, NULL);

    gaSerial[minor].ev_send &= ~EV_TX_AVAIL;
    return i;
}
/*控制串行口*/
int SerCtrl (int minor, WORD command, DWORD *para)
{
    int ret;
    ret = ERROR;
    /*取接收缓冲区可用数据个数*/
    if((command&CCC_RXBUF_NUM)==CCC_RXBUF_NUM) {
        ret =(gaSerial[minor].rx_buf.wp - gaSerial[minor].rx_buf.rp + SERIAL_BUF_NUM)&
            (SERIAL_BUF_NUM-1);
    }else { /*未实现命令传给驱动*/
        if( gaSerial[minor].ctrl == NULL ) return ERROR;
        ret = gaSerial[minor].ctrl(minor, command, para);
    }
    return ret;
}
/*底层串口驱动注册*/
int SerRegister(void *ctrl)
{
    int i;
    for(i=0; i< SERIAL_MAX_NUM; i++)
        if (gaSerial[i].used == 0) break;

```

```

    if( i >= SERIAL_MAX_NUM) return ERROR;
    /*分配接收和发送缓冲区*/
    gaSerial[i].rx_buf.size = SERIAL_BUF_NUM;
    gaSerial[i].rx_buf.addr = (BYTE *)malloc(gaSerial[i].rx_buf.size);
    if(gaSerial[i].rx_buf.addr == NULL) return ERROR;
    memset(gaSerial[i].rx_buf.addr, 0, gaSerial[i].rx_buf.size);

    gaSerial[i].tx_buf.size = SERIAL_BUF_NUM;
    gaSerial[i].tx_buf.addr = (BYTE *)malloc(gaSerial[i].tx_buf.size);
    if(gaSerial[i].tx_buf.addr == NULL) return ERROR;
    memset(gaSerial[i].tx_buf.addr, 0, gaSerial[i].tx_buf.size);

    gaSerial[i].used = 1;
    gaSerial[i].ctrl = ctrl;    /*存储底层驱动 ioctl 函数指针*/

    return i;
}
/*定时扫描各串口，向上层任务发送事件和消息*/
void SerScanOnTimer (void)
{
    int minor, old; DWORD countr,countw; VSerBuf *pbuf; VSerial *ps;

    for(minor=0; minor<SERIAL_MAX_NUM; minor++){
        if(gaSerial[minor].refer != 1) continue; /*如果没有上层任务引用，则略过*/
        old = intLock();
        pbuf = &gaSerial[minor].rx_buf;
        countr=(pbuf->wp - pbuf->rp + pbuf->size)&(SERIAL_BUF_NUM-1);
        pbuf = &gaSerial[minor].tx_buf;
        countw=(pbuf->wp - pbuf->rp + pbuf->size)&(SERIAL_BUF_NUM-1);
        intUnlock(old);
        /*接收缓冲区扫描*/
        if(!(gaSerial[minor].ev_send & EV_RX_AVAIL)){
            if(countr >= gaSerial[minor].rx_avail_num ){
                gaSerial[minor].ev_send |= EV_RX_AVAIL;
                if( evSend(gaSerial[minor].thid, EV_RX_AVAIL) < 0)
                    gaSerial[minor].ev_send &= ~EV_RX_AVAIL;
            }
        }
        /*发送缓冲区扫描*/
        if(countw < gaSerial[minor].tx_empty_num){
            if(((gaSerial[minor].ev_send & EV_TX_AVAIL)) ){
                gaSerial[minor].ev_send |= EV_TX_AVAIL;
                if( evSend(gaSerial[minor].thid, EV_TX_AVAIL) < 0)
                    gaSerial[minor].ev_send &= ~EV_TX_AVAIL;
            }
        }
    }
}
}

```

● 16C552 驱动实现

本节描述一个与硬件相关的串口驱动的实现。对应的串口硬件芯片为 16C552，在实际中有广泛应用，如 PC 兼容平台。

该芯片驱动 VxWorks 也提供（在 “\target\src\drv\sio\” 目录下），不过是与上层库关联紧密，不方便与自己设计的通用通信层接口，所以需要自己实现驱动。驱动为应用代码，不属于通常意义的 BSP 层。这种方式减少了定制硬件与具体软件平台的耦合，提高了代码的移植性，在某些工控系统应用较多。注意 VxWorks 的 Console 可能定向在串口上，接管串口后 Console 不再可用，需要设置 CONSOLE_TTY 为 NONE，否则会影响系统使用，特别会影响 Telnet 上 Target Shell 的使用。

16C55x 系列芯片提供的通道是独立的，各通道有自己的中断和寄存器。下面的代码按通道的概念编写。各通道共用一个中断处理函数，根据入口参数或特殊寄存器访问特定的通道。各通道寄存器也由通道号参数 (phy_no) 加以区分，每个通道对应一个抽象结构 (VphyChan)。所以该段代码适合 16C55x 系列所有芯片，包括提供 8 通道的 PC104 多串口卡。

为了节省篇幅，略去一些由控制接口 (ioctl) 调用的辅助函数代码，如波特率设置、中断控制和 Modem 操作等，但这不会影响驱动设计结构的理解。

```
/*-----
Module:      16c55x.c
Author:      amine
Project:     dsxxxx
State:
Creation Date: 2002-07-15
Description:  16c55x 芯片的驱动程序
-----*/

#include <dsxxxxcfg.h>

#if (TYPE_OS == OS_NUCLEUS)
#include <stdlib.h>
#include <string.h>
#include "../platform/cpu.h"          /*端口抽象访问接口定义，如 sysInByte() 等*/
#endif

#include <mytypes.h>

#include "../comm_cfg.h"
#ifdef INCLUDE_16C55X
#include "16c55x.h"                  /*包括寄存器地址定义，如 LINECTL 等*/

#if (TYPE_CPU == CPU_X86_6231)
#include <arch/i86/ivI86.h>
#define CLOCK_INPUT    1843200
#else
#define CLOCK_INPUT    8000000
#endif

#if ( (TYPE_CPU&TYPE_MASK) == CPU_5272 )
#include "../../platform/xcf5272/bsp.h"
#endif

#if ( (TYPE_CPU&TYPE_MASK) == CPU_332 )
#include "../../platform/mc68332/reg32.h"
#include "../../platform/mc68332/bsp.h"
#endif
#endif
```

```

#include "../comm.h"
#include "serial.h"

static BYTE chan_max_num; /*本驱动支持的实际通道数*/
static VPhyChan g_Chان[CHAN_MAX_NUM]; /*数组下标为通道物理号 phy_no, 从 0 计数*/

static int v16C55x_open(BYTE phy_no);
static void v16C55x_int(BYTE int_no);
static void v16c55x_int_rx(BYTE phy_no, BYTE lstat);
static void v16c55x_int_tx(BYTE phy_no);
static void v16c55x_baud_set(BYTE phy_no, DWORD baudrate);
static int v16C55x_int_ctrl(BYTE phy_no, BYTE rx_tx, BYTE on_off);
static BYTE v16c55x_modem_in(BYTE phy_no);
static void v16c55x_modem_out(BYTE phy_no, BYTE modstat);
/*-----
Procedure:      V16C55xInit ID:1
Purpose:        芯片整体初始化
Input:
Output:         0:OK; -1:ERROR.
Errors:
-----*/

int V16C55xInit(void)
{
    int i, j; DWORD imrMask;

    chan_max_num = 0;
    memset(g_Chان, 0, sizeof(g_Chان[0])*CHAN_MAX_NUM);

    /*根据硬件填充地址和中断号*/
#if (TYPE_CPU == CPU_X86_6231)
    chan_max_num = 4;
    g_Chان[0].addr = 0x3F8; g_Chان[0].int_no = 4;
    g_Chان[1].addr = 0x2F8; g_Chان[1].int_no = 3;
    g_Chان[2].addr = 0x3E8; g_Chان[2].int_no = 7;
    g_Chان[3].addr = 0x2E8; g_Chان[3].int_no = 5;
#endif

#if ( (TYPE_BACK&TYPE_MASK) == BACK_XXXX_5272)
    chan_max_num = 2;
    g_Chان[0].addr = BSP_16C55_0; g_Chان[0].int_no = INT_NUM_INT4;
    g_Chان[1].addr = BSP_16C55_1; g_Chان[1].int_no = INT_NUM_INT5;
#endif

#if ( (TYPE_BACK&TYPE_MASK) == BACK_XXXX_68332 )
    chan_max_num = 2;
    g_Chان[0].addr = BSP_16C55_0; g_Chان[0].int_no = V_IRQ + 4;
    g_Chان[1].addr = BSP_16C55_1; g_Chان[1].int_no = V_IRQ + 3;
#endif

    /*根据上面的硬件配置, 初始化各通道*/
    for(i=0; i<chan_max_num; i++) v16C55x_open(i);
    return 0;
}

```

```

/*-----通道控制函数，minor 为串口统一编号，可能包括别的硬件串口-----*/
int V16C55xCtrl(int minor, WORD command, DWORD *para)
{
    int phy_no; BYTE lcr;
    for(phy_no=0; phy_no<chan_max_num; phy_no++)
        if( g_Chan[phy_no].minor == minor ) break;
    if(phy_no >= chan_max_num) return ERROR;

switch(command&0xFF00) {
    case CCC_BAUD_SET:
        v16c55x_baud_set(phy_no, *para);
        break;
    case CCC_INT_CTRL:
        if( command & RXINT_ON) v16C55x_int_ctrl(phy_no, RXINT, 1);
        if( command & RXINT_OFF) v16C55x_int_ctrl(phy_no, RXINT, 0);
        if( command & TXINT_ON) v16C55x_int_ctrl(phy_no, TXINT, 1);
        if( command & TXINT_OFF) v16C55x_int_ctrl(phy_no, TXINT, 0);
        break;
    case CCC_DATA_BITS:
        if(*para > 8 || *para < 5) return ERROR;
        lcr = sysInByte(g_Chan[phy_no].addr + LINECTL);
        lcr = lcr&0xFC;
        lcr = lcr + (*para-5)&0x03;
        sysOutByte(g_Chan[phy_no].addr + LINECTL, lcr);
        break;
    case CCC_STOP_2BITS:
        lcr = sysInByte(g_Chan[phy_no].addr + LINECTL);
        lcr = lcr|0x04;
        sysOutByte(g_Chan[phy_no].addr + LINECTL, lcr);
        break;
    case CCC_PARITY_SET:
        if(*para==0 || *para>2) return ERROR;
        lcr = sysInByte(g_Chan[phy_no].addr + LINECTL);
        lcr = lcr|0x08;
        if(*para == PARITY_SET_ODD) lcr = lcr&0xEF;
        else lcr = lcr|0x10;
        sysOutByte(g_Chan[phy_no].addr + LINECTL, lcr);
        break;
    default:
        return ERROR;
        break;
};
    return OK;
}

/*-----通道初始化-----*/
static int v16C55x_open(BYTE phy_no)
{
    int minor,i; static BYTE iir1, iir2;

    minor = SerRegister((void *)V16C55xCtrl);
    g_Chan[phy_no].minor = minor;

#if ( (TYPE_CPU&TYPE_MASK) == CPU_X86 )

```



```

        if (phy_no < chan_max_num)
            intConnect( INUM_TO_IVEC(INT_VEC_GET(g_Chan[phy_no].int_no)),
                        v16C55x_int, g_Chan[phy_no].int_no);
    #endif/*( (TYPE_CPU&TYPE_MASK) == CPU_X86 )*/

    #if ( (TYPE_CPU&TYPE_MASK) == CPU_5272 )
        intConnect( INUM_TO_IVEC(g_Chan[phy_no].int_no),
                    v16C55x_int, g_Chan[phy_no].int_no);
    #endif

    #if (TYPE_OS == OS_NUCLEUS)
        /*intConnect 在 Nucleus 为自定义接口， 实际调用 NU_Register_LISR()*/
        intConnect((void *) g_Chan[phy_no].int_no, v16C55x_int, 0);
    #endif

    sysOutByte(g_Chan[phy_no].addr+IENB, 0);
    iir1 = sysInByte(g_Chan[phy_no].addr + IENB);
    sysOutByte(g_Chan[phy_no].addr + LINECTL, 3);
    iir2 = sysInByte(g_Chan[phy_no].addr + LINECTL);
    if(iir1 != 0 || iir2 != 3)
        printf("serial %d init fail!\n", phy_no);

    v16c55x_baud_set(phy_no, 9600);
    /*Modem 控制*/
    sysOutByte(g_Chan[phy_no].addr + MODCTL, OUT2|DTR|RTS);
    g_Chan[phy_no].ctrl_modem = OUT2|DTR|RTS;

    sysOutByte(g_Chan[phy_no].addr + FCR, 0);
    v16C55x_int_ctrl(phy_no, RXINT, 1);
    /*!!!!必须在开中断后启动 FIFO, 否则有数据进入导致不能启动*/
    sysOutByte(g_Chan[phy_no].addr + FCR, FIFO_THRES_14 | ENABLE_FIFO);
}
/*-----通道中断处理函数-----*/
static void v16C55x_int(BYTE int_no)
{
    BYTE phy_no;  DWORD iir_addr;  BYTE istat, lstat; int old;

    for(phy_no=0; phy_no<chan_max_num; phy_no++)
        if( g_Chan[phy_no].int_no == int_no ) break;

    if(phy_no >= chan_max_num) return;

    old = intLock();
    iir_addr = g_Chan[phy_no].addr + IIR;
    lstat = sysInByte(g_Chan[phy_no].addr + LSTAT);
    while( (! ((istat = sysInByte(iir_addr)) & INTR_NOTHING)) ) {
        switch (istat & INTR_ID_MASK) {
            case INTR_RCV_AVAIL:
            case INTR_RCV_ERR:
                lstat = sysInByte(g_Chan[phy_no].addr + LSTAT);
                v16c55x_int_rx(phy_no, lstat);
                break;
            case INTR_MOD_STAT:

```

```

        /*读 modem 状态寄存器以清除中断*/
        sysInByte(g_Chان[phy_no].addr + MODSTAT);
        break;
    case INTR_XMT_EMPT:
        v16c55x_int_tx(phy_no);
        break;
    default :
        break;
    }
}
intUnlock(old);
}
/*-----接收中断处理-----*/
static void v16c55x_int_rx(BYTE phy_no, BYTE lstat)
{
    DWORD lsr_addr, rx_addr;  BYTE data;  WORD minor; VSerBuf *pbuf;

    lsr_addr = g_Chان[phy_no].addr + LSTAT;
    rx_addr  = g_Chان[phy_no].addr + RX;
    minor    = g_Chان[phy_no].minor;

    pbuf = &(gaSerial[minor].rx_buf);

    while(1){
        if(!(lstat & LSR_DR)) return;
        data = sysInByte(rx_addr);
        /*data 填充缓冲区*/
        *(BYTE *) (pbuf->addr + pbuf->wp) = data;
        pbuf->wp = (pbuf->wp+1)&(SERIAL_BUF_NUM-1);

        lstat = sysInByte(lsr_addr);
    }
}
/*-----发送中断处理-----*/
static void v16c55x_int_tx(BYTE phy_no)
{
    DWORD lsr_addr, tx_addr;  BYTE lstat;  WORD minor; VSerBuf *pbuf;

    lsr_addr = g_Chان[phy_no].addr + LSTAT;
    tx_addr  = g_Chان[phy_no].addr + TX;
    minor    = g_Chان[phy_no].minor;
    pbuf = &(gaSerial[minor].tx_buf);
    /*循环发送缓冲区*/
    while(pbuf->rp != pbuf->wp){
        sysOutByte(tx_addr, *(BYTE *) (pbuf->addr + pbuf->rp));
        pbuf->rp = (pbuf->rp+1)&(SERIAL_BUF_NUM-1);

        lstat = sysInByte(lsr_addr);
        if(lstat & (LSR_OE | LSR_PE | LSR_FE | LSR_BI | LSR_DR) ||
            !(lstat & LSR_THRE) )
            break;
    }
    /*发送缓冲区空处理*/

```

```

lstat = sysInByte(lsr_addr);
if( (pbuf->rp == pbuf->wp) && (lstat & LSR_TENT) ){
    v16C55x_int_ctrl(phy_no, TXINT, 0);
}
}

```

10.4 移植实践

在实际的工作中，一般会有代码移植的工作。比如，可能需要引用其他平台的成熟组件代码，或者需要将自己的应用代码移植到其他平台上工作，或者需要在异种平台之间进行信息交流。

代码移植和编写时，需要注意一些常见问题，以提高移植工作效率。高低字节序问题和 CPU 相关，Intel 的 CPU 一般用 **little-endian** 模式（先低字节），Motorola 的 CPU 一般用 **big-endian** 模式（先高字节），而 ARM 芯片能选择两种模式。字节对齐问题一般在结构定义中出现，同样的结构在不同平台上有不同表现。数据类型大小问题一般和编译器相关，同样“int”类型可能有不同的解释。另外，软件平台接口不兼容也是移植工作的主要问题。

在编写代码时，需要注意代码的可移植性。最简单的方法是使用宏条件编译，用不同的宏来对应不同的平台。字节对齐问题一般可用编译器的伪指令解决，如果编译器无此功能，结构定义时可能需要填充字节，或者按字节流处理结构变量。不同软件平台间代码移植通常需要定义自己的抽象库接口，或者修改移植代码的库接口。

10.4.1 多操作系统移植

多操作系统间代码移植最常遇见的工作。移植有两种方式，一种是定义自己的统一接口；另一种是用新操作系统仿真旧操作系统接口，两者在本质上是一样的。

● 操作系统抽象层[OSAL]

定义自己的操作系统抽象层，需要在项目设计阶段就进行，也就是事先考虑未来可能出现移植工作。抽象层接口应根据实际的应用需求来定义，也需要参考大多操作系统所能提供的共性服务。抽象层接口一般是一个最小集合，不可能覆盖操作系统提供的所有服务，主要目的是将上层应用和实际的操作系统分离。当代码需要移植到新平台上时，只需要修改具体的抽象层实现，而不用在整个应用代码范围内进行修改。

操作系统抽象层一个额外的好处是能维持开发组习惯的编程风格，在新平台上工作也不会降低程序员的效率，也方便开发人员之间的交流。比如，作为 pSOS 用户，一般都习惯如下的任务编写风格。

```

void TaskFun()
{
    tmEvEvery(thid, 10/TICK2MSE, EV_TIMER1);
while (1) {
    evReceive(thid, EV_TIMER1|EV_DATA_RX, &events) < 0 )
if(events & EV_TIMER1){
    /*做一些定时循环处理*/
}
if(events & EV_DATA_RX){
    /*处理接收数据*/
    evSend(xxx, EV_DONEXT);    /*通知别上层任务继续处理数据*/
}
}

```

```
}  
}/*while(1)*/  
}
```

而在 VxWorks 上工作时，这种风格就难以继续，因为 VxWorks 不提供多事件等待机制（VxWorks 5.5 开始提供事件），定时器也不和任务绑定。但 VxWorks 提供一些近似机制，如信号量和 watchDog，可利用其来实现自己的操作系统抽象层，以保持编程风格。

参考 pSOS 的接口，笔者设计了自己的操作系统抽象层。在 VxWorks 上实现了事件、定时器、事件绑定消息等。细节参见前面章节的描述

在笔者的实际工作中，还需要在别的软硬件平台上编程，如 MC68K+Nucleus 和 8051 + μ C/OS 等。为了保持源代码基的统一，也在 Nucleus 和 μ C/OS 上实现了操作系统抽象层，为了节省篇幅，略去具体实现代码，有兴趣的读者可以通过前言中提供的 Email 和作者联系。不过基本实现原理是一样的，读者也可以参考前面的 VxWorks 实现编写自己的代码。

操作系统抽象层虽然可以提高代码的移植性，但在某些方面会损失系统的效率，也不能使用操作系统某些新特性。这种两难的折衷需要设计者根据项目情况决定。


● 仿真旧操作系统接口


如果在项目设计时未考虑平台移植问题，可以采用仿真旧操作系统接口的方式来完成代码移植。移植时，只需要用新操作系统服务实现出原来软件平台的接口，不需要对应用代码进行大的改动。

某些其他情况也会用到这种方式，比如，脱离硬件目标机，在 Windows 的 VC++ 中调试硬件无关的上层代码。一个大型的嵌入式系统一般会包含大量的上层代码，需要多个开发人员协作才能完成。而硬件开发设备一般是有限的，不能满足多个开发人员同时开发的要求。VC++ 开发环境大多数程序员都很熟悉，代码编写和调试都很方便，将上层应用脱离出来开发会提高效率。另外一个好处，在硬件设备出来前，代码编写和调试工作就可以提前进行，能提高整个项目的开发速度。所以，很多大公司都在 VC++ 环境中实现了仿真 VxWorks 接口的代码（VxWorks 任务用 Windows 线程来仿真）。

Tornado 也提供一个仿真开发环境（vxSim），能脱离目标机在独立主机上完成部分开发工作。但和上面讲的 VC++ 仿真实现还有不同，调试还是在 Tornado 中进行，某些方面不如 VC++ 中调试方便。

网上也有人在做这方面的工作，如铁峰（tiefeng@vip.sina.com）提供的一个实现，不过只公布了二进制库，没有源代码。

 铁峰，“嵌入式系统的 off-target 调试”，http://drew.nease.net/friends/tief_01.htm

 铁峰，“Windows API 实现 VxWorks 调用”，<http://drew.nease.net/friends/tief.htm>

10.4.2 升级到 Tornado 2.2

随着 WindRiver 公司推出升级版本 Tornado 2.2 和 VxWorks 5.5，大多数用户都面临升级移植的工作。本节描述升级时需要关注的一些问题，包括工程移植、BSP 移植、API 接口变化和编译器移植。

● 工程移植

Tornado 2.2 提供了一些新特性、升级工具和代码，遵循新的规范。老工程不能直接在

Tornado 2.2 上使用，需要借助“prjMigrate”工具完成移植过程。

首先启动 DOS 命令行窗口，并运行 torVars.bat 建立环境。运行 prjMigrate 命令。

```
prjMigrate -windbase F:\T20ppc -type vxWorks -newproject F:\T22ppc\target\proj\newImageProj  
F:\T20ppc\target\proj\oldImageProj.
```

如果转换的工程是 Downloadable 的，类型参数替换为“-type vxApp”。

PrjMigrate 命令有两种执行模式，自动[AUTO]模式和查询[QUERY]模式。上面的命令行示例使用自动模式。

QUERY 模式用于查询映射到 Tornado 2.2 中的编译宏或组件的值，不会执行实际工程转换。使用 QUERY 模式的命令格式如下。

```
prjMigrate -windbase your_old_WIND_BASE -bsp bsp_name -tool toolchain_name  
[  
  {  
    -component COMPONENT_NAME  
    | -macro build_macro [-value macro_value]  
  }  
]
```

其中“-tool”参数指定 Tornado 2.2 中使用的新工具链，一般为“gnu”或“diab”类型。后面的组件或宏参数设置与要查询的信息相关。

PrjMigrate 工具缺省只支持 WindRiver 提供 BSP 上的 VxWorks 工程移植，两个 Tornado 版本中都需要有该 BSP，缺省支持 BSP 可以查看 Tornado 2.2 中的 infolookupTables.tcl 文件。如果需要添加自己定制的 BSP，需要在该文件中添加一行描述，格式如下。

```
set cpuList (bspxxxx) MCF5200
```

为了在该定制 BSP 上移植工程，还需要手动移植该 BSP 到 Tornado 2.2 中。在使用 prjMigrate 移植工程前，确保 Tornado 2.2 能在移植 BSP 上建立工程。

如果工程移植时出现错误，可以查看日志文件（“\$WIND_BASE/target/proj/BSPNameMigrated/migration.log”）。

● BSP 移植

BSP 移植时有两种策略，一种是将旧 BSP 移植到 Tornado 2.2 中，使旧 BSP 和新软件环境兼容。另一种就是在 Tornado 2.2 的新 BSP 上进行修改，引入旧 BSP 定制，这种方法多用于 Tornado 2.2 中新添加的 BSP 类型。两种方式各有优缺点，前者工作量较小，可以快速完成；后者可以充分利用新版本提供的增强特性。后面主要针对第一种策略进行描述，不过对后者也有些提示作用。

BSP 移植时一般会遇到两种类型的问题，与体系结构无关的或与体系结构相关的。后面主要描述体系结构无关的方面。体系结构相关的移植一般改动较小，比如 Coldfire 的 BSP 需要修改 Makefile 的“TOOL”为“diab”，而不用原来的“gnu”，用户可以根据自己的 CPU 类型参考相关文档。

首先需要移植 BSP 的 Makefile，主要是修改一些编译规则的包含语句。Tornado 2.2 提供了工具“bspCnvtT2_2”用于转换 Makefile，可同时多个 BSP 操作，命令格式如下。

```
bspCnvtT2_2 bspName1 bspName2 ...
```

该工具会备份原来的 Makefile 为 Makefile.orig，注释掉一些不用的“include”语句和 HEX_FLAGS，并对一些 HEX 格式命令提出警告（Tornado 2.2 推荐使用 GNU 的 objcopy）。

对于使用 TFFS 和 dosFS 的用户，要注意适应两组件的升级变化。如果用户以前使用

Tornado 2.1 或 dosFs 2.0, 则不需要移植。

● VxWorks API

VxWorks 5.5和VxWorks 5.4接口大部分兼容, 应用代码只需要少量修改, 就可在VxWorks 5.5上运行。API改变分为3类: 修改的函数、过时库和新接口。新接口对程序移植没有影响, 如果需要使用新功能(如事件), 可在代码移植完成后再修改。

影响应用的 API 修改涉及很多函数和库, 用户可以参考 WindRiver 提供的移植手册, 上面有详细的列表。用户可以一一查看各条目, 对照自己的应用代码, 看是否需要修改。

 WindRiver, “TechTips-BSP Migrating Tips from Tornado 2.x to Tornado 2.2”

 WindRiver, “Tornado 2.2 Migration Guide”。

10.4.3 goAhead 移植实例

VxWorks 中自带了 WebServer 组件, 在 “network applications” 下选择 “http server” 即可。也可以采用 “rapid control for Web”。这里介绍 GoAhead WebServer, 它是一个源码免费、功能强大、可以运行在多个平台的嵌入式 Web Server。

GoAhead WebServer 的主要特性有:

- ✧ 支持 ASP;
- ✧ 嵌入式的 JavaScript;
- ✧ 标准的 CGI 执行;
- ✧ 内存中的 CGI 处理 GoForms;
- ✧ 扩展的 API;
- ✧ 快速响应, 每秒可处理超过 50 个请求;
- ✧ 完全和标准兼容;
- ✧ 如果不包含 SSI, 仅要求 60KB 的内存; 包含 SSI, 要求 500KB 内存
- ✧ Web 页面可以存在 ROM 或文件系统中;
- ✧ 支持多种操作系统, 包括 eCos、Linux、LynxOS、QNX、VxWorks、WinCE 和 pSOS 等。

下面讲述一下通过 Downloadable 工程来架设 GoAhead WebServer 的过程, 当然也可以用类似的方法把它集成到 Bootable 工程中。

- (1) 到 <http://12.129.4.11/webserver/webserver.htm> 下载最新的 GoAhead Webserver。
- (2) 建立基于 BSP 的 Downloadable 工程, 名称为 goaheadweb。
- (3) 将下列下载的源文件加入到工程中:

balloc.c、base64.c、default.c、ejlex.c、ejparse.c、emfdb.c、form.c、h.c、handler.c、md5c.c、mime.c、misc.c、page.c、ringq.c、rom.c、security.c、sock.c、sockGen.c、sym.c、uemf.c、um.c、umui.c、url.c、value.c、webrom.c、webs.c、asp.c、websuemf.c、cgi.c、/vxworks/main.c。

- (4) 参考 “/vxworks/makefile” 来修改工程的 Makefile。

在 Makefile 中增加定义 “-DWEBS -DUEMF -DVXWORKS”, 如下所示:

```
CFLAGS = -g -m486 -ansi -nostdinc -DRW_MULTI_THREAD -D_REENTRANT -DWEBS -DUEMF -DVXWORKS  
-fvolatile -nostdlib -fno-builtin -fno-defer-pop -I. -IE:/Tornado-x86/target/h -DCPU=I80486
```

如果需要增加用户管理、存取控制支持, 则还需要增加下列定义:

```
-DUSER_MANAGEMENT_SUPPORT  
-DDIGEST_ACCESS_SUPPORT
```

- (5) 根据文档修改 “/vxworks/main.c”。

设置根目录，例如：

```
#define ROOT_DIR T("/tffs0/webs")
```

设置缺省主页：

```
websSetDefaultPage(T("index.asp"));
```

利用 FTP 等工具将 index.asp 等主页下到 “tffs0” 中。

- (6) 编译为 goaheadweb.out 下载。

- (7) 在 WindSh 下加载 WebServer。

```
sp websvxmain
```

- (8) 在浏览器中键入 “<http://192.168.0.2/index.asp>” 就可以访问到 “/tffs0/webs/index.asp” 文件了。

- (9) GoAhead ASP 编程首先需要预先定义 ASP 函数，在 main.c 的函数 initWebs 中有如下的例子。

```
websAspDefine(T("aspTest"), aspTest);
```


这里将 aspTest 定义为 ASP 函数，用户可提供相关的函数来完成想要的功能，例如下面所示的代码：

```
/* Test Javascript binding for ASP. This will be invoked when "aspTest" is
 * embedded in an ASP page. See web/asp.asp for usage. Set browser to
 * "localhost/asp.asp" to test.
 */
static int aspTest(int eid, webs_t wp, int argc, char_t **argv)
{
    char_t *name, *address;

    if (ejArgs(argc, argv, T("%s %s"), &name, &address) < 2) {
        websError(wp, 400, T("Insufficient args\n"));
        return -1;
    }
    return websWrite(wp, T("Name: %s, Address %s"), name, address);
}
```

然后在 ASP 的页面中调用 ASP 函数即可，如调用上述例子。

```
<h2>Expanded ASP data: <% aspTest("Peter Smith", "112 Merry Way"); %></h2>
```

 如果要停止或重起 WebServer，不要使用 taskDelete 或者 td 来删除 WebServer 任务，而是调用 kill 来给 WebServer 任务发送 9 或者 15 信号。

10.5 常见问题解答

- 用类成员函数作为任务入口的问题

```
#include "vxWorks.h"
#include "taskLib.h"
#include "stdio.h"
class my_class
{
public:
    my_class();
    int Task( void );
};
my_class::my_class()
```

```

{
    taskSpawn( "Task", 90, 0, 10000, (FUNCPTR)Task, 0, 0, 0, 0, 0, 0, 0, 0, 0 );
}
my_class::Task()
{
    while( 1 ){
        taskDelay(100);
        printf("Hi\n");
    }
}

```

上面这段代码用 Tornado 2.0 时能正常工作, 只是 gcc2.72 编译有警告。而用 Tornado 2.2 的 gcc2.96 却不能编译通过, 出现如下错误。该如何解决此问题?

```

..\post.cpp: In method `my_class::my_class()':
..\post.cpp:16: no matches converting function `Task' to type `int(*) (...)'
..\post.cpp:9: candidates are: int my_class::Task()
make: *** [post.o] Error 0x1

```

解答: 因为新编译器更为严格, 可以将代码修改如下。

```

class my_class
{
public:
    my_class();
    int Task( void );
    static int TaskEntry(my_class *myClass);
};
int TaskEntry(my_class *myClass)
{
    return myClass->Task();
}

```

其中使用了静态成员函数作为任务的入口, 并将对象实例指针显式地传给任务, 任务中需要用该指针来访问其他的类成员。

附录 A 参考资源

A.1 参考文献

- [01]. WindRiver, “Tornado 2.0 Getting Started Guide”。
- [02]. WindRiver, “Tornado 2.0 User’s Guide”。
- [03]. WindRiver, “Tornado 2.2 User’s Guide”。
- [04]. WindRiver, “Tornado Reference”。
- [05]. WindRiver, “Tornado API Reference”。
- [06]. WindRiver, “Tornado 2.0 API Programmer’s Guide”。
- [07]. WindRiver, “GNU Toolkit User’s Guide”。
- [08]. WindRiver, “GNU Make Version 3.74”。
- [09]. WindRiver, “Debugging with GDB The GNU Source-Level Debugger 4.17”。
- [10]. WindRiver, “Tornado Training Workshop”。
- [11]. WindRiver, “Vxworks 5.4 Programmer’s Guide”。
- [12]. WindRiver, “Vxworks 5.4 Reference Manual”。
- [13]. WindRiver, “BSP Developer’s Kit User’s Guide”。
- [14]. WindRiver, “BSP Reference”。
- [15]. WindRiver, “VxWorks Network Programmer’s Guide”。
- [16]. WindRiver, “VxWorks Network Protocol Toolkit Programmer’s Guide”。
- [17]. WindRiver, “WindView User’s Guide”。
- [18]. WindRiver, “dosFs for Tornado 2.0 Release Notes and Supplement 2.0”。
- [19]. WindRiver, “TrueFFS for Tornado Programmer’s Guide 1.0 Edition 1”。
- [20]. WindRiver, “PCMCIA for x86 Release Notes and Supplement”。
- [21]. WindRiver, “Tornado 2.2 Migration Guide”。
- [22]. WindRiver, “Tornado Training Workshop”。
- [23]. WindRiver, “Tornado BSP Training Workshop”。
- [24]. WindRiver, “Tornado Device Driver Workshop”。
- [25]. WindRiver, “WhitePaper”
- [26]. WindRiver, “TechTip”
- [27]. WindRiver, “AppNote”

- [28]. M-Systems, “Two Technologies Compared: NOR vs. NAND White Paper”。
- [29]. Logix, Rhapsody4.1的“RTOS Adapter Guide”文档 (rtos.pdf)。

- [30]. JohnGordan, “VxWorks CookBook”。
- [31]. Mitch Neilsen, “CIS 721, Lecture 13, Priority inheritance protocols and vxworks”。
- [32]. Kurt Keutzer, “Software Enviroments for Embedded Systems”, 2000.03。
- [33]. Pierre-Alain Darlet, “Runtime Loader-Linker Technologies”, 2001.04, ESC。
- [34]. Ron Fredericks, “FAQ: What is a Board Support Package?”, Wind River, 2002.12。
- [35]. “Google 的 VxWorks 和 TcpIp 讨论组”。
- [36]. “VxWorks Users Group mailing list”。

- [37]. 孔祥营 等, “嵌入式实时操作系统 VxWorks 及其开发环境 Tornado”, 2002.01, 中国电力出版社。
- [38]. 罗国庆 等, “VxWorks 与嵌入式软件开发”, 2003.09, 机械工业出版社。
- [39]. W.Richard Stevens, “UNIX Network Programming”, 清华大学出版社。

- [40]. 郑更生等, “基于 Vxworks 的产品映象设计”, 电子设计应用, 2003.04。

- [41]. 秦遵明、黄峰,“CASE 工具 Rhapsody 在综合接入服务器中的应用”。
- [42]. 孙强、张振华,“使用 Rhapsody 软件框架和 UML 的实时系统开发”。
- [43]. 徐异婕,“测试跟踪工具 Bugzilla 介绍”,UML 软件工程组织,2002.05。
- [44]. seasoblue,“51 flash 文件系统 DIY”,<http://bbs.edw.com.cn>,2002,05。
- [45]. alahover,“VxWorks 文件系统”,<http://bbs.edw.com.cn>,2002,12。
- [46]. gem2000,“vxworks 压缩技术”,<http://bbs.edw.com.cn>,2003.04。
- [47]. longyong,“VxWorks 的启动方式”,<http://bbs.edw.com.cn>。
- [48]. amine,“Vxworks 新手探路系列”,<http://bbs.edw.com.cn>,2002.04。
- [49]. vxFree,“MPC860 上电初始化流程分析”。
- [50]. VxFree,“X86 romInit.s 分析”。
- [51]. 铁峰,“嵌入式系统的 off-target 调试”,http://drew.nease.net/friends/tief_01.htm。
- [52]. 铁峰,“Windows API 实现 vxWorks 调用”,<http://drew.nease.net/friends/tief.htm>。
- [53]. drew,“关于 BSP 配置精华”,<http://drew.diy.163.com>。
- [54]. 寄存器,“NAND 和 NOR flash 详解”,C51BBS 论坛。
- [55]. “VxWorks 串行设备驱动的编写”。
- [56]. “Vxworks 培训讲稿”。
- [57]. “VxWorks 操作系统指南”。

A.2 基础书籍

- [01]. 吕骏,“嵌入式系统设计”,电子工业出版社。
- [02]. 张晓林,“嵌入式系统固件揭秘”,电子工业出版社。
- [03]. 路晓村,“嵌入式系统 TCP/IP 应用层协议”,电子工业出版社。
- [04]. 张君施“嵌入式软件测试”,电子工业出版社。
- [05]. 绍贝贝,“ μ C/OS-II——源码公开的实时嵌入式操作系统“,中国电力出版社。
- [06]. 于志宏,“C/C++嵌入式系统编程”,中国电力出版社。
- [07]. 孙玉芳等,“嵌入式计算系统设计原理”,机械工业出版社。
- [08]. 陈向群,“嵌入式系统 Web 服务器: TCP/IP Lean”,机械工业出版社。
- [09]. 裘宗燕,“程序设计实践”,机械工业出版社,2000.08。
- [10]. Bruce Powel Douglass,“Real-Time UML”,科学出版社。
- [11]. Hassan Gomaa,“Designing Concurrent,Distributed,Real-Time Applications with UML”,科学出版社。
- [12]. Albert M. K. Cheng,“Real-Time Systems: Scheduling, Analysis, and Verification”, John Wiley & Sons,2002。
- [13]. Bruce Powel Douglass,“Real-Time Design Patterns”,Addison-Wesley,2002.09。
- [14]. Bruce Powel Douglass,“Doing Hard Time”,Addison-Wesley,1999.05。
- [15]. D.M. Auslander,“Design and Implementation of Real Time Software”。
- [16]. Frank Vahid,“Embedded System Design:A Unified Hardware/Software Approach”,1999。
- [17]. Herman Bruyninckx,“Real-Time and Embedded Guide”,2002。

- [18]. Jack Ganssle, “The Art of Designing Embedded Systems”, Newnes, 1999.11。
- [19]. Jack Ganssle & Michael Barr, “Embedded Systems Dictionary”, CMP, 2003。
- [20]. John Levine, “Linkers and Loaders”, Morgan Kaufmann。
- [21]. Jürgen Sauermann & Melanie Thelen, “Concepts and Implementation of Microkernels”。
- [22]. Ken Arnold, “Embedded Controller Hardware Design”, LLH, 2000。
- [23]. Manu Bhatia, “Component Based Software Engineering to Design Real-Time Software”, 1999。
- [24]. Miro Samek, “Practical Statecharts in C/C++ Quantum Programming for Embedded Systems”, CMP, 2002.07。
- [25]. Mathai Joseph, “Real-time Systems Specification, Verification and Analysis”, Prentice Hall, 1995.12。
- [26]. National Research Council, “Embedded Everywhere”, National Academy Press。
- [27]. Qing Li & Carolyn Yao, “Real-Time Concepts for Embedded Systems”, CMP, 2003。
- [28]. R. J. Wieringa, “Design Methods for Reactive Systems”, Morgan Kaufmann, 2003。
- [29]. Robert Betz, “Introduction to Real-Time Operating Systems”, 2001。
- [30]. Ross Albert Mckegney, “Application of Patterns to Real-Time Object-Oriented Software Design”, 2000.07。
- [31]. T. Sridhar, “Designing Embedded Communications Software”, CMP, 2003.07。
- [32]. Ted Van Sickle, “Reusable Software Components”, Prentice Hall, 1996.11。

A.3 网络资源

Wind River 公司

<http://www.windriver.com/>

WindRiver 公司的官方网站, 提供论坛, 补丁, techtip

<http://www.windriver.com/windsurf>

正式用户的技术支持网站, 用 license 号登录。可以获得各种文档, 包括正式手册、TechTips 和 AppNote。用户可以提交 TSR, 下载 SPR 列表 (或 Proactive Alert 邮件), 下载更新补丁等。网站还提供一个技术论坛, 回答很专业, 只是人比较少。

<http://developer.windriver.com/>

风河开发者网络, 非正式用户也可以登录。

Wind River 的第 3 方公司

<http://www.windriver.com/partnerships/directory/index.html>

可查找与 WindRiver 合作的第 3 方厂商。

北京奥索公司

<http://www.autosoft.com.cn/index.htm>

WindRiver 在中国的最大代理商

Google VxWorks 讨论组

<http://groups.google.com/groups?hl=zh-CN&group=comp.os.vxworks>

VxWorks 邮件组

<http://www-csg.lbl.gov/mailman/listinfo/vxwexplo>

The VxWorks Archive

<ftp://ftp.atd.ucar.edu/pub/archive/vxworks/vx>

一些老的 VxWorks 软件组件，可能对平台扩展有帮助。

VxWorks and Tornado II FAQ

<http://www.xs4all.nl/~borkhuis/vxworks/vxworks.html>

JohnGordan 的公开发表的 vxworks 相关的电子书籍

<http://www.bluedonkey.org/cgi-bin/twiki/bin/view/Books/VxWorksCookBook>

书中共包含 14 章，包括内核，内存管理，IO 系统，网络，文件系统等章节。未完全完成，最新版本为 2003-4-26 的 r1.7

<http://www.bluedonkey.org/cgi-bin/twiki/bin/view/Books/MigratingVxWorksToLinux>

VxWorks 向 Linux 移植的指南。

RTOS State of the Art Analysis

<http://www.mnis.fr/opensource/ocera/rtos/book1.html>

Ismael Ripoll 网上发布的电子书，对多种 RTOS 进行了分析，包括 VxWorks。

Embedded System Programming

<http://www.embedded.com/>

嵌入式系统软件开发相关的最出色的电子杂志，还可以下载 ESC 会议论文。

Jack Ganssle 的个人网站

<http://www.ganssle.com/>

Jack Ganssle 是 ESP 的常务编辑，负责 “Break Points” 专栏。

电子产品世界论坛

<http://bbs.edw.com.cn/>

作者 Amine 个人主页

<http://amine.nease.net/>