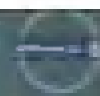


VxWorks 开发人员的在线书



VxWorks BSP开发人员指南

美 [Wind River] 著

王雷彪 译 周 杨 编

清华大学出版社

VxWorks 开发人员指南丛书

VxWorks BSP 开发人员指南

[美] Wind River 著

王金刚 苏 琪 杨锡劭 译

清华大学出版社

北 京

内 容 简 介

本书是《VxWorks 开发人员指南丛书》的 BSP 开发分册,根据 Wind River 公司的技术文档 *VxWorks 5.5 BSP Developer's Guide* 翻译而成。

主要内容包括: BSP 组织部分,讲述 BSP 文件和 VxWorks 引导顺序;创建新 BSP,讲述如何编写初始化代码等;硬件指南部分,讲述 CPU 分类结构、存储器和其他硬件设备、驱动程序指南、BSP 编写规范、测试、归档和产品打包等内容;还讲述了 BSP 升级、CPU 结构、VxWorks 引导顺序、组件描述语言、通用驱动程序、时间戳驱动程序、网络驱动程序、SCSI 驱动程序等内容。

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

版权所有,翻印必究。

本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。

图书在版编目(CIP)数据

VxWorks BSP 开发人员指南/(美)风河公司著;王金刚,苏琪,杨锡勋译.

—北京:清华大学出版社,2003

(VxWorks 开发人员指南丛书)

ISBN 7-302-06895-X

I.V… II.①风…②王…③苏…④杨… III. 实时操作系统,VxWorks—指南
IV. TP316.2-62

中国版本图书馆 CIP 数据核字(2003)第 056886 号

出 版 者:清华大学出版社

<http://www.tup.com.cn>

社 总 机:010-62770175

地 址:北京清华大学学研大厦

邮 编:100084

客户服务:010-62776969

组稿编辑:曾 刚

文稿编辑:许存权

封面设计:秦 铭

版式设计:俞小红

印 刷 者:北京国马印刷厂

发 行 者:新华书店总店北京发行所

开 本:185×260 印张:21.75 字数:497 千字

版 次:2003 年 9 月第 1 版 2003 年 9 月第 1 次印刷

书 号:ISBN 7-302-06895-X/TP·5106

印 数:1~5000

定 价:30.00 元

授权声明

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

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

WIND RIVER SYSTEMS, INC.

By: 

Name, title *Steve Kennedy*
Group Vice President

Date *3-4-03*



前 言

当前,国内单片机技术的研究与应用已经非常成熟,而对面向高端产品的嵌入式系统的研究正在国内悄然兴起,以 ARM 核为代表的 32 位嵌入式处理器的价格不断降低,技术也不断成熟,嵌入式系统必将有非常广阔的应用前景。

随着 Internet 的普及,我们已经进入了后 PC 时代。在后 PC 时代,出现了很多信息家电,如掌上电脑、个人数字助理(PDA)、可视电话、移动电话、TV 机顶盒、电视会议机和数码相机等嵌入式设备。这些嵌入式设备的功能都非常强大,其中的大部分都嵌入了 TCP/IP 网络协议。这样,不仅可以用 PC 机上网,使用各式各样的嵌入式设备同样可以上网,在网络时代,这给用户带来了极大的方便。而在这些强大的功能背后,少不了各种嵌入式操作系统的支持。

VxWorks 操作系统是美国 Wind River 公司推出的一种嵌入式实时操作系统。自 20 世纪 80 年代问世以来,以其不断推出的升级版本、高性能内核以及友好的用户开发环境,在嵌入式实时操作系统领域逐渐占据一席之地,在国内外拥有很多用户。尤其已成功应用于火星探测车和爱国者导弹等高科技产品而声名鹊起。自 1996 年登录中国以来,越来越多地占据了国内嵌入式实时应用市场。

在 VxWorks 操作系统与目标板底层硬件之间,还有一个重要部分——板级支持包 BSP (Board Support Package)。板级支持包的开发,在整个嵌入式系统的开发过程中起着至关重要的作用。板级支持包对各种板子的硬件功能提供了统一的软件接口,它包括硬件初始化、中断的产生和处理、硬件时钟和计时器管理、本地和总线内存地址映射、内存空间大小、实时内核载入等。每个板级支持包包括一个 ROM 启动(Boot ROM)或其他的启动机制。

本书是根据 Wind River 公司提供的技术手册《VxWorks 5.5 BSP DEVELOPER'S GUIDE》翻译而成,并且在翻译过程中完全保持了原书风貌。有关 BSP 开发的资料, Wind River 公司一直是对外保密的。在这部分资料公开伊始,我们便在第一时间对其进行了翻译,希望它能尽快出版,尽早与广大读者见面。我们相信,本书中的内容一定会对众多的 VxWorks 开发人员提供帮助。

本书主要包括两大部分:第一部分为正文,在这一部分中包含了 BSP 代码的编写、注释,以及测试等各个方面的全部内容,详细地讨论了适用于 C 语言源代码和注释的规范标准,而且还提供了有关如何编写网络接口驱动程序、SCSI 设备驱动程序,以及其他设备驱动程序等内容;第二部分为附录,在附录中也包括了很多 BSP 开发中十分重要而且很实用的知识。

天津大学电子信息工程学院的王金刚教授,以及苏琪、杨锡勋、冀清颖、谢子薇和颜祺等负责了本书的全部翻译工作。在本书的翻译过程中,还得到了 Wind River 公司的大力支持。官霄霖和丁大尉等几位研究生也牺牲了很多宝贵的时间,为本书的校对及出版作出了大量的工作,在此一并表示衷心的感谢!

由于时间仓促,且这部分内容目前在国内尚无中文资料可供参考,因此在翻译中错误与疏漏之处在所难免,恳请国内广大同行批评、指正。

译 者

2003 年 6 月于天津大学

目 录

第 1 章 绪论	1
第 2 章 BSP 结构	7
2.1 简介	7
2.2 BSP 组件	7
2.3 VxWorks 引导顺序	22
2.4 需要避免的错误	27
第 3 章 创建一个新的 BSP	29
3.1 概述	29
3.2 建立开发环境	30
3.3 编写 BSP Pre-Kernel 初始化代码	31
3.4 使用最小化的内核	40
3.5 目标板代理和 Tornado	42
3.6 完成安装工作	42
第 4 章 硬件介绍	47
4.1 概述	47
4.2 结构体系注意事项	47
4.3 内存	50
4.4 设备	56
4.5 启用虚拟内存库	61
第 5 章 驱动程序指导	65
5.1 绪论	65
5.2 设计目标	65
5.3 设计的问题	67
5.4 设计指导	69
5.5 逐步实现	72
5.6 关于缓存的问题	74
5.7 有益的忠告	74
5.8 驱动程序使用缓存器库	75

第 6 章 组件	81
6.1 简介	81
6.2 组件描述语言	82
6.3 创建组件	91
6.4 发布组件	99
第 7 章 确认测试	102
7.1 简介	102
7.2 后台	102
7.3 配置 BSP VTS	105
7.4 运行 BSP VTS	112
第 8 章 编写可移植 C 程序	117
8.1 可移植 C 代码	117
8.2 执行工具	125
第 9 章 文件指导	128
9.1 概述	128
9.2 书写风格	128
9.3 格式	132
9.4 子段	135
9.5 创建参考页	139
第 10 章 产品封装	144
10.1 安装开发工具包	144
10.2 BSP 封装	144
10.3 组件封装	147
10.4 工程封装	148
附录 A 升级 BSP 以兼容 Tornado 2.0	149
A.1 过渡问题	149
A.2 代码接口的改变	149
A.3 工程问题	150
A.4 产品内容	151
A.5 小结	153
附录 B 升级 BSP 以兼容 Tomado2.2	155
B.1 对 BSP 进行独立于体系结构的修改	155

B.2 独立于体系结构的 BSP 问题.....	157
附录 C 升级 BSP 以兼容 VxWorks 网络堆栈	163
附录 D VxWorks 引导顺序	165
附录 E 组件语言	171
E.1 组件描述语言 (CDL)	171
E.2 文件夹等级	176
E.3 工程初始化顺序	179
附录 F 普通驱动程序	188
F.1 介绍	188
F.2 串行驱动程序	188
F.3 多模式串行驱动程序 (SIO)	189
F.4 定时器	191
F.5 非易失性存储器	192
F.6 VME 总线	193
F.7 DMA	193
F.8 中断控制器	193
F.9 多功能	193
F.10 PCI 总线	194
附录 G 升级 4.3 BSD 网络驱动程序	195
G.1 简介	195
G.2 升级到 4.4 BSD	197
G.3 转换一个网络接口驱动程序到最终模块	201
附录 H 执行基于多用户 (MUX) 的网络接口驱动程序	203
H.1 简介	203
H.2 VxWorks 如何装载并使用驱动程序	205
H.3 处理驱动程序包接收问题的指南	209
H.4 指明错误条件	213
H.5 所需的驱动程序入口点和结构	214
H.6 编写使用 MUX API 的网络协议	228
H.7 将一个 END 驱动程序从 RFC1213 转化到 RFC2233	236
附录 I 编写 SCSI-2 设备驱动程序	242
I.1 概述	242

I.2	SCSI 总括.....	242
I.3	SCSI 库.....	245
I.4	SCSI 驱动程序编程接口.....	248
I.5	BSP 接口	280
I.6	SCSI 驱动器开发指导.....	282
I.7	测试程序组.....	283
I.8	故障查找以及调试	285
附录 J BSP VTS 参考条目		287
附录 K BSP 合法性检查表		318
附录 L Refgen		328
附录 M BSP 产品目录.....		334

第 1 章 绪 论

VxWorks 操作系统的一个强项是它可以使应用程序编码在很大程度上与目标板的硬件和结构无关。这种便利功能是由于 VxWorks 操作系统的模块化设计,它把所有特定的硬件功能都集成在一个被称作板级支持包 Board Support Package(BSP)的库中。BSP 库为所有的硬件功能板提供了一种相同的软件界面。它包括硬件初始化,中断的产生和处理,硬件时钟和计时器管理,本地和总线存储空间的规划,存储容量管理等功能。

广泛地讲, VxWorks 有下列几种移植操作:

- 主机移植操作: 在以前不支持此系统的主机上安装 Tornado 和 VxWorks 的开发功能。
- 结构移植操作: 在以前不能支持此功能的目标处理类型和结构中安装 VxWorks 和 Tornado 系统。
- 目标板移植操作: 在一个新的目标板上安装 VxWorks 操作系统, 该目标板已经满足结构和主机的安装要求。
- 升级以前 Tornado 版本的 BSP 到最新版本。
- 将一个可选组件安装在 Tornado 中。

主机移植操作和结构移植操作需要访问整个 VxWorks 的源码, 这已经超出了本书范围。目标板移植操作是本书的重点, 它只需访问在 VxWorks 中和其调试器范围内与硬件相关的代码, 这一源代码就是 BSP。

几年来, Wind River 公司为用户提供了一系列广泛的 BSP。但是, 随着许多 CPU (其中许多是用户自己设计的) 的发展, Wind River 公司的研制并没能保持与新 BSP 的需求一致。

《Tornado BSP 开发工具包》将会帮您创建、编写并且测试新的 BSP 及其组件和工程。在新的目标板上安装 VxWorks 的过程可能很具挑战性, 首先因为 VxWorks 的开发工具在安装完成之前都无法使用。Wind River 公司制定了开发策略, 从而使安装更加容易。这个 BSP 工具包提供了这些策略和对系统各部分如何相互作用的详细介绍。它同时也包括一些第三方所需的创建、分配基于 Tornado 2.x 的各组件和方案的技术信息。

这个 BSP 工具包的版本是和 Tornado 2.0 以及更高版本相兼容的。将以前版本较低的 BSP 升级以使其与 Tornado 2.x 产品相兼容是一项相对简单的工作。如需更多的信息, 可查阅附录 A、B。

1. Tornado 2.x 的新特征

- 不再使用 SCSI 1。支持 SCSI 1 的驱动程序和工具库已不再使用。在 Tornado 2.0 中仍将包括这个工具库, 但它已经过时了, 并不包含在新的版本中。
- 删除宏 TYCOS_2。不再支持宏 TYCOS_2。用户需使用 SIO 驱动程序来支持 Tornado

2.x。

- **修改 BSP 工具包。**有关组装需求的详细介绍参见“10.2 节 BSP 封装”。请查阅附录 M，这是一个典型的产品目录摘要。
- **文档。**现在提供的所有联机文档都是 HTML 格式。系统中包含一种被称作 `refgen` 的应用程序，它是从现有的 `nroff` 输入中产生 HTML 格式文档。
- **VxWorks 网络堆栈。**网络堆栈对于 Tornado 1.0.1 的用户是一个选择项，它现在是惟一被支持的堆栈。它称作 VxWorks 网络存储器。所有的 BSP 都使用驱动程序，这些驱动程序使用 VxWorks 网络堆栈。在 BSP 可以使用 BSD44 和 END 驱动程序，END 驱动器被选为 BSP 的默认驱动器。
- **新的驱动器。**许多新的 END 网络驱动器已经被添加到驱动器产品中，这样就可以为尽可能多的 BSP 提供至少一个可用的 END 驱动程序。
- **集成模拟器。**Tornado 2.x 的基本产品包括一个集成模拟器。这就让开发者可以在硬件环境准备好之前就开始应用程序的开发。
- **集成 WindView 工具。**Tornado 2.x 产品包含一个集成的 WindView 工具来检查产品的即时运行情况。有关 WindView 的特性参见《WindView 用户指南》。
- **VTs 测试资料。**有效性测试套件只做了很少的改进，它可以很好的适应目标机命令解释器新增加的虚拟 I/O 特性。
- **SCSI 测试套件。**在以前的端口操作工具包版本中所没有的 SCSI 性能测试套件现在已包含在 CD-ROM 中了。
- **ROM_WARM_ADRS。**在一些 BSP 模板和已存在的 BSP 中已经添加了这个最新的宏程序。它为 boot ROM 编码提供了热启动的入口通道，并且它省去了人为计算基于某一类 CPU 的入口地址。通常来说，ROM_WARM_ADRS 是基于 ROM_TEXT_ADRS 的偏移量，比如说：

```
#define ROM_WARM_ADRS      (ROM_TEXT_ADRS + 8)
```

- **新的 SIO IOCTLs。**已经编写出新的 SIO IOCTL 程序来支持调制解调器的控制线。当设备打开和关闭时，IOCTLs 的 SIO_OPEN 和 SIO_HUP 命令将会通知驱动程序。新的 IOCTLs SIO_HW_OPTS_SET 和 SIO_HW_OPTS_GET 命令使得用户能够控制字符的长度，奇偶性，中止字节和调制解调器控制线。而可选控制位 CLOCAL 则使得调制解调器控制线在驱动程序中禁止使用。在后向兼容性上，CLOCAL 是 Wind River 公司提供的 BSP 中默认的选择项。
- **删除 target.txt 文件。**删除了 target.nr 文件的文本格式文件（用于 Windows 主机用户）。UNIX 和 Windows 主机用户现在都使用浏览器来阅读 HTML 格式下的文件。主机中的 `refgen` 应用程序使用 `nroff` 输入并产生 HTML 输出。

2. 先决条件

BSP 工具包的用户应该具备一些初级 C 语言和汇编语言的基本知识，熟悉驱动器装置

和中断设备，以及一些软硬件相结合的操作经验。另外，用户必须熟悉 Tornado 的开发工具（至少应该看过《Tornado 用户指南》）。最后，用户应该参考一个基于相同 CPU 的已发行 Tornado BSP 版本。

用户应该有基于自己结构的 BSP 模板和驱动器模板。大部分客户也都购买一个或更多的驱动源代码产品。

3. BSP 工具包组件

BSP 工具包包含一个文件编制装置和带驱动程序的 CD-ROM。这个文件编制装置包含一个安装指导和这个手册，这在下面的文件中有概括。而这个软件包含以下几点：

- **BSP VTS。**这些程序的目的是要检查 BSP 的基本功能，并且在这个过程中通知和报告任何故障。这些程序涉及目标板器和其所支持的主机。可以任意地增强已存在的测试功能或创建新的程序。但是，由 VTS 提供的基本测试套件是判断 BSP 的标准。

这个测试套件是高度自动化的。一旦开始，BSP 的测试将不需要用户的介入，它可以确保测试的可重复性并且也降低了手动操作的乏味性。

VTS 被分布在各种代码形式中用来维持和扩展。当进行正式的测试确认时，测试不能修改其初始代码。

- **BSP 模板。**这些模板是为所有结构类型提供的。它们为 BSP 的开发提供了一个初始介入点。每一个模板都可以用来编辑，但是禁用了几乎所有的任选特点。这就允许从模板目录的副本中快速启动，并且不需要作任何变化。
- **模板驱动器。**我们为所有类型的装置提供了模板驱动器。
- **SCSI 测试套件。**包括了一套 SCSI 测试程序。涉及到测试程序的内容请查阅 `target/src/test/scsi` 目录。

当使用 Tornado 时，需要一个与用户规定结构相对应的参考 BSP。这是一个完全运行的 BSP。在所创建的 BSP 中使用这个参考 BSP 作为许多头文件和对象模块的来源。

对于如何安装 BSP 工具包、参考 BSP，或者其他 Wind River 公司的软件产品，可参阅《Tornado 入门指导》。

4. 文件

BSP 工具包文件包含《Tornado 入门指导》和本书。

(1) 关于本书

用户指南包含各方面的信息，包括：书写、文档制作以及 BSP 测试。本书详细叙述了使用 C 语言编码和文件的标准，以及关于如何书写网络界面驱动程序、SCSI 驱动器和其他驱动器的信息。本书各章内容概括如下：

第 1 章，绪论（本章内容）

第 2 章，BSP 结构。本章是对 BSP 结构和需求的总括，它讨论了必须的和任选的文件和常规程序。这里概述了如何构成 BSP 及其与 VxWorks 如何互相影响。

第 3 章, 创建一个新的 BSP。本章介绍了 BSP 的安装策略。这里讨论了一些使用不同的硬件配置的启动方法。在新板上安装 VxWorks 的最困难的部分是不需要引导 ROM 来启动。

第 4 章, 硬件介绍。本章讨论 CPU 结构的考虑以及不同的硬件部分如何影响 BSP 的安装。

第 5 章, 驱动程序指导。本章概述了装置驱动器的使用, 关于驱动的详述可在一些特定装置类型的附录中找到。

第 6 章, 组件。这章叙述了 BSP 各部分如何交互工作, 提供了各部分创建和使用的基础。

第 7 章, 确认测试。本章描述了 BSP VTS 的运行和使用, 并搜集在主机和目标板上的运行程序来检查 BSP 的基本功能和检测并报告所发现的错误。关于测试套件程序和单独测试的文件可在附录 J 中找到, 以下对 BSP 测试的核对列表可在附录 K 中找到。

第 8 章, 编写可移植 C 程序。本章描述了如何将可移植 C 代码程序写入独立编码器, 目的是编写不需要改变的编码程序, 使得不同的编码器都能正确工作。

第 9 章, 文件指导。它包含了 Wind River 公司传统的类型和格式, 以及从模块源中产生 BSP 文件的过程。与 BSP 工具包一起提供的 BSP 模板给出了写入类型、文件格式、模块布局以及在本节中所讨论的文件命令的样例。

第 10 章, 产品封装。本章讨论了直到终端用户的 BSP 产品的内容和格式。

附录 A, 升级 BSP 以兼容 Tornado 2.0。讲述了如何将现有的 Tornado 1.0.1/SENS BSP 升级至 Tornado 2.0 BSP。

附录 B, 升级 BSP 以兼容 Tornado 2.2。讨论了将 BSP 从 Tornado 2.0 升级至 2.2 版本。它包含了独立和非独立结构组织的信息。

附录 C, 升级 BSP 以兼容 VxWorks 网络堆栈。描述了网络堆栈的更新过程。这仅仅适用于以前未更新的 Tornado 1.0.1 SENS 存储器产品的 BSP。

附录 D, VxWorks 引导顺序。详述了完整的 VxWorks 引导顺序。它通过其作用、名字和源地址来确认每一步。

附录 E, 组件语言。总结了在组件描述文件 (CDFs) 中使用的组件描述语言 (CDL) 的方法。它同时也给出了关于组件文件夹的缺陷和其初始组和排序的各层观点。

附录 F, 普通驱动程序。详述了关于如何在 VxWorks 系统中写入其他驱动程序。

附录 G, 升级 4.3BSD 网络驱动程序。是关于转变 4.3BSD 类型驱动程序至新网络堆栈所需要的 4.4BSD 接口的应用介绍。

附录 H, 执行基于多用户 (MUX) 的网络接口驱动程序。描述了一种新的增强型网络驱动程序 (END)。这种驱动程序使用 MUX 来达到新网络堆栈的所有特征。END 包括在基本 NETIF 驱动中不可用的登记和多点传送支持。

附录 I, 编写 SCSI-2 设备驱动程序。描述了 VxWorks SCSI-2 设备驱动程序如何适用于 VxWorks I/O 系统, 以及如何在 VxWorks SCSI-2 库 (scsi2Lib) 界面接口中写入 SCSI-2 驱动程序, 这种库提供了高层次 SCSI-2 独立装置的接口程序。

附录 J, BSP VTS 参考条目。提供了使用 BSP-VTS 的详细文件。

附录 K, BSP 合法性检查表。提供了 Wind River 公司提供的 BSP 合法性检查列表的副本。

附录 L, Refgen。提供了使用 refgen 工具来从传统 BSP 源代码中产生 HTML 所涉及的页面。

附录 M, BSP 产品目录。列出了 BSP 产品的典型目录。

(2) 其他文件

以下的文件提供了关于 VxWorks 和 Tornado 的重要背景信息。

- 《Tornado 入门指导,2.2》 (Tornado Getting Started Guide,2.2) ;
- 《Tornado 发行手册,2.2》 (Tornado Release Notes,2.2);
- 《Tornado 用户指南,2.2》 (Tornado User's Guide,2.2);
- 《VxWorks 程序员指南,5.5 》 (VxWorks Programmer's Guide,5.5);
- 《VxWorks API 参考: 驱动程序,5.5 》 (VxWorks API Reference:Driver,5.5);
- 《VxWorks API 参考: OS 库,5.5 》 (VxWorks API Reference:OS Libraries,5.5);
- 《Wind River 技术手册 》 (Wind River Technical Notes)。

5. 文档规范

本章其他的部分描述了这些文档规范的交叉引用、路径名及类型。

(1) 交叉引用

在本书中对于工具和模块交叉引用的参考条目是指涉及到了 VxWorks API Reference (目标库或者子程序) 的一个入口或《Tornado 用户指南》(主机工具) 中的和 Tornado man 目录下特定 BSP 的参考附录。在《Tornado 联机手册》(Tornado Online Manuals) 中也提供了这个参考。如果想得到关于如何取得联机文档的更多信息, 请查阅《Tornado 用户指南》中的文档指南 (Tornado User's Guide: Documentation Guide)。

书中的其他参考, 通常是以章节为层次的, 一般采取这样的形式: 书名: 章节名。

(2) 路径名

最高层次的 Tornado 目录结构包含三个主要目录 (参阅《Tornado 用户指南》: 目录和文件)。因为所有 VxWorks 文件都归于 target 目录, 本书在这个初始目录下使用相关路径名。例如, 如果想在 /usr/wind 下安装 Tornado, 文件 config/all/configALL.h 的完整路径名将显示为 /usr/wind/target/config/all/configAll.h。



注意: 在本手册中, 使用 UNIX 和 Windows 文件名中的前置分隔线来分隔路径名。

(3) 格式规则

本书使用各种规则 (样例在表 1-1) 来区别不同的组件。圆括弧通常用来指示子程序名, 就像 printf()。

表 1-1 术语的使用

术 语	样 例
文件，路径名	/etc/hosts
库，驱动	memLib,nfsDrv
主机工具	more,chkdsk
子程序	Sem Take()
引导命令	P
代码显示	main ();
键盘输入	make CPU=MC68040...
显示输出	value = 0
用户提供参数	name
常数	INCLUDE_NFS
C 语言关键字，C++ 指令	#define
按键	RETURN
控制字	CTRL+C
首字母缩写（小写）	fd

第2章 BSP 结 构

2.1 简 介

本章讲述了 BSP 的各组件，并列述了所有与 BSP 有关的源码和所包含的文件的内容。对于.h 文件，本章讲述了定义的和未定义的标准符号常数。对于.c 文件，本章讲述了所有与文件相关的必须的和任选的函数。本章还介绍了派生文件，比如 sysLib.o（与目标板相关和与目标板无关的程序代码之间的接口）和 bootrom。

本章有一节列举了 BSP 中一些主要的函数和它们被调用的顺序（对于表格形式的启动顺序，请参阅附录 D）。在调试 VxWorks 启动阶段，这些信息为我们提供了一些注意事项。

在本章的结尾讨论了 BSP 开发新手通常所犯的错误。

2.2 BSP 组件

BSP 中的应用程序为 VxWorks 操作系统与硬件环境之间提供了一个接口。图 2-1 说明了 VxWorks 的各组件，简要的说明了独立于和非独立于硬件的部分。

常规的 BSP 包括大量的用户必须新建（或修改，如果以 BSP 模板为起始）的 C 语言和汇编语言文件。本节的其他部分概括了 BSP 文件和目录。

2.2.1 BSP 中所包括的文件

本节讲述了 target/config/all 和 target/config/bspname 目录。严格地说，target/config/all 中的文件并不是 BSP 的一部分，但是所有 BSP 共享在这个目录下定义的模块。如果想了解用户特定的 BSP，通晓这些模块是很重要的。当构建工程时，target/config/comps/src 中的组件配置文件取代了 target/config/all 中的文件。该组件配置文件可为任意的 C 语言源代码，它是由工程工具所编译的，作为工程构建步骤的一部分。这些文件只提供了最终工程构造的一小部分。

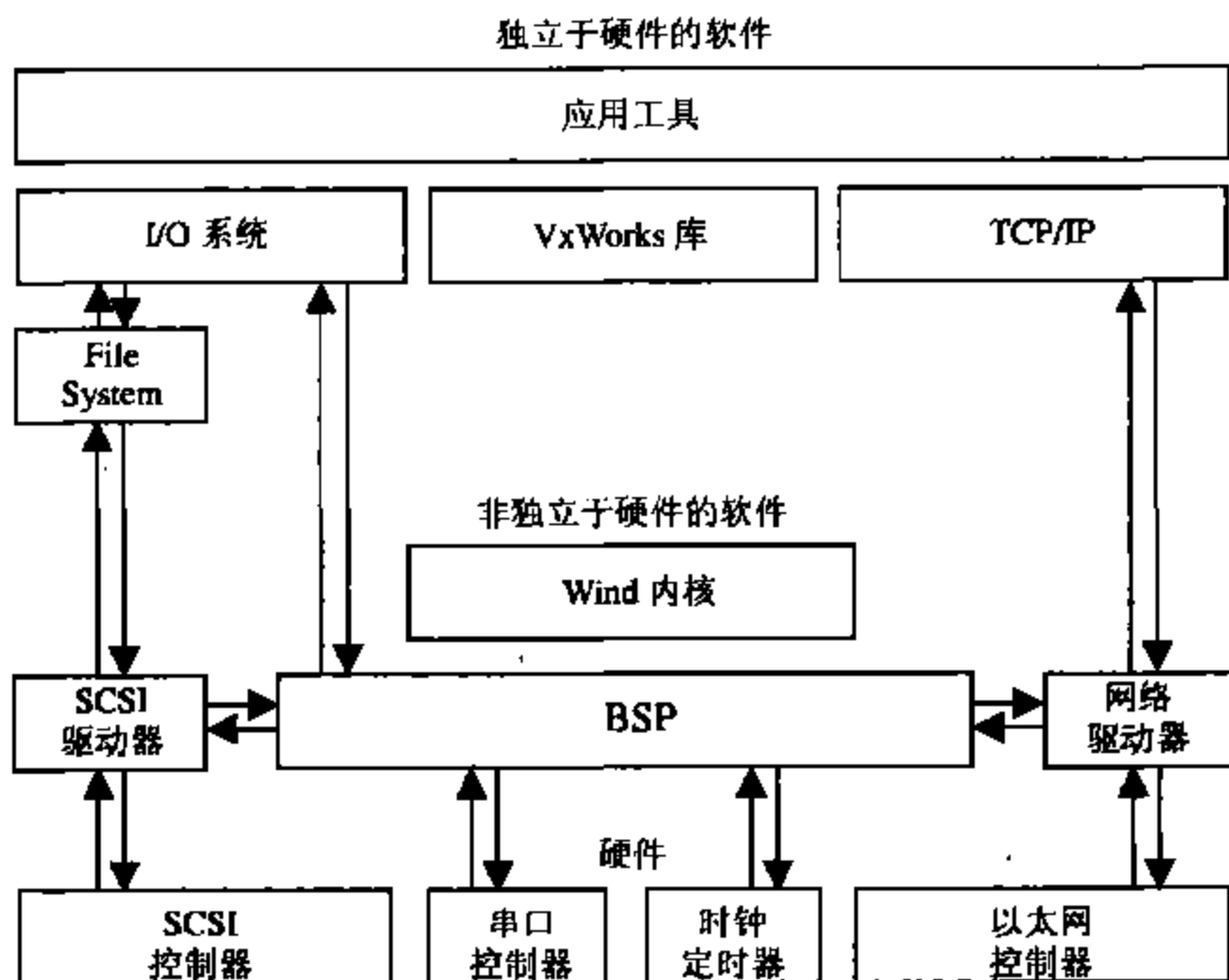


图 2-1 VxWorks 组件

1. target/config/all 目录下的文件

target/config/all 目录下的文件被作为 VxWorks 结构产品的部分而给出。一般情况下不要修改这些文件。特别是 configAll.h 文件。本文件为所有的 VxWorks 映像设置了默认的配置。在此定义的值应该是通用的，以便对于其他 VxWorks 开发者也是有效的。如果需要一个与模板构建不同的 VxWorks 映像，使用 BSP 的 config.h 文件来取代 configAll.h 的默认值。

在创建工程时并不直接使用 config/all 中的文件。当需创建一个新的工程时这些文件仅被浏览一次。当一个工程被创造以后，对于 target/config/all 中文件的任何改变对此工程都没有作用。如果要求修改，使用者必须在工程区域中直接做出改变。



注意：在命令行中修改 configAll.h 文件会对所在位置下的其他 VxWorks 使用者产生不良影响。这个注释仅对命令行下的构建方式起作用。Tornado 工具的一个好处就是修改后并不影响已有的工程。仅仅对在修改 configAll.h 后在 BSP 下创造的新工程产生影响。

■ bootConfig.c——引导 ROM 映像的主初始化文件

bootConfig.c 文件是对所有引导 ROM 映像的主初始化和控制文件。它包括完整的引导 ROM 工作。它包含了一个网络装置初始化的表格。此模块是 nsrConfig.c 引导 ROM 映像的

子集,而这个映像并没有提供在完整的 VxWorks 映像下的所有可选特性。需特别指出的是引导映像并不使用 MMU 库 (SPARC 除外)。

Tornado 工具并没有能力创建和操作引导 ROM 工程。对于 BSP 下创建的工程,在 Build 菜单下有一个按钮是运行创建引导 ROM 映像的传统命令行。当构建引导 ROM 映像时请注意特定工程构建的信息没有任何作用。只有普通的 config.h、configAll.h、bootConfig.c 文件才对引导 ROM 映像的构建起作用。

■ bootInit.c——ROM 初始化的第二步

bootInit.c 文件是引导 ROM 初始化 (romsInit.o 以后) 的第二步。在 romInit.s 文件中的 romInit() 程序跳到在此文件下定义的 romStart() 程序后终止。romStart 程序运行了对于 ROM 映像所需的解压缩和重定位工作。首先它将文本段和数据段从 ROM 中复制到 RAM 中。然后它清除了没有使用的部分主 RAM。最终它解压缩了那些已压缩的映像部分。不同的选项设置可以修改这些操作的运行方式。

在 Tornado 2.x 下构建一个工程时,romstart.c 配置文件取代了 bootInit.c 文件的功能(仅适用于 VxWorks 存储器映像,并不适用于引导 ROM 映像)。

■ dataSegPad.s——VxVMI 文本段的保护

dataSegPad.s 文件仅用作 VxVMI 文本段保护。它确保文本段和数据段不共用一个 MMU 页面。

■ usrConfig.c——VxWorks 映像的初始化编码

userConfig.c 文件包含适用于传统 VxWorks 映像的主初始化编码。与完全自包含的 bootConfig.c 不同,usrConfig.c 文件包括 target/src/config/usrExtra.c,它包含了供子系统初始化和配置的其他文件。

此文件仅适用于在 Tornado 1.0 下构建命令行。当使用此装置构建工程时,工程的初始化决定了初始化组件的顺序。每一组件都有特定的设置来规定其属于哪一个初始化组和那个初始化组属于哪部分。

2. Target/config/comps/VxWorks 目录下的文件

此目录下的文件是 VxWorks 实时内核的基本组件描述文件 (CDF 文件)。在第 6 章中有关于组件如何创建、使用和发布的更多信息。在附录 E.1 中有关于 CDL 文件语法的详述。

3. Target/config/comps/src 目录下的文件

此目录下的文件代表了与那内核组件相关的配置文件。它代表了以前在 target/config/all/usrConfig.c 和 target/src/config 目录下的编码片断。

4. Target/config/bspname 目录下的文件

Target/config/bspname 子目录包含了 BSP 的非独立于系统和硬件的文件。

5. README

这个 README 文件包含了 BSP 版本记录。它记录了每一个版本的形式,修订和文件

作为一个整体在 BSP 下工作的情况。此文件在 Tornado 1.0 中首先有所介绍。

6. Makefile 和 depend.bspname

Makefile 文件通过命令行方式对创建映像文件进行控制。从 VxWorks 5.2 开始，标准 make 工具是 GNU make，当前的编译技术继续使用了许多 GNU make 的高级特性。BSP 的 make 系统包括在 target/h/make 目录下的 make 子文件。如果一个依赖文件不存在，make 将会自动产生一个 depend.bspname 文件来跟踪所有模块的依赖性。Depend.bspname 依赖文件并不是 BSP 自带的文件，而是随后产生的文件。如需了解更多 makefile 的信息，请查阅《Tornado 用户指南 2.2》。

在此 makefile 中，必须定义以下宏模块：

CPU	目标板 CPU 结构（比如，MC68040）
TOOL	主机工具链（比如，gnu）
TARGET_DIR	目标板目录（BSP）名
VENDOR	目标制造者名
BOARD	目标名
ROM_TEXT_ADRS	规定十六进制下的引导 ROM 入口地址。对于大部分的目标板，它被规定在 ROM 地址的起始区域。但是，如果硬件配置使用 ROM 的起始区域作为复位矢量，应该给出相应的偏移量
ROM_WARM_ADRS	引导 ROM 代码重启时的入口点
ROM_SIZE	十六进制 ROM 区域的大小
RAM_LOW_ADRS	加载 VxWorks 的地址
RAM_HIGH_ADRS	复制引导 ROM 映像到 RAM 所使用的目标地址

7. Config.h

config.h 文件包含特定 CPU 板的所有包含文件和定义。Config.h 的标准组织内容大致如下：

- BSP 版本和版本 ID 号码；
- configAll.h(#included)；
- 缓冲存储器和 MMU 配置；
- 定义共享网络存储器；
- 板载存储器地址和大小；
- ROM 地址和大小；
- 非易失性存储器（NVRAM）参数；
- 默认引导行参数；
- 时标驱动支持；
- 外部总线地址映射；

- 网络设备和中断向量;
- bspname.h 文件 (#included)。

 **注意:** config.h 文件应包括 configAll.h 和 bspname.h

(1) BSP 发行号。

BSP 发行号包括其版本号码和修订本号码。BSP 版本号码指明了 BSP 的发展。BSP 的修订本号码是一个附加号码, 这个号码表明了 BSP 的发行时间。

BSP 1.0 版本是为一直到并包括 VxWorks 5.2 在内的所有 VxWorks 版本标准所作的。而 BSP1.1 版是为 Tornado 1.0 和 1.0.1 版所作的。而 BSP1.2 版则是专门为 Tornado 2.x 所设计的。BSP1.1 和 1.2 的主要接口区别在于是否支持网络存储器和不同的存储方式。随着每一个新的版本顺序, 修订版号码从 0 开始, 并且为支持产品的 BSP 每次升级一个版本。

可以定义一串字符“1.2”或者“1.1”为 BSP_VERSION 来规定一个 BSP 版本号。要不然可以定义宏模块 BSP_VER_1_1 或者 BSP_VER_1_2 中所适用的任意一个。为了设置修订版号码, 可以使用 BSP_REV, 应该设定序列值比如为“/0”。修订版号码包含了斜字符。完整的 BSP 版本号码是 BSP_VER 和 BSP_REV 两个宏模块序列的并置, 例如: 1.2/0。

因此, 如要设置 BSP 的版本号码为 1.2/0, 需要如下的定义:

```
#define BSP_VER_1_1
#define BSP_VER_1_2
#define BSP_VERSION "1.2"
#define BSP_REV "/0" /* Increment for each new release */
```

(2) configAll.h。

在 config.h 文件中应该包括 configAll.h 文件 (在 target/config/all 目录下), 这样就可采用合理的 BSP 默认配置。使用 config.h 作为你的 BSP 的用户可编辑的设置文件。

使用 config.h 作为 BSP 的主用户编辑设置文件。为了在你的地址下为 BSP 设置合理的默认设置, 你的 config.h 文件应该包括 configAll.h (在 target/config/all 目录下)。

在 config.h 文件中, 用户可以覆盖这个默认值, 而采用更适合于此特殊 BSP 的值。同时用户也应该将 config.h 文件作为后续用户应该编辑来覆盖 configAll.h 或者设置其已创建好的 BSP 的任意特性的设置文件。

当创建一个工程的映像时, 使用这个工程所选用的组件和参数会覆盖在 configAll、config.h 甚至 bspname.h 文件中所设置的值。工程工具中被设置值的宏模块会覆盖任意其他设置值。

 **注意:** 一个 config.h 文件在 BSP_VERSION 和 BSP_REV 被定义以前, 不应该包括 configAll.h, 如果他们未被定义, configAll.h 赋予默认值。

如果在 `config.h` 中包含特定 BSP 的设置选项，尽可能把这些选项做成用户友好的。理论上，用户应该仅通过其名字直接理解选项的使用方法。另外，不要要求用户通过计算来设置选项。例如：用户应该使编译器根据是否定义了一个可以理解的名字的常数来计算出的寄存器值。

(3) 存储器和 MMU 配置。

默认的存储器和 MMU 配置是与 CPU 结构有关的。他们通常设置为能使目标板发挥最高的性能。根据结构，存储器和 MMU 可以完全独立或者高度结合。通常的，当 MMU 被激活时，我们可以认为对存储器的控制是采取基于页的形式。这意味着 `USER_I_CACHE_MODE` 和 `USER_D_CACHE_MODE` 宏模块在 MMU 被激活时通常并不影响缓存的运行。如需得到关于存储器和 MMU 之间交互作用的详细信息，请查阅相关的 VxWorks 结构附录文件。

大部分 BSPs 取消在 `configAll.h` 中对存储器和 MMU 的默认设置，用 `config.h` 对他们重新定义，这样是为了允许用户更容易的做出其自己的选择。这样做就使得当用户对设置进行修改时，就需要十分注意。当然，如果存储器状态并不是用户可改变的选项，就应该在 `bspname.h` 而非 `config.h` 中定义这些宏模块。

(4) 共享存储器网络定义。

这部分描述了共享存储器网络驱动所需设置的参数（如需更多的信息，请查阅《VxWorks 网络程序员指南》：数据链路层网络组件）。共享存储器网络使得系统中的多个 CPU 可以通过共享存储器进行通信。以下的常数对于共享存储器网络初始化是十分重要的：

■ SM_ANCHOR_ADRS

底板定位地址 (anchor address)。此 anchor 包含一个指向实际共享内存池的指针。如果板载存储器是双重端口，则其所分配的存储器的低位地址+0x600 就是 Wind River 使用的默认地址。如果使用一个存储器板，则使用的是此存储器的基址，共享存储器的中断类型见表 2-1。

■ SM_MEN_ADRS

共享存储器的位置。定义为 NONE 意味着存储器将使用 `malloc()` 动态地被定位到主控的板载双端存储器。

■ SM_MEM_SIZE

共享内存池的大小。

■ SM_INT_TYPE

在底板上与其他 CPU 板的通信方式。

表 2-1 共享存储器中断类型

SM_INT_TYPE	SM_INT_ARG1	SM_INT_ARG2	SM_INT_ARG3	Meaning
SM_INT_NONE	—	—	—	Polling
SM_INT_BUS	level	vector	—	Bus interrupt

续表

SM_INT_TYPE	SM_INT_ARG1	SM_INT_ARG2	SM_INT_ARG3	Meaning
SM_INT_MAILBOX_1	address space	address	value	1-byte write mailbox
SM_INT_MAILBOX_2	address space	address	value	2-byte write mailbox
SM_INT_MAILBOX_4	address space	address	value	4-byte write mailbox
SM_INT_MAILBOX_R1	address space	address	—	1-byte read mailbox
SM_INT_MAILBOX_R2	address space	address	—	2-byte read mailbox
SM_INT_MAILBOX_R4	address space	address	—	4-byte read mailbox

如需关于 Tornado 2.2 共享存储器子系统修改的信息, 请查阅 Tornado Migration Guide。

(5) RAM 地址和大小。

■ LOCAL_MEN_LOCAL_ADRS

板载存储器区的起始地址。

■ LOCAL_MEN_SIZE

固定的(静态的)存储器大小。

■ LOCAL_MEN_AUTOSIZE

运行时的(动态的)存储器大小。更多信息, 请查阅 Wind River Technical Note#42。

■ USER_RESERVED_MEN

保留存储器的大小(字节)。更多信息, 请查阅 Wind River Technical Note#41。

■ RAM_HIGH_ADRS

复制引导 ROM 映像所使用的目标地址。它通常是板载存储器起始地址加上 0x00100000。它必须和在 makefile 文件中所定义的值是一样的(请查阅 Makefile and depend.bspname)。

(6) ROM 地址和大小。

■ ROM_BASE_ADRS

ROM 十六进制起始地址。

■ ROM_TEXT_ADRS

十六进制的引导 ROM 入口地址。对于大部分目标板, 它被设置为 ROM 地址的开端。但是, 有些目标板将 ROM 地址开端的部分区域用于存放复位向量。它必须和 makefile 中所定义的值是相同的(请查阅: Makefile and depend.bspname)。

■ ROM_LINK_ADRS

如果使用它, 则这个宏模块以十六进制的形式指明了引导 ROM 的连接地址。对于大部分目标板, 这一项被设置为 ROM 地址的起点。如果这个地址已经给出, 则连接器用这个地址连接引导 ROM 映像。否则, ROM_TEXT_ADRS 则是作为连接地址来使用。

■ ROM_WARM_ADRS

热启动引导入口地址。这通常定义为 ROM_TEXT_ADRS 加上一个偏移量。

■ ROM_SIZE

十六进制形式的 ROM 的大小。这必须和 makefile 中的值相同，请查阅 Makefile and depend.bspname。

(7) 非易失性存储器 NVRAM。

sysNvRamSet()和 sysNvRamGet()程序有一个偏移量参数。如果 NV_BOOT_OFFSET 的值远大于零，你可以通过指定一个负偏移量来访问引导行之前的字节内容。要想覆盖 NV_BOOT_OFFSET 的默认值，首先删除 config.h 文件中该宏的定义，然后再重新定义。

对于没有 NVRAM 的目标板，需要包含有 men/nullNvRam.c 文件，这样在调用 NVRAM 程序时就会返回 ERROR。在 BSP 工具包光盘的 target/src/drv/men 目录中提供了此文件。

有一种文件名为 templateNvRam.c 的通用 NVRAM 驱动程序。请参阅此文件中介绍的在使用此驱动程序时的一些条件以及所需的宏模块。TemplateNvRam.c 文件对所有按位访问（区别于按块访问）的 NVRAM 目标板都是适用的。

■ NV_RAM_SIZE

可以使用的 NVRAM 总字节数。在 config.h 文件中定义了 NV_RAM_SIZE。对于没有 NVRAM 的目标板，设置 NV_RAM_SIZE 为 NONE。

■ BOOT_LINE_SIZE

保留给 VxWorks 引导行的 NVRAM 的字节数。默认值是 255，它是在 config.h 文件中定义的。BOOT_LINE_SIZE 必须小于或等于 NV_RAM_SIZE。若不想使用 BOOT_LINE_SIZE 的默认值，可以先在 config.h 中删除对这个的宏定义，然后再重新进行定义。

■ NV_BOOT_OFFSET

NVRAM 中 VxWorks 起始引导行的字节偏移量。默认值是 0，并在 config.h 中定义。

(8) 默认引导行参数。

DEFAULT_BOOT_LINE 宏模块提供了一个引导行例子，终端用户能够做出一些改变来反映他们的目标板环境的变化。这些对于没有 NVRAM 的目标板是十分有用的，因为它意味着终端用户没有必要在每次重新启动后重新输入引导参数。

(9) 时标支持。

每一个 BSP 都应该对 WindView 产品提供时标支持。用户通过定义或者取消定义 INCLUDE_TIMESTAMP 宏来配置时标支持。可以参阅定时器驱动程序源文件中的有关内容，其中描述了特定驱动程序所需要的一些其他的支持。

(10) 外部总线地址映射

VME 总线映射是由一系列的九个宏模块控制进行主控映射窗口，并由另外的一组九个宏模块来控制进行受控窗口映射。用户通过设置 config.h 中的 INCLUDE_VME 来配置 VME。对 A16,A24,A32 主访问空间的 VME 映射，每一个是由三个宏依次来控制的。

假定 BSP 使用了所有这些宏模块。如果硬件是已经准备好的并且这些宏模块中一组或多组必须有固定值，那么那些宏模块就必须从 config.h 移动到 hspname.h 文件中。只有用户能够合理改变的宏模块可以放在 config.h 中。

使用下列宏模块来定义从本地 CPU（总线主控访问）到 VME 总线的映射窗口：

VME_A16_MSTR_SIZE	A16 存储空间视窗的大小
VME_A16_MSTR_BUS	A16 视窗的总线首地址
VME_A16_MSTR_LOCAL	A16 视窗的本地地址
VME_A24_MSTR_SIZE	A24 存储空间视窗的大小
VME_A24_MSTR_BUS	A24 视窗的总线首地址
VME_A24_MSTR_LOCAL	A24 视窗的本地地址
VME_A32_MSTR_SIZE	A32 存储空间视窗的大小
VME_A32_MSTR_BUS	A32 视窗的首总线地址
VME_A32_MSTR_LOCAL	A32 视窗的本地地址

使用下列宏模块来定义从 VME 总线到本地存储器的视窗。默认情况下，只有 CPU 0 将其存储器内容输出到外部总线。这些宏模块控制总线受控窗口，使得本地存储对于 VME 总线是可见的。由于其容量很小，主存储器很少被映射到 A16 受控窗口中。将视窗的大小设定为 0 就可以禁用这个视窗。

VME_A16_SLV_SIZE	A16 存储空间视窗的大小
VME_A16_SLV_BUS	A16 视窗的总线地址
VME_A16_SLV_LOCAL	A16 视窗的本地地址寻址
VME_A24_SLV_SIZE	A24 存储空间视窗下的大小
VME_A24_SLV_BUS	A24 视窗的总线地址
VME_A24_SLV_LOCAL	A24 视窗的本地地址寻址
VME_A32_SLV_SIZE	A32 存储空间视窗的大小
VME_A32_SLV_BUS	A32 视窗的总线地址
VME_A32_SLV_LOCAL	A32 视窗的本地地址寻址

目前还没有建立起一组标准的描述 PCI 地址映射的宏。可以通过联系 Wind River 公司来取得最新的 PCI 接口标准。如需更多信息，请查阅 Wind River Technical Note #44 和 `target/src/drv/vme/templateVme.c`。

（11）PCI 地址宏

对于 PCI，应该描述至少一个或者通常四个从本地总线存储空间到 PCI 存储空间的视窗。它们被称为主窗口，因为本地 CPU 是用于处理这些数据交换的总线控制器。

PCI_MSTR_MEM_SIZE	PCI 存储空间视窗的大小
PCI_MSTR_MEM_BUS	PCI 存储视窗的总线地址
PCI_MSTR_MEM_LOCAL	PCI 存储视窗的本地地址寻址
PCI_MSTR_IO_SIZE	PCI I/O 存储空间视窗的大小
PCI_MSTR_IO_BUS	PCI I/O 视窗的总线地址
PCI_MSTR_IO_LOCAL	PCI I/O 视窗的本地地址寻址
PCI_MSTR_MEMIO_SIZE	PCI 存储（非预取）视窗的大小
PCI_MSTR_MEMIO_BUS	PCI 非预取存储视窗的总线地址

PCI_MSTR_MEMIO_LOCAL 非预取视窗的本地地址寻址

并非在所有的情况下 PCI 总线空间都是可以实现的。注意有不同的视窗映射了主存储器。默认的视窗是用于可预取的存取操作的。第二视窗 PCI_MSTR_MEMIO 是用于执行非预取操作的。这些视窗中的一个或者两个通常是映射到主存储器的。使一个窗口处于非活动状态的标准信号是设置大小为 0，这样可以禁止从本地存储器访问那个存储空间。

如需关于 PCI 宏模块的更多信息，请查阅 Wind River Technical Note #49。

(12) 网络设备和中断向量

如果此 BSP 使用外部板卡作为网络设备，那么对总线地址和中断向量的指派可以在 config.h 中由用户来改变。

8. bspname.h

此文件通常包含在所有的选项之后。这样就允许在 bspname.h 中插入测试来检查适当的选项联接。举个例子，有一个目标板有两个不同的以太网供选工程，但一次只能选择其中的一个。在此情况下，如果可能的话 bspname.h 文件应检查是否对两者都进行了定义并相应的产生错误信息。

另一个例子是添加额外的特性，这些都是为了支持用户所选择的功能所必须的。例如，用户可能选择支持 NFS，这时 bspname.h 文件应该添加所需的 RPC 和网络支持来支持 NFS。这正好是 usrExtra.c 为 usrConfig.c 所做的。当然，这仅仅是为说明此原理的一个假设例子，而非实际例子。

9. romInit.s

这个文件包含用于初始化的汇编语言源文件代码，这些都是 VxWorks 的引导 ROM 入口和 VxWorks 基于 ROM 版本的入口。

入口点 romInit() 是开机后执行的第一段程序。它设置 BOOT_COLD 参数来将其传给普通 romStart() 程序。如果硬件需要直接存储器映射或者设置特殊的寄存器，在这里进行处理。大部分硬件初始化使用 sysLib.c 中的 sysHwInit() 程序。

romInit() 必须完成以下 3 个主要的功能：

- 禁止中断并设置 CPU。
- 设置存储器系统。通常是禁用高速缓冲存储器并根据需要设置存储控制器。对于 SPARC 结构，MMU 必须是激活的。
- 设置堆栈区指针和其他寄存器来开始执行 C 语言程序。计算 romStart() 程序的地址并访问此程序，而 romStart() 没有返回值。

要注意 romInit() 必须按照 PIC 来编码。这是为支持 VxWorks 的复杂引导策略所必须的，如果地址不是与 PC 相联系的，就必须通过先减去 romInit 符号再加上 ROM_TEXT_ADRS 值来重新计算。这通常是由 ROM_ADRS 宏来完成的。

```
#define ROM_ADRS(X) ((X) - __romInit + ROM_TEXT_ADRS)
```

对于 BSP 编写者来说，企图在 romInit() 中进行太多的初始化是一个常见的错误。大多

数其他初始化功能。使用 `sysHwInit()` 程序来完成。另外, `romInit.s` 的编码不应调用其他的模块和程序。这引起了压缩引导 ROM 映像相关的问题并否定了压缩的任何好处。

从 `romInit.s` 调出 C 程序是可能的, 但却是不方便的。某些 C 语言使用寄存器地址而非 PC 相关编码 (当然就不是 PIC)。如果需要 C 程序, 它不应该成为 `sysLib.c` 的一部分。这种程序应该被放在一个单独的模块中, 并通过将该模块添加到 `makefile` 文件的变量 `BOOT_EXTRA` 和程序包含列表中。

另一个 BSP 编写者常犯的错误是假定由 `romInit.s` 初始化的设备将被由 `sysALib` 或 `sysLib.c` 重新初始化。VxWorks 映像不假定是由 VxWorks boot ROM 程序来引导的, 它必须重新设置和初始化所有设备和特性。如果没有这么做, 将导致只依赖于 boot ROM 映像程序的 VxWorks 映像才能正常工作。以这种方式来连接 boot ROM 和 VxWorks 映像是不可取的。

10. sysALib.s

此文件包含特定目标板的汇编程序, 是与系统相关的程序。所有 BSP 程序都用 C 语言来编写。但是, 如果有一些技术原因需要使用汇编程序, 可以将程序放在 `sysALib.s` 中。

VxWorks 映像的入口点 `_sysInit` 在 `sysALib.s` 文件中。这是启动之后首先执行的程序 (查阅 `sysALib.s:sysInit()`)。除系统存储器设置外, 通常在映像中完成了 `romInit.s` 运行功能。存储器系统应该在启动阶段进行初始化。

不像 `romInit.s`, `sysALib.s` 的代码不需按照 PIC 来编写或使用 `ROM_ADRS` 来指定映射绝对地址。它可以调用其他模块或程序。

11. sysLib.c

文件包含特定目标板的 C 语言源程序, 非独立于系统。 `sysLib` 体系。这些程序提供了目标板级接口, 这样可以独立于系统的方式建立 VxWorks 及用户程序。注意 `sysLib.c` 不包括 `mdrv` 目录的驱动。这些驱动可包含在下面列出的程序中。如需更多信息, 请查阅第 5 章和附录 F。

驱动器设置及初始化通常包含 `sysLib.c` 的子文件中。普通子文件被命名为: `sysSerial.c`, `sysCsi.c`, `sysNet.c` 等。这些子文件是为了使对专用设备的设置/初始化。所有体系结构中, 结果是不变的。对于利用专用设备的目标板的设置文件, 应该与任何其他目标板上的该设备的目标板是一致的。

BSP 抽象并不完全支持特定目标板的特性。有些目标板可以通过跳线或 PAL 来代替软件可控的寄存器的寄存器。其他硬件配置可能不支持正常情况下的基本特性。

当删掉某些特性时, 编写一个空程序段来返回 `ERROR` 或 `NULL` 通常是安全的。

12. sysSerial.c

此可选文件包含所有 SIO 设备驱动程序 (串行 I/O) 的设置和初始化。它并不单独编译, 它是包含在 `sysLib.c` 中的。从 `sysLib.c` 中调出串行设备初始化的目的是使编译模块和

许重复使用 BSP 程序。

13. sysScsi.c

此可选文件包含 BSP 的 SCSI-2 设置和初始化。就像 sysSerial.c 一样，它是包含在 sysLib.c 中的文件。

14. sysNet.c

可选的网络接口设备的设置和初始化。

15. bspname.h

使用 bspname.h 来设置所有非可选的与特定目标板相关的信息，包括串行接口、计时器、I/O 设备的定义。可以通过访问 target/h/drv 目录了解许多信息。在目标板中安装 VxWorks 所需要的驱动程序的头文件应包括在 bspname.h 的起始部分。

此文件包含不受用户影响的固定信息。如果 bspname.h 中定义的宏或值能够改变以使系统用户化，则可以在 config.h 中定义这些宏或值。

使用一个头文件样本是非常有帮助的，因为大部分常量名、设备基地址等已经定义了，仅需要重新进行优化。下列内容应该定义在 bspname.h 中：

- 中断向量/优先级；
- I/O 设备地址；
- 设备寄存器位的意义；
- 系统和辅助时钟参数（最高和最低频率）。

同时，使用一个描述目标板的宏前缀（例如：Heurikon 可被定义为 HK_）来指明特定目标板的值。

(1) 中断向量和优先级。定义所有的中断向量和优先级，这些是由硬件控制的，用户是不可以在 bspname.h 文件中改变的。

(2) I/O 设备地址。定义所有由硬件固定的 I/O 地址，用户是不能在 bspname.h 文件中改变的。

(3) 设备寄存器位的意义。对于板载控制寄存器，为每一个寄存器中每一位或几个位定义一个宏值。典型的，如果没有更好的位置，这种宏定义一般放在 bspname.h 中。

(4) 系统和辅助时钟。当给定一个新的时钟速率，计时器驱动应使用这些宏来判断新的速率是否正确。

SYS_CLK_RATE_MIN	设置最小系统时钟速率
SYS_CLK_RATE_MAX	设置最大系统时钟速率
AUX_CLK_RATE_MIN	设置最小辅助时钟速率
AUX_CLK_RATE_MAX	设置最大辅助时钟速率

16. target.nr

target/config/bspname 目录下的 target.nr 文件包含了为运行 VxWorks 所需的特定目标板

的信息。为了方便地开始编写一个新的 BSP，BSP 安装工具包光盘中包含一些特殊结构模板，比如 `target/config/template68k/target.nr`。

对于 Tornado 2.x，`target.nr` 文件并不像 UNIX 类型关联页面或者 Windows 兼容文本页面那样来处理。它是以 HTML 文件形式传送到用户的。请参阅 L.Refsen 获取 `refgen` 文件有用的详细信息。

参考信息被划分成以下几部分：

NAME	目标板的名称
INTRODUCTION	目标板默认状态的概括。关于引导 ROM 和所需跳线设置的介绍
FEATURES	所有的目标板支持或不支持的特性的概括
HARDWARE DETAILS	可用装置的概括，例如：串行，以太网和 SCSI
SPECIAL CONSIDERATION	特别的限制或特性
BOARD LAYOUT	介绍目标板布局，对说明 boot ROM 和跳线是很有用的。跳线和跳线状态是由表 2-2 中的内容说明的
SEE ALSO	请查阅其他 VxWorks 手册
BIBLIOGRAPHY	概括了资料和其他相关目标板和设备的技术信息

表 2-2 目标板线路的 ASCII 描述

跳线类型	状态	特性
垂直跳线	Installed	x
垂直跳线	Open	:(colon)
垂直跳线 (三针)	Up	U
垂直跳线 (三针)	Down	D
水平跳线	Installed	— (dash)
水平跳线	Open	" (quote)
水平跳线 (三针)	Left	L
水平跳线 (三针)	Right	R

2.2.2 衍生文件

下列文件是从源文件，系统头文件、驱动源文件和 VxWorks 文档库的模块中衍生出来的。这些文件仅仅是为了示例而存在的。用户在配置系统时应重新建立这些文件的一部分或全部。

当在使用 Tornado 2.x 时, 这些衍生文件在工程目录下。它们在 BSP 目录下的副本只有当使用传统的命令行方法创建工程时才被使用。



注意: 对于此版本, 引导 ROM 不能使用 Tornado 工具构建。用于配置和构建引导 ROM 映像的增强工具是已经计划好的。

■ **bootInit.o、bootInit_res.o 和 bootInit_uncmp.o**

有三种可用的启动策略: 压缩 ROM, 未压缩 ROM 和常驻 ROM。所有这三种策略都可用于引导 ROM 或 VxWorks ROM。模块的名称表示其使用了哪种策略。

■ **bootrom**

此文件是一个包含二进制的 VxWorks 引导 ROM 的目标模块。

■ **bootrom.hex**

这是一个包含 VxWorks boot ROM 程序的 ASCII 文件, 适合于通过串行接口下载到 PROM 编程器。该文件的默认格式是采用 Motorola 公司的 S-Records。

■ **bootrom.Z.s 和 bootrom.Z.o**

这是用于生成 bootrom.o 压缩映像模块的源文件和目标文件。

■ **ctdt.o**

此模块起 C++构造器/析构器的作用。

■ **dataSegPad.o**

此模块确保文本和数据段不共享一个通用存储器页面。对于 VxVMI 选项, 此模块迫使将数据段分配在与文本段不同的 MMU 页面中。

■ **sysTbl.c 和 sysALib.o**

这些是嵌入的符号表的源文件和目标文件。

■ **sysLib.o 和 sysALib.o**

这些模块通常是成对出现。它们是 sysALib.s 和 sysLib.c 模块的输出。

■ **romInit.o**

此模块是所有 boot ROM 映像或 VxWorks ROM 映像的启动程序。对于 Tornado, 它与 romStart.o 模块(而非 bootInit.o)结合使用。

■ **VxWorks 和 VxWorks.sym**

VxWorks 文件是完整的, 将在目标板中运行已连接的 VxWorks 二进制代码。VxWorks.sym 文件是 VxWorks 二进制代码的符号表。两个文件都是由所提供的配置文件生成的。

■ **VxWorks.st**

此文件是完整的, 已连接的, 独立的二进制 VxWorks 文件, 并且是与符号表相关联的。

target/proj/bspname_gnu/'and target/proj/bspname_diab/'

此目录包含 BSP 工程文件和默认构建输出。有两种创造新工程的方法: 首选的方法是

使用工程工具；其次是使用命令行中的 `make prj_gnu` 或 `make prj_diah` 命令。

`doc/VxWorks/bsp/bspname/“.”`

此目录包含 HTML 的记录文件，当安装产品时此文件与用户帮助系统相结合。可以在命令行方式下用 `make man` 命令打开这个文件。

2.2.3 必须的程序

下列程序在 `sysLib.o` 模块中。如果缺少了任一程序，在连接时就可能出现无法解决的全局错误。

<code>sysBspRev()</code>	返回 BSP 版本和修订版序号
<code>sysClkConnect()</code>	连结到系统时钟中断程序
<code>sysClkDisable()</code>	关闭系统时钟中断
<code>sysClkEnable()</code>	开启系统时钟中断
<code>sysClkInt()</code>	处理系统时钟中断（内部的）
<code>sysClkRateGet()</code>	取得系统时钟频率
<code>sysClkRateSet()</code>	设置系统时钟频率
<code>sysHwInit()</code>	初始化系统硬件
<code>sysHwInit2()</code>	初始化附加的系统硬件
<code>sysMemTop()</code>	取得逻辑存储器的顶端地址
<code>sysModel()</code>	返回 CPU 目标板的模块名称
<code>sysNvRamGet()</code>	获取 NVRAM 中的内容
<code>sysNvRamSet()</code>	设置 NVRAM
<code>sysSerialHwInit()</code>	将 BSP 串行设备初始化为静态状态
<code>sysSerialHwInit2()</code>	连结 BSP 串行设备中断
<code>sysSerialChanGet()</code>	取得与串行通道关联的 SIO_CHAN 设备
<code>sysToMonitor()</code>	将控制权交给 ROM Monitor

2.2.4 可选的程序

下列的程序通常出现在所有的目标板中。那些标记为“internal”是 BSP 内部的，并不能被应用程序调用。

<code>sysAbortInt()</code>	处理 ABORT 按键中断
<code>sysAuxClkConnect()</code>	连结到辅助时钟中断程序
<code>sysAuxClkDisable()</code>	关闭辅助时钟中断
<code>sysAuxClkEnable()</code>	打开辅助时钟中断
<code>sysAuxClkInt()</code>	处理辅助时钟中断
<code>sysAuxClkRateGet()</code>	取得辅助时钟频率
<code>sysAuxClkRateSet()</code>	设置辅助时钟频率
<code>sysPhysMemTop()</code>	取得物理存储器的顶端地址

下列的程序通常只出现在有总线接口的目标板中:

<code>sysProcNumGet()</code>	取得处理器号码
<code>sysProcNumSet()</code>	设置处理器号码
<code>sysBusIntAck()</code>	确认总线中断
<code>sysBusIntGen()</code>	产生总线中断
<code>sysBusTas()</code>	测试并设置交叉总线位置
<code>sysBusTasClear()</code>	清除由 <code>sysBusTas()</code> 设置的位
<code>sysBusToLocalAdrs()</code>	转换总线地址为本地地址
<code>sysIntDisable()</code>	禁用总线中断电平
<code>sysIntEnable()</code>	启用总线中断电平
<code>sysLocalToBusAdrs()</code>	转换本地地址为总线地址
<code>sysMailboxConnect()</code>	连结邮箱中断程序
<code>sysMailboxEnable()</code>	启用邮箱中断
<code>sysMailboxInt()</code>	处理邮箱中断 (内部的)
<code>sysNanoDelay()</code>	设置校准延迟

2.3 VxWorks 引导顺序

本节讲述了典型的 VxWorks 引导工程。从根本上说,在初始化和载入 VxWorks 时所有的处理器执行相同的逻辑步骤,尽管有些可能需要一、两步额外的步骤,而其他的可跳过某些步骤。

从最低限度讲，初始化处理器包含提供一部分程序，以及安装位于存储器特殊位置中的一些表格，这通常正是在重新启动或开机时处理器跳到的地方。此程序将处理器设置为某种特殊状态，初始化存储器和存储器地址，禁止中断，并且将控制权交给附加引导程序。

本节对各种不同的配置和体系结构，一步一步地介绍了引导顺序。在某些情况下，此说明可能只提供粗略的处理器引导顺序介绍，因为可能有非常独特的特殊芯片，比如说 M68302 嵌入式控制器。

如需了解由传统命令行构建的映像的引导顺序，请查阅附录 D. VxWorks 引导序列。如需了解使用工程工具构建的映像的引导顺序，请查阅 E.3 工程初始化顺序。

2.3.1 处理器的差异

Motorola 680x0 系列在 VxWorks 引导顺序中是典型的；幸运的是，大部分处理器符合这一情况。

Intel i960 是很特殊的，它需要初始引导记录 (IBR)，它包含详尽的处理器配置表，其中定义了存储器区域，中断表信息，异常处理等。在某种程度上，这是处理器首先执行的程序，并且必须在固定的位置，例如许多其他处理器的堆栈指针和程序计数器。

摩托罗拉 M68040 必须是透明的存储器寻址方式。摩托罗拉 M683xx 系列是一特殊情况。这些处理器在无总线的目标板上运行，并通过串行线路与外部通信；因此，引导 ROM 包含嵌入式独立 VxWorks shell (VxWorks.st_rom.hex)。这种配置类型可能会在今后的无总线系统下更加流行。

2.3.2 顺序概述

处理器首先跳至 ROM 入口点，它设置了状态字并创建了虚拟栈；然后跳至一段 C 程序。虚拟栈的一个参数决定是否清理处理器（冷启动），然后将 ROM 中余下内容复制到 RAM 中（如果 ROM 中余下的内容是压缩的，那么在复制的时候要进行解压缩）。接下来，处理器跳至 RAM 入口。

bootConfig.c 的 RAM 入口程序禁用缓存器，清除 bss 段，初始化向量表，运行目标板特定初始化，然后使用户引导任务启动多任务内核。

VxWorks 引导 ROM 映像是一个独立的应用。开发者使用其来通过网络引导新的 VxWorks 映像并连接到应用程序代码。引导 ROM 的核心被丢弃了：这里没有软中断机制也没有代码的重用。应用程序与在主机中构建的新的映像相连接。

2.3.3 文件和程序

本节包含了关于系统如何启动的更加详细讲述，由文件和程序名组成。为了更清楚，引导顺序被分为许多主要步骤或功能调用。关键的作用作为标题并按照调用命令的顺序列出来。

1. romInit.s: romInit()

冷启动时，处理器从 romInit() 开始执行，通常是 romInit.s 中的第一段程序。对于热启动，处理器对 romInit() 加上小的偏移量（查阅 sysLib.c 中的 sysToMonitor()）开始执行。romInit() 程序禁止中断，将引导类型（冷/热）入栈，清空缓存器，并转移到 bootInit.c 中 romStart()。堆栈被定位在文本段之前，并向相反方向增长。在 romInit() 程序中开始执行 C 程序之前必须尽可能少的安装设备。硬件初始化是 sysLib.c 文件中的 sysHwInit() 程序的任务。

2. bootInit.c:romtart()

文本和数据段从 ROM 复制到 RAM 中是通过如下方法：

- 如果文本段不是常驻 ROM 的，则文本和数据段将被复制。
- 如果文本段是常驻 ROM 的，只有数据段将被复制。
- 未使用的存储器将被清空来对其初始化。
- 如果需要，可以解压缩。

非典型的，i960 调用 sysInit(bootType)，和其他所有结构一样，其最终将调用 usrInit(bootType)。

3. sysALib.s:sysInit()

VxWorks 在正常重启下调用 sysInit() 来避免程序的冗余，i960 在 ROM 初始化时也调用该程序。SysInit() 程序通常必须是 sysALib.s 中首先定义的程序，它禁用缓存器，使用默认程序段初始化系统中断表、初始化系统故障表，并初始化所有的处理寄存器为已知默认值。它启动跟踪功能，清除所有的等待中断并最终调用 usrInit(bootType)。

此程序必须重复由 romInit() 已完成的硬件初始化。如果不这样，引导 ROM 程序设置的特性可能转移到 VxWorks 映像。这是不合理的，因为引导 ROM 的设置和初始化作用对运行 VxWorks 映像有影响。这有可能会在配置改变时用户需要重新建造引导 ROM。这也可能会导致用户不得不使用成对的引导 ROM 和 VxWorks 映像套件。

4. usrConfig.c 和 bootConfig.c:usrInit()

usrInit() 程序（在 usrConfig.c 中）保存了关于引导类型的信息，处理了所有在核心启动之前就必须完成的初始化工作，然后启动核心执行。这是 VxWorks 下首先运行的 C 程序。

这是在切断所有的硬件中断的管理模式下调用的。

许多 VxWorks 功能在此程序下不能调用。因为到目前为止还没有任务环境（没有 TCB 和任务堆栈），需要任务环境的功能不能被调用。这包括任何可能使主叫用户优先的功能，比如信号量，或者任何使用这种功能的程序，比如 `printf()`。`UsrInit()` 程序仅完成创建一个初始任务 `usrRoot()` 所需要的工作。然后此任务完成启动。

`usrInit()` 下的初始化包括如下内容：

- 高速缓存初始化。在 `usrInit()` 的开始初始化高速缓存起始的程序，设置了高速缓存的模式并将其设置为安全状态。在 `usrInit()` 的末端默认将指令缓存和数据缓存启用。
- 系统 bss 段清零。C 和 C++ 语言规定所有未初始化的变量初始值必须是 0。这些未初始化的变量被放在 bss 段。此段在引导指令下并没有被载入，因为它被认为全部是零。由于 `usrInit()` 是首先执行的 C 程序，它消除了包括 bss 在内的存储器部分，作为它最先的行为。当 VxWorks 引导 ROM 清除了所有的存储器，VxWorks 并不认为使用了引导 ROM。
- 初始化中断向量。在启动中断和内核之前必须设置异常向量。首先，调用 `intVecBaseSet()` 建立向量表基地址。在 `intVecBaseSet()` 调用之后，`excVecInit()` 程序初始化所有的异常向量到默认的处理程序，它会中断并报告因程序错误或者不可预料的硬件中断而产生的异常。



注意：对于某些结构，此规则会有一些例外，`intVecBaseSet()` 必须在启用中断和启动内核之前调用。请查阅相应的 VxWorks 结构补充文件。

初始化系统硬件为静态。系统硬件使用非独立于系统的程序 `sysHwInit()` 来初始化。这主要包括复原和禁用在中断启用之后能够引起中断硬件设备（在内核启动之后）。这是非常重要的，因为 VxWorks ISRs（for I/O 设备，系统时钟等）直到系统初始化在 `usrRoot()` 程序中完成后，才能连接到它们的中断向量。但是，因为内存池仍然未被初始化，你不能试图在 `sysHwInit()` 中将中断处理器与一个中断相连接。

5. lib/*:a:kernellnit()

`kernellnit()` 程序初始化多任务环境而且没有返回值。它使用了下列参数：

- 应用程序被生成为“root”任务，典型的例子为 `usrRoot()`。
- 堆栈的大小。
- 可用存储器的启动；这就是，在 VxWorks 映像主文本、数据、和 bss 段之后的存储器。在此区域之后的所有存储器被添加至由 `memPartLib` 控制的系统内存池。对于动态模块加载、任务控制块、栈等的分配，都不在此区域。
- 由 `sysMemTop()` 指示的存储器的顶端。如果存储器的连续模块在正常的存储器分配中被保留下来，则设置 `config.h` 中的宏 `USER_RSERVED_MEM` 为需要的保留存储器

的大小。

- 中断堆栈大小。中断堆栈与栈空间的最大值一致，这个堆栈可被任何可能被调用的中断级程序使用（加上嵌套中断的安全余量）。
- 中断封锁优先级。对于有优先级概念的结构，这是最大优先级。对于没有优先级概念的结构，这是为禁止中断而产生的屏蔽。请查阅适当的 VxWorks 结构附录。

kernelInit() 程序调用 intLockLevelSet()，禁止轮询模式，如果结构支持还会创建一个中断堆栈。然后再从内存池顶层创建一个 root 堆栈和 TCB，产生 root 任务 usrRoot()，并中断正在执行的 usrInit() 线程。在此时，中断允许已经开启；很关键的是所有的中断源被禁用并清除等待状态中断。

usrConfig.c 和 bootConfig.c:usrRoot()

对于一般的 VxWorks 开发环境，usrRoot() 按 configAll.h 和 config.h 中的配置来初始化 I/O 系统、安装驱动、创建设备并设置网络。

usrRoot() 程序调用 memInit()。还可以有选择地调用 memShowInit() 和 usrMmuInit()。

SysClkConnect() 程序是系统开始执行多任务以后，在 BSP 中调用的第一段程序。它马上调用 sysHwInit2()。这是进一步完成在 sysHwInit() 中不能完成的目标板初始化工作最合适的时间；例如：可以在 sysHwInit2() 中完成附加设备的 intConnect()。

60Hz 系统时钟是由调用 sysClkRateSet() 和 sysClkEnable() 来设置的。而 VxWorks 可以不用运行系统时钟来引导。但是，网络驱动如果已经挂上，那么它通常执行一个 taskDelay() 来允许硬件重新工作全部完成。如果没有运行系统时钟，此延迟没有终止，将会导致系统挂起。

系统时钟可以动态的通过应用程序或命令解释器来改变。但是，取得时钟频率的“snapshot”的功能可能被这种意外的频率改变而损坏。

如果在 configAll.h 中定义了 INCLUDE_IO_SYSTEM，那么 VxWorks I/O 系统是由调用 iosInit() 程序初始化的。理论规定了随后安装驱动器的最大个数、系统同时可以打开的文件的最大数和 VxWorks I/O 系统包括的“null”设备的名字。此 null 设备的输出是“bit-bucket”而输入通常返回文件结束符。

在 INCLUDE_IO_SYSTEM 中包括的和未包括的设置也影响着是否创建控制台设备，是否设置标准输入、标准输出，标准错误输出设备；请参阅下两节以获取更多信息。

如果包含板载串行端口的驱动程序 (INCLUDE_TTY_DEV)，它通过调用驱动程序中的初始化程序来安装 I/O 系统，默认函数是 ttyDrv()。实际的设备是通过调用驱动程序中的设备创建程序来创建和命名的，默认函数是 ttyDevCreate()。此程序参数包括设备名，一个串行 I/O 通道描述符（来自 BSP），以及输入和输出缓冲器大小。

宏 NUM_TTY 规定 tty 端口的数值（默认是 2）；CONSOLE_TTY 规定了哪个端口是控制台输出端口（默认是 0）；并且 CONSOLE_BAND_RATE 规定了传输速率（默认是 9600）。这些宏定义在 configAll.h 中，但对非标准端口数目的目标板，可以在 config.h 中修改。

计算机可以使用键盘输入和 VGA 输出的交互控制台；请查阅用户的 PC 工作站文件的详细信息。

调用 `ExcInit()` 程序来初始化异常处理。其他的功能是按照在 `configAll.h` 中的宏所规定进行有选择的初始化（请查阅 VxWorks 程序员指南：配置）。

如果定义了 `INCLUDE_WDB`，就要调用 `src/config/usrWdb.c` 中的 `wdbConfig()`。此程序初始化了目标板代理的通信接口，然后启动代理服务。如需更多的有关配置目标板代理和目标板代理初始化顺序的信息，请查阅 Tornado 用户指南：设置和启动。

如果定义 `INCLUDE_USER_APPL` 了，程序执行宏 `USER_APPL_INIT`。但是，用户必须确认宏 `USER_APPL_INIT` 是合法的。

2.4 需要避免的错误

下面所列的大部分错误可以被总结为：在错误的地方或时间做了正确的事情。环境是重要的。

1. 忘记 `LOCAL_MEM_LOCAL_ADRS`

许多 BSP 编写者假定 `LOCAL_MEM_LOCAL_ADRS` 是零并且没有将它包括在需要对存储器起始地址进行补偿的宏模块中。大部分使用者没有改变 `LOCAL_MEM_LOCAL_ADRS` 的值，但是此问题趋向于被复制并在开发工具中重复。

2. 在 `romInit.s` 中做了太多

许多 BSP 编写者试图将过多的设备初始化程序写入 `romInit.s`。使用 `romInit.s` 中的初始化程序作为初步步骤。利用 `sysLib.c` 文件中的 `sysHwInit()` 函数来处理实际的设备初始化。请查阅 `romInit.s:romInit()`。

3. 在 `sysALib.s` 中做的太少

许多 BSP 编写者认为 `romInit.s` 中的任何初始化只需要做一次。`sysALib.s` 中的 `sysInit()` 程序应该重复 `romInit.s` 中所有的初始化。它在一些情况下可能会忽略存储控制器的设置。如没有这样做，则需要用户对 VxWorks 映像中简单的配置改变而重新组建引导 ROM，请查阅 `sysALib.s:sysInit()`。

4. 将修改过的驱动程序存放在错误目录下

BSP 编写者经常修改 Wind River 公司的设备驱动程序，也提供他们自己的驱动程序。这些特定 BSP 的驱动程序只能在 BSP 的特定目录下，而不能放在 Wind River 的 `target/src/drv` 和 `target/h/drv` 目录下。特定 BSP 程序属于特定 BSP 目录 `target/config/bspname`。这对使用来自各种 BSPs 的客户可能有不好的作用。只有 Wind River 公司的源程序才能被存放在这些目录下。

5. 配置选项的混乱

在文件 `config.h` 中，应该对用户说明 BSP 配置的明确选项。用户不能选择的材料不应在 `config.h` 中，而应在 `bspname.h`。用户不应该需要计算进入 `config.h` 的值。如果寄存器需要使用高 16 位字节地址载入，用户应该输入完整的地址。计算值的程序已载入寄存器中。用户应该不需要做这些；请查阅 BSP 内容。

6. 使用不可屏蔽中断 (NMI)

因为 Wind 内核使用 `intLock()` 作为保护内核数据结构的首要办法，应该避免使用不可屏蔽中断 (NMI)。如果一个 NMI 中断服务程序 (ISR) 访问 VxWorks，并导致内核目标被改变了，则失去了保护的作用并有可能出现意外现象。关于高中断优先级的 ISR 的更多信息，请查阅“VxWorks 程序员指南：基础 OS”。

同时也要注意一个标记为中断安全的 VxWorks 程序并不意味着对于 NMI 中断也是安全的。相反的，许多标记为中断安全的程序实际上对于 NMI 中断是不安全的。

第3章 创建一个新的 BSP

3.1 概 述

在 Tornado 环境中使用有效的工具创建一个新的 BSP，需要按步骤一步一步地进行开发，每一步都是建立在前一步基础之上，步骤如下：

- (1) 建立一个开发环境的基础；
- (2) 编写 BSP pre-kernel 初始化代码；
- (3) 启动一个最小的 VxWorks 内核，然后向其中添加定时器、串口设备、中断控制器的基本驱动程序；
- (4) 启动目标板，连接 Tornado 开发工具；
- (5) 完成 BSP。主要包括网络、引导 ROMs、SCSI、缓存、MMU 初始化代码和 DMA；
- (6) 使用一个新的工程工具生成一个默认的项目工程。

本章对上面列出的每一步进行了详细的介绍。本章的目标不仅是要新建一个 BSP，而且还要尽量在最短的时间里，使你了解 Tornado 的开发工具，尤其是 WDB 服务器。因为 WDB 服务器是与内核连接的，所以它可以与内核共享初始化代码。因此，在初始化代码运行之后，你可以启动 WDB 服务器，也可以启动 VxWorks 内核，或者将两者同时启动。

使用 WDB 服务器，可以调试与之连接的 VxWorks 映像。这一点与传统的 ROM-monitor 方法不同，传统的方法需要你首先将监视器安装到目标板，然后安装操作系统。

与传统的方法相比，在目标板中嵌入 WDB 代理的方法具有以下一些优势：

- 只需要编写一段初始化代码模块。在传统的 ROM-monitor 方法中，必须编写两段初始化代码模块：一个是为监视器编写，另一个是为了操作系统。另外，传统的方法需要进行非标准化的修改，以便解决 MMU（存储器管理单元）、向量表、设备初始化之间相互争抢的问题^①。
- 代码量少，因为 VxWorks 和目标板可以共享一般的库函数中的函数，例如 memcpy()。
- 传统的 ROM 监视器调试只能在“系统模式”下进行，整个操作系统被看作是一个线程进行调试。而 WDB 服务器，除了系统模式外，还提供了一个完整的 VxWorks-aware 任务模式。这种模式允许选择操作系统中的一部分（例如一个独立的任务）进行调试，

^① 传统的 ROM-monitor 方法暂时地修改操作系统初始化代码，来确保监视器使用的 MMU、部分向量表和设备没有被复位。仅仅通过运行一段初始化代码，就可以解决这些问题。

而不会影响系统中的其他部分。

如何下载 WDB 代理程序和 VxWorks 内核取决于所处的开发阶段。在编写和调试目标板初始化代码时，必须建立自己的下载方式，这一点与传统的方法相比既没有优势也没有劣势，在传统的 ROM-monitor 方法中，需要为监视器建立一个下载路径。当目标板初始化代码可以正常工作之后，接下来如何做，取决于下载路径的速度。如果有一条高速的下载路径，就可以在进一步的内核开发中继续使用它。这比只能使用串口下载路线的 ROM-monitor 方法更有优势。如果下载路径速度比较慢，应该将内核、代理程序、以及尽可能多的 VxWorks 一般性代码烧写进 ROM。

因为 Tornado 开发环境工具允许动态地下载和调试代码，所以可以下载扩充代码（例如应用程序代码、新的驱动程序、附加硬件的初始化代码等）并且使用 WDB 服务器调试这些扩充代码。这一点比传统的方法具有优势，在传统的方法中，每一次都必须下载全部的内核代码，即使大部分代码都没有变化。

3.2 建立开发环境

建立开发环境就是建立一个将代码下载到目标中的机制并且调试下载后的代码。通常，从一些简单的测试代码映像开始要比从一个完整的已烧录的 VxWorks 映像开始好。通过你的参考 BSP，并且编译 VxWorks（一个 RAM 映像）或者 VxWorks_resrom_nosym（一个 ROM 映像），就可以找到需要的编译器和连结器的标志。

1. 为将代码下载到目标板选择一种技术

下面是一些将代码下载到目标板中的比较常用的技术：

- 使用在目标板供应商的调试 ROMs 中提供的下载协议。这种方法的缺点是下载速度一般都很慢，优点是比较容易建立。
- 使用 JTAG 调试器，例如 visionICE 或者 visionPROBE 仿真器。这种方法的主要缺点是不能对所有的构架或 CPU 都获得有效的支持。然而，它可以获得较快的下载速度，而且允许你单步地遍历初始化代码。
- 使用 ROM 仿真器（例如来自 AMC 的 NetROM）。这种方法的缺点是，它会占用你很多时间去学习如何使用这种方法。优点包括：下载速度快，对于多数目标板比较轻便，通信协议允许调试信息通过 ROM 插座从目标板传送到主机中。

当把代码下载到目标板后，要检查存储器或 ROM 以确保代码被装载到了正确的位置。Wind River 公司提供的 GNU 工具 nm 和 objdump 可用于已编译好的映像中，以检查目标存储器中应有的内容。需要特别注意的是文本和数据段的开始地址。

2. 为调试已下载代码选择一种方法

下一步是建立一个用于跟踪你的初始化序列进程的机制。下面是一些比较通用的技术：

- 使用 JTAG 设备, 例如 visionICE 或 visionPROBE。
- 使用目标板上的本地调试 ROM (如果它具有中断支持)。
- 使用逻辑分析仪跟踪处理器的地址线。
- 编写一个调试库完成类似如下的功能:
 - 点亮 LED。
 - 写入永久性内存 (本地的, 目标板以外的, 或有电池支持的存储器)。
 - 以查询模式通过串口传送一个字符。
 - 将数据通过 NetROM 调试端口传送。

在已经选择了一种调试机制后, 核实可以跳到代码的入口处并成功的运行, 测试代码应该按需要设置板载的 RAM, 并通过向其中写入并读出数据来证明它正在正常工作。测试程序将有可能使用与 VxWorks 相同的初始化代码。有关 romInit.s 和 sysALib.s 文件的详细内容, 请参见 3.3.1 编写 BSP 文件。

 **注意:** 如果测试基于 ROM 的 VxWorks 映像中的调试库, 要注意数据段和 bss 段还没有建立, 尤其是依赖于全局变量的正确初始化的库还不能工作。

3.3 编写 BSP Pre-Kernel 初始化代码

这一部分描述了如何编写 BSP 文件, 编译、下载 VxWorks, 调试初始化代码, 以及在启动内核之前如何和何时启动 WDB 服务器。

3.3.1 编写 BSP 文件

通过复制 target/config/templateCPU 中的 BSP 模板文件创建一个 BSP 目录。下面将逐个介绍在修改文件时需要考虑和注意的一些事项。

1. makefile 文件

makefile 文件中至少必须定义以下一些宏:

CPU	目标板 CPU 必须与参考 BSP 中的相同。
TOOL	主机的工具链 (例如 gnu) 必须与参考 BSP 中的相同。
TGT_DIR	目标板目录的路径, 默认值为 installDir/target。
TARGET_DIR	BSP 目录名。
VENDOR	目标板生产商的名字。
BOARD	目标板的名称。

ROM_TEXT_ADRS	引导 ROM 的十六进制入口地址。对于多数目标板，它被设置在 ROM 地址区的开始部分，然而有一些硬件结构体系使用 ROM 中的这一部分作为复位向量，因此就要设定相应的偏移量，偏移量通常依赖于体系结构。所以，这个宏的低字节可以从参考 BSP 中复制得到。
ROM_WARM_ADRS	引导 ROM 的十六进制热入口地址。这通常是一个固定的 8 字节偏移量，超出了冷启动引导入口点 ROM_TEXT_ADRS。当在硬件 ROM 代码中需要一个开关时，sysToMonitor() 中的代码应该直接跳到 ROM_WARM_ADRS。
ROM_SIZE	用十六进制表示的 ROM 区域的大小。
RAM_LOW_ADRS ^②	装载 VxWorks 的地址。
RAM_HIGH_ADRS ^②	在将引导 ROM 中的映像复制到 RAM 中时使用的目标地址。
HEX_FLAGS	特定体系结构中用于产生 S-record 文件的 objcopy 工具的标志。
MACH_EXTRA	任何额外的与目标板相关的文件，现在先把它作为一个空的声明。

 **警告：** 上述的十六进制地址不用 0x 开头。

在 makefile 文件中可能还会用到其他一些特定体系结构的宏，例如，i960 需要知道从哪里连接初始引导纪录。对于特殊体系结构的内容，可以参见相应的 VxWorks 体系结构的补充内容或者参见“第 4 章硬件介绍”。

2. bspname.h 文件

参见 bspname.h。

3. config.h 文件

有关 config.h 文件的基础知识，已经在前文中提到过了，但是需要作一些小小的改动。首先，确定将下面一段代码添加到包括 configAll.h 文件的代码行之前：

```
#define BSP_VER_1_1 1
#define BSP_VER_1_2 1
#define BSP_VERSION "1.2"
#define BSP_REV "0" /* 0 for first revision */
```

^② RAM_LOW_ADRS 和 RAM_HIGH_ADRS 都是绝对地址，通常是指定体系结构的从 DRAM 开始的偏移量。通过查阅《VxWorks 程序员指南》一书的附录中的存储器分配图，就可以快速的了解 VxWorks 存储器的分配。

在开始之前，将下面几行添加到头文件接近结尾的地方：

```
#undef INCLUDE_MMU_BASIC          /* bundled mmu support */
#undef INCLUDE_CACHE_SUPPORT      /* cache support */
```

确保对 ROM_TEXT_ADRS, ROM_SIZE, 和 RAM_LOW_ADRS 的定义与在 makefile 文件中的定义相一致，确保已经对 LOCAL_MEM_LOCAL_ADRS 和 LOCAL_MEM_SIZE 作了正确的定义。

4. sysLib.c 文件

由于 sysLib.c 文件是到目前为止最大的 BSP 文件，在这个阶段只能使用一些基本的函数：sysModel(), sysBspRev(), sysHwInit(), sysHwInit2() 和 sysMemTop()。与 VxWorks 5.3 版本相比，sysBspRev() 是一个新的函数，这个函数的使用方法如下：

```
char * sysBspRev (void)
{
    return (BSP_VERSION BSP_REV);
}
```

sysLib.c 文件还应该包括下列驱动程序段：

```
#include "mem/nullNvram.c"
#include "vme/nullVme.c"
```

sysHwInit() 程序是 sysLib.c 文件的核心，大部分工作都在这一部分完成。这个程序将所有的设备都复位到静止状态，这样在中断允许状态下它们也不会产生中断信号。

5. romInit.s 文件

在接通电源后（冷启动），处理器从 romInit() 函数开始执行，这个函数必须是 romInit.s 文件的文本段的第一段代码。对于热启动，处理器从 romInit() 加上一个小的偏移量开始执行（见 sysLib.c 文件中的 sysToMonitor() 函数）。

大多数硬件和设备的初始化工作都在后面的初始化序列中由 sysLib.c 文件中的 sysHwInit() 函数完成。romInit() 的工作是完成将控制传送给 romStart()（在 config/all/bootInit.c 文件中）所需要的最少的设置：

- 初始化处理器（这些代码专门用于处理器而不用于目标板，因此可以从参考 BSP 中复制）：
 - 屏蔽处理器中断；
 - 设置 STACK_ADRS (在 configAll.h 文件中定义) 的初始化堆栈指针；
 - 禁止处理器缓存。
- 如下述所需的对目标板的 DRAM 的初始化访问（这些代码是目标板专用的）：

- 等待状态;
- 刷新率;
- 芯片选择;
- 禁止二级缓存 L2 (如果需要的话)。

最后, 通过了启动类型, `romInit()` 跳到 `bootInit.c` 文件中的 `romStart()`。对于冷启动, 启动类型应该是 `BOOT_COLD`, 对于热启动, 启动类型应该是从 `sysToMonitor()` 传过来的参数。这个文件还必须包括一个叫做 `sdata` 的数据变量, 用来标示数据段的开始。这个变量通常存储了一个字符串, 例如“start of data”。

请查阅这个产品的附件 BSP 模板中的 `romInit.s` 文件的模板。

6. `sysALib.s` 文件

这个文件包含了基于 RAM 的映像的入口点, 例如 VxWorks。这个入口点 `sysInit()`, 完成了将控制传送给 `usrInit()` (定义于 `config/all/usrConfig.c` 文件中) 所需的最少量的设置, 这个函数与 `romInit()` 是相似的。`sysInit()` 函数对于每一种体系结构几乎都是相同的, 同样的, 从 BSP 模板中复制这段代码是最好的开始办法。`sysInit()` 函数通常仅仅屏蔽处理器中断, 并设置堆栈指针。对于堆栈向下增长的处理器, 堆栈指针设置在 `sysInit()` 中, 对于堆栈向上增长的处理器, 堆栈指针被设置在 `sysInit()` 下面的数百字节, 然后它调用 `usrInit()` 函数, 将变量 `BOOT_WARM_AUTOBOOT` 传送给它。多数的硬件和设备的初始化工作是由 `sysLib.c` 文件中的 `sysHwInit()` 函数在后面的初始化序列中完成的。

如果可能的话, BSP 程序最好用 C 语言编写, 但是, 如果由于一些技术原因被迫要求助于汇编语言时, 那么将程序放在 `sysALib.s` 文件中。

7. `config/all/usrConfig.c` 文件

在开发过程中, 可能需要添加调试代码, 或者对一般的配置文件 (非 BSP) 作一些类似的修改。这时, 不需要对一般性代码进行直接的修改, 可以如下述在 BSP 目录中创建本地备份:

(1) 首先将 `config/all/usrConfig.c` 文件和 `config/all/bootInit.c` 文件复制到 BSP 目录中, 然后, 在 BSP 的 `makefile` 文件中, 在对 `HEX_FLAGS` 的定义之后, 添加下列代码行:

```
BOOTINIT = bootInit.c
USRCONFIG = usrConfig.c
```

这样, 在编译 VxWorks 时, `makefile` 文件将会使用这些配置文件中的复制。

(2) 下一步, 修改 `usrConfig.c` 文件的复制, 使用 `#if FALSE/#endif` 语句来删除了 `pre-kernel` 初始化代码外的全部内容, 这样, `usrExtra.c` 文件就不包含在内了, 而且 `usrRoot()` 函数的内容也变成空的了:

...

```
#if FALSE
#include "usrExtra.c"
#endif
...
void usrRoot
{
...
}
#endif
}
```

包含了 `usrExtra.c` 文件，就可以与 `src/config` 中的其他配置文件连接在一起，尽管现在不需要它们，但是将来在增加某些新特性时可能会需要它们。例如，当准备好开始启动内核和目标板工作时，需要连接如下合适的配置代码：

```
#if FALSE
#include "usrExtra.c"
#else
#include "usrWdb.c"
#endif
```

3.3.2 编译和下载 VxWorks

装载到目标板中的 VxWorks 映像取决于所使用的下载方法，主要的映像如下所述：

1. VxWorks

这个映像是从 RAM 开始执行的，必须通过一些外部手段，如利用目标板本地的调试 ROM 将其装载到 RAM 中。

2. VxWorks_rom

这个映像从 ROM 开始执行，但是它的文本段和数据段是连接到 RAM 的地址的，开始时它将自身复制到 RAM 中，然后从那里继续执行。

3. VxWorks_resrom_nosym

这个映像从 ROM 开始执行，只有数据段被复制到 RAM 中。

如果下载路径将映像放入 RAM 中（例如，当使用销售商的调试 ROM 时），使用 VxWorks。如果下载路径将映像放入 ROM 中（例如，使用 NetROM 时），可以使用

VxWorks_rom 或者 VxWorks_resrom_nosym。VxWorks_rom 的优势是它调试起来比较容易，因为软件的中断点只能设置在 RAM 的地址上。VxWorks_resrom_nosym 的优势是它使用的目标板中的内存较少。这两个 ROM 映像的 makefile 文件允许指定可选择的 hex 扩展名(例如, VxWorks_rom.hex)，来产生一个 S-record，而不是一个目标文件。

有一个名为 depend.cputool 的文件，它包含了 makefile 文件使用的附属文件规则，如果依赖文件不存在的话，makefile 文件将会自动的生成一个附属文件。当向 BSP 中添加了新的文件后，要将附属文件删除，并使 makefile 文件自动重新生成一个。

在将代码下载到目标板之后，检查内存和/或 ROM 以确保代码已经被装载到了正确的地方。对 VxWorks 映像，使用 nm 和 objdump 工具，来查看目标板的内存中应有的内容，要格外注意文本段和数据段开始的地址。

3.3.3 调试初始化代码

ROM 和 RAM 的初始化序列的开端是不同的，但是它们在其他方面是相同的。每一个 BSP 程序所需要做的具体的工作在前一部分已经作了详细的介绍，这一部分将要复习初始化序列的步骤，并提供一些当在某一步失败时如何来检查错误的方法。

1. ROM 初始化

这一部分描述了基于 ROM 的 VxWorks 映像 VxWorks_rom 和 VxWorks_resrom_nosym 的初始化序列。

(1) romInit.s: romInit()

在接通电源后（冷启动），处理器从 romInit() 开始执行，它的工作是完成将控制传送给 romStart()（在 config/all/bootInit.c 中）所需要的最基本的设置，大多数硬件和设备的初始化工作都由 sysLib.c 文件中的 sysHwInit() 在后面的初始化序列中完成。

(2) bootInit.c: romStart()

文本段和数据段以下述的一种方式从 ROM 中复制到 RAM 中：

- 对于 VxWorks_rom, 文本段和数据段都要复制到 RAM 中。
 - 对于 VxWorks_resrom_nosym, 仅仅将数据段复制到 RAM 中。
- 在完成复制工作之后，检查数据段已经正确的初始化了，例如：

```
int thisVal = 17; /*some data segment variable */
...
if (thisVal != 17)
    somethingIsWrong();
```

如果有错误，检查 RAM 是否能够正确的访问，例如：

```

int dummy;
...
dummy = 17;
if (dummy != 17)
somethingIsWrong( );

```

如果能够正常访问 RAM, 则检查数据段是否以正确的偏移量复制到内存当中, 只有 VxWorks_resrom_nosym 映像可能会发生这样的问题。romStart() 函数默认数据的位置在 ROM 中文本段末尾的特定体系结构的偏移量处, 如同 bootInit.c 文件中的如下代码段所示:

```

#if (CPU_FAMILY == SPARC)
bcopyLongs ((UINT *) (etext + 8), (UINT *) RESIDENT_DATA,
#elif ((CPU_FAMILY == MIPS) || (CPU_FAMILY == PPC))
bcopyLongs ((UINT *) (etext + 0), (UINT *) RESIDENT_DATA,
#else
bcopyLongs ((UINT *) (etext + 4), (UINT *) RESIDENT_DATA,
#endif

```

然而, 如果使用可选择的工具来创建 ROM 映像, 那么这个偏移量可能不同, 在这种情况下, 需要相应的调整偏移量。

romStart() 函数所作的最后一项工作是调用 usrConfig.c 文件中的普通的初始化函数 usrInit()。i960 是一个特例, 它首先跳到 sysInitAlt() 函数 (在 sysALib.s 文件中), 这是为了在跳到 usrInit() 函数之前重新装载处理器平台。其他的初始化序列将在后面的一般性初始化部分讲述。

2. RAM 初始化

这一部分讨论了基于 RAM 的 VxWorks 映像的初始化函数。

sysALib.s: sysInit()

VxWorks 的入口点是 sysInit(), 它的工作是完成将控制权传到 usrInit() (在 usrConfig.c 文件中) 所需的最基本的设置, 大多数硬件和设备的初始化是由 sysLib.c 文件中的 sysHwInit() 函数在后面的初始化序列中完成的。

3. 一般性初始化

初始化代码的其他部分, 对于基于 ROM 和基于 RAM 的映像是通用的。

(1) usrConfig.c: usrInit()

从一个 BSP 程序员的角度来看, usrInit() 函数的主要意义是清除了 bss 段 (这样所有还未初始化的 C 语言全局变量, 现在都为零), 然后调用 sysHwInit() (在 sysLib.c 文件中) 来初始化硬件。如果内存已经作了适当的设置, 这段程序一般不会发生错误。

(2) sysLib.c: sysHwInit()

这时 BSP 初始化代码的核心部分，必须将所有的硬件复位到静止状态，这样在将来允许中断的时候，就不会产生未初始化的中断。

(3) usrConfig.c: usrInit()

在 sysHwInit() 工作完成之后控制权再次交给 usrInit(), usrInit() 的最后一项工作是调用 kernellInit() 函数来启动 VxWorks 内核。pre-kernel 初始化代码到此结束。kernellInit() 并不返回，然而，作为第一项任务它通过 usrRoot() 函数来启动内核。在 kernellInit() 的内部中断允许最终被打开，如果在 sysHwInit() 中没有将所有的中断源都禁止并且清除，可能会引起严重的混乱。

如果用户希望在目标板的控制下启动内核，那么就在这时启动目标板，这一步是可供选择的，更加详细的内容，请参见“3.3.4 在启动内核之前启动 WDB 服务器”。

(4) usrConfig.c: usrRoot()

余下的 VxWorks 初始化工作是在内核已经在 usrRoot() 中启动之后完成的，详细的内容将在后续章节中进行讲解。在这个阶段，如果 usrRoot() 能够证实 sysHwInit() 函数编写的正确就已经足够了。

如果 kernellInit() 函数已经被调用，但是还未开始执行 usrRoot() 便产生错误而失败了，那么，错误可能是下面两种可能之一，一种情况是 sysMemTop() 函数返回了一个错误的地址，另一种可能性更大的情况是，某个硬件设备没有复位并产生了一个中断。对于第二种情况。

必须修改 sysHwInit(), 来复位产生中断的设备。

为了找到这个中断源，首先计算产生的中断向量，然后运用下面介绍的任何一种方法：

- 使用逻辑分析仪，来查找访问中断向量表的指令。
- 使用 ICE 在中断向量表中设置断点。
- 修改 sysHwInit(), 通过中断控制器来屏蔽可疑的中断向量。
- 修改 sysHwInit(), 使用 intVecSet() (不能使用 intConnect(), 因为它要调用 malloc(), 而且 VxWorks 内存分配算符还未被初始化) 将调试程序连接到可疑的中断向量。

此时，已拥有了一个可以工作的内核，但是没有设备驱动程序，VxWorks 仅仅需要一个定时器和一个中断控制器（如果可能的话）驱动程序，多数 BSP 还具有串行设备的驱动程序。如何添加 VxWorks 支持的基本驱动程序，将在“3.4 使用最小化的内核”中介绍。

3.3.4 在启动内核之前启动 WDB 服务器



注意：这一部分只能运用于由命令行编译生成的映像。

在创建一个新 BSP 时，这一步是可以选择的而且很少使用。在内核之前启动 WDB 服务器的缺点是：

- 一旦硬件的初始化代码编写完成之后，启动内核要比启动服务器占用较少的时间，因

为大多数开发者认为使内核开始工作是一个重要的里程碑，所以应该从内核开始。

- 在内核之前启动服务器对于基本内核的正常工作没有带来任何帮助，这是因为基本内核没有向已经编写好的代码中添加什么内容，仅仅是一个定时器驱动程序和一个中断控制器这两个简单的设备。
- 在内核之前启动服务器限制只能使用系统模式进行调试。

较早启动服务器的惟一原因是，下载环境速度非常慢。在那种情况下，可能会希望将所有的内容尽早的装进 ROM 中，以便节省下载周期。

有关在启动内核之前启动服务器的内容，在《Tornado 用户指南》一书的“设置和启动”部分进行了介绍。下面两部分讲述了其他的重要知识。

 **警告：** 因为虚拟 I/O 驱动需要使用系统内核，应向 config.h 文件中添加下列代码：

```
#undef INCLUDE_WDB_VIO
```

如果计划使用目标板的串行通信端口，那么这里有一个重要的警告。当内核一开始启动，处理器的中断允许便被打开，但是驱动程序的中断处理尚未连接，必须确保所使用的用于服务器通信的串行设备没有产生中断。如果目标板有中断控制器，在 sysHwInit() 中使用它来屏蔽串行中断。要注意目标板代理试图使所有的驱动设备工作在“interrupt on first packet”模式。因此，需要修改串行驱动程序，以便拒绝进入中断模式，即使服务器请求中断。

系统模式调试技术

当服务器开始正常工作之后，将要使用它调试它所连接的 VxWorks 映像，为了节省下载时间，应该将想要测试的 VxWorks 代码连接到 ROM 映像中。尤其是，应该浏览一下“3.4 使用最小化的内核”这一部分内容，以便了解与 VxWorks 连接的驱动代码。为了避免重新擦写 ROM，可以考虑像下面这样在 BSP 和驱动程序中添加钩子：

```
void (*myHwInit2Hook)(void); /* declare a hook routine */
...
void sysHwInit2 (void)
{
    if (myHwInit2Hook != NULL) /* 有条件的调用 */
    {
        myHwInit2Hook( );
        return;
    }
    ... /*默认代码*/
```



```
}
```

这样允许动态的从调试器替换程序，例如，为了覆盖上述的 `sysHwInit2()` 的作用，只需要在叫作 `myLib.o` 的模块中新建一个名为 `myHwInit()` 的版本，然后输入：

```
(gdb) load myLib.o
(gdb) set myHwInit2Hook = myHwInit2
(gdb) break myHwInit2
(gdb) continue
```

然而，如果在启动内核之前启动服务器，必须在调用 `sysHwInit()` 后再启动它，因此不能覆盖 `sysHwInit()`。在另一方面，可能想要添加在内核启动之前就被调用的附加的硬件初始化代码，在这种情况下，可以在调用 `kernelInit()` 之前添加一个异常分支，作为异常分支的另一种选择，可以使用 GDB 的调用程序来调用调试器的函数，例如：

```
(gdb) call myHwInit2
```

使用异常分支代替调用机制的优势在于：

- 异常分支允许避免执行原始代码；
- 异常分支速度更加快。

如果目标板有“放弃”按钮，可以考虑将调试程序连接到“放弃”中断，然后通过调试器中断处理器中设置一个断点。这样就提供了一种方法，使在系统近似崩溃的时候仍能获得对系统的控制权。在这种情况下，最好将放弃开关连接到非屏蔽中断（NMI）。



警告： 只有像“放弃”这样的致命的中断才可以连接到 NMI，如果一个设备的中断连接到 NMI，可能会导致内核不能正常工作。

3.4 使用最小化的内核

一个最小化的内核包括添加仅仅几个最简单的设备驱动，因为这些驱动包括连接中断服务程序，所以这个部分将从介绍 ISR 指导开始。

3.4.1 ISR 指导

在内核启动之后，可以使用 `intConnect()` 函数和 `INUM_TO_IVEC` 宏来将中断服务程序（ISRs）连接到中断向量表，例如：

```
intConnect (INUM_TO_IVEC (intVec), proc, param);
```

其中参数 `intVec` 是中断向量, `proc` 是与中断相对应的被调用的 C 语言代码, `param` 是用于传递 `proc` 程序代码的参数。更详细的内容, 请参见 `intConnect()` 函数的调用入口。

在编写 `ISRs` 时常犯的一个错误是, 常常忽略用于清除中断源的中断应答, 这样就会导致一个中断刚刚响应完成, 另一个中断接着又产生了。

非屏蔽中断 `NMIs` 不能与 `VxWorks` 中的很多库函数联合使用, 它们必须非常小心的使用和执行。有关 `ISRs` 高级中断的更多的知识, 请参见《`VxWorks` 程序员指南》一书的“操作系统基础”部分。

 **警告:** `ISRs` 不能调用 `VxWorks` 中的某些函数, 有关中断层可以调用的函数的列表, 请参见《`VxWorks` 程序员指南》一书的“操作系统基础”部分中对 `ISR` 的一些特殊限制的讨论。

3.4.2 需要的驱动程序

`VxWorks` 所需要的惟一的驱动程序是系统时钟, 尽管对于某些体系结构, 例如 `80x86` 和 `PowerPC`, 同样需要一个中断控制器驱动程序。执行系统时钟驱动程序通常可以使用 `src/drv/timer` (和 `src/drv/intrCtl`, 如果还需要中断控制器驱动的话) 中已经存在的驱动程序, 如果重用一段已经存在的驱动程序, 需要做的全部工作就是在 `sysHwInit()` 中完成特定目标板的硬件初始化, 并且通过调用 `sysHwInit2()` 中的 `intConnect()` 来连接中断。

定时器驱动器是比较简单的设备, 可以通过修改 `usrRoot()` 来周期的完成某个动作, 如点亮 `LED`, 例如:

```
void myTestCode (void)
{
    while (1)
    {
        taskDelay (5*sysClockRateGet ()); /* 每 5 秒钟一次 */
        sysFlashLed ( ); /* 点亮一个 LED */
    }
}
```

3.4.3 串口驱动

尽管不总是需要, 但大多数 `BSP` 都包括串口驱动, 随着引入 `Tornado`, 串口驱动已经

发生了深刻意义的变化（事实上，那是对 BSP 所作的最主要的变化）。为了介绍这些变化，见附录 A。新的串口驱动程序位于 `src/drv/sio` 目录中，这些驱动程序中应该有一个能够适合需求。如果没有的话，可以使用 `src/drv/sio/templateSio.c` 驱动程序模板，作为自己的串口驱动程序的基础。

在这之前，使用 `#if FALSE/endif` 语句将 `usrRoot()` 函数中的大部分内容删掉了，现在，需要进一步向下移动 `#if FALSE` 这一语句，到串口设备初始化位置之后。

应该修改 `usrRoot()` 来产生一些可以用于测试驱动程序的应用程序测试代码，例如，可以周期地向控制台打印一条消息：

```
void myTestCode (void)
{
    extern int vxTicks;
    char * message = "still going...\n";
    while (1)
    {
        taskDelay (5*sysClockRateGet()); /* 每 5 秒钟一次 */
        write (1, message, strlen (message)); /* 打印一条消息 */
    }
}
```

3.5 目标板代理和 Tornado

调试服务器是通过调用 `wdbConfig()` 来初始化的，对于默认情况，这一调用是在 `usrRoot()` 的最后产生的。然而，曾经将 `usrRoot()` 改为了一个空函数（见 `config/all/usrConfig.c`），为了恢复服务器，可以在串行线路初始化代码的最后调用 `wdbConfig()`。目标板的默认配置使用网络与主机建立连接，不幸的是，到目前为止网络还不能正常的工作，因此，必须选择可以使用的另一种方法来建立连接。有关如何完成这项工作的知识，请参见《Tornado 用户指南》中“设置和启动”一章的“修改 Back End”部分。

3.6 完成安装工作

这一部分概括了完成安装工作的各种各样的不同的基本任务，包括对清除、定时器、网络和对其他一些问题的讨论。

3.6.1 清除

到目前为止，一直在使用个人的 `usrConfig.c` 和 `bootInit.c` 文件的复制，BSP 应该能够与通用的版本一起工作，为了重装通用的版本，删除你的 BSP 的 Makefile 文件中的这些行：

```
BOOTINIT = bootInit.c
USRCONFIG = usrConfig.c
```

然后做一次编译清除。

在此之前，使用 `#if FALSE/#endif` 语句屏蔽掉了不需要的配置代码，现在需要使用更标准的方法来删除不需要的代码，也就是应该取消在 `config.h` 文件中对相应的宏的定义，在 `src/config/usrDepend.c` 文件中列出了必须牢牢记住的与宏相关的内容。

3.6.2 NVRAM

VxWorks 定义了一个向固定的存储区域中读写的接口，这个接口被称作非易失性的内存库。一些普通的驱动存在 `target/src/drv/mem` 目录中。

3.6.3 添加其他的定时器

驱动程序可能在包括时标驱动的高级解决工程的同时，还包括一个辅助的时钟驱动程序。

这个辅助的时钟被 VxWorks 的监视功能使用，同时也被 Tornado 的主机工具——浏览器使用，有关辅助时钟的更详细的内容，可以参见 `sysAuxClk*()` 函数的入口参数，附录 F，《WindView 用户指南》：创建 VxWorks 时标驱动。

高分辨率的时标驱动通常仅仅被 WindView 使用，但是编写这段程序会对你将来的调试工作带来帮助，接口可以在 `target/h/drv/timer/timestampDev.h` 这个头文件中找到。

3.6.4 网络

这个手册讲述了如何使用一个独立的 VxWorks 映像，`VxWorks.st`。这个映像具有两个属性：

- 它具有一个相连接 VxWorks 符号表，因为对于调试信息 Tornado 使用基于主机的符号表，所以这已经没用了；

■ 它与网络子系统相连接，但是并没有对它进行初始化。

继续使用 VxWorks 或 VxWorks_rom，来代替使用 VxWorks.st 映像。为了与网络相连接，但并不初始化，可以在调用 usrNetInit() 之前，在 usrConfig.c 文件中加入一个虚拟的条件：

```
if (FALSE)                /* add this line */
usrNetInit (BOOT_LINE_ADRS); /* so that this is not called */
```

然后，可以在后面调用 usrNetInit()，并且使用 CrossWind 来调试你的网络驱动程序。这是在不能用缓存和 MMU 的情况下先使网络运行起来的的最简单的方法。

下面是 src/config/usrNetwork.c 文件中的表格，它是用来配置 BSD4.4 的网络设备：

```
#ifdef NETIF_USR_DECL
NETIF_USR_DECL /* declaration from BSP */
#endif
...
LOCAL NETIF netIf [] = /* network interface */
{
#ifdef NETIF_USR_ENTRIES /*Additional entries,fromBSP */
NETIF_USR_ENTRIES
#endif
#ifdef INCLUDE_DC
{"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_EGL
{"egl", eglattach, (char*)IO_ADRS_EGL, INT_VEC_EGL, INT_LVL_EGL},
#endif /* INCLUDE_EGL */
...
}
```

对于 END 驱动，网络设备是通过 configNet.h 文件中下列代码段初始化的，代码段的名称是 endDrvTbl。

```
#define DEC_LOAD_FUNC dec21x40EndLoad
#define DEC_BUFF_LOAN 1
/*
* <devAdrs>:<PCIadre>:<ivec>:<ilevel>:<numRds>:<numTds>:<memBase>: \
* <memSize>:<userFlags>
```

```

*/
# define DEC_LOAD_STRING
"0x81020000:0x80000000:0x12:0x12:-1:-1:-1:0:0x80800
000"
IMPORT END_OBJ* DEC_LOAD_FUNC (char*, void*);
END_TBL_ENTRY endDevTbl [] =
{ 0, DEC_LOAD_FUNC, DEC_LOAD_STRING, DEC_BUFF_LOAD, NULL, FALSE},
{ 0, END_TBL_END, NULL, 0, NULL, FALSE},
};

```

就象所看到的那样，初始化的驱动设备是由一系列的宏来控制的，修改 config.h 文件，定义相应的宏和 I/O 参数，这样就可以为一个已经存在的 VxWorks 的网络驱动添加 BSP 支持，正确地定义宏 NETIF_USR_ENTRIES 和 NETIF_USR_DECL，可以添加自定义的 BSD 驱动支持。END 的用户只需要向 endDrvTbl 中添加适当的入口，来包含驱动。准确的驱动程序初始化语句的格式和语法，可以参考具体的驱动文件。

至于 Tornado 2.0 版本的标准产品，已经放弃了对基于 VME 的网卡，Excelan EXOS-202 和 CMC ENP-10 的支持。

3.6.5 缓存、MMU、DMA 设备

下一步是在缓存和 MMU 能够工作的情况下，使 BSP 开始工作。更多的内容，请参见“5.6 和 4.5 节”。缓存和 MMU 的配置情况能够严重的影响 DMA 设备的工作，例如网络驱动。

3.6.6 引导 ROM

引导 ROM 使用 VxWorks 内核，引导 ROM 映像和 VxWorks 映像的两个主要不同点是：

引导 ROM 映像使用 target/config/all/bootConfig.c 文件来代替 target/config/all/usrConfig.c 文件，这两个文件非常相似，运行引导 ROM 的步骤与前面所讲述的在本质上是相同的。

引导 ROM 映像是按照默认方式被压缩的，它在 bootInit.c 文件中被解压缩并复制到 RAM 中。

3.6.7 SCSI

VxWorks 支持 SCSI-2 驱动程序，参见附录 I。SCSI-1 驱动程序已经不再支持了。

3.6.8 项目工程

Tomado 文件组包含了有关创建、配置、编译一个项目工程所必需的信息。这些步骤多可以通过 GUI 项目工具来完成，关于工程工具 GUI 工作的详细内容，参见第 6 章。

传统的 ROM-monitor 方法通过暂时的修改操作系统初始化代码来确保 MMU、部分向量表、和监视器使用的设备不会被复位。通过仅仅运行一组初始化代码，这些问题便可以得到解决。

第4章 硬件介绍

4.1 概 述

VxWorks 可以运行于各种不同的销售商的硬件体系结构和目标板上, 包括: 一般的和嵌入式的硬件、VMEbus、Multibus、PCibus、单片机、工作站和 PC 机目标板。VxWorks 同样可以使用各种各样的通信媒介从不同的 UNIX 和 Windows 主机中引导启动。

由于构建的组合方式多种多样, Wind River 已经搜集了足够的目标板设计经验, 为设计最适宜 VxWorks 运行环境的目标板提供建议。然而, 这些内容不能用来确定潜在的和已经存在的端口的需求, 这样的决策还需要考虑很多的要素, 例如成本分析, 投放市场时间, 以及供电和冷却要求, 而这些要素都不属于本书所讨论的内容。

本章列举了运行时的功能需求, 提取运行时的重要组件的特征和对于潜在缺陷的警告。本章的主要目标是协助开发者为 VxWorks 运行选择合适的目标板和组件。在本章中讨论了下列的问题:

- 体系结构需考虑的问题;
- 内存;
- 总线;
- 设备。

一个独立的体系结构如何解决上述问题, 将在 VxWorks 体系结构的附录中对于每一种体系结构进行详细的讨论。

本章是对这些问题的一般性的论述, 而不是对于如何具体操作细节的指导。

4.2 结构体系注意事项

任何 VxWorks 运行环境的核心是目标体系结构, 这一部分主要专注于体系结构选择中的性能和运行时的分支, 接下来还有一些一般性的观测报告, 但是更详细的内容可以参见专门讨论特定体系结构的部分。

有关 VxWorks 体系结构支持, 请参见下列书籍 (适当的 VxWorks 体系结构补充文献):

- VxWorks API 参考手册: 驱动程序, 5.5;
- VxWorks API 参考手册: OS 库, 5.5;

- Tornado 用户指南, 2.2;
- BSP 文献资料 (与你的目标系统相似的);
- Wind River 技术手册, 可以通过网络从 WindSurf 获得。

4.2.1 中断处理

中断功能异步地将外部事件与系统连接起来,而且对于实时系统这通常是重点。VxWorks 采取了向量中断策略,即把 ISRs (Interrupt Service Routines) 与一个由中断组件产生的惟一向量连接起来。VxWorks 提供了可以动态地为这些向量编程的函数,以使其包含那些非常短小且高速的代码段的地址,这些代码调用了应用程序的 C 语言 ISR,然后再将控制权交还内核。

普通中断服务的较为复杂的因素是中断确认 (IACK),多数系统结构都提供了自动地中断确认,对于少数没有涉及这个问题的系统,ISRs 必须人工的通过寄存器访问或其他较复杂的方式来确认一个中断。

最后,不同的体系结构之间的中断延迟是不同的,中断延迟是从处理器中断请求开始到中断服务处理开始的最大时间。硬件和软件都会对中断延迟有影响。硬件会为外部中断确定优先级,因而对低优先级的中断引入一个固有的延迟。体系机构经常具有一些不可分割的指令,它们的执行时间惊人的长。

主要的问题是高速缓冲寄存器写入操作,它可能会占用数十微秒的不可中断时间。VxWorks 操作系统通过抑制处理器获取中断信号而增加了中断延迟,每一种体系结构已经将这些中断上锁优化到绝对的最小值,但是要注意在不同的体系结构之间比较时,它们还是有不同的表现的。

有关非屏蔽中断 (NMIs) 的知识,请参见《VxWorks 程序员指南》“操作系统基础”中对中断服务代码的讨论。非屏蔽中断是可以使用的,但是它们绝对不可以调用 VxWorks 内核中的程序作为服务程序的一部分,这会严重的限制 NMI 处理的功能。

4.2.2 有关缓冲寄存器的问题

很多近期的体系结构已经引入了一些指令和数据缓存来提高处理器的性能,并降低 CPU 总线的活动性,内存缓冲寄存器最困难的问题是那些技术过于强调缓存的一致性。

缓存一致性问题是指缓存中相对于内存中的信息多余的信息。如果另一个总线控制器或者 DMA 设备更新了内存,缓冲寄存器中的数据就不能再反映内存中的实际数值。在没有足够的硬件支持的情况下,解决相关性问题的只能靠软件。

不幸的是,缓存的管理在不同的体系结构之间的差异是非常大的,在某些情况下,体系结构提供了缓存管理指令,缓存管理与虚拟内存管理函数绑定在一起。

VxWorks 提供了一个缓存库接口,它在完全不同的 CPU 结构之间是统一的。这样可以

使可移植的和高性能的驱动设备在 VxWorks 中实现。更多的内容, 请参见缓存库的访问入口和 5.8.1 部分。

考虑到硬件侦听, 只有完全的缓存侦听才是有意义的, 一些处理器执行局部的侦听, 但是局部侦听不能满足内存一致性的需求, 只有当侦听硬件使内存完全保持一致时, 它对于 VxWorks 才是有用的。

没有侦听功能和 copyback 缓存的结合是不安全的, 即便所有的用户缓冲区都被定位了, 已使得它们不会与其他的缓冲区共享缓存队列 (这样可以降低危险性)。用户可以通过 memDefaultAlignment 变量设置缓存队列的大小, 来确保用户缓冲区的首地址处于高速缓存的边界处。这将导致所有的 malloc() 请求都与高速缓存的开始处对齐。

用户可以通过增加一个缓存队列所需大小的内存, 来保护缓冲区的末端。这便保证了其他的缓冲区不会与这个缓冲区共享一个缓存队列。完成了这些工作之后, 用于 DMA 操作的内存缓冲区在 copyback 情形下不会受内存一致性的影响。

很多新型的处理器实现了写入管道, 它可以在总线控制器比较繁忙的时候, 对写入操作进行缓冲。设备驱动程序需要使用于普通设备驱动器的 CACHE_PIPE_FLUSH 宏, CACHE_PIPE_FLUSH 宏操作应该插入在任何的 I/O 写操作和 I/O 读操作之间。如果一段程序是以 I/O 读操作开始的, 那么需要假设在它之前有一个 I/O 读操作。

4.2.3 MMU 支持

VxWorks 通过虚拟内存库支持一些不同的 MMU, 由于虚拟内存不适合于一些实时应用, VxWorks 可以被配置为不包括虚拟内存的支持。

更多的关于 VxWorks 虚拟内存支持的信息, 请参见如下:

- vmBaseLib 和 vmLib 的调用入口;
- 《VxWorks 程序员指南》: 虚拟内存接口;
- 《VxWorks 程序员指南》: 配置;
- “4.5 启用虚拟内存库”。

4.2.4 浮点操作支持

浮点操作被作为任务的扩展而得到支持, 这样对于不使用浮点操作的任务可节省宝贵的上下文交换时间, 可以使用特殊的浮点选项发起任务, 而且上下文交换编号提供了一种浮点环境的初始化、保存和恢复的机制。通过交换浮点数据和控制寄存器的输入与输出, 每一个任务都有效地共用了一个单一的浮点单元。通过以下方式 VxWorks 支持较高级的超越函数。

- 一般避免使用任何浮点指令的可移植式的工程, 但是对于某些具有浮点运算能力的结构可以使用优化的汇编语言来代替, 请阅读《VxWorks 程序员指南》: 配置部分中

对于如何选择可选特征的讨论。

- 对于浮点运算密集的应用，协处理器为更好的性能表现发挥了重要的作用。

4.2.5 一些其他问题

一些值得考虑的其他特性包括如下：

- endian 字节顺序选择对于整个 VxWorks 功能是透明的。
- 为了使底层网络通信具有更好的表现，一个具有写入-修改-读出操作不可分割性的体系结构（例如，测试和设置）是必需的。
- 对于不需要操作系统支持的事件，限制非屏蔽中断。

4.3 内 存

这一部分讨论了下面这些问题：

- RAM;
- ROM;
- 以太网 RAM;
- NVRAM;
- 奇偶校验;
- 地址。

4.3.1 RAM

VxWorks CISC 处理器需要 1MB 的内存来开发一个包括所有标准 VxWorks 特性的系统，如：命令行解释器、网络、文件系统、程序装载器等。RISC 处理器通常需要更多的 RAM 空间：至少要 2MB，建议用 4MB 内存。对于一个 scaled-down 的产品系统，所需要的 RAM 大小取决于应用的规模以及所选择的选项。

控制内存使用的首要配置评价值是 LOCAL_MEM_LOCAL_ADRS, LOCAL_MEM_SIZE, USER_RESERVED_MEM, ROM_TEXT_ADRS, ROM_WARM_ADRS, 和 ROM_SIZE。有关这些值得更多信息，参见“2.2.1 BSP 中所包括的文件”。

4.3.2 ROM

VxWorks CISC 处理器至少需要 128KB ROM，这刚刚满足了 VxWorks 的压缩引导

ROM 的需要。一般 RISC 处理器需要更多的 ROM 空间，通常认为至少需要 256KB ROM。这些配置不包括常驻 ROM 的应用代码。

驻留 ROM 的应用通常运行很慢，由于 16bit 或者 8bit（更加常见）的数据宽度，降低访问速度等。VxWorks 通过将引导 ROM 中的内容复制到 RAM 中从而避免了这个问题。

关于创建一个常驻 ROM 的 VxWorks 映像的更多内容，请参见《VxWorks 程序员指南》：配置和《Tornado 用户指南》：交叉开发部分中对于在 ROM 中执行 VxWorks 的讨论。

配置宏 ROM_TEXT_ADRS 和 ROM_SIZE 控制了 ROM 内存空间的实际使用。更多的有关内容，请参见“2.2.1 BSP 中所包括的文件”。

宏 ROM_WARM_ADRS 是 sysToMonitor() 程序使用的热启动入口点，它通常被定义为一个不变的偏移量，在冷启动入口点 ROM_TEXT_ADRS 上面通常为 8 字节。

4.3.3 以太网 RAM

一些应用程序设计者已经选择为基于 DMA 的以太网芯片提供专门的内存池，从而使 CPU 访问自己的内存不会被即将到来的以太网数据包所阻碍，这样就确保了实时响应。

如此专门的内存池被相关的 VxWorks 以太网驱动所支持。这些内存池应该完全支持 32 位的数据访问，否则网络的表现会受到严重的影响。

可以参考用户手册中相关的驱动程序代码，来获得关于配置专用 DMA 内存池的信息。

4.3.4 NVRAM

VxWorks 可以使用 255 字节的非易失性存储器 (NVRAM)，用来存储引导行信息，如果没有 NVRAM，正确的引导行就必须烧写进引导 ROM 中，或者在开发过程中的每一次重启后重新输入。

NVRAM 可以在电池支持的静态 RAM，EEPROM，或者其他技术中实现，很多目标板使用 Mostek MK48T02，它除了具有 2024 字节由电池支持的静态 RAM 之外，还包括了日历时钟（也是可选择的）。

参考有关合适的内存驱动的内容，来获得更多的关于配置一个特殊的驱动的知识。几乎所有的驱动程序都使用了配置宏 NV_RAM_SIZE, NV_BOOT_OFFSET, 和 NV_RAM_ADRS。这些宏通常在文件 config.h, bspname.h, 或者 configAll.h 中定义。更多的内容参见“2.2.1 BSP 中所包括的文件”。

4.3.5 奇偶校验

VxWorks 不在 RAM 中直接使用内存奇偶校验，如果需要奇偶校验，通常由 BSP 或者由用户来启用奇偶校验并执行奇偶校验错误处理程序。一些体系结构会指定一个中断向量

或者异常事件向量，用来处理奇偶校验中的错误。

4.3.6 地址分配

VxWorks 本身的地址分配并不重要，然而，工具链可能不会支持存储器中代码和数据的复杂分配。

地址分配中的关键性地址是：RAM_HIGH_ADRS, RAM_LOW_ADRS, 和 ROM_TEXT_ADRS（参见上面讲述的 RAM 配置宏）。

4.3.7 总线

这一部分描述了在考虑到下述总线类型时需要关注的问题：

- VME 总线；
- 多总线 II；
- PCI、cPCI 和 PMC；
- 无总线（busless）。

1. VME 总线

这一部分讨论了使用 VME 总线的 BSP 开发者需要关注的问题。

（1）VME 规范 C.1

VME 的协同工作能力是这个标准成功的关键所在。总线的特殊扩展应该限制在 P2 的 A、C 用户插脚范围内，而且应该被清楚地说明。在任何情况下，都不允许偏离规定的定时要求。

VME-64 规范是对早期规范的扩充，在这个时候 Wind River 公司并没有提供在 VME-64 规范或其扩充范围内的对即插即用特性的支持。

（2）地址

在 VxWorks 中地址分配的选择不是一个很重要的问题，本地地址可以使部分 VME 总线地址空间变得模糊。一些目标板不能对内存总线的低地址进行寻址，因为它们的本地地址从 0 开始。对于 VxWorks 这不是一个问题，因为所有的 VME 设备驱动程序都是可以配置的。然而，设备冲突可能是一个系统问题。

（3）VME 总线访问的动态总线规模

在这个规范中定义了 3 种地址类型：

- A16 短地址；
- A24 标准地址；
- A32 扩展地址。

除此之外，对于板外的设备通常有数据长度的限制，很多工具对相同的 VME 总线提

供了具有不同数据长度 (D16 或 D32) 的不同窗口。

将 D32 访问宽度自动变成两个 D16 访问宽度的窗口是非常有用的, 这可以通过使用一些具有动态总线大小的体系结构来实现 (如, 68K)。

一些目标板需要寄存器在初始化时, 除了要指定 AM (Address Modifier) 之外, 还要指定数据的“方向”(读/写), 这是非常不方便的。

参见 config.h 文件中有关为 VME 总线主/从访问窗口的标准的宏配置。

Wind River 并不直接支持 64 位的地址和数据, 然而, 这不会妨碍目标板的特定程序为用户提供这样的支持。

(4) 双端口内存

很多 CPU 目标板都有本地 (板载) RAM, 在 VME 总线中创建一个从属访问窗口可以使总线上的其他 CPU 和 DMA 设备使用本地内存, 这通常也被称为双端口。

这是由需要将本地处理器的 RAM 作为底板共享内存的系统决定的。这样的双端口内存应该完全支持下面将要介绍的 RMW 周期。

板外设备也需要双端口, 一般可以对目标板内存进行直接存储器存取, 例如 Excelan EXOS-302。

在存储器的外部地址上使本地处理器看见上端口内存将是非常有用的, 尽管通常没有提供这样的支持 (而且也不被 VxWorks 使用)。

在下载过程中双端口存储器也是非常有用的, 它可以便利底板调试器的使用。

(5) RMW

RMW (read-modify-write) 必须以一种不可分割的方式提供。

目标板必须与 VME 规范中定义的 RMW 机制相一致, 就是说, 在一个周期的读和写阶段, 目标板必须保持地址选通电平为低。一个正常工作的双端口从板, 通过检查地址选通电平持续为低, 来保持 RMW 的两个端口不被分割。

不幸的是, 只有在用户读写一个相同的单一地址时, 才能够保持地址选通为低电平。更复杂的不可分割指令, 如 CAS, 涉及到多地址则不能使用它, 这样就没有用于双端口内存的方法了。由于 VxWorks 只使用 TAS, 因此并不会带来麻烦。由于这个原因, 一些厂家为 P2 总线增加了 Lock 脚。

然而, 该管脚并不是这个标准的一部分, 也因此不能很充分地支持这个机制, 对于多数目标板这并不是个问题。

缓存和监听不是 VME RMW 循环中要讨论的问题, 共享内存的目标板绝对不能引入缓存一致性问题。对于适当的 VME 从属访问, 它必须是无缓存的, 或者是被完全的监听能力所保护的。

对于一些 PowerPC 工具, 有必要使用总线优先级作为用于 VME RMW 操作的全局信号量, 当目标板既不能产生也不能响应 RMW 循环时, 使用总线作为全局信号量很奏效。任何不能使用 RMW 的目标板, 在读写循环完成时会判断优先级并维持 VME 总线。除此之外, 即使控制板能够正确的执行 RMW 循环, 存储共享目标代码的总线控制板也必须实现总线锁保护。这一模式已在 BSP 的 sysBusTas() 和 sysBusTasClear() 函数中实现了。

(6) 仲裁

目标板应该默认为总线请求级别为 3，并且如果支持可选择优先判断级别的话，还要提供跳线机制。能够选择总线释放的方式包括 RWD(release when done)、ROR(release on request)、RAT(release after timeout)，这通常是非常方便的。

多总线优先级对于在底板中有多个控制器的系统来说是非常重要的，使用循环调度法进行优先级判断，可以保证每个总线控制器能够及时访问总线。

如果同一层的控制器采用菊花链方式，则离总线很远的控制器将会变得非常“饥渴”。

(7) 仲裁/系统控制器

系统控制器的功能应该是可选择的并通过跳线来选择。

系统控制器不是由一个由软件设置的寄存器来激活的。对于软件来说，具有能够读取系统控制器状态（开/关）的能力是很有用的。

当一个本地的复位信号被激发时，总线系统控制器应该声明总线复位信号。

系统控制器不需要判断所有的总线优先级，但如果它只判断一个级别，那通常是第三级别。

系统控制器负责中断协处理器并且声明一个 BERR 信号，较好的工具允许这个暂停时间在最短 16 毫秒到无限长的时间内选择。系统控制器 LED 可用于判断用户优先级的逻辑状态。

(8) 邮箱中断

邮箱中断和位置监视拥有相似的机制，此机制可作为内部处理器的同步方法。

邮箱允许中断器传递一些信息（通常是一个短字节或是一个长字节）到接收器，而位置监视机制只能引起一个中断却不能传递数据。

当在底板的节点之间传递网络数据包时，VxWorks 使用这些机制作为同步的方法。没有它们的话，VxWorks 就必须依赖于效率较低的查询方法，那样将会降低底板网络的吞吐能力。目前使用一个邮箱或者位置监视，但是使用两个或以上会更加出色。

VxWorks 能够使用 VME 中断来驱动底板的驱动程序，这样比轮询的方式更可取，但是仍旧不如邮箱中断方式好。

事实上，在使用邮箱中断时 VxWorks 并没有传递任何信息，而只是用了它的中断能力。

(9) VME 总线中断

尽管 VxWorks 并不要求 VME 中断，但是对所有的 VME 总线中断均给与支持是一个较好的想法，尤其是在需要板外以太网的功能时。通过跳线来允许或禁止这些中断是可以实现的，但是我们推荐使用软件控制的方法，允许软件读取中断的状态是很有价值的。

(10) VME 总线中断确认

VME 总线中断请求必须得到接收器的确认。一些实施者选择迫使 ISR 来回应中断的方法，但是更加规范的首选方法是使硬件自动确认中断请求，并给 CPU 提出正确的中断向量。

我们并不推荐使用软件中断确认的方法，因为那样会严重影响系统的性能。

VME 中断产生能力也是非常必要的，尤其是如果板卡与那些没有邮箱中断的板卡一起用于底层网络中时，则更是如此。其中一个最重要的例子是在底板上安装使用 Sun 操作系

统的 Sun 目标板时，如果从板没有提供 VME 中断产生的能力，那么在 Sun 操作系统这方面底板的驱动程序需要进行轮询。

(11) 电源使用

VME 总线标准规定了每伏电压的最大能源消耗。设计者必须遵守这些限制，或者清楚地标明额外的功率需求。

每一块板卡典型的功率消耗是 7 瓦特，那些需要更多能源消耗的板卡需要特别标明。

(12) 提取器

VME 板应该将板卡提取器安装在前板上。

(13) VLSI 芯片

很多现有的 VLSI 芯片都提供了全面的 VME 总线接口功能。

2. 多总线 II

这一部分讨论了使用 Multibus II 总线的开发者需要关注的一些问题。

(1) Multibus II 规范

Multibus II 实际上是一个总线标准的集合：

- IPSB，一种用于板卡之间通信的消息传送总线，它使用地区地址。
- ILBX，内存扩展总线。
- ISBX，对于可以插入到 Multibus II 板卡的模块定义。
- Multibus II，传输协议规范 (Inter)。
- Multibus II，互连接口规范 (Inter)。

在总线上有 4 个地址空间：内存、I/O、消息和互连。

(2) 配置

Multibus II 板卡不使用 DIP 开关或跳线来进行配置，配置选项可以通过访问目标板的多总线 II 的互连空间来设置。典型的选择是存储器的开始和结束地址。

(3) 硬件中断

没有使用硬件总线中断，取而代之的是使用消息传达信息（如设备就绪状态）或实际的 I/O 数据。信令传递提供了一种可靠的并且易于扩充的，用于在 Multibus II 代理器之间传递信息的方法。所有代理器之间的底层通信都能够通过消息来处理。不需要主/从内存接口——事实上，内存接口并不适用于板间的通信，因为消息含有数据，一个 I/O 数据块可以像状态或注意信息一样在一个相同的消息中被传输。在总线中不会有内存地址冲突产生，而且也不会有总线内存缓冲的问题，因为消息是在代理器之间传递，而不是在内存地址之间。

(4) 处理器序号

多总线板卡的插槽是相互独立的。在系统初始化时，系统控制器 (slot 0) 将插槽的 ID 号传送给每一块目标板，然后每一块板子上的初始化程序就会设置消息 ID。

(5) 初始化

在初始化过程中，所有的目标板都能够使用互连空间为总线上的特定目标板定位。例

如，每一个设备驱动程序都能够在总线中搜索它能够识别并能够控制的板卡。这个方法可以在运行时决定板卡的类型和位置以及系统的配置。在互连记录中包含的互连寄存器和数据是由 Intel 来定义的。遵守 Intel 规范的生产商称自己为（通过互连数据）服从规范的，我们也有可能使用不服从规范的目标板，这样的板子称自己为不服从规范的。

（6）消息传送协议

Intel 已经定义了一个用于传送消息的传输协议，这个协议被很多目标板生产商使用。这就很容易增加一个新设备，因为设备之间的基本通信可以使用已存在的消息传送程序和约定。一旦初始的消息传送程序写好之后，在增加一个新设备时，通常就不需要再编写针对特定硬件的代码。

（7）设备驱动程序

BSP 通常包括一个为 MPC 芯片编写的硬件设备驱动程序和一个传输软件模块。所有 VxWorks 的 I/O 设备驱动程序传送或接受的消息都是通过使用传输层（如果需要的话，可以直接使用 MPC 驱动器）在软件层次上进行通信的。

传输层和 MPC 驱动器负责消息排队，并将响应回馈给适当的设备驱动器。可以同时控制多个设备，因为所有的消息都包括源地址和目标地址，还包括序列号等，这都保证了消息的正确传输。

（8）PCI, cPCI 和 PMC

Wind River 提供的驱动程序满足 PCI 2.1 修订版规范的要求。这些驱动程序为人工或自动的进行设备的配置和列举提供了支持，同时还提供了一个中断库模块，用于处理 PCI 中断元的超负荷情形。可以参考 pciConfigLib, pciConfigShow, pciAutoCfg 和 pciIntLib 的入口参量来获取更多的内容，Wind River Technical Note #49 中也包含了有关 PCI 支持的信息。

3. 无总线

当需要将 VxWorks 装载到无总线服务器中的时候，Wind River 强烈推荐使用 NetROM 或者电路仿真器。仿真器目标文件的格式和操作就像它们的销售商不同一样，也是不同的。请联系 Wind River 以便获得适当的结合信息。

最初的目标是打通串行通信口，启动一个最小系统，编译生成一个轮询的串口驱动程序，并且从这里开始扩展。要充分利用 LED，当代码中的检测点通过这些 LED 的时候它们会闪烁。可以使用示波器来观测中断的产生以及得到正确的确认。

参见第 3 章，以便获得更多的有关用于一个新建 BSP 初始化启动策略的信息。

4.4 设 备

设备应该同时支持读写访问，对软件来说，保存寄存器或者其他写入设备的数据备份开销很大而且易于出错。有强制访问时间要求的设备不应该寄希望于使用软件在两次访问

间增加延迟（易于出错并且不便于移植），而是应该自动的将下一个访问挂起。

在这一部分的其他内容中，我们将讨论关于可能在实时系统中使用设备的更具体的问题。

1. 中断

在任何实时系统中，中断都是最需要注意的事项，中断延迟将会对系统的设计产生重要的影响，VxWorks 的中断处理器机制是为在中断事件和中断处理程序的执行之间提供一个最小化接口而设计的。最小的响应时间已经在基准程序中公布。

如果硬件中断电路的设计使中断处理变得非常笨重并且效率很低，则所有这些努力都将付诸东流。这也是某些硬件不适用于实时系统设计的原因之一。

另一个重要的事实是经常有意地编写一段设备驱动程序使在一个短暂的事件段内禁止中断，这样做是为了保护共享的数据变量（在任务级别代码和中断处理器之间）不遭到破坏。理想情况下，驱动器只能禁止在驱动器控制下的设备产生的中断，然而，由于一些硬件具有限制性的设计，驱动器必须在某些时候禁止目标板中的所有中断。这不是人们所希望的，因为在这个被锁住的时期内不能执行任何中断服务程序。

一些设备有能力在中断确认总线周期内提供中断向量，除此之外，还有一些设备可以提供修改过的向量，向量的某些位做了相应的变化来指明设备产生中断的确切原因。这是人们想要的特性，因为它使得驱动中适当的处理程序运行得更快，比只提供一个共享中断处理程序运行得更快（一段单一的处理程序需要不断的轮询设备，来决定是哪一个引起了中断）。

很多硬件设计都包含了一类专门的设备：中断控制器。这些设备可以使目标板的所有中断源都参与进来，并且可以通过编程为每一个中断源来选择中断参数。这些参数是：中断产生的优先级，中断产生的向量和中断源是被允许还是被禁止。一些中断控制器能够通过编程，为设置为相同优先级的中断源产生的中断提供循环优先级判断法。

下面是一些中断电路设计的方法指导：

（1）选择的设备应该能够提供中断向量，并且根据特定事件修改向量。如果不可能的话，那么选择只提供一个单一向量的设备。

（2）中断控制器设备具有非常好的特征，它们可以替代或者扩充（1）中提到的需求。目前在市场上有一些 Wind River 支持的程序。若用户自行编写，则设备一定要明确记录。

（3）板上的每一个中断源都应该具有单独地控制允许/禁止中断信号传到处理器的能力。因此驱动器从不需要改变 CPU 的中断级别，中断控制器设备通常提供了这一特性，还可以使用一个简单的控制寄存器来阻止某一设备产生中断信号。多数设备在可编程的寄存器中都包含一个中断允许/禁止位，然而，事实上并不总是这样：Wind River 已经发现了一些设备不具有禁止使用中断信号的机制。

（4）在硬件复位或是加电启动之后，所有的中断源都必须处于未声明状态。这一条件通常只是一个良好的习惯，可是在 VxWorks 系统中这样做尤其重要。这是由 VxWorks 在不同的功能的模块分层组合方式所决定的，如果一个设备在复位之后正在声明它的中断信号，

那么在内核初始化时，就必须完成对这个设备或其他电路的初始化工作，以促成设备取消声明它的中断信号。完成这个工作所需要的这一类特定设备的内容通常包含在设备驱动程序中，不要在内核初始化程序中复制这个顺序。

对于中断设计的其他必要条件只剩下用图表和文件适当表示中断向量的规划，即使是设计较灵活的末端开放系统，文件也应包括目标板设备的默认向量和优先级（如果可以选择的话）。

一些读者可能会觉得在讨论中断时用到的术语比较容易混淆。中断级别是指一个中断输入的优先级。优先级级别是设备处理器中断的手段。中断向量表是用于鉴别响应一个中断的正确中断服务程序的 ID 号。驱动程序将中断服务程序与中断向量表绑定，驱动程序允许和禁止中断级别，同时还要注意，在设备初始化时驱动程序允许中断级别是正常的，但是在断开与设备的连接时禁止中断级别是不可以的，这是因为其他的设备可能正在共享这个相同的中断级别。

Wind River Technical Note #46 中讨论了用于某些体系结构的标准中断控制器设备的创建方法。请参考这本技术手册来获取进一步的知识。

2. 系统时钟

系统时钟中断是强制性的，VxWorks 的系统时钟要求中断频率在 30Hz 至 2kHz 之间。

默认值是 60Hz，并且要求精确地产生这个频率，依赖波特率产生器常常达不到这个精确度要求。

3. 辅助时钟

VxWorks 使用一个从 30Hz 到 2kHz 的辅助时钟来剖析 CPU 的利用情况，这是监测功能所需要的，具有一个工作在更高频率(10MHz)的 24 位（或者更大）计数器，对于剖析系统性能是非常有用的。

4. 时标时钟

很多在 `target/src/drv/timer` 中的普通的定时器都包含了时标功能，时标驱动程序为时间的测量提供了高分辨率的工具，通常被 WindView 产品使用。参见《WindView 用户指南》：创建 VxWorks 时标驱动，来获取关于 API 的更多知识。

5. 串行端口

VxWorks 目前需要一个 RS-232 串行端口，它在开发和调试阶段被使用。设备应该可以通过配置来选择工作在不同的标准波特率下，推荐使用 9600。标准的通信参数有 8 个数据位，1 个停止位，无奇偶校验位，没有调制解调器控制线路。可以在没有串行接口的情况下引导启动 VxWorks，而且在嵌入式系统的产品中没有串行接口也是可能的。

首先应该考虑的是目标服务器目标板应为数据通信设备，因此，RS-232 连接器应该是一对一连接的，而不是空调制解调器（Null-Modem）的（针脚 2 和针脚 3 相交叉）。这样考虑到了与一标准终端的直接连接。

VxWorks 支持软件流控制, 忽略硬件流控制信号, 因此, 在工作过程中只需要 3 条线路 (传送、接收和信号地)。

关于 SIO 类型串行驱动程序的 API 函数已经扩展为包括对硬件流控制和改变每一个串行设备工作参数的支持。可以通过改变每一个设备的硬件选项标志, 来实现改变数据位、数据奇偶校验位、停止位的个数, 并且选择启用或是禁用硬件流控制。默认的设置通常使用 8 个数据位, 1 个停止位, 无奇偶校验位, 无硬件流控制。用于改变硬件选项的 `ioctl` 代码是 `SIO_HW_OPTS_SET (0x1005)`。 `SIO_HW_OPTS_GET (0x1006)` 代码能够用来读取当前的硬件选项。在 `target/h/sioLib.h` 文件中可以看到选项标志位的值。不是所有的驱动程序都已经升级到能够响应这些新的 `ioctl` 命令, 目前只有 Zilog 8530、Motorola MC68681 和 Intel 8250 可以完全支持这个特性。

另外, 串口是由硬件提供的, 并被应用程序使用, 而不是 VxWorks 所必需的。如果有多个串行口存在, 他们的波特率应该分别单独设置。

6. 以太网控制器

VxWorks 是非常“面向网络”的, 至少在开发阶段是这样的, 因此具有一个有效的网络接口是非常必要的。这个接口可以用于引导和下载应用程序代码, 同时还包括具体应用的内部处理器之间的通信。VxWorks 通过 `netLib` 为以太网控制器提供了一个独立于设备的接口, 它允许使用 RPC 和 NFS 以及其他的网络协议。

在市场上可以见到两种基本类型的以太网设备: 一种是与 CPU 和其他的总线设备共享内存, 另一种是保留一个为 CPU 不可见的私有包缓冲区。

与 CPU 共享内存的设备是通过 DMA 总线周期实现的, 这意味着在传送和接收数据时, 设备分享了总线上的一段时间, 这会为应用带来无法预测的影响。

保留有私有包缓冲区的设备通常需要 CPU 的协助来完成将数据包写入或读出私有缓冲区的工作, 这一般是以字节或字的形式移入移出设备寄存器的方式来完成。这种模式能够提供更好的行为预见性, 因为数据包的移动情况完全是在 CPU 的控制之下的。

对于共享内存这一类设备还有另一种分类方式: 一类设备只能处理连续的内存, 另一类设备可以处理内存碎片。对于 VxWorks 一般建议使用能够处理内存碎片的设备, 因为网络接口通过协议模块交换 `mbuf`, 而 `mbuf` 基本上都是内存碎片。如果设备只能处理连续的内存, 那么驱动器就要在 `mbuf` 和这些连续的内存之间复制数据包, 这样将会影响系统的性能。

如果使用共享内存这类设备, 选择一种没有任何地址段或内存段限制的设备是很有利的, 这将使得驱动程序更加简单、有效以及通用。

如果数据缓存是所选择处理器的一个问题, 则最好以太网控制 / DMA 设备和 CPU 具有硬件一致性支持, 例如 MC68040 和 Intel82596, 或者如果它使用的以太网设备和存储器能够被标识不进行缓存。如果不能得到硬件支持, 那么驱动程序就必须考虑缓存一致性问题, 需要在适当的时间执行缓存无效或者缓存刷新操作。这就使得驱动程序更加复杂并且不具有一般性。

设计一个包含以太网芯片的 CPU 目标板可以在一个“全总线”环境下节省开发额外板外网络硬件的费用，并且具有潜在的更高性能。

在附录 H 中有关于编写 VxWorks END 网络设备驱动程序的详细的介绍。

7. SCSI 控制器

SCSI (Small Computer Systems Interface) 控制器能够用于控制硬盘、软盘、磁带驱动器。这些设备可以用于本地的引导启动以及应用程序所需要的数据存储。VxWorks 通过 scsiLib 为 SCSI 控制器提供了一个独立于设备的接口，同时它允许使用 MS-DOS 和 RT-11 等相互兼容的文件系统以及“低级别”原始文件系统。

并不需要同时使用 SCSI 控制器与内部或外部 DMA，但是 DMA 的使用会大大提高驱动器的性能。适用于以太网控制器解决缓存一致性和地址问题的方法同样适用于 SCSI 控制器。

选用的 SCSI 控制器应该能够支持应用所需要的 SCSI 目标服务器类型，这一点是非常重要的。如果高级的 SCSI 特性是重要的，那么 SCSI 控制器就必须能够提供它们所需要的特性。

Wind River 不再继续支持他们早期的 SCSI-1 产品，SCSI-1 库和驱动程序仍旧存在于过渡产品中，但是它们不会包含在未来的新产品中，将不会继续提供以技术咨询和修正为形式的支持，完全的支持仅限于 SCSI-2 库和驱动程序。

附录 I 包含有如何编写 VxWorks SCSI 设备驱动程序的详细的介绍。

8. DMA 控制器

DMA 控制器可以释放处理器向 I/O 设备或者板外的内存中复制的冗长数据，并且可被 SCSI 设备驱动程序使用。

9. 复位按钮

应该设计一个复位按钮，它在功能上与接通电源复位是等价的。如果是在全总线环境中运行，复位信号应该通知给总线。

10. 放弃按钮

如果能够产生一个非屏蔽中断 (NMI)，这将使得能够从一个中断处理器中重新获得对系统的控制权，而不需要重新启动整个系统。

11. DIP 开关

对于选择地址和中断，使用 DIP 开关或者旋转式开关比使用跳线方式更加方便，尤其是在配置过程中，易于被软件读取的 DIP 开关更具有优势。

总体上讲，建议除了启用系统控制器之外的所有功能都应该能够被软件设置。

12. 用户 LED

LED 是非常有用的调试工具，特别是以无总线为目标的时候。这样可以快速地通过视

觉来检查目标板以核实它的功能。使用一排 LED，例如代码错误这样的信息就会显示出来。（LED 应该被放置在容易看到的地方，例如面板的正面。）

13. 并行端口

VxWorks 对并行端口提供了简单的支持，有关支持的设备和功能，请参考 `target/src/drv/parallel` 目录。

4.5 启用虚拟内存库

在进行这一部分学习之前，先回顾一下 `vmBaseLib` 和 `vmLib` 访问入口，以及《VxWorks 程序员指南》：虚拟内存接口。

`VmBaseLib`（基本虚拟内存支持库）和 `vmLib`（VxVMI 选项）具有相似的对 BSP 的要求。因为 `vmBaseLib` 与 VxWorks 是绑定在一起的，所以在编写 BSP 时默认包含它。如果安装了 VxVMI 选项，则终端用户在将来的编译中只需要在 `target/config/bspname/config.h` 文件中将 `INCLUDE_MMU_BASIC` 改为 `INCLUDE_MMU_FULL`，这样就可以将未绑定的库包括进来。

这一部分的其他内容还介绍了在 `sysLib.c` 文件和 `config.h` 文件中必须要做的一些改动，以适应所有支持的使用 MMUs 的目标服务器，以及为 SPARC 目标服务器所必须做的其他的改动。

 **警告：** 由于目前 VxWorks 工具的一些限制，当使用 68K 的 MMU 时，虚拟地址必须与物理地址相同。

1. 修改 `sysLib.c` 文件

虚拟内存支持要有两个必要条件：

- 增加物理内存描述。这需要包括一个头文件并且定义两个变量，以例 4-1 中的代码为例开始，并对它作适当的修改。
 - 本地 RAM 是合法的，可以写入，也可以进行缓存。ROM 也是合法的，只读，并且通常不能进行缓存。I/O 设备是合法的，可以写入，不能进行缓存。如果条件允许，VME 总线域也是合法的，可以写入，通常不能进行缓存。如果能够从 CPU 被直接地写入，闪存一般是可以写入的但不能进行缓存。
 - 地址和长度参数必须是 `VM_PAGE_SIZE` 的倍数，在 `configAll.h` 文件中定义为 8192。
 - 由于每一页面映射需要一个页面列表入口存储在物理内存中，我们需要一个与寻址空间上限有关的可分配的物理存储空间。不要为 VME 总线空间分配大量空间。为安全起见，每个记录不要分配超过 32MB。

- 当虚拟内存支持被激活后（见下），CPU 访问超出了物理内存时导致总线错误。除非被包含在 `sysPhysMemDesc[]` 之中，内存映射设备(以太网，SCSI，串行设备及其他)不可访问。

■ 在 `sysToMonitor()` 的开头添加下面内容：

```
VM_ENABLE (FALSE); /* disable MMU */
```

例 4-1：为支持虚拟内存存在 `sysLib.c` 文件中增加的部分。

```
/* includes */
#include "private/vmLibP.h"
/* globals */
PHYS_MEM_DESC sysPhysMemDesc [] =
{
/* adrs and length parameters must be page-aligned */
/* RAM */
{
(void *) LOCAL_MEM_LOCAL_ADRS,
(void *) LOCAL_MEM_LOCAL_ADRS,
LOCAL_MEM_SIZE,
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
VM_STATE_MASK_CACHEABLE,
VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE
},
/* ROM */
{
(void *) ROM_BASE_ADRS,
(void *) ROM_BASE_ADRS,
ROM_SIZE,
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
VM_STATE_MASK_CACHEABLE,
VM_STATE_VALID | VM_STATE_WRITABLE_NOT | VM_STATE_CACHEABLE_NOT
}, /* a16 VME */
{
(void *) SHORT_IO_ADRS,
(void *) SHORT_IO_ADRS,
0x10000, /* 64 Kbytes */
VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
VM_STATE_MASK_CACHEABLE,
VM_STATE_VALID | VM_STATE_WRITABLE | VM_STATE_CACHEABLE_NOT
}
};
```

```
int sysPhysMemDescNumEnt = NELEMENTS (sysPhysMemDesc);
```

2. 修改 config.c 文件

定义 INCLUDE_MMU_BASIC 来实现对基本虚拟内存的支持。

3. SPARC 目标板的额外条件

在 SPARC BSP 中,使用文件 sysLib.c 中的两个全局变量 sysCacheLibInit 和 sysMmuLibInit 来选择合适的缓存和 MMU 库。下面这个代码段说明了这个机制:

```
/* Sun-4 cache library */
IMPORT mmuSun4LibInit( );
FUNCPTR sysCacheLibInit = (FUNCPTR) cacheSun4LibInit;
FUNCPTR sysMmuLibInit = (FUNCPTR) mmuSun4LibInit;
```

(1) Sun-4 MMU

Sun-4 MMU 的一页大小通常是 8KB, 与在 target/config/all/configAll.h 文件中对 VM_PAGE_SIZE 的定义相一致。一些硬件设备使用不同的页面大小, 对于这样的问题, 可以通过在 config/bspname/config.h 中添加下列#undef/#define 序列来覆盖掉 VM_PAGE_SIZE 的默认值 (这个例子假设页面大小为 4KB):

```
#undef VM_PAGE_SIZE
#define VM_PAGE_SIZE 4096
```

Sun-4 MMU 支持只有 8 个虚拟内存的环境, 另外虚拟地址空间只有 66MB 可在任何时间被映射 (为不同的文本所共享), 因为包含有转换表格的静态 RAM 的大小是有限的。

当决定 sysPhysMemDesc[] 的全局映射时, 保证虚拟内存容量不超过 66MB。同时注意到额外的虚拟内存空间可在运行时期分配, 作为 8 个文本中的个人内存。

(2) SPARC 参考 MMU

SPARC 参考 MMU 页面大小为 4KB。由于这个值与 config/all/configAll.h 文件中对 VM_PAGE_SIZE 设定的默认值不一致, 因此在 config/bspname/config.h 文件中增加下面的#undef/#define 序列:

```
#undef VM_PAGE_SIZE
#define VM_PAGE_SIZE 4096
```

由于在 vmLib 定义的 32 比特物理地址, 无法完全显示 SPARC 参考 MMU 模型的 36 比特物理地址。因此, 对于目前版本的 mbLib, 物理地址必定为物理页面号 (即把物理地址右移 PAGE_SHIFT 位, 或者 12 比特)。这会影响到 vmGlobalMap(), vmMap() 和

vmTranslate 的调用，也会影响 sysPhysMemDesc[] 的规定。对于上述调用（和规定）对物理地址移位 PAGE_SHIFT 中规定的位数。

为了支持多种文本，SPARC 参考 MMU 通过当前上下文号为四层页面表而不是三层建立索引。此表大小可变并由 BSP 引进的常量所决定，如下：

mmuInitNumContexts，列表文本的初始码（为 2 的方幂，符合物理页面要求）。

MmuMaxNumContexts，列表文本最大号码（不一定与 mmuInitNumContexts 不同）。

MmuContextTableAlign，列表文本的队列需求。

上述每一项都是由特定 SPARC 参考 MMU 芯片文件所决定的。BSP 设计者需限制 mmuMaxNumContexts() 来阻止列表过长（在执行过程中需要很多上下文条目）。

下面是在 sysLib.c 中对全局变量进行定义的一个例子：

```
/* export parameters of context table */
int mmuInitNumContexts = TI390_SPARC_CONTEXTS;
int mmuMaxNumContexts = TI390_SPARC_CONTEXTS;
int mmuContextTableAlign = PAGE_SIZE;
```

第 5 章 驱动程序指导

5.1 绪 论

在本章描述了有关编写设备驱动程序的一般性问题。关于特殊设备驱动程序编写的其他问题，可以在 Wind River 技术注释中找到，在下列附录中也有这样的内容：

- 附录 F、通用设备驱动程序（串口、SIO、内存、定时器等）；
- 附录 G、升级 4.3BSD 网络驱动程序；
- 附录 H、执行基于 MUX 的网络接口驱动程序；
- 附录 I、编写 SCSI-2 设备驱动程序。

与 BSP 包一起发布的软件中包含了每一类型驱动程序的模板，这些模板提供了属于某一类型的几乎所有驱动程序所共用的逻辑框，它们还解释了一个特殊的驱动程序是如何与它的 VxWorks 系统本身相互作用的。

本章还包含了在 DMA 驱动器设计中，对于缓存器需要考虑的一些问题的一般性讨论。然而，关于在特定体系结构下缓存工作的具体细节问题，可参见 VxWorks 体系结构增刊文献中的适当内容。

5.2 设计目标

任何设计计划书都必须从陈述目标和目的开始，第一个目标就是实时性，其他的目标包括灵活性、可维护性、可读性、可配置性。

1. 性能上的设计

驱动程序必须表现出良好的性能以便与实时内核能力相匹配。性能上的设计包括了很多内容，它当然也意味着采用一种有效的方式使用 DMA 和中断。在编码方面，它意味着将子程序嵌套在最适宜的层次上，过多的子程序调用和存储操作会降低系统的性能。我们必须在正确使用子程序来保持代码量较少和保证设计的简单并易于理解这两个问题上找到一个平衡点。

性能上的设计还意味着保持中断延迟为最小，在任何设计中，中断处理器都必须受到最多的关注。整个系统的性能表现与某一特定驱动器的性能是同样重要的。

2. 代码的灵活性和可移植性

代码的灵活性与适用于新板的配置有关。这里重要的原理是结构化的设计风格和对于所有硬件访问使用的宏。灵活性包括两个方面：运行时的灵活性和编译时的灵活性。为了使用指针访问目标模块程序来获得我们所要的灵活性，通常要牺牲一些实时性能。运行时的灵活性通常也被称为可移植性。编译时的灵活性是指使用预处理器宏在编译时定制系统来获得较好的性能。

Wind River 建议无论在何种情况下都尽量使用这两种方法，这为编译过的目标模块提供了所需要的可移植性，并且允许使用一系列不同的宏来编译相同的源代码以产生一个优化的模块。最好的方法是使用编译时间宏来执行实时向量程序。这样可以实现两个目标，正常编译产生的目标模块在运行时是可定制的以实现灵活性，而且，相同的代码可以与重新定义的编译时的宏一起使用来创建一个性能更加优化的模块。

3. 可维护性和易读性

大多数代码工作都是需要维护的，因此，任何为使维护工作更加简单而做的努力都是值得的。严格遵守编码规则和品质文件使编码易于阅读、理解和维护。应特别关注为什么事件会发生。低品质文件如同不完全的文件。所有新编写的文件都应该让至少一个其他人审查。

4. 配置的简易性

驱动程序不能限制终端用户的选择和需求，不要强加限制所支持的设备数目或其他特征。可能不能支持一个设备的所有特征和模式，但是在设计中必须考虑将来要包括这些特性。

5. 性能测试

必须对所有驱动程序的性能进行测试。除了编写驱动程序之外，设计工程师还必须考虑测试程序。这包括插入调试消息，也包括基准测试。如果不可能进行标准基准测试，那么工程师就要考虑自己编写一个。性能测试应该考虑到所有的设备驱动程序、以太网、SCSI、串口、定时器、中断控制器等的类型。

6. 代码量

在嵌入式 RTOS 市场中，代码量是很重要的。应该通过很好的结构化设计来尽量的减小代码量。减小代码量将会影响性能，设计工程师必须在设计中权衡这一问题，在尽量提高系统性能的同时不要使代码过长。

7. 重入机制

驱动程序应该是完全可以重入的，这样才可以支持任意数量的设备。限制了所支持设备数目的驱动程序不是我们想要的。应该为每一个新设备创建一个结构并且将它传送给驱动程序来进行初始化和控制，这样代替设备信息固定的排列方式。换句话说，作为设备初

始化调用的一部分，驱动程序可以使用 `malloc()` 为每一个设备分配相应的结构。访问全局的数据结构必须被一些同步方式所保护，常用的同步方法包括：`intLock()`、`taskLock()`、或者互斥信号量。

```
#define NUM_LN 2
LN_CTRL ln_softc[NUM_LN]; /* BAD IDEA */
```

5.3 设计的问题

这一部分讨论了与各种各样硬件设计内存映射芯片、I/O 映射芯片、多功能芯片、多总线和中断控制器有关的设计问题。

1. 各种类型的硬件设计

硬件设计的种类可以说是无限多的，不能随意创造硬件连接方式。设计者应继续使用软件设计全系统的方法。

在理想的情况下，第一次遇到一个新的芯片将会使你学到将来遇到的所有芯片所应该了解的很多知识，但是经验与理想中的是完全不同的。在技术成熟之后，最初运行的硬件和软件驱动程序一定会需要全面的修改。

2. 存储器映射芯片

在存储器映射系统中，设计者需要选择如何将芯片寄存器映射到内存空间的方式。使用相同芯片的系统能够以完全不同的方式对设备的 I/O 寄存器进行映射。

考虑一个拥有 4 字节宽度寄存器的简单串口芯片，一些设计者可能会将它们映射为 4 个连续字节的地址，其他一些使用 16 位内存系统的设计者可能会仅仅使用 16 位内存总线的低 8 位来映射每一个寄存器。在另一种系统中，寄存器可以被映射为长字节地址。因此对于三种不同的寻址机制有三种不同执行方法。

还要考虑到选择长字节设备的设计者可能也会选择要求驱动程序仅仅使用长字节进行读写访问。驱动程序甚至不能使用一个字节读操作来读取一个字节寄存器中的数据。

解决这个问题的一个方法是把所有对该芯片的访问限制到尽可能少，更进一步，要求每一个对该芯片的访问都要在预处理程序宏中进行声明，这个宏可以被重新定义，以便适应任何特定系统的特殊需要。

3. I/O 映射芯片

大多数工程师的工作经验都是使用 Motorola 处理器的，这些处理器使用专门的内存映射。然而，还有另一大类的处理器使用独立的被称为 I/O 空间的地址空间。不幸的是，C

语言没有提供一种用来指明一个地址是内存地址还是 I/O 地址的方法。必须使用汇编语言子程序利用特殊的指令来访问 I/O 空间中的数据。无法在 C 程序中直接编写一段代码来完成上述工作。

值得庆幸的是，用来解决上述的存储映像问题的解决工程同样也可以解决这个问题。通过定义专门的硬件访问宏来使用一个特定的 I/O 子程序，可以操作映射到 I/O 空间的设备。

4. 多功能芯片

设计者喜欢在一块芯片中包含更多的通道（设备），于是他们想到了一个使用 ASIC 芯片的方法，就是在一块单独的硅片上结合了不同类型的多个设备。设计者仅仅被将众多的硅块结合到一起的方法所限制。

应该尽量像每一个部分都是独立设备一样来编写驱动程序，而不是直接对一个完整的 ASIC 编写驱动程序。一个驱动程序需要另一个驱动程序的支持也可能是必须的。在这种情况下，需要在所来的驱动文件中予以明确说明。用于整个 ASIC 的单一大型单片电路与系统的可伸缩性相违背，应该能够排除应用中所不需要的特性。

5. 多总线

CPU 本地总线、VME 总线、VXI、QBus、SBus、PCI 本地总线：到底有尽头吗？计算机正在变得越来越复杂，并且已经进入了嵌入式计算机领域。一些总线定义了其自身的字节顺序，而且与本地 CPU 字节顺序不同。需要在最小的配置变化前提下使自己的驱动程序从而可以工作在任何总线上。幸运的是，为了实现灵活性而定义的硬件访问宏又一次帮助了我们。定义宏来执行任何字节交换或其他与总线有关的操作是很简单的。

在设置系统日期方面最复杂的系统是 Motorola MVME-1600 目标板。本地总线是 PowerPC MPC 总线。MPC105 芯片将 MPC 总线与 PCI 总线连接，PCI 芯片与 ISA 总线连接，另一不同的 PCI 芯片将 PCI 总线与 Motorola 68040 总线连接，从 68040 总线，有一 VME 芯片连接 VME 总线。一块目标板，五种不同的总线。VME 总线支持 7 个中断请求级别和 256 个可能的向量，PCI 总线支持 4 个中断级别，ISA 总线支持 16 个中断级别。从定义上来看 PCI 为 little-endian，040 总线为 big-endian。MPC 总线为双工的（任何一端均可作为终点）。

6. 中断控制器

一些硬件设计需要专门的中断确认这一步骤，有些时候中断是通过中断控制器的层叠链来实现的，这种控制器需要配合每个中断执行多个步骤来复位和清除中断控制硬件。

在 PCI 总线设备的高峰时，必须确保所有中断服务程序都与中断链条一致，这意味着驱动程序代码必须能够判断是否一个设备事实正在发出一个中断请求，如果不是的话，那么代码必须立刻退出。

在理论上，设备驱动程序不需要过多关注连接中断向量，这明显属于 BSP 的职责范围。驱动程序的中断服务程序应该是全局的程序，作为 sysHwInit2() 的一部分 BSP 连接到中断向量。若在一个中断连接步骤中必须包括一个驱动程序，则需要通过一个硬件抽象宏。这

需要一个灵活接口以便不同中断结构的不同板卡可以由一个驱动程序来支持。

5.4 设计指导

这一部分讨论了在设计与 Wind River 公司产品一起使用的驱动程序时，应该遵守的一些设计要求。包括对下属问题的探讨：函数命名规则、文件规范标准、每一个驱动程序和每一个设备的数据结构、中断服务程序以及用于访问芯片寄存器的宏。

1. 命名及位置

要遵循 Wind River 驱动程序命名规则，必须十分注意程序和变量的名字。每一个模块都应该有一个与众不同的前缀，并且每一个程序和变量的命名都必须以这个前缀开始。编码规则建议名字的结构使用模块——名词——动词的方式。也可以认为它是从一般到具体、从左到右的规律。相关的程序必须具有一个根，并由它们的后缀来相互之间区别。

一个简单的例子：

```
STATUS fooStartPoll (void);  
STATUS fooStopPoll (Void);
```

应该命名为（假设它们都是与轮询操作有关的）：

```
STATUS fooPollStart (void);  
STATUS fooPollStop (void);
```

只有 Wind River 的通用驱动程序被存储在 target/src/drv/xxx 中。这说明了所有的驱动程序被一个或更多的 BSP 使用，或者是希望被更多的 BSP 使用。第三方的 BSP 编写者常常认为这些目录是用于所有的驱动程序的，也包括他们自己的。由于未来的不可预见性，Wind River 保留了关于这些目录的所有权利。第三方使用者必须将他们的驱动程序放在一个相同的目录中，就像 BSP。

即使是 Wind River 专用的驱动程序也被放在 BSP 目录中，它们通常是通用驱动程序经过修改后的版本。有些情况下它们只是封装了对一组特定宏的声明并且#include 通用驱动程序。

2. 文件和标准

在特性和操作模式方面描述整个芯片。代码维护人员需要在不求助于全面的库搜索的前提下，了解芯片的基本情况。大多数生产商现在都将它们的资料放在网络上，这就使得查找这些资料变得更加容易，但是如果有一段较好的关于某个芯片的概括性资料，便可以省去大量的查找和阅读时间。

说明驱动程序的模式和限制，模块说明应该是对芯片和驱动程序两者的初步介绍。

遵守 Wind River 的编码标准，这样可以提高程序的可读性并易于维护。让一名专业的程序员来检查文件的明确性和语法。

做一次正式的代码检查，核实是否符合编码标准。没有比让其他人来查看你的代码更好的办法来确保代码的质量和可读性的了。

关注设备和驱动程序共同工作的文档。除了丢失的文件，还存在无用文件的问题。过于琐细和冗余的文件是并不少见的。一个典型的例子是：

```
x = 0; /* clear x */  
fooReset (&myFoo); /* reset myFoo */
```

这是一个很普通的例子。汇编语言编程用来记录每行代码。在 C 语言时代，程序员习惯于在代码之前加上一段注释，每行程序都加注释产生了很多无用文件。程序编写者需要尽量避免每行程序都加注释的问题。

另外一个需要注意的问题是，手册中的文件往往比代码段中的注释重要得多。这些是给用户的信息，它必须是准确的并且清楚的，如果做得好，读者就会明白代码将要完成的任务。在用户清楚地理解了它的目标之后，就非常容易发现程序中的问题。

文献资料将会告诉工程人员如何将驱动程序合成为一个完整的 BSP 包。应该记录所有特殊外部程序。应该包括有关驱动程序初始化的介绍。参考 BSP 合成代码的实例是非常有帮助的。

3. 每一设备的数据结构

作为面向对象设计的一部分，每一个设备在系统中应该由一个单独的结构来表示，这个结构应该包含了所有的状态信息。对象方法（子程序）使用一个对象句柄（指向代表着某一个对象的数据结构的指针），完成了对对象的操作。

一个设备对象的新的请求可以通过调用驱动程序库（xxxDevCreate）中的设备创建函数来创建。在设备创建时期，驱动程序应该探测核实设备事实是否存在，如果不存在，那么设备创建操作将失败并返回一个 null 指针。

4. 每一驱动程序的数据结构

在遵守面向对象设计的方法论中，应该有一个结构来代表驱动程序本身，这个结构将包括所有驱动程序的状态信息（类变量）。从实用的观点来看，在一个单独的结构中包含了所有的驱动程序数据，这将使在 CrossWind 调试工具显示变得容易。

5. 驱动器中断服务程序

由于中断可能会涉及到总线问题，驱动程序不能直接调用 intConnect()，驱动程序应该定义一个可以被客户修改的宏，以此来调用正确的中断连接程序，它不会是 intConnect()。

```
#ifndef FOO_INT_CONNECT
#define FOO_INT_CONNECT(vec, rtn, arg) intConnect(vec, rtn, arg)
#endif
```

如果设备没有声明一个中断，那么设备驱动程序的 ISR 必须立刻退出。不要认为在中断向量和中断句柄之间有一个一对一的映射。就 PCI 系统而言，中断线路用于服务一个以上的设备是完全可能的。中断程序必须检查设备并判断是否事实具有中断的条件，如果没有，则中断句柄代码应该以尽量快的速度退出。

6. 访问宏

每一次对芯片寄存器的访问都应该通过宏来完成，这些宏能够被重新定义以适应不同的访问方式。通常一个读取宏和一个写入宏就足够了，在一些情况下，用于单独改变寄存器中一个位的宏也是需要的，例如：

```
M68681_READ(addr, pData)
M68681_WRITE(addr, data)
M68681_CLR_SET(addr, clear_bits, set_bits)
```

访问内存或者总线上的命令模块也同样需要一个可以被重新定义的宏以便适应字节交换和地址操作。

要注意，读取宏传递的是读取到的数据所要返回的地址，它们不会像子程序那样返回一个值。这是故意的，用这方式写的宏可以被一个简单的声明、子程序调用或者模块声明替换。为返回一个量而编写的宏不能够被一个 C 模块声明代替。例如：

```
xxx = M68681_READ (x); /* Limited */
M68681_READ (x, &xxx); /* Better */
```

尽量减少预处理程序代码模块中的条件语句，这样可以提高代码的可读性。如果条件表达式仅仅对程序改变了一个参数的数值，那么就应该在文件的一开始使用#define 语句来完成。只有在实际上改变了程序的逻辑流程的条件语句后才可以放在函数中。

在这个例子中，仅仅改变了在子程序调用中的一个参数，在子程序中加入了条件声明只能会使读者变得糊涂，因为它对执行的流程毫无影响。

```
#ifdef INCL_FOO
fooReset (&myFoo,
#else
fooReset (&yourFoo,
#endif
arg2, arg3, arg4);
```


为了调整这一情况，可以新建一个代表这个参数的宏，在文件开头部分的合适位置对它进行定义，然后在代码中的适当位置使用它。这就会使子程序一看就非常容易理解。例如：

```
/* a better way */
#ifdef INCL_FOO
# define THE_FOO myFoo
#else
# define THE_FOO yourFoo
#endif
. . .
fooReset (&THE_FOO, arg2, arg3, arg4);
```

一般的规则是，如果它真的改变了程序运行时的流程，就应该使用条件编译，如果只是参数值、变量的名字、子程序的名字被改变了，那么就使用上述的方法定义一个中介宏。要注意宏的名字，不应该太简单也不应该太复杂，下面的两个例子都是不合适的：

```
SIO_READ
NS16550_CONTROL_REGISTER_BIT_7_CLEAR
```

不要使用结构体来声明硬件寄存器，或者内存控制块。使用宏来代表偏移量常量或将基本地址转变为寄存器地址。结构体本身具有不易于移植的特点，当为体系结构安装驱动程序时，这是惟一最大的错误源。即使使用相同的工具链，默认结构体的建立在不同的体系结构之间也是不同的。C 语言规范并没有为结构体成员的排列制定一个标准。

```
typedef struct { /* I8250_DEV */
char CSR;
char DATA;
char MBR;
} I8250_DEV;
/* A better way */
#define I8250_CSR 0
#define I8250_DATA 1
#define I8250_MBR 2
```

5.5 逐步实现

在设计时一般的顺序应该是从上到下，但是在执行时和测试时的一般顺序应该是从下到上。

1. 从上到下设计

从上到下设计意味着在开始正式编码之前, 要进行计划和证明。

(1) 模板文件

从模板文件或现有驱动程序开始, 我们建议使用模板, 从现有的文件开始通常会引进其他驱动程序代码的很多问题。Wind River 为驱动器的所有类别提供了模板文件。

(2) 模块描述

找到芯片手册并将芯片的介绍性描述复制到模块的描述部分, 然后再加上一段详细描述这个驱动程序是如何工作的, 芯片的那些部分将会被控制, 以及可以得到支持的操作模式等内容。

(3) 设备结构

从定义每一设备的基本数据结构和每一个驱动程序的基本数据结构开始, 这些结构定义了驱动程序和每一设备的状态。像结构体一样, 只用一个简单的打印命令, 调试器就可以显示出它们的全部状态。

(4) 宏定义

证明所有用来为不同应用所定制的驱动程序的宏, 比如访问芯片宏、访问存储器宏等。告诉用户这些宏是作什么的, 以及它们的默认值是如何定义的。

(5) 子程序声明

声明用户及 BSP 编写者使用这个驱动器可能会需要的所有的子程序。尽可能地使硬件设计与软件设计一致。

(6) 无阻塞函数

为所有用户计划编写的程序建立标志和声明, 先使它们的内容是空的, 直到已经确实准备好编写其中的代码。在注释块中写上对子程序的描述。注释块中的注释比代码部分中的注释要重要得多, 它们可以帮助别人研究这个设计而没有必要去阅读代码中的内容。

2. 从底向上执行

在完成对驱动程序的计划, 以及初步形成了由函数和数据结构构成的骨架之后, 就要开始准备执行和测试了。

(1) 编写代码

从设备初始化开始, 编写代码来接受一个设备的数据结构并为以后的使用而对它进行初始化。为所有其他低级别的驱动程序函数添加内容。可以进行一次测试编译, 来核实所有必须的程序是否都已被声明。检查目标模型中的符号来确认只有所需的外部程序未被执行。

(2) 测试、调试、重新编译

一般的测试、调试和重新编译是一个循环过程。个人的不同选择和偏好影响着这一阶段, 一些人一次只对一个程序进行编码和测试。

(3) 一次完成一层

对代码的每一层重复地进行编写、测试、调试、重新编译的循环工作。对每一层进行

全面的测试可以提高工程的可信度。企图一次编写全部的代码是难以做到的。

(4) 性能测试

应该进行一系列的基准程序的测试，来证实这个设备可以满足一般的客户需求。仅能提供 5MB/s 网络吞吐量的宽带 SCSI 设备，是不能很好的满足客户要求的。工程师有必要编写一个完整的性能测试基准程序作为整个项目工程的一部分。

5.6 关于缓存的问题

VxWorks 缓存库 (cacheLib) 的设计目的是从系统其他部分中隐藏体系结构和目标具体细节，并且为保持缓存的一致性提供一种机制。缓存一致性是指缓存中的数据必须与 RAM 中的数据保持同步（或者一致）。

设备驱动程序是在 VxWorks 系统中，能够产生与数据缓存一致性有关问题的模块类型之一（注意到在系统的其他部分保持了 CPU 指令缓存的一致性）。这部分内容描述了驱动程序编写者要关注的一些问题，并且描述了如何使用 cacheLib 来处理这些问题。

这部分内容还描述了如何启用虚拟内存库，如果使用 68K 或 SPARC 体系结构，那么这就是必要的。在这些体系结构中，VxWorks 是通过调用虚拟内存库操作存储器管理单元 (MMU) 来控制缓存的。

只有使用直接存储器访问 (DMA) 的驱动器是与缓存问题有关的。只有在 DMA 范围内，缓存问题才是设备驱动程序中需要考虑的问题。对于具有写入缓冲器的体系结构来说，驱动程序需要注意与 WRITE_PIPING 有关的问题。



注意：如果你的目标体系结构包括 MMU，而且禁止了对虚拟内存的支持但是启用了数据缓存，那么 cacheDmaMalloc() 不能够保证缓存器一致的安全。

如果使用 MMU，那么缓存模式由 sysPhysMemDesc[] 表中的缓存模式值来决定，而不是由 USER_I_CACHE_MODE 和 USER_D_CACHE_MODE 这两个配置宏来决定。认为这些宏总是能够控制缓存的工作模式是一个普遍的错误。

5.7 有益的忠告

1. 避免在驱动程序中使用 printf()

即使目的是为了调试，也要避免在驱动程序中使用 printf()。而使用 logMsg() 来代替。

在驱动程序和 I/O 系统之间可能会有系统的交互作用，这可能会使 `printf()` 导致系统瘫痪。

2. 调用 `IntConnect()`

不要从 `sysHwInit()` 内部调用 `intConnect()`，或者任何其他调用 `malloc()` 的程序，因为内存分配库还没有被初始化，系统将会瘫痪。

5.8 驱动程序使用缓存器库

在阅读这一部分之前，先回顾一下 `cacheLib` 的访问入口。这一部分描述了如何使用 VxWorks 的缓存库 (`cacheLib`) 来维持设备驱动程序的数据缓存一致性。同时也描述了控制 CPU 书写通路的边缘效应，并且给出了一些关于处理用户设备驱动程序中与缓存相关问题的忠告。

5.8.1 回顾 `cacheLib` 功能

`cacheLib` 库的访问入口详细描述了这个库的功能，本节将介绍文件设备的简要定义。请记住，`cacheLib.h` 这个头文件包含了函数原型，并且宏需要使用 `cacheLib` 库中的功能。在用户驱动程序代码中要包括这个文件。

这个文件也涉及到下列程序和宏：

- `cacheDmaMalloc()` 这个程序从系统中分配出一些内存，它尽可能使用潜在的硬件特性来提供一块相对于缓存一致性问题较安全的内存区域。这个内存区域确保了从缓存队列分界线开始，而且不会使缓存队列与邻近的区域相互交叠。
- `CACHE_DMA_xxxx` 宏(例如 `CACHE_DMA_FLUSH`)刷新，使无效，并且获取由 `cacheDmaMalloc()` 程序提供的内存区域的属性。
- `CACHE_DRV_xxxx` 宏(例如 `CACHE_DRV_INVALIDATE`)刷新，使无效，并且获取由驱动程序控制的普通存储器的属性。
- `CACHE_USER_xxxx` 宏(例如 `CACHE_USER_IS_WRITE_COHERENT`)刷新，并且使驱动程序范围之外的用户存储器无效。

5.8.2 驱动程序分析指导

每一个驱动程序都有一系列的属性，它们定义了设备与缓存有关的行为。这一部分为深入地分析用户驱动程序潜在的缓存一致性问题提供了一些指导。当了解了驱动程序的属性，就可以制订出在驱动程序中使用 `cacheLib` 程序中的策略。

在开始行动之前，先检测设备是否具有 DMA 能力。设备能够对与 CPU 共用的内存直

接进行读写访问吗？如果答案是不可以，则你的驱动程序将不需要任何 `cacheLib` 中与缓存有关的功能。缓存问题只会影响那些能够访问与 CPU 共用内存的设备。

如果 CPU 的体系结构要执行缓冲写入，那么即便设备中不包括 DMA 功能，也需要处理 `WRITE_PIPING`。在大多数硬件环境下，都提供了一种硬件机制，它维持了 I/O 地址不会被缓存。然而，需要记住的是即使是不具有 DMA 功能类型的设备，仍旧会有与写入管道传送技术有关的问题（参见“驱动程序属性”）。

1. 共享内存的类型

对于 DMA 类型的设备，驱动程序和设备共享一块或者更多块内存区域，这个共享的内存可能会是下列这些类型之一：

- 由 `cacheDmaMalloc()` 分配的内存。这个内存与 `CACHE_DMA_xxxx` 宏有关，并且对于缓存问题是由驱动程序和底层的硬件来控制的。
- 使用 `malloc()` 或者 `memalign()` 程序分配的内存，或是在模块的数据段或 `bss` 段声明的内存（堆栈内存绝对不能和设备共享）。这种类型的内存与 `CACHE_DRV_xxxx` 宏有关，而且对于缓存问题是单独由驱动程序控制的。

因为用户不能控制由这种方法获取数据的位置，因此它有一个内在的问题：与邻近的不属于这个驱动程序的内存区域共享一个缓存队列的可能性。这就意味着在这个内存区域刷新和使无效操作会干扰到共享的缓存队列中临近数据的一致性，因而就要对这种类型的内存的使用加以限制，排除这个区域中的第一个和最后一个缓存队列。因为在不同的系统中缓存队列的大小是不同的，所以这就为程序移植带来问题。

通过使用 `memalign()`，可以保证缓冲器在其初始地址对齐。如果需要保护缓冲器的末端，那么应该将这个请求的大小增加到至少一个缓存队列的大小，这样就确保了缓冲器的末端不会与任何其他的缓冲器共享一个缓存队列。

起始地址是未知的内存。这种内存是与 `CACHE_USER_xxxx` 宏相关的，而且是在驱动程序的控制之外的。换句话说，驱动程序事实上并不知道内存是如何来分配的，但是 `CACHE_USER_xxxx` 宏假定这个内存是使用 `malloc()` 来分配的。

具有固定或专门用途的内存。这种内存是由目标硬件提供一块特殊的区域，是为 DMA 类型的设备所专用的。这样做通常是出于对性能因素的考虑。这类内存不是系统层中的一部分，因此它与 `malloc()`、`cacheDmaMalloc()`、数据段、或 `bss` 段没有关系。没有与这种类型内存有关的缓存宏。通常是由硬件提供了一种方法，使得这个完整的区域不能缓存。

2. 驱动程序属性

在这一部分中，列举了在为用户的 `cacheLib` 策略做计划时，需要注意的一些驱动程序的属性。为了使后面讨论起来比较容易，我们为每一个属性起了一个名字。不同于写入管道传送技术，这些属性都是仅仅与 DMA 类型的设备有关的。

(1) `WRITE_PIPING`

CPU 使用写入管道传送技术。写入管道传送技术的意思是 CPU 的写操作先保留在管道

中，直到外部总线中产生了适宜的状态为止。写入管道传送技术在 RISC 体系结构中使用，例如 MIPS。这将会对用户的驱动程序产生严重的影响，因为这样会对传送给用户驱动程序控制下的设备数据和命令产生延迟。

如果驱动程序在具有写入管道的 CPU 上运行，那么这个属性就是“TRUE”。

(2) USER_DATA_UNKNOWN

用户数据处于未知的状态。如果用户驱动程序在设备和驱动程序的用户之间通过地址直接地传送数据，那么这个属性为“TRUE”。在这种情况下，由用户传送给驱动程序的数据正处在一种未知的缓存状态。其中一个例子是，一个指向数据缓冲器的指针为了向 SCSI 磁盘中写入数据，而被交给了 SCSI 磁盘驱动程序，在这种情况下，驱动程序不知道缓冲器是否已被刷新，或者还是处于缓存中。从相反方面说，设备获取到的数据必须在指向那个数据的指针被交给用户使用之前，保证缓存与内存之间的一致。

(3) MMU_TAGGING

硬件提供了缓存区域的 MMU 标签。在目标硬件上，当一个 MMU 允许存储区域的标记为无缓存器状态时，属性为 TRUE。

(4) DEVICE_WRITES_ASYNCHRONOUSLY

硬件提供了对共享内存的异步写入操作。如果设备能够异步地与驱动程序的行为在共享的内存中执行写入操作，那么这个属性就是“TRUE”。

(5) SHARED_CACHE_LINES

硬件允许驱动程序和设备两者向一个单独的缓存队列中写入数据。如果任何单一的缓存能够被驱动程序和设备两者写入数据，则这个属性就是“TRUE”。其中一个例子是共享数据结构，如下：

```
struct
(
    short command;
    short status
);
```

在这个例子中，驱动程序负责写入命令，而设备负责写入状态。

(6) SNOOPED

硬件提供了总线侦听功能。如果目标服务器硬件支持总线侦听，那么这个属性就为“TRUE”。总线侦听使得缓存问题对于驱动程序来说变为透明。只有能够保持完全一致性的完全地总线侦听才能够得到支持，一些处理器提供了部分侦听能力，它不能满足完全的内存一致性的要求。如果没有完全的侦听能力，这个属性就必须是“FALSE”。

(7) SHARED_POINTERS

设备的控制模型允许驱动程序和设备之间交换内存指针。如果在驱动程序和设备之间交换内存指针，则这个属性应该为“TRUE”。

5.8.3 开发 cacheLib 策略

这一部分介绍了如何在用户的驱动程序属性的基础之上，设计出一种 cacheLib 策略。

在开始之前，先要确定所支持的目标板的范围，例如：一个单一的目标板，多种不同的目标板，或者更大的支持。这个决定将会对用户的缓存策略的选择产生影响。例如，一个为更大的支持设计的驱动程序会被限制在 cacheLib 库中程序的某一个范围内，因为目标板的具体细节是未知的。然而，为仅仅支持一个单独的已知目标板而设计的驱动程序，设计师就可以使用 cacheLib 中的任何程序，来充分地利用目标板硬件的性能。

在驱动程序中设计的缓存器库支持等级应在代码中明确记录。这是一个绝对不能被忽视的重要问题。

cacheLib 库中的很多程序和宏在设计时都隐藏了底层硬件的具体细节问题，宏可能会实际调用 cacheLib 库内部的程序，它们也可能会根本什么都没有做。它们的行为取决于在运行时某一个功能指针是如何被初始化的。cacheLib 库在运行时的初始化是由 BSP 模块中的代码来决定的，也是获知缓存硬件实际细节的地方。这种隐藏掉具体细节的方法使得驱动程序的目标更加广泛，并且提高了可移植性。

好的驱动程序设计包括可移植性，在下面的一小部分内容中假设可移植性是最主要的目标，并且最低程度地讨论了特定目标的应用软件。

1. 刷新和使无效宏

通常，需要在其驱动程序中添加刷新和使无效宏。这个宏依据需要控制的内存类型（参见“共享内存类型”）来设置用户的设备。刷新宏用在存储器的位置上，此位置已告知设备以便其后续读取。使无效宏用在驱动程序读取内存中的特定区域之前，这个区域中的数据是由设备写入的。

可移植式驱动程序在所有需要的地方都使用了刷新宏和使无效宏。如果在某些特殊的目标服务器中不需要这些宏，那么它们就可以被编写为执行空操作。

2. WRITE_PIPING 属性

cacheLib 中的宏 CACHE_PIPE_FLUSH 刷新了 CPU 的写入管道，通常，需要将这个宏添加到用户的整个驱动程序中的适当位置，这个适当位置是指在向设备写入一个或者更多的命令之后的位置。需要立刻传送设备命令是最重要的，这些命令常见于初始化序列中或者中断处理器中（一个中断条件需要被清除的地方）。

要判断何时需要使用这个宏，这需要对正为之编程的设备有深入的了解。一些设备不会受到由写入管道传送技术所引起的延迟的影响，这样的设备在驱动程序中不需要管道刷新宏。而其他一些设备在没有和驱动程序严格地同步时，就不能正常的工作，这样的设备需要在驱动程序中充分的使用管道刷新宏。

我们建议在 I/O 写入操作和随后的 I/O 读取操作之间使用 CACHE_PIPE_FLUSH 宏。

连续的写入或者读取操作不需要任何特殊的保护，只有在写入操作向读取操作转换时，管道的刷新才是必要的。

要注意，对写入管道的刷新并不意味着与此同时缓存也被自动的刷新，必须还要处理驱动程序中的缓存问题。从反面来说，刷新或者禁用缓存的同时并不会自动刷新或禁用写入管道。写入管道处于体系层次中的第一级。

3. SNOOPED 属性

这个属性对于所有的内存类型都是透明的。可以移植的驱动程序使用了适当的刷新和使无效宏。然而，当 SNOOPED 属性为“TRUE”时，宏被设定为执行空操作。特定目标服务器的驱动程序不需要使用刷新和使无效宏。

一些系统运行部分的侦听，它们可以读取侦听数据，并且仅能够使写入缓存数据无效。这样并不能满足完全内存一致性的需求，在这种情况下，SNOOPED 属性是“FALSE”，而不是“TRUE”。

4. MMU_TAGGING 属性

这个属性对使用 `cacheDmaMalloc()` 而得到的内存类型是透明的。可移植式驱动程序使用 `CACHE_DMA_FLUSH` 和 `CACHE_DMA_INVALIDATE` 宏。然而，如果 `MMU_TAGGING` 属性是“TRUE”，这个宏被赋予空操作，特定目标服务器的驱动程序不需要 `CACHE_DMA_FLUSH` 或者 `CACHE_DMA_INVALIDATE` 宏。

这个属性一般不会影响使用 `malloc()`、数据段、或 `bss` 段得到的内存。因此，必须适当的使用 `CACHE_USER_XXXX` 和 `CACHE_DRV_XXXX` 刷新和使无效宏，就像在“刷新和使无效宏”部分介绍的一样。

5. USER_DATA_UNKNOWN 属性

使用这个属性，用户的驱动程序必须维持在设备与用户之间直接传送任何数据的缓存一致性。对于流出的数据，在命令设备访问这个数据之前，首先使用 `CACHE_USER_FLUSH` 宏对从缓存器获得的用户数据进行刷新。对于流入的数据，在将数据交给用户之前，首先使用 `CACHE_USER_INVALIDATE` 宏为设备中的数据使缓存器无效。

6. DEVICE_WRITES_ASYNCHRONOUSLY 属性

这个属性只有在与 `SHARED_CACHE_LINES` 属性相结合时，才是值得关注的。请参看下面的两个部分。

7. SHARED_CACHE_LINES 属性

只要驱动程序能够将所有的操作与可能共享一个缓存队列的数据帧同步（如果驱动程序不能够同步这些操作，请参见下一部分），那么驱动程序就可以简单的使用这个属性实现一些功能。如果只有当设备受到由驱动程序发出的明确命令时才能对数据帧进行写入操作，那么驱动程序可以以一种能够确保数据完整性的方式对刷新宏、使无效宏和设备指令进行

排序。

8. DEVICE_WRITES_ASYNCHRONOUSLY 和 SHARED_CACHE_LINES 属性

DEVICE_WRITES_ASYNCHRONOUSLY 属性和 SHARED_CACHE_LINES 属性的结合会引起一些问题。因为驱动程序不能对设备的写操作进行同步，所以数据出现错误的潜在可能性还是存在的，如果设备执行了写操作，然后驱动程序执行了刷新操作，那么错误就会发生，在这个顺序下，由设备写入的数据被缓存队列的刷新操作清除了。

对于这一组属性，驱动程序不能使用普通的 CACHE_DRV_FLUSH 和 CACHE_DRV_INVALIDATE 宏，因为驱动程序不再能够控制共享内存的完整性了。这个共享的内存必须使用 cacheDmaMalloc() 函数来获得，此外，这个函数必须提供具有写入一致性的内存，因为驱动程序不能执行任何刷新操作。驱动程序可以通过使用 CACHE_DMA_IS_WRITE_COHERENT 宏来检查由 cacheDmaMalloc() 提供的内存属性。如果这个宏被设置为“FALSE”，则驱动程序就不能可靠的运行并且应该放弃。

9. SHARED_POINTERS 属性

只有当为驱动程序和设备所共享的内存是使用 cacheDmaMalloc() 函数得到的时候，才关注此属性。因为这个函数会调用 MMU 或其他特殊的硬件服务，所以内存提供的地址可能是一个虚拟的地址。然而，设备只能处理物理地址。

在这种情况下，使用 CACHE_DMA_VIRT_TO_PHYS 和 CACHE_DMA_PHYS_TO_VIRT 宏来在驱动程序角度的地址和设备角度的地址之间进行转换。

5.8.4 其他的关于缓存库的忠告

下面是一些对于使用缓存库的其他忠告：

- 在向用户驱动程序添加任何对 cacheLib 的支持之前，先要确保这个驱动程序能够在没有数据缓存时能可靠地工作。在禁用数据缓存时，将不会碰上缓存一致性问题。当开始增加对 cacheLib 的支持时，可以启用数据缓存。这样，任何驱动程序可靠性的改变都可以归结为缓存问题。
- 先使用特定目标服务器的技术来指导 cacheLib 的开发策略，确保具有 cacheLib 支持的驱动程序能够在一个单一的目标服务器上正常地工作。在完成这一步之后，就可以修改驱动程序的 cacheLib 策略，来实现更广泛的、不受目标服务器约束的支持。

第6章 组 件

6.1 简 介

Tornado 2.x 介绍了一种新的配置方法，更好地适用于越来越复杂的 VxWorks 环境。这种方法使用了 Tornado 2.x 的图表配置工具和工程工具，提供给使用户进入 VxWorks 的配置世界的一种观点。这种工程工具显示了一系列按等级排列的可选择的软件组件，它利用点对点技术，很容易地适用于 VxWorks（对 Tornado 2.x 而言，可下载的应用软件不用组件）。

对大多数用户而言，这种配置系统的 GUI 水平的观点已足够满足它们的发展需求。然而，这种工程工具包含了一种 API 和一种语言——组件描述语言（CDL），这用语言用来定义组件，以及它们的参数，可测量性，和其他组件的关联。本章讨论了怎样应用这些建筑板块，API 和 CDL，使之能创建或修改被选择的软件组件，并适用于用户所想象的或应用的 VxWorks 系统。

组件是一种将代码和配置数据并入功能单元的一种高水平方法，比如说“消息日志”。可启动的目标是组件，及它们的配置参数值，构造选择的搜集。在 Tornado 2.x 中，使用工程工具是工程操纵的惟一方式，也就是说，工程不能够由命令行配置（更多的有关怎样使用工程工具的信息请见《Tornado 用户指南》：工程）。

构建 VxWorks 的传统方法是基于构造头文件，它在快速发展的软件的复杂性面前显得越来越低效。自 VxWorks 5.0 发布以来，大约有 20 个为 VxWorks 映像所配置的可选择的组件。直到 VxWorks 5.4 发布以来，大约有将近 300 个组件，其中不包括第三方卖主提供的组件。使用配置头文件对于其发展在下列三方面起作用：

- 可用性。

随着系统发展越来越复杂，很难形象化系统中所包含的部分。通读一遍或是逐行修改配置资源代码成为一种挑战。而且，如果系统被错误配置，没有任何机制会来提醒用户。

- 可测量性。

配置头文件没有提供任何组件之间相互依赖的可视性。如果用户不知道它们的关联性如何，用户就不知道需要什么或是用户可以删除些什么。

- 最新版本。

当发布一个新的版本时或第三方组件或用户修改时，系统组件中总会出现配置冲突。这是因为所有的源代码变化总是出现在两个文件中——usrConfig 和 configAll.h。

执行工程工具例如配置工具就解决了以上问题。

在以下的篇章中将讨论怎样运用组件描述语言 (CDL) 以及怎样分解和合成这些系统组件。本章中举了些例子, 将传统的配置头文件与 Tornado 2.x 工程工具对比, 阐明了两个工程的不同, 以及它们的补充。

我们编本书时假设用户已经熟练掌握 Tornado 的配置, 包括其工程工具。更多的信息请见《Tornado 用户指南》: 工程。



注意: 这个版本并不限制用户使用配置头文件来配置 VxWorks 及应用软件。在其他环境中使用 config.h 和 configAll.h 来补充工程工具。

6.2 组件描述语言

在过去, 描述一个组件的配置头文件的惟一信息就是它的预处理程序宏 (例如, INCLUDE_NFS)。Tornado 2.x 介绍了一种组件描述语言, 它包含每个组件的更多有用信息的图表描述, 例如:

- 与相关组件的任何等级制分组;
- 与其他组件的关联依赖性;
- 配置特性;
- 如何集成到一个已有系统 (为了组件的发布)。

这些信息被 CDL 目标组织起来, 一共分 5 个等级:

- 组件目标;
- 参数目标;
- 文件夹目标;
- 选择目标;
- 初始化群组目标。

因为组件是用户配置系统的中心构造, 必须首先讨论它们。这个选择提供了关于每个 CDL 目标和它们各自工具的看法。

6.2.1 组件

组件是一个配置软件的基本单元。它们是系统中最小的, 可升级的单元。通过工程工具, 使用户能添加或是删减一个组件, 也可以修改一些特征参数。

CDL 是用组件描述文件 (CDF) 来标准化软件的组件。每个 CDF 带着后缀.cdf。组件描述文件的传统在 6.3.1 CDF 协定中有所描述。每个文件可定义不止一个组件。

在 Tornado 2.x 中, 代码产生器基于用户所选择的配置组件输出系统配置文件。一个工

程的全部配置只是一张包含组件及其特征值的清单。有关工程工具的详细资料和代码产生器是如何工作的请见《Tornado 用户指南》：工程。

在过去用户通过修改配置文件（config.h 或 configAll.h）从而把组件认为是包含在系统中的代码和参数，如下：

```
#define INCLUDE_FOO
```

对于 Tornado 2.x 而言，用户使用 CDF 定义它的特征参数来描述一个组件，如下：

```
component INCLUDE_FOO
{ // Define a component called INCLUDE_FOO.
NAME          Foo example component
SYNOPSIS      This is just an example component
...//other properties
}
```

组建特性属于以下 4 个种类之一：

- 代码。这可以是建构一个工程使用的目标代码（模块）或是源代码（configlet 和 stubs）的合集或是子集。
- 配置信息。CDL 提供了丰富的特性参数用于描述一个组件。这些参数可以被指定，改变，增加或是删除。参数是配置的参数，例如数据的类。典型的预处理程序宏用在一个组件的配置代码中。个别的 CDL 目标用于定义这些参数。
- 集成信息。这些特征参数控制一个组件是如何集成到一个可执行“系统映像”中的，例如，一个初始化程序和初始化过程中何时调用程序的描述。

集成特性参数也定义了组件间的依赖性，代替了以前的 Tornado 版本中的 #ifdef/#endif 的陈述。

- 用户介绍。很多的参数控制了一个组件怎样被介绍给用户，也就是一个组件怎样呈现在 Tornado GUI 中。例如，特征信息定义了一些特征参数，最基本的是一个组件的名称，一个说明组件配置的简洁大纲。它们也影响了在工程工具组件层级中在哪里一个组件，或一个文件夹，或一个选集应被陈列。

也可以为一个特殊的组件指定 HTML 参考纪录，如支持材料等。

在 6.2.2 CDL 目标类型部分详述了不同特征值和它们的用途。

关于组件描述文件的更多信息请参见“6.3 创建组件及 6.4 分布组件”。

6.2.2 CDL 目标类型

组件描述语言支持很多目标类型。随着组件的增加，一种把它们分组到相关的字系统的方式已经发展起来，它被称为文件夹和选集，CDL 目标描述了每个方式。初始化分组提供了一种指定一个组件的初始化顺序的方式，这是另外一种目标类型。当然了，CDL 提供

了一个目标来定义一个组件以及它的特征参数。还有为定义参数的附加目标类型，数据类型用来表示组件特征参数。

图 6-1 显示出了工程工具组件的等级。在工程工具组件等级中，若被包含在配置中，则条（组件，选集和文件夹）用黑体字标明，不被包含，用普通的字体标明，当它们还没被安装时用斜体字标明（也就是说，当一个 CDF 已经描述了一个组件，但是这个组件的模块还没有被安装。）

完整的每个 CDL 目标类型的用法，请见附录 E.1 组件描述语言（CDL）。

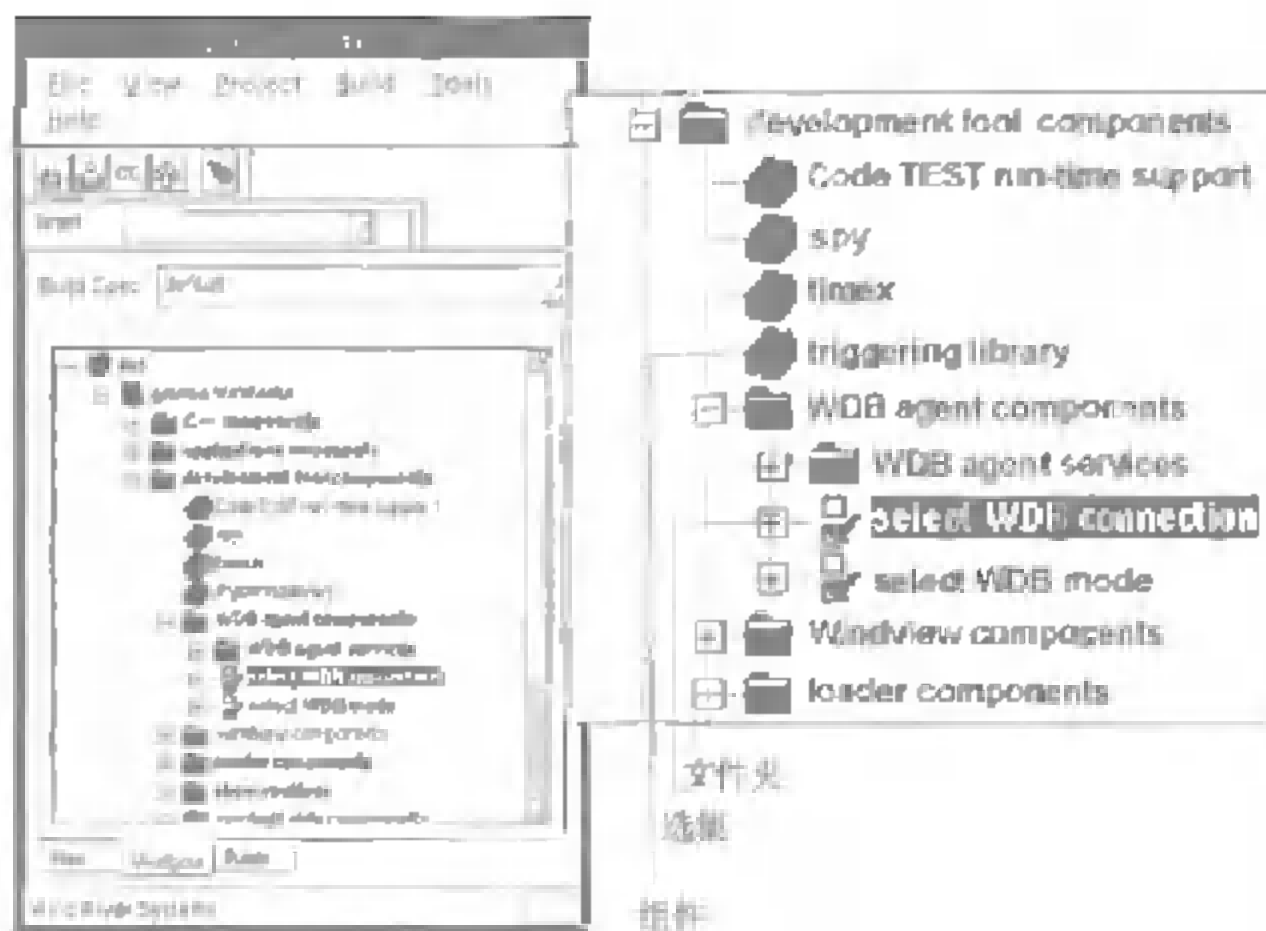


图 6-1 工程组件层次结构

1. 文件夹

文件夹提供了一个组件分组的目录类型的等级，它们是逻辑相关的。WDB 代理子系统，ANSI，组建 POSIX 子系统都是被文件夹分组的组件。在工程工具中，文件夹被文件夹图表文件所描述，如图 6-1 所示。

文件夹包含了不止一个组件。在 Tomado 2.x 以前，组件分组通常是被嵌套完成的，其中的陈述例如：

```
#ifdef INCLUDE_ANSI_ALL
#define INCLUDE_ANSI_ASSERT /* ANSI C assert library functionality */
#define INCLUDE_ANSI_CTYPE /* ANSI C ctype library functionality */
#define INCLUDE_ANSI_LOCALE /* ANSI C locale library functionality */
...
#endif
```

对于 Tornado 2.x 而言, ANSI 功能性包含在 CDL 所描述的文件夹中, 如下所示:

```
Folder FOLDER_ANSI (
NAME ANSI C components (libc)
SYNOPSIS ANSI libraries
CHILDREN INCLUDE_ANSI_ASSERT \
INCLUDE_ANSI_CTYPE \
INCLUDE_ANSI_LOCALE \
INCLUDE_ANSI_MATH \
INCLUDE_ANSI_STDIO \
INCLUDE_ANSI_STDLIB \
INCLUDE_ANSI_STRING \
INCLUDE_ANSI_TIME \
INCLUDE_ANSI_STDIO_EXTRA
DEFAULTS INCLUDE_ANSI_ASSERT INCLUDE_ANSI_CTYPE \
INCLUDE_ANSI_MATH INCLUDE_ANSI_STDIO \
INCLUDE_ANSI_STDLIB INCLUDE_ANSI_STRING \
INCLUDE_ANSI_TIME
)
```

文件夹在分组组件中比以前的 Tornado 2.0 提供了更加灵活的方法。以前, #ifdef 陈述表明了所有的详细定义也同时被包含在其中。现在, 文件夹允许分组包含不只一个组件的默认值, 这些默认值被放在一起 (在 Tornado 2.x 中通过文件夹目标的默认特征值所定义); 组件可以被增加或是被配置个别地移动 (被文件夹目标的 CHILDREN 特征值所定义)。

文件夹信息很大程度上影响了用户的表达。这跟初始化分组等级没有关系, 因此和启动顺序与关, 启动顺序仅仅取决于一个组件的初始化分组。文件夹可以包含一个或更多的组件, 选集和其他的文件夹; 它们没有相关的参数或初始化分组目标。

一个文件夹有以下特性:

- 名称。一个易读的名称, 在一个工程工具组件等级中出现在文件夹图表旁边。
- 大纲。对一个文件夹的简洁的描述。
- 子文件夹。属于这个文件夹的组件, 文件夹和选集称为子文件夹。
- 默认值。如果没有使用任何代理, 把文件夹加进去的话, 它将包含组件的默认值。

当一个文件夹被加进去的时候, 文件夹分组能够影响配置, 这是因为组件会同时被文件夹的默认值所指定。

用户可以通过看相关的 CDF 或看内含的文件夹对话框为一个文件夹确定组件所设定的默认值 (图 6-2), 双击工程工具组件等级的特殊文件夹可打开这个对话框。这个内含的文件夹对话框只显示了那些还没被设置的组件。如果一个组件被设置为默认值, 它将以复选标记出现在新开的窗口中。为了看到一系列完整的默认值, 用户必须去掉文件夹中的所有组件。

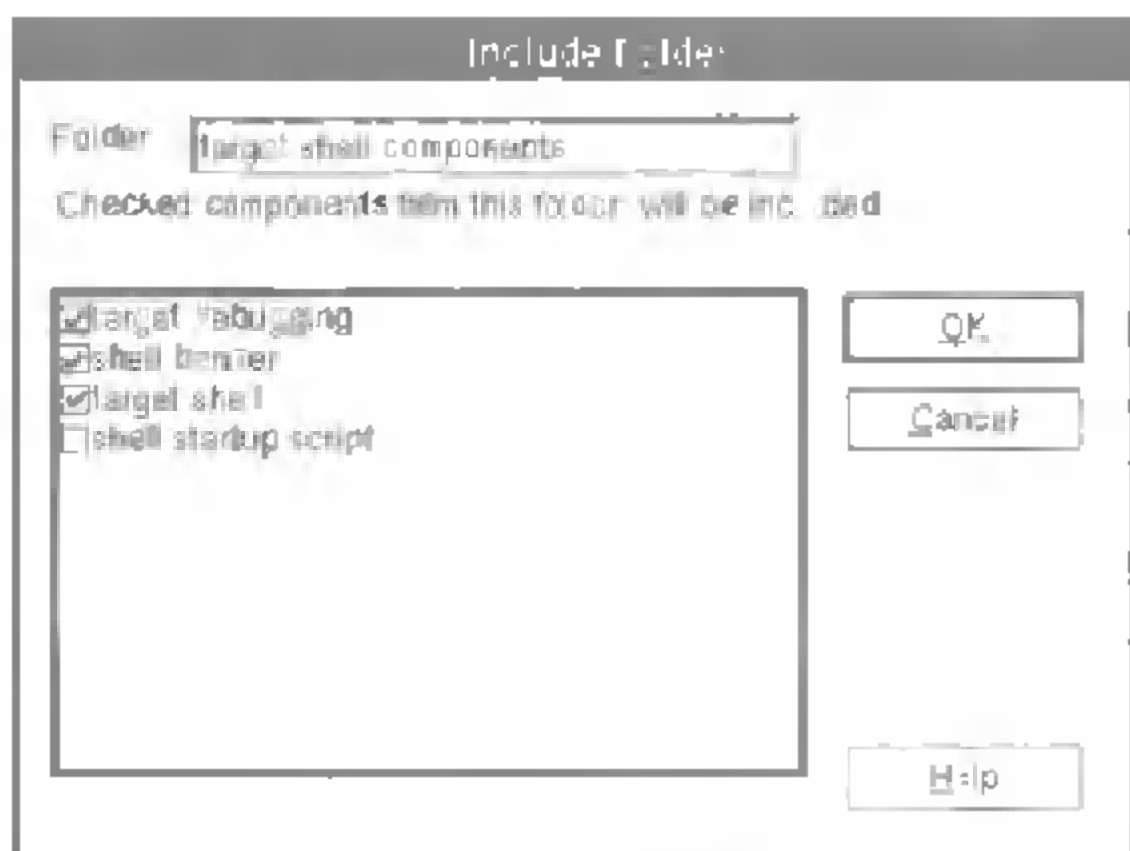


图 6-2 包含文件夹对话框

注意：仅当创建一个新的组件的群组（新的或正在创建的）时才能使用文件夹目标。不要为了增加新的组件而修改正在创建的文件夹。CDL 通过预先考虑下划线的特性名称而提供了那些配置，例如，CHILDREN。更多的信息请见 6.4 分解组件。

Tornado 2.x 分布的文件夹的等级清单，请见附录 E.2 文件夹等级。

2. 选集

选集类似于文件夹，但它们是执行一个普通界面的组件，例如，串口驱动器，WindView 时标机制，以及 WDB 通信界面。这些组件为相同的服务提供了多种选择，可为单个工程工具提供一个或多个选择。在工程工具中，它们以校验盒图标显示，如图 6-1 所示。

在 Tornado 2.0 以前，`#ifdef/#else` 配置完成同样的任务。在 Tornado 1.0.1 中的例子，用户可选择用 WindView 时标设备：

```
#ifdef INCLUDE_USER_TIMESTAMP
... /* user timestamp driver */
#elif defined INCLUDE_TIMESTAMP
... /* system timestamp driver */
#else
... /* DEFAULT: sequential timestamp driver */
#endif
```

文件夹的选择信息是仅为用户介绍的。这与初始化分组等级无关，因为它也和启动顺序

无关。选集仅包含一个或更多的组件。

选集像文件夹一样工作，但它能为一系列的可用到的组件增加了一个计数器：例如，一个包含三个组件的选集能告诉用户一次配置成功或三次中能有两次成功。由于这个计数器，选集并不包含文件夹或其他选集；也没有参数值或相关的初始化分组目标。

选集有以下特征参数：

- 名称。一个可读的名字，在工程工具组件等级中出现在选集图表的旁边。
- 大纲。对选集的简洁的描述。
- 计数器。为可用的选项设定一个最小值和最大值。
- 子选集。可选择的组件。
- 默认值。依赖计数器的组件默认值。

对于 Tornado 2.x 而言，以上 Tornado1.0.1 代码所捕获的同样的特性可用 CDL 描述为：

```
Selection SELECT_TIMESTAMP {
NAME select timestamping
COUNT 1-1
CHILDREN INCLUDE_SYS_TIMESTAMP \
INCLUDE_USER_TIMESTAMP \
INCLUDE_SEQ_TIMESTAMP
DEFAULTS INCLUDE_SEQ_TIMESTAMP
}
```

一共有 3 个可用的时标驱动器，分别为 CHILDREN 特征的 3 个值所指明。计数器允许选择一个，也就是在最小值和最大值之间选一个。

3. 组件

组建目标等级定义了和一个组件相关的源代码和目标代码，集成信息以及任何相关的参数。在工程功能中，组件被建构块图标所代表，如图 6-1 所示。

以前，这种信息被包含在 usrConfig.c 或 src/config/* 文件中，放在 #ifdef/#endif 语句周围，如下所示：

```
#ifdef INCLUDE_LOGGING
logInit (consoleFd, MAX_LOG_MSGS);
#endif /* INCLUDE_LOGGING */
```

Tornado 2.x 结构文件仅仅包含必要的信息来建构应用程序或 VxWorks，并不是所有的成百的其他组件的相关代码能被包含其中。而且，组件间的依赖性能被探测，相关的组件能被工程功能自动的增加。他通过分析每个相关的目标模块的通用信号和分辨哪些组件被包含其中来具备以上功能。以上同样的组件信息日志可用 CDL 写为：


```

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

```

组件目标包含大量的特征。以下描述了更多的普遍使用的特征；剩下的定义请见 E.1.1 组件特征：

- 名称。一个可读的名称，在工程功能组件等级中出现在组件图表的旁边。
- 大纲。对一个组件的简洁的描述。

以下的四个特征与一个工程的配置代码有关：

- 模块。与组件关联的目标文件。
- 配置。普通的配置源文件有典型的使用相关联的组件参数的源代码。
- BSP_STUBS。通用配置源文件被复制到 BSP 目录中作为第一使用。用户能把 BSP 指定的选项放到复制的文件当中而不影响其他的 BSP。请注意修改的存根文件使用同样的 BSP 作为基础被所有的工程共享。



注意：源代码能以 BSP 存根的形式进入。Tornado 2.2 支持 BSP 存根；而 Tornado 2.x 以前的版本不支持 BSP 存根。

- HDR 文件。与用户的配置代码或是初始化程序相关的头文件。以下是必须被包含的头文件，这样才能编译用户的配置或是初始化程序。
- 档案文件。在档案文件中找寻存储的目标模块而不是在标准位置上。

以下特性提供了配置信息：

- CFG_PARAMS。一系列的跟组件相关的配置参数，通常为一系列的宏。每个参数能被各自的参数目标描述。

以下为一组在系统映像中的组件控制集成特性。这里有初始化信息和依赖信息，例如被连接符号传送的，需要#include 语句和#exclude 语句的，及 if/then#include 语句：

- INIT_RTN。一条初始化程序。
- LINK_SYMS。包含存档文件组件一系列可查询的符号。
- REQUIRES。如果组件被包含的话，也必须包含一系列没有结构依赖性的组件。代表性地，只列出了那些不能被 MODULES 追踪的组件，也就是说被相关的目标文件追踪，例如，只是配置组件或通过功能指示器涉及的其他组件。在后面一种情况，REQUIRES 能指定一个选集。

- **EXCLUDES**。一系列的如果这个组件正在被包含就不能被包含的组件。
- **INCLUDE_WHEN**。当一个组件被包含时（也就是说它处理嵌套包含）设定一个依赖关系能自动地包含特定的组件。
- **INIT_BEFORE**。在被这个特性指定化之前调用这个组件的初始化程序。这个特性仅当与 **_INIT_ORDER** 在一起时有效。
- **_INIT_ORDER**。这个组件属于指定的初始化群组。这个特性把指定的组件放在 **INIT_ORDER** 初始化群组目标的特性清单的末尾。

正如它们前面的名字和大纲一样，一下特性也影响用户的表达：

- **帮助**。与组件相关的参考专栏。
- **子目录**。这个组件是指定文件夹的子组件。
- **默认值**。这个组件是指定文件夹的默认组件。这个特性必须与子目录一起使用。

4. 参数

参数是用户配置一个系统的主要方式之一。通常，一个或更多的参数控制一个组件的特性。例如，我们考虑一个日志队中的日志信息的最大数量。在 Tomado 1.0.1, 在 `target/config/configAll.h` 中为：

```
#define MAX_LOG_MSGS 50
```

对于 Tomado 2.x 而言，使用参数目标等级描述被一个特殊组件的 **CFG_PARAMS** 特性列出的那些参数。日志例子如下：

```
Parameter      MAX_LOG_MSGS
(  NAME        max #queued messages
  TYPE         unit
  DEFEAULT     50
)
```

以下的特性描述一个参数：

- **名称**。一个可读的名称，出现在工程工具的特性窗口的参数表的参数 ID 清单的下面，如图 6-3 所示。
- **数据类型**。参数的数据类型，也就是，`int`，`uint`，`bool`，`string`，`exists` 或 `untyped`。如果参数的值是正确的，则存在的参数的类被定义，如果值是错误的，则不被定义。
- **默认值**。参数的默认值出现在工程工具特性窗口的参数表的参数 Id 清单的上面，如图 6-3 所示。

5. 初始化群组

一个组件必须包含可控制怎样合成一个可执行的“系统映像”的信息。例如，一个初始化程序和一个关于在初始化顺序中何时调用那个程序的描述。与组件初始化有关的

Initializations groups 或 initGroups 汇编语言定义系统开始顺序。初始化群组等级仅与运行时间相关，对 GUI 没有影响。

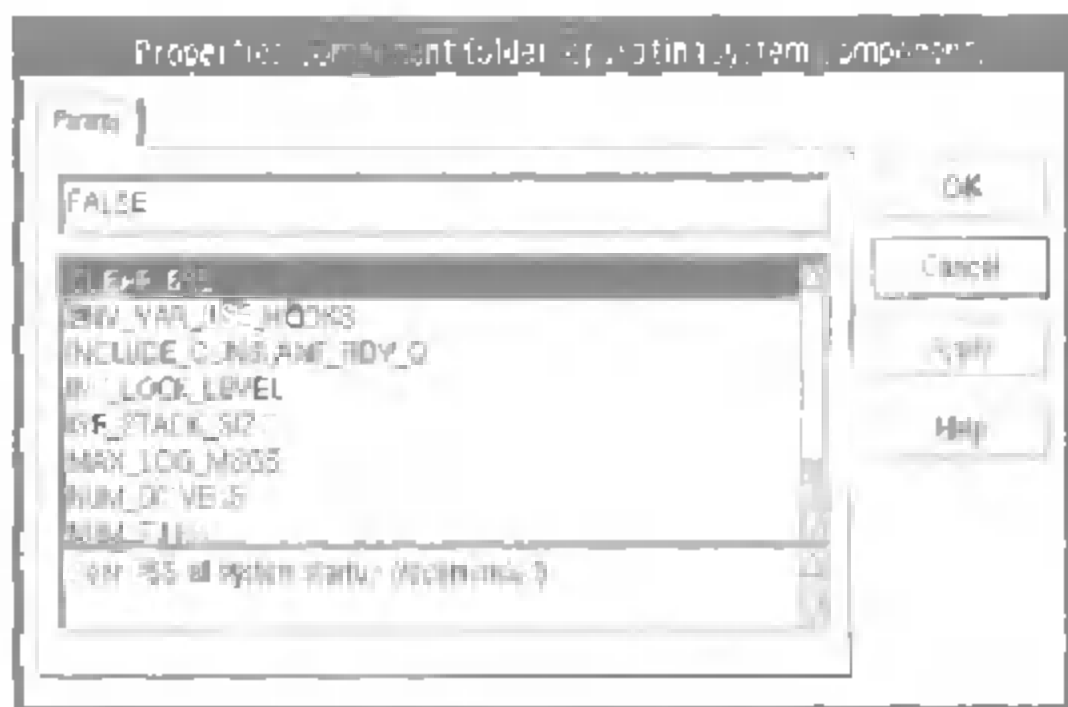


图 6-3 属性窗

对于 Tornado 1.0.1 而言，target/config/all/userConfig.c 或者 target/src/config/*.c 文件以及 #ifdef/#endif 语句决定初始化顺序，如下：

```
#if defined(INCLUDE_EXC_HANDLING) && defined(INCLUDE_EXC_TASK)
Init ( ): /* initialize exception handling */
#endif /* defined(INCLUDE_EXC_HANDLING) & defined(INCLUDE_EXC_TASK) */
#ifdef INCLUDE_LOGGING
logInit (console, MAX_LOG_MESSAGES); /* initialize logging */
#endif /* INCLUDE_LOGGING */
...
```

BSP 文件 sysLib.c 决定硬件的初始化顺序。

initGroup 是一个调用组件和其他 initGroups 的程序。此程序中的代码包含的组件及其初始化代码片段所决定的。例如：

```
InitGroup perIoeExtraInit
{INIT_RTN perIoeExtraInit ( )
  S/NOPSIS extended I/O system
  INIT_ORDER INCLUDE_EXC_TASK
  INCLUDE_LOGGING \
  ...
}
```

在这个例子中，组件 `INCLUDE_EXC_TASK`、`INCLUDE_LOGGING`，和 `INCLUDE_PIPES` 是初始化群组 `usrIosExtraInit` 的一部分。如果包含那些组件，它们的初始化程序作为这个初始化群组的一部分以指定的顺序被调用。

初始化群组中的代码被工程工具合成到文件 `prjConfig.c` 始终是每个 BSP 工程的一部分。为了看到一个系统的初始化顺序，建立这个工程，检查 `prjConfig.c` 中的发生代码。为了看到初始化顺序的默认值，请见附录 E.3 工程初始化顺序。

以下的特性描述一个初始化群组：

- 名称。
- 大纲。名称用来提供一个初始化群组的简短名称；然而，它不能出现在 GUI 的任何地方。大纲用来提供一个简洁的可读的初始化群组的定义。
- `INIT_RTN`。初始化程序初始化一个相关的组件。
- `INIT_ORDER`。按照它们初始化顺序所陈列的属于这个初始化群组的组件和初始化群组。

6.3 创建组件

这部分介绍如何去定义用户自己的组件或是修改一个已经存在的组件。当用户使用 CDL 时必须遵循一些协定。在一个组件描述文件中的组件描述后，用户必须把文件放入合适的路径以保证工程工具能读出哪些信息，并且包含等级中的适当的组件。

6.3.1 CDF 规则

当使用组件描述文件时必须遵循以下规则：

- 所有组件以 `INCLUDE_FOO` 的形式命名。
- 所有文件夹以 `FOLDER_FOO` 的形式命名。
- 所有选集以 `SELECT_FOO` 的形式命名。
- 参数的名称不能与任何其他的目标类匹配，除此以外没有任何限制。例如，用户可以使用 `FOO_XXX`，但不能使用 `INCLUDE_FOO`。
- 所有初始化群组必须以 `initFoo` 形式命名。然而，Wind River `initGroups` 使用 `usrFoo` 形式（为了以后的兼容性）。
- 所有的组件描述文件必须有 `.cdf` 后缀。
- 所有的 `.cdf` 文件名称以两个十进制数字开始，也就是 `00xxxx.cdf`。这开始的两个数字控制一个目录中 `.cdf` 文件读取的顺序。

6.3.2 CDF 的优先和路径

组件描述文件在下列两个时刻被读取：

- 当工程被创建时。
- 当组件描述文件被更改，工程创建发生时，对于用户而言 CDFs 读取的顺序是很重要的。如果两个文件描述同一个组件的同样特征，它将读最后一个而不考虑以前的。这是为了组件描述文件有优先权。后一个文件比以前所有的都优先。

优先权通常以两种补充方式建立：

- CDF 文件放在确定的目录下，这些目录以指定的顺序读取。
- 在一个这样的目录中，CDF 以文字数字的顺序读取。

工程工具源文件.cdf 在以下的目录中。这些目录以它们陈列的顺序读取。

(1) comps/VxWorks

包含所有通用 VxWorks 组件。

(2) comps/VxWorks/arch/arch

包含所有特定体系机构的 VxWorks 组件（或不考虑组件）。

(3) config/bsp

包含所有的特定 BSP 组件。

(4) 工程目录

包含所有的其他组件。

在一个目录中，为了控制文字数字的顺序，优先考虑每个.cdf 文件的两个阿拉伯数字。Wind River 保留开始的 50 个数字，也就是 00xxxx.cdf 到 49xxxx.cdf。剩下的数字，从 50 到 99 预留给第三方。.cdf 文件上面的 xx 是那些数字的占位符（也就是 on+1 优先于 on）。

这种设置优先权的方法允许工程、BSP 和 CPU 体系机构制定的通用组件或参数不予考虑。如果一个 BSP 提供商想要改变文件的最大数字，从默认值 50 到 75，可用 BSP 指定的 CDF 操作如下：

```
Parameter NUM_FILES{  
    DEFAULT    75  
}
```

如果用户正在创建一个新的 CDF，必须根据内容的特性和优先权级别把它放在适当的路径中，用户可选择的路径列于以上优先权的讨论中：

- 仅为通用 VxWorks 组件描述准备的 comps/VxWorks。
- 为特定体系机构的 VxWorks 组件描述准备的 comps/VxWorks/arch/arch。
- 为指定板的组件描述准备的 config/bsp。
- 为所有其他组件准备的工程目录。

6.3.3 定义组件

这个部分介绍定义用户自己的组件的过程。为了允许最大限度的灵活性，这里没有任何的规则来控制描述一个组件的特性的顺序或是 CDL 目标进入到组件描述文件的顺序。以下创建组件 INCLUDE_FOO 的步骤仅仅是个参考顺序；这个顺序并不是强制性的。用户也不必使用所有的特性和所描述的目标类。

1. 步骤 1：命名和准备一个大纲

为了创建一个新的组件，首先要给它命名，并给它准备一个关于它的用途的大纲。

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

在虚拟的组件 INCLUDE_FOO 中，名称就是 foo。大纲告诉用户这就是一个组件的例子。

名称和大纲仅仅影响用户的陈述；它们对初始化顺序或是依赖性没有影响。

2. 步骤 2：描述与代码相关的部分

下一步，用户应该通过定义模块和在创建时应被配置的源配置来描述用户的组件的代码部分。



注意：Tornado 2.2 支持 BSP 存根，而以前的版本不支持。

如果用户的虚构组件 INCLUDE_FOO 有与它相关的目标模块，则使用模块来指定它。用户可以以任何方式来指定它。在下面的一个例子当中，陈列了 fooLib.o 和 fooShow.o：

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

Tornado 2.x 通过绘制现存的组件关联的图形使组件从属物明显可见，这些图形在工程工具窗口中的包含组件及不包含组件当中（如图 6-4 所示）。

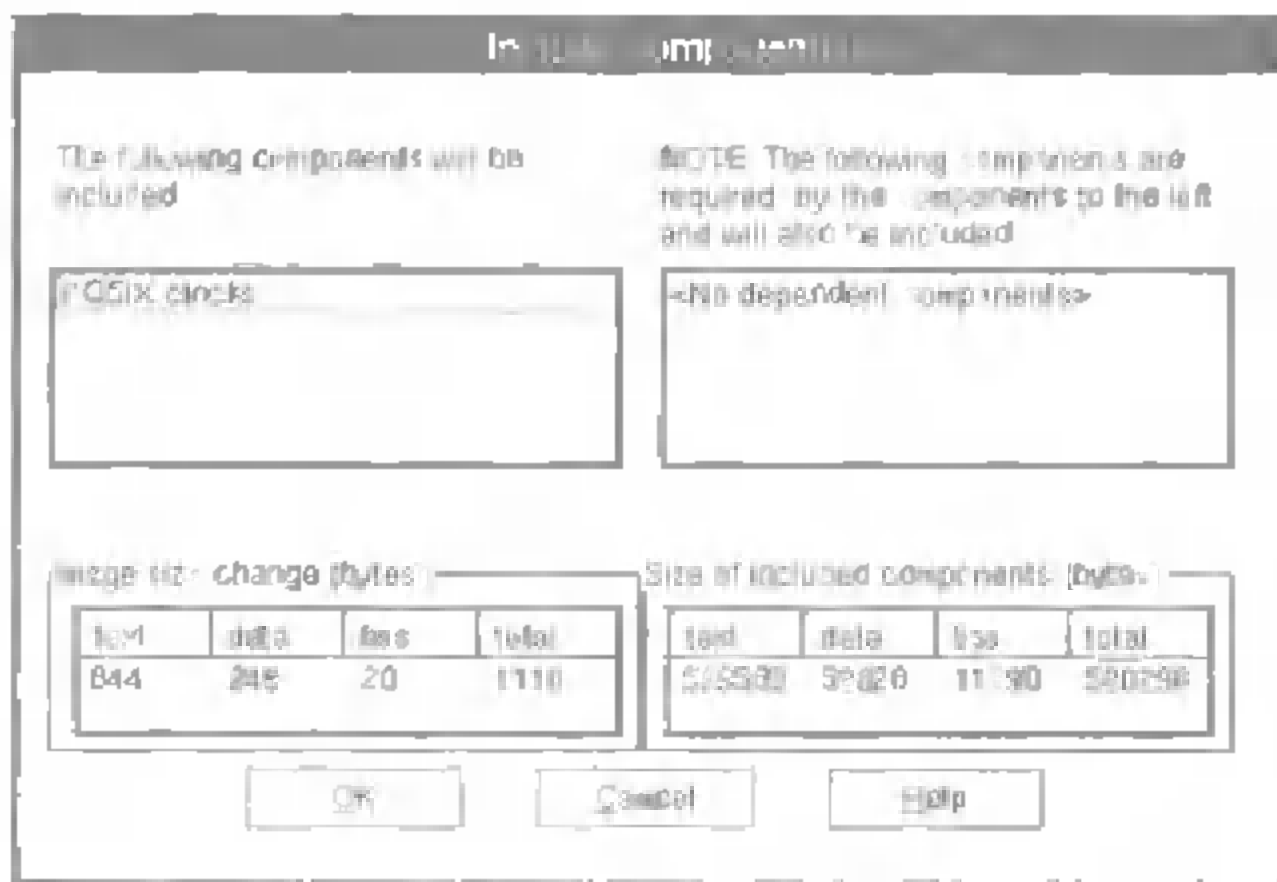


图 6-4 包含组件窗口

工程工具自动的分析目标模块依赖性来计算组件从属物：例如，如果 `fooLib.o` 有未解决的 `logMsg()`，一个组件 `INCLUDE_LOGGING` 自动地从属于这个组件将会被追踪。对于那些，目标模块格式中没有装载的组件，CDL 支持组件依赖性清单。如果源部分包含 `logMsg()` 的调用，工程工具不追踪那些从属物；相反的，应使用 `REQUIRES` 来申明 `INCLUDE_LOGGING` 的依赖关系。请见步骤 6。

如果用户的模块不放在标准位置，使用 `ARCHIVE` 来指定存档文件的名称，例如，`/path/fooLib.a`。另外的指导请参考 `ARCHIVE` 的用法，步骤 10。

⚠ 注意：第三方应创建他们自己的存档文件。不要修改 Wind River 存档文件。

使用 `HDR_FILES` 属性来指定任何与组件相关的 `.h` 文件，例如 `foo.h`。这些文件由 `prjConfig.c` 发出，提供了初始化程序的说明。

如果有应被汇编的源代码作为组件的一部分，把它放在 `a.c` 文件中，在 `CONFIGLETTES` 中指定文件名称，例如，`fooConfig.c`。配置中或初始化程序中应涉及组件参数；否则它们没有任何效用。

3. 步骤 3：设定初始值

如果用户的组件必须被初始化，使用组件目标类的 `INIT_RTN` 来指定初始化程序并征求它的意见，例如在 `fooInit(arg1,arg2)` 中。如果用户的组件需要不止一条初始化程序，创建或增加初始化代码到 `a.c` 文件中，并使用 `CONFIGLETTES` 替换。通过把初始化程序和配置联系起来用户就可以确定初始化顺序中配置的位置。

```
...
INIT_RTN fooInit(arg1, arg2);
...
```

如果用户不是用组件目标的模块，使用 `LINK_SYMS` 可使用户获取一个连接存档文件的目标模块。系统产生一个未解决的符号 `_fooRtn1`，从而产生了一个连接通过从文档文件中提取用户模块来解决它。

```
...
LINK_SYMS _fooRtn1
...
```

4. 步骤 4: 建立初始化顺序

初始化顺序是很重要的。用户可以控制在初始化顺序中何时组件以 `_INIT_ORDER` 初始化。一个被限制在一个已存在的初始化群组中的组件（或初始化群组），使用默认值 `_INIT_ORDER` 最后初始化。这就是代表性地想要的作用；然而，用户可以通过使用 `INIT_BEFORE` 而不考虑这些行为。

```
...
_INIT_ORDER usrRoot
INIT_BEFORE INCLUDE_USER_APPL
...
```

在这个例子中，`INCLUDE_FOO` 被申明为 `usrRoot` 初始化群组的成员之一。`INIT_BEFORE` 有很好的控制性，`INCLUDE_FOO` 在 `INCLUDE_USER_APPL` 之前被初始化。

用户还可以有另一种选择，创建一个新的初始化群组并申明 `INCLUDE_FOO` 为一个成员；然而，用户本来可以申明这个新的初始化群组为一个已存在的初始化群组的一个成员。更多的有关初始化群组的信息请见 E. 组件语言。



注意： `INIT_BEFORE` 只影响初始化群组内的顺序。不要涉及不在初始化群组内的组件；那时没有任何效用的。

`INIT_AFTER` 在 Tornado 2.x 执行时无效用。

Tornado 2.x 的默认初始化顺序请见 E.3 工程初始化顺序。

5. 步骤 5: 链接有帮助的文件

以帮助指定相关的参考条目（以 HTML 格式）。


```
...  
HELP fooMan.html  
...
```

默认地，一个结构自动的包含相关的值的参考条目，MODULES 和 INTT_RTN 声明了这些条目。在 INCLUDE_FOO 的场合中，除被帮助指定的 fooMan.html 之外，结构把 fooLib 和 fooShow 库和 fooInit() 程序联系起来。

6. 步骤 6：定义从属物

使用 REQUIRES, EXCLUDES 和 INCLUDE_WHEN 来申明组件之间的从属物（请参考步骤 2 学习怎样一个工程工具能自动的配置目标模块相关的从属物）。

当一个组件不装载在目标模块格式中时工程工具并不追踪隐藏的从属物。同样地，当指示器在运行引用符号时，也不追踪从属物。在这两种情况下要求用户明确的指明从属物。

```
...  
REQUIRES INCLUDE_LOGGING  
EXCLUDES INCLUDE_SPY  
INCLUDE_WHEN INCLUDE_POSIX_AIO INCLUDE_POSIX_MQ  
...
```

在这个例子中，REQUIRES 申明 INCLUDE_LOGGING 必须被配置，INCLUDE_FOO.EXCLUDES 申明 INCLUDE_SPY 不能以 INCLUDE_FOO 配置。INCLUDE_WHEN 告诉系统，当组件 INCLUDE_POSIX_AIO 和 INCLUDE_POSIX_MQ 被包含时，INCLUDE_FOO 也必须被包含。



注意：总体而言，工程工具是被设计来增加选择组件的灵活性的，也就是说，来增加可测量性；指定 REQUIRES 关联会降低灵活性。在执行 REQUIRES 之前确定在用户的情况下它是最好的程序。

7. 步骤 7：陈列相关的参数

在组件目标中，使用 CFG_PARAMS 来申明所有相关的参数，例如，FOO_MAX_COUNT。

```
...  
CFG_PARAMS FOO_MAX_COUNT  
...
```

8. 步骤 8：定义参数

对于每个 CFG_PARAMS 申明的参数，创建一个参数目标来描述它。使用 NAME 来提

提供一个名称。

使用 TYPE 来指定数据类，即 int, uint, bool, string, exists 或 untyped。

使用 DEFAULT 来指定每个参数的默认值。

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

 **警告：** 如果一个组件包含一个没有赋值的参数，它将被认为是被错误配置。确定要赋予默认值，否则没有合理地默认值，必须强迫用户明确的设定它。

9. 步骤 9：定义群组成员

一个组件必须与一个文件夹或一个选集联系，否则它在工程工具 GUI 中是看不到的。举个例子，由于组件在群组中合理的内涵物必须分配一个组件给一个文件夹，这些内涵物被文件夹定义，它们有相似的或共享的功能。通过包含一个文件夹中的一个组件用户可以使用户把其他组件同时放在这个群组中，这是通过使用文件夹目标的 DEFAULTS 申明它们为文件夹的默认值办到的。

```
...
_CHILDREN FOLDER_ROOT
...
```

CHILDREN 申明 INCLUDE_FOO 是文件夹 FOLDER_ROOT 的子组件。预先的底线 (“_”) 用 CHILDREN 颠倒相互关联。用户也可以用 _DEFAULTS 和 _CHILDREN 一起指定一个组件作为一个文件夹的默认组件。关于 CDL 协定的更多信息请参见“6.4 发布组件”。

如果用户认为一个组件变得越来越复杂了，用户可以把它分为一系列的分配给文件夹或选集目标的组件。用户可以增加群组或从用户的配置中把群组去除，这时它作为一个单位而不是单个的组件。

对于文件夹而言，如果配置了群组则 DEFAULTS 可指定包含的基本组件。

对于选集而言，如果用户不提供其他的可选的值 DEFAULTS 可指定包含的组件来满足计数（被 COUNT 申明）。

在一个选集群组中，COUNT 指定了包含组件的最小数目和最大数目。如果用户超过了这些限制，系统标志这些选集位错误配置。

```
Selection SELECT_FOO {
```

```
NAME Foo type
SYNOPSIS Select the type of desired FOO support
COUNT 0-1
CHILDREN INCLUDE_FOO_TYPE1 \
INCLUDE_FOO_TYPE2 \
INCLUDE_FOO_TYPE3
DEFAULTS INCLUDE_FOO_TYPE1
}
```

10. 步骤 10: 创建虚拟组件

只有当工程工具与包含的组件联系起来时它才会分析档案文件。这就产生了一个子问题: 为了了解一个特殊的档案文件, 工程工具需要在组件真正被加进去之前分析它们。也就是说, 如果用户把 ARCHIVE 声明的一个组件加进去的话, 配置分析会在不知道 ARCHIVE 的值的条件下完成。所以, 如果用户的组件包含一个档案文件以及几个目标模块的话, 用户应该创建一个虚拟组件, 这个虚拟组件是始终要被包含的, 它使工程工具知道一个新的档案文件应被读取。这个组件叫做 INSTALL_FOO。它应该仅仅包含 NAME, SYNOPSIS 和 ARCHIVE。在 INSTALL_FOO 被加进去之前用户不能够增加来自相同的档案文件其他组件。



警告: 不要直接改变 Wind River-supplied CDFs。使用赋名来创建一个文件, 这个文件的优先权高于 Wind River-supplied 地默认组件。

11. 步骤 11: 产生工程文件

工程工具基于以下两点产生每个工程的工程配置文件 (prjConfig.c):

这一系列的组件是用它从组件等级中选出来的。

信息来自相关组件的描述文件。

为了产生这个文件, 点击工程工具的 Build 制表符的 Build 按钮。



注意: 产生的工程文件 (prjConfig.c) 不需要传统静态的配置文件的与处理程序 #ifdefs, 因此很容易读取。

6.3.4 修改组件

在任何更高优先级别的 CDF 文件中通过重新指定以存在的组件来修改它们。传统的第三部分的 CDF 文件是被最后读取, 因此有最高的优先权。使用赋名可创建一个更高优先权

的 CDF 文件,它超越 Wind River 的任何默认组件。关于文件赋名和优先权的信息请见“6.3.2 CDF 的优先和路径”。

在下面的这个例子中,标准 Wind River 组件 INCLUDE_IO_SYSTEM 中的打开文件描述符 (NUM_FILES) 的数量被更改了,默认值是 50。

```
Parameter NUM_FILES {
  DEFAULT 75
}
```

通过把例子中的那一系列代码加到第三部分 CDF 文件中,如果组件已经存在于配置中通过去除或是增加组件,以及通过重建工程,NUM_FILES 的值被改变到 75。初始的 Wind River CDF 文件 00VxWorks.cdf 没有被改变;默认值被改变是因为第三部分文件有更高的级别。除非被特殊的重新定义,其他的特征值依然保持原样。



警告: 不要改变 target/config/comps/src 中的 Wind River-supplied 源配置文件,如果有必要的话,使用一个 BSP 或工程 CDF 来取消 CONFIGLETES 的申明,使用 BSP_STUB (只在 Tomado 2.2 中可用) 作为替换。于是一个类文件的副本被放到 BSP 目录中,它可以被改变而不影响其他的 BSPs,也不改变源文件。
不要直接改变 Wind River-supplied CDFs。使用赋名来创建一个文件,它的优先权高于 Wind River-supplied 的默认组件。

6.4 发布组件

发布一个 Tomado 组件的过程排除对一个现存的配置文件修改的可能性。新的组件描述文件应该与现存的文件无关,有以下两种例外:

- 新的组件目标与一个现存的文件夹或选集相关联。
- 新的组件目标初始化程序与一个现存的初始化群组相关联。

一个特殊的 CDL 是一个新的组件目标连接到一个现存的文件夹或选集或一个现存的初始化群组中,并且它不修改现存的元素。通过预先在某个组件划底线 (“_”),用户可以颠倒这个特性的意义。例如,如果已经有一个文件夹 FOLDER_EXISTING 和一个初始化群组 initExisting,用户可以把一个新组件连接到它们上面(不修改它们),如下所示:

```
Component INCLUDE_FOO {
  ...
  _CHILDREN FOLDER_EXISTING
```

```
_INIT_ORDER initExisting  
}
```

CHILDREN 与 FOLDEREXISTING 有相应的关联：那就是，_CHILDREN 与“父母”等同。也就是说它提供与在 children 清单上包含组件 INCLUDE_FOO 的 FOLDER_EXISTING 相同的效果，而且它不修改任何包含 FOLDER_EXISTING 目标的 CDF。

如果用户的工程或 VxWorks 系统包含来自以前版本的 Tornado 组件（也就是 2.0 以前的版本），为了保证以后的兼容性，使用老的 INCLUDE_XXX 格式来创建组件 ID。在 6.3.3 定义组件中，INCLUDE_FOO 和旧配置中的与处理程序宏都是组件 ID。

用户可以一次发布不止一个组件，因为一个 CDF 可定义不止一个组件。用户还可以同时发布任何数量的 CDFs。

6.4.1 测试新组件

用户可以进行好几个测试来校验组件的写入正确：

1. 核对句法和语义

这个支配线测试是最简单的校验测试。首先编辑文件 installDir/host/resource/tcl/app-config/Project/cmpTestLib.tcl，设置 TCL 可变的 bsp 文件一个有效的 BSP 目录名（例如 mv162）。然后运行 cmpTest。

```
% cd $(WIND_BASE)/host/resource/tcl/app-config/Project  
% wtxtcl  
wtxtcl> source cmpTestLib.tcl  
wtxtcl> cmpTest
```

句法和语义的 CmpTest 测试会在用户的.cdf 文件（事实上在所有的.cdf 文件中）出错。根据测试输出作相应的改变。一直运行这个测试直到没有错误。

2. 核对组件从属物

用户可以通过运行 cmpInfo 来测试用户组件中的可测量的 bug，如下所示：

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

这报告反馈给用户组件的子树（用户的组件需要的其他组件）和前树（需要用户的组件的其他组件）的信息。

3. 核对工程工具组件等级

通过核对工程工具组件等级来校验用户所增加的选集，文件夹和新组件被正确包含。打开工程工具并进入 VxWorks。

看看用户的新组件是怎样出现在文件夹树中的。调用 `Properties` 来核对和一个组件相关的参数和它们的默认值。

如果用户已经增加了一个包含组件的文件夹，并且已经把文件夹放在用户的配置中，工程工具等级会默认地用黑体字陈列那个文件夹的所有组件（也就是 `DEFAULTS` 的值）。校验因而发生的映像构造和 `boots`。

6.4.2 包装组件

更多的包装组件的信息请见“10.3 组件封装”。

第7章 确认测试

7.1 简介

本章记述了板级支持包(BSP)的确认测试序列 (BSP VTS)。BSP VTS 是一个用于测试板的由主机引起的测试程序的集合。这些测试的目的是提供一个客观的报告，可以用它来判断一个 VxWorks BSP 的基本程序。BSP VTS 有高度的自动化特性使之能帮助确定测试的可重复性。一旦开始了，一个 BSP 测试不再需要用户的干涉。

这个版本的 BSP VTS 仅仅只在 Tornado 环境中运行，它支持 UNIX 和 Windows 主机。为了提供 UNIX 和 Windows 主机的可携带性，BSP VTS 被写入 Tcl/WTX。这些测试检查 BSPs 的基本程序，追踪和报告它们发生的错误。

如果这些测试不足以满足你的需求，可以编辑源代码来增强现存的测试或者增加你自己的测试。然而，如果选择修改 BSP VTS，应该保证这些测试的基本装置的完整性。这些测试是 BSPs 被评判的标准。

其他文件

BSP VTS 和它的参考组件测试的参考信息，请见 J.BSP 测试组确认参考信息。更多的关于 Tcl 语言的信息请见 Tornado 用户指导：Tcl 或任何标准的 Tcl 参考信息，例如 Brent B.Welch 著的《Tcl 和 TK 中的编程实战》，或 John Ousterhout 著的《Tcl 和 Tk Toolkit》。

7.2 后台

这个部分用来提供 BSP VTS 的后台信息，包括设计目标板，执行工具和一个测试环境的总体配置。

7.2.1 设计目标板

下面是指导 BSP VTS 运行的设计目标板的清单。注意到这些目标板能帮助你更好地理解怎样设置这些测试，运行测试以及解释结果。如果决定修改 BSP VTS 你应该把这些设计目标板记住。

- Tornado 是可兼容的。BSP VTS 应该在 Tornado 2.0 或更新的版本中运行。
- 广泛可用的。BSP VTS 应该被广范围的用户所使用（板集卖主，终端用户，以及 Wind River 的开发者）。另外，这些测试应该能够处理全线（VME）和独立的配置。
- 可移植的。BSP VTS 应该既是可对象建构又可主机建构的。
- 可重复的。所有的测试都应该是可重复的。因此，对于一个给定的测试环境，不管它们在哪里或是何时运行，都应该同样的执行任务。
- 自动的。用户应该能够运行来自单一要求的 BSP VTS，测试结果应该放在一个测试日志中。用户应该可以运行来自单一要求的子测试。
- 特定 BSP。BSP 应该仅仅与属于 BSP 运行和配置的 VxWorks 方面相关。
- 主机控制的。这个测试环境应包含单一的测试主机，一个 VxWorks 支持的主机，这个主机配置必要的硬件和软件来使 BSP VTS 正确的执行任务。它包含：
 - 合适的主机工具（编译器或与之类似的）；
 - 完整的 Tornado 开发环境；
 - 完整的正确版本的 BSP 开发工具箱；
 - 完整的在测试下的 BSP 源文件。

这个测试主机应该控制测试的执行。这是很必要的，因为没有关于目标板的任何设想，除非它能够运行 VxWorks 以及能与主机和目标板控制台串行端口的目标板服务程序相互作用。

- 可以使用第二目标板机。Network, busTas 和 rlogin 测试要求使用一个第二目标板机的板集，即参考板集。这个参考板即必须能运行以前的有效的 BSP。
- 有目标板的。目标板服务程序 BSP VTS rlogin 测试要求有一个 VxWorks 目标板。

7.2.2 有效测试地址的软件体系结构

这个部分描述了典型的单一目标板和多目标板的 BSP VTS 软件体系结构。

1. 单一目标板

图 7-1 显示了单一目标板的总体配置。

典型的测试情况如下：

- (1) Tel/WTX 测试在 UNIX/Windows 主机上运行。
- (2) 这个测试连接到目标板服务程序，它通过使用目标板工具联系终端目标板机的目标板服务程序。如果要求串行的通信，这个测试使用主机上一个一个串行的线（UNIX 上的 ttya 或 Windows 上的 COM1）来打开目标板/tyCo/0 的通道或者使用控制台配置的其他端口。
- (3) 当要求控制台和目标板机相互作用时，测试信息将发送 WTX 或者 WindSh 请求和读控制台输出来联系每个目标板。

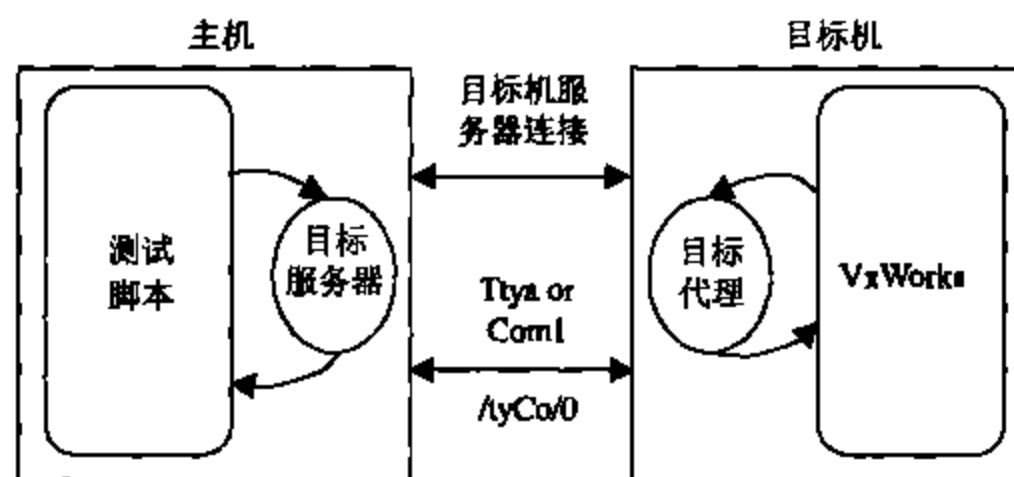


图 7-1 BSP VTS 软件体系结构举例：单目标机

2. 多个目标板

图 7-2 显示了两个目标板的测试系统的总体配置。

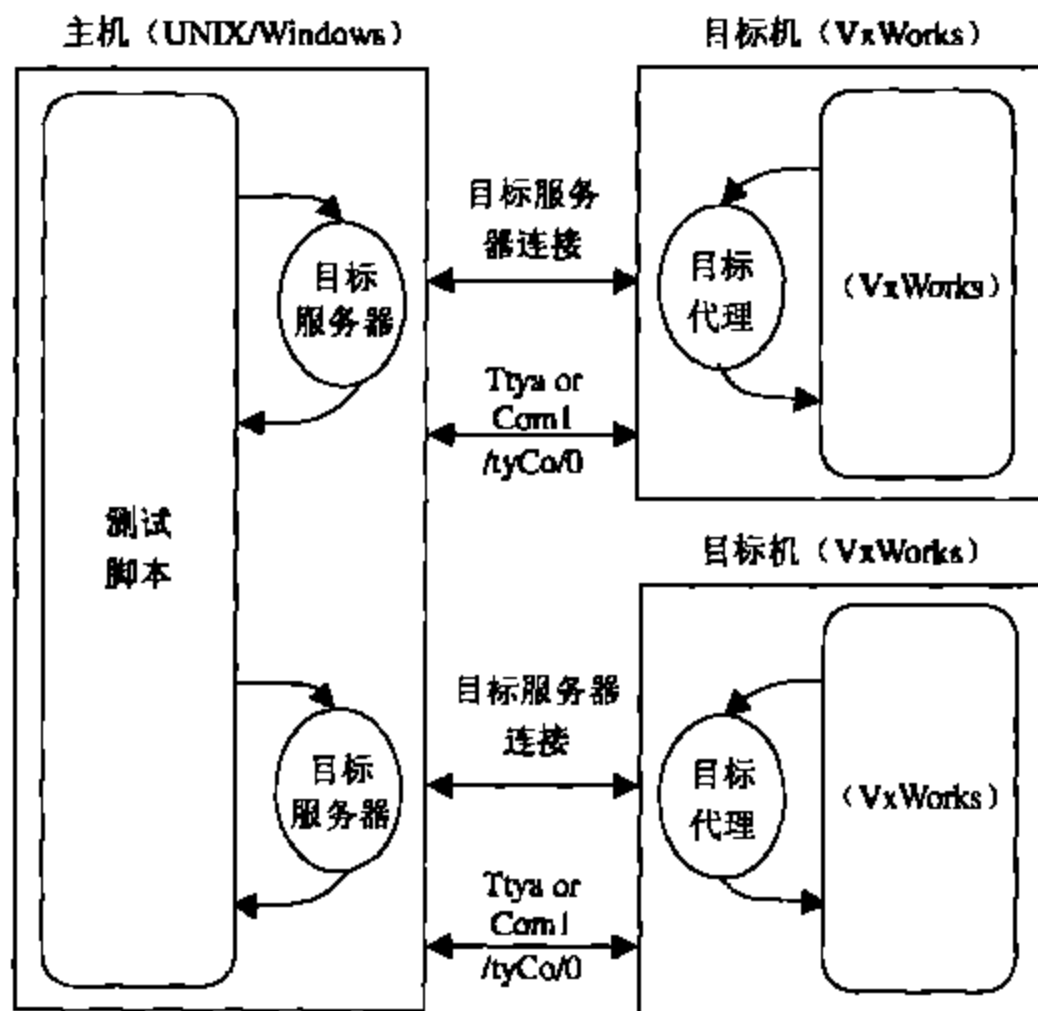


图 7-2 BSP VTS 软件体系结构举例：双目标机

典型的测试情况如下：

- (1) Tcl/WTX 测试在 UNIX 或 Windows 主机上运行。
- (2) 这个测试连接到第一个目标板服务程序，它通过使用目标板代理来联系终端目标板服务程序到目标板 0。如果要求串行的通信，一个从 ttya (UNIX) 或 COM1 (Windows) 到目标板 0's/tyCo/0 或其他端口的通道被配置，控制台经由第一条串行路线被打开。
- (3) 这个测试连接到第二个目标板服务程序，它通过使用目标板代理来联系终端目标板服务程序到目标板 1。如果要求串行的通信，一个从 ttya (UNIX) 或 COM1 (Windows)

到目标板 l's/tyCo/0 或其他端口的通道被配置, 控制台经由第二条串行路线被打开。

(4) 当要求控制台与目标板相互作用时, 测试信息将发送 WTX 或 WindSh 请求和读控制台输出来联系每个目标板。

7.3 配置 BSP VTS

这个部分描述了以下内容:

- 需要的硬件配置
- BSP VTS 的安装
- 配置系统: 主机, 目标板, VxWorks 和 BSP VTS。

与《Tornado 用户指导: 安装和启动》联系起来学习这个部分。

7.3.1 硬件设置

BSP VTS 假定一个最小的目标板系统设置。这可以是一个全线环境 (传统的) 或一个独立的配置。对于一些测试, 例如底板测试, 需要第二个 CPU 板。

1. 全总线

图 7-3 显示了一个典型的全线配置, 它包括以下内容:

- 类。一个有底板的板卡罩, 典型的例子是 VMEbus 和电源。
- CPU 板。已安装 VxWorks 的单板计算机。这个板运行测试下的 BSP。
- 以太网卡。一个以太网控制板 (一些 CPU 板包括一个在板上的以太网控制程序)。
- 串行端口。在工作台或是目标板的一个或使更多的串行的端口。这是初始化安装和测试执行和串行端口测试所要求的。

2. 独立的板

图 7-4 显示了一个典型的独立的配置:

- CPU 板。已安装 VxWorks 的独立的计算机板 (目标板)。它运行测试下的 BSP。它包含一个以太网控制程序。
- 串行端口。在工作台和目标板中的一个或使更多的串行的端口。这是初始化安装和测试执行和串行端口测试所需要的。

BSP VTS 支持在独立的 ROM 中的 VxWorks。为了做到这个, 当启动目标板时 BSP VTS 检查并使用独立的 ROM 映像。然而, 只有当你已经建构了 VxWorks 的必要的配置宏, 这样才能允许目标板服务程序和目标板联系起来。例如, 对于一个 VxWorks 独立的目标板和目标板服务程序的网络连接, 定义宏为 INCLUDE_NETWORK, STANDALONE_NET 和

INCLUDE_WDB。

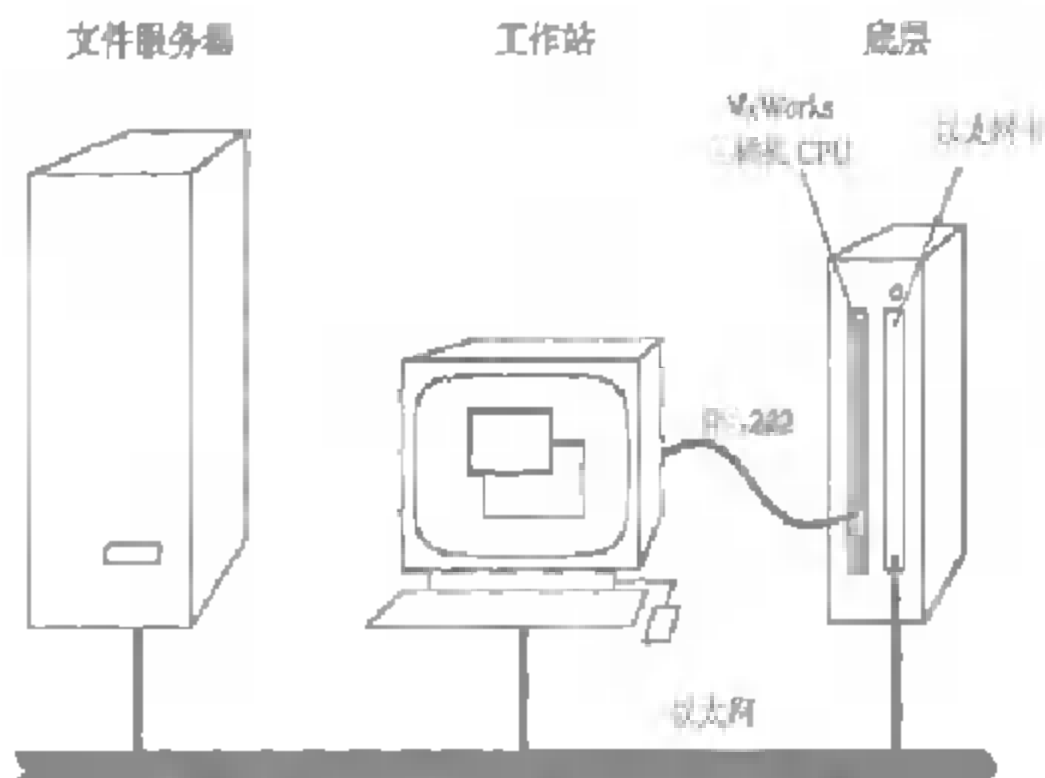


图 7-3 典型 BSP 测试配置: VME

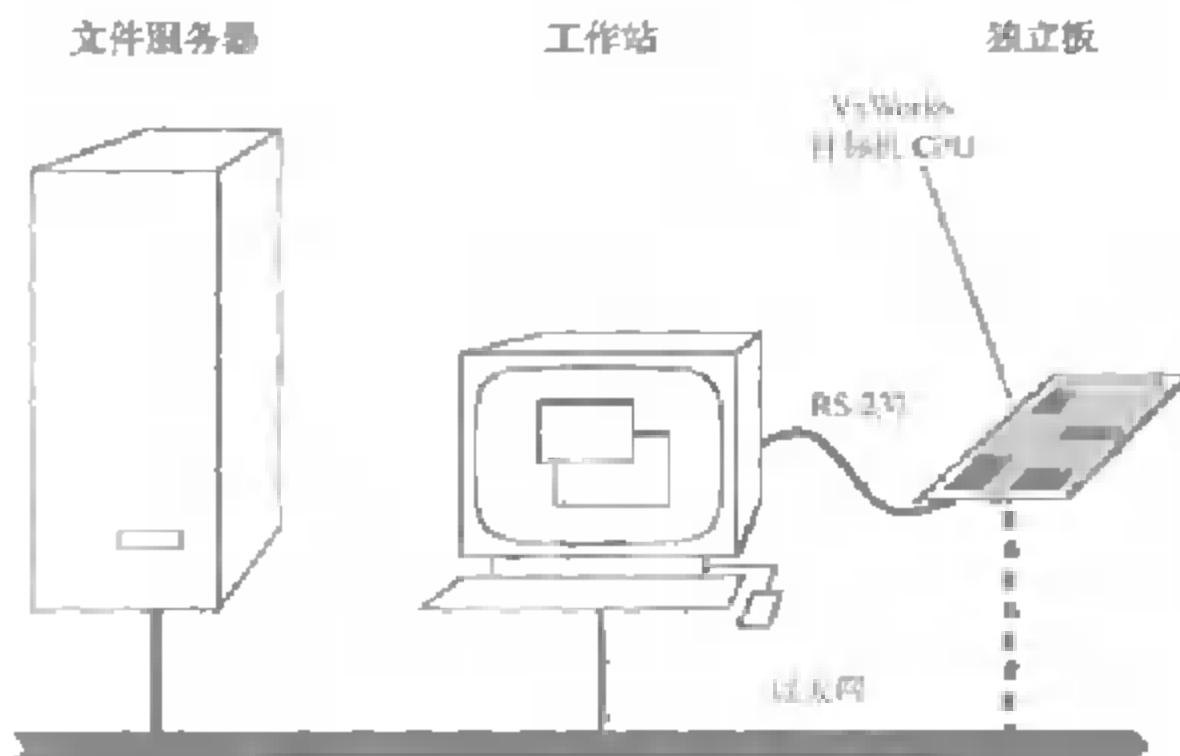


图 7-4 标准 BSP 测试配置: 独立式

所有应用到一个 VxWorks 的目标板的测试需要一个在那个目标板主机上运行的目标板服务程序。终端目标板服务程序与目标板的通信是通过以太网或使用 SLIP 的串行的连接或其他有效的目标板服务程序能配置的通信装置来实现的。更多的有关配置目标板服务程序的信息请见《Tornado 用户指导: 目标板服务程序》(Windows 版本)或《Tornado 用户指导: 安装和启动》(UNIX 版本)。有了这个连接, BSP VTS 就可以通过目标板服务程序来建构和承载测试库 (pkLib.o)。更多的信息请见配置 VxWorks。

7.3.2 安装 BSP VTS

安装 BSP VTS 和其他的 Wind River 产品的大体步骤是一样的。请见 Tornado 安装指导的软件安装指导。如果 BSP 以前没有被安装过，参考和 BSP 一起的安装指导。

1. 目录和文件

BSP VTS 测试在 `host/src/test/bsp` `Val/src/test` 目录下。建立 BSP VTS 的源文件 Tcl 在 `host/src/tcl` 目录下。Tclserial 库的源文件在 `host/src/test/tclserial` 下。运行 BSP 的工具和执行文件的二进制代码在 `host/hostType/bin` 目录下。BSP VTS 的配置文件在 `host/resource/test/bsp` `Val` 目录下。

2. 主机支持

BSP VTS 在 Tcl, 7.6 版本中运行。BSP VTS 脚本以 UNIX 脚本开始，然后在 `wtxtcl` 下运行 Tcl，`wtxtcl` 是 WTX 协议对 Tcl 语言的约束。事实上，BSPVTS 脚本作为连接到目标板服务程序的工具运行。

Tclserial 提供了一系列能够与串行端口通信的程序。如果你正在使用一个被支持的主机配置则没有必要产生 Tcl 库和 `tclserial` 库。为所支持的主机配置的 Wind River 在 Tcl 库和 `tclserial` 库之前安装，被放入 `host/hostType/bin` 目录下。^①

完整的源代码是用来帮助维护和有待扩展的。Tcl 和 `tclserial` 库是可移植的，它在主机上而不是被支持的主机上汇编。

Tcl 的完整源代码在 `host/src/tcl` 目录下可用。为了在不被 Wind River 支持的主机上建构和测试 Tcl，为了被选择的 `hostType` 运行这个目录下的 `make`。虽然 Tcl 应该运行时不发生故障，但是自我测试可支持一定的错误而不是很明显的影响 BSP 的运行。

建立一个 `tclserial` 库，为了被选择的 `hostType` 运行 `host/src/test/tclserial` 目录下的 `make`。



注意：Wind River 对 Tcl 源代码作了一些修改，使之允许 Tcl 在不同的主机上运行。Wind River 对这些更改的有效性并不予以保证，因为 Tcl 是一个公开的软件，它不是被 Wind River 完全支持的。不同的主机上安装的 Tcl 在运行被供应的测试和被测试的指定的主机上是完整的。任何超越这个目的的 Tcl 的使用都没有经过测试，是不被支持的。

^① Tclserial 库是一个共享的或连接的库，它适合于平台（Solaris 或 SunOS 系统下的 `pkgctclserial.so`，HP-UX 下的 `pkgctclserial.sl`，Windows 下的 `pkgctclserial.dll`）。

7.3.3 系统配置

这个部分描述了怎样配置一个主机系统，目标板硬件，VxWorks 和 BSP VTS。

1. 配置主机系统

除了 BSP VTS，主机还要求标准的 Tornado 开发环境。关于配置主机系统的信息请见《Tornado 用户指导：安装和启动》。

2. 配置目标板硬件

VxWorks 目标板硬件仅仅需要最小的配置。BSP VTS 在 Tornado 环境中运行。大部分时候它使用目标板服务程序连接来与目标板通信。这些使用控制台输出的测试要求串行的连接到主机的控制台。^① rlogin, network 和 busTas 测试要求主机连接到两个串行的控制台上。



注意：目标板服务程序必须在开始 BSP VTS 之前运行。

一些测试，例如 error1.tcl 和 bootline.tcl，必须重新启动目标板。然而，第一次运行 BSP VTS 之前，应该手动的启动 VxWorks 来核对启动参数是否设置正确。除此之外，在运行 BSP VTS 之前，确定 VxWorks 目标板已经开机了并且对控制台做出回应。但是目标板并不需要启动。

Serial.tcl 测试要求一个循环连接来终止所有闲置的串行端口（那些没被 VxWorks 控制台或 SLIP 使用的端口）。为了做到这个，运行一个来自 transmit/output 到 receive/input 的跳线。端口和硬件握手时，需要运行从 RTS 到 CTS 的跳线。



警告：scsi.tcl 测试通过写入和读出每个 SCSI 驱动程序的每个地址来工作。

它擦除驱动程序。确定在运行这个测试之前指定合适的 SCSI 驱动程序。

3. 配置 VxWorks

运行 BSP VTS 需要一系列的补充程序。Target/src/test/bsp Val/pkLib.o 提供这些程序的源代码。BSP VTS 在 target/config/bspname/pkLib.o 中寻找这个代码的目标板版本。如果 BSP VTS 没有找到这个目标板文件，它会自动的复制源代码到 target/config/bspname 目录下并汇编它。如果对 pkLib.c 做任何的修改，则必须复制文件到 target/config/bspname 并重新汇编它。

^①你必须重启目标板并且与主机上的控制台通信。当你配置和建立目标板的启动 ROM 映像时记住这个。

多数 BSP VTS 测试能使用默认的 VxWorks 配置。以下陈列了一些例外:

(1) 网络

要求能运行 ping 和 pkUdpEchoTest 的第二个目标板。当建立参考板级的 VxWorks (从机) 时, 定义 INCLUDE_PING 宏。运行这两个目标板的 VxWorks 应该已经定义了 INCLUDE_SHOW 宏。

(2) 时标

要求在建立 VxWorks 和 pkLib.c 之前定义 INCLUDE_TIMESTAMP。如果这个宏没有被定义, BSP VTS 将跳过时标测试。不会产生严重的错误。

(3) 远程登录命令

要求使用目标板。在建立这个测试的 VxWorks 之前, 必须定义以下宏:

- INCLUDE_SHELL
- INCLUDE_RLOGIN
- INCLUDE_SHOW_ROUTINES
- INCLUDE_NET_SYM_TBL
- INCLUDE_LOADER

(4) SCSI

要求 VxWorks 映像和 pkLib.c 必须在包含的 SCSI 的支持下建立。对于大多数的 BSP 而言, 能通过定义 INCLUDE_SCSI 来做到这个。

(5) busTas

要求这两个目标板的 VxWorks 映像包含共享的记忆组件。定义以下宏:

- INCLUDE_SM_NET
- INCLUDE_SM_SEQ_ADDR
- INCLUDE_SM_NET_SHOW

一个目标板的 VxWorks 映像也应该定义 SM_OFF_BOARD 为 TRUE, 其他的定义为 FALSE。

更多的信息请见《J.BSP 测试确定参考条目》的测试描述。

4. 配置 BSP VTS

BSP VTS 是通过一些作为环境变量的参数来配置。所有的这些参数都是在配置文件(以下要说明的)中定义的, 没有一个例外。可选择的环境变量 TEST_LOG_FILE 提供给用户一种指定 BSP VTS 测试被保存在何处的方法, 使用合适的要求去设置这个变量到文件名(包含一个详尽的路径), 测试结果被登入日志。当变量没有被设置时, 测试被登入到 target/config/bspname/vtsLogs/bsp ValidationLog.PID。在这两种情况之一, 屏幕输出作为日志文件被复制到同样的地址下, 与文件名的搜索路径_screenDump 的日志文件一样命名。

控制单一目标板的 BSP VTS 的行为的配置参数放在 host/resource/test/bsp Val/bspName.T1 文件下。如果一个测试需要两个目标板, 则 host/resource/test/bsp Val/bspName.T2 文件提供

了第二个目标板的参数。^④

这些参数包含 VxWorks 映像启动的路径，主机和目标板的特性，以及测试怎么进行的具体过程。很多参数是可选择的，或者有在多数情况下都合理的默认值。有些参数仅对某些测试是适用的。指定目标板的参数使用前缀 T1_和 T2_来区分一个目标板和其他目标板。

表 7-1 和表 7-2 总结了可在 bspName.T1 或 bspName.T2 文件中设置的配置参数。对于陈列默认值的参数，甚至当参数没被定义时那些默认值依然起作用。除此以外，需要为你的串行连接和超时值修改配置参数。这个部分的结果描述了这些参数。

配置参数的附加信息请看 bspName.T1 或 bspName.T2 中的论述。与指定测试相关的参数的信息请看 J.BSP 提供的测试参考条目。

5. 配置串行的连接

需要修改以下串行连接参数来适应你的主机：

{T1,T2}_SER_DEVICE 串行设备连接的主机名称。

{T1,T2}_SER_BAUD 串行设备连接的波特率。

也可以选择使用另一种方法，即指定串行参数为 bspVal 中的要求线选项。请看《J.BSP 测试参考条目》的详细清单。

6. 配置超时值

以下配置参数决定了 BSP 测试应等待多长时间的 VxWorks 启动：

{T1,T2}_TMO_BOOT 启动超时。

如果在启动后没有在规定时间内出现 VxWorks 命令，将出现一个致命的错误。因此，如果目标板启动超越串行线 Internet 协议 (SLIP)，那么需要增加超时。

另外，还有一些与个体测试相关的超时参数选项。这些参数包含宏名称中的 TMO。对于具有大型设备或者较低访问速度的目标板，需要将超时值变量设得大一些。这些参数在 host/resource/test/bsp Val 中的 bspName.T1 或 bspName.T2 文件中设置。

表 7-1 确认测试包配置参数：目标板引导参数

参 数	默认值	描 述
{T1, T2}_BOOT_B	N/A	底板 inet (b)
{T1, T2}_BOOT_DEVICE	N/A	引导设备
{T1, T2}_BOOT_E	N/A	以太网 inet (e)
{T1, T2}_BOOT_F	N/A	标志符 (f)
{T1, T2}_BOOT_FILE	N/A	VxWorks 引导映像
{T1, T2}_BOOT_G	N/A	网关 inet (g)
{T1, T2}_BOOT_H	N/A	主机 inet (h)
{T1, T2}_BOOT_HOST	N/A	主机名称

当创建一个 bspname.T1 或 bspname.T2 文件，你能根据需要复制 mv147.T1 和 mv147.T2 文件和编辑副本。

续表

参 数	默认值	描 述
{T1, T2}_BOOT_O	N/A	其他 (o)
{T1, T2}_BOOT_PROCNUM	N/A	处理器序号
{T1, T2}_BOOT_PW	N/A	密码
{T1, T2}_BOOT_S	N/A	启动脚本 (s)
{T1, T2}_BOOT_TN	N/A	目标板名称 (tn)
{T1, T2}_BOOT_U	N/A	用户 (u)

表 7-2 确认测试包配置参数：特定目标板参数

参 数	默 认	描 述
T1_BOOT_LOCAL_BUS_ERR_MSG	N/A	引导 ROM 级的本地总线错误信息关键字。见 bootline.tcl。
T1_BOOT_LOCAL_ERR_ADRS	N/A	引导 ROM 级的引起总线错误的本地存储地址。见 bootline.tcl。
T1_BOOT_OFFBOARD_BUS_ERR_MSG	N/A	引导 ROM 级的板外总线错误信息关键字。见 bootline.tcl。
T1_BOOT_OFFBOARD_ERR_ADRS	N/A	引导 ROM 级的引起总线错误的板外地址。见 bootline.tcl。
T1_BUS_MASTER	TRUE	表明那个目标板是总线主管。见 busTas.tcl。
T1_CATASTROPHIC_ERR_MSG	N/A	致命总线错误信息关键字。见 error2.tcl。
T1_COUNT_ADRS	not used	共享计数器的主管地址。见 busTas.tcl。
T1_DIV_ZERO_MSG	N/A	除 0 错误信息关键字。见 error1.tcl。
T1_EXTRA_AUXCLK	25	测试的另一速度。见 auxClock.tcl。
T1_EXTRA_SYSCLK	20	测试的另一速度。见 sysClock.tcl。
T1_LOCAL_BUS_ERR_MSG	N/A	本地总线错误信息关键字。见 error1.tcl。
T1_LOCAL_ERR_ADRS	N/A	引起总线错误的本地存储地址。见 error1.tcl。
T1_MODEL	N/A	sysModel() 的期望结果。见 model.tcl。
T1_OFFBOARD_BUS_ERR_MSG	N/A	板外总线错误信息关键字。见 error1.tcl。
T1_OFFBOARD_ERR_ADRS	N/A	引起总线错误的板外地址。见 error1.tcl。
T1_SCSI_BUFSIZE	65536	缓冲器大小（比特）。见 scsi.tcl。
T1_SCSI_ID	N/A	测试用 SCSI 设备的 ID。见 scsi.tcl。
T1_SCSI_LUN	N/A	测试设备的逻辑单元号。见 scsi.tcl。
T1_SCSI_NBBUF	0	SCSI 测试功能的缓冲器序号。见 scsi.tcl。
T1_SCSI_NBLOOP	0	SCSI 测试功能的循环号。见 scsi.tcl。
T1_SEM_ADRS	not used	共享信号量的主管地址。见 busTas.tcl。
{T1, T2}_SER_BAUD	N/A	目标板控制台的默认波特率。见 bspVal。

续表

参 数	默 认	描 述
{T1, T2}_SER_DEVICE	N/A	与目标板控制台相连的系列设备主机名称。见 bspVal。
{T1, T2}_TAS_DELAY	20	测试调整测试中的忙-等待延迟。见 busTas.tcl。
T1_TMO_SLICES	4	时标测试度量时间间隔。见 timestamp.tcl。
T1_TMO_TIMESTAMP	300	时标定时器测试持续时间（秒）。见 timestamp.tcl。
T1_TMO_SCSI	3600	测试 SCSI 的超时（秒）。见 scsi.tcl。
{T1, T2}_TMO_BOOT	60	引导 VxWorks 的超时（秒）。
T2_UDP_TRANS	5000	需发送 UDP 数据包的序号。见 network.tcl。

7.4 运行 BSP VTS

BSP VTS 的每个测试执行一个或更多的 BSP 基本功能。在极大程度上，这些测试的源代码（Tcl/WTX 执行的）被包含在 host/src/test/bsp Val/src/tests 目录下。然而，如果很多的测试共享一个程序，代码就被存储在 host/src/test/bsp Val/src/lib 目录下。这些共享的程序控制 VxWorks 的启动，PASS/FALL 的结果报道等。

例如，BSP VTS 测试的系统时钟和辅助时钟存储在 tests/sysClock.tcl 和 tests/auxClock.tcl 文件下。这两个文件从 lib/clockLib.tcl 中调用程序。除此以外，很多 BSP 测试使用 target/src/test/bsp Val/pkLib.c 中的功能，那是一个在 VxWorks 目标板中（尽管主机依然控制测试的进行）执行 C 函数的库。这些 C 函数作为低水平的支持路线工作。

7.4.1 启动 BSP VTS

BSP VTS 在 Windows 主机或 UNIX 主机上运行。在开始 bash 后，在一个 Windows 主机上运行 BSP VTS，与在 UNIX 主机上运行几乎是一样的。



注意：BSP VTS 需要一条惟一的进入串行设备的途径，它在 bspname.T1 被指定为 T1_SER_DEVICE（或在 bspname.T2 中被指定为 T2_SER_DEVICE）。因此，在能够运行一个 BSP VTS 测试之前必须停止所有的存取指定的串行设备的应用程序或是模块。除此以外，T1_SER_DEVICE(或 T2_SER_DEVICE)连续设备必须是目标控制器连接的设备。

1. Windows 主机

在 Windows 主机上运行 BSP VTS 时步骤为：

(1) 目录改变到：

```
windbase\host\x86-win32\bin
```

windbase 是被 WIND_BASE 环境变量指向的目录。

(2) 编辑 bashrc.bspVal 来配置 Tornado 环境。在文件中具有这些说明。

(3) 调用 bash:

```
bash-rcfile bashrc.bspVal
```

在 bash 以命令 bash\$ 反馈以后, bsp Val 中的类如同在 UNIX 主机上(在下文描述)一样支配。例如:

```
bash$ bspVal tgtsvr1 -b mv147 -s tgtsvr2 -sb mv147 -all
```

2. UNIX 主机

为了开始运行 BSP VTS 测试, 运行主机上的 UNIX 文件 host/hostType/bin/bsp Val。例如:

```
% bspVal tgtsvr1 -b mv147 -s tgtsvr2 -sb mv147 -all
```

主机应该回应:

```

                                BSP VALIDATION TEST
                                -----
Target server                   : tgtSvr1
BSP                             : mv147
Second target server           : tgtSvr2
Second BSP                     : mv147
Log file                       : /folk/lyle/bspValidationLog.671
Model Test                     :
sysModel() return value        : PASS
sysProcNumGet Test             :
sysProcNumGet() return value   : PASS
. . .

```

bspVal 建立一个环境默认值然后调用 bspVal.tcl,Tcl 运行个体测试。

BspVal 只在一批代码中运行。使用命令行选项, 可以指定你想要运行的测试。以上的那个例子使用-all 选项来运行所有的 BSP VTS 的测试。关于 bspVal 的更多信息, 请参考《J.BSP 测试参考条目》。



注意：为了中止测试，进入主机中的 `tty` 中断特征（通常是 `CTRL+C`），这使主机上的测试停止，尽管目标板可继续执行本地任务。在这种情况下，需要重启目标板。

7.4.2 监控一个 BSP 测试

在启动的时候 BSP VTS 打印标准输出的一个头文件。这个头文件陈列了目标服务器的名称，BSP 和日志文件。在每个测试开始的时候，BSP VTS 打印测试名称。一个测试的子测试完成以后，BSP VTS 打印一些 PASS/FAIL 状态信息和子测试的名称。PASS/FAIL 信息被写入日志文件，那是用来捕捉支配输出（如 `ifShow` 或 `smNetShow`）。

正常地，如果一个测试失败了，所有的其他的测试依然会运行。可是，一些错误是致命的。在这种情况下，BSP VTS 中止任务并且陈列一个致命的错误的信息，例如：

```
FATAL ERROR:error in ioctl() command
```

如果你设置环境变量 `TEST_LOG_FILE` 为一个可写下的文件，BSP VTS 搜索这个文件的输出路径。为了在第二终端监控 BSP VTS，运行文件中的 `tail-f`。更多的关于日志文件的信息，请参考 J.BSP 确认测试参考条目中的 `bspVal` 条目。

例子 7-1：测试辅助时钟。

这个例子阐明了 BSP VTS 对测试一个辅助时钟的使用。请注意在这个部分中显示的例子是在 MVME147 BSP 中运行的一个测试。这个例子对于显示怎样与一个单目标的 BSP VTS 工作是很有用的。

(1) 校验目标板能启动 VxWorks。使用 Tornado 发射台，开始一个目标服务器并打开一个 WindSh。通过 WindSh 来确认是否与目标板通信。

(2) 修改 `host/resource/test/bspVal/mv147.T1` 中的配置参数。前缀 `T1_` 的参数必须根据你的环境和 `mv147.T2` 恰当地设置。（`mv147.T2` 中的 `T2_` 参数是为了一个第二目标准备的，辅助时钟测试并不需要它）。为了匹配 `mv147BSP`，设置辅助时钟的速度如下所示：

```
T1_EXTRA_AUXCLK=4321; export T1_EXTRA_AUXCLK
```

(3) 从 `target/src/test/bspVal` 复制 `pkLib.c` 到 VxWorks 目标目录（在这个例子中是 `target/config/mv147`），并使用 VxWorks Makefile 来构造 `pkLib.o`。在测试中 `pkLib.o` 目标文件被下载到目标板中。注意到如果 BSP VTS 脚本没有找到 `bspname` 目录下的 `pkLib.o` 它会自动处理这些步骤。

(4) 核对 `mv147` 目标板连接到主机的合适的串行端口，连接到网络，然后再打开。

(5) 执行主机上的 `bspVal`，在命令行上指定 `auxClock` 测试。测试的运行时间大概是 7 分钟。`% bspVal tgtsvr -b mv147 auxclock`

屏幕上显示：

BSP VALIDATION TEST

```

-----
Target server          : tgtSvr
BSP                    : mv1604
Log file               : /tmp/bapValidationLog.19938
Auxiliary Clock Test   :
auxClk at 4321 hertz (rate = 4327 error = 0%)      : PASS
auxClk at 40 hertz (rate = 40 error = 0%)          : PASS
auxClk at 5000 hertz (rate = 5001 error = 0%)       : PASS
auxClk at 60 hertz (rate = 60 error = 0%)          : PASS
sysAuxClkDisable() disables auxClk                 : PASS
sysAuxClkRateSet() parameter checking              : PASS
sysAuxClkRateGet() return value                    : PASS
Tests RUN 7
Tests PASSED 7
Tests FAILED 0
Tests SKIPPED 0

```

日志文件内容如下:

BSP Validation Test Suite version 2.0

```

-----
BSP Validation Test Started : Wed Aug 27 08:14:25 PDT 1997
Test started : Wed Aug 27 08:14:26 PDT 1997
Auxiliary Clock Test :
optional extra auxClk rate selected is 4321
default auxClk rate is 60
minimum auxClk rate is 40
maximum auxClk rate is 5000
## Starting auxClk test at 4321 hertz rate ##
1 second through auxClk test: ticks1=4510
11 seconds through auxClk test: ticks2=47804
141 seconds through auxClk test: ticks3=610388
corrected value for 120 ticks is 519290
ticks divided by rate = 120
auxClk at 4321 hertz (rate = 4327 error = 0%)      : PASS
## Starting auxClk test at 40 hertz rate ##
1 second through auxClk test: ticks1=120
11 seconds through auxClk test: ticks2=520
141 seconds through auxClk test: ticks3=5722
corrected value for 120 ticks is 4802

```

```
ticks divided by rate = 120
auxClk at 40 hertz (rate = 40 error = 0%)           : PASS
## Starting auxClk test at 5000 hertz rate ##
1 second through auxClk test: ticks1=4995
11 seconds through auxClk test: ticks2=55152
141 seconds through auxClk test: ticks3=705447
corrected value for 120 ticks is 600138
ticks divided by rate = 120
auxClk at 5000 hertz (rate = 5001 error = 0%)       : PASS
## Starting auxClk test at 60 hertz rate ##
1 second through auxClk test: ticks1=150
11 seconds through auxClk test: ticks2=750
141 seconds through auxClk test: ticks3=8554
corrected value for 120 ticks is 7204
ticks divided by rate = 120
auxClk at 60 hertz (rate = 60 error = 0%)           : PASS
sysAuxClkDisable() disables auxClk                  : PASS
sysAuxClkRateSet() parameter checking               : PASS
sysAuxClkRateGet() return value                     : PASS
End of test : Wed Aug 27 08:23:54 PDT 1997
Final Results are :
Tests RUN 7
Tests PASSED 7
Tests FAILED 0
Tests SKIPPED 0
End of BSP Validation Test : Wed Aug 27 08:23:54 PDT 1997
```

(6) 研究这些结果。在上面的输出显示中，BSP 通过了执行的 7 个辅助时钟测试中的子测试。因此，它通过了整个测试。更多的关于确认 BSP VTS 结果的信息，请看附录 K 核对确认。

第 8 章 编写可移植 C 程序

本章介绍如何编写独立于编译器的可移植 C 程序。目的是要无需任何修改，使程序适用于不同的编译器。

针对独立编译器的问题，Wind River 公司已经对它本身 VxWorks 代码进行了分析。这一章就是基于这套分析而展开的。

本章中出现的代码变换不能代替原始主机工具，如 VxSim 或其他由主编译器和交叉编译器完成的汇编工具。

本章也不包括 C++ 的可移植性；GNU 和 Diab C++ 的执行是有根本差别的。关于 C++ 的可移植性将在以后的版本中介绍。

背景知识

随着美国国家标准化组织（ANSI）关于 C 和编译器规范的进一步具体化，每一种编译器需要用不同的方法来实现标准。结果使得很多原码和编码系统并不能真正匹配。为了使编译器能用一种特定的方式产生代码，工程师们需要在编码中插入特殊的非规范性指令。

本章所介绍的知识应归功于 Wind River 公司的努力，现在不需要大的原程序变化就可以支持多编译系统，这样为将来使用新型编译系统排除了不兼容的后顾之忧。

对 Wind River 公司现有的数据库分析，揭示了 3 个可以利用非标准化编译特性来产生运行代码的主要领域：

- 结构封装定义；
- 内联汇编码；
- 数据元素队列。



注意：本章语言仅适用于 GNU 和 Diab 编译器。

本章内容仅限于 GCC 2.96 和 Diab 5.0 以上版本，早期版本彼此之间不完全兼容，与在这部分讨论的规范会有出入。

8.1 可移植 C 代码

8.1.1 数据结构

一些结构定义受制于外部约束。使用结构定义来说明硬件设备寄存器分配是很平常的。

我们也可以经常用结构来定义来自另一系统的数据包内容的结构。然而会出现一些问题，因为根据 ANSI 规定，为了优化数据访问，编译器可以在结构中插入填充元素。这样就可能使结构声明与人们预期的结果和外部世界不再匹配。从结构开始部分到数据开始部分的编译偏移与我们的希望值出现分歧。需要一种既可以识别特殊编码方法产生的特定结构又允许其他结构提高效率的方法。

我们定义未受填充的结构为“包”。每一个主要的编译器都有自己的方法来指定结构包，但是没有一个统一的标准。GNU 和 Diab 在说明书内使用了一种特征声明。微软编译器使用 #pragma 声明。

为了说明独立于编译器的包，我们使用宏来定义结构。宏 `_WRS_PACK_ALIGN(x)` 为一种特征声明。它位于大括号后，实例或初始化软件前。建立名为 `VxWorks.h` 的头文件，一个特定的工具链文件随即产生来定义此宏以供编译工具使用。

例如：

```
struct aPackedStruct {
    UINT32 int1;
    INT16  aShort;
    INT8   aByte;
} _WRS_PACK_ALIGN(1);

struct aPackedStruct {
    UINT32 int1;
    INT16  aShort;
    INT8   aByte;
} _WRS_PACK_ALIGN(1) anInstance = {0x1, 2, 3};

typedef struct {
    UINT8  aByte;
    UINT16 aShort;
    UINT32 aLong;
} _WRS_PACK_ALIGN(1) myPackedStruct;
```

1. 指定字段宽度

在包结构声明中使用特定段宽。基本数据类型 `int`、`long`、`unsigned` 和 `char` 没有规定大小，所以应该尽量避免使用。头文件 `VxWorks.h` 需用特定大小来定义基本数据类型。并在诸如 `INT8`、`INT16`、`INT32`、`UINT8` 等包结构中使用这些数据类型。通常，优先考虑用大于 32bit 正整数构建结构。

2. 应避免字节段

不要在包结构中包含字节域定义。编译器是从最不重要字节或最重要字节开始标记的。

可以使用宏常量在数据段中定义特殊字节模式，从而使上述问题得以解决。

例如：

```
struct aPackedStruct {
    UINT32 int1;
    INT16  aShort;
    INT8   aByte;
} _WRS_PACK_ALIGN(1) anInstance = {0x1, 2, 3};

/* Bits in the aByte field */

#define ABYTE_ERROR    0x01
#define ABYTE_OFLOW    0x02
#define ABYTE_UFLOW    0x04
#define ABYTE_DMA       0x08
#define ABYTE_POLL     0x10
```

8.1.2 内联汇编 (in-line assembly)

内联汇编是最艰巨的任务，因为它同时涉及到编译器与汇编语言。Wind River 公司的编码基于 MIT 汇编语言结构，与很多汇编语言兼容。可移植 C 语言的内联汇编就其本质来讲与体系结构是不相吻合的。因此，这里所要解决的即是内联汇编的兼容问题。

在解决如何使用汇编指令而不会与编译优化性能产生相互影响的问题上，目前不同的编译器差别很大。在没有一个理想的解决办法的情况下，内联汇编只能在不与 C 语言编码相互影响的时候使用。这意味着适合的内联汇编应是与 C 变量或返回变量不产生相互作用的。而那些不能满足此条件的编码必须全部用汇编语言来编写。

头文件 VxWorks.h 定义一个名为 _WRS_ASM 的宏，用来在 C 程序中插入内联汇编。

例如 (PowerPC)：

```
VOID foo (void)
{
    routineA (args);
    _WRS_ASM(" eieio; isync;");
}
```

假设编译器不能自由的与 C 语言环境同时优化或重排内联汇编代码。

8.1.3 静态数据队列

有时需要使用特定边界类型排列一组通用静态数据元素或结构。经常应用于需要高速缓冲存储器边界排列的特定 CPU 数据结构。

为了解决这一问题，引入另一特定工具链宏 `_WRS_DATA_ALIGN_BYTES(bytes)`，使用指定的字节队列排列数据如下：

```
_WRS_DATA_ALIGN_BYTES(16) int myData = 0xFE00235F;
```

这种队列宏只适用于定义过的全局数据。未定义数据在数据段不应出现，与宏无关。未定义数据可以在运行期间使用 `memalign()` 或者其他程序处理。

8.1.4 运行时队列检验

1. 检验数据项队列

在运行期间我们有时需要知道一条特定数据的队列情况。大多数编译器为得到此类信息提供扩展，但没有一定标准。在 Wind River 公司的源码里，宏 `_WRS_ALIGNOF(x)` 用字节单元反馈队列信息。

```
If (WRS_ALIGNOF(itemA) < 4 ==  
(  
    printf ("Error: itemA is not longword aligned");  
})
```

2. 校验指针队列

对于不同的队列需求，指针可以指向不同类型的目标。严格的类型检验在编译阶段是十分有益的，但是也有只能在运行阶段检验的问题。因此，我们定义 `_WRS_ALIGN_CHECK(ptr,type)` 这个宏来评估是 TRUE 还是 FALSE。若 `type` 类型条目的指针 `ptr` 充分排列则返回 TRUE。经常可以通过检查指针的低比特位来测试。

```
void * pVoid;  
if (!WRS_ALIGN_CHECK(pVoid, long))  
(  
    printf ("Error: pVoid is not longword aligned");  
})
```

3. 非队列访问和复制

有时候需要获得在运行期间没有被正确排列的数据。在这种情况下，建议把数据复制到一个结构或其他已经适当排列的地方。这样做可以避免发生非队列访问异常。由此定义宏 `_WRS_UNALIGNED_COPY(pSrc, pDst, size)`。尽管标准 VxWorks 的 `bcopy()` 程序可以用来解决此问题，多数编译器使用小规模复制可以提高效率。使用宏在运行程序时可以得到理想的效果。

在下例中同时使用 `_WRS_ALIGN_CHECK` 和 `_WRS_UNALIGNED_COPY` 检验一个未知指针。若指针被完全排列，则它可以转化为其他形式，得到条目信息。若指针未被排列，则宏将数据复制到一个可用的变量位置。

```
struct structA {
    long item1;
    long item2;
    char item3;
} itemA;

void * pVoid;
long along;

if (WRS_ALIGN_CHECK(pVoid, (struct structA)))
{
    /* Alignment is okay, reference directly */
    aLongItem = ((struct structA *)pVoid) ->item2;
}
else
{
    /*alignment is not okay, use unaligned copy */
    _WRS_UNALIGNED_COPY(pVoid,&along,sizeof(along));
}
```

8.1.5 其他问题

1. 遵守严格的 ANSI 编译规则

编译单元应严格遵守 ANSI 协议要求。这就需要具体的原型声明。对于 GNU 编译系统，编译器标记应包含：

```
-WALL -W -Wmissing-declarations -Wstrict-prototypes -Wmissing-prototypes
```

2. 消除编译警告

可移植编码应尽量避免警告。应用最严格的编码校验来处理所有报警。查错时，编译器能够很好地识别非可移植代码。

3. 避免使用强制类型转换

每个强制类型转换语句意味着隐含的错误。经常会使用强制类型转换语句来处理编译器的报警。然而，我们应该仔细检查每一事例以保证合理地选用强制类型转换语句。通常的报警表明出现了需要被纠正的自变量传输错误。使用强制类型转换语句会破坏编译器在处理重要数据时的纠错能力。

4. 避免内联关键字

这里应避免 C 语言内联关键字，除非 GNU 和 Diab 编译器能用一种相容的方式实现它。根据最新 ANSI 规范 C99，却是需要内联关键字的。然而，两个编译器都不能完全满足此规范。尽管每个编译器都可以使用内联关键字，在实现方法上它们还是有细微但很重要的区别的。当支持内联关键字成熟的时候，本书会有升级版详细介绍。

5. 避免 alloca() 函数

很多编译器把 alloca() 程序作为 C 程序的扩展。和定义变量一样，这需要在堆栈内分配存储空间。由于是堆栈内存储空间，这一区域不需要被释放，因为堆栈会在函数推出后立即重建。

尽管 alloca() 广泛运用于其他操作系统的编程模型，它对 VxWorks 却不十分适用。其他操作系统也许会支持自动堆栈扩展和堆栈校验，而 VxWorks 却不会。在嵌入式设计里，预计耗时和堆栈使用是十分关键的。VxWorks 编码绝对应该避免使用 alloca() 程序。在堆栈中直接分配存储空间，或实在需要时使用 malloc() 和 free() 程序。

6. 小心使用 void 指针算法

对于 void 指针数据类型经常会使用指针算法。因为 void 字段大小是不确定的，而这是 ANSI 标准所不允许的。然而 GNU 编译器却允许这种情况，并规定数据长度与 char 数据类型相同，为一个字节。

例如，下列编码片段是错误的：

```
{
    void * pVoid;

    pVoid += 1;           /* WRONG */
    pVoid++;              /* WRONG */
    pVoid = pVoid + sizeof (char); /* WRONG */
}
```

以上例子的错误是因为 ANSI 指针算法是基于所指目标大小的。当指针指向 void 时，大小未定。对于上述第 1 个错误，有下列惟一更正方法：

```
{
void * pVoid;

pVoid = (char *)pVoid + 1; /* RIGHT */

(char *)pVoid++;          /* WRONG */
(char *)pVoid += 1        /* WRONG */
}
```

上例中的最后两句依然是错误的，因为 ANSI 不允许模型应用于 lval 类型表达。

7. 使用 volatile 和 const 属性

适当使用 volatile 和 const 属性可提高编译器纠错功能。volatile 关键字对于能够被外部服务器中正在运行的线程改变的数据元素是十分重要的。（例如，DMA 设备，另一个共享数据的任务，或者一个需要数据共享的中断程序。）如果用 volatile 标记共享数据失败了，会间断产生难以追踪的系统差错。

VxWorks 5.4 和早期版本经常使用 -fvolatile 编译选项，以使所有指针均指向 volatile 数据字段。将来的可移植编码依赖于此种机制。已知所有数据的正确特性可以使编译器优化更加高效。

const 属性告诉编译器自变量为输入自变量，而且其值在传输过程中不变。这样不仅提高了编译器的检错能力，而且可以产生更优化的编码。

8. 误用 register 特性

register 特性（即 FAST）的误用是十分普遍的。Wind River 公司的代码分析为我们揭示了当输入自变量标记为 FAST 特性的时候，一些子程序可以被编码，本地程序值也是一样。当前编译器可以很好地优化程序而无需在代码段使用 FAST 特性。实际上，过度使用特性反而会妨碍编译器产生有效优化代码。对于现在使用的绝大多数编码，FAST 特性是不必要的。只有当出现很多本地变量，而只引用其中一小部分的时候，才可以适当利用它。

9. 避免使用 vector 为变量名

摩托罗拉公司在 ANSI C 语言规范的扩展中定义 vector 为一关键字。运行代码中不应定义 vector 为变量名。编译器中的这一变化支持 AltiVec 处理器和它所提供的内建 DSP 功能。

10. 陈述标号

ANSI 规范规定每一句语言后面应注明标号。在代码块后做标号而没有任何语言注释的

情况也是经常发生的。在 ANSI 严格检查下这就意味着错误。在下面的例子中 default 标记就没有任何注释。可以去掉标记或写入空白注释来改正这种错误。

```
Switch (xxx)
{
  case X: statement;
  case Y: statement;
  default: /* WRONG - statement here */
}
```

11. 编译宏摘要

对于每个 C 工具链，当编译中出现 VxWorks.h 时，本段中描述的宏被定义。

(1) _WRS_PACK_ALIGN(n)

这个宏用来详细说明结构中出现的捆绑和队列特性。捆绑不仅可以保证没有填充数据，而且保证结构中的最小队列长为 1 字节。使用者可以用自变量 *x* 按字节定义队列整体长度。*x* 值应为 2 的幂 (1, 2, 4, 8, …)。结构长度应为此值的 2 倍。若总结构队列为 1，则编译器认为此结构和任何指向此结构的指针可以存在于任意可能队列。对于不能处理未对准数据传输的体系结构，编译器会产生代码来单独得到各字节，然后以 word 和 longword 单元把这些数据集合起来。

在结构段描述的大括号后和变量定义前使用宏，或者定义 typedef。

最好使用明确宽度定义段，比如 UINT8、UINT16、INT32 等。不要在已封装结构中使用字节段。

(2) _WRS_ASM("X")

这个宏用来在 C 程序声明中插入汇编语言代码。插入的代码不与 C 变量相互作用或改变程序返回值。此代码采用 MIT 汇编语言记忆法和句法。

假设编译器不优化和重排任何特定内联汇编代码。插入一段空内联汇编声明可以阻止编译器用声明后的 C 代码来重排声明前的 C 代码。

(3) _WRS_DATA_ALIGN_BYTES(n)

这个宏用在前置符号中以表明初始 C 数据为一特殊队列。自变量 *n* 为字节队列 1, 2, 4, 8, 16 等。通常只用于通用数据元素。使用这个宏需注意：过度使用可能会有损内存利用。若很多变量需要特殊队列，最好将它们直接在汇编代码中独立的部分中进行声明。由载入程序再把它们连接到一起。

(4) _WRS_GNU_VAR_MACROS

GNU 编译系统有一种分配不同数目自变量给前期处理器宏程序的方法，在 VxWorks 代码里就有一些使用 GNU 定义的句法的例子。而后，ANSI 标准委员会又制定了一套与这不同的标准。目前，GNU2.96 编译器还不能支持 ANSI 标准，而 Diab 编译器却只能支持 ANSI 标准。

使用变量宏的编码应注意既符合 GNU 标准又符合 ANSI (Diab) 标准。我们这里使用一个新的宏特征名, `_WRS_GNU_VAR_MACROS`, 而无须选择一个工具链或编译器名。GNU 工具链对此名进行了定义, 而 Diab 却没有。

如果我们想把代码转化成另一工具链, 必须考虑同时支持 GNU 和 ANSI 句法标准。例如, 下列代码段使用了 `#ifdef` 语句来满足上述要求:

```
#ifdef WRS_GNU_VAR_MACROS /* GNU Syntax */
#define MY_MACRO(x, y, args...) printf (x, y, args)
#else /*ANSI Syntax */
#define MY_MACRO(x, y, ...) printf (x, y, _VA_ARG_)
#endif
```

将来, 当 GNU 工具链满足 ANSI 标准时, 基于宏的 `#ifdefs` 语句可以省略。

(5) `_WRS_ALIGNOF(item)`

此宏以字节形式返回指定项的队列和项目类型。大多数结构的队列值为 4, 通常也就是 longword 队列数据值。更大一些的数据项目或类型返回值有可能为 1, 2, 4, 8, 16 等 2 的幂数。

(6) `_WRS_ALIGN_CHECK(ptr,type)`

此宏返回一布尔数学值 TRUE 或 FALSE。TRUE 意味着分配了一个指向数据项目或类型的有效指针值。可以用检查指针值的低字节信息来确认它是否为给定类型的队列模。

(7) `_WRS_UNALIGNED_COPY(pSrc,pDst,size)`

此宏为标准 Wind River bcopy 操作的编译器优化版本。它可以把数据块从原位置移动到目的位置。在考虑到指针数据类型和块长度的基础上, 此宏允许编译器优化复制操作。经常在执行大面积操作和数据块较小的情况下使用。其他情况下的误用应予以避免。

8.2 执行工具

本节讨论当前的 VxWorks 运行编码是如何利用和维护新的编译工具的。

8.2.1 新工具宏文件

基于工具的宏的定义必须置于编译单元中。可以通过在 `target/h/VxWorks.h` 文件中加入 `toolMacros.h` 文件得以实现。此文件用来定义新宏, 和其他基于工具的选项, 适用于整个编译运行期。所有编译运行文件应包含 `VxWorks.h` 以便使用新工具宏定义。

8.2.2 新工具目录

每一工具在各自的目录下均有一 toolMacros.h 文件。修改 VxWorks.h 可以找出建立在处理器变量 TOOL_FAMILY 上的 toolMacros.h 文件。对于逆序编译, 若 TOOL_FAMILY 无明确定义, 其值根据建立 TOOL 宏的值而产生。

这种新的特定工具链目录可以使所有与工具有关的文件置于与其他工具系统文件分立的常目录下。现在若要增加一个新的工具系统就无须改变任何系统普通文件了。添加新工具目录 target/h/tool/newTool, 然后完成体系构造, 使系统重新分析所有工具链, 重建工具链数据库。

8.2.3 BSP makefile 文件的修改

所有 VxWorks 5.5(Tornado 2.2)BSP 的 makefile 文件需要用基于工具的新路径将旧 make 工具修改到新路径中。由于这一释放, 需要在平台释放文件中做一些修改。

在以下 BSP makefile 文件的各行前插入#。其余含有 include 的各行, 引导所有需要的 makefile 文件。

```
#include $(TGT_DIR)/h/make/make.$(CPU)$(TOOL)
#include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)
#include $(TGT_DIR)/h/make/rules.$(WIND_HOST_TYPE)
```

8.2.4 GNU 和 Diab 的宏定义

当前 GNU 工具链所需宏的执行如下 (以 GCC2.96 为例):

```
#define _WRS_PACK_ALIGN(x)      _attribute__((packed,aligned(x)))
#define _WRS_ASM(x)             _asm_volatile(x)
#define _WRS_DATA_ALIGN_BYTES(x) _attribute__((aligned(x)))
#define _WRS_GNU_VAR_MACROS
#define _WRS_ALIGNOF(x)         _alignof_(x)
#define _WRS_ALIGN_CHECK(ptr,type) \
(((int)(ptr)&(_WRS_ALIGNOF(type) - 1)) == 0 ? TRUE : FALSE)
#define WRS_UNALIGNED_COPY(pSrc,pDst,size)\
(_builtin_memcpy((pDst, (void *) (pSrc), size))
```

当前 Diab 工具链所需宏的执行如下（以 Diab 5.0 为例）：

```
#define _WRS_PACK_ALIGN(x)      _attribute__((packed,aligned(x)))
#define _WRS_ASM(x)             _asm volatile (x)
#define _WRS_DATA_ALIGN_BYTES(x) _attribute__((aligned(x)))
#undef _WRS_GNU_VAR_MACROS
#define _WRS_ALIGNOF(x)         sizeof(x,1)
#define _WRS_ALIGN_CHECK(ptr,type) \
(((int)(ptr) & (WRS_ALIGNOF(type) - 1)) == 0 ? TRUE : FALSE)
#define _WRS_UNALIGNED_COPY(pSrc,pDst,size) \
(memcpy((pDst), (pSrc), size))
```


第9章 文件指导

9.1 概 述

Wind River 板级支持包 (BSP) 的参考文件包括模块 `sysLib.c` 和 `target.nr` 的 UNIX 形式访问入口。HTML 格式的文件也是使用 Wind River 的工具 `refgen` 而产生的。在 BSP 构建中, `make` 运行 `refgen`, 同时把 HTML 的输出存放到 Tornado/VxWorks 的 `docs` 目录下。其产生的参考页面可用 HTML 浏览器在线浏览。

本章介绍 Wind River 关于风格、格式约定和产生 BSP 文件的步骤的一些知识。BSP 开发模板即 VxWorks BSP 开发工具包为我们提供了关于书写风格、文体格式、模块布局及程序命令等方面的例子。这些内容将在以下各节陆续讨论。

你的 BSP 应该是按照 Wind River 规范所编写的, 那么这里所讨论的模块就会与所有 Wind River 文件的宏以及格式化脚本兼容。

9.2 书 写 风 格

本节介绍 Wind River 技术性出版物里面的一些关于书写风格的规范。以下所列各条仅是其书写标准的一部分, 而且是被经常误用的。

关于 BSP 特定规范见 9.4 节。

1. 语句

- 使用简单扼要语句, 以简练直接的方式表达信息。
- 注意语句完整。
- 使用现在时陈述语句。除非需要传达概念, 否则不要使用将来时或过去时。
- 避免缩写——不要省略冠词 `the, a, an`。
- 用 2 个空格分隔语句。

2. 标点

- 在引导例子、展示、名单或列表的习语或语句后使用冒号。
- 在连词 `and, or, nor` 前用逗号来分隔最后几段或 3 个以上的单词或词组。逗号不能省。
例如:

apple, orange, and bananas

- 避免使用引号。如若需要，只能使用直双引号（“ ”）。关于单引号的使用“特殊元素”一节。不要根据传统的 troff 方法随意创造后向双引号和撇号。

3. 语言使用

- 不用大写字母强调而使用斜体字。关于如何改变字体见表 9-3。总之，最好也不要使用斜体字来强调重点，最好的方法为使用模式语句。
- 不要使用 thus 或 therefore 之类的语句。连词 so 是可以使用的。
- 不要使用缩写，如 don't, doesn't, can't 等。
- 不要省略冠词 the, a, an。

4. 拼写

表 9-1 列出了与通常的计算机体系有出入的 Wind River 标准术语。表中还列出了经常拼错的命令、缩写及需要语言环境才可以理解的词句。

表 9-1 拼写规则

正 确	错 误
and so forth, among others	etc.
back end	backend
boot line	bootline, boot-line
boot ROM	bootrom
bps	BPS, baud
caching	cacheing
cacheable	cachable
callback	call-back
cannot	can not
CD-ROM	CDROM, cdrom
coprocessor	co-processor
countdown	count-down
cross-compiler	cross compiler
cross-development	cross development
cross-reference	cross reference
data type	datatype
dialog	dialogue
e-mail	email, E-mail, Email
Ethernet	EtherNet
Excelan	Excellan
fax	FAX
fd	FD

续表

正 确	错 误
filename	file name
for example	e.g.
FTP	ftp
HP-UX	HP/UX, HP/UX
I/O	i/o, IO, io
ID	id
Internet	internet
Intertask	inter-task
inline	in-line
ioctl	IOCTL, IOctl, IOCtl
Iostreams (非黑体字)	IOStreams, iostreams, IoStreams
log in (动词)	login, log-in
login (名词, 形容词)	log in, log-in
lowercase	lower-case
MC680x0	M68000, M68k
MC68020	M68020, 68020
MS-DOS	MSDOS, MS DOS
motherboard	mother-board, mother board
multiprocessor	multi-processor
multitasking	multi-tasking
multi-user	Multiuser
nonvolatile	non-volatile
nonzero	non-zero
on-board	on board, onboard
online	on-line
PAL	Pal
pop-up	Popup
preemptive	pre-emptive
printout	print-out
real-time, Real-time	realtime, Real-Time
reentrant	re-entrant
RSH	Rsh
run-time, Run-time	runtime, Run-Time
SBus	S-Bus, Sbus
SCSI	Scsi, scsi
set up (动词)	Set-up
setup (名词, 形容词)	Set-up

续表

正 确	错 误
shell script	shellscript
single-stepping	single stepping
standalone	stand-alone
start up (动词)	startup, start-up
startup (名词, 形容词)	start-up
stdio	STDIO
sub-class	subclass
subdirectory	sub-directory
SunOS	SUN OS
super-class	superclass
task ID	task id
Tcl	TCL, tcl
TFTP	Tftp
that is	i.e.
timeout	time-out
timestamp	time stamp, time-stamp
tty	TTY
UNIX	Unix
uppercase	upper-case, upper case
User's Group	User's Group, Users Group
VxWorks	VxWORKS, VXWORKS
workaround (名词)	work-around

5. 缩写词

除去可识别缩写词, 将常用缩写词列于表 9-2。先给出全称, 然后在括号内标注缩写, 例如:

Internet Control Message Protocol (ICMP)

不要使用撇号 (') 表达缩写的复数形式。CPU 的复数可表示为 CPUs。

表 9-2 常见缩写

缩 写	全 称
ASCII	American Standard Code for Information Interchange
ANSI	American National Standards Institute
CPU	Central Processing Unit

续表

缩 写	全 称
EOF	End-Of-File
fd	file descriptor
FTP	File Transfer Protocol
IP	Internet Protocol
NFS	Network File System
PPP	Point-to-Point Protocol
pty	pseudo terminal device
RAM	Random Access Memory
ROM	Read-Only Memory
RSH	Remote Shell
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
tty	terminal device

6. 目标板名称

目标板 VME 目标板的名称应与其提供者相一致：如 MV135 在 MVME135 中不合法。

当一个板级支持包包含多个目标板模型而这些模型的部分名字用“/”或“-”来分隔时，这些部分可用逗号和空格来分隔。见下例：

Force SYS68K/CPU/-21, -29, -32

Heurikon HK68/V2F, V20, V2FA

特例：

Motorola MVME147, MVME147S-1

9.3 格 式

本节讨论 Wind River 参考文件的一般格式要求。关于 BSP 的相关规范详见 9.4 节。

1. 布局

- 模块布局。在调用 `refgen` 时，源模块中的文件须遵守一些简单原则。这些布局规范见 Tornado 使用指导：编码规则。（Tornado User's Guide: Coding Conventions）
- 常规。正文需写满（设为 75 字）。不要写一句换一行。

2. 特殊元素

- 子程序。要在所有子程序名后加小括号，即使对那些通常被按照 shell 命令的方式直译执行的程序。不要在括号中或程序名字后加空格（这一点不同于 Wind River 的编码规则）：

正确：

```
taskSpawn()
```

错误：

```
taskSpawn( ), taskSpawn (), taskSpawn
```

注意此处有一特例，在定义子程序时，名字无需加括号：

正确：

```

/*****
 *
 * xxxFunc ~ do such and such

```

错误：

```

/*****
 *
 * xxxFunc() _ do such and such

```

避免句子开头使用程序名或模块名，如果必须这样，一定不要大写开头。

- 参量。当需要引入参量时，用尖括号<>把自变量扩起来。比如，定义子程序 getName()如下：

```

VOID getName
(
    int      tid,      /* task ID */
    char *   pTname /* task name */
)

```

也许你会写以下一句：

```
This routine gets the name associated with the specified task and copies
it to <pTname>.
```

- 用单引号标注命令、文件、任务、通用变量、操作、C 关键字、网络界面的名字。例如：

```
This routine maps an error number in 'errno' to an error message.
```


- 终结符。终结符须大写；例如：RETURN, ESC。在控制语句前加 CTRL+；例如，CTRL+C。
- 有关参考书目。有关章节的参考书目应采用以下形式。出版物标题：章节名称。例如：

For more information, see the
.I "VxWorks Programmer's Guide: I/O System."

不要写章节数。文件卷册参考规定用斜体字写。通常情况下，使用带.I 的宏。然而在 SEE ALSO 段中，VxWorks 编程指导和 Tornado 使用指导中分别使用带.pG 和.tG 的宏。详情请见 Markup 命令。

- 交错段落号。不要在库或子程序中使用 UNIX 的括号加数字的标记方法。正确：

sysLib, vxTas()

错误：

sysLib(1), vxTas(2)

特殊元素格式见表 9-3。

表 9-3 特殊元素格式

组 成	输 入	输出 (mangen + troff)
库标题	sysLib.c	sysLib
库原文	sysLib	sysLib
子程序标题	sysMemTop	sysMemTop()
子程序原文	sysMemTop()	sysMemTop()
子程序参数	<ptid>	ptid
参考文献 (非 SEE ALSO 的)	.I "Tornado User's Guide"	Tornado User's Guide
VxWorks 编程指导 (SEE ALSO 的)	.pG "Configuration"	VxWorks Programmer's Guide: Configuration
Tornado 用户指导 (SEE ALSO 的)	.tG "Cross-development"	Tornado User's Guide: Cross-Development
省略	.I must	must

3. 显示

关于下列宏及命令的显示详见 9.5.3 部分内容。

- 编码。使用带.CS 和.CE 的宏程序显示编码及输入/输出结果。左起空 4 格显示如下：

```
* . CS
*     struct stat statStruct;
*     fd = open ("file", READ);
```

```
* status = ioctl (fd, FIOFSTATGET, &statStruct);
* .CE
*
```

对于库路径，这里指从第 5 栏开始显示；对于子程序，是从第 7 栏显示（因为子程序中的有效行从第 3 栏开始，由于 `refgen` 去掉了初始星号和空格）。

- 板图。在 `target.nr` 模块的 **BOARD LAYOUT** 标题下使用带 `.bS` 和 `.bE` 的宏显示板图。
- 列表。用 `tbl` 标记工具创建表格是很容易的，只要坚持用 `basics` 就可以在几乎所有情况下满足要求。关于创建表格的命令请见“表格”部分。在格式上通常需要考虑以下 2 点：
 - 栏标题头用黑体。
 - 用一波折号分隔标题头与表体。
- 分项。用标准控制宏 `.IP` 来分项。不要使用 `.CS/.CE`。

9.4 子 段

本章按部分讨论了 BSP 文件的格式规则。想了解详细信息和具体例子请参见 BSP 模板。

在下面的例子中，`mfr&board` 表示厂家名称和目标板模型全称，见“目标板命名”部分内容。

对于目标信息参考页(`target.nr`)有特定要求，因此独立于标准数据库和子程序参考页而单独讨论。

1. 库与子程序参考页

在 `target/config/templateCPU/sysLib.c` 中有一个展示 `sysLib.c` 文件的模板。库和子程序的参考页面通常顺序包含以下分段：其他分段如若需要也可包含：

- **NAME**。库或子程序名。
- **SYNOPSIS**。在库中为子程序摘要；在子程序中，为声明（由 `refgen` 自动生成）。
- **DESCRIPTION**。库或子程序总介。
- **INCLUDE FILES**。内含 `.h` 文件（只用于库）。
- **RETURNS**。返回值（只用于子程序）。
- **SEE ALSO**。关于其他库，文件或用户手册的交叉参考文件。

关于这些分段的注意事项将在下面的内容进行讨论。

(1) NAME 段

- 库。在程序中简要描述。连字号必须用在两个空格间，不使用反斜线符号或双引号。常用格式为：

```
nameLib.c - such-and-such library
```


例如:

```
sysALib.s - mfr&board system-dependent assembly routines
sysLib.c - mfr&board system-dependent library
```

一定要包含文件扩展名 (.c, .s); 注意到 `refgen` 过程会把它们去掉而不会在最终参考页面中显示。

- 子程序。对于一行的标题/定义这种结构, 使用下列祈使句和动作。通常格式为:
名称 - 动作

例如:

```
sysMemTop - get the address of the top of memory
```

不要在标题中插入子程序小括号; `refgen` 过程会把它们加入而在最终参考页面路径中显示。

(2) DESCRIPTION 段

用 "This library..." 或 "This routine..." 开始描述段——在这里不要重复模块或子程序名称; 因为它已经被描述得很清楚了。接下来的语句应该是一个关于库和程序提供服务的总结 (比 NAME 行做更深入描述)。

(3) INCLUDE FILES 段

每一个库和驱动程序入口需要一个副标题 INCLUDE FILES, 列出任何有关标题名称, 例如:

```
INCLUDE FILES: sysLib.h
```

(4) RETURNS 段

RETURNS 段应是 SEE ALSO 前的最后一段。

- 所有子程序都包含 RETURNS 段。若无返回值 (在 void 情况下), 只需键入:

```
RETURNS: N/A
```

- 在 RETURNS 段中只使用真返回——不是假设作为自变量复制到缓冲器里的值。
- 用大写字母开始返回值描述, 用逗号结束; 不要使用缩写。如:

```
RETURNS: The address of the top of memory.
```

不论格式的返回规则、返回值为不完全语句; 包括主语和谓语。

- 用现在时书写返回语句 (当返回 ERROR 时, 可认为是过去时)。
- 在返回 STATUS 时, ERROR 后紧跟一限定语句。

在 OK 后跟逗号, 因为限定句属于 ERROR 而不是 OK。比如:

```
RETURNS: OK, or ERROR if memory is insufficient.
```

不要用多余的空格开头——首字符为空格的输入行会导致溢出暂停。以前, 很多作者在 RETURNS 段中使用这种技巧为返回的独立元素产生段间隔——这里不使用这种技巧。

例如:

正确:

- * RETURNS: OK, or ERROR if the tick rate is invalid or the timer
- * cannot be set.

错误:

- * RETURNS: OK, or ERROR
- * if the tick rate is invalid or
- * the timer cannot be set.

(5) SEE ALSO 段

SEE ALSO 段为参考页面的最后一段。

- 若为子程序, `refgen` 在 SEE ALSO 段内自动包含根数据库。
- 不使用 UNIX 的括号加数字模式来书写交叉参考手册段数码。

正确:

SEE ALSO: `sysLib`, `vxTas()`

错误:

SEE ALSO: `sysLib(1)`, `vxTas(2)`

- 在 SEE ALSO 段中, 关于 VxWorks 编程指导和 Tornado 用户向导的交叉参考章节分别用带 `.pG` 的宏和带 `.tG` 的宏。点 “.” 必须为有效行的第一个字符 (对于一个子程序段, `refgen` 可以去掉星号和空格)。例如:

```
* SEE ALSO: someLib, anotherLib,
* .pG "Basic OS, Cross-Development,"
* .tG "Setup and Startup"
* /
```

上例表明, 关于另一栏的交叉参考应该置于关于其他库与子程序的交叉参考之后。 `.pG` 和 `.tG` 宏只用于 SEE ALSO 段后。

2. 目标信息参考页面: `target.nr`

有关目标信息参考页面的模板由 `target/config/templateCPU/target.nr` 提供。通常包含以下分段, 如有需要也可包含其他分段。摘要如下:

- NAME. 目标板名称。
- INTRODUCTION. 范围和前提摘要。
- FEATURES. 目标板支持和不支持特征。
- HARDWARE DETAILS. 目标板驱动器及硬件信息。
- SPECIAL CONSIDERATIONS. 特征及特殊限制。

- **BOARD LAYOUT.** ASCII 格式目标板布局。
- **SEE ALSO.** Wind River 的文件参考书目。
- **BIBLIOGRAPHY.** 附加文件参考。

(1) NAME 段

NAME 段的信息只用一单行表示，以单一自变量形式写入宏 `aX`，格式如下：

```
.aX "mfr&board"
```

同前，`mfr&board` 为厂家名称和目标板模型生产商名。例如：

```
.aX "Motorola MVME147, MVME147S-1"
```

(2) INTRODUCTION 段

此段包括初始信息，包括 VxWorks 操作中 ROM 安装及跳转设置的细节问题。

(3) FEATURES 段

此段描述所有目标板支持或不支持的特征。目标板的任何特征总属于下列子段之一，Supported Features 或 Unsupported Features。每一目标板配置选项均为一特征。而第 3 个子段，Feature Interactions，描述一个特征或目标板配置如何影响其他。

(4) HARDWARE DETAILS 段

此段讨论硬件元素和驱动设备，比如 serial, Ethernet 和 SCSI。它还包含每个总线的存储器分配情况，中断级别清单和/或每一中断源的向量序号。

(5) SPECIAL CONSIDERATIONS 段

此段描述目标板特性。包含不适用于其他段的所有用户所需信息。

对于拥有 BSP 确认测试组件的用户，此段必须含有 VTS 测试的已知错误。通常目标板没有一定特征或者没有特定方式展示特征。BSP 编写者对所有在 VTS 测试阶段出现的异常负责。

(6) BOARD LAYOUT 段

用 `bS/bE` 宏程序展示目标板图表。参看 BSP 模块图表转移位置指导，或见 `target.nr`。

(7) SEE ALSO 段

此段经常涉及《Tornado 用户指导》的“设置和启动”章节。其他 Wind River 手册也可根据需要列出。

在 Tornado 用户指导参考中使用带 `tG` 的宏。在 VxWorks 编程指导参考中使用带 `pG` 的宏。对于其他手册使用带 `iB` 的宏。例如：

```
.SH "SEE ALSO"
.tG "Setup and Startup,"
.pG "Configuration, Motorola MC680x0"
```

(8) BIBLIOGRAPHY 段

此段描述任何附加技术手册，数据清单，或者用户需要的补充信息。对于这些参考文

件使用带.iB 的宏。例如：

```
.SH "BIBLIOGRAPHY"
.iB "Motorola MC68620 User's Manual"
.iB "Motorola MC68030 User's Manual"
```

9.5 创建参考页

本节讨论创建 BSP 文件的方法：使用的文件和工具，文本格式命令，以及把文件从源码转换为可打印参考页面的 makefile 系统。

9.5.1 文件

有以下几种文件扩展名：

- .s 汇编语言源文件。
- .c C 语言源文件。
- .nr nroff/troff 源文件。
- .html HTML 格式产生文件。

1. 原地址目录 (target/config/bspname)

此目录包含特定 BSP 的 C 和汇编源代码，以及源文件；bspname 反映生产商和目标板型号的目录名；比如，mv147 为 Motorola MNME147。和文件有关的文件为：

- Makefile。创建 BSP 和 VxWorks 模型的主要 makefile 文件。需定义 3 个常量：TARGET_DIR, VENDOR 和 BOARD。
- depend.bspname。Makefile 产生的独立规则。
- sysLib.c。独立于主机的 C 程序库。
- target.nr。目标板信息参考页的资源程序，包含有关 VxWorks 目标板安装要求和能力的一般信息。此文件“手工”产生的 nroff/troff 源文件。

2. 文件目录 (docs/vxworks/bsp/bspname)

此目录包括 BSP 的 HTML 参考页面文件。所有文件通过 make 运行 refgen 而在 target/config/bspname 内由源文件产生。这些文件有：

- bspname.html 由 target.nr 产生的目标信息参考页面。
- sysLib.html sysLib.c 的参考文件。
- libIndex.html BSP 库级参考页面索引。
- rtnIndex.html BSP 子程序参考页面索引。

9.5.2 工具 (host/host/bin/refgen)

此工具是 Tcl 脚本，它通过特殊格式源代码产生 HTML 文件，这些代码可以为 C 语言模型、汇编语言模型或 target.nr 文件。关于 refgen 的命令行句法总结于 L.Refgen 的参考页中。

9.5.3 文本格式

1. markup 命令

这段讲述使用 UNIX nroff/troff man 宏在源文件中创建 markup 格式的基础。

对 mark-up 命令的理解有助于构建 target.nr 文件。然而，在.c 和.s 文件中，mark-up 应该用.CS/.CE 来显示例子，.IP 来创建目录——在这些模型中应该适当地使用 markup，因为大多数情况下它们是自动生成的。

下面的列表显示了在 Wind River 参考文献中的 mark-up。从 UNIX man 宏发展而来的 mark-up 命令由 2 个大写字母组成。由 Wind River 添加的宏则由大写字母加小写字母组成。

每一个命令自变量须用双引号括起来；比如：

.SH "BOARD LAYOUT"

.TH 12345

标题名。此宏添加到标题头和页脚处。经常是调用手册入口的第一个宏。它的自变量如下：

- 1——模块名（对于 target.nr 为目标目录名）。
- 2——字母段号（“T”为 target.nr 文件）。
- 3——目标板生产者和模型（位于运行页脚中部）。
- 4——当前数据，形式“Rev:01 Jan 97”。
- 5——“VXWORKS 参考手册”（位于运行标题中部）。

.SH heading

子标题（名字，描述等）。

.CS

.CE 代码开始/代码结束。这一对宏用来显示代码——显示结果不做任何修改，字体长度固定。

.bS

.bE 目标板开始/目标板结束。这对宏用来在 target.nr 中显示目标板图表。它们与.CS/.CE 相似，不同的是显示缩小字体。关于用 ASCII 码显示表格转移位置的指导见 target.nr 部分。

.IP x Indented Paragraph。此宏用来产生详细项目。自变量 x 为项目标记。遇到.LP 缩排结束。

.IP "FIODISKFORMAT"

通过适当的硬件路径和段落标记格式化整个磁盘。通过执行此命令不初始化文件系统。

.IP "FIODISKINIT"

在磁盘空间内初始化 DOS 文件。

.LP

需要处理的其他 `ioctl()` 程序码被转移到块设备驱动器。

输出举例：

■ FIODISKFORMAT

通过适当的硬件路径和段落标记格式化整个磁盘。通过执行此命令不初始化文件系统。

■ FIODISKINIT

在磁盘空间内初始化 DOS 文件系统。

需要处理的其他 `ioctl()` 程序码被转移到块设备驱动器。

.LP left paragraph。此宏存储 IP 定义以左，见上例。

.pG chapter

编程指导章节。此宏用在 SEE ALSO 段中特指 VxWorks 编程指导交叉参考章节。见后面的 SEE ALSO 段。

.tG chapter

Tornado 用户手册章节。此宏用在 SEE ALSO 段中特指 Tornado 拥护手册交叉参考章节，见 SEE ALSO 段。

.iB text

参考书目。此宏用于 SEE ALSO 和 BIBLIOGRAPHY 段中补充说明 VxWorks 编程指导和 Tornado 用户手册没有涉及的参考文献。

.I text

斜体字。此宏用来设定文本为斜体字。

.aX text

此宏不产生任何特殊格式；`wrs.an` 逐条输出其自变量。然而，在 Wind River 宏程序包和文本中它用来为内容和索引表格收集数据。只需调用 `target.nr`，见“NAME 段”。

2. 表格

只要遵循一定规则，创建普遍适用的表格是非常容易的。



注意：不要使用 `.CS/.CE` 宏来创建表格。在代码举例中这些宏被限用。

下面一例采用一般格式：

.TS


```

expand;
1 1 1 .
第 1 栏头      第 2 栏头      第 3 栏头
第 1 栏第 1 行 第 2 栏第 1 行 第 3 栏第 1 行
第 1 栏第 2 行 第 2 栏第 2 行 第 3 栏第 2 行
.TE

```

作为选择，在 `target.nr` 文件中，可以用 `tab` 键代替“ ”间隔分栏。

9.5.4 处理

以下各步展示了使用基于 BSP 开发工具产生的 `makefile` 文件创建 BSP 文件的过程。

(1) 在 `target/config/bspname` 路径中，在 `Makefile` 中查询以下常量：

- `TARGET DIR` 目标目录名 (`bspname`)：例如，“`mv147`”。
- `VENDOR` 生产家名：如“`Motorola`”。
- `BOARD`。板型名称：如“`MVME147,MVME147S-1`”。

(2) 运行 `make man` 命令产生参考页面。

这会实现以下各项：

- 创建一适当文件 `depend-bspname`。
- 通过 C 预处理程序运行 `sysLib.c` 来采集所有包含在 `#include` 指示下的驱动程序。
- 运行 `refgen-mg`。
- 分配 HTML 参考页到相应的 `WIND_BASE/docs` 目录下。

图 9-1 说明处理过程是如何把 BSP 参考页分配到 `docs/vxworks/bsp` 目录下的。

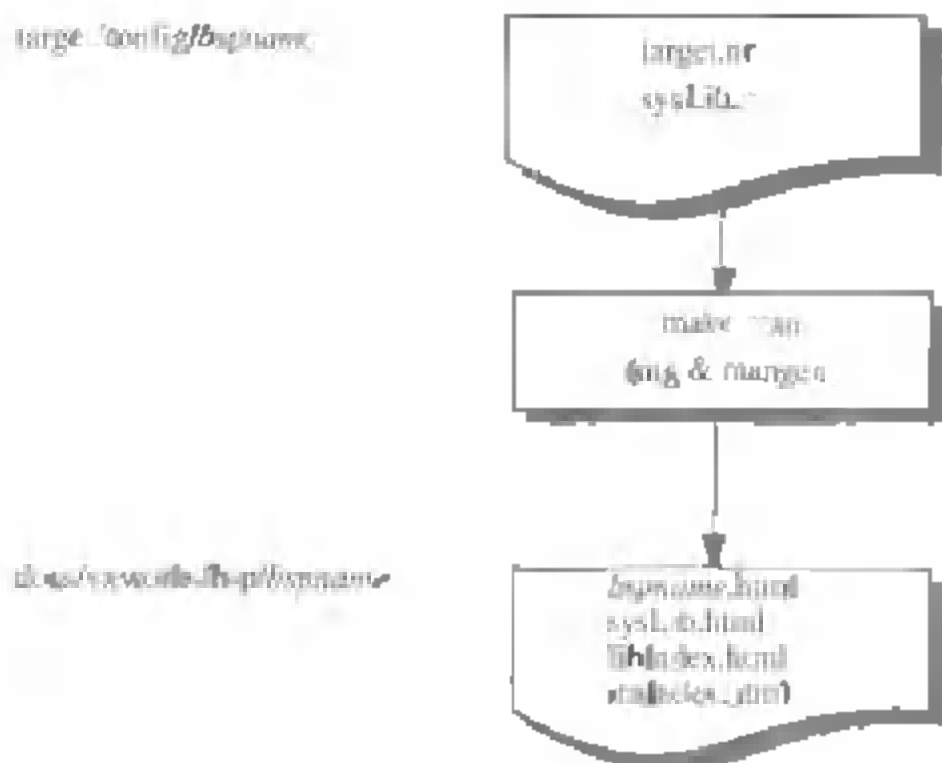


图 9-1 BSP 文件的产品流程

第 10 章 产 品 封 装

10.1 安装开发工具包

目前 Wind River 公司产品的发行使用安装功能来制作只读存储器光盘。它们经过加密，只有通过安装密钥才能够破解。安装功能在所有主机支持平台上均可使用。

安装盘不仅包含适用于所有主机平台的解包程序，而且还有一些诸如 README.TXT, FIXED.TXT 和 PROBLEM.TXT 等文本文件。

详情请见 Tornado 安装软件开发工具包编程指导（WindSurf 出版）。

10.2 BSP 封装

封装包括所发行产品的内容和形式。Wind River 公司所出品的所有 BSP 软件都集成在光盘中，适宜在任何主机上安装。过去，BSP 是在 UNIX tar 磁带中或包含 UNIX 安装文件和 Windows 自摘录功能的光盘中发行的。

所有 VxWorks 的 BSP 均含有类似的目录结构和以下所描述的几个常用文件。遵循以下各小节所介绍的标准，将有助于确保 BSP 在 VxWorks 开发系统中的顺利运行。

1. BSP 所包含的内容

明确文件和目录作为发行 BSP 产品的一部分是制作 BSP 软件最终和最重要的步骤之一。BSP 必须包含所有重新汇编 BSP 所需的文件，而没有使另一 BSP 与用户系统不兼容的文件。

2. 默认配置

在 BSP 的封装阶段以前，必须建立和设置它的默认配置。这包含在 config.h 中设置配置选项以及在 makefile 文件中利用 RELEASE 变量选择默认值。

在一些情况下，BSP 的目标板所使用的配置结构有所不同。例如，它们也许会包含不同的存储器模型，外围设备控制器，处理器种类或不同等级修订本。可适用于所有可能配置的 BSP 才是我们的最佳选择。

为此，使用的可能配置应尽量最小化。这可以使目标板机的所有变量运行所装载的只

读存储器设置和 VxWorks 映像。因此，默认存储器容量越小越好，而且所有可选择的外围设备控制器应配置在 BSP 之外。若终端用户拥有大存储器或一个可选模型，就可以在需要的情况下制定所支持的配置以及重建 VxWorks 映像和引导只读存储器。

任何调整目标板支持的 BSP 配置机制均需通过 config.h 里的参量来实现。这些参量一般为宏或条件编辑指示。任何非常规或复杂配置机制也应该在 config.h 和 target.nr 中的特殊考虑一段中予以解释。

Wind River 公司的 BSP 标准发行批处理目标包括 bootrom, VxWorks 和 vxWorks.st。标准 make 变量 RELEASE 定义为这些目标。若定义一套不同目标（例：bootrom_uncmp.hex 代替 bootrom.hex），RELEASE 应在 makefile 文件中重新定义。比如：

```
--
include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)
--
RELEASE +=bootrom_uncmp.hex
--
include $(TGT_DIR)/h/make/rules.bsp
--
```

为用户提供可执行文件的目的只为了实证和确认。假设所有用户在家都会重新配置和改建可执行参量。其中包括引导只读存储器映像。

同时还应包括获得支持的工具链的工程。这些工程的默认名为 bspName_gnu 或 bspName_diab。黑体字部分可使用其他起名方法。这些工程是用 RELEASE 宏设定默认值的。它们可由 make 目标 prj_gnu 或 prj_diab 单独建立。

3. 包含文件

关于 BSP 产品内嵌文件的详细内容请见附录 M。BSP 文件分为以下几类：

- 特定目标板文件：所有在 target/config/bspname 和 docs/vxworks/bsp/bspName 目录下的目标参考页面。
- 与 BSP 默认工程同时产生的文件。工程可在工程设备中产生或使用 make release 使用传统命令产生。
- HTML 格式文件。见 htmlBook 命令访问入口（《Tornado 用户手册》：设备参考）。

4. 外部文件

BSP 产品应包含下列文件：

- VxWorks 文件。

产品不包含向用户提供的其他 Wind River 产品所需的文件。

```
target/config/all/...
```

```
target/config/compa/...  
target/src/...  
target/h/...
```

这些目录下的文件是 VxWorks 体系/工具链产品的一部分,而且不应被任何第 3 类产品所改变。

BSP 从属文件(target/config/bspname/depend.bspname)不应与其他 BSP 文件同时发行。这样可以在用户第 1 次使用 make 命令汇编时强迫系统重新产生它。

■ 驱动程序目录。

产品不应在以下列出的目录中包含第三方文件。这些目录只为 Wind River 原始文件所保留。第三方不可以添加或修改这些目录中的任何文件。若 BSP 包括 Wind River 驱动程序的特殊 BSP 版本,驱动程序应分配到 BSP 目录下而非 target/src/drv 目录下。

```
target/h/drv/...  
target/src/drv/...
```

■ 修改的 Wind River 目标文件。

BSP 编写者不要制作任何覆盖或修改 Wind River 公司所提供的目标文件。惟一和独特的目标文件会在 BSP 目录下自行产生。

■ 版权保护文件。

Wind River 公司拥有其网络和 SCSI 驱动程序的所有版权。若无 Wind River 系统的明确书面认可,任何第三方不准在这些驱动程序或派生驱动程序中分配源程序代码。Wind River 组成版本的源程序代码是受保护的。

```
target/src/drv/netif/any.c  
target/src/drv/scsi/any.c  
target/src/drv/end/any.c
```

5. 文件源

在 BSP 工具包产品中的 BSP 模型不产生通用驱动程序或特定机构驱动文件。BSP 样本起初是为帮助开发者创建目标明确文件和新驱动程序而用的。

开发者常会从 BSP 参考中复制通用文件(每个 BSP 与一个 VxWorks 目标许可相捆绑)。

6. 卖方提供的共享文件

有些文件看上去是目标明确的,然而实际上是为许多 BSP 软件所共享的。这些共享文件包括上述“通用”和“特定机构”,以及包含驱动目标工具的文件。

以下各目录为分配在 Tornado 机制产品和 Tornado 驱动成分产品中的目录:

```
target/config/all
target/src/config
target/src/drv/anything
target/h/drv/anything
target/h/make
target/h/tool/anything
```

任何情况下都不允许第三方开发者改变和装载这些目录和文件。尽管 BSP 开发者是可以创建通用安装设备驱动程序的，但是他们不能在 `target/src/drv` 和 `target/h/drv` 目录中设置源码；Wind River 所有现有驱动程序均不可被修改。若一终端用户准备改变共享文件，在原有第三方 BSP 系统中安装 Wind River BSP 或新的 VxWorks 版本，第三方文件的改变将会被 Wind River 文件覆盖。

通常情况下，BSP 是可被编写的，以便所有特定目标板文件存放到 `target/config/bspname` 和 `docs/vxworks/bsp/bspName` 目标参考页面目录下。少数情况下，这种方法不可取，BSP 必须包含共享文件的修正版本。然而，这些共享文件本身是不能被改变的；而改变这些所需文件的复本并将它们存放在合适的位置（如：`target/config/bspname`）。然后 BSP makefile 文件就可以升级到开发版本；见《Tornado 用户手册》2.2 节所介绍的 makefile 文件机制。

BSP 开发者可使用 `CONFIG_ALL` makefile 文件变量引导 BSP makefile 文件，寻找 `target/config/all` 文件中的修正版本。首先，开发者应把这些文件的 Wind River 版本复制下来，存储在 BSP 的 `bspname` 目录下，以便需要时修改它们（注意此操作只适用于传统的命令行系统，对工程配置或构建不造成任何影响）。

通过使 VxWorks 体系用户化，BSP 开发者可以创建共享文件的特定目标板版本而不会与其他 BSP 软件发生冲突。

自 Tornado 2.0 以来，每个体系结构的只读存储光盘均包含一个“驱动程序目标/标头”产品。此产品包含所有已编译的此体系机构所需的以太网和 SCSI 驱动程序，以及它们的标题头。还包括以源代码形式出现的所有未受保护的 Wind River 驱动程序。用户可以从网上下载升级驱动产品。

10.2.1 BSP 媒体

自 Tornado 1.0.1，所有 Wind River 产品均使用集成 UNIX/Windows 安装方法封装在只读存储光盘中。UNIX 和 Windows 用户都可通过 `SETUP` 命令开始安装。所有用户将在窗口界面的指导下完成产品和许可管理器安装过程。

详细内容请见《Tornado SETUP SDK 开发指导》（WindSurf 出版）。

10.3 组件封装

组件封装也需解决很多问题。Wind River 公司将部分组件安装在其传统位置上。

- 源码模块可在 `target/src` 或 `target/config` 目录下找到。
- 标题头置于 `target/h` 中；目标模块在 `target/lib/objARCH` 中。
- 组件描述文件在 `target/config/comps/vxWorks` 中。
- 组件配置文件（源码片段）位于 `target/config/comps/src`。

第三方受限于这些规定，组件元素位置在组件描述文件中完全给出。建议第三方把组件源和目标元素置于一个特定目录下，如 `target/config/vendor_name`。组件描述文件（CDF）的位置取决于组件集成在系统中的位置。

若要在一个完整系统中集成新的组件，CDF 应该置于 `target/config/comps/vxWorks` 目录下。若它是一个 BSP 特定组件，则它应该放于 BSP 目录下。若其特别服务于一个单一工程，则它应放于工程目录（`target/proj/project_name`）下。注意遵守合适的命名规则。任何第三方 CDF 名字均须以 50 到 99 之间的数字开头（见 6.3.2 CDF 的优先及路径）。

第三方组件不允许覆盖 Wind River 支持的文件。但可在其自身组件描述文件中忽略 Wind River 的组件描述。考虑到文件优先级，没有必要为了改变组件行为而覆盖一个文件。

10.4 工程封装

工程通常在一单一目录下作为一组文件而被封装起来。可以使用创建向导来产生工程，或用 `make release` 书写命令行来实现。

以下为标准 Wind RiverBSP 工程的常见文件：

```
bsp_gnu
bsp_gnu/bsp_gnu.wpj
bsp_gnu/usrAppInit.c
bsp_gnu/prjParams.h
bsp_gnu/linkSyms.c
bsp_gnu/prjConfig.c
bsp_gnu/Makefile
bsp_gnu/default
bsp_gnu/default/vxWorks
bsp_gnu/default_rom/vxWorks_rom
```

```
bsp_gnu/default_romCompressed/vxWorks_romCompressed  
bsp_gnu/default_romResident/vxWorks_romResident  
(省略了带.o的中间文件)
```

任何特殊组件和工程需要的 CDF 都需在工程中建立起来。若一个工程需要从一个地方安装到另一个地方，那么工程重建后应在新安装位置上使用组件描述文件。

第三方也应该用此种方式封装其向用户提供的工程。可以在 `target/proj` 目录下书写工程目录名，这样就不会与任何 Wind River 提供的工程发生冲突了。

附录 A 升级 BSP 以兼容 Tornado 2.0

这个附录描述了怎样把 BSP 从 Tornado 1.0.1/VxWorks 5.3.1 过渡到 Tornado 2.0/VxWorks 5.4。

为 VxWorks 5.2 或更早版本写入的 BSPs 被指定为版本 1.0BSPs。为 Tornado 1.0.1 或更早版本写入的 BSPs 被指定为版本 1.1BSPs。在这个文件中概要的提及了写入标准 BSPs，特别是为了 Tornado 2.0 准备的 BSPs 被指定为版本 1.2BSPs。

所有版本的 1.1BSPs 已经被修改了，它们使用 END 驱动程序和 SENS 网络堆栈，它们与 Tornado 2.0 兼容。使用指定顺序和修改了的内核文件版本的第三部分的 BSPs 能否转化为工程。如果传统的命令行被用来建立工程的话它们会正确的工作。

A.1 过渡问题

把一个已存在的 Tornado 1.0.1 BSP 过渡到 Tornado 2.0 需要以下几个步骤：

- 升级 SENS 网络堆栈的 BSP。如果一个 END 类型的驱动程序可以被 BSP 所使用，那么它应该成为默认的网络驱动程序。任何不与 END 或 BSD4.4 兼容的传统网络驱动都必须为 SENS 升级。参看《VxWorks 网络堆栈》。
- 创建一个 BSP 的默认工程和 HTML 文件。这是通过使用命令 `make release man` 来实现的。
- 创建一个 BSP 产品。这仅仅适用于那些需要把自己的 BSP 分发给其他 Wind River 用户的人。参看“A.4 产品内容”，其中有关于为了传递给用户怎样包装一个 BSP 的信息。

A.2 代码接口的改变

在 Tornado 1.0.1 之前没有任何一个在 BSP 中使用的标准 API 已经被改变了。《Tornado 发行手册》2.0，提到了对 `pciConfigLib` 和访问机制 0 的 API 修改；尽管如此，这些修改仅仅会影响到很少的用户。

产品包装的改变使得 Wind River BSPs 的版本被升级到版本 1.2。Wind River 使用一个 X.X/Y 形式的 BSP 命名计划，X.X 是 BSP 的版本，Y 是修正 ID。每次新发布的或升级的

BSP 增加修正 ID。任何新的 BSP 的第一个修正号都是 0,而不是 1。Config.h 文件包含宏 BSP_REV, 那是一个表明 BSP 修正数的字符串, 例如:

```
#define BSP_VER_1_2 1
#define BSP_VER_1_1 1
#define BSP_VERSION "1.2" /* A Tornado 2.0 BSP */
#define BSP_REV "/0" /*First Tornado 2.0 revision */
```

config.h 文件也以字符串或是宏的形式来说明 BSP 版本号。BSP_VERSION 宏是一个字符串, 例如 "1.2"。宏 BSP_VER_1_2 被定义为整数 1, 可以在 #if 和 #ifdef 表达中使用。因为 1.2 版本的 BSPs 不与 1.1 版本的 BSPs 兼容, 宏 BSP_VER_1_1 也要被定义。

在 config.h 的结尾, 在工程工具中建立工程时, 添加指令来包含工程参数头文件。

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

A.3 工程问题

Tornado 1.0.1 与 Tornado 2.0 的主要不同是它们的工程工具。对于命令行操作, Tornado 1.0.1 与 Tornado 2.0BSP 没有什么不同。在 Tornado2.0 中, 工程建立方法是全新的, 他们不包含文件 target/config/all。旧的 usrConfig.c 文件已经被分别放到初始化群组 and 组件配置中, 它们由工程建立机制编译和连接。用工程工具建立和从命令行建立是大不相同的。

从命令行建立使用 BSP Make 和 config.h 和 configAll.h 头文件来控制的正在建立的目标的大部分的配置。当使用工程工具建立时, 配置将由工程头文件决定。工程建立不包含 BSP makefile。如果有一个 config.h 或 configAll.h 中的项目, 它通过 CDF 文件描述到工程工具, 并且它仅仅被工程信息所控制; 则对 config.h 和 configAll.h 的改变并不影响它, 它也不影响一个已存在的工程。

注意, 如果在 config.h 或 configAll.h 中有一些工程工具没有注意到的东西, 则 config.h 和 configAll.h 选项在工程建立的过程中优先考虑。

这里有两种产生一个 BSP 默认工程的方法:

- 调用工程工具向导。为了打开这个向导, 从菜单条里选择 File>New Project, 或者如果 Create Project 或 Open Workspace 窗口打开了, 选择 New 标签。
- 使用命令行, 如下所示:

```
make release
```

A.4 产品内容

1.2 版本的 BSPs 的最明显的改变就是产品内容清单，附录 M 中提供了这个内容。

在 Tornado 1.0.1 中，每个 BSP 产品携带了大量的驱动文件，那些文件也是作为部分最基本的 Tornado 产品传送的。这些冗余为用户带来诸多不便，用户不很清楚文件的哪个版本应该被使用。

在 Tornado 2.0 中，BSP 和基本的产品都不包含任何一般的 Wind River 的驱动程序。一般的驱动程序是 CD-ROM 中的一个单个产品，在内容中它们经常被列为“驱动程序目标/头”。这个产品包含网络和 SCSI 驱动程序的所有的目标模块。它也包含其他驱动程序的所有源文件和头文件。

对于 Tornado 2.0 而言还有一个东西是新的——BSP 工程。每个 BSP 应该包含一个默认工程，它包含 BSP 的默认配置。应该鼓励终端用户把新工程基于一个 BSP 的默认工程；而不应该直接从一个 BSP 中开发。

可交付的文件也被修改了。新的文件标准是基于可浏览的 HTML 页。它代替了 UNIX 形式的 man 页。更多的信息请参看附录 L。

A.4.1 产品限制

- 不要提供任何超越产品中的 Wind River 提供的文件或模块：

```
target/config/anything
target/lib/obj$(CPU)$(TOOL)vx/anything
target/config/all/anything
target/src/drv/any.c
target/h/drv/any.c
```

- 不要包含任何在一个被限制的 Wind River 目录下的 BSP 指定的文件或模块（也就是说，第三方必须仅仅传递 target/config/xxxx 目录下的文件）：

```
target/config/all/*
target/src/*
target/h/*
target/lib/*
```

- 不要包含任何与命令行一起使用的 BSP 独立文件。不包含这个文件，构造系统被

强迫重新建立用户第一次给出的 make 命令的信息:

```
target/config/target/depend.target
```

- 不要包含旧式的 man 页:

```
target/man/*.*
```

```
target/man/manX/*.*
```

A.4.2 产品需求

- 产品包含 BSP 指定的文件, 他们一直是 BSP 产品的一部分 (典型的 BSP 中):

```
target/config/bspname/Makefile  
target/config/bspname/README  
target/config/bspname/bootrom.hex  
target/config/bspname/config.h  
target/config/bspname/configNet.h  
target/config/bspname/romInit.a  
target/config/bspname/eyeALib.s  
target/config/bspname/eyeLib.c  
target/config/bspname/sysScsi.c  
target/config/bspname/eyeSerial.c  
target/config/bspname/target.nr  
target/config/bspname/vxWorks  
target/config/bspname/vxWorks.st  
target/config/bspname/vxWorke.sym
```

- 包含工程工具向导产生的工程文件:

```
target/proj/bspname_vx/Makefile  
target/proj/bspname_vx/prjCompa.h  
target/proj/bspname_vx/prjObja.lst  
target/proj/bspname_vx/userAppInit.c  
target/proj/bspname_vx/bspname_vx.wpj  
target/proj/bspname_vx/linkSyma.c  
target/proj/bspname_vx/prjConfig.c  
target/proj/bspname_vx/prjParams.h
```

- 包含工程工具向导产生的默认建立文件:

```
target/proj/bspname_vx/default/*.*
```

- 包含 make man 创建的 BSP 帮助文件:

```
docs/vxworks/bsp/bspname/*.*
```

A.5 小 结

可以通过以下步骤来模拟过渡活动, 它们是被 Wind River 系统采取的, 用来把 BSP 1.1 升级到版本 1.2。

(1) 有的 Wind River BSP 升级到使用 END 形式的驱动程序或可与 SENS1.1 产品兼容的修改了的 BSP 4.4 驱动程序。当 Tornado 1.0.1 的 SENS 产品被传递时才会这样。

(2) 一些 BSP 和接受的驱动程序保持正常的碎片。这些与执行工程或组件没有任何关系。

(3) 宏 BSP_VERSION, BSP_VER_1_2, 和 BSP_REV 被加入到 config.h。这些宏必须插入到优先#include “configAll.h” 的位置, 正如它出现在以下引用的文件中:

```
...
/* New BSP version/revision id */
#define BSP_VER_1_2 1
#define BSP_VER_1_1 1 /* compatible with 1.2 */
#define BSP_VERSION "1.2"
#define BSP_REV "/0" /* increment with each revision */
#include "configAll.h"
...
```

(4) 在 config.h 文件的结尾, 有条件的包含 prjParams.h 文件, 如下所示:

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

(5) 在新版本的 BSP 中 README 文件被修改了。一个新的通道被加到清单的顶部, 它显示了新的版本号和修正号, 例如:

```
1.2/0 - First release for Tornado 2.0.
```

(6) 产生了默认工程文件:

```
make release
```

(7) 产生了默认帮助文件:

```
make man
```

(8) 新的产品被创建了。所有的驱动程序都从 BSP 文件清单中去除了。新的工程基本文件被增加到文件清单中。同时也增加了新的 HTML 文档文件, 以替换目录 `target/man/manX` 中的旧式的 `man` 页。

附录 B 升级 BSP 以兼容 Tornado 2.2

本附录讨论在 VxWorks 5.5 系统中移植 BSP 的问题。所介绍内容包括了独立于体系结构和非独立于体系结构的问题。

B.1 对 BSP 进行独立于体系结构的修改

1. BSP makefile 文件的变化和 bspcnvt2_2 工具

BSP makefile 文件可以通过减少所需要的 include 语句而得到简化。特别优先于 Tornado 2.2 的是一个 makefile 文件需要几个 includes 语句，如例 B-1 所示。

例 B-1：优先于 Tornado 2.2 的 makefile 文件 includes 语句。

```
include $(TGT_DIR)/h/make/defs.bsp
include $(TGT_DIR)/h/make/make.$(CPU)$(TOOL)
include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)
--
include $(TGT_DIR)/h/make/rules.bsp
include $(TGT_DIR)/h/make/rules.$(WIND_HOST_TYPE)
```

在 Tornado 2.2 中，只有 2 个这样的 include 语句是必须的：

```
--
include $(TGT_DIR)/h/make/defs.bsp
--
include $(TGT_DIR)/h/make/rules.bsp
--
```

defs.bsp 文件和 rules.bsp 文件包含建立 BSP 所需要的其他文件。若要修改 BSP，只需在 makefile 文件中为不必要的 includes 语句添加注释。

另外，makefile 文件转换工具 bspCnvt2_2 存放于 installDir/host/\$(WIND_HOST_TYPE) bin 目录下。可用下列句法调用它：

```
% bspCnvt2_2 bspName1 bspName2 ...
```

bspCnvT2_2 通过一下操作对 makefile 文件进行转换:

- 在 BSP 目录下将原 makefile 文件以 Makefile.old 文件名保存。
- 为所有不必要 includes 语句添加注释。
- 提醒用户所有使用 16 进制卷标的地方 (见“十六进制的使用和 objcopy”部分内容)。

2. 十六进制的使用和 objcopy

我们这里不推荐使用 Wind River 公司提供的十六进制和二进制使用方法, 即 aoutToBinDec 和 coffHexArm, 而提倡使用 GNU 提供的 objcopy。请参见 GNU 工具包用户指导一章关于二进制使用的介绍。

3. WDB_COMM_TPYE 的新默认值

WDB_COMM_TPYE 的默认值由 WDB_COMM_NETWORK 变为 WDB_COMM_END。若想使用一种不同的通信模式, 可把它明确定义为 config.h。例如, 若默认 BSP 设置 WDB 为串行通信, 应在 config.h 中包含下列语句:

```
--
/* make sure this appears after inclusion of configAll.h */
#define WDB_COMM_TYPE WDB_COMM_SERIAL
...
```

4. 共享存储器子系统的变化

为了支持共享存储器网络和可选择组件 VxMP, VxWorks 5.5 的板级支持包 BSP 需要对 config.h 和 sysLib.c 做适当修改。在过去发行的 VxWorks 版本中, 要确定包含共享存储器组件只需通过证实包含共享存储器底板网络即可。但是在 VxWorks 5.5 中, 可以对 VxMP 进行配置而无需支持共享存储器网络。

一个具有包含宏 INCLUDE_SM_COMMON 的新组件被添加到 VxWorks 5.5 中。可以使用此宏来测试共享存储器支持。大多数支持共享存储器的 BSP 都使用如下的条件编辑语句:

```
#ifdef INCLUDE_SM_NET
    /* shared memory-specific code */
#endif
```

在 VxWorks 5.5 中, 这些语句必须升级以便测试 INCLUDE_SM_COMMON:

```
#ifdef INCLUDE_SM_COMMON
    /* shared memory-specific code */
```

```
#endif
```

修改 `sysLib.c` 时, 遵循把所有 `INCLUDE_SM_NET` 修改为 `INCLUDE_SM_COMMON` 的简单原则。

除了一些特殊情况, 我们可以使用同样的方法修改 `config.h`。对 `INCLUDE_SM_NET` 的测试在与网络有关的语句中依然有效, 但在测试如下普通共享存储器参量时无效:

<code>SM_ANCHOR_ADRS</code>	<code>SM_ANCHOR_OFFSET</code>	<code>SM_CPUS_MAX</code>
<code>SM_INT_ARG1</code>	<code>SM_INT_ARG2</code>	<code>SM_INT_ARG3</code>
<code>SM_INT_TYPE</code>	<code>SM_MASTER</code>	<code>SM_MAX_WAIT</code>
<code>SM_MEM_ADRS</code>	<code>SM_MEM_SIZE</code>	<code>M_OBJ_MEM_SIZE</code>
<code>SM_OFF_BOARD</code>	<code>SM_TAS_TYPE</code>	

这些共享存储器参数不需要条件编译而且可以预留定义 (注意一些参数定义为 `configAll.h` 里面的默认值, 而且在 `config.h` 重新定义以前, 必须要首先取消定义)。

在 `config.h` 里对 `INCLUDE_SM_NET` 的定义会引入 VxWorks 5.5 中改变的组件, 比如 `INCLUDE_NET_SHOW` 和 `INCLUDE_BSD`。这些成分可以被舍弃; `smNetShow()` 程序现在是一个独立组件, 叫做 `INCLUDE_SM_NET_SHOW`, 适当的 BSD 和其他网络配置包含在其他的文件中。

5. 其他运行时功能的改变

几个 Tornado 2.2/VxWorks 5.5 的可选择产品需要有一些变动, 这样一来, 对 BSP 的修改就很必要。

- 真 Flash 文件系统 (TrueFFS)。具体内容请见《VxWorks 程序员指南》Flash 存储器设备接口一章。也可以查询《VxWorks API 参考手册》中 `tffsConfig` 和 `tffsDrv` 库的入口。
- 通用串行总线架构 (USB)。详见《USB 开发工具包程序员指南》1.1.2, 和《USB 开发工具包发行注释》1.1.2。
- DosFs 2.0。这个支持 VxWorks 的 DOS 文件系统升级版本使改变 BSP 成为必要。由于 `dosFsNLib 2.0` 版本在 Tornado 2.0 发行不久后也出台了, 很多 Tornado 2.0.x 和以后的 BSP 已经可以支持它, 而且不需要做任何改变。

B.2 独立于体系结构的 BSP 问题

以下各节内容提供了升级 Tornado 2.2/VxWorks 5.5 的板机支持包的具体步骤, 以及特定体系结构 BSP 的升级信息。其他特定结构的信息, 参考相关的《VxWorks 结构增刊手册》(由 WindSurf 出版)。



注意：本节所介绍的结构信息在出版之日是准确无误的。关于现在的结构信息，参见《Tornado 移植指导》2.2（可在 Wind River 支持网站找到）。

B.2.1 对所有体系结构的通用移植变化



注意：本节所提供的材料包括 B.1 介绍的独立于体系结构的 BSP 部分。

以下变化是所有 BSP 均需要的，无论它们的结构如何：

- 升级 makefile 文件。所有用户都需要执行此步。使用 bspCnvT2_2 脚本来升级 BSP makefile 文件。这个脚本文件对无用 include 语句和任何现有 HEX_FLAGS 值进行注释。
- TFFS 支持。BSP 需要此步的 TFFS 支持。从 syslib.c 文件中删除包含 sysTffs.c 文件的语句。

B.2.2 68K/CPU32

所有由 Wind River 提供的与 Tornado 2.2 产品共同发行并适用于 68K/CPU32 的 BSP 都已升级以适应 VxWorks 5.5 的使用。传统的基于 VxWorks 5.4 的 BSP 只需要根据 B.2.1 节所有体系结构的通用移植变化所介绍的内容做相应修改以便与 VxWorks 5.5 兼容，而无需进行特定体系结构的修改。

关于在 68K/CPU32 目标板中使用 VxWorks 的问题，请参见《基于 68K/CPU32 的 VxWorks 体系结构附录》。

B.2.3 ARM



注意：本章介绍从 Tornado 2.1\VxWorks 5.4 到 Tornado 2.2\VxWorks 5.5 的 BSP 变化。关于如何将 BSP 从 Tornado 2.0.x 升级到 Tornado 2.1 请参见 WindSurf 提供的基于 ARM 的 Tornado 发行与结构附录手册。

除了在 B.2.1 所有体系结构的通用移植变化中介绍的步骤之外，下面 ARM 特定移植变化也是需要的：

- 只用于汇编文件。增加了新宏 GTEXT, GDATA, FUNC 和 FUNC_LABEL 用以协助装载汇编文件。通用汇编标题名的前置下划线应去掉。使用这些宏可使

Tornado 2.1.x 和 Tornado 2.2 之间得以兼容。

- Diab 支持。由于 GNU 和 Diab 工具链之间汇编语法的不同，需要把 GNU 汇编宏改为 Diab 的语法形式。关于 Diab 的汇编语法详情请参见《基于 ARM 的 Diab C/C++ 编译器用户指导》。

有关以 ARM 为目标的 VxWorks 的更多信息请参见《基于 ARM 的 VxWorks 体系结构附录》。

B.2.4 ColdFire

除了在 B.2.1 所有体系结构的通用移植变化中介绍的内容，下面有关特定 ColdFire 的问题也需要在移植用户 BSP 时予以考虑。

- Diab 支持。基于 ColdFire 的 VxWorks 含有与以前 Tornado 2.1/VxWorks 5.4 相同的结构和功能。然而，它不再支持 GNU 工具链。

关于在 ColdFire 目标板中使用 VxWorks 的具体问题请参见《基于 ColdFire 的 VxWorks 体系结构手册》。

B.2.5 MIPS



注意：本章介绍从 Tornado 2.1/VxWorks 5.4 到 Tornado 2.2/VxWorks 5.5 的 BSP 移植。有关 BSP 从 Tornado 2.0.x 到 Tornado 2.1 的升级方法，请见 WindSurf 提供的基于 MIPS 的 Tornado 发行注释与结构附录手册。

除了在 B.2.1 所有体系结构的通用移植变化中介绍的内容，下面有关 MIPS 的特定问题也需要在升级用户 BSP 到 Tornado 2.2/VxWorks 5.5 时予以考虑。

- CPU 变量。MIPS CPU 由 CPU 变量组织起来。这允许 VxWorks 内核利用一个变量的特定体系结构特征而不相互给对方以消极影响。结果是，所有 MIPS BSP 现在均需在 makefile 文件后的 MACH_EXTRA 行后加入 CPU_VARIANT 行。例如，归为 Vr54xx 变量的 CPU 使用以下语句：

```
CPU_VARIANT = _vr54xx
```

关于 MIPS CPU 的目录及相关 CPU_VARIANT 值参看《基于 MIPS 的 VxWorks 体系结构手册》。

- MIPS64 库。MIPS64 库 (MIPS64gnu, MIPS64diab, MIPS64gnule 和 MIPS64diab) 支持 64 比特 ISA 三级以上版本的 MIPS 工具。在以前版本的 VxWorks 中，这些库只支持 ISA 四级以上版本的 MIPS 工具。关于 MIPS 库的编译器选择，请见《体系结构附录》。

- **Achemy Semiconductor BSP。**改变 Achemy Semiconductor BSP—pb1000 为 pb1500BSP 提供支持。因此需要在 API 中做相应变化来为这两个 BSP 提供常用支持。所有以前利用 au1000 的宏、驱动器和文件名变为简单形式 au。例如：高速缓冲存储器库 cacheAu1000Lib 改为 cacheAuLib。关于这些变化请参阅 BSP 及其支持驱动器。

有关在 MIPS 目标板中使用 VxWorks，请见《基于 MIPS 的 VxWorks 体系结构手册》。

B.2.6 奔腾处理器

除去在 B.2.1 所有体系结构的通用移植变化中介绍的内容，下面有关奔腾的特定问题也需要在升级用户 BSP 到 Tornado 2.2/VxWorks 5.5 时予以考虑。

- **新的 CPU 型号 PENTIUM2, PENTIUM3 和 PENTIUM4 被添加进来并删除 CPU_VARIANT。**因此，应该用一个与用户处理器兼容的 CPU 取代 CPU_VARIANT。
- **在这个版本中添加了 3 个新的编码选择器 sysCsSuper, sysCsExc 和 sysCsInt，去掉了 sysCodeSelector。**现有的 BSPs 中，应用 sysCsSuper 来代替 sysCodeSelector。
- **由于 GNU 汇编器的改进，ROM_IDTR, ROM_GDTR, ROM_GDT, ROM_INIT2 这些偏移量宏已经被删去了，在 romInit.s 文件中也不再使用这些宏了。**
- **仅用于汇编文件：新的宏 GTEXT, GDATA, FUNC 和 FUNC_LABEL 添加进来协助装载汇编文件。**通用汇编标题名的前置底线应去掉。使用这些宏使 Tornado 2.0.2 和 Tornado 2.2 之间得以兼容。
- **仅用于汇编文件：用.balign 代替.align。**
- **PC 主机工具 mkboot 现使用已知的 VxWorks 名，但不能使用用户提供名，除非它们是二进制文件 (*.bin)。**对于所有其他映像，有以下 2 种选择：
 - 在运行 mkboot 前把映像改为 bootrom.dat。
 - 修改 mkboot.dat 以支持用户名。遵循 mkboot.bat 文件里的例子。
- **电源管理由默认值决定。若使之失效，修改 config.h：**

```
#undef VX_POWER_MANAGEMENT
```

- **默认值控制台设置在 COM1 中。在以前版本的 VxWorks 中，x86 目标板设置默认控制台为 VGA 控制台。改变 config.h 以使用 VGA 控制台：**

```
#define INCLUDE_PC_CONSOLE
```

- **IDE 驱动器的结构参数 ideDrv 由 ATA 驱动器 ataDrv 所代替，这已在 Tornado 2.0 的默认配置中所使用。**
- **升级 CPUID 结构 (sysCpuId) 来支持奔腾 III 和奔腾 IV 处理器。**sysCpuId.version, sysCpuId.vendor 和 sysCpuId.feature 分别被 sysCpuId.singature, sysCpuId.signature,

`sysCpuId.vendorId` 和 `sysCpuId.featuresEdx` 所代替。

- 程序 `sysCpuProbe()` 支持奔腾 III 和奔腾 IV 处理器。
- 升级程序 `sysIntEoiGet()`。
- 本地和 IO APIC/xAPIC 驱动器。 `IoApicIntr.c`, `ioApicIntr.c` 和 `IoApicTimer.c` 现支持奔腾 IV 里的 xAPIC。这些驱动器的显示程序分别存在于 `IoApicIntrShow.c` 和 `ioApicIntrShow.c` 中。

有关在奔腾目标板中使用 VxWorks 的更多信息, 请见《基于奔腾的体系结构手册》。

B.2.7 PowerPC

除了在“B.2.1 所有体系结构的通用移植变化”中介绍的内容之外, 下面有关 PowerPC 的特定问题也需要在升级用户 BSP 到 Tornado 2.2/VxWorks 5.5 时予以考虑。

- 反对使用程序 `vxImmrGet()`。现有 BSP 用不同方法执行此程序: 有的返回整个 IMMR 寄存器, 有的去掉 PARTNUM 比特。BSP 的现有操作不变。

首选 `vxImmrLsbGet()` 和 `vxImmrDevGet()` 来代替程序, 它们在 `vxALib.s` 中执行, 不应被 BSP 所覆盖。标准 Wind River 驱动器使用新的接口。

- 一些早期 MPC74xx/Altivec 支持包含一个程序来初始化 Altivec 异常向量, 典型的是 `vmxExcLoad()`。如:

```
{
    bcopy ((char*)(LOCAL_MEM_LOCAL_ADRS + 0x0100),
           (char*)(LOCAL_MEM_LOCAL_ADRS + _EXC_VMX_UNAVAIL),
    bcopy ((char*)(LOCAL_MEM_LOCAL_ADRS + 0x0100),
           (char*)(LOCAL_MEM_LOCAL_ADRS + _EXC_VMX_ASSIST),
           sizeof_exception),
}
```

这样的编码必须删除。Altivec 异常向量由 `altivecInit()` 初始化。

下列变化在 PowerPC BSPs 中是可选择的:

- 改变汇编文件以使用在 `installDir/target/h/arch/ppc/toolsPpc` 目录下的 Wind River 标准宏:

```
FUNC_EXPORT
FUNC_IMPORT
_WRS_TEXT_SEG_START
FUNC_BEGIN
FUNC_LABEL
FUNC_END
```

通常情况下不需要用这种方法改变汇编文件。然而，这种改变（特别是变为 `_WRS_TEXT_SEG_START`）有时会产生一个无声干扰。

有关以 PowerPC 为目标板的 VxWorks 使用方法详情请见《基于 PowerPC 的 VxWorks 体系结构手册》。

B.2.8 XScale/StrongARM



注意：本节介绍如何使 BSP 从 Tornado 2.1/VxWorks 5.4 升级到 Tornado 2.2/VxWorks 5.5。有关 BSP 从 Tornado 2.0.x 升级到 Tornado 2.1 参见 WindSurf 提供的有关 StrongARM/XScale 的 Tornado 发行注释及结构手册。

除去在 B.2.1 所有体系结构的通用移植变化中介绍的内容之外，下面有关 Xscale/StrongARM 的特定问题也需要在升级用户 BSP 到 Tornado 2.2/VxWorks 5.5 时予以考虑。

- 只用于汇编文件：新的宏 `GTEXT`, `GDATA`, `FUNC` 和 `FUNC_LABEL` 添加进来协助装载汇编文件。通用汇编标题名的前置下划线应去掉。使用这些宏可使 Tornado 2.1.x 和 Tornado 2.2 之间得以兼容。
- Diab 支持。由于 GNU 和 Diab 工具链之间汇编语法的不同，需要把 GNU 汇编宏改为 Diab 的语法形式。关于 Diab 的汇编语法详情请见《基于 ARM 的 Diab C/C++ 编译器用户指导》。

关于在 XScale/StrongARM 目标板中使用 VxWorks，参阅《基于 Intel XScale/StrongARM 的 VxWorks 结构手册》。

B.2.9 SuperH

除了在 B.2.1 所有体系结构的通用移植变化中介绍的内容，下面有关 SuperH 的特定变化也需要在升级用户 BSP 到 Tornado 2.2/VxWorks 5.5 时予以考虑。

- 电源管理设置。BSP 需要此步来使处理器电源管理得到授权。`sysHwInit()` 程序的 `sysLib.c` 文件初始化取决于所使用的 SuperH 处理器的 `vxPowerModeRegs` 结构。
- Diab 支持。当 BSP 需用 Diab 工具链构建的时候使用此步。升级汇编文件，使用 `.short` 指示代替 `.word` 指示。
- 使用 `NULL`。在以前的版本中，`NULL` 被定义为整数 0。这种定义需适应 C 语言标准，使之成为一个空指针。为了避免编译器告警，确保 `NULL` 只分配给指针。

有关在 SuperH 目标板中使用 VxWorks，参阅《基于日立公司 SuperH 的 VxWorks 结构手册》。

附录 C 升级 BSP 以兼容 VxWorks 网络堆栈

像增加别的应用程序一样把驱动程序加入到目标板 VxWorks 系统中。第一步要做的就是汇编和包含 VxWorks 映像中的驱动程序代码。有关的一般步骤请参看《Tornado 用者指南》，它告诉你如何汇编源代码和如何产生适合的目标板代码。

在 VxWorks 映像中包含大多数的软件模块所要做的工作不过是建立一些 `#define`。增加一个网络接口驱动程序并不需要那么多。然而，因为 VxWorks 允许创建不止一个网络驱动程序，也必须建立一个表格来把 `#define` 分组成指定的驱动程序组。这个表格在 `target/src/config/BSP/configNet.h` 中定义，在那里 BSP 是你的板级支持包的名称，例如 `mv162` 和 `pc486`。

例如，如果想要 VxWorks 创建两个网络驱动程序，一个支持缓冲器输出，另外一个不支持，应该首先编辑 `configNet.h` 来包含以下的叙述：

```
/* Parameters for loading the driver supporting buffer loaning. */
#define LOAD_FUNC_0 ln7990EndLoad
#define LOAD_STRING_0 "0xfffffe0:0xffffffffe2:0:1:1"
#define BSP_0 NULL

/*Parameters for loading the driver NOT supporting buffer loaning.*/
#define LOAD_FUNC_1 LOAD_FUNC_0
#define LOAD_STRING_1 "0xffffee0:0xffffffffee2:4:1:1"
#define BSP_1 NULL
```

为这些内容设置合适的值，考虑以下的问题：

1. END_LOAD_FUNC

指定你的驱动程序的 `endLoad()` 入口点的名称。例如，如果驱动程序的 `endLoad()` 入口点的名称是 `ln7990EndLoad()`，应该编辑 `config.h` 来包含以下行：

```
#define END_LOAD_FUNC ln7990EndLoad
```

2. END_LOAD_STRING

指定进入 `muxDevLoad()` 的初始化字符串的 `initString` 参数。

也必须编辑 `endTbl`（在 `configNet.h` 中的一个表格，它指定了包含在映像中的 `ENDs`）的定义来包含以下内容：

```
END_TBL_ENTRY endTbl  
{  
  { 0, LOAD_FUNC_0, LOAD_STRING_0, BSP_0, FALSE},  
  { 1, LOAD_FUNC_1, LOAD_STRING_1, BSP_1, FALSE},  
  { 0, END_TBL_END, 0, NULL},  
};
```

每个表格入口中的第一个数字指定了设备的单元数字。在以上例子中的第一个入口指定了单元 0。因此，这个装载的驱动程序是 `deviceName0`。在每个入口的结尾的 `FALSE` 表明了入口还未被处理。在系统成功的加载一个驱动程序后，在这个表格的运行时它改变这个值为 `TRUE`。为了防止系统自动的加载你的驱动程序，把这个值设为 `TRUE`。

最后，必须编辑你的 BSP 的 `config.h` 文件来定义 `INCLUDE_END`。这使建立程序包含 `END/MUX` 接口。在一这点上，准备好了重新建立 VxWorks 来包含你的新驱动程序。当启动这个重新建立的映像时，它与其他表格中陈列的每个指定的驱动程序调用 `muxDevLoad()`。

附录 D VxWorks 引导顺序

表 D-1 记录了一个只读存储器 VxWorks 或者根只读存储器典型的初始化过程。此过程只适用于传统命令语言构建的典型。



注意：预期在未来的版本中，这种老的初始化过程将被工程和组件初始化组所取代。

由项目工具生成的映像遵循一个不同但又相似的初始化顺序，这个顺序是基于在组件描述文件中描述的 `initGroups` 的。有关项目组件初始化命令参阅附录页 E.3。

表 D-1 VxWorks 启动顺序

程 序	作 用	文 件
1. <code>romInit()</code>	(a) 禁止中断	<code>romInit.s</code>
	(b) 保存启动类型（冷启/热启）	
	(c) 激活和刷新 DRAM（独立于硬件）	
	(d) 分支到 <code>romStart()</code> （位置独立编码）	
2. <code>romStart()</code>	(a) 从 ROM 中复制文本或数据段到 RAM	<code>bootInit.c</code>
	(b) 清除内存（若需要）	
	(c) 解压缩（若需要）	
	(d) <code>if(i960CPU)</code> 激活 <code>sysInitAlt(boot Type)</code> <code>else</code> 激活 <code>usrInit(根型)</code>	
3. <code>sysInitAlt()</code> （只用于 i960）	(a) 使高速缓冲存储器无效	<code>sysALib.s</code>
	(b) 用默认残余初始化系统中断表	
	(c) 用默认残余初始化系统错误表格	
	(d) 初始化所有处理寄存器为默认值	
	(e) 执行跟踪	
	(f) 清除所有挂起中断	
	(g) <code>usrInit(boot Type)</code>	<code>usrConfig.c</code>
4. <code>usrInit()</code>	(a) 确认正确数据段位置。若排列不正确产生无限循环。	
	(b) <code>if (SPARC CPU)</code> 初始化积存器窗口管理	<code>usrConfig.c</code>

续表

程 序	作 用	文 件
4. usrInit()	(c) if (SYS_HW_INIT_0) 执行宏 SYS_HW_INTT_0	
	(d) if (CACHE_SUPPORT) cacheLibInit()	
	(e) if (MIP CPU) sysGpInit(), 初始化通用指针	sysALib.a
	(f) zero bss (未初始化数据)	usrConfig.c
	(g) 在 sysStartType 中保存引导类型	
	(h) intVecBaseSet(), 设置向量基础表	lib/*.a
	(i) excVecInit(), 初始化所有系统和默认中断向量	lib/*.a
	(j) sysHwInit(), 程序初始化依赖板机的硬件	sysLib.c
	(k) usrKernelInit(), 程序配置 wind 内核	usrKernel.c
	(l) if (instruction cache) 激活指令缓冲器 if (data cache) 激活数据缓冲器	
	(m) kernelInit()	lib/*.a
5. kernelInit()	初始化和启动内核	lib/*.a
	(a) intLockLevelSet()	lib/*.a
	(h) if (interrupt stack) 在内存池开端创建中断堆栈	
	(c) 在存储器末尾创建根堆栈和 TCB	
	(d) taskInit(tRoot Task)	lib/*.a
	(e) taskActivate(usrRoot)	lib/*.a
	(f) usrRoot()	usrConfig.c
6. usrRoot()	初始化输入输出系统、安装驱动器、创建工具、依照 configAll.h 和 config.h 设置网络	usrConfig.c
	(a) memInit() 程序初始化存储器 if (show routines) memShowInit() if (virtual memory) usrMmuInit()	lib/*.a lib/*.a usrMmuInit.c
	(b) sysClkConnect()	sysLib.c
	(c) sysClkRateSet()	sysLib.c
	(d) sysClkEnable()	sysLib.c
	(e) selectInit()	

续表

程 序	作 用	文 件
	(f) iosInit()	lib/*.a
	(g) if(INCLUDE_TTY_DEV)and(NUM_TTY) ttyDrv() 建立控制台端口、标准输入输出及错误	ttyDrv.c
	(h) if (target symbol table) hashLibInit() andsymLibInit()	lib/*.a
	(i) If (LSTLIB) lstLibInit()	
	(j) 初始化异常处理程序 exclnit() if (logging) , logInit() if (singal), siglnit()	lib/*.a lib/*.a lib/*.a
	(k) if (native debugging) dbgInit()	lib/*.a
	(l) if (pipes) pipeDrv()	lib/*.a
	(m) if (standard I/O) stdioInit()	lib/*.a
	(n) if (POSIX_SIGNALS) sigqueueInit()	
	(o) if (POSIXsemaphores) semPxLibInit() if (show routines) semPxShowInit()	lib/*.a
	(p) if (POSIX_THREADS) pthreadLibInit()	
	(q) if (POSIX message queues) mqPxLibInit() if (show routines) mqPxShowInit()	lib/*.a
	(r) if (POSIX asynchronous I/O) aioPxLibInit() if (show routines) aioPxShowInit()	
	(s) if (CBIO), cbioLibInit()	
	(t) if(DOS file system) dosFsInit()	lib/*.a
	(u) if (raw file system) rawFsInit()	lib/*.a

续表

程 序	作 用	文 件
	(v) if (rt11 file system) rt11FsInit()	lib/*.a
	(w) if (RAM disk) ramDrv()	lib/*.a
	(x) if (USB) usbInit()	
	(y) if (SCSI-2 support) scsiIfInit() if (SCSI-1 support) scsiIfInit() sysScsiInit() usrScsiConfig()	sysLib.c usrScsi.c sysScsi.c
	(z) if (floppy disk support) fdDrv()	
	(aa) if (IDE disk support) ideDrv()	
	(ab) if (ATA driver support) ataDrv()	
	(ac) if (parallel printer support) lptDrv()	
	(ad) if (PCMCIA support) pcmciaInit()	
	(ae) if (flash file system support) rffsDrv()	
	(af) if (formatted I/O support) fioLibInit()	
	(ag) if (floating point support) floatInit()	lib/*.a
	(ah) if (software floating-point) mathSoftInit() if (hardware floating-point) mathHardInit() if (hardware floating-point and show routines) fppShowInit()	lib/*.a lib/*.a lib/*.a
	(ai) if (DSP) usrDsplInit()	
	(aj) if (AltiVec) usrAltiVecInit()	

续表

程 序	作 用	文 件
	(ak) if (spy) spyStop()	lib/*.a
	(al) if (timex) timexInit()	lib/*.a
	(am) if (environment variables) envLibInit()	lib/*.a
	(an) if (loader) moduleLibInit()	lib/*.a
	(ao) if (a.out format) loadAoutInit() else if (ecoff format) loadEcoffInit() else if (coff format) loadCoffInit() else if (ELF format) loadElfInit()	lib/*.a lib/*.a lib/*.a lib/*.a
	(ap) if (SYM_TBL_SYNC) symSyncLibInit()	
	(aq) if (network initialization) usrBootLineInit() usrNetInit()	usrNetwork.c usrNetwork.c
	(ar) if (shared memory objects) usrSmObjInit()	usrSmObj.c
	(as) if (VxFusion) usrVxFusionInit()	
	(at) if (text protect and VxVMI) vmTextProtect()	lib/*.a
	(au) if (vector table protect and VxVMI) intVecTableWriteProtect()	lib/*.a
	(av) if (select) selTaskDeleteHookAdd()	lib/*.a
	(aw) if (symbol table) symTblCreate() netLoadSymTbl()	lib/*.a usrLoadSym.c
	(ax) if (C++) cplusCtorsLink() cplusLibInit()	lib/*.a lib/*.a
	(ay) if (COM) comLibInit()	

续表

程 序	作 用	文 件
	(az) if (DCOM) dcomLibInit()	
	(ba) if (http) httpd()	
	(bb) if (WindView) windviewConfig()	
	(bc) wdbConfig(): 确认并初始化目标板	lib/*.a
	(bd) if (shell) shellInit()	lib/*.a
	(be) if (WindML) usrWindMlInit()	
	(bf) if (Java) javaConfig()	lib/*.a
	(bg) if (HTML) usrHtmlInit()	lib/*.a
	(bb) if (USER_APPL) 执行 USER_APP_INIT 宏	sysLib.o

附录 E 组件语言

E.1 组件描述语言 (CDL)

本附录提供了描述工程配置工具的一个组件的组件描述语言(CDL)技术上的简介。CDL 用一个组件描述文件(CDF)来描述一个或更多的组件。按照规定, 一个 CDF 有文件扩展名.cdf。这个后缀提供了五个 CDL 目标类型的句法; 更多的信息请参看“6.2.2 CDL 目标类型”。

特性描述了它们的目标, 而且还描述了相关的组件。在本附录中, 如果一个特性不指定一个默认值, 就什么都不存在了。

本附录也包含了 Tornado 2.x 的默认文件夹等级, 还有默认的初始化群组顺序。文件夹等级决定了一个组件出现在工程工具的组件等级的什么位置。初始化群组顺序决定了在系统启动时哪个文件夹中的一个组件被初始化。

E.1.1 组件特性

Component <i>component</i> {	// required for all components
NAME <i>name</i>	// readable name (e.g., "foo manager").
	// should be in all lower case.
SYNOPSIS <i>desc</i>	// one-line description
MODULES <i>m1 m2 ..</i>	// object modules making up the service.
	// used to generate dependency
	// information.
	// it is important to keep this list
	// small,since the tool's dependency
	// engine assumes that the component is
	// included if *any* of the modules are
	// dragged in by dependency. It may make
	// sense to split a large number of
	// modules into several distinct
	// components.

```

CONFIGLETTES l s2 ..           // source files in the component that are
                                // #included in the master configuration
                                // file.
                                // file paths are assumed to be relative
                                // to $(WIND_BASE)/target/config/comps/src
BSP_STUBS s1 s2 ..             // source file stubs that should be copied
                                // into the BSP and customized for the
                                // component to work.
                                // file paths are assumed to be relative
                                // to $(WIND_BASE)/target/config/comps/src
                                // (BSP_STUBS are supported for Tornado 2.2
                                // only; they are not supported in earlier
                                // Tornado 2.x releases)
HDR_FILES h1 h1 ..            // header files that need to be included
                                // to use this component. Typically
                                // contains prototypes for the
                                // initialization routine.
CFG_PARAMS p1 p2 ..           // configuration parameters, typically
                                // macros defined in config[All].h, that
                                // can change the way a component works.
                                // see Parameters, below, for more info.
INTT_RTN init(..)             // one-line initialization routine.
                                // if it needs to be more than one line,
                                // put the code in a CONFIGLETTE.
LINK_SYMS s1 s2 ..            // reference these symbols in order to drag
                                // in the component from the archive.
                                // this tells the code generator how to
                                // drag in components that don't need to
                                // be initialized.
REQUIRES r1 r2 ..             // other components required. Note:
                                // dependencies are automatically calculated
                                // based on a components MODULES and
                                // LINK_SYMS. For example, because nfsLib.o
                                // calls rpcLib.o, the tool is able to
                                // figure out that INCLUDE_NFS requires
                                // INCLUDE_RPC. One only needs to list
                                // requirements that cannot be detected from

```

```

// MODULE dependencies. Typically only needed
// for components that don't have associated
// MODULES (e.g., ones with just
// configlettes).
EXCLUDES el el .. // other components that cannot
// coexist with this component
HELP h1 h2 .. // reference pages associated with the
// component.
// The default is the MODULES and INIT_RTN
// of the component. For example, if the
// component has MODULE fooLib.o, then the
// manual page for fooLib is automatically
// associated with the component (if the
// manual page exists). Similar for the
// components INIT_RTN. If there are any
// other relevant manual pages, they can
// be specified here.
MACRO_NEST expr // This is for Wind River internal use only.
// It is used to tell the bsp2prj script
// how to create a project corresponding
// to a given BSP.
// The "bsp2prj" script assumes that if
// a BSP has macro INCLUDE_FOO defined,
// the corresponding component INCLUDE_FOO
// should be included in the project.
// That is not always the case. For example,
// The INCLUDE_AOUT component should only
// be included if both INCLUDE_AOUT *and*
// INCLUDE_LOADER are defined.
// So for the INCLUDE_AOUT component, we set
// MACRO_NEST = INCLUDE_LOADER.
// Similarly all WDB subcomponents have
// MACRO_NEST = INCLUDE_WDB.
ARCHIVE al // archive in which to find the MODULES.
// default is lib$(CPU)$(TOOL)vx.a.
// file path is assumed to be relative
// to $(WIND_BASE)/target/lib.

```



```

// any archive listed here is
// automatically added to the VxWorks
// link-line when the component is
// included.
// Note: the tool only analyzes archives
// associated with included components.
// This creates a chicken-and-egg problem,
// because the tool analyzes components
// before they are actually added.
// So if you add a component with an ARCHIVE,
// analysis will be done without the ARCHIVE.
// As a work-around, if a separate archive is
// used, create a dummy component that
// lets the tool know that a new archive
// should be read. Such a component
// should be called INSTALL_something.
// It should contain only NAME, SYNOPSIS,
// and ARCHIVE attributes. Only after the
// user adds it can he or she add other
// components from the archive.
INCLUDE_WHEN c1 c2 .. // automatically include this component
// when some other components are included.
// All listed components must be included to
// activate the INCLUDE_WHEN (AND
// relationship). This allows, for example,
// msgQShow to be included whenever msgQ and
// show are included. Similarly, WDB fpp
// support can be included when WDB and fpp
// are included.
INIT_BEFORE c1 // if component c1 is present, our init
// routine must be called before c1.
// Only needed for component releases.
_CHILDREN fname // Component is a child of folder or
// selection fname.
_INIT_ORDER gname // Component is a member of init group gname
// and is added to the end of the
// initialization sequence by default (see

```

```

    // INIT_BEFORE).
}

```

E.1.2 参数特性

```

Parameter parameter {
  NAME name                // readable name (e.g., "max open files")
  SYNOPSIS desc            // one-line description.
  STORAGE storage          // MACRO, REGISTRY, ... Default is MACRO.
  TYPE type                // type of parameter:
                           // int, uint, bool, string, exists.
                           // Default is untyped.
                           // more types will be added later.
  DEFAULT value            // default value of the parameter.
                           // default is none - in which case the user
                           // must define a value to use the component.
                           // for parameters of type "exists," the
                           // value is TRUE if the macro should be
                           // defined, or FALSE if undefined.
}

```

E.1.3 文件夹特性

工程工具组件等级使用文件夹来把组件逻辑分组。不是在一个表中列出所有的组件，而是工程工具按等级合理排列，如在 `configAll.h` 中。例如，在顶端的文件夹在头文件下组织组件，例如网络、驱动程序、OS 和应用程序等。

```

Folder folder {
  NAME name                // readable name (e.g., "foo libraries").
  SYNOPSIS desc            // one-line description
  CHILDREN i1 i2 ..        // containers and components
                           // that belong to this container.
  DEFAULTS i1 i2 ..        // default CHILDREN.
                           // if the folder represents a complex
                           // subsystem (such as the WDB agent),
                           // this is used to suggest to the user
                           // which components in the folder are

```



```

// considered "default." That way the user
// can add the whole subsystem at once,
// and a reasonable set of subcomponents
// will be chosen.
}

```

E.1.4 选择特性

```

Selection selection {
NAME name // readable name (for example , "foo
// communication path")

SYNOPSIS desc // one-line description
COUNT min-max // range of allowed subcomponents.
// 1-1 means exactly one.
// 1- means one or more.

CHILDREN i1 i2 .. // components from which to select
DEFAULTS i1 i2 .. // default CHILDREN.
// this is not used for anything except to
// to suggest to the user which components
// in the selection we consider "default."
}

```

E.1.5 初始化群组特性

```

InitGroup group {
INIT_RTN rtn(..) // initialization routine definition
INIT_ORDER i1 i2 .. // ordered list of init groups and
// components that belong to this init group

INIT_AFTER i2 i2 .. // Only needed for component releases.
}

```

E.2 文件夹等级

以下的等级是Tornado 2.x的默认文件夹目录和仅仅在Tornado 2.x开发环境中的上下文

应该被参考的。

 **注意：** 不要假设以后的版本能保存包含在 Tornado 2.x 文件夹等级中的任何文件夹的名称或是特性，除了 FOLDER_ROOT。

```

FOLDER_ROOT
{ // all components
FOLDER_APPLICATION           // user application component(s)
FOLDER_TOOLS                  // Development Tools
{
FOLDER_WDB                    // WDB agent components
{
FOLDER_WDB_OPTIONS            // optional WDB agent services
}
FOLDER_SHELL                  // Target shell components
FOLDER_LOADER                 // OMF loader components
FOLDER_SYMTBL                 // symbol table components
{
FOLDER_SYM_TBL_INIT           // Symbol table initialization
}
FOLDER_SHOW_ROUTINES          // Kernel show routine components

E
FOLDER_WINDVIEW               // WindView components
}
FOLDER_NETWORK                // Network components
{
FOLDER_CORE_COMPONENTS        // basic network initialization options
{
FOLDER_BOOTLINE_SETUP         // network init from bootline
}
FOLDER_NET_DEV                // Support for network device types
{
FOLDER_SMNET_COMPONENTS       // Shared memory component
FOLDER_BSD_NET_DRV            // BSD Ethernet drivers
}
FOLDER_NET_PROTOCOLS          // networking protocols

```

```
{
FOLDER_NET_FS                // Network file systems
{
FOLDER_NET_REM_IO            // Network remote I/O components
}
FOLDER_NET_APP                // Network applications
{
FOLDER_MIB2                  // MIB2 application
FOLDER_STREAMS               // Streams application
}
FOLDER_NET_API                // Networking APIs
FOLDER_NET_ROUTE              // Network routing protocols
FOLDER_NET_DEBUG              // Network show routines
FOLDER_TCP_IP                 // Core TCP/IP components
{
FOLDER_TCPIP_CONFIG          // TCP/IP configuration components
}
}
}
FOLDER_CPLUS                  // C++ class libraries
{
FOLDER_CPLUS_STDLIB           // C++ Standard Library
FOLDER_CPLUS_WFC              // Wind Foundation Classes
}
FOLDER_OS                     // Operating system components
{
FOLDER_IO_SYSTEM              // IO system components
FOLDER_KERNEL                 // kernel components

FOLDER_ANSI                   // ANSI libraries
FOLDER_POSIX                   // POSIX components
FOLDER_UTILITIES              // Utility components
}
FOLDER_OBSOLETE               // Obsolete components
FOLDER_HARDWARE               // Hardware components
{
FOLDER_MEMORY                 // Memory components
```

```

{
FOLDER_MMU                // MMU options
}
FOLDER_BUSES              // Bus components
{
FOLDER_SCSI               // SCSI components
}
FOLDER_PERIPHERALS        // Peripheral components
{
FOLDER_HD                 // Hard Disk components
FOLDER_CLOCK              // System and Aux Clock component
FOLDER_SERIAL             // SIO driver support
FOLDER_FPP                // Floating Point libraries
}
FOLDER_BSP_CONFIG         // BSP configuration components
}
FOLDER_GRAPHICS           // Graphic components
{
FOLDER_UGL                // Optional product
FOLDER_HTML               // Optional product
}
}

```

E.3 工程初始化顺序

一个已命名了的初始化群组。它也可以申明它应该在哪个初始化群组的其他组件之前还是之后初始化。

工程编译的两个基本的初始化群组是 `usrInit` 和 `usrRoot`。每个初始化群组翻译成 C 函数，函数本身就是每个组件的初始化代码。

在以下的初始化顺序中，以大写字母出现的名称是组件的名称。以小写字母出现的名称是程序的名称。每个初始化群组有一条和初始化群组一样的名称的程序。在这个图表中缩进用来描绘标准组件的等级，这些等级被 Tornado 2.x 定义。当产生 `prjConfig.c` 文件时，每个被包含的组件被它的初始化代码片断所代替，那些初始化代码片断在 CDF 中被定义。不包含的组件不产生初始化群组的任何代码片断。



注意：工程工具建立过程并不应用到启动 ROMs 中。Tornado 2.x 的启动 ROMs 仅仅能够从传统的命令行建立。

E.3.1 romInit.s

仅仅为 ROMmed VxWorks 映像所使用，这个集合代码初始化了处理器和内存。这个代码通常以调用 romStart() 结束。

E.3.2 romStart.c

仅仅为 ROMmed VxWorks 映像所使用，这个代码重新部署了从 ROM 内部到 RAM 的正文和数据。它以调用位于 RAM_DST_ADRS 的程序来结束。这通常是 usrEntry() 的地址。

E.3.3 usrEntry.c

仅仅为 ROMmed VxWorks 映像所使用，这个程序完成一些特定的体系结构的初始化，并以调用 prjConfig.c 中的 usrInit() 结束。

E.3.4 sysALib.s

仅仅为一个已下载的映像所使用，执行时以 sysInit() 开始，代替了 romInit(), romStart() 和 usrEntry()。这个程序通过使用 romInit() 来重复所有的处理器和硬件初始化，除了主要的内存。这个程序以调用 prjConfig.c 中的 usrInit() 结束。

E.3.5 prjConfig.c



注意：不要假设以后的版本会保留 Tornado 2.x 中的任何初始化群组名称和特性，除了 usrInit 和 usrRoot。

```
usrInit
{
    // pre-multitasking init group
    INCLUDE_SYS_START
    INCLUDE_CACHE_SUPPORT
    INCLUDE_EXC_HANDLING
```

```

INCLUDE_SYSHW_INTT
INCLUDE_CACHE_ENABLE
INCLUDE_WINDVIEW_CLASS
INCLUDE_KERNEL                // DO NOT REMOVE.
}                             // end of usrInit

```

要求的 INCLUDE_KERNEL 必须是 usrInit() 最后被初始化的组件。它通过创建 root 任务和安排执行程序 usrRoot() 来进行多重任务处理。

```

usrRoot
{
    // post-multitasking init group
    usrKernelCoreInit
    {
        // kernel core features init group
        INCLUDE_SEM_BINARY
        INCLUDE_SEM_MUTEX
        INCLUDE_SEM_COUNTING
        INCLUDE_MSG_Q
        INCLUDE_WATCHDOGS
        INCLUDE_TASK_HOOKS
    }
    // end of usrKernelCoreInit
    INCLUDE_MEM_MGR_BASIC
    INCLUDE_MEM_MGR_FULL
    E
    INCLUDE_MMU_BASIC
    INCLUDE_MMU_FULL
    INCLUDE_SYCLK_INTT        // makes indirect call to sysHwInit2.
    usrIosCoreInit
    {
        // I/O system core init group
        INCLUDE_HW_FP
        INCLUDE_SW_FP
        INCLUDE_BOOT_LINE_INIT
        INCLUDE_TTY_DEV
        INCLUDE_TYCODRV_5_2
        INCLUDE_SIO
        INCLUDE_PC_CONSOLE
    }
    // end of usrIosCoreInit
    usrKernelExtraInit
    {
        // kernel extra features init group
        INCLUDE_HASH

```



```

INCLUDE_SYM_TBL
INCLUDE_ENV_VARS
INCLUDE_SIGNALS
INCLUDE_POSIX_AIO
INCLUDE_POSIX_AIO_SYSDRV
INCLUDE_POSIX_MQ
INCLUDE_POSIX_MEM
INCLUDE_POSIX_SIGNALS
INCLUDE_SM_OBI
INCLUDE_PROTECT_TEXT
INCLUDE_PROTECT_VEC_TABLE
usrIosExtraInit
{
    // I/O system extras init group
    INCLUDE_EXC_TASK
    INCLUDE_LOGGING
    INCLUDE_PIPES
    INCLUDE_STDIO
    INCLUDE_DOSFS
    INCLUDE_RAWFS
    INCLUDE_RT11FS
    INCLUDE_RAMDRV
    INCLUDE_SCSI
    INCLUDE_FD
    INCLUDE_IDE
    INCLUDE_ATA
    INCLUDE_LPT
    INCLUDE_PCMCIA
    INCLUDE_TFFS
    INCLUDE_FORMATTED_IO
    INCLUDE_FLOATING_POINT
}
// end of usrIosExtraInit
}
// end of usrKernelExtraInit
usrNetworkInit
{
    // network system init group
    INCLUDE_NET_SETUP
usrNetProtoInit
{
    // Network Protocol initializations

```

```
INCLUDE_BSD_SOCKET
INCLUDE_ZBUF SOCK
BSD43_COMPATIBLE
INCLUDE_ROUTE_SOCKET
INCLUDE_HOST_TBL
INCLUDE_IP
INCLUDE_IP_FILTER
INCLUDE_UDP
INCLUDE_UDP_SHOW
INCLUDE_TCP
INCLUDE_TCP_SHOW
INCLUDE_ICMP
INCLUDE_ICMP_SHOW
INCLUDE_IGMP
INCLUDE_IGMP_SHOW
INCLUDE_SM_NET_SHOW
INCLUDE_MCAST_ROUTING
INCLUDE_OSPF
INCLUDE_NET_LIB
INCLUDE_TCP_DEBUG
INCLUDE_ARP_API
INCLUDE_NET_SHOW
} // end of usrNetProtoInit
INCLUDE_MUX
INCLUDE_END
INCLUDE_PPP
INCLUDE_PPP_CRYPT
INCLUDE_SLIP
E
INCLUDE_NETWORK
usrNetworkBoot
{ // configure the network boot device, if specified
INCLUDE_NET_INIT
usrNetworkAddrInit
{ // get a network address
INCLUDE_DHCPC_LEASE_GET
INCLUDE_DHCPC_LEASE_CLEAN
```



```
    }                                // end of usrNetworkAddrInit
    INCLUDE_PPP_BOOT
    INCLUDE_SLIP_BOOT
    INCLUDE_NETMASK_GET
    INCLUDE_NETDEV_NAMEGET
    INCLUDE_SM_NET_ADDRGET
    usrNetworkDevStart
    {                                // attach and configure the network device(s)
    INCLUDE_SM_NET
    INCLUDE_END_BOOT
    INCLUDE_BSD_BOOT
    INCLUDE_LOOPBACK
    }                                // end of usrNetworkDevStart
    INCLUDE_SECOND_SMNET
    }                                // end of usrNetworkBoot
    usrNetworkAddrCheck
    {                                // verify IP address
    INCLUDE_DHCPC
    INCLUDE_DHCPC_LEASE_TEST
    }                                // end of usrNetworkAddrCheck
    usrNetRemoteInit
    {                                // network remote I/O access
    INCLUDE_NET_HOST_SETUP
    INCLUDE_NET_REM_IO
    INCLUDE_NFS
    INCLUDE_NFS_MOUNT_ALL

    }                                // end of usrNetRemoteInit
    usrNetAppInit
    {                                // start network applications
    INCLUDE_RPC
    INCLUDE_RLOGIN
    INCLUDE_TELNET
    INCLUDE_SECURITY
    INCLUDE_TFTP_SERVER
    INCLUDE_FTP_SERVER
    INCLUDE_FTPD_SECURITY
```

```
INCLUDE_NFS_SERVER
INCLUDE_DHCPC_SHOW
INCLUDE_DHCPR
INCLUDE_DHCPD
INCLUDE_SNTPC
INCLUDE_SNTPS
INCLUDE_PING
INCLUDE_RIP
INCLUDE_DNS_RESOLVER
INCLUDE_SNMPD
INCLUDE_MIB2_ALL
INCLUDE_STREAMS
INCLUDE_STREAMS_ALL
INCLUDE_STREAMS_AUTOPUSH
INCLUDE_STREAMS_DEBUG
INCLUDE_STREAMS_DLPI
INCLUDE_STREAMS_SOCKET
INCLUDE_STREAMS_STRACE
INCLUDE_STREAMS_STRERR
INCLUDE_STREAMS_TLI
} // end of usrNetAppInit
} // end of usrNetworkInit
INCLUDE_SELECT
usrToolsInit
{ // user tools init group
INCLUDE_SPY
INCLUDE_TIMEX
INCLUDE_MODULE_MANAGER
INCLUDE_LOADER
INCLUDE_NET_SYM_TBL
INCLUDE_STANDALONE_SYM_TBL
INCLUDE_STAT_SYM_TBL
INCLUDE_TRIGGERING
usrWdbInit
{ // WDB init group
INCLUDE_WDB
E
```

```

INCLUDE_WDB_MEM
INCLUDE_WDB_SYS
INCLUDE_WDB_TASK
INCLUDE_WDB_EVENTS
INCLUDE_WDB_EVENTPOINTS
INCLUDE_WDB_DIRECT_CALL
INCLUDE_WDB_CTXT
INCLUDE_WDB_REG
INCLUDE_WDB_GOPHER
INCLUDE_WDB_EXIT_NOTIFY
INCLUDE_WDB_EXC_NOTIFY
INCLUDE_WDB_FUNC_CAL
INCLUDE_WDB_VIO_LIB
INCLUDE_WDB_VIO
INCLUDE_WDB_BP
INCLUDE_WDB_TASK_BP
INCLUDE_WDB_START_NOTIFY
INCLUDE_WDB_USER_EVENT
INCLUDE_WDB_HW_FP
INCLUDE_WDB_TASK_HW_FP
INCLUDE_WDB_SYS_HW_FP
INCLUDE_WDB_BANNER
INCLUDE_SYM_TBL_SYNC
}                                     // end of usrWdbInit

usrShellInit
{                                     // target shell init group
INCLUDE_DEBUG                       // brkpts and stack tracer on target
INCLUDE_SHELL_BANNER               // display the Wind River banner on startup
INCLUDE_STARTUP_SCRIPT
INCLUDE_SHELL
}                                     // end of usrShellInit

usrWindviewInit
{                                     // WindView init group
INCLUDE_WINDVIEW
INCLUDE_SYS_TIMESTAMP
INCLUDE_USER_TIMESTAMP
INCLUDE_SEQ_TIMESTAMP

```

```
INCLUDE_RBUFF
INCLUDE_WV_BUFF_USER
INCLUDE_WDB_TSFS
INCLUDE_WVUPLOAD SOCK
INCLUDE_WVUPLOAD_TSFS SOCK
INCLUDE_WVUPLOAD_FILE
}                                     // end of usrWindviewInit
usrShowInit
{                                     // show routines init group
INCLUDE_TASK_SHOW
INCLUDE_CLASS_SHOW
INCLUDE_MEM_SHOW
INCLUDE_TASK_HOOKS_SHOW
INCLUDE_SEM_SHOW
INCLUDE_MSG_Q_SHOW
INCLUDE_WATCHDOGS_SHOW
INCLUDE_SYM_TBL_SHOW
INCLUDE_MMU_FULL_SHOW
INCLUDE_POSIX_MQ_SHOW
INCLUDE_POSIX_SEM_SHOW
INCLUDE_HW_FP_SHOW
INCLUDE_ATA_SHOW
INCLUDE_TRIGGER_SHOW
INCLUDE_RBUFF_SHOW
INCLUDE_STDIO_SHOW
}                                     // end of usrShowInit
|                                     // end of usrToolsInit
INCLUDE_CPLUS
INCLUDE_CPLUS_DEMANGLER
INCLUDE_HTTP
INCLUDE_USER_APPL                   // start user application
}                                     // end of usrRoot
```

附录 F 普通驱动程序

F.1 介 绍

本附录为适合于大多数 BSP 的一些设备提供书写普通驱动程序的指导。尽管不同的 BSP 在具体细节方面会有出入，对于绝大多数 BSP 还是有一些共同的需求的。例如，很多 BSP 需要一串行工具或计时器驱动程序。理想状态下，这些工具的程序应该比较普通，只需要做简单变动就可以兼容于一个新的 BSP。这种代码复用可以减少维护而使我们关注于检测结果，这样可以产生更好的测试代码。例如，使用普通驱动程序可以使 Wind River 在 3 个相似的 BSP 中节省 500 行 C 源码。^①

为了帮助读者在品种繁多的相似 BSP 中使用普通驱动程序，BSP 工具包在 `target/src/drv` 目录下提供了普通驱动程序源码。在编译阶段，`sysLib.c`（一个在任何 BSP 中均会复制的文件）可从 `target/src/drv` 中复制普通驱动程序。当需要升级 BSP 的驱动程序时，用户需要创建适合于多种 BSP 的普通驱动程序。可以把这些驱动程序添加到 `target/src/drv` 目录下。

然而，当处理非典型性硬件或遗传代码时，使用普通驱动程序往往是不合理的。一些硬件设计差别如此之大以至于不可能使用相同的驱动程序。遇到遗传代码的时候是有可能使用普通驱动程序的。然而，这种改写 BSP 以使用普通驱动程序的方法是不可取的，特别是当我们只是想升级 BSP 到一种最新版本 VxWorks 的时候。在以上两种情况下用户必须创建或维持一个 BSP 专用驱动程序。这个驱动程序在 `target/src/drv` 目录下是找不到的，而存在于 BSP 专用目录（在此目录下，BSP 的 `sysLib.c` 文件可在需要时包含驱动程序）。

F.2 串行驱动程序

普通串行驱动程序存在于 `target/src/drv/serial` 目录下。此目录还包括 `templateSerial.c` 文件及其相应的头文件 `templateSerial.h`。这些文件包含了典型串行驱动程序的设计及创建的十分详细的信息。当编写串行驱动程序时，先在模板上创建驱动程序，然后修改 BSP 的 `sysLib.c` 或 `sysSerial.c` 文件，以使它们包括所需的驱动程序。

^① 这里不包括 VxWorks 网络和 SCSI 工具驱动程序（以目标文件形式分布），关于它们的介绍会在此手册的其他附录中予以说明。



注意：串行驱动程序与 VxWorks 5.2 及以前版本后向兼容。所有 Tornado 时代的 BSP 应使用 SIO 驱动程序。

为了控制串行驱动程序信息，sysLib.c 使用了一个工具描述符。此工具描述符也可以压缩目标板特定信息。例如，尽管不是很详细，但它一般包含时钟频率和寄存器地址信息。在 sysLib.c 文件中，串行工具描述符公布于程序定义之外：

```
TY_CO_DEV tyCodv [NUM_TTY]; /* structure for serial ports */
```

运行时期，此队列在 sysHwInit() 中被初始化。TY_CO_DEV 结构在工具头文件中予以定义（例如，target/h/drv/serial/z8530.h）。下列 TY_CO_DEV 结构成员适用于所有串行驱动程序：

- TyDev。tyLib 所必须，是 VxWorks 的 tty 驱动程序支持的库。
- Created。一个必须在 sysHwInit() 文件中被初始化为 FALSE 的标记。
- NumChannels。用做参数检验。

在所有 BSP 中必须定义以下宏：

- NUM_TTY。定义所支持的串行通路数量。在 configAll.h 中默认为 2。要想覆盖默认值，首先去定义，然后在 config.h 中定义 NUM_TTY。若无串行通路，定义 NUM_TTY 为 NONE。
- CONSOLE_TTY。此宏定义控制台的通路数量。在 configAll.h 中默认值为 0。要想覆盖默认值，首先去定义，然后在 config.h 中定义 CONSOLE_TTY。

F.3 多模式串行驱动程序（SIO）

普通多模式串行驱动程序存在于 target/src/drv/sio 目录下。这些驱动程序被称为 SIO 驱动程序以区别于先前那些只含有一种单一中断模式操作的驱动程序。

SIO 驱动程序提供了一种硬件选项安装接口，比如断点比特、数据比特、奇偶等的数量。另外，这些驱动程序还提供了用于外部调试的无首通信接口（例如 ROM 监控类型调试）。目前只支持异步模式的 SIO 驱动程序。

每一个 SIO 工具都由一个 SIO_CHAN 结构来控制。这种结构包含一个单一的成员，即一个指向 SIO_DRV_FUNCS 结构的指针。这些结构在 target/h/sioLib.h 中定义如下：

```
typedef struct sio_chan /* a serial channel */
{
    SIO_DRV_FUNCS *pDrvFuncs;
    /* device data */
} SIO_CHAN;
```



```

typedef struct sio_drv_funcs SIO_DRV_FUNCS;

struct sio_drv_funcs /* driver functions */
{
    int (*ioctl)
        (
            SIO_CHAN *      pSioChan,
            int              cmd,
            void *           arg
        );

    int (*txStartup)
        (
            SIO_CHAN *      pSioChan
        );

    int (*callbackInstall)
        (
            SIO_CHAN *      pSioChan,
            int              callbackType,
            STATUS           (*callback)(),
            void *           callbackArg
        );

    int (*pollInput)
        (
            SIO_CHAN *      pSioChan,
            char *           inChar);

    int (*pollOutput)
        (
            SIO_CHAN *      pSioChan,
            char outChar
        );
};

```

SIO_DRV_FUNCS 结构程序成员如下:

■ **ioctl。**

指向驱动程序的标准 I/O 控制接口程序。此程序为任何驱动程序提供原始控制接口。为获得服务于一个标准 SIO 工具的 I/O 控制, 使用下列符号常数:

SIO_BAUD_SET	设置一个新的波特速率。
SIO_BAUD_GET	得到当前波特速率。
SIO_HW_OPTS_SET	新硬件设置。
SIO_HW_OPTS_GET	得到当前硬件设置。
SIO_MODE_SET	设置一个新的控制模式。
SIO_MODE_GET	得到当前控制模式。
SIO_AVAIL_MODES_GET	得到有效模式。
SIO_OPEN	打开一通道。
SIO_HUP	关闭一通道。

■ txStartup

当需要传送新数据时，指向系统响应程序的指针。一般情况下，此程序只在 `tryDrv.o` 从模块中调用。此模块提供了一个更高级的功能性，可以使原串行通道与行控制和规范字符处理同时运转。

■ callbackInstall

为驱动程序提供了指向回叫信号程序的指针，从而使驱动程序可以和字符取舍控制异步进行。驱动程序负责保存从 `callbackInstall()` 程序中得到的回叫信号程序和自变量。有效回叫信号为：`SIO_CALLBACK_GET_TX_CHAR` 和 `SIO_CALLBACK_PUT_RCV_CHAR`。

定义 `SIO_CALLBACK_GET_TX_CHAR` 指向一个接收输出新字符的程序。驱动程序使用补充自变量和一个表示新输出字符地址位置的附加自变量来调用此回叫信号程序。回馈程序返回 `OK` 表示字符成功传递，返回 `ERROR` 表示无有效字符。

定义 `SIO_CALLBACK_PUT_RCV_CHAR` 指向一个驱动程序可用来向上传送字符的程序。对于每一个引用字符，回叫信号用补充自变量调用，新字符作为第二自变量。驱动程序通常不关心这种调用的返回值。通常若更高一级不能接收驱动程序会留下一字符，除此之外不做任何响应。

■ pollInput 和 pollOutput

提供一种为驱动程序无首模式操作服务的接口。只在通过 `SIO_MODE_SET` 操作将设备置于无首模式操作的情况下才调用此程序。

关于更多的有关典型 SIO 设备驱动程序内部运行的信息参看 `target/src/drv/sio/templateSio.c`。

F.4 定 时 器

普通定时驱动程序存在于 `target/src/drv/timer` 目录下。此目录还包括 `templateTimer.c`。当书写一定时器驱动程序时，将驱动程序置于此模板上，然后修改 BSP 的 `sysLib.c` 文件以使之包括所需驱动程序。若一 BSP 只有一单一定时器，则此 BSP 必须支持系统时钟而非辅

助时钟。这就意味着 `sysAuxClkConnect()` 必须返回 `ERROR`。

以下各宏用于在 VxWorks 定时器驱动程序中的参数检验, 它们必须在 BSP 的 `bspname.h` 文件中予以定义:

- `SYS_CLK_RATE_MIN`。定义系统时钟运行的最小速率。除非硬件限制有其他规定, `SYS_CLK_RATE_MIN` 必须小于或等于 60 Hz。
- `SYS_CLK_RATE_MAX`。定义系统时钟运行的最大速率。除非硬件限制有其他规定, `SYS_CLK_RATE_MIN` 必须大于或等于 60 Hz。
- `AUX_CLK_RATE_MIN`。定义辅助时钟运行的最小速率。为了支持 `spy()`, `AUX_CLK_RATE_MIN` 必须小于等于 100 Hz。
- `AUX_CLK_RATE_MAX`。定义辅助时钟运行的最大速率。为了支持 `spy()`, `AUX_CLK_RATE_MIN` 必须大于等于 100 Hz。

F.5 非易失性存储器

普通 NVRAM 和 flash 存储器驱动程序存在于 `target/src/drv/mem` 目录下。此目录还包括 `templateNvRam.c`。此文件可使模板驱动程序作为非易失性存储器驱动程序使用, 包括闪速驱动程序。但是, 不要将此模板用于可选择真闪速文件系统 (TFFS) 产品。对于 TFFS, 使用闪速设备的特殊 MTD 驱动程序文件。

所有 BSP 均需要有一些非易失性存储器接口, 即使这种非易失性存储器不可利用。需要两个必须的程序 `sysNvRamGet()` 和 `sysNvRamSet()`, 它们都需要一个偏移参数。内部的偏移参数为:

```
offset += NV_BOOT_OFFSET; /* boot line begins at <offset> = 0 */
if ((offset < 0) || (strLen < 0) || ((offset + strLen) >
    NV_RAM_SIZE))
    return (ERROR);
```

因此, 偏移变量可以置偏, 而当设置为 0 时指向 VxWorks 导入行的起始比特。即使在导入行不在非易失性存储器区域的首部, 这也是成立的。

所有 BSP 均需定义以下宏:

- `NV_RAM_SIZE`。定义 NVRAM 的所有可用比特。在 `config.h` 或 `bspname.h` 中定义 `NV_RAM_SIZE`。对于没有 NVRAM 的目标板, 定义 `NV_RAM_SIZE` 为 `NONE`。
- `BOOT_LINE_SIZE`。定义为 VxWorks 导入行所预留的 NVRAM 的比特数量。默认值为 255, 定义在 `configAll.h` 中。

`BOOT_LINE_SIZE` 必须小于等于 `NV_RAM_SIZE`。若想覆盖 `BOOT_LINE_SIZE` 的默认值, 首先去定义, 然后在 `config.h` 或 `bspname.h` 中定义宏。

- **NV_BOOT_OFFSET**. 定义在 NVRAM 中 VxWorks 导入行首行的比特偏移量。默认值为 0, 定义在 configAll.h 中。与 BOOT_LINE_OFFSET 不同的是, 导入行的偏移保存在 RAM 中。

sysNvRamSet() 和 sysNvRamGet() 程序均有一偏移参数。若 NV_BOOT_OFFSET 大于 0, 可以通过指定一负偏移访问导入行前的比特。若想覆盖 NV_BOOT_OFFSET 的默认值, 首先去定义, 然后定义宏。

对于没有 NVRAM 的目标板, 为返回 ERROR 的程序的截断版本添加 mem/nullNvRam.c 文件。

F.6 VME 总线

通用 VME 总线驱动程序存在于 target/src/drv/vme 目录下。此目录还包括 templateVme.c 文件。使用此文件的驱动程序作为 VME 驱动程序的模板。此模板文件还包括有关 VME 驱动程序建构的详细信息。

对于没有 VME 总线接口的目标板, 使用 nullVme.c 驱动程序。若应用程序无须网络支持, 可跳过所有 VME 驱动程序。尽管在这种情况下, 用户需要提供一个经常会返回 FALSE 的 sysBusTas() 程序。

Wind River 公司目前还没有支持 VME-64 规格及其扩展规格的即插即用设备。

F.7 DMA

VxWorks 目前没有使用 DMA 驱动程序模型。DMA 支持是可选择的, 而这种支持的体系结构留到执行阶段予以考虑。

F.8 中断控制器

普通中断控制设备的驱动程序存在于 target/src/drv/intrCtl 目录下。有关中断控制器驱动程序设计和结构的详细介绍参看模板驱动程序 templateIntrCtl.c。

一些 CPU 的体系结构使用外部中断控制来获得和区分外部设备中断的先后次序。Wind River 公司准备了一个草拟文件用以定义标准中断控制设备接口。有关此文件上网查看 WindSurf, 或 Wind River 技术手册。

F.9 多 功 能

以前，驱动程序需要有一个 `target/arc/drv/multi` 目录配合 ASIC 芯片以实现多功能的合并。Wind River 不再使用此驱动程序模型。

ASIC 芯片和多功能目标板的驱动程序被分割成一些独立的实现各功能领域的驱动程序。一些这样的驱动程序取决于另一驱动程序的特征。若需要，它应在非独立驱动程序源码中备份。

依靠这种实现不同功能的单独驱动程序，用户可以测量那些他们没有申请使用的功能领域的支持程度。

F.10 PCI 总线

数据库 `PciConfigLib.c`、`pciConfigShow.c`、`pciIntLib.c` 和 `pciAutoCfg.c` 为 PCI 总线规格 2.1 提供支持。这些库支持基本和自动运行结构。关于如何使用这些数据库程序，请参见 Wind River 技术注释#49 和相关参考条目。

附录 G 升级 4.3 BSD 网络驱动程序

G.1 简介

这个附录描述了两种 4.3 BSD 网络驱动程序的升级程序。一个方法是把 4.3 BSD 网络驱动程序移植到 BSD 4.4 模型。另外一个方法是把 4.3 BSD 网络驱动程序升级到 END 驱动程序（在附录 H 中有所描述）。

把一个驱动程序移植到 4.4 BSD 模型仅仅需要对代码作最小的修改。事实上，移植一些驱动程序仅仅需要一天的工作。然而，移植一个不使用传输启动函数的旧版驱动程序需要花费更多的时间。

移植一个网络驱动程序到一个 END 驱动程序需要更多的工作。然而，如果驱动程序处理以下各项工作，则值得做这样的修改：

- 多址通信；
- 注册码以太网（为一个网络中 WDB 内核调试所必需的，一个有些命令比串行连接快得多的代码）；
- zero-copy 传输；
- 支持除了 IP 以外的网络协议。



注意：这一章假设你已经熟练掌握 BSD 网络驱动程序，应该也已经阅读过了附录 H。

G.1.1 4.3 BSD 的网络驱动程序的结构

当前装载在 VxWorks 中的网络驱动程序是基于那些在 BSD UNIX 版本 4.3 中可用的程序上。这些驱动程序仅仅定义了一个全球化（用户可随时使用的）的程序，驱动程序的 attach() 程序。一般而言，这个程序的名称包含单词 attach，以驱动程序名称中的两个字母作为前缀。例如，AMD Lance 驱动程序的 attach 程序被叫做 lnattach()。Xxattach() 程序在分配到 ifnet 结构中的 5 个功能指针中分支。在表 G-1 中陈列的这些功能都是在 IP 协议栈中从不同的位置调用的，IP 协议栈对驱动程序有很好的理解。

表 G-1 网络接口步骤管理

函 数	函数指针	特定驱动程序
初始化	if_init	xxInit()
输出	if_output	xxOutput()
控制	if_ioctl	xxioctl()
复位	if_reset	xxReset()
看门狗	if_watchdog (optional)	xxWatchdog()

当调用驱动程序的中断时，开始接受数据包了。这个中断仅仅执行把数据包从本地硬件上拿下来的最少的必要工作，通过调用 `netJobAdd()` 来确定一个输入处理器的运行时间，然后返回。TNetTask 调用增加到它的工作序列中的函数。在接受数据包的时候，这个驱动程序是 `xxReceive()` 函数。

`XxReceive()` 函数最后通过调用 `do_protocol_with_type()` 来把数据包上传到协议。这个程序是一个 switch 说明，它指出了哪个协议来控制下载数据包。在图 G-1 中有调用序列。

图 G-2 显示了数据包传输的调用图表。在一个协议已经选择了传送一个数据包的接口时，它调用那个接口的 `xxOutput()` 程序。输出程序调用 `ether_output()` 函数，把指针及数据传送到地址信息（通常是一个 `arpcom` 结构）。在数据已经被合理打包后，它在输出序列（使用 `IF_ENQUEUE` 宏）中被陈列，驱动程序的启动程序被调用。`XxTxStartup()` 程序列出大量的数据包，并通过物理媒介来传输它们。

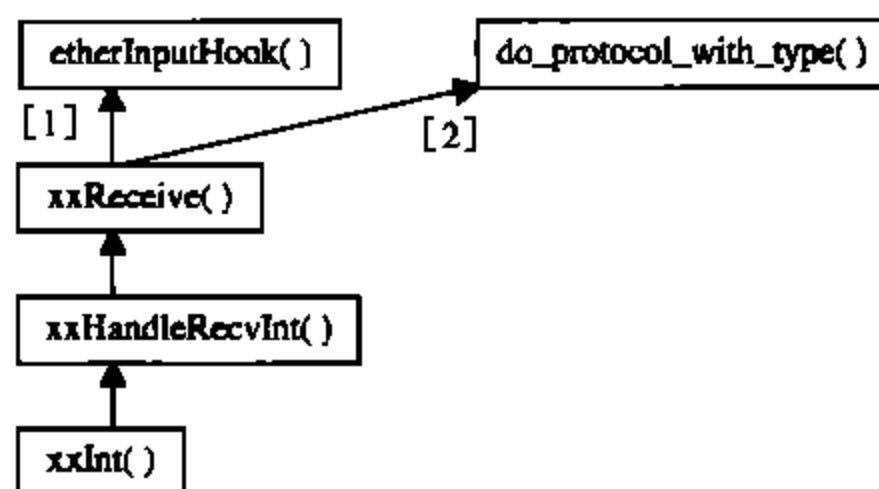


图 G-1 数据包接收调用图表

图 G-1 中[1] `xxReceive()` 首先显示了传送到 `etherInputHook()` 的数据包。

图 G-1 中[2] 如果 `etherInputHook()` 不传送这个数据包，`xxReceive()` 把数据包传送到 `do_protocol_with_type()`。

图 G-2 中[1] `xxTxStartup()` 首先显示了传送到 `etherOutputHook()` 的数据包。

图 G-2 中[2] 如果 `etherOutputHook()` 不传送这个数据包，那么 `xxTxStartup()` 把数据包

传送到媒介。

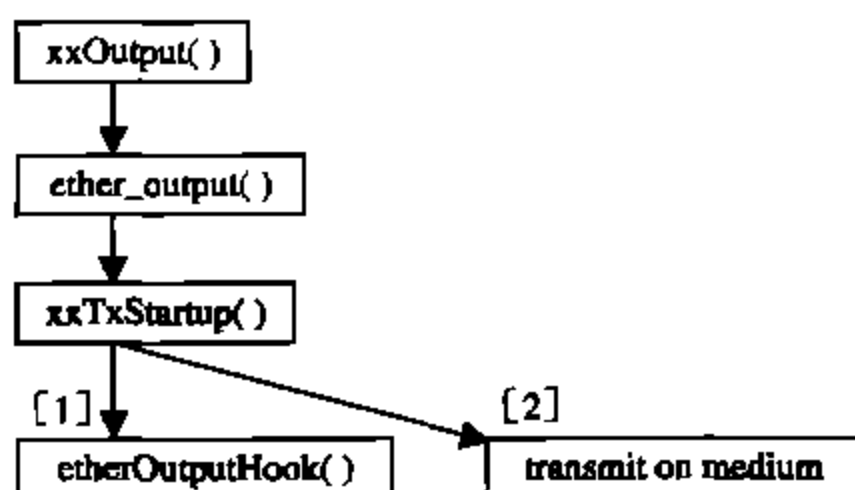


图 G-2 数据包传递调用图表

G.1.2 提供访问原始数据包的 Etherhook 程序

可以使用 `etherInputHook()` 和 `etherOutputHook()` 程序来通过 TCP/IP 栈，因此可以访问原始数据包。一旦接收到数据包，如果安装了 `etherInputHook()` 函数，它在驱动程序完成接受之后在数据包进入到协议之前接受数据包。如果 `etherInputHook()` 决定不让别人看到这个数据包，`etherInputHook()` 返回一个非零的值，驱动程序考虑传送数据包。如果 `etherInputHook()` 返回零，则驱动程序把数据包传送到 TCP/IP 栈。

一旦发送了数据包，一个已被安装了的 `etherOutputHook()` 在发送以前接受一个数据包。如果 `etherOutputHook()` 决定阻止传送数据包，`etherOutputHook()` 返回一个非零值，驱动程序考虑接受数据包。如果 `etherOutputHook()` 返回零，则驱动程序发送数据包。

每个驱动程序仅有可能安装一个 `etherInputHook()` 和一个 `etherOutputHook()`。这限制了可选择的协议的数量为一个，除非这些 `ether*Hook()` 程序作为一个多路通道为更多的协议工作。

更多的关于 etherhooks 的信息请参看《Tornado 用户指南》1.0.1: G.4。



注意：以后版本的 VxWorks 不支持 etherhooks。

G.2 升级到 4.4 BSD

为了把一个驱动程序从 4.3 BSD 升级到 4.4 BSD，必须改变驱动程序使用 `ether_attach()` 的方法。这个程序是从驱动程序的 `xxattach()` 程序中调用的，它负责把驱动程序的入口点放在 TCP/IP 协议追踪驱动程序的 `ifnet` 结构中，这些入口点陈列在表 G-1 中。考虑调用以下

的程序:

```
ether_attach( ):
ether_attach(
(IFNET *) & pDrvCtrl->idr,
unit,
"xx",
(FUNCPTR) NULL,
(FUNCPTR) xxIoctl,
(FUNCPTR) xxOutput,
(FUNCPTR) xxReset
);
```

这个程序期待一个接口数据纪录 (IDR)，一个单位数字，一个引用的字符串，这个字符串是程序的名称，在以上的例子中是“xx”。以下的四个指针是相关的驱动程序的功能指针。

第一个功能指针参考驱动程序的 `init()` 程序，驱动程序不需要或者根本没有这个程序。第二个功能指针参考驱动程序的 `ioctl()` 接口，这个接口允许更高层来控制驱动程序状态。第三个功能指针参考输出在物理媒介中的数据包的程序。如果 TCP/IP 栈决定它需要重新设置驱动程序的话，最后一个功能指针参考这个程序。

在 4.4 BSD 中，有一个输出程序，它调用所有的 Ethernet 驱动程序能够使用的 `ether_output()`。因此，为了把以上的 `ether_attach()` 调用转换成一个 4.4 类型的调用，应该调用以下的 `ether_attach()`：

```
ether_attach(
(IFNET *) & pDrvCtrl->idr,
unit,
"xx",
(FUNCPTR) NULL,
(FUNCPTR) xxIoctl,
(FUNCPTR) ether_output, /* generic ether_output */
(FUNCPTR) xxReset
);
pDrvCtrl->idr.ac_if.if_start = (FUNCPTR)xxTxStartup;
```

这个命令中，有一个额外的代码行紧跟对 `ether_attach()` 的调用之后。这个代码行增加了一个到接口数据纪录的发送启动程序。在 `ether_output()` 程序被调用之后 TCP/IP 栈调用了发送启动程序。这个额外的代码行假设驱动程序已经有一个发送启动程序。如果一个驱动程序缺乏一个单独的发送启动程序，必须写一个。请在 G.2.4 中参看模块。

G.2.1 去除 xxOutput() 程序

如果一个 4.3 BSD 驱动程序有一个 xxOutput() 程序，它应该类似于如下所示的程序：

```
static int xxOutput
(
    IDR * pIDR,
    MBUF * pMbuf,
    SOCK * pDestAddr
)
{
    return (ether_output ((IFNET *)pIDR, pMbuf, pDestAddr,
        (FUNCPTR) xxTxStartup, pIDR));
}
```

这个程序在内部调用 ether_output() 程序，这个程序期望一个指针启动程序。然而，在 4.4 BSD 模块，所有的工作被控制在 TCP/IP 栈。因此，在一个 4.4 BSD 驱动程序中，这个代码是不必要的，应该被去除。

G.2.2 改变发送启动程序

在 4.3 BSD 中，一个发送启动程序的功能原型如下所示：

```
static void xxTxStartup (int unit);
```

在 4.4 BSD 下，原型改变为：

```
static void xxTxStartup (struct ifnet * pDrvCtrl);
```

4.4 BSD 版本期望到一个驱动程序的指针来控制结构。这个改变减轻了启动程序的负担。不是找到自己的驱动程序来控制结构，而是接受一个驱动程序的指针作为输入来控制结构。

如果一个驱动程序使用 netJobAdd() 来为任务级别的执行确定发送启动程序的时间，在一个 DRV_CTRL 结构指针而不是一个单元中编辑 netJobAdd() 调用。

G.2.3 在接受数据包时的改变

在 4.3 BSD 下，驱动程序调用 do_protocol_with_type()。例如：


```
do_protocol_with_type (etherType, pMbuf, &pDrvCtrl->idr, len);
```

这个调用期望一个 `etherType`，一个到包含数据包的 `mbuf` 的指针，一个接口数据纪录和一个数据的长度。

在 4.4 BSD 下，把以上的调用替换成对 `do_protocol()` 的调用。例如：

```
do_protocol (pEh, pMbuf, &pDrvCtrl->idr, len);
```

第一个参数期望一个到每个数据包的开头的指针（包括连接头文件）。其他参数保持原状。这个驱动程序不再需要指明协议的 `etherType`。

G.2.4 创建一个发送启动程序

一些 4.3 BSD 驱动程序没有一个发送启动程序。对于每个驱动程序，必须创建一个，模块如下所示：

```
void templateStartup
(
    DRV_CTRL *pDrvCtrl
)
{
    MBUF * pMbuf;
    int length;
    TFD * pTfd;
    /*
     * Loop until there are no more packets ready to send or we
     * have insufficient resources left to send another one.
     */
    while (pDrvCtrl->idr.ac_if.if_snd.ifq_head)
    {
        /* Dequeue a packet from the send queue. */
        IF_DEQUEUE (&pDrvCtrl->idr.ac_if.if_snd, pMbuf);
        /*
         * Device specific code to get transmit resources, such as a
         * transmit descriptor, goes here.
         */
        /*
         * If (Insufficient Resources)
         */
        m_freem (pMbuf); /* Make sure to free the packet. */
        return;
    }
    /*
```

```

* pData below is really the place in your descriptor,
* transmit descriptor, or equivalent, where the data is
* to be placed.
*/
copy_from_mbufs (pData, pMbuf, length);
if ((etherOutputHookRtn != NULL) &&
    (* etherOutputHookRtn)
    (&pDrvCtrl->idr, (ETH_HDR *)pTfd->enetHdr, length))
    continue;
/*
* Do hardware manipulation to set appropriate bits
* and other stuff to get the packet to actually go.
*/
/*
* Update the counter that determines the number of
* packets that have been output.
*/
pDrvCtrl->idr.ac_if.if_opackets++;
) /* End of while loop. */
) /* End of transmit routine. */

```

G.3 转换一个网络接口驱动程序到最终模块

基于 MUX 的网络驱动程序的模块包含标准化的入口点, 这些入口点不在 BSD 模块中。表 G-2 显示了在 BSD 4.3 驱动程序中的类似功能和 ENDS 所必需的那些功能。应该能够重复使用来自 BSD 4.3 网络驱动程序的大部分代码。

表 G-2 所需的驱动程序入口点及其来源

END 入口点	BSD 4.3 模式入口点
endLoad()	xxattach()
endUnload()	N/A
N/A	xxReceive()
endSend()	xxOutput()
endIoctl()	xxIoctl()
endMCastAddrAdd()	N/A
endMCastAddrDel()	N/A
endMCastAddrGet()	N/A
endPollSend()	N/A
endPollReceive()	N/A

 **警告：**当把一个 BSD 4.3 网络驱动程序代码转换成一个 END，必须把进入协议的所有调用替换成进入到 MUX 的合适调用。同样地，必须去除执行或使用 `etherInputHook()` 和 `etherOutputHook()` 程序的所有代码。

1. 重写 `xxattach()` 来使用一个 `endLoad()` 接口

重写 `xxattach()` 接口来与 `endLoad()` 入口点相配。

控制硬件初始化细节的大部分代码应该是一样的。然而，当分派上传到协议的接受包的缓冲的内存时，应该使用 MUX 缓冲管理。请参看 H.3.1，以及 `muxBufInit()` 的参考入口。

去除支持 `etherInputHook()` 和 `etherOutputHook()` 程序执行的 `xxattach()` 的任何代码。

2. `xxReceive()` 依然控制任务级别的接收数据包

因为 MUX 不直接调用驱动程序的接收数据包代码，所以没有任何 `endReceive()` 入口点。然而，你的驱动程序依然需要控制任务级别的接受包。驱动程序的大部分代码需要大量的修改。不是直接调用协议，而是这个程序使用一个 MUX-supplied 函数来把数据包传送到协议（参看 H.3）。同样地，接收程序应该使用 MUX-managed 的内存池作为它的接受缓冲区域。

3. 重写 `xxOutput()` 来使用一个 `endSend()` 接口

重写你的 `xxOutput()` 接口来与 `endSend()` 程序相配。

如果需要修改的话，直接处理硬件中的数据包的大部分代码应该需要很小的改变。然而，应该改变你的代码来使用被分配在 `endBufLib-managed` 内存池的 `mblk` 链接。更多的信息请参看 `netBufLib` 的参考入口。

4. `xxioctl()` 是 `endioctl()` 的基础

重写你的 `xxioctl()` 接口来与 `endioctl()` 程序匹配。

如果你的驱动程序以前使用 `xxioctl()` 来执行多址通信，必须把这些操作分成独立的 `endMCastAddAdd()`，`endMCastAddDel()` 和 `endMCastAddrGet()` 程序。

附录 H 执行基于多用户 (MUX) 的网络接口驱动程序

H.1 简 介

一个专为使用网络堆栈而编写的网络接口驱动程序被称为增强的网络驱动程序 (END)。本章介绍如何使用 END 及书写使用 END 的协议方法。

本附录适用于熟悉网络原理 (包括协议层) 的软件开发人员使用。熟悉 4.4 BSD 网络内部结构对于理解本章也是有帮助的。本章决非书写网络接口驱动程序的指南。用户应把本附录作为运行 VxWorks 的网络接口驱动程序细节的指南。

若这是读者第一次编写网络接口驱动程序, 需考虑使用一现有驱动程序, 修改之以满足用户需求。有关 TCP/IP 的详细介绍, Wind River 公司推荐阅览以下 Addison Wesley 期刊:

- TCP/IP Illustrated, 第 1 卷, 协议, Richard Steven 著。
- TCP/IP Illustrated, 第 2 卷, 执行, Gary R. Wright 和 W. Richard Steven 同著。



注意: 若用户使用的是 Ipv6, 应同时参阅 WindNet Ipv6 程序员指南 1.0 或者 VxWorks 网络程序员指南的后续版本 (即继 WindNet Ipv6 1.0 后发行的版本)。

src/drv/end 目录包含了 templateEnd.c 和 In7990End.c, 为配合 Lance 芯片的使用提供了一个 END 驱动程序样本。

1. 多用户接口为一个连接数据链接和协议层的接口

网络堆栈的一个特征就是在数据链接和网络协议层之间使用——API。此 API 为一访问网络硬件以获得多种网络协议的多元接口。这种接口被称为 MUX。图 H-1 展示了协议与数据链接层的关系。

在协议层方面, 尽管用户可以载入和使用其他协议, 但 VxWorks 通常使用的是网络协议。在数据链接层方面, 尽管 VxWorks 也支持其他物理媒质的数据传送, 但它通常使用以太网。例如, 对于长距离传送, VxWorks 支持使用串行线路。在连接更紧密的环境中, VxWorks 的互联网协议 (IP) 也可在通用目标板上使用共享内存来作为物理媒介。然而, 无论什么媒介, 网络接口驱动程序都使用 MUX 与协议层通信。

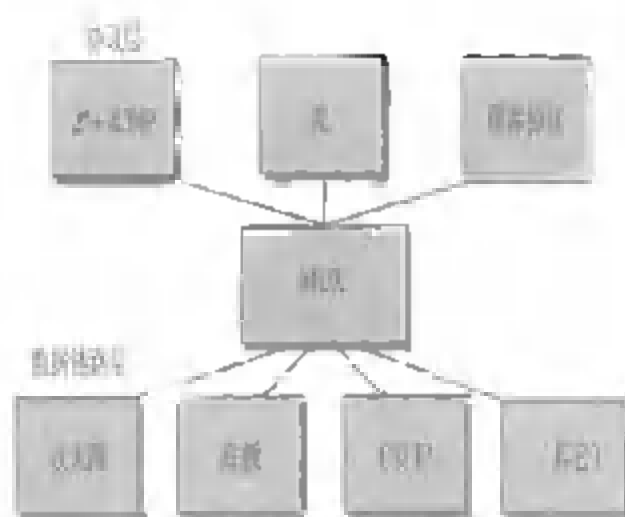


图 H-1 MUX 是数据链路层和协议层之间的接口



注意：数据链接层是一种抽象。网络接口驱动程序是执行在这个抽象中描述的功能代码。同样地，协议层也是一种抽象。执行协议层功能的代码被称为协议接口驱动程序。然而，这本书中将此代码简称为“协议”。

2. MUX 的分解网络接口驱动程序和协议

在 BSP4.3 模型下，VxWorks 网络驱动程序和协议紧密结合着。它们彼此紧密依赖于对方的数据结构。在基于 MUX 的模式下，网络驱动程序与协议之间没有内部交换数据。它们只通过 MUX 间接相互作用。例如，接收到数据包后，网络接口驱动程序并不直接访问协议中的任何结构。当驱动程序准备给协议传递数据时，驱动程序会调用一 MUX 辅助程序。然后此程序用于处理给协议传递数据时的具体问题。

MUX 的目的是分解协议和网络驱动程序，从而使它们几乎独立于彼此。这种独立使添加新的驱动程序和协议变得简单。例如，若想添加新的 END，所有现有基于 MUX 的协议均可使用新的驱动程序。同样地，若想添加新的基于 MUX 的协议，任何现有的 END 均可 MUX 来访问新的协议。

3. MUX、协议和驱动程序 API 总览

图 H-2 显示了协议、MUX 和一个网络接口驱动程序。协议执行以下入口点：

- `stackShutdownRtn()`;
- `stackError()`;
- `stackRcvRtn()`;
- `stackTxRestartRtn()`。

当 MUX 需要访问协议时调用这些入口点。为了加载协议以使用 MUX，用户必须至少执行以上所有入口点。

MUX 会执行入口点 `muxBind()`、`muxUnbind()`、`muxDevLoad()` 等。MUX 使用这些入口点与网络接口驱动程序相互作用。编写或加载网络接口驱动程序以使用 MUX 时，用

户必须执行表 H-1 中所有入口点。

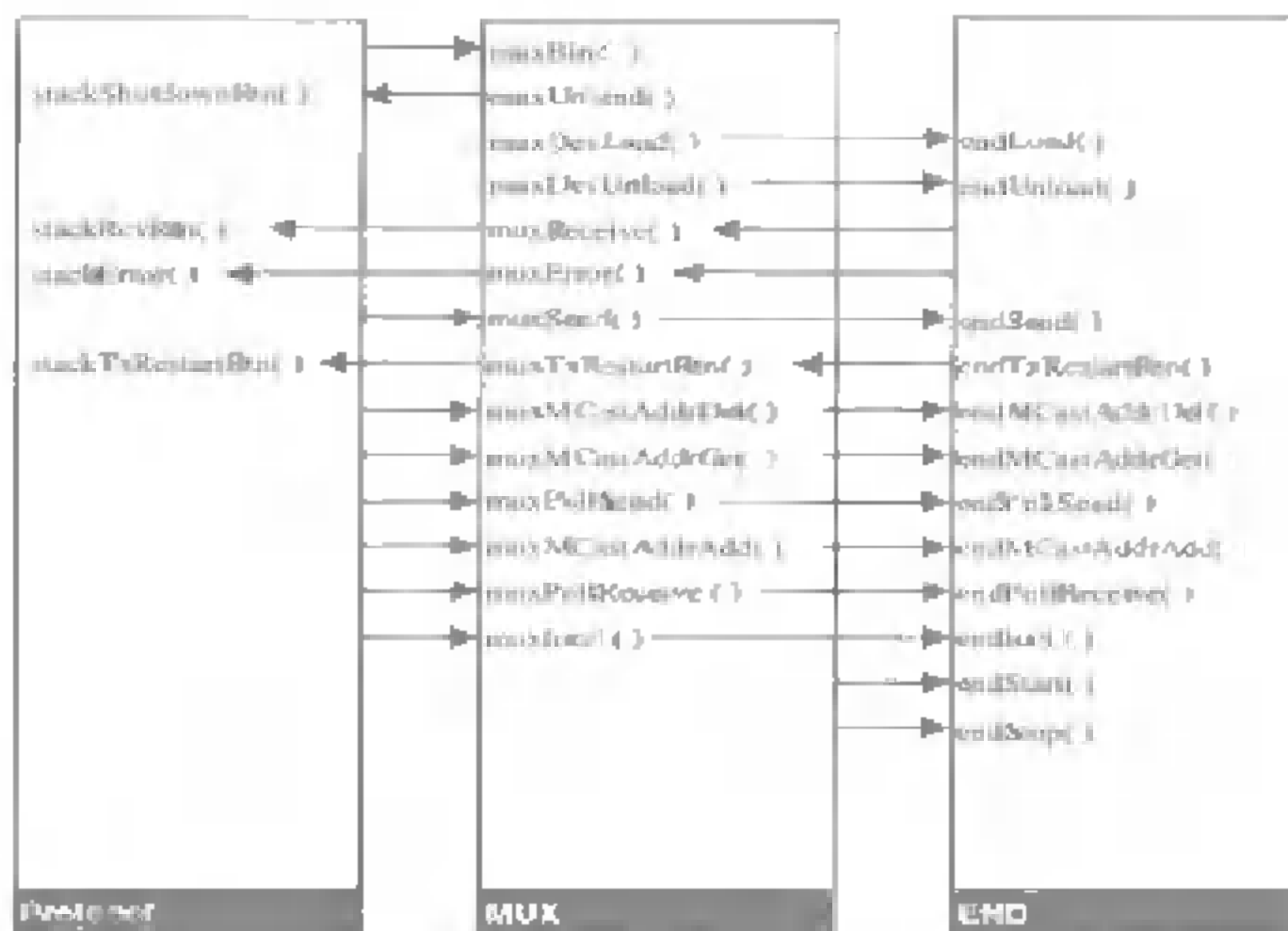


图 H-2 MUX 接口

箭头表示每个调用程序的入口点。例如，最上面的箭头表示协议调用一个在 MUX 中执行的程序 `muxBind()`。若基于 MUX 的 API 指定调用程序首尾，图 H-2 在每一箭头末尾标注了程序名。例如，`muxSend()` 调用 `endSend()`。注意到，尽管协议通过调用 `muxSend()` 开始传送，图 H-2 却并没有指出调用它的协议程序名。此程序是在标准 API 的外部。

H.2 VxWorks 如何装载并使用驱动程序

我们首先来关注 MUX 设备和标准 END 入口点。无论什么情况，当需要设计或调试用户驱动程序的入口点时，需要知道入口点执行的内容。因此，需要知道以下内容：

- 实际装载和启动用户驱动程序的调用任务。
- 驱动程序的一般寄存中断管理任务。
- 使用驱动程序来处理数据包的任務。

H.2.1 装载驱动程序

当启动系统时, VxWorks 执行任务 tUsrRoot 来完成以下各项:

- 网络任务的工作队列初始化。
- 产生 tNetTask 来处理网络任务的工作队列中的条目。
- 调用 muxDevLoad() 加载网络驱动程序。
- 调用 muxDevStart() 启动驱动程序。

1. 加载用户驱动程序到 MUX 中

使用 tUsrRoot 调用 muxDevLoad() 加载用户网络驱动程序。作为调用的输入, tUsrRoot 规定了用户驱动程序 endLoad() 的入口点。从内部来看, muxDevLoad() 调用规定了 endLoad() 的入口点。这个 endLoad() 入口点类似于 xxattach() 入口点, 是 4.3BSP 驱动程序的惟一公共入口点。

endLoad() 程序用于处理任何特殊设备的初始化和返回一个 END_OBJ 结构。用户的 endLoad() 必须包含大多数这样的结构 (参见网络设备摘要: END_OBJ)。这包括一个指向 NET_FUNCS 结构的指针, 该结构与用于处理接收和发送等问题的用户驱动程序接口的功能指针同时组装。

在从 endLoad() 返回控制值到 muxDevLoad() 之后, MUX 完成了 END_OBJ 结构 (通过给程序赋予一指针, 用户的驱动程序可以传递数据包到 MUX 中)。MUX 然后把返回的 END_OBJ 添加到 END_OBJ 结构的连接清单中。此清单列出了当前系统所运行的所有网络设备状态。在 muxDevLoad() 返回控制信息后, 用户的驱动程序装载完毕以备用。

2. 用户驱动程序中断程序的寄存

需调用 sysIntConnect() 来寄存用户驱动程序中断处理器。此调用的典型地址为用户驱动程序 endStart() 的入口点。

当 muxDevLoad() 加载用户驱动程序时, 调用 muxDevStart() 来访问 endStart() 入口点。

H.2.2 网络任务队列的中断服务程序 (ISR) 移动

当网络设备产生中断时, VxWorks 调用用户驱动程序先前寄存的中断服务程序。中断服务程序应做尽可能少的工作以完成将信息包从本地硬件送出。为了最小化中断阻塞时间, 用户中断服务程序应只处理那些只需要最小执行时间的任务, 如差错和状态变换。用户中断服务程序应包含所有任务级的耗时处理工作。

为了排列任务级别的数据包接收处理工作, 用户中断服务程序必须调用 netJobAdd()。此程序作为输入, 接收一个程序指针和最多 5 个附加自变量 (由程序指针引用的程序参量)。

```

STATUS netJobAdd
(
    {FUNCPTR}    routine, /* routine to add to netTask work queue */
    int          param1,  /* first arg to added routine */
    int          param2,  /* second arg to added routine */
    int          param3,  /* third arg to added routine */
    int          param4,  /* fourth arg to added routine */
    int          param5,  /* fifth arg to added routine */

```

在用户调用 `netJobAdd()` 时, 应把驱动程序用来处理数据包的入口点规定为任务级别。然后 `netJobAdd()` 程序把程序调用 (和自变量) 放入 `tNetTask` 的工作队列。VxWorks 使用 `tNetTask` 来处理任务级别的网络处理。



注意: 用户可以使用 `netJobAdd()` 来排列任务而不处理接收到的数据包。



警告: 谨慎使用 `netJobAdd()`。 `netJobAdd()` 为一同时被网络堆栈所使用的有限资源。若其溢出将会造成通用网络堆栈的瘫痪。

H.2.3 执行等待在网络任务队列中的调用任务

`TNetTask` 任务在引入工作队列时闲置。当响应一个引入数据包时, 用户中断服务程序调用 `netJobAdd()`。作为 `netJobAdd()` 的参量, 用户中断服务程序让驱动程序接口处理任务级的数据包接收工作。`netJobAdd()` 的调用添加此入口点到 `tNetTask` 工作队列中。`netJobAdd()` 调用同时自动产生适当的信号量来激活 `tNetTask`。

`TNetTask` 被激活后, 使各个程序调用任务及相关自变量从它的工作队列中出列。然后在它的文件中执行这些程序。只要队列中有一个任务, `TnetTask` 便会运行。当队列为空, 所有数据包均被成功地传送到 MUX 中时, `TnetTask` 回到队列继续闲置。

H.2.4 在 VxWorks 中添加用户网络接口驱动程序

在 VxWorks 系统中安装驱动程序和安装其他应用程序很相似。第一步是在 VxWorks 映像中汇编和包含驱动程序代码。有关一般步骤的描述请参见《Tornado 用户指南》。此书介绍了如何汇编源程序代码来产生相适合的目标代码。

要在 VxWorks 映像中包含大多数软件模块一般只需要写一些 `#define` 语句。而添加网络接口驱动程序就不需要了。然而, 因为 VxWorks 允许用户创建多个网络工具, 用户必须建立一张表格以使所有 `#define` 语句归入特定工具组。此表在 `target/src/config/BSP/configNet.h`

目录下有所定义，在这里 BSP 为板机支持包的缩写，如 mv162 等。

例如，如果用户想要 VxWorks 创建两个网络工具，一个支持缓冲器借出，另一个不支持，我们需要首先在 configNet.h 中写入下列语句：

```
/* Parameters for loading the driver supporting buffer loaning. */
#define LOAD_FUNC_0 In7990EndLoad
#define LOAD_STRING_0 *0xfffffe0:0xffffffe2:0:1:1*
#define BSP_0 NULL

/*Parameters for loading the driver NOT supporting buffer loaning. */
#define LOAD_FUNC_1 LOAD_FUNC_0
#define LOAD_STRING_1 *0xfffffee0:0xfffffee2:4:1:1*
#define BSP_1 NULL
```

当给这些常量赋值的时候，应考虑以下几点：

- **END_LOAD_FUNC**。指定用户驱动程序 endLoad() 的入口点。例如，若用户驱动程序 endLoad() 的入口点为 In7990EndLoad()，需要在 config.h 中包含下列语句：

```
#define END_LOAD_FUNC In7990EndLoad
```

- **END_LOAD_STRING**。指定初始串作为 initString 参数传入 muxDevLoad()。

用户在定义 endTbl（在 configNet.h 中包含的表格，定义了映像中的 ENDS）的时候需要包括以下语句：

```
END_TBL_ENTRY endTbl
{
    { 0, LOAD_FUNC_0, LOAD_STRING_0, BSP_0, FALSE},
    { 1, LOAD_FUNC_1, LOAD_STRING_1, BSP_1, FALSE},
    { 0, END_TBL_END, 0, NULL},
};
```

每行开始的数字表示设备的单元数。如第 1 行规定单元数为 0。因此，它装载的设备为 deviceName()。每行末尾的 FALSE 表示此行没有被处理。当系统成功装载了一个驱动器后，在此表运行时，此值将变为 TRUE。若想避免系统自动装载用户驱动程序，只需把此值设置为 TRUE。

最后，用户必须编写 BSP 中的 config.h 文件来定义 INCLUDE_END。^① 这会使创建过程包括 END/MUX 接口。此时，我们已做好了重建 VxWorks 以添加新的驱动程序的工作。当用户重建映像时，需调用每个在命令栏表格中列出设备的 muxDevLoad() 文件。^②

^① 对于一个 END 设备的 config.h 文件，BSP 默认去定义 INCLUDE_END。

^② 关于 muxDevLoad() 参数的介绍请参见相关参考文献。



注意: 若用户不重建导入 ROM, 则它们不包括新的 END。这意味着必须从导入 ROM 中已存在的非 END 驱动程序中导入。为此, 用户需在 BSP 的 config.h 中定义 END_OVERRIDE。系统被导入后就可以使用 VxWorks 映像中新的 END 设备。

H.3 处理驱动程序包接收问题的指南

END 设备目录入口点 (见表 H-1) 不包括 endReceive() 入口。这是因为 END 不需要此入口。当然, 用户驱动程序必须包括处理包接收问题的代码, 但 MUX 不会直接调用此代码。因此, 包接收代码的细节问题留给用户自行处理。

然而, 尽管 MUX API 不需要 endReceive() 入口, 当设计驱动程序包接收代码时, 用户也需要考虑到 VxWorks 系统。例如, 用户网络接口驱动程序必须包括中断服务程序的入口。另外, 用户也需要设定不同的入口来完成不同任务级别的包接收。

从内部来看, 任务级包接收入口负责把数据包传给 MUX, 比如保证数据的一致性。同样地, 在它接收数据 (3) 做任何处理之前, 此入口需使用一种非直接方式来检查和避免竞争。当一切就绪后, 用户驱动程序把数据包传给 MUX。为此, 调用在 END_OBJ 结构中 receiveRtn 成员所引用的函数 (见网络工具摘要: END_OBJ)。

尽管用户驱动程序的 endLoad() 入口分配了 END_OBJ 结构而且设定了大多数其成员值, 但它却没有设定 receiveRtn 的成员值。在调用 muxDevLoad() 装载用户驱动程序后, MUX 设定 receiveRtn 成员值。然而, 在驱动程序被激活和 muxDevLoad() 装载完毕之间有一短的时间间隔。在此间隔内, receiveRtn 也许还未被赋值。因此, 在实行上述程序之前检查 receiveRtn 均应为 NULL。

从装载的驱动程序中去除 Etherhook 代码

VxWorks 网络接口驱动程序的 BSD4.3 模型支持一组输入输出钩子, 用来截取在驱动器和协议之间传输的数据包。这些输入输出钩子被称作 etherhook。当为一个 END 模型装载驱动程序时, 需要去掉执行 etherhook 的代码 (如果有的话)。END 是不需要 etherhook 的。此功能包含在 MUX 中。详情请见《Tornado 用户指南》的有关 etherhook 的讨论。

H.3.1 设置并使用用于接受和传送缓冲的内存池

和网络堆栈共存的是 netBufLib, 它为一个用来安装和管理网络应用 (诸如 END 和网络协议) 的内存池的库。为了支持缓冲器 loaning 和其他特征, netBufLib 程序以 mBlks、clBlks 和机群的形式处理数据。结果是, 一个用 netPoolInit() 建立起来的内存池安装在 mBlk

结构池，cBlk 结构池和一个或多个机群池周围。

netBufLib 程序使用 mBlk 和 cBlk 结构来跟踪有用信息以管理机群中数据。机群包括用 mBlk 和 cBlk 结构描述的数据。用户主要通过 mBlk 结构来访问和传递在由 netPoolInit() 建立的内存池中存放的数据。由于 mBlk 仅仅涉及数据，这使得网络层不需要复制数据便可交换数据。另一个 mBlk 的特征为它的连锁性，这使得用户可以通过在 mBlk 链首传递 mBlk 而任意传送大批数据。见图 H-3。

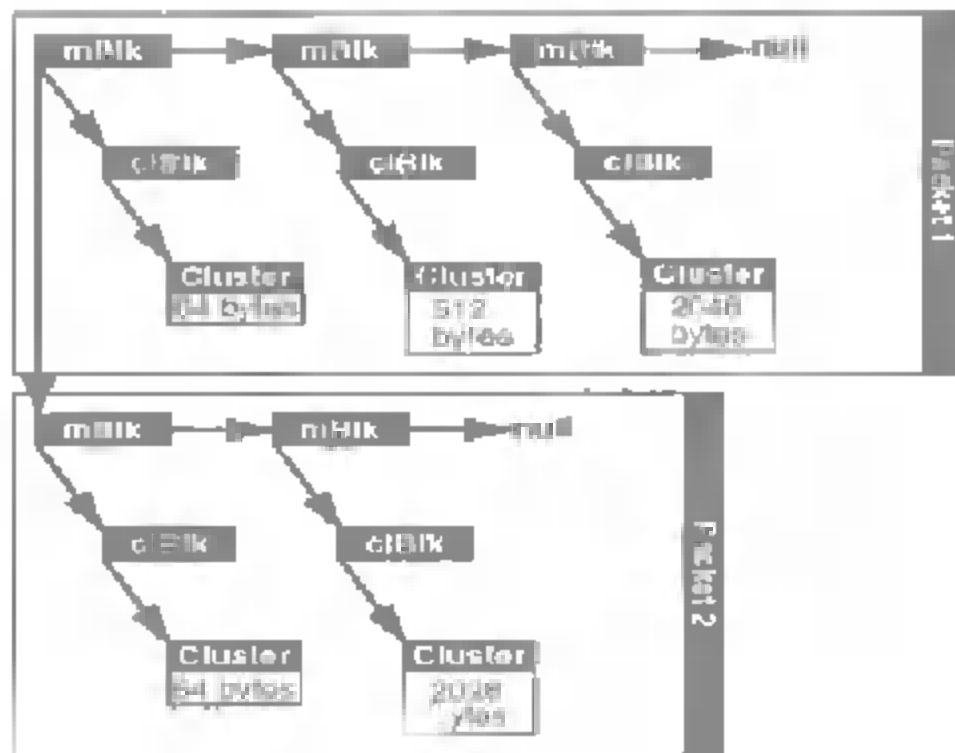


图 H-3 在 TCP 层出现两个数据包

另外，mBlk 结构通过 cBlk 结构涉及数据，这使得多种 mBlk 共享同一个机群而变得容易。当用户想要复制 mBlk 而不复制所涉及数据的时候，此功能很有用。例如，图 H-4 中的 mBlk 为 mBlk A 的复制。它们两个都属于同一个 cBlk，因此共享同一个机群。

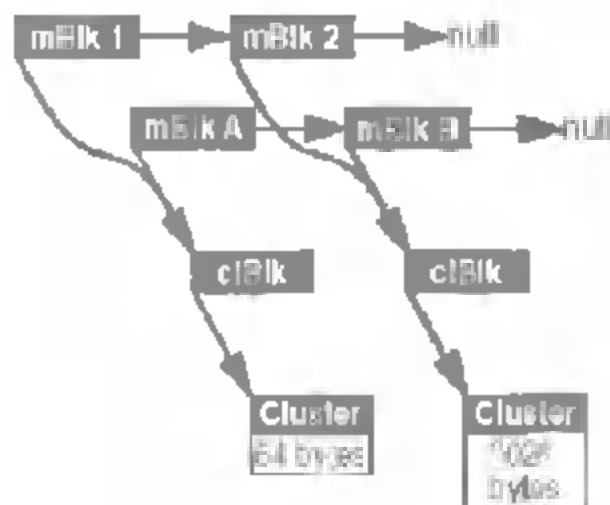


图 H-4 两个 mBlk 共享相同的 Cluster

注意这 使用 `cBlk` 结构而不用指针来进行一种间接操作是很有必要的。`cBlk` 结构会统计有多少 `mBlk` 共享其下层机群。当释放 `mBlk`、`cBlk` 或机群构造的时候这是很重要的。若用户使用 `netMblkCIFree()` 来释放构造, `mBlk` 被释放到层中, `cBlk` 中的计数增加。若计数为 0, `cBlk` 和机群也同样被释放到内存池。

1. 内存池的建立

每一个 END 元都需要有其自身的内存池。如何配置内存池取决于用户想要其作为网络协议 (如 Ipv4) 还是 END。

如上所述, 所有内存池均在 `mBlk` 结构、`cBlk` 结构和机群周围组织起来。然而, 由于一个网络协议通常需要不同大小的机群, 它的内存池也必须包含很多机群层 (每一机群层对应一个大小的机群)。

另外, 每个机群的大小必须为 2 的方幂。这种结构内存池的常用机群大小为 64、128、256、512、1024 比特。见图 H-5。

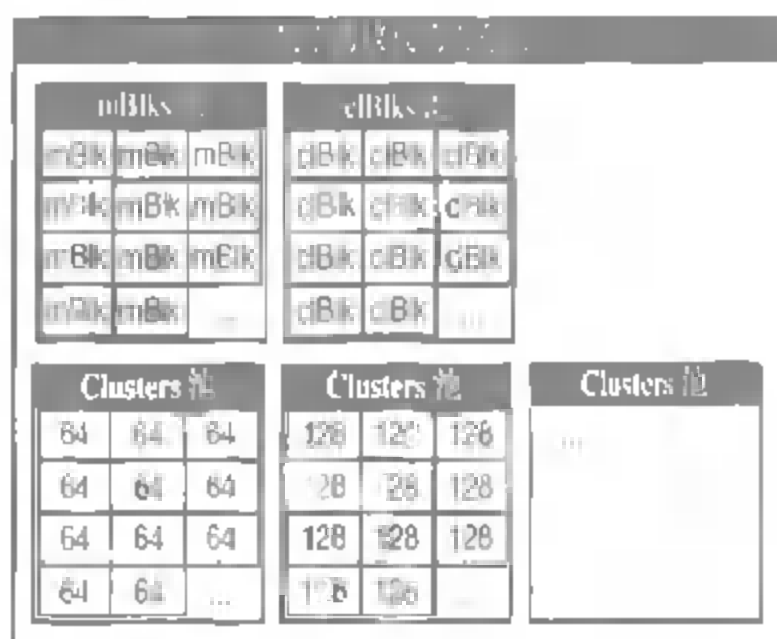


图 H-5 协议内存池

相比较而言, END 的内存池一般只包含一个机群层 (除 `mBlk` 池和 `cBlk` 池外), 并且机群的大小不一定是 2 的方幂。因此, 用户可以根据网络 MTU 任意选择方便的机群大小。例如, 在 Lance 以太 END 中, 考虑到以太 MTU 再加上一些余量, 机群大小设为 1520 比特, 见图 H-6。



图 H-6 END 内存池



注意： mBlk 和机群的比例推荐为 3 : 1。

调用 netPoolInit() 建立内存池。对于 END，在驱动程序初始化阶段调用 netPoolInit()。在此池外，用户驱动程序应保存 mBlk、cBlk 和机群构造以便其接受数据包。详情参见 netBufLib 参考条目。

2. 接收数据放入机群中

在 END 的初始化阶段，调用 netClusterGet() 保存一些合理数量的机群来接收即将传递的数据包（根据用户需要，以缓冲环或其他结构形式在这些保留机群中存储参考量）。当一个数据包来到时，用户 END 应直接接收并把它保留到这些机群中。

当用户 END 准备传递一个接收数据包到 MUX 时，它使用一个 mBlk、cBlk 或机群结构。创建这个结构需要：

- (1) 调用 netCBlkGet() 来保存 cBlk 结构。
- (2) 调用 netCBlkJoin() 把 cBlk 加入到包所在的机群内。
- (3) 调用 netMblkGet() 来保存 mBlk 结构。
- (4) 调用 netMblkCJoin() 把 mBlk 结构加入到 cBlk 或机群构造中。

现在 mBlk、cBlk 或机群构造中已经包含了数据包，END 可以使用不同的 netBufLib 程序在 mBlk、cBlk 或机群构造中来调整或检查数据。例如，若要读取在构造中的包数据，用户 END 可以调用 netMblkToBufCopy()。此外，END 可以使用 mBlk 的链接性在包中计划和添加数据。



注意： 不同的 net*Get() 程序在预分配池中保留了存储空间。从内部来看，它们不需要信号量。因此，它们在不可能调用 malloc() 的地方却可以被安全调用。

3. 释放 mBlk、cBlk 和机群

当使用过 mBlk、cBlk 或机群链接后需要释放它们的时候，调用 netMblkCChainFree()。这样会释放链中的所有 mBlk。同时链中所有 cBlk 中的参考计数器也会增值。若 cBlk 的参考计数器值为 0，则这个 cBlk 和其相关的机群被释放到池中。若要释放单一的 mBlk、cBlk 或机群到内存池，使用 netMblkCFree()。有关此方法和其他程序的详细介绍请参见 netBufLib 的参考条目。

H.3.2 协议和驱动程序之间的交换缓冲器

为将一个缓冲区中的数据传送给 MUX，一个名为 muxReceive() 的驱动程序轮流的调用协议中的 stackRcvRtn() 函数。当 muxReceive() 返回控制信息时，驱动程序认为数据被成

功传递, 可以忽略传递到 MUX 中的缓冲器。当上层处理完数据后, 释放缓冲器到驱动程序内存池。

此系统为一暗含的 buffer loaning 系统。若一设备不能直接接收数据到机群中, 它必须首先分配其内存 (除网络缓冲池外), 然后在使其包含在 mBlk 中传递给 MUX 之前, 从私有内存中复制数据到机群中。

H.3.3 使用个人内存管理程序

若设备不允许使用 MUX 存储管理设备, 用户可以自行编写 netBufLib 提供的种种程序。然而, 这些程序必须遵守 API。

H.3.4 支持驱动程序中的离合

一些设备支持把一个网络包裹分成几个独立的内存块。这使得以 mBlk、clBlk 和机群构造形式传递数据包成为可能 (例如 Berkeley 类型 TCP/IP 协议的 mbuf 清单)。

当一个驱动程序得到一个 mblk 链后, 便决定以其喜好的链接形式传递机群。若它可以聚集编写便不需要数据复制。若不可以, 则它需要在传送数据前把所有链接的数据放入一个内存空间中。这是由执行的驱动程序的网络设备的类型所驱动的。

H.4 指明错误条件

有些时候 END 会遇到错误或者其他与协议使用 END 有关的情况。例如, 设备死机或者时断时续。这种情况下, END 应该调用 muxError()。此程序传送错误信息到 MUX, 接着将这些信息传递给所有已经注册了接受函数的协议。muxError() 程序被定义如下:

```
void muxError
(
void* pCookie, /* pointer to END_OBJ */
END_ERR* pError /* pointer to END_ERR structure */
)
```

在输入中, 此程序需要一个指向 end_err 结构的指针, 此指针在 end.h 是这样定义的:

```
typedef struct end_err
(
INT32 errCode; /* error code, see above */
```



```
char* pMesg; /* NULL-terminated error message, can be NULL */
void* pSpare; /* pointer to user defined data, can be NULL */
} END_ERR;
```

和 MUX 一起注册的协议错误接收程序必有以下原型：

```
void xxError
(
  END_OBJ* pEnd, /* pointer to END_OBJ */
  END_ERR* pError, /* pointer to END_ERR */
  void* pSpare /* pointer to protocol private data passed in muxBind */
)
```

end_err 结构的 errCode 成员为 32 比特长。Wind River 保留了 errCode 的低 16 比特为其自身的错误信息。然而，高 16 位可以为用户使用。使用这些比特来编写需要在驱动程序和协议之间传输的任何错误信息。目前定义错误代码如下：

```
#define END_ERR_INFO 1 /* information only */
#define END_ERR_WARN 2 /* warning */
#define END_ERR_RESET 3 /* device has reset */
#define END_ERR_DOWN 4 /* device has gone down */
#define END_ERR_UP 5 /* device has come back on line */
#define END_ERR_FLAGS 6 /* device flags have changed */
#define END_ERR_NO_BUF 7 /* device's cluster pool is exhausted */
```

这些错误代码具有以下含义：

- END_ERR_INFO。此错误只为信息。
- END_ERR_WARN。发生了一个不严重的错误。
- END_ERR_RESET。错误发生，强迫设备复位，但是设备可以恢复。
- END_ERR_DOWN。发生了一个严重的错误，致使设备死机，不能再用来传递和接收数据包。
- END_ERR_UP。设备以前死机但现在又恢复正常，可以接收和传递数据包。
- END_ERR_BLOCK。设备忙，稍后再做处理。
- END_ERR_FLAGS。改变设备标志。
- END_ERR_NO_BUF。耗尽了设备的机群池。

H.5 所需的驱动程序入口点和结构

本节描述 END 的 API。介绍了驱动程序的重要结构及执行所需的入口点。



注意: END 的组织结构不同于标准 VxWorks I/O 驱动程序模型。驱动程序不能通过 `open()` 程序或其他文件的 I/O 程序得到。驱动程序是为配合 MUX 而组织形成的。MUX 控制与网络协议的联系。

H.5.1 驱动程序的必需结构

在驱动程序中, 用户必须分配和初始化 END_OBJ。用户驱动程序还需要分配和初始化 END_OBJ 中的结构参量, 如 DEV_OBJ, NET_FUNCS 和 M2_INTERFACETBL。使用 mBlk 结构把数据包传送给 MUX。

1. 网络工具抽象: END_OBJ

`endLoad()` 入口点必须分配、初始化和返回一个 END_OBJ 结构。MUX 使用此结构作为一个存储控制堆栈和设备驱动程序工具的地方。这些工具包括数据和程序指针。END_OBJ 结构在 `end.h` 中定义如下:

```
typedef struct end_object
{
    NODE node; /* root of the device hierarchy */
    DEV_OBJ devObject; /* accesses your device's ctrl struct */
    FUNCPTR receiverRtn; /* routine to call on reception */
    BOOL attached; /* indicates unit is attached */
    SEM_ID txSem; /* transmitter semaphore */
    long flags; /* various flags */
    struct net_funcs *pFuncTable; /* function table */
    M2_INTERFACETBL mib2Tbl; /* MIBII counters */
    struct ETHER_MULTI *pAddrList; /* head of the multicast address list */
    int nMulti; /* number of elements in the list */
    LIST protocols; /* protocol node list */
    BOOL snarfProto; /* is someone snarfing us? */
    void* pMemPool; /* memory cookie used by MUX bufr mgr. */
    M2_ID* pMib2Tbl; /* RFC 2211 MIB objects */
} END_OBJ;
```

用户驱动程序负责设置和控制一些这样的成员。其他成员由 MUX 控制。关于如何分配这些成员, 请阅读以下成员描述:

- **node.** 设备层次根部。由 MUX 设定其值。对于用户驱动程序不透明。
- **DevObject.** 此设备的 DEV_OBJ 结构。在装载时期用户驱动程序设定其值。详细内容请参见跟踪设备控制结构: DEV_OBJ。

- **receiveRtn**。涉及 **muxReceive()** 函数的指针。MUX 调用 **muxDevLoad()** 装载完毕驱动程序后支持此指针。用户驱动程序使用此函数指针向协议传送数据。
- **attached**。一个 **BOOL** 表明设备是否被连接上。MUX 设定和控制此值。
- **txSem**。控制访问设备传输工具的信号量。MUX 设定和控制此值。
- **flags**。由 **IFF_*** 标志常量中的 **Oring** 产生的值。除了 **IFF_LOAN** 和 **IFF_SCAT**，这些常量均为与 TCP/IP 堆栈有关的 **IFF_*** 标志。

IFF_UP	接口驱动程序完成。
IFF_BROADCAST	广播地址有效。
IFF_DEBUG	进行调试。
IFF_LOOPBACK	逆环网络。
IFF_POINTOPOINT	接口为一点对点网络。
IFF_NOTRAILERS	设备避免使用跟踪器。
IFF_RUNNING	设备成功分配所需资源。
IFF_NOARP	没有地址决定协议。
IFF_PROMISC	设备接收所有数据包。
IFF_ALLMULTI	设备接收所有多点传送数据包。
IFF_OACTIVE	传送进程。
IFF_SIMPLEX	设备不知道其自身传送。
IFF_LINK0	每一连接层定义的比特。
IFF_LINK1	每一连接层定义的比特。
IFF_LINK2	每一连接层定义的比特。
IFF_MULTICAST	设备支持多点传送。
IFF_LOAN	设备支持缓冲器 loaning。
IFF_SCAT	设备支持分散和集合。

- **pFuncTable**。指向 **net_funcs** 结构的指针。此结构包含指向用户驱动程序负责控制标准需求诸如卸载和发送的入口点的函数指针。当设备转载完毕后，用户驱动程序分配和初始化此结构。参见鉴别用户网络驱动程序入口点：**NET_FUNCS**。
- **mib2Tbl**。一个跟踪驱动程序中使用的 **MIB-II** 变量的 **M2_INTERFACETBL** 结构。用户驱动程序必须初始化此结构，然后驱动程序和 MUX 将会调整表中存储值。



注意： **mib2Tbl** 与 **RFC1213** 后向兼容。对于新的驱动程序最好不使用它，而使用 **RFC2233** 接口。详情请见附录 H.7。

- **pAddrList**。指向多点传送地址列表首部的指针。MUX 使用用户驱动程序中的 **endMCastAddrAdd()**、**endMCastAddrDel()** 和 **endMCastAddrGet()** 入口点设定和控制此列表。
- **nMulti**。表明在 **multiList** 成员中所涉及清单的地址数。MUX 使用驱动程序的

`endMCastAddrGet()` 的返回信息设定此值。

- `protocols`。网络驱动程序范围内的协议清单首部。MUX 控制这个清单。
- `snarfProto`。一个 `BOOL` 表明数据包阻塞协议是否在驱动程序范围内。这个协议可以阻止数据包传递给低优先级的协议 (见阻止数据包传向低优先级协议的阻塞协议)。MUX 设定和控制此值。
- `pMemPool`。一个指向 `netBufLib` 所控制的内存池的指针。MUX 设定此成员值, 把它作为不透明处理。
- `pMib2Tbl`。兼容于 RFC2233 的接口列表。

2. 跟踪用户设备的控制结构: `DEV_OBJ`

用户驱动程序使用 `DEV_OBJ` 结构通知 MUX 设备名称, 并给 MUX 一个指针来指向设备的控制结构。这个控制结构为根据用户需求所定义的特定工具结构。用户驱动程序使用此控制结构来跟踪诸如标志、内存池、地址等信息。在控制结构中存储的信息一般对于每一个驱动程序入口点都很重要。`DEV_OBJ` 结构在 `end.h` 中定义如下:

```
typedef struct dev_obj
{
    char name[END_NAME_MAX]; /* device name */
    int unit; /* to support multiple units */
    char description[END_DESC_MAX]; /* text description */
    void* pDevice; /* pointer back to the device data. */
} DEV_OBJ;
```

- `name`。指向一个最多含有 8 字节的字符串指针。此字符串定义了网络设备的名称。
- `pDevice`。指向驱动程序内部控制结构的指针。为了保持协议和数据连接层的独立性, MUX 把 `pDevice` 指针作不透明处理。因此, MUX 不需要知道控制结构。然而, 当调用用户驱动程序入口点时, MUX 会传递一 `cookie` (除 `endLoad()` 程序外)。此 `cookie` 为一指向由 `endLoad()` 分配的 `END_OBJ` 的指针。通过这个 `cookie`, 用户入口点可以与 `pCookie.devObject.pDevice` 中的设备控制结构连接起来。
- `unit`。这是一个特定名称设备数量单元。单元数量从 0 开始, 随同一驱动程序控制的设备数量增加而增加。例如, 若系统拥有 2 个 Lance 以太网设备 (称为 `ln`), 分别命名为 `ln0` 和 `ln1`。若这个系统还拥有一个 DEC21x40 以太网, 则设备 (名称为 `dc`) 为 `dc0`。
- `description`。此为设备驱动程序的文字描述。例如, Lance 以太驱动程序在此位置写入 “AMD7990 Lance 以太增强型网络驱动程序”。在调用 `muxShow()` 程序时显示此字符串。

3. 鉴别用户网络驱动程序入口点: `NET_FUNCS`

MUX 使用 `NET_FUNCS` 结构来维护 `END` 的入口点列表。`NET_FUNCS` 结构定义如下:

```

typedef struct net_funcs
{
    STATUS (*start) (void*); /* driver's start func */
    STATUS (*stop) (void*); /* driver's stop func */
    STATUS (*unload) (void*); /* Driver's unload func */
    int (*ioctl) (void*, int, caddr_t); /* driver's ioctl func */
    STATUS (*send) (void*, M_BLK_ID); /* driver's send func */
    STATUS (*mCastAddrAdd) (void*, char*); /* driver's mcast add func */
    STATUS (*mCastAddrDel) (void*, char*); /* driver's mcast delete func */
    STATUS (*mCastAddrGet) (void*, MULTI_TABLE*);
    /* driver's mcast get fun. */
    STATUS (*pollSend) (void*, M_BLK_ID); /* driver's poll send func */
    STATUS (*pollRcv) (void*, M_BLK_ID); /* driver's poll receive func */
    STATUS (*addressForm) (M_BLK_ID, M_BLK_ID, M_BLK_ID);
    /* driver's addr formation func */
    STATUS (*packetDataGet) (M_BLK_ID, M_BLK_ID);
    /* driver's pkt data get func */
    STATUS (*addrGet) (M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID);
    /* driver's pkt addr get func */
} NET_FUNCS;

```

在 `endLoad()` 中, 初始化这些成员指向相应的驱动程序入口点。因此, `start` 应包含一个指向 `endStart()` 的指针, `stop` 应包含一个指向 `endStop()` 的指针, `unload` 应包含一个指向 `endUnload()` 的指针, 等等。

4. 跟踪连接级信息: LL_HDR_INFO

MUX 使用 `LL_HDR_INFO` 结构来跟踪从 END 到 MUX 继而传送到协议的数据包的连接级首部信息。`LL_HDR_INFO` 结构是一个对所有栈接收程序均适用的自变量 (见向协议传递数据包: `stackRcvRtn()`)。

```

typedef struct llHdrInfo
{
    int destAddrOffset; /* destination addr offset in mBlk */
    int destSize; /* destination address size */
    int srcAddrOffset; /* source address offset in mBlk */
    int srcSize; /* source address size */
    int ctrlAddrOffset; /* control info offset in mBlk */
    int ctrlSize; /* control info size */
    int pktType; /* type of the packet */
    int dataOffset; /* data offset in the mBlk */
} LL_HDR_INFO;

```

- **destAddrOffset**。mBlk 结构中的偏移量，标明终点地址位置。
- **destSize**。终点地址大小。
- **srcAddrOffset**。mBlk 结构中的偏移量，标明起点地址位置。
- **srcSize**。源地址大小。
- **ctrlAddrOffset**。预留。
- **ctrlSize**。预留。
- **pktType**。数据包类型。有效数据包类型请见 RFC1700。
- **dataOffset**。mBlk 结构中的偏移量，标明数据包起始位置。

5. 跟踪驱动程序与协议之间传送的数据：mBlk

使用 mblk 结构在驱动程序与协议层之间传递数据包。mBlk 结构在 netBufLib.h 中定义如下：

```
typedef struct mBlk
{
    M_BLK_HDR mBlkHdr; /* header */
    M_PKT_HDR mBlkPktHdr; /* pkthdr */
    CL_BLK * pClBlk; /* pointer to cluster blk */
} M_BLK;
```

- **mBlkHdr**。包含一个指向 mHdr 结构的指针。大多数情况下，用户不需要直接访问或设定此成员值而只需予以不透明处理。惟一的例外为，当须连接多个 mBlk 的时候，用户需要设定 mBlk.mHdr.mNext 或 mBlk.mBlkHdr.mNextPkt 或两者的值。使用 mBlk.mBlkHdr.mNextPkt 在 mBlk 链中指向下一个 mBlk。使用 mBlk.mHdr.mNext 指向含有下一个数据包首部的 mBlk。
- **mBlkPktHdr**。包含一个指向 pktHdr 结构的指针。用户无须直接访问或设定这个成员而可把它作为不透明处理。
- **pClBlk**。包含一个指向 clBlk 结构的指针。用户无须直接访问或设定这个成员而可把它作为不透明处理。然而，若不是使用 netBufLib 来管理驱动程序的内存池，用户就必须为内存的相关机群提供自由程序。此程序必须使用与 netBufLib 自由程序相同的 API。这就意味着必须升级 mBlk.pClBlk.pFreeArg1、mBlk.pClBlk.pFreeArg2 和 mBlk.pClBlk.pFreeArg3 成员。

为上述成员（和所有相关结构成员）设定适当值只需为创建一个 mBlk、clBlk 或机群结构调用相关的 netBufLib 程序。详情请见附录 H.3.1。

H.5.2 必需的驱动程序入口点

本附录所描述的所有入口点名称均以 end 开头。这表明它们为通用驱动程序入口点。

在一个特定的网络驱动程序中，特定的入口点需用一个表明属于哪个驱动程序的前缀开头。例如，与 AMD Lance 驱动程序有关的入口点中应使用 In 前缀。因此用户网络接口驱动程序定义入口点为 InLoad()、InUnload()、InReceive()等。

对于驱动程序入口点，这种命名规则是非常好的。因为 VxWorks 使用在 NET_FUNCS 结构中装载的函数指针来访问这些入口点，对于入口点用户可以自由根据其他规则来为之命名，见表 H-1。

表 H-1 必须的驱动程序入口点

程 序	目 的
endLoad()	初始化驱动程序并装载入 MUX 中
endUnload()	自由驱动程序资源
endStart()	启动驱动程序
endStop()	终止驱动程序
endSend()	为硬件传递数据包
endIoctl()	访问驱动程序控制函数
endMcastAddrAdd()	为设备的多点传送地址表添加一地址
endMcastAddrDel()	从设备的多点传送地址表中去掉一地址
endMcastAddrGet()	从设备的多点传送地址表中获得一地址
endPollSend()	polling 发送
endPollReceive()	polling 接收
endAddressForm()	在 mBlk 中添加相关连接级信息，准备数据传送
endPacketDataGet()	从一个 mBlk 中提取包数据（去掉连接级信息），并写入其他 mBlk
endPacketAddrGet()	从一个 mBlk 中提取地址信息（去掉包数据）并在自身 mBlk 中写入每一源地址和目标地址。对于一个以太网数据包，这里需要 2 个输出 mBlk。然而，对于一些非以太网数据包，这里需要 4 个输出 mBlk，因为本地源地址和目标地址与终点的源地址和目标地址有所不同



注意：若从 BSD4.3 模型中装载一个驱动程序，我们也许倾向与使用已有的 xxIoctl() 入口点作为 endIoctl() 入口点，这样看上去省略了为不同 McastAddr 入口点创建独立入口点的工作。但是千万不要这样做，驱动程序必须执行所有表 H-1 中所列的入口点。

1. 装载设备：endLoad()

在基于 4.3BSD 的系统中，endLoad() 入口点与 attach() 入口点的作用相同。endLoad() 是每一个网络接口驱动程序的初始入口点。当 tUserRoot 任务调用 muxDevLoad() 来装载驱

动程序的时候, 它规定 `endLoad()` 为一输入参数。

`endLoad()` 必须遵守以下格式:

```
END_OBJ* endLoad
(
char* initString /* a string encoded for the device to use for its
/* initialization arguments. */
)
```

在 `endLoad()` 中, 用户需要处理所有特定设备的初始化。还需要为 `END_OBJ` 的大多数成员赋值。需要特别关注的是 `END_OBJ` 的 `receiveRtn`、`pFuncTable` 和 `devObject` 成员。见网络设备摘要中的成员描述: `END_OBJ`。

`endLoad()` 应返回一个指向初始化的 `END_OBJ` 结构的指针。若有错误发生则返回 `ERROR`。

参数为:

`initString`, 在任何所需初始化自变量中忽略。

2. 卸载设备: `endUnload()`

`endUnload()` 入口点处理从系统中去掉网络驱动程序的任何问题。在 `endUnload()` 中, 用户需要处理诸如清除所有本地数据结构等问题。`endUnload()` 无须通报卸载设备协议。在调用 `endUnload()` 前, `MUX` 为与设备所连接的每一个协议发送一个关闭通告。

`endUnload()` 须使用以下形式:

```
void endUnload
(
void* pCookie /* pointer to device-identifying END_OBJ */
)
```

此函数被定义为 `void`, 因此需返回一函数值。

参数为:

`pCookie`, 指向由 `endLoad()` 返回的 `END_OBJ` 结构的指针。在 `endUnload()` 中, 也许需要释放其相关的存储空间。

保证删除任何在驱动程序中产生的信号量。

3. 为驱动程序提供一不透明控制接口: `endIoctl()`

`endIoctl()` 入口点处理设备状态的任何变化要求, 比如启动、关闭、切换到混合模式等。用户也可使用 `endIoctl()` 来访问 MIB-II 接口统计表。



警告: 若使用 BSD4.3 模型装载驱动程序, 我们也许倾向于使用现有的 `xxIoctl()` 入口点作为 `endIoctl()` 入口点, 这样做看上去可以跳过为

McastAddr 入口点创建独立的入口点的工作,但千万不要这样做!用户驱动程序必须执行表 H-1 中所列的所有入口点。

endIoctl() 必须遵守以下格式:

```
STATUS endIoctl
(
    void* pCookie, /* pointer to device-identifying END_OBJ */
    int cmd, /* value identifying command */
    caddr_t data /* data needed to complete command */
)
```

此函数返回 OK 或 ERROR。若发生错误,它应设置 errno。

参数为:

- pCookie。为 endLoad() 返回的 END_OBJ 结构分配一指针。
- cmd。可以传递表 H-2 中第一栏所示的任何值。endIoctl() 对每一值均需有相应回答。
- data。为 endIoctl() 所需的数据传递数据或指针,以便 endIoctl() 执行 cmd 中规定的命令。

表 H-2 ioctl 命令和数据类型

命 令	函 数	数据类型
EIOCSFLAGS	设定设备标志	int; 见有关 END_OBJ.flags 的描述
EIOCGFLAGS	获得设备标志	int
EIOCSADDR	设定设备地址	char*
EIOCGADDR	获得设备地址	char*
EIOCMULTIADD	添加多点传送地址	char*
EIOCMULTIDEL	去除多点传送地址	char*
EIOCMULTIGET	获得多点传送清单	MULTI_TABLE*
EIOCPOLLSTART	设定设备为 polling 模式	NULL
EIOCPOLLSTOP	设定设备为中断模式	NULL
EIOCGFBUF	获得链的最小第一缓冲器	int
EIOCGMIB2	从驱动程序中获得 MIB-II 的计数器	M2_INTERFACETBL*

4. 为设备传送数据: endSend()

当需要为设备传送数据时, MUX 调用 endSend() 入口点。endSend() 须遵循以下格式:

```
STATUS endSend
```

```
(
void* pCookie, /* device structure */
M_BLK_ID pMblk, /* data to send */
)
```

此函数返回 OK、ERROR 或 END_ERR_BLOCK。

若由于 polling 模式或资源缺乏而数据包在这一时刻不能被传送, END_ERR_BLOCK 的值需被返回。在以上两种情况下, 不要从 mBlk 链中释放数据包。

当数据包被成功接收后返回 OK 值。若有错误发生则返回 ERROR 并设定 errno。在这种情况下, mBlk 链释放数据包。

参数为:

- pCookie。为 endLoad() 返回的 END_OBJ 结构分配一指针。
- pMblk。为包含所需传送数据的 mBlk 结构分配一指针。有关如何设置 mBlk 详情请见附录 H.2.1。

5. 启动一暂停的载入驱动程序: endStart()

endStart() 入口点用来激活驱动程序。例如, 它注册了用户设备驱动程序的中断服务程序。endStart() 必须遵守以下格式:

```
Statue endStart
(
void* pCookie /* pointer to device-identifying END_OBJ structure */
)
```

此函数返回 OK 或 ERROR。若有错误发生, 设定 errno。

参数为:

- pCookie。为 endLoad() 返回的 END_OBJ 结构分配一指针。然而, endStart() 应该将此指针作为一个 sysIntConnect() 调用的参量用来注册 ISR。ISR 也许对于这个 END_OBJ 指针没有任何直接作用, 但是它却将指针指向驱动程序入口点从而处理任务级的数据包接受进程。

当向 MUX 传递数据包的时候, 驱动程序须调用在 pCookie.receiveRm 中的 MUX 所支持的函数。见网络设备摘要: END_OBJ。

6. 停止而不卸载驱动程序: endStop()

endStop() 入口点假设驱动程序已经装载完毕并且用户已经调用了 endLoad()。在 endStop() 中, 用户应该停止而不是卸载驱动程序。endStop() 应遵守以下格式:

```
STATUS endStop
(
void* pCookie /* pointer to a device-identifying END_OBJ structure */
)
```


)

此函数返回 OK 或 ERROR。若有错误发生，设定 `errno`。

参数为：

- `pCookie`。为 `endLoad()` 返回的 `END_OBJ` 结构分配一指针。

7. 处理 Polling 传送：endPollSend()

`endPollSend()` 用来处理有关 Polling 传送的所有问题（比如调试代理）。因此，`endPollSend()` 应该可以直接向网络堆栈中存放数据包而无须在输出队列中等待。



注意：当系统调用 `endPollSend()` 的时候，也许其正处于不能为内核调用服务的模式。因此，入口点不应调用任何内核函数，诸如执行信号量或分配内存。同样地，此入口点不应处于阻塞或忙状态，因为这样会暂停整个系统。

`endPollSend()` 应遵循以下结构：

```
STATUS endPollSend
(
    void* pCookie, /* device structure */
    M_BLK_ID pMblk, /* data to send */
)
```

在 `endPollSend()` 中，检查以保证设备处于 `polled` 模式（由先前的 `endIoctl()` 调用所设置）。然后 `endPollSend()` 直接把数据包（使用 `pNBuf` 传送）传递给网络。`endPollSend()` 使数据包避免了在任何输出队列中等待。

此函数应返回 OK 或 ERROR。若有错误发生，设定 `errno`。

参数为：

- `pCookie`。为 `endLoad()` 返回的 `END_OBJ` 结构分配一指针。
- `pMblk`。为包含所需传送数据的 `mBlk` 结构分配一指针。关于 `mBlk` 的建立参见附录 H.3.1。

8. 处理 Polling 接收：endPollReceive()

`endPollReceive()` 用来处理有关 Polling 接收的所有问题（比如调试代理）。



注意：当系统调用 `endPollReceive()` 的时候，也许其正处于不能为内核调用服务的模式。因此，入口点不应调用任何内核函数，诸如执行信号量或分配存储空间。同样地，此入口点不应处于阻塞或忙状态，因为这样会暂停整个系统。

endPollReceive() 必须遵守以下格式:

```
int endPollReceive
(
    void* pCookie, /* device structure */
    M_BLK_ID pMblk /* place to return the data */
)
```

endPollReceive() 应该检查设备处于 **polled** 模式 (通过调用 **endIoctl()**)。然后 **endPollReceive()** 直接从网络中获得一数据包, 并通过 **pMblk** 参数将它复制到 **mblk** 中。

endPollReceive() 返回 **OK** 或相关错误值。很可能的错误返回信息为 **EAGAIN**。若递交的 **mblk** 不足以存储接收数据包或者无有效数据包的时候, **endPollReceive()** 应返回 **EAGAIN**。

参数为:

- **pCookie**。为 **endLoad()** 返回的 **END_OBJ** 结构分配一指针。
- **pMblk**。分配一指向 **mblk** 结构的指针。它为一个输出参数。**endPollReceive()** 须从堆栈中复制数据到 **mblk** 结构。

9. 添加多点传送地址: **endMCastAddrAdd()**

endMCastAddrAdd() 入口点用来向设备所存储的多点传送表中添加地址。它须遵守以下格式:

```
STATUS endMCastAddrAdd
(
    void* pCookie, /* pointer to a device-identifying END_OBJ structure */
    char* pAddress /* pointer to address to add */
)
```

为了帮助用户控制多点传送地址清单, **VxWorks** 提供了 **etherMultiLib** 库。

此函数应返回 **OK** 或者 **ERROR**。若有错误发生, 设定 **errno**。

参数为:

- **pCookie**。为 **endLoad()** 返回的 **END_OBJ** 结构分配一指针。
- **pAddress**。分配一个指向用户想要添加到清单中的地址指针。为了帮助用户控制多点传送地址, **VxWorks** 包含了一个 **etherMultiLib** 库。

在 **endMCastAddrAdd()** 中, 必须以特定硬件方式重新装配接口。这种重新装配使得驱动程序从特定地址接收帧, 然后将这些帧传送到高级层中。

10. 删除多点传送地址: **endMCastAddrDel()**

endMCastAddrDel() 入口点用来从设备所存储的多点传送表中删除地址。它必须遵循以

下格式:

```
STATUS endMCastDelAddr
(
    void* pCookie, /*pointer to a device-identifying END_OBJ structure */
    char* pAddress, /* pointer to address to delete */
)
```

此函数返回 OK 或 ERROR。若有错误发生, 设定 `errno`。

参数为:

- `pCookie`。为 `endLoad()` 返回的 `END_OBJ` 结构分配一指针。
- `pAddress`。为将删除地址分配一指针。为了帮助用户控制多点传送地址清单, VxWorks 包含了 `etherMultiLib` 库。

11. 获取多点传送地址表: `endMCastAddrGet()`

`endMCastAddrGet()` 必须获得一个多点传送地址表, 然后将它放在由 `pMultiTable` 参量所涉及的缓冲器中。这些地址为接口正在使用的多点传送地址。`endMCastAddrGet()` 必须遵守以下格式:

```
STATUS endMCastGetAddr
(
    void* pCookie,
    MULTI_TABLE* pMultiTable
)
```

使用 `etherMultiLib` 中的程序获得多点传送地址清单。此函数返回 OK 或 ERROR。若有错误发生, 设定 `errno`。

参数为:

- `pCookie`。为 `endLoad()` 返回的 `END_OBJ` 结构分配一指针。
- `pMultiTable`。为缓冲器分配一指针。这是一个输出参数。`endMCastAddrGet()` 须在相关缓冲器中写入 `MULTI_TABLE` 结构。`end.h` 定义 `MULTI_TABLE` 如下:

```
typedef struct
{
    long len; /* length of table in bytes */
    char *pTable; /* pointer to entries */
} MULTI_TABLE;
```

修改 `MULTI_TABLE` 中的 `len` 成员, 指出返回了多少地址。将地址写入 `MULTI_TABLE` 成员 `pTable` 的缓冲器中。

12. 在传送数据包中创建一个地址: endAddressForm()

endAddressForm() 必须包含一个源地址和目标地址, 以与连接级相关的形式复制信息到 mBlk 结构的数据段。尽管 endLib 提供了一些常见情况, 驱动程序复写器负责执行此功能。在 mBlk 中添加完地址后, 程序 endAddressForm() 应该相应调整 mBlk.mBlkHdr.mLen 和 mBlk.mBlkHdr.mData 成员。此程序须遵循以下格式:

```
M_BLK_ID endAddressForm
(
  M_BLK_ID pMblk, /* packet data */
  M_BLK_ID pSrcAddress, /* source address */
  M_BLK_ID pDstAddress /* destination address */
)
```

此函数返回 M_BLK_ID, 它为一个 mBlk 结构链的潜在首部。

若 pMblk 的相关机群的空间不足以包含首部和包数据, 此程序需要保存一额外 mBlk、clBlk 或机群结构来存储首部。然后将 pMblk 中的 mBlk 链接到额外 mBlk 的预留首部, 并返回一指针指向首部 mBlk 作为函数值。

参数为:

- pMblk。包含传送数据包的 mBlk。
- pSrcAddress。包含连接级源地址的 mBlk。
- pDstAddress。包含连接级目的地址的 mBlk。

13. 获取一个只存储数据的 mBlk: endPacketDataGet()

此程序提供了一个 mBlk 复制, 这个复制包含原始包数据而忽略了首部信息。endLib 提供了一些常见情况。此程序返回 OK 或 ERROR, 若错误发生设定 errno。

程序为:

```
STATUS endPacketDataGet
(
  M_BLK_ID pBuff, /* packet data and address information */
  LL_HDR_INFO* pLinkHdrInfo /* structure to hold link-level info. */
)
```

参数为:

- pBuff。指向同时含有首部和包数据的 mBlk。
- pLinkHdrInfo。返回一个包含首部信息的 LL_HDR_INFO 结构, 此首部信息与 END 执行的特定数据连接层有关。详情请见“跟踪连接级信息: LL_HDR_INFO”。

14. 返回地址信息: endPacketAddrGet()

此程序用来得到与数据包有关的地址信息。endLib 提供了一些常见情况。最后两个参

数用于终端的硬件源地址和目标地址与本地源地址和目标地址不同的网络中。此程序返回 OK 或 ERROR。若有错误发生，则设定 `errno`。

程序形式为：

```
STATUS endPacketAddrGet
(
    M_BLK_ID pMblk, /* M_BLK_ID of the packet */
    M_BLK_ID pSrc, /* local source of packet */
    M_BLK_ID pDst, /* local destination of packet */
    M_BLK_ID pESrc, /* end source of packet */
    M_BLK_ID pEDst /* end destination of packet */
)
```

参数为：

- `pMblk`。一个指向 `mBlk` 结构的指针，从此结构中提取地址信息。
- `pSrc`。为 `mBlk` 结构赋值 NULL 或设定指针，将提取的数据的源地址（本地）写入 `mBlk` 结构中。
- `pDst`。为 `mBlk` 结构赋值 NULL 或设定指针，将提取的数据包的目的地址（本地）写入 `mBlk` 结构中。
- `pESrc`。为 `mBlk` 结构赋值 NULL 或设定指针，将提取的数据包的源地址（终点若不同）、终点地址写入 `mBlk` 结构中。
- `pEDst`。为 `mBlk` 结构赋值 NULL 或设定指针，将提取的包的终点地址（终点若不同）写入 `mBlk` 结构中。

H.6 编写使用 MUX API 的网络协议

本节介绍如何为基于 MUX 的模型装载协议。如图 H-1 所示，基于 MUX 的协议受限于上层 MUX，而网络接口驱动程序（END）受限于下层 MUX。因此，协议分布在 MUX 的顶端，也在 END 顶端。每一层的任务列于表 H-3。

表 H-3 层及任务

层	任 务
协议	传输层的接口，通过它申请任务、复制所需包数据
MUX	装载和卸载驱动程序 捆绑和松绑协议 调用每一范围内协议的接收程序 管理缓冲器 loaning 的存储空间

续表

层	任 务
网络接口驱动程序	处理硬件设备 为系统装载驱动程序 为系统卸载驱动程序 为上层提供缓冲器

协议复写器需要处理 MUX 的调用。任何与数据连接层 (即 MUX 之下的一层) 驱动程序有关的特定设备问题。

1. 协议启动

任何要接收数据包的协议必须首先将其自身与 MUX 连接起来。协议调用 `muxBind()` 完成此任务。函数返回值为一个“cookie”，表明了 MUX 所限制的协议的网络接口驱动程序。此协议必须保存这个 cookie 以便在 MUX 的后续调用中使用。

作为 `muxBind()` 的输入，用户必须规定网络设备的名称 (例如，`ln` 和 0、`ln` 和 1、`ei` 和 0 等)；与协议相关的接收、重启和关闭的函数；与附加协议有关的协议类型 (从 RFC1700) 和名称。

MUX 使用此协议类型区分协议的优先顺序。这种优先级决定哪个协议先接收数据包。当驱动程序传给 MUX 一数据包时，驱动程序包含一个指向 `END_OBJ` 结构的指针。此结构还包含一个 `protocols` 成员、关注设备的协议清单 (`NET_PROTOCOL` 结构清单) 首部。此清单中的协议顺序决定了哪个协议首先接收数据包。MUX 负责保存此清单。

若协议在它的 `muxBind()` 中规定一种 `MUX_PROTO_SNARF` 类型，则 MUX 在 `END_OBJ.protocols` 清单首部添加一协议。在任何给定时刻，MUX 只允许一个 `MUX_PROTO_SNARF` 类型的协议激活。若协议在它的 `muxBind()` 中规定一种 `MUX_PROTO_PROMISC` 类型，则 MUX 在 `END_OBJ.protocols` 清单尾部添加一协议。若协议在 `muxBind()` 中还规定了其他类型，则 MUX 在清单中 `MUX_PROTO_SNARF` 协议后添加这些类型的协议 (若无 `MUX_PROTO_SNARF` 协议则在清单首部添加)。

这种协议添加方式保证了在接收数据包之前，低优先级的协议无法盗用或破坏高优先级的协议的数据包。

■ 拦截协议阻塞数据包到低优先级协议。

若协议的接收程序返回 `TRUE`，并且协议不是 `MUX_PROTO_PROMISC` 类型的时候，MUX 将使协议拦截数据包 (即 MUX 使协议掌握数据包)。协议截取数据包后，MUX 就不会把它传送给 `END_OBJ.protocols` 清单中的其他协议了。

若想把协议作为防火墙，协议需要以 `MUX_PROTO_SNARF` 类型注册，并且从接收函数中应该返回 `TRUE`。若要协议显示设备上所有没有被截取的数据包，以 `MUX_PROTO_PROMISC` 类型注册协议。

■ 协议和网络地址。

写入此系统来支持 RFC894 类型的以太网数据包和 802.3 编码数据包。若协议使用 802.3 类型数据包，需以 SAP（服务访问点）形式注册协议类型。否则，协议必须使用 RFC1700 中的可用协议类型。

■ 输出协议。

单一协议可以被限制在用于过滤输出数据包的每一设备上。这种功能用于特定设备输出的每一个数据包的应用中。当协议注册时，MUX_PROTO_OUTPUT 类型被传递到 muxBind() 中。在这种类型下，只有 stackRcvRtn 参数有效。

2. 传送数据

为了在缓冲器中写入相关地址首信息，协议调用 muxAddressForm()。最后，协议调用 muxSend() 来传送数据包，通过由 muxBind() 返回的 cookie 以及含有所传递数据包的 mBlk。然后 MUX 将包传给驱动程序。

3. 接收数据

为了响应网络设备的中断请求，VxWorks 执行设备先前注册的中断服务程序。此程序从设备上获得数据包然后排列起来等待任务级处理，此时驱动程序为 MUX 准备数据包有关此处理详情请见附录 H.3。

调用 muxReceive() 将数据包传给 MUX。muxReceive() 程序决定了数据包的协议类型（IP 的 0x800，ARP 的 0x806 等），然后在协议清单中搜索任何使用此类型协议注册的协议。

若有协议处理此数据包，MUX 即将此包传递给协议的 muxBind() 调用所规定的 stackRcvRtn() 中。在把数据包传递给一些有限的协议（即非 MUX_PROTO_SNARF 或 MUX_PROTO_PROMISC 协议）前，muxReceive() 调用 muxPacketDataGet() 程序并向协议传递 2 个 mBlk。

第一个 mBlk 包含所有连接级的信息。第二个 mBlk 包含连接级首部后的所有信息。这种数据分割使协议跳过首部信息（在 do_protocol_with_type() 接口打破了 BSD4.3 模型）。然后协议负责数据包处理。

这种多路接受数据包的新方法打破基于 etherInputHook() 和 etherOutputHook() 程序的方法。若协议要获取一个接口上的所有未被传递数据包，应先复制数据。由于其他协议也需要调用原始数据，因此在此地点不要修改数据（即在接收缓冲器中）。

4. 支持分散集合设备

一些设备支持把一个单一的网络数据包分解成若干独立的存储块。这使得以 mBlk、cBlk 或机群结构链形式向下传递数据包（例如一个 Berkeley 类型 TCP/IP 协议的 mbufs 清单）成为可能。链中的每一个 mBlk 均通过 mBlk.mBlkHdr.mNext 成员与下一个 mBlk 连接起来。在传输过程中，驱动程序适当地对链进行处理。

5. 协议传输重启

MUX/END API 的特征之一就是当资源耗尽时，允许设备通知协议。通常协议拥有一

队等待传输的数据包。若通过调用 `muxSend()` 设备返回 `END_ERR_BLOCK`，则设备没有传送数据的所需资源。这时协议需要停止传送。当协议知道拥有可以重新传输所需要的资源后，调用 `muxTxRestart()` 程序，此程序会调用特定设备的每个附加协议的 `stackTxRestartRtn()`。协议应执行一个重启程序以便能够调动系统优势。

6. 协议的关闭

当协议使用完一个接口，或由于某种原因想要关闭协议的时候，调用 `muxUnbind()` 程序。此程序会通知 MUX 去分配 `NET_PROTOCOL` 以及为此协议所特定的其他内存。

H.6.1 MUX API 协议

本节介绍了协议用来与 MUX 作用的一些程序和数据结构。大多数工作是由 MUX 程序（表 H-4 中所列）所控制的。不同于以前的驱动程序入口点，用户无须执行 MUX 程序。这些程序可以直接从用户协议中调用。关于这些 MUX 程序的详细情况见有关的入口参考。

然而，这些 MUX 程序并没有包含完整的 MUX/协议接口。另外，一个协议必须执行一套处理诸如关闭、重启协议和为协议传递数据及错误信息的标准协议，MUX 接口程序见表 H-4。

表 H-4 MUX 接口程序

MUX 程序	作 用
<code>muxDevLoad()</code>	装载一个设备到 MUX 中
<code>muxDevStart()</code>	从 MUX 中启动一个设备
<code>muxBind()</code>	为 MUX 添加一协议
<code>muxSend()</code>	从协议中接收数据包然后传递给设备
<code>muxDataPacketGet()</code>	得到一个只含有包数据的 <code>mBlk</code> 。省略了连接级的首部信息
<code>muxAddressForm()</code>	为输出包分配一地址
<code>muxIoctl()</code>	访问控制函数
<code>muxMCastAddrAdd()</code>	为设备所含清单添加一多点传送地址
<code>muxMCastAddrDel()</code>	从设备所含清单中删除一多点传送地址
<code>muxMCastAddrGet()</code>	获取设备清单列表中的多点传送地址
<code>muxUnbind()</code>	从 MUX 中分离协议
<code>muxDevStop()</code>	终止一设备
<code>muxDevUnload()</code>	卸载一设备
<code>muxPacketDataGet()</code>	从递交的 <code>mBlk</code> 中提取包数据（省略了连接级数据）并写入一新 <code>mBlk</code>
<code>muxPacketAddrGet()</code>	从递交的 <code>mBlk</code> 中提取源地址和目标地址数据（省略了包数据）并将每一地址写入其自身的 <code>mBlk</code> 中。若本地源地址或目标地址与终点源地址和目标地址不同，则此程序需要写入 4 个 <code>mBlk</code> 中

续表

MUX 程序	作 用
<code>muxTxRestart()</code>	若想设备在阻塞后除去障碍，此程序会调用与每一个相关协议对应的。 <code>stackTxRestartRtn()</code> 程序
<code>muxReceive()</code>	从设备向 MUX 传递数据包
<code>muxShutdown()</code>	关闭此设备上的所有协议
<code>muxAddrResFuncAdd()</code>	在地址工程函数列表中添加一地址工程函数
<code>muxAddrResFuncGet()</code>	从列表中获取一个特定地址工程函数
<code>muxAddrResFuncDel()</code>	删除列表中的一个特定地址工程

1. 协议数据结构 NET_PROTOCOL

MUX 为设备的每一协议分配了一个 NET_PROTOCOL 结构。MUX 使用此结构来存储与协议有关的信息，比如协议类型、接收程序及关闭程序。这些被链接到连接清单中，此清单首部存放在 END_OBJ 结构的 protocols 成员中，MUX 使用这个结构来控制设备。在 end.h 中 NET_PROTOCOL 结构定义如下：

```
typedef struct net_protocol
{
    NODE node; /* How we stay in a list. */
    char name[32]; /* String name for this protocol. */
    long type; /* Protocol type from RFC 1700 */
    int flags; /* Is protocol in a promiscuous mode? */
    BOOL (*stackRcvRtn) (void *, long, M_BLK_ID, M_BLK_ID, void*);
    /* The routine to call when we get */
    /* a packet. */
    STATUS (*stackShutdownRtn) (void*, void*);
    /* The routine to call to shutdown */
    /* the protocol stack. */
    STATUS (*stackTxRestartRtn) (void*, void*);
    /* Callback for restarting on blocked tx. */
    void (*stackErrorRtn) (END_OBJ*, END_ERR*, void*);
    /* Callback for device errors. */
    void* pSpare; /* Spare pointer that can be passed to */
    /* the protocol. */
} NET_PROTOCOL;
```

2. 向协议传送一数据包：stackRcvRtn()

每一协议均应为 MUX 提供一个使它可以为协议传递数据包的程序。此程序格式为：

```

void stackRcvRtn
(
void* pCookie, /* returned by muxBind() call */
long type, /* protocol type from RFC 1700 */
M_BLK_ID pNetBuff, /* packet with link level info */
LL_HDR_INFO* pLinkHdr, /* link-level header info structure */
void* pSpare /* a void* the protocol can use to get info */
/* on receive. This was passed to muxBind(). */
)

```

协议必须默认 `stackRcvRtn()` 为空。因此，函数不返回值。

参数为：

- `pCookie`。调用 `muxBind()` 后的返回指针。此指针用来识别 MUX 将协议分配到的那个设备。
- `type`。从 RFC1700 或 SAP 中获得的协议类型。
- `pNetBuff`。一个指向包含包数据和连接级信息的 `mBlk` 结构的指针。
- `pLinkHdr`。返回包含首部信息的一个 `LL_HDR_INFO` 结构，首部信息取决于 END 执行的特定数据连接层。详情请见“跟踪连接级信息：LL_HDR_INFO”。
- `pSpare`。一个指向传递给 MUX 的多余信息（如果有的话）的指针，MUX 通过调用 `muxBind()` 的 `pSpare` 参数得到。此信息通过每个 `receiveRtn` 调用返回到协议。这种信息的使用是可选的，也是基于特定协议的。

3. 向协议返回错误信息：stackError()

MUX 使用 `stackError()` 程序将设备中的错误信息返回到协议中。此程序代码会适当响应所有可能的错误信息。`stackError()` 程序源码如下：

```

void stackError
(
END_OBJ* pEnd, /* pointer to END_OBJ */
END_ERR* pError, /* pointer to END_ERR */
void* pSpare /* pointer to protocol private data passed in muxBind */
)

```

须默认 `stackShutdownRtn()` 的返回值为空。因此，此程序没有返回函数值。参数为：

- `pEnd`。此程序的 `muxBind()` 函数值的返回指针。此指针表明 MUX 将协议分配给的那个设备。
- `pError`。指向 `END_ERR` 结构的指针，在 `end.h` 中定义如下：

```

typedef struct end_err
{

```

```

INT32 errCode; /* error code, see above */
char* pMesg; /* NULL-terminated error message, can be NULL */
void* pSpare; /* pointer to user defined data, can be NULL */
} END_ERR;

```

在 `stackError()` 程序的代码中, 用户必须对在 `errCode` 成员中存储的标志予以适当响应。Wind River 预留了 `errCode` 的低 16 位比特作为它自身的错误信息, 定义如下:

<code>END_ERR_INFO</code>	只是信息错误。
<code>END_ERR_WARN</code>	非致命错误。
<code>END_ERR_RESET</code>	使设备复位但可以恢复的错误。
<code>END_ERR_DOWN</code>	使设备瘫痪的错误。设备不能再传送或接收数据包。
<code>END_ERR_UP</code>	设备关闭但现在已经重新启动, 可以接收和传递数据包。

`errCode` 成员的高 16 位为用户使用。可使用这些比特来编写任何需要在驱动程序和协议之间传送的错误信息。

- `pSpare`。指向特定协议数据的指针。最初, 当调用 `muxBind()` 时, 协议将此数据传递给 MUX。此数据是可选择的, 也是基于特定协议的。

4. 关闭一个协议: `stackShutdownRtn()`

MUX 使用 `stackShutdownRtn()` 来关闭一个协议。在这个程序中, 用户必须顺序关闭所使用协议。 `stackShutdownRtn()` 格式如下:

```

void stackShutdownRtn
(
void* pCookie /* Returned by muxBind() call. */
void* pSpare /* a void* that can be used by the protocol to get
/* info on receive. This was passed to muxBind(). */
)

```

默认 `stackShutdownRtn()` 的返回为空。因此, 此程序无返回函数值。

参数为:

- `pCookie`。指向此协议 `muxBind()` 函数值的指针。这个指针用来识别 MUX 为哪个设备分配协议。
- `pSpare`。作为 `pSpare` 传递给 `muxBind()` 的指针。

5. 重新启动协议: `stackTxRestartRtn()`

MUX 使用 `stackTxRestartRtn()` 来重新启动那些由于设备没有资源而中断传送的协议。在高流量情况下, `muxSend()` 返回 `END_ERR_BLOCK`。这种错误返回表明设备已经没有传递更多数据包所需要的资源, 并且说明在传送更多数据包之前, 协议需要等待。

当设备确定其已含有足够资源以开始重新传送时, 调用 `muxTxRestart()` 函数, 从而调

用了协议的 `stackTxRestartRtn()`。

`stackTxRestartRtn()` 须遵循以下格式：

```
void muxTxRestart
(
void* pCookie /* Returned by muxBind() call. */
)
```

参数为：

- `pCookie`。指向此协议 `mucBind()` 函数值的指针。此指针用来识别 MUX 为哪个设备分配协议。

H.6.2 网络层到数据连接层的地址工程

MUX 为连接网络层到数据连接层的地址工程函数提供了若干函数。分解网络层地址到数据连接层地址通常通过独立协议来完成。对于以太网环境下的多数 IP 来说，是通过 ARP（地址分析协议）来实现的。

通过使用 MUX，任何协议/数据链均可以注册其自身地址分析函数。这些函数由以下两个程序添加或删除：

```
STATUS muxAddrResFuncAdd
(
long ifType, /* Media interface type from m2Lib.h */
long protocol, /* Protocol type from RFC 1700 */
FUNCPTR addrResFunc /* Function to call. */
)
STATUS muxAddrResFuncDel
(
long ifType, /* Media interface type from m2Lib.h */
long protocol /* Protocol type from RFC 1700 */
)
```

这些函数添加或删除地址工程程序。程序的自变量负责协议复写器的确定。当前 Wind River 所提供的惟一地址分析程序为 `arpresolve()`。

为了找出一个特定网络/数据链接对所使用的地址分析程序只需调用以下程序：

```
FUNCPTR muxAddrResFuncGet
(
long ifType, /* ifType from m2Lib.h */
```

```
long protocol /* protocol from RFC 1700 */
}
```

此程序返回一个指向某一特定函数的指针，此函数用于解决网络协议的第二自变量数据链接地址。

H.7 将一个 END 驱动程序从 RFC1213 转化到 RFC2233

若要转化一个 END 驱动程序使之使用 RFC2233，执行以下各步：

- (1) 修改 END 的装载程序，在 END_OBJ 中建立 RFC2233 类型的 MIB。
- (2) 修改 END 的卸载程序，从 END_OBJ 中删除 RFC2233 类型的 MIB。
- (3) 改变硬件地址宏来指向物理地址的新位置。
- (4) 在 ioctl 程序中添加 EIOCGMIB2233。

(5) 用 RFC2233 接口 API 替代旧的 RFC1213 接口 API；然后删除旧 RFC1213 接口 API。

所有的 MIB 调用均能在一有效位置上，这一点是很重要的。换句话说，在确定数据包可以被成功接收或发送之前不要记入一数据包的成功接收和发送。

特别注意保证所有的失败情况都要有适当记录。

1. 第一步：修改 END 的装载程序，在 END_OBJ 中建立 RFC2233 类型的 MIB

在 xxxEndLoad() 程序中初始化 MIB-II 接口（对于 RFC2233）。例如：

```
pDrvCtrl->endObj.pMib2Tbl = m2IfAlloc(M2_ifType_ethernet_cemacd,
(UINT8*) enetAddr, 6,
ETHERMTU, speed,
DEV_NAME, pDrvCtrl->unit);
if (pDrvCtrl->endObj.pMib2Tbl == NULL)
{
logMsg ("%s%d - MIB-II initializations failed\n",
DEV_NAME, pDrvCtrl->unit, 0, 0, 0, 0);
goto errorExit;
}
/*
* Set the RFC2233 flag bit in the END object flags field and
* install the counter update routines.
*/
```

```

m2IfPktCountRtnInstall(pDrvCtrl->endObj.pMib2Tbl,
m2If8023PacketCount);
/*
 * Make a copy of the data in the mib2Tbl struct as well. We do this
 * mainly for backward compatibility issues. There might be some
 * code that references the END pointer and does lookups on the mib2Tbl,
 * which causes all sorts of problems.
 */
bcopy ((char *)&pDrvCtrl->endObj.pMib2Tbl->m2Data.mibIfTbl,
(char *)&pDrvCtrl->endObj.mib2Tbl, sizeof (M2_INTERFACETBL));/*
Mark the device ready */
END_OBJ_READY (&pDrvCtrl->endObj,
IFF_NOTRAILERS | IFF_MULTICAST | IFF_BROADCAST |
END_MIB_2233);

```

2. 第二步：修改 END 的卸载程序，从 END_OBJ 中删除 RFC2233 类型的 MIB

在 xxxEndUnload() 程序中添加 MIB-II 自由程序接口（对于 RFC2233）。例如：

```

m2IfFree(pDrvCtrl->endObj.pMib2Tbl);
pDrvCtrl->endObj.pMib2Tbl = NULL;

```

3. 第三步：改变硬件地址宏来指向物理地址的新位置

硬件地址宏的名称通常定义为 END_HADDR(pEnd)，但有时也会用表明实际驱动程序的前缀来命名，如在 fei82557End 驱动程序中，它被定义为 FEI_HADDR(pEnd)。

设定硬件地址宏指向在 RFC2233 类型的 MIB 中所存储的物理地址。对于 RFC2233 类型的 MIB，一个以太网设备的物理地址储存在 END_OBJ 的 pMib2Tbl 中。例如：

```

#define END_HADDR(pEnd) \
((pEnd)->pMib2Tbl->m2Data.mibIfTbl.ifPhysAddress.phyAddress)

```

4. 第四步：在 ioctl 程序中添加 EIOCGMIB2233

例如：

```

/* New RFC 2233 mib2 interface */
case EIOCGMIB2233:
if ((data == NULL) || (pEndObj->pMib2Tbl == NULL))
error = EINVAL;
else

```



```
*((M2_ID **)data) = pEndObj->pMib2Tbl;
break;
```

5. 第五步：用 RFC2233 接口 API 替代旧的 RFC1213 接口 API；然后删除旧 RFC1213 接口 API

- RFC1213 接口 API。旧的 RFC1213 接口使用 END_ERR_ADD 宏来升级数据包计数器并进行错误情况计数。

(1) 替代所有 END_ERR_ADD 调用的常量。替换后删除。例如：

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

(2) 替代删除的 RFC1213 接口。

- RFC2233 接口 API。RFC2233 接口 API 包含 m2IfGenericPacketCount()

m2IfCounterUpdate()。通过在 END_OBJ 的 pMib2Tbl 中存储的指针访问这些程序。指向 m2IfGenericPacketCount() 的指针通过 pDrvCtrl-> endObj.pMib2Tbl->m2PktCountRtn 获得。指向 m2IfCounterUpdate() 的指针通过 pDrvCtrl->endObj.pMib2Tbl-> m2CtrUpdateRtn 获得。

这些程序的 API 如下：

m2If8023PacketCount。802.3 设备的接口数据包计数器增量。

此函数用来升级数据包的基本接口计数器。Ctrl 自变量规定了数据包是被发送还是接收 (M2_PACKET_IN 或 M2_PACKET_OUT)。此函数只适用于 802.3 设备，因为它读取以太网数据包格式。有下列计数器被升级：

ifInOctets	ifInUcastPkts
ifInNUcastPkts	ifOutOctets
ifOutUcastPkts	ifOutNUcastPkts
ifInMulticastPkts	ifInBroadcastPkts
ifOutMulticastPkts	ifOutBroadcastPkts
ifHCInOctets	ifHCInUcastPkts
ifHCOctets	ifHCOctets
ifHCInMulticastPkts	ifHCInBroadcastPkts
ifHCOctetsMulticastPkts	ifHCOctetsBroadcastPkts
ifCounterDiscontinuityTime	

此函数应在完成 netMblkToBufCopy() 函数后立即调用。输出缓冲器的起始 6 个字节包含目标 MAC 地址；接下来的 6 个字节必须包含源 MAC 地址。

MAC 地址类型（即广播、多点传送和点对点）由以下地址决定：

广播地址：	ff:ff:ff:ff:ff:ff
多点传送地址：	设定第一个比特
点对点地址：	任何不符合以上情况的地址

返回值: 若 M2_ID 为空或 ctrl 为无效返回 ERROR, 若计数器升级完毕返回 OK。

```
STATUS m2If8023PacketCount
(
    M2_ID * pId, /* The pointer to the device M2_ID object */
    UINT ctrl, /* Update In or Out counters */
    UCHAR * pPkt, /* The incoming/outgoing packet */
    ULONG pktLen /* Length of the packet */
)
```

m2IfCounterUpdate. 增加接口计数器。

此函数用来直接升级接口计数器。计数器是由 ctrlId 规定的且所增加的数量由 value 规定。

若计数器计数, 和当前系统一起升级 ifCounterDiscontinuityTime。

返回值: 若 M2_ID 为空返回 ERROR, 若计数器升级完毕返回 OK。

```
STATUS m2IfCounterUpdate
(
    M2_ID * pId, /* The pointer to the device M2_ID object */
    UINT ctrlId, /* Counter to update */
    ULONG value /* Amount to update the counter by */
)
```

从 RFC1213 到 RFC2233 的代码转换:

(1) 发送程序。

在 Send() 和 PollSend() 程序中, 为输出数据包添加 RFC2233MIB-II 计数器。

例如, 改变一个 Send() 程序如下:

```
if (pDrvCtrl->endObj.pMib2Tbl != NULL)
{
    pDrvCtrl->endObj.pMib2Tbl->m2PktCountRtn(pDrvCtrl->endObj.pMib2Tbl,
    M2_PACKET_OUT, pEnetHdr,
    len);
}
```

例如, 改变一个 PollSend 程序如下:

```
if (pDrvCtrl->endObj.pMib2Tbl != NULL)
{
    pDrvCtrl->endObj.pMib2Tbl->m2PktCountRtn(pDrvCtrl->endObj.pMib2Tbl,
```



```
M2_PACKET_OUT, pEnetHdr,  
len);  
}
```

(2) 接收程序。

在 `Reveive()` 和 `PollReceive()` 程序中, 为即将到来的数据包添加一个 RFC2233MIB-II 计数器升级。

例如, 改变一个 `Receive()` 程序如下:

```
if (pDrvCtrl->endObj.pMib2Tbl != NULL)  
{  
pDrvCtrl->endObj.pMib2Tbl->m2PktCountRtn(pDrvCtrl->endObj.pMib2Tbl,  
M2_PACKET_IN,  
pMblk->mBlkHdr.mData,  
pMblk->mBlkHdr.mLen);  
}
```

例如, 改变一个 `PollReceive()` 程序如下:

```
if (pDrvCtrl->endObj.pMib2Tbl != NULL)  
{  
pDrvCtrl->endObj.pMib2Tbl->m2PktCountRtn(pDrvCtrl->endObj.pMib2Tbl,  
M2_PACKET_IN,  
pMblk->mBlkHdr.mData,  
pMblk->mBlkHdr.mLen);  
}
```

6. 第六步: 特别注意保证所有的失败情况都要有适当记录

RFC2233 接口提供了 `m2IfCounterUpdate()` 程序来存储所有与失败情况有关的多种计数器。所有失败情况多被认为是错误。然而, 通常将这些情况分为两类。设备返回一个错误状态不是由于没有完成所需任务就是因为由于缺乏有效资源驱动程序不能够处理数据包。

进一步划分设备失败情况可分为只发生错误和发生错误即丢掉两种, 这取决于是否要将发生错误的数据包丢掉。若不丢掉数据包, 则失败只是错误; 若丢掉数据包, 则失败不仅为错误而且需要被去除。大多数设备同时含有两种类型。

若驱动器收到的数据包接收后被破坏了, 这只构成一个错误。然而若由于缺乏资源, 驱动程序不能处理一个完好的数据包, 则它不仅是一个错误而且也应被丢掉。

RFC2233 接口拥有负责记录即将接受的和要输出的错误和丢弃的计数器。API 通过提供与指定的每一个特定计数器的相应标志来选择计数器。这个标志作为第二自变量传送到 `m2IfCounterUpdate()` 中。

相关的标志为:

M2_ctrlId_ifInDiscards

M2_ctrlId_ifInErrors

M2_ctrlId_ifOutDiscards

M2_ctrlId_ifOutErrors

例如:

```
/* New RFC 2233 mib2 interface */  
if (pDrvCtrl->endObj.pMib2Tbl != NULL)  
{  
    pDrvCtrl->endObj.pMib2Tbl->m2CtrUpdateRtn(pDrvCtrl->endObj.pMib2Tbl,  
    M2_ctrlId_ifInErrors, 1);  
    pDrvCtrl->endObj.pMib2Tbl->m2CtrUpdateRtn(pDrvCtrl->endObj.pMib2Tbl,  
    M2_ctrlId_ifInDiscards, 1);  
}
```

附录 I 编写 SCSI-2 设备驱动程序

I.1 概 述

VxWorks SCSI-2 子系统包括下列组件：

- SCSI 函数库，一个独立于系统体系结构的组件；
- SCSI 控制器驱动，一个指定体系结构的组件；
- SCSI-2 子系统初始化代码，一个指定目标板的组件。

在能够扩展 SCSI 库的功能或者添加新的 SCSI 控制器驱动之前，必须要首先理解这些组件中的每一个的基本功能。为了帮助获得对这些组件的理解，这一章描述了各种 SCSI 模块的总体布局，讨论了 SCSI 库的内容（以及它们使用 SCSI 库控制器驱动程序的接口），并且描述了开发指定控制器 SCSI 驱动程序的步骤。

关于 I/O 系统和 SCSI 库之间接口的内容，以及在 VxWorks 中配置 SCSI 外围设备，请参见《VxWorks 程序员指南》：I/O 系统。



注意：在这一章中，术语 SCSI 在所有情况下都是指 SCSI-2。在本章中描述的 SCSI 库接口和 SCSI 控制器驱动只涉及 SCSI-2。VxWorks 仅仅对 SCSI-1 提供了有限的支持，VxWorks 将会在最终取消对 SCSI-1 的所有支持。

I.2 SCSI 总括

这一部分讲述了各种 SCSI 模块之间的关系，介绍了不同的 SCSI 对象和数据结构，并且告诉如何来产生 SCSI 命令。

1. SCSI 模块的结构

图 I-1 说明了所有的 SCSI 库模块、它们之间的关系和一些典型的驱动程序。SCSI 库包含了不同的数据结构，重要的数据结构以及它们之间的关系将在后面的内容中进行描述。一般的数据结构的设计是面向对象的，数据结构代表了真实的和抽象的 SCSI 对象，例如外围设备、控制器和块设备。

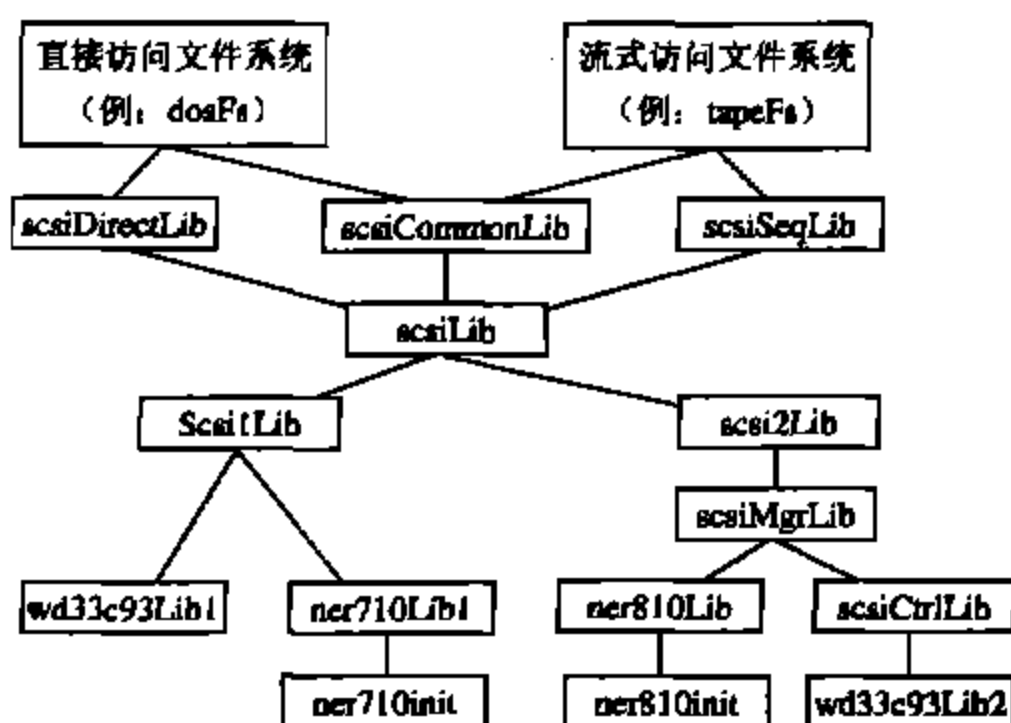


图 I-1 SCSI 模块结构

2. SCSI 对象和数据结构

图 I-2 说明了各种物理和逻辑 SCSI 对象之间的关系以及相应的数据结构。

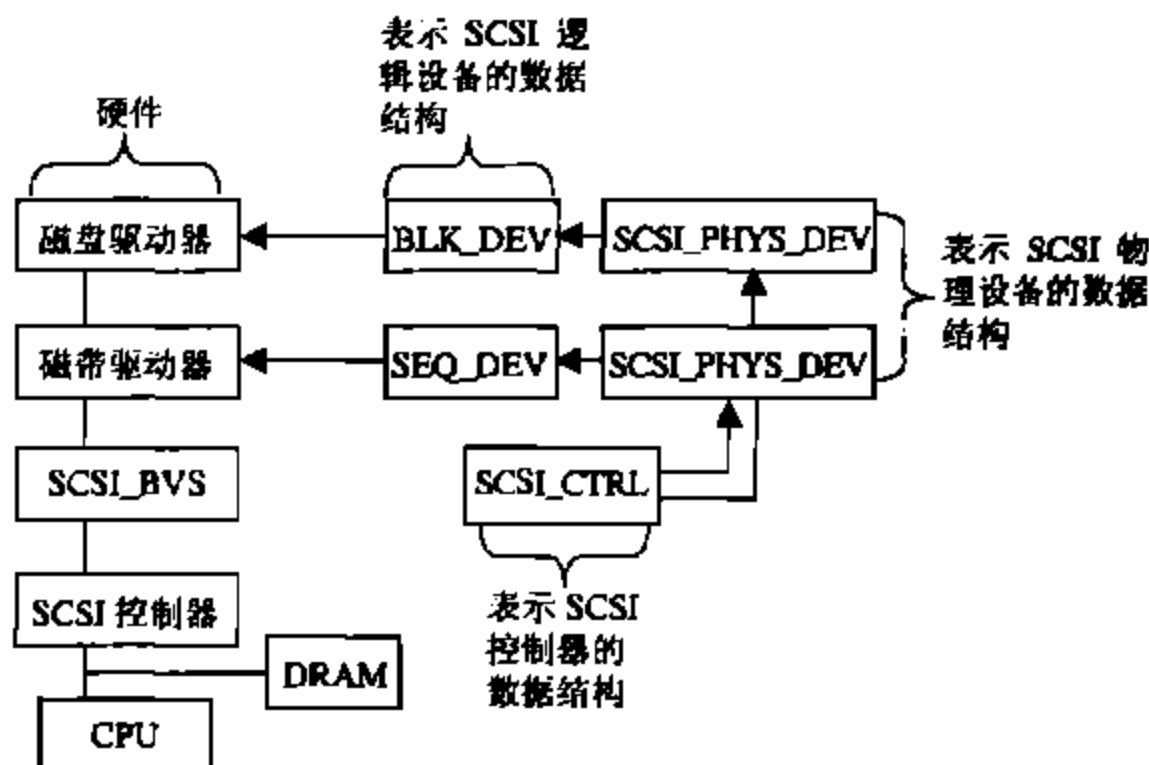


图 I-2 SCSI 设备和数据结构的关系

图 I-3 更加具体地描述了这些数据结构和它们之间的关系。

- **SCSI_CTRL**。这个结构包含了所有的物理设备和所有被分配 SCSI 线程。
- **SCSI_THREAD**。每一个线程由一个动态的数据结构来代表，这个数据结构被 `scsi2Lib`、`scsiMgrLib` 和设备驱动程序中的不同层次控制。它包含了 `SCSI_TRANSACTION` 和其他的有关线程状态的信息。
- **SCSI_TRANSACTION**。每一个来自 I/O 系统的 SCSI 命令都被解释成这些结构中

的一种，它们包括了 SCSI 命令解释块和所有需要的指针地址。

- SCSI_PHYS_DEV。这个结构包含了关于有效的逻辑设备的信息以及有关各种线程的信息。
- SEQ_DEV。这个结构代表了流逻辑设备，例如磁带驱动器。
- BLK_DEV。这个结构代表了一个块设备，例如磁盘驱动器。

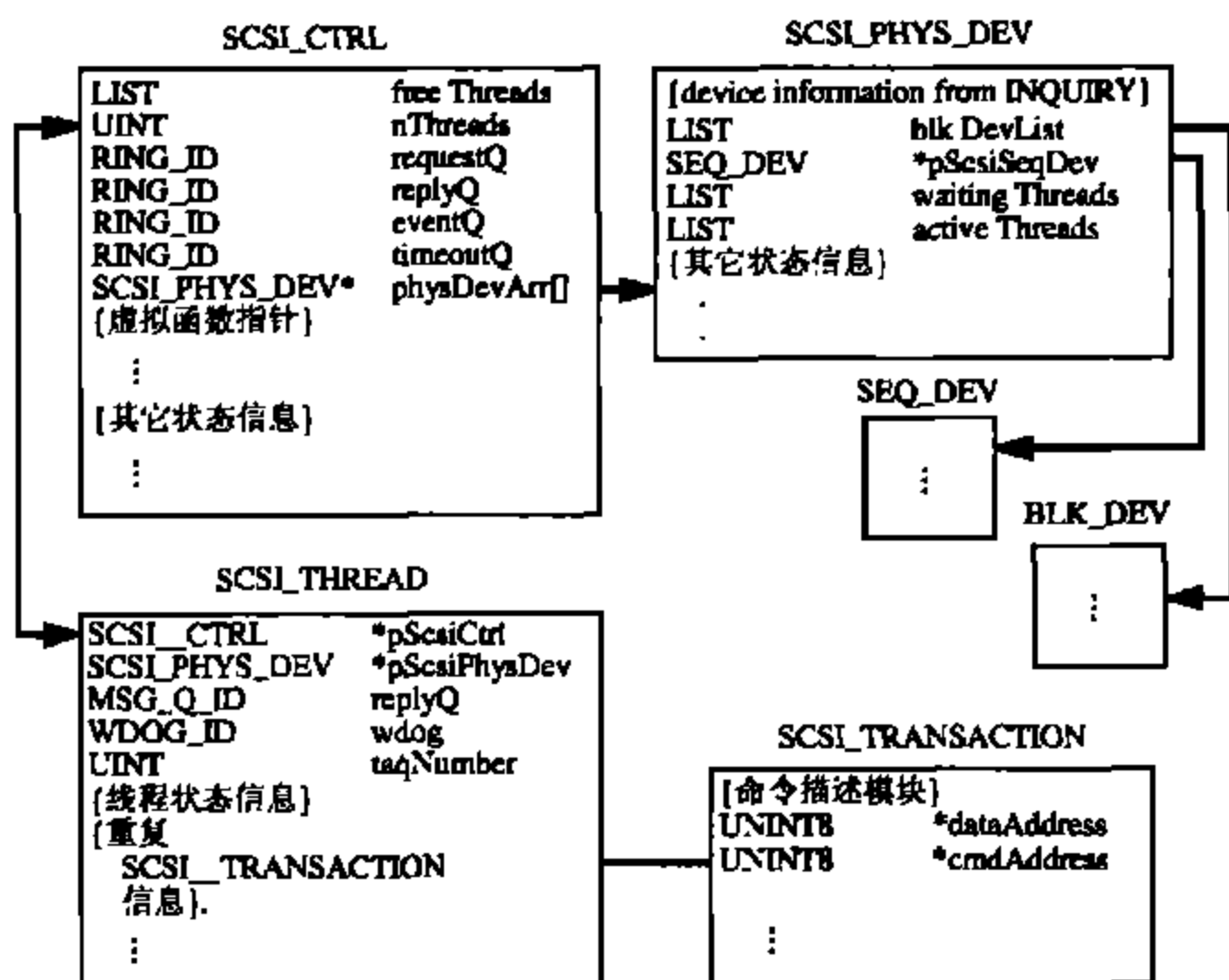


图 I-3 指定控制器和驱动器数据结构

3. 生成 SCSI 命令

在 SCSI 库内部，SCSI 命令都是以一种相似方式工作。命令所需要的所有信息都是通过传送适当的参数来传递。这些命令首先使用指向所有需要的数据的指针建立一个 SCSI 命令描述模块，并且在 SCSI_TRANSACTION 结构中储存这个模块，然后调用 `scsiTransact()` 程序，将结构 SCSI_TRANSACTION 和 SCSI_PHYS_DEV 传送给它。

`scsiTransact()` 函数是 `scsi2Lib` 库中的一段一般性程序，它处理了在 `scsiDirectLib`、`scsiCommonLib` 和 `scsiSeqLib` 库中所有 SCSI 命令产生的过程。下面这个范例可以用于扩展 SCSI 库对其他设备类库（`scsiXXXLib`）的支持。

```
STATUS scsiXxxCmd
(
    char * buf
    SCSI_PHYS_DEV * pScsiPhysDev
```

```

;
)
SCSI_COMMAND xxxCommand;
SCSI_TRANSACTION scsiXaction;
...
/* build a SCSI command descriptor block in xxxComand */
...
scsiXaction.cmdAddress = xxxComand;
scsiXaction.dataAddress = buf;
scsiXaction.dataLength = strlen(buf);
...
return ((*pScsiPhysDev->pScsiCtrl->scsiTransact)
(pScsiPhysDev, &scsiXaction));
...

```

I.3 SCSI 库

这个部分描述了下面这些库：

- SCSI 管理器 (scsiMgrLib)；
- SCSI 控制器库 (scsiCtrlLib)；
- SCSI 直接访问库 (scsiDirectLib)；
- SCSI 顺序访问库 (scsiSeqLib)；
- SCSI 一般性访问库 (scsiCommonLib)。

在这一部分的最后，简要的讨论了在执行 SCSI 命令时 VxWorks 通常是如何处理的。

1.3.1 SCSI 管理器 (scsiMgrLib)

SCSI 管理器作为 VxWorks 内部的一个任务来运行，每一个 SCSI 控制器都有一个 SCSI 管理器，它负责管理在 VxWorks 任务和 SCSI 控制器之间的所有的 SCSI 交互处理。任何 VxWorks 任务都可以从 SCSI 外围设备请求服务，SCSI 总线是一个可以共享的重要资源，它需要多任务支持和同步性。

为了提高性能和系统的效率，SCSI 管理器将所有的 SCSI 通信控制在操作系统内部。SCSI 通信包括通过 VxWorks 任务请求 SCSI 服务。这些请求相当于来自 SCSI 总线的同步事件，它们包括 SCSI 重新连接，SCSI 连接超时设定和 SCSI 请求响应。这个工作由 SCSI 线程管理，它是指定的 SCSI 库的抽象部分。一个 SCSI 线程被分配到 SCSI 工作的每一个单元，换句话说，对每一个 SCSI 请求都分配一个 SCSI 线程。

每个 SCSI 线程是在调用 VxWorks 任务的上下文环境中创建的。这个线程由 SCSI 管理器来管理，此时调用 VxWorks 任务保持在阻塞状态。当 SCSI 线程完成时，VxWorks 任务就解除阻塞状态，并且 SCSI 线程被删除。

每一 SCSI 线程拥有它们自己的上下文和状态变量，它们由 SCSI 库和控制器驱动程序来操作。SCSI 线程能够最大限度的在任何时间内执行。除了管理 SCSI 线程的状态信息外，SCSI 管理器还负责为这些 SCSI 线程安排时间。

当同时存在多个线程时，不同的线程可以处于不同的状态来代表不同的请求。一个 SCSI 线程能够代表一个新的服务请求、连接超时、服务完成或者来自 SCSI 总线的事件。由于服务请求是通过 VxWorks 任务服从于 SCSI 管理器的，因此相关的线程必须基于优先级来执行，或者当它们的优先级相同时基于先来先服务的原则来执行。

当多个线程的活动都是合法的时候，SCSI 管理器要遵循一个严格的处理层次。同步的总线事件具有最高的优先级，在任何其他类型 SCSI 线程之前执行，处理的顺序是：事件、超时设定、请求、和最终的响应。SCSI 管理器对在开发中产生的来自 SCSI 总线的请求的激活状态和事件的同步出现之间相互竞争资源的情况进行了处理。

一旦选择了一个适当的 SCSI 线程来执行，SCSI 管理器就分派此线程，实际上是由指定控制器的驱动程序来执行的。

1. 限制情况

SCSI 管理器使用标准的 VxWorks 环形缓冲器来管理 SCSI 请求，使用环形缓冲器会更加高速和有效。SCSI 工作的总量可以按被分配的环形缓冲的大小来排序。SCSI 管理器同样也有一些限制，例如最大允许的线程数 (`scsiMaxNumThreads`)，可以被放入 SCSI 管理器请求队列中的来自 VxWorks 任务的 SCSI 请求的最大个数 (`scsiMgrRequestQSize`)，可以放入 SCSI 管理器事件队列中的 SCSI 总线事件的最大个数 (`scsiMgrEventQSize`)，可以放入回复队列中的最多的回复个数 (`scsiMgrReplyQSize`)，可以放入超时设定队列中的超时设定的最大个数 (`scsiMgrTimeoutQSize`) 以及超时设定的数值。

2. 配置情况

我们可以通过调节环形缓冲器的大小和 SCSI 线程的个数来对指定的环境进行优化。然而，在多数情况下，默认值是比较令人满意的。参数 `scsiMaxNumThreads`、`scsiMgrRequestQsize`、`scsiMgrReplyQSize`、`scsiMgrEventQSize`、`scsiMgrTimeoutQSize` 被定义为 SCSI 库内部的全局变量，并且在 `scsiLib.h` 文件中被分配了默认值。这些值可以在调用驱动程序的 `xxxCtrlInit()` 函数之前在 BSP 函数 `sysScsiInit()` 中重新设定。然后，当驱动程序的 `xxxCtrlInit()` 调用函数 `scsiCtrlInit()` 时，新的参数就会被用于数据结构的分配。

任务 `scsiMgr` 的名字、优先级和堆栈的大小也能够通过控制器驱动程序 `xxxCtrlCreate()` 来达到用户化需求，默认值在 `scsiLib.h` 文件中指定。例如，默认的任务名称 `SCSI_DEF_TASK_NAME` 是 `tScsiTask`，默认的优先级 `SCSI_DEF_TASK_PRIORITY` 是 5，默认的堆栈大小 `SCSI_EDF_TASK_STACK_SIZE` 是 4000。



注意：可能的 VxWorks SCSI 任务的数目越多，所需要的堆栈空间就越大。
当设置 SCSI 管理器时，对堆栈大小参数的设置要多加考虑。

1.3.2 SCSI 控制器库 (scsiCtrlLib)

SCSI 控制器库是为老一代的 SCSI-2 控制器设计的，老一代控制器需要较高层的软件来处理协议状态机（和传输）。这些基本的 SCSI 控制器驱动程序（那些需要使用由 SCSI 库提供的 SCSI 状态机的）使用 SCSI 控制器库。更加先进的 SCSI 控制器允许这样的协议状态机在 SCSI 控制器层执行。这样有效地减少了每一个 I/O 过程对 CPU 产生的 SCSI 中断个数，从而提高了性能。

在 SCSI 库和这类驱动器的控制器驱动程序之间已经有了较成熟的接口，这个接口在“1.4 SCSI 驱动程序编程接口部分”进行了介绍。

1.3.3 SCSI 直接访问库 (scsiDirectLib)

SCSI 直接访问库 `scsiDirectLib` 封装了所有的执行在 SCSI ANSI 规范 I 中定义的 SCSI 直接访问命令的函数。除了所有的直接访问函数之外，`scsiDirectLib` 还提供了其他一些函数，这些函数为 SCSI 直接访问周边设备提供了 BLK_EDV 映象。

1.3.4 SCSI 顺序访问库 (scsiSeqLib)

SCSI 顺序访问库 `scsiSeqLib` 提供了所有的执行在 SCSI ANSI 规范 I 中定义的 SCSI 顺序访问命令的函数。一些可选的特性也是能够实现的。这个库中还提供了用于控制 SEQ_DEV 映象的函数。

1.3.5 SCSI 一般访问库 (scsiCommonLib)

在 SCSI 一般访问库中提供了对所有的 SCSI 外围设备类型通用的 SCSI 命令。这些命令在 SCSI ANSI 规范 I 中进行了描述。对这些命令的编程接口可以在相关的访问入口中或者通过查找头文件 `scsi2Lib.h` 找到。

1.3.6 一个可执行的例子

让我们一起简单地看一下当 VxWorks 任务通过调用 SCSI 库函数例如 `scsiInquiry()` 来请

求 SCSI 服务时,会发生什么情况。由于我们假设了一种 SCSI-2 配置,首先调用 `scsi2Inquiry()` 函数,它又依次调用了 `scsiTransact()` 函数(参见“生成 SCSI 命令”)。`scsiTransact()`调用了 `scsiCommand()`函数,这个函数分配了一个 SCSI 线程,执行这个线程,然后删除它。

线程的执行通过 `scsiThreadExecute()`来通知 SCSI 管理器一个新的线程在执行,然后接下来阻塞消息队列中的 VxWorks 任务,直到接受到一个响应为止。这里是 VxWorks 任务被阻塞和 SCSI 管理器被激活开始执行一个新的线程同时管理其他需要处理的线程的一个转折点。

在 SCSI 线程执行完成并且得到了一个响应后,调用 VxWorks 任务被解除阻塞,并且与那个任务相关的 SCSI 线程最终被删除。

I.4 SCSI 驱动程序编程接口

为了更好的解释控制器驱动程序和两种类型 SCSI 控制器(基本型和高级型)的 SCSI 库之间的接口,这一部分对每种类型的驱动器分别进行了讨论。与 SCSI 库和控制器驱动程序之间的编程接口一起,我们还提供了一个驱动程序框架。控制器驱动程序提供了对所有硬件寄存器的访问以及指定控制器的功能。为了更加简单,这里没有对这些访问和特定控制器信息进行说明。驱动程序模板的目的是示例了整体的结构以及驱动程序、SCSI 库和 BSP 之间的程序接口。

I.4.1 基本的 SCSI 控制器驱动程序

这一部分介绍了基本的编程接口 SCSI 控制器和 SCSI 库。然后,在这一部分我们还提供了在编写自己的 SCSI 控制器驱动程序时应该使用的驱动程序模板。

1. 编程接口

在任何基本 SCSI 控制器的控制器驱动程序和 SCSI 库之间都存在严格定义的编程接口。每一个基本的控制器驱动程序都必须向 SCSI 库提供下列函数:

(1) `xxxDevSelect()`。

这个函数使用已经声明的 `Attention (ATN)` 信号来选择一个 SCSI 外围设备。

(2) `xxxInfoXfer()`。

所有的信息传输部分都由这个函数来处理,包括 `DATA_IN`、`DATA_OUT`、`MSG_IN`、`MSG_OUT`、和 `STATUS`。

(3) `xxxXferParamsQuery()`。

这个函数对同步数据传输参数进行更新,以便配合驱动器的能力,并且返回最优的同步偏移量和周期。

(4) `xxxXferParamsSet()`。

这个函数设置了 SCSI 控制器的同步数据传输参数。

(5) `xxxBusControl()`。

这个函数控制了控制器的一些 SCSI 总线，它必须复位 SCSI 总线，声明 ATN，或者取消 ACK。

相似地，控制器驱动程序调用了下列函数以便获得 SCSI 库服务。

(6) `scsiCtrlInit()`。

这个函数初始化了 SCSI 库的数据结构，每个 SCSI 控制器只调用它一次。

(7) `scsiMgrEventNotify()`。

这个函数用于通知 SCSI 管理器一个 SCSI 事件已经产生了，事件是在 `scsi2Lib.h` 文件中定义的。然而，控制器驱动程序可以定义更多的事件，而且事件也可以被驱动程序绑定起来。在这种情况下，在驱动程序初始化时 `SCSI_STRL` 区段 `scsiEventProc` 必须按照指定的驱动程序函数来设置。

2. 模板驱动程序

下面这个例子是基本的 SCSI 控制器驱动程序示例模板，没有任何特定硬件限制。这个驱动程序的结构是与任何其他 VxWorks 驱动程序都很相似。下面列出了包括的主要函数：

- `xxxCtrlCreate()` 函数，它是由位于 BSP 文件 `sysScsi.c` 中的 BSP 函数 `sysScsiInit()` 调用的。
- ISR 调用函数 `xxxIntr()`，它处理了所有的中断，说明了所产生的 SCSI 事件，并且通过 `scsiMgrEventNotify()` 函数将这些事件信息传送给 SCSI 管理器。

SCSI 库通过 `xxxDevSelect()` 函数和 `xxxInfoXfer()` 函数来指挥驱动程序，控制器驱动程序利用 `scsiMgrEventNotify()` 函数向库反馈信息。

例 I-1 基本 SCSI 控制器驱动程序。

```
/* xxxLib.c - XXX SCSI-Bus Interface Controller library (SCSI-2) */
/* Copyright 1989-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"
/*
modification history
-----
01a,12sep96,dds written
*/
/*
DESCRIPTION
This library contains part of the I/O driver for the XXX family of SCSI-2
Bus Interface Controllers (SBIC). It is designed to work with scsi2Lib.
The driver routines in this library depend on the SCSI-2 ANSI specification;
for general driver routines and for overall SBIC documentation, see xxxLib.
```

```

INCLUDE FILES
xxx.h
SEE ALSO: scsiLib, scsi2Lib,
.pG "I/O System"
*/
#include "vxWorks.h"
#include "drv/scsi/xxx.h"
typedef XXX_SCSI_CTRL SBIC; /* SBIC: SCSI Bus Interface Controller struct
*/
/* globals */
int xxxXferDoneSemOptions = SEM_Q_PRIORITY;
char *xxxScsiTaskName = SCSI_DEF_TASK_NAME;
IMPORT SCSI_CTRL *pSysScsiCtrl;
/*****
*
* xxxCtrlCreate - create and partially initialize a SCSI controller structure
*
* This routine creates a SCSI controller data structure and must be called
* before using a SCSI controller chip. It should be called once and only
* once for a specified SCSI controller. Since it allocates memory for a
* structure needed by all routines in xxxLib, it must be called before
* any other routines in the library.
* After calling this routine, at least one call to xxxCtrlInit() should
* be made before any SCSI transaction is initiated using the SCSI controller.
*
* RETURNS: A pointer to the SCSI controller structure, or NULL if memory
* is insufficient or parameters are invalid.
*/
XXX_SCSI_CTRL *xxxCtrlCreate
(
    FAST UINT8 *sbicBaseAdre, /* base address of the SBIC */
    int regOffset, /* address offset between SBIC registers */
    UINT clkPeriod, /* period of the SBIC clock (nsec) */
    FUNCPTR sysScsiBusReset, /* function to reset SCSI bus */
    int sysScsiResetArg, /* argument to pass to above function */
    UINT sysScsiDmaMaxBytes, /* maximum byte count using DMA */
    FUNCPTR sysScsiDmaStart, /* function to start SCSI DMA transfer */
    FUNCPTR sysScsiDmaAbort, /* function to abort SCSI DMA transfer */
    int sysScsiDmaArg /* argument to pass to above functions */
)
{
    FAST SBIC *pSbic; /* ptr to SBIC info */

```

```

/* calloc the controller info structure; return NULL if unable */
pSbic = (SBIC *) calloc (1, sizeof (SBIC))
/*
 * Set up sizes of event and thread structures. Must be done before
 * calling "scsiCtrlInit()".
 */
/* fill in driver-specific routines for scsiLib interface */
pSbic->scsiCtrl.scsiDevSelect = xxxDevSelect;
pSbic->scsiCtrl.scsiInfoXfer = xxxInfoXfer;
pSbic->scsiCtrl.scsiXferParamsQuery = xxxXferParamsQuery;
pSbic->scsiCtrl.scsiXferParamsSet = (FUNCPTR)xxxXferParamsSet;
/* Fill in driver specific variables for scsiLib interface */
pSbic->scsiCtrl.maxBytesPerXfer = sysScsiDmaMaxBytes;

/* fill in generic SCSI info for this controller */
xxxCtrlInit (&pSbic->scsiCtrl);
/* initialize SBIC info transfer synchronization semaphore */
if (semBInit (&pSbic->xferDoneSem, xxxXferDoneSemOptions, SEM_EMPTY)
== ERROR)
{
(void) free ((char *) pSbic);
return ((XXX_SCSI_CTRL *) NULL);
}
/* initialize state variables */
/* fill in board-specific SCSI bus reset and DMA xfer routines */
/* spawn SCSI manager - use generic code from "scsiLib.c" */
pSbic->scsiCtrl.scsiMgrId = taskSpawn (xxxTaskName,
xxxTaskPriority,
xxxTaskOptions,
xxxTaskStackSize,
(FUNCPTR) scsiMgr,
(int) pSbic,
0, 0, 0, 0, 0, 0, 0, 0, 0);
return (pSbic);
;
/*****
 *
 * xxxCtrlInit - initialize a SCSI controller structure
 *
 * After a SCSI controller structure is created with xxxCtrlCreate, but
 * before using the SCSI controller, it must be initialized by calling this
 * routine.

```

```

* It may be called more than once if desired. However, it should only be
* called while there is no activity on the SCSI interface.
*
* RETURNS: OK, or ERROR if out-of-range parameter(s).
*/
LOCAL STATUS xxxCtrlInit
(
FAST SBIC *pSbic, /* ptr to SBIC info */
FAST int scsiCtrlBusId, /* SCSI bus ID of this SBIC */
FAST UINT defaultSelTimeout /* default dev. select timeout (microsec) */
)
{
pSbic->scsiCtrl.scsiCtrlBusId = scsiCtrlBusId;
/* initialize the SBIC hardware */
xxxHwInit (pSbic);
return (OK);
}
/*****
*
* xxxHwInit - initialize the SCSI controller to a known state
*
* This routine puts the SCSI controller into a known quiescent state. It
* does not reset the SCSI bus (and any other devices thereon).
*/
LOCAL void xxxHwInit
(
SBIC *pSbic /* ptr to an SBIC structure */
)
{
/*
* Initialize the SCSI controller hardware registers and place the
* chip in a known quiescent state
*/
}
/*****
*
* xxxDevSelect - attempt to select a SCSI device
*
* RETURNS: OK (no error conditions)
*/
LOCAL STATUS xxxDevSelect
(

```

```

SCSI_CTRL *pScsiCtrl, /* ptr to SCSI controller info */
int devBusId, /* SCSI bus ID of device to select */
UINT selTimeOut, /* select t-o period (usec) */
UINT8 *msgBuf, /* ptr to identification message */
UINT msgLen /* maximum number of message bytes */
}
{
int lockKey; /* saved interrupt lock key */
lockKey = intLock ();
/* Select device */
intUnlock (lockKey);
}
/*****
*
* xxxXferParamsQuery - get (synchronous) transfer parameters
*
* Updates the synchronous transfer parameters suggested in the call to match
* the SCSI controller's capabilities. Transfer period is in SCSI unite
* (multiples of 4 ns).
*
* RETURNS: OK    I
*/
LOCAL STATUS xxxXferParamsQuery
(
SCSI_CTRL *pScsiCtrl, /* ptr to SBIC info */
UINT8 *pOffset, /* max REQ/ACK offset [in/out] */
UINT8 *pPeriod /* min transfer period [in/out] */
)
{
/* read offset and period values */
return (OK);
}
/*****
*
* xxxXferParamsSet - set transfer parameters
*
* Programs the SCSI controller to use the specified transfer parameters.
An
* offset of zero specifies asynchronous transfer (period is then irrelevant).
*
* RETURNS: OK if transfer parameters are OK, else ERROR.
*/

```



```

LOCAL STATUS xxxXferParamsSet
(
    SCSI_CTRL *pScsiCtrl, /* ptr to SBIC info */
    UINT8 offset, /* max REQ/ACK offset */
    UINT8 period /* min transfer period */
)
{
    /* set the appropriate SCSI controller registers */
    return (OK);
}

/*****
 *
 * xxxInfoXfer - transfer information bytes to/from target via SCSI bus
 *
 * Executes a "Transfer Info" command to read (write) bytes from (to) the
 * SCSI bus. If the transfer phase is DATA IN or DATA OUT and there is a
 * DMA routine available, DMA is used - otherwise it's a tight programmed
 * i/o loop.
 *
 * RETURNS: Number of bytes transferred across SCSI bus, or ERROR.
 */
LOCAL int xxxInfoXfer
(
    FAST SCSI_CTRL *pScsiCtrl, /* ptr to SCSI controller info */
    int phase, /* SCSI phase being transferred */
    FAST UINT8 *pBuf, /* ptr to byte buffer for i/o */
    UINT buflen /* number of bytes to be transferred */
)
{
    pSbic = (SBIC *) pScsiCtrl;
    /* Handle phase changes */
    /* Start DMA, if used, or programmed i/o loop to transfer data */
    /* Wait for transfer to complete: find out how many bytes transferred */
    semTake (&pSbic->xferDoneSem, WAIT_FOREVER);
    /*
     * If there are bytes left to be transferred return ERROR
     * If DMA is used for transfer do a SCSI DMA Abort
     */
    xxxXferCountGet (pSbic, &bytesLeft);
    return (buflen - bytesLeft);
}

/*****

```

```

*
* xxxXferCountSet - load the SCSI controller transfer counter with count.
*
* RETURNS: OK if count is in range 0 - 0xffffffff, otherwise ERROR.
*
*/
LOCAL STATUS xxxXferCountSet
{
    FAST SBIC *pSbic, /* ptr to SBIC info */
    FAST UINT count /* count value to load */
}
{
    /* set the appropriate SCSI controller registers */
}
/*****
*
* xxxXferCountGet - fetch the SCSI controller transfer count
*
* The value of the transfer counter is copied to *pCount.
*
*/
LOCAL void xxxXferCountGet
{
    FAST SBIC *pSbic, /* ptr to SBIC info */
    FAST UINT *pCount /* ptr to returned value */
}
{
    /* read the appropriate SCSI controller registers */
}
/*****
* I
* xxxCommand - write a command code to the SCSI controller Command Register
*
*/
LOCAL void xxxCommand
{
    SBIC *pSbic, /* ptr to SBIC info */
    UINT8 cmdCode /* new command code */
}
{
    /* set the appropriate SCSI controller registers */
}

```



```

/*****
*
* xxxIntr - interrupt service routine for the SCSI controller
*
*/
LOCAL void xxxIntr
{
    SBIC *pSbic /* ptr to SBIC info */
}
{
    SCSI_EVENT event;
    /* Check the SCSI status. Handle state transitions */
    switch (scsiStatus)
    {
        ...
        /* the list of event types is defined in scsi2Lib.h */
        use ...
        event.type = SCSI_EVENT_XFER_REQUEST;
        event.phase = busPhase;
        break;
        case ...
    }
    /* Synchronize with task-level code */
    semGive (&pSbic->xferDoneSem);
    /* Post event to SCSI manager for further processing */
    scsiMgrEventNotify ((SCSI_CTRL *)pSbic, &event, sizeof (event));
}
/*****
*
* xxxRegRead - Get the contents of a specified SCSI controller register
*
*/
LOCAL void xxxRegRead
{
    SBIC *pSbic, /* ptr to an SBIC structure */
    UINT8 regAddr, /* address of register to read */
    int *pDatum /* buffer for return value */
}
{
    /* read the appropriate SCSI controller registers */
}
/*****
*

```

```

* xxxRegWrite - write a value to a specified SCSI controller register
*
*/
LOCAL void xxxRegWrite
(
    SBIC *pSbic, /* ptr to an SBIC structure */
    UINT8 regAddr, /* address of register to write */
    UINT8 datum /* value to be written */
)
{
    /* write the appropriate SCSI controller registers */
}

```

1.4.2 高级 SCSI 控制器驱动程序

高级 SCSI 控制器在驱动程序内部结合了所有的低层的状态机功能, 这个功能代替了由 `scsiCtrl.Lib` 提供的功能。大多数高级 SCSI 控制器都具有它们自己的 SCSI I/O 处理器, 这样通过管理 SCSI 总线上的所有低层行为可以增强系统的性能, 例如相位变化和 DMA 数据传输。通常对 I/O 处理器的指令是机器语言指令, 它们是使用高级的汇编语言编写然后再编译产生机器指令。这些指令驻留于主 DRAM 区域, 并且使用 SCSI 程序计数器和一些间接寻址的形式来通过 I/O 处理器从 DRAM 中获取指令。

在高级 SCSI 控制器环境下, 在特定驱动程序结构中, 例如 `XXX_EVENT` (`XXX` 是指 SCSI 驱动程序模块前缀), 总是描述了额外的事件信息。在控制器驱动程序中有许多部分的线程管理程序, 在基本 SCSI 控制器驱动程序中就不是这样的。

1. 程序接口

高级 SCSI 控制器驱动程序和 SCSI 库之间的编程接口包含了被驱动程序调用的驱动程序和库函数所必须提供的函数。驱动程序不需要遵守这里使用的命名规则, 因为这些程序是使用在 `xxxCtrlCreate()` 中设置的函数指针来访问, 但是我们建议使用这个命名规则。驱动程序必须提供的函数 (或等价的) 是:

(1) `xxxEventProc()`^①。

这个函数被 SCSI 管理器调用, 来对事件进行分析, 并且做出适当的处理。

(2) `xxxThreadInit()`。

这个函数初始化了 SCSI 线程结构, 并且添加了由 `scsiThreadInit()` 提供之外还需要的指定驱动器的初始化代码。

^① 函数名中的 `xxx` 是指为自己的 SCSI 驱动程序模块分配的任意前缀。

(3) `xxxThreadActivate()`。

这个函数激活了一个 SCSI 连接，在 `SCSI_THREAD` 数据结构中设置合适的线程环境，并且为所有的寄存器设置适当的值。它可能会调用 SCSI 库中的程序和其他的驱动程序。

(4) `xxxThreadAbort()`。

如果线程事实上没有连接，这个函数不进行任何操作。如果线程确实连接好了，这个函数将发送一个 `ABORT TAG` 消息，从而使 SCSI 目标设备断开连接。

(5) `xxxBusControl()`。

这个函数控制了一些 SCSI 总线，它必须复位 SCSI 总线，声明 `ATN`，或者忽略 `ACK`。

(6) `xxxXferParamsQuery()`。

这个函数对同步数据传输参数进行更新以便配合驱动器的能力，并且返回最优的同步偏移量和周期。

(7) `xxxXferParamsSet()`。

这个函数设置了 SCSI 控制器的同步数据传输参数。

(8) `xxxWideXferParamsQuery()`。

这个函数对调用中的宽带数据传输参数进行更新，为了与 SCSI 控制器的相匹配。

(9) `xxxWideXferParamsSet()`。

这个函数设置了 SCSI 控制器的宽带数据传输参数。

高级控制器驱动程序同样使用了许多由 SCSI 库提供的工具。所有被 SCSI 控制器库调用的程序都可以被驱动程序调用。分析一下 SCSI 控制器库和 `scsi2Lib.h` 头文件，就可以发现对控制器驱动程序有效的所有函数。下面列出了典型的可以被驱动程序调用的函数，但这并不是全面的。

(10) `scsiCtrlInit()`。

这个函数初始化了 SCSI 库的数据结构，每个 SCSI 控制器只调用它一次。

(11) `scsiMgrEventNotify()`。

这个函数通知 SCSI 管理器在 SCSI 总线上产生了一个事件。

(12) `scsiWideXferNegotiate()`。

这个函数发起或者继续宽带数据传输流通。参见相关的访问入口和 `scsi2Lib.h` 文件来获得更详细的内容。它通常从 `xxxThreadActivate()` 函数被调用。

(13) `scsiMgrCtrlEvent()`。

这个函数向 SCSI 控制器状态机发送一个事件，它通常在选择、重选或者断开连接之后被驱动程序 `xxxEventProc()` 函数调用。

(14) `scsiMgrBusReset()`。

一旦启动的总线复位了，这个函数就要复位所有的物理设备。它通常由 `xxxEventProc()` 调用。

(15) `scsiMgrThreadEvent()`。

这个函数向线程状态机发送一个事件。它在驱动程序内部被线程管理程序调用；线程程序的入口点是 `xxxEventProc()` 函数。一般说来，`xxxEventProc()` 是一个通用的函数，它调

用了其他的特定驱动程序线程管理程序。为了更深入的理解，可以看一看高级 SCSI 控制器驱动程序模板，同样还要分析一下实际的驱动程序。

(16) `scsiMsgOutComplete()`。

在发送了一个 SCSI 消息之后，这个函数进行一些后处理。它同样是由驱动程序线程管理程序调用。

(17) `scsiMsgInComplete()`。

这个函数在接受到了一个 SCSI 消息之后进行一些后处理。它是由驱动程序线程管理程序调用。

(18) `scsiMsgOutReject()`。

当一个向外发送的消息被拒绝时，这个函数进行一些补偿处理。

(19) `scsiIdentMsgParse()`。

当 VxWorks 已经被选择或重新选择时，这个函数要分析这个进来的识别消息。

(20) `scsiIdentMsgBuild()`。

这个函数在调用的缓冲器中建立一个识别消息。

(21) `scsiCacheSnoopEnable()`。

这个函数通知库硬件缓存监听已经被启用，并且没有必要调用指定缓存函数。

(22) `scsiCacheSnoopDisable()`。

这个函数通知库硬件缓存监听已经禁用或者根本不存在，并且库必须要遵守缓存的一致性。

(23) `scsiCacheSynchronize()`。

这个函数被驱动程序为所有的缓存一致性所调用。

(24) `scsiThreadInit()`。

这个函数完成总体的线程初始化，它被驱动程序 `xxxThreadInit()` 调用。

例 I-2 提供了一个先进得 SCSI 控制器驱动程序模板，例 I-3 示例了一个 SCSI I/O 处理器汇编语言模板。这些例子说明了如何来建立这些驱动程序的结构。在这些模板中并不包括很多细节内容，它们只是从总体上说明了所涉及的问题。一旦理解了模板中的基本结构，分析一个实际的高级控制器驱动程序将会明确所涉及的一些问题，尤其是线程管理。

例 I-2 模板：先进控制器驱动程序。

```
/* Copyright 1989-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"
/*
modification history
-----
01g,19aug96,dds written
*/
/*
DESCRIPTION
```

This is the I/O driver for the XXX SCSI I/O Processor (SIOP). It is designed to work with scsiLib and scsi2Lib. This driver runs in conjunction with a script program for the XXX controller. These scripts use DMA transfers for all data, messages and status. This driver supports cache functions through scsi2Lib.

USER-CALLABLE ROUTINES

Most of the routines in this driver are accessible only through the I/O system. The following routines must be called directly: xxxCtrlCreate() to create a controller structure, and xxxCtrlInit() to initialize it. The XXX SCSI Controller's hardware registers need to be configured according to the hardware implementation. If the default configuration is not proper, the routine xxxSetHwRegister() should be used to properly configure the registers.

INTERNAL

This driver supports multiple initiators, disconnect/reconnect, tagged command queueing, synchronous data transfer and wide data transfer protocols.

In general, the SCSI system and this driver automatically choose the best combination of these features to suit the target devices used. However, the default choices may be over-ridden by using the function scsiTargetOptionsSet() (see scsi2Lib).

There are debug variables to trace events in the driver.

<scsiDebug> scsiLib debug variable, trace event in scsiLib, xxxScsiPhase(), and xxxTransact().

<scsiIntsDebug> prints interrupt informations.

INCLUDE FILES

xxx.h, xxxScript.h and scsiLib.h

```
*/
#define INCLUDE SCSI2
#include "vxWorks.h"
#include "memLib.h"
#include "ctype.h"
#include "stdlib.h"
#include "string.h"
#include "stdio.h"
#include "logLib.h"
#include "semLib.h"
#include "intLib.h"
#include "errnoLib.h"
#include "cacheLib.h"
#include "taskLib.h"
#include "drv/scsi/xxx.h"
```

```

#include "drv/scsi/xxxScript.h"
/* defines */
typedef XXX_SCSI_CTRL SIOP;
/* Configurable options */
int xxxSingleStepSemOptions = SEM_Q_PRIORITY;
char *xxxScsiTaskName = SCSI_DEF_TASK_NAME;
int xxxScsiTaskOptions = SCSI_DEF_TASK_OPTIONS;
int xxxScsiTaskPriority = SCSI_DEF_TASK_PRIORITY;
int xxxScsiTaskStackSize = SCSI_DEF_TASK_STACK_SIZE;
/*****
*
* xxxCtrlCreate - create a control structure for the XXX SCSI controller
*
* This routine creates a SCSI controller data structure and must be called
* before using a SCSI controller chip. It should be called once and only
* once for a specified SCSI controller. Since it allocates memory
* for a structure needed by all routines in xxxLib, it must be called before
* any other routines in the library. After calling this routine,
* xxxCtrlInit() should be called at least once before any SCSI transactions
* are initiated using the SCSI controller.
*
* RETURNS: A pointer to XXX_SCSI_CTRL structure, or NULL if memory
* is unavailable or there are invalid parameters.
*/
XXX_SCSI_CTRL *xxxCtrlCreate
(
    UINT8 *baseAdrs, /* base address of the SCSI controller */
    UINT clkPeriod, /* clock controller period (nsec*100) */
    UINT16 devType /* XXX SCSI device type */
)
(
    FAST SIOP *pSiop; /* ptr to SCSI controller info */
    /* check that dma buffers are cache-coherent */
    /* cacheDmaMalloc the controller structure and other driver structures */
    pScsiCtrl = (SCSI_CTRL *) pSiop;
    /* inform the SCSI libraries about the size of an XXX event and thread */
    pScsiCtrl->eventSize = sizeof (XXX_EVENT);
    pScsiCtrl->threadSize = sizeof (XXX_THREAD);
    pScsiCtrl->scsiTransact = (FUNCPTR) scsiTransact;
    pScsiCtrl->scsiEventProc = (VOIDFUNCPTR) xxxEvent;
    pScsiCtrl->scsiThreadInit = (FUNCPTR) xxxThreadInit;
    pScsiCtrl->scsiThreadActivate = (FUNCPTR) xxxThreadActivate;

```



```

pScsiCtrl->scsiThreadAbort = (FUNCPTR) xxxThreadAbort;
pScsiCtrl->scsiBusControl = (FUNCPTR) xxxScsiBusControl;
pScsiCtrl->scsiXferParamsQuery = (FUNCPTR) xxxXferParamsQuery;
pScsiCtrl->scsiXferParamsSet = (FUNCPTR) xxxXferParamsSet;
pScsiCtrl->scsiWideXferParamsQuery = (FUNCPTR) xxxWideXferParamsQuery;
pScsiCtrl->scsiWideXferParamsSet = (FUNCPTR) xxxWideXferParamsSet;
/* the following virtual functions are not used with this driver */
pScsiCtrl->scsiDevSelect = NULL;
pScsiCtrl->scsiInfoXfer = NULL;
/* fill in generic SCSI info for this controller */
scsiCtrlInit (&pSiop->scsiCtrl);
/* fill in SCSI controller specific data for this controller */
/* initialize controller state variables */
/*
 * Initialize fixed fields in client shared data area. This "shared"
 * area of memory is shared between this driver and the scripts I/O
 * processor. Fields like data pointers, data size, message pointer,
 * message size, status pointer and size, etc. are typically the
 * pieces of information shared. These fields are updated and managed
 * before and after an I/O process.
 */
xxxSharedMemInit (pSiop, pSiop->pClientShMem);
/* spawn SCSI manager - use generic code from "scsiLib.c" */
pScsiCtrl->scsiMgrId = taskSpawn (xxxScsiTaskName,
xxxScsiTaskPriority,
xxxScsiTaskOptions,
xxxScsiTaskStackSize,
(FUNCPTR) scsiMgr,
(int) pSiop, 0, 0, 0, 0, 0, 0, 0, 0, 0);
return (pSiop);
)
/*****
 *
 * xxxCtrlInit - initialize a XXX SCSI controller structure
 *
 * This routine initializes an SCSI controller structure, after the structure
 * is created with xxxCtrlCreate(). This structure must be initialized before
 * the SCSI controller can be used. It may be called more than once if
 * needed; however, it should only be called while there is no activity on
 * the
 * SCSI interface. A detailed description of the input parameters follows:
 */

```

```

*
* RETURNS: OK, or ERROR if parameters are out of range.
*/
STATUS xxxCtrlInit
(
    FAST XXX_SCSI_CTRL *pSiop, /* ptr to SCSI controller struct */
    int scsiCtrlBusId /* SCSI bus ID of this SCSI controller */
)
{
    SCSI_CTRL * pScsiCtrl = (SCSI_CTRL *) pSiop;
    /* initialize the SCSI controller */
    xxxHwInit (pSiop);
    /*
     * Put the scripts I/O processor in a state whereby it is ready for
     * selections or reselection from the SCSI bus. Such a state continues
     * until either a selection or reselection occurs or the driver interrupts
     * the scripts processor and resets its program counter to begin
     * execution elsewhere.
     */
    xxxScriptStart (pSiop, (XXX_THREAD *) pScsiCtrl->pIdentThread,
        XXX_SCRIPT_WAIT);
    return (OK);
}
/*****
*
* xxxHwInit - initialize the SCSI controller chip to a known state
*
* RETURNS: N/A
*/
LOCAL void xxxHwInit
(
    FAST SIOP *pSiop /* ptr to a SCSI controller info structure */
)
{
    /* initialize hardware independent registers */
}
/*****
*
* xxxScsiBusReset - assert the RST line on the SCSI bus
*
* Issue a SCSI Bus Reset command to the XXX SCSI controller. This should put
* all devices on the SCSI bus in an initial quiescent state.

```



```

*
* RETURNS: N/A
*/
LOCAL void xxxScsiBusReset
(
FAST SIOP *pSiop /* ptr to SCSI controller info */
)
{
/* set appropriate register values in order to reset the SCSI bus */
}
/*****
*
* xxxIntr - interrupt service routine for the SCSI controller
*
* Find the event type corresponding to this interrupt, and carry out any
* actions which must be done before the SCSI controller is re-started.
* Determine whether or not the SCSI controller is connected to the bus
* (depending on the event type - see note below). If not, start a client
* script if possible or else just make the SCSI controller wait for something
* else to happen.
*
* Notify the SCSI manager of a controller event.
*
* RETURNS: N/A
*/
void xxxIntr
(
SIOP *pSiop
)
{
XXX_EVENT event;
SCSI_EVENT pScsiEvent = (SCSI_EVENT *) &event;
BOOL connected = FALSE;
BOOL notify = TRUE;
int oldState = (int) pSiop->state;
/* Save (partial) SCSI controller register context in current thread */
/* Get event type */
pScsiEvent->type = xxxEventTypeGet (pSiop);
/* fill in event information based upon the nature of the event */
/* controller is now idle: if possible, make it run a script. */
xxxScriptStart (pSiop, (XXX_THREAD *) pScsiCtrl->pIdentThread,
XXX_SCRIPT_WAIT);

```

```

/* Send the event to the SCSI manager to be processed. */
scsiMgrEventNotify ((SCSI_CTRL *) pSiop, pScsiEvent, sizeof (event));
}
I
/*****
*
* xxxEventTypeGet - parse SCSI and DMA status registers at interrupt time
*
* RETURNS: an interrupt (event) type code
*/
LOCAL int xxxEventTypeGet
(
SIOP * pSiop
)
{
/* Read interrupt status registers */
key = intLock ();
/* Check for fatal errors first */
/* No fatal errors; try the rest (order of tests is important) */
return (INTERRUPT_TYPE);
}
/*****
*
* xxxThreadActivate - activate a SCSI connection for an initiator thread
*
* Set whatever thread/controller state variables need to be set. Ensure that
* all buffers used by the thread are coherent with the contents of the
* system caches (if any).
*
* Set transfer parameters for the thread based on what its target device
* last negotiated.
*
* Update the thread context (including shared memory area) and note that
* there is a new client script to be activated (see "xxxActivate()").
*
* Set the thread's state to ESTABLISHED.
* Do not wait for the script to be activated. Completion of the script is
* signalled by an event which is handled by "xxxEvent()".
*
* RETURNS: OK or ERROR
*/
LOCAL STATUS xxxThreadActivate

```

```

(
SIOP * pSiop, /* ptr to controller info */
XXX_THREAD * pThread /* ptr to thread info */
)
(
acsiCacheSynchronize (pScsiThread, SCSI_CACHE_PRE_COMMAND);
acsiWideXferNegotiate (pScsiCtrl, pScsiTarget, WIDE_XFER_NEW_THREAD);
acsiSyncXferNegotiate (pScsiCtrl, pScsiTarget, SYNC_XFER_NEW_THREAD);
if (xxxThreadParamsSet (pThread, pScsiTarget->xferOffset,
pScsiTarget->xferPeriod) != OK)
return (ERROR);
/* Update thread context; activate the thread */
xxxThreadUpdate (pThread);
if (xxxActivate (pSiop, pThread) != OK)
return (ERROR);
pScsiCtrl->pThread = pScsiThread;
xxxThreadStateSet (pThread, SCSI_THREAD_ESTABLISHED);
return (OK);
}
/*****
*
* xxxThreadAbort - abort a thread
*
* If the thread is not currently connected, do nothing and return FALSE to
* indicate that the SCSI manager should abort the thread.
*
* RETURNS: TRUE if the thread is being aborted by this driver (i.e. it is
* currently active on the controller, else FALSE.
*/
LOCAL BOOL xxxThreadAbort
(
SIOP * pSiop, /* ptr to controller info */
XXX_THREAD * pThread /* ptr to thread info */
)
(
xxxAbort (pSiop);
xxxThreadStateSet (pThread, SCSI_THREAD_ABORTING);
return (TRUE);
}
/*****
*
* xxxEvent - XXX SCSI controller event processing routine

```

```

*
* Parse the event type and act accordingly. Controller-level events are
* handled within this function, and the event is then passed to the current
* thread (if any) for thread-level processing.
*
* RETURNS: N/A
*/
LOCAL void xxxEvent
(
    SIOP * pSiop,
    XXX_EVENT * pEvent
)
(
    SCSI_CTRL * pScsiCtrl = (SCSI_CTRL *) pSiop;
    SCSI_EVENT * pScsiEvent = (SCSI_EVENT *) pEvent;
    XXX_THREAD * pThread = (XXX_THREAD *) pScsiCtrl->pThread;
    I
    /* Do controller-level event processing */
    /* If there's a thread on the controller, forward the event to it */
    if (pThread != 0)
        xxxThreadEvent (pThread, pEvent);
)
/*****
*
* xxxThreadEvent - SCSI controller thread event processing routine
*
* Forward the event to the proper handler for the thread's current role.
*
* If the thread is still active, update the thread context (including
* shared memory area) and resume the thread.
*
* RETURNS: N/A
*/
LOCAL void xxxThreadEvent
(
    XXX_THREAD * pThread,
    XXX_EVENT * pEvent
)
(
    SCSI_EVENT * pScsiEvent = (SCSI_EVENT *) pEvent;
    SCSI_THREAD * pScsiThread = (SCSI_THREAD *) pThread;
    SIOP * pSiop = (SIOP *) pScsiThread->pScsiCtrl;

```

```

XXX_SCRIPT_ENTRY entryPt;
switch (pScsiThread->role)
{
case SCSI_ROLE_INITIATOR:
xxxInitEvent (pThread, pEvent);
entryPt = XXX_SCRIPT_INIT_CONTINUE;
break;
case SCSI_ROLE_IDENT_INIT:
xxxInitIdentEvent (pThread, pEvent);
entryPt = XXX_SCRIPT_INIT_CONTINUE;
break;
case SCSI_ROLE_IDENT_TARG:
xxxTargIdentEvent (pThread, pEvent);
entryPt = XXX_SCRIPT_TGT_DISCONNECT;
break;
case SCSI_ROLE_TARGET:
default:
logMsg ("xxxThreadEvent: thread 0x%08x: invalid role (%d)\n",
(int) pThread, pScsiThread->role, 0, 0, 0, 0);
entryPt = XXX_SCRIPT_TGT_DISCONNECT;
break;
}
/* Resume thread if it is still connected */
xxxResume (pSiop, pThread, entryPt);
}
/*****
*
* xxxResume - resume a script corresponding to a suspended thread
*
* NOTE: the script can only be resumed if the controller is currently idle.
* To avoid races, interrupts must be locked while this is checked and the
* script re-started.
*
* Reasons why the controller might not be idle include SCSI bus reset and
* unexpected disconnection, both of which might occur in practice. Hence
* this is not considered to be a major software error.
*
* RETURNS: OK, or ERROR if the controller is in an invalid state (this
* should not be treated as a major software failure).
*/
LOCAL STATUS xxxResume
(

```

```

SIOP * pSiop, /* ptr to controller info */
XXX_THREAD * pThread, /* ptr to thread info */
XXX_SCRIPT_ENTRY entryId /* entry point of script to resume */
)
{
    STATUS status;
    int key;
    /*
     * Check validity of connection and start script if OK
     */
    key = intLock ();
    xxxScriptStart (pSiop, pThread, entryId);
    pSiop->state = NCR810_STATE_ACTIVE;
    status = OK;
    intUnlock (key);
    return (status);
}
/*****
 *
 * xxxInitEvent - XXX SCSI controller initiator thread event processing rout
 *
 * Parse the event type and handle it accordingly. This may result in state
 * changes for the thread, state variables being updated, etc.
 I
 *
 * RETURNS: N/A
 */
LOCAL void xxxInitEvent
(
    XXX_THREAD * pThread,
    XXX_EVENT * pEvent
)
{
}
/*****
 *
 * xxxSharedMemInit - initialize the fields in a shared memory area
 *
 * Initialize pointers and counts for all message transfers. These are
 * always directed to buffers provided by the SCSI_CTRL structure.
 *
 * RETURNS: N/A

```



```

*/
LOCAL void xxxSharedMemInit
(
    SIOP * pSiop,
    XXX_SHARED * pShMem
)
{
}

/*****
 *
 * xxxThreadInit - initialize a client thread structure
 *
 * Initialize the fixed data for a thread (i.e., independent of the command).
 * Called once when a thread structure is first created.
 *
 * RETURNS: OK, or ERROR if an error occurs
 */
LOCAL STATUS xxxThreadInit
(
    SIOP * pSiop,
    XXX_THREAD * pThread
)
{
    scaiThreadInit (&pThread->scsiThread);
    return (OK);
}

/*****
 *
 * xxxActivate - activate a script corresponding to a new thread
 *
 * Request activation of (the script for) a new thread, if possible; do not
 * wait for the script to complete (or even start) executing. Activation
 * is requested by signalling the controller, which causes an interrupt.
 * The script is started by the ISR in response to this event.
 *
 * NOTE: Interrupt locking is required to ensure that the correct action
 * is taken once the controller state has been checked.
 *
 * RETURNS: OK, or ERROR if the controller is in an invalid state (this
 * indicates a major software failure).
 */
LOCAL STATUS xxxActivate

```

```

(
SIOP * pSiop,
XXX_THREAD * pThread
)
{
key = intLock ();
/* Activate controller for the current thread */
intUnlock (key);
return (status);
}
/*****
*
* xxxAbort - abort the active script corresponding to the current thread
*
* Check that there is currently an active script running. If so, set the
* SCSI controller Abort flag which halts the script and causes an
* interrupt.
*
* RETURNS: N/A
*/
LOCAL void xxxAbort
(
SIOP * pSiop /* ptr to controller info */
)
{
STATUS status;
int key;
key = intLock ();
/* Abort the active script corresponding to the current thread */
intUnlock (key);
}
/*****
*
* xxxScriptStart - start the SCSI controller executing a script
*
* Restore the SCSI controller register context, including the shared memory
* area, from the thread context. Put the address of the script entry point
I
* into the DSP register. If not in single-step mode, start the script.
*
* NOTE: should always be called with SCSI controller's interrupts locked.
*

```



```

* RETURNS: N/A
*/
LOCAL void xxxScriptStart
(
    SIOP *pSiop, /* pointer to SCSI controller info */
    XXX_THREAD *pThread, /* ncr thread info */
    XXX_SCRIPT_ENTRY entryId /* routine address entry point */
)
{
    static ULONG * xxxScriptEntry [] =
    {
        xxxWait, /* wait for re-select or host cmd */
        xxxInitStart, /* start an initiator thread */
        xxxInitContinue, /* continue an initiator thread */
        xxxTgtDisconnect, /* disconnect a target thread */
    };
    /* Restore the SCSI controller register context for this thread. */
    /*
     * Set the shared data address, load the script start address,
     * then start the SCSI controller.
     */
}
/*****
 *
 * xxxXferParamsQuery - get (synchronous) transfer parameters
 *
 * Updates the synchronous transfer parameters suggested in the call to match
 * the XXX SCSI controller's capabilities. Transfer period is in SCSI units
 * (multiplies * of 4 ns).
 *
 * RETURNS: OK
 */
LOCAL STATUS xxxXferParamsQuery
(
    SCSI_CTRL *pScsiCtrl, /* ptr to controller info */
    UINT8 *pOffset, /* max REQ/ACK offset [in/out] */
    UINT8 *pPeriod /* min transfer period [in/out] */
)
{
    return (OK);
}
/*****

```

```

*
* xxxWideXferParamsQuery - get wide data transfer parameters
*
* Updates the wide data transfer parameters suggested in the call to match
* the XXX SCSI controller's capabilities. Transfer width is in the units
* of the WIDE DATA TRANSFER message's transfer width exponent field. This is
* an 8 bit field where 0 represents a narrow transfer of 8 bits, 1 represents
* a wide transfer of 16 bits and 2 represents a wide transfer of 32 bits.
*
* RETURNS: OK
*/
LOCAL STATUS xxxWideXferParamsQuery
(
    SCSI_CTRL *pScsiCtrl, /* ptr to controller info */
    UINT8 *xferWidth /* suggested transfer width */
)
{
}

/*****
*
* xxxXferParamsSet - set transfer parameters
*
* Validate the requested parameters, convert to the XXX SCSI controller's
* native format and save in the current thread for later use (the chip's
* registers are not actually set until the next script activation for this
* thread).
*
* Transfer period is specified in SCSI units (multiples of 4 ns). An offset
* of zero specifies asynchronous transfer.
*
* RETURNS: OK if transfer parameters are OK, else ERROR.
*/
LOCAL STATUS xxxXferParamsSet
(
    SCSI_CTRL *pScsiCtrl, /* ptr to controller info */
    UINT8 offset, /* max REQ/ACK offset */
    UINT8 period /* min transfer period */
)
{
}

/*****
*

```

```

* xxxWideXferParamsSet - set wide transfer parameters
*
* Assume valid parameters and set the XXX's thread parameters to the
* appropriate values. The actual registers are not written yet, but will
* be written from the thread values when it is activated.
*
* Transfer width is specified in SCSI transfer width exponent units.
*
* RETURNS: CK
*/
LOCAL STATUS xxxWideXferParamsSet
(
SCSI_CTRL *pScsiCtrl, /* ptr to controller info */
UINT8 xferWidth /* wide data transfer width */
)
{
}

```

例 1-3 模板：高级 I/O 处理器驱动程序

```

; xxxInit.n Script I/O processor assembly code for xxxLib Driver
;
; Copyright 1989-1996 Wind River Systems, Inc.
;
; /*
; Modification history
; -----
; 01a, 28jun95, jds Created. Adapted from ncr710init.n
;
;
; INTERNAL
; This file contains the assembly level SCSI scripts instructions which are
; used in conjunction with a higher level controller driver. To operate in
; SCSI SCRIPTS mode the SCSI I/O Processor requires only a SCRIPTS start
; address and a signal to begin operation. At that point, the processor
; begins fetching instructions from external memory and then executes them.
; The start address is written to the DMA SCRIPTS Pointer (DSP) register,
; which acts like a typical program counter. All SCRIPT instructions are
; fetched from external memory. The SCSI I/O Processor fetches and executes
; its own instructions by becoming a bus master on the host bus. Instructions
; are executed until a SCSI SCRIPTS interrupt instruction is encountered or
; until an unexpected interrupt causes an interrupt to the external

```

```
;processor. Once an interrupt is generated, the SCSI I/O Processor halts all
;operations until the interrupt is serviced. The further execution of
;SCRIPTS is then controlled by the SCSI controller driver which decides
;at which entry point should the SCRIPT processor start executing.
;
;There are four SCRIPT entry points which could be used by the controller
;driver. Execution thereafter is a function of the logic flow within the
;SCRIPTS and cannot be controlled by the driver. Thus, control is
;transferred to the SCRIPTS processor by the controller driver at well known
;entry points and this control is returned to the controller driver by the
;SCRIPTS by generating a SCRIPTS interrupt. The four SCRIPTS entry points
;are described below:
;
;1) xxxWait
; If the SCSI controller is not connected to the bus, this entry point is
; used. The SCRIPTS processor waits for selection or re-selection by a SCSI
; target device (which acts as an initiator during selection), or can be
; interrupted by a new command from the host. This is done by signalling
; the processor via register bits. Thus this entry point puts the SCRIPTS
; processor into a passive mode.
;
;2) xxxInitStart
; This entry point is used to start a new initiator thread or I/O process
; (in SCSI parlance), selecting a target, sending the identify message and
; thus establishing the ITL nexus, and then continuing to follow the SCSI
; protocol as dictated by the SCSI target, which drives the bus; thus,
; transferring the command, data, messages and status. This processing is
; actually done, within the code of the xxxInitContinue entry point. i.e
; if no stopping condition is encountered, execution continues on into the
; next logical entry point.
;
;3) xxxInitContinue
; This entry point resumes a suspended SCSI thread. SCSI threads are
; when further processing is required by the controller driver and an int
; instruction is executed. However, when the higher level management has
; been worked out, control comes back to a suspended thread and the process
; of cycling through all the SCSI information transfer phases continues. In
; essence, this entry point is the "meat" of an I/O process. The following
; phases are managed by this entry point.
; DATA_OUT
; DATA_IN
; COMMAND
```

```

; STATUS
; MSG_OUT
; MSG_IN
; XXX_ILLEGAL_PHASE
;
;4) xxxTgtDisconnect
; Disconnects a target from the SCST bus. It is the last entry point in
; an I/O process.
;
;
;The description level of the code is close to assembly language and is
;infact the language of the SCRIPTS processor. The assembly code is compiled
;using an NCR compiler which generates opcodes in the form of a static C
;language structure, which is then compiled and loaded into memory.
;
;The opcode is a pair of 32bit words, that allow operations and offsets for
;the SCRIPTS processor. A deailed discussion can be found in the chip's
;programmer's guide. Some of the important instructions and their formats
;are listed below.
;
;block move instruction.
; move from <offset> when PHASE_NAME
; .....
;I/O instructions
; set target
; wait DISCONNECT
; wait RESELECT
; select from <offset>,@jump
; .....
;read/write register instructions
; move REG_NAME to SFBR
;SFBR acts like an accumulator allowing branch instructions based on its
;value
; .....
I
;
;control transfer instructions
; jump <Label>
; int <value> when PHASE_NAME
; .....
;
;INTERRUPT SOURCES

```

```

;The SCSI I/O Processor has three main kind of interrupt, scsi, dma interrupt
;and script interrupt. The int instruction allows the controller driver to
;be interrupted with an interrupt value which is stored in the DSPS register.
;*/
#define NCR_COMPILE
#include "xxxScript.h"
; /*****
; *
; * xxxWait - wait for re-selection by target, selection by initiator, or
; * new command from host
; */
PROC xxxWait:
;setup instructions here
wait reselect REL(checkNewCmd)
;
; have been re-selected by a SCSI target
;
reselected:
; handle reselects, insert the reselect logic
int XXX_RESELECTED ; all seems OK so far
;
; May have a new host command to handle
;
checkNewCmd:
; insert logic for checking if the processor is connected to the bus
int XXX_READY ; processor is ready for a new thread
; /*****
; *
; * xxxInitStart - start new initiator thread, selecting target and
; * continuing to transfer command, data, messages as requested.
; *
; * At this point the script requires some data in the scratch registers.
; * This is the threads context information.
; *
; * When the script finishes, these registers are updated with the new context
; * information
; *
; */
PROC xxxInitStart:
;
; If required to identify, select w. ATN and try to transfer IDENTIFY message
; (if this fails, continue silently). Otherwise, select without ATN.

```



```

;
select atn from OFFSET_DEVICE, REL(checkNewCmd)
; add code to test various processor states and conditions interrupt driver
; if neccessary.
jump REL(nextPhase)
; /*****
; *
; * xxxInitContinue - resume an initiator thread
; *
; * At this point the script requires the threads context information in
; * scratch registers
; *
; * When the script finishes, these scratch registers are updated with the
; * the latest context information
; */
PROC xxxInitContinue:
; some setup code...
nextPhase:
;
; Normal info transfer request processing
;
phaseSwitch:
jump REL(doDataOut), when DATA_OUT
jump REL(doDataIn) if DATA_IN
jump REL(doCommand) if COMMAND
jump REL(doStatus) if STATUS
jump REL(doMsgOut) if MSG_OUT
jump REL(doMsgIn) if MSG_IN
int XXX_ILLEGAL_PHASE
; /*****
; *
; * doDataOut - handle DATA OUT phase
; */
doDataOut:
; ...
jump REL(nextPhase)
I
; /*****
; *
; * doDataIn - handle DATA IN phase
; */
doDataIn:

```

```

;...
jump REL(nextPhase)
; /*****
; *
; * doCommand - handle COMMAND phase
; */
doCommand:
;...
jump REL(nextPhase)
; /*****
; *
; * doStatus - handle STATUS phase
; */
doStatus:
;...
jump REL(nextPhase)
; /*****
; *
; * doMsgOut - handle MSG OUT phase
; */
doMsgOut:
;...
jump REL(nextPhase)
; /*****
; *
; * doMsgIn - handle MSG IN phase
; *
; * Note: there is little point in having the '810 parse the message type
; * unless it can save the host some work by doing so; DISCONNECT and
; * COMMAND COMPLETE are really the only cases in point. Multi-byte messages
; * are handled specially - see the comments below.
; */
doMsgIn:
;...
int XXX_MESSAGE_IN_RECVD ; driver handles all others
;
; Have received a DISCONNECT message
;
disconn:
;...
int XXX_DISCONNECTED
;

```



```

; Have received a COMMAND COMPLETE message
;
complete:
;...
int XXX_CMD_COMPLETE
extended:
int XXX_EXT_MESSAGE_SIZE
contExtMsg:
int XXX_MESSAGE_IN_RECVD ; at last !
/*****
*
* xxxTgtDisconnect - disconnect from SCSI bus
*
*/
PROC xxxTgtDisconnect:
;...
disconnect
int XXX_DISCONNECTED

```

1.5 BSP 接口

BSP 在它调用初始化程序时，向驱动程序提供了目标板的信息。在文件 `sysScsi.c` 中的 BSP `sysScsiInit()` 函数的主要任务是：

- 初步地完成所有指定目标板硬件的初始化工作。
- 通过调用驱动程序的 `xxxCtrlCreate()` 函数，并提供指定目标板硬件信息，例如 SCSI 控制器的寄存器基地址，来创建一个控制器驱动程序对象。
- 将 SCSI 控制器的中断向量表连接到驱动程序的中断服务子程序（ISR）。
- 通过调用函数 `xxxCtrlInit()` 和可选的函数 `xxxHwInit()`，来完成额外的驱动器初始化工作，并提供特定目标板的信息，例如 SCSI 初始总线 ID，指定硬件寄存器值。
- 如果使用了外部 DMA 控制器而且它不是 SCSI 控制器驱动程序的一部分，那么就要提供 DMA 函数。

任何其他初始化 SCSI 外围设备的指定目标板配置，例如硬盘、磁带机、块/流设备和文件系统，也必须由 `sysScsi.c` 来完成。这样配置的初始化应该位于 `sysScsiConfig()` 中。

下面的一小部分内容介绍了位于 `sysScsi.c` 中的 `sysScsiInit()` 函数模板。

例 1-4 BSP 中 SCSI 初始化模板 (`sysScsi.c`)

```
/* sysScsi.c - XXX BSP SCSI-2 initialization for sysLib.c */
```

```

/* Copyright 1984-1996 Wind River Systems, Inc. */
#include "copyright_wrs.h"
/*
modification history
-----
01a,29nov95,jde written
*/
/*
Description
This file contains the sysScsiInit() and related routines necessary for
initializing the SCSI subsystem for this BSP.
*/
#ifdef INCLUDE_SCSI
/* external inclusions */
#include "drv/scsi/xxx.h"
#include "tapeFsLib.h"
/*****
*
* sysScsiInit - initialize XXX SCSI chip
*
* This routine creates and initializes an SIOP structure, enabling use of the
* on-board SCSI port. It also connects the proper interrupt service routine
* to the desired vector, and enables the interrupt at the desired level.
*
* RETURNS: OK, or ERROR if the control structure is not created or the
* interrupt service routine cannot be connected to the interrupt.
*/
STATUS sysScsiInit ()
{
/* perform preliminary board specific hardware initializations */
/* Create the SCSI controller */
if ((pSysScsiCtrl = (SCSI_CTRL *) xxxCtrlCreate
(
(UINT8 *) SCSI_BASE_ADRS,
(UINT) XXX_40MHZ,
devType
)) == NULL)
{
return (ERROR);
}
/* connect the SCSI controller's interrupt service routine */
if (intConnect (INUM_TO_IVEC (SCSI_INT_VEC),

```

```

xxxIntr, (int) pSysScsiCtrl) == ERROR)
{
return (ERROR);
}
/* Enable SCSI interrupts */
intEnable (SCSI_INT_LVL);
/* initialize SCSI controller with default parameters (user tuneable) */
if (xxxCtrlInit ((XXX_SCSI_CTRL *)pSysScsiCtrl,
SCSI_DEF_CTRL_BUS_ID) == ERROR)
return (ERROR);
#if (USER_D_CACHE_MODE & CACHE_SNOOP_ENABLE)
scsiCacheSnoopEnable ((SCSI_CTRL *) pSysScsiCtrl);
#else
scsiCacheSnoopDisable ((SCSI_CTRL *) pSysScsiCtrl);
#endif
/* Set the appropriate board specific hardware registers for the SIOP */
if (xxxSetHwRegister ((XXX_SCSI_CTRL *)pSysScsiCtrl, &hwRegs)
== ERROR)
return (ERROR);
/* Include tape support if configured in config.h */
#ifdef INCLUDE_TAPEFS
tapeFsInit (); /* initialize tapeFs */
#endif /* INCLUDE_TAPEFS */
return (OK);
}

```

I.6 SCSI 驱动器开发指导

在这一部分中提供了开发一个新 SCSI 控制器的一些有用的建议。将整个工程分为几个较小的易于管理的步骤通常是最好的办法。

(1) 理解模板中的驱动程序和与 SCSI 库的接口。

(2) 将模板驱动程序复制到你的新的驱动程序目录下，用选择的驱动程序的名字来替换各个函数和宏的名字（例如，xxxShow() 可以变为 myDriveShow()）。

(3) 确保中断机制能够正常地工作，这样在获得一个 SCSI 中断信号时，可以激活驱动程序的中断服务程序。可以确保 ISR 被调用的一个较好的方法是写入到内存中公开的位置或者 NVRAM，这样一旦目标板重新初始化，就能够说明已经进入 ISR。使 ISR 开始工作是开发过程中的一个重要的里程碑。

(4) 使驱动程序选择一个 SCSI 外围设备。SCSI 总线分析器能够分析出总线上正在发生什么, 而且 xxxShow() 函数也非常有用。选择一个设备是另一个重要的里程碑。

(5) 按照标准的一步一步编程的过程来改进你的驱动程序, 直到获得了想要的结果。

(6) 运行标准的 WindRiver SCSI 测试, 来检验 SCSI 总线的各个方面, 包括多线程、多启动程序和多个外围设备同时工作, 以及性能和驱动器数据吞吐量。

I.7 测试程序组

在下面的内容中, 列举并且介绍了由 Wing River 提供的测试程序。这些测试程序的源代码都位于目录 target/src/test/scsi 中。

(1) scsiDiskThruputTest()。

这个测试是将 16MB 的块设备分为大小为 4096、65536 或者 1048576 字节大小的块, 每个扇区由 512 字节大小的块组成。测试程序向磁盘驱动器中写入和读取块的大小, 并且计算使用的时间, 如此来计算出吞吐量。

按如下的方式调用这个测试程序:

```
scsiDiskThruputTest "scsiBusId devLun numBlocks blkOffset"
```

单独的参数必须符合下面所描述的:

scsiBusId	目标设备 ID
devLun	设备的逻辑单元 ID
numBlocks	在块设备中块的个数
blkOffset	卷中第一个块的地址

例如: scsiDiskThruputTest "4 0 0x0000 0x0000"

(2) scsiDiskTest()。

这个测试可以完成下面所介绍的测试中的一个或者全部。调用 scsiDiskTest() 的方式如下:

```
scsiDiskTest "test scsiBusId devLun Iterations numBlocks blkOffset"
```

每个独立的参数必须符合下面所描述的:

test 下述之一:

- #1: 只运行 commonCmdsTest()
- #2: 只运行 directRwTest()
- #3: 只运行 directCmdsTest()
- ~[a]: 运行所有磁盘测试

scsiBusId	目标设备 ID
devLun	设备逻辑单元 ID

Iterations 执行读/写测试的次数
 numBlocks 在块设备中块的个数
 blkOffset 卷中第一个块的地址

例如，下面这个调用进行了所有的磁盘测试，每次读/写测试重复 10 次：

```
scsiDiskTest "-a 4 0 10 0x0000 0x0000"
```

默认的测试模式是进行下述的所有三种测试：

(3) commonCmdsTest()。

这个测试函数测试了对 SCSI 外围设备所有强制性的 SCSI 一般性访问命令。这些一般性访问命令是：

TEST UNIT READY
 REQUEST SENSE
 INQUIRY

(4) directRwTest()。

这个函数测试了下述命令的读、写、并且检查数据类型：

6-byte SCSI 命令
 10-byte SCSI 命令

(5) directCmdsTest()。

这段程序测试了下面列出的所有直接访问命令。其中，FORMAT 命令也可以通过参数 doFormat 指定一个 TRUE 值来测试。

MODE SENSE
 MODE SELECT
 RESERVE
 RELEASE
 READ CAPACITY
 READ
 WRITE
 START STOP UNIT
 FORMAT (可选择的)

(6) scsiSpeedTest()。

这个测试为与 dosFs 文件系统一起使用初始化了一个块设备。这个测试使用了一个较大的缓冲器来对使用具有缓冲和不具有缓冲 I/O 系统的连续的文件进行读写操作。

scsiSpeedTest() 程序反复运行数次，并使用 timex 对读操作和写操作进行计时。速度测试应该一次只在一个驱动器上运行，以获得最大吞吐量。

调用这个测试函数的方法如下：

```
scsiSpeedTest "scsiBusId devLun numBlocks blkOffset"
```

参数的设置必须符合下述要求:

<code>scsiBusId</code>	目标设备 ID
<code>devLun</code>	设备逻辑单元 ID
<code>numBlocks</code>	在块设备中块的个数
<code>lkOffset</code>	卷中第一个块的地址

例如:

```
scsiSpeedTest "4 0 0x0000 0x0000"
```

(7) `tapeFsTest()`。

这个测试建立一个磁带文件系统,并产生各种命令来测试磁带设备。可以选择来测试块大小固定的磁带设备、块大小可变的磁带设备、或者两者都包括。固定块大小的测试认为块的大小为 512 字节。

这个测试的调用方式如下:

```
tapeFsTest "test scsiBusId devLun"
```

参数要符合下述要求:

`test` 下述之一:

- `-f` 只运行固定块大小测试
- `-V` 只运行可变块大小测试
- `-a` 运行两种测试

`scsiBusId` 目标设备 ID

`devLun` 设备逻辑单元 ID

例如,下面这个调用进行了两种测试:

```
tapeFsTest "-a 1 0"
```

I.8 故障查找以及调试

这一部分针对查找错误的技术和调试中的捷径提供了一些建议。

1. SCSI 电缆和终端

劣质的电缆连接和劣质的 SCSI 终端是运行不稳定、在 SCSI 运行时 VxWorks 目标板挂起、甚至产生未知中断这些错误的最普遍的原因。SCSI 总线必须在两端都要结束,但是要确保在菊花链中间没有设备有上拉终端电阻或者其他形式的终端。

2. SCSI 库配置

核实测试没有超出在库内部对内存的限制,例如允许的 SCSI 线程个数、环形缓冲器的

大小、以及 SCSI 管理器堆栈的大小。在多数情况下，默认值是较合适的。

3. 数据一致性问题

数据一致性问题常常会在 CPU 支持数据缓存的硬件环境中产生。首先禁用数据缓存，检查数据错误是否还发生。如果在缓存禁用的条件下错误消失了，则说明一致性问题与缓存有关（缓存可以通过 `#undef USER_D_CACHE_ENABLE`，在 BSP 中关掉。）为了进一步查找数据一致性问题产生的原因，可以在驱动程序中对所有的内存分配使用 `cacheDmaMalloc()`。

4. 虚拟内存环境中的数据地址

如果 CPU 目标板中有存储器管理单元 (MMU)，那么驱动程序开发者在为 DMA 传输设置数据地址指针时就要非常小心。当在这种环境使用 DMA 时，必须使用物理存储器地址来代替虚拟存储器地址。这是因为在由 SCSI 总线进行 DMA 传输时，SCSI 或者 DMA 控制器是总线的控制者，从而 CPU 上的 MMU 就不能够将虚拟地址翻译为物理地址。取而代之，在向 DMA 控制器提供数据地址时，必须要使用 `CACHE_DMA_VIRT_TO_PHYS` 宏。

附录 J BSP VTS 参考条目

本附录介绍了主要的 BSP VTS 脚本参考条目、bspVal 及其所有的辅助脚本。

bspVal	—配置和运行 BSP VTS 的命令解释程序脚本。
auxclock	—辅助时钟测试。
baudConsole	—控制波特速率测试。
bootline	—引导行的功能测试。
busTas	—总线测试和设置测试。
eprom	—EPROM 测试。
error1	—错误处理测试。
error2	—错误处理测试。
model	—sysModel()测试。
network	—网络的 I/O 测试。
nvRam	—非易失性 RAM 测试。
procNumGet	—sysProcNumGet()测试。
ram	—RAM 测试。
rlogin	—测试 rlogin。
scsi	—SCSI 测试。
serial	—串行 I/O 测试。
sysClock	—系统时钟测试。
timestamp	—测试时间戳定时器。

1. bspVal

(1) 名称

bspVal—配置和运行 BSP VTS 的命令解释程序脚本。

(2) 提要

```
bspVal [-h] [-all] [-r] [-c] serverID -b bspName  
[-s secondServerID] [-sb secondBspname]  
[-t1SerDev T1SerialDevice] [-t1SerBaud T1SerialBaud]  
[-t2SerDev T2SerialDevice] [-t2SerBaud T2SerialBaud]  
[testName ... ] *
```

(3) 说明

此脚本提供了 BSP VTS 的高层用户接口。在初始化所需的测试环境之后，此脚本在目标机上执行 Tcl 脚本，运行特定的 BSP 测试。可以使用 -all 选项来运行所有可能的测试。

该 VTS 是为了使用用来运行测试而基于主机的 windsh 而设计的。在内部，这些测试将 windsh 命令和 WTX Tcl API 调用结合起来使用。这就允许相同的脚本在 UNIX 或 Windows 下运行。它也引用了特殊的函数套件，测试的时候用该函数套件来访问目标/控制交互作用的主机串行接口。如需更多关于 WTX 协议功能和 WTX Tcl API 信息，请查阅《tornado API Guide》。



注意：目标机必须与目标服务器相连接，并由某些类型的 WDB 线路来运行 VTS。如果目标机并不支持默认线路的以太网连通性，请参考不同类型 WDB 线路的 Tornado 使用手册。

VTS 也可以在引导 VxWorks ROM 映像的目标机上运行，不过目标机必须与目标服务器连结。如果目标机正在引导 VxWorks ROM 映像，某些需要 VxWorks 即时引导的测试（如引导行测试）就会失败。

在 UNIX 系统下，此脚本可直接从命令行中调用。在 Windows 系统下，就必须首先调用 bash：

① 改变目录至：*windbasehost+*

在这里 *windbase* 是由 *WIND_BASE* 环境变量所指定的目录。

② 编辑 *bashrc.bspVal* 来配置 Tornado 环境（相关的说明在文件中）。

③ 调用 bash：

```
bash -rcfile bashrc.bspVal
```

当 bash 做出即时响应（*bash\$*）时，输入与在 UNIX 主机下输入的相同的 *bspVal* 命令（以下将会讲到）。例如：

```
bash$ bspVal tgtSvr1 -b mv147 -s tgtSvr2 -sb mv147 -all
```

（4）选项

如果规定了双目标测试，必须确认第二个目标是一个有效的 BSP。另外，还要根据所选择的是单目标测试或双目标测试来更改要求的 *bspVal* 命令行变量。对于单目标测试，要求的参数是：

- *serverID*。指定目标服务器名称。
- *-b bspName*。指定有效的 BSP 的名称。
- *testName*。指定要运行的测试名称。

对于双目标测试，除了必须指定上面列出的所有参数外，还必须指定以下参数：

- *-s secondServerID*。指定引用目标板目标服务器名称。
- *-sb secondBspName*。指定引用目标板 BSP 名称。

其他 *bspVal* 选项的使用如下所示：

- `-t1SerDev T1SerialDevice`。指定主目标机的主机串行设备。
- `-t1SerBaud T1serialBaud`。指定主目标机的主机串行设备的波特率。
- `-t2SerDev T2SerialDevice`。指定引用目标板目标的主机串行设备。
- `-t2SerBaud T2SerialBaud`。指定引用目标板目标的主机串行设备的波特率。
- `-h`。打印 VTS 的使用信息。
- `-r`。在启动测试之前取消目标的重新启动。
- `-c`。删除以前的记录文件。
- `-all`。运行所有的测试。

(5) 设置文件

VTS 使用的配置文件称作 `bspName.T1`，`bspname` 是想要使其生效的 BSP 的名称。它必须在 `$WIND_BASE/host/resource/test/bspVal` 目录下。如果想运行涉及两个目标机 (`network`, `rlogin`, `busTas`) 的测试，还必须指定一个称作 `secondBspName.T2` 的配置文件，`secondBspName` 是在次目标机引用目标板下运行的 BSP 名称。对于首目标的所有配置参数应该使用 `T1_prefix`，而对于引用目标板的所有参数应该使用 `T2_prefix`。请查阅在目录 `$WIND_BASE/host/resource/test/bspVal` 下的配置文件的样本套件。

作为可选项的参数不必进行定义，因为 BSP 测试拥有与这些宏相关的默认值。而其他所有的参数就必须进行定义，即使它们已被设置为“”。

如果两个目标机测试是在普通的 VME 底板上运行的，则此测试就会将 T1 目标机假定为默认的系统控制器。用户可将任意的配置参数 `T1_BUS_MASTER` 设置为 `FALSE` 来取消此默认配置。这对于在作为普通 VME 目标板从目标机的 T2 上运行 `busTas` 测试是十分有用的。这对目标机的重新启动序列有影响，因为在任何 VME 底板上的从目标机重新启动之前的都要重新启动系统控制器。

VTS 的配置参数 `T1_SER_DEVICE`、`T1_SER_BAUD`、`T2_SER_DEVICE` 和 `T2_SER_BAUD` 可被指定为 `bspVal` 命令的命令行参数，这些参数是可选的。如果把这些配置参数中的任一个规定为 `bspVal` 命令的命令行参数，那么配置文件中指定的默认值就会改变。

(6) 记录文件

默认情况下，`bspVal` 脚本在 BSP 目录下建立了名为 `vtlogs` 的目录。在此目录下对于每个 VTS 的运行都将建立 `bspValidationLog.PID` 形式的记录文件 (PID 是 `bspVal` 运行脚本的相关过程的 ID)。必须确认 BSP 目录有权限允许建立 `vtLogs` 目录。

或者，也可以使用 `TEST_LOG_FILE` 环境变量来指定用户自定义的记录文件的完整路径名。注意：只要 `TEST_LOG_FILE` 环境变量已经定义了，所有随后的 `bspVal` 的运行将要把它们的记录输出附加到此文件中。重新设置 `TEST_LOG_FILE` 就可以将其返回默认记录文件的位置。

(7) VTS 结构

Tornado 树的 VTS 结构如下所示：

`$WIND_BASE/`

`target/src/test/bspVal/ (pklib.c, 目标测试库)`

```

host/
resource/test/bspVal/ (配置文件)
src/test/
tclserial/ (所有需要的源文件)
bspVal/
src/tests/ (所有测试)
src/lib/ (所有库函数)

```

对于每个 BSP, 目标测试库 `pkLib.c` 是通过将文件复制到 BSP 目录来进行编译的。`BspVal` 脚本检查 `pkLib.o` 是否存在, 如果需要的话, 则复制源文件至 BSP 目录, 产生 `pkLib.o` 并将 `pkLib.o` 装载到目标机上。如果要对 `$WIND_BASE/target/src/test/bspVal` 目录下的文件作任何更改, 就必须删除 `pkLib.o`, 并将目标机上先前的对象模块卸载从而可以使用最新的程序。

`Tclserial` 目录包含用于访问绑定到 Tcl 的主机的串行端口的函数的源文件。`SerLib.tcl` 文件包含一组 `tclserial` 延伸的函数库。

每个测试必须使用 `bspVal` 命令以及要求的选项才能够激活。单目标和非重新启动的测试将首先运行, 接下来是双目标的测试, 接着是时钟测试, 再后是重启目标机的测试。在这些限制下, 可以指定测试运行的顺序。

在对命令行参数进行分析之后, `bspVal`:

- ① 检查使用 `checkServerName` 库函数指定的目标服务器的有效性。
- ② 在资源目录下取得配置参数。
- ③ 如果需要则重新启动目标机 (这一项可以指定)。
- ④ 加载 `pkLib.o` 至目标机。
- ⑤ 打开到 `windsh` 的通道来发送命令以取得 Tcl 脚本并运行测试。

可以在任何时候使用中断信号来中断测试脚本 (`CTRL+c`)。不过, 目标机底板仍可能继续执行任务。这种情况下, 可能会使得底板需要进行电源循环复位。

(8) 测试清单

此节列出了 BSP VTS 中当前所包含的所有测试。一些测试是独立运行的, 而一些则需要引用底板。下列的测试是独立运行的。

- `auxClock` 测试辅助时钟的功能性。
- `budConsole` 在支持的所有连续波特率下测试通信的控制能力。
- `bootline` 检验 VxWorks 即时引导程序的可执行命令。
- `eprom` 检验 ROM 读取操作。
- `error1` 检验本地目标机总线访问、板外总线访问以及由从目标机命令解释程序初始化的零异常事件进行的分割。
- `error2` 当需要重新测试和发生重大错误时的另一组错误测试。
- `model` 测试 `sysModel()` 函数的返回值。
- `nvRam` 检测非易失性 RAM 的操作函数的功能。

- **procNumGet** 检测 sysProcNumGet()函数的返回值。
- **ram** 检测 RAM 的读操作。
- **scsi** 检测 scsi 的读写操作。
- **serial** 测试 serial 驱动程序的功能。
- **sysClock** 测试系统的时钟功能。
- **timestamp** 检测 BSP 时间戳定时器功能。

下面的测试需要引用底板，一个有效的 BSP

- **busTas** 检测在一块普通的 VEM 底板上的两块 VxWorks 目标板之间的 sysBusTas()的运行情况。
- **network** 检测基本网络功能。
- **rlogin** 检测 rlogin 的性能。

(9) 使用 bsp 配置文件配置目标机

很多 bspVal 测试可以在默认的 VxWorks 5.4 配置下运行，但是有一些测试则需要用到默认配置中不包含的功能。下表中列出了需要运用宏的测试。在建立 VxWorks 映像前，需要在目标机的 config.h 文件中加入一些宏，如下所示：

auxClock	INCLUDE_AUXCLK
busTas	INCLUDE_SHOW_ROUTINES
error2	INCLUDE_AUXCLK
network	INCLUDE_PING
	INCLUDE_NET_SHOW
rlogin	INCLUDE_SHOW_ROUTINES
	INCLUDE_RLOGIN
	INCLUDE_SHELL
	INCLUDE_NET_SYM_TBL
	INCLUDE_LOADER
timestamp	INCLUDE_TIMESTAMP

(10) 运用工程工具进行目标板配置

如果采用 Tornado 工程工具对目标机进行配置，那么每个测试要求的组件仅与使用 BSP 的 config.h 文件进行的目标机配置仅有轻微的区别。下面列出了每个测试中需要配置的组件。

auxClock	INCLUDE_AUX_CLK
busTas	INCLUDE_SM_NET_SHOW
error2	INCLUDE_AUX_CLK
network	INCLUDE_PING
	INCLUDE_NET_SHOW
rlogin	INCLUDE_RLOGIN
	INCLUDE_SHELL

```

INCLUDE_NET_SYM_TBL
INCLUDE_LOADER
INCLUDE_SM_NET_SHOW
INCLUDE_SYM_TBL_SHOW
INCLUDE_SHELL_BANNER
timestamp    INCLUDE_TIMESTAMP

```

另外，scsi 测试（上面未列出）需要为 SCSI 单独对目标机进行配置并且与一个 SCSI 驱动器连接。

下面是用户在运行 VTS 时可能遇到的错误的列表：

FATAL ERROR 测试的任何重要功能失败时，显示此错误信息。

SKIP count 某些测试如果检测到目标机与运行的测试不相匹配时，跳过此测试（例如：timestamp）。

FAIL count 测试失败时计数。

Target server ambiguity 在不同设备上的同名目标服务器同时运行时，就会发生该错误。

Networking 如果在子网络中具有高性能的网络连接，由于网络问题，用户会得到不同种类的 WTX 错误信息，测试可能失败。

File error message 如果测试中发现所需文件丢失，显示此信息。

Abnormal termination (CTRL+c) 告知用户测试被非正常中止。

2. auxClock

(1) 名称

auxClock——辅助时钟测试

(2) 提要

```
bspVal options auxClock
```

(3) 说明

此测试检测 BSP 的辅助时钟功能，运行时间为 7~10 分钟。请参看 bspVal.sh 的 option 的说明。

此脚本中的测试运用 sysAusClkConnect() 把函数与辅助时钟中断控制器相连接，但与已经通过 BSP 连接到辅助时钟 ISR 的任意程序不连接。

要运行此项测试，目标必须使用辅助时钟配置，就需要在 BSP 的 config.h 文件中加入 INCLUDE_AUXCLE 或运用工程工具包含 INCLUDE_AUX_CLK 组件。



注意：此参考条目只列出了测试失败的最有可能的原因，但并不是所有原因。

第一项测试至 4 是检验在七个不同频率时，辅助时钟的准确性：一个任选的特殊速率，

最小值，最大值和默认速率。为了测量辅助时钟的速率，使用 `sysAuxClkConnect()` 将一个简单的反馈程序连接到辅助时钟的 ISR 上。此反馈程序在时钟跳跃时计数。然后计数器清零，将辅助时钟在相应的速率下使能。计数器在 10 秒时和 130 秒时分别进行读数，就可以使用计数器的值计算辅助时钟的平均间隔速率。用三次测量来消除测试中的固定的测量潜在错误，其中 1% 的可能为时钟速率计算错误。若所测的时钟速率偏离测试值的 10% 以上，则测试速率失败。

如果有必要，则若任意一项测试失败，就应该检验定时芯片是否被正确初始化且使用正确的比例因数程序化。定时开始前，必须在 `sysAuxClkEnable()` 中使能中断。检查 `sysAuxClkRoutine()` 是否通过 `sysAuxClkConnect()` 连接，以及每个辅助时钟中断是否调用 `sysAuxClkRoutine()`。

第五项测试是检验 `sysAuxClkDisable()` 的运行情况，此测试是周期性的检查前四项测试中应用的计数器。在 `sysAuxClkDisable()` 被调用后，不能继续增加计数器的值。若测试失败，检验 `sysAuxClkDisable()` 函数是否已经禁用了定时中断，关闭了定时器，并且设置运行标志设置为 FALSE。

第六项测试进行了对 `sysAuxClkRateSet()` 的参数检验。此测试是检测对错误的输入参数能否给出适当的返回值。如果测试失败，在宏 `AUX_CLK_RATE_MIN` 和 `AUX_CLK_RATE_MAX` 下，检测 `sysAuxClkRateSet()` 是否运行了错误检测。

第七项测试检测了错误速率后通过的第一个有效速率时，`sysAuxClkRateSet()` 和 `sysAuxClkRateGet()` 的返回值。如果测试失败，则需检测 `sysAuxClkRateSet()` 是否设置了全部速率的变化，以及 `sysAuxClkRateSet()` 是否读入了相同的变量。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标机底板必须通过所有有效的 BSP 测试。

(4) 配置参数

TI_EXTRA_AUXCLK

另一个需要测试的速率（可选）

例 J-1：输出如下内容：

```

BSP VALIDATION TEST
Target server          : t53-160
BSP                    : mv147
Log file               : /tmp/bspValidationLog.5219
Auxiliary Clock Test  :
auxClk at 4321 hertz (rate = 4325 error = 0%)      : PASS
auxClk at 3 hertz (rate = 3 error = 0%)            : PASS
auxClk at 5000 hertz (rate = 5002 error = 0%)      : PASS
auxClk at 60 hertz (rate = 60 error = 0%)          : PASS
sysAuxClkDisable() disables auxClk                 : PASS
sysAuxClkRateSet() parameter checking              : PASS

```



```
sysAuxClkRateGet( ) return value : PASS
Tests RUN 7
Tests PASSED 7
Tests FAILED 0
Tests SKIPPED 0
```

(5) 参考

bspVal.sh, clockLib.tcl, bspPkCommonProc.tcl, envLib.tcl, pkLib.c

3. baudConsole

(1) 名称

baudConsole——控制波特速率测试。

(2) 提要

bspVal options baudConsole

(3) 说明

此测试为检测在所有支持的连续波特率下目标控制台的通信能力，该测试需要运行几分钟。请参看 bspVal.sh 的选项说明。



注意：此参考条目只列出了测试失败最有可能的原因，但未列出所有的原因。

此脚本包含了所有主机和目标所支持的连续波特率测试。宏 BSPTEST_HOST_RATES 提供了主机支持速率，而目标机支持速率则由连续驱动程序的 ioctl()调用返回值决定。若返回值为 OK(0)，说明支持要求的波特率，若返回值为 ERROR(-1)，说明不支持该波特率。目标机支持但主机不支持的波特率不能被测试。实际通信确认能够从命令解释程序中对现行波特速率进行检测，并且能够调用 ioctl()对下一个波特速率的控制进行设置。如果任意一个测试失败，需检测连续设备是否支持所给的波特率。同时如果有波特率发生器存在的话，必须保证其正确初始化和激活。当运用预定的波特率检验连续性时，需检验连续驱动器的 ioctl()程序运行结果是否为错误。

如果测试操作提前结束，目标板会被设置为一定的波特率而非 T1_SER_BAUD。这样的情况下，就可能需要对电源进行循环复位才能够继续以下的测试。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标机底板必须通过所有有效 BSP 测试。

(4) 配置参数

BSPTEST_HOST_RATES：主机支持的波特率（必须）。

T1_SER_DEVICE：主机应用的串行设备（必须）。

T1_SER_BAUD：默认串行波特率（必须）。

例：输出如下内容：

BSP VALIDATION TEST

```

-----
Target server          : t53-160
BSP                    : mv147
Log file               : /tmp/bspValidationLog.5219
Baud Console Test     :
console at 150 baud    : PASS
console at 300 baud    : PASS
console at 600 baud    : PASS
console at 1200 baud   : PASS
console at 1800 baud   : PASS
console at 2400 baud   : PASS
console at 4800 baud   : PASS
console at 9600 baud   : PASS
console at 19200 baud  : PASS
console at 38400 baud  : PASS
Tests RUN 10
Tests PASSED 10
Tests FAILED 0
Tests SKIPPED 0

```

(5) 参考

bspVal.sh, bspCommonProc.tcl, serLib.tcl, envLib.tcl, pkLib.c

4. bootline

(1) 名称

bootline——引导行的功能测试。

(2) 提要

bspVal options bootline

(3) 说明

此测试即时检测 VxWorks 引导 ROM 命令的可执行性，运行时间至多为几分钟。

参阅 bspVal.sh 选项说明。



注意：此参考条目中只列出了测试失败的最可能的原因，但未列出所有的原因。

当给串行控制台传送字符时或者由于串行中断挂起而导致一些可能发生的问题，第一项测试需要重新启动目标机。如果测试失败，检测在由 romInit.s 定义的 romInit() 中 CPU 中断是否被禁用，以及在 sysLib 函数 (sysSerialHwInit()) 中的串行控制中断是否被禁用和清零。

第二项测试和第三项测试检测总线错误异常处理功能（分别为本地地址和板外地址）。如果测试失败，检测存储器控制器是否被正确初始化以及存储器看门狗定时器是否被设定为合适的值。

访问时，如果并非本地（板内）存储器地址导致总线错误，则宏 `TI_BOOT_LOCAL_ERR_ADRS` 需设定为一个无效的板外地址。

第四项测试检测引导行命令和引导机制是否能够正确运行。测试失败的原因是多种多样的，需检测存储器控制器是否正确设置和是否连接网络。

在 ROM 中的 VxWorks 独立映像上，该测试脚本不能正确运行。如果目标机仅能够在 ROM 之外正确执行 VxWorks，则该测试脚本就不属于 BSP 的有效程序中的一部分。

确保配置文件中的所有引导行的参数代表实际目标机的引导参数。如果不符合，目标机将会出现如下情形：随着输入参数的变化，需采用上电期间重置。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标机底板必须通过所有的有效 BSP 测试。

（4）配置参数

`TI_BOOT_LOCAL_ERR_ADRS`: 产生总线错误的本地存储器地址（必须）。

`TI_BOOT_OFFBOARD_ERR_ADRS`: 产生总线错误的板外地址（必须）。

`TI_BOOT_LOCAL_BUS_ERR_MSG`: 本地总线错误信息关键字（必须）。

`TI_BOOT_OFFBOARD_BUS_ERR_MSG`: 板外总线错误信息关键字（必须）。

`TI_SER_DEVICE`: 主机使用的串行设备（必须）。

`TI_SER_BAUD`: 默认串行波特率（必须）。

`TI_TMO_BOOT`: 引导 VxWorks 的超时值。

例 J-2: 输出如下内容:

```

BSP VALIDATION TEST
-----
Target server           : t53-160
BSP                     : mv147
Log file                : /tmp/bepValidationLog.5219
Bootline Test           :
Control X test          : PASS
Bus error test for local error address : PASS
Bus error test for off-board error address : PASS
boot commands test      : PASS

Tests RUN 4
Tests PASSED 4
Tests FAILED 0
Tests SKIPPED 0

```

（5）参考

bspVal.sh, bspPkCommonProc.tcl, serLib.tcl, envLib.tcl, pkLib.c

5. busTas

(1) 名称

busTas——总线测试和设置测试。

(2) 提要

bspVal options busTas

(3) 说明

此模块提供检测-设置测试的 Tcl 编码，检测在一块共用 VME 底板上两块 VxWorks 目标板之间的 sysBusTas() 操作。此模块实现 Tcl 和 WTX 协议功能。运行整个测试大约需要 10 分钟，请参阅 bspVal.sh 的选项说明。

如果测试在两个共用存储器的目标机间运行，确保目标机配置中包含共享存储器显示程序。要实现此项功能，就需要在 BSP 的 config.h 文件中加入 INCLUDE_SHOW_ROUTINES，或者运用工程工具加入 INCLUDE_SM_NET_SHOW 组件。



注意: 此参考条目只列出了测试失败最有可能的原因，但未列出所有的原因。

此脚本并未重新引导目标机，而是假设两个目标机与其相应的目标服务器已打开并运行。

默认状态下，该测试假定目标机 T1 为共用 VME 第板的系统控制器。若设置可选配置参数 T1_BUS_MASTER 设置为 FALSE，则可覆盖默认配置，在这种情况下，就将目标机 2 假定为为系统控制器。要确保在运行任何测试之前，在目标要被重新引导的情况下，T1_BUS_MASTER 具有正确值。

通过调用主控制器上的 cacheDmaMalloc() 函数可以获得两块连续的高速缓冲存储器单元，从而使测试开始。这两块存储器单元用于信号量和存取计数。函数 pkTestRamBytecacheDmaMalloc() 的返回地址是否可读写的进行检测。

接着，通过调用控制器上的 sysLocalToBusAdrs() 函数，运用本地地址来计算总线地址。sysLocalToBusAdrs() 返回的总线地址传递给受控器，受控器使用函数 sysLocalToBusAdrs() 得到本地地址，以便访问控制器共享存储器中的信号量和计数单元。调用 SmAdrsFind 程序来检测信号量和计数单元对于主控制器和受控器是否均是可见的。然后，初始化存储单元，通过发起主控制器和受控器上的 pkTasTest 任务启动测试。运用函数 wtxContextCreate() 和 wtxContextResume()，发起一个测试。函数 pkTasTest() 使用信号量单元用于互斥。当控制器接收到信号量时，若为奇数，计数器的值随之增加，则运行忙碌-等待循环，直到信号量被消除为止。受控器以相同的方式运行，只不过为偶数时，计数器的值增加。此过程保证了在两项任务中接收信号和计数器的增加能轮流进行。

控制计数器从脚本进行监控，所以计数器读操作每秒进行两次。若第二次比第一次长，则短期测试是成功的。在 10 分钟的暂停后，重新进行读操作，若计数器仍增加，则长期测

试是成功的。

如果测试失败，则需检验控制器和受控器是否同时访问相同的控制存储单元。若控制程序 `sysLocalToBusAdrs()` 或受控程序 `sysBusToLocalAdrs()` 不能正确运行，测试就不能找到访问控制存储单元的合适受控器地址。控制存储单元必须是连续高速缓存而且是可写的，VME 控制芯片必须被初始化为正确的值。为了加快目标板的速度，可能会有必要通过宏 `T1_TAS_DELAY` (`T2_TAS_DELAY`) 来增加忙碌-等待延迟时间。

脚本中给出的两项测试需要在能够被控制器和受控器访问且基于控制器上的两个不同的高速缓存单元上进行。默认值状态下，这些单元能够通过调用控制器上的 `cacheDmaMalloc()` 获得。或者，宏 `T1_COUNT_ADRS` 和/或者 `T1_SEM_ADRS` 能够从默认值为 -1 变化为指向控制器上的惟一共享存储器地址。在这种情况下，使用者必须确保这些存储器是空闲的高速缓存单元。详细的了解这些单元的功能是为了调试程序和降低板上的硬件局限性。最后需确认宏的值设置为“-1”，并且通过调用 `cacheDmaMalloc()`，动态地对存储单元进行分配。

除了严重的硬件和软件缺陷（比如是 VxWorks 的可确认问题）以外，目标板必须通过所有有效 BSP 测试。当然，如果 VME 板与目标板不兼容，则 BSP 不需要此测试。

(4) 配置参数

`T1_COUNT_ADRS`: 控制共享计数器的地址（可选）。
`T1_SEM_ADRS`: 控制共享信号量的地址（可选）。
`T1_TAS_DELAY`: 控制器在 TAS 过程中的忙延迟（可选）。
`T2_TAS_DELAY`: 受控器在 TAS 过程中的忙延迟（可选）。

例 J-3: 输出包括如下内容:

```

                                BSP VALIDATION TEST
                                -----
Target server                   : t53-160
BSP                             : mv147
Second target server           : t214-2
Second BSP                     : mv147
Log file                       : /tmp/bspValidationLog.6425
busTas                         :
short-term TAS test            : PASS
long-term TAS test             : PASS

                                Tests RUN 2
                                Tests PASSED 2
                                Tests FAILED 0
                                Tests SKIPPED 0

```

(5) 参考

`bspVal.sh`, `smLib.tcl`, `pkLib.c`

6. eprom

(1) 名称

eprom-EPROM 测试。

(2) 提要

bspVal options eprom

(3) 说明

此测试检测了 ROM 的读操作。基于脚本运行此测试的执行时间取决于 CPU 的速度和所需测试的 ROM 的数量。请查阅 bspVal.sh 的选项说明。



注意：此参考条目仅列出了测试失败最有可能的原因。并没有列出所有可能的原因。

基于此脚本的三个测试调用 pkRead() 来运行在 ROM_BASE_ADRS 和 ROM_BASE_ADRS + ROM_SIZE 之间的 ROM 单元读操作。使用 1-、2-或者 4-比特通路来进行读取操作，这取决于正在进行哪个测试。如果此测试失败了，但是存储硬件运行良好，则检查存储控制器是否正确初始化。如果给出了 MMU，为使整个 ROM 空间可被访问，需对其进行正确配置。

除了严重的硬件和软件缺陷（比如是 VxWorks 的可确认问题）以外，目标板必须通过所有有效 BSP 测试。

(4) 配置参数

无。

例 J-4：输出包括如下内容：

BSP VALIDATION TEST

```
-----
Target server           : t53-160
BSP                     : mv147
Log file                : /tmp/bspValidationLog.5219
EPROM Test             :
1-byte read of EPROM   : PASS
2-byte read of EPROM   : PASS
4-byte read of EPROM   : PASS
```

```
Tests RUN 3
Tests PASSED 3
Tests FAILED 0
Tests SKIPPED 0
```

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, pkLib.c

7. error1

(1) 名称

error1——错误处理测试。

(2) 提要

`bspVal options error1`

(3) 说明

此脚本测试目标机的本地总线访问。由零异常变量实现的板外总线的访问和划分从目标机的命令解释程序开始。请查阅 `bspVal.sh` 的选项说明。



注意: 此参考条目只列出了测试失败最有可能的原因,但未列出所有的原因。

第一项测试和第二项测试检测总线错误的异常操作功能(分别为本地和板外地址)。若测试失败,须检测存储控制器是否被正确初始化和任意存储看门狗定时器是否设置为合理值。

访问时,若并非本地(板上)存储地址产生总线错误,则设置宏 `T1_LOCAL_ERR_ADRS` 为无效板外地址。

第三项测试会引起由零异常变量进行的分配,并且对控制台的返回信息进行检测。

(4) 配置参数

<code>T1_LOCAL_ERR_ADRS:</code>	产生总线错误的本地错误地址(必须)。
<code>T1_OFFBOARD_ERR_ADRS:</code>	产生总线错误的板外错误地址(必须)。
<code>T1_DIV_ZERO_MSG:</code>	总线错误信息关键字(必须)。
<code>T1_LOCAL_BUS_ERR_MSG:</code>	本地总线错误信息关键字(必须)。
<code>T1_OFFBOARD_BUS_ERR_MSG:</code>	板外总线错误信息关键字(必须)。
<code>T1_SER_DEVICE:</code>	主机串行设备(必须)。
<code>T1_SER_BAUD:</code>	默认串行波特率(必须)。
<code>T1_TMO_BOOT:</code>	引导 VxWorks 的超时值(必须)。

例 J-5: 输出包括如下内容:

```

BSP VALIDATION TEST
-----
Target server           : t53-160
BSP                     : mv147
Log file                : /tmp/bspValidationLog.5219
First Error Test       :
bus-error test for local address : PASS
bus-error test for offboard address : PASS
Divide by zero test    : PASS

```

Tests RUN 3

```
Tests PASSED 3
Tests FAILED 0
Tests SKIPPED 0
```

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, serLib.tcl, envLib.tcl, pkLib.c

8. error2

(1) 名称

error2-错误处理测试。

(2) 提要

```
bspVal options error2
```

(3) 说明

此测试检测当处理来自于 VxWorks shell 的错误时，目标板是否挂起。整个测试需要运行几分钟时间。请参阅 bspVal.sh 的选项说明。

为了运行此测试，目标机必须以辅助时钟的方式来进行配置。为了达到此目的，需要在 BSP 的 config.h 文件中加入 INCLUDE_AUXCLK，或者在使用工程工具时，包含 INCLUDE_AUX_CLK 组件。



注意:此参考条目只列出了测试失败最有可能的原因,但未列出所有的原因。

第一项测试实现引起总线错误的程序与目标机的系统时钟中断之间的连接。这会引起中断时的异常情况，称为“灾难性”错误。此异常情况通常会导致目标板的重新启动。目标机会显示相应的错误信息。若测试失败，请检测 sysLib.c 中的 sys sToMonitor() 函数是否正确运行。同时，检测是否正确设定了 sysLib.c 中的 sysExcMsg 地址。

当特征值被输入到串行控制台，并且辅助时钟工作于最大速率时，第二项测试就从命令解释程序中重新启动目标机。此测试保证重新启动时，等待中断不会产生问题。若测试失败，请检测是否在 romInit() 中禁用 CPU 中断，romInit() 在 romInit.s 中作了定义，以及是否禁用和清除在 sysSerialHwInit() 中所有的串行设备中断。同时，检验辅助时钟程序 sysAuxClkRoutine() 是否被初始化为 NULL，若已经为 NULL，确保在 sysAuxClkInt() 中辅助时钟不被调用。

保证在配置文件中的所有引导行参数为真正的目标机引导参数。否则，会使目标机产生如下情况：随着引导参数的变化，需要进行上电期间重置。

除了严重的硬件和软件缺陷（比如是 VxWorks 的可确认问题）以外，目标板必须通过所有有效 BSP 测试。

(4) 配置参数

T1_OFFBOARD_ERR_ADRS: 产生总线错误的板外地址（必须）。

T1_CATASTROPHIC_ERR_MSG: 重大错误信息关键字 (必须)。
 T1_SER_DEVICE: 主机上的串行设备 (必须)。
 T1_SER_BAUD: 使用的串行波特速率 (必须)。
 T1_TMO_BOOT: 引导 vxWorks 的超时值 (必须)。

例 J-6: 输出包括如下内容:

```

                                BSP VALIDATION TEST
                                -----
Target server                   : c53-160
BSP                             : mv147
Log file                       : /tmp/bspValidationLog.5219
Second Error Test              :
catastrophic error test        : PASS
reboot with interrupts         : PASS

                                Tests RUN 2
                                Tests PASSED 2
                                Tests FAILED 0
                                Tests SKIPPED 0
  
```

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, serLib.tcl, envLib.tcl

9. model

(1) 名称

model——sysModel()测试。

(2) 提要

```
bspVal options model
```

(3) 说明

此测试检测目标板是否与目标机配置文件相吻合。为了进行此测试,需检测 sysModel() 程序的返回值,此值从目标服务器储存的信息中获得。整个测试过程只需几秒钟。请参阅 bspVal.sh 的选项说明。

若指定了 bspVal 的 -all 选项,首先将运行此测试。



注意: 此参考条目只列出了测试失败最有可能的原因,但未列出所有的原因。

第一项测试 (惟一) 对 sysModel() 的返回值和 T1_MODEL 指定的字符进行比较。若测试失败,请检测 sysLib 中的 sysModel() 函数是否返回了正确的目标模型信息。

除了严重的硬件和软件缺陷 (比如是 VxWorks 的可确认问题) 以外,目标板必须通过

所有的有效 BSP 测试。

(4) 配置参数

T1_MODEL: sysModel()的期望值 (必须)。

例 J-7: 输出包括如下内容:

```

                                BSP VALIDATION TEST
                                -----

Target server                   : t53-160
BSP                             : mv147
Log file                        : /tmp/bspValidationLog.5219
Model Test                     :
sysModel( ) return value       : PASS

                                Tests RUN 1
                                Tests PASSED 1
                                Tests FAILED 0
                                Tests SKIPPED 0

```

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, envLib.tcl

10. network

(1) 名称

network——网络的 I/O 测试。

(2) 提要

bspVal options network

(3) 说明

此测试检验基本网络功能, 网络媒质的独立性。此脚本写入了 Tcl 中并且使用了一些 WTX 协议的函数, 同时它还使用 tclSerial 库函数访问串行端口。整个测试需要运行几分钟时间。请参阅 bspVal.sh 的选项说明。



注意: 此参考条目只列出了测试失败最有可能的原因, 但未列出所有的原因。

本测试需要两个目标机。被测试的目标机为控制器, 第二个目标机作为受控器, 用于运行 ping 和 udpEcho。运行于受控器上的 VxWorks 映像必须建立在以定义的宏 INCLUDE_PING 的基础之上。两个目标机都需要拥有由已定义的宏 INCLUDE_NET_SHOW 建立的 VxWorks 映像。如果运用工程工具, 应在两个目标机的 VxWorks 映像中分别加入 INCLUDE_PING 和 INCLUDE_NET_SHOW 组件。

第一项测试运用 ping 来发出大约 50 个 ICMP 包 (数据包) 到控制器的 ICMP 服务器。

每一个数据包包含一个来自于服务器响应的请求；而应该受控器接收该响应。每个数据包均会显示出往返回程，即发送和接收的时间。只要受控器接受大约五十次响应，测试就能通过。若测试失败，请检测是否正确连接网络以及目标机 IP 地址是否正确。

第一项测试需要受控器上存在 ping，并保证它处于路径中。若 ping 丢失，测试就会中断，由于 UDP 测试需要运用 ping 的统计信息。

第二项测试发起控制器上的 pkUdpEchoTest()（一个 pkLib.c 函数）。然后发送大量的 UDP 包到控制器上的一个 UDP 的端口监督应答程序，此程序也是由发起 pkUdpEchoTest()（一个 pkLib.c 函数）开始的。每一个 UDP 包都是由受控器发送到控制器，然后由受控器应答和读回，然后对数据包的大小和内容是否正确进行检验。共有 16 种不同大小的数据包被传送和接收。当等待回应时，若 pkUdpEchoTest() 计时超时，数据包将被重新发送。若现有的数据包不能满足要求，测试显示为失败。每一个 UDP 数据包的计时值是从 ping 测试中的统计信息得到的，该值取决于所采用的网络媒质。但是，数据包往返时间的真实值应小于所计算的值。如测试失败，则确保使用的主机可靠并且网络没有因为其他的数据包通信量太大而造成过载。

第二项测试使用 UDP/IP 网络协议。由于 UDP/IP 并不能确保数据包被传送，因此有时主机或目标机会丢失数据包。虽然丢失一些数据包并不能导致测试的失败，但是如果丢失大量的数据包（>>1%）则说明可能有如下方面存在问题：网络配置，网络设备，设备驱动器，网络软件。

除了严重的硬件和软件缺陷（比如是 VxWorks 的可确认问题）以外，目标板必须通过所有的有效 BSP 测试。

若目标机连续失败，则会使第二项测试不能正确的运转。如果证明了是由于主机的问题导致数据包丢失（例如：运行 etherfind），则不能说明目标机是否通过测试。当评定测试结果时，需考虑到硬件和网络环境的局限性。

（4）配置参数

T1_BOOT_TN:	控制器目标机名称（必须）。
T1_BOOT_E:	控制器目标机 IP 地址（必须）。
T1_BOOT_B:	控制器目标底板地址（必须）。
T2_BOOT_TN:	受控器目标机名称（必须）。
T2_BOOT_E:	受控器目标机 IP 地址（必须）。
T2_BOOT_B:	受控器目标机底板地址（必须）。
T2_UDP_TRANS:	UDP 测试中使用的数据包数量。

例 J-8：输出包括如下内容：

BSP VALIDATION TEST

```
-----
Target server          : t53-160
BSP                    : mv147
```

```

Second target server          : t214-2
Second BSP                    : mv147
Log file                      : /tmp/bspValidationLog.6425
network                      :
ping target test              : PASS
UDP packet echo test         : PASS
Tests RUN 2
Tests PASSED 2
Tests FAILED 0
Tests SKIPPED 0

```

(5) 参考

bspVal.sh, networkLib.tcl, pkLib.c

11. nvRam

(1) 名称

nvRam——非易失性 RAM 的测试

(2) 提要

```
bspVal options nvRam
```

(3) 说明

此测试检测了 sysNvRamGet() 和 sysNvRamSet(), 它们是使用非易失性存储器的 sysLib.c 函数。



注意: 此参考条目只列出了测试失败最有可能的原因, 但未列出所有的原因。

若目标机中不包含非易失性 RAM, 那么 BSP 中 NV_RAM_SIZE 应该被设置为 NONE (-1)。在这种类型的目标板中, 仅仅能进行 sysNvRamSet() and sysNvRamGet() 的参数检测。若 NVRAM 只有一部分是可写的, 或者一部分被保留用于硬件操作, 一定要指定文件 config.h 中的宏 NV_RAM_SIZE_WRITEABLE 的可写部分的大小。

此脚本测试的执行时间是由所检测的 NVRAM 的大小和 CPU 的速度决定的。整个测试运行时间只需几分钟。

此脚本仅通过八次调用 pkTestNvRam(), 就可完成不同的非易失性 RAM 的功能测试。

第一项测试检测 sysNvRamGet() 对 NV_RAM_SIZE 字节大小的数据进行的读操作。若测试失败, 请检验 NV_RAM_SIZE 是否被设置为与 NVRAM 相同的大小, 并且 NV_BOOT_OFFSET 已被正确定义和使用。同时要保证 sysNvRamGet() 将要求的整个数据长度从 NVRAM 复制到了输入缓冲器中。

第二项测试进行读、写和读 NVRAM 中大小为 NV_RAM_SIZE 的字符数从而检验 sysNvRamGet() 的功能性。若测试失败, 需检验与第一项测试中相同的问题。同时要保证

`sysNvRamGet()` 将要求的整个数据长度从输入缓冲器复制到了 NVRAM 中。

第三项测试检测了 `sysNvRamGet()` 的读取零字节数的操作。若测试失败，请检测 `sysNvRamGet()` 是否允许读入长度为零。当缓冲器通过 EOS 正常中止时，函数的返回值为 OK。

第四项测试进行 `sysNvRamSet()` 的参数检验。此测试检验，若给了错误输入参数，是否能返回适当的输出值。若测试失败，请检测错误测试是否由 `sysNvRamSet()` 提供的，它对于负偏移量或长参数返回值应该为 ERROR。同时，偏移量值加上长参数应该小于 NV_RAM_SIZE 的值。

第五项测试进行 `sysNvRamGet()` 的参数检验。该测试检验，若给了错误的输入参数，是否能返回适当的输出值。若测试失败，请检测错误测试是否由 `sysNvRamSet()` 提供的，它对于负偏移量或长参数返回值应该为 ERROR。同时，偏移量值加上长参数应该小于 NV_RAM_SIZE 的值。

第六项测试写入一些数据 (0xff)，然后从相同的 NVRAM 单元中读数据。这是为了检测读出的数据是否相同，并且缓冲器是否返回 NULL 终止符。若测试失败，请检验写入 NVRAM 的数据 0xff 是否有问题，输入缓冲器是否以 EOS 的形式正确中止。

第七项测试写入一些数据 (0x00)，然后从相同的 NVRAM 单元读数据。这是为了检验读出的数据是否相同，并且缓冲器是否返回 NULL 终止符。若测试失败，请检验将数据 0x00 写入 NVRAM 是否会产生问题以及输入缓冲器是否以 EOS 的形式正确中止。

测试 8 将 NVRAM 中的 NV_RAM_SIZE 的字节数返回到第一项测试的读入值中。若测试失败，请检验 NVRAM 中的 NV_RAM_SIZE 的字节数是否能够用 `sysNvRamSet()` 和 `sysNvRamGet()` 进行读写。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标板必须通过所有有效 BSP 测试。

(4) 配置参数

无。

例 J-9：输出包括如下内容：

```

BSP VALIDATION TEST
-----
Target server           : t53-160
BSP                     : mv147
Log file                : /tmp/bspValidationLog.5219
Non-Volatile RAM Test  :
sysNvRamGet() of boot line           : PASS
sysNvRamSet() and sysNvRamGet() of complemented bootline : PASS
sysNvRamGet() with length zero       : PASS
sysNvRamSet() parameter checking     : PASS
sysNvRamGet() parameter checking     : PASS

```

```

sysNvRamSet( ) and sysNvRamGet( ) of 0xff data      : PASS
sysNvRamSet( ) and sysNvRamGet( ) of 0x00 data      : PASS
sysNvRamSet( ) and sysNvRamGet( ) of boot line      : PASS

```

```

Tests RUN 0
Tests PASSED 0
Tests FAILED 0
Tests SKIPPED 0

```

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, pkLib.c

12. procNumGet

(1) 名称

procNumGet——procNumGet 测试。

(2) 提要

```
bspVal options procNumGet
```

(3) 说明

此测试检测 sysProcNumGet() 函数的返回值。整个测试需要几分钟。请参看 bspVal.sh 的选项说明。



注意: 此参考条目只列出了测试失败最有可能的原因,但未列出所有的原因。

第一项测试(惟一)比较 sysProcNumGet() 的返回值和 T1_BOOT_PROCNUM 指定的数值。若测试失败,请检验 sysProcNumGet() 是否返回了正确的目标处理器的数值信息。

除了严重的硬件和软件缺陷以外(比如是 VxWorks 的可确认问题),目标板必须通过所有有效 BSP 测试。

(4) 配置参数

T1_BOOT_PROCNUM sysProcNumGet() 的期望值(必须)。

例 J-10: 输出包括如下内容:

```

BSP VALIDATION TEST
-----
Target server          : t53-160
BSP                    : mv147
Log file               : /tmp/bspValidationLog.5219
sysProcNumGet Test     :
sysProcNumGet( ) return value      : PASS
Tests RUN 1
Tests PASSED 1

```

Tests FAILED 0

Tests SKIPPED 0

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, envLib.tcl

13. ram

(1) 名称

ram——RAM 测试。

(2) 提要

bspVal options ram

(3) 说明

此测试检测 RAM 的读操作。请参阅 bspVal.sh 的选项说明

 **注意:** 此参考条目只列出了测试失败最有可能的原因,但未列出所有的原因。

此脚本的三项测试都是调用 pkRead() 来进行 RAM 内 LOCAL_MEM_LOCAL_ADRS 和 sysMemTop() 之间单元的读操作。读操作运用 1-, 2-, 4-比特通路, 由运行的测试种类所决定。若任意测试失败, 但是存储硬件运行良好, 需检测存储控制器是否被初始化为正确值。同时要确保 LOCAL_MEM_LOCAL_ADRS 和 LOCAL_MEM_SIZE 被设定为正确值。如果给出了 MMU, 要使整个 RAM 空间可访问, 就需对其进行正确的配置。请检验 sysMemTop() 是否返回正确值。

除了严重的硬件和软件缺陷以外 (比如是 VxWorks 的可确认问题), 目标板必须通过所有有效 BSP 测试。

(4) 配置参数

无。

例 J-11: 输出包括如下内容:

BSP VALIDATION TEST

```
Target server           : t53-160
BSP                     : mv147
Log file                : /tmp/bspValidationLog.5219
RAM Test                :
1-byte read of RAM      : PASS
2-byte read of RAM      : PASS
4-byte read of RAM      : PASS
```

Tests RUN 3

```

Tests PASSED 3
Tests FAILED 0
Tests SKIPPED 0

```

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, pkLib.c

14. rlogin

(1) 名称

rlogin——测试 rlogin。

(2) 提要

```
bspVal options rlogin
```

(3) 说明

此测试检测两块 VxWorks 目标板之间的 rlogin 的性能。整个测试需要几分钟的时间。请参阅 bspVal.sh 的选项说明



注意：此参考条目只列出了测试失败最有可能的原因，但未列出所有的原因

为了运行此测试，用于建立 VxWorks 映像的配置文件必须包含以下宏模块：

INCLUDE_SHELL

INCLUDE_RLOGIN

INCLUDE_SHOW_ROUTINES

INCLUDE_NET_SYM_TBL

INCLUDE_LOADER

若运用工程工具建立 VxWorks 映像，需在 VxWorks 结构中加入以下组件：

INCLUDE_RLOGIN

INCLUDE_SHELL

INCLUDE_NET_SYM_TBL

INCLUDE_LOADER

INCLUDE_SM_NET_SHOW

INCLUDE_SYM_TBL_SHOW

INCLUDE_SHELL_BANNER

此脚本运行的所有测试均与检测 rlogin 进程有关。当对引用底板上的目标机发出 rlogin 命令时，第一项测试检测目标板的控制台是否出现“IN USE”。第二项测试通过从调用 VxWorks 命令解释程序的 version() 函数以及尝试使目标板的地址与其相匹配来确认 rlogin 会话。第三项测试通过运行 printLogo() 来检测 pty。第四项测试检测 logout() 的操作。然后，rlogin 的该进程就在目标板到引用目标板之间重复进行，并且第五项测试-8 来确认 rlogin

会话。若任意测试失败，请检验目标板的 IP 地址和底板地址(T1_BOOT_E 或者 T1_BOOT_B)、引用目标板的 IP 和底板地址(T2_BOOT_E 或者 T2_BOOT_B)是否正确设置。同时还需检验它们相应的目标名(T1_BOOT_TN 和 T2_BOOT_TN)是否正确设定。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标板必须通过所有有效 BSP 测试。



注意：此测试并不需要底板，只须两个目标机之间有网络连接即可。若将目标板置于单独的子网上，则需正确设定每一个相应的路由表。

(4) 配置参数

T1_BOOT_TN: 目标机名（必须）。
T1_BOOT_E: 目标机 IP 地址（必须）。
T2_BOOT_B: 目标机底板地址（若 T1_BOOT_E 未设定，则必须）。
T2_BOOT_TN: 引用目标机名（必须）。
T2_BOOT_E: 引用目标机 IP 地址（若 T2_BOOT_B 未设定，则必须）。
T2_BOOT_B: 引用目标板地址（若 T2_BOOT_E 未设定，则必须）因为默认值需应用以太网,若同时设定 T2_BOOT_E 和 T2_BOOT_B，测试将应用引用目标机 IP 地址。

例 J-12: 输出包括如下内容:

```

BSP VALIDATION TEST
-----
Target server           : t53-160
BSP                     : mv147
Second target server    : t214-2
Second BSP              : mv147
Log file                : /tmp/bspValidationLog.6425
rlogin                  :
IN USE message when rlogin to t53-160      : PASS
reach shell of t53-160                     : PASS
rlogin pty usage on t53-160                 : PASS
logout from t53-160                        : PASS
IN USE message when rlogin to t214-2       : PASS
reach shell of t214-2                      : PASS
rlogin pty usage on t214-2                 : PASS
logout from t214-2                        : PASS

Tests RUN 8
Tests PASSED 8
Tests FAILED 0

```

Tests SKIPPED.0

(5) 参考

bspVal.sh, loginLib.tcl

15. scsi

(1) 名称

scsi——SCSI 测试。

(2) 提要

bspVal options scsi

(3) 说明

此测试检测 SCSI 的读写操作。请参阅 bspVal.sh 的选项说明。

为了运行此测试，目标必须在 SCSI 的支持下配置。如要进行配置，目标必须通过定义 BSP 的 config.h 文件中的 INCLUDE_SCSI_SHOW 或者使用工程工具来加入 INCLUDE_SCSI_SHOW 组件。



警告：所有 SCSI 设备上的数据在测试后都会被覆盖。

此脚本中测试所需的时间由被检测的 SCSI 设备的大小和速度、CPU 的速度、测试缓冲器的大小(TI_SCSI_BUFSIZE)来决定。小型缓冲器会减缓测试。对于特殊的缓慢目标机、运行缓慢的设备、大型 SCSI 设备和/或小型缓冲器会通过宏 TI_TMO_SCSI 来增加超时值。整个过程需要一小时以上的时间来完成。

对于 RAM 有限的目标板，有必要从它的默认值（64k 比特）减去 TI_TMO_SCSI。而缓冲器的大小必须是进行测试的设备的大小的整数倍。



注意：此参考条目只列出了测试失败最有可能的原因，但未列出所有的原因。

第一项测试检测了 scsiShow()程序的返回值。若测试失败，可认为是由于未正确配置 SCSI，并且下一测试（设备的读/写操作）将被忽略。

第二项测试调用 pkTestOneUnit()，并通过 TI_SCSI_ID 和 TI_SCSI_LUN 来测试 SCSI 设备。设备的每一部分均以一定的格式写入内容，然后使用每部分的读入值来检验此格式。若测试失败，需检测 SCSI ID 和 LUN 是否正确配置，并且出现在 scsiShow()命令的输出中。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标板必须通过所有有效 BSP 测试。当然，如果目标板不支持 SCSI，则 SCSI 测试不是验证 BSP 有效的必要部分。

(4) 配置参数

TI_SER_DEVICE: 目标机使用的串行设备。

T1_SER_BAUD: 目标机使用的串行波特率。
T1 SCSI_ID: 测试时设备的 SCSI ID。
T1 SCSI_LUN: 测试时设备的 SCSI LUN。
T1 SCSI_BUFSIZE: 缓冲器的大小 (单位: 比特) (可选)。
T1 SCSI_NBBUF: 可写的缓冲器的数量 (可选)。
T1 SCSI_NBLOOP: 执行的循环次数 (可选)。
T1_TMO SCSI: 超时值 (单位: 秒) (可选)。

例 J-13: 输出包括如下内容:

```

                                BSP VALIDATION TEST
                                -----
Target server                   : t53-160
BSP                             : mv147
Log file                       : /tmp/bspValidationLog.5219
SCSI Test                      :
scsiShow( ) test               : PASS
SCSI write/read/verify device  : PASS

                                Tests RUN 2
                                Tests PASSED 2
                                Tests FAILED 0
                                Tests SKIPPED 0
  
```

(5) 参考

bspVal.sh, serLib.tcl, pkLib.c, bspPkCommonProc.tcl, envLib.tcl

16. serial

(1) 名称

serial——串行 I/O 测试。

(2) 提要

bspVal options serial

(3) 说明

此测试检测在不利情况下串行驱动器的性能。请参阅 bspVal.sh 的选项说明。整个测试需要几分钟的时间。



注意: 此参考条目只列出了测试失败最有可能的原因,但未列出所有的原因。

第一项测试重点测试串行控制端口,当输入信号来自于主机时,通过目标函数重复向控制台 tty 发送输出信号来实现。此测试检测目标机是否处于挂起状态,并且所有输入信

号是否全部被目标机接收。本测试在由主机和目标机支持的高速波特率下运行。若测试失败，请检验串行线路。串行驱动器的接收和传送设备必须稳定且正常地运转。总体地说，若两事件由同一个 ISR 处理时，驱动器必须在处理转移中断前处理接收中断。

第二项测试也重点测试了串行控制台端口。`pkConsoleEcho()` 函数把控制端口设置为回送模式，即所有从主机发出的信息很快就被回送。此测试保证，主机每次向控制台发送的信息返回主机时仍具有正确的顺序和数量。本测试在由主机和目标机支持的最高速波特率下运行。若测试失败，请检验串行驱动器可能发生中断的所有情况。

后面两项测试需要连接所有的空闲串行端口（除了用于控制台和 SLIP 的 ttys），并运行在回送的模式下。通常需要在传输/输出线到接收/输入线之间连接一个跳线。同时，也需要使用硬件交换器在 RTS 线和 CTS 线之间连接一个跳线。

第三个测试对回送模式下连接的每个串行端口发起一个 `pkLoopbackTest()` 任务。每个端口看见连续字节传送方式并在最大波特速率下接收。所有字节必须在顺序传送下接收。如果此测试失败了，检查是否正确设置了回送模式下所有测试的串行端口。确保串行驱动程序正确处理中断发送和中断接收。确保所有测试的串行通路都已被设置并初始化以正确值。

第四次测试为每个连接到回送模式地串行设备都发起几个 `pkLoopbackTest()` 任务。每个任务将一个特定的字母发送到端口。测试中每个串行端口都有多个任务。每个端口都应该接收到发送的所有字母。如果该测试失败，就应检查建议第三个测试检查的可能出现问题的地方。

第三和第四测试产生了对回送模式下连接的每个串行端口的单个或多个 `pkLoopbackTest()` 任务。这些任务同时运行，在主机和目标机都支持的最高速率下发送和接收字母。拥有较低 CPU 速率的目标板和许多串行设备可能不具有支持测试所需的初始处理功率。如果由于硬件限制使得回送测试失败，则可对测试进行设置以来减轻处理器的负荷。`Serial.tcl` 测试在宏 `bspVal` 规定的最高波特速率下运行，`BSPTEST_HOST_RATES`（提供了主机和目标机都支持的速率）。

修改 `BSPTEST_HOST_RATES` 以防最高共用波特速率使其在较低速下运行测试，从而可以尽量避免可能的相关硬件问题。另外 `pkLib.c` 常量 `PK_LOOP_TASK_NUM` 可以从默认值 4 改变为第四个测试下同时运行的任务减少的数量。在此情况下，目标代码文件 `pkLib.o` 必须在改变 `pkLib.c` 源文件的情况下重新建立。

配置有几个串行设备和较少数量本地 RAM（1M 比特或更低）的目标板在测试时可能发生存储器容量不足。在这种情况下，可以将目标机设置为使用外部存储器，或者减少执行任务的数量。

如果此测试过早的结束，可能要对目标板设置一个波特速率而不是 `T1_TIP_BAUD`。这种情况下，可能会使目标板需要上电期间复位来继续其他测试。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标板必须通过所有的有效 BSP 测试。

（4）配置参数：

BSPTEST_HOST_RATES: 主机支持的波特速率（必须）。

T1_SER_DEVICE: 主机使用的串行设备（必须）。

T1_SER_BAUD: 使用的默认串行波特速率（必须）。

例 J-14: 输出包括如下内容:

```

                        BSP VALIDATION TEST
                        -----
Target server          : t53-160
BSP                   : mv147
Log file              : /tmp/bspValidationLog.5219
Serial Test           :
console I/O test      : PASS
console echo test     : PASS
sequential loopback test : PASS
multiple loopback test : PASS

                        Tests RUN 4
                        Tests PASSED 4
                        Tests FAILED 0
                        Tests SKIPPED 0

```

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, envLib.tcl, serLib.tcl, pkLib.c

17. SysClock

(1) 名称

sysClock-系统时钟测试。

(2) 提要

```
bspVal options sysClock
```

(3) 说明

此测试检验了 BSP 系统时钟的性能。整个测试大约需要 7~10 分钟。请查阅 bspVal.sh 的选项说明。

基于此脚本的测试与使用 sysClkConnect() 函数将一个函数与系统时钟中断处理器相连, sysClkConnect() 函数使用 sysClkRoutine 函数指针变量。SysClkRoutine 的 sysClkConnect() 函数调用就会将 BSP 以前连接为的系统时钟 ISR 的任何程序断开,。



注意: 此参考条目仅列出了测试失败最有可能的原因,但是它并没列出所有的原因。

第一项测试到 4 检查了系统时钟在几种频率下的准确度：任意的特别速率、最小、最大和默认速率。为了测定系统时钟的速率，使用 `sysClkConnect()` 函数将一个简单反馈函数与系统时钟 ISR 相连接。此反馈函数在时钟跳跃时计数。然后此计数器清零，并且系统时钟将被赋予测试的速率。在 10 秒钟后读取计数器，下一次读取是 130 秒后。我们可用计数器值来计算系统时钟的平均中断速率。用三次测量来消除测试中的明显错误，其中 1% 的可能为时钟速率计算错误。若所测的时钟速率偏离测试值的 10% 以上，则测试速率失败。

如果有必要，则若任意一项测试失败，就应该检验定时芯片是否被正确初始化和具有正确的比例因数。必须在启动计数器之前在 `sysClkEnable()` 中激活中断。检查 `sysClkRoutine` 是否与 `sysClkConnect()` 相连接，以及每个系统时钟中断是否调用 `sysClkRoutine()`。

第五个测试检验了 `sysClkDisable()` 的运行。此测试是周期性的检查前四项测试中应用的计数器。调用 `sysClkDisable()` 后，此计数器不能再继续计数。如果此测试失败了，检查 `sysClkDisable()` 程序是否已经禁用了计时中断，关闭计时器，并将运行运行标志位设置为 `FALSE`。

第六项测试进行了对 `sysAuxClkRateSet()` 的参数检验。此测试检验了对于错误输入参数是否给出了适当的返回值。如果测试失败，在宏 `AUX_CLK_RATE_MIN` 和 `AUX_CLK_RATE_MAX` 下，检测 `sysAuxClkRateSet()` 是否运行了错误检测。

第七项测试检测了错误速率后通过的第一个有效速率时，`sysAuxClkRateSet()` 和 `sysAuxClkRateGet()` 的返回值。如果测试失败，则需检测 `sysAuxClkRateSet()` 是否设置了全部速率的变化，以及 `sysAuxClkRateSet()` 是否读入相同的变化值。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标板必须通过所有的有效 BSP 测试。

(4) 配置参数

`T1_EXTRA_SYSCLK`：另外需要测试的速率（可选）。

例 J-15：输出包括如下内容：

```

                                BSP VALIDATION TEST
                                -----
Target server                   : t53-160
BSP                             : mv147
Log file                        : /tmp/bspValidationLog.5219
System Clock Test               :
sysClk at 4321 hertz (rate = 4325 error = 0%)      : PASS
sysClk at 3 hertz (rate = 3 error = 0%)            : PASS
sysClk at 5000 hertz (rate = 5002 error = 0%)      : PASS
sysClk at 60 hertz (rate = 60 error = 0%)          : PASS
sysClkDisable() disables sysClk                    : PASS
sysClkRateSet() parameter checking                 : PASS
sysClkRateGet() return value                       : PASS

```

```
Tests RUN 7
Tests PASSED 7
Tests FAILED 0
Tests SKIPPED 0
```

(5) 参考

bspVal.sh, bspPkCommonProc.tcl, envLib.tcl, clockLib.tcl, pkLib.c

18. Timestamp

(1) 名称

timestamp-测试时间戳计时器。

(2) 提要

```
bspVal options timestamp
```

(3) 说明

此测试检验了 BSP 时间戳计时器的性能。整个测试大约需要 5 分钟。请查阅 bspVAL.sh 的选项说明。

为了运行此测试，目标必须配置时间戳计时器。我们可以在 BSP 的 config.h 文件中加入 INCLUDE_TIMESTAMP；如果使用工程工具，则可加入 INCLUDE_TIMESTAMP 组件。



注意：此参考条目列出了测试失败最有可能的原因，但是它并没列出所有的原因。

确认时间戳计时器没有运行，则可开始运行此脚本。如果发现时间戳计时器运行时，将调用 sysTimestampDisable() 函数，并且检验计时器是否已停止。

第一个测试检验 sysTimestampEnable() 是否在工作。测试调用该函数并且进行了一个短期测试来检验计时器是否在计数。

第二个和第三个测试检验了使用 sysTimestamp() 或 sysTimestampLock() 的长期时间戳操作。在相等时间间隔下，测试需要进行大量时间戳计时器的读取操作。T1_TMO_TIMESTAMP 的值指定了测试读取操作所需的时间。把每个时间间隔的时间戳点的数量与计算得出的数量相比较，如果计时器错误超过 1%，则测试失败。

第四个测试检验了 sysTimestampDisable() 是否在工作。测试调用了该函数并且检验了计时器是否已经停止计数。

第五个测试检验了计时器在被禁用后是否可被重新激活。测试调用了 sysTimestampEnable() 并且检验了计时器已经开始计数。

除了严重的硬件和软件缺陷以外（比如是 VxWorks 的可确认问题），目标板必须通过所有有效 BSP 测试。

(4) 配置参数

T1_BOOT_TN: 目标机名 (必须)。
 T1_TMO_TIMESTAMP: 长期测试持续时间 (可选)。
 T1_TMO_SLICES: 长期测试时间间隔 (可选)。

例 J-16: 输出包括如下内容:

```

          BSP VALIDATION TEST
          -----

Target server      : t53-160
BSP                : mv147
Second target server : t214-2
Second BSP         : mv147
Log file           : /tmp/bspValidationLog.6425
enable the timestamp timer      : PASS
sysTimestamp( ) long-term test  : PASS
sysTimestampLock( ) long-term test : PASS
disable the timestamp timer     : PASS
re-enable after disable        : PASS

          Tests RUN 5
          Tests PASSED 5
          Tests FAILED 0
          Tests SKIPPED 0
  
```

(5) 参考

bspVal.sh, timestampLib.tcl, pkLib.c

附录 K BSP 合法性检查表

此附录包含两个检查表：BSP 合法性检查表和 VTS 检查表。Wind River 公司在测试和验证新的 BSP 时使用这些检查表。

使用 BSP 合法性检查表作为在 BSP 产品各方面的检验指南：安装、测试和软件包的检验。使用 VTS 检查表来记录 BSP VTS 的测试结果。VTS 检查表仅仅是所有 BSP 合法性检查表中的一小部分。

如需运行这些检查表的拷贝，则显示如下 PDF 文件：

[host/src/test/bspVal/validation.cklist.pdf](#)

[host/src/test/bspVal/bspVts.cklist.pdf](#)

以下为检查表样式：

BSP 合法性检查表		
结构产品 ID:	BSP 产品 ID:	
结构补丁 ID:		
可选产品 1:		
可选产品 2:		
可选产品 3:		
功能测试:		
项 目	日期	状态
1. 将所有相关的补丁安装到结构产品的新副本上。		
2. 安装 BSP 的产品。		
3. 为 BSP 的安装目录作副本。为此 BSP 的方案的安装目录作副本。每个 BSP 的方案都将命名为 BSP_toolchain，并可在目录 target/proj 下找到它。		
4. 测试安装映像（基本性能的简单测试）。 这包括此方案目录下安装的所有映像。		
5. 重新建立 BSP 的组件。确保正确建立所有映像。然后比较新旧映像，以获得映像大小的任何变化。		
VxWorks.....		
vxWorks.sl.....		
vxWorks_rom.....		
vxWorks.st_rom.....		
vxWorks.res_rom.....		

vxWorks.res_rom.res_low.....		
bootrom.....		
bootrom_uncmp.....		
bootrom_res.....		
6. 重新建立所有 BSP 的方案功能程序的映像。对于 GNU 和 Diab 工具链，确保正确建立所有映像。比较新旧映像，以获得映像大小的任何变化。		
GNU		
VxWorks.....		
vxWorks_rom.....		
vxWorks_romCopy.....		
vxWorks_romResident.....		
Diab		
VxWorks.....		
vxWorks_romCopy.....		
vxWorks_romCopy.....		
vxWorks_romResident.....		
建立测试		
项 目	日期	状态
1. 对于以前没有建立和测试的所有标准映像，进行建立并且测试。验证：在标题 SPECIAL CONSIDERATIONS 下，已经把所有未能建立或不能运行的映像 in target.nr 中进行标记。		
vxWorks		
vxWorks.st.....		
vxWorks_rom.....		
vxWorks.st_rom.....		
vxWorks.res_rom.....		
vxWorks.res_rom_nosym.....		
vxWorks.res_rom.res_low.....		
vxWorks.res_rom_nosym.res_low.....		
bootrom.....		
bootrom_uncmp.....		
bootrom_res.....		
bootrom_res_high.....		
2. 建立并且测试方案功能程序的 GNU 工具链的下列映像。验证：在标题 SPECIAL CONSIDERATIONS 下，已经把所有未能建立或不能运行的映像 in target.nr 中进行标记。		
vxWorks		
vxWorks_rom		
vxWorks_romCopy		
vxWorks_romResident		

3. 建立并且测试方案功能程序的 Diab 工具链的下列映像。验证： 在标题 SPECIAL CONSIDERATIONS 下，已经把所有未能建立或不能运行的映像 in target.nr 中进行标记。 vxWorks..... vxWorks_rom..... vxWorks_romCopy..... vxWorks_romResident.....		
4. 检查 BSP 中所有 C 语言文件的内联可编译宏 <code>_asm_</code> 。而使用此宏的所有文件也都必须使用新的宏 <code>_WRS_ASM</code> 。请查阅《BSP 开发指南》：通用 C 程序标准。		
产品封装测试		
项 目	日期	状态
1. 为由 BSP 产品安装的文件建立列表。		
2. 验证：BSP 没有提供 WRS 普通驱动程序的头文件、源文件和对象模块。		
3. 验证：由 BSP 产品安装的文件中不包括相关文件 <code>depengd.bspname</code> 。		
4. (可选) 验证：BSP 没有提供不必要的文件，例如：在 BSP 或者方案目录下的对象文件、 <code>ctdt.c</code> 和 <code>symTbl.c</code> 。		
5. 检查 BSP 的版本是否正确。有效 BSP 版本分别是： VxWorks 5.2 BSP 或者更早的版本 -1.0/x Tornado 1.0 和 1.0.1 BSP 版本-1.1/x Tornado 2.X BSP 版本 -1.2/x 其中 x 表示 BSP 修订版本号。 验证：BSP 目录下的 README 文件的版本是正确的。 验证： <code>config.h</code> 文件中的宏 <code>BSP_VERSION</code> 和 <code>BSP_REV</code> 也对应了相同的 BSP 版本。		
6. 验证：在 <code>target/h</code> 、 <code>target/src/drv/</code> 、 <code>target/config/all</code> 和 <code>target/src/config</code> 目录下没有第三用户文件。		
7. 验证：BSP 所有的特殊驱动程序都是 <code>MACH_EXTRA</code> 和 <code>sysLib.c</code> 的一部分。		
8. 验证：BSP 不包含有版权的文件和 Wind River 公司所限制的文件。 Wind River 公司并没有对用户公开 Ethernet 和 SCSI 驱动程序的源代码。		
配置测试		
项 目	日期	状态
1. 测试由默认 BSP 配置支持的引导程序。安装并引入引导 ROM，检查是否可用引导程序下载映像。 对于每个引导程序，尝试下载一个无效的 VxWorks 映像，然后验证：在尝试失败以后，此引导仍可下载并执行一个无效映像。选用一独立的表格，详述引导程序的测试及结果。		

2. 基于 target.nr 的信息, 建立并引导不同的 BSP 配置, 它包括一些 BSP 配置所没有提供的特性。选用一独立的表格, 详述不同配置下的测试, 哪些特性被隐藏了, 哪些没有, 以及测试结果。 依据第一步测试每个引导程序。		

BSP 合法性测试套件		
项 目	日期	状态
1. 验证: 所有应用测试都成功了, 附加一张 VTS 检查表的打印输出。		
2. 如果在此 BSP 下的测试没有成功, 则检验其是否在 target.nr 中被标记。		

可选产品测试		
项 目	日期	状态
1. 如果 BSP 已经支持 USB, 对这个 BSP 进行 USB 支持测试。按照下述步骤完成这项工作。使用工程工具来配置 USB。有关 USB 的更多信息, 请参见 USB 资料。 工程工具配置 —选择 USB 主堆栈组件 —选择至少一个主机控制器, OHCI 和 UHCI 则依赖于 BSP 的支持 —如果硬件允许的话, 带上鼠标或者键盘设备组件 注意: 不要包括任何初始化组件。如果希望使用目标机 shell 来测试, 请确保目标机可以使用符号表格。 USB 测试 —引导目标机, 并包括对 USB 的支持。 —在目标机或者主机 shell 中, 通过调用 usbTool() 命令激活 USB 工具。这样就产生了 usb 工具提示符。在提示符下运行下列命令: <pre>usb>usbic(初始化代码)</pre> <pre>usb>attach ohci(或者是 uhci, 取决于配置)</pre> 这样可以保证在目标机中, USB 得到了适当的配置。 如果有 USB 键盘或 USB 鼠标设备, 按下述方式测试它们: 鼠标测试: —将鼠标设备连接到一个端口。 —通过检查总线, 确保设备已经被检测到。在 usb 工具中调用 usbnum 命令。 —在 usb 工具中调用 mousetc 命令来运行鼠标测试。在鼠标运动时应该显示出鼠标的坐标。		

键盘测试 —将键盘连接好，通过在 <code>usb</code> 工具中调用 <code>kbdi</code> 来初始化键盘。 —调用 <code>kbdpo</code> 命令来运行键盘测试，在 <code>shell</code> 中应该显示敲在键盘上的字符。																										
2. 如果 BSP 支持 TrueFFS，测试 BSP 的 TrueFFS 支持。按照下述步骤，完成这个工作。关于 TrueFFS 的更多信息，请参见 TrueFFS 资料。可以使用命令行或者工程工具来测试 TrueFFS。 命令行配置： —在 <code>config.b</code> 中定义 <code>INCLUDE_TFFS</code>。 —使用 MTD 和 TL 组件来选择合适的介质和变换层 —如需要 TrueFFS 显示程序，选择 <code>INCLUDE_SHOW_ROUTINES</code> 方案功能程序配置： —配置 <code>INCLUDE_TFFS</code> 组件。 —使用 MTD 和 TL 组件来选择合适的介质和变换层。 —如需要 TrueFFS 显示程序，选择 <code>TFFS_SHOW</code> 组件。 测试 TrueFFS： —通过 TrueFFS 配置引入目标。 —在目标或者主机下运行 <code>sysTffsFormat()</code> 或 <code>tffaDevFormat()</code>。 —运行 <code>usrTffsConfig()</code>。 —把文件从主机复制到所建立的显示装置，并检查复制操作。																										
3. 为使用串行后置，通过串行通信和目标服务器来配置 WDB。使用正确的串行线路来合理的设置目标，并打开 <code>windsh</code> 或使用 <code>i</code> 和 <code>version</code> 这样的命令来测试 WDB 线路。																										
目标信息工作表 <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 60%;">项 目</th><th style="width: 20%;">日期</th><th style="width: 20%;">状态</th></tr> </thead> <tbody> <tr> <td data-bbox="212 1976 1381 2148">1. 将 BSP 目标信息工作表附到此检查表，并打印。一般来说，此工作表在 <code>target.nr</code> 中，而一些开发者则可能将它放在其他文件中。请查阅《BSP 开发指南》：目标信息参考页面： <code>target.nr</code>。</td><td data-bbox="1381 1976 1633 2148"></td><td data-bbox="1633 1976 1877 2148"></td></tr> <tr> <td data-bbox="212 2148 1381 2258">2. 为 BSP 建立文本。验证：文本是否完整，并且符合 Wind River 公司的文本标准。并附上文本的打印输出。</td><td data-bbox="1381 2148 1633 2258"></td><td data-bbox="1633 2148 1877 2258"></td></tr> <tr> <td data-bbox="212 2258 1381 2368">3. 验证：为此 BSP 附上的文本包括经销、工程技术和联系方式（e-mail 和电话）。</td><td data-bbox="1381 2258 1633 2368"></td><td data-bbox="1633 2258 1877 2368"></td></tr> <tr> <td data-bbox="212 2368 1381 2478">4. 添加注释，以解释在测试中标记的所有偏移量的类型和程度。</td><td data-bbox="1381 2368 1633 2478"></td><td data-bbox="1633 2368 1877 2478"></td></tr> <tr> <td data-bbox="212 2478 1381 2558">BSP 合法性： 通过： 不合格：</td><td colspan="2" data-bbox="1381 2478 1877 2558">完成检查表：</td></tr> <tr> <td data-bbox="212 2558 1381 2638">工程师：</td><td colspan="2" data-bbox="1381 2558 1877 2638">日期：</td></tr> <tr> <td data-bbox="212 2638 1381 2718">签名：</td><td colspan="2" data-bbox="1381 2638 1877 2718"></td></tr> </tbody> </table>			项 目	日期	状态	1. 将 BSP 目标信息工作表附到此检查表，并打印。一般来说，此工作表在 <code>target.nr</code> 中，而一些开发者则可能将它放在其他文件中。请查阅《BSP 开发指南》：目标信息参考页面： <code>target.nr</code> 。			2. 为 BSP 建立文本。验证：文本是否完整，并且符合 Wind River 公司的文本标准。并附上文本的打印输出。			3. 验证：为此 BSP 附上的文本包括经销、工程技术和联系方式（e-mail 和电话）。			4. 添加注释，以解释在测试中标记的所有偏移量的类型和程度。			BSP 合法性： 通过： 不合格：	完成检查表：		工程师：	日期：		签名：		
项 目	日期	状态																								
1. 将 BSP 目标信息工作表附到此检查表，并打印。一般来说，此工作表在 <code>target.nr</code> 中，而一些开发者则可能将它放在其他文件中。请查阅《BSP 开发指南》：目标信息参考页面： <code>target.nr</code> 。																										
2. 为 BSP 建立文本。验证：文本是否完整，并且符合 Wind River 公司的文本标准。并附上文本的打印输出。																										
3. 验证：为此 BSP 附上的文本包括经销、工程技术和联系方式（e-mail 和电话）。																										
4. 添加注释，以解释在测试中标记的所有偏移量的类型和程度。																										
BSP 合法性： 通过： 不合格：	完成检查表：																									
工程师：	日期：																									
签名：																										
1. 将 BSP 目标信息工作表附到此检查表，并打印。一般来说，此工作表在 <code>target.nr</code> 中，而一些开发者则可能将它放在其他文件中。请查阅《BSP 开发指南》：目标信息参考页面： <code>target.nr</code> 。																										
2. 为 BSP 建立文本。验证：文本是否完整，并且符合 Wind River 公司的文本标准。并附上文本的打印输出。																										
3. 验证：为此 BSP 附上的文本包括经销、工程技术和联系方式（e-mail 和电话）。																										
4. 添加注释，以解释在测试中标记的所有偏移量的类型和程度。																										
BSP 合法性： 通过： 不合格：	完成检查表：																									
工程师：	日期：																									
签名：																										

合法性测试套件检查表		
VxWorks 版本:		BSP 版本:
生产商:		模型:
部件号:		序列号:
修订版:		条形码标签:
用于测试的 PROMS		
PROMS	套接字 ID	总结
0		
1		
2		
3		
附注:		
工程师:		日期:
签名:		

合法性测试套件检查表		
生产商:	模型:	
测试:	日期	检查
1. AuxClock.tcl		
特殊测试的频率下的 Auxclock		
最小测试的频率下的 Auxclock		
最大测试的频率下的 Auxclock		
默认测试的频率下的 Auxclock		
sysAuxClkDisable() 测试		
sysAuxClkRateSet() 参数检查		
sysAuxClkRateGet() 返回值		
2. baudConsole.tcl		
150 波特控制台		
300 波特控制台		
600 波特控制台		
1200 波特控制台		
1800 波特控制台		
2400 波特控制台		
4800 波特控制台		
9600 波特控制台		
19200 波特控制台		
38400 波特控制台		
3. bootline.tcl		
控制/X 测试		
本地错误地址的总线错误测试		
板外错误地址的总线错误测试		
引导命令测试		
4. busTas.tcl 从属:		
短期 TAS 测试		
长期 TAS 测试		
5. eprom.tcl		
1 比特 eprom 读取		

续表

合法性测试套件检查表		
生产商:	模型:	
测试:	日期	检查
2 比特 eeprom 读取		
4 比特 eeprom 读取		
6. error1.tcl		
本地地址的总线错误测试		
板外地址的总线错误测试		
0 间隔测试		
7. error2.tcl		
严重错误的测试		
有中断的重新启动		
8. model.tcl		
sysModel() 返回值		
9. network.tcl 从属:		
脉冲信号目标测试		
UDP 包返回测试		
10. nvRam.tcl		
引导序列的 sysNvRamGet()		
补充引导序列的 sysNvRamSet() 和 sysNvRamGet()		
长度为零的 sysNvRamGet()		
SysNvRamSet() 参数检查		
SysNvRamGet() 参数检查		
数据 0xff 的 sysNvRamSet() 和 sysNvRamGet()		
数据 0x00 的 sysNvRamSet() 和 sysNvRamGet()		
引导序列的 sysNvRamSet() 和 sysNvRamGet()		
11. procNumGet.tcl		
sysProcNumGet() 返回值		
12. ram.tcl		
1 比特 RAM 读取		
2 比特 RAM 读取		
4 比特 RAM 读取		

续表

合法性测试套件检查表		
生产商:	模型:	
测试:	日期	检查
13. rlogin.tcl 从属:		
当 rlogin 到目标 0 时, 出现 IN USE 信息		
目标 0 的搜索		
目标 0 下 login pty 使用法		
目标 0 下的 logout		
当 rlogin 到目标 1 时, 出现 IN USE 信息		
目标 1 的搜索		
目标 1 下 login pty 使用法		
目标 1 下的 logout		
14. scsi.tcl		
scsiShow() 测试		
SCSI 写入/读取/验证 设备		
15. serial.tcl		
控制台 I/O 测试		
控制台返回测试		
连续回送测试		
多路回送测试		
16. sysClock.tcl		
特殊频率的 sysClk		
最小频率的 sysClk		
最大频率的 sysClk		
默认频率的 sysClk		
sysClkDisable() 禁用 sysClk		
sysClkRateSet() 参数检查		
sysClkRateGet() 返回值		
17. timestamp.tcl		
启用时标计时器		
sysTimestamp() 长期测试		
sysTimestampLock() 长期测试		

续表

合法性测试套件检查表		
生产商:	模型:	
测试:	日期	检查
禁用时标计时器		
重新启用		
18. 确认底板类型测试:		
登记		
总线中断		
信箱		
注解 (参考以上的项目号):		

附录 L Refgen



注意：在 Tornado1.0.1 和更早的版本中，UNIX 帮助页面产生于使用 `mg` 和 `mangen` 两种脚本的源程序。在 Tornado 2.0 中，将不再分配 UNIX 帮助页面，它们由 HTML 文件代替，HTML 文件产生于一种被称作 `refgen` 的新 Tcl 脚本，就像其前期产品一样，它是由 `make` 命令控制的，并作为建立过程的一部分。

1. 名称

`refgen`—Tornado 参考文件的生成工具。

2. 提要

```
refgen [-book bookName] [-chapter chapterName] [-config configFile]
[-cpp] [-expath pathList] [-exbase basedir] [-h] [-int]
[-l logFile] [-mg] [-out outDir] [-verbose] fileList
```

3. 说明

此工具执行了源程序文件摘要的表驱动进程。输入表定义了“*meta-syntax*”，其详述了如何将文档嵌入到使用特殊的编程语言实现的源文件中。同样的，输出表定义了输出文件的细节。

4. 总体约定

有些关于文档如何嵌入至源代码的规定并不取决于源程序语言，因而不需要改变配置表。

■ 输入约定概要。

函数都将被组织成库的形式，并且每个库都以 `DESCRIPTION` 段开始。如果不是一段 `DESCRIPTION` 开头，则该描述段将被放在修改历史纪录后的第一个说明程序块中。有些输入程序语言（比如 `shellscript`）可能以 `SYNOPSIS` 段作为开头。

源文件库的第一行是下列格式的一行标题：

```
sourceFileName - simple description
```

这就是说，此行（在要求的提示开始字母之后）是由其文件名开头的，并且由一个空格、一个连字符和另一空格与库的简单描述分开。函数描述的第一行是相同格式的一行标题。

无论有没有确切的给出 **DESCRIPTION** 标志, 描述程序都将在函数标题行下面开始。段标题是由大写的字符组成的, 起始于一个新的行, 并且结束于一个新的行或结束于一个冒号。

斜体显然包括文本变量——也就是说, 文件中不是逐字打印的, 不过, 反过来就是说可以由使用的每个例子中的一些特定的值代替——在源文件中, 将用角括号标出。因此, 如要产生输出 `textVar`, 则需键入 `<textVar>`。

黑体字的使用: 一般结构: 在源文件中, 用单引号。例如, `'sysIntIdtType'` 代表 `sysIntIdtType`. **Special**. 关键字: 以 `Lib` 结尾或者有文件名后缀的将自动的以黑体表示。例如, `fileName.c`, `object.o` 和 `myStuff.tcl` 都以黑体字表示。

简单的列表可以在源文件中通过缩进空白缩进行来构造 (如果在一个特殊的源程序语言中要求使用说明字符的话, 可以使用说明字符来缩进)。例如:

```
line one
line two
```

■ 输出约定概要。

记录库一开始就是其中程序的提要, 它是由所有程序的标题行构成的。

对于一些输入程序语言, 也将自动为其提供一个 **SYNOPSIS** 段, 它是从输入 `meta-syntax` 表规定的独立语言方式的定义程序中提取的。而不支持它的输入程序语言对于 `$SYNTAX (declDelim)` 是空的; 在此语言中, **SYNOPSIS** 段必须作为子程序的说明部分而明确给出。例如, 对于所有的组件和 `Tcl` 都必须有明确的 **SYNOPSIS** 段接在 **DESCRIPTION** 后面。下列的例子给出了正确的组件标志:

```
* SYNOPSIS
* \cs
* void xxxMemoryInit(void)
* \ce
```

对于一些程序语言, 程序定义也自动的用来建立一个 **PARAMETERS** 段。

假定文档的在线形式适合一种包括原始文件和程序索引的结构。则此库中的一些程序需要为这些文件命名或标记, 这样就可以在输出中包含它们。在此库中的程序执行时, 这些原始文件和程序索引是不需要实际存在的。

无论在源文件中是否提供了 **DESCRIPTION** 标志, 它在所有描述段中都是自动提供的。

无论在源文件中是否给出 **SEE ALSO** 段, 它在程序描述的输出中通常都是给出的。程序描述的 **SEE ALSO** 段将自动包括所包含的库的索引。

5. 输出指令

下列的关键字通常由大写字母组成, 并且出现在每行的开始, 并改变输出的文本。一

些指令在相同行中将采用特殊形式的变量。

■ NOROUTINES

表明子程序文件将不能产生（必须在库段中显示）。

■ NOMANUAL

无论是一个完整的程序文件还是库文件，只要程序段出现就删除它。可以通过特殊语言来删除程序文件，这种语言是通过在 LOCALdecls 表中匹配正规表达式来规定的。请查阅 refDocGen，用命令行选项来覆盖它。

■ INTERNAL

删除程序段。请查阅 refDocGen，用命令行选项来覆盖它。

■ APPEND filename

包括输出的 *filename* 文档，就好比将它的源文件添加至包含 APPEND 关键字的文件。

6. 明确的标志

为了支持大部分普通输出，refgen 应用程序必须支持源文件中的简单标志语言。

下列的列表概括了 refgen 的确切标志：

附 注	标 志	描 述
	\ " text to end of line	文件说明
	'text...' or 'text ...'	黑体字
	<...>	斜体字
[1]	\\ or \ /	\ 字符
[2]	\< \> \& \'	<>和'字符
	\\	表格内的\字符
[3]	\ss ... \se	预先格式化的字符
	\cs ... \ce	书面语（字符）
	\bs ... \be	小规模展示书面语
[4]	\is \i ... \i ... \ie	项目清单
[5]	\ml \m ... \m ... \me	标记清单
[6]	\ts ... \te	一张表格
	\h text	副标题
	\b reference	引用后换新行

7. 标志的附注

[1] 输入反斜杠\是很容易辨认的，但是在其他文本中使用反斜杠符号会有其他作用，这就可能产生混淆。而这时就需要使用 \，通常使用 \ 是安全的，不会引起混淆。

[2] 必须键入<and>，以识别嵌入的 HTML 标志。必须键入&，以识别 HTML 的字符。必须键入单引号，以识别 refgen 自动的粗体字符。

[3] 在\ss 和\se 之间必须保留空白和新的一行，不过格式不是这样就会无效。这些对于范例中的文本变量将起到作用。

[4] \is 和\ie 标记了项目列表的起始和结尾。仅仅在\is 和\ie 之间支持\i，它标记了项目的开始：也就是说，它使得一段结束，并且消除了空余的部分。

因此，示例如下：

```
\is
\i boojum
A boojum is a rare
tree found in Baja
California.
\i snark
A snark is a different
matter altogether.
\ie
```

输出大概如下列所示：

```
boojum
boojum 是在 Baja California 发现的一种罕见的树。
snark
snark 是一种完全不同的事物。
```

[5] \ml 和\me 为标记列表确定了界限：它们与输出时的详细列表不同：除\m 之外，所有标记在同一行中出现，并作为下节的开始。

[6] 尽管相对于初始的 `mangen` 表格语句有些简化，但在\ts 和\te 之间仍然都是特殊语句。

三个重要的规定：

表格必须有一个标题段和主体段：这通常由仅包括字符`-`，`+`，`|`和水平空白的一行来分界。

无论是在标题还是主体中，表格的单元格都是由字符`|`来分界的。也可能用`\`来表示字符`|`。

新的一行表示了标题或主体中一行的结束。

例如，下列规定了一个只有一个标题行而具有三列的表格：

```
\ts
Key | Name          | Meaning
----|-----
\& | ampersand      | bitwise AND
\|  | stile          | bitwise OR
#   | octothorpe     | bitwise NAND
\te
```

在源文件中，单元格并不需要对准，不过这样做可方便阅读（也更容易在编辑时避免错误）。

以上的输入将会产生表格：

符号	名称	意义
&	"&"符号	AND 比特
	" "符号	OR 比特
#	"#"符号	NAND 比特

8. 参数

-book

可通过此选项定义有效程序或者库属于哪本书。默认的是《Wind River 系统参考手册》。

-chapter

与-book 选项相关，可通过此选项设置有效程序段或者库。默认的设置是《Wind River 系统参考程序库》。

-category

与-book 选项相关，此选项允许设置有效程序和程序库的类别。例如，它可用来区别不同主机平台下的可用程序。

-config configname

如果给出此参数，则从 configname 文件中阅读配置信息；默认的配置是 C2html。

-cpp

此选项规定：所列出的文件将被认为是 C++ 文件。然后将运行特别的程序来恢复所有的类型成分。

-expath pathList

在明确使用冒号间隔路径名的文件中，此选项将预先占用 EXPANDPATH 设置记录。

-exbase basedir

为了简化在不完全树形结构下的工作，此选项使用 basedir 而不是 WIND_BASE 作为扩展文件搜索的引导。

-int

如果给出了此参数，它对所有的可用文件，即使标记 INTERNAL 或 NOMANUAL 的文件，甚至 local 程序进行格式化。

-l logfile

此选项规定使用 logfile 来记录 refgen 错误。如果没有指定-l 选项，则使用标准输出。

-mg

将标记 mangel 改为 “on the fly”。

-out outputDirectoryName

将输出文本文件保存在 outputDirectoryName。

-verbose

显示可靠信息以表明有事件的发生。

SourceFileList

使用其他任何参数作为源文件名，从中可摘录和格式化文本。

例： 产生 BSP sysLib.c 文件的 HTML 参考页面：

```
% cd $WIND_BASE/target/config/myBsp
% refgen -mg -book BSP_Reference -chapter myBsp -category myBsp \
-out $WIND_BASE/vxworks/bsp/myBsp sysLib.
```

9. 包括

refLib.tcl

10. 参考

《VxWorks 程序员指南》：C 语言指令

htmlLink, htmlBook, windHelpLib

附录 M BSP 产品目录

当完成了 BSP，产品目录应该与此附录给出的目录相同。BSP 仅包含关联特殊类型目标板的文件。此列表仅仅是一个例子，而并非所需要的或不需要的文件的最终列表。

下列是传统的 BSP 索引目录：

```
target/config/bspname/Makefile
target/config/bspname/README
target/config/bspname/bootrom
target/config/bspname/config.h
target/config/bspname/bspname.h
target/config/bspname/romInit.s
target/config/bspname/sysALib.s
target/config/bspname/sysScsi.c
target/config/bspname/sysLib.c
target/config/bspname/sysSerial.c
target/config/bspname/target.nr
target/config/bspname/vxWorks
target/config/bspname/vxWorks.st
target/config/bspname/vxWorks.sym
```

包含了相应类型的特殊 BSP 的任何驱动程序：源文件、目标文件和库文件。不包括 Wind River 限制网络和 SCSI 驱动程序的源文件：

```
target/config/bspname/specialDrvr.c
target/config/bspname/specialDrvr.obj
target/config/bspname/specialLib.a
下列文件是新的 HTML 文本页面：
docs/vxworks/bsp/bspname/*.html
docs/vxworks/bsp/bspname/*.lib
docs/vxworks/bsp/bspname/*.rtn
```

下列文件表示了 BSP 的默认工程：

```
target/proj/bspname_gnu/Makefile
target/proj/bspname_gnu/linkSyms.c
target/proj/bspname_gnu/prjObjs.lst
target/proj/bspname_gnu/bspname_gnu.wpj
```

target/proj/bspname_gnu/prjComps.h
target/proj/bspname_gnu/prjParams.h
target/proj/bspname_gnu/prjConfig.c
target/proj/bspname_diab/usrAppInit.c
target/proj/bspname_diab/Makefile
target/proj/bspname_diab/linkSyms.c
target/proj/bspname_diab/prjObjs.lst
target/proj/bspname_diab/bspname_diab.wpj
target/proj/bspname_diab/prjComps.h
target/proj/bspname_diab/prjParams.h
target/proj/bspname_diab/prjConfig.c
target/proj/bspname_diab/usrAppInit.c

下列文件表示了工程的默认编译输出:

target/proj/bspname_gnu/default/vxWorks
target/proj/bspname_gnu/default_rom/vxWorks_rom
target/proj/bspname_gnu/default_romCompressed/vxWorks_romCompressed
target/proj/bspname_gnu/default_romResident/vxWorks_romResident
target/proj/bspname_diab/default/vxWorks
target/proj/bspname_diab/default_rom/vxWorks_rom
target/proj/bspname_diab/default_romCompressed/vxWorks_romCompressed
target/proj/bspname_diab/default_romResident/vxWorks_romResident

不包括任何下列的 BSP 产品:

target/config/bspname/depend.bspname
target/config/all/anything
target/config/comps/anything
target/h/anything
target/src/anything
target/lib/objARCHgnu/vx/anything

在 Tornado2.2 下, Wind River 公司不再使用包括任何普通 BSP 产品的中间目标文件, 这一改变将会降低分布式 BSP 的大小。第三用户可继续提供中间目标文件 (*.o), 以作为 BSP 的分布。

读者建议反馈表

- 1、姓名_____2、性别_____3、年龄_____4、电话_____
- 5、单位_____6、职务/职称_____
- 7、通信地址_____邮编_____
- 8、电子信箱_____单位网站_____
- 9、您的文化程度: ☐ 中专以上 ☐ 大专 ☐ 本科 ☐ 研究生以上
- 10、您所学专业: ☐ 通信工程 ☐ 电子工程 ☐ 计算机 ☐ 自动化 ☐ 制导控制
- 11、您所在行业: ☐ 通信网络 ☐ 工控设备 ☐ 视听设备 ☐ 电脑外设 ☐ 数字消费品
☐ 航空航天 ☐ 高等院校 ☐ 通信网络 ☐ 高级智能机械装置
- 12、您的工作性质: ☐ 设计开发 ☐ 大学教学 ☐ 普通培训 ☐ 学生
- 13、您目前使用哪种嵌入式操作系统、哪种 32 位芯片

14、您认为风河公司应该为用户提供哪些指导和服务

15、您对本书的建议和意见

16、您今后需要哪些关于 VxWorks 的图书

表格填好后请寄:

有关 VxWorks 产品或技术问题请按照以下方式联系:

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, 41 技术支持或销售 传真: 010-84603644
公司网站: www.windriver.com 公司电子信箱: support-china@windriver.com

有关本丛书的建议和意见或邮购本丛书请按照以下方式联系:

北京清华大学校内金地公司 VxWorks 编辑部 收 邮编: 100084
电话: 010-62788951-215 邮购电话: 010-62770384 传真: 010-62788903

有关嵌入式软硬件、DSP、EDA、单片机等工控电子类写作意向,
请按照以下方式联系:

曾刚 电子信箱: zengg@tup.tsinghua.edu.cn 电话: 010-62788951-215

VxWorks 开发人员指南丛书

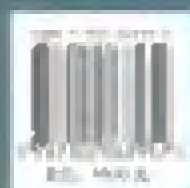
第1卷：第1版：第1版：第1版：第1版：第1版

《VxWorks 程序员指南》

《VxWorks 网络程序员指南》

《Tornado 用户指南》

《VxWorks 5.2.0 开发人员指南》



定价：44.00元