Vxworks

设备驱动

中国科学技术大学

（安徽，合肥）

近代物理系

快电子实验室

曹桂平（著）

This page is intentionally left blank.

前言

Vxworks 是较为常用的嵌入式硬实时操作系统，在很多领域都有其应用的身影，然而由于 Vxworks 操作系统源代码不公开，虽然文档中对各种驱动设计都有说明，但是实际中还是会遇到很多问题。本书根据作者的一些驱动经验，结合 Wind River 提供的一些官方文档和开发环境下的源代码写成，对 Vxworks 各种类型设备驱动都做了比较详细的介绍和分析，可以作为 Vxworks 下设备驱动设计的指南。另外对于各种类型的 Vxworks 启动方式以及映像文件组成进行了较为详细的分析，将澄清 Vxworks BSP 开发者具有的很多疑问。

本书内容

本书共分为 11 章，各章内容如下：

第一章简单介绍了嵌入式系统，并对 Vxworks 操作系统特性进行了简单的说明。

第二章对 Vxworks 操作系统几个主要组成方面进行了比较详细的介绍，包括任务，任务调度，任务间通信，内存管理，中断处理。我们不做翻译官方文档的工作，结合作者经验有感而发。

第三章对 Vxworks 内核映像类型以及启动方式进行了详细分析，并对下载方式中使用的 bootrom 进行了较为深入的分析和介绍，此后对 Vxworks 操作系统启动过程进行了梳理。本章将澄清读者对 Vxworks 启动方面的很多疑问。

第四章讨论了驱动程序的基本功能和结构，对驱动程序中常用的一些策略以及注意事项进行了介绍。

第五章介绍了 Vxworks 设备驱动的内核结构层次，着重对 IO 子系统及其维护的三张系统表进行了讨论，并对 Vxworks 下已有的几个较为常用的驱动以代码示例的方式介绍了其使用方法。

第六章开始进入具体设备驱动的设计，在本章中我们将从结构层次最为简单的普通字符设备驱动开始讲起，以一个 SPI 接口驱动代码为例，着重讨论了普通字符设备驱动的结构，设计方式和具体实现。

第七章对串口驱动设计和实现进行了详细的分析。串口也是字符设备的一种，由于其常用性，Vxworks 内核专门提供了 TTY 中间层来提高串口驱动设计的效率，降低串口驱动设计的复杂度。

第八章进入到第二大类设备-块设备驱动的设计和实现的分析中。我们将从数据结构出发，分析块设备驱动的基本结构，进而讨论其具体实现。Vxworks 下块设备驱动工作方式比较特殊，其使用阻塞读写方式，不同于通用操作系统下中断读写方式，这与 Vxworks 特殊的工作环境有关。

第九章我们将对 FLASH 设备驱动进行详细介绍。Flash 设备是嵌入式平台上最为常见和常用的设备，用以存储操作系统内核映像和用户数据。本章将以 Vxworks 内核提供的 TrueFFS 中间层进行展开，分析 Flash 设备驱动涉及的各个方面。

第十章进入到第三大类设备-网络设备驱动的设计和实现中。网络设备由于其独特的工作方式，其内核驱动层次不同于其他两类设备（字符，块设备），其不属于 IO 子系统管理，而是直接工作在内核网络栈实现下，为了简化网络设备驱动设计的复杂度，Vxworks 提供 MUX 中间层，在该层次下实现的驱动通常被称为增强型网络驱动，本章同样也是从数据结构出发，以实际项目中使用的网口驱动代码为例，逐步完成对网络设备驱动的设计和实现。

第十一章分析了 USB 设备驱动的设计和实现。本章首先对 USB 本身进行了详细的介绍，之

后对我们要驱动对象进行了澄清，一般而言，USB 设备驱动指的是对 USB 主机或者目标机控制器的驱动，这个驱动由于与内核 USB 栈耦合较紧密，故必须对内核 USB 栈的实现有一个很清楚的了解才能成功完成 USB 主机控制器的驱动开发。本章首先跟随一个 USB 类驱动层读数据请求，对请求在内核 USB 栈中的传递路径进行了跟踪，对路径上调用的关键函数以及使用的数据结构进行了较为详细的分析和介绍，之后以 Mass Storage 类驱动为例，介绍了类驱动的初始化过程，并以 UHCI 控制器驱动为例，介绍了主机控制器驱动的初始化过程，之后总结出了 USB 主机控制器的驱动结构，并给出了驱动中两个中心函数的实现框架。

中国科学技术大学
曹桂平
2010 年 06 月

目 录

第一章 嵌入式系统概述

本章首先从嵌入式系统的定义，组成和特点，发展趋势三个方面简单的对嵌入式系统进行了介绍，之后对实时操作系统特点进行了说明，并从内核结构的角度介绍了微内核和宏内核结构的区别，最后我们对嵌入式操作系统 Vxworks 进行了简单的讨论。

1.1 嵌入式系统定义

嵌入式系统的定义有很多，比较通用的定义如下：以应用为中心，以计算机技术为基础，软硬件可裁剪，迎合特定应用环境，对功能，可靠性，成本，体积，功耗方面要求严格的专用计算机系统。
按照电器工程协会（IEEE）的定义，嵌入式系统是用来控制、监控、或者辅助操作机器、装置、工厂等大规模系统的设备（devices used to control, monitor, or assist the operation of equipment, machinery or plants）。这个定义主要是从嵌入式系统的用途方面来进行定义的。

1.2 嵌入式系统组成和特点

根据以上嵌入式系统的定义，我们可以看出，嵌入式系统是由硬件和软件相结合组成的具有特定功能、用于特定场合的独立系统。其硬件主要由嵌入式微处理器、外围硬件设备组成；其软件主要包括底层系统软件和用户应用软件组成。嵌入式系统具有如下特点：
（1）专用、软硬件可剪裁可配置
从嵌入式系统定义可以看出，嵌入式系统是面向应用的，和通用系统最大的区别在于嵌入式系统功能专一。根据这个特性，嵌入式系统的软、硬件可以根据需要进行精心设计、量体裁衣、去除冗余，以实现低成本、高性能。也正因如此，嵌入式系统采用的微处理器和外围设备种类繁多，系统不具通用。
（2）低功耗、高可靠性、高稳定性
嵌入式系统大多用在特定场合，要么是环境条件恶劣，要么要求其长时间连续运转，因此嵌入式系统应具有高可靠性、高稳定性、低功耗等性能。
（3）软件代码短小精悍
由于成本和应用场合的特殊性，通常嵌入式系统的硬件资源（如内存等）都比较少，因此对嵌入式系统设计也提出了较高的要求。嵌入式系统的软件设计尤其要求高质量，要在有限资源上实现高可靠性、高性能的系统。虽然随着硬件技术的发展和成本的降低，在高端嵌入式产品上也开始采用嵌入式操作系统，但其和 PC 资源比起来还是少得可怜，所以嵌入式系统的软件代码依然要在保证性能的情况下，占用尽量少的资源，保证产品的高性价比，使其具有更强的竞争力。
（4）代码可固化
为了提高执行速度和系统可靠性，嵌入式系统中的软件一般都固化在存储器芯片或单片机本身中，而不是存储于磁盘中。

（5）实时性
很多采用嵌入式系统的应用具有实时性要求，所以大多嵌入式系统采用实时性系统。但需要注意的是嵌入式系统不等于实时系统。
（6）弱交互性
嵌入式系统不仅功能强大，而且要求使用灵活方便，一般不需要类似键盘、鼠标等。人机交互以简单方便为主。
（7）嵌入式系统软件开发通常需要专门的开发工具和开发环境
（8）要求开发、设计人员有较高的技能
嵌入式系统是将先进的计算机技术、半导体技术和电子技术与各个行业的具体应用相结合后的产物。这一点就决定了它必然是一个技术密集、资金密集、高度分散、不断创新的知识集成系统，从事嵌入式系统开发的人才也必须是复合型人才。

1.3 嵌入式系统发展趋势

未来嵌入式系统的发展趋势主要有：
（1）小型化、智能化、网络化、可视化
随着技术水平的提高和人们生活的需要，嵌入式设备（尤其是消费类产品）正朝着小型化便携式和智能化的方向发展。如果你携带笔记本电脑外出办事，你肯定希望它轻薄小巧，甚至你可能希望有一种更便携的设备来替代它，目前的上网本、MID（移动互联网设备）、便携投影仪等都是因类似的需求而出现的。对嵌入式而言，可以说是已经进入了嵌入式互联网时代（有线网、无线网、广域网、局域网的组合），嵌入式设备和互联网的紧密结合，更为我们的日常生活带来了极大的方便和无限的想像空间。嵌入式设备功能越来越强大，未来我们的冰箱、洗衣机等家用电器都将实现网上控制；异地通讯、协同工作、无人操控场所、安全监控场所等的可视化也已经成为了现实，随着网络运载能力的提升，可视化将得到进一步完善。人工智能、模式识别技术也将在嵌入式系统中得到应用，使得嵌入式系统更具人性化、智能化。
（2）多核技术的应用
人们需要处理的信息越来越多，这就要求嵌入式设备运算能力更强，因此需要设计出更强大的嵌入式处理器，多核技术处理器在嵌入式中的应用将更为普遍。
（3）低功耗（节能）、绿色环保
在嵌入式系统的硬件和软件设计中都在追求更低的功耗，以求嵌入式系统能获得更长的可靠工作时间。如：手机的通话和待机时间，mp3 听音乐的时间等等。同时，绿色环保型嵌入式产品将更受人们青睐，在嵌入式系统设计中也会更多的考虑如：辐射和静电等问题。
（4）云计算、可重构、虚拟化等技术被进一步应用到嵌入式系统中
简单讲，云计算是将计算分布在大量的分布式计算机上，这样我们只需要一个终端，就可以通过网络服务来实现我们需要的计算任务，甚至是超级计算任务。云计算（Cloud Computing）是分布式处理（Distributed Computing）、并行处理（Parallel Computing）和网格计算（Grid Computing）的发展，或者说是这些计算机科学概念的商业实现。在未来几年里，云计算将得到进一步发展与应用。
可重构性是指在一个系统中，其硬件模块或（和）软件模块均能根据变化的数据流或控制流对系统结构和算法进行重新配置（或重新设置）。可重构系统最突出的优点就是能够根据不同的应用需求，改变自身的体系结构，以便与具体的应用需求相匹配。
虚拟化是指计算机软件在一个虚拟的平台上而不是真实的硬件上运行。虚拟化技术可以简化

软件的重新配置过程，易于实现软件的标准化。其中 CPU 的虚拟化可以单 CPU 模拟多 CPU 并行运行，允许一个平台同时运行多个操作系统，并且都可以在相互独立的空间内运行而互不影响，从而提高工作效率和安全性，虚拟化技术是降低多内核处理器系统开发成本的关键。虚拟化技术是未来几年最值得期待和关注的关键技术之一。

随着各种技术的成熟与在嵌入式系统中的应用，将不断为嵌入式系统增添新的魅力和发展空间。

（5）嵌入式软件开发平台化、标准化、系统可升级，代码可复用将更受重视

嵌入式操作系统将进一步走向开放、开源、标准化，组件化。嵌入式软件开发平台化也将是今后的一个趋势，越来越多的嵌入式软硬件行业标准将出现，最终的目标是使嵌入式软件开发简单化，这也是一个必然规律。同时随着系统复杂的的提高，系统可升级和代码复用技术在嵌入式系统中得到更多的应用。另外，因为嵌入式系统采用的微处理器种类多，不够标准，所以在嵌入式软件开发中将更多的使用跨平台的软件开发语言与工具，目前，Java 语言正在被越来越多的使用到嵌入式软件开发中。

（6）嵌入式系统软件将逐渐 PC 化

需求和网络技术的发展是嵌入式系统发展的一个源动力，随着移动互联网的发展，将进一步促进嵌入式系统软件 PC 化。如前所述，结合跨平台开发语言的广泛应用，那么未来嵌入式软件开发的概念将被逐渐淡化，也就是嵌入式软件开发和非嵌入式软件开发的区别将逐渐减小。

（7）融合趋势

嵌入式系统软硬件融合、产品功能融合、嵌入式设备和互联网的融合趋势加剧。嵌入式系统设计中软硬件结合将更加紧密，软件将是其核心。消费类产品将在运算能力和便携方面进一步融合。传感器网络将迅速发展，其将极大的促进嵌入式技术和互联网技术的融合。

（8）安全性

随着嵌入式技术和互联网技术的结合发展，嵌入式系统的信息安全问题日益凸显，保证信息安全也成为了嵌入式系统开发的重点和难点。

1.4 实时操作系统

1.4.1 实时操作系统定义

实时操作系统（RTOS）是指当外界事件或数据产生时，能够接受并以足够快的速度予以处理，其处理的结果又能在规定的时间之内来控制生产过程或对处理系统作出快速响应，并控制所有实时任务协调一致运行的操作系统。因而，提供及时响应和高可靠性是其主要特点。实时操作系统有硬实时和软实时之分，硬实时要求在规定的时间内必须完成操作，这是在操作系统设计时保证的；软实时则只要按照任务的优先级，尽可能快地完成操作即可。我们通常使用的操作系统在经过一定改变之后就可以变成实时操作系统。

实时操作系统是保证在一定时间限制内完成特定功能的操作系统。例如，可以为确保生产线上的机器人能获取某个物体而设计一个操作系统。在“硬”实时操作系统中，如果不能在允许时间内完成使物体可达的计算，操作系统将因错误结束。在“软”实时操作系统中，生产线仍然能继续工作，但产品的输出会因产品不能在允许时间内到达而减慢，这使机器人有短暂的不生产现象。一些实时操作系统是为特定的应用设计的，另一些是通用的。一些通用目

的的操作系统称自己为实时操作系统。但某种程度上，大部分通用目的的操作系统，有实时系统的特征。这就是说，即使一个操作系统不是严格的实时系统，它们也能解决一部分实时应用问题。

1.4.2 实时操作系统的特征

1）多任务；
2）有线程优先级
3）多种中断级别
小的嵌入式操作系统经常需要实时操作系统，内核要满足实时操作系统的要求。

1.4.3 实时操作系统的相关概念

（1）基本概念

代码临界段：指处理时不可分割的代码。一旦这部分代码开始执行则不允许中断打扰；

资源：任何为任务所占用的实体；

共享资源：可以被一个以上任务使用的资源；

任务：也称作一个线程，是一个简单的程序。每个任务被赋予一定的优先级，有它自己的一套 CPU 寄存器和自己的栈空间。典型地，每个任务都是一个无限的循环，每个任务都处在以下五个状态下：休眠态，就绪态，运行态，挂起态，被中断态；

任务切换：将正在运行任务的当前状态（CPU 寄存器中的全部内容）保存在任务自己的栈区，然后把下一个将要运行的任务的当前状态从该任务的栈中重新装入 CPU 的寄存器，并开始下一个任务的运行；

内核：负责管理各个任务，为每个任务分配 CPU 时间，并负责任务之间通讯。分为不可剥夺型内核于可剥夺型内核；

调度：内核的主要职责之一，决定轮到哪个任务运行。一般基于优先级调度法；

（2）关于优先级的问题

任务优先级：分为优先级不可改变的静态优先级和优先级可改变的动态优先级；

优先级反转：优先级反转问题是实时系统中出现最多的问题。共享资源的分配可导致优先级低的任务先运行，优先级高的任务后运行。解决的办法是使用“优先级继承”算法来临时改变任务优先级，以遏制优先级反转。

（3）互斥

虽然共享数据区简化了任务之间的信息交换，但是必须保证每个任务在处理共享共享数据时的排他性。使之满足互斥条件的一般方法有：关中断，使用测试并置位指令（TAS），禁止做任务切换，利用信号量。

因为采用实时操作系统的意义就在于能够及时处理各种突发的事件，即处理各种中断，因而衡量嵌入式实时操作系统的最主要、最具有代表性的性能指标参数无疑应该是中断响应时间了。中断响应时间通常被定义为：

中断响应时间=中断延迟时间+保存 CPU 状态的时间+该内核的 ISR 进入函数的执行时间。

中断延迟时间=MAX(关中断的最长时间，最长指令时间) + 开始执行 ISR 的第一条指令

的时间。

1.5 微内核和宏内核

宏内核：也称为单内核（Monolithic kernel），将内核从整体上作为一个大过程实现，并同时运行在一个单独的地址空间。所有的内核服务都在一个地址空间运行，相互之间直接调用函数，简单高效。单内核是个很大的进程。他的内部又能够被分为若干模块（或是层次或其他）。但是在运行的时候，他是个单独的二进制大映象。其模块间的通讯是通过直接调用其他模块中的函数实现的，而不是消息传递。

微内核（Micro kernel）：在微内核中，大部分内核都作为单独的进程在特权状态下运行，他们通过消息传递进行通讯。在典型情况下，每个概念模块都有一个进程。因此，假如在设计中有一个系统调用模块，那么就必然有一个相应的进程来接收系统调用，并和能够执行系统调用的其他进程（或模块）通讯以完成所需任务。在这些设计中，微内核部分经常只是个消息转发站：当系统调用模块要给其他系统模块发送消息时，消息直接通过内核转发。这种方式有助于实现模块间的隔离。（某些时候，模块也能够直接给其他模块传递消息。）在一些微内核的设计中，更多的功能，如 I/O 等，也都被封装在内核中了。但是最根本的思想还是要保持微内核尽量小，这样只需要把微内核本身进行移植就能够完成将整个内核移植到新的平台上。其他模块都只依赖于微内核或其他模块，并不直接直接依赖硬件。
微内核设计的一个长处是在不影响系统其他部分的情况下，用更高效的实现代替现有系统模块的工作将会更加容易。我们甚至能够在系统运行时将研发出的新系统模块或需要替换现有模块的模块直接而且迅速的加入系统。另外一个长处是无需的模块将不会被加载到内存中，因此微内核就能够更有效的利用内存。
通常嵌入式操作系统（如 Vxworks）采用微内核设计结构，以节省内存空间，而通用操作系统（如 Linux）则采用宏内核设计结构。

1.6 Vxworks操作系统简介[1]

VxWorks 是专门为实时嵌入式系统设计开发的操作系统内核，为程序员提供了高效的实时多任务调度、中断管理，实时的系统资源以及实时的任务间通信。在各种 CPU 平台上提供了统一的编程接口和一致的运行特性，尽可能的屏蔽了不同 CPU 之间的底层差异。应用程序员可以将尽可能多的精力放在应用程序本身，而不必再去关心系统资源的管理。基于 VxWorks 操作系统的应用程序可以在不同 CPU 平台上轻松移植。
VxWorks 是美国 Wind River System 公司（ 以下简称风河公司 ，即 WRS 公司）推出的一个实时操作系统。WRS 公司组建于 1981 年，是一个专门从事实时操作系统开发与生产的软件公司，该公司在实时操作系统领域被世界公认为是最具有领导作用的公司。
VxWorks 是一个运行在目标机上的高性能、可裁减的嵌入式实时操作系统。它以其良好的可靠性和卓越的实时性被广泛地应用在通信、军事、航空、航天等高精尖技术及实时性要求

[1] 本节内容主要摘自http://www.hwacreate.com.cn/chanpin/dlcp/windriver/WorkBench.html，2010.06。

极高的领域中，如卫星通讯、军事演习、弹道制导、飞机导航等。在美国的 F-16、FA-18 战斗机、B-2 隐形轰炸机和爱国者导弹上，1997 年 4 月在火星表面登陆的火星探路者上也使用了 VxWorks。

VxWorks 是一种功能强大而且比较复杂的操作系统，包括了进程管理、存储管理、设备管理、文件系统管理、网络协议及系统应用等几个部分。VxWorks 只占用了很小的存储空间，并可高度裁减，保证了系统能以较高的效率运行。所以，仅仅依靠人工编程调试，很难发挥它的功能并设计出可靠、高效的嵌入式系统，必须要有与之相适应的开发工具。TornadoII 就是为开发 VxWorks 应用系统提供的集成开发环境，TornadoII 中包含的工程管理软件，可以将用户自己的代码与 VxWorks 的核心有效的组合起来，可以按用户的需要裁剪配置 VxWorks 内核；vxSim 原型仿真器可以让程序员不用目标机的情况下，直接开发系统原型，作出系统评估；功能强大的 CrossWind 调试器可以提供任务级和系统级的调试模式，可以进行多目标机的联调；优化分析工具可以帮助程序员从多种方式真正地观察、跟踪系统的运行，排除错误，优化性能。

1.6.1 高性能的微内核设计

处于 VxWorks 嵌入式实时操作系统核心的是高性能的微内核 wind。这个微内核支持所有的实时特征：快速任务切换、中断支持、抢占式和时间片轮转调度等。微内核设计减少了系统开销，从而保证了对外部事件的快速、确定的反应。

运行环境也提供了有效的任务间通信机制，允许独立的任务在实时系统中与其行动相协调。开发者在开发应用程序时可以使用多种方法：用于简单数据共享的共享内存、用于单 CPU 的多任务间信息交换的消息队列和管道、套接口、用于网络通信的远程过程调用、用于处理异常事件的信号等。为了控制关键的系统资源，提供了三种信号灯：二进制、计数、有优先级继承特性的互斥信号灯。

1.6.2 可裁剪的运行软件

VxWorks 之所以设计为具有可裁剪性，是为了使开发者能够根据自己的应用程序需要，而不是根据操作系统的需要，来分配稀少的内存资源。从需要几个 KB 字节内存的深层嵌入式设计到需要更多的操作系统的功能的复杂的高端的实时系统，开发者也许需要从 100 多个不同的选项中进行选择以产生上百种的配置方式。许多独立的模块都是在开发时要使用而在产品中却不再使用。

而且，这些子系统本身也是可裁剪的，这样就允许开发者为最广泛的应用程序进行更为优化的 VxWorks 运行环境配置。例如，如果应用程序不需要某些功能模块，就可以将它移出 ANSI C 运行库；如果应用程序不需要某些特定的内核同步对象，这些对象也可以忽略。还有，TCP、UDP、套接口和标准 Berkeley 服务也可以根据需要将之移出或移入网络协议栈。

这些配置选项可以通过 TornadoII 的项目工具图形接口轻易地选择。开发者也可以使用 TornadoII 的自动裁剪特性，自动地分析应用程序代码并合并合适的选项。

1.6.3 综合的网络工具

VxWorks 是第一个支持工业标准 TCP/IP 的实时操作系统。创新的传统伴随着 VxWorks TCP/IP 协议栈，它支持最新的 Berkeley 网络特性，包括：
IP, IGMP, CIDR, TCP, UDP, ARP
RIP v.1/v.2
Standard Berkeley sockets and zbufs
NFS client and server, ONC, RPC
Point-to-Point Protocol
BOOTP, DNS, DHCP, TFTP
FTP, rlogin, telnet, rsh
WindRiver 也支持可选的 WindNet 产品：SNMP v.1/v.2c, OSPF v.2, STREAMS.
WindRiver 还通过提供工业级最广泛的网络开发环境来加强这些核心技术，这主要是通过 WindLink for TornadoII 伙伴计划来实现的。高级的网络解决方案还包括：
ATM, SMDS, frame relay, ISDN, SS7, X.25, V5 广域网网络协议
IPX/SPX，AppleTalk，SNA 局域网网络协议
分布式网络管理的 RMON，CMIP/GDMO，基于 Web 网的解决方案
CORBA 分布式计算机环境

1.6.4 兼容 POSIX 1003.1b 标准

VxWorks 支持 POSIX 1003.1b 的规定和 1003.1 中有关基本系统调用的规定，包括：过程初始化、文件与目录、I/O 初始化、语言服务、目录处理；而且 VxWorks 还支持 POSIX 1003.1b 的实时扩展，主要包括：异步 I/O、记数信号量、消息队列、信号、内存管理和调度控制。

1.6.5 平台的选择

WindRiver 还提供现成的一整套的商业和评估板。VxWorks 开放的设计具有高度的可移植性并且支持几乎所有的处理器，这样，应用程序就可以在不同的体系结构之间毫不费力的移植。

1.6.6 方便地移植到用户硬件上

能否将操作系统和应用程序以一种合适的方式进行移植是嵌入式软件开发方面的关键。如果事先就考虑了操作系统和应用程序代码的可移植性，那么这个过程就会变得非常容易。这需要明确划分低级的依赖于硬件的代码和高级的应用程序和操作系统代码，这样，移植时只需要改变整个依赖于硬件的低级代码，而不需要改变操作系统和应用程序。
依赖于硬件的这一层称为板极支持包(BSP, Board Support Package)。板极支持包是运行 VxWorks 的任何目标板都需要的。BSP Developer’s Kit 使开发者很容易地在用户硬件上使用

VxWorks；如果使用商业硬件，WindRiver 提供了 2000 个板极支持包。当为用户板开发板极支持包时，开发者可以获得大量的标准设备驱动程序，这些程序对应于所有的目标体系。

1.6.7 操作系统选件

操作系统选件产品为开发者提供了意想不到的特性和操作系统扩展。这些选件主要包括：
板极支持包开发工具(BSP Developer’s Kit)
支持闪存文件系统的 TrueFFS for TornadoII
支持图形应用程序
支持虚拟内存管理 VxVMI
支持多处理的 VxMP、VxDCOM 和 VxFusion

1.7 Vxworks 应用范围

Vxworks 应用范围非常广，以应用领域分，可以分成如下类应用：
（1）消费电子
掌上电脑
机顶盒
可视电话
汽车导航系统

（2）航空航天
飞行模拟器
航班管理系统
卫星跟踪系统
航空电子设备

（3）数字图象设备
打印机、传真机
数字复印机
数字相机

（4）数据通讯网络
交换机
路由器
远程访问服务器
ATM 及帧中继交换机

（5）电信设备
PBXs，ACDs

CD 交换系统
移动通讯基站
蜂窝式电话
Cable Modem

（6）交通运输
汽车发动机控制
交通信号控制
高速列车控制

（7）工业控制
机器人
测试与测量设备
过程控制系统
计算机外设
网络计算机
X 终端
RAID 存储系统
I/O 控制设备

第二章 Vxworks 操作系统

本章我们将对 Vxworks 操作系统如下方面进行介绍：任务，任务调度，任务间通信，内存管理，中断处理。

2.1 Vxworks 任务

Vxworks 是实时操作系统，这决定了他的任务调度必须是基于优先级的，而且必须是可抢占式调度方式，这样才能够区分实际情况下不同状态的处理级别，对高优先级的情况进行优先响应。

Vxworks 内核维护三个队列：tick 队列，ready 队列，active 队列。另外还有一个队列涉及到任务，即任务等待资源时所处的队列，这个队列可以是 Vxworks 内核提供的，也可以是用户提供的，此处令其为 pend 队列。

所谓 tick 队列，即当调用 taskDelay 函数让任务延迟一段固定的时间时，任务所处的队列，此时任务为设置为 Delay 状态，无资格竞争使用 CPU；ready 队列即有资格竞争使用 CPU 的所有任务，该队列以优先级为序排列任务，队列头部即除了当前运行任务外，系统中最高优先级的任务；active 队列有些误导，实际上称之为 task 队列更合适，因为系统中所有任务，无论任务当前状态为何，都将在这个队列中，这个队列维护者系统中当前所有的任务，即通过该队列可以查找到当前系统中的所有任务，在 shell 下运行‘i’命令，显示系统中所有的任务，就是通过遍历 active 队列完成的；pend 队列即当任务竞争使用某资源，而资源当前不可得时，任务就被设置为 pend 状态，进入 pend 队列。

函数 taskSpawn 创建一个新的任务，首先其创建一个任务控制结构，对该结构进行初始化后，将结构加入 active 队列以作系统任务管理之用，此时任务仍无资格竞争使用 CPU，taskSpawn 函数的最后一步就是将这个任务结构再加入到 ready 队列，此时这个任务才真正可以称为已经在竞争时用 CPU 了，当系统中所有优先级高于这个任务的其他任务运行完毕或者由于等待资源而处于阻塞时，这个新创建的任务就将被调度运行。所以在 Vxworks 下，如果一个新创建的任务优先级不高，创建后将等待一段时间才能被真正执行。在实际项目中，有时需要一个新的任务被创建后立刻得到执行，那么就需要在创建任务时，指定一个较高的任务优先级。Vxworks 内核将任务分为 256 个优先级，标号从 0 到 255。其中 0 表示最高优先级，255 表示最低优先级。任务在调用 taskSpawn 函数进行创建时就指定了优先级，当然任务的优先级并非在创建后就无法改变，用户可以通过调用 taskPrioritySet 函数在任务创建后重新设定优先级，taskPrioritySet 专门针对嵌入式平台下不同情况对同一任务不同运行级别的需求而设置的。值得注意的是，taskPrioritySet 函数不同于通用操作系统提供的类似函数，taskPrioritySet 函数可以提高或者降低任务的运行级别，而不是通用操作系统下在任务创建后就只能动态的降低任务优先级。

Vxworks 下对于应用层任务，推荐使用 100-250 之间的优先级，驱动层任务可以使用 51-99 之间的优先级，特别要注意的是内核网络数据包收发任务 tNetTask 的优先级为 50，如果使用网口进行调试，则一定要注意不要创建任务优先级高于 50 的任务，否则 tNetTask 任务将无法得到运行，表现形式是死机，因为系统将无法再从网口接收调试命令，无法响应 Tornado

Shell 或者 Telnet 下输入的任何命令。以上只是推荐值，事实上，由于嵌入式系统下的特殊应用，对于某个任务优先级的设置需要根据该任务完成的具体工作而定，而不可一味遵循推荐值。如笔者曾在项目中，为了与 Shell“争读”从串口发送的命令，就将一个应用层任务优先级设置为 0，只是要谨记在任务达到某一特殊目的后，必须将任务优先级设置回正常（推荐）值。

无论何种操作系统，任务在设计中都有一个数据结构表示。这个数据结构包含一个任务运行时需要的所有信息，我们一般将这些信息称为任务上下文。具体的，任务上下文（广义上）包括：

1. 所在平台 CPU 内部所有寄存器值，特别是指令寄存器，这代表了任务当前的执行点。这是一般狭义上的任务上下文。除了寄存器值，每个任务有自己的内存映射空间，任务名称，任务优先级值，任务入口函数地址，打开文件句柄数组，信号量，用于各种目的的队列等等。
2. 任务运行时暂时存放函数变量以及函数调用时被传递参数的栈。从操作系统底层实现来看，很多操作系统将表示任务的数据结构和任务栈统一管理，如 Linux 下在分配任务结构的同时分配任务的内核栈，这二者作为一个整体进行内存分配，通常将一页（如 4KB）的开始部分作为任务结构，用以存储任务关键信息，而将页的末尾作为任务内核栈的顶部，如此，实际上用一页页面的大小（通常为 4KB）减去任务信息占据的空间，剩下的空间都作为任务的内核栈在使用。Vxworks 与 Linux 在栈的分配和管理上基本类似，不过 Vxworks 区分与 Linux 的一个最大不同是 Vxworks 下所有的代码都运行在一个状态下，不区分内核态和用户态（当然 Vxworks 下也有内核态的概念，但含义完全不同，见下文分析），所以不存在 Linux 下的内核态栈和用户态栈，Vxworks 下任务自始至终都在使用同一个栈，不论这个任务在运行过程中调用了任何 Vxworks 内核函数，都不存在栈的切换。正因如此，Vxworks 对栈的大小无法预先进行把握，栈的大小将由被创建的任务决定，而且不同于通用操作系统，Vxworks 下任务栈在任务创建时就被确定，而且此后不可以改变栈的大小。所以对于一个存在很多递归调用的任务，必须在任务创建时指定一个较大的任务栈，防止在后续的运行中造成栈的溢出，导致任务异常退出。
3. 各种定时信息。实际上这些信息都作为任务结构中的一个字段而存在。任何操作系统都必须有一个系统时钟进行驱动，该系统时钟通常称为系统的脉搏。系统时钟一定与一个高优先级的中断联系，如此每当时钟前进一个滴答（tick），操作系统就会响应一次中断，该中断通常就被作为操作系统进程调度的触发点。每次滴答，操作系统都会增加内核维护的一个全局变量（如 Vxworks 操作系统维护的 vxTick 变量），通过该变量为系统各种定时器提供定时依据。每个任务固定的都有一个内部定时器，用于任务内部特定的需求，每次系统时钟产生一个中断，操作系统都会对当前系统内所有需要关注的任务定时器进行处理，从而完成任务定时器特殊的用途。定时器的一个特殊变相应用即 Round-Robin 任务调度（简称 RR 调度）。RR 调度实际上对每个支持 RR 调度的任务内部都维护有一个定时器。当一个支持 RR 调度的任务被调度进入运行状态时，在任务运行期间，每次系统时钟前进一个滴答时，该定时器指针都会前进一个单位，当到达预定的值时，该任务就要主动让出 CPU，以便让相同优先级的其他任务运行。定时器可以以加法或者减法运算运行。对于减法运算，一般根据定时时间计算出一个 Tick 数，每次系统前进一个滴答，Tick 数减一，当到达 0 时，表示定时器到期。而对于加法运算，则一般需要使用操作系统维护的全局 Tick 变量。Vxworks 下，如设置定时时间间隔为 N，则定时器到期时间为 vxTick+N=T0，每次系统前进一个滴答，操作系统会对当前系统内维护的所有定时器进行检查，判断 T0 是否大于 vxTick，一旦 T0≤vxTick，则表示该定时器到期，此时将根据定时器的目的作对应的响应。

4. 信号处理函数。事实上，操作系统都每个信号都有一个默认的响应方式，如用户在命令行敲入“Ctrl+C”时，则系统默认响应方式是中止当前前台任务。每个任务可以根据自身定制对某个信号的响应方式。如一个任务可以将用户的“Ctrl+C”操作响应为打印输出当前任务中某个变量的值。每个任务内部对每个信号都维护一个响应函数句柄，操作系统在创建任务时已经将所有的句柄设置为系统默认方式，用户在创建任务后，可以针对某个信号安装自己的信号响应句柄。
5. 其他辅助信息。这些信息包括一些统计上的数据，如任务运行总时间，任务最终返回值等等。

前文中说到Vxworks是基于优先级方式的抢占式任务调度操作系统，同时对于相同优先级的任务，支持Round-Robin循环调度方式（以下简称RR调度）。RR调度方式通过kernelTimeSlice函数使能。该函数调用原型如下：

```
STATUS kernelTimeSlice
(
int ticks /* time-slice in ticks or 0 to disable round-robin */
)
```

RR 调度默认情况下是禁用的。当用户以非零参数明确调用 kernelTimeSlice 函数后，RR 调度方才使能，此时相同优先级任务可以轮流使用 CPU，但是整个系统仍然是按优先级方式进行调度的。换句话说，只能当前系统中最高优先级下存在多个任务运行时，RR 调度才有效，如果存在多个低优先级的任务，那么系统仍然在运行着高优先级的任务，这些低优先级任务根本无法使用到 CPU。注意 kernelTimeSlice 函数参数为 tick 系统时钟嘀答数，一般 Vxworks 下一个滴答为 1ms，故大多直接将参数作为以 ms 为单位的时间值。如果在调用 kernelTimeSlice 函数使能 RR 调度方式后，又想禁用 RR 调度方式，则只需要以参数 0 再次调用 kernelTimeSlice 函数即可。

控制任务调度的另一个函数是**taskPrioritySet**，该函数通过改变任务优先级来控制调度。注意**taskPrioritySet**函数可以任意改变任务的优先级，该函数的调用原型如下：

```
STATUS taskPrioritySet
(
int tid, /* task ID */
int newPriority /* new priority */
)
```

第一个参数为需要改变优先级的任务的 ID 号，这是一个整型值，是调用 taskSpawn 创建任务时的返回值。参数二即对应任务需要设置的优先级，该参数范围为 0 到 255，即可以改变某个任务到任意优先级，而不是只能降低任务的优先级。

注意：一个任务可以在运行过程中通过**taskPrioritySet**函数按照需要自己改变自己的优先级，此时只需要将第一个参数设置为0即可。这种可以动态改变任务自身优先级的方式给任务的运行提供了极大的灵活性，特别当任务需要在某个特定时刻从特殊渠道获取数据时，必要提高自身的优先级才能读取到数据，而在读取数据后，又必须退回到正常优先级进行数据的处理，**taskPrioritySet**函数就是针对这种情况专门设计的。通常任务不需要使用taskPrioritySet函数。

另外一个控制任务调度的关键函数对是taskLock，taskUnlock。函数调用原型分别如下。

```
STATUS taskLock (void)
STATUS taskUnlock (void)
```

taskLock函数即关闭任务调度，taskUnlock函数即重新开启任务调度，这两个函数必须成对使用。这两个函数通常用于系统级的资源共享上，即是一种变相的互斥机制。不过需要注意的是这两个函数并不禁止中断，所以并不能与在中断中运行的函数形成互斥。另外注意的是taskLock并不是信号量，它仅仅是暂时禁止了任务调度机制，保证了接下来的一段时间内（直到调用taskUnlock），当前任务一直主动占用CPU，taskLock主要使用在需要以原子方式执行一段代码的情况下。如对某个变量进行某种操作（如加1或减1）。刚才说到任务主动占用CPU，即没有其他外界手段可以将任务调度出去，但是任务自身可以进入阻塞状态（如阻塞于某个资源，虽然不应在此种方式下调用taskLock），此时taskLock应有的作用将被取消，任务调度机制重新工作，直到该任务又重新被调度运行。换句话说，只要调用taskLock的任务处于运行状态，则任务调度机制一直被禁止，而一旦该任务主动由于某种原因主动让出CPU，则任务调度机制重新启动。taskLock的作用对于某个任务而言，只有taskUnlock可以取消。

Vxworks 提供任务创建函数 taskSpawn，该函数调用原型如下。

```
int taskSpawn
(
char * name, /* name of new task (stored at pStackBase) */
int priority, /* priority of new task */
int options, /* task option word */
int stackSize, /* size (bytes) of stack needed plus name */
FUNCPTR entryPt, /* entry point of new task */
int arg1, /* 1st of 10 req'd task args to pass to func */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8,
int arg9,
int arg10
)
```

参数 1：**char * name**

任务名。可以为 NULL，此时系统将使用默认任务名。默认任务名形式为“tN”，其中‘t’为前缀，‘N’是一个数字，系统将对所有任务创建时未提供任务名的任务依次进行编号，从 1 开始。

参数 2：**int priority**

任务优先级。该优先级决定了任务的调度级别。在任务创建完毕后，仍然可以使用taskPrioritySet 函数对任务优先级进行动态改变。

参数 3：**int options**
任务选项。控制任务的某些行为。任务可用选项如下图 2-1 所示。一般情况下，将该参数设置为 0。

任务选项		
选项	值	说明
VX_FP_TASK	0x0008	运行时使用浮点协处理器
VX_NO_STACK_FILL	0x0100	不要对栈进行0xee填充
VX_PRIVATE_ENV	0x0080	使用私有环境运行任务
VX_UNBREAKABLE	0x0002	禁止设置断点
VX_DSP_TASK	0x0200	DSP处理器支持
VX_ALTIVEC_TASK	0x0400	ALTIVEC处理器支持

图 2-1 任务可用选项

参数 4：**int stackSize**
任务栈大小。注意任务栈在创建任务时指定，此后不可更改。任务从创建到结束自始至终都在使用这个栈。Vxworks 下任务栈和表示任务的结构连续存放，在内存分配时是作为一个整体进行分配的。

参数 5：**FUNCPTR entryPt**
任务开始执行时入口函数地址。当任务被调度运行时，从该参数指定的地址开始执行。

参数 6-15：**int arg1 ~ int arg10**
入口函数参数，最多可同时传递 10 个参数。多于 10 个时，可以通过指针的方式传递。

返回值：
taskSpawn 函数返回一个整型数据，表示刚创建任务的 ID。实际上这个 ID 是一个内存地址，指向这个被创建任务的 TCB（Task Control Block）。故原则上，可以通过以下方式通过任务 ID 得到任务的 TCB：

```
WIND_TCB   *tPcb;
tPcb = ((WIND_TCB *) tid);
…
```

但是并不建议这样做，因为对于任务 ID 的解释可以随着 Vxworks 内核版本的不同该改变，故要从任务 ID 得到任务对应的 TCB，建议使用内核提供的函数 taskTcb，taskTcb 函数调用原型如下。

WIND_TCB *taskTcb

```
(
int tid /* task ID */
)
```

使用方式如下代码所示。

```
WIND_TCB   *tPcb;
tPcb = taskTcb(0);
```

注意：taskTcb 要求传入任务 ID，当以 0 作为参数传递给该函数时，taskTcb 将返回当前任务的 TCB 结构地址。以参数 0 表示得到当前任务的某种信息，这种处理方式在 Vxworks 下很常见。

taskSpawn 调用举例：

```
taskSpawn ("demo", 20, 0, 2000, (FUNCPTR)usrDemo, 0,0,0,0,0,0,0,0,0,0);

void usrDemo (void)
{
    ……
}
```

taskSpawn 函数调用完毕后，任务就进入运行状态，即与其它任务竞争使用 CPU。有时创建一个任务后，需要暂时使其处于挂起状态，不具有调度运行的资格，这可以通过调用 taskCreate 函数完成。taskCreate 调用原型如下。

```
int taskCreate
(
char * name, /* name of new task */
int priority, /* priority of new task */
int options, /* task option word */
int stackSize, /* size (bytes) of stack needed */
FUNCPTR entryPt, /* entry point of new task */
int arg1, /* 1st of 10 req'd args to pass to entryPt */
int arg2,
int arg3,
int arg4,
int arg5,
int arg6,
int arg7,
int arg8,
int arg9,
int arg10
)
```

taskCreate 函数参数与 taskSpawn 完全一致。其区别于 taskSpawn 函数之处就在于 taskCreate 创建后的任务暂不具有运行的资格，需要另一个函数 taskActivate“激活”。taskActivate 函数

将任务结构加入到 ready 队列，从而使得任务有资格竞争使用 CPU。至于任务确切地运行时间根据系统当前任务情况决定。一般而言，某个任务结构被加入到系统 ready 队列，我们就认为其已经运行状态，因为余下的情况的已经无法由系统和用户掌控。如果用户需要在任务被激活后立刻得到运行，可以设置任务为较高优先级。任务激活函数 taskActivate 调用原型如下。

```
STATUS taskActivate
(
int tid /* task ID of task to activate */
)
```

该函数接收 taskCreate 函数返回的任务 ID，对指定 ID 的任务进行激活，即将任务结构添加到系统任务 ready 队列中，使得任务成为竞争使用 CPU 的一员。

任务栈再议

任务在 Vxworks 内核中作为调度的基本单元，每个任务是程序的一个实例，程序可以由多个或单个函数构成。任务在运行这些函数的过程中，不可避免的使用一些函数变量（定义在函数内部的变量），这些函数变量的存放地就在任务栈中；当存在函数间调用时，需要某种机制传递参数，使用较多的就是栈机制。栈实际上就是一块地址连续的内存块，只不过由于其特殊的使用方式从而被封装成一个特殊的结构类型。一般，栈的使用是向下递减地址方式，即栈顶位于内存高地址处，栈底位于内存低地址处，使用时从栈顶开始使用，逐渐向栈底逼近。当然也有相反方向工作模式。无论方向如何，其目的都是一样的，只要在某个平台保持全局范围内的统一性，就可以保证系统工作的正常性。注意：栈的方向一般是由 CPU 硬件决定的，更进一步说，是由指令集决定的。如 Intel x86 系列提供的 push，pop 指令就是向下递减性栈，操作系统编写者对此必须了解。

通用操作系统进程栈（通用操作系统一般称每个调度单元为进程）分为两个层次，这是由于通用操作系统一般具有两个运行态-内核态和用户态-决定的。通用操作系统需要提供高安全性的运行环境，其必须明确的隔离两个不同层次上的操作，从而知道哪些操作是被允许的，哪些是被禁止的，当然安全措施不单是通过运行态来实现，运行态的区分只是其中重要措施之一。通用操作系统对于进程在两个不同运行态下具有不同的栈，当进程运行在用户态时，其使用用户态栈，当发生系统调用或由于中断进入到内核态时，其切换到进程的内核态栈。一般而言，用户态栈是使用不尽的，由平台内存空间决定，地址空间一般不造成限制（如 Linux 下用户态栈原则下有接近 3G 的空间可用），而内核态栈大小是由很大限制的，如 Linux 下内核态栈一般只有大约 4KB 的大小。所以进程运行在内核态时，要求大空间的结构必须使用堆的方式分配，不要使用栈进行分配。事实上，由于运行在内核态的代码都是严格限制的，所以一般都不会发生内核态栈溢出的问题。而由于用户态栈可使用空间极大，虽然其先分配给用户态栈的实际内存较小，但是可以根据需要动态的增加用户态栈内存，所以很难使得用户态栈溢出（你能想象一个应用程序可以使用完 3GB 的栈吗？）。

Vxworks 内核不使用通用操作系统下的运行态，虽然 Vxworks 下也有内核态的概念，不过去本质上就是一个内核数据结构的简单保护机制，仅仅由一个全局变量 kernelState 表示是否在内核态，当用户进行内核函数调用或者中断发生进入到 Vxworks 内核运行时，其并不发生运行态的本质切换，而任务自始至终都在使用同一个栈。换句话说，Vxworks 下任务栈即被应用函数（用户编写的函数）使用，也被内核函数（Vxworks 内核提供）使用，其 Vxworks 下任务栈在初始创建后，不可以根据需要在后期动态的增加内存容量。这就对 Vxworks 任务栈的初始创建大小提出了一个要求，即其在创建栈（即创建任务）时指定的栈大小必须足

够大，以满足任务后续运行期间对栈容量的需求，而这一点是很难由用户在创建时进行判断的。故实际上，用户在创建任务时会指定一个比实际需要大得多的栈容量，这对于一个嵌入式系统而言，对已然宝贵的内存资源造成了极大的浪费。Vxworks 官方文档的建议是在通过试验法，在程序开发期间，在任务创建时，指定一个比较大的栈，在任务运行期间，不断地使用 checkStack 函数查看任务的栈使用情况，从而得到一个任务栈的大致使用数据统计。其后，用这个得到的统计值作为任务栈的实际容量。checkStack 函数的调用原型如下。

```
void checkStack
(
int taskNameOrId /* task name or task ID; 0 = summarize all */
)
```

参数指定要检查的栈对应任务的 ID。以参数 0 调用该函数时，会打印出系统内所有任务的栈的使用情况。checkStack 函数的一个使用实例如下。

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

任务名长度

当在 shell 下使用‘I’命令查看当前系统中的任务时，对于任务名的显示有些令人误解：对于任务名长度较长的字符串（如大于 11 个字节）都发生了截断。这让很多人误解为任务名最大只能有 10 个字节长度。故在创建任务时刻意减少任务名的长度。其实这只是显示上的一些问题，taskShow 函数（’I’命令的底层调用函数）在大印任务名时进行了截断操作，而非在 taskSpawn 函数中对传入的任务名参数进行截断。事实上，Vxworks 支持任意长度的任务名。换句话说，在调用 taskSpawn 或者 taskCreate 创建任务时，用户可以指定一个其需要的任何名称。Vxworks 内核在内部以无损方式保存了这个用户传入的任意名称。只是在调用 taskShow 对任务信息进行显示时，在打印格式上的一些操作只打印出了任务名前 11 个字节。用户可以根据需要自编写一个任务信息显示函数，将任务名称全部打印出来。

Vxworks 下并不要求任务名的全局唯一性，两个完全不同的任务可以使用相同的名称而不会造成底层（内核）使用上的任何问题，但是这回让查看任务信息的用户产生疑问。同样上文中讨论到的 taskShow 对于任务名的截断显示也会造成同样的问题。故如果仍然使用内核提供的 taskShow 显示任务信息，则建议用户在任务创建时尽量将任务限制在 11 个字节之内，或者两个不同任务的名称最好在前 11 个字节内有所区别。任务信息显示函数 taskShow 调用原型如下。

```
STATUS taskShow
(
int tid, /* task ID */
int level /* 0 = summary, 1 = details, 2 = all tasks */
)
```

参数1：需要显示信息的任务ID。

参数2：任务信息开放程度。

注意 taskShow 根据第二个参数值对任务信息和任务范围作不同程度的信息显示。当参数 2 传入的值为 2 时-即对系统内所有任务进行信息显示，此时将忽略第一个参数值。否则将只

显示参数 1 指定的任务信息。而参数 2 的值表示显示任务信息的详细程度。0 表示只显示大概信息，这些信息包括函数名称，入口函数，任务栈使用情况总结；1 表示显示任务的详细信息，这些信息除了以上所说的大概信息外，还包括任务栈的详细信息（栈基址，栈容量），任务所包含的事件，任务当前寄存器值。

任务结束

除了极少数几个任务自始至终保持存活外，大多数任务都只是在一段时间内保持活动，最后都不可避免的要消亡。消亡由两个方式：一是任务正常运行到结束，通过 exit 函数结束；二是被直接删除，即非正常方式结束。一般情况下，用户创建一个任务，完成某个特定的功能，最后“功成身退”，从系统中消失，这是一般任务的工作方式。例如任务执行一个 main 函数，最后通过 reture 返回，代码在编译时编译环境实际上在函数最后加上了一个 exit 函数，这是对用户透明的，用户只需要调用 reture 语句或者直接退出都可以，但是从底层代码运行来看，在运行完所有用户代码后，还会执行一个 exit 函数，这个函数是由编译环境在编译用户程序时自动加入的，与在函数结束时加上一个 exit 函数同样的道理，实际上编译环境在调用用户提供的入口函数之前，也加上了一个类似于 init 的函数，创建程序的运行环境，进而调用用户提供的入口函数，真正执行用户的代码。这些机制有些类似于 C++中自动加入的初始化和消亡函数。Vxworks 下 exit 函数结束一个任务的寿命，其底层调用 windDelete 函数，完成表示任务的数据结构的释放和任务栈的释放。注意：windDelete 只释放任务栈和表示任务本身的结构，并不负责释放用户在任务运行过程中分配的任何内存空间。exit 函数用于一个任务正常的自动消亡。内核同样提供了一个非正常结束任务的方式：taskDelete 函数。该函数的调用原型如下。

```
STATUS taskDelete
(
int tid /* task ID of task to delete */
)
```

注意：当以参数 0 调用 taskDelete 时，表示删除当前运行的任务自身。此时的功能有些类似于被动调用 exit 函数退出。一般而言，如果一个任务需要删除自身，那么直接调用明确调用 exit 皆可，当然也可以以参数 0 调用 taskDelete。但是 taskDelete 更多的被用于删除一个其他正常运行的任务，而非自身。该函数提供的功能必须慎用。通常对于普通的用户不需要调用这么一个任务删除函数，这个函数极容易造成内核的不一致性，从而最后导致内核崩溃。设想一个任务刚刚获取一个资源的使用权，正在所谓的“关键区域”执行代码。而另外一个任务却“霸道”的将其直接删除。此时被删除任务将停止运行余下的代码，表示任务的结构和任务栈被释放，进而到内核资源状态的不一致性，因为任务退出之前没有相应的执行释放其已获取资源的使用权，导致其他任何竞争使用该资源的任务处于无限等待状态。以一个简单的例子说明，如多个函数需要竞争使用同一项资源，该资源有一个变量进行保护，任何使用该资源的任务在使用之前必须检查该变量的值，如果该变量为 1，则表示资源可用，为 0，则表示资源已被其他任务使用，所有其他任务必须等待，知道该变量重新变为 1.为了保证对变量本身操作的原子性，一般会使用某种特定编码方式，如在 Intel x86 结构下，会使用 lock 指令修改该变量的值。某个竞争使用资源的任务准备要使用资源时，其检查该变量的值，如果为 0，则等待，等待一段时间后，其重新检查该变量的值，此时该变量值为 1，表示资源可以访问，那么任务将会将该变量的值设置为 0，表示“我”这个任务现在正在使用资源，其他任务必须进行等待。而在该任务使用资源期间，一个“外来不明是非者”将这个任务进行了删除，即中断了正在使用资源的任务的执行，致使其非正常消亡，即系统中已经不存在这个任务了，那么这个任务原先在使用完资源后将该变量重新设置为 1 表示资源可用的代码

永远也无法得到运行，换句话说，该资源被一个“死亡”的任务无限期的“使用”，所有其他竞争使用该资源的任务都无限期的处于等待状态。
所以不正当的删除一个正在运行的其他任务很非常危险的一个操作，除非明确知道被删除任务当时所处的状态，如被删除任务已经设置了一个标志位表示其可被删除，因为其已经释放了其所使用的一切资源，正在处于“无为”状态，但是在外界其他某个条件满足之前，其又不能提前结束，故等待其他任务对其进行删除。但是这种情况除了某个特殊情况，一般都不会发生。所以 taskDelete 函数的使用场合较为有限，用户只需对其进行了解，建议不要使用。
与 taskDelete“分庭抗礼”的一对函数就是 taskSafe，taskUnsafe。一个任务为了防止在预先无任何提示的情况下被其他任务删除，其可以调用它 taskSafe 函数进行自身的保护，一个任务在调用 taskSafe 函数后，则其他任何任务都不可对其进行删除操作，这个函数就是为了应对 taskDelete 而设计的，以防止上文所述的一个处于“关键区域”的任务被野蛮删除的行为发生。taskSafe 和 taskUnsafe 函数调用原型如下。

```
STATUS taskSafe (void)
STATUS taskUnsafe (void)
```

如下代码例举了这两个函数的典型应用环境。

```
taskSafe ();
semTake (semId, WAIT_FOREVER); /* Block until semaphore available */
.
. /* critical region code */
.
semGive (semId); /* Release semaphore */
taskUnsafe ();
```

Vxworks 内核还提供如下三个函数对运行中的任务施加影响。taskSuspend 挂起一个正在执行的函数，该函数调用原型如下。

```
STATUS taskSuspend
(
int tid /* task ID of task to suspend */
)
```

当以参数 0 调用 taskSuspend 时，表示挂起当前正常执行的任务。而以非 0 任务 ID 方式指定一个其他被挂起的任务。这个其他被挂起的任务原先可以是由于优先级较低而等待运行，也可以是一个处于延迟（delay）睡眠的任务或者是一个已然被挂起的任务。

taskResume 函数的功能取消 taskSuspend 的作用，其将一个被挂起的任务重新设置为可运行状态。当然此时被恢复的任务一定是一个其他任务，因为一个任务是不可能 Resume 其自身的。taskResume 函数的调用原型如下。

```
STATUS taskResume
(
int tid /* task ID of task to resume */
)
```

第三个比较重要的控制任务运行的函数时 taskDelay，即置当前正在运行的任务为睡眠状态，睡眠时间长度由调用 taskDelay 时输入的参数为据。taskDelay 函数的调用原型如下。

```
STATUS taskDelay
```

```
(
int ticks /* number of ticks to delay task */
)
```

注意：参数以系统滴答数为单位，即系统时钟的时间间隔为单位，一般而言，这个间隔为1ms，所以可以简单的认为在毫秒级的任务延迟。任务调用 taskDelay 进行延迟在代码中使用较多，特别是以轮询方式对某个设备进行操作的任务一般都是以 while 循环加 taskDelay 的方式工作。

另外注意 taskDelay 的一种特殊工作方式，就是以 NO_WAIT 参数调用 taskDelay，此种方式主要是用以在 RR 调度禁止使用的前提下，给相同优先级的其他任务一个使用 CPU 的机会。当以 NO_WAIT 调用 taskDelay，Vxworks 内核将当前任务置于 ready 队列中所有相同优先级的任务之后，从而给相同优先级的其他任务提供一个运行的机会。注意这种方式一般使用在 RR 调度被禁用的情况下。如果使能了 RR 调度，则一般无需使用此种方式，因为 RR 调度会循环让相同优先级的任务使用 CPU 资源的。

涉及 Vxworks 任务一个重要的使用是可以在任务创建，消亡，调度之时调用用户注册的钩子函数，这在某些情况下会特别有意义。Vxworks 提供一种机制可以让用户注册一种钩子函数，当系统中有新的任务被创建，一个任务消亡以及任务调度时，可以执行用户注册的这些函数，从而达到用户的特殊目的。如对于低功耗的实现，可以利用 Vxworks 内核提供的这种机制，虽然这不是很优美的一种实现，但可以用于说明钩子函数的作用。低功耗要求当 CPU 中没有用户任务运行时，将整个平台尽量置于低功耗状态。当 Vxworks 下没有用户任务时，Vxworks 将执行其自身提供的一个 Idle 任务，Idle 任务具有最低优先级，换句话说，只有存在一个可运行的用户任务，无论这个任务优先级是什么，其都将被调度运行，如此我们可以使用一个最低优先级的后台任务加上一个任务调度钩子函数实现平台低功耗目的。在系统初始化阶段，创建一个最低优先级的后台任务，这个任务的优先级要低于所有其他有“责任”在身的其他任务，换句话说，当这个后台任务被调度运行时，就表示当前系统可以处于低功耗模式。我们同时注册一个任务调度时被调用的钩子函数。后台任务实现代码就是一个 FOREVER 语句，其伪代码如下所示。

```
FOREVER
{
    if(powerdown==FALSE){
        put board to power down state;
        powerdown=TRUE;
    }

}
```

钩子函数实现如下。

```
void taskSwitchHook(WIND_TCB *pOldTcb, WIND_TCB *pNewTcb){
    if(old task is our daemon task){ /*即后台任务被其他任务替代了*/
        put board back to normal state;
        powerdown=FALSE;
    }
}
```

一般而言，将平台从低功耗模式唤醒的方式有很多种，如可以使用一个外部中断专门用于唤醒平台，使脱离低功耗模式，上述代码只是一个简单示例，要使用到实际中要需要很多考虑，但是这给我们提供了钩子函数的一种使用方式。

Vxworks 提供的钩子函数注册函数如下。

1> 任务创建钩子函数注册和注销

```
STATUS taskCreateHookAdd
(
FUNCPTR createHook /* routine to be called when a task is created */
)
```

参数为注册的钩子函数，其将在任何一个新任务创建时被调用，其必须具有的定义形式如下：即以新创建任务的结构为参数。钩子函数可以这个代表新创建任务的结构做一些个性化的处理，如检查优先级使限定在特定的范围等。

```
void createHook
(
WIND_TCB *pNewTcb /* pointer to new task's TCB */
)
```

taskCreateHookDelete用以注销之前注册的钩子函数。其调用原型如下。

```
STATUS taskCreateHookDelete
(
FUNCPTR createHook /* routine to be deleted from list */
)
```

2> 任务调度钩子函数注册和注销

```
STATUS taskSwitchHookAdd
(
FUNCPTR switchHook /* routine to be called at every task switch */
)
```

taskSwitchHookAdd 用于注册一个发生任务调度时调用的一个钩子函数，该钩子函数必须具有如下的定义形式。其中 pOldTcb 表示被调度出去的任务结构，而 pNewTcb 则表示被调度进来成为 CPU 的新的使用者的任务结构。

```
void switchHook
(
WIND_TCB *pOldTcb, /* pointer to old task's WIND_TCB */
WIND_TCB *pNewTcb /* pointer to new task's WIND_TCB */
)
```

taskSwitchHookDelete 用以注销通过 taskSwitchHookAdd 添加的钩子函数。其调用原型如下。

```
STATUS taskSwitchHookDelete
(
FUNCPTR switchHook /* routine to be deleted from list */
```

```
)
```

3> 任务消亡（删除）钩子函数注册和注销

```
STATUS taskDeleteHookAdd
(
FUNCPTR deleteHook /* routine to be called when a task is deleted */
)
```

用以注册在任何一个任务消亡时被调用的钩子函数。钩子函数 deleteHook 必须具有如下的定义形式，参数是消亡的任务的结构。

```
void deleteHook
(
WIND_TCB *pTcb /* pointer to deleted task's WIND_TCB */
)
```

taskDeleteHookDelete 用以注销之前通过 **taskDeleteHookAdd** 注册的钩子函数。其调用原型如下。

```
STATUS taskDeleteHookDelete
(
FUNCPTR deleteHook /* routine to be deleted from list */
)
```

除了这些注册和注销函数外，Vxworks 同时还提供了三个信息显示函数用于显示当前注册到系统中的所有钩子函数。这三个函数的调用原型如下。

```
void taskCreateHookShow (void)
```

显示当前注册到系统的所有在任务创建时被调用的钩子函数列表。

```
void taskDeleteHookShow (void)
```

显示当前注册到系统中所有在发生任务切换时被调用的钩子函数列表。

```
void taskSwitchHookShow (void)
```

显示当前注册到系统中的在任务消亡时被调用的钩子函数列表。

至此，我们完成对 Vxworks 任务的说明，Vxworks 下任务即通用操作系统下所述的进程，是内核基本运行单元，也是内核调度算法的处理单元。纵观所有的操作系统，进程或者任务在底层上都有一个数据结构表示，这个数据结构一般称为任务（或者进程）控制块 TCB（或者 PCB），用以保存一个任务（或者 CPU 执行单元）的所有信息，这些信息中有些是保证一个任务可以被 CPU 运行的基本关键信息（如所有寄存器值，内存映射），但很多是为管理需要存在的“辅助”信息，这些“辅助”信息构成了某个操作系统的实现方式。
Vxworks 任务给用户提供了极大的灵活性，如果深入到任务控制结构这一层面（即内核编程和驱动编程），那么可以对任务执行其任何需要的操作，即便让系统立刻崩溃也在所不惜。这也是底层编程提供的极大“娱乐性”。

Vxworks 任务的几个重要方面总结如下：

1. 任务具有优先级，是任务调度运行的依据和获取 CPU 资源的“竞争卡”，优先级越大，

其获得 CPU 资源的可能性就越大，Vxworks 以 0~255 的数字表示优先级，数字越大，优先级越小，0 表示最高优先级。任务优先级可以在任务运行过程中动态改变，可以改变一个任务到任意优先级。

2. 任务具有任务栈。任务栈的容量在任务创建时就被确定而且其后不可进行更改。任务栈与表示任务的内核结构作为一段连续的区域进行分配。任务栈是任务运行过程中各种局部函数变量存放地和函数调用参数传递的渠道。由于 Vxworks 下任务栈容量不可动态改变，故在创建任务时必须指定一个足够大的容量，通常可以在函数调试阶段通过 checkStack 统计出一个任务的栈使用情况，进而指定一个合理的任务栈容量，避免浪费较多的内存空间。
3. Vxworks 提供了一种机制可以在涉及任务的三个关键状态变化出调用用户注册的钩子函数，这三个状态变化为任务初始创建时，任务被调度使用或放弃 CPU 时，任务消亡时。通过利用 Vxworks 提供的这种灵活性可以实现一些嵌入式平台下有意义的策略。
4. 每个任务在操作系统底层实现上都是有一个数据结构表示，通过直接更改这个数据结构中某些字段值，可以控制任务某些运行行为，这在内核层编程时十分有效。Vxworks 任务结构定义在 taskLib.h 头文件中，感兴趣用户可以查看 Vxworks 任务结构 WIND_TCB 的具体定义。

2.2 Vxworks 进程（任务）调度

Vxworks 是实时操作系统，进程调度时机对于其而言至关重要。基本上所有的操作系统的进程调度时机都选择在同一个地方：从内核状态退出之时。从内核状态退出时机主要有两个渠道：系统调用和中断。外部中断可有多个源，一般操作系统只是使用系统时钟中断作为固定的进程调度时刻点，而其他中断源以及系统调用由于其执行时机的不确定性，只是作为进程调度触发的辅助手段。实时操作系统区别于通用操作系统的关键点在于其必须能够迅速的响应外部中断，所以实时操作系统只是在响应速度上更快，而在响应时机上与通用操作系统并无区别。中断响应速度即中断产生到对应中断处理程序的调用之间的延迟时间。实时操作系统将这个延迟尽量减少到最小。一个理解的误区是系统时钟频率对实时性起决定性影响，事实并非如此。例如 Vxworks 系统时钟间隔 1ms 量级，而现在绝大多数通用操作系统的系统时钟间隔也是在 1ms 量级。并非时钟间隔越小越好，当时钟间隔过小时（如间隔为 0.5ms 时），操作系统将花费大量的时间用于处理进程调度上，会严重影响整个系统的工作性能。注意：系统时钟是进程调度的脉搏。另一个误区是将系统时钟与 CPU 主频混为一谈。CPU 主频是指 CPU 内硬件单元处理速度，一般以每秒内执行的指令数为指标参数。CPU 主频越高，对系统的实时性越好，因为可以在更快的时间内执行完过渡指令，从而使得中断响应的时间更少。所以系统实时性一方面由操作系统决定，另一方面也是由操作系统运行的平台决定，或者更进一步，由平台上 CPU 主频决定。

Vxworks 操作系统区别于通用操作系统的一个显著的特点是其划分运行级别。应用层程序可以直接调用内核函数，而不需要通过软中断的方式从应用层进入到内核层。故 Vxworks 系统调用的概念与通用操作系统（如 Linux）有本质上的差别。下面代码分别演示了在 Vxworks 和 Linux 下一个应用层函数（userRtn）是如何调用一个内核层函数（kernelRtn）的。

Vxworks 下系统调用：

```
void userRtn（）{
    kernelRtn（）;
```

```
}
```
Linux 下系统调用：
```
void userRtn（）{
    _asm_();
}
```

Vxworks 下所谓内核态仅仅有一个内核布尔变量 kernelState 表示。当 kernelState 设置为 TRUE 时，表示此时代码是允许在内核状态。Vxworks 内核态的本质即保护内核数据结构，防止多处代码对内核数据结构的同时访问，所以它不同于通用操作系统内核态的概念。通用操作系统内核态是为了保护内核数据结构不受应用层的影响。所以说，Vxworks 内核态更多的是一个信号量。如下代码显示了 kernelState 的作用，从而我们可以看出内核态的基本含义：防止多处代码对内核数据结构的同时操作。

```
if (kernelState)                    /@ defer work if in kernel @/
{
    workQAdd1 (windResume, tid);        /@ add work to kernel work q @/
    return (OK);
}
kernelState = TRUE;                 /@ KERNEL ENTER @/
…                                   /@ OPERATE ON KERNEL STRUCTRUES @/
windExit ();                    /@ KERNEL EXIT @/
```

当代码需要对内核数据结构进行操作时，其首先检查 kernelState 的状态，如果 kernelState 设置为 TRUE，则表示当前已经有代码正在操作内核数据结构，故将当前代码要做的工作加入到内核工作队列中，稍候再完成。可以说，内核工作队列以及任务队列构成了 Vxworks 内核的架构，基本上内核代码都是对各种队列的操作，所以上文中所说的 Vxworks 内核数据结构主要就是指这些内核队列。Vxworks 内核态的进入通过简单的设置 kernelState 变量值为 TRUE 完成，而退出内核态则是由 windExit 函数完成，该函数除了将 kernelState 设置为 FALSE 之外，完成的另外两个重要工作就是运行内核工作队列中延迟的工作以及进程调度。进程调度函数（reschedule）在 windExit 中被调用用以调度更高优先级的进程运行。而 windExit 函数被调用的时机则是从内核态退出，而中断和系统调用则是进入内核态的唯一手段，所以从这个意义上说 Vxworks 进程调度的时机即在系统调用和中断发生之时。更进一步，中断一定会引起进程调度，系统调用在绝大多数情况下也会引起进程调度。系统调用并不总是能够引起进程调度，因为可能调用的内核函数较为简单，不需要对内核共享数据结构进行操作，此时就不会执行 windExit 函数。

中断可以分为时钟中断和其他硬件中断。时钟中断即以固定时间间隔的产生中断，这个中断专门用于系统定时器和进程调度。任务状态的改变主要有时钟中断触发。其他硬件中断则是嵌入式系统的关键组成部分，用以控制特定设备。这些硬件中断的实时响应可以使某个嵌入式系统的主要功能。一个嵌入式系统可以没有任务（当然除了 Idle 任务外，这是 Vxworks 操作系统自带的后台进程），但是不可以没有硬件中断。

下面将以 ARM 处理器为例详细介绍时钟中断的注册和执行流程，这个中断是系统脉搏，了解它对于整体掌握 Vxworks 操作系统将有很大帮助，这对于在嵌入式平台下实现一些有意义的策略也很有启示。

时钟中断的注册在 Vxworks 操作系统初始化过程中完成，这主要是在 userRoot 中调用的如下三个函数完成。

```
sysClkConnect ((FUNCPTR) usrClock, 0);   /* connect clock ISR */
sysClkRateSet (SYS_CLK_RATE);      /* set system clock rate */
sysClkEnable ();
```

其中 sysClkConnect 完成时钟中断程序的注册。事实上传递给 sysClkConnect 函数的 userClock 函数并非是直接中断响应函数。ARM 处理器支持两类中断：普通中断 irq 和快速中断 fiq。Vxworks 操作系统只使用了 irq。对于一个嵌入式平台而言，单个中断源显然不够用，故实际上各种中断都是以 irq 的形式存在的。Vxworks 内核维护一个 irq 中断的总入口中断处理程序，再由这个总入口中断处理函数根据中断向量号进一步调用对应中断处理程序。时钟中断中断向量号一般设置为 0。时钟中断对应的中断响应函数为 sysClkInt，这是在 sysHwInit2 函数中完成注册的。事实上，sysHwInit2 是在 sysClkConnect 中被调用的。sysClkConnect 调用 sysHwInit2 注册 sysClkInt 作为时钟中断响应函数，基本代码如下：

```
void sysHwInit2(){
    …
    (void)intConnect ( INUM_TO_IVEC(INT_TINT0), sysClkInt, 0);
    intEnable ( INT_TINT0 );
    …
}
```

而后，sysClkConnect 将 userClock（sysClkConnect 的参数）作为二次函数调用注册到 Vxworks 内核中。所谓二次函数调用即当发生一个时钟中断时，sysClkInt 作为时钟中断响应函数将首先被调用，而 sysClkInt 又进一步调用 userClock 函数。userClock 函数定义在 userConfig.c 中，其代码如下：

```
void usrClock ()
{
    tickAnnounce ();    /* announce system tick to kernel */
}
```

从以上代码可以看出，userClock 直接调用 tickAnnounce 函数作为响应。

tickAnnounce 函数主要完成如下工作：

1. 对 vxTick 变量作加一运算。vxTick 表示系统自启动之时到现在的 tick 数，所以用 vxTick 乘以系统时钟间隔就是开机时间。
2. 对处于等待状态的任务进行检查，将超时任务（那些调用 taskDelay 延迟的任务）重新设置为 ready 状态，并转移到调度队列中。
3. 遍历内核工作队列，对延迟的内核工作进行执行。
4. 进程调度。选择最高优先级任务作为当前任务运行。

时钟中断注册过程总结如下：

userConfig.c：userRoot()→arm_timer.c：sysClkConnect()→sysLib.c：sysHwInit2

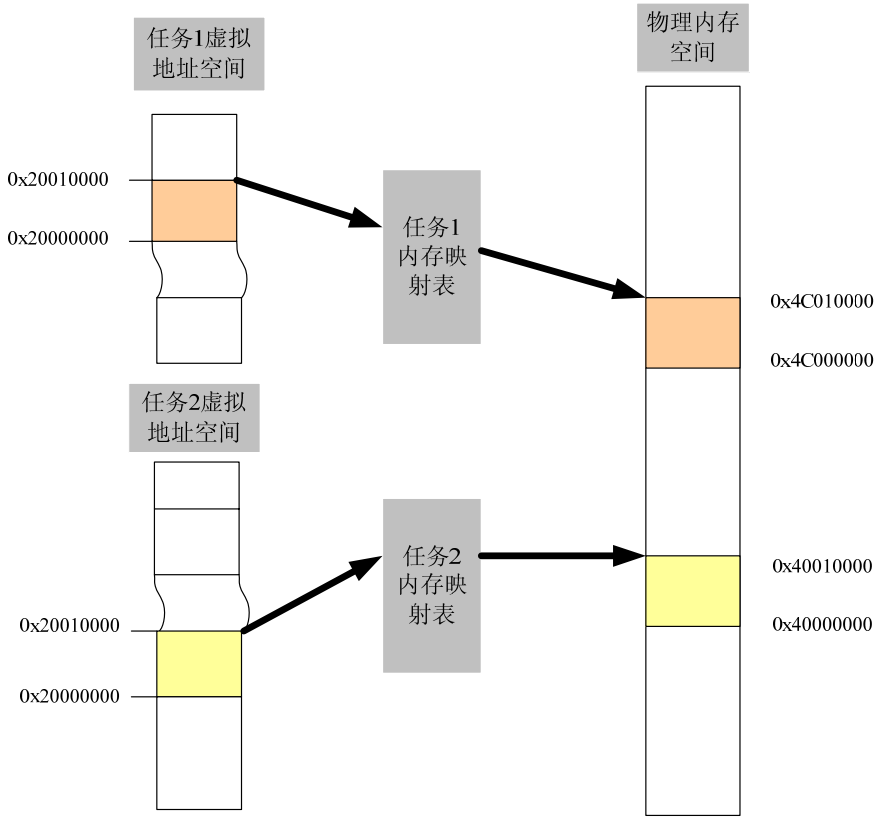
其中 arm_timer.c 为 ARM 平台 BSP 中定时器驱动文件。诚如前文所述，sysClkConnect 函数

调用 sysHwInit2 将 sysClkInt 函数注册为时钟中断处理函数，同时将作为参数传入的 userClock 函数作为二次函数调用注册到 Vxworks 内核中。（注：二次函数调用即被 sysClkInt 调用执行。）

时钟中断执行流程总结如下：
时钟中断产生→Vxworks 内核 IRQ 总入口中断处理函数→时钟中断处理函数（sysClkInt）→userClock 函数→tickAnnounce 函数。

2.3 任务间通信

每个任务都有自己的内存空间，无论内存是否通过 MMU 机制进行管理。通过 MMU 机制进行管理时，内存空间被分为两个区域：虚拟内存区域和物理内存区域。此时程序中使用的是虚拟内存地址，当一个程序需要访问外部内存时，一般 CPU 内存 MMU 硬件单元会自动完成虚拟地址到物理地址的转换。基于 MMU 机制的操作系统，任务所运行代码中使用虚拟地址，且由于需要通过虚拟地址到物理地址的变换，故不同任务之间虚拟地址的重叠不会造成任何问题，而实际上基本所有的操作系统在 MMU 机制下将系统内所有任务的虚拟地址空间都设置为完全重叠的区域，如 Linux 操作系统下一个任务（进程）的虚拟地址空间范围为 0~3G，最高 1G 被内核和所有任务共享（即内核代码使用的虚拟地址空间）。不同任务具有完全重叠的虚拟地址空间并不会对任务的实际运行造成任何影响，因为不同任务虽然具有相同的虚拟地址，但是通过 MMU 的映射都被映射到不同的物理地址，而物理地址才是访问外部实际 DDR RAM 时使用的地址。由于不同任务具有完全相同的虚拟地址，故为了保证任务间不相互影响，每个任务都具有自己的内存映射表，这个映射表作为任务自身组成的一个关键部分而存在。如下图 2-2 所示表示了两个不同任务地址映射关系。



不同任务相同虚拟地址空间映射到不同的物理地址空间

图 2- 2 虚拟地址空间到物理地址空间的映射

由于物理地址空间有限，故对于暂时不使用的物理页面可以先将其置换到硬件缓冲区中，即通常所述的交换区，从而让出相应的物理页面供其他任务使用。使用这种交换的方法，从表面上或者从任务使用页面的角度，表现为一个比实际内存设备容量大得多的物理存储。

对于无 MMU 机制下的任务而言，虚拟地址就是物理地址，此时不同任务的地址空间必须严格隔离开（当然除了刻意使用共享内存），保证不同任务间不会造成相互影响。

结合以上的说明，对于任务间通信，其本质就是在使用共享物理内存的机制。虽然手册上将内核提供的任务间通信机制与共享内存机制区分开来，只把用户层次的诸如全局变量之类的使用方式作为共享内存来看，但是从操作系统底层来看，所有的任务间通信实现方法的内在本质都是共享物理内存。

Vxworks 内核本身主要提供了三种任务间通信的机制（不包括信号机制）：信号量，消息队列，管道。这三种机制无一例外的都是在使用共享物理内存机制，只不过这块共享的内存由内核在进行管理，任务无法直接进行访问，而必须通过内核提供的接口函数进行访问，这就提供了一种保护和管理机制，使得任务间通信安全有序的进行。

1. 信号量

信号量的主要用途是互斥和同步。互斥主要保护资源，即某个时刻只允许只有一个任务在使用该资源。由于使用该资源的潜在用户可能很多，故信号量此时此地就用作一把“钥匙”，要使用该资源的任务，必须首先取得这把钥匙，方可进行资源的使用，否则就等待。同步则是任务间协同完成某一项共同工作的机制，典型的例子即一个任务产生资源，另一个任务使用资源，使用任务的资源平时处于等待状态，等待产生资源的任务完成其资源的生成，而一

旦资源产生完毕，此时产生资源的任务就会触发同步信号量，让等待使用资源的任务启动（即唤醒）其处理资源的工作。

信号量的底层实现可以简单的看做是一个内核维护的全局变量，对于用于互斥机制的信号量，这个内核全局变量初始化为 1，当一个任务需要访问该信号量保护的资源时，其首先检查这个内核全局变量的值是否为 1，如非 1，则表示已然存在其他任务已经在使用资源，则等待；如为 1，则表示资源当前可被访问，则这个任务首先将这个内核全局变量的值设置为 0，阻止其他任务的访问，而自身就可以安全的使用该资源。此处一个漏洞是，在当前任务修改内核全局变量的同时，另一个任务可能同时在检查这个全局变量的值，很可能造成另一个任务检查到全局变量值为 1 后，当前任务才完成全局变量 0 值的设置，此时就有两个任务在使用资源，造成内核状态的不一致，极端情况下，将造成整个系统的崩溃。内核对这种情况进行了特殊处理，一般是将变量的改变操作作为一个原子操作（如 x86 下提供的 Lock 指令）完成。这也是内核提供的任务间通信机制区别于用户层任务间通信机制的根本：内核提供的机制已经从根本上保证了足够的安全性。

基于各种资源不同的使用方式，Vxworks 信号量机制具体提供了三种信号量：通用信号量；互斥信号量；资源计数信号量。通用信号量既可用于同步也可用于资源计数，此时资源数通常为 1（当资源数为 1 时，也可以称之为互斥）。互斥信号量针对在使用过程中一些具体问题（如优先级反转）做了优化，更好的服务于任务间互斥需求；资源计数信号量用于资源数较多，同时可供多个任务使用的场合。

2. 消息队列

消息队列内核实现上实际是一个结构数组，数组大小和数组中元素的容量在创建消息队列时被确定。在创建消息队列时指定的另外一个参数是消息队列满时任务等待基于的策略：FIFO 或者优先级排序。消息队列是 Vxworks 内核提供的任务间传递较多信息的一种机制，不过这种机制存在的很大的局限性，即每个消息的最大长度是固定的。当然在这个最大长度范围内从用户层而言是可变的，但是对于内核维护而言，所有消息都具有相同的长度，因为无论实际消息的长度如何，内核都将按最大长度分配内存空间。当然如果对每个消息都采用动态内存分配方式，可以消除最大长度限制，但是这并不是 Vxworks 提供的消息机制。Vxworks 内核提供的消息机制在创建消息队列时就必须指定单个消息的最大长度以及消息的数量，在消息队列成功创建后，这些参数都是固定不变的。我们可以如此想象内核对于消息队列的实现，在消息队列创建之时，内核分配一个大小为单个消息最大长度与消息数量乘积的内存区域，可以将此看做是一个数组，数据元素个数为消息数量，每个元素的大小为单个消息最大长度。当用户发送一个消息时，内核将消息内容存入数组中下一个空闲元素中，用户读取消息时，将读取数组中下一个非空元素，底层基本实现为一个环形缓冲区。Vxworks 对消息最多只区分两个优先级的消息，对于高优先级的消息将从数组开始处存储，对于普通优先级的消息将从数组尾部开始存储，而读取时从数组头部开始读取，从而保证高优先级的消息优先被传递。当然以上只是一种简单的类比，帮助读者理解 Vxworks 内核对于消息队列的实现。由于消息队列在创建时指定了消息数量，一旦任务传递的消息总数大于这个数量，那么传递消息的任务则需要等待，即将任务暂时设置为挂起状态，并统一挂起到一个内核分配的专门附属于对应消息队列的任务队列中，在任务被放入这个任务队列时，需要依据一定的策略，这个策略也是在消息队列创建时指定，可以是 FIFO，也可以是基于优先级。FIFO 方式即任务将以先进先出的方式挂入到队列中，如果消息队列中空出一个元素可供传递消息，那么最早挂入队列的任务将得到消息传递权；而基于优先级的任务队列在任务挂起时，将根据优先级将任务插入到队列中，如此当消息队列可用时，最高优先级的任务将优先得到消息传递权。

3. 管道

管道相比消息队列提供了一种更为流畅的任务间信息传递机制。消息队列对于每个消息的大小存在限制，而且必须将信息分批打包，而管道可以像文件那样进行读写，是一种流式消息机制。其提供的基本操作方式类似于对一个文件的读写，支持 select 函数，所以可以对多个管道进行信息监测。管道在底层实现上是一种更为直接的共享物理内存机制。信号量和消息队列还需要对传递的数据进行某种方式的封装，则管道不会传递信息做任何包装，直接分配一块连续的内存空间作为任务间信息交互的中转站。传统意义上，管道分为两种：命名管道和非命名管道。通常任务间通信使用的一般都是命名管道。非命名管道使用在线程意义上，如父子进程，进程关系密切，且某些变量存在继承关系，如文件描述符，一般无法使用在两个执行路线完全不同的任务之间。

Vxworks 内核提供的管道机制在创建时如同消息队列也要指定一个消息数量以及单个消息最大长度，事实上，Vxworks 提供的管道机制在底层实现上完全基于消息队列，用户层使用上完全类似于对一个文件的操作，但是对于每次读写的字符数存在最大长度限制，这个限制就是管道创建时指定的最大消息长度。由于管道底层实现上是基于消息队列，故管道只是在应用层进行了文件系统层次的封装，可以支持 open，write，read，close 等普通文件操作行为，在文件系统层次下，为字符驱动层，提供 pipe_open，pipe_write，pipe_read，pipe_close 等响应函数，字符驱动层就是消息队列实现函数，诸如 msgQReceive, msgQSend 等函数。所以，本质上讲，Vxworks 下管道机制只是在消息队列之上提供了文件系统层次的支持，其本质上还是一个消息队列，每次 read，write 读写的最大数据长度受限，且 read，write 连续调用的次数受限，即如果连续调用 read，而无对应 write 操作，内核维护的消息队列为空，此时 read 任务将挂起等待。注意：基于管道的消息队列任务等待策略是基于 FIFO 方式的。

4. Socket

Socket 是一种特殊的任务间通信机制，其特殊之处在于通信的任务双方不需要限制在同一台 PC 上，可以是联网的任何两台计算机上的两个任务。由于传递的信息需要通过非稳定的网络介质，故用户需要指定信息传输策略：是否要求数据可靠性。通常而言，使用 Socket 机制进行信息传递的两个任务间大多使用在批量数据传输上，但也有进行控制信息传输的。Socket 通信虽然底层实现相比其他机制要复杂的多，但这些都是操作系统实现的一部分，在用户使用层次上，与以上讨论的其他三种机制没有区别。用户只需遵循一套规定的操作接口即可完成不同主机上两个任务间的信息传输。Socket 机制下的通信由于涉及不同主机，故传输方式从根本上区别于其他任务间通信机制，其底层实现上由网络栈加网卡驱动组成，网卡驱动完成传输介质上（如双绞线）的解析，通过网络栈的层层剖析，最后将传递纯数据暂存于内核空间供任务读取，发送方任务发送的信息经过网络栈的层层封装，暂存于内核空间，依次按序的通过网卡驱动最终将数据发送到网络传输介质上。由于内核提供的暂存空间有限，故无论发送方还是接收方都有数据率的限制。

任务间通信的特殊机制：信号

信号不是信号量，二者不是一个概念。信号量是一种任务间互斥和同步机制，而信号则用于通知一个任务某个事件的发生。一个信号产生后，对应任务暂停当前执行流程，转而去执行一个特定的函数进行处理。信号机制有些类似于中断，也可以将信号看作是一种用户层提供的软件中断机制（区别于 CPU 本身提供的软件中断指令）。这种用户层软件中断机制相比硬件中断和中断指令而言具有较长的延迟时间，即从信号产生到信号的处理之间大多将经历较

长的延迟。Vxworks 下信号处理有些特别，当一个任务接收到一个信号时，在这个任务下一次被调度运行之时进行信号的处理（即调用相关信号处理函数）。事实上，由于 Vxworks 在退出内核函数时都会进行任务调度，故一个任务，无论是否是当前执行任务，都将在被调度运行时执行信号的处理。

通用操作系统上对于某些信号将不允许用户修改其默认处理函数，如 SIGKILL，SIGSTOP，然而 Vxworks 操作系统中可以对任何信号的处理函数都可以进行更换。

2.4 内存管理

内存管理并非简单的使得物理内存可被使用，通常说到内存管理，就认为是否使能 MMU，更进一步说，即是否使用虚拟地址。一般 CPU 都包含 MMU 硬件单元，但是 MMU 的使用与否由操作系统决定，当然使用 MMU 硬件单元是需要做一系列准备工作的，如系统页表的建立，配置 CPU 硬件控制寄存器等。虽然存在这些准备工作，这与使用 MMU 机制之后的优势相比也是值得的。MMU 可以管理物理内存非连续的应用环境，可以对单个页面控制是否进行 cache 以及应用权限限制，可以让一个很大的程序与其他程序一同运行，而非独占物理内存或者无法运行。
是否使能 MMU 只是内存管理的一个重要方面，内存管理还包括操作系统如何安排各组件（内核层次的和应用层次的）在物理内存中的布局；如何提供接口可让 BSP 开发人员提供内存映射的关系，因为每个平台的物理内存地址空间不同，需要根据特定平台的特殊情况定制映射关系；是否提供某种策略可让用户预留一段物理内存以供私用，如专供某个驱动存储信息等等。这些都可以看作是内存管理的一部分。本节将介绍涉及内存使用以及内存管理的各个方面。

由于各处理器下内存管理的方式有些差别（如有些处理器包含 MMU 硬件支持，有些则不然），故下文讨论中我们以 ARM 处理器为例进行介绍。ARM 处理器内部包含 MMU 硬件单元，MMU 的使能通过 CP15 控制寄存器的相关位来进行控制。系统上电执行的初始阶段，MMU 是禁止使用的，如果使用 bootrom，那么在 bootrom 运行的整个过程中，MMU 都是禁止的，一般只在 Vxworks 操作系统中才使能 MMU。使用 MMU 之前，必须在内存中创建一张页表，对于 ARM 处理器而言，这个页表分为两个层次（实际上，为了节省内存，基本上所有操作系统都是通过分层次的方式建立页表），第一层次称为页目录表，第二层次才称为页表。

在 MMU 使能的情况下，系统使用两套地址：虚拟地址和物理地址。虚拟地址是指令中使用的地址，这个地址必须经过 MMU 单元的转换方能输出到 CPU 外部总线上访问实际 RAM；物理地址即实际出现在地址总线上的地址，这个地址直接输入到 RAM 的地址管脚，用以寻址 RAM 中的某个存储单元。MMU 单元的转换必须要有页表的配合方能完成，这个页表一般由操作系统进行创建。具体到 Vxworks 下，为了适应各种平台的地址映射关系，内核提供一个内核变量供 BSP 开发人员使用，用以表示该平台上的地址映射关系，这个内核变量在 sysLib.c 文件中被初始化，变量名为 sysPhysMemDesc，这个变量的初始化示例如下。

```
PHYS_MEM_DESC sysPhysMemDesc [] =
```

```
{
    /* adrs and length parameters must be page-aligned (multiples of 0x1000) */

    /* iram 16K */
    {
        (void *)ARM_I_RAM_BASE,      /* virtual address */
        (void *)ARM_I_RAM_BASE,      /* physical address */
        ROUND_UP(ARM_I_RAM_SIZE, PAGE_SIZE), /* length, then initial state: */
        VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
                VM_STATE_MASK_CACHEABLE,
        VM_STATE_VALID| VM_STATE_WRITABLE | VM_STATE_CACHEABLE
    },
    …
    /* DDRAM */
    {
            (void *) ARM_DDR2_BASE,      /* virtual address */
            (void *) ARM_DDR2_BASE,      /* physical address */
            ROUND_UP (ARM_DDR2_SIZE, PAGE_SIZE), /* length, then initial state: */
            VM_STATE_MASK_VALID | VM_STATE_MASK_WRITABLE |
                VM_STATE_MASK_CACHEABLE,
            VM_STATE_VALID| VM_STATE_WRITABLE | VM_STATE_CACHEABLE
    }
};
```

sysPhysMemDesc 更确切的说是一个结构数组，数组中每个元素都是一个 PHYS_MEM_DESC 结构，这个结构定义在 h/vmLib.h 文件中，定义如下：

```
typedef struct phys_mem_desc
{
    void *virtualAddr;
    void *physicalAddr;
    UINT len;
    UINT initialStateMask;          /* mask parameter to vmStateSet */
    UINT initialState;              /* state parameter to vmStateSet */
} PHYS_MEM_DESC;
```

Vxworks 内核初始化过程将依据 sysPhysMemDesc 变量构建系统页表。

对于一般通用操作系统（如 Linux），其页表的构建是动态的，即在操作系统初始化过程中，只对操作系统本身代码已使用的内存空间进行页表项的构建，则其他内存空间用作动态内存在以后使用的过程中不断对系统页表项进行添加。更进一步说，每个用户进程都有自己的一份页表，这份页表中对于内核代码部分的映射都是相同的，只在自己地址空间所对应的页表项上映射到的物理页表才有差别。在进程进行调度时，系统页表的更换是作为调度的一部分完成。所以在 Linux 下所有的用户进程可以使用完全相同的虚拟地址 0x80000000，程序运行前都被链接到这个相同的虚拟地址处。只是在映射到物理内存时，才分配到不同的物理地址处，由于内核代码在操作系统初始化过程中已被复制到固定的物理内存中，故所有的用户

进程对内核代码的映射都是一致的，且提供给用户的内核代码一般都是可重入的（reentrable），所以各进程对于内核代码的调用都不存在问题。注意：通用操作系统中一个可运行的用户程序一般都是通过链接的，如 Linux 下的.out 文件，Windows 下的.exe 文件，这些文件中的代码都具有实际运行时使用的绝对虚拟地址。

Vxworks 下使用 MMU 时虽然也存在页表表示的映射关系，但是其工作机制却从根本上不同于通用操作系统，首先 Vxworks 运行的所有任务（进程）都使用相同的页表，换句话说，Vxworks 操作系统中只有一张页表，所有的任务都使用这张页表；其次非常关键的一点，Vxworks 运行的文件都是未链接的，在程序运行之前，都需通过 loadModule 函数的即时动态链接。Tornado 环境下编译的.out 文件实际上也是一个未链接文件，或者更准确的说，是一个可链接的文件，如此不同的程序经过 loadModule 函数被动态链接到不同的物理地址处，才将各个任务的地址空间区分开来而不造成冲突。如此就引起另一个问题，即 Vxworks 的内存分配都是针对物理内存直接进行的，而在使能 MMU 的情况下，访问内存使用的是虚拟地址，这就表示物理地址被直接用作了虚拟地址！这个关系决定了系统页表中虚拟地址到物理地址的映射必须是相同的！即虚拟地址必须等于物理地址，否则整个系统将无法正常工作。所以在 Vxworks 下使用 MMU 和使用系统页表并非实现通常操作系统下灵活的映射机制，而是更多的是提供面向页的 cache 机制和权限控制机制。实际上 Vxworks 下的地址映射关系是非常固定的，在系统初始化过程中，这个系统页表就被完全建立，在建立完成后，基本不存在改变系统页表项的操作（虽然 Vxworks 支持动态添加页表项），在这一点上完全不同于通用操作系统。

我们可以对 Vxworks 下内存映射机制作一总结。

1. 首先 Vxworks 下整个系统只有一张页表，所有的任务都在使用这张页表。
2. 其次 Vxworks 系统页表在初始化过程中就被完全创建，其后基本不会改变该页表，虽然 Vxworks 支持动态改变页表项。这同时也表示 BSP 开发任务必须仔细编写 sysPhysMemDesc 变量定义，对其后操作系统程序以及所有要运行的用户程序需要访问的地址都必须进行映射，包括主内存，所有外设的寄存器地址区域。
3. 在页表中，所有的物理内存都被进行了映射，但并不表示对应的物理内存已被使用，内存的使用与否由 Vxworks 内核另外的机制表示。
4. Vxworks 下虽然使用 MMU 和页表，但是使用的虚拟地址实际上就是物理地址，这是由 Vxworks 的内存分配方式决定的。Vxworks 下分配内存函数返回的地址直接就是物理内存地址，这个返回的物理内存地址被直接使用访问物理内存，即物理地址被直接用作虚拟地址，这就从根本上决定了页表项中的内容必须是虚拟地址等于物理地址。
5. Vxworks 所有程序的运行都要经过 Vxworks 内核的重新动态链接操作，这个链接由 loadModule 及其同族函数完成，包括 Tornado 环境下编译生成的.out 文件也是要经过 Vxworks 内核的重新链接，才被执行的，且生成的.out 文件本身就是被设计在可被重新链接的。
6. Vxworks 下使用 MMU 更多的是提供面向页（一页一般为 4KB）的 cache 机制和权限控制机制，而非提供虚拟地址和物理地址之间灵活的映射关系。

内存管理的另一个重要方面是如何安排 Vxworks 内核以及各种信息在内存中的布局。如图 1 所示是 ARM 处理器平台下物理内存的布局示意图。注意不同版本的 Vxworks 内核内存布局存在差别，图 2-3 中所示为 Vxworks 5.5 内核对应的布局。

图 2-3 中几点注意：

1. 某些组件的包含与否也会影响内存布局，例如如果不包含 WDB 组件，那么图 2-3 中 WDB 内存池将被并入系统内存池。
2. ‘Persistent Mem Reserved’为错误探测和报告组件预留的内存空间，这个空间的大小由 PM_RESERVED_MEM 表示。
3. 异常向量表（Vectors）和异常跳转指针（Exception Pointers）一般都是以绝对地址 0 为基址。ARM 处理器要求系统异常向量表在绝对地址 0 或者高端地址处创建，ARM 处理器内部绝对地址 0 处集成有 IRAM，故一般将异常向量表创建在这个 IRAM 首端（地址为 0）。图 1 中没有表示出这一点，而是以 LOCAL_MEM_LOCAL_ADRS 为基址进行了表示，这一点是不确切的。此时就必须要求 LOCAL_MEM_LOCAL_ADRS 为 0.除此之外从 BootLine 开始都是以 LOCAL_MEM_LOCAL_ADRS 进行偏移是正确的。但诚如前文所述，这个偏移量的确切值根据 Vxworks 内核版本的不同而有所差别，但是从总体顺序上是没有变化的。
4. “initial stack”使用在 usrRoot 函数执行之前，usrRoot 函数执行在任务上下文中，其前所有的代码都没有任务上文下，我们将其类比于中断上下文，“initial stack”就被这些代码使用。每个任务都具有自己独立的栈，故从 usrRoot 函数开始就在使用任务自身的栈，“initial stack”自此以后将不再被使用。“initial stack”大小在 ARM 平台下设置为 512 个字节，其栈顶就是 RAM_LOW_ADRS，栈底则为（RAM_LOW_ADRS-512），当然这是对于一般“向下增长型（grows down）”栈而言。

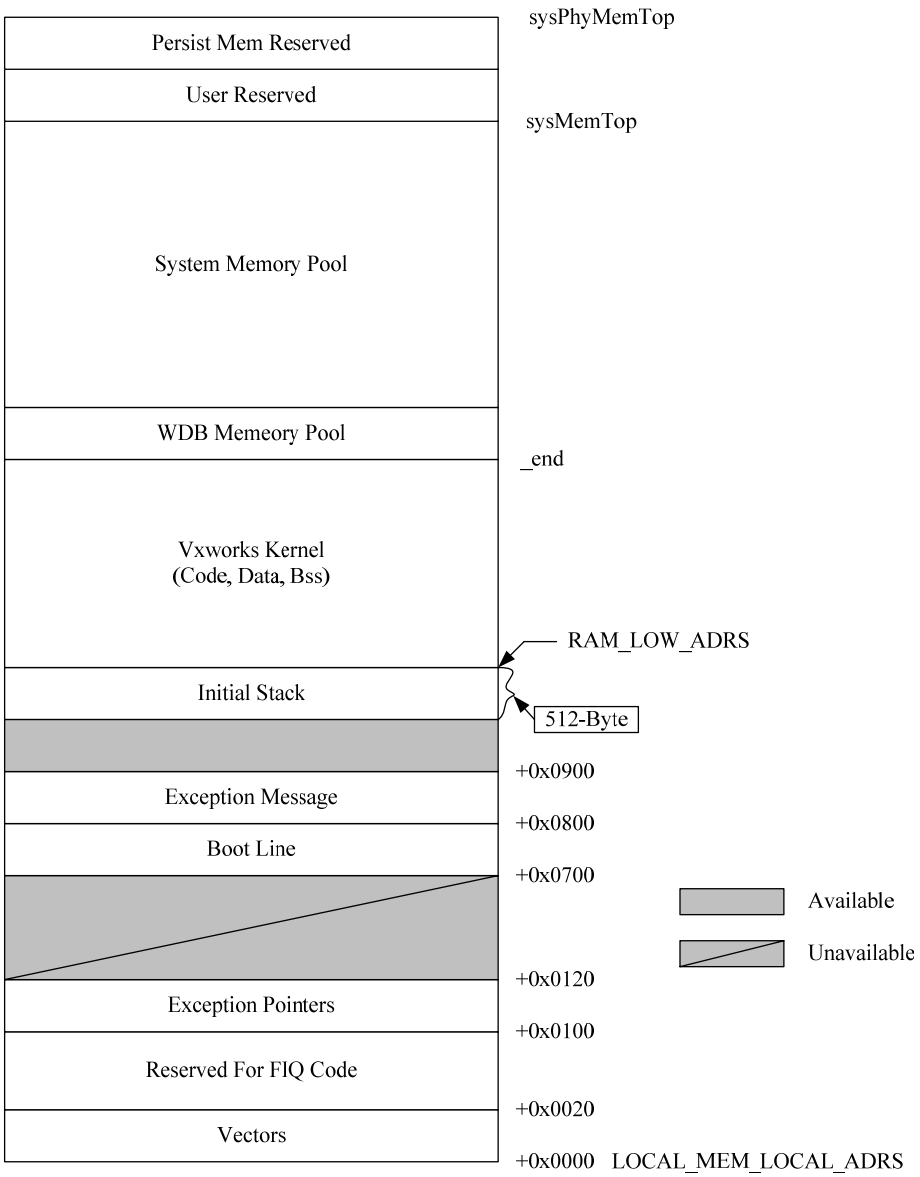


图 2-3 内核布局

Vxworks 内存管理组件配置选项

要使得 Vxworks 包含内存管理相关组件，必须在 configAll.h 或者 config.h 文件中包含如下宏定义。

INCLUDE_MMU_GLOBAL_MAP

该宏定义表示根据 sysLib.c 文件中 sysPhysMemDesc 变量定义构建系统页表。

INCLUDE_MMU_BASIC

该宏定义表示包含 vmBaseLib API，这些 API 用于对系统页表的动态管理，如添加一个新的页表项或删除一个已有的页表项或改变某个页表项的状态位（如是否使能 cache）。

注意：对于 5.5 版本的 Vxworks 内核，需要使用 INCLUDE_MMU_FULL 宏定义来进行 MMU 的初始化。

Vxworks 操作系统初始化过程中，调用 usrMmuInit 函数完成 MMU 的配置，包括系统页表

的建立以及相关硬件寄存器的配置以使能 MMU。注意：如果映射的内存空间较大，在 usrMmuInit 函数中将执行较长的时间，内核需要分配内存，创建页表项。

内存管理是操作系统实现的一个重要方面，其决定了内存的使用效率。而内存管理的实现一般都要涉及 MMU 机制，不同处理器架构具有不同的 MMU 硬件单元，虽然从总体上说，都需要一个系统页表配合硬件 MMU 单元工作，但是系统页表的层次结构以及页表项的定义都不相同，不同处理器都有自己的特殊要求。从 BSP 移植和开发角度而言，读者只需要知道要使能 MMU，必须首先在内存中创建一个系统页表，该页表表达了虚拟地址与物理地址之间的映射关系，Vxworks 下这个映射关系由 sysLib.c 文件中 sysPhysMemDesc[]数组表示，BSP 开发人员只需要根据特定平台的地址空间分配来初始化这个数组即可。另外为了包含 MMU 内核相关组件，如 usrMmuInit 函数的调用，用户还必须进行相关宏的声明，如 INCLUDE_MMU_FULL,INCLUDE_MMU_BASIC，具体配置选项，请用户参考文献“vxworks kernel progammer’s guide”。Vxworks 下对 MMU 机制以及虚拟地址和物理地址关系处理上都比较特殊，其中最为关键的一点是 Vxworks 的系统页表在初始化的过程中基本上一次性完全创建，其后基本无需进行改变，这就决定了虚拟地址和物理地址之间的映射关系是固定的，且是一一对应的关系，虽然 Vxworks 内核提供了可动态修改系统页表项的接口函数，但是很少使用。基于 Vxworks 对于 MMU 机制的特殊使用方式，Vxworks 下的文件都是可链接的文件，需用通过 Vxworks 自身提供的 loadModule 及其同族函数进行动态链接后方可执行，包括 Tornado 环境下编译生成的.out 文件也是一个可链接文件。
最后简单介绍 Vxworks 对内存的使用布局。

2.5 中断处理

中断是外设为得到 CPU 服务而向 CPU 发送一个信号，CPU 在接收到一个中断信号后，暂停当前执行的任务，转而去执行中断处理程序，完成对外设的服务。为使一个中断能够得到服务，除了操作系统需要进行配合外，还必须配置 CPU 硬件寄存器。以 ARM 处理器为例。ARM 处理器支持两个中断管脚，换句话说，ARM 处理器可以从外界接收两个不同源产生的服务需求，IRQ 为普通中断服务请求，FIQ 为快速中断服务请求，FIQ 优先级较高，当 IRQ，FIQ 同时有中断请求时，ARM 处理器优先处理 FIQ 中断。Vxworks 操作系统只使用了 IRQ 中断，FIQ 闲置未用，“未用”的含义即负责 FIQ 的内核中断响应程序只是简单的返回，而不进行任何有效的操作。实际上，对于一个平台而言，处理器提供的中断管脚根本无法满足外设的数量需求，故一般外设发出的中断都会通过一个中断控制器输入到 CPU。中断控制器是可编程的，可接收多个中断源，并对这些中断源进行优先级排序的中断源控制设备，其可以单独禁止和使能某个中断。一般而言，中断控制器具有多个中断源输入管脚，而只有一个中断输出管脚，与 ARM 处理器的 IRQ 或者 FIQ 直接相连。故而平台所有外设的中断都作为 ARM 处理器 IRQ 或者 FIQ 中断。由于 Vxworks 操作系统只支持 IRQ 中断管脚，故在 Vxworks 下，中断控制器的输出管脚必须与 ARM 处理器的 IRQ 中断输入管脚连接在一起，此时 ARM 处理器的 FIQ 管脚闲置不用。
由于此时所有外设中断源都通过中断控制器的管理以处理器单一中断的形式而存在，故操作系统对中断的响应采用了多层分级查找的方式。首先 Vxworks 内核提供一个 IRQ 中断总入口函数，由这个函数查询中断控制器得到当前请求服务的最高优先级中断号，进而有这个中断号调用对应该中断的具体的中断函数，完成一次中断的响应过程。为此，Vxworks 内核单

独维护一张外设中断程序表，该表实现上实际为一个结构数组，这个结构定义可以如下形式：

```
typedef struct {       /* VEC_ENTRY */
    VOIDFUNCPTR    routine;
    ULONG  arg;
} VEC_ENTRY;
```

根据特定平台总的外设中断源的数目，在 Vxworks 内核初始化过程中，将创建这样一个结构数组，外设中断源编号作为数据索引，而用户注册的中断处理程序则以中断源编号为索引，在数组对应的某项中填入中断函数地址和参数，完成中断程序的注册过程。当 ARM 处理器接收到一个 IRQ 中断后，首先由 Vxworks 内核提供的 IRQ 中断响应函数进行处理，该总入口函数作为 IRQ 处理函数注册到 ARM 处理器系统中断向量表中。IRQ 总入口函数并不对具体中断进行服务，其主要完成中断服务查询和转移的工作，具体如下：

1> 读取中断控制器相关寄存器，获取当前中断号。注意，中断控制器自身具有中断源优先级排序功能，故当某个时刻有多个中断产生时，中断控制器将选择最高优先级的中断进行服务，此时中断控制器相关寄存器中存储的就是这个最高优先级中断的中断号。
2> 根据获取的中断号，索引 Vxworks 自身维护的外设中断程序表，得到中断号对应的表项，调用表项中之前注册的中断处理程序，完成中断服务的转移。
3> 用户注册的中断程序被调用，完成特定中断的服务。

基于此二级中断服务机制，需要注意以下几个方面。

首先 Vxworks 内核提供了 IRQ 中断的总入口函数，所有外设中断的服务首先进过该总入口函数，之后总入口函数通过调用外设中断程序表中对应的中断函数，完成中断服务。ARM 处理器为了完成中断服务，其要求在绝对地址 0 处必须建立中断向量表，我称之为系统中断向量表，以区别于 Vxworks 操作系统自身维护的用户中断向量表。为了使能第一层次中断响应，必须配置 ARM 处理器 CPCR 寄存器，使能 IRQ 中断，即将 CPCR 寄存器中 I 位清零。这一点很重要，在基于 ARM 处理器的平台初始化代码中，我们一般都会将 CPCR 寄存器中 I，F 位置 1，禁止系统一切中断，并同时设置模式为系统特权模式，从而使具有足够的优先级进行 ARM 处理器其他寄存器的设置。在 Vxworks 下，在响应外设中断之前，必须重新将 I 位清零，事实上，在此后的内核初始化代码中，并没有具体的代码完成 I 位清零的工作，这个工作是在任务创建时附带完成的。usrInit 函数最后通过调用 kernelInit 函数创建一个内核任务完成余下的内核初始化工作，新创建的任务默认将 I 位清零，即 usrRoot 函数默认运行在系统中断使能的情况下，故此时为了完成外设中断的服务，此时只需使能二级中断即可，即配置中断控制器相关寄存器去使能某个中断，注意：在 Vxworks 内核初始化最初阶段，中断控制器被配置为禁止了所有外设中断。以后将根据具体需要单个的使能各个中断源。为了使能某个外设中断，BSP 开发人员必须提供 intEnable，intDisable 的底层实现函数，这两个函数分别完成使能和禁止某个外设中断的目的。具体工作的完成是通过配置中断控制器的相关寄存器位实现的。另 intConnect 函数完成外设中断函数的注册，即注册到 Vxworks 内核维护的外设中断程序表中。

所以为了完成二级中断转移，BSP 开发人员必须提供如下函数的底层实现。

1> intEnable 底层实现函数，对应 sysIntLvlEnableRtn 函数指针。
2> intDisable 底层实现函数，对应 sysIntLvlDisableRtn 函数指针。
3> 中断号获取函数，对应 sysIntLvlVecChkRtn 函数指针。
4> 中断应答函数，对应 sysIntLvlVecAckRtn 函数指针。

这四个函数都将被 Vxworks 内核调用，而具体实现则有 BSP 开发人员提供，故内核提供函数指针的形式，由 BSP 开发人员进行初始化，而后由 Vxworks 通过函数指针的方式调用这些底层实现函数。
其中中断号获取函数和中断应答函数将被 IRQ 中断总入口函数调用，完成中断转移和应答。这四个函数的实现都必须具体的操作中断控制器，而中断控制器是平台相关的，故必须作为 BSP 的一部分来实现。

Vxworks 内核提供 intEnt（intALib.s）作为 IRQ 中断总入口函数，该函数完成相关设置后，调用__func_armIrqHandler 函数(excArchLib.c)指针指向的函数完成具体的中断服务。__func_armIrqHandler 函数指针默认初始化为指向 excIntHandle 函数(excArchLib.c)，excIntHandle 函数只是调用 excIntInfoShow(excArchShow.c)打印出相关中断信息，并不服务中断，在 Vxworks 内核中断服务初始化过程中(sysHwInit2 函数中通过调用 intLibInit，sysHwInit2 在 sysClkConnect 中被调用)，__func_armIrqHandler 函数指针被重新初始化为指向 intIntRtnNonPreempt 或者 intIntRtnPreempt。其中 intIntRtnPreempt 函数支持中断嵌套，即允许高优先级的中断打断当前低优先级服务，转而服务高优先级中断；而 intIntRtnNonPreempt 则不支持中断嵌套，即当系统在服务低优先级中断时，不允许高优先级中断发生，此时 ARM 处理器中 I 位被置 1. 如果工作在允许中断嵌套的情况下，系统对于嵌套的层次有所限制。当嵌套层次超过系统最大允许值时，将不再允许中断嵌套的发生。由于中断发生时，ARM 处理器 I 位被置 1（intEnt 函数中完成），故实际上 intIntRtnNonPreempt 和 intIntRtnPreempt 区别主要在于 intIntRtnPreempt 函数中重新将 ARM 处理器 I 位清零，从而允许其他高优先级中断在服务当前中断的过程中产生，即产生中断嵌套。通过以上分析，基于 ARM 处理器 Vxworks 的中断服务流程如下：
中断产生→intEnt→intIntRtnNonPreempt 或者 intIntRtnPreempt→用户注册的具体的中断函数。

注意以上中断服务流程中，具体完成中断服务的是最后被调用的用户注册的中断函数，intEnt 以及 intIntRtnNonPreempt 或者 intIntRtnPreempt 函数都是做辅助性的工作，当然这些辅助性的工作对于内核而言是至关重要的。

由于 ARM 处理器只有两个中断管脚，而 Vxworks 操作系统只使用了 IRQ 普通中断管脚，故平台下所有外设中断都是通过一个中断控制器外部设备进行管理，包括系统时钟中断也是通过中断控制器向 ARM 处理器请求中断服务。系统时钟中断是整个 Vxworks 操作系统的脉搏，其是任务调度的固定触发点，也是系统各种定时器的源，对于整个系统而言具有至关重要的作用。以下我们即以系统时钟中断的初始化和响应过程为例介绍 Vxworks 下中断注册和服务过程。

Vxworks 系统时钟中断用户层注册函数为 sysClkInt。
系统时钟中断的注册在 sysClkConnect 中完成。sysClkConnect 被 usrRoot 调用专门负责系统时钟的初始化。该函数被调用情景如下代码所示。

```
sysClkConnect ((FUNCPTR) usrClock, 0);/* connect clock interrupt routine */
sysClkRateSet (SYS_CLK_RATE);        /* set system clock rate */
sysClkEnable (); //配置外设时钟，使其正常工作，按固定时间间隔产生时钟中断
```

注意：sysClkConnect 的第一个输入参数为系统时钟中断的服务例程，然而这并非直接中断服务例程，换句话说，从上文分析的中断服务流程来看，其还不属于用户注册的中断函数，而是更低一级，由用户注册的中断函数调用。事实上，对于系统时钟中断而言，用户注册的中断函数是 sysClkInt，这在 sysHwInit2 中注册完成的。sysHwInit2 由 sysClkConnect 函数调用，如下为 sysClkConnect 函数（xxx_timer.c，其中‘xxx’表示相关平台）实现。

```
STATUS sysClkConnect
(
    FUNCPTR routine, /* routine to be called at each clock interrupt */
    int arg          /* argument with which to call routine */
)
{
    static BOOL beenHere = FALSE;

    if (!beenHere)
    {
        beenHere = TRUE;
        sysHwInit2 ();
    }

    sysClkRoutine    = NULL;
    sysClkArg          = arg;
    sysClkRoutine    = routine;

    return (OK);
}
```

作为参数传入的 usrClock 函数地址被存储在 sysClkRoutine 函数指针中，而不是作为 intConnect 的函数注册到外设中断程序表中。其中调用的 sysHwInit2 完成系统时钟中断程序的注册，如下代码所示。

```
/*sysLib.c*/
void sysHwInit2 (void)
{
    static BOOL initialised = FALSE;

    if (initialised)
        return;

    /* initialise the interrupt library and interrupt driver */
    intLibInit (NUM_OF_INTERRUPT,NUM_OF_INTERRUPT,
                        INT_NON_PREEMPT_MODEL );
    xxxIntDevInit ();


    /* connect sys clock interrupt */
```

```
    (void)intConnect ( INUM_TO_IVEC(INT_TINT0), sysClkInt, 0);
    intEnable ( INT_TINT0 );

    …

    initialised = TRUE;

}
```

sysHwInit2 首先调用 intLibInit 完成内核中断服务的相关初始化，其中最重要的是重新初始化__func_armIrqHandler 函数指针，intLibInit 函数的第三个参数指定是否允许中断嵌套，此处调用中不逊于中断嵌套，故__func_armIrqHandler 函数指针被初始化为指向 intIntRtnNonPreempt 函数；继而调用 xxxIntDevInit 函数完成前文所述的四个中断底层实现函数的注册，即初始化 sysIntLvlEnableRtn，sysIntLvlDisableRtn，sysIntLvlVecChkRtn，sysIntLvlVecAckRtn 四个函数指针（这四个函数指针的含义见上文分析）；而后调用 intConnect 进行外设中断程序的注册，即将 sysClkInt 函数作为系统时钟中断的服务程序注册到内核维护的外设中断程序表中，最后调用 intEnable 使能该中断，intEnbale 实现上调用 sysIntLvlEnableRtn 指向的函数（BSP 开发人员实现）完成中断控制器的配置，从而使能系统时钟中断。注意可以将此处的 intEnable 函数放入 usrRoot 中调用的 sysClkEnable 函数（xxx_timer.c）中，如果此处进行了 intEnable 的调用，则无需在 sysClkEnable 中再次进行调用，此时 sysClkEnable 只需进行定时器外设的硬件配置即可。所以单从系统时钟中断的角度来看，sysHwInit2 将 sysClkInt 函数注册到内核中作为了系统时钟中断的服务程序。sysClkInt 定义在 xxx_timer.c 中，其代码结构如下。

```
void sysClkInt (void)
{
    if( ( sysClkRunning == TRUE ) && ( sysClkRoutine != NULL ) )
    {
        /* call system clock service routine */
        (* sysClkRoutine) (sysClkArg);
    }
}
```

在 sysClkConnect 函数中，sysClkRoutine 被设置为第一个参数指向的函数，即 usrClock。由此我们可以将系统时钟中断产生时整个系统的服务流程表示如下：

系统时钟中断产生→intEnt→intIntRtnNonPreempt→ sysClkInt→usrClock→tickAnnounce

usrClock 调用的 tickAnnounce 函数由 Vxworks 操作系统提供，tickAnnounce 函数主要完成如下工作：

5. 对 vxTick 变量作加一运算。vxTick 表示系统自启动之时到现在的 tick 数，所以用 vxTick 乘以系统时钟间隔就是开机时间。
6. 对处于等待状态的任务进行检查，将超时任务（那些调用 taskDelay 延迟的任务）重新设置为 ready 状态，并转移到调度队列中。
7. 遍历内核工作队列，对延迟的内核工作进行执行。
4. 进程调度。选择最高优先级任务作为当前任务运行。

系统时钟中断的服务过程相对其他中断较为“曲直”，其在用户层又进行了分层。对于其他硬件中断，通过 intConnect 函数注册到系统外设中断程序表中的中断函数一般作为服务具体完成的场合，而不再有进一步中断转移的过程。系统时钟中断的特殊之处在于其具体中断服务函数（tickAnnounce）是由内核提供的，但又必须在用户层对时钟进行管理，故必须经过中间一步转换的过程。

中断上下文

无论是在阅读相关操作系统理论书籍或者实际代码时（如 Linux），基本都说到了在中断处理程序中，不可以调用可以引起睡眠或者阻塞的函数。经典的解释如下：

“Code running in interrupt context is unable to sleep, or block, because interrupt context does not has a backing process with which to reschedule. Therefore, because interrupt handler is not associated with a process, there is nothing for the scheduler to put to sleep and, more importantly, nothing for the scheduler to wake up …”

这是我们听到的最为经典的解释：因为中断是运行在中断上下文中，所以不可以调用可引起阻塞或者睡眠的函数。再进一步：因为调度必须是以进程为条件的，而中断并没有这样一个对应的进程存在。具体到 Linux，我们知道，当前执行进程控制块由 current 指针指向，当进程调度时，current 指针被赋值指向下一个即将运行的进程。当某个进程运行时，发生一个中断，此时 CPU 按照一定的流程从用户或者内核进程切换到中断处理程序运行。注意，此时 current 指针仍然指向被中断的进程。对于在用户态被中断的情况来看，其与系统调用的情景差不多，最大的差别在于中断时硬件自动完成全局中断的禁止（注意这一点在下文的讨论中是至关重要的）。此时将从用户态切换到内核态，以下寄存器将被压入进程的内核堆栈：用户态 SS，用户态 ESP，EFLAGS，用户态 CS，用户态 EIP，出错码（if any，对于外部中断，都没有）。中断处理程序将使用进程的内核堆栈。对于在内核态被中断的情景，此时不发生堆栈的切换，其他相同。以从用户态被中断的情景为依据进行分析，我们可以分析得出，其与用户进程进行系统调用时发生的情况基本完全相同。一个最为本质的区别是中断时硬件自动关闭了系统中断使能位，故直到中断处理程序退出或者其主动开启使能位之前，所有其他系统中断都将被屏蔽，包括 timer interrupt，这个进程调度的脉搏和激励源。由此不同之处，我们可以分析出，为何一般的系统调用可以进行睡眠，而由中断引起的进程暂停不可以进行睡眠。

首先需要提及的一点是，系统调用一般是为进行系统调用的进程服务的，或者说，就是我们通常所说的服务于用户进程，即此时的内核执行代码与当前 current 指向的进程是相绑定的。如果内核执行过程中某个条件不满足，需要进行等待，那么其绑定的进程进入睡眠状态是合乎情理的。对于中断的情况，则完全不同，中断的发生完全是异步的，这就是说当中断发生时，我们无法预知当前执行的是哪一个进程。换句话说，单从上下文来看，中断发生时，当前执行进程被这个中断强行绑架了。或者说，中断处理程序的执行并不是以当前进程的“利益”为其首要准则，他是一个显然的侵犯者，他为着自身的利益强行中断了当前进程的执行。此时除了全局中断被硬件处理流程关闭外，系统所有的上下文环境与普通的系统调用没有任何差异！注意以上这一点。对于中断处理程序（或者说中断上下文中）不可以调用可引起睡眠或者阻塞的函数的传统经典解释中，仅仅表达了“因为是中断上下文，不是进程上下文，所以不可调用。”简直就是废话！从以上的说明中，我们可以看出，中断就是执行在进程上下文中，current 指针依然指向一个有效的位置：就是当前被中断的进程（无论这个进程与这个中断相关还是不相关，相关不相关只是人类的理解，计算机可没有这点认识）。所以如

果此时中断处理程序中调用一个可引起睡眠（即引起进程调度）的函数，那么 switch_to 仍然可以毫无困难的执行，至多只是将一个被绑架的进程调度了出去。当这个被绑架的进程被调度回来后（暂不考虑是如何被调度回来的，下文讨论这一点），那么他将仍然继续之前的处理：先执行完尚未执行完的中断处理代码，而后退出，从被绑架进程内核堆栈中弹出 CS，EIP 等等，恢复这个被绑架的可怜的进程的执行，有何不可？？

以上讨论中跳过了一点，即被绑架进程被调度出去后，是如何被调度回来的。由于我只作为反方进行论证，所以我只给出一种可能的情况，只要能被调度回来就可以。例如中断程序调用 kmalloc 函数引起睡眠，那么此时被绑架进程将被挂入到一个内核指定的队列中，当系统又有可用内存时，内核相关代码会唤醒由于 kmalloc 调用而睡眠的进程，当然包括我们的这个可怜的被中断绑架的进程。在某个时刻这个被绑架进程重新被调度回来进行执行，有何不可？

聪明的读者可能已经意识到了，以上的讨论中省略了最为关键的一点：即调度器要在工作着。换句话说，系统的脉搏要在搏动。但是注意到当中断发生时，硬件处理流程关闭了系统中断，包括系统脉搏中断 timer interrupt。

首先我们讨论一下没有 timer interrupt 进程调度的可能性。没有了 timer interrupt，系统能否工作？回答是：不能。如此情况下，进程的调度将完全依赖于一个执行的进程主动调用 schedule 函数。对于某些进程这一点是可以满足的，但是对于 Linux 内核中绝大多数进程这一点是不满足的，只要当前被调用执行的进程不曾主动调用 schedule 函数，那么我们只能等待他执行到死。对于用户而言，此时将表现为系统死机。因为通常我们用户判断系统是死是活，是通过界面是否响应用户的命令，对于一个一直在执行某特定单一进程的操作系统而言，这一点是无法完成的，那就是我们干什么（你可以摔鼠标，捶键盘，或者对着机箱飞一脚），系统都没有响应，那不就是系统死了。（此处讨论的很不深入，不过到此我已经达到了说明的目的了。）

以上是最为平常的情况，即中断处理程序一般的工作方式（禁止嵌套中断）。现在我们讨论一下另一种可能性：中断处理程序在一开始就将系统关闭着的中断开启，即允许中断嵌套。此时 timer interrupt 获得了执行的资格，系统重新拥有了脉搏。那么被中断绑架的进程将有希望被重新调度。问题是，如果这个中断的中断源自系统开机到关机之间终其一生只发出一个中断，那么将不会造成任何问题。最多被这个中断绑架的进程倒霉一点，无故多延迟等待一会执行而已，不至于造成整个系统死掉。问题的关键有两点：其一，中断会发生 N 多次；其二，每发生一次，系统就多一个进程被强行绑架。第一点引起的问题是，基本每次中断都会对同一个问题进行加重，如对同一个资源的重新申请；第二点引起的问题是，系统到最后可能所有的进程都被绑架，到最后系统无进程可以调度，因为所有的进程都在被绑架着睡大觉，甚至包括 idle 进程。这个陷阱的出路是：每次下一个中断来临前，前一次中断退出。但考虑到如果一旦允许中断处理程序中可以调用可引起阻塞或者睡眠的函数，那么引起该类问题的中断源就不只一个了，某个小中断源可以（怎么定判，再说了），串口可以否，硬盘可以否，网口可以否，继而 timer interrupt 可以否？显然不可以。既然有不可以，那么大家就都不可以。这一点不是可以商量的，那是必须的。所以就有中断处理程序中（或者说中断上下文中）不可以调用可引起阻塞或者睡眠的函数。这是一条必须严格遵守的游戏规则。但是如果强行不遵守它，是不是一定出问题？不见得，如果中断频率不高，而调用的可引起阻塞的函数不是 VIP 高级会员，如 kmalloc（在系统内存足够时），那么调用一下不会引起什么

问题，但是这是一个错误，是一个潜在的危险源，你只是由于侥幸而未得到惩罚。如果有一天系统死机了，那么这可能就是罪魁祸首！

第三章 Vxworks 映像及启动

本章我们将着重分析 Vxworks 映像类型以及启动方式，主要分为如下几个方面进行介绍，首先简单介绍一个 Vxworks 操作系统的几种启动方式，每种启动方式下映像组成和基本启动流程；其次详细介绍下载启动方式下 bootrom 组成和执行流程；其次详细介绍 Vxworks 本身的启动过程；最后我们对 BSP 下的文件组成进行较为详细的介绍。

3.1 Vxworks 启动流程分析

启动类型

按 vxWorks 内核的下载形式，vxWorks 启动总体上分为两种方式：

1. bootrom+vxWorks
   此时 bootrom 被烧入 ROM 中，而 vxWorks 内核映像通过串口或者网口下载到系统 RAM 中。
2. vxWorks 直接从 ROM 中运行，不借助于 bootrom 引导程序。
   此时 vxWorks 内核映像被烧入 ROM 中，无 bootrom 程序

映像构成

1. bootrom 引导程序，一般被称为 vxWorks boot Image。
   该程序由如下文件创建而成：romInit.s, bootInit.c, sysALib.s, sysLib.c, bootConfig.c, 设备驱动程序。注意：虽然 bootrom 中包含 sysALib.s,文件，但是并未使用其中定义的任何函数，这与 ROM 形式的 Vxworks 内核映像类似。

2. vxWorks 内核映像，一般被称为 vxWorks application Image。
   vxWorks 内核映像是系统启动后实际运行的程序（操作系统本身）。对于下载形式的 vxWorks 内核映像，由于需要对硬件环境进行重新初始化（即不依赖于 romInit 函数所作的初始化工作），所以最先需要运行 sysInit 函数进行重新初始化，这些初始化工作与 romInit 函数基本相同，但是不再需要对内存控制器进行重新初始化，这是 sysInit 区别于 romInit 函数之处。sysInit 最后跳转到 usrInit 函数。而对于 ROM 方式直接运行的 vxWorks 内核映像，由于 romInit.s 函数此时是作为内核映像的一部分，所以就省去了对 sysInit 函数的调用。下载型和 ROM 型 Vxworks 内核文件构成上有一些差别。
   a) 下载形式的 vxWorks 内核映像组成
   sysALib.s, sysLib.c, usrConfig.c, 设备驱动程序文件。
   b) ROM 形式的 vxWorks 内核映像组成
   romInit.s, bootInit.c, sysALib.s, sysLib.c, usrConfig.c, 设备驱动程序文件。
   注意：虽然 ROM 形式的 vxWorks 内核映像并未使用 sysALib.s 文件中函数（sysInit），但是该文件仍然作为一部分存在于该 vxWorks 内核映像中。当然用户可以修改 defs.bsp 文件中 MACH_DEP 宏定义，将 sysALib.o 从如下形式定义中去掉即可。
   MACH_DEP = sysALib.o sysLib.o $(MACH_EXTRA) $(ADDED_MODULES)

压缩 vs 非压缩

在 Vxworks 操作系统过程中可能使用到的 BootRom 以及 Vxworks 内核映像本身都可以存在两种方式：压缩的和非压缩的。

如果没有进行压缩，则只有一次重定位，即从 ROM 到 RAM 只存在一次代码拷贝过程，所有 ROM 中存储的代码一次性都被拷贝到 RAM 中：

1. 对于 BootRom 而言，所有代码一次性被拷贝到 RAM_HIGH_ADR 指定地址处。如下图 3-1 所示。

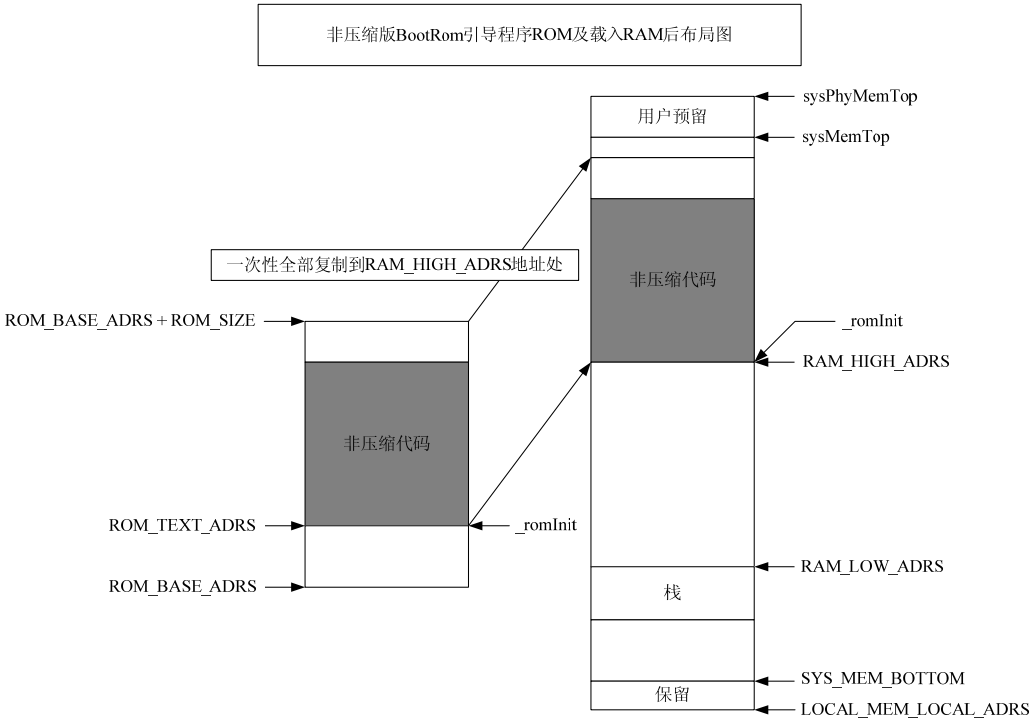


图 3- 1 非压缩版 BootRom 引导程序 ROM 及载入 RAM 后分布图

2. 对于 Vxworks_rom 而言，所有代码一次性被拷贝到 RAM_LOW_ADR 指定地址处。如下图 3-2 所示。

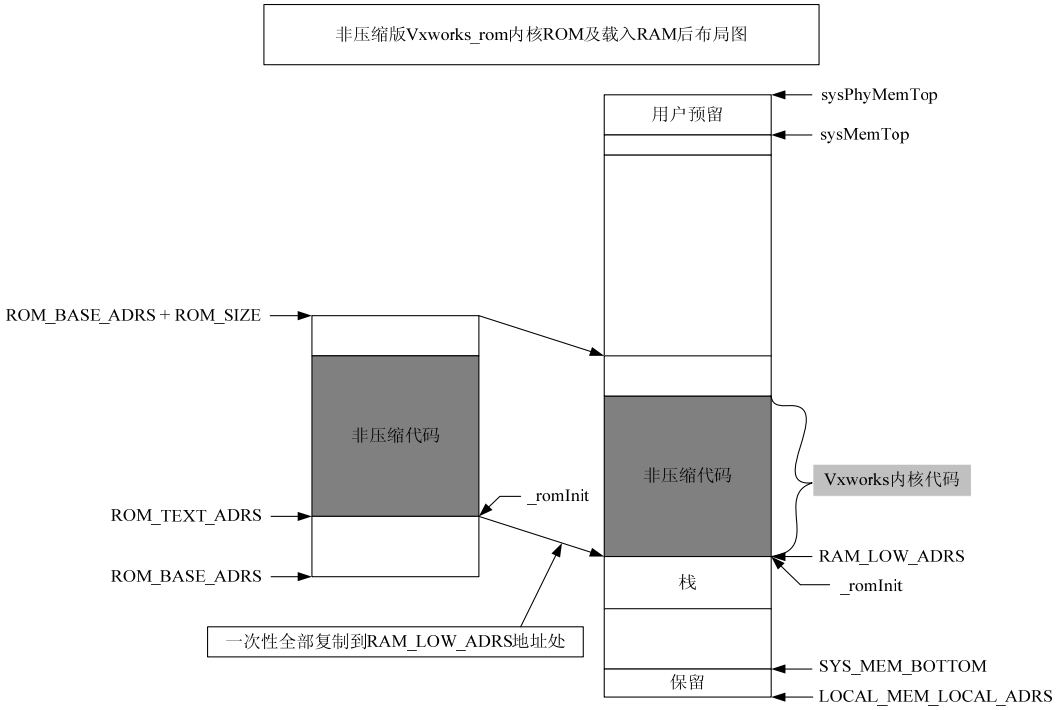


图 3- 2 非压缩版 Vxworks_rom 内核映像类型 ROM 及载入 RAM 后布局图

如果有进行压缩，则代码拷贝过程将分为两次，一次是非压缩代码，另一次是压缩代码，且二者拷贝到内存不同位置处。

1. 对于 BootRom 而言，非压缩代码（romInit.s，bootInit.c）被直接拷贝到 RAM_LOW_ADR 处；压缩代码拷贝到 RAM_HIGH_ADR 处，并在拷贝过程中完成解压缩。如下图 3-3 所示。

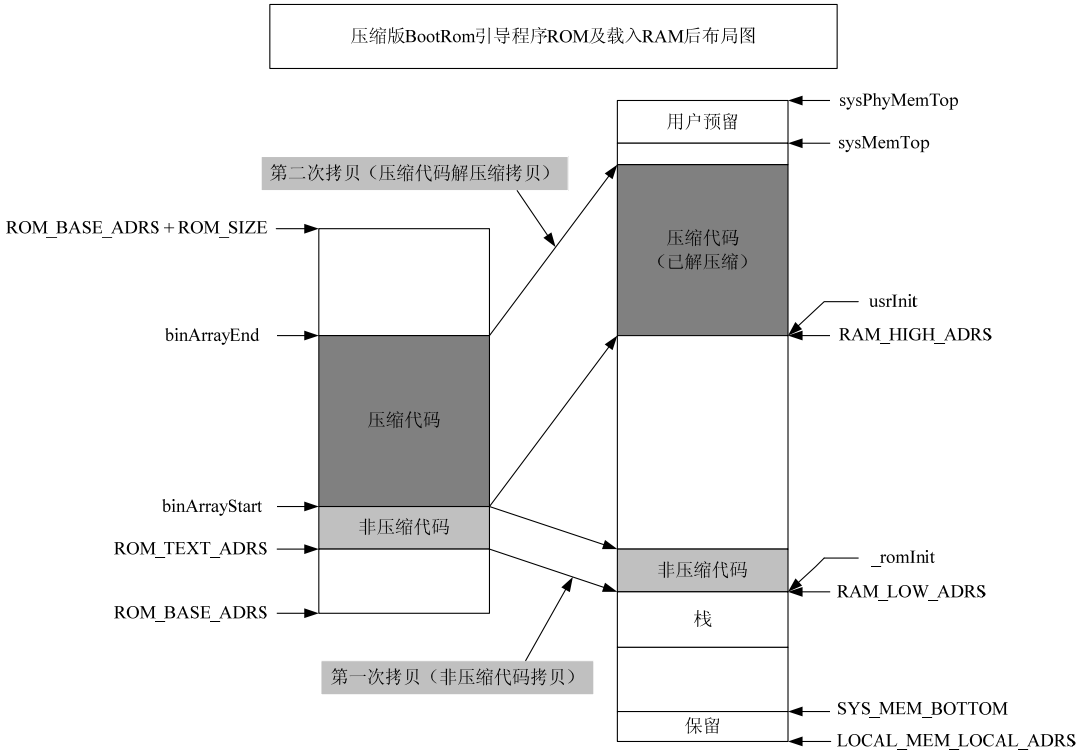


图 3- 3 压缩版 BootRom 引导程序 ROM 及载入 RAM 后分布图

2. 对于 Vxworks_rom 而言，非压缩代码（romInit.s，bootInit.c）被直接拷贝到 RAM_HIGH_ADR 处；压缩代码拷贝到 RAM_LOW_ADR 处，并在拷贝过程中完成解压缩。如下图 3-4 所示。

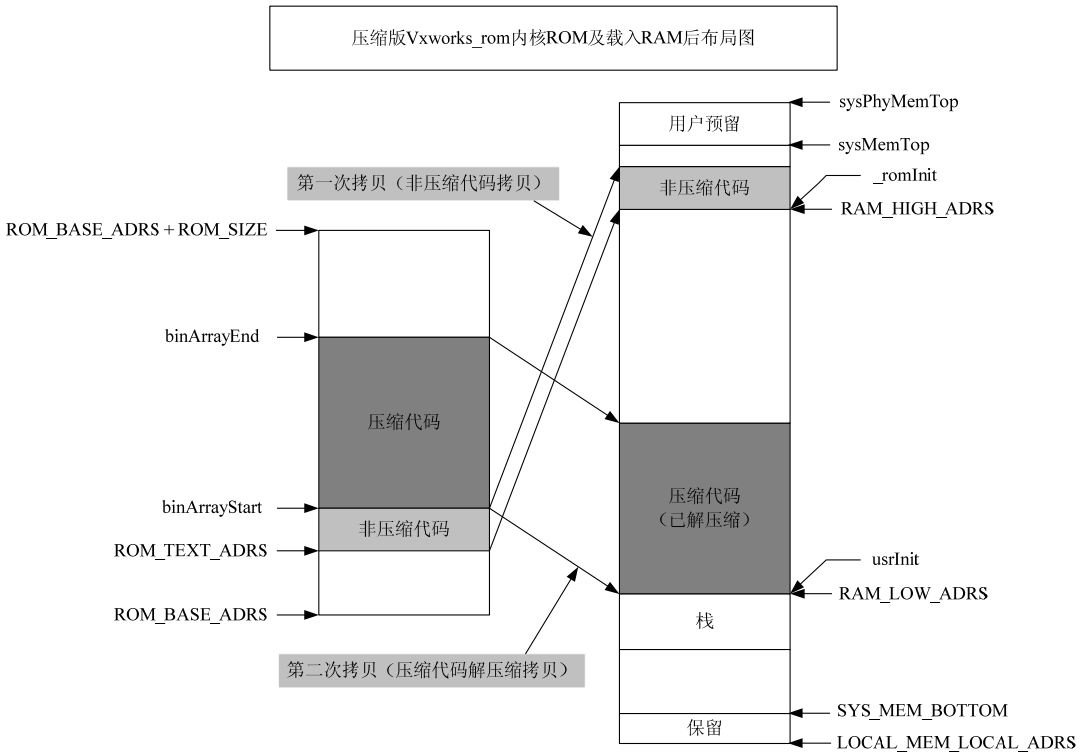


图 3- 4 压缩版 Vxworks_rom 内核映像类型 ROM 及载入 RAM 后分布图

事实上，对于压缩版本，在进行编译时，进行了两次代码链接：一次是针对压缩代码的链接，另一次是将非压缩代码和压缩代码整合在一起时的链接。注意在进行非压缩代码和压缩代码的整合时，压缩代码部分是作为数据进入整个映像的，所以不会对其进行重新链接。对压缩代码的链接实际上是在代码被压缩之前完成的，完成代码的链接后，生成特定格式的目标文件（如 ELF 或者 COFF），为了在解压缩后直接可以执行，所有必须首先将其转成二进制可执行文件，通过 objcopy 函数完成此项功能，此后对生成的二进制可执行文件调用 deflate 函数进行压缩，由于需要将压缩后代码最后整合到整个映像中，故压缩后文件还需要通过 binToAsm 工具进行转换，转成一个汇编源文件，该文件将压缩后二进制代码作为数据部分进行保存，从而避免在与非压缩代码最后进行二次链接时被修改。两步链接过程中各自指定了不同的链接地址，对于压缩代码而言，其分为两种情况，对于 BootRom，压缩代码被链接到了 RAM_HIGH_ADRS 地址处，而对于 Vxworks_rom，则被链接到了 RAM_LOW_ADRS 地址处，在于非压缩代码进行整合链接时指定的链接地址（即_romInit 函数地址）与压缩代码链接时指定的地址相对应：如果压缩代码为 RAM_HIGH_ADRS，则最后整合时就为 RAM_LOW_ADRS；如果压缩代码为 RAM_LOW_ADRS，则整合时就为 RAM_HIGH_ADRS。

下载启动方式详解

下载形式的 vxWorks 启动方式，需要 bootrom 引导程序，该程序将被烧录到开发板的 ROM 或者 Flash 中，在上电时，系统将自动跳转到 ROM 或者 Flash 起始地址处运行该 bootrom 引导程序，该引导程序进行必要的系统初始化从而为下载 vxWorks 内核映像做准备，如通过网口下载时则需要先调用网口驱动程序进行网口初始化。由于 bootrom 的主要工作是下载真正的 vxWorks 内核映像到系统 RAM 中，所以其进行的初始化工作也是为这一目的，在文件组织上，虽然从上文中可以看到其与 vxWorks 内核映像的文件组成非常相似，但其使用 bootConfig.c 文件，而 vxWorks 内核映像则使用 usrConfig.c 文件。而这两个文件中虽然定义有相同的函数名（如 usrInit，usrRoot），但在实现上存在很大的差异：bootConfig.c 文件实现完成从某个服务器下载真正的 vxWorks 内核映像到系统 RAM 中（由变量 RAM_LOW_ADRS 指向的内存地址处），并跳转到 sysInit 函数（定义在 sysALib.s 文件中第一个（且必须是第一个）函数，该函数即被装载到变量 RAM_LOW_ADRS 指向的内存地址处）执行；usrConfig.c 文件实现完成一个操作系统正常运行所需要的所有初始化工作。简言之，bootConfig.c 文件完成 vxWorks 内核映像的下载，而 usrConfig.c 完成初始化 vxWorks 操作系统的所有工作。

按 bootrom 执行方式的不同，bootrom 文件存在如下三种文件类型：

A．bootrom.bin

压缩 bootrom 文件形式，此种文件形式主要为解决开发板上 ROM 或者 Flash 的空间限制。注意：压缩并非对所有的文件进行压缩，而是除了 romInit.s, bootInit.c 文件的其它所有文件，因为解压缩程序必须是非压缩形式的。

Bootrom 执行流程如下：以下采用“文件名：函数名：初始是否被压缩”形式进行说明。

romInit.s：romInit：非压缩

romInit 函数完成系统硬件环境的必要的初始化工作，如从系统角度禁止中断，初始化相关寄存器到可知状态，初始化内存控制器，初始化函数调用所需的栈，最后跳转到 romStart

（）函数执行。

bootInit.c：romStart：非压缩
romStart 函数主要完成如下任务：
首先将 romInit 函数及其自身拷贝到 RAM_LOW_ADRS 变量指向的 RAM 区，以便从 RAM 执行。注意此后在下载 vxWorks 内核映像时，内核映像也被下载到 RAM_LOW_ADRS 变量指向的内存处，所以这部分代码此后将被覆盖，同样 bootrom 占用的所有其它内存最后也将被 vxWorks 内核回收（如由 RAM_HIGH_ADRS 指向的内核空间）。

1> 将 bootrom 程序其它部分（压缩部分）从 ROM 区拷贝到 RAM 区(由 RAM_HIGH_ADRS 变量指向的内存地址处)，并解压缩。
2> 对于 cold boot（冷启动）方式，将其它 RAM 区清零。
3> 最后跳转到 usrInit（注意是 bootConfig.c 文件中定义的）函数进行执行。

bootConfig.c：usrInit：压缩
usrInit 函数此处主要进行外围硬件初始化，为下载 vxWorks 内核映像做初始准备，而后创建 tUsrBoot 进程调用 usrRoot 程序进一步完成驱动程序初始化工作，为下载 vxWorks 内核映像做进一步工作，最后创建 tBoot 进程调用 bootCmdLoop 函数完成 vxWorks 的内核映像下载。在下载完成后，调转到下载起始地址处执行，此时下载型 Vxworks 内核入口函数 sysInit 将被调用执行，真正开始 Vxworks 操作系统的启动过程。最终在 usrConfig.c:usrRoot 函数执行完毕后，vxWorks 操作系统即完成启动，系统进入正常运行状态。函数执行流程如下：(采用“映像类型：文件名：函数名”形式)
bootrom:romInit.s:romInit→bootrom:bootInit.c:romStart→bootrom:bootConfig.c:usrInit→bootrom:bootConfig.c:usrRoot→bootrom:bootConfig.c:bootCmdLoop→
完成 vxWorks 内核映像下载，并跳转到 Vxworks 内核入口函数（sysInit）执行代码
→vxWorks:sysALib.s:sysInit→vxWorks:usrConfig.c:usrInit→vxWorks:usrConfig.c:usrRoot
→vxWorks 操作系统完成启动

如下图 3-5 所示为 bootrom 完成下载 vxWorks 内核映像后 RAM 布局，此时之前 bootrom 占用的所有区域均被 Vxworks 操作系统回收。

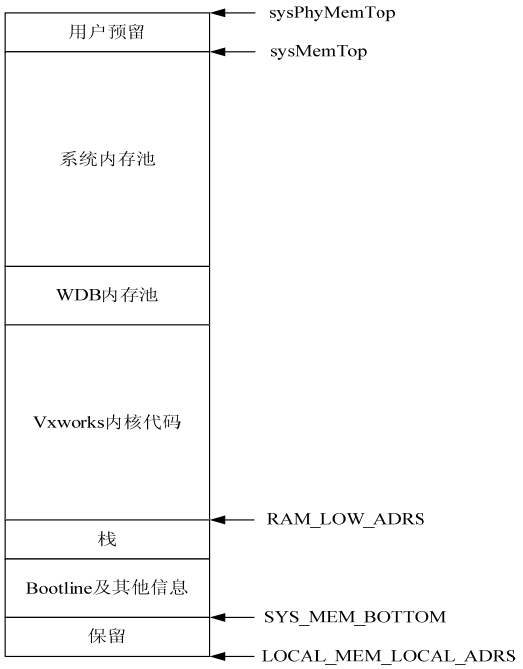


图 3- 5 Vxworks 操作系统启动后内存分布图

B．bootrom_uncmp.bin

C．bootrom_res.bin

这两种形式的 bootrom 执行流程基本同于 bootrom.bin。差别在于：

1．bootrom_uncmp.bin

该文件类型为非压缩类型，所以区别以上压缩形式的地方在于 romStart 函数将 bootrom 从 ROM 移动到 RAM 中时，无须进行解压缩工作，移动到的 RAM 区地址仍然由 RAM_HIGH_ADRS 变量指定。注意：对于非压缩 bootrom，所有代码（包括已经执行的 romInit 函数，当前执行的 romStart 函数，构成 bootrom 的其他所有函数）都一次性从 ROM 拷贝到 RAM 中，RAM 中目的地址由 RAM_HIGH_ADRS 变量指定，如前文图 3-1 示。函数执行流程同压缩版本。

2．bootrom_res.bin

res 全词为 resident，此处的意思为程序代码部分将驻留在 ROM 中，所以不同于上文中的地方即 romStart 函数只复制 bootrom 引导程序的数据段到 RAM_HIGH_ADRS 变量指向的 RAM 区。代码执行还是从 ROM 区中取，此种方式一般在开发板 ROM 区较大时采用，但会增加执行时间，另外由于需要从 ROM 中直接执行，故也不能采用代码压缩方式。

下载方式中使用的 Vxworks 内核映像一般命名为’vxworks’（不包含内核符号表，此时创建一个独立的内核符号表文件 vxworks.sym，如果需要将内核符号表包含其中，则在 config.h 文件中定义 INCLUDE_STANDALONE_SYM_TBL 组件）或者‘vxworks_st’（内含内核符号表），而其他启动方式（一般就为 ROM 启动方式）下的 Vxworks 内核映像名都具有一个后缀，如 vxworks_rom 表示从 ROM 直接启动，vxworks_res_rom 表示代码部分驻留在 ROM

中，当然也是从 ROM 直接启动。下载方式下 Vxworks 内核映像文件构成以及完成下载后 Vxworks 内核函数的调用流程前文已有交代，此处不再赘述。

ROM 型启动方式详解

所有 vxWorks 内核映像类型，其中只有 vxWorks 类型使用上文中介绍的 bootrom 引导程序进行启动，此时 Vxworks 内核映像放置在主机端，由目标板 bootrom 完成 Vxworks 的下载，一般通过网口（或串口）方式进行。其他类型（文件名中带有 rom 字样的）都无须 bootrom 的配合，也即不需要 bootrom。因为这些 vxWorks 类型的内核映像自身（而非 bootrom）被烧入开发板系统 ROM 或者 Flash 中，均无须进行下载，系统上电时，将直接跳转到 vxWorks 内核映像入口函数执行操作系统初始化工作。

ROM 型启动方式下的 VxWorks 内核映像类型如下：

vxWorks_rom.bin

非压缩版Vxworks内核映像类型不含内核符号表。如果需要包含内核符号表，则在config.h 文件中定义INCLUDE_STANDALONE_SYM_TBL组件。

vxWorks.res_rom.bin

非压缩版代码驻留 ROM 中执行的 Vxworks 内核映像类型（含内核符号表）。

vxWorks.res_rom_nosym.bin

非压缩版代码驻留 ROM 中执行的，不包含内核符号表的 Vxworks 内核映像类型。

vxWorks.st_rom.bin

压缩版本的，内含内核符号表的 Vxworks 内核映像类型。

vxWorks_romCompress.bin

压缩版本，不含内核符号表的 Vxworks 内核映像类型。

不同 Vxworks 内核版本下，在内核映像名称上会有所变化，但一般都包含在以上类型之中。从这些映像类型来看，主要有以下区分：

1. 是否 ROM 驻留方式，即代码是否被拷入 RAM 执行。
2. 是否进行了压缩
3. 是否包含符号表

注意：Vxworks 内核映像是否包含内核符号表对于后续函数调用非常重要，如在串口命令行下敲入一个函数名称，如果内核映像不包含内核符号表，则即便内核中包含该函数定义，则在终端也会给出‘undefined symbol‘错误。

以下从三个不同角度介绍他们的不同点，不过首先以 vxWorks_rom.bin 文件类型介绍一下执行的基本流程：（采用“文件名：函数名”方式）

romInit.s：romInit

此处 romInit 函数完成的工作同 bootrom，实际上二者使用相同的 romInit 函数实现。

bootInit.c：romStart
首先将其自身拷贝到 RAM_HIGH_ADRS 变量指向的 RAM 区，以便从 RAM 执行。注意与上文中 bootrom 的区别，虽然使用的是相同的 romStart 函数实现，但在 RAM 地址上存在差别，关键点在于 vxWorks 内核映像被复制到 RAM_LOW_ADRS 指向的内存地址处。
romStart 函数完成如下任务：
1> 由于是非压缩版本的 Vxworks 内核映像，故所有代码一次性从 ROM 拷贝到由 RAM_LOW_ADRS 变量指向的 RAM 内存处。基本拷贝过程见前文图示。
2> 对于 cold boot（冷启动）方式，将其它 BSS 区清零。
3> 最后跳转到 usrInit（注意是 usrConfig.c 文件中定义的）函数进行执行。

此后的执行流程如下：
usrConfig.c:usrInit→usrConfig.c:usrRoot
usrConfig.c:usrRoot 函数执行完毕后，vxWorks 操作系统即已完成启动，系统进入正常运行状态。

其他 ROM 型 Vxworks 内核映像启动流程差别如下：
1. 是否 ROM 驻留
   对于 ROM 驻留方式而言，在以上第一步中只将数据部分从 ROM 载入到由 RAM_LOW_ADRS 指向的内存区域，代码仍然滞留在 ROM 中，此后一直从 ROM 中读取代码执行，此种执行方式造成效率的些微损失，一般也较少使用。涉及的文件类型如下：
   vxWorks.res_rom.bin，vxWorks.res_rom_nosym.bin
2. 是否进行了压缩
   对于压缩的 Vxworks 内核映像，在以上第一步复制中，将分为两个阶段完成：第一阶段完成非压缩代码（romInit 函数，romStart 函数）的复制，这部分代码被复制到 RAM_HIGH_ADRS 指向的 RAM 内存处；第二阶段完成压缩代码的解压缩和复制过程，这部分代码被解压缩到 RAM_LOW_ADRS 指向的 RAM 内存处。参见前文图示说明。
3. 是否包含了符号表
   所谓符号表是内核中定义的所有函数与其（虚拟）地址的对应关系表。符号表与 vxWorks 内核映像一般是分离的，一般为调试方便，需要独立的载入符号表，当然也可以将符号表纳入 vxWorks 内核映像中作为一个整体。vxWorks.st_rom.bin 文件类型即将符号表作为了内核映像的一部分。

符号表最大的作用使得在命令行直接敲入函数名即可运行该函数，内核查询符号表获得对
应的地址并转到该地址处执行；另外在调试时，也可以对地址进行函数名标注，从而方便调试，符号表与 vxWorks 内核通常是独立的，所以无论包不包含符号表都不会对内核初始化流程造成影响。当 vxWorks 内核映像被载入 RAM 后，进入到 usrInit 函数执行时，其最终分布图如前文图 3-5 所示。

ROM 型 Vxworks 内核映像类型启动流程总结如下，由于其中只涉及 vxWorks 内核，故采用“文件名：函数名“方式。

romInit.s:romInit→bootInit.c:romStart→完成 vxWorks 内核从 ROM 到 RAM 的复制（和解压缩）→usrConfig.c:usrInit→usrConfig.c:usrRoot→vxWorks 操作系统启动完成

下一节将以压缩版本的 bootrom 为例，详细介绍如何生成压缩的 bootrom，以及其较为详细的执行流程和调试方法。一般在开发阶段，都需要采用 bootrom+Vxworks 的启动方式，因为 bootrom 相对较小，烧录时间较短，基本上 bootrom 如果能够运行到可以将 Vxworks 从外部下载到开发板 RAM 中，就基本完成了 BSP 的移植过程。此外，我们也借助压缩版本 bootrom 的生成方式，介绍二次链接的具体细节，读者也可以借此理解压缩版本 Vxworks_rom 的生成方式。

3.2 深入理解 bootrom

在开发阶段，Vxworks 操作系统启动大多采用 bootrom+ Vxworks 方式，即下载型方式进行。一方面由于 Vxworks 本身调试的需要，另一方面 bootrom 相比 Vxworks 内核较小，可以较快的烧录到平台 ROM 中。在下载型方式中，bootrom 的主要任务就是从主机端（相对运行 Vxworks 的目标板而言）通过串口或者网口将 Vxworks 内核映像载入目标板 RAM 中，而后跳转到 Vxworks 内核映像入口处执行。所有 bootrom 完成的所有的工作基本上都是为了下载做准备。
bootrom 在构成上基本类似于 Vxworks 内核本身，即二者使用同一套函数，但是也有一个较大的区别：bootrom 使用 bootConfig.c 文件，而 Vxworks 内核本身则使用 usrConfig.c 文件。
在下载型启动方式下使用的 Vxworks 内核映像由如下文件构成：
sysALib.s, sysLib.c, usrConfig.c, 设备驱动程序文件。
bootrom 映像则由如下文件构成：
romInit.s, bootInit.c, sysALib.s, sysLib.c, bootConfig.c, 设备驱动程序。
注意：bootrom 映像中虽然包含 sysALib.s 文件代码，但是并非使用其中定义的任何函数。

其中 sysLib.c 以及设备驱动程序都是相同的，下载启动方式下，Vxworks 内核映像不包含 romInit.s 和 bootInit.c 文件。但是一旦处于产品阶段，当采用 ROM 启动方式时，Vxworks 内核映像构成将基本类似于 bootrom 映像构成，如下所示：（注意 ROM 启动方式下，sysALib.s 文件没有使用，但是仍然包含在内核映像中，可以修改系统文件中的相关宏定义，去掉该文件，但如果需要下载型 Vxworks 内核映像，还是要加上 sysALib.s 文件，故建议一直包含该文件）
romInit.s, bootInit.c, sysALib.s, sysLib.c, usrConfig.c, 设备驱动程序文件。

其中 romInit.s，bootInit.c，sysLib.c，设备驱动程序与 bootrom 中使用的都是同一套文件，然而无论 Vxworks 映像是基于下载方式的，还是 ROM 方式的，其总是使用 usrConfig.c 文件，而 bootrom 则总是使用 bootConfig.c 文件。这两个文件虽然定义有相同的函数（usrInit，usrRoot），但基本实现却大不相同，bootConfig.c 也进行一些初始化，如当使用网口下载 Vxworks 内核映像时，其需要进行网口初始化，但是正如上文所述，bootConfig.c 中完成的所有工作都是为了能够从外部主机上下载真正的 Vxworks 操作系统映像，其本身不具有 Vxworks 操作系统功能部件；而 usrConfig.c 则不然，其需要完成维持 Vxworks 操作系统正常运行时所需的所有组件的初始化工作，所以 usrConfig.c 才真正是进行 Vxworks 操作系统

的启动工作。

以下我们以压缩版 bootrom 为例，基于 Powerpc 平台，详细介绍压缩版 bootrom 的生成过程及执行流程，从而使读者对 bootrom 有一个彻底的了解。这对于 Vxworks 内核本身的移植和 BSP 开发都具有重要意义。

bootrom是通过命令行脚本生成的，虽然Tornado开发环境中包含生成bootrom的菜单子命令，但是最终还是通过调用命令行脚本进行bootrom的生成。在执行生成bootrom映像的make命令之前，我们首先需要设置一些环境变量，最直接的方式是从$(WIND_BASE)/host/$(WIND_HOST_TYPE)/bin目录下运行torVars脚本文件。该文件基本实现如下：

```
rem Command line build environments
set WIND_HOST_TYPE=x86-win32
set WIND_BASE=C:\Tornado2.2
set PATH=%WIND_BASE%\host\%WIND_HOST_TYPE%\bin;%PATH%

rem Diab Toolchain additions
set DIABLIB=%WIND_BASE%\host\diab
set PATH=%DIABLIB%\WIN32\bin;%PATH%
```

由此我们可以在target/config/<bspName>（target/ config/wrSbc824x）目录下创建bootrom生成脚本如下：

```
rem bootrom creator file：bootrom.bat
rem Command line build environments
set WIND_HOST_TYPE=x86-win32
set WIND_BASE=C:\T22\ppc
set PATH=C:\T22\ppc\host\x86-win32\bin;C:\WINNT\SYSTEM32;C:\WINNT;

rem Diab Toolchain additions
set DIABLIB=C:\T22\ppc\host\diab
set PATH=C:\T22\ppc\host\diab\WIN32\bin;C:\T22\ppc\host\x86-win32\bin;C:\WINNT\SYSTEM32;C:\WINNT;

make bootrom
pause
```

最后pause命令的加入是为了在执行完毕后，等待用户输入任意键关闭DOS窗口，这样做的目的是为了查看执行结果，否则运行过程将一闪而过，无法得知运行过程及结果。现在我们可以执行该脚本生成bootrom，如下是使用Tornado 2.2 wrSbc824x BSP执行该脚本的结果。

```
ccppc -M -MG -w -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -Wall -I/h
-I. -IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu eeprom.c i8250Sio.c m8240AuxClk.c m8240Epic.c
sysCacheLockLib.c sysFei82557End.c sysLib.c sysNet.c sysPci.c sysPciAutoConfig.c
```

```
sysPnic169End.c sysSerial.c sysVware.c C:\T22\ppc\target\config\all/bootConfig.c
C:\T22\ppc\target\config\all/bootInit.c C:\T22\ppc\target\config\all/dataSegPad.c
C:\T22\ppc\target\config\all/usrConfig.c C:\T22\ppc\target\config\all/version.c
>depend.wrSbc824x

ccppc -E -P -M -w -mcpu=603 -mstrict-align -E -xassembler-with-cpp -I/h -I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu romInit.s >>depend.wrSbc824x

ccppc -E -P -M -w -mcpu=603 -mstrict-align -E -xassembler-with-cpp -I/h -I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu sysALib.s >>depend.wrSbc824x
```

以上三个语句进行预处理，生成源文件所依赖头文件列表，将这些头文件列表写入depend.wrSbc824x文件中，注意这个文件的扩展名，以BSP的目录名为后缀，这是一个约定。

```
ccppc -c -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -Wall -I/h -I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu C:\T22\ppc\target\config\all\bootInit.c
```

bootInit.c 文件包含 romStart()函数，是第一个被执行的 C 函数，其完成将代码和数据从 ROM 拷贝到 RAM 中，如果代码存在压缩，其在拷贝过程中一并完成代码的解压缩工作，在完成拷贝后，其跳转到已拷贝到 RAM 中的 usrInit 函数进行执行。

```
ccppc -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -I/h -I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu -P -xassemblerwith-cpp -c -o romInit.o romInit.s
```

romInit.s 文件包含了 bootrom 入口函数 romInit()，该函数在系统上电时第一个被执行的函数，其编码上必须注意在函数起始处必须放置一个中断向量表或复位向量，因为系统上电起始阶段，CPU 都会收到一个复位中断，跳转到复位中断向量表处执行。romInit 完成硬件相关寄存器的初始化，注意某些硬件寄存器只能在上电复位后被配置一次，这些寄存器的配置就在 romInit()函数中完成。romInit()函数执行完毕后，将跳转到 romStart()函数执行。

```
ccppc -c -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -Wall -I/h-I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu C:\T22\ppc\target\config\all\bootConfig.c
```

bootConfig.c 文件包含 usrInit()函数，完成平台的进一步初始化（主要是外围设备初始化），Vxworks 内核的下载等工作。不建议直接修改 target/config/All 目录下的 bootConfig.c 文件，

因为 All 目录下文件将被所有的 BSP 共享，所以对于一个特定的 BSP，如果需要修改 bootConfig.c 文件，建议用户从 All 目录下拷贝一份 bootConfig.c 文件的副本到 BSP 目录下(假设重命名为 bootConfig_copy.c)，并在 Makefile 中定义如下的宏定义：
BOOTCONFIG=./bootConfig_copy.c
则在编译时将使用 BSP 目录下的 bootConfig_copy.c 文件，此时我们按照需要自动对 bootConfig_copy.c 文件进行修改，而不影响其他 BSP 对 All 目录下系统 bootConfig.c 文件的依赖。

```
ccppc -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -I/h -I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu -P -xassemblerwith-cpp -c -o sysALib.o sysALib.s
```

sysALib.s 文件虽然也包含在 bootrom 映像中，但是并非使用该文件中定义的任何函数。当然由于这是一个汇编文件，如果需要在 romInit.s 文件之外编写一段汇编代码实现某种特殊目的，可以加入到 sysALib.s 文件中。但是要保证 sysInit 函数必须是 sysALib.s 文件中定义的第一个函数，下载方式 Vxworks 内核启动过程依赖这一点。一般而言，对于 bootrom 和 ROM 型 Vxworks 内核映像而言，都不需要使用 sysALib.s 文件中的代码。这个文件只被下载型 Vxworks 内核映像使用，该文件中定义的 sysInit 函数是下载型 Vxworks 内核映像执行的入口函数。

```
ccppc -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -Wall -I/h -I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu-c sysLib.c
```

sysLib.c 文件中定义了关键结构数组，完成内存映射过程。一般驱动源码也被直接包含在该文件中。该文件是 BSP 中必须文件，其中定义了一些初始化过程中调用的关键函数；包括文件名本身也是事先约定的，必须命名为 sysLib.c，不可随意更改。

```
ccppc -c -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -Wall -I/h-I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu -o version.o C:\T22\ppc\target\config\all/version.c
```

version.c 文件是一个实现上较为简单的文件，用以生成映像的版本信息，映像创建时间和日期。

```
ldppc -o tmp.o -X -N -e usrInit -Ttext 01F00000 bootConfig.o version.o sysALib.o
sysLib.o --start-group -LC:\T22\ppc\target/lib/ppc/PPC603/gnu
-LC:\T22\ppc\target/lib/ppc/PPC603/common -lcplus -lepcommon -lepdes -lgnucplus
-lsnmp -lvxcom -lvxdcom -larch -lcommoncc -ldcc -ldrv -lgcc -lnet -los -lrpc
-ltffs-lusb -lvxfusion -lvxmp -lwdb -lwind -lwindview
C:\T22\ppc\target/lib/libPPC603gnuvx.a --end-group
```

```
-TC:\T22\ppc\target/h/tool/gnu/ldscripts/link.RAM
```

创建 bootrom 中压缩部分可执行代码，这部分代码在 romStart()函数将被解压缩到 RAM_HIGH_ADRS 指定的内存地址处。这是压缩版 bootrom 映像类型中的压缩部分代码，这部分代码进行独立链接，我们称之为 bootrom 创建过程中的第一次链接。注意这次链接指定的链接地址‘-Ttext 01F00000’，其中 01F00000’就是 RAM_HIGH_ADRS 常量的值。这一点非常重要，压缩版 bootrom 映像中的压缩部分被解压缩到 RAM_HIGH_ADRS 地址处，在完成 romStart()函数执行后，将直接跳转到 usrInit()函数进行执行，而 usrInit()函数已被解压缩到 RAM_HIGH_ARDS 地址处，所以 usrInit()函数的链接地址也必须是 RAM_HIGH_ADRS。注意 sysALib.o 也被包含进 bootrom 映像中，虽然实际上 bootrom 并不需要 sysALib.s 中定义的任何函数。
从以上语句可以看出，这次链接并不包括 romInit.s，bootInit.c 两个文件，因为这两个文件的代码作为压缩 bootrom 映像中的唯一非压缩代码存在，所有压缩代码并不能直接执行，故硬件的原始初始化代码（即 romInit.s）以及解压缩代码本身(即 bootInit.c)都必须是非压缩状态的。
由于只有非压缩代码和压缩代码整合成单一映像文件时，还需要进行一次链接（即二次链接），且链接地址与本次并不相同，故必须对本次链接后的文件（tmp.o）进行处理，避免二次链接时对已链接的代码造成修改。我们首先调用 objcopyppc 将连接后 ELF 格式文件转换成纯二进制可执行文件。

```
C:\T22\ppc\host\x86-win32\bin\objcopyppc -O binary --binary-without-bss tmp.o tmp.out
```

而后对这个纯二进制可执行文件完成压缩操作：

```
C:\T22\ppc\host\x86-win32\bin\deflate < tmp.out > tmp.Z
Deflation: 60.52%
```

由于需要在二次链接中包含这个被压缩文件，所以必须以一种特殊的方式将这些二进制代码嵌入最后 bootrom 映像中，且不对其中的内容（已链接的可执行二进制纯代码）造成任何影响。

```
C:\T22\ppc\host\x86-win32\bin\binToAsm tmp.Z >bootrom.Z.s
```

binToAsm 将压缩后文件转换成一个汇编文件，压缩块作为汇编文件中一个数据块而存在，这样避免了二次链接过程中对压缩块的链接操作。在这个汇编文件中，专门定义了两个变量表示这个压缩块的开始和结尾，便于解压缩时对压缩块进行定位。其中 binArrayStart 表示压缩块的起始地址，binArrayEnd 则表示压缩块的结束地址。这两个变量包含的压缩数据最后将被 romStart()函数解压缩到 RAM_HIGH_ADRS 指定的内存地址处。
由于是一个汇编源文件，当然，首先我们需要将其编译成目标文件，代码如下。

```
ccppc -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -I/h -I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu -P -xassemblerwith-cpp -c -o bootrom.Z.o bootrom.Z.s
```

```
ccppc -c -mcpu=603 -mstrict-align -ansi -O2 -fvolatile -fno-builtin -Wall -I/h-I.
-IC:\T22\ppc\target\config\all -IC:\T22\ppc\target/h
-IC:\T22\ppc\target/src/config -IC:\T22\ppc\target/src/drv -DCPU=PPC603
-DTOOL_FAMILY=gnu -DTOOL=gnu -o version.o
C:\T22\ppc\target\config\all/version.c
```

再次编译 version.c 文件，这次是针对 bootrom 本身，而上一次则是用于压缩块中的代码。

```
ldppc -X -N -e _romInit -Ttext 00100000 -o bootrom romInit.o bootInit.o version.o
bootrom.Z.o --start-group -LC:\T22\ppc\target/lib/ppc/PPC603/gnu
-LC:\T22\ppc\target/lib/ppc/PPC603/common -lcplus -lepcommon -lepdes -lgnucplus
-lsnmp -lvxcom -lvxdcom -larch -lcommoncc -ldcc -ldrv -lgcc -lnet -los -lrpc
-ltffs -lusb -lvxfusion -lvxmp -lwdb -lwind -lwindview
C:\T22\ppc\target/lib/libPPC603gnuvx.a --end-group
-TC:\T22\ppc\target/h/tool/gnu/ldscripts/link.RAM
```

完成二次链接过程，将非压缩部分和压缩部分最终整合成压缩版本的 bootrom 映像文件。指定的入口函数为 romInit，链接地址指定为 RAM_LOW_ADRS=0x00100000。那么为何不是 ROM_TEXT_ADRS？这一点下文在做解释。

最后一步是检查 bootrom 的大小，查看平台 ROM 或 FLASH 是否能够放得下，平台 ROM 或 FLASH 大小通过 ROM_SIZE 定义，这个常量如同 RAM_HIGH_ADRS，RAM_LOW_ADRS 等常量一样，同时定义在 Makefile 和 config.h 文件中，且这两个文件中定义的值必须一致！

```
C:\T22\ppc\host\x86-win32\bin\romsize ppc -b 00080000 bootrom
bootrom: 16784(t) + 206048(d) = 222832 (301456 unused)
```

至此，我们完成压缩版本 bootrom 的生成。对于非压缩版本的 bootrom，其生成过程相对比较简单，此时不需要第一次的链接，只需要在最后一次链接中包含所有的目标文件即可。结合以上的说明，这一点应该不难理解。

bootrom 中较为关键的一个函数就是代码从 ROM 向 RAM 重定位的过程，这个过程集中实现在 romStart()函数中，对于压缩版本的 bootrom 映像，该函数将分两个阶段进行代码的复制过程：第一阶段完成非压缩代码从 ROM 到 RAM 的拷贝；第二阶段完成压缩代码从 ROM 到 RAM 的拷贝，并同时完成解压缩操作。非压缩代码和压缩代码在 RAM 中的目的地址将不同，这与各自链接时指定的链接地址有关。romStart()函数实现包含大量的宏定义，在阅读该函数时很不方便。可以使用一个有效的调试技巧，即通过编译器的预处理功能，去掉宏定义，这样可以让代码简洁的多。用户可以通过在命令行使用如下命令达到以上目的：
make ADDED_CFLAGS=-E file.o >file.i
这将取出源代码中所有的空行和以‘#’打头的语句，同时宏定义将被解析，此时可以直接的看到哪些代码被使用，避免源码中大量条件宏对阅读分析代码造成的严重不便。如下使用在命令行使用命令‘make ADDED_CFLAGS=-E bootInit.o >bootInit.i’得到的 romStart()函数实现。
注意：某些情况下如上命令不可用，此时可以使用如下命令对源文件直接进行预处理。

C:\Tornado2.2\target\config\all>ccarm -E –I\h -I..\<bspName> -I. -Ic:\Tornado2.2\target\config\all -Ic:\Tornado2.2\target/h -Ic:\Tornado2.2\target/src/config -Ic:\Tornado2.2\target/src/drv -DCPU_926E bootInit.c >bootInit.i

```
void romStart
(
register int startType
)
{
    Volatile FUNCPTR absEntry;
    ((FUNCPTR)(((UINT) copyLongs - (UINT)romInit) + 0xFFF00100 ) ) (0xFFF00100,
        (UINT)romInit,((UINT)binArrayStart - (UINT)romInit)/ sizeof (long));
    ((FUNCPTR)(((UINT) copyLongs - (UINT)romInit) + 0xFFF00100 ) )
    ((UINT*)((UINT)0xFFF00100 + ((UINT)(((int)( binArrayEnd ) & ~( sizeof(long) - 1)) )
    -(UINT)romInit)), (UINT *)(((int)( binArrayEnd ) & ~( sizeof(long) - 1)) ) ,
        ((UINT)wrs_kernel_data_end - (UINT)binArrayEnd) / sizeof (long));
    if (startType & 0x02 ) //检查是否是上电启动（即冷启动），如是，则对内存相关区域清零。
    {
        fillLongs ((UINT *)((0x00000000 + 0x4400 ) ),((UINT)romInit - 0x1000 -
            (UINT)(0x00000000 + 0x4400 ) ) / sizeof(long), 0);
        fillLongs ((UINT *)wrs_kernel_data_end,((UINT)(0x00000000 + 0x04000000 -
            0x02000000 ) - (UINT)wrs_kernel_data_end) / sizeof (long), 0);
        *(((char *) (0x00000000 + 0x4200 )) ) = '\0' ;
    }
    {
        if (inflate ((UCHAR *)(((UINT) binArrayStart - (UINT)romInit) + 0xFFF00100) ,
                (UCHAR *)0x01F00000 , binArrayEnd - binArrayStart) != 0 )
            return;
        absEntry = (FUNCPTR)0x01F00000 ;
    }
    (absEntry) (startType);
}
```

代码行：

```
((FUNCPTR)(((UINT) copyLongs - (UINT)romInit) + 0xFFF00100 ) ) (0xFFF00100,
(UINT)romInit,((UINT)binArrayStart - (UINT)romInit)/ sizeof (long));
```

完成第一阶段非压缩代码从 ROM 向 RAM 的拷贝过程。注意由于代码仍然执行在 ROM 中，故对 copyLongs 函数的调用必须使用相对寻址，如果直接使用 copyLongs，由于 copyLongs 函数链接地址是在 RAM 空间，而目前代码尚未从 ROM 拷贝到 RAM，故将造成非法指令异常，系统直接崩溃。此次 copyLongs 调用将 romInit 以及 romStart 函数代码从 ROM 直接拷贝到 RAM 中，由于这些代码是非压缩的，所有直接拷贝即可。参数 1（0xFFF00100）的值是 ROM_TEXT_ADRS 常量指定的值，也是 bootrom 烧录入的 ROM 或 FLASH 在全局地址空间的地址。如果 ROM_TEXT_ADRS 等于 ROM_BASE_ADRS，那么就是 ROM 或 FLASH 的起始地址，系统上电后，将首先从这里开始执行代码。参数直接使用

romInit 函数地址作为目的地址，在链接时 romInit 函数被链接到 RAM_LOW_ADRS 地址处，也就是说 romInit()和 romStart()函数被拷贝到了 RAM_LOW_ADRS 指定的内存处，这也是为何对于 romInit 实现中以及当前对于 copyLongs 函数的调用必须做到地址无关，或者只能使用相对地址进行调用，因为一旦直接进行调用，那么 CPU 的指令寄存器将使用链接时的地址作为地址去读取指令，即从 RAM_LOW_ADRS 指定的内存区域读取指令，而在 romInit 函数执行时以及当前 copyLongs 函数调用之时，ROM 中代码尚未拷贝到 RAM 中，所以必须避免在这之前进行函数的直接调用，而要使用相对调用，即通常所说的 PIC（Positon Independent Code）：位置无关代码。

语句行：

```
((FUNCPTR)(((UINT) copyLongs - (UINT)romInit) + 0xFFF00100 ) )
((UINT*)((UINT)0xFFF00100 + ((UINT)(((int)( binArrayEnd ) & ~( sizeof(long) - 1)) )
-(UINT)romInit)), (UINT *)(((int)( binArrayEnd ) & ~( sizeof(long) - 1)) ) ,
    ((UINT)wrs_kernel_data_end - (UINT)binArrayEnd) / sizeof (long));
```

完成数据段由 ROM 到 RAM 的拷贝。注意 binArrayEnd 指向压缩块的结束为止，即实际上是代码段的尾部，其后是数据段，而 wrs_kernel_data_end 变量则表示数据段的尾部，其后是 BSS 段，故以上代码即完成数据段的复制。

语句行：

```
if (inflate ((UCHAR *)(((UINT) binArrayStart - (UINT)romInit) + 0xFFF00100) ,
        (UCHAR *)0x01F00000 , binArrayEnd - binArrayStart) != 0 )
    return;
```

完成压缩块的解压缩。

参数 1：(UINT) binArrayStart - (UINT)romInit) + 0xFFF00100

计算压缩块在 ROM 中的地址，注意 binArrayStart 在二次链接中指向压缩块的起始地址，而 binArrayEnd 则指向压缩块的结束地址。0xFFF00100 为 ROM_TEXT_ADRS 的值，即存放 bootrom 的起始 ROM 地址。

参数 2：(UCHAR *)0x01F00000

这个常量实际上就是 RAM_HIGH_ADRS，在经过预处理后，被直接置换成 RAM_HIGH_ADRS 表示的值。

参数 3：binArrayEnd - binArrayStart

表示压缩块的大小，inflate 函数调用时需要指定被解压缩块的大小。

实际上以上 inflate 函数的操作就是将压缩块从 ROM 中解压缩到 RAM_HIGH_ADRS 指定的 RAM 地址处，我们可以参见前文中压缩块的生成过程，则可以看到压缩块入口函数为 usrInit，即 romStart 函数完成对压缩 bootrom 的两次拷贝后，就会跳转到 usrInit 函数执行。

另外注意到此处对于 inflate 的调用已经是函数直接调用方式，而非位置无关调用方式，因为在 romStart 函数中对两个 copyLongs 的调用已经将 inflate 代码从 ROM 处拷贝到 RAM_LOW_ADRS 指定的 RAM 地址处，且数据段也已经拷贝完毕，故可以直接使用通常的函数调用方式，这个 inflate 函数实际上是执行的在 RAM 区域的代码，已经完全脱离 ROM

中代码了！

语句行：

```
absEntry = (FUNCPTR)0x01F00000 ;
(absEntry) (startType);
```

完成跳转，进入 usrInit 函数进行执行。

romStart 函数中完成的二次拷贝过程可以参见前一节中图 1 所示。

对于 bootrom 而言，其使用 bootConfig.c 文件，故 romStart 最后跳转到 bootConfig.c 文件中定义的 usrInit 函数。该函数在 bootrom 生成过程中的第一次链接中被链接到 RAM_HIGH_ADRS 指定的地址处（此处即为 0x01F00000），进入 usrInit 函数后，接下来所有执行的代码都是在 RAM 中进行的了。现在我们看一下 bootConfig.c 文件中定义的 usrInit，usrRoot，bootCmdLoop 三个函数具体实现的功能。

我们对 bootConfig.o 依然使用‘make ADDED_CFLAGS=-E bootConfig.o >bootConfig.i’命令。

```
void usrInit
(
int startType
)
{
    while (trapValue1 != 0x12348765 || trapValue2 != 0x5a5ac3c3 )
    {
        ;
    }
    cacheLibInit (0x02 , 0x02 );
    bzero (edata, end - edata);
    sysStartType = startType;
    intVecBaseSet ((FUNCPTR *) ((char *) 0x0) );
    excVecInit ();
    sysHwInit ();
    usrKernelInit ();
    cacheEnable (INSTRUCTION_CACHE);
    kernelInit ((FUNCPTR) usrRoot, (24000) ,(char *) (end) ,sysMemTop (), (5000) , 0x0 );
}
```

语句行：

```
    while (trapValue1 != 0x12348765 || trapValue2 != 0x5a5ac3c3 )
    {
        ;
    }
```

用以进行检查代码段拷贝是否成功完成，成功完成的含义如是否对齐到合适的内存位置，以及在拷贝过程中数据是否完好等。

trapValue1，trapValue2 是定义在 bootConfig.c 文件中的两个 volatile 型变量，如下所示。

```
#define TRAP_VALUE_1    0x12348765
#define TRAP_VALUE_2    0x5a5ac3c3
LOCAL volatile UINT32    trapValue1    = TRAP_VALUE_1;
LOCAL volatile UINT32    trapValue2    = TRAP_VALUE_2;
```

语句行：

```
cacheLibInit (0x02 , 0x02 );
```

初始化 cache 内核库，且根据参数值配置 CPU 相关寄存器，开启或者关闭系统 cache。由于 bootrom 并不使用 CPU MMU 单元，故 cache 与否只能通过系统寄存器进行控制，这是全局范围内控制 cache 的方式，Vxworks 内核映像中使用 MMU 单元，此时可以控制单个页面 cache 与否。如果在 bootrom 调试阶段配置外设时出现一些问题，建议关闭系统 cache，可以通过在 config.h 文件定义如下语句完成系统 cache 的关闭。

#undef INCLUDE_CACHE_SUPPORT

注意：外设寄存器操作必须设置为 non-cachable，否则将可能一些很难调试的硬件问题，所以如果使用 cache，对于表示外设寄存器的变量都必须使用 volatile 修饰符进行修饰，或者如上文建议的，在 bootrom 中直接关闭系统 cache，当然这会对 RAM 的使用造成一定程度的性能影响，但对于 bootrom 而言，这一点可以不用过分计较。

语句行：

```
bzero (edata, end - edata);
```

对 BSS 段清零。从数据段尾部到 bootrom 映像尾部都将被清零。

语句行：

```
sysStartType = startType;
intVecBaseSet ((FUNCPTR *) ((char *) 0x0) );
excVecInit ();
```

完成异常表（系统中断向量表）的建立。注意 `intVecBaseSet` 函数内核实现为空，不依赖于输入参数，异常表的位置将根据特定平台上处理器的要求进行，如 ARM 处理器一般将系统中断向量表建立在绝对地址 0 处。此处我们还将启动类型赋值给一个全局变量 sysStartType，便于 bootConfig.c 中定义的其他函数直接使用，而不用一直以参数形式进行传递。这三条语句的执行基本都会成功，不会有什么问题，除非在创建系统中断向量表的内存区域不可写入。

语句行：

```
sysHwInit ();
```

完成平台所有外设的配置，将所有外设配置到有效状态，但是都暂时进入“安静“或”预知“状态。这表示外设已经被配置到可以随时准备工作了，现在就剩下启动有关工作”使能“位了。当然对于有些需要使用中断方式工作的外设而言，此时还没有完成中断服务程序的注册，这个注册将在 sysHwInit2 函数中完成。sysHwInit 函数定义在 sysLib.c 文件中。

注意：对于外设中断服务程序注册的时机，必须将其放在 sysHwInit2 函数被调用时，这一点非常重要，因为在 intLibInit 函数尚未调用之前，bootrom 映像尚未创建外设中断表，故如果在此之前进行注册，那么将无从对这些注册的中断服务程序句柄进行保存。而对于

intLibInit 函数的调用一般将其放置在 sysHwInit2 函数开始处。

语句行：

```
usrKernelInit ();
```

完成bootrom内核的一些初始化工作，bootrom虽然主要功能是从外部下载一个Vxworks内核映像，但其本身也是一个小的内核，其支持任务创建，任务调度等等一些列Vxworks内核功能，只是其并不作为一般应用开发的平台。`usrKernelInit`函数完成对内核一些关键数据结构的初始化，如涉及任务的三个队列readyQHead，activeQHead，tickQHead等。`usrKernelInit`函数定义在$(WIND_BASE)/target/src/config/usrKernel.c文件中，用户可以直接查看其源代码。

语句行：

```
cacheEnable (INSTRUCTION_CACHE);
```

开启系统指令cache。如果调试过程中出现一些诡异的问题，建议首先关闭cache。

语句行：

```
kernelInit ((FUNCPTR) usrRoot, (24000) ,(char *) (end) ,sysMemTop (), (5000) , 0x0 );
```

创建内核任务，这是系统第一个任务，usrRoot 作为入口函数。

注意：在 romInit 函数中，我们从系统角度，通过配置 CPU 的相关控制寄存器关闭了系统中断，而当任务创建时，系统中断将默认被开启，这一点非常重要。有时我们会怀疑既然在 romInit 函数中从全局角度禁止了中断，为何在后续代码中都找不到中断开启的代码，然而中断却能正常响应。实际上这个系统中断重新开启的工作由此处任务创建过程中完成，这是作为任务本身创建（其中包括对硬件寄存器的初始化）的一部分完成的。故在后续代码中，我们只需要开启中断控制器相关中断屏蔽位即可。有关中断方面的详细说明，请参考本书中相关章节对中断的分析。

kernelInit 函数调用可能会出现问题，因为其在任务创建过程中已经自动开启的系统中断，故一旦进入到 usrRoot 函数运行，CPU 就可对中断进行响应，但是现在 sysHwInit2 函数尚未调用，换句话说，现在所有中断服务程序都没有进行注册，所有如果某个外设由于在 sysHwInit 函数中配置不合理，没有进入到应有的“安静“状态，从而不适合的产生了一个中断，那么将直接导致系统的死机，因为 CPU 跳转到对应中断句柄执行时，将遇到‘undefined instruction’系统异常。

注意：在调试过程中，如果系统运行后就死机，那么首先怀疑的应该就是不合适的中断产生，而对应中断服务程序却尚未注册。笔者曾经调试一个网卡驱动，一旦开始启动网卡驱动（即调用 endStart 函数后），系统就立刻死机。检查了所有寄存器配置，都没有发现问题，最后问题出在搞错了网卡设备的中断号，即将网卡中断服务程序注册到另一个设备的中断号上，结果是网卡中断号没有对应的中断服务程序，所以网卡一产生中断，系统就死机。故“不适合的中断产生”并非不应该产生中断，而是首先确定系统当前是否允许该中断产生，其次该中断是否已经注册有中断服务程序。特别注意不要搞错中断号！

`kernelInit` 函数将不再返回，在任务创建后，usrRoot 任务将立刻得到运行，因为当前系统内只有这一个任务。注意之前的代码都没有任务上下文，读者可以将之前代码的运行想象为

运行在中断上下文中。当进入 usrRoot 函数执行时，此时所有的代码将在一个任务上下文中运行，故可以调用 malloc 分配内存等等。usrRoot 函数用以初始化 bootrom 小系统的其他内核组件。

```
void usrRoot
(
    char * pMemPoolStart,
    unsigned memPoolSize
)
{
    char tyName [20];
    int ix;
    int count;
    END_TBL_ENTRY* pDevTbl;
    memInit (pMemPoolStart, memPoolSize);
    sysClkConnect ((FUNCPTR) usrClock, 0);
    sysClkRateSet (60 );
    sysClkEnable ();
    selectInit (50 );
    iosInit (20 , 50 , "/null");
    consoleFd = (-1) ;
    if (1 > 0)
    {
        ttyDrv();
        for (ix = 0; ix < 1 ; ix++)
        {
            sprintf (tyName, "%s%d", "/tyCo/", ix);
            (void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);
            if (ix == 0 )
            {
                strcpy (consoleName, tyName);
                consoleFd = open (consoleName, 2 , 0);
                (void) ioctl (consoleFd, 4 , 9600 );
                (void) ioctl (consoleFd, 3 ,0x01 | 0x02 | 0x04 | 0x08 );
            }
        }
    }
    ioGlobalStdSet (0 , consoleFd);
    ioGlobalStdSet (1 , consoleFd);
    ioGlobalStdSet (2 , consoleFd);
    pipeDrv ();
    excShowInit ();
    excInit ();
    excHookAdd ((FUNCPTR) bootExcHandler);
```

```
    logInit (consoleFd, 5);
    bootElfInit ();
    muxMaxBinds = 8 ;
    if (muxLibInit() == (-1) )
        return;
    for (count = 0, pDevTbl = endDevTbl; pDevTbl->endLoadFunc != ((void *)0) ;
            pDevTbl++, count++)
    {
        cookieTbl[count].pCookie = muxDevLoad (pDevTbl->unit, pDevTbl->endLoadFunc,
        pDevTbl->endLoadString,pDevTbl->endLoan, pDevTbl->pBSP);
        if (cookieTbl[count].pCookie == ((void *)0) )
        {
            printf ("muxLoad failed!\n");
        }
        cookieTbl[count].unitNo=pDevTbl->unit;
        bzero((void *)cookieTbl[count].devName,8 );
        pDevTbl->endLoadFunc((char*)cookieTbl[count].devName, ((void *)0) );
    }

    taskSpawn ("tBoot", bootCmdTaskPriority, bootCmdTaskOptions,
    bootCmdTaskStackSize, (FUNCPTR) bootCmdLoop,0,0,0,0,0,0,0,0,0,0);
}
```

语句行：

```
    memInit (pMemPoolStart, memPoolSize);
```

初始化系统内存堆，从而 malloc/free 函数可用。此函数调用完成后，接下来的代码就可用 malloc 分配内存空间了。

语句行：

```
    sysClkConnect ((FUNCPTR) usrClock, 0);
    sysClkRateSet (60 ); //设置系统时钟tick间隔，即1s将产生60次系统时钟中断
    sysClkEnable ();
```

完成 bootrom 小系统内核外设中断表的创建，注册系统时钟中断，配置相关外设注册其中断服务程序。注意 sysHwInit2 函数在 sysClkConnect 函数中被调用。关于这三个函数的详细说明，请读者参考本书中相关章节对中断的说明。

以上三个函数完成执行后，系统将正式具备“脉搏”，系统时钟将以固定的时间间隔产生中断，此时 taskDelay，wdStart 函数都将变得有效。

语句行：

```
    selectInit (50 );
    iosInit (20 , 50 , "/null");
    consoleFd = (-1) ;
    if (1 > 0)
    {
```

```
        ttyDrv();
        for (ix = 0; ix < 1 ; ix++)
        {
            sprintf (tyName, "%s%d", "/tyCo/", ix);
            (void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);
            if (ix == 0 )
            {
                strcpy (consoleName, tyName);
                consoleFd = open (consoleName, 2 , 0);
                (void) ioctl (consoleFd, 4 , 9600 );
                (void) ioctl (consoleFd, 3 ,0x01 | 0x02 | 0x04 | 0x08 );
            }
        }
    }
    ioGlobalStdSet (0 , consoleFd);
    ioGlobalStdSet (1 , consoleFd);
    ioGlobalStdSet (2 , consoleFd);
```

进行串行终端的初始化，并设置 0，1，2 三个默认句柄指向串口，此后程序中所有的打印将通过串口输出。串口设备属于字符设备的一种，但是 Vxworks 串口驱动编写又与普通字符设备存在一些差异，这是由于 Vxworks 内核本身为串口提供了一个额外的 TTY 中间层，如此可以简化串口驱动的设计并提供串口字符收发效率，有关串口和字符驱动，请参考本书后面章节内容。

在完成以上语句的执行后，接下来的代码就可以调用 printf 语句显示信息了。当然 logMsg 函数尚不可进行调用，因为尚未初始化 log 库，这将在接下来的代码中完成。

如果在 Shell 下敲入‘devs’命令，将显示当前的设备列表，串口设备名显示为“/tyCo/0”，如果存在多个串口，最后的数字依次增加，如”/tyCo/1”表示第二个串口设备。

语句行：

```
    pipeDrv ();
    excShowInit ();
    excInit ();
    excHookAdd ((FUNCPTR) bootExcHandler);
```

完成管道设备内核初始化（pipeDrv），异常信息（如异常发生时各寄存器值）显示库初始化（excShowInit），异常处理任务 tExcTask 创建（excInit）以及异常发生时钩子函数注册（excHookAdd）。

语句行：

```
    logInit (consoleFd, 5);
    bootElfInit ();
```

完成对 tLogTask 任务（logInit）的创建，此后就可以调用 logMsg 进行信息显示。logMsg 可以在任何环境下被调用，包括中断上下文。通过 logMsg 打印的信息将暂时被存储到一个内核队列中，由 tLogTask 任务负责将这个队列中信息在任务上下文中发送出去。在外设驱动调试时，如果使用 logMsg 打印，如果外设驱动存在问题造成系统死机，那么要显示的信息可能无法打印，此时可以使用 printf 进行替代。logMsg 大多使用在系统一切正常时一般信息

的显示，调试时建议直接使用 printf 函数。bootElfInit 完成内核中 ELF 模块的初始化，将系统模块的默认读取器设置为 ELF 格式，如下载型 Vxworks 内核映像就是 ELF 格式的。

语句行：

```
    muxMaxBinds = 8 ;
    if (muxLibInit() == (-1) )
        return;
    for (count = 0, pDevTbl = endDevTbl; pDevTbl->endLoadFunc != ((void *)0) ;
            pDevTbl++, count++)
    {
        cookieTbl[count].pCookie = muxDevLoad (pDevTbl->unit, pDevTbl->endLoadFunc,
        pDevTbl->endLoadString,pDevTbl->endLoan, pDevTbl->pBSP);
        if (cookieTbl[count].pCookie == ((void *)0) )
        {
            printf ("muxLoad failed!\n");
        }
        cookieTbl[count].unitNo=pDevTbl->unit;
        bzero((void *)cookieTbl[count].devName,8 );
        pDevTbl->endLoadFunc((char*)cookieTbl[count].devName, ((void *)0) );
}
```

完成网络设备驱动的初始化。endDevTbl 数组定义在 configNet.h 文件中，定义了当前平台具有的所有可用网络接口设备以及对应的初始化入口函数（如 armEndLoad）。以上代码遍历 endDevTbl 数组各个元素，对其调用 muxDevLoad 函数，该函数将进一步调用用户网口驱动中实现的初始化函数(如 armEndLoad).

注意：

1. 对于网口驱动初始化函数存在两次调用：第一次传入初始化函数的第一个参数被设置为 NULL，这表示让初始化函数仅仅返回网口设备名称，不要对网络设备进行硬件初始化；第二次才是正规的调用，此时初始化函数需要对网络设备硬件寄存器进行配置，使网络设备进入到准备工作状态。关于网络设备驱动的细节，请参见本书后续相关章节内容。
2. 以上代码的执行是有条件的，如果 Vxworks 内核映像的下载是通过串口进行的（虽然较慢，但是没有办法），那么就不需要在 bootrom 中对网口进行配置，以上语句可以全部删除。然而对于外设驱动的调试，一般都是通过调试 bootrom 来进行的，而非直接调试 Vxworks。因为二者使用同一套外设驱动程序，只是在内核组件初始化代码上存在差别。当然对于外设驱动调试，一般通过是先让 bootrom 运行到 shell 下，而后在 shell 下调用驱动的相关函数进行调试，而不是作为内核启动的一部分进行，这样更有效率些。

如果成功完成网络设备驱动的初始化，且驱动可用，那么下面就要开始下载 Vxworks 内核了，当然在这之前还需要一些列准备工作，如启动网络设备工作（注意以上仅仅是初始化，还没有启动进入到工作，启动过程由 muxEndStart 函数完成，其调用驱动中启动函数（如 armEndStart）完成中断注册，开启工作使能位等），解析 bootline 参数，获取主机地址，Vxworks 内核映像名，主机服务器用户名和密码等等，这些工作都将由 bootCmdLoop 函数完成。

语句行：

```
taskSpawn ("tBoot", bootCmdTaskPriority, bootCmdTaskOptions, bootCmdTaskStackSize,
(FUNCPTR) bootCmdLoop,0,0,0,0,0,0,0,0,0,0);
```

创建“tBoot”任务，执行 bootCmdLoop 函数，由其具体完成 Vxworks 内核映像的下载工作。bootCmdLoop 函数较长，我们将不再跟随该函数的执行过程，读者可以自行对该文件进行分析。bootCmdLoop 函数定义在 bootConfig.c 文件中。

有时需要在 bootrom 中加入自己的一些代码，如 romInit 函数调用的平台初始化代码（如对内存控制的初始化，平台 PLL 初始化，管脚复用寄存器初始化等等），此时需要注意由于这些代码需要在 romInit 函数中调用，故不可以是压缩的，因为在 romInit 执行时，还没有解压缩操作，解压缩知道 romStart 函数执行后才完成。那么如何做到让我们自定义的代码以非压缩形式进入 bootrom 映像中呢？用户需要借助 BOOT_EXTRA 宏，对于需要以非压缩形式加入到 bootrom 映像中的用户代码，用户需要在 Makefile 中以如下形式操作 BOOT_EXTRA 宏定义。

BOOT_EXTRA=myOwnCode.o myOwnAnotherCode.o

如此定义后，myOwnCode.o，myOwnAnotherCode.o 中包含的代码将作为非压缩部分进入 bootrom 中，romStart 函数将在第一次拷贝中将这些文件中代码和 romInit 函数，romStart 函数一道拷贝到 RAM_LOW_ADRS 指定的内存地址处。

如果需要以压缩形式进入 bootrom，那么就需要使用到另一个宏 MACH_EXTRA，如下所示：

MACH_EXTRA=myOwnCode.o myOwnAnotherCode.o

如此定义后，myOwnCode.o，myOwnAnotherCode.o 中包含的代码将作为压缩部分进入 bootrom 中，这些代码将在 romStart 函数的第二次拷贝中被解压缩到 RAM_HIGH_ADRS 指定的内存地址处。

注意：

1. BOOT_EXTRA 和 MACH_EXTRA 宏对于压缩型 ROM 启动方式 Vxworks 内核映像同样有效。
2. 在 BOOT_EXTRA 和 MACH_EXTRA 中可以指定相同的文件，此时该文件中代码即以压缩形式也以非压缩形式存在于映像中。由于在载入 Vxworks 内核映像后，RAM_LOW_ADRS 区域的 bootrom 代码将被覆盖，如果在跳转到 Vxworks 内核映像执行之前，还需要执行位于 RAM_LOW_ADRS 区域的 bootrom 代码，那么这些代码就需要通过以非压缩方式存在，这样 RAM_HIGH_ADRS 区域的代码就可以以相对方式进行调用，一个典型的文件就是 version.o，其同时存在压缩和非压缩部分。虽然其并非以 BOOT_EXTRA 和 MACH_EXTRA 宏的方式指定的。

某些读者可能对同时以压缩和非压缩形式保存两份相同的代码是否会造成地址上的冲突有疑问，实际上并不会出现读者所担心的问题。因为压缩部分代码只供压缩部分其他代码调用，而非压缩部分代码一般只供 romInit 函数调用，且压缩和非压缩代码分别在不同的链接过程中进行包含的，属于完全不同地址空间（一个位于 RAM_LOW_ADRS，一个位于 RAM_HIGH_ADRS），故不会造成冲突，因为二者都有代码的不同副本。

常量定义说明

LOCAL_MEM_LOCAL_ADRS

平台外部存储器（如 DDR RAM）基地址。

LOCAL_MEM_SIZE
平台外部存储器容量。

RAM_LOW_ADRS
bootrom 中非压缩代码 RAM 中运行地址。
Vxworks 载入 RAM 中基地址。

RAM_HIGH_ADRS
bootrom 中压缩代码解压缩基地址以及压缩代码运行起始地址。

ROM_TEXT_ADRS
bootrom ROM 中存储起始地址；romInit 函数 ROM 中运行地址，复位跳转指令地址。romInit 函数开始处必须放置复位向量表。

注意：RAM_HIGH_ADRS 与 RAM_LOW_ADRS 之间的内存容量必须足够大，能够放置被下载的 Vxworks 内核映像，否则将导致 bootrom 的一部分代码被覆盖，将产生一些比较诡异的问题，很难调试。

调试常用命令：nm<arch> -n

```
C:\T22\ppc\target\config\wrSbc824x>nmppc -n bootrom
00100000 T _romInit
00100000 T _wrs_kernel_text_start
00100000 T romInit
00100000 T wrs_kernel_text_start
00100038 t cold
00100044 t warm
00100048 t start
…
00103db4 T inflate
00104190 A _etext
00104190 D _wrs_kernel_data_start
00104190 A _wrs_kernel_text_end
00104190 A etext
00104190 D runtimeName
00104190 D wrs_kernel_data_start
00104190 A wrs_kernel_text_end
00104194 D runtimeVersion
00104198 D vxWorksVersion
0010419c D creationDate
001041a0 D _binArrayStart
001041a0 D binArrayStart
0010c190 T _SDA2_BASE_
001363d0 D _binArrayEnd
001363d0 D binArrayEnd
```

```
…
001502f0 A _end
001502f0 A _wrs_kernel_bss_end
001502f0 A end
001502f0 A wrs_kernel_bss_end
```

可以通过 nm 输出直接查看函数的链接地址，在某些情况下将非常有利于调试的进行，也可以帮助对 bootrom 映像构成的理解。

最后需要提请注意的一点，由于 romInit 的链接地址是在 RAM_LOW_ADRS 指定的 RAM 中，而该函数所有的代码的执行在 ROM 中完成，所以 romInit 代码的编写必须自始至终做到 PIC（位置无关），不可以进行直接跳转，不可以直接进行函数调用，读者可以查看 romInit 的具体实现代码，将看到对于函数调用都是通过相对调用完成的。在进行 romInit 函数的编写时必须特别注意这一点，否则在 romInit 函数执行阶段，系统就将死机，而这一般是比较低级的错误。

小结
本节比较详细的分析了 bootrom 的生成过程和执行流程，其中对于压缩映像的二次链接以及二次拷贝做了较为详细的说明，读者在充分理解本节内容，应该十分清楚相关文件之间的关系，各函数的执行环境，以及整个系统的启动过程。理解这些内容对于 Vxworks 操作系统的移植将具有十分重要的意义。

3.3 深入 Vxworks 启动过程

Vxworks 内核映像类型总体上分为两种：下载型 Vxworks 映像和 ROM 型 Vxworks 映像。下载型 Vxworks 映像需要借助于 bootrom 的前期准备工作，由 bootrom 负责 Vxworks 内核映像的下载并最终跳转到 Vxworks 内核入口函数执行。ROM 型 Vxworks 映像不需要借助任何外部程序，系统上电时就跳转到 Vxworks 内核入口函数进行执行。两种映像类型在启动过程的早期阶段有些不同，当进入到 usrConfig.c 文件中定义的 usrInit 函数后，则启动流程将完全相同。故我们首先从两种映像类型的不同角度分别介绍启动过程的各自早期流程，在分别到达 usrInit 函数后再合二为一进行介绍。

3.3.1 ROM 型映像早期启动流程

所谓 ROM 型映像即不借助任何外部程序的帮助，自从系统上电之时，就执行 Vxworks 内核代码的启动方式。ROM 型映像将预先被烧录入平台 ROM 中（一般为 NorFlash 介质），平台被设计为上电时，CPU 自动跳转到 ROM 起始地址处开始执行第一条指令，而 ROM 型 Vxworks 内核映像的第一条指令也就被放置在 ROM 起始地址处。
ROM 型 Vxworks 内核映像由如下文件生成：romInit.s, bootInit.c, sysALib.s, usrConfig.c, sysLib.c, 外设驱动文件。注意虽然 ROM 型 Vxworks 内核自始至终都没有使用 sysALib.s 文

件中的任何代码，但这个文件总是作为一部分存在于映像之中。系统上电执行的第一条语句就定义在 romInit.s 文件中，该文件完全使用汇编编程，其中主要实现了一个 romInit 函数，该函数开始处按照特定平台要求，一般放置一个系统异常表，即一些跳转指令。事实上，系统上电对于 CPU 而言就是一个复位异常，故其将跳转到复位异常处理程序处执行代码，这个复位异常处理程序就是整个系统的入口，通常这个入口被设置为 romInit 函数。基于 romInit 的链接地址位于 RAM 中，而其整个代码的执行都处于 ROM 中，故 romInit 函数的实现必须自始至终做到位置无关 PIC，这就要求对于函数的调用不能以通常直接的形式，而必须使用如下形式。

#define ROM_ADRS(x) ((x) - _romInit + ROM_TEXT_ADRS)

（(FUNCPTR)ROM_ADRS(romStart)）(startType);

这个位置无关的要求直到 romStart 函数将代码和数据从 ROM 拷贝到 RAM 后才解除。

romInit 函数需要实现如下功能：

1. 配置 CPU 相关寄存器，如设置运行模式，屏蔽系统中断。
2. 初始化平台，这包括外部存储器控制器的初始化，目标板上关键硬件的初始化如时钟，PLL，管脚复用模块，电源管理模块等。
3. 初始化初始栈，供任务创建之前代码使用，具体的，所有在 usrRoot 函数之前运行的代码都没有任务上下文，而是在复位异常上下文中运行，类似于在中断上下文中运行，在涉及到函数调用时，需要一个栈进行参数传递，这个工作必须在 romInit 函数中完成，因为只有它才有机会直接操作栈寄存器，对其赋予一个合理的值。
4. 跳转到 romStart 函数。

BSP 开发中的一个重要组成部分：平台初始化代码基本全部在 romInit 函数中实现或者被其调用。对于 CPU 硬件寄存器的配置一般直接在 romInit 函数中硬编码，而平台外设硬件资源如 DDR 控制器，管脚复用模块等的配置则使用 C 语言编程，而后在 romInit 函数中进行调用。注意由于在 DDR 控制器初始化之前，内存还不可用，而进行函数调用时一般都需要使用栈，此时可以将栈寄存器（SP）初始化成指向 CPU 内部的 DRAM 或者 IRAM。事实上，这是唯一的途径，用户可以一直到 usrRoot 函数一直使用 CPU 内部 RAM 作为初始栈使用。如下是一个实际系统中（ARM926EJS 处理器核）使用的 romInit.s 文件的例子。我们将分段对其进行简单分析。

```
#define     _ASMLANGUAGE
#include "vxWorks.h"
#include "sysLib.h"
#include "asm.h"
#include "regs.h"
#include "config.h"
#include "arch/arm/mmuArmLib.h"

        .data
        .globl      VAR(copyright_wind_river)
        .long       VAR(copyright_wind_river)

/* internals */
        .globlFUNC(romInit)          /* start of system code */
```

```
        .globlVAR(sdata)                /* start of data */
        .globl    _sdata
        .globlVAR(daviciMemSize)  /* actual memory size */

/* externals */
        .extern       FUNC(romStart) /* system initialization routine */
        .extern       FUNC(Platform)
        .extern       FUNC(led_test)

_sdata:
VAR_LABEL(sdata)
        .asciz"start of data"
        .balign       4

        .data
VAR_LABEL(daviciMemSize)
        .long 0

        .text
        .balign 4

/*******************************************************************************
*
* romInit - entry point for VxWorks in ROM
*
* romInit
*         (
*         int startType     /@ only used by 2nd entry point @/
*         )

* INTERNAL
* sysToMonitor examines the ROM for the first instruction and the string
* "Copy" in the third word so if this changes, sysToMonitor must be updated.
*/

_ARM_FUNCTION(romInit)
_romInit:
        B       cold
        B        myexcEnterUndef
        B        myexcEnterSwi
        B        myexcEnterPrefetchAbort
        B        myexcEnterDataAbort
        B        cold
        B        myintEnt
```

```
    B       cold
```

正如前文中一再声称的，在 romInit 函数的开始处必须放置一个系统异常向量表。前文中并没有对此进行解释，实际上其中有一个十分重要的原因。对于 ARM 处理器而言，其硬件上要求必须在绝对地址 0 或者高端地址 0xffff0000 处建立系统异常向量表。至于低端还是高端由一个寄存器位控制，通常情况下设置为低端地址。此时如当一个复位异常产生后，ARM CPU 将 IP 寄存器的值设置为 0，即跳转到复位中断向量处执行，完成复位异常的响应。由于系统异常向量表中每个异常只能使用 4 个字节，故一般都是一个跳转语句，跳转到真正的处理程序进行异常的处理。某些平台在集成 ARM CPU 时会修改该默认行为，如当一个复位异常产生时，其并非跳转到绝对地址 0 处，而是跳转到平台 Flash 或者 ROM 占据的地址起始处，从 ROM 或者 Flash 起始地址处开始执行代码，因为在绝对地址 0 处没有存储介质对应或者存储介质为易失的，在上电之时尚无有效代码。也就是说，这些平台刻意的将 ARM 处理器的系统异常向量表移到另一个地址处。通常这个地址就是系统上电时执行的第一条指令所在地址，通常也就是平台上 Flash 或者 ROM 所在的起始地址。而对于 ROM 型启动方式下的 Vxworks 映像而言，这个起始地址就存放着 romInit 函数的实现代码，故在 romInit 函数的开始处必须放置一张系统异常向量表！这个向量表将在系统运行期间一直被使用，包括 IRQ，FIQ 中断响应都要经过这个系统异常向量表的过渡，这一点十分重要，将直接影响代码编写方式。我们需要仔细编写 romInit 函数实现将 IRQ 中断响应程序设置为 Vxworks 内核提供的函数 intEnt。事实上，在代码预处理阶段，所有的常量都将被设置为一个数字值，烧入 ROM 或者 Flash 后都是不可更改的，由于在 romInit 执行时，必须做到位置无关，故我们不可以直接将 IRQ 的跳转指令写成“B intEnt”，如此将造成执行路线跳转到 RAM 中，因为 intEnt 链接的地址为 RAM_LOW_ADRS 为基地址的 RAM 中地址，而此时 RAM 还不存在任何有效指令（知道 romStart 执行完毕才可以进行如此跳转）。更深层次的原因在于 ROM 型 Vxworks 内核映像一般都是经过压缩的，此时 romInit 以及 romStart 函数与压缩部分不是在一起进行链接的，而是经过二次链接过程（二次链接过程请参考本书前文中“深入理解 bootrom”一节），即在 romInit 函数中根本无法“看到”intEnt 函数的存在。必须在进入 usrInit 函数之后才可以使用 intEnt 函数地址。所以在 romInit 函数 IRQ 向量的跳转方式采用二次跳转的方式进行。首先跳转到 romInit 函数中定义的 myintEnt 标号处，这是一个相对跳转，myintEnt 则将（MY_EXC_BASE+0x10）RAM 地址处的值作为 PC 值进行第二次跳转，这个（MY_EXC_BASE）RAM 地址指向的区域将在 sysHwInit 函数被初始化，具体的，对于（MY_EXC_BASE+0x10）RAM 地址处，将以 intEnt 函数地址进行初始化。经过这样一个二次跳转，最终将 IRQ 的中断入口设置为内核提供的 IRQ 入口函数 intEnt。这个二次跳转的思想十分巧妙和实用。

基于以上分析，我们有二次跳转过渡语句定义如下，读者需要结合 romInit.s 文件尾部一系列诸如 L$_promUndef 之类的定义来看。这些过渡语句将特定 RAM 内存处（MY_EXC_BASE）存储的值作为 PC 值进行跳转，内核初始化过程中（如在 sysHwInit 函数中完成）将使用 Vxworks 内核提供的异常响应函数地址初始化这片由 MY_EXC_BASE 指向的 RAM 内存区域，从而完成向 Vxworks 内核提供的异常处理程序的衔接。

```
myexcEnterUndef:
    sub sp,sp,#4
    stmfd sp!,{r0}
    ldr r0,L$_promUndef
    ldr r0,[r0]
```

```
        str r0,[sp,#4]
        ldmfd sp!,{r0,pc}
myexcEnterSwi:
        sub sp,sp,#4
        stmfd sp!,{r0}
        ldr r0,L$_promSwi
        ldr r0,[r0]
        str r0,[sp,#4]
        ldmfd sp!,{r0,pc}
myexcEnterPrefetchAbort:
        sub sp,sp,#4
        stmfd sp!,{r0}
        ldr r0,L$_promPrefectchAbort
        ldr r0,[r0]
        str r0,[sp,#4]
        ldmfd sp!,{r0,pc}
myexcEnterDataAbort:
        sub sp,sp,#4
        stmfd sp!,{r0}
        ldr r0,L$_promDataAbort
        ldr r0,[r0]
        str r0,[sp,#4]
        ldmfd sp!,{r0,pc}
myintEnt:
        sub sp,sp,#4
        stmfd sp!,{r0}
        ldr r0,L$_promintEnt
        ldr r0,[r0]
        str r0,[sp,#4]
        ldmfd sp!,{r0,pc}


cold:
        MOV     r0, #BOOT_COLD      /* fall through to warm boot entry */
warm:
        B       start
```

冷启动时设置对应参数值，romStart 函数将根据这个参数决定是否对有关内存区域进行清零处理。

```
        /* copyright notice appears at beginning of ROM (in TEXT segment) */

        .ascii      "Copyright 1999-2004 ARM Limited"
        .ascii      "\nCopyright 1999-2006 Windriver Limited"
```

```
        .balign 4

start:

        /* Enabling the Main Oscillator*/
        /*
         * There have been reports of problems with certain boards and
         * certain power supplies not coming up after a power-on reset,
         * and adding a delay at the start of romInit appears to help
         * with this.
         */

        TEQS    r0, #BOOT_COLD
        MOVEQ   r1, #ARM926EJS_DELAY_VALUE
        MOVNE   r1, #1

delay_loop:
        SUBS    r1, r1, #1
        BNE delay_loop
```

如上代码根据是否是冷启动（即上电启动）决定是否等待一段时间。某些平台要求在上电后等待一段时间才能进行相关外设的初始化过程。当然这个等待时间从宏观上将都比较短。

```
#if defined(CPU_926E)

        /*
         * Set processor and MMU to known state as follows (we may have not
         * been entered from a reset). We must do this before setting the CPU
         * mode as we must set PROG32/DATA32.
         *
         * MMU Control Register layout.
         *
         * bit
         *    0 M 0 MMU disabled
         *    1 A 0 Address alignment fault disabled, initially
         *    2 C 0 Data cache disabled
         *    3 W 0 Write Buffer disabled
         *    4 P 1 PROG32
         *    5 D 1 DATA32
         *    6 L 1 Should Be One (Late abort on earlier CPUs)
         *    7 B ? Endianness (1 => big)
         *    8 S 0 System bit to zero } Modifies MMU protections, not really
         *    9 R 1 ROM bit to one      } relevant until MMU switched on later.
         * 10 F 0 Should Be Zero
```

```
        * 11 Z 0 Should Be Zero (Branch prediction control on 810)
        * 12 I 0 Instruction cache control
        */

       /* Setup MMU Control Register */

       MOV     r1, #MMU_INIT_VALUE          /* Defined in mmuArmLib.h */
       MCR CP_MMU, 0, r1, c1, c0, 0    /* Write to MMU CR */

       /*
        * If MMU was on before this, then we'd better hope it was set
        * up for flat translation or there will be problems. The next
        * 2/3 instructions will be fetched "translated" (number depends
        * on CPU).
        *
        * We would like to discard the contents of the Write-Buffer
        * altogether, but there is no facility to do this. Failing that,
        * we do not want any pending writes to happen at a later stage,
        * so drain the Write-Buffer, i.e. force any pending writes to
        * happen now.
        */

       MOV     r1, #0                       /* data SBZ */
       MCR CP_MMU, 0, r1, c7, c10, 4   /* drain write-buffer */

       /* Flush (invalidate) both I and D caches */

       MCR CP_MMU, 0, r1, c7, c7, 0    /* R1 = 0 from above, data SBZ*/


          /*
        * Set Process ID Register to zero, this effectively disables
        * the process ID remapping feature.
        */

       MOV     r1, #0
       MCR CP_MMU, 0, r1, c13, c0, 0
#endif
       /* disable interrupts in CPU and switch to SVC32 mode */

       MRS r1, cpsr
       BIC   r1, r1, #MASK_MODE
       ORR r1, r1, #MODE_SVC32 | I_BIT | F_BIT
       MSR cpsr, r1
```

如上代码设置 ARM CP15 控制寄存器相关控制位，如清除 cache，禁止 MMU；其后设置运行模式为系统管理模式，禁止系统 IRQ，FIQ 中断。

```
        /*
         * CPU INTERRUPTS DISABLED
         *
         * disable individual interrupts in the interrupt controller
         */

        MOV         r1, #0x00000000
        LDR r0, =IC_EINT0
        STR  r1, [r0]
        LDR r0, =IC_EINT1
        STR  r1, [r0]
```

配置外设中断控制器寄存器，禁止所有外设中断。

```
        /*
         * Jump to the normal (higher) ROM Position. After a reset, the
         * ROM is mapped into memory from* location zero upwards as well
         * as in its normal position at This code could be executing in
         * the lower position. We wish to be executing the code, still
         * in ROM, but in its normal (higher) position before we remap
         * the machine so that the ROM is no longer dual-mapped from zero
         * upwards, but so that RAM appears from 0 upwards.
         */

        LDR pc, L$_HiPosn
HiPosn:
/*********************************************
          set up Pinmiux
**************************************************/
        LDR R6, =_PINMUX0
        LDR R7, =0x80000C1F
        STR R7,[R6]



        LDR R6, =_PINMUX1
        LDR R7, =0x000404F1
        STR R7,[R6]

        LDR R6, =VDD3P3V_PWDN
        LDR R7, =0x0
```

```
    STR R7,[R6]
```

以上语句配置管脚复用寄存器，即根据平台要完成的特定功能设定某些硬件管教的功能。

```
    /*
     *
     * Initialize the stack pointer to just before where the
     * uncompress code, copied from ROM to RAM, will run.
     */

    LDR sp, =0xB000
```

设置初始栈，为下面调用平台初始化代码（C 语言编写）做准备。注意由于此时尚未初始化 DDR 控制器，故外部存储器不可用，此处将 SP 设置为 CPU 内部的 RAM 空间地址，使用内存 RAM 作为初始栈。

```
    mov lr,pc
    LDR pc,L$_rRomInit_C
```

以上两个语句首先保存返回地址到 lr 寄存器中，其后调用 Platform 函数进行平台初始化。注意函数的特殊调用方式，读者结合 romInit.s 文件末尾对于 L$_rRomInit_C 的定义来看，这是采用一种相对调用方式，即位置无关调用方式在进行。

```
    /* jump to C entry point in ROM: routine - entry point + ROM base */

#if    (ARM_THUMB)
    LDR r12, L$_rStrtInRom
    ORR r12, r12, #1        /* force Thumb state */
    BX   r12
#else
    /*LDR      sp,=0x88000000*/
    MOV      r0, #BOOT_COLD
    LDR pc, L$_rStrtInRom

#endif      /* (ARM_THUMB) */
```

romInit 函数最后通过位置无关方式跳转到 romStart 函数执行。

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

/*
 * PC-relative-addressable pointers - LDR Rn,=sym is broken
 * note "_" after "$" to stop preprocessor performing substitution
```

```
 */

        .balign      4


L$_HiPosn:
        .long ROM_TEXT_ADRS + HiPosn - FUNC(romInit)

L$_rStrtInRom:
        .long ROM_TEXT_ADRS + FUNC(romStart) - FUNC(romInit)

L$_rRomInit_C:
        .long ROM_TEXT_ADRS + FUNC(Platform) - FUNC(romInit)
/*
L$_rEmacInRom:
        .long ROM_TEXT_ADRS + FUNC(emacStart) - FUNC(romInit)
*/
L$_STACK_ADDR:
        .long STACK_ADRS
L$_memSize:
        .long VAR(daviciMemSize)

L$_promUndef:
        .long MY_EXC_BASE
L$_promSwi:
        .long MY_EXC_BASE+4
L$_promPrefectchAbort:
        .long MY_EXC_BASE+8
L$_promDataAbort:
        .long MY_EXC_BASE+0xc
L$_promintEnt:
        .long MY_EXC_BASE+0x10
```

romInit 函数末尾是对一些符号常量的定义，前面一些表示函数的相对地址，后面则为异常向量表各表项在 RAM 中的位置。

romInit 函数最后跳转到 romStart 函数进行执行，romStart 根据映像类型将原先存储在 ROM 中的代码和数据（对于 rom resident 映像类型而言，则是单纯的数据，不包括代码）拷贝到 RAM 中，如果存在压缩，则在拷贝过程中进行解压缩。本书前文“深入理解 bootrom”一节已经对 romStart 函数做了详细的分析，Vxworks 内核映像与 bootrom 映像使用相同 romStart 函数，故此处不作详细论述。另前文“映像类型“一节已经对于压缩型和非压缩型代码的拷贝方式进行了总体叙述，此处为了加深读者映像，我们再次进行说明。
非驻留 ROM 映像类型分为压缩和非压缩两种，压缩映像类型中只有 romInit.s，bootInit.c 文件是非压缩的，其余部分代码都是压缩的。压缩映像类型从 ROM 向 RAM 拷贝时需要分两

个阶段进行，第一阶段将非压缩代码（romInit.s，bootInit.c）从 ROM 中拷贝到 RAM_HIGH_ADRS 指向的 RAM 区域；第二阶段将压缩代码（usrConfig.c，sysLib.c，sysALib.c，外设驱动代码）从 ROM 中拷贝到 RAM_LOW_ADRS 指向的 RAM 区域，并在拷贝过程中进行解压缩操作。注意解压后代码的入口为定义在 usrConfig.c 文件中的 usrInit 函数。如图 3-6 所示为压缩型 Vxworks 映像类型 ROM 及完成拷贝后 RAM 布局图。

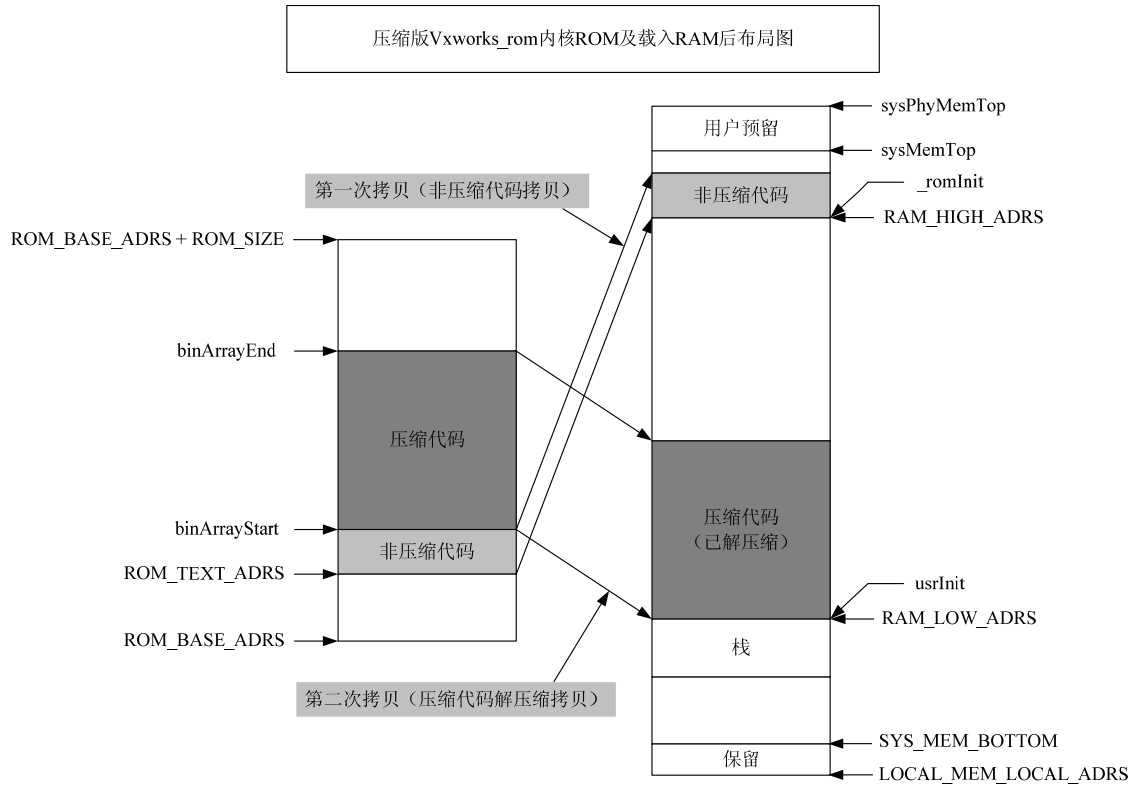


图 3- 6 压缩型 Vxworks 映像类型 ROM 及拷贝后 RAM 布局图

对于非 ROM 驻留非压缩型 Vxworks 映像类型，拷贝过程则要简单的多，此时只存在一次拷贝，所有 ROM 中代码和数据都被一次性拷贝到由 RAM_LOW_ADRS 指定的 RAM 区域。如图 3-7 所示。

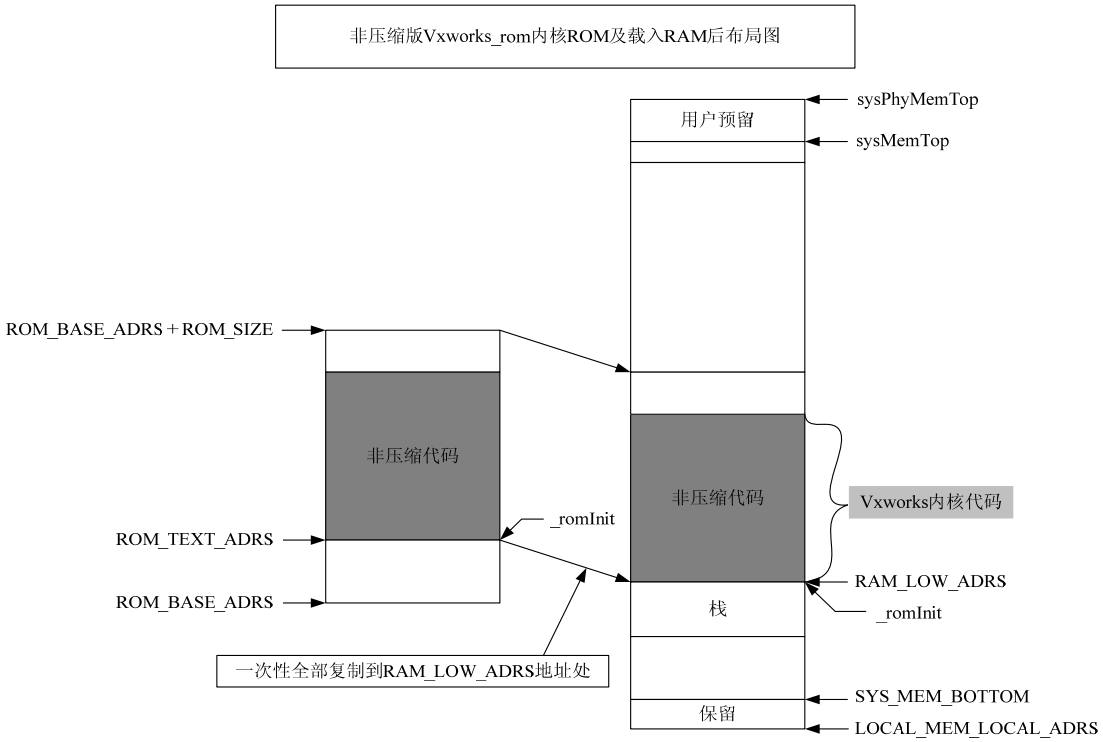


图 3- 7 非压缩型 Vxworks 映像类型 ROM 及拷贝后 RAM 布局图

读者分析 romStart 函数实现，会发现非常有趣的一个现象：当映像是非压缩类型时，入口函数指针被赋值为 usrInit 函数地址；而当映像是压缩类型时，入口函数指针则被赋值为(FUNCPTR)RAM_DST_ADRS，对于 Vxworks 映像而言，RAM_DST_ADRS 即等于 RAM_LOW_ADRS，即对于压缩类型，入口函数指针被直接赋值为 RAM_LOW_ADRS。实际上这种不同的入口函数指针赋值方式是有深刻含义的。对于非压缩 Vxworks 映像类型而言，映像的生成过程只存在一次链接过程，所有的代码以 romInit 为入口函数被链接到 RAM_LOW_ADRS 地址处，注意在非压缩映像拷贝过程中，所有代码从 romInit 开始到 data 段的结束都是一次性拷贝到 RAM_LOW_ADRS 指向的 RAM 地址处，换句话说，实际上此时 RAM_LOW_ADRS 地址处放置的是 romInit 函数实现，其后跟随着 romStart 函数实现，其后跟随着 usrInit 函数实现，所以此时我们不可以使用 RAM_LOW_ADRS 地址再次作为跳转地址，如是，则又将跳转回 romInit 函数执行了！所以必须明确以 usrInit 函数地址的方式对 romStart 之后执行的入口函数指针进行初始化。对于压缩性 Vxworks 映像类型则不然，该映像的生成使用了二次链接过程：第一次是压缩代码本身一次链接，其以 usrInit 函数为入口被链接到 RAM_LOW_ADRS 指向地址处；第二次是非压缩代码（romInit.s，bootInit.c）与压缩代码整合在一起时的链接过程，此时压缩代码作为数据部分被毫无改变的集成到最后映像中，此时以 romInit 函数为入口链接到 RAM_HIGH_ADRS 指向的地址处。所以 romStart 拷贝过程中分为两次进行拷贝，在第二次对压缩代码拷贝过程同时进行了解压缩，将 usrInit 为入口的代码块解压到 RAM_LOW_ADRS 地址处，即此时 RAM_LOW_ADRS 地址处存储着 usrInit 函数实现的第一条指令，故 romStart 之后执行的入口函数指针被简单赋值为 RAM_LOW_ADRS，实际上此时赋值为 usrInit 也是可行的。Vxworks 内核提供者使用 RAM_LOW_ADRS 直接赋值，其用意是否正是告知用户其内在差别之处。所以看似简单的两种不同赋值方式，其后有深刻的含义。

对于 ROM 驻留型映像类型，首先必须是非压缩的，其次 romStart 无需拷贝代码部分，只需将数据部分拷贝到 RAM_LOW_ADRS 指向 RAM 区域，较为简单，结合以上两种映像类型拷贝方式，对此应不难理解。

无论何种 ROM 型映像类型，romStart 完成拷贝后，都将跳转到 usrConfig.c 文件中定义的 usrInit 函数执行，与下载型 Vxworks 映像类型殊途同归。

3.3.2 下载型映像早期启动流程

相对于 ROM 型映像类型，下载型映像类型早期启动流程则较为简单。在进入 usrInit 函数前，其只执行一个函数，即 sysALib.s 文件中定义的 sysInit 函数。下载 Vxworks 内核映像既可以是压缩的，也可以是非压缩的。如果是非压缩的，那么 bootrom 将直接将 Vxworks 映像下载到 RAM_LOW_ADRS 指定的 RAM 地址处，而对于压缩类型而言，则需要进行解压缩的工作，此时首先需要将压缩映像下载到一个其他地址处（相对于 RAM_LOW_ADRS），再从这个地址解压缩到 RAM_LOW_ADRS，无论如何，最后位于 RAM_LOW_ADRS 地址处的一定是一个非压缩的 Vxworks 内核映像，第一条指令就是 sysInit 函数实现。这就要求 sysInit 函数必须时 sysALib.s 文件中的第一个函数，否则无法做到 RAM_LOW_ADRS 的第一条指令就是 sysInit 函数实现。

当然这同时还要从 bootrom 执行完毕后跳转到 Vxworks 内核映像的跳转语句编码方式说起。我们以常见的网络下载方式为例。bootrom 后期执行流程如下(注意如下这些都定义在 bootConfig.c 文件中)：

usrInit→usrRoot→bootCmdLoop→autoboot→bootLoad→netLoad→bootLoadModule

bootLoadModule 函数由 Vxworks 内核提供，传递给这个函数的参数有两个：第一个参数为打开的远程主机上 Vxworks 内核映像的文件句柄；第二个参数为 Vxworks 内核入口函数指针，该参数需要 bootLoadModule 函数进行初始化。当以上函数层层返回后，autoboot 函数中将跳转到 bootLoadModule 函数返回的 Vxworks 内核入口函数指针指向的地址处，即进入 Vxworks 内核映像执行。autoboot 函数中的相关语句如下：

```
if (bootLoad (BOOT_LINE_ADRS, &entry) == OK)
        go (entry);       /* ... and never return */
```

以上代码中的第二个参数将层层传递，最后传递给 bootLoadModule 函数，由其进行赋值。事实上，bootLoadModule 函数也只是一个封装函数，其根据 Vxworks 内核映像的文件格式进一步调用下层函数。Vxworks 映像格式一般为 ELF 格式，故 bootLoadModule 将调用 bootElfModule 函数，该函数最后根据 ELF 文件头部中的入口函数地址对 entry 变量进行初始化，换句话说是使用 sysInit 函数地址对 entry 变量进行了赋值。从这个意义上讲，我们并不需要将 sysInit 函数定义为 sysALib.s 文件中的第一个函数，此时也不会对正常启动构成任何映像，不过 Wind River 要求将 sysInit 函数定义为 sysALib.s 文件中的第一个函数，建议读者遵守这个要求。

sysInit 函数完成的功能如同 romInit 函数，但是由于其执行在其链接地址上，故无需位置无关。采用 bootrom + Vxworks 的下载启动方式时，Wind River 要求 Vxworks 的初始化完全不依赖于 bootrom，换句话说，其要求在完成 Vxworks 映像的下载，Vxworks 必须“忘记”

bootrom 的存在，即其必须进行一切使其进入正常运行状态的准备工作，如平台初始化。基于这个要求，事实上，我们可以将 romInit 函数实现中除了开始处的系统异常表之外的其他所有代码原封不动的作为 sysInit 函数的实现。但是一般我们并不这样做，我们还需要对 bootrom 进行一下“怀念”，所以我们同时将平台初始化代码也删除掉，当然 sysInit 最后是跳转到 usrInit，而非 romStart。至此，我们完成 sysInit 函数的实现。
注意：我们在 sysInit 函数的实现中，并未“忘记”bootrom 的存在，这一点看似与 Wind River 的建议相违背，但是这仅仅是一个建议，事实上，下载启动方式下，Vxworks 不可能脱离 bootrom 的存在，即 bootrom 完成 Vxworks 映像的下载后，依然在起着关键作用，这就是 romInit 函数开始处的“系统异常向量表”，这个向量表一直被 Vxworks 内核使用或者说一直作为所有系统异常到 Vxworks 内核的“桥梁”。所以也就没有必要在 sysInit 函数中做些无用功，还有一个更根本的原因就是，某些平台设备已经不可能重新初始化，如 DDR 控制器，因为 Vxworks 内核映像就是运行在 RAM 中，如重新初始化 DDR 控制器，初始化的基本流程：先复位一个器件，再配置这个器件的寄存器，而一旦对 DDR 控制器进行复位，以后就无可能再运行 Vxworks 内核代码了，何谈再配置寄存器？系统将直接死机。基于这个根本原因，sysInit 实现中需要将平台初始化代码剔除。
注意：这一点非常重要，必须将平台初始化代码从 sysInit 函数中剔除，否则将造成系统死机。当然如果 sysInit 函数对其他一些非关键外设硬件（即不包括 DDR 控制器，电源管理模块等）进行初始化则是可行的。

sysInit 函数最后跳转到 usrConfig.c 文件中定义的 usrInit 函数执行，与 ROM 型 Vxworks 内核映像殊途同归。

下面一节我们将从 usrInit 函数出发，讲述所有 Vxworks 内核映像类型共同的启动流程。

3.3.3 公共启动流程

任何 Vxworks 内核映像从 usrInit 函数开始，都将具有相同的执行流程。不同映像类型在进入 usrInit 函数之前根据各自的特殊情况有所差别，这些差别主要从平台初始化的时机而言的，如 ROM 型映像类型，由于无其他外部任何代码可以依赖，其必须靠自身完成所有的初始化工作；而下载型映像类型，由于可以借助于 bootrom 完成的工作，故其只是做些简单的初始化。当所有映像类型的代码执行到 usrInit 函数时，此时将开始内核组件和外设驱动的初始化，这对于所有的映像类型都将是一致的，所有无论何种 Vxworks 内核映像类型，无论其早期启动流程为何，在进入 usrInit 函数后，都具有相同的启动流程。本节即从 usrInit 函数出发，较为详细介绍所有 Vxworks 内核映像类型共同的启动流程，直到 Vxworks 操作系统完全启动，进入正常运行状态。
如下代码示例是一个实际系统（基于 ARM926EJS 核）中运行的 BSP usrConfig.c 文件源码，我们使用如下预处理命令对其进行了预处理。
C:\Tornado2.2\target\config\all>ccarm -E –I\h -I..\arm_cao -I. -Ic:\Tornado2.2\target\config\all -Ic:\Tornado2.2\target/h -Ic:\Tornado2.2\target/src/config -Ic:\Tornado2.2\target/src/drv -DCPU_926E usrConfig.c >usrConfig.i
其中-E 选项指定预处理，-I 指定头文件搜索路径，‘arm_cao’为 BSP 目录名，‘CPU_926E’为 CPU 类型定义。

以下是经过预处理后的 usrConfig.c 文件中 usrInit 函数实现代码。

```
void usrInit (int startType){
    while (trapValue1 != 0x12348765 || trapValue2 != 0x5a5ac3c3){
        ;
     }

    sysHwInit0 ();
    cacheLibInit (0x01, 0x02);
    bzero (edata, end - edata);
    sysStartType = startType;
    intVecBaseSet ((FUNCPTR *) ((char*)0));
    excVecInit ();

    sysHwInit ();

    usrKernelInit ();

    cacheEnable (INSTRUCTION_CACHE);

    cacheEnable (DATA_CACHE);

    kernelInit ((FUNCPTR) usrRoot, 0x4000,
        (char *) ((end) + ((sysMemTop() - (end))/16)),
        sysMemTop (), 0x2000, 0);
}
```

Vxworks 内核映像中 usrInit 函数实现与 bootrom 映像中 usrInit 实现基本类似，主要完成如下几个方面的工作：

1. 外设硬件初始化工作，此处的初始化将所有使用的外设硬件设置为“安静”状态：完成所有相关寄存器的配置，只等注册中断和使能工作。这部分工作由 sysHwInit0，sysHwInit 两个函数完成。
2. cache 内核组件的初始化以及 CPU 内存 cache 控制寄存器的配置。这部分工作由 cacheLibInit，cacheEnable 两个函数完成。
3. Vxworks 内核在绝对地址 0 处（位于 CPU 的内部 IRAM 空间）创建系统异常向量表。这部分工作由 excVecInit 函数完成。注意 intVecBaseSet 设置的是内核维护的外设中断向量表的基地址，事实上，Vxworks 内核并未使用该函数，换句话说，intVecBaseSet 内部实现为空，外设中断向量表的位置由内核自行进行分配。有关外设中断向量表的更多内容请参考本书前文“中断”一节。
4. 对 BSS 段清零，设置全局变量 sysStartType 为启动类型：冷启动或热启动。此处设置避免了将启动类型老是作为参数进行函数间传递的麻烦。
5. 最后创建系统的第一个任务，usrRoot 为任务入口函数。注意：‘end’表示 Vxworks 内核映像中 BSS 段的结束地址，即整个 Vxworks 内核映像的结束地址。kernelInit 的调用原型如下，对照该原型，我们可以直接看到各参数的含义。

```
void kernelInit
    (
    FUNCPTR      rootRtn,   /* user start-up routine */
    unsigned  rootMemSize, /* memory for TCB and root stack */
    char *       pMemPoolStart,     /* beginning of memory pool */
    char *       pMemPoolEnd,       /* end of memory pool */
    unsigned  intStackSize,   /* interrupt stack size */
    int          lockOutLevel  /* interrupt lock-out level (1-7) */
    );
```

自 usrRoot 函数开始，代码开始运行在任务上下文中。Vxworks 每个任务都具有自己的栈空间，这是与任务控制结构 TCB 作为一个整体进行分配。这个分配的连续地址空间底部用于 TCB，顶部用于任务栈。第二，第三个参数指定系统内存池的起始和结束地址。注意起始地址相对于 Vxworks 内核映像偏移了((sysMemTop() - (end))/16)，即整个可用内存空间的 1/16 被用于 WDB 内存池，我们可以从 configAll.h 文件中的如下定义看到如上计算式的由来。

```
#define WDB_POOL_SIZE    ((sysMemTop() - FREE_RAM_ADRS)/16) /* memory pool for host tools */
```

intStackSize 指预留给中断栈的内存大小，这个内存不是 malloc 分配的，而是直接从系统内存池预留，相关代码如下：

```
    vxIntStackBase = pMemPoolStart + intStackSize;
    vxIntStackEnd    = pMemPoolStart;
    bfill (vxIntStackEnd, (int) intStackSize, 0xee);

    windIntStackSet (vxIntStackBase);
    pMemPoolStart = vxIntStackBase;
```

我们可以看到中断栈直接从内存池起始处预留，而内存池起始地址相对的被向上偏移了 intStackSize。对于 ARM 体系结构，共有 7 中工作模式，实际上每种模式都需要有自己的栈空间，此处只指定了 IRQ 的中断栈大小，Vxworks 不使用 FIQ 中断，而对于其他模式下栈的大小将按内核默认值进行分配。最后一个参数指定了 intLock 调用时被锁定（即被禁止）的中断优先级别，参数 0 表示禁止任何级别的中断。

注意：实际上中断锁定级别的使用十分有限，如在 ARM 体系结构下，此处传入的任何参数都不会影响 intLock 的行为。ARM 体系结构下 intLock 直接从系统角度禁止了 IRQ 中断（即将模式寄存器中的 I 位置 1），即禁止了所有中断，包括系统时钟中断。

```
void usrRoot(char *     pMemPoolStart, unsigned     memPoolSize){
    char tyName [20];
    int    ix;

    memInit (pMemPoolStart, memPoolSize);
    memShowInit ();
```

初始化系统内存池以及内存信息显示组件。

```
sysClkConnect ((FUNCPTR) usrClock, 0);
sysClkRateSet (60);
sysClkEnable ();
```

注册系统时钟中断函数，设置时钟中断频率，最后使能中断。对于以上三个语句的底层含义请参考本书前文“中断”一节的讨论。

```
sysTimestampEnable();
```

启动系统时间戳定时器。

```
selectInit (50);
```

select 内核组件初始化。Select 功能可以同时监听多个文件输入输出行为。该函数调用原型如下。

```
void selectInit
    (
    int             numFiles    /* maximum number of open files */
    );
```

参数表示的是最大同时可以监听的文件句柄数。

```
iosInit (20, 50, "/null");
```

初始化 IO 子系统。iosInit 调用原型如下，从中可以得知各参数含义。

```
STATUS iosInit
    (
    int max_drivers,                /* maximum number of drivers allowed */
    int max_files,                  /* max number of files allowed open at once */
    char *nullDevName                   /* name of the null device (bit bucket) */
    );
```

iosInit 函数参数：

参数 1：系统支持的最大驱动数。

参数 2：系统支持的最大同时打开的文件数。

参数 3：null 文件文件名。null 文件可以吸收所有写入它的内容，这个功能可以屏蔽某些打印。

```
consoleFd = (-1);
if (3 > 0){
        ttyDrv();

        for (ix = 0; ix < 3; ix++)
        {
        sprintf (tyName, "%s%d", "/tyCo/", ix);
        (void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);
```

```
        if (ix == 0)
            {
                strcpy (consoleName, tyName);
                consoleFd = open (consoleName, 2, 0);
                (void) ioctl (consoleFd, 4, 115200);
                (void) ioctl (consoleFd, 3, (0x01 | 0x02 | 0x04 |
                                                    0x10 | 0x08 | 0x20 | 0x40));
            }
        }
    }

    ioGlobalStdSet (0, consoleFd);
    ioGlobalStdSet (1, consoleFd);
    ioGlobalStdSet (2, consoleFd);
```

串口设备初始化并设置标准输入，输出，错误输出句柄均指向串口设备。自此之后 printf 语句可用。

```
    printf("Constructing components to enable cache and mmu!\n");
    printf("This may take half a minute, please wait...\n\n");

    usrMmuInit ();
```

MMU 内核组件初始化，使支持虚拟地址空间。有关 MMU 的详细内容，请参考本书前文“内存管理”一节。

```
    printf("Succeed enabling mmu and instruction,data cache!\n\n\n");

    excShowInit ();
    excInit ();
```

创建 tExcTask 任务，在任务上下文中处理异常。当一个异常发生时，Vxworks 可以暂缓对异常的处理，而将异常处理工作交给 tExcTask 任务，在任务上下文中进行处理。这种功能对于某些异常十分必要，因为这些异常需要一个上下文环境。注意：此处异常不同于处理器本身的异常，处理器本身的异常必须即时处理，一般不可延迟。

```
    logInit (consoleFd, 50);
```

系统打印任务初始化，该函数将创建一个系统任务，在任务上下文打印用户信息。logMsg 用于向任务维护的信息队列中添加信息。这个函数可以在任何场合进行调用，包括中断上下文。logInit 函数调用原型如下。

```
STATUS logInit
    (
```

```
    int fd,                 /* file descriptor to use as logging device */
    int maxMsgs             /* max. number of messages allowed in log queue */
    );
```

参数 1：logMsg 信息的输出文件句柄。
参数 2：消息队列中的最大信息条数。

```
    sigInit ();
```

信号内核组件初始化。信号是任务间通信的一种有效方式，也是控制任务运行的一种方式。

```
    pipeDrv ();
```

管道内核组件初始化。Vxworks 下管道底层实现建立在消息队列之上，故提供了一种任务间面向包的信息交互方式。

```
    stdioInit ();
    stdioShowInit ();
```

标准输入输出内核组件初始化。这个函数完成标准输入输出支持库的安装。这个库是一个中间层，将用户层和文件系统层衔接在一起。

```
    hashLibInit ();
    dosFsLibInit( 0 );
    dosVDirLibInit();
    dosDirOldLibInit();
    dosFsFatInit();
    dosChkLibInit();
    dosFsFmtLibInit();
    hashLibInit ();
    dosFsInit (20);
```

MS-DOS 兼容型文件系统组件初始化，该部分内容较大，此处不作展开。

```
    ramDrv ();
```

RAM 文件系统内核组件初始化。RAM 文件系统支持以 RAM 为介质创建一个文件系统。这个功能可用以暂存系统关键信息。我们可以创建一个 RAM 文件系统，以文件方式存储运行时信息，而系统下电后，这些信息又必须得到清除，RAM 文件系统是一种十分有效的方式。

```
    tffsDrv ();
```

TFFS 内核组件初始化。TFFS 是一个中间层，连接文件系统和底层 FLASH 驱动。其将 FLASH 模拟成硬盘设备进行读写，在 Flash 上创建文件系统。

```
fioLibInit ();
floatInit ();
mathSoftInit ();
timexInit ();
envLibInit (1);
moduleLibInit ();
loadElfInit ();
```

ELF 文件格式解析模块初始化。这个模块将负责对 ELF 格式文件（如 Vxworks 内核映像）的解析，包括动态链接 ELF 文件等。

```
usrBootLineInit (sysStartType);
```

获取启动 bootline 参数，并将其写入((0x80000000)+0x700)内存地址处，将被 usrNetInit 函数使用。usrBootLineInit, usrNetInit 函数均定义在 target/src/config/usrNetwork.c 文件中。读者可以具体查看二者的实现。

```
usrNetInit (((char *) ((0x80000000)+0x700)));
```

内核网络组件初始化，usrNetInit 的调用原型如下。

```
STATUS usrNetInit
(
char *bootString          /* boot parameter string */
);
```

参数表示 bootline 值，Vxworks 内核将根据 bootline 解析出 IP 地址，网卡 MAC 地址等等信息，用以对内核网络支持组件进行初始化并设置网口 IP 地址等等。

```
selTaskDeleteHookAdd ();

cplusCtorsLink ();

rBuffLibInit();

rBuffShowInit ();

windviewConfig ();

wdbConfig();
```

WDB 调试引擎初始化，该引擎将负责对主机发送调试命令的响应。

```
printLogo ();

printf ("                                        ");
```

```
printf ("CPU: %s.    Processor #%d.\n", sysModel (), sysProcNumGet ());
printf ("                                                    ");

printf ("Memory Size: 0x%x.", (UINT)(sysMemTop () - (char *)(0x80000000)));

printf ("    BSP version " "1.2" "/0" ".");

printf ("\n                                                  ");
printf ("WDB Comm Type: %s", "WDB_COMM_END");
printf ("\n                                                  ");
printf ("WDB: %s.",((wdbRunsExternal () || wdbRunsTasking ())?
        "Ready" : "Agent configuration failed") );

printf ("\n\n");

shellInit (0x10000, 1);
```

启动 tShell 任务，创建 shell 命令终端。shellInit 调用原型如下。

```
STATUS shellInit
(
int stackSize,          /* shell stack (0 = previous/default value) */
int arg                   /* argument to shell task */
);

startUsrRoutine();
```

启动用户任务。startUsrRoutine 为用户自定义函数，在其中启动用户层任务，完成平台特定功能。在 usrRoot 函数最后调用一个用户自定义函数是 Vxworks 提供给用户的在操作系统启动后启动用户任务的一种机制。对于嵌入式系统，一般都需要使用这种机制。

```
}
```

usrRoot 最后启动 shell 终端命令行，完成 Vxworks 操作系统的启动。整个启动界面如下所示。前面一部分斜体字体显示为 bootrom 启动和下载 Vxworks 内核映像过程，后面一部分加粗字体部分显示为 Vxworks 内核自身启动过程。

VxWorks System Boot

Copyright 1984-2002 Wind River Systems, Inc.

CPU: DAVICI DM6446 - ARM926E (SEED)

Version: VxWorks5.5.1

BSP version: 1.2/0

Creation date: Apr 28 2009, 23:22:51

sysNvRamGet: 0 ff 10000

atemac-tffs=0,0(0,0)vxw:vxWorks e=192.168.1.176:ffffff00 h=192.168.1.210 g=192.168.1.1

u=vxw pw=vxw tn=ARM926EJS o=12:87:59:e3:9b:06

Press any key to stop auto-boot...
0
auto-booting...

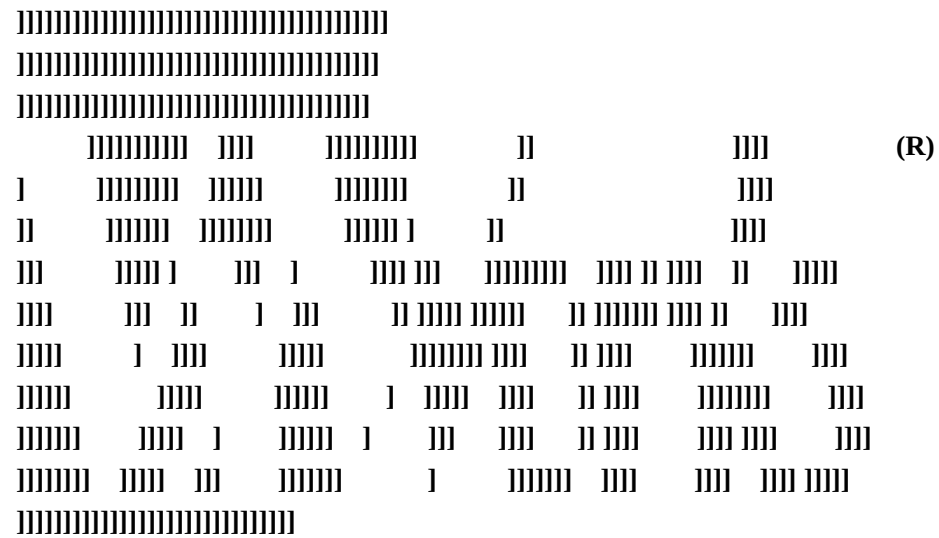
boot device : atemac-tffs=0,0
unit number : 0
processor number : 0
host name : vxw
file name : vxWorks
inet on ethernet (e) : 192.168.1.176:ffffff00
host inet (h) : 192.168.1.210
gateway inet (g) : 192.168.1.1
user (u) : vxw
ftp password (pw) : vxw
flags (f) : 0x0
target name (tn) : ARM926EJS
other (o) : 12:87:59:e3:9b:06

bootLoad:loading network interface.
findCookie-input param:unitNo=0,devName=atemac.
bootLoad:start netdriver...
bootLoad:Done starting netdriver!
Attached TCP/IP interface to atemac0.
Attaching loopback interface ...
Attaching network interface lo0... done.
Attaching to Tffs ... done.
Loading /tffs/vxWorks ...

Cannot open "/tffs/vxWorks".
tffsLoad Error: cannot load vxworks image from tffs file system: errno = 0x380003.
Trying load image from network ...
netLoad-NetLoading...
1049840
Starting at 0x80004000...

sysNvRamGet: 0 ff 10000
armGetMacAddr:mac addr=12:87:59:e3:9b:06
Attached TCP/IP interface to atemac unit 0
Attaching interface lo0...done
Unable to add route to 192.168.1.0; errno = 0xffffffff.

Adding 3456 symbols for standalone.



->

注意：以上 usrRoot 函数是经过预处理后的输出，对于 usrConfig.c 文件中定义的原函数，其要比此处复杂的多。usrRoot 函数执行完毕即表示 Vxworks 操作系统进入正常运行状态。一般我们需要在操作系统启动后运行用户应用程序，Vxworks 提供这样一种机制如下（如下代码被放置在 usrRoot 函数的最尾端，用以在 Vxworks 操作系统启动完成后，启动用户任务）：

```
#ifdef INCLUDE_USER_APPL
    /* Startup the user's application */
    USER_APPL_INIT;     /* must be a valid C statement or block */
#endif
```

用户可以通过定义 INCLUDE_USER_APPL 宏，并将 USER_APPL_INIT 定义为用户程序入口函数即可。如下示例。

```
/*config.h*/
#define INCLUDE_USER_APPL
#define USER_APPL_INIT          (startUsrRoutine)
```

用户可以在 startUsrRoutine 自定义函数中创建用户任务，实现用户需要的功能。这种工作方

式在操作系统启动过程中启动用户层任务。一般嵌入式系统中都需要使用USER_APPL_INIT 宏启动用户任务。

3.4 BSP 文件组成

BSP 是 Board Support Packet，即板级支持包的简称，组成上主要由平台初始化代码文件以及外设驱动文件构成。当然这主要是从 BSP 开发人员实际所需工作的角度而言的。实际上，BSP 开发人员真正要完成的是 Vxworks 操作系统向特定平台的移植工作。这些工作除了包含平台初始化代码和外设驱动代码的编写外，还需要关心其他一些列文件并理解这些文件的功能，如此当出现相关问题时，可以快速的定位问题所在。本节从 Vxworks 操作系统移植的角度详细介绍一个典型 BSP 的文件构成以及文件具体的功能。

一个典型 BSP 的文件组成有如下类型：

1. 源文件
2. 头文件
3. Makefile 文件
4. 扩展文件（编译过程中生成的文件，如.o 文件）
5. 说明文件（可选）

其中源文件和头文件主要存在 target/config 目录下，这是源文件和头文件包括内核提供的以及 BSP 开发人员提供的文件；Makefile 文件指 target/config/<bspName>以及 target/h/make 目录下的文件，<bspName>下的 Makefile 是属于某个特定 BSP 工程的文件，而 target/h/make 目录下的相关文件被所有 BSP 工程使用；扩展文件指编译过程中生成的文件，这部分文件主要存在于<bspName>目录下；说明文件指存在于<bspName>目录下说明 BSP 信息的相关文件。

对于源文件和头文件需要另外说明的一点是，除了 target/config 目录下的文件，内核提供（安装 Tornado 开发环境时安装的文件）的源文件主要存在于 target/src 目录下，而内核头文件则存在于 target/h 目录下。BSP 开发人员在进行相关代码的查看时，可以到这些目录下去寻找。注意：禁止对内核源文件和头文件进行修改。BSP 开发主要是修改和完善 target/config 目录下相关源文件和头文件。

安装 Tornado 开发环境时，BSP 是作为一个产品独立进行安装的，这是由 Wind River 公司针对一些平台提供的官方 BSP，如 integrator946es，前文中<bspName>就表示一个特定 BSP 的目录名称，可以使用任何对于用户有意义的名称作为<bspName>。

源文件

Tornado 开发环境安装完成后，在 target/config 目录下就包含官方提供的一些 BSP 以及共用目录 All，BSP 开发人员一般复制一个近似平台的 BSP，对其中文件进行修改以及添加一些文件来开始 BSP 的开发过程，不要从零开始创建 BSP，如此可以大大减少 BSP 的开发时间和难度。target/config/all 目录被 config 目录下所有 BSP 使用，这个目录下的文件被所有 BSP 使用，所以不要直接修改其中的文件，下文将介绍在修改 all 目录中文件时如何进行处理。target/config/all 目录下定义有如下源文件，所有这些都是一个 BSP 的组成部分。

bootConfig.c
只被 bootrom 使用，Vxworks 内核则使用 usrConfig.c 文件。

usrConfig.c
只被 Vxworks 内核使用，bootrom 则使用 bootConfig.c 文件。
注意：bootConfig.c 和 usrConfig.c 文件中定义的函数基本类似，如 usrInit，usrRoot 等函数。有关各函数在初始化过程中被调用的时机，请参考本书前面章节内容。

bootInit.c
其中包含 romStart 函数定义，在函数被 romInit 函数调用，负责将代码从 ROM 拷贝到 RAM 中，对于压缩映像，在拷贝过程中，其还将完成解压缩的工作。

version.c
版本信息文件，使用_DATA_和_TIME_宏来表示映像的创建时间，这些信息将在内核启动过程中被显示。

dataSegPad.c
当使用 vxVMI 组件时，该文件中代码确保数据段和代码段使用不同的物理页面，避免共享同一个页面。该文件实现的功能在操作系统运行过程中起作用，在启动过程中不会用到该功能。

configAll.h
该头文件定义了一系列系统默认组件和系统常量。用户可以打开 configAll.h 文件具体查看一下即可明白该文件的意义。注意：禁止修改该头文件，用户可以在 BSP 目录下的 config.h 文件中对 configAll.h 文件中的一些默认设置进行修改。

console.c
usrShell.c
终端交互功能支持文件。

再次提请注意 all 目录下定义的以上这些文件被所有 BSP 共享，且每个文件都是不可缺少的，该目录下所有文件都是由开发环境提供的，所以无需用户编写，但是在某些条件下，用户可能需要修改这些文件中的某些代码，以显示针对某个特定 BSP 的某种机制或功能。Vxworks 提供了这种修改机制，而同时又不影响其他 BSP。如用户需要修改 usrConfig.c 文件中的代码，此时从 all 目录下拷贝一份 usrConfig.c 到 BSP 目录下（如 integrator946es），并在 BSP 的 Makefile 文件中，添加如下宏定义 BOOTCONFIG=./usrConfig.c，此时 Tornado 编译环境将使用当前 BSP 目录下的 usrConfig.c 文件作为 all 目录下 usrConfig.c 文件的替代，即 all 目录下 usrConfig.c 文件将被不再被这个 BSP 使用。当然用户可以在复制过程中修改文件名，如将 usrConfig.c 改为 myUsrConfig.c，那么相应的宏定义就为 BOOTCONFIG=./myUsrConfig.c，只要维持 BOOTCONFIG 宏名称不变，至于指定的文件名称则不作要求，即不一定非要命名为‘usrConfig.c’。
对于 all 目录下其他文件的修改类似，如对应 bootConfig.c 文件的 BOOTCONFIG 宏定义，对应 bootInit.c 文件的 BOOTINIT 宏定义。一般而言，只需要对这三个文件进行修改。

除了 all 目录下共用文件外，Tornado 编译环境还对 BSP 目录下的文件提出了要求，即对于某些文件包括文件名以及文件中定义的函数都做了要求，这些文件有三个。

romInit.s

该文件定义 romInit 函数，该函数是整个系统启动的入口函数，系统上电时，执行的第一行代码就是 romInit 函数实现的第一条语句。基于处理器的特殊需求，一般在 romInit 函数实现的开始处放置一个系统异常表。可以参考 arm 平台下 BSP 的 romInit 函数的实现。该文件完全使用汇编代码编写，文件实现的功能主要完成平台的初始化（如 RAM 控制器初始化，屏蔽系统中断等），此后初始化 C 函数调用环境（主要指栈寄存器的初始化），调用 romStart 函数。romInit.s 文件中一般只定义一个 romInit 函数，基于 romInit 函数特殊的执行环境（即在 ROM 中执行），该函数代码编写必须始终保持 PIC（Position Independent Code），即位置无关。实现 PIC 的根本原因请参考本书前文中“深入理解 bootrom”章节。

该文件被 bootrom 和 ROM 型 Vxworks 内核映像使用。

sysALib.s

该文件中定义了 sysInit 函数，该函数完成的功能类似于 romInit，且 sysALib.s 文件只被下载型 Vxworks 内核映像使用。此时 bootrom 完成 Vxworks 内核映像的下载，所以 romInit 函数已在 bootrom 中被执行，故诸如 RAM 控制器初始化工作无需再在 sysInit 函数中进行，这应该可以说是 sysInit 函数与 romInit 函数实现上的一个区别，且 sysInit 函数是直接运行在 RAM 中的，所以其实现上不需要做到 PIC。这两点区别可以说是二者的所有不同之处，实际上可以将 romInit 函数实现完全复制到 sysInit 函数实现中，当然要删除 RAM 控制器初始化的一段代码，对于 PIC 则无需关心。

sysLib.c

该文件名称必须定义为 sysLib.c，换句话说，Tornado 编译环境要求 BSP 目录下必须有一个文件命名为 sysLib.c，且对该文件中定义的函数进一步作出了要求，即该文件必须实现一些具有指定函数名和指定功能的函数。这些函数如下。

sysBspRev( ) - return the BSP version and revision number

sysClkConnect( ) - connect a routine to the system clock interrupt

sysClkDisable( ) - turn off system clock interrupts

sysClkEnable( ) - turn on system clock interrupts

sysClkInt( ) - handle system clock interrupts (internal)

sysClkRateGet( ) - get the system clock rate

sysClkRateSet( ) - set the system clock rate

sysHwInit( ) - initialize the system hardware

sysHwInit2( ) - initialize additional system hardware

sysMemTop( ) - get the address of the top of logical memory

sysModel( ) - return the model name of the CPU board

sysNvRamGet( ) - get the contents of non-volatile RAM

sysNvRamSet( ) - write to non-volatile RAM

sysSerialHwInit( ) - initialize the BSP serial devices to a quiescent state

sysSerialHwInit2( ) - connect BSP serial device interrupts

sysSerialChanGet( ) - get the SIO_CHAN device associated with a serial channel

sysToMonitor( ) - transfer control to the ROM monitor

除了以上必须的函数外，还有一些函数是可选的，但一旦要在 sysLib.c 中包含这些函数，函数名和实现功能依然是固定的。这些可选函数如下。

sysAbortInt( ) - handle the ABORT button interrupt

sysAuxClkConnect( ) - connect a routine to the auxiliary clock interrupt

sysAuxClkDisable( ) - turn off auxiliary clock interrupts

sysAuxClkEnable( ) - turn on auxiliary clock interrupts

sysAuxClkInt( ) - handle auxiliary clock interrupts

sysAuxClkRateGet( ) - get the auxiliary clock rate

sysAuxClkRateSet( ) - set the auxiliary clock rate

sysPhysMemTop( ) - get the address of the top of physical memory

除了 romInit.s，sysALib.s，sysLib.c 三个文件必须按这些名称命名外，还包括其他可选文件，这些文件的文件名并没有固定要求，但一般都有约定的名称，故也建议用户按约定名称进行命名。这些可选文件如下。

sysSerial.c

串口驱动相关代码。

sysScsi.c

SCSI 驱动相关代码。

sysNet.c

局域网 LAN 驱动相关代码。

以上三个文件中的代码主要在 sysHwInit 和 sysHwInit2 函数执行期间被调用。

这些文件中代码通过在 sysLib.c 文件中包括到内核映像中。

例如针对以上三个文件，可以在 sysLib.c 文件某个位置中添加如下语句：

#include “sysSerial.c”

#include “sysScsi.c”

#include “sysNet.c”

用户也可从此处看出为何不对这些文件的命名加入硬性限制。

除了以上文件外，对于 BSP，当然还需要特定平台上一些外设的驱动文件，这些文件也包含了 BSP 目录下，如网口驱动文件可以命名为 myEndDriver.c，SPI 口驱动文件 mySPIDriver.c，那么此时在 sysLib.c 文件中加入如下语句即可将这些驱动代码加入内核映像中。

#include myEndDriver.c

#include mySPIDriver.c

除了直接在 sysLib.c 文件中加入以上 include 语句外，还可以通过 Makefile 中 MACH_EXTRA 宏定义来将某个文件代码包含到内核映像中。如以上的例子中，还可以通过如下方式将 myEndDriver.c，mySPIDriver.c 文件代码包含到内核映像中。

MACH_EXTRA=myEndDriver.o mySPIDriver.o

头文件

头文件主要有三个：all/configAll.h，<bspName>/config.h，<bspName>/<bsp>.h，当然一些驱动文件还配属有各自的相关寄存器的头文件。

configAll.h 文件是 Tornado 开发环境提供的，禁止对其进行修改。这个头文件定义内核构成的所有默认选项和系统常量。如果需要修改一些默认选项，则通过 config.h 文件进行修改。<bsp>.h 文件则是针对特定平台寄存器地址和结构的相关定义。configAll.h 和<bsp>.h 文件都被 config.h 文件使用：configAll.h 文件被 config.h 包含在最开始处，这一点是必须的，因为只有如此，才能通过 config.h 文件的后续定义改变某些选项定义，<bsp>.h 文件一般则被 config.h 文件包含在最尾处。

三个头文件中，configAll.h 文件禁止修改，<bsp>.h 文件平台相关，只有 config.h 文件具有操作的灵活性。config.h 文件必须包括以下功能或者定义。

1. BSP 版本号以子版本 ID 号。

   可以使用 BSP_VERSION 宏定义 BSP 版本号，如下所示。

   #define BSP_VERSION “1.2”

   子版本号通过 BSP_REV 宏进行定义，如下所示。

   #define BSP_REV “/0”

   注意子版本号定义中包括一个反斜杠，因为最后的发行号（release number）是版本号和子版本号的字符串连接，所以必须加一反斜杠进行隔离。前文中我们讨论到 sysLib.c 文件中必须定义的函数时，其中之一就是 sysBspRev 函数，该函数即返回 BSP 的发行号。其实现如下。

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

   根据以上的定义，其返回“1.2/0”字符串，这正是我们想要的结果。

   注意：BSP 版本号和子版本号必须位于 config.h 文件的最开始处，在包含 configAll.h 文件之前，因为 configAll.h 中某些组件行为可以依赖于版本号。

2. 包括 configAll.h 文件。

   configAll.h 文件定义内核所需的一系列默认组件，特定 BSP 所属的 config.h 文件必须以此为基础构建 Vxworks 内核，故包含 configAll.h 文件是必须的，而且应该放在 config.h 文件的开始处，紧接着版本号定义之后，因为我们需要对一些默认行为进行重新定义，故必须将组建功能的修改宏定义放在 configAll.h 文件之后。

3. 内存 cache 和 MMU 策略（即是否使能系统 cache 以及是否使能 MMU）。

   此处所指内存 cache 策略是整个系统的 cache 策略，是由 CPU 相关寄存器对所有的 cache 一种全局控制。我们知道，MMU 使能后具有面向页面的 cache 策略控制，但是这个控制必须时在全局 cache 使能的情况下，如果全局 cache 不使能，那么无论 MMU 页表项中属性为何，都无法达到 cache 的目的。内存 cache 策略对指令和数据分别进行控制，由 USER_I_CACHE_MODE，USER_D_CACHE_MODE 进行控制。如下所示。

   ```
   #undef    USER_I_CACHE_MODE
   #define USER_I_CACHE_MODE            CACHE_WRITETHROUGH

   #undef    USER_D_CACHE_MODE
   #define USER_D_CACHE_MODE            CACHE_COPYBACK
   ```

由于 configAll.h 文件中已经定义了默认的 cache 行为，故此处为了修改这个默认的行为，首先我们必须取消默认行为（undef 语句），此后重新定义 cache 行为（define 语句），这是 config.h 文件对 configAll.h 文件中默认行为进行修改的通用方式。

MMU 机制的控制通过 INCLUDE_MMU_FULL 和 INCLUDE_MMU_BASIC 两个宏进行控制，值得注意的是不同版本的 Vxworks 内核用以控制相关行为的宏名称并不相同。这一点在开发 BSP 时特别要注意，要弄清楚 BSP 移植所基于的 Vxworks 内核版本。

4. 平台 RAM 基地址和大小。

涉及平台 RAM 的宏定义如下。

LOCAL_MEM_LOCAL_ADRS：平台 RAM 基地址。

LOCAL_MEM_SIZE：平台 RAM 总大小。

USER_RESERVED_MEM：在 RAM 顶部用户预留的内存大小，这部分内存将排除在 Vxworks 内核的管理之外，即 Vxworks 内核将“看不到”这部分内存空间。

RAM_HIGH_ADRS：bootrom 压缩代码拷贝到 RAM 中的目的地址。当非压缩形式时，bootrom 映像所有代码将拷贝到 RAM_HIGH_ADRS 指定的地址。

RAM_LOW_ADRS：Vxworks 压缩代码拷贝到 RAM 中的目的地址。当非压缩形式时，Vxworks 映像的所有代码（即包括 romInit，romStart 函数）都将被拷贝到 RAM_LOW_ADRS 指定的 RAM 地址处。

注意：有关 RAM_HIGH_ADRS，RAM_LOW_ADRS 的具体使用请参考本书前文中“映像类型”章节的说明，此处的叙述不甚准确。

另外一个宏用以支持 RAM 容量的动态改变，一般我们不使用这种工作方式，这个宏为 LOCAL_MEM_AUTOSIZE。感兴趣用户可以参考文献“vxworks bsp programmer’s guide”。

5. 平台 ROM 基地址和大小。

当采取 bootrom + Vxworks 下载启动方式时，bootrom 将被烧录到 ROM 中；或者采用 ROM 型 Vxworks 内核直接启动方式时，Vxworks 内核映像本身（一般为压缩版本）将被直接烧录到 ROM 中。可以说，ROM 是系统上电后跳转的目的地址，其中存储着上电后运行的第一条指令。基本上所有平台都必须存在某种形式的 ROM 存储介质。ROM 是“Read Only Memory”，即只读存储器的简称，但是现今平台上基本上都在使用 Flash 替代 ROM，一方面由于 Flash 的存储容量大，另一方面成本也比较低。当然 Flash 从工作方式上主要分为两大类：NorFlash 和 NandFlash。为了能在其中在线执行 EIP（Execute In Place），都是用 NorFlash 作为启动介质（虽然现在有号称直接从 NandFlash 启动，但是都必须借助于中间 bootloader 代码，并非完全意义上的 EIP，事实上，NandFlash 不可能达到真正意义上的 EIP 的）。NorFlash 支持重新擦除后编程，编程次数可达 10 万次。故使用 NorFlash 作为启动介质较为普遍，反而真正意义上的 ROM 基本已经退出开发阶段，可能在产品发布时，为了进一步降低成本，会使用 ROM。虽然使用的是 Flash，但是在定义上还是被看做是 ROM，涉及到的相关宏如下。

ROM_BASE_ADRS：ROM 基地址。

ROM_TEXT_ADRS：存放启动代码的首地址。一般 ROM_TEXT_ADRS 会相对 ROM_BASE_ADRS 做一个偏移，以存放一些参数（如 Bootline）。但是很多平台直接将 ROM_TEXT_ADRS 定义为 ROM_BASE_ADRS 的值。

ROM_SIZE：ROM 大小。

ROM_COPY_SIZE：当代码非压缩时，所有代码将一次性被拷贝到 RAM 中，ROM_COPY_SIZE 即指定拷贝代码的大小。一般将这个值直接设置为 ROM_SIZE，虽然在拷贝时会多做一些无用功，但是较为方便，而且对于拷贝代码的大小会随着代码的改变而作改变，经常性修改这个参数也比较厌烦，而且一旦忘记时，如果实际代码大小大于此处定义的拷贝大小，还会出一些诡异的错误，故还是建议用户将其直接设置为 ROM_SIZE。

如下给出了以上宏定义的简单示例。

```
#define ROM_BASE_ADRS       0x02000000      /* base of Flash/EPROM */
#define ROM_TEXT_ADRS       ROM_BASE_ADRS   /* code start addr in ROM */
#define ROM_SIZE            0x00200000      /* size of ROM holding VxWorks*/
#define ROM_COPY_SIZE       ROM_SIZE
```

6. 非易失性 RAM（NVRAM）大小。NVRAM 一般容量较小或者直接从存储启动代码的 Flash 中预留一段空间，用以存储诸如 Bootline 之类的参数。Vxworks 启动过程中，有一个倒计时的过程，在这个过程中，用户如按下键盘上任何一个按键，就会进入到 bootline 修改模式，用户可以动态修改 bootline 的参数值，如可以修改启动方式，在下载启动中，可以修改下载文件名，主机 IP，目标机 IP 等。修改后的参数一般被写入 NVRAM 中，这样就可以保存本次修改，而当下次启动时，将使用修改后的参数进行启动。涉及到 NVRAM 的相关宏如下。

   NV_RAM_SIZE：NVRAM 大小。如果平台不包括 NVRAM，则将这个宏定义为 NONE。

   BOOT_LINE_SIZE：NVRAM 中预留给 Bootline 参数的存储空间，这个预留值不可以大于 NVRAM 本身的大小 NV_RAM_SIZE。这个参数在 configAll.h 中的默认值被设置为 255 个字节。

   NV_BOOT_OFFSET：这个常量定义了 Bootline 参数存储在 NVRAM 中相对于 NVRAM 基地址的偏移量。这个参数将被 sysNvRamGet 和 sysNvRamSet 函数使用用以获取和修改 Bootline 参数。

   注意：以上并没有表示 NVRAM 基地址的宏。因为实际平台上基本没有单独使用一片存储介质去存储参数，大多是从启动代码存储介质中（如使用较多的 NorFlash 中）预留一段空间供 Bootline 参数使用，故并没有一个内核特定常量来表示 NVRAM 基地址，这个基地址完全由用户决定，用户可以定义一个 NV_RAM_ADRS 常量来表示 NVRAM 的基地址，这个基地址和 NV_BOOT_OFFSET 以及 BOOT_LINE_SIZE 将被 sysNvRamGet 和 sysNvRamSet 两个函数使用。

7. 默认 Bootline 参数定义。

   Bootline 定义了 Vxworks 启动过程中的一些参数，如目标板的 IP 地址；如果使用下载启动方式，则定义了下载源，下载文件名等等。当然，在 Bootline 中还可以定义一些平台特定参数，用户可以在 bootConfig.c 或者 usrConfig.c 文件中对 Bootline 中自定义参数进行解析，已决定启动过程中的某些自定义行为。内核提供了 bootStringToStruct 函数将 Bootline 将特定格式进行解析并返回解析后的数据结构，这个结构中的字段将决定内核启动行为。

一个实际应用中的 BSP Bootline 定义，如下所示。

#define DEFAULT_BOOT_LINE "atemac(0,0) vxw:vxWorks h=192.168.1.6 e=192.168.1.254:ffffff00 g=192.168.1.1 u=www pw=www tn=AT91RM9200 o=00:00:0a:0b:0d:00"

其中定义了启动方式是通过 EMAC 网口启动，Vxworks 内核映像名为 vxWorks，映像所在主机 IP 地址为 192.168.1.6，要求目标机 IP 地址设置为 192.168.1.254，子网掩码设置为 255.255.255.0，网关 IP 地址为 192.168.1.1，用户名和密码都是 www，要求目标机网卡 MAC 地址设置为 00：00：0a：0b：0d：00.

8. 其他对默认组件的宏定义以修改 configAll.h 文件中定义的默认选项。
   默认行为的修改由两条语句完成：undef 和 define。undef 取消组件默认行为，define 重新定义组件行为。

9. bspname.h 文件，即特定平台外设寄存器结构和地址定义，外设中断号分配等等平台特定常量定义。

Makefile 文件

BSP 目录下 Makefile 文件决定了 Vxworks 和 Bootrom 的映像类型以及对一些参数的定义，这个 Makefile 是一个上层封装文件，其底层将调用 Tornado 开发环境提供的 Makefile 文件，这些文件存在于 h/make 目录下，主要有两个：rules.bsp 和 defs.bsp。rules.bsp 文件定义了针对各种不同映像类型的编译规则，以及 BSP 中模块的编译规则；defs.bsp 文件定义了控制编译行为的一些标志位和常量的定义。h/make 目录下其他一些文件定义了针对不同特定平台的编译时参数和标志位。

如下给出了一个实际 BSP 的 Makefile 文件，这个 BSP 已经完成移植且运行稳定。

```
CPU         = ARMARCH5
TOOL            = gnu
EXTRA_DEFINE    = -Wcomment -DCPU_926E \
            -DARMMMU=ARMMMU_926E -DARMCACHE=ARMCACHE_926E

TGT_DIR = $(WIND_BASE)/target

include $(TGT_DIR)/h/make/defs.bsp
include $(TGT_DIR)/h/make/make.$(CPU)$(TOOL)
include $(TGT_DIR)/h/make/defs.$(WIND_HOST_TYPE)

## Only redefine make definitions below this point, or your definitions will
## be overwritten by the makefile stubs above.

TARGET_DIR  = arm926ejs_bsp
VENDOR      = HFei
BOARD       = seed dm6446

MACH_EXTRA = Dm644x.o
```

```
#BOOT_EXTRA    =


RELEASE         +=    bootrom.bin

#
# The constants ROM_TEXT_ADRS, ROM_SIZE, and RAM_HIGH_ADRS are defined
# in config.h and Makefile.
# All definitions for these constants must be identical.
#

ROM_TEXT_ADRS    = 0x02000000 # ROM entry address
ROM_SIZE         = 0x00200000 # 16M number of bytes of ROM space

RAM_LOW_ADRS     = 0x80004000 # 32M RAM text/data address
RAM_HIGH_ADRS    = 0x86000000 # RAM text/data address

VMA_START     = 0x$(ROM_TEXT_ADRS)

# Binary version of VxWorks ROM images, suitable for programming
# into Flash using tools provided by ARM.    If other ROM images need to
# be put into Flash, add similar rules here.

bootrom.bin: bootrom
        - @ $(RM) $@
        $(EXTRACT_BIN) -O binary bootrom $@

bootrom_res.bin: bootrom_res
        - @ $(RM) $@
        $(EXTRACT_BIN) -O binary bootrom_res $@

bootrom_uncmp.bin: bootrom_uncmp
        - @ $(RM) $@
        $(EXTRACT_BIN) -O binary bootrom_uncmp $@



vxWorks_rom.bin: vxWorks_rom
        - @ $(RM) $@
        $(EXTRACT_BIN) -O binary vxWorks_rom $@

vxWorks.st_rom.bin: vxWorks.st_rom
        - @ $(RM) $@
        $(EXTRACT_BIN) -O binary vxWorks.st_rom $@
```

```
vxWorks.res_rom.bin: vxWorks.res_rom
    - @ $(RM) $@
    $(EXTRACT_BIN) -O binary vxWorks.res_rom $@

vxWorks.res_rom_nosym.bin: vxWorks.res_rom_nosym
    - @ $(RM) $@
    $(EXTRACT_BIN) -O binary vxWorks.res_rom_nosym $@

## Only redefine make definitions above this point, or the expansion of
## makefile target dependencies may be incorrect.

include $(TGT_DIR)/h/make/rules.bsp
include $(TGT_DIR)/h/make/rules.$(WIND_HOST_TYPE)
```

Makefile 文件中必须定义如下宏：
CPU：平台 CPU 架构，如上定义为 ARMARCH5，即 ARM V5.
TOOL：主机编译工具链。
TARGET_DIR：BSP 目录名。
VENDOR：生产商。
BOARD：平台代号。
ROM_TEXT_ADRS：启动代码存储介质基地址。
ROM_WARM_ADRS：热启动基地址，这个地址相对于ROM_TEXT_ADRS将存在一个小的偏移。热启动时将保留当前RAM中的所有内容。
ROM_SIZE：ROM容量。
RAM_LOW_ADRS
RAM_HIGH_ADRS
RAM_LOW_ADRS 和 RAM_HIGH_ADRS 两个常量定义启动过程中从 ROM 向 RAM 拷贝时的 RAM 目的地址，具体含义参考本书前文中“映像类型”章节。

注意：Makefile 文件中定义的一些常量如 ROM_TEXT_ADRS, ROM_SIZE, RAM_LOW_ADRS, RAM_HIGH_ADRS 在 config.h 文件中也有相同的常量定义，两个文件中对于这些常量必须定义为相同的值！这一点非常重要。某些用户可能认为只需在一个文件中进行定义即可，两个文件中进行定义岂不是重复？事实上并非如此，两个文件中定义的宏被不同的部分使用，config.h 文件中的宏定义被内核组件使用控制内核的某些行为或者初始化内核的某些变量，而 Makefile 中的定义则控制编译过程，如链接地址。所以对于以上重合的常量定义并非有意重复，而是使用在不同的地方，由于最后在运行时都要汇集在一起，所以两个文件中相同的常量必须保证相同的值！

扩展文件
这些文件主要存在于 BSP 目录下，如果使用 Tornado 开发环境，则存在于工程目录 default 子目录下。扩展文件主要是在编译过程中生成的一些中间目标文件，如 bootInit.o 之类，以及动态创建的文件，如符号表文件 symTbl.c，以及最终的编译得到的链接文件，如 bootrom，vxworks 映像。用户一般不用关心具体这些扩展文件，本书前文“深入理解 bootrom”一节较为详细介绍了 bootrom 的生成过程，用户可以借此对这些文件进行理解，另外感兴趣用户

可以参见文献“vxworks bsp developer’s guide”2.2.2 节内容了解扩展文件方面的内容，此处不再讨论。

说明文件

这些文件存在于 BSP 目录下，对一个 BSP 的相关信息进行介绍，主要有如下两个文件：README 和 target.nr。其中 README 文件包含了 BSP 的发行纪录，每次发行的版本号；target.nr 文件包含了运行 Vxworks 内核映像的平台特定信息，这些信息包括：

平台名称；平台简要介绍；平台支持的功能；平台可用硬件资源；平台布局（如跳线器等信息）；参考资源信息等等。

说明文件在将一个 BSP 作为产品发布时是必要的，一般而言，大多数情况下，开发人员开发针对特定平台的 BSP，用于公司某个产品内部，整个产品作为一个整体发布，并不单独发布 BSP（BSP 只对公司内部可见），故对于说明文件要求各异，且说明文件格式并没有特殊要求，故此处只做简单说明。

第四章 驱动程序概述

驱动程序简单的说就是设置某个硬件完成其固有功能的程序。如网卡设备驱动程序就是设置网卡相关寄存器使其能够正常的收发网络数据包的程序。驱动程序直接与硬件设备交互，其大多数的工作就是操作硬件相关寄存器。

首先寄存器也是一种 RAM，在系统下电后，寄存器中的内容都会丢失，系统上电复位过程中，硬件寄存器一般都复位到一个默认值，默认状态下硬件是不能正常工作的，如中断使能被屏蔽，工作使能位也被屏蔽，还有一些决定硬件工作情况的关键控制寄存器也需要被重新配置。而这些工作都有赖于设备驱动完成。驱动一般都作为操作系统内核组成的一部分，即便现在很多系统支持驱动的动态加载，但是驱动代码在执行时，依然是以内核代码模式进行执行的，换句话说，驱动代码具有系统特权级，除了其自身资源，对应硬件设备资源，其还对操作系统资源具有完全的访问权。所以一个驱动程序如果存在 BUG，将直接会导致整个操作系统的崩溃。故调试驱动是一项十分关键的工作，必须对驱动进行仔细检查，并需要经受长时间运行考验。应用层程序员往往对属于内核编程的外设驱动心存敬畏，认为驱动编程是一项非常复杂的工作。实际上，底层驱动编程往往比应用层编程具有更大的灵活性，就如没有调试不出来的硬件，也没有调试不出来的底层驱动，但是应用层 BUG 有时就是无法调试出来。底层驱动的调试过程是同时对硬件和驱动进行验证的过程。底层驱动很多时候用来定位硬件设计错误或者硬件芯片本身可能的问题，故底层驱动程序员必须对所要驱动的硬件设备有一个比较充分的了解，以及对与硬件交互的其他硬件或外界环境也需要有一个比较清楚的理解。

驱动程序对上需要匹配操作系统提供的一套规范接口，对下必须驱动硬件设备进行工作，其起着一个关键的中间转换角色，将操作系统的具体请求转换为对硬件的某种操作。驱动程序在操作系统中扮演着一个非常特殊的角色，其类似一个黑盒子，让所有的硬件对操作系统的一套内部规范接口进行响应，驱动程序屏蔽了硬件的所有复杂性，应用层对于某个设备的操作通过操作系统提供的一套标准接口完成，操作系统最终将这些操作请求传递给驱动程序，驱动硬件完成这些请求。

Vxworks BSP 移植的主要工作就是编写平台特定硬件设备的驱动程序，可以说驱动程序编写能力对于 BSP 的移植具有重要影响。本书从此章开始详细介绍 Vxworks 下各种外设的驱动程序基本设计原理，框架及内核相关代码，使读者能够对 Vxworks 下驱动程序的编写做到胸有成竹。在各种类型外设驱动介绍中，尽量在交代清楚驱动结构和内核接口之后，借助具体的例子帮助读者理解，但是基于每种设备的一个实际驱动代码动辄上千行（网口驱动可达 3 千多行），且无法对一个具体设备交代其所有方面，故只能以一种代码片段示例的方式进行讲述，着重加深读者对于某些概念的理解。

了解一个操作系统最好的方式不是一上去就去阅读操作系统源代码，对于一个初学者而言，操作系统的庞大和复杂会立刻消磨掉初学的锐气。在这个操作系统下编写设备驱动应该是最直接最有效的方式。编写驱动可以让你直接对操作系统的某个子系统有详细的了解，明白为何代码并不像你想象的那么工作，当你完全调通一个驱动（无论多么简单），你都会对原先学习一些理论和概念有一个全新的认识。

4.1 设备驱动功能

作为一个驱动程序员，你在驱动编写时间和驱动所提供的灵活性之间有很大的决定权进行选择。一个驱动可以设计为只提供基本的功能或者实现一种非常灵活的底层机制供用户使用，完全由驱动程序员自己决定。值得提出的是，驱动仅仅提供了一种机制，如何使用这个驱动则是用户的策略。驱动程序中包括着主动和被动两个方面的因素，其主动性体现在对所驱动硬件上，一个典型的例子就是网卡驱动，其平时不间断的从网卡主动接收数据包，而不需要用户的任何干预；被动性则体现在对上层用户的响应上，如果用户不提出请求，底层驱动是不会主动进行服务的，如网卡驱动在接收网络数据包后只是将数据包交给操作系统本身进行管理，而并非交给某个用户。只当用户进行数据请求时，这些数据才递交给用户。底层驱动这两个方面的表现正好体现了驱动仅仅是提供了一种服务机制，而如何使用这种机制就完全由用户决定。

驱动程序员在编写底层驱动代码时，必须谨记的一点是：不可以对用户使用该驱动强加某种前提条件。换句话说，不能将驱动设计为将硬件固定为某种工作方式，用户只能通过该方式进行服务请求。驱动仅仅是驱动硬件工作，至于如何工作则应该完全交由用户决定。我们将这种驱动称为具有灵活性的驱动程序。

诚如前文所述，驱动程序是位于用户和硬件之间一个软件层，驱动程序员有完全的决定权决定一个硬件设备以何种形式呈现给用户，不同的驱动程序可以使得同一个硬件设备以不同的方式对用户可见。我们可以将一个实际块设备以字符设备对用户可见，将一个实际 Flash 设备以硬盘设备对用户可见，以何种形式表现一个实际设备完全由底层驱动控制。驱动程序员可以提供一系列方式让用户对设备进行控制，甚至可以让用户直接操作硬件设备的每一个寄存器，由用户在寄存器层次对硬件设备进行操作；或者只提供一个读或写操作，屏蔽其他所有操作等等。

故从宏观角度而言，驱动程序实现的功能即提供一种底层服务机制供用户进行选择。从微观角度而言，驱动程序需要对下配置硬件寄存器，完成对设备数据的读写，对设备本身的控制，对上使得设备能够响应用户的服务请求，这些服务请求如下：打开设备；读写设备；控制设备；关闭设备。

4.2 设备驱动结构

可以说，所有的设备驱动都具有相同的结构组织。这或许可以从用户层进行理解：所有的设备在用户层都以同一套标准的接口对用户可见。这些标准的用户请求经过层层传递，最后到达底层驱动，对于每个标准请求，底层驱动都必须有一个对应的响应函数，这就决定了基本所有的底层驱动都具有相同的结构。然而对用户请求进行响应并非驱动全部，这仅仅构成底层驱动被动的一面，其主动的一面则体现在对硬件的操作上。设备驱动被动和主动两个方面的特性决定了设备驱动基本的架构：对上提供一套内核标准接口响应函数，对下提供硬件驱动和硬件中断响应函数。

为了实现用户层调用的标准化，操作系统对每种类型设备的驱动都有一套标准的接口函数，底层驱动必须实现这些函数并将这些函数的地址注册给操作系统，如此当用户请求某项服务时，操作系统可以调用底层驱动对应的函数进行响应。此时操作系统作为主动方主动调用底层驱动提供的函数，以达到用户与底层驱动交互。另一方面为了与实际硬件设备交互，底层

驱动也需要一种通知机制，这种机制通常就是中断。一个用户请求到来时，底层驱动某个函数被操作系统调用进行响应，该函数配置硬件相关寄存器服务该请求。硬件完成该请求后，必须反过来通知底层驱动表示服务已经完成。硬件通过的方式一般就是中断。故底层驱动必须有一个针对驱动硬件中断发生的中断响应函数。底层驱动中断响应函数一旦被调用，就表示用户请求的某项服务已经完成，大多数时候，这个完成的状态需要通过给用户，故从这个方面来看，内核本身也需要提供一个回调函数给底层驱动，从而完成底层驱动向用户的主动通知，用户在得到通过可以采取下一步的操作。由此我们可以总结设备驱动的通用架构：

1. 一套需要提供给操作系统的标准操作函数。
2. 一个硬件中断响应函数。
3. 一个（或一套）由内核提供给底层驱动的回调函数指针，用以存储内核回调函数的地址。
4. 设备驱动注册和初始化函数。

以上四点基本构成了所有设备驱动的结构组织。至于对前三个方面如何进行封装则将根据具体的设备类型决定。对于第一个方面底层驱动提供给操作系统的标准操作函数，我们可以进一步的细化，细化的基本依据就是用户层标准接口。这些函数包括：

A． 设备打开函数：完成设备的初始化，注册底层驱动硬件中断响应函数，使能设备工作。
B．设备读写函数：完成与设备数据的交互。
C．设备关闭函数：禁止设备工作，注销底层驱动硬件中断响应函数。
D． 设备控制函数：对设备的工作方式进行控制。

以上讨论中将设备的初始化作为设备打开函数实现的一部分，这通常不准确的。底层驱动是作为操作系统映像的一部分而存在，即其属于内核代码，设备初始化通常在操作系统启动过程中完成。换句话说，设备初始化将作为一个独立的函数被实现，在操作系统启动过程中作为启动的一部分为调用。此时设备打开函数实现中将剔除设备初始化的代码，一般只需要完成中断函数的注册和设备的使能。另很多开发者将中断注册也作为设备初始化的一部分，这并非是一个错误，但基于现在中断号一般都是共享的，故推荐只当设备真正工作时，才注册中断处理函数。设备打开函数中注册中断基本上已经成为一个约定。

另外以上列出的四个函数需要注册给内核，这个注册的过程以及设备节点的创建通常也是在设备初始化函数中完成的。这样操作系统启动完成后，用户才有机会通过这些函数对设备进行操作。所以基于这些考虑，我们还需要为设备驱动通用架构再加上一条：即设备驱动注册和初始化函数，这即是上面第 4 点的由来。

4.3 设备驱动相关方面

4.3.1 驱动代码执行环境

设备驱动代码是执行在内核环境下的这是笼统的说法。实际上驱动代码具有两个执行环境。一个是任务上下文，一个是中断上下文。除了中断响应函数执行在中断上下文之外，其他所有函数均执行在任务上下文。这个任务上下文对应的任务就是当前对设备进行某种服务请求的用户任务或内核任务。在 Vxworks 下处于一个任务的上下文中仅仅是指在执行驱动代码

时可以被挂起，代码使用的栈是任务栈，代码可被中断抢断，包括其自身中断。所以如果驱动中某个函数与驱动中断处理程序共享同一资源，则要避免形成资源的破坏，在这种情况下大多使用 intLock+taskLock 组合保护任务上下文。内核提供的回调函数虽然不是驱动自身的代码，但是也是在驱动中被调用，这些回调函数一般由底层驱动中断响应函数进行调用，故是执行在中断上下文中。内核在实现这些回调函数时已经注意到这一点，故在中断上下文中执行不会造成问题。关于内核回调函数具体实例读者可参考本书中“串口驱动”一章。

4.3.2 设备类型

根据设备的工作方式和数据存储或者来源不同，可以将设备分为三大类型。

A． 字符设备类型

字符设备即以字节流的方式被访问，就如同一个文件，但不同于文件之处是字符设备一般不可以移动文件偏移指针，而只能顺序的访问数据。终端设备以及串口设备都属于字符设备类型。字符设备驱动至少需要实现 open，close，read 和 write 四个系统调用底层实现函数。

B．块设备类型

块设备一般通过文件系统访问。而块设备的最多使用方式也是文件方式。块设备一般不能对单个字节进行访问，而是一个块的方式（如硬盘以一个扇区（512B）为单元进行访问）进行。块设备允许同一数据的反复读取和写入。最典型的块设备就是硬盘，Flash 设备也是一种块设备。

C．网络设备类型

这是比较特殊的一类设备，这类设备用于与网络上其他主机进行通信。其数据读取方式有些类似于字符设备，不可以对同一数据进行反复读写，只能顺序读写数据。且该类型设备区别于字符设备和块设备的一个很大的不同是，其不提供文件节点，任务要访问一个网络设备必须使用另一套网络套接字接口函数进行，与文件系统则完全不相关。网络设备底层数据传输上以块的方式进行，但是又不同于块设备中数据块的概念，网络设备中块的大小可以改变，但是有一个区间范围。

以上只是设备类型的划分方式之一，事实上，按以上的划分标准，某些设备接口在某些情况下可以表现为任意以上三种形式之一，如 USB 接口，可以是一个字符设备，如 USB 串口；也可以是一个块设备，如 USB 内存卡；也可以是一个网络设备，如 USB 网络接口。

4.3.3 安全性

Vxworks 不支持驱动程序的动态加载，驱动程序一般实现为内核代码的一部分，故安全性方面的问题出现较少。但是一些编程上应遵循的基本安全规则必须谨记：对于用户输入的任何参数都必须经过检查后方可使用，且对于不合法或不合理的参数，必须终止服务。

在某些特殊情况下，对于一个外设硬件的控制可能通过一个用户层任务进行，此时可以认为这个驱动实现在用户层，此时就需要特别注意。对于 Vxworks 这样一个不区分运行级别的

操作系统而言，不存在通用操作系统下的用户层和内核层的概念，通常我们在 Vxworks 讲到用户层，仅仅是指任务运行的代码是用户代码，而非内核代码，且仅此而已。事实上，在 Vxworks 下，用户可以直接调用任意的内核函数而不会出现调用权限不够的情况，这主要是 Vxworks 操作系统应用环境决定的。故安全性在 Vxworks 下更多的是指尽量排除代码中固有的 BUG，而不是防止某些用户恶意的行为。

4.3.4 驱动工作模式

设备驱动从总体上分为两种工作模式：轮询模式和中断模式。轮询模式通过检测相关寄存器状态位来决定是否进行下一步操作；而中断则用以通知某个操作已然完成，可以进行下一个操作了。二者的根本区别在于轮询方式在等待操作完成的过程中需要 CPU 等待，而中断则将 CPU 从等待中解放出来，在硬件完成一个用户请求的过程中，CPU 可以运行其他任务，当硬件完成请求后，发出一个中断，再次引起驱动的注意，驱动可以在中断响应函数中启动下一个操作。从 CPU 使用效率而言，好像中断方式一定优于轮询方式，其实不然。轮询方式被使用在很多场合，即便这些场合支持中断模式。如串口，SPI 口等速率较低的设备，实际工作中很多都是通过查询状态位来进行读写的。以串口为例，其支持中断工作模式，但是使用中断有如下缺点，如果没发送或者接收一个字节就产生一个中断，由于中断响应需要消耗资源，频繁的中断不断不能加快数据收发速率，而且会极大的影响着整个系统的性能。为了更好的服务中断，现在串口设备大多内部集成 FIFO，可以设置 FIFO 级别，控制中断发生时 FIFO 的空闲度或者占用度，将串口从单个字节的中断中解除出来，但是这种 FIFO 一般容量较小（如常见的 16 个字节），但串口进行大量数据传输，中断频率依然较大，但相比查询方式而言，效率还是有很大的提高；其次，对于小量数据，由于没有达到中断产生的字节数，这些数据一直无法提供给驱动，造成数据的延迟，在某些情况下这是不可容忍的。所以需要根据实际情况选择工作模式。通常情况下，串口只是用于少量信息的现实，Vxworks 下 shell 通常也是建立在串口之上，所以需要使用串口传递信息量较小的命令，此时使用 FIFO 就有些不合适，故大多时，串口都工作在轮询模式下。以发送为例，每次驱动将一个字节的数据写入串口数据发送寄存器后，就不断的检查状态寄存器，查看这个字节是否被发送出去，如果没有则不断的进行查询，一旦查询得到这个字节已成功发送，则取下一个待发送的字节进行发送，直到当前内核串口缓存中所有字节均已发送完毕。这种方式可以将写入串口的数据及时的通过串口打印出来。只不过在打印之时会占用 CPU 资源，不过从宏观上来看，并不会对整个系统的性能造成影响，顶多对进行打印的任务造成一些延迟。
所以从表面上看，中断好像优于轮询模式，这只是单从 CPU 使用的角度考虑问题的，从实际工作情况出发，二者使用的范围都比较广。可以做如下总结：中断使用在数据量较大的场合；轮询使用在数据量较小的场合。

4.3.5 与硬件数据交互方式

驱动与硬件之间的数据交互方式主要分为两中方式：DMA 方式和直接拷贝方式。DMA 方式将 CPU 从数据拷贝中解放出来，数据拷贝由 DMA 控制器专门负责，最典型的使用实例即硬盘数据拷贝。DMA 方式不能单独工作，一定要借助于中断，所以 DMA 方式一般使用在大批量数据拷贝上，即便数据输入输出速率很大的网口设备也较少使用 DMA 方式，而是

驱动直接从网卡设备硬件缓冲区中直接拷贝网络数据包。DMA 方式在嵌入式系统下使用的概率较小，因为嵌入式系统很少有机会进行大批量数据传输，而且 DMA 方式从创建数据拷贝环境到数据拷贝完成到 CPU 的后续处理都具有较大的延迟，不利用对速度要求较高的场合。DMA 最大的好处是在数据拷贝过程中无需 CPU 的干预，CPU 可以独立出来运行其他任务，然而在很多场合，如果数据没有完成拷贝，CPU 也没有其他什么“活”可以做，这一点在嵌入式系统下就尤为特出。故虽然基本上很多书本上介绍与硬件之间的数据交互方式时，都会介绍 DMA 方式，但是实际上从外设驱动的角度，DMA 使用的几率比较小。

4.3.6 其他注意事项

外设驱动代码编写中有一个问题值得特别注意，不单是外设驱动，所有对外设寄存器进行操作的代码都必须注意一个问题：即对外设寄存器的操作必须使用 volatile 修饰符。虽然可以在 Vxworks 提供的 sysPhysMemDesc 数组初始化时将外设寄存器区间设置为 non-cachable，但是还是要使用 volatile 修饰符，因为 MMU 机制并非在任何条件下都有效，如 bootrom 运行期间就不使用 MMU 机制，且 Vxworks 启动的早期阶段也没有 MMU 机制的帮助，而在这过程中都需要对外设进行操作。

很多驱动程序出现一些诡异的问题，仔细检查每个寄存器的配置都没有问题，甚至从网上下载一个针对该硬件的标准配置程序，在其他平台上可用，但是就是在自己的平台上运行不正常，此时就需要特别注意设备的寄存器区域是否都定义在 volatile 型。volatile 是一个 C 语言规范中定义的修饰符，当一个变量使用该修饰符进行定义时，这就表示 CPU 对该变量的每次访问都从 RAM 中（寄存器实际上也是 RAM）取，而不要使用 CPU 内部的 cache 值，或者简单的说，使用 volatile 修饰符，就是从单个变量的层次上禁止 cache。

第五章 Vxworks 下设备驱动结构

本章我们将介绍 Vxworks 下设备驱动的结构层次，首先介绍 IO 子系统下维护的三张表及之间的关系，而后我们对内核已有驱动支持进行说明，并介绍 Vxworks 下文件系统的支持，最后讨论如何添加一个驱动到内核中。

5.1 内核驱动层次

从上一章内容，我们知道底层设备驱动并不直接对用户可见，还需要经过操作系统中间层作为“桥梁”进行交互。这个操作系统中间层根据具体应用需要可能又被分为几个子层次，我们将从用户层开始到底层硬件设备之间的所有由内核提供的中间软件层称为内核驱动层次（注意：底层驱动也是内核的一部分）。为了保持应用层用户程序的平台无关性，操作系统为应用层提供了一套标准的接口函数，这些接口函数在所有平台上都保持一致，只是随着平台的变化，底层驱动或接近驱动部分操作系统中间层可能会随着调整。如此一来可以使得用户程序脱离平台，增加了应用层开发的效率，避免重复编码。通用操作系统下，将这套提供给应用层的标准接口函数从操作系统中独立出来，专门以标准库的形式存在，这样更进一步增加了应用程序的平台无关性，平台差别完全被操作系统屏蔽。Vxworks 下也对应用层提供了一套标准文件操作接口函数，实际上与通用操作系统提供的一致，我们将其称作为标准 IO 库，Vxworks 下由 ioLib.c 文件提供。ioLib.c 文件提供如下标准接口函数：creat，open，unlink，remove，close，rename，read，write，ioctl，lseek，readv，writev 等。Vxworks 操作系统区别于通用操作系统的一个很大的不同点是 Vxworks 下不区分用户态和内核态，用户层程序可以直接对内核函数进行调用而无需使用指令中断之类的机制或者存在使用权限上的限制。所以 Vxworks 提供给应用层的接口无需通过外围库的方式，而是直接以内核文件的形式提供。用户程序可以直接使用 ioLib.c 文件定义的如上这些函数，这些函数名称与通用操作系统标准库下的函数名一致，是 Vxworks 对标准库的模拟。由于是 Vxworks 内核提供，我们将 ioLib.c 文件中定义的如上这些函数看作是内核的一部分。ioLib.c 文件提供的函数仅仅是一个最上层的接口，其并不完成具体的用户请求，而是将请求进一步向其他内核模块进行传递，位于 ioLib.c 模块之下的模块就是 iosLib.c。我们将 ioLib.c 文件称为上层接口子系统，将 iosLib.c 文件称为 IO 子系统，注意二者的区别。上层接口子系统直接对用户层可见，而 IO 子系统则一般不可见（用户当然也可以直接调用 iosLib.c 中定义的函数，但一般需要做更多的封装，且违背了内核提供的服务层次），其作为上层接口子系统与下层驱动系统的中间层而存在。如图 5-1 所示显示了内核驱动层次。

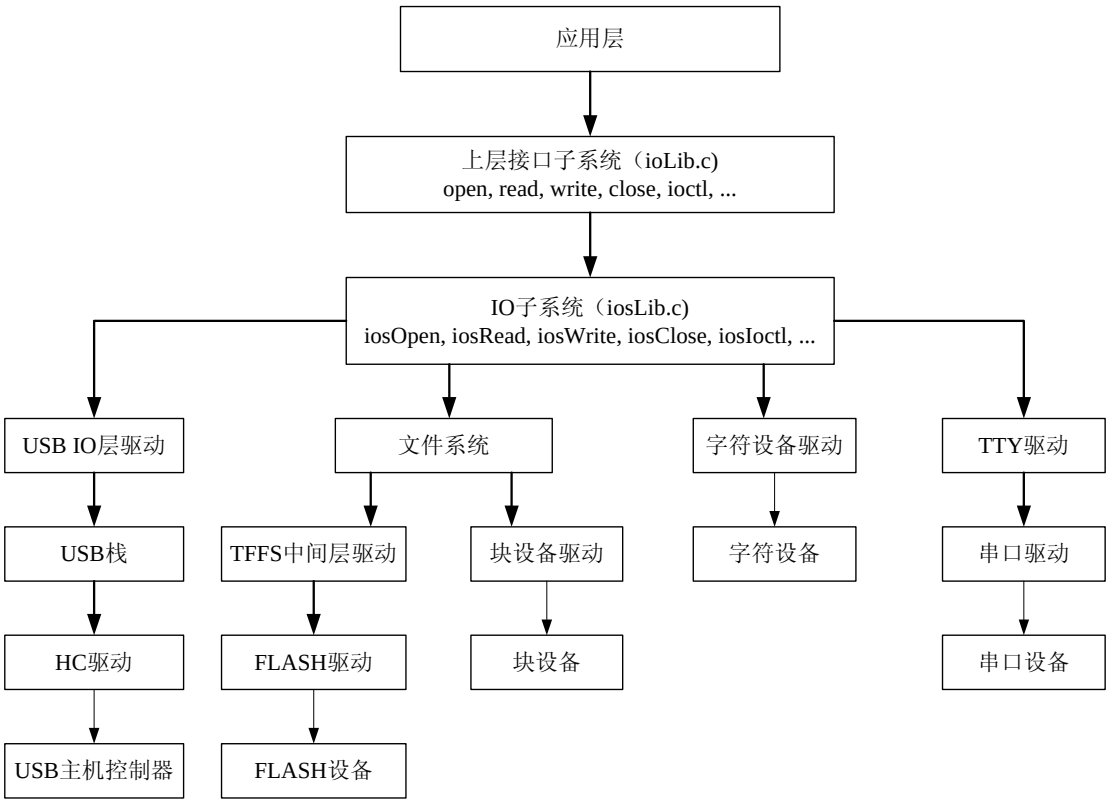


图 5- 1 Vxworks 内核驱动层次

从图 5-1 中可以看出，IO 子系统在整个驱动层次中起着十分重要的作用，其对下管理者所有类型的设备驱动。换句话说，所有类型（包括网络设备）的设备都必须向 IO 子系统进行注册方可被内核访问。所以在 IO 子系统这一层次，内核维护着几个十分关键的数组用以对设备驱动，设备本身以及当前系统文件句柄进行管理。

对于 USB 设备，通过 USB IO 层驱动与 IO 子系统进行交互，而与底层的交互则通过 USB 栈完成，HC 驱动即主机 USB 控制器驱动，Vxworks 内核提供 UHCI，OHCI，EHCI 三种标准 USB 主机控制器的驱动支持，但是在嵌入式平台下有时会存在一个定制的 USB 主机控制器，此时可能需要编写底层 HC 驱动。

块设备从当前应用来看主要分为两大类：其一就是长期以来使用的硬盘设备；其二就是 FLASH 设备。硬盘设备驱动支持通过一个文件系统中间层，这个中间层在操作系统中十分关键，其对硬盘数据进行管理，提供硬盘数据的内核缓存区，避免频繁访问硬盘对整个系统性能造成非常不利的影响。对于硬盘这类块设备，直接在文件系统层下通过一个硬盘驱动即可完成。另一类块设备是 FLASH 设备，FLASH 设备的最大特点式擦除次数有限，且擦除单元一般都较大，花费时间长，必须在文件系统层与 FLASH 驱动层之间插入一个针对 FLASH 设备这些特点的管理层，而且 FLASH 设备每次写入的数据块相对硬盘设备都比较大，而在文件系统的统一管理下，我们需要将 FLASH 设备模拟成一个硬盘设备，故在文件系统层和 FLASH 驱动层还需要一个转换层或称映射层，Vxworks 内核提供 TFFS(True Flash File System)中间层完成以上对 FLASH 管理和映射的功能。TFFS 中间层内部又进行分层，每层对应一个驱动文件，这一点在本书“FLASH 驱动”一章将有较详细的讨论。

字符设备即以字节流方式进行数据交互的设备，该类设备只能顺序的读写数据，不能像块设备那样对同一数据进行反复操作。如一个 ADC 设备，其接收外界检波器模拟信号，转换为

数字信号，通过串行接口将数据送给 CPU，此时就需要一个驱动对这个 ADC 设备进行驱动以获取转换后的数据。从图 1 层次来看，这个驱动将直接工作在 IO 系统的管理之下。
串口设备本身也是一个字符设备，但是由于串口使用范围广，为了提供串口驱动的编程效率以及串口数据的收发效率，Vxworks 内核将串口设备区分一般的字符设备对待，其提供一个 TTY 中间层驱动对串口数据进行管理（提供缓存）。TTY 驱动向 IO 系统注册，底层串口设备驱动对 TTY 注册。从底层驱动编程难易程度来看，TTY 中间层的加入并非显著降低底层串口的编程难度，但是 TTY 中间层提供的一个最为关键的优点是极大的提高了数据收发的效率，而且也提供了很大的灵活性，如提供较多的选项用以对串口设备进行控制。

在上一章中，我们将设备类型从总体上分为了三类：字符设备，块设备和网络设备。其中的两类设备：字符设备和块设备均有 IO 子系统进行管理。而网络设备由于其特殊的工作方式将由另一套用户层标准接口函数（套接字函数）和内核网络栈进行支持。对于网络设备驱动的支持，现在使用较多的是 Vxworks 提供的 MUX 接口，在 MUX 接口下编写的底层网口驱动被称为增强型网口驱动（END：Enhanced Network Driver），网络设备内核驱动层次如图 5-2 所示。

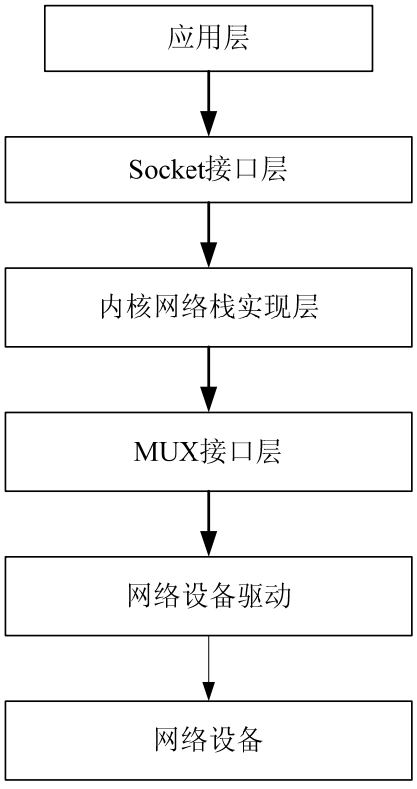


图 5- 2 网络设备内核驱动层次

网络数据由于其传输介质的不稳定性造成了网络设备内核驱动层次和应用层接口层的特殊性。网络编程应用层接口函数被称为 socket（套接字），其区别于其他所有设备编程使用的标准接口函数。为了在网络上两台主机之间通信，我们首先必须约定一个协议，这样发送方以这个事先约定好的协议封装数据，接收方按这些协议进行解析，如此方能达到数据的传送，如果二者没有共同的协议作约束，那么接收方根本就无法知道接收的报文哪些是数据，哪些是为了使报文到达接收方的帮助信息。所以网络数据传输从一开始就好设计好一套网络上所

有主机都必须遵守的通信协议。但是“一步到位”的协议设计方式为后期的扩展带来了极大的不便，如果一个新的更有效的协议被发明，那么网络上所有主机的网络驱动都要进行修改，所以网络协议从开始时就被设计为分层次的结构，每层使用本层上具有的协议对数据进行封装。最经典的当前使用范围最广的就是四层分层结构，从上到下依次为：应用层，传输层，网络层，链路层。虽然 OSI 后来开发出了一个七层分层结构，但是基于四层分层结构的应用已经十分广泛。当然 OSI 也考虑到这一点，所以七层分层结构在核心层上（传输，网络，链路）设计成与四层分层结构兼容，如此原先的四层实现只需在传输层实现之上插入一个模块即可完成对七层的支持，目前基本所有声称支持七层分层结构的实现都是如此实现的。当然有些读者可能会有疑问，为何设计分这么多层？原因在网络上传输一个报文不像主机内部传递一个报文，在网络上传输的报文首先要指定目的主机的 IP 地址以及任务绑定的端口号，同时为了使得接收方可以验证报文的合法性，还需要加上发送主机的 IP 地址以及报文发送任务绑定的端口号；其次网络接口设备接收报文并不以 IP 地址作为判断依据，IP 地址只是被操作系统网络栈实现代码使用，网络接口设备使用硬件地址，在以太网下称为 MAC 地址，一般是 6 个字节长度的字符串，所有为了使得一个主机能够从网络介质接收数据，报文中还必须提供网卡设备硬件地址；其次为了共享网络，网络上每个主机每次发送的报文长度必须有限，然而用户层一次发送的数据并不关心这一点，所以在发送报文前还必须对报文进行切割，将原先的一个报文切割成几个报文分批发送，而在接收端又需要将这几个报文进行重组后送给相关进程，这些都需要在不同的层次完成；其次，网络传输介质是不稳定的，可能有报文的丢失，不同报文可能经过不同的传输路径（想象一下网络的四通八达的点）到达，故一个先发送的报文可能较后到达等等，这些问题都需要接收双方进行考虑。所有网络栈必须设计成多个层次，每个层完成不同的功能，如果所有的功能都集中在一起进行实现，代码复杂且不易维护和修改。分成多层可以在每层实现特定的功能，且在后期加入一个新的功能也较为容易，不会对其他层实现造成明显的影响。网络栈中“栈”的概念就是起源于操作系统对于网络报文的收发使用分层的结构。网络栈实现是操作系统实现的一个固定组成部分，Vxworks 在基本的网络栈实现之下还实现了一个 MUX 接口层，专门与底层网络设备驱动进行交互。MUX 接口层将底层网络设备驱动与核心网络栈实现隔离开来，其提供的接口函数极大了降低了底层网络设备驱动设计的复杂度，因为 MUX 层本身提供了一系列函数对网络栈请求进行了响应，而且也提供一些辅助函数供底层驱动进行调用，帮助进行一些辅助功能的实现。Vxworks 将在 MUX 接口下实现的网络设备驱动称为 END 驱动，以区别于早期版中的 BSD 驱动，即直接实现在 BSD 网络栈下的网络设备驱动。

网络传输方式下，由于需要对传输的数据经过分层次的特殊的封装，且需要对网络上两台主机之间的通信的不稳定性进行处理，故网络设备并不工作在 IO 子系统下，而是使用完全不同的另一种机制，包括应用层接口函数都是独立的一套函数集合。当然这种工作方式并非 Vxworks 特有，所有的操作系统对于网络设备编程都采用与其他设备（字符，块设备）不同的方式。当然所有的平台对于网络设备的处理都是基本一致的：应用层采用 socket 编程接口，操作系统提供网络栈实现和网络设备驱动。

5.2 内核驱动相关结构

由于 IO 子系统在整个驱动层次中起着管理的功能，其维护着系统设备和驱动的关键的三张表。故本节着重介绍 IO 子系统层次的相关数据结构。具体设备的驱动结构在本书下文具体

章节进行介绍。

以一个字符设备为例，假设我们已经将该字符设备的驱动向 IO 子系统进行了注册，并创建了一个该字符设备的文件节点。下面以用户打开设备操作为例，介绍用户层请求传递到底层驱动的调用流程。
用户要使用一个设备之前，必须先打开该设备。以字符设备的文件节点为路径名调用 open 函数，open 函数将请求转移给 iosOpen。IO 子系统维护着当前系统所有驱动和所有设备。故其根据 open 函数调用时传入的设备节点名从系统设备列表中查询设备，查询到设备后，由设备结构，其继而得知该设备对应的驱动程序索引号，IO 子系统根据该驱动程序号从系统驱动列表中获取该设备对应的驱动函数集合，调用底层驱动 open 响应函数，底层驱动 open 响应函数将进行中断注册，使能硬件工作，完成用户层打开设备的服务请求。
open 函数返回一个整型数，我们将其称为文件描述符，IO 子系统除了系统驱动和系统设备两张表外，其维护的第三张表就是系统当前打开的所有文件描述符表，该表中每个表项都是一个数据结构，表项在表中的位置索引号就是文件描述符本身，而表项的内容则表明了该文件描述符对应的设备以及驱动。此后对设备的读写，控制，关闭或其他任何操作将以文件描述符为依据。由文件描述符可以直接寻址到被操作设备的驱动程序。

5.2.1 系统设备表

Vxworks 内核对每个设备使用 DEV_HDR 数据结构进行表示，该结构定义如下。

```
/*h/iosLib.h*/
typedef struct        /* DEV_HDR - device header for all device structures */
    {
    DL_NODE    node;          /* device linked list node */
    short      drvNum;        /* driver number for this device */
    char *     name;          /* device name */
    } DEV_HDR;
```

该结构中给出了链接指针（用以将该结构串入队列中），驱动索引号，设备节点名。内核提供这个结构较为简单，只存储了一些设备关键系统。底层驱动对其驱动的设备都有一个自定义数据结构表示，其中包含了被驱动设备寄存器基地址，中断号，可能的数据缓冲区，保存内核回调函数的指针，以及一些标志位。最为关键的一点是 DEV_HDR 内核结构必须是这个自定义数据结构的第一个成员变量，因为这个用户自定义结构最后需要添加到系统设备队列中，故必须能够在用户自定义结构与 DEV_HDR 结构之间进行转换，而将 DEV_HDR 结构设置为用户自定义结构的第一个成员变量就可以达到这个目的。如下所示一个用户自定义设备结构的简单示例。

```
typedef struct   xxDev
{
    DEV_HDR    devHdr;   //内核提供的结构，必须是自定义结构的第一个成员变量。
    UINT32     regBase;  //设备寄存器基地址。
    UINT32     buffPtr;  //数据缓冲区基地址。
    BOOL       isOpen;   //设备已打开标志位。
    UINT8      intLvl;   //设备中断号。
```

```
    FUNCPTR     putData;   //内核回调函数指针，该指针指向的函数向内核提供数据。
    FUNCPTR     getData;   //内核回调函数指针，该指针指向的函数从内核获取数据。
    …                      //其他设备参数。
}
```

为了能够让用户对设备进行操作，驱动程序必须将设备注册到 IO 子系统中，这个过程也被称为创建设备节点。

IO 子系统提供的 iosDevAdd 函数用以被驱动程序调用注册一个设备。该函数调用原型如下。

```
STATUS iosDevAdd
    (
    DEV_HDR *pDevHdr, /* pointer to device's structure */
    char *name,           /* name of device */
    int drvnum            /* no. of servicing driver, */
                  /* returned by iosDrvInstall() */
    );
```

注意传入 iosDevAdd 的参数：

参数 1 是一个 DEV_HDR 结构类型，一般我们将用户自定义结构作为第一个参数传入，这也是必须将 DEV_HDR 结构类型的成员变量作为用户自定义结构的第一个成员的原因所在；

参数 2 表示设备节点名，这个名称将被用户程序调用作为打开设备时的路径使用。

参数 3 是设备对应的驱动程序索引号。这个驱动号是 iosDrvInstall 函数的返回值。在设备初始化函数中，我们首先调用 iosDrvInstall 注册驱动，然后使用 iosDrvInstall 函数返回的驱动号调用 iosDevAdd 添加设备到系统中，这两步完成之后，设备就可以被用户程序使用了。

iosDevAdd 函数将一个设备添加到由 IO 子系统维护的系统设备列表中，该列表是一个队列，队列中成员通过指针链接在一起，这是由 DEV_HDR 结构中 node 成员变量完成的。系统设备列表由 iosDvList 内核变量指向，如图 5-3 所示系统设备列表示意图。

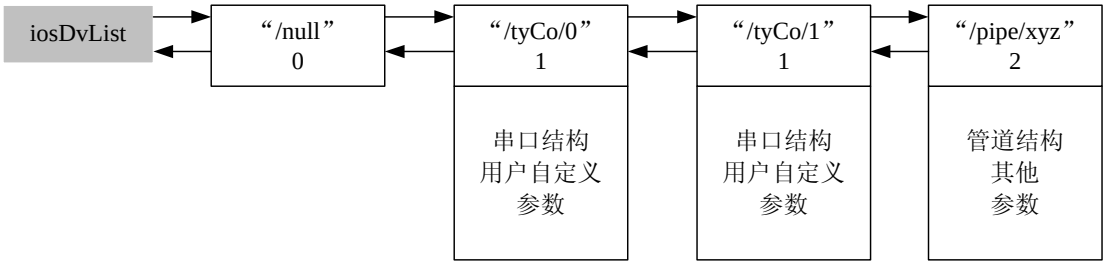


图 5- 3 系统设备表示意图

注意：图 5-3 中对于 iosDvList 的指针表示只是一种简单方式，实际上 iosDvList 还有一个尾部指针指向队列的尾部，图 5-3 中没有显示出来。

系统设备列表中第一个设备是内核本身添加的，这是一个 null 设备，所有写入 null 设备的数据都将被直接丢弃，这种机制对于屏蔽一些输出十分有效。null 设备是内核内部设备，驱动号 0 被专门预留给 null 设备。null 设备单独有一个 DEV_HDR 结构表示，不存在其他参数，故图 5-3 中对 null 设备只显示了一个 DEV_HDR 结构，而其他设备一般都需要在 DEV_HDR 结构之外定义额外的参数。

另图 5-3 中显示了系统中存在两个串口设备，这两个串口使用相同的驱动，实际上此处显示

的驱动索引号是 TTY 驱动的驱动号，还不是真正的底层串口驱动号，底层串口驱动通过 TTY 进行管理，故对不同串口的操作在 TTY 驱动层才进行分离，所有的串口驱动都首先需要通过相同的 TTY 驱动层的处理，而后请求被转发到具体的底层串口驱动。
用户可在命令行下输入 iosDevShow 或 devs，显示系统设备中的所有设备。
用户调用 open 函数打开一个设备文件时，IO 子系统将以传入的文件路径名匹配系统设备中的设备节点名，匹配方式是最佳匹配：即名称最相近的设备被返回。如输入的文件路径为“/pipe/xyzu”，如果系统设备表中存在如下两个设备：“/pipe/xy”，“/pipe/xyz”，那么“/pipe/xyz”设备将被返回，无论其位置在前在后。当然如果传入的文件路径名长度较小，那么此时系统设备表最前面的设备将被返回。例如如果传入的文件路径名为“/pipe/x”，那么对于系统设备中的“/pipe/xy”，“/pipe/xyz”两个设备，谁位于设备表的前面，谁就被返回。路径名比设备名长的情况，在对块设备的操作中比较普遍。一般我们在块设备上创建一个文件系统，我们对块设备创建一个设备节点，而对块设备的所有操作，都是在这个根节点下，此时块设备节点就成为判断一个被操作的文件或者目录到底属于哪个块设备，如果系统中存在多个块设备的话。

5.2.2 系统驱动表

IO 子系统维护的系统驱动表包含了当前注册到 IO 子系统下的所有驱动。这些驱动可以是直接驱动硬件工作的驱动层，如一般的字符驱动，也可以是驱动中间层，如文件系统中间层，TTY 中间层，USB IO 中间层等。对于中间层驱动，下层硬件驱动将由这些中间层自身负责管理，而不再通过 IO 子系统。如串口底层驱动将通过 TTY 中间层进行管理，而不再通过 IO 子系统。
系统驱动表底层实现是一个数组，数组元素数目在 Vxworks 内核初始化过程中初始化 IO 子系统时指定。iosInit 函数用以初始化 IO 子系统，iosInit 函数调用原型如下。

```
STATUS iosInit
    (
    int max_drivers,            /* maximum number of drivers allowed */
    int max_files,              /* max number of files allowed open at once */
    char *nullDevName             /* name of the null device (bit bucket) */
    );
```

参数 1 指定系统驱动表元素数目，即系统最多支持的驱动数。
参数 2 指定系统同时打开的最大文件数，这个参数实际上指定了系统文件描述符表的元素数目。
参数 3 指定了 null 设备的设备节点名，一般为“/null”。

系统驱动表在内核中由 drvTable 表示，其声明如下：

```
DRV_ENTRY *     drvTable;           /* driver entry point table */
```

在 iosInit 函数根据传入的最大驱动数目对 drvTable 进行初始化，如以下代码所示。

```
/* allocate driver table and make all entries null */
size = maxDrivers * sizeof (DRV_ENTRY);
drvTable = (DRV_ENTRY *) malloc ((unsigned) size);
```

系统驱动表中每个表项都是一个 DRV_ENTRY 类型的结构，该结构定义在 h/private/iosLibP.h

文件中，如下：

```
typedef struct        /* DRV_ENTRY - entries in driver jump table */
    {
    FUNCPTR     de_create;
    FUNCPTR     de_delete;
    FUNCPTR     de_open;
    FUNCPTR     de_close;
    FUNCPTR     de_read;
    FUNCPTR     de_write;
    FUNCPTR     de_ioctl;
    BOOL     de_inuse;
    } DRV_ENTRY;
```

可以看出 DRV_ENTRY 实际上就是一个函数指针结构，结构中每个成员都指向一个完成特定功能的函数，这些函数与用户层提供标准函数接口一一对应。成员 de_inuse 用以表示一个表项是否空闲。

iosInit 函数创建系统驱动表，从以上代码示例来看，该表实际上是由 drvTable 指向的一个数组，数组大小由传入 iosInit 函数的第一个参数决定，每个表项在 drvTable 中位置索引就作为驱动号。索引号为 0 的表项被内核预留，专门用作 null 设备的驱动号，故实际上驱动号的分配是从 1 开始的。

IO 子系统提供 iosDrvInstall 供驱动程序注册用，iosDrvInstall 函数调用原型如下。

```
int iosDrvInstall
    (
    FUNCPTR pCreate,      /* pointer to driver create function */
    FUNCPTR pDelete,      /* pointer to driver delete function */
    FUNCPTR pOpen,         /* pointer to driver open function */
    FUNCPTR pClose,       /* pointer to driver close function */
    FUNCPTR pRead,         /* pointer to driver read function */
    FUNCPTR pWrite,       /* pointer to driver write function */
    FUNCPTR pIoctl        /* pointer to driver ioctl function */
    );
```

一个设备驱动在初始化过程中一方面完成硬件设备寄存器的配置，另一方面就是向 IO 子系统注册驱动和设备，从而使得设备对用户可见。可以看到 iosDrvInstall 函数参数为一系列函数地址，这些函数对应了为用户层提供的标准接口函数。一个驱动无需提供以上所有函数的实现，对于无需实现的函数，直接传递 NULL 指针即可。iosDrvInstall 函数基本实现即遍历 drvTable 数组，查询一个空闲表项，用传入的函数地址对表项中各成员变量进行初始化，并将 de_inuse 设置为 TRUE，最后返回该表项在数组中的索引作为驱动号。设备初始化函数将使用该驱动号调用 iosDevAdd 将设备添加到 IO 子系统中。此后用户就可以使用 iosDevAdd 函数调用时设置的设备节点名对设备进行打开操作，打开后进行读写或控制等其他操作，完成用户要求的特定功能。

用户可在命令行下输入 iosDrvShow，显示系统驱动表中当前存储的所有驱动。

如图 5-4 所示为系统驱动表一个简单示意图。

索引	de_create	de_delete	de_open	de_close	de_read	de_write	de_ioctl	de_inuse
0	NULL	NULL	NULL	NULL	NULL	nullWrite	NULL	1
1	y_create	y_delete	y_open	y_close	y_read	y_write	y_ioctl	1
2	z_create	z_delete	z_open	z_close	z_read	z_write	z_ioctl	1
...	...	...	...	...	...	...	...	0

图 5- 4 系统驱动表示意图

5.2.3 系统文件描述符表

IO 子系统维护的第三张表就是系统文件描述附表，即当前系统范围内打开的所有文件描述符都将存储在该表中。文件描述符表底层实现上也是一个数组，正如设备驱动表表项索引用作驱动号，文件描述符表表项索引被用作文件描述符 ID，即 open 函数返回值。对于文件描述符有一点需要注意：标准输入，标准输出，标准错误输出虽然使用 0，1，2 三个文件描述符，但是可能在系统文件描述附表中只占用一个表项，即都使用同一个表项。Vxworks 内核将 0，1，2 三个标准文件描述符与系统文件描述符表中内容分开进行管理。实际上系统文件描述符中的内容更多的是针对硬件设备，即使用一次 open 函数调用就占用一个表项。0，1，2 三个标准文件描述符虽然占用 ID 空间（即其他描述符此时只能从 3 开始分配），但是其只使用了一次 open 函数调用，此后使用 ioGlobalStdSet 函数对 open 返回值进行了复制。如以下 usrConfig.c 文件中对于三个标准文件描述符的初始化代码。

```
 consoleFd = (-1);
if (3 > 0){
        ttyDrv();

        for (ix = 0; ix < 3; ix++)
       {
        sprintf (tyName, "%s%d", "/tyCo/", ix);
        (void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);

        if (ix == 0)
              {
                    strcpy (consoleName, tyName);
                    consoleFd = open (consoleName, 2, 0);
                    (void) ioctl (consoleFd, 4, 115200);
                    (void) ioctl (consoleFd, 3, (0x01 | 0x02 | 0x04 |
                                                          0x10 | 0x08 | 0x20 | 0x40));
              }
        }
}

ioGlobalStdSet (0, consoleFd);
ioGlobalStdSet (1, consoleFd);
ioGlobalStdSet (2, consoleFd);
```

可以看到 usrRoot 函数中将“/tyCo/0”作为了三个标准输入输出，此处只使用了一次 open 函数调用，如下语句：consoleFd = open (consoleName, 2, 0);
实际上 consoleFd=3，因为标准输入输出占用了 0，1，2 三个文件描述符，所以系统文件描述符表中存储的描述符最小值就是 3.
此后使用 ioGlobalStdSet 函数将这个描述符（3）指向的设备作为 0，1，2 三个描述符的默认设备。即此处将串口作为了标准输入输出设备。
内核将 0，1，2 三个文件描述符预留给了标准输入输出，并将其与系统文件描述符表中的表项隔离开来，内核专门使用 ioStdFd 数组表示 0，1，2 三个文件描述符指向的具体系统文件描述符表中哪个表项。

```
int ioStdFd [3];            /* global standard input/output/error */
```

所以 ioGlobalStdSet（0，consoleFd）；语句实际上完成如下工作：
ioStdFd[0] = consoleFd；
其他两条语句实际结果即：
ioStdFd[1] = consoleFd；
ioStdFd[2] = consoleFd；

而 consoleFd 等于 3，实际上是系统文件描述符表中的第一个表项，其索引为 0，但是在作为文件描述符返回时，基于 0，1，2 已被预留为标准输入输出，故做加 3 处理，实际上系统文件描述符表项索引作为文件描述符返回时都做加 3 处理。

当使用一个文件描述符进行操作时，如调用 write 函数，内核首先检查文件描述符是否是 0，1，2 标准输入输出描述符，如是，则依次为索引查询 ioStdFd，以 ioStdFd[fd]作为索引查询系统文件描述符表，获得驱动号，进而索引系统驱动表，调用对应表项 de_write 指向的函数，完成对设备的写入操作；如果文件描述符大于 2，表示这是一个普通的文件描述符，那么就直接以该描述符作为索引查询系统文件描述表，获得驱动号，进而索引系统驱动表，调用相关函数。
系统文件描述符表中每个表项都是一个 FD_ENTRY 类型的结构，该结构定义在 h/private/iosLibP.h 中，如下所示。

```
typedef struct          /* FD_ENTRY - entries in file table */
    {
    DEV_HDR *  pDevHdr;/* device header for this file */
    int    value;           /* driver's id for this file */
    char *     name;            /* actual file name */
    int        taskId;          /* task to receive SIGIO when enabled */
    BOOL       inuse;           /* active entry */
    BOOL       obsolete;  /* underlying driver has been deleted */
    void *     auxValue;/* driver specific ptr, e.g. socket type */
    void *     reserved; /* reserved for driver use */
    } FD_ENTRY;
```

注意 FD_ENTRY 结构的第一个成员就是 DEV_HDR 结构类型，该结构中存储了设备节点名和驱动号。FD_ENTRY 结构中 value 成员表示驱动附加信息，并非驱动号，实际上这个字段

被用以保存底层驱动中 open 实现函数的返回值. 这个返回值意义重大，因为其后驱动中 read，write 等实现函数被调用时，IO 子系统就以这个返回值作为这些函数的第一个参数。所以底层驱动 open 实现函数一般返回一个驱动自定义结构句柄。name 成员变量按理应该设置为文件名，但是 DEV_HDR 结构中已经有设备节点名（也就是文件名），故该成员变量当前被设置为 NULL，节省内存空间。
系统文件描述符表有内核变量 fdTable 指向，该变量声明如下。

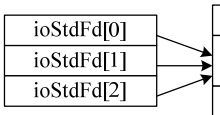
```
FD_ENTRY * fdTable;                /* table of fd entries */
```

fdTable 的初始化在 iosInit 函数中完成，该函数调用原型如前文所示，传入该函数的第二个参数指定了 fdTable 数组的大小，该变量的初始化代码示例如下。

```
size = maxFiles * sizeof (FD_ENTRY);
fdTable = (FD_ENTRY *) malloc ((unsigned) size);
```

用户程序每调用一次 open 函数，系统文件描述符表中就增加一个有效表项，直到数组满，此时 open 函数调用将以失败返回。表项在表中的索引偏移 3 后作为文件描述符返回给用户，作为接下来其他所有操作的文件句柄。

用户可在命令行下输入 iosFdShow，显示系统文件描述附表中当前所有有效表项。
如下图 5-5 所示为系统文件描述符表的一个简单示意图。



文件描述符号	索引	pDevHdr	value	name	...	inuse	...
3	0	“/tyCo/0” 1	0	NULL	...	1	...
4	1	“/tyCo/1” 1	0	NULL	...	1	...
5	2	“/pipe/0” 2	0	NULL	...	1	...
6	3	“/null” 0	0	NULL	...	1	...
...	...	...	...	...	...	...	...

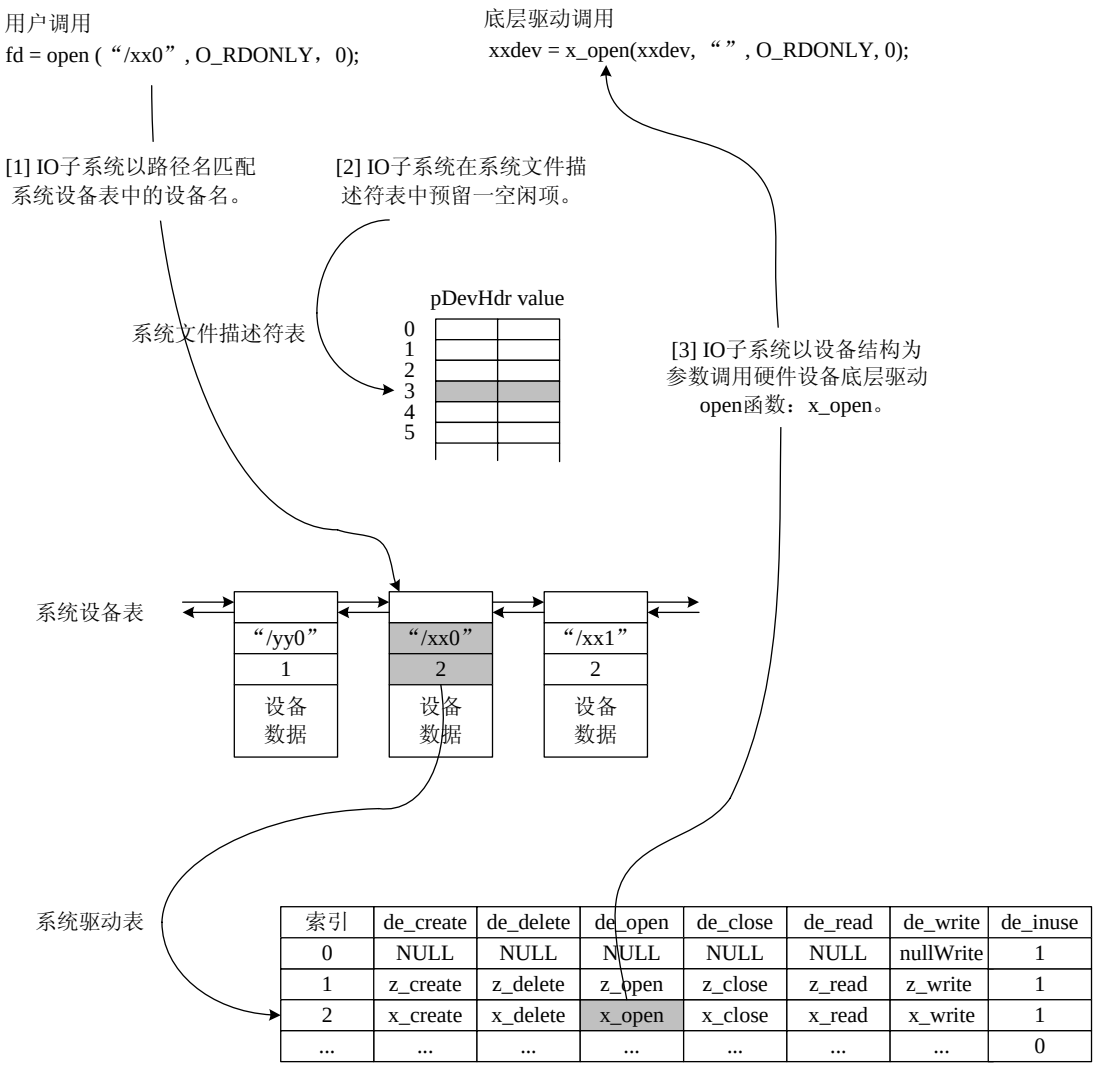
图 5- 5 系统文件描述符表示意图

注意每个文件描述符的确定由对应设备文件被打开的时机决定，一般而言，串口被用作标准输入输出是最早被打开的设备，其他设备只在需要时由用户程序打开。当然这也不可一概而论。
图 5-5 中也显示了 0，1，2 三个标准文件描述符对系统文件描述符表的索引。此时“/tyCo/0”设备同时也被用于标准输入输出。ioStdFd 数组有且仅有三个元素，0 号元素表示标准输入，1 号元素表示标准输出，2 号元素表示标准错误输出。ioStdFd 数组索引本身表示文件描述符，而元素内容表示实际操作设备时使用的文件描述符，usrRoot 函数将 ioStdFd 三个元素均设置为 3，即第一个串口设备对应的文件描述符（3），也就是使用第一个串口设备作为标准输入输出设备。

5.2.4 三张表之间的联系

下面我们以一个 open 调用为例介绍 IO 子系统维护的三张表是如何相互协作完成用户请求的。
如图 5-6，5-7 所示，用户调用 open 函数打开”/xx0”文件，Vxworks 内核 IO 子系统将进行如下一系列响应。

[1] 其使用文件路径名匹配系统设备表，查询一个匹配设备。此处在设备列表中找到了一个匹配的设备。
[2] 其在系统文件描述符表中预留一个空闲项，用以创建一个文件描述符，如果后续调用成功，将以这个空闲项对应的索引（偏移 3）值返回给用户，作为文件描述符使用。
[3] 其根据设备列表匹配项中信息得到驱动号，进而以此驱动号为索引从系统驱动表中获取底层设备驱动对应用户层 open 调用的响应函数 x_open。x_open 的第一个参数被设置为对应硬件设备结构，第二个参数为除去设备名本身余下的部分，此处为 NULL，第三，四个参数为用户传入的权限和模式参数。x_open 将完成硬件设备的配置，使能工作，注册中断等等，为用户接下来可能的读写设备操作做好准备。x_open 同时对传入的第一个参数：设备结构进行初始化，这个结构将在后续操作中一直被底层驱动使用。



索引	de_create	de_delete	de_open	de_close	de_read	de_write	de_inuse
0	NULL	NULL	NULL	NULL	NULL	nullWrite	1
1	z_create	z_delete	z_open	z_close	z_read	z_write	1
2	x_create	x_delete	x_open	x_close	x_read	x_write	1
...	...	...	...	...	...	...	0

图 5- 6 用户请求服务过程 A

[4] 底层驱动返回设备结构，表示底层打开设备成功，否则返回 NULL 或 ERROR 表示调用失败。
[5] IO 子系统对文件描述符表中预留的空闲项进行初始化，填入驱动号和设备结构。

[6] 最后 open 函数调用返回一个文件描述符，这个描述符是文件描述符之前被预留空闲项（现在已得到初始化，被占用使用）在表中的索引值（偏移 3）。此处即文件描述符表中的第一个表项，即 fd=0+3=3.

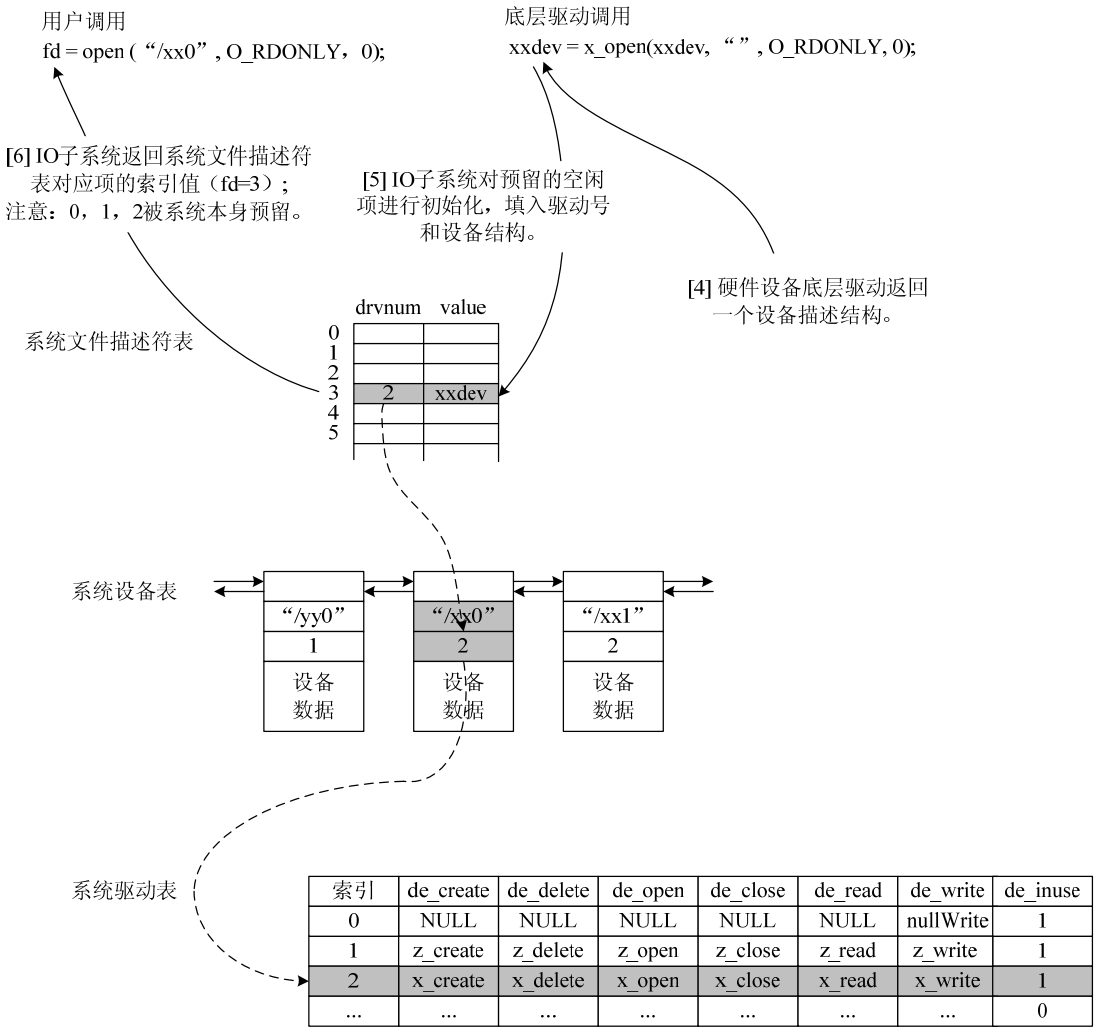


图 5- 7 用户请求服务过程 B

5.3 内核驱动支持

Vxworks 除了提供 IO 子系统对 Vxworks 下所有设备类型（当然除了网络设备）进行管理外，其本身还提供一些驱动程序，这些驱动程序中有些作为驱动中间层（如 TTY 驱动，TFFS 驱动，USB IO 层驱动），帮助简化底层硬件设备驱动的设计；有些则是直接的底层驱动，驱动一个虚拟设备工作。对于中间层驱动，我们在本书具体驱动章节中进行介绍，本节将介绍 Vxworks 内核提供的虚拟设备驱动。所谓虚拟设备即完全使用软件进行模拟工作的设备，这些设备通过封装成特定的结构，完成特定的功能，大多建立在内存之上，用于数据的转换或者某种任务间通信机制。最常见且使用较多的有如下虚拟设备：管道设备（pipeDrv），虚拟内存设备（memDrv），RAM 磁盘设备 RamDisk（ramDrv）以及网络设备（netDrv）。

5.3.1 管道虚拟设备

Vxworks 下管道设备主要被用以任务间通信。管道虚拟设备直接工作在 Vxworks IO 子系统的管理之下，管道设备在内核初始化过程中通过调用 pipeDrv 函数完成自身的初始化。pipeDrv 函数在 usrRoot 中被调用，如下所示。

```
#ifdef      INCLUDE_PIPES
      pipeDrv ();                       /* install pipe driver */
#endif     /* INCLUDE_PIPES */
```

为了在 Vxworks 内核映像中包含管道虚拟设备组件，必须进行定义 INCLUDE_PIPES 宏。pipeDrv 函数完成管道虚拟设备相关资源的分配和内核结构的初始化，调用 iosDrvInstall 函数向 IO 子系统注册管道虚拟设备驱动。注意 pipeDrv 函数并非进行管道虚拟设备的创建，故用户在使用管道虚拟设备之前还必须使用 iosDevAdd 函数首先创建一个管道设备。Vxworks 内核为此提供了更高层次的接口函数 pipeDevCreate，用户使用该函数完成管道设备的创建，其后就可以使用 open 函数对该管道设备进行打开，进而进行读写，控制操作。pipeDevCreate 函数调用原型如下。

```
STATUS pipeDevCreate
      (
      char *name,              /* name of pipe to be created          */
      int nMessages,          /* max. number of messages in pipe */
      int nBytes               /* size of each message                */
      );
```

pipeDevCreate 函数参数：

参数 1：管道设备节点名。该名称在调用 open 函数将作为路径名使用用以打开该管道设备。

参数 2：创建的管道设备可存储的最大消息数量。

参数 3：每个消息的最大长度。

Vxworks 下管道的底层实现基于消息队列，故此处参数 2，3 的含义同消息队列创建时输入的参数含义。这也是在管道创建时需要传入这两个参数的原因所在。

pipeDevCreate 实现中最后将调用 iosDevAdd 函数将这个管道设备添加到 IO 子系统中。pipeDevCreate 函数调用之后，用户就可以使用调用 pipeDevCreate 函数传入的设备名作为路径调用 open 函数打开这个管道进行读写操作。如下代码示例。

```
void pipe_test(){
      int pfd;
      //管道设备名为“/pipe/x”，最大消息数量为 10，每个消息的最大长度为 1024 个字节。
      pipeDevCreate(“/pipe/x”, 10, 1024);
      pfd = open(“/pipe/x”, O_WRONLY,0);
      if(pfd<0){
            printf(“open pipe error.\n”);
            return;
      }
      write(pfd, “hello,vxworks.”, 14);
      close(pfd);
```

```
    return;
}
```

以上示例简单了演示了管道的使用方式，管道虚拟设备是 Vxworks 下任务间通信机制之一，使用较多，Vxworks 通过底层驱动的方式以消息队列为基础模拟出一个信息通道，我们将其称为管道。注意：要在内核中添加管道设备支持，必须包含 INCLUDE_PIPES 宏定义，否则管道设备将不可用。

5.3.2 虚拟内存设备

虚拟内存设备即将一段内存空间模拟成一个设备供用户进行读写。虚拟内存有些类似于管道，但是相比较 Vxworks 内核对管道的封装，内核对虚拟内存设备的处理则较为简单，实际上它仅仅作为一个“设备”，内核没有对其进行任何封装。此类设备大多数时候被用作驱动程序设备设计的典型示例，因为其只需要一段可用的内存即可，无需任何实际设备支持。虚拟内存设备在内存中模拟出一个设备或者更确切的说是一个文件，用户在创建出一个虚拟内存设备后，可以使用 open 函数打开该设备，向该设备写入任何数据，这些数据被原封不动的写入到表示设备的内存区域中，当然也可以将这个写入的内容重新读出来，其外在表现其实就是一个文件，可以移动文件偏移指针，读取固定偏移处的数据或向该处写入数据。从数据读写和存储方式来看，虚拟内存设备可以被作为最原始的任务间通信手段，不过数据的同步需要任务自身完成或者说给任务预留了很大的灵活性。

虚拟内存设备并非仅用作设备驱动编写的教案，其具有一个很有用的使用场合：当下载型启动方式下 Vxworks 内核是压缩型映像时，虚拟内存设备就显得非常有用。我们将在其后代码示例中进行详细介绍，首先我们讨论一下虚拟内存设备的相关接口函数。

虚拟内存设备驱动并不作为 Vxworks 内核初始化的一个组成部分。只当用户需要使用虚拟内存设备提供的功能时，可以即时通过调用 memDrv 函数完成虚拟内存设备驱动的注册，该调用 iosDrvInstall 函数向 Vxworks IO 子系统注册虚拟内存设备驱动。在完成驱动的注册后，在使用设备之前，我们还必须创建一个虚拟内存设备的节点，Vxworks 为此提供了 memDevCreate 函数，该函数调用原型如下。

```
STATUS memDevCreate
    (
    char * name,        /* device name                  */
    char * base,        /* where to start in memory */
    int    length       /* number of bytes        */
    );
```

参数说明：

参数 1（name）：虚拟内存设备节点名，也是 open 函数调用时使用的路径名。

参数 2（base）：虚拟内存设备所使用内存基地址。

参数 3（length）：虚拟内存设备所使用内存大小。

注意：用户在调用 memDevCreate 函数创建虚拟内存设备时必须自己指定该设备所用内存的基地址以及大小，不可以让 memDevCreate 自己负责分配，即参数 2 不可以为 NULL。用户可以通过预留一段内存空间专门用作虚拟内存设备的使用空间。如果只在内核启动过程中使

用内存设备，那么就可以直接使用任何空闲的内存空间。后文中内存设备使用示例中就是直接使用空闲内存空间作为内存设备使用。memDevCreate 函数将最终将调用 iosDevAdd 函数完成虚拟内存设备的添加。此时用户就可以使用 open 函数打开着内存设备进行操作了。

虚拟内存设备的使用场合不多，但是在某些情况下却很有用。下面是一种使用虚拟内存设备的情况，作为虚拟内存设备的使用示例。
下载启动方式中，由 bootrom 完成 Vxworks 内核映像的下载。一般情况下这个被下载的 Vxworks 映像文件都是非压缩的，故通过 bootrom 直接下载到 RAM_LOW_ADRS 地址处即可。但是某些情况下我们会将映像文件进行压缩（注意这个压缩是对映像文件的整体压缩，而不是指压缩版本的 Vxworks 映像，对于下载方式下的 Vxworks 映像，其只经过一次链接，即所有代码都是非压缩的），这可能是对某种工作方式的试验，如先通过网络下载方式试验 Vxworks 映像压缩后的工作情况，进而压缩后的映像文件写入 FLASH，由 bootrom 从本地 FLASH 中下载压缩映像启动。

注意：此处再次声明，以上一段所说的压缩生成非压缩版本的 Vxworks 内核映像文件后，在命令行下使用 deflate 工具对整个文件的压缩，而并非 Vxworks 内核映像中一部分是压缩的，另一部分是非压缩的。下面一段中的压缩也是指对映像文件调用 deflate 后得到的 Vxworks 映像。

下载方式下，Vxworks 映像一般放置在主机服务器上，由目标机中的 bootrom 通过网络将这个映像下载到目标机 RAM 中（这个下载过程由 bootLoadModule 函数完成，下议），进而跳转到 RAM 中执行 Vxworks 内核映像，完成 Vxworks 操作系统的启动。

实际上 bootrom 最后是调用内核函数 bootLoadModule 完成 Vxworks 内核映像的解析和下载工作的。bootLoadModule 是 Vxworks 内核提供的，其将根据 Vxworks 映像文件格式（ELF）调用具体的处理函数对文件进行解析，将文件中代码和数据通过网络直接从主机文件中拷贝到其链接地址处（即 RAM_LOW_ADRS），最后返回入口地址。bootLoadModule 调用原型如下：

```
STATUS bootLoadModule
    (
    FAST int fd,            /* fd from which to read module */
    FUNCPTR *pEntry         /* entry point of module */
    );
```

通常情况下，我们传递一个非压缩的 Vxworks 内核映像文件的文件描述符和一个入口函数指针给 bootLoadModule 函数。对于非压缩版本的内核映像文件，我们直接通过网络方式打开主机上的 Vxworks 内核映像文件，将返回的文件句柄传递给 bootLoadModule 函数即可。然而对于压缩的 Vxworks 映像文件，我们就不能简单的通过网络打开主机上这个压缩的 Vxworks 映像文件，将返回的文件句柄传给 bootLoadModule 函数，我们首先需要将这个压缩版的 Vxworks 映像下载到本地（即目标机）内存中，对其解压缩，然后将解压缩后的映像封装成文件形式，对其调用 open 函数，再将这个 open 函数返回的文件句柄传入 bootLoadModule 函数才能达到我们的目的，此处虚拟内存设备就起着将解压缩后的 Vxworks 映像的封装成文件形式的作用。

由于内核函数最终将 Vxworks 内核代码和数据拷贝到 RAM_LOW_ADRS 指向的内存区域，

故解压缩后的 Vxworks 映像必须在另一个内存地址处，且不可与 RAM_LOW_ADRS 处内核映像结束地址（end 变量指向）重合（否则会出现一些诡异的问题）。我们可以将其放置在 RAM_HIGH_ADRS 处。而对于 Vxworks 映像压缩文件的下载地址，则可以放置在 RAM_LOW_ADRS 处。总结一下即：我们首先通过网络将 Vxworks 映像文件经过 deflate 函数压缩后的版本下载到 RAM_LOW_ADRS 地址处，然后调用 inflate 函数对压缩文件解压缩，解压到 RAM_HIGH_ADRS 地址处，然后使用虚拟内存设备将解压缩后位于 RAM_HIGH_ADRS 处的 Vxworks 映像封装成文件，对其调用 open 函数，此时返回的文件句柄就指向一个正常格式（指没有经过 deflate 处理）的 Vxworks 内核映像，将这个句柄传递给 bootLoadModule 函数，bootLoadModule 函数将对其进行解析，将文件中代码和数据拷贝到 RAM_LOW_ADRS 指向的地址处，这与正常情况下通过网络的方式拷贝代码和数据达到相同的效果，只不过此时打开的文件位于本地内存中，而通常打开的文件位于主机服务器上。

这种工作方式看似没有必要，不过确是实际中常用的一种方式。首先 Vxworks 启动方式选择是 bootrom+Vxworks，Vxworks 是非压缩版本的，即 Vxworks 内核映像的生成只经过一次链接过程。但是我们对整个 Vxworks 内核映像文件进行压缩，Vxworks 工具箱专门提供了命令行方式下工作的 deflate 命令对文件进行压缩。可能某些读者有疑问，既然如此，为何不采用对 Vxworks 内核映像使用二次链接生成方式，这样也可以进行压缩。如果这样做，那么就只能采用 ROM 型启动方式了，因为下载启动方式下，Vxworks 自身无法完成解压缩工作，必须由 bootrom 完成，而对于二次链接方式生成的 Vxworks 内核映像，bootrom 基本上是束手无策。故一般采用一次链接方式生成一个非压缩的映像文件，而后采用 deflate 工具对整个映像文件进行压缩。这样 bootrom 就可以使用 inflate 函数对整个文件进行解压缩。对整个文件进行压缩，并利用 bootrom 进行解压缩，并采用 bootrom+Vxworks 启动方式的环境是：bootrom 较小（500KB 左右），故可以将其烧入平台 EEPROM 或者容量较小的 NORFLASH 中，Vxworks 内核映像可能非常大（如 40M 以上），即便采用二次链接也无法使用目标板上有限的 NorFlash 资源（注意 ROM 型 Vxworks 映像必须存在在具备在线执行能力 EIP）的介质中。此种情况下，以上所介绍的方式就可以解决该问题，一般目标板上 EEPROM（4MB 左右）或者 NorFlash（16MB 左右）资源有限，但是 NandFlash 资源却十分丰富（可达 256MB），此时可以将 bootrom 烧入 EEPROM 中，而将 Vxworks 写入 NandFlash 中，由于嵌入式系统下 NandFlash 大多需要同时用作为平台文件系统存储，故需要尽量减小 Vxworks 内核映像的大小，一般都是对其调用 deflate，对整个映像文件进行压缩。bootrom 先从 NandFlash 中读取 Vxworks 压缩文件，而后解压缩，最后还是需要通过调用 bootLoadModule 函数对解压后文件进行解析。故其中还是必须将解压后内存中内容封装成文件形式。

无论经过 deflate 压缩后的映像文件存在于何处（网络主机上或是本地 NandFlash 中），bootrom 首先将其下载到 RAM_LOW_ADRS 地址处，然后调用 inflate 函数进行解压缩，并将解压缩后内容存放到 RAM_HIGH_ADRS 地址处。现在的问题就是这个解压缩后的内容仍然是一个文件的格式：有文件头，段头部，符号表等等，我们要调用 bootLoadModule 函数对这个解压缩后内容进行解析，但是 bootLoadModule 函数只接收文件描述符，故我们需要将 RAM_HIGH_ADRS 开始的内存空间封装成一个文件形式，然后对其进行打开操作，返回一个文件描述符，之后才能调用 bootLoadModule 函数完成解析。虚拟内存设备就是为此设计的。下面是利用虚拟内存设备将从 RAM_HIGH_ADRS 开始的内存空间封装成文件的代码。

```
printf(“Load image … \n”);
memDrv();
memDevCreate(“/mem_vx”, RAM_HIGH_ADRS, IMAGE_MAX_SIZE);
if(（fd=open(“/mem_vx”, O_RDONLY, 0)) < 0）{
    printErr(“Cannot open memory device. \n”);
    goto err_out;
}
if(bootLoadModule(fd, pEntry) != OK){
    printErr(“Error Loading Image: errno = %d.\n”, errnoGet());
    goto err_out;
}
```

以上这段代码示例了虚拟内存设备的使用方法，也显示了虚拟内核设备在某些情况下十分有效。最后 bootLoadModule 函数将对文件进行解析，将文件中代码和数据段拷贝到 RAM_LOW_ADRS 地址处，并将 pEntry 初始化为指向 sysInit 函数，该函数是下载启动方式下 Vxworks 内核的入口函数。

5.3.3 ramDisk 设备

ramDisk 设备也是基于内存的一种设备，但其与虚拟内存设备的使用场合大不相同。虚拟内存设备将一段内存空间封装成一个文件的形式；而 ramDisk 则在这段内存空间创建一个文件系统，如同基于硬盘的文件系统一样，可在其中建立目录和文件。在完成 ramDisk 文件系统的创建后，其操作方式完全等同于一般的基于硬盘的文件系统，但是 ramDisk 文件系统中所有的改变只存在于内存中，换句话说，一旦系统掉电，其中的内容将全部丢失。这在某些特殊情况下将十分有用，如某些高密级应用中，可使用 ramDisk 保存核心参数，以文件系统方式进行操作，可以对各种参数分类保存，便于程序使用。硬件设计上的某些特殊处理同时可以从硬件角度保证参数的安全性。

ramDisk 设备驱动可在 Vxworks 操作系统初始化过程中进行注册，这个注册的任务由 ramDrv 函数完成。ramDrv 在 usrRoot 中被调用，其调用环境如下。

```
#ifdef      INCLUDE_RAMDRV
    ramDrv ();                          /* initialize ram disk driver */
#endif      /* INCLUDE_RAMDRV */
```

故 ramDisk 组件的包含与否由 INCLUDE_RAMDRV 宏进行控制。

事实上 ramDrv 函数是一个空实现。因为 ramDisk 并非直接由 IO 子系统进行管理。ramDisk 的使用方式是在一段内存空间创建一个文件系统，故 ramDisk 驱动工作在文件系统层之下，以常用的 MS-DOS 兼容型文件系统为例，ramDisk 将通过 dosFs 层管理，ramDisk 使用方式与以上两种设备就有些差别。首先我们需要调用 ramDevCreate 函数设置 ramDisk 所使用内存的某些参数。该函数调用原型如下。

```
BLK_DEV *ramDevCreate
    (
    char *ramAddr,      /* where it is in memory (0 = malloc) */
```

```
    FAST int bytesPerBlk,   /* number of bytes per block */
    int         blksPerTrack,  /* number of blocks per track */
    int         nBlocks,  /* number of blocks on this device */
    int         blkOffset /* no. of blks to skip at start of device */
    );
```

ramDisk 使用 RAM 内存模拟硬盘设备，故对内存参数的设置也是基于硬盘的一些参数。
参数说明：
参数 1（ramAddr）：ramDisk 所使用 RAM 内存基地址。以 0 参数传入时，表示由内核通过 malloc 函数调用分配内存，内存大小由参数 2，4 决定。
参数 2（bytesPerBlk）：指定每个块的大小。此处块的概念类似于硬盘中扇区。
参数 3（blksPerTrack）：指定每道中的块数。此处道的概念类似于硬盘中磁道。
参数 4（nBlocks）：总块数，可以理解硬盘设备中的总扇区数。
参数 5（blkOffset）：ramDisk 起始处跳过不使用（或用以特殊使用）的块数。该参数通常设置为 0.

由于 ramDisk 并非由 IO 子系统直接管理，而是在文件系统管理之下，故 ramDrv 函数并不调用 iosDrvInstall 函数向 IO 子系统注册 ramDisk 驱动，ramDrv 函数实际上是一个空实现。ramDisk 的驱动实际上就是在 ramDevCreate 函数中完成注册的，注册的对象不是 IO 子系统，而是文件系统。注册的驱动函数地址就被保存在 ramDevCreate 函数返回的 BLK_DEV 结构中。
这个 BLK_DEV 类型的返回值将被 dosFsDevCreate 函数使用用以创建一个文件系统设备，并同时将 ramDisk 驱动注册到文件系统中。注意：在 dosFsDevCreate 函数中块设备节点才被创建，此时块设备将如同其他设备一样通过 iosDevAdd 函数添加到系统设备列表中。每次用户对 ramDisk 中文件进行操作时，请求首先通过 IO 子系统传递给 dosFs 驱动层，然后通过 dosFs 驱动层传递给 ramDisk 驱动，完成请求的最终服务。

如下代码示意了 ramDisk 的使用方法。

```
BLK_DEV *pBlkDev;
pBlkDev = ramDevCreate (0, 512, 400, 400, 0);
if (dosFsDevCreate ("/ramdisk", pBlkDev, 20, NULL) == ERROR)
{
    printErrno( );
};
/*  格式化。对于 ramDisk，一般在创建后都需要进行格式化。*/
if (dosFsVolFormat ("/ramdisk", 2, NULL) == ERROR)
{
    printErrno( );
}
```

如上代码中，ramDisk 内存大小为 512x400B=200KB，且 ramDevCreate 的第一个参数为 0，表示由内核调用 malloc 函数分配 200KB 的内存作为 ramDisk 使用。ramDevCreate 函数返回一个 BLK_DEV 结构类型，这个结构将被用以调用 dosFsDevCreate 函数创建一个 dos 文件系统设备。所以事实上，这个 ramDisk 设备节点直到 dosFsDevCreate 函数才被创建。注意

ramDevCreate 并不创建设备节点，其只是设定 ramDisk 所用内存的相关参数，初始化一个 BLK_DEV 结构，为 dosFsDevCreate 函数调用做准备。在 dosFsDevCreate 函数才完成 ramDisk 设备节点的创建和 ramDisk 驱动向 dosFs 文件系统的注册。dosFsVolFormat 用以格式化 ramDisk 内存区域，即在 ramDisk 设备上创建 dosFs 文件系统。

以上代码执行后，用户就可以在/ramdisk 下创建目录和文件了，用户可以在 shell 下使用如下命令对 ramDisk 进行试验。

```
-> devs
-> cd “/ramdisk”
-> mkdir “/test”
-> copy “/tffs/vxWorks”, “vxWorks”
-> ls
```

如果一段内存区域之前已被用作 ramDisk，而且已经创建过文件系统，现在需要重新挂载，那么此时就可以省去格式化这一步，而且必须省去，因为一旦进行格式化，所有的数据将丢失，重新挂载只是一种工作方式，一般情况下很少会使用，此处只给出相关代码，并做简单介绍。

```
pBlkDev = ramDevCreate (0xc0000, 512, 400, 400, 0);
if (dosFsDevCreate ("/ramdisk", pBlkDev, 20, NULL) == ERROR)
{
    printErrno( );
}
```

注意 ramDevCreate 函数第一个参数必须是之前用作 ramDisk 内存空间的起始地址，这个地址可以是用户自行分配的，也可以是内核分配的（此时可以通过返回的 BLK_DEV 结构获取，实际上中间还需要进行一次计算，因为 BLK_DEV 中并不保存这个地址，我们需要先从 BLK_DEV 得到 RAM_DEV），ramDevCreate 函数后 4 个参数必须与之前调用 ramDevCreate 函数完全一致！

5.3.4 网络设备（netDrv）

网络设备即通过 RSH（Remote Shell protocol）或者 FTP（File Transfer Protocol）协议将远程主机上某个设备作为本地设备访问的一种方式，操作系统屏蔽了底层网络通信的所有细节，用户层通过标准的文件操作接口函数（open，read，write 等）对远程主机上的文件进行打开，读写和控制。当使用 RSH 或者 FTP 打开远程主机上的一个设备时，整个文件将通过网络首先拷贝到本地 RAM 中，接下来的所有操作将对本地 RAM 中的文件副本进行，最后关闭文件时，如果有修改，那么这个文件又将通过网络写回到远程主机上。由于操作系统将底层网络通信细节对用户进行了完全屏蔽，所以对于一个用户层应用而言，就表象就是在对一个本地文件进行操作。网络设备在开发阶段十分有用，我们可以将要测试的程序放置在远程主机上，通过网络设备形式进行打开，测试，修改。由于这些程序的编译都是在远程主机上完成，故操作上十分方便。

网络设备初始化在 Vxworks 内核初始化过程中完成，具体的是在 usrNetInit 函数（被 usrRoot

函数调用）中完成。用户必须定义 INCLUDE_NET_INIT 用以在初始化过程调用 usrNetInit 对整个网络栈进行初始化；同时必须定义 INCLUDE_NET_REM_IO 用以初始化网络设备。内核将根据 bootline 参数值确定远程主机 IP，主机名，以及根据是否提供了密码决定是使用 RSH 还是 FTP 协议进行网络设备的创建。

网络设备节点名默认使用远程主机名，不过也进行了些微修改，即在主机名后添加了一个冒号，假设主机为“vxw”，则网络设备节点名则为“vxw：”。

netDrv 函数负责网络设备驱动的注册，其调用 iosDrvInstall 函数向 IO 子系统注册网络设备驱动函数集合，为了使得网络设备对用户可见，还必须调用 netDevCreate 函数创建一个网络设备节点。netDevCreate 函数调用原型如下。

```
STATUS netDevCreate
    (
    char *devName,          /* name of device to create */
    char *host,             /* host this device will talk to */
    int protocol            /* remote file access protocol 0 = RSH, 1 = FTP */
    );
```

参数 1：网络设备节点名。

参数 2：远程主机名。

参数 3：访问协议设置。网络设备只支持两种访问协议：RSH 和 FTP。

netDevCreate 函数最后将调用 iosDevAdd 函数将这个网络设备添加到系统设备列表中。

usrNetInit 函数在 usrNetwork.c 文件中定义，如下代码片段是该函数中用以初始化网络设备驱动层以及创建网络设备的过程。

```
#ifdef INCLUDE_NET_REM_IO
    if (netDrv () == ERROR)      /* init remote file driver，注册网络设备驱动 */
        NET_DIAG(("Error initializing netDrv; errno = 0x%x\n", errno));
    else
    {
        sprintf (devName, "%s:", params.hostName);          /* make dev name */
        protocol = (params.passwd[0] == EOS) ? 0 : 1;    /* pick protocol : 0-RSH, 1-FTP*/

        /* create device */
        //创建网络设备
        if (netDevCreate (devName, params.hostName, protocol) == ERROR)
        NET_DIAG(("Error creating network device %s - errno = 0x%x\n",
                devName, errno));
        else   // ioDefPathSet 函数将操作系统默认根目录设置为网络设备
        if (ioDefPathSet (devName) == ERROR) /* set the current directory */
            NET_DIAG(("Error setting default path to %s - errno = 0x%x\n", devName, errno));
    }
    iam (params.usr, params.passwd);                    /* set the user id */

    taskDelay (sysClkRateGet () / 4);         /* 1/4 of a second */
#endif /* INCLUDE_NET_REM_IO */
```

网络设备这个词是从目标机而言的，对于主机而言，其相当于 FTP 服务器某个用户的根目录。所以网络设备是一个具有目录层次的类似于文件系统的块设备。实际上读者可以将其等效为从目标机登陆到了一个 FTP 服务器上，所以通常的 cd，ls，以及打开一个文件，读写文件等操作都可以对网络设备使用。
我们将其称之为（网络）设备主要是因为从目标机操作系统来看，对其操作方式上就相当于对本地的一个（块）设备进行操作，然而本质上其底层驱动要比块设备驱动简单的多，其（netDrv）仅仅对网络通信进行了封装，没有做任何优化。正如前文所述，对于 netDrv 机制下的网络设备，当对一个文件进行操作时，这个文件将首先从远程主机完全复制到本地 RAM 中，在文件关闭之前所有的操作都是对本地这个副本进行的，最后当文件被关闭时，这个副本才被写回到远程主机上。底层实现上十分简单，仅仅对 socket 通信进行了封装而已。虽然效率不高，但是对于用户而言，这种机制十分有用，在开发的早期基本上都在依赖网络设备进行程序的调试。

默认情况下，Vxworks 内核启动过程后将根据 bootline 中的如下信息：远程主机 IP，远程主机名，用户名和密码等信息创建一个网络设备，并将操作系统根目录设置为指向这个网络设备，实际上就是将 bootline 中指定的用户在远程主机 FTP（通常都是使用 FTP）服务器上的根目录作为了操作系统启动后的系统默认根目录。
注意：如果目标机 Vxworks 操作系统包含了 FTP 服务器，那么这个 FTP 服务器的根目录就是通过 ioDefPathSet 设置的路径。如果用户不作改变，那么这个根目录就是网络设备，换句话说，就是远程主机 FTP 服务器上 bootline 中指定用户的根目录。如果用户需要将目标机上本地 FTP 服务器根目录该为本地文件系统中的某个目录，调用 ioDefPathSet 函数重新设置一下即可。

最后需要注意一点的是网络设备与网络文件系统是两个完全不同的概念，Vxworks 下网络设备由 netDrv 负责驱动，其直接工作在 IO 子系统下，较为简单；而网络文件系统则由 nfsDrv 驱动，其虽然也是直接工作在 IO 子系统下，但是其底层使用网络文件系统协议，与 netDrv 实现完全不同。由于网络文件系统需要远程主机对其进行配合，一般在嵌入式系统下使用较少，netDrv 提供的功能基本可以满足网络传输文件的需要，且远程主机仅需要启动一个 FTP 服务器即可完成目标机上网络设备的创建，使用上较为方便。

5.4 文件系统支持

文件系统是操作系统实现的一种重要组成部分。文件系统在原始块设备之上提供了一个视图，可以让用户以文件和目录的方式对块设备进行操作，而同时屏蔽了块设备底层复杂的驱动方式。基本上只要操作系统提供任务支持，其同时也就必须提供文件系统支持。文件系统实现是一个复杂和要求巧妙设计的内核组件，其并非简单的一堆数据结构，由于块设备的访问时间相对 CPU 运行速率而言都比较大，故内核文件系统层必须提供内存缓冲区之类的机制对块设备数据进行缓存，这些缓存块的管理，分配，块中数据的刷新以及刷新时机选择等等都需要仔细的设计。文件系统层设计优劣将直接影响整个操作系统的性能。本节我们并不对文件系统实现本身进行讨论，主要是简单介绍一下 Vxworks 提供的几种文件系统类型。在这些内核提供的文件系统中，最常用的是 MS-DOS 兼容型文件系统。在本书块设备驱动一章，我们将较为详细介绍文件系统的层次性以及文件系统与底层块设备驱动之间如何进行交互和一个用户层文件操作请求是如何通过 IO 子系统到达文件系统层，又进而到达块设备驱动层的。

5.4.1 虚拟根文件系统 VRFS

Vxworks 操作系统中一般无需包含对 VRFS 的支持，只当应用层需要一个 POSIX 根文件系统支持时，才需要在内核组件中包括对 VRFS 的支持。VRFS 仅仅提供一个根目录，为其他文件系统提供挂载点。当然无 VRFS 时，可以直接将一个其他类型的文件系统设置为根目录。
注意：由于 Vxworks 内核对于设备特殊的管理方式，所有的设备都被一视同仁的挂接在系统设备列表中。当用户打开一个文件时，将以文件路径名匹配系统设备列表中的设备名，从这个角度来看，根本不需要一个根目录。根目录的意义是针对一个任务上下文而言，每个任务在创建之时都被设置为一个根目录，当在任务中以相对路径名打开一个文件时，此时内核就将根目录字符串添加到相对路径名之前，将这个合成后的绝对路径作为一个字符串去匹配系统设备列表中的设备名称。
另外 Vxworks 操作系统本身并不需要 VRFS 的支持，通常情况下，应用层也无需 VRFS 的支持，所以 VRFS 只作为一个选项提供，通常我们无需包含该选项。如果用户需要在内核中包含对 VRFS 的支持，必须包含 INCLUDE_VRFS 宏定义。

5.4.2 事务（transactional）型文件系统 HRFS

事务型文件系统又被称为高可靠性文件系统，该类型文件系统将用户的每个操作记录在案，一旦发生突然掉电的情况，系统回滚到起先一个一致的状态，工作方式有些类似于数据库。除了保证数据的高可靠性，其对文件和目录的管理方式和操作如同一般的文件系统（如 dosFs）。
用户必须进行如下宏定义以在内核中包含对 HRFS 的支持。
INCLUDE_HRFS：HRFS 核心库支持。
INCLUDE_HRFS_FORMAT：HRFS 格式化功能支持。
INCLUDE_HRFS_CHKDSK：HRFS 文件系统完整性检查功能支持。

注意：Vxworks 内核早期版本（如 5.5）尚无对 HRFS 文件系统组件的支持。

5.4.3 MS-DOS 兼容型文件系统 dosFs

dosFs 是 Vxworks 内核中历史最悠久的一种文件系统，Vxworks 早期版本就包含对 dosFs 文件系统的支持，其使用范围也最广。dosFs 支持层次性文件和目录管理方式，支持 VFAT 长文件名，支持 FAT12，FAT16，FAT32 文件格式。
不同 Vxworks 内核版本支持 dosFs 文件系统的宏定义存在差别，且 dosFs 层与底层块设备之间的缓冲层组件也有所不同。对于 5.5 版本等早期版本，使用如下宏定义在 Vxworks 内核中包含对 dosFs 文件系统的支持。
INCLUDE_DOSFS_MAIN：全局控制宏，当且仅当首先包含对该宏的定义，其他子宏定义

才有意义。在包含 INCLUDE_DOSFS_MAIN 宏定义后，内核将初始化 dosFs 基本组件，向 IO 子系统注册 dosFs 文件系统中间驱动层。
如下功能包含必须定义有相应的子宏定义。
INCLUDE_DOSFS_FMT：包含 dosFs 格式化功能组件。
INCLUDE_DOSFS_CHKDSK：包含 dosFs 文件系统完整性检查功能组件。
INCLUDE_DOSFS_DIR_VFAT：包含 VFAT 目录处理功能组件。
INCLUDE_DOSFS_DIR_FIXED：包含 VxLong 目录处理功能组件。

dosFs 是使用最为广泛的文件系统，由于块设备访问速度较慢，故内核对于块设备的访问都提供一个缓存机制，在内存中专门开辟一段区域用以保存从块设备读取的数据，程序中对块设备的写入操作也是先将数据写入内存缓冲区域中，由专门的内核任务负责对这些已修改缓冲区进行刷新，即将修改后数据写入块设备中。
老版本 Vxworks 内核（如 5.5）使用 CBIO（Cached Block I/O）辅助组件作为缓存中间层，新版本 Vxworks 内核（如 6.4）使用 XBD（eXtened Block Device）作为缓存中间层。此处以 CBIO 为例介绍其使用方法。
对于 dosFs，rawFs，用户必须在内核包含对 CBIO 的支持，即需要定义 INCLUDE_CBIO 来包含 CBIO 的核心组件。如图 5-8 所示层次结构。

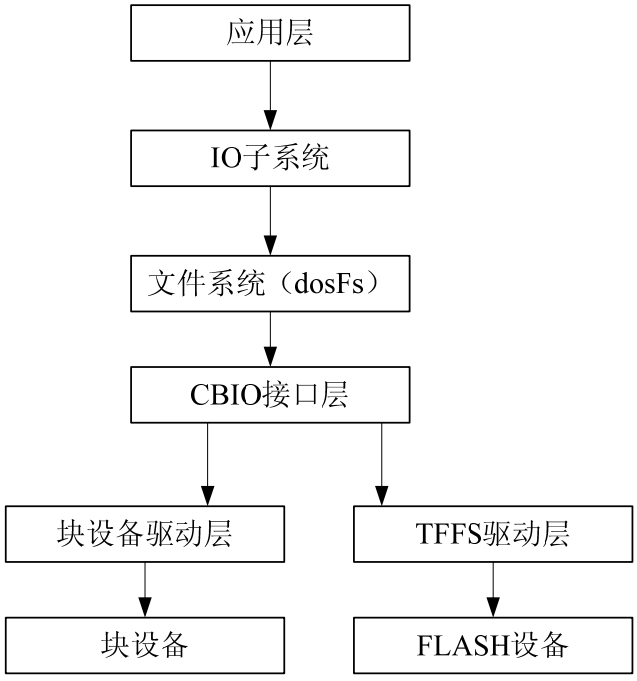


图 5- 8 CBIO 接口层在内核驱动层次中的位置

INCLUDE_CBIO 只是将 CBIO 直接与底层块设备驱动交互的子层包含进内核，为了包含对数据 cache 的支持，提供块设备的使用效率，用户还必须包含如下宏定义。
INCLUDE_DISK_CACHE：块设备缓存管理组件。
INCLUDE_DISK_PART：块设备分区工具组件。

注意：块设备缓存管理组件使用 INCLUDE_CBIO 对应的 CBIO 子层，CBIO 子层提供了一系列功能函数供块设备缓存管理组件调用，实际上图 5-8 中 CBIO 接口层可以继续细分为三个层次：分区管理层，数据缓冲层和 CBIO 基本功能层。

5.4.4 原始（raw）文件系统 rawFs

rawFs 即将整个块设备作为单个文件进行操作，不存在文件和目录层次性管理功能。对于一个挂接的块设备，如果 Vxworks 无法辨别其已有文件系统类型，则就以 rawFs 作为该块设备的默认文件系统。使用 rawFs 文件系统的块设备在打开时只需指定块设备名，不要在其后添加文件或者目录层次。系统内所有的任务都可以打开一个 rawFs 类型的块设备进行读写，可以通过控制文件偏移指针控制读写的位置。

用户需要定义 INCLUDE_RAWFS 宏在内核中包含 rawFs 组件。

rawFs 如同 dosFs 文件系统一样，Vxworks 早期版本（如 5.5）就已经对其提供支持。rawFs 在某些情况下十分有用，如下代码示例演示了 rawFs 文件系统的创建。

```
RAW_VOL_DESC *pVolDesc;
BLK_DEV *pBlkDev;
pBlkDev = ramDevCreate (0, 512, 400, 400, 0);
if (（pVolDesc = rawFsDevInit("/ramraw", pBlkDev) ）== NULL)
{
printErrno( );
};
```

我们首先利用内存模拟了一个块设备，其后使用 rawFsDevInit 函数在其上创建一个 raw 文件系统。下面就可以使用 open 函数打开该设备进行读写了。

```
int rawFd;
rawFd = open(“/ramraw”, O_RDWR, 0);
if(rawFd<0){
      printErrno();
      return;
}
write(rawFd, “hello”, 5);
close(rawFd);
```

5.4.5 CD-ROM 文件系统 cdromFs

cdromFs 是 Vxworks 提供的从标准 ISO9660 格式 CD 上读取数据的内核组件。Vxworks 可以使用 cdromFs 将一个 CD 设备挂载到系统中，进而从 CD 设备中读取数据。Vxworks 老版本中（5.5）尚未提供对 cdromFs 的支持，且虽然新版内核中（如 6.4）包含了对 cdromFs 组件的支持，但是基于 Vxworks 的使用环境一般是嵌入式系统，故 cdromFs 很少使用，故此处不再讨论，感兴趣读者请参考文献 vxworks kernel programmer’s guide，7.7 节内容。

5.4.6 只读文件系统 ROMFS

ROMFS 是 Vxworks 提供的一种非常有用的机制，可以将任何格式的文件存储在 ROMFS 下将其编入内核，作为内核映像的一部分。在内核启动过程中，这个 ROMFS 文件系统将被载入 RAM 中，此时内核其他组件可以以普通文件操作方式对 ROMFS 文件系统中的文件或者目录进行操作。这种工作方式有些类似于 Linux 下的 Initial RamDisk。注意：ROMFS 是指只读（Read-Only）型的文件系统，其与 ROM 存储介质毫无关系，事实上，这个文件系统只存在于 RAM 中。虽然在 Vxworks 内核编译过程中被作为内核的一部分编入内核映像中，但是其整体上还是作为一个独立的部分存在，并不与内核代码进行链接之类的操作。
用户必须定义 INCLUDE_ROMFS 使得内核包含对 ROMFS 的支持。此外还需要加进行如下配置。
A． 在 Vxworks 内核工作目录下创建一个 ramfs 目录。注意目录名必须是 ramfs。
B．将需要包含进 ROMFS 的任何文件复制到 ramfs 目录下。
C．重新生成 Vxworks 内核映像。
如下示例。

```
cd c:\Tornado2.2target\projs\arm926ejs_proj //进入到内核工程目录下
mkdir romfs   //创建 ramfs 目录
copy c:\allMyVxApps\myDemoApp.out romfs //拷贝文件到 ROMFS 文件系统中
copy c:\allMyVxApps\kernelParams.dat romfs
make TOOL=gnu //重新生成内核
```

除了在内核工程目录下直接创建一个名为 ramfs 的目录外，编译环境提供 ROMFS_DIR 变量用以表示 ROMFS 文件系统根目录地址，如下所示。

```
make TOOL = gnu ROMFS_DIR = ”c:\allMyVxApps”
```

Vxworks 内核启动过程中，这个 ROMFS 文件系统被拷贝到 RAM 中，并以 ROMFS 文件系统的形式挂载到系统中，设备节点名为“/ramfs”。用户在 shell 下使用 devs 命令时，将显示该名称。使用‘ls “/ramfs”’命令可以查看“/ramfs”根目录下的内容，对于以上的示例，其将显示 myDemoApp.out，kernelParams.dat 两个文件。可以通过标准的文件操作接口函数（open，read）对 ROMFS 下的文件进行操作。
如下示例。

```
int fd;
fd=open(“/ramfs/kernelParams.dat”, O_RDONLY, 0);
read(fd, parmBuf, parmSize);
…
```

ROMFS 文件系统作为 Vxworks 内核映像的一部分编入映像中，在 Vxworks 操作系统启动过程中被拷贝到 RAM 中并以一个文件系统挂载到内核中供其他组件访问，无需借助任何物理的磁盘设备或者网络连接，为某些特殊应用场合提供了一种选择。其与 ramDisk 有些类似，但是 ramDisk 只能在 Vxworks 操作系统启动完成后即时创建，而 ROMFS 则是在编译内核映像的过程中创建，可以通过 ROMFS 传递任何需要传递给 Vxworks 内核的批量参数。

5.4.7 目标机文件系统 TSFS

TSFS 即通过网络将主机一个文件系统挂载到目标机系统下。注意 TSFS 不是网络文件系统（NFS），TSFS 使用 WDB 目标机引擎本地驱动与主机上的目标机服务引擎进行通信。目标机上所有的请求将由本地驱动通过网络传递给主机服务引擎。
用户必须定义 INCLUDE_WDB_TSFS 宏使得内核包含对 TSFS 的支持，此时内核将以“/tgtsvr”设备名创建一个 TSFS 文件系统。
TSFS 文件系统工作机制类似于网络设备驱动 netDrv，只是二者在网络通信使用协议上有些差别，netDrv 使用 RSH（无密码时）和 FTP（有密码时），而 TSFS 使用较为底层的协议（如 TCP 协议）在目标机（由 WDB 负责）与主机（target server）之间进行信息的交互。另外一点不同之处是 netDrv 直接在 IO 子系统管理之下，且 netDrv 本身即负责与主机（FTP）服务器的通信，而 TSFS 属于文件系统，其层次相对较为复杂。
有关 TSFS 的更多内容，请读者参考 vxworks kernel programmer’s guide 以及 vxworks programmer’s guide 相关章节。

5.5 添加驱动到内核

添加驱动到内核主要包括两个方面的内容：（1）将代码编入内核映像；（2）注册驱动和创建设备。

（1） 将驱动代码编入内核映像
根据驱动所实现的功能，驱动的使用时机基本可以分为三个时间段：
A.非压缩代码执行期间；
处于这个时间段工作的驱动一般较少，但是也是一种可能。此种情况下，驱动代码必须也是非压缩的。所以驱动代码编入内核映像的方式不能简单的将其包括进 sysLib.c 文件中。而是要通过 BOOT_EXTRA 宏来完成。BOOT_EXTRA 指向的代码在编入内核映像时将不进行压缩，而是与 romInit，romStart 函数一样作为非压缩部分存在于内核映像中。在 romInit 函数执行期间可以被调用完成对某个外设的操作。
注意：这种情况下由于操作系统代码尚未得到执行，故驱动不受任何其他组件管理，底层驱动函数将直接被调用，不经过任何中间层（如 IO 子系统）。同时当然也无需驱动注册和设备节点创建。

B.压缩代码执行期间；
压缩代码最终将由 romStart 函数完成解压缩的工作，最后 romStart 跳转到这些代码运行，真正开始内核的初始化过程，直到 usrRoot 函数执行完毕，完成 Vxworks 操作系统的启动都可以视作压缩代码的执行过程。一般在这个阶段驱动将被内核其他组件使用，而且通常而言，必须在 IO 子系统完成初始化之后才对这些驱动进行初始化。对于这个阶段使用的驱动可以通过如下两种方式将驱动代码编入内核映像之中。
[1] 将驱动源文件使用 include 语句直接加入 sysLib.c 文件中。如#include “spi.c”。

[2] 在 Makefile 中使用 MACH_EXTRA 包含驱动代码。如 MACH_EXTRA=platform.o spi.o。注意：以上两种方式不可同时使用。

C.操作系统系统启动完成后。
在这个阶段使用的驱动编入内核映像的方法与 B 情况相同。

（2） 注册驱动和创建设备
驱动注册和设备创建通常是指通过调用 iosDrvInstall 和 iosDevAdd 将驱动和设备分别添加到系统驱动表和系统设备表的过程。根据设备使用范围，这个注册和创建的时机也不同。一般而言对于标准设备如串口，驱动注册和设备创建在内核初始化过程中完成；而对于一些驱动中间层，驱动注册主要在内核初始化中完成，而设备创建需要等到下层驱动层注册之时完成。一个典型的例子就是 FLASH 设备驱动，位于其上的 TFFS 中间层和文件系统中间层在内核初始化过程中即已完成注册，而设备的创建可以在操作系统启动后任意时刻进行创建（一般通过调用 usrTffsConfig 函数）。另一些只在某些特殊场合使用的驱动如 memDrv，则在真正使用时才进行驱动注册和设备创建。
按照约定，驱动注册一般在形如 xxxDrv 之类的函数中完成，在设备创建则在形如 xxxDevCreate 的函数中完成。根据是否直接受内核 IO 子系统管理，xxxDevCreate 函数的功能又有所差别：（1）如果直接受 IO 子系统管理，xxxDevCreate 通常都在函数实现的最后通过调用 iosDevAdd 函数完成设备的创建；（2）如果通过中间层驱动管理，那么底层驱动中 xxxDevCreate 函数一般不进行 iosDevAdd 函数的调用，而是对一些中间层驱动提供的数据结构进行初始化，再由中间层驱动根据底层驱动返回的这个结构进行设备的最终创建。换句话说，设备的创建一般都是在 IO 子系统紧接着的下一层进行，即直接受 IO 子系统管理的驱动层（无论是直接的底层硬件驱动还是一个中间层驱动）上才进行设备节点的创建（即将设备添加到系统设备列表中）。

5.6 本章小结

本章我们详细介绍了 Vxworks 下内核驱动层次，着重对 IO 子系统进行了介绍。Vxworks 内核使用三张表对系统内所有的驱动，设备以及打开文件进行管理。这三张表是 Vxworks 内核驱动层次的核心。Vxworks IO 子系统在驱动层次中的作用十分重要，系统内所有的驱动直接或者间接地受 IO 子系统的管理，所有的用户请求都必须通过 IO 子系统进行传递。对于较为复杂的设备（如磁盘设备，FLASH 设备，USB 设备），为了简化底层硬件驱动的设备复杂度，Vxworks 内核专门提供了驱动中间层与这些底层驱动接口，此时驱动中间层直接受 IO 子系统的管理，而底层驱动则委托给这些中间层进行管理。
在对 Vxworks 驱动层次进行介绍后，我们进一步介绍了 Vxworks 内核本身提供的一些设备驱动支持，这些驱动有些是中间层驱动，有些则直接驱动一个虚拟设备工作，如管道设备，虚拟内存设备，ramDisk 设备。对于这些虚拟设备的基本功能和使用方法都进行了比较详细的介绍。网络设备虽然并不属于虚拟设备，但是从目标机的的角度而言，其也可以看做一个“没有（本地）存储介质“的虚拟设备。网络设备在开发阶段十分有用，文中对其基本功能和使用方法也一并进行了介绍。
之后对 Vxworks 下文件系统的支持进行了简单说明，文件系统在内核驱动层次中实际上作为块设备驱动层次中的一个中间层而存在的，其向 IO 子系统进行注册，而将底层块设备驱

动置于自身的管理之下以提供数据访问的效率。在这些文件系统中，dosFs 和 rawFs 是最为常用的两种文件系统类型，在 Vxworks 早期版本就包含对这两种文件系统的支持，故对这两种文件系统我们进行了较为详细的介绍，对于其他文件系统类型一方面由于其使用范围相对较小，且老版本内核（如 5.5）并不包含对这些文件系统的支持，故介绍的较为简单，感兴趣用户可以参考文献 vxworks kernel programmer’s guide 和 vxworks programmer’s guide 的相关章节进行了解。

本章最后我们介绍了如何将驱动代码添加到内核映像之中以及驱动注册和设备创建的时机选择。对于一个直接受 IO 子系统管理的底层驱动，驱动注册和设备创建的时机比较确定，而受中间驱动层管理的底层驱动则需要根据实际情况进行选择。总之基本原则就是不影响对驱动的使用。

在完成内核驱动层次相关内容的介绍后，从下一章开始，我们将详细分析各种设备类型的具体驱动编程细节。基本上所有的设备驱动都直接或间接地与 IO 子系统有着联系，尤其是对系统三张表作用的深刻理解将非常有利于对底层驱动结构的理解。

第六章 字符设备驱动

字符设备是一类比较简单的设备，其以字节流的方式对数据进行操作，数据只能顺序的读写。I2C，SPI，UART 等等接口类型的设备都可以作为字符设备进行驱动。字符设备位于 Vxworks 内核 IO 子系统直接管理之下，不经过任何中间层。值得注意的是，内核为了简化某些常用字符设备驱动的设备，也会提供一个中间层作为缓冲，如 TTY 中间层。用户可以自行选择是否使用这些中间层。一般而言，对于 UART 设备，建议读者使用内核提供的 TTY 中间层，以提高设备使用效率，因为 UART 串口设备在内核启动过程中默认的被设置为标准输入输出通道，在整个操作系统运行期间使用比较频繁。
对于 I2C，SPI 接口，其对应的设备可以是一个块设备如 EEPROM，也可以是一个类似于 UART 之类与外界进行数据交互的设备，都可以当做字符设备进行驱动。本章即以一个 SPI 接口设备 VK3224 为例，详细介绍字符设备驱动的各个方面。

从本书前文可知，Vxworks 内核 IO 子系统维护着三张系统表：系统驱动表，系统设备表和系统文件描述符表。这三张表构成了 Vxworks 下所有驱动的管理中心。如图 6-1 所示为 Vxworks 下内核驱动层次图。

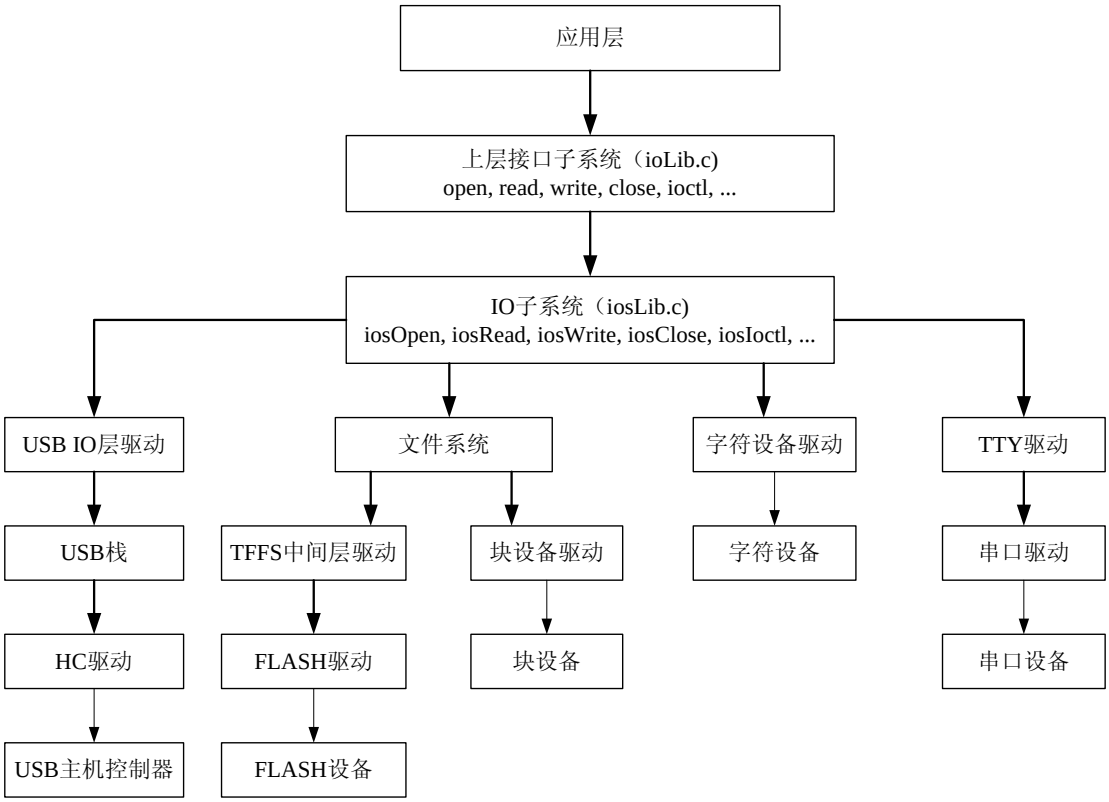


图 6- 1 Vxworks 内核驱动层次

上层接口子系统为用户提供了一套标准接口函数，用户可以无视底层设备类型，对所有的设备都使用这一套接口函数进行操作，由接口子系统之下的 IO 子系统根据设备类型调用不同的底层驱动进行响应。上层接口子系统提供的标准接口函数如下图 6-2 所示。

函数名	调用原型	说明
open	open(filename, flags, mode)	打开设备
creat	creat(filename, flags)	创建设备
read	read(fd, &buf, nBytes)	从设备读取数据
write	write(fd, &buf, nBytes)	向设备写入数据
ioctl	ioctl(fd, command, arg)	配置设备参数
close	close(fd)	关闭设备
lseek	lseek(fd, offset, whence)	移动文件指针
rename	rename(oldname, newname)	重命名设备
remove	remove(filename)	删除设备

图 6- 2 标准接口函数

底层驱动通过 iosDrvInstall 函数向内核 IO 子系统注册自己的驱动函数集合，iosDrvInstall 函数调用原型如下。

```
int iosDrvInstall
    (
    FUNCPTR pCreate,      /* pointer to driver create function */
    FUNCPTR pDelete,      /* pointer to driver delete function */
    FUNCPTR pOpen,         /* pointer to driver open function */
    FUNCPTR pClose,       /* pointer to driver close function */
    FUNCPTR pRead,         /* pointer to driver read function */
    FUNCPTR pWrite,       /* pointer to driver write function */
    FUNCPTR pIoctl        /* pointer to driver ioctl function */
    );
```

可以看到，底层驱动对用户层 open，creat，read，write，ioctl，close，remove 调用都具有对应的响应函数，而 lseek，rename 两个函数事实上都间接地调用 ioctl 函数，换句话说，底层实现上实际上对用户层所有可能的操作都提供了响应函数。

当然用户通过调用图 6-2 所示的标准函数接口发出一个服务请求时，这个请求首先将由内核 IO 子系统进行处理，在完成 IO 子系统层的处理后，才将请求进一步传递给底层驱动进行请求的具体响应。IO 子系统在其中起着非常关键的承上启下的作用：其对下管理驱动函数，维护设备列表，管理当前系统打开的所有文件句柄；其对上对用户层维护统一的接口。整个系统因为 IO 子系统的存在而层次分明。

为了使得底层设备对用户层可见，进而是设备可被用户使用，底层驱动必须完成两个关键的初始化工作：（1）向 IO 子系统注册驱动；（2）通过 IO 子系统将设备添加到系统设备列表中。内核 IO 子系统也专门提供了 iosDrvInstall 和 iosDevAdd 两个接口函数供底层驱动使用。图 6-3 显示了 IO 子系统与底层驱动之间的函数调用关系。

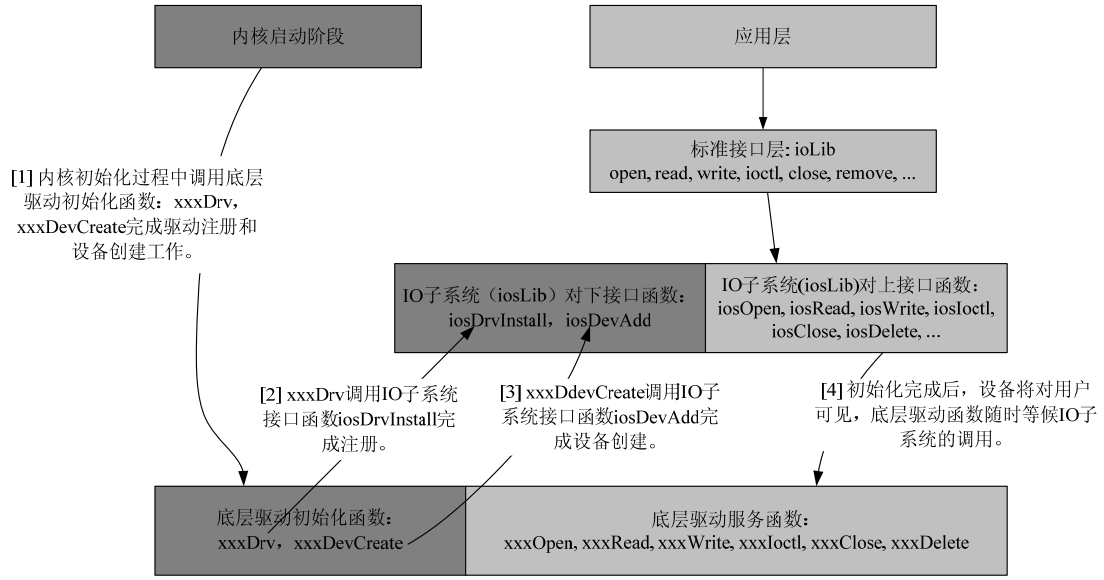
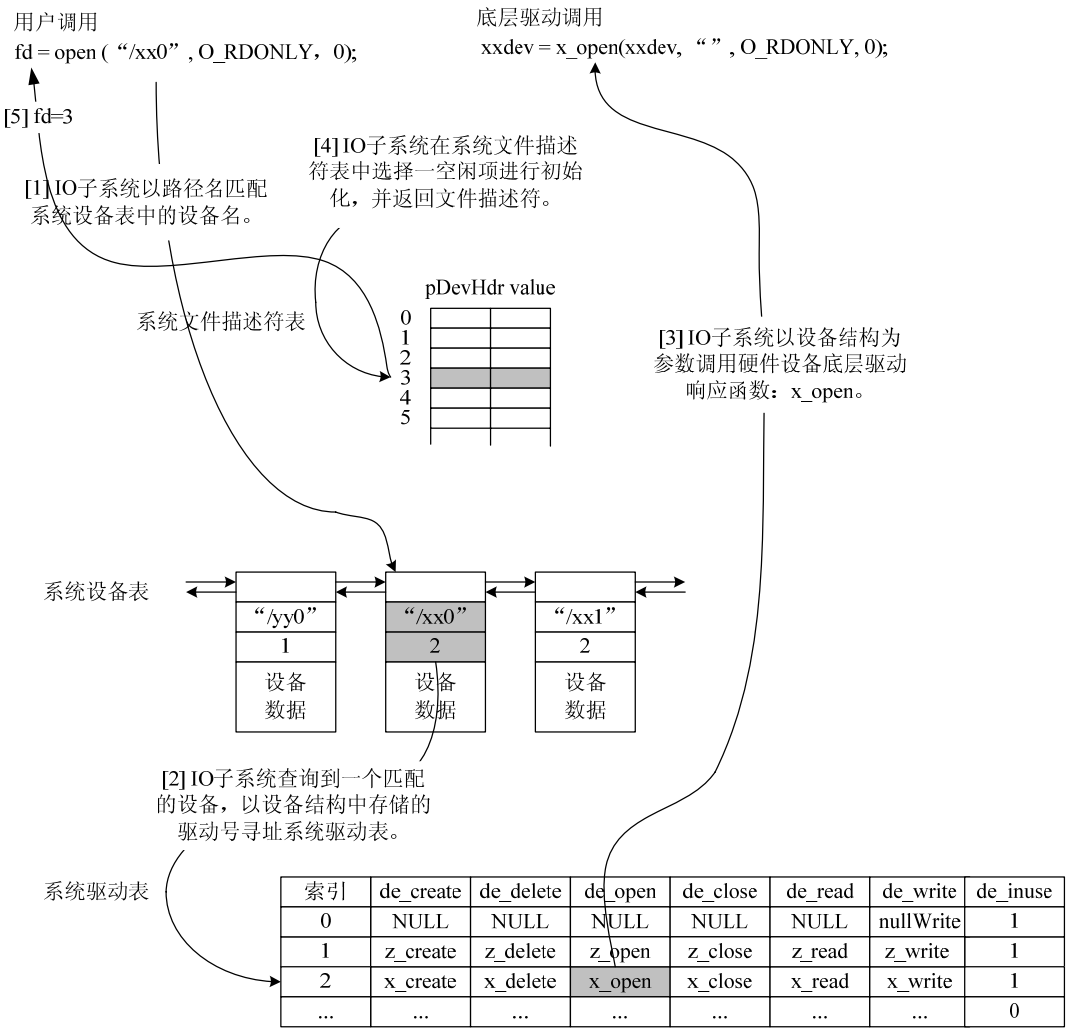


图 6- 3 底层驱动与 IO 子系统之间的函数调用关系



索引	de_create	de_delete	de_open	de_close	de_read	de_write	de_inuse
0	NULL	NULL	NULL	NULL	NULL	nullWrite	1
1	z_create	z_delete	z_open	z_close	z_read	z_write	1
2	x_create	x_delete	x_open	x_close	x_read	x_write	1
...	...	...	...	...	...	...	0

图 6- 4 用户请求到底层驱动的传递过程

图 6-4 给出了一个用户 open 调用请求到底层驱动的传递过程。底层驱动 xxxDevCreate 进行设备创建时，在每个设备结构中都存储了该设备的驱动号（xxxDrv 函数调用时产生），IO 子系统可根据设备列表中设备结构直接查询到该设备对应的驱动程序，如图 6-4 中所示，当 IO 子系统以文件路径名在系统设备列表中匹配到一个设备时，其直接根据存储在设备结构中的驱动号（图中驱动号为 2）在系统驱动表中获得对应设备驱动，并调用 open 底层驱动响应函数 x_open，完成用户层打开设备请求。

IO 子系统在调用底层驱动函数时，将使用设备结构作为函数调用的第一个参数。这个设备结构是由底层驱动自定义的，用以保存底层硬件设备的关键参数。对于自定义的设备结构必须遵循内核的约定：即将内核结构 DEV_HDR 作为自定义设备结构的第一个成员，该成员被 IO 子系统使用统一表示系统内的所有设备。换句话说，IO 子系统将所有驱动自定义的设备结构作为 DEV_HDR 结构使用，DEV_HDR 结构之后的字段由各自具体的驱动进行解释和使用，内核不关心这些字段的含义。

下面我们以一个 SPI 接口设备 VK3224 为例一步一步介绍如何编写一个字符设备驱动。
VK3224 是一个 SPI 接口的 4 通道 UART 串口扩展器件，其使用 SPI 从接口与主机通信，扩展成 4 个 UART 输出口。从主机驱动端来看，VK3224 就是一个 SPI 从设备。VK3224 SPI 接口特性如下：（1）最高速度 5Mbits/s；（2）仅支持 SPI 从模式；（3）16 位，SPI 模式 0.
SPI 是 Serial Peripheral Interface 的简称，是当前比较常用的一种串行接口。操作上由四根信号线构成：数据输出线 SDOUT，数据输入线 SDIN，片选 SCS，时钟 SCK，是一种全双工的通信工作模式：在主端发送数据的同时，从端也在发送数据，可以将主从通道想象成如下图 6-5 所示的两个环形移位寄存器。

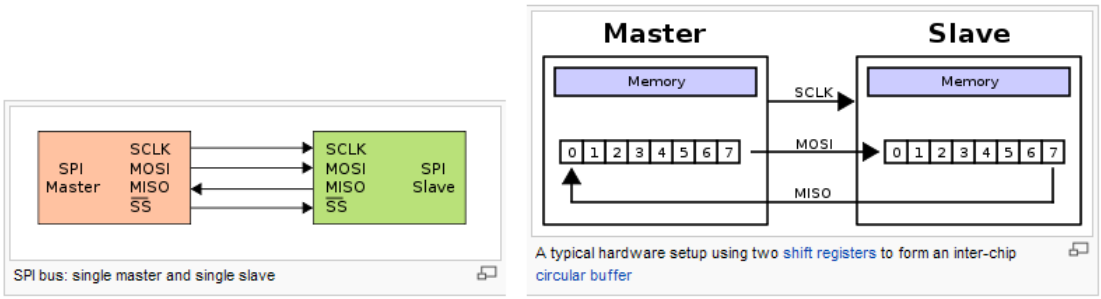


图 6- 5 SPI 接口工作方式

SPI 共有四种工作模式，由 CPOL，CPHA 两个控制位进行选择。
CPOL 控制在 idle 状态时的时钟电平：CPOL=0，时钟 idle 电平为低电平；CPOL=1，时钟 idle 电平为高电平。所谓 idle 电平即平时不进行数据传输时的时钟电平。
CPHA 控制数据和时钟之间的延迟：CPHA=0，无延迟；CPHA=1，数据相对时钟上升沿提前半个周期出现在总线上。
由于 SPI 并非作为一个标准规范，不同产商的 SPI 设备对四种模式的定义可能所有区别，但是大多数产商都遵守以上的约定。
VK3224 只能工作在 SPI 模式 0 下，即主机端作为时钟供应方和控制方必须设置 CPOL=0，CPHA=0. 对于 SCS 片选信号，SPI 约定在只有一个从设备的情况下，可以从硬件上直接将从设备 SCS 片选管脚做接地处理，但是 VK3224 不可以作如此处理，即便系统中只有一片 VK3224 作为从设备，也必须对 vk3224 SCS 片选信号线进行控制，平时不进行数据传输时，

必须将 VK3224 SCS 片选信号线做拉高处理，这一点非常重要，否则 VK3224 数据输出管脚将一直为低电平，芯片无法正常工作。
另外一个值得注意的地方是 vk3224 本身支持 2.5V，3.3V，5V 供电，在 5V 供电时，其要求的输入高电平最小电平为 3.6V，当前许多平台上 IO 接口标准都是 3.3V，而且像 SPI 接口之类的串行信号线一般都通过 10K 电阻上拉到 3.3V，换句话说，这些信号线上最大电平是绝对不会超过 3.3V 的，如果使用 5V 对 VK3224 供电，那么无论 IO 管脚输出电平是高是低，对于 VK3224 而言都是 0 电平。这一点要非常注意：主机 IO 接口电平与接口芯片的电平必须匹配，这一点不单是针对 VK3224 芯片而言。
我们将在主机端使用 GPIO 模拟 SPI 工作时序对 VK3224SPI 从接口进行操作。由于 VK3224 通过 SPI 接口可以扩展为四个 UART 串口，此处能够说明问题，同时保持代码尽量简单，我们只使用其中两个 UART 输出通道：通道 1 和通道 2。VK3224 对四个 UART 通道内部有不同的寄存器映射，故此种情况下，我们只需对通道 1 和通道 2 对应的寄存器进行控制即可。此时在驱动初始化过程中，我们需要创建两个串口设备与这两个通道对应。

6.1 定义设备结构

所有的底层驱动都要对其驱动的硬件设备维护一个结构，用以保存设备的关键参数：设备寄存器基地址，中断号，设备硬件缓存区基地址等等。这些信息将随着设备类型的不同而有所差别。对于我们此处要驱动的 VK3224，我们仅需要保存如下信息： GPIO 模块寄存器基地址。
我们在主机端直接使用 GPIO 管脚模拟 SPI 工作时序对 VK3224 进行操作。SPI 接口需要四根信号线，故我们需要预留四个 GPIO 管脚。GPIO 管脚通常与平台上其他器件复用，所以一般需要对 GPIO 模块相关寄存器进行配置，首先将其配置为 GPIO 管脚，其次配置其输入输出方向，初始值以及是否上拉。
VK3224SPI 接口底层驱动设备结构定义如下：

```
typedef struct _spi_dev{
    DEV_HDR    pDevHdr;
    UINT32     gpioRegBase;
    UINT8      channel;
    UINT8      flags;
}SPI_DEV;
```

DEV_HDR 结构定义在内核头文件 h/iosLib.h 中，如下。

```
typedef struct      /* DEV_HDR - device header for all device structures */
{
    DL_NODE    node;          /* device linked list node */
    short      drvNum;        /* driver number for this device */
    char *     name;          /* device name */
} DEV_HDR;
```

注意：自定义设备结构的第一个成员必须是 DEV_HDR 结构类型。SPI_DEV 自定义结构仅对驱动本身有效，对于内核 IO 子系统而言，其将所有的设备结构都看作为 DEV_HDR 类型，

内核仅对 DEV_HDR 结构进行管理，在系统设备列表中，内核只使用 DEV_HDR 结构中成员，对于自定义结构中的其他成员的含义将由具体的底层驱动进行解释和使用，内核对此不关心。

以上针对 SPI 设备的结构定义中，我们添加了三个自定义成员，因为我们直接使用 GPIO 进行控制，所以需要 GPIO 控制寄存器基地址；其次我们还需要字段用以表示通道号，因为底层驱动相当于在驱动两个串口（虽然是通过共同的 SPI 接口与主机交互数据），所以我们需要一个字段指示当前是对哪个串口进行操作；再次我们需要保存特定通道的操作类型，这个由 flags 成员表示。对于一个复杂的字符设备，其自定义成员可能有很多。

对于 GPIO 模拟 SPI 接口时序的代码，我们预先给出。这与特定平台相关，如下代码基于 S3C2440 平台。

```
#define SPISPD    (100000000)
GPIO_REG *gpioReg=(GPIO_REG *)GPIO_REG_BASE;
CLKPW_REG *gCLKPWReg=(CLKPW_REG *)CLKPW_REG_BASE;
//GPIO 管脚配置，该函数将在 spiDevCreate 函数中被调用。
void spi_gpio_init(){
    //mosi,miso; mosi-output, miso-input
    gCLKPWRreg->rCLKCON &= ~(1<<18);
    gpioReg->rGPECON = (gpioReg->rGPECON & ~(0x3F<<22)) | (0x4<<22);
    gpioReg->rGPEUP |= (0x3<<11); //use external pullup, disable gpio pullup
    gpioReg->rGPEDAT &= ~(0x3<<11); //initial low
    //clk
    gpioReg->rGPGCON = (gpioReg->rGPGCON & ~(0x3<<4)) | (0x1<<4); //output
    gpioReg->rGPGUP |= (0x1<<2); //use external pullup, disable gpio pullup
    gpioReg->rGPGDAT &= ~(0x1<<2); //spi mode 0, sck inactive-low
    //cs
    gpioReg->rGPFCON = (gpioReg->rGPFCON & ~(0x3<<2)) | (0x1<<2); //output
    gpioReg->rGPFUP |= (0x1<<1); //use external pullup, disable gpio pullup
    gpioReg->rGPFDAT |= (0x1<<1); //chip select, inactive-high
}
inline void delay(unsigned int t)
{
    int i;
    while(t--){
        i=t-i;
    }
}
//GPIO F1 用作片选 SCS
void scs(int d)
{
    if (d)
        gpioReg->rGPFDAT |= (1<<1);
    else
        gpioReg->rGPFDAT &= ~(1<<1);
```

```
}
//GPIO E12 用作 MOSI
void mosi(int i)
{
	if(i)
		gpioReg->rGPEDAT |= (1<<12);
	else
		gpioReg->rGPEDAT &= ~(1<<12);
}
//GPIO E11 用作 MISO
int miso()
{
	return (gpioReg->rGPEDAT >>11)&0x1;
}
//GPIO G2 用作时钟 SCK
void clk(int i)
{
	if(i)
		gpioReg->rGPGDAT |= (0x1<<2);
	else
		gpioReg->rGPGDAT &= ~(0x1<<2);
}
//SPI 接口读写函数，每次 16-bit。
unsigned short spi_bitbang(unsigned short dat){
	int i;
	scs(0);
	for(i=0;i<16;i++) {
		mosi(dat&0x8000);
		dat<<=1;
		delay(SPISPD);
		clk(1);
		dat|= miso();
		delay(SPISPD);
		clk(0);
	}
	scs(1);
	return dat;
}
```

底层驱动中将使用 spi_bitbang 函数对 VK3224 SPI 从接口进行读写，SPI 接口使用两根数据线进行数据的传输，SPI 主从接口之间数据的交互是全双工的，一方在写出数据的同时也在读入数据，主从双方构成一个环形寄存器。如上文中图 5 所示。

6.2 驱动注册和设备创建

底层驱动一般提供形如 xxxDrv 和 xxxDevCreate 之类的函数完成驱动注册和设备创建的工作。这些工作的完成一般是在内核启动过程中进行。当然并非一定如此，只要不影响用户层的最终使用，对于注册和设备创建的时机比较随意（其实完全可以等到用户要使用设备时，由用户自己调用 xxxDrv 和 xxxDevCreate 函数完成底层驱动的初始化）。对于此处的 SPI 驱动我们定义 spiDrv 和 spiDevCreate 两个初始化函数。

spiDrv 完成底层 SPI 驱动的注册，即创建一个 SPI_DEV 结构并对其进行初始化，最后调用 iosDrvInstall 函数将 SPI 驱动添加到系统驱动表中，完成注册。

该函数原型为 STATUS xxxDrv();

一般而言，xxxDrv 形式的函数名不带参数，如果需要参数，则函数名称定为如下形式：

STATUS xxxInit(int arg, …);

对于此处的 SPI 驱动，我们不需要参数，故函数基本实现如下。

```
//注意：Vxworks 下 LOCAL=static
LOCAL int spiDrvNum=-1; //定义一个整型全局变量，保存 iosDrvInstall 返回的驱动号。
STATUS spiDrv(){
    if(spiDrvNum!=-1) //如果 spiDrvNum 不等于-1，则表示驱动已经注册，此时直接返回。
        return (OK);

    spiDrvNum=iosDrvInstall(
                        spiOpen,    /*creat*/
                        spiDelete,  /*delete*/
                        spiOpen,    /*open*/
                        spiClose,   /*close*/
                        spiRead,    /*read*/
                        spiWrite,   /*write*/
                        spiIoctl);  /*ioctl*/

    return (spiDrvNum==ERROR?ERROR:OK);

}
```

spiDrv 函数实现“干净利落”，完成了其需要完成的工作：调用 iosDrvInstall 向 IO 子系统注册驱动本身，此时这个驱动被保存在系统驱动表中，等待设备与其衔接。这个衔接的工作交由 spiDevCreate 函数完成。

VK3224 可将一个 SPI 接口扩展成 4 个 UART 串口，底层驱动将根据当前用户打开的设备节点通过 SPI 接口驱动 VK3224 的特定通道。为了区分不同通道，我们需要多个设备分别表示被驱动的通道。底层驱动将底层实现细节对用户进行了屏蔽，对于用户而言，其并不知道底层硬件实现上实际是通过一个 SPI 接口扩展了四个串口，用户从上层看到的就是四个可用的普通串口，即系统有四个串口设备可用使用。所以我们对每个扩展 UART 口都创建一个对应的设备供用户使用。

注意：iosDrvInstall 函数的返回值被保存到 spiDrvNum 变量中，这是底层驱动自身维护的一

个全局变量，供驱动的其他部分代码使用，如 spiDevCreate 函数，这个函数在调用 iosDevAdd 函数添加设备时，需要指定设备对应的驱动程序驱动号，就是 spiDrvNum 中存储的驱动号。从而将设备与底层驱动进行“衔接”，当用户层对该设备进行操作时，可以调用我们这儿定义的底层驱动函数。

对于我们的 SPI 驱动，spiDevCreate 函数原型如下。

STATUS spiDevCreate（int channel）;

参数：通道号.

通道号参数可以让用户在创建设备时指定使用哪个通道。注意：此种方式下，如果用户需要同时使用多个通道，那么需要多次调用 spiDevCreate 函数。这是必要的，因为底层驱动无法得知用户究竟需要使用哪个通道，以及同时使用几个通道串口。正如本书在驱动程序概述一章中所述的，底层驱动只能提供一种机制，策略则由用户选择。

单独一个通道号并不能作为设备节点名，底层驱动将使用一个默认前缀“/spiUart”，即如果传入的通道号为 0，则创建的设备节点名则为“/spiUart/0”.

spiDevCreate 函数实现如下。

```
char *spiDevNamePrefix=”/spiUart”;
STATUS spiDevCreate(int channel){
    SPI_DEV *pSpiDev;
    char    devName[256];

    sprintf(devName, “%s/%d”, spiDevNamePrefix, channel);

    pSpiDev=(SPI_DEV *)malloc(sizeof(SPI_DEV));
    bzero(pSpiDev, sizeof(SPI_DEV));
    pSpiDev->channel=channel;
    pSpiDev->gpioRegBase=（UINT32）GPIO_REG_BASE;

    spi_gpio_init();   //配置 GPIO 模块。
    …   //调用 spi_bitbang 函数通过 SPI 接口配置 VK3224，初始化由 channel 指定的通道。

    //将设备添加到系统设备列表中。
    if(iosDevAdd(&pSpiDev->pDevHdr, devName, spiDrvNum) == ERROR){
        free((char *)pSpiDev);
        return (ERROR);
    }

    return (OK);
}
```

假设用户两次调用 spiDevCreate，并分别传入 0，1，表示使用 VK3224 通道 1 和通道 2，此时系统设备列表中将包含如下两个设备：”/spiUart/0”, “/spiUart/1”. 这两个设备使用相同的驱动。注意二者使用不同的设备结构句柄，”/spiUart/0”设备对应的设备结构中 channel 被初始化为 0，而”/spiUart/1”设备对应的设备结构中 channel 字段被初始化为 1，底层驱动函数被调用时，第一个参数就是设备的结构句柄，底层驱动据此可以确定此次操作是针对通道 1 还是

通道 2. 当然实际上 SPI_DEV 中包含的 DEV_HDR 结构中设备名也有所差别，也可以根据 DEV_HDR 结构进行判断，但使用 channel 字段更直接有效。
如果两次以相同的通道号对 spiDevCreate 函数进行调用，那么第二次调用将失败，即内核（IO 子系统）不允许创建两个相同名称的设备，即便他们使用不同的底层驱动（更何况是使用同一个底层驱动）。
现在驱动已经就绪，设备也已创建完毕，用户可以对其进行操作了。系统设备中现在经过 SPI 接口驱动的可用串口有两个，实际上对于用户而言，其并不知道底层硬件上是通过一个 VK3224 芯片将 SPI 接口扩展成的串口，用户可见的就是两个可用的如普通串口无异的 UART 口。例如用户需要对”/spiUart/1”进行读写操作测试。如下函数所示。

```
void main（void）{
    int fd;
    char *msg = “hello, world.\n”;

    fd=open(“/spiUart/1”, O_WRONLY, 0);
    if(fd<0){
        printErrno();
        return;
    }
    write(fd, msg, strlen(msg));
    close(fd);

    return;
}
```

用户运行该程序，查看 VK3224 通道 2 UART 输出口（可接至 PC 机串口，通过查看超级终端）是否显示“hello，world.”信息即可检测驱动是否正常工作。

当用户以”/spiUart/1”为路径调用 open 函数，这个请求首先被 IO 子系统截获，其以路径名匹配系统设备列表。由于之前已经以参数 1 调用了 spiDevCreate 函数，故 IO 子系统在系统设备列表中发现了一个匹配项，获得设备的驱动号 spiDrvNum，根据该驱动号调用底层 SPI 驱动中 spiOpen 函数。这个过程如上文中图 4 所示。

下面我们分析底层驱动中通过 iosDrvInstall 函数注册到系统驱动表中的一系列服务函数的实现，这些函数包括 spiOpen, spiRead, spiWrite, spiClose, spiIoctl。

6.3 底层驱动服务函数

SPI 底层驱动服务函数即通过调用 iosDrvInstall 注册到系统驱动表中的五个函数，下面我们一一对其进行介绍。

6.3.1 设备打开函数

用户在使用一个设备之前必须先打开这个设备，底层驱动响应函数中根据设备的需要将进行中断注册和使能设备工作配置等操作。对于我们的 SPI 底层驱动而言，在主机端直接使用 GPIO 进行 SPI 时序的模拟，故我们没有使用中断，所以底层驱动打开函数 spiOpen 的实现只需要通过 SPI 接口配置对应的通道寄存器，使能通道工作即可。对于通道波特率等参数的配置在 spiDevCreate 函数中已然完成。

spiOpen 函数原型如下。

SPI_DEV *spiOpen(DEV_HDR *pDevHdr, char *name, int flags, int mode);

注意传入 spiOpen 的第一个参数是由 IO 子系统提供的，IO 子系统在根据驱动号寻址到对应驱动函数时，其将系统设备列表中存储的设备结构作为第一个参数调用 spiOpen。实际上该参数是一个 SPI_DEV 结构类型，但是 IO 子系统只“看到“DEV_HDR 结构。所以在 spiOpen 函数中我们需要首先将这个 DEV_HDR 结构转换成 SPI_DEV，这基本上是所有底层驱动函数实现中第一条语句需要完成的任务。当然，驱动程序员也可以直接将第一个参数的类型设置为自定义结构类型，那么对于我们 SPI 驱动，以上 spiOpen 函数的调用原型就变为：

SPI_DEV *spiOpen(SPI_DEV *pDevHdr, char *name, int flags, int mode);

这并不会造成什么影响，因为 IO 子系统传递过来实际上就是 SPI_DEV 结构类型，只不过 IO 子系统对于驱动自定义参数并不关心，其只需要对 DEV_HDR 进行操作就可满足 IO 子系统本身管理的需要，其他自定义参数完全由底层驱动本身进行解释和使用。所以将第一个参数直接设置为底层驱动自定义结构类型不会有任何问题。实际上为了省去函数起始处结构类型的强制转换（即此处的从 DEV_HDR 转换成 SPI_DEV），大多数驱动就将驱动中所有服务函数的第一个参数类型均设置为驱动自定义结构类型。不过为了规范起见，在我们这儿的 SPI 驱动中，我们还是使用 DEV_HDR。spiOpen 的第二个参数是设备名匹配后的剩余部分，在我们的应用中，由于 open 函数调用时输入的路径名与系统设备列表中的设备名完全匹配，故此处的 name 应为空字符串。但是对于在文件系统层下的块设备而言，此处 name 指向的就是块设备节点名后的子目录和文件名。spiOpen 的第三，四个参数就是用户 open 调用时传入的第二，三个参数，IO 子系统原封不动的将他们传递给了 spiOpen 函数。

spiOpen 返回值为 SPI_DEV 结构类型，一般返回如下两种值：一个有效的 SPI_DEV 结构指针表示 spiOpen 调用成功，ERROR 则表示 spiOpen 调用失败，IO 子系统根据有效指针或者 ERROR 返回一个文件描述符或者返回错误。注意：spiOpen 函数的返回值非常重要，驱动程序员必须让他们的底层 open 函数（如 spiOpen）在调用成功后返回一个驱动自定义结构指针，这个指针将被 IO 子系统保存，用于其后对驱动中读写，控制函数的调用（如 spiWrite，spiRead，spiIoctl），这个返回的指针将作为这些函数的第一个参数。

对于我们 SPI 底层驱动，spiOpen 函数实现如下。

```
SPI_DEV *spiOpen(DEV_HDR *pDevHdr, char *name, int flags, int mode){
    SPI_DEV *pSpiDev;
    int status=OK;
    //首先将 DEV_HDR 结构强制转换成驱动自定义结构。对于我们的 SPI 底层驱动而言，
    //这个自定义结构就是 SPI_DEV。
    pSpiDev = (SPI_DEV *)pDevHdr;

    //忽视 name，mode 参数，SPI 底层驱动不需要这两个参数。
    pSpiDev->flags=flags;
```

```
    switch(pSpiDev->channel){
        case 0:
            …  //调用 spi_bitbang 函数配置 VK3224，使能通道 1 工作。
            break;
        case 1:
            …  //调用 spi_bitbang 函数配置 VK3224，使能通道 2 工作。
            break;
        case 2:
            …  //调用 spi_bitbang 函数配置 VK3224，使能通道 3 工作。
            break;
        case 3:
            …  //调用 spi_bitbang 函数配置 VK3224，使能通道 4 工作。
            break;
        default:
            printErr("bad channel num: %d.\n", pSpiDev->channel);
            status=ERROR;
            break;
    }
    if(status == OK)
        return pSpiDev;
    else
        return (SPI_DEV *)ERROR;
}
```

spiOpen 根据设备结构中的通道号（channel）决定对哪个通道采取措施。底层驱动同时驱动四个通道，故 SPI_DEV 结构中专门定义了一个 channel 字段用以对各个通道进行区分。在调用 spiDevCreate 函数进行设备创建时，每个创建的设备都对应有自己的设备结构实例，该结构中的通道号即表示该串口设备对应 VK3224 中的某个通道，这样用户对某个串口设备的操作，其底层表现为对 VK3224 的指定通道进行操作。VK3224 的每个通道都作为一个串口设备对用户可见。

打开设备后，用户就需要对其进行读写了，注意，用户使用 open 函数调用时，传入的第二个参数表示操作权限，这个权限被底层驱动保存了下来，对后续操作进行了限制。

6.3.2 设备读写函数

在成功打开一个设备后，用户程序将得到一个文件描述符，此后用户就可以使用这个文件描述符对设备进行读写，控制操作。底层驱动读写函数原型如下。

```
int spiRead(SPI_DEV *pSpiDev, char *buffer, int nbytes);
int spiWrite(SPI_DEV *pSpiDev, char *buffer, int nbytes);
```

注意：此处将两个函数的第一个参数类型直接设置为 SPI_DEV 结构类型，spiRead，spiWrite 函数的第一个参数是 spiOpen 函数的返回值，这个返回值被 IO 子系统保存，在进行 spiRead，spiWrite，spiIoctl 函数调用作为第一个参数传入。IO 子系统的这个行为不单是针对我们这里

示例用的 SPI 驱动，对于所有的驱动，IO 子系统都是如此进行操作的。故在前文中，我们一再强调了 spiOpen 返回值的重要性，作为约定底层驱动 open 实现函数（如 spiOpen）都返回一个驱动自定义的结构类型指针（如 SPI_DEV 类型指针）。

spiRead 以及 spiWrite 实现代码如下。注意代码中 CHNx_xxx_RD，CHNx_xxx_WR 表示操作 VK3224 读写的命令常量。

```
#define MAX_WAIT_CYCLE      (1000)
int spiRead(SPI_DEV *pSpiDev, char *buffer, int nbytes){
    int dataRdy, cycles;
    int bytesRead;

    bytesRead=0;
    cycles=0;

    if(pSpiDev->flags == O_WRONLY){
        return ERROR;
    }
    switch(pSpiDev->channel){
        case 0:
            while(bytesRead<nbytes&&cycles<MAX_WAIT_CYCLES)
                //虽然 VK3224 每次操作 16-bit，但是每次只能读取 8-bit 的有效数据。
                //使用 spi_bitbang 函数从 VK3224 通道 1 读取数据。
                dataRdy=spi_bitbang(CHN1_STATUS_RD<<8|0); //检查数据是否就绪。
                if(dataRdy &RD _RDY){ //读就绪标志位有效，读取一字节数据。
                    buffer[bytesRead] = (char)spi_bigbang(CHN1_DATA_RD<<8|0);
                    bytesRead++;
                    cycles=0;
                }
                else{ //尚未就绪，循环等待。
                    cycles++;
                }
            }
            break;
        case 1:
            while(bytesRead<nbytes&&cycles<MAX_WAIT_CYCLES)
                dataRdy=spi_bitbang(CHN2_STATUS_RD<<8|0);
                if(dataRdy &RD _RDY){
                    buffer[bytesRead] = (char)spi_bigbang(CHN2_DATA_RD<<8|0);
                    bytesRead++;
                    cycles=0;
                }
                else{
                    cycles++;
                }
            }
```

```
            break;
        case 2:
            while(bytesRead<nbytes&&cycles<MAX_WAIT_CYCLES)
                dataRdy=spi_bitbang(CHN3_STATUS_RD<<8|0);
                if(dataRdy &RD _RDY){
                    buffer[bytesRead] = (char)spi_bigbang(CHN3_DATA_RD<<8|0);
                    bytesRead++;
                    cycles=0;
                }
                else{
                    cycles++;
                }
            }
            break;
        case 3:
            while(bytesRead<nbytes&&cycles<MAX_WAIT_CYCLES)
                dataRdy=spi_bitbang(CHN4_STATUS_RD<<8|0);
                if(dataRdy &RD_RDY){
                    buffer[bytesRead] = (char)spi_bigbang(CHN4_DATA_RD<<8|0);
                    bytesRead++;
                    cycles=0;
                }
                else{
                    cycles++;
                }
            }
            break;
        default:
            printErr("bad channel num: %d\n", pSpiDev->channel);
            bytesRead=-1;
            break;
    }
    return bytesRead;
}
```

spiWrite 函数实现如下。

```
int spiWrite(SPI_DEV *pSpiDev, char *buffer, int nbytes){
    int dataRdy, cycles;
    int bytesWrite;

    bytesWrite=0;
    cycles=0;

    if(pSpiDev->flags == O_RDONLY){
```

```
        return ERROR;
    }
    switch(pSpiDev->channel){
        case 0:
            while(bytesWrite < nbytes&&cycles<MAX_WAIT_CYCLES)
                dataRdy=spi_bitbang(CHN1_STATUS_ RD <<8|0);
                if(dataRdy &WR_RDY){ //写就绪标志位有效，写入一字节数据。
                    spi_bigbang(CHN1_DATA_WR<<8| buffer[bytesWrite]);
                    bytesWrite ++;
                    cycles=0;
                }
                else{
                    cycles++;
                }
            }
            break;
        case 1:
            while(bytesWrite < nbytes&&cycles<MAX_WAIT_CYCLES)
                dataRdy=spi_bitbang(CHN2_STATUS_ RD <<8|0);
                if(dataRdy &WR _RDY){
                    spi_bigbang(CHN2_DATA_WR<<8| buffer[bytesWrite]);
                    bytesWrite ++;
                    cycles=0;
                }
                else{
                    cycles++;
                }
            }
            break;
        case 2:
            while(bytesWrite < nbytes&&cycles<MAX_WAIT_CYCLES)
                dataRdy=spi_bitbang(CHN3_STATUS_ RD <<8|0);
                if(dataRdy &WR _RDY){
                    spi_bigbang(CHN3_DATA_ WR<<8| buffer[bytesWrite]);
                    bytesWrite ++;
                    cycles=0;
                }
                else{
                    cycles++;
                }
            }
            break;
        case 3:
            while(bytesWrite < nbytes&&cycles<MAX_WAIT_CYCLES)
```

```
                dataRdy=spi_bitbang(CHN4_STATUS_RD<<8|0);
                if(dataRdy &WR _RDY){
                    spi_bigbang(CHN4_DATA_ WR<<8| buffer[bytesWrite]);
                    bytesWrite ++;
                    cycles=0;
                }
                else{
                    cycles++;
                }
            }
            break;
        default:
            printErr("bad channel num: %d\n", pSpiDev->channel);
            bytesWrite =-1;
            break;
    }
    return bytesWrite;
}
```

spiRead 和 spiWrite 的实现非常相似，只是更换了一下数据的传输方向。VK3224 每次以 16-bit 为单位进行操作，但是高 8-bit 都是状态位，只有低 8-bit 才是有效数据位，所以 nbytes 个字节需要 nbytes 次读写操作。在每次进行数据读写之前，必须检查相关的寄存器状态位，对于读操作而言，查看下一个数据有无准备好，而对于写操作而言，检查前一个写入的数据是否已得到处理。另外为了避免可能的 UART 端出现问题，对于状态位的检查不是无限制的，代码中设置了一个循环等待次数（1000 次），如果在检查了这么多次后数据依然没有准备好（对于读操作）或者没有被处理（对于写操作），则退出本次读写操作，返回当前已处理的字节数。这种行为是允许的，也是标准文件接口函数的正常行为，用户对此应有相应的处理。

VK3224 对于每个通道都设置有独立的寄存器组用以进行各自的配置和控制操作，其中包括状态和数据寄存器。以上诸如 CHNx_xxx_RD，CHNx_xxx_WR 就是对每个通道各自的状态和数据寄存器进行读写的命令，这些命令以常量的形式在底层驱动中定义。

6.3.3 设备控制函数

设备控制简单的说用户对于设备某些工作行为的再配置。基于设备的类型，这些由驱动和 IO 子系统提供给用户的再配置参数会有所差别。Vxworks（其他通用操作系统也是如此）对各种类型的设备都抽取了一组共同属性作为配置选项，如串口波特率再配置就是一个串口标准属性。事实上，虽然有所约定，底层驱动完成可以按照自己的标准对这些再配置属性进行选择：可以选择只实现其中某些再配置参数，可以按照特定设备的特殊情况选择对某个再配置选项的响应方式或者转移再配置参数等等。可以说，设备控制函数既提供给了用户控制设备的方便性，也对底层设备的实现提供了极大的方便性，当然，底层驱动程序员不可以“欺骗”用户，必须完成用户要求的基本配置要求方可根据需要在做一些辅助性的配置工作，这是底层驱动设备控制实现函数的基本原则。

除了 Vxworks 操作系统本身提供的控制参数外，对于一个特定设备也有自己的特定参数，这些也可以作为选项提供给用户进行控制。一般而言，底层驱动需要定义一个头文件，将设备特定参数在其中进行定义，而后将这个头文件提供给用户程序，当用户对设备进行操作时，其包含这个头文件，使用其中定义的特定参数对设备进行控制。IO 子系统实际上不加任何改变的将用户使用的选项参数或者控制命令传递给了底层驱动，由底层驱动完成对选项参数或控制命令的解释和使用。
设备控制函数原型如下。
int spiIoctl(SPI_DEV *pSpiDev, int command, int arg);
其中第一个参数如同 spiRead 和 spiWrite，是 spiOpen 函数的返回值。第二个参数表示用户进行再配置的选项或命令，第三个参数是选项或命令附带的数据。当改变一个设备参数时，用户必须自己提供这个参数的最新值。

对于我们的 SPI 驱动，在实际使用中，再配置参数和命令有很多，作为示例我们只提供如下再配置参数和命令。
VK3224_CHN_BAUDRATE：VK3224 通道波特率参数再配置。
VK3224_CLEAR_FIFO：VK3224 通道 FIFO 清空命令。
VK3224_ENA_FIFO：VK3224 通道 FIFO 使能。
VK3224_DISA_FIFO：VK3224 通道 FIFO 禁止。
注意：所有选项和命令将统一定义在一个头文件中，同时提供给驱动本身和用户层使用。见下文介绍。

spiIoctl 函数实现如下。

```
int spiIoctl(SPI_DEV *pSpiDev, int command, int arg){
    int status=OK;
    if(pSpiDev->channel>3||pSpiDev->channel<0){
        printErr("bad channel number:%d.\n", pSpiDev->channel);
        return ERROR;
    }
    switch(command){
        case VK3224_CHN_BAUDRATE:
            spi_bitbang(makeOptions(pSpiDev->channel, VK3224_CHN_BAUDRATE , arg));
            break;
        case VK_CLEAR_FIFO:
            spi_bitbang(makeCommand(pSpiDev->channel, VK_CLEAR_FIFO));
            break;
        case VK3224_ENA_FIFO:
            spi_bitbang(makeCommand(pSpiDev->channel, VK3224_ENA_FIFO));
            break;
        case VK3224_DISA_FIFO:
            spi_bitbang(makeCommand(pSpiDev->channel, VK3224_DISA_FIFO));
            break;
        default:
            errno=S_ioLib_UNKNOWN_REQUEST;
            status=ERROR;
```

```
                break;
        }
        return status;
}
```

spiIoctl 函数实现中调用 makeOptions, makeCommand 函数根据通道号和具体参数值创建一个写入 VK3224 的数据，通过 spi_bitbang 函数将这个数据通过 SPI 接口写入到对应通道的波特率设置寄存器中完成指定通道波特率的设置。
为了使用户能够对 spiIoctl 函数实现的这些选项进行使用，SPI 底层驱动必须在一个头文件中专门对这些选项进行定义，之后将这个头文件提供给用户，用户程序中包含这个头文件即可使用这些选项或者命令对设备进行再配置或控制。
此处 SPI 底层驱动定义头文件 spicommand.h，该文件基本实现如下。

```
/ * spicommand.h
 * commands and options provided to user.
 */
#define VK3224_OPTIONS              0x8000
#define VK3224_CHN_BAUDRATE         (VK3224_OPTIONS|0x1)
#define VK3224_CLEAR_FIFO           (VK3224 _OPTIONS|0x2)
#define VK3224_ENA_FIFO             (VK3224 _OPTIONS|0x3)
#define VK3224_DISA_FIFO            (VK3224 _OPTIONS|0x4)
```

注意：系统内每个设备驱动都需要通过如此方式提供设备选项或者命令给应用层用户使用，这些选项或命令不需要在数值上做到不同，因为不同的选项将被不同的用户使用，不会再驱动之间造成冲突。对于 SPI 底层驱动，其提供 spicommand.h 头文件，当用户需要对底层 SPI 接口设备进行再配置和控制时，其首先包含 spicommand.h 头文件，而后就可以在程序中使用头文件中定义的这些选项对设备进行控制了。如下代码片段示例。

```
#include “spicommand.h”
void main(void){
    int fd;
    fd=open(“/spiUart/0”, O_RDWR, 0);
    if(fd<0){
        printErr(“cannnot open device.\n”);
        return;
    }
    ioctl(fd, VK3224_CHN_BAUDRATE, 9600); //设置通道波特率为 9600.
    ioctl(fd, VK3224_DISA_FIFO,0); //禁用通道 FIFO 缓冲。

    write(fd, msg, strlen(msg));
    close(fd);
    return;
}
```

如上所示是一个简单的用户层测试程序，其首先包含底层驱动提供的设备控制选项和命令定义头文件，其后打开一个设备，使用 ioctl 函数对通道波特率和通道 FIFO 进行了再配置，此后使用 write 函数向该通道写入一些数据，之后关闭通道。用户可将 VK3224 通道 1 UART 输出口接至 PC 机串口，使用超级终端（注意超级终端波特率也要相应的设置为 9600，否则乱码）进行信息的查看。
注意：spicommand.h 同时也被 SPI 底层驱动使用。

6.3.4 设备关闭函数

用户打开一个设备，对设备进行操作，完成用户程序的功能后，其最后一般都需要（通过调用 close 函数）关闭设备，释放资源。底层驱动对于关闭设备操作也有一个专门的服务函数，对于我们的 SPI 驱动来说，这个函数就是 spiClose。一个设备被关闭，驱动需要做的工作将根据设备类型的不同而差别很大，有些设备仅仅需要配置硬件相关寄存器将工作使能位清除即可，而有些设备除了硬件相关寄存器的配置外，还需要释放一些在打开操作中申请的内核资源，如内核缓冲区等等。对于我们的 SPI 驱动而言，关闭操作仅仅禁止使用 VK3224 中相应的通道即可。当然对于底层驱动从硬件上关闭通道的操作还有一些注意事项，事实上，我们在这儿只是说明了一种非常简单的情况，或者说我们的 SPI 驱动是不可重入的：每次在某个时刻只能有一个用户程序打开我们的驱动在使用，如果有多个用户程序同时打开我们的设备，使用我们的 SPI 驱动，则此时如果某个用户发出关闭设备请求，那么其他正在操作设备的用户程序也将无法正常工作，因为对应通道已经从硬件上被关闭了。
这是驱动设计上的一个漏洞，对于像串口如此频繁使用的设备，允许多个用户同时进行使用。
注意：此处的情况与 Vxworks 内核中将一个普通串口作为标准输入输出，而各用户程序调用 printf 通过这个串口打印的情况从本质上是不同的。虽然很多程序通过 printf 语句通过串口打印，但是实际上串口设备只被打开了一次，大家只不过都在使用同一个文件描述符而已。而对于我们的 SPI 设备，这个设备可能被打开了多次，每个用户程序在使用我们的 SPI 设备之前，都调用 open 函数进行了打开。每个用户程序的文件描述符都是不同的，虽然底层上都在使用同一个设备，同一个驱动。对于这种情况，我们需要采取文件系统中的常用策略，在驱动中维护一个使用变量，这个变量维护着驱动中 spiOpen 函数被调用的次数。当第一次 spiOpen 函数被调用时，我们进行设备初始化等等工作，其后 spiOpen 函数被调用时，仅仅增加调用计数即可，无需在进行初始化。另外由于我们的驱动同时支持四个设备，故调用计数的维护必须在设备结构中增加字段完成，而不可直接在驱动中采用全局变量的形式。为此，我们重新定义 SPI_DEV 结构，添加一个使用计数成员变量，新的 SPI_DEV 结构如下。

```
typedef struct _spi_dev{
    DEV_HDR    pDevHdr;
    UINT32     gpioRegBase;
    UINT8      channel;
    UINT8      flags;
    UINT8      refcnt;   //设备打开次数，最大支持 256 次打开。
}SPI_DEV;
```

由此为了支持用户层并发对设备和驱动的使用，我们重新实现 spiOpen 函数如下。

```
SPI_DEV *spiOpen(DEV_HDR *pDevHdr, char *name, int flags, int mode){
    SPI_DEV *pSpiDev；
    int status=OK;

    pSpiDev = (SPI_DEV *)pDevHdr;
    if(pSpiDev->refcnt>0)
        return pSpiDev;

    //不再使用 flags 成员，原因将代码后解释。
    //pSpiDev->flags=flags;
    switch(pSpiDev->channel){
        case 0:
            …  //调用 spi_bitbang 函数配置 VK3224，使能通道 1 工作。
            break;
        case 1:
            …  //调用 spi_bitbang 函数配置 VK3224，使能通道 2 工作。
            break;
        case 2:
            …  //调用 spi_bitbang 函数配置 VK3224，使能通道 3 工作。
            break;
        case 3:
            …  //调用 spi_bitbang 函数配置 VK3224，使能通道 4 工作。
            break;
        default:
            printErr(“bad channel num: %d.\n”, pSpiDev->channel);
            status=ERROR;
            break;
    }
    if(status == OK){
        pSpiDev->refcnt++; //增加打开计数。
        return pSpiDev;
    }
    else
        return (SPI_DEV *)ERROR;
}
```

注意：对 spiOpen 函数实现如此修改后，虽然基本上可以支持用户层的多次打开操作，但是还是存在问题，即 SPI_DEV 中的 flags 变量只能保存某个用户的参数，其他用户的参数将被丢弃，换句话说，如果第一个打开设备的用户传入的 flags 为 O_RDONLY，那么其他之后所有的用户对设备的操作都将被限制为只读。在处理上，我们可以不使用 flags，但是这对用户后续读写的合法性就得不到检查，当然这个问题可以通过增加 SPI_DEV 结构定义的复杂度来得到解决，此处只是提出一个警告。以上 spiOpen 函数实现中我们不再使用 flags 成员变量，相应的 spiRead，spiWrite 函数将对 flags 的检查语句删除。

此时 spiClose 函数的实现如下。

```
STATUS spiClose(SPI_DEV *pSpiDev){
    pSpiDev->refcnt --;
    if(pSpiDev->refcnt > 0){
        return OK;
    }
    switch(pSpiDev->channel){
        case 0:
            …  //调用 spi_bitbang 函数配置 VK3224，禁止通道 1 工作。
            break;
        case 1:
            …  //调用 spi_bitbang 函数配置 VK3224，禁止通道 2 工作。
            break;
        case 2:
            …  //调用 spi_bitbang 函数配置 VK3224，禁止通道 3 工作。
            break;
        case 3:
            …  //调用 spi_bitbang 函数配置 VK3224，禁止通道 4 工作。
            break;
        default:
            printErr(“bad channel num: %d.\n”, pSpiDev->channel);
            break;
    }
    return OK;
}
```

注意：spiClose 只有一个参数，这个参数也是 spiOpen 的返回值。

spiClose 函数总是返回 OK，这与用户层对 close 函数的使用方式匹配，一般用户并不对 close 函数的返回值进行检查。

spiClose 函数开始处将对设备的使用计数做减 1 处理，这是对此次的关闭操作的一个响应，如果这是最后一个使用设备的用户在进行设备关闭，那么做减 1 处理后，设备的使用计数将为 0，此时才进行真正意义上的设备关闭，即从硬件上禁止 VK3224 的相应通道工作。

至此，我们以一个实际使用的 SPI 接口设备底层驱动为例，介绍完底层驱动基本所有服务函数（还剩一个删除设备函数，下议）的定义和实现，读者需要对这些函数的原型特别留意，有一点必须谨记的是 spiRead，spiWrite，spiIoctl，spiClose 函数的第一个参数都是 spiOpen 函数的返回值。IO 子系统将 spiOpen 的返回进行保存，并将其作为第一个参数传入以上四个函数中，所以 spiOpen 函数必须返回一个设备结构类型的指针。这是所有驱动应该遵循的约定，否则其后的操作没有操作对象。当然很多驱动将设备结构作为一个全局变量在驱动各函数中使用，这种编码方式不值得提倡，当一个驱动同时驱动多个相同设备时，就有可能造成问题，当然，驱动此时可以维护一个设备结构数组，对每个设备使用数组中一个元素，但是这种方式在进行 Read，Write，Ioctl，spiClose 函数依然无法对各个设备进行区分。所以底层驱动中 open 函数的返回值是至关重要的，驱动程序员需要对此特别注意，必须使用一个有意义的可以区分设备的参数作为返回值（当然当在驱动中使用全局设备结构变量时，可

以不用返回设备结构，但是还是需要一个区分设备的参数）。

本书驱动程序概述一章中总结了驱动程序的架构，相比较我们此处示例的 SPI 驱动，我们的 SPI 驱动少了一个中断响应函数。原因很简单，SPI 驱动没有使用到中断，故也就无从有中断响应函数。我们使用 GPIO 直接模拟了 SPI 接口时序，是一种轮询工作模式。用户请求服务是即时提供的，如当一个用户请求向 VK3224 的某个通道（当然应用层代码是通过向”/spiUart/0”之类的设备）读取数据时，底层 SPI 驱动即时的从 VK3224 读取数据，而用户程序在阻塞等待数据就绪。这是轮询工作方式的基本特点。中断工作方式平时在用户没有数据请求的情况下也在从 VK3224 读取数据，并缓存在底层驱动维护的缓存区中，用户发出数据读取请求时，如果缓存区中有数据，那么就直接从缓存区中读取，而无须即时的从 VK3224 读取，所以中断工作方式效率更高。用户一般不需要等待。中断工作方式最大的优点是数据收集的过程不需要加借助于用户程序，而是自动进行的，不特意占用用户程序执行的时间。即当数据准备好，中断发生时，CPU 可能（大多数情况）执行另一个与此设备毫无关系的任务，此时占用的是这个任务的执行时间。
如果一切正常，中断和轮询模式在写入数据的效率相差不大。如果仔细分析，写入数据时，轮询模式可能具有更高的效率，相比中断而言，省去了一次中断响应过程。

正如本书驱动程序概述一章中所述，中断方式和轮询方式都是底层驱动中常用的工作模式，通常人们所说的中断优于轮询或者轮询方式极大的消耗 CPU 都是一种表层上的描述，实际上中断更消耗 CPU。基本上所有的串行设备都支持中断模式，但是实际工作中使用的大多都是轮询模式。一方面由于这些串行设备每次传输的信息量少，而是轮询模式下驱动设计较为简单，调试方便，也不会对实际性能造成影响（相比较中断而言）。我们不再深入讨论中断和轮询孰优孰劣，此处想要表达的思想是：根据具体情况选择，哪种方式都可以，而不要受某些人云亦云的固有偏见的影响，一定要使用某种方式。

当然中断是一些较为复杂设备必不可少的工作方式，在本书的下一章我们将以串口设备为例介绍中断工作方式下的串口驱动编写。

一个设备被创建后，一般而言在整个 Vxworks 操作系统运行期间，是不需要进行删除的，这个设备将一直存在于系统设备列表中，同样设备驱动也将一直存在于系统驱动表中。但是可能在某些特殊情况下，我们在使用完设备之后，需要将设备从系统设备列表中删除，也将设备驱动从系统驱动表中注销。虽然不常用，但是 IO 子系统也提供了这些功能实现的接口函数。在以上底层驱动服务函数介绍中，我们还遗留一个 spiDelete 函数的实现，下面就以 spiDelete 函数的实现为例，介绍 IO 子系统提供的设备删除和驱动注销接口函数。

6.3.5 设备删除函数

虽然使用较少，但是在某些特殊情况下还是会使用，此处以 SPI 底层驱动为例，介绍如何删除一个设备。删除一个设备的确切含义：即将设备从系统设备列表中删除，底层驱动中所有为设备分配的资源将被释放，包括设备结构占用的内存空间。设备删除函数调用后，系统中所有设备的信息将得到清除。虽然底层驱动中可以提供设备删除函数实现，但是普通用户通

常没有权限删除一个设备。这个删除操作只被某些特殊用户使用。用户层删除一个设备的标准接口函数是 remove。Vxworks 下该函数调用原型如下。

```
STATUS remove
    (
    const char *name            /* name of the file to remove */
    );
```

参数指定了被删除的设备名。

SPI 底层驱动中的 remove 响应函数为 spiDelete，该函数原型如下。

```
int spiDelete(SPI_DEV *pSpiDev, char *devName);
```

spiDelete 的第一个参数为设备结构，第二个参数为设备名。注意：此处的 devName 与 remove 函数调用时传入的 name 参数可能有所差异，这种差异主要是针对文件系统下文件的，对于一般的硬件设备而言，这两个参数是相同的。

在文件系统中删除一个文件时，给出的可能直接是一个文件名，但是一个单独的文件名并不能用于匹配系统设备列表中的设备名，因为系统设备列表中的设备名表示的是整个一个硬件设备，所以对于一个文件名在匹配系统设备列表之前，必须加上一个默认路径前缀，这个默认路径就是一个任务的根目录。当一个块设备驱动中的 delete 函数被调用时，此时传入的第一个参数将是默认路径与文件名的组合。

spiDelete 函数需要完成的功能如下。

A. 检查设备是否正常被使用，如果还有用户在操作设备，则直接返回。即一个被删除的设备首先必须是一个已经关闭的设备。
B. 调用 iosDevDelete 将设备从系统设备列表中删除。
C. 释放底层驱动中为设备分配的其他任何资源。
D. 释放设备结构本身占用的内存资源。

spiDelete 函数代码实现如下。

```
int spiDelete(SPI_DEV *pSpiDev, char *devName){
    //检查设备是否被使用，如被使用，则直接返回。
    if(pSpiDev->refcnt>0){
        printErr("Device in use.\n");
        return ERROR;
    }
    //将设备从系统设备列表中删除。
    if(iosDevDelete(&pSpiDev->pDevHdr)!=OK){
        printErr("Cannot delete device from kernel device list.\n");
        return ERROR;
    }
    //然后释放底层驱动中为设备分配的其他任何资源，对于我们的 SPI 驱动，这一项为空。
    //最后释放设备结构本身。
    free((char *)pSpiDev);
    return OK;
}
```

设备删除函数被调用后，系统内（当然包括底层驱动）所有设备信息都被清除。如果用户需要再次对这个设备（既然下决心删除了，为何现在又要用，暂且不管）进行操作，那么必须首先调用 spiDevCreate 函数重新创建一个设备：分配设备结构创建设备并将设备添加到系统设备列表中。

到此为止，方可称得上“大功告成”，我们实现了底层驱动注册的所有函数：spiOpen，spiDelete，spiClose，spiRead，spiWrite，spiIoctl。这些函数在 spiDrv 函数中通过调用 iosDrvInstall 函数注册到系统驱动表中。每个函数基本上都是用户层调用的一一对应的响应实现。如下图 6-6 所示。

用户层接口函数	底层驱动响应函数
creat	spiOpen
remove	spiDelete
open	spiOpen
close	spiClose
read	spiRead
write	spiWrite
ioctl	spiIoctl

图 6- 6 用户层接口和底层驱动响应函数对应关系

实际上，creat，remove 一般只用于文件系统管理下的块设备中，对某个文件的创建或者删除。对于普通的设备文件，底层驱动并不需要提供这两个函数的实现。

6.4 设备卸载和驱动卸载

虽然在上一节中，我们对 SPI 底层驱动实现了一个设备删除函数 spiDelete，但是这并不是合适的处理方式。事实上，系统驱动表中 de_delete 函数指针指向的实现只在文件系统中间层驱动中一般才有效，用于删除块设备上的一个文件或者目录，而非用于删除整个设备。SPI 底层驱动是一个字符设备驱动，直接受 IO 子系统管理，没有通过文件系统中间层，所以实现 spiDelete 更多的是出于帮助读者理解的目的，在实际应用中，不会作如此方式的实现。
我们使用 spiDevCreate 函数创建了一个设备，事实上删除一个设备应有的实现并不属于底层驱动服务函数集合中，而是与 spiDevCreate 函数位置平行的 spiDevRemove 函数。
spiDevRemove 函数用以删除设备，其取消 spiDevCreate 函数所做的所有工作。
spiDevRemove 函数具有和 spiDevCreate 函数相同的原型，如下。
STATUS spiDevCreate(int channel);
STATUS spiDevRemove(int channel);
同样参数表示通道号，表示具体通道对应的设备。
另外由于设备总是存在的，软件上无法做到删除一个设备，所以我们给出一个更准确的说法：即卸载设备，即将设备的软件设施全部销毁。同样，在卸载设备之后，我们可以更进一步的卸载驱动。本节就介绍如何卸载设备和驱动。

6.4.1 卸载设备

与设备创建（或者加载）相反的过程称为设备卸载，即将设备从系统中注销，使得设备不再对系统的其他部分可见。
设备下载函数命名方式与创建函数类型，一般形式为 xxxDevRemove，如 spiDevRemove。该函数完成：
A． 设备节点从系统设备列表中删除。
B．释放 xxxDevCreate 函数分配的所有资源，包括设备结构本身。

我们以 SPI 底层驱动实现的 spiDevRemove 函数为例，说明设备卸载函数基本结构。

```
STATUS spiDevRemove(int channel){
    char devName[256];
    SPI_DEV *pSpiDev;

    //首先使用通道号从系统设备列表中查询设备，获取设备结构指针。
    sprintf(devName, “%s/%d”, spiDevNamePrefix, channel);
    pSpiDev=(SPI_DEV *)iosDevFind(devName, NULL);
    //检查设备是否还在被使用，如果有，则打印警告信息，终止卸载过程。
    if(pSpiDev->refcnt>0){
        printErr(“device in use, cannot unload device.\n”);
        return ERROR;
    }
    //将设备从系统设备列表中删除。
    if(iosDevDelete(&pSpiDev->pDevHdr)!=OK){
        printErr(“Cannot delete device from kernel device list.\n”);
        return ERROR;
    }
    //然后释放底层驱动中为设备分配的其他任何资源，对于我们的 SPI 驱动，这一项为空。
    //最后释放设备结构本身。
    free((char *)pSpiDev);
    return OK;
}
```

由于传入 spiDevRemove 函数的参数仅仅是一个通道号，故首先我们根据通道号组成一个设备名，用以匹配系统设备列表中的表项，获取设备结构指针。然后检查设备使用计数，如果仍然有用户在使用设备，则终止卸载过程。在设备在被使用的情况下，不可将设备强行卸载，否则将造成内核状态的不一致。因为其后我们需要对设备结构进行释放，如果还有用户使用这个设备，在对设备进行操作时，会使用到这个设备结构，而由于设备结构已被释放，很有可能造成无效指针访问，造成整个操作系统的崩溃。另外有些书籍上对 xxxDevRemove 函数实现上，不检查用户使用情况直接就对设备进行卸载，也不释放设备结构占用的内存，且声明不影响已打开设备的用户使用，这是一个严重的误导和编程错误，这将造成内存泄露（设备结构占用的内存自始至终无法得到释放），且是一种极不规范的编程习惯。xxxDevRemove 函数与 xxxDevCreate 函数是两个对称的函数，xxxDevRemove 函数的“使命”就是取消

xxxDevCreate 函数中的所有工作。
spiDevRemove 实现中调用了 IO 子系统提供的两个接口函数：iosDevFind，iosDevDelete。这两个函数的调用原型如下。

```
DEV_HDR *iosDevFind
    (
    char *name,                    /* name of the device */
    char **pNameTail               /* where to put ptr to tail of name */
    );

void iosDevDelete
    (
    DEV_HDR *pDevHdr               /* pointer to device's structure */
    );
```

iosDevFind 函数第一个参数指定要查询的设备名，第二个参数是一个指针，指向设备名中匹配后多余的字符部分。如当以“/tffs/vxw/vxworks”设备名进行匹配时，如果系统列表中有一个设备名为“/tffs”的元素，那么 pNameTail 就指向“/vxw/vxworks”，表示匹配完成后 name 中剩余的部分字符串，可以看出这一般在进行一个文件系统下的文件名匹配时较为常见，因为系统设备列表中存储的是整个设备名，对于块设备而言，其上创建有文件系统，具有复杂层次性，故除了设备名外（设备名相当于根目录），其后还有目录名和文件名。对于一般的不使用文件系统的设备而言，将 pNameTail 设置为 NULL 即可，因为此时大多都是完全匹配。ioDevFind 函数返回一个 DEV_HDR 结构指针，实际上是一个驱动自定义的设备结构指针，如 SPI_DEV。
iosDevDelete 较为简单，此处不再讨论。

6.4.2 卸载驱动

在 spiDrv 中我们使用 iosDrvInstall 函数向 IO 子系统注册了我们的驱动，同时 IO 子系统也提供了另外一个相反作用的函数注销我们的驱动，这个函数就是 iosDrvRemove。该函数调用原型如下。

```
STATUS iosDrvRemove
    (
    int drvnum,          /* driver to remove, returned by iosDrvInstall()*/
    BOOL forceClose      /* if TRUE, force closure of open files */
    );
```

参数 1：要卸载的驱动对应的驱动号。
参数 2：如果卸载驱动时，还有使用该驱动的用户，该参数指定是否强制进行卸载，并将所有与此驱动有关的文件描述符关闭。如果强制关闭，则 IO 子系统将遍历系统文件描述符表，检查每个描述符对应结构中的驱动号是否等于要卸载驱动的驱动号，如果相同，则调用这个驱动的 close 实现函数进行关闭，同时释放文件描述符表中该表项，此时用户层的文件句柄将自动失去功效，如果用户其后使用这个文件描述符，将直接得到一个错误返回。这是比较野蛮的一种方式，对于一个良好编程习惯的程序员，会在调用 iosDrvRemove 函数之前，确

定没有程序使用该驱动。这就是我们 spiDevRemove 函数需要完成的任务。在卸载驱动之前，先完成设备的卸载，确保系统范围内已经没有用户在使用底层驱动。

我们的 SPI 底层驱动卸载函数命名为 spiDrvUninstall，该函数原型如下。
STATUS spiDrvUninstall();
注意：作为一个良好的编程习惯，用户必须首先调用 spiDevRemove 函数卸载设备，在 spiDevRemove 函数成功返回后，才进行 spiDrvUninstall 函数的调用，如此将维护内核的一致性，不会造成任何系统隐患。

spiDrvUninstall 函数实现如下。

```
STATUS spiDrvUninstall(){
    //检查驱动号合法性，如果驱动号为-1，则表示驱动已经被卸载。
    if(spiDrvNum != -1){
        if(iosDrvRemove（spiDrvNum，TRUE）==ERROR){
            printErr(“Fail to unoad driver.\n”);
            return ERROR;
        }
        spiDrvNum=-1;
    }
    return OK;
}
```

6.5 本章小结

本章首先讨论了字符设备在内核驱动层次中的基本位置，以及与 IO 子系统之间的函数调用关系，并简单了介绍了一个用户层调用如何将请求传递到底层驱动。而后，对字符设备驱动分为三个大的方面进行了分析说明：（1）设备创建和驱动注册；（2）底层驱动服务函数实现；（3）设备卸载和驱动卸载。我们以一个实际中使用的 SPI 接口设备 VK3224 为例详细介绍了各个函数的原型和实现细节，帮助读者更好的理解字符设备驱动的编写。
本章介绍的字符设备驱动是所有驱动结构层次上最为简单的一种驱动，其直接受 IO 子系统管理，中间不经过任何内核或者第三方提供的中间层。理解本章内容，可以更好的理解内核 IO 子系统，这个所有驱动的基础，后面章节中无论驱动层次多么复杂，其本质上还是驱动层次上的简单叠加，每个层次管理具体的一项功能（类似于网络栈），最后还是起源于 IO 子系统。

下一章我们将介绍基于 TTY 中间层的通用串口驱动，TTY 中间层是 Vxworks 内核提供了为提供串口使用效率而在 IO 子系统之下，串口驱动之上添加的中间层。此时 TTY 中间层直接受 IO 子系统管理，而串口驱动则委托给 TTY 中间层管理。当应用层发出一个请求后，首先经过 IO 子系统传递给 TTY 中间层，再由 TTY 中间层传递给底层串口驱动。具体细节将在下一章中讨论。

第七章 串口驱动

串口是一类非常常用的典型的字符设备，正是由于其常用性，所以 Vxworks 在 IO 子系统层之下提供了一个 TTY 内核驱动中间层，用以管理串口驱动，这就使得底层串口驱动结构与一般的字符设备有所差异。本章将详细介绍 Vxworks 下基于 TTY 中间层的串口驱动的编写。

从驱动的基本功能出发，一个驱动必须完成上层应用层和底层硬件之间的数据交互。串口驱动也不例外：其一方面接收用户发送到串口的数据，通过配置串口硬件设备将数据通过串口发送出去；其另一方面从串口接收外界输入的数据，传递给用户层。

我们以最终需要完成数据的发送和接收为例，分析一下 TTY 驱动既然作为一个中间层，其应该完成哪些功能，提供哪些功能的接口函数。

首先用户层向串口写入一些数据，这些数据首先通过 IO 子系统，IO 子系统并不对数据本身进行处理，只是对底层函数调用时的参数进行封装，这些数据被原封不动的传递给了 TTY 中间层，由于 TTY 中间层仍然起着管理的作用，其并不直接与硬件打交道，无论其完成哪些工作，其最终也毫无例外的需要将数据传递给底层串口驱动，那么 TTY 中间向底层串口驱动传递数据的方式无外乎两种：其一，TTY 中间层主动调用底层串口驱动的某个函数，将数据传递给底层驱动；其二底层串口驱动主动向 TTY 中间层要数据，在这种方式下，底层串口驱动索要数据的时机必须进行把握，因为可能 TTY 中间层暂时没有数据，如果底层驱动不停的在索要数据，那么就在白白的消耗资源。那么我们看第一种方式：当用户层需要通过串口发送数据时，TTY 中间层主动的调用底层驱动中的某个函数，TTY 中间层提供接口注册函数，让底层驱动在其初始化过程中将这个被调用的函数注册到 TTY 中间层中。这是一种可行的方案。但是这里存在一个问题，即当用户想通过串口发送一些数据时，这些数据一般不止一个字节，换句话说，即是一堆数据，而串口操作方式是每次最多只能发送一个字节。如何解决这之间的矛盾？当然我们可以在底层驱动分配一个缓冲区，当 TTY 中间层主动调用底层驱动函数发送一批量数据时，这个底层驱动函数暂时将数据存入缓冲区中，然后慢慢的一个一个将这些数据发送出去。这是一种比较合理的工作方式，这样底层驱动函数每次发送完一个字节，就从缓冲区中取下一个字节，直到缓冲区变空。这就要求这个被 TTY 驱动层主动调用的底层驱动函数必须是可重入的，因为在底层驱动发送缓冲区中之前的数据时，可能用户又写入了数据需要发送，此时就需要将这些数据缓存到底层缓冲区其他数据的末尾处，而且不能覆盖其它之前写入的尚未发送的数据，另外当缓冲区满时，必须通知用户暂时不可接收其他数据等等。

对于用户读数据，我们按照以上用户写数据的实现思想，我们在底层驱动中再分配一个读缓冲区，底层驱动将从硬件接收的数据暂时缓存到底层缓冲区中，当用户读数据时，我们从缓冲区中复制数据给用户。当然也要注意不可对用户尚未读取的数据进行覆盖，当缓冲区满时，选择何种策略处理新读取的数据等等问题。

对于以上硬件设备的读写操作，很多实现中如果硬件设备的速度跟不上用户读写的速度，基本上都是如此实现的。那么问题是，既然底层驱动需要做这么多工作，而且从效率上也是可行的，那么 TTY 中间层存在的必要性在何处？

实际上，Vxworks 操作系统开发者正是基于以上的实现思想，将底层驱动中维护的读写缓冲区从底层驱动中分离了出来，改由内核本身进行维护，这个进行维护的模块我们就称之为 TTY 中间层。所以 TTY 中间层主要是为了简化串口这个常用设备的驱动编写而专门设计的

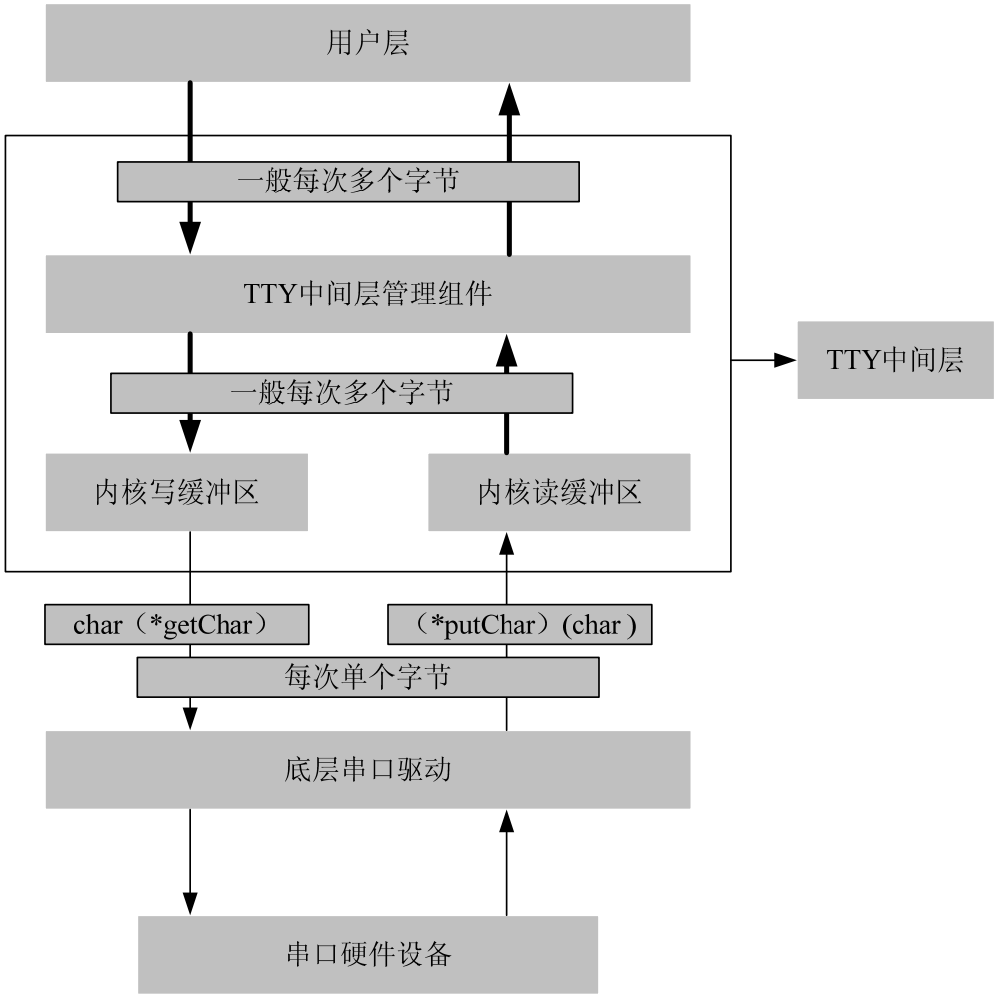
一个内核模块。其管理本来应由底层驱动管理的数据读写缓冲区，提供回调函数供底层驱动从缓冲区中读数据以及将从硬件接收的数据存入缓冲区中。原先如果由底层驱动自身进行缓冲区的维护，那么从缓冲区中读写数据直接操作缓冲区即可，而现在这个缓冲区交由内核进行管理，那么底层驱动就不可直接对缓冲区进行操作了，而是通过 TTY 中间层提供的接口函数，这两个（读写）接口函数将在串口驱动初始化过程中，由 TTY 中间层提供给底层驱动。

基于 TTY 中间层，现在我们重新设计底层串口实现思想。

用户向串口设备写入了一些数据，这些数据首先传递给 IO 子系统，IO 子系统原封不动的将这些数据传递给 TTY 中间层，由于底层串口每次只能发送一个字节，所以 TTY 中间层先将这些数据缓存到其维护的内核写缓冲区中，然后主动调用底层驱动提供的一个数据发送接口函数，发送一个字节。调用一次底层驱动接口函数，就只发送一个字节，TTY 中间层要将其维护的内核写缓冲区中缓存的数据用户数据发送出去，那么就要不停的调用底层驱动接口函数，事实上这种实现基本也是可行的。但是当我们从用户读数据的角度考虑这个问题时，这种由内核 TTY 中间层不停的调用底层驱动函数的方式就不可取了。因为对于用户读数据时，就无法通过让 TTY 中间层不停调用底层驱动某个函数来实现了。因为外界数据的到来时机是比较随机的，而且对于串口这一类设备，很多时间都是处于 Idle 状态，即没有数据的收发行为。如果让 TTY 中间层不停的调用底层驱动某个接口函数来从串口硬件接收数据，其将大大的浪费 CPU 资源，而且效率极低。最好的方式莫过于让底层驱动在有数据到达时，主动的提供数据给 TTY 层，实现上，可以让 TTY 中间层提供一个接口函数，该接口函数接收一个字节，并将其放入 TTY 中间层维护的内核读缓冲区中。基于中断的工作方式，每次硬件接收一个字节的数据，底层驱动调用 TTY 中间层提供的这个接口函数将这个字节存入内核读缓冲区中，既可以做到数据接收的及时性，也免去了让 TTY 中间层不断的轮询，浪费 CPU 资源。而对于向串口写数据的情况，TTY 中间层实现上也摒弃了由 TTY 中间层不断的调用底层驱动接口函数的方式，而是改由底层驱动不断地调用 TTY 中间层提供的从内核缓冲区读取数据发送的方式。即 TTY 中间层对于串口数据的读写提供了两个接口函数给底层驱动：（1）内核写缓冲区读取函数；（2）内核读缓冲区写入函数。

内核写缓冲区即从用户的角度而言，由用户写入的，需要底层驱动通过串口硬件发送出去的数据暂存之地；内核读缓冲区，即由底层驱动写入的，最后被用户读取的数据暂存之地。那么从底层驱动的角度而言，内核写缓冲区就是底层驱动需要发送的数据的读取之地，而内核读缓冲区就是底层驱动从硬件串口读取的数据的缓存之地。

TTY 中间层维护两个内核缓冲区，一个供用户写入，一个供底层驱动写入，如下图 7-1 所示。



注意：内核缓冲区的命名是从用户层角度而言的，即写缓冲区即让用户写入的，底层驱动读取的缓冲区；而读缓冲区即让底层驱动写入的，用户层读取的缓冲区。

图 7- 1 底层驱动与 TTY 中间层之间的数据交互

从如上图 7-1 中，我们可以看到，底层驱动维护两个函数指针，存储 TTY 中间层提供的内核缓冲区读写函数，其中 getChar 函数指针存储从内核写缓冲区读取数据的函数地址；putChar 函数指针存储向内核读缓冲区写入数据的函数地址。这两个函数每次操作都以一个字节为单位进行。

现在只遗留最后一个问题了，即如何触发底层驱动主动发送数据。对于底层驱动数据接收是由串口设备中断触发的。每次串口硬件设备从外界接收到一个字节，其发出一个中断，底层中断响应函数，从串口数据寄存器中读取这个字节，调用 putChar 指向的内核读缓冲区写入函数将这个字节写入内核读缓冲区中，完成串口对外界数据的接收。对于向外界发送数据，TTY 确实是提供了一个内核写缓冲区读取函数，可以让底层驱动不停的从这个缓冲区中读取数据，进而通过串口发送出去，但是一定有一个触发点，即第一次从内核写缓冲区中读取数据如何进行触发。底层驱动本身不可能进行触发，如果由底层驱动完成触发，那么就又回到轮询方式下了，而且是漫无目的的轮询（注意不同于轮询工作模式），因为缓冲区是由 TTY 中间层维护，用户何时发送数据无法把握。另外一个关键的原因在于现在缓冲区由 TTY 中间层维护，即由内核维护，底层驱动没有对缓冲区的直接操作权，只能通过 TTY 中间层提

供的接口函数对缓冲区进行操作，但是实际上 TTY 只提高了一个写缓冲区读取函数和读缓冲区写入函数，并无缓冲区空或者满状态的查询函数。当然当谢缓冲区读取函数返回空时，也就表示写缓冲区空的状态。但是这仍然没有解决最初始的触发问题。而且当这次内核写缓冲区被读空后，下一次触发又该如何触发？既然这个触发底层驱动完成不了，那么只能由 TTY 中间层完成。
我们可以这样来想象 TTY 的触发实现，首先触发的目的就是让底层驱动知道写缓冲区中现在有数据了，需要将这些数据发送出去了，那么基本实现上定是 TTY 中间层调用底层驱动中的某个函数，这个函数一旦被调用，就表示内核写缓冲区存在要发送的数据，此后底层驱动不停的调用 getChar 函数指针（如图 1 所示）指向的内核写缓冲区读取函数读取数据，并通过串口发送出去，直到 getChar 函数返回空，即表示写缓冲区中所有数据已被发送完毕，此时底层驱动停止这一次的一连串的发送操作，等待写缓冲区中再次存入数据。下一次用户再次写入数据，TTY 中间层再次调用底层驱动中的函数，其中下一次的一连串发送操作，直到写缓冲区再次被清空。以上描述的过程正是当前 TTY 中间层的实际实现方式。底层驱动在初始化过程中，除了从 TTY 中间层“要了”两个缓冲区操作函数外，其还需要“提供”一个函数给 TTY 中间层，当内核写缓冲区存在数据时，可以让 TTY 中间层通过调用该函数通知底层驱动，此后，底层驱动将不停地读取内核写缓冲区中的数据进行发送，直到内核写缓冲区中当前存储的数据被发送完毕。内核写缓冲区读取函数则是底层驱动从 TTY 中间层“索要”的两个函数中的其中之一，另一个函数则是向内核读缓冲区写入的函数。当串口设备从外界接收到一个字节的数据时，其发出一个中断，底层驱动中断响应函数从串口设备中读取该字节，通过内核读缓冲区写入函数将这个字节写入缓冲区中，完成数据的接收。

经过以上的分析，TTY 中间层与底层串口驱动之间的关系就比较清楚了：其一，TTY 中间层向底层驱动提供两个内核缓冲区读写函数，使得底层驱动可以从 TTY 层读取需要通过串口发送的数据以及向 TTY 层写入从串口设备接收的数据；其二，底层驱动向 TTY 中间层提供一个数据发送触发函数，从而使得当内核写缓冲区中从空变为非空时，可以通知到底层驱动，将这些新存入的数据发送出去。
注意：TTY 中间层对底层驱动提供的数据发送触发函数调用时机是内核写缓冲区由空变为非空时。这就表示如果在底层驱动对内核写缓冲区中已存数据发送的过程中，如果又有一批用户写入的数据存入内核写缓冲区中，TTY 中间层并不调用底层驱动发送触发函数。这也从另一方面要求底层驱动发送触发函数以及实际数据发送函数实现上必须能够做到“百折不饶”，一旦被触发，那么就必须将写缓冲区中已存入的所有数据（包括在发送过程中写入的数据）都发送完毕才可“罢休”，更确切的说，只当 getChar 返回空时，才结束本次发送操作。

在对 TTY 中间层和底层串口驱动之间的基本关系理解后，下面我们就开始从实际代码出发一步一步的介绍 Vxworks 串口驱动的编写。

7.1 数据结构

从上文分析中我们知道，TTY 中间层和底层串口驱动之间存在着一个相互注册的过程，注册是一方向另一方提供某种信息的手段。通常而言，非内核组件向内核组件的注册通过如下两个步骤：（1）初始化内核提供的一个特定数据结构；（2）以这个初始化后的特定数据结构

调用内核提供的接口函数。内核组件向非内核组件的注册则是直接输出相关信息。
底层串口驱动向 TTY 中间层注册用的数据结构就是 SIO_CHAN 结构。该结构定义如下。

```
/*h/sioLib.h*/
typedef struct sio_drv_funcs SIO_DRV_FUNCS;

typedef struct sio_chan                 /* a serial channel */
    {
    SIO_DRV_FUNCS * pDrvFuncs;
    /* device data */
    } SIO_CHAN;

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 *, ...),
            void *      callbackArg
            );

    int     (*pollInput)
            (
            SIO_CHAN * pSioChan,
            char *      inChar
            );

    int     (*pollOutput)
            (
            SIO_CHAN * pSioChan,
            char        outChar
            );
```

};

SIO_CHAN 是一个封装后的结构名称，其实际就是一个 sio_drv_funcs 结构类型。从以上定义中可以看出，该结构所有成员都是函数指针。除了上文分析中介绍了底层发送触发函数指针 txStartup 外，还存在如下函数指针：

A． ioctl：设备控制函数，该函数工作机制如同一般的直接受 IO 子系统管理的字符设备驱动函数。注意上文中的讨论中我们只对数据读写两个主要方面进行了论述，总结出了三个函数（底层驱动发送触发函数和内核缓冲区读写函数）。

B．callbackInstall：这个函数就与上文讨论的数据读写操作相关了。我们提到 TTY 中间层需要向底层串口驱动提供两个函数：内核写缓冲区读取函数和内核读缓冲区写入函数。既然是内核向底层驱动输出信息，那么通常实现，就是内核提供两个接口函数，每次调用输出一个函数地址，由底层驱动进行保存即可；或者以硬编码方式，在底层驱动中直接调用这两个函数，因为底层驱动作为也是作为内核一部分编入 Vxworks 内核，这不会造成使用上的问题，不过这种方式对操作系统本身的升级提出了很大的限制。此处 TTY 内核中间层实现采用了另一种方式，即将原先 TTY 中间层向底层驱动提供函数的过程变为了由底层驱动向 TTY 中间层提供函数的过程。我们查看 callbackInstall 函数的参数即可明白。这个函数第二，三个函数表示了该函数的真实面目。由于 TTY 中间层需要向底层串口驱动提供两个函数，故必须有一个参数（即第二个参数）指定此次提供的是哪个函数，其次就是函数本身（第三个参数），底层驱动提供的这个 callbackInstall 函数必须在底层驱动中根据函数的不同目的将这两个函数地址保存到底层驱动中，供之后操作内核缓冲区使用。

C．pollInput，pollOutput：这两个函数的目的就是实现上文分析中所述的由 TTY 中间层不停的调用底层驱动函数发送缓冲区中数据以及从设备接收数据的功能。即此时底层驱动：（1）不再使用中断主动从设备接收数据，而完全由 TTY 中间层主动索取数据；（2）不再主动从内核写缓冲区读取数据发送，而是每发送一个字节，都需要 TTY 中间进行驱动，即不是原先单次触发后，紧随一系列发送操作，而是每需要发送一个字节，就必须触发一次，即多次触发。当然此时不再调用 txStartup 指向的触发函数，而是调用 pollOutput 指向的触发函数，txStartup 函数只使用在单次触发的工作方式中。这种由 TTY 中间层完全负责每个字节发送和接收过程的工作方式用于使用串口建立主机和目标机之间的调试通道时。如果系统不使用串口作为调试通道，则可以不对这两个函数进行实现。作为示例，我们将给出两个函数的完全实现。

从以上分析中可见，sio_drv_funcs 结构实际上完成了两个方向上的注册功能。作为底层驱动编写的惯例，底层驱动需要定义一个自己的数据结构用以保存底层硬件的固有信息和关键参数，当然同时为了服务内核（对于串口设备而言就是 TTY 中间层）组件的需要，同时也需要一个内核提供的数据结构，对于串口设备，毫无疑问就是如上的 sio_drv_funcs 结构，那么毫无疑问，sio_drv_funcs 结构自然作为底层驱动自定义数据结构的第一个成员，其他设备参数和信息作为其他成员。如此这个结构既可以被内核组件使用，也可以被底层驱动使用，且所有信息集中在一起便于使用和管理，这基本上已经是底层驱动编写的固定流程和操作方式。

下面我们将基于 Texas Instruments（TI）公司的 TMS320DM6446 平台串口接口介绍串口驱动的组成和编写。TMS320DM6446 平台是内嵌 ARM926EJS 处理器核以及 TI C64x+DSP 核的基于 TI“达芬奇”技术的一款面向网络多媒体应用的高性能嵌入式平台。

DM6446 平台提供了三个串口接口，这些串口遵循工业标准的 TL16C550 异步通信接口。其编程方式如同通用串口：可配置波特率；可配置有效数据位，校验位，停止位；可配置硬件 FIFO 使能与否；DMA 支持；中断支持；回环测试支持等等。其内部寄存器映射按照标准的串口设备。关于 DM6446 平台提供的串口接口更多信息请参见 TMS320DM644x Universal Asynchronous Receiver Transmitter User Guide 手册，有关 DM6446 平台的更多信息请参见 TMS320DM644x Digital Media System on Chip 手册。

基于上文中的分析，我们的底层串口自定义数据结构如下所示。

```
typedef volatile unsigned char    ARMREG
typedef struct    _CSL_UART_S { //DM6446 平台上串口接口硬件寄存器布局定义。
    ARMREG RBR;
    ARMREG IER;
    ARMREG IIR;
    ARMREG LCR;
    ARMREG MCR;
    ARMREG LSR;
    ARMREG MSR;
    ARMREG SCR;
    ARMREG DLL;
    ARMREG DLH;
    ARMREG PID1;
    ARMREG PID2;
    ARMREG PWREMU_MGMT;
} CSL_UART_S;

#define THR RBR     /* RBR & THR have same offset */
#define FCR IIR     /* IIR & FCR have same offset */

typedef struct _ARM926_CHAN
{
    /* must be first */
    //是的，第一个成员变量必须是内核提供的，这已是老生常谈了。
    SIO_CHAN        sio;        /*standard SIO_CHAN element */

    /* callbacks */
    //以下两个函数指针保存 TTY 中间层提供了两个内核缓冲区操作函数地址以及函数。
    //TTY 中间层使用这两个特殊参数分辨是对内核哪个缓冲区进行操作。这两个参数在
    //内核提供回调函数指针时一并提供，底层驱动必须也对他们进行保存。
    STATUS          (*getTxChar) ();
    STATUS          (*putRcvChar) ();
    void *          getTxArg;
    void *          putRcvArg;

    /* register addresses */
```

```
    //串口设备硬件寄存器基地址。
    CSL_UART_S          *regs;          /*UART Registers*/
    //串口设备中断号。
    UINT32      level;          /* Interrupt Level for this device*/
    //串口设备波特率计算因子。这个值将被写入串口设备中决定波特率的两个寄存器中。
    UINT32      clkdiv;


    /* misc */
    //其他管理信息。
    UINT32      options;    /* Hardware options */
    int         intrmode;           /* current mode (interrupt or poll) */
    int         frequency;          /* input clock frequency */
    UINT32      errcount;

} ARM926_CHAN;
```

底层驱动中自定义数据结构中成员的确定是在驱动不断编写中完善的，当然对于一些显然的参数开始时就已存在于结构中了。有些成员要到实际代码编写后才能确定。有时为了实现某种功能，需要添加一个变量进行配合，这时就要考虑到底是以全局变量还是成员变量的形式存在，进而根据实际情况做到决定，此时就可能对原先定义的结构增添成员或者更改或删除成员，很少有“一蹴而就”的。有时对于某个功能的实现方法有问题，随意添加成员造成结构中成员混乱，意义不明的情况经常发生，特别是使用在公司内部，不对外发行的驱动程序中这种情况尤为严重。这就需要底层驱动程序员具有较深的功底，对设备驱动具有一定的经验时，才能编出一个代码结构清晰，易读易懂的驱动来。

数据结构定义完成后，接下来我们将围绕该数据结构介绍串口驱动的各个方面。首先让我们“追寻”一下内核 TTY 中间层的初始化过程以及底层驱动的初始化过程。从本书前面章节内容可知，Vxworks 下内核驱动层次决定了 Vxworks 下所有的驱动都必须置于 IO 子系统的直接或间接管理之下，对于串口设备驱动而言，IO 子系统下是 TTY 驱动中间层，TTY 中间层之下才是底层串口驱动。基于 IO 子系统管理中的三张表，首先 TTY 中间层需要向 IO 子系统注册其驱动中间层，从而使得让 IO 子系统可以访问到 TTY 中间层，进而将用户请求转发给 TTY 中间层，至于其后的处理，则是 TTY 中间层与底层驱动之间的事情了。在内核提供了 TTY 中间驱动层的情况下，IO 子系统只与 TTY 中间层交互，底层串口驱动完全委托给了 TTY 中间层管理，所以对于 IO 子系统而言，其是无法“看到”底层串口驱动存在。对于所有的串口设备操作，IO 子系统都将调用 TTY 中间层注册在系统驱动表中的函数进行处理。而 TTY 中间层实现再将请求传递给底层串口驱动，不过这已与 IO 子系统没有任何关系了。换句话说，对于 IO 子系统而言，TTY 中间层就是串口设备驱动。
所以在介绍真正的串口设备驱动之前，我们必须对 TTY 中间层的初始化过程做一了解，以及真正的串口驱动本身是如何初始化的，串口设备又是如何以及何时添加到系统设备表中的。

7.2 TTY 中间层初始化

TTY 中间层作为 Vxworks 操作系统组成的一部分，如同其他内核组件一样，在操作系统启动中被初始化。具体初始化代码定义在 usrRoot 函数（usrConfig.c）中。代码如下。

```
#ifdef   INCLUDE_TTY_DEV
    if (NUM_TTY > 0)
    {
    ttyDrv();                   /* install console driver */

    for (ix = 0; ix < NUM_TTY; ix++)/* create serial devices */
        {
#if      (defined(INCLUDE_WDB) && (WDB_COMM_TYPE == WDB_COMM_SERIAL))
        if (ix == WDB_TTY_CHANNEL)      /* don't use WDBs channel */
        continue;
#endif
        sprintf (tyName, "%s%d", "/tyCo/", ix);
        (void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);

        if (ix == CONSOLE_TTY)          /* init the tty console */
        {
        strcpy (consoleName, tyName);
        consoleFd = open (consoleName, O_RDWR, 0);
        (void) ioctl (consoleFd, FIOBAUDRATE, CONSOLE_BAUD_RATE);
        (void) ioctl (consoleFd, FIOSETOPTIONS, OPT_TERMINAL);
        }
        }
    }
#endif   /* INCLUDE_TTY_DEV */
```

TTY 组件初始化必须首先定义 INCLUDE_TTY_DEV 宏。

TTY 组件具体的初始化工作由 ttyDrv 函数完成，该函数实现的基本功能如同本书前面章节中普通字符设备类似函数实现的功能：向 IO 子系统注册驱动。此处，即 TTY 驱动中间层将自己的驱动函数注册到系统驱动表中，作为所有串口设备驱动的最上层次，也即所有用户对串口设备的操作都首先经过 TTY 驱动中间层的处理。我们可以类比普通字符设备给出 ttyDrv 函数的基本实现，代码如下。

```
LOCAL int ttyDrvNum=-1;
STATUS ttyDrv (void){
    /* check if driver already installed */
    if (ttyDrvNum > 0)
        return (OK);

    ttyDrvNum = iosDrvInstall (ttyOpen, (FUNCPTR) NULL, ttyOpen,
                                        ttyClose, tyRead, tyWrite, ttyIoctl);
```

```
    return (ttyDrvNum == ERROR ? ERROR : OK);
}
```

此处 TTY 驱动中间层向 IO 子系统维护的系统驱动表中注册了如下函数 ttyOpen，ttyClose，tyRead，tyWrite，ttyIoctl。注意没有 ttyDelete 函数，事实上除了文件系统管理下的块设备外，基本其他所有的设备驱动都不存在 xxxDelete 函数，类似的功能实现由 xxxDevRemove 函数完成。这个函数与 xxxDevCreate 函数形成对称，取消 xxxDevCreate 函数所有的操作，当然包括将设备节点从系统设备列表中删除。

ttyDrv 函数执行完成后，TTY 中间层就将中间层驱动注册到系统驱动表了。现在根据平台上串口的数目（TTY_NUM 表示），初始化每个底层串口驱动，创建对应的串口设备，并添加到系统设备列表中。这些功能的实现当然是由 ttyDevCreate 函数完成。该函数的调用原型如下。

```
STATUS ttyDevCreate
    (
    char *          name,               /* name to use for this device          */
    SIO_CHAN * pSioChan,      /* pointer to core driver structure */
    int             rdBufSize,          /* read buffer size, in bytes           */
    int             wrtBufSize          /* write buffer size, in bytes          */
    );
```

参数 1：创建串口设备时使用的设备节点名。

参数 2：这是一个 SIO_CHAN 类型的结构指针，事实上，其本质上是一个底层驱动自定义的数据结构，SIO_CHAN 结构作为了第一个成员变量，故同时也可以当做一个 SIO_CHAN 使用，具体情景参见前一小结“数据结构”的介绍。注意传递给 ttyDevCreate 的这个 SIO_CHAN 必须是经过底层串口驱动初始化后的，此时 TTY 中间层将根据该结构中的内容保存底层驱动相关函数地址，以便此后进行调用，如对发送触发函数（由 SIO_CHAN 结构中 tyStartup 函数指针指向）的调用。同时 ttyDevCreate 还将调用 SIO_CHAN 结构中 callbackInstall 指向函数向底层驱动提供内核缓冲区的两个操作函数。

参数 3，4：TTY 中间层维护的内核读写缓冲区的大小。这两个参数给用户一个指定内核缓冲区大小的手段。

我们查看 usrRoot 中对 ttyDevCreate 的调用：

```
sprintf (tyName, "%s%d", "/tyCo/", ix);
(void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);
```

参数 1 被初始化为形如“/tyCo/0”，“/tyCo/1”之类的字符串，系统内串口设备都具有相同的前缀，只在最后一个数字部分有所区别，表示这个系统的第几个串口设备。注意用于 WDB 调试的串口通道被预留。这个字符串最终用于设备节点名，当用户在 shell 下输入 devs 命令时，串口设备将显示为如上的字符串。

参数 2 为 sysSerialChanGet 函数的返回值，下文中我们再详细说明。

参数 3，4 均为 512，即表示 TTY 中间层将分别分配 512 个字节用于串口的内核读写缓冲区。

ttyDevCreate 函数调用中最重要的参数是第二个参数：一个 SIO_CHAN 结构指针，指向底层串口驱动自定义结构。这个 SIO_CHAN 是已经经过底层驱动初始化的，即对结构中的函数指针都进行了初始化。

sysSerialChanGet 函数一般定义在 sysSerial.c 文件中，sysSerial.c 文件专门是针对串口设备的实现文件，除了 sysSerialChanGet 函数外，其还定义有如下重要的函数：sysSerialHwInit，sysSerialHwInit2；以及一个数组：sysSioChans，该数组维护着系统内所有串口设备对应的 SIO_CHAN 结构地址。实际上 sysSerialChanGet 函数实现就是简单的根据输入参数，返回 sysSioChans 数组中对应的元素。既然调用 ttyDevCreate 函数时要求传递一个经过底层驱动初始化后的 SIO_CHAN 结构，而 sysSerialChanGet 函数仅仅返回 sysSioChans 数组中对应元素，其并不完成任何结构的初始化工作，这就表示在调用 ttyDevCreate 函数之前，底层驱动已然经过了初始化，并对驱动设备对应的 SIO_CHAN 结构进行了初始化，所以现在调用 ttyDevCreate 函数时只需简单的返回这个初始化的结构即可。那么底层串口驱动是何时进行初始化的？这就要从 sysSerial.c 文件中定义的其他两个重要函数出发：sysSerialHwInit，sysSerialHwInit2。

Vxworks 内核启动过程中，将通过调用 sysHwInit 以及 sysHwInit2 完成平台外设硬件的初始化工作（对于 sysHwInit，sysHwInit2 函数具体调用，请参见本书前面介绍 Vxworks 启动流程的章节）。而 sysSerialHwInit 函数即被 sysHwInit 函数调用专门对串口设备进行初始化，sysSerialHwInit2 则被 sysHwInit2 函数调用进行串口设备初始化的后续工作。从本书内核启动部分的介绍中可知，只有当 sysHwInit2 函数调用后，内核外设中断向量表才被创建，即 sysHwInit2 函数调用后，系统中断才能工作，也是在 sysHwInit2 函数完成外设中断向量表的初始化后，内核本身才进行系统时钟中断的注册，对于其他所有工作于中断模式或者需要借助中断的设备都必须在 sysHwInit2 之后初始化中断相关的内容：注册中断，使能中断。对于串口设备也不例外，其首先在 sysSerialHwInit 函数中完成除中断之外的所有初始化工作，包括对底层串口驱动自定义数据结构的初始化（当然包括了 SIO_CHAN 结构的初始化），配置串口设备使进入待工作状态，此后在 sysSerialHwInit2 函数中注册中断。自此串口就进入到预工作状态，由于 sysSerialHwInit2 函数只是注册了中断，没有使能中断，所以串口硬件上虽然现在可以从外界接收数据了，但是在接收到一个字节后就不再接收，因为没有中断，底层驱动不会读取这个字节，换句话说底层驱动软件上还有待使能中断，让串口中断响应函数可以将串口数据寄存器中已接收的数据读取出来，从而使得串口硬件可以继续接收下一个字节，这才是串口真正进入到工作状态。这个使能中断的工作就是在 ttyDevCreate 函数中完成的，其根据由底层驱动初始化后的 SIO_CHAN 结构完成 TTY 中间层与底层串口驱动之间的相互注册过程，最后调用底层驱动 ioctl 函数使能中断，使得串口进入到正常的数据收发工作状态。为此，底层串口驱动必须在其 ioctl 实现中，实现以下控制选项：SIO_MODE_SET。这个选项被 ttyDevCreate 函数使用使能串口中断。

对串口以及 TTY 中间层初始化过程总结如下。

A. 底层串口驱动先于 TTY 中间层驱动进行初始化，但是使能串口中断这一步将遗留给 TTY 中间层完成，为此底层串口驱动 ioctl 函数实现中，必须实现 SIO_MODE_SET 控制选项。注意串口中断使能的工作具体是由 ttyDevCreate 函数完成的。而且串口中断使能的工作必须交由 TTY 中间层完成。因为底层驱动接收的数据最终将交给 TTY 中间层，而这个转交的完成是通过调用 TTY 中间层提供的内核读缓冲区写入函数完成的。所以如果 TTY 中间层没有完成初始化，即便底层驱动接收了数据也无法向上传递，换句话说，底层驱动虽然先于 TTY 中间层进行初始化，其也必须等待 TTY 中间层初始化完成后，才能进入到真正的工作状态。这个真正的工作状态就是中断工作模式。没有中断，底层串口驱动将无法完成数据的连续接收。

B. 底层串口驱动的初始化过程分为两个阶段完成，其第一阶段完成相关数据结构的初始化

和串口硬件设备的配置工作。相关数据结构即底层串口驱动自定义的结构，其中关键是对自定义结构中第一个成员 SIO_CHAN 结构的初始化工作，这个 SIO_CHAN 结构最终将被 TTY 中间层（ttyDevCreate）使用，完成 TTY 中间层与底层串口驱动的相互注册过程：TTY 中间层提供内核写缓冲区读取函数和内核读缓冲区写入函数；底层串口驱动提供发送触发函数，设备控制函数，轮询工作模式下收发函数。其中底层串口驱动向 TTY 中间层提供的这些函数，TTY 中间层将通过直接保存 SIO_CHAN 结构中相关函数指针的值即可，而 TTY 中间层向底层驱动提供的两个函数由 SIO_CHAN 结构中 callbackInstall 指向的底层函数实现进行保存。此时 TTY 中间层将调用 callbackInstall 指向的函数，并传递内核缓冲区操作函数的地址，所以虽然是 TTY 中间层向底层驱动提供接口函数，但是最终也是通过调用底层驱动提供的函数完成。所以底层驱动在初始化 callbackInstall 函数指针时，其指向的函数必须将由 TTY 中间层调用自身时传递的函数句柄保存到底层驱动中，以便之后数据收发时使用。第二阶段完成串口中断响应函数的注册，但是不使能中断，中断的使能工作将专门留给 TTY 中间层完成，具体是在 ttyDevCreate 函数中完成的。

C．ttyDevCreate 函数是由 TTY 中间层提供的接口函数，供用户调用进行串口设备中断使能，内核读写缓冲区的创建，TTY 层和底层串口驱动之间的相互注册以及串口设备的创建工作。ttyDevCreate 函数执行完毕后，串口设备及其驱动都进入正常的工作流程，与 TTY 中间层配合着完成应用层用户的串口使用。

CONSOLE_TTY 表示作为终端使用的串口通道。这个通道将作为所有任务中打印信息的默认输出通道以及系统标准输入输出通道。

注意：ttyDevCreate 函数将保存传入的第二个参数，此后对于底层驱动函数的调用都将使用该参数。这类似于普通字符设备中 IO 子系统对 xxxOpen 函数返回参数进行的保存。另外串口设备通常被作为标准输入输出设备，即一个任务在通过调用 printf 语句打印一些信息到终端时，其无需专门打开一个串口设备。但是在系统存在多个串口的情况下，用于对于除被用作标准输入输出串口的其他串口进行操作时，就必须首先打开该串口，如下代码片段所示。

```
int fd;
fd=open(“/tyCo/1”, O_RDWD, 0);
if(fd < 0){
      printErr(“Cannot open serial channel 1.\n”);
      return -1;
}
write(fd, “hello”,5);
close(fd);
```

如上代码片段打开串口 1（串口 0 被用作标准输入输出），写入“hello”信息。此时我们就不能直接使用 printf（“hello”）；进行输出了，因为 printf 语句将通过串口 0 将信息发送出去，而不是我们要求的串口 1. 也就是说，此时除了串口 0 外，对于其他串口的操作还是必须按照一般的设备操作流程进行：先打开这个设备，而后才能对这个设备进行操作。但是从下文串口驱动结构来看，底层串口驱动中并无相应的 open，close 函数实现。是不是 TTY 中间层完全包办了？事实并非如此，实际上，TTY 中间层将用户对串口设备 open，close 请求通过底层驱动中 ioctl 函数传递给了底层驱动。当用户对一个串口设备发出 open，close 请求时，

其通过 SIO_OPEN，SIO_HUP 选项分别调用底层驱动 ioctl 函数，将请求传递给底层驱动。至于用户的读写请求，则是从 TTY 中间层维护的内核读写缓冲区直接读写数据。具体的，对于用户写入的情况，如果当前内核缓冲区非空，则表示底层驱动正在发送数据，此时 TTY 中间层将这些用户数据缓存到内核写缓冲区中，进而直接返回，不调用底层驱动的任何函数，如果当前内核写缓冲区为空，则表示上一次触发发送后，底层驱动已经将内核写缓冲区中的所有数据都已经发送出去，现在又有一批新的数据到来，那么必须再次触发底层驱动进行发送，此时 TTY 中间层将调用之前初始化过程中底层驱动提供的触发发送函数（SIO_CHAN 结构中 txStartup 指针指向的函数）。对于用户读数据的情景，由于底层驱动对于数据的接收是完全主动，无需 TTY 中间层索要，所以对于用户读数据请求，TTY 中间层将直接拷贝内核读缓冲区数据，而不对底层驱动做任何申请或者请求。即用户读数据请求将根据内核读缓冲区中数据进行满足，满足不了时，也只能返回空数据给用户，不会调用底层驱动中的任何函数。

虽然 TTY 中间层希望底层驱动使用 ioctl 函数实现对串口设备的打开和关闭操作进行支持，但是基于串口设备的简单性，底层驱动实际上在初始化时就完成了打开操作所需完成的一切配置工作，而不是等到用户调用设备打开函数时才进行串口的配置工作。只是除了被用作标准输入输出的串口外，其他串口平时一直无数据的交互，所以从表象上看似不工作，实际上我们从 usrRoot 中对 TTY 中间层的初始化代码可以看出，平台上所有的串口设备都一并进行了初始化和设备创建工作。即系统启动后，所有的串口都处于正常的工作状态。所以当用户打开一个非标准输入输出的串口时，底层串口驱动实际上没有做任何工作，事实上，很多串口驱动 ioctl 函数实现中，根本没有对 SIO_OPEN，SIO_HUP 的响应语句，而是作为 default 选项由 TTY 中间层进行处理，而 TTY 中间层除了调用底层驱动 ioctl 函数外，也没有任何其他实质性操作。从这个意义上，用户对于串口设备的打开操作更多的意义是获得一个用于读写的文件描述符，而对于串口设备的关闭操作则更多的是释放这个文件描述符资源。

TTY 中间层初始化总结：
TTY 中间层初始化主要由两个接口函数完成：ttyDrv，ttyDevCreate。其中 ttyDrv 函数完成 TTY 中间层驱动的注册，该注册的意义在于将系统上所有的串口设备都置于 TTY 中间层的管理之下，而串口设备对上（IO 子系统以及用户层）的交互则完全由 TTY 中间层代劳。ttyDevCreate 函数完成 TTY 中间层与底层驱动之间的交互：（1）创建内核读写缓冲区，提供数据收发效率；（2）完成 TTY 中间层与底层驱动之间的相互注册，TTY 中间层保存底层驱动初始化后的 SIO_CHAN 结构用以此后对底层所有函数的调用，其次 TTY 中间层通过调用 SIO_CHAN 结构中 callbackInstall 指向的底层驱动函数将内核读写缓冲区操作的两个函数地址提供给底层驱动；（3）使用 SIO_MODE_SET 选项调用底层驱动 ioctl 函数使能串口中断，是串口设备以及驱动本身进入到正常工作状态；（4）创建串口设备，将串口设备添加到系统设备列表中。
对 ttyDrv，ttyDevCreate 函数的调用完成后，TTY 中间层，底层串口驱动以及串口设备都进入正常工作状态，进行着数据收发工作。

7.3 串口驱动基本结构

串口驱动涉及的文件有 sysSerial.c，xxx_uart.c。其中 sysSerial.c 文件属于 BSP（板级支持包）组成的标准文件之一，一般包含串口设备的平台都需要在 BSP 中包含这个文件，且文件名

遵照约定必须命名为 sysSerial.c。xxx_uart.c 则是具体串口驱动的底层实现代码，完成着与串口硬件设备的交互工作，这也是底层串口驱动的真正实现文件，sysSerial.c 只是与内核组件交互的一个上层文件，其主要实现如下三个函数：sysSerialHwInit，sysSerialHwInit2，sysSerialChanGet，以及一个 SIO_CHAN 结构数组。该数组将被 sysSerialChanGet 函数实现根据串口通道号返回对应串口的 SIO_CHAN 结构供 ttyDevCreate 函数使用。除了以上两个文件外，还有一些涉及串口的宏定义，其中 NUM_TTY 表示平台串口数目或者更确切的说在 Vxworks 中使用的串口数目。一个平台可能包含多个串口通道，但是只需要使用其中的一个进行信息的输出，那么就可以将 NUM_TTY 定义为 1，即只使用一个串口，此时其他串口硬件虽然存在，但是对于 Vxworks 操作系统而言，其他串口并不对其可见。如下语句是包含在 config.h 文件中的对 NUM_TTY 的定义。

```
#define    N_SIO_CHANNELS     2
#define    INCLUDE_SERIAL
#undef     NUM_TTY
#define    NUM_TTY            N_SIO_CHANNELS
```

其中 N_SIO_CHANNELS 表示串口通道数。注意对 INCLUDE_SERIAL 的定义，这个宏定义十分重要，因为 sysSerialHwInit，sysSerialHwInit2 两个函数的调用前提就是必须定义 INCLUDE_SERIAL。

除了以上的宏定义外，涉及串口设备的其他定义如下：

```
#undef CONSOLE_TTY
#define CONSOLE_TTY                0

#undef CONSOLE_BAUD_RATE
#define CONSOLE_BAUD_RATE          115200
```

CONSOLE_TTY 表示默认为标准输入输出的串口通道，此处设置为通道 0，而 CONSOLE_BAUD_RATE 则表示所有串口初始的默认波特率，此处设置为 115200。

对于 DM6446 平台，其包含三个串口通道，为了说明问题，且尽量简单，我们使用其中的两个通道。通道 0 被用作标准输入输出设备，任务无需打开操作即可通过调用 printf 或 logMsg 语句通过该通道输出信息。通道 1 作为通常的字符设备使用，所以用户使用该通道输出信息时，必须遵循一般的设备操作流程：打开设备，操作（读写，控制）设备，关闭设备。

两个通道对应的串口设备节点在 usrRoot 函数中完成创建，所以操作系统启动完成后，这两个串口设备就已经处于正常的工作状态。

基于本章开始处的分析以及上文中对 TTY 中间层初始化介绍，我们总结底层串口驱动基本结构如下。

（1）函数实现

A. xxxUartInit 函数：被 sysSerialHwInit 函数调用完成驱动自定义结构的初始化，尤其是第一个成员变量 SIO_CHAN 结构的初始化，完成串口硬件的配置，使串口进入到待工作状态。DM6446 底层串口驱动中具体实现函数为 arm926UartInit。

B. xxxUartInit2 函数：被 sysSerialHwInit2 函数调用完成串口驱动中断响应函数的注册。中断使能延迟至 ttyDevCreate 函数中进行。DM6446 底层串口驱动中具体实现函数为 arm926UartInit2。

C．xxxUartIoctl 函数：底层驱动 ioctl 控制函数实现，必须提供 SIO_MODE_SET 选项的实现，该选项基本实现功能是使能中断，包括系统级中断使能以及硬件中断使能两个方面的使能工作（实际上可以将系统级中断使能放入 xxxUartInit2 函数中进行）。如果底层驱动支持可变波特率，还必须提供 SIO_BAUD_SET 选项的实现。关于串口设备的更多选项，请查阅内核头文件 h/sioLib.h。DM6446 底层串口驱动中具体实现函数为 arm926UartIoctl。

D． xxxUartInt 函数：底层驱动中断响应函数，主要实现数据的接收。DM6446 底层串口驱动中具体实现函数为 arm926UartInt。

E．xxxUartPollInput，xxxUartPollOutput 函数：底层驱动轮询工作方式实现函数，这两个函数提供了一种工作模式由 TTY 中间层负责每一个字节的接收和发送，此时底层驱动完全处于被动调用。这种模式一般在串口通道被用作调试通道时使用。Dm6446 平台上底层串口驱动中具体实现函数为 arm926UartPollInput，arm926UartPollOutput。

F．xxxUartCallbakInstrall 函数：该函数作为 SIO_CHAN 结构中 callbackInstall 成员变量指向的函数，供 TTY 中间层调用用以将 TTY 中间层提供的两个内核缓冲区操作函数地址传递给底层驱动。底层驱动此后对用户需要发送数据的获取以及向 TTY 中间层传递接收数据的操作都需要通过这两个函数完成，底层驱动需要对这两个 TTY 中间层提供的函数进行妥善保存。DM6446 平台底层串口驱动中具体实现函数为 arm926UartCallbackInstall。

G． xxxUartTxStartup 函数：该函数是由底层驱动提供给 TTY 中间层的发送触发函数。这个函数将用以初始化 SIO_CHAN 结构中 txStartup 函数指针，从而在 ttyDevCreate 函数调用中传递给内核 TTY 中间层。对于 DM6446 平台底层串口驱动中具体实现函数为 arm926UartTxStartup。

H． 串口硬件操作函数集合：这些函数用以对串口设备寄存器进行读写操作。对于这些函数，下文介绍中我们只交代其实现的功能，而不再给出细致的代码，这部分代码平台相关。

（2）结构定义

对于 DM6446 平台，其上有三个串口通道，我们使用其中的两个进行说明。对于两个通道，我们需要定义两个 ARM926_CHAN 结构（该结构定义见 5.1 节内容）。为了对所有串口进行管理，我们定义为数组形式。在上文“TTY 中间层初始化”一节中我们给出了一个 sysSioChans 数组，并提到这个数组定义在 sysSerial.c 文件中，事实上，这个数组的定义位置也可以放在底层驱动文件中（如 DM6446 平台串口驱动文件 arm926uart.c），由于这个数据直接被 sysSerial.c 文件中的 sysSerialHwInit，sysSerialHwInit2，sysSerialChanGet 三个函数使用，故最好还是定义在 sysSerial.c 文件中比较合适。我们在 sysSerial.c 文件中定义如下 ARM926_CHAN 结构数组（注：如下定义的结构和变量都在 sysSerial.c 文件中）：

```
LOCAL ARM926_CHAN arm926UartChan[ N_SIO_CHANNELS ]; //N_SIO_CHANNELS=2
```

而 sysSioChans 数组则定义为：

```
/*
 * Array of pointers to all serial channels configured in system.
 * See sioChanGet(). It is this array that maps channel pointers
 * to standard device names.    The first entry will become "/tyCo/0",
 * the second "/tyCo/1", and so forth.
 */
SIO_CHAN *sysSioChans [] =
```

```
{
    &arm926UartChan[0].sio, /* /tyCo/0 */
    &arm926UartChan[1].sio, /* /tyCo/1 */
};
```

事实上，只使用 arm926UartChan 这一个数组就可以满足要求，但是为了表示不同使用方式（sysSioChans 数组由 sysSerialChanGet 函数返回，供 TTY 中间层使用；arm926UartChan 数组由 sysSerialHwInit 和 sysSerialHwInit2 使用；二者本质上是指向相同内存区域的），我们增加了一个 sysSioChans 数组。注意对于这个数组的命名没有特别要求，串口设备驱动程序员可以使用一个任何其想要的名称。

为了能够在 sysSerialHwInit 和 sysSerialHwInit2 函数使用数组方式进行初始化，我们将两个串口设备的参数也用数组的方式表示。故我们定义如下结构：

```
typedef struct
{
    UINT32  vector;
    CSL_UART_S *baseAdrs;
    UINT32  intLevel;
    UINT32  clkdiv;
} SYS_ARM926_CHAN_PARAS;
```

基于该结构，我们定义如下数组：

```
LOCAL SYS_ARM926_CHAN_PARAS devParas[] =
{
    { INUM_TO_IVEC(INT_UARTINT0), (CSL_UART_S *)UART0_BASE_ADDR,
            INT_UARTINT0, 1 },
    { INUM_TO_IVEC(INT_UARTINT1), (CSL_UART_S *)UART1_BASE_ADDR,
            INT_UARTINT1, 1 },
};
```

注意：devParas 数组中类似于 INT_UARTINT0 常量定义于平台头文件中（dm6446.h），此处不再给出其具体数组，读者理解意义即可。

在完成底层串口驱动基本结构的总结以及所有数据结构的准备工作后，现在我们就可以进入到底层串口具体函数实现的阶段了。首先我们分析底层串口驱动内核接口文件 sysSerial.c 的实现。

7.4 串口驱动内核接口文件 sysSerial.c 实现

串口驱动文件组成一般有三个：sysSerial.c，xxxUartDriver.c, xxxUartDriver.h。以 DM6446 平台串口驱动为例，则是 sysSerial.c, arm926Uart.c, arm926Uart.h。其中 arm926Uart.c 以及 arm926Uart.h 文件的意义明显，c 文件是驱动代码实现文件，h 文件则是一些结构定义以及

寄存器地址和中断号等等相关参数的定义文件。对于 sysSerial.c 文件，我们将其称之为串口驱动内核接口文件，这个文件的作用主要体现在串口驱动的初始化过程中，这个文件的实现具有标准的模板，包括函数命名方式都应遵照一定的约定。当然内核级编程，完全可以实现自己的一套，只要能正常工作。但是遵循一定的编程规范和约定可以让代码的维护变得更加容易，也便于其他人的接手。Vxworks 官方文档推荐：当平台包括串口设备（并在内核中使用）时，在平台 BSP 目录下定义一个 sysSerial.c 文件，该文件由 BSP 编写者实现，具有标准的模板可供参考，BSP 编写者应该保持 sysSerial.c 模板文件中函数声明方式，只对函数的具体实现代码进行修改即可。用户在按照 Tornado 开发环境时，会一并安装 Wind River 公司提供的某些平台的官方 BSP 参考，这些 BSP 中就包括一个对应平台的模板 BSP，其中就存在 sysSerial.c 模板文件。基于 ARM 平台的 sysSerial.c 模板文件如下。

```
/*sysSerial.c Template File*/
#define MAX_SIOS       4
SIO_CHAN * sysSioChans [MAX_SIOS];

/* definitions */

/* Defines for template SIO #0 */

#define TEMPL_SIO_BASE0       0x00100000    /* base addr for device 0 */
#define TEMPL_SIO_VEC0    0x05            /* base vector for device 0 */
#define TEMPL_SIO_LVL0  TEMPL_SIO_VEC0      /* level 5 */
#define TEMPL_SIO_FREQ0 3000000       /* clk freq is 3 MHz */

/*******************************************************************************
*
* sysSerialHwInit - initialize the BSP serial devices to a quiescent state
*
* This routine initializes the BSP serial device descriptors and puts the
* devices in a quiescent state.    It is called from sysHwInit() with
* interrupts locked.
*
* RETURNS: N/A
*
* SEE ALSO: sysHwInit()
*/

void sysSerialHwInit (void)
{
      /*
       * TODO - Do any special board-specific init of hardware to shut down
       * any hardware that may be active (dma, interrupting, etc).
       *
       * Any code called here cannot use malloc/free, or do any intConnect
       * type functions.
```

```
        */

}

/*******************************************************************************
*
* sysSerialHwInit2 - connect BSP serial device interrupts
*
* This routine connects the BSP serial device interrupts.    It is called from
* sysHwInit2().
*
* Serial device interrupts cannot be connected in sysSerialHwInit() because
* the    kernel memory allocator is not initialized at that point and
* intConnect() calls malloc().
*
* This is where most device driver modules get called and devices are created.
*
* RETURNS: N/A
*
* SEE ALSO: sysHwInit2()
*/

void sysSerialHwInit2 (void)
{
    SIO_CHAN * pChan;

    /* TODO - create/connect all serial device interrupts */

    /* Create the first instance of a template SIO device */
    pChan = templateSioCreate (TEMPL_SIO_BASE0, TEMPL_SIO_VEC0,
                               TEMPL_SIO_LVL0, TEMPL_SIO_FREQ0);

    /* install this device as SIO channel #0 */
    sysSioChans[0] = pChan;

}

/*******************************************************************************
*
* sysSerialChanGet - get the SIO_CHAN device associated with a serial channel
*
* This routine returns a pointer to the SIO_CHAN device associated
* with a specified serial channel.    It is called by usrRoot() to obtain
* pointers when creating the system serial devices, `/tyCo/x'.    It
```

```
* is also used by the WDB agent to locate its serial channel.
*
* RETURNS: A pointer to the SIO_CHAN structure for the channel, or ERROR
* if the channel is invalid.
*/

SIO_CHAN * sysSerialChanGet
(
    int channel             /* serial channel */
)
{
    if (channel < 0
      || channel >= NELEMENTS(sysSioChans) )
    return (SIO_CHAN *) ERROR;

    return sysSioChans[channel];
}
```

针对 DM6446 平台修改后的 sysSerial.c 文件实现如下。

```
/* device initialization structure */
typedef struct
{
    UINT32  vector;
    CSL_UART_S *baseAdrs;
    UINT32  intLevel;
    UINT32  clkdiv;
} SYS_ARM926_CHAN_PARAS;


/* Local data structures */
LOCAL SYS_ARM926_CHAN_PARAS devParas[] =
{
    { INUM_TO_IVEC(INT_UARTINT0), (CSL_UART_S *)UART0_BASE_ADDR,
            INT_UARTINT0, 1 },
    { INUM_TO_IVEC(INT_UARTINT1), (CSL_UART_S *)UART1_BASE_ADDR,
            INT_UARTINT1, 1 },
};
/*驱动自定义结构数组*/
//N_SIO_CHANNELS=2，定义在 config.h 文件中，参见上文相关内容。
LOCAL ARM926_CHAN arm926UartChan[ N_SIO_CHANNELS ];

/*
 * Array of pointers to all serial channels configured in system.
 * See sioChanGet(). It is this array that maps channel pointers
```

```
 * to standard device names.    The first entry will become "/tyCo/0",
 * the second "/tyCo/1", and so forth.
 */
SIO_CHAN *sysSioChans [] =
{
      &arm926UartChan[0].sio, /* /tyCo/0 */
      &arm926UartChan[1].sio, /* /tyCo/1 */
};

/*******************************************************************************
*
* sysSerialHwInit - initialize the BSP serial devices to a quiescent state
*
* This routine initializes the BSP serial device descriptors and puts the
* devices in a quiesent state.    It is called from sysHwInit() with
* interrupts locked.
*
* RETURNS: N/A
*
* SEE ALSO: sysHwInit()
*/
void sysSerialHwInit (void)
{
      int i;
      for (i = 0; i < N_SIO_CHANNELS; i++)
      {
            //串口设备寄存器基地址初始化，每个串口通道都对应一些列需要配置的寄存器。
            //而每个串口通道对应的这些寄存器基地址都不相同，此处对不同通道的寄存器
            //基地址进行保存，此后底层驱动中将不再区分哪个串口，而是直接操作基地址。
            //注意：一个串口驱动负责多个串口设备。
            arm926UartChan[i].regs = devParas[i].baseAdrs;
            arm926UartChan[i].baudRate = CONSOLE_BAUD_RATE;
            arm926UartChan[i].clkdiv = devParas[i].clkdiv;

            //每个串口通道具有不同的中断号。
            //在 sysSerialHwInit2 中，我们需要使用此处保存的中断号对每个通道注册中断。
            /* receive and send always use same interrupt*/
            arm926UartChan[i].level = devParas[i].intLevel;

            /*
             * Initialise driver functions, getTxChar, putRcvChar and channelMode
             * and initialise UART
             */
            arm926UartInit (&arm926UartChan[i]);
```

```
    }
}

/*******************************************************************************
*
* sysSerialHwInit2 - connect BSP serial device interrupts
*
* This routine connects the BSP serial device interrupts.    It is called from
* sysHwInit2().    Serial device interrupts could not be connected in
* sysSerialHwInit() because the kernel memory allocator was not initialized
* at that point, and intConnect() may call malloc().
*
* RETURNS: N/A
*
* SEE ALSO: sysHwInit2()
*/

void sysSerialHwInit2 (void)
{
    int i;
    for (i = 0; i < N_SIO_CHANNELS; i++)
    {
    /*
     * Connect and enable the interrupt.
     * We would like to check the return value from this and log a message
     * if it failed. However, logLib has not been initialised yet, so we
     * cannot log a message, so there's little point in checking it.
     * URAT still in quiet state
     */
        //将底层驱动函数 arm926UartInt 注册为中断响应函数。
        //注意：中断响应函数调用时被传入的参数是 arm926UartChar[i]，所以虽然所有的
        //串口设备使用相同的中断函数，但是中断函数被调用时，传入的参数是不同的。
        //底层中断响应函数就是直接操作传入参数达到对不同串口设备进行操作的目的。
        //注意：不同串口具有不同的寄存器基地址。
        (void) intConnect ( INUM_TO_IVEC(devParas[i].vector),
                    arm926UartInt, (UINT32) &arm926UartChan[i] );

        //我们将系统级中断使能放在此处，而将设备级中断使能放在 ttyDevCreate 中。
        //系统级中断使能此处的含义即使能中断控制器中串口设备的中断。
        //而设备级中断使能的含义即配置串口相关寄存器，使得串口中断能够产生。
        intEnable (devParas[i].intLevel);
        arm926UartInit2( &arm926UartChan[i] );
    }
```

```
}

/*******************************************************************************
*
* sysSerialChanGet - get the SIO_CHAN device associated with a serial channel
*
* This routine returns a pointer to the SIO_CHAN device associated with
* a specified serial channel.    It is called by usrRoot() to obtain
* pointers when creating the system serial devices '/tyCo/x'.    It is also
* used by the WDB agent to locate its serial channel.
*
* RETURNS: A pointer to the SIO_CHAN structure for the channel, or ERROR
* if the channel is invalid.
*/

SIO_CHAN * sysSerialChanGet
  (
      int channel                 /* serial channel */
)
{
      if (channel < 0 || channel >= (int)(NELEMENTS(sysSioChans)))
            return (SIO_CHAN *)ERROR;

      return sysSioChans[channel];
}
```

sysSerial.c 文件中定义的三个函数主要用户底层串口驱动初始化过程中。内核初始化代码通过调用 sysSerial.c 文件中定义的标准函数接口完成对底层串口驱动的初始化，避免了直接对内核代码进行修改。sysSerial.c 文件中三个函数的实现是平台相关的，当时基本的代码结构相似，特别是 sysSerialChanGet 函数基本可以不做改变的使用到任一平台上。而 sysSerialHwInit 和 sysSerialHwInit2 函数只需做微小的改正即可，主要是调用底层驱动函数时的改正。

sysSerialHwInit 实现中，我们初始化每个串口设备关键参数：串口设备对应寄存器基地址，初始波特率，中断号，之后调用底层驱动串口设备初始化函数 arm926UartInit，该函数配置串口设备寄存器使得串口处于待工作状态。
我们查看 sysSerialHwInit 函数的调用环境，该函数被 sysHwInit 函数调用，相关代码如下。

```
void sysHwInit (void)
{
      …   //初始化其他设备的代码。

//以下是对 sysSerialHwInit 函数调用，用以初始化平台串口设备。
#ifdef INCLUDE_SERIAL
```

```
    /* initialise the serial devices */
    sysSerialHwInit ();         /* initialise serial data structure */
#endif /* INCLUDE_SERIAL */

}
```

sysSerialHwInit 调用完成后，底层驱动已经完成设备硬件的配置工作，并且完成了串口设备 SIO_CHAN 结构的初始化工作。

sysSerialHwInit2 函数是串口设备的后续初始化工作。对于使用中断的所有设备，都需要一个二次初始化的过程，在第二次初始化过程中，完成中断注册和使能中断的工作，这个二次初始化的过程是 Vxworks 内核特殊启动方式造成的，而非有意为之，将中断注册和中断使能单独作为第二次初始化过程。具体细节请参考本书前文“Vxworks 启动流程”相关章节。对于一般的设备，如网口，SPI 接口设备，I2C 接口设备等等，在 sysHwInit2 调用期间完成中断的注册和使能，自此设备进入正常工作模式。然而对于串口设备，却有些特殊。虽然串口设备的中断注册确实是在 sysHwInit2 调用期间完成，但是其并不进行使能中断的工作。一个中断的使能原则上包括三个步骤，CPU 系统中断位使能，这个已经在 usrRoot 任务创建时完成，其次中断控制器对应中断通道使能，再次设备中断使能。由于 CPU 可用中断输入管脚有限（通常为 1 个，ARM 为 2 个，但是 Vxworks 只使用 IRQ，不使用 FIQ），所以通常都使用各外部设备管理平台上所有设备的中断，这个外部设备我们就称之为中断控制器，中断控制器内部对每个中断通道都有一个中断使能位对应，要使该通道发出的中断可以传递给 CPU 中断管脚，则对应的中断使能位必须开启。另外每个设备都有自己的中断控制寄存器，控制中断的发出与否。所以中断开启一方面需要使能中断控制器的相关通道，还需要使能设备本身。对于串口设备我们可以在 sysHwInit2 调用期间，使能中断控制器，但是设备使能的工作留给 ttyDevCreate 函数，或者将这两个工作都留给 ttyDevCreate 调用时完成。总之，从工作角度而言，却没有进入到中断模式。

串口设备的二次初始化具体是在 sysSerialHwInit2 函数中完成的，该函数将被 sysHwInit2 函数调用专门对串口设备进行二次初始化。从此处可以看出，Vxworks 内核对串口设备确实是“情有独钟”，对串口设备驱动提供 TTY 中间层，初始化过程也使用专门的函数进行（sysSerial.c 文件中定义的 sysSerialHwInit，sysSerialHwInit2）。

在如上代码实现中，sysSerialHwInit2 函数完成了中断的注册以及中断控制器通道的使能工作，而只将设备的中断使能留到 ttyDevCreate 函数调用时完成。至于对于 arm926UartInit2 底层驱动函数的调用，则是让底层驱动在二次初始化过程中进行一些其需要完成的额外工作。通常，底层驱动二次初始化函数实现为空，即 arm926UartInit2 函数简单返回，不做任何实质性工作。

sysSerial.c 文件中定义的最后一个函数 sysSerialChanGet 被内核调用，用以获取 ttyDevCreate 函数调用时所需的设备结构参数。这可以从上文中 TTY 中间层初始化代码看出。具体调用语句如下。

```
void) ttyDevCreate (tyName, sysSerialChanGet(ix), 512, 512);
```

注意 sysSerialChanGet 函数名称是预先约定的，换句话说，为了能够让内核启动过程中正常的初始化 TTY 中间层，BSP 程序员必须在 sysSerial.c 文件中定义一个名为 sysSerialChanGet 的函数，该函数返回一个 SIO_CHAN 结构指针，指向由参数指定的串口设备的数据结构。

在以上的实现中，我们根据 sysSerialChanGet 调用时传入的参数返回 sysSioChans 数组中的相应元素。sysSioChans 数组维护着当前系统内所有被使用串口的 SIO_CHAN 数据结构。当然本质上这些 SIO_CHAN 只是底层驱动自定义结构的一部分而已。实际上 sysSerialChanGet 返回的是驱动自定义结构，但是 SIO_CHAN 作为第一个成员变量，故同时也可以用作 SIO_CHAN 结构。这在（基本所有设备类型的）驱动设计中已经是约定所成的规则了。

本节小结：
如果平台使用串口，则必须在平台 BSP 目录下定义一个 sysSerial.c 文件，该文件中必须实现如下三个指定名称的函数：sysSerialHwInit，sysSerialHwInit2，sysSerialChanGet，一般还需要一个 SIO_CHAN 结构数组。其中 sysSerialChanGet 函数和 SIO_CHAN 结构数组在 TTY 中间层初始化过程中被调用用以获取每个串口设备对应的 SIO_CHAN 结构，从而使得 TTY 中间层完成其与底层驱动之间的交互；sysSerialHwInit 和 sysSerialHwInit2 则在串口设备本身的初始化过程中被调用，完成每个串口设备对应 SIO_CHAN 结构的初始化以及串口设备硬件寄存器的配置和中断注册过程。
基于串口设备的普遍性和常用性，Vxworks 内核提供了 TTY 中间层对底层串口驱动进行管理。在 Vxworks 启动过程中，为了便于初始化底层串口驱动，Vxworks（应该是 Wind River）提供了 sysSerial.c 内核接口文件。实际上我们也可以将 sysSerial.c 文件视为 TTY 中间层的一个组成部分，从而使得 sysSerial.c 文件显得更为正式一些。

在完成对 TTY 中间层初始化以及 sysSerial.c 文件的介绍后，下面就进入到串口驱动代码的实际编写了。上文中我们已经对串口驱动的组成进行了说明，而且对于串口驱动和 TTY 中间层之间的交互关系也做了较为详细的分析，基于这些理解，串口驱动本身的设计就只是编写代码了。下一节我们将以 TI DM6446 平台串口驱动为例，详细介绍底层串口代码的实现。

7.5 串口驱动函数实现

基于上下层的关系，我们的串口驱动需要实现如下函数：

A. arm926UartInit 函数：被 sysSerialHwInit 函数调用完成驱动自定义结构的初始化，尤其是第一个成员变量 SIO_CHAN 结构的初始化，完成串口硬件的配置，使串口进入到待工作状态。
B. arm926UartInit2 函数：被 sysSerialHwInit2 函数调用完成串口驱动中断响应函数的注册。中断使能延迟至 ttyDevCreate 函数中进行。
C. arm926UartIoctl 函数：底层驱动 ioctl 控制函数实现，必须提供 SIO_MODE_SET 选项的实现，该选项基本实现功能是使能中断，包括系统级中断使能以及硬件中断使能两个方面的使能工作（实际上可以将系统级中断使能放入 xxxUartInit2 函数中进行）。如果底层驱动支持可变波特率，还必须提供 SIO_BAUD_SET 选项的实现。关于串口设备的更多选项，请查阅内核头文件 h/sioLib.h。
D. arm926UartInt 函数：底层驱动中断响应函数，主要实现数据的接收。
E. arm926UartPollInput，arm926UartPollOutput 函数：底层驱动轮询工作方式实现函数，这两个函数提供了一种工作模式由 TTY 中间层负责每一个字节的接收和发送，此时底层驱动完全处于被动调用。这种模式一般在串口通道被用作调试通道时使用。
F. arm926UartCallbackInstall 函数：该函数作为 SIO_CHAN 结构中 callbackInstall 成员变量

指向的函数，供 TTY 中间层调用用以将 TTY 中间层提供的两个内核缓冲区操作函数地址传递给底层驱动。底层驱动此后对用户需要发送数据的获取以及向 TTY 中间层传递接收数据的操作都需要通过这两个函数完成，底层驱动需要对这两个 TTY 中间层提供的函数进行妥善保存。

G. arm926UartTxStartup 函数：该函数是由底层驱动提供给 TTY 中间层的发送触发函数。这个函数将用以初始化 SIO_CHAN 结构中 txStartup 函数指针，从而在 ttyDevCreate 函数调用中传递给内核 TTY 中间层。

H. 串口硬件操作函数集合：这些函数用以对串口设备寄存器进行读写操作。对于这些函数，下文介绍中我们只交代其实现的功能，而不再给出细致的代码，这部分代码平台相关。

7.5.1 初始化函数实现

串口驱动中初始化函数主要有两个：arm926UartInit，arm926UartInit2，arm926UartInit 将被 sysSerialHwInit 函数调用完成串口设备结构的初始化以及串口设备硬件寄存器的配置，使得串口设备进入待工作状态，arm926UartInit2 函数被 sysSerialHwInit2 调用。由于 sysSerialHwInit2 函数本身在调用 arm926UartInit2 函数之前已经完成串口设备中断注册以及中断控制器相关通道的使能工作，所以 arm926UartInit2 函数就没有实质性的工作需要完成，实现中将直接返回。

除了以上两个比较明显的函数外，在初始化过程被 TTY 中间层直接调用的函数有 arm926UartIoctl，arm926UartCallbackInstall。其中 arm926UartCallbackInstall 被 TTY 中间层使用向底层串口驱动提供两个内核缓冲区操作函数；而 arm926UartIoctl 函数则被调用设置一些串口设备工作参数，如波特率，还有一个最为关键的工作是通过 arm926UartIoctl 函数使能串口设备中断。sysSerialHwInit2 函数中完成的是对中断控制器相关通道的使能，而此处完成的是对设备本身的中断使能。

（1） arm926UartInit 函数实现

该函数在 sysSerialHwInit 中的调用环境如下：

```
arm926UartInit (&arm926UartChan[i]);
```

其中 arm926UartChan 是 sysSerial.c 文件中定义的 ARM926_CHAN 结构数组，每个数组成员对应一个串口设备。此处 sysSerialHwInit 函数以指定设备对应的 ARM926_CHAN 结构为参数调用 arm926UartInit 函数。在进行调用时，传入的 ARM926_CHAN 结构是一个空白的结构，即其中的成员都还没有进行初始化，传入的目的就是让 arm926UartInit 函数根据底层驱动以及设备本身的具体情况对其进行初始化，而后便于 TTY 中间层使用。

arm926UartInit 实现的功能：初始化串口设备结构，配置串口设备寄存器，具体代码如下。

```
LOCAL   SIO_DRV_FUNCS arm926UartDrvFuncs =
{
    arm926UartIoctl,
    arm926UartTxStartup,
    arm926UartCallbackInstall,
    arm926UartPollInput,
    arm926UartPollOutput,
```

```
};
LOCAL STATUS dummyCallback (void)
{
    return (ERROR);
}
void arm926UartInit
(
    ARM926_CHAN * pChan
)
{
    UINT32 dummy;

    /* initialize each channel's driver function pointers */
    pChan->sio.pDrvFuncs  = &arm926UartDrvFuncs;

    /* install dummy driver callbacks */
    pChan->getTxChar      = dummyCallback;
    pChan->putRcvChar     = dummyCallback;

    /* reset the chip */
    CSL_uartReset(pChan);
    CSL_uartConfig(pChan);

    pChan->intrmode = FALSE;
}
```

注意：SIO_CHAN 结构本质上是一个 SIO_DRV_FUNCS 结构。arm926UartInit 函数实现的第一条语句就是对 SIO_CHAN 结构进行初始化。arm926UartDrvFuncs 变量是一个驱动内部全局变量，指向驱动内部实现的需要提供给 TTY 中间层的五个函数：ioctl，txStartup，callbackInstall，pollInput，pollOutput。每个函数在 TTY 中间层中都有使用的意义。ioctl 指向的函数用以控制串口设备；txStartup 用以触发一个发送过程；callbackInstall 函数是底层驱动提供的一个回调函数，只在初始化过程中被调用两次，TTY 中间层使用其向底层驱动提供两个内核缓冲区操作函数；pollInput 以及 pollOutput 用于调试。在完成 SIO_CHAN 结构的初始化后，我们将底层驱动用以保存 TTY 中间层提供的内核缓冲区操作函数的两个函数指针初始化为指向一个 dummy 函数，防止内核启动异常可能造成的未初始化问题，此后在使用时将造成空指针引用，此处初始化为指向一个 dummy 函数可以避免这个问题。对于 ARM926_CHAN 结构中的 getTxChar，putRcvChar 这两个函数指针的真正初始化将在 TTY 中间层调用 arm926UartCallbackInstall 函数中完成。
arm926UartInit 函数最后完成串口设备寄存器的配置工作，这些工作由 CSL_uartReset 和 CSL_uartConfig 函数完成，这些代码与具体平台相关，我们不在给出其具体实现代码。由于尚未使能中断，故设置 intrmode 成员变量为 FALSE，表示处于非中断工作模式（实际上也不是轮询模式，可以称为待工作模式）。

（2） arm926UartInit2 函数实现

该函数被 sysSerialHwInit2 函数调用完成串口设备的第二次初始化工作。在第二次初始化过程中，我们将进行中断注册以及中断控制器串口通道的中断使能工作。由于这些工作已经由 sysSerialHwInit2 函数完成（具体参见前文对 sysSerial.c 文件的介绍内容），故此处 arm926UartInit2 函数无实质性工作需要完成，故实现为一个空函数。当然，在底层驱动的具体实现中，可以将 arm926UartInit 中的一部分工作挪到 arm926UartInit2 中完成，完全由驱动程序员控制，并不受约束。如果所有工作在 arm926UartInit 中完成，实际上我们完全可以去除 arm926UartInit2 函数的定义；或者将 sysSerialHwInit2 中的工作交由 arm926UartInit2 完成等等。总之效果只有一个：就是在二次初始化期间完成中断的注册和部分使能工作，具体将工作安排在何处完成则由底层驱动程序员决定。此处我们预留 arm926UartInit2 函数的位置，以提供最大的灵活性。

```
void arm926UartInit2
(
    ARM926_CHAN * pChan
)
{
    return;
}
```

7.5.2 arm926UartCallbackInstall 函数实现

虽然 arm926UartCallbackInstall 函数实际上也是在初始化过程中被调用，但是其完成的功能相对比较特殊，故我们将其专门作为一节进行介绍。arm926UartCallbackInstall 由底层串口驱动提供，被 TTY 中间层调用，将 TTY 中间层实现的两个函数提供给底层串口驱动，即完成本章开始处介绍中所述的 TTY 中间层维护两个内核缓冲区的操作函数：内核读缓冲区写入函数和内核写缓冲区读取函数。

内核读缓冲区即被用户读取的缓冲区，其数据来源是底层驱动，所以对于底层驱动而言，其进行写入操作。当底层驱动接收到一个字节时，其调用读缓冲区写入函数将这个字节存储读缓冲区中，当用户启动一个串口读操作时，TTY 中间层将直接拷贝读缓冲区中的已有数据给用户，而不会对底层驱动做任何打扰。

内核写缓冲区即被用户写入的缓冲区，其数据来源是用户层。对于底层串口驱动而言，其必须读取其中的数据，将它们通过串口一个字节一个字节的发送出去。每次发送完一个字节的数据，底层驱动将使用写缓冲区读取函数从内核写缓冲区中读取一个字节，操作串口硬件设备发送出去，直到写缓冲区清空，底层驱动才终止发送过程。当下一次用户又写入一批数据时，TTY 中间层将调用 arm926UartTxStartup 函数再次启动发送过程，底层驱动继而又不断的读取写缓冲区中的数据进行发送，直到写缓冲区再次清空，如此循环反复的进行着数据的发送行为。注意：前文中已经交代，TTY 中间层对于 arm926UartTxStartup 发送触发函数的调用，只在用户写入数据时，内核写缓冲区为空的情况下才进行，如果用户写入数据时，内核写缓冲区还存在数据，则表示底层驱动当前已经在不断地发送数据，此时就无需对 arm926UartTxStartup 发送触发函数进行调用了。

由于内核读写缓冲区由 TTY 中间层进行维护，底层驱动无权直接对其进行操作，故需要 TTY 中间层提供两个接口函数给底层驱动，从而完成底层串口驱动与 TTY 中间层之间的数据交互。

arm926UartCallbackInstall 函数实现的功能就是存储 TTY 中间层对其进行调用时传入的函数句柄（即函数地址）。其实现代码如下。

```
LOCAL int arm926UartCallbackInstall
 (
    SIO_CHAN * pSioChan,          /* channel */
    int        callbackType,      /* type of callback */
    STATUS  (*callback)(),        /* callback */
    void *     callbackArg        /* parameter to callback */
)
{
    ARM926_CHAN * pChan = (ARM926_CHAN *)pSioChan;
    switch (callbackType)
    {
    case SIO_CALLBACK_GET_TX_CHAR:
        pChan->getTxChar = callback;
        pChan->getTxArg  = callbackArg;
        return (OK);

    case SIO_CALLBACK_PUT_RCV_CHAR:
        pChan->putRcvChar     = callback;
        pChan->putRcvArg= callbackArg;
        return (OK);

    default:
        return (ENOSYS);
    }
}
```

实际上，arm926UartCallbackInstall 函数的实现代码可以被复制到 Vxworks 下所有串口驱动的 callbackInstall 函数实现中，只需对设备结构进行细微调整即可。该函数实现是非常标准的，就是根据调用类型保存函数句柄和对应参数。由于 TTY 中间层需要提供两个接口函数给底层驱动，故 arm926UartCallbackInstall 函数将被 TTY 中间层调用两次，每次传入一个接口函数的地址，至于究竟是哪种函数的句柄，则由第二个参数指定：SIO_CALLBACK_GET_TX_CHAR 表示此次传入的是内核写缓冲区读取函数；SIO_CALLBACK_PUT_RCV_CHAR 则表示此次传入时内核读缓冲区写入函数。注意在传递函数句柄的同时还有一个参数需要进行保存，该参数在进行对应函数的调用时必须作为第一个参数传入，这个参数被 TTY 中间层内部使用区分不同设备的缓冲区。

前文中在介绍 arm926UartInit 函数实现中，将 ARM926_CHAN 结构中的 getTxChar，putRcvChar 初始化为指向一个 dummy 函数，此处这两个函数指针才得到真正的初始化。此处保存的两个内核缓冲区操作函数将在串口驱动工作期间频繁的被调用用以与 TTY 中间层进行数据的交互。

本章开始处讨论中，我们说到对于 TTY 中间层提供的这两个函数完全可以以硬编码的方式提供，即底层驱动直接调用指定名称的函数，无需进行此处繁琐的回调注册过程。但是这种

硬编码的工作方式给后期的代码扩展带来了极大的不便，而且对内核代码本身的更新升级也带来了极大地限制。因为内核代码的升级必须保证在建立在原有版本上的驱动代码也可以工作，这就表示所有的后续操作系统版本必须定义早期版本中以硬编码方式提供给驱动的这两个函数。
使用函数指针赋值的方式，就彻底解决了以上操作系统代码升级的问题，内核可以任意更换内部实现函数，而不会对底层驱动代码的运行造成任何影响。

7.5.3 arm926UartIoctl 函数实现

该函数在初始化过程中也被调用，用以设置初始波特率以及使能设备中断。但是不同于以上介绍的三个函数，其并非只使用在初始化过程中。这个设备配置，控制函数可以在设备工作期间任何时间点被调用对设备进行相关控制。
arm926UartIoctl 函数的实现具有最大的灵活性，其在实现基本功能之下可以提供任意的其他功能，也可以不提供其他任意功能。此处定义的基本功能有二：（1）设备中断使能，禁止功能；（2）波特率设置，获取功能。
arm926UartIoctl 实现代码如下。

```
LOCAL int arm926UartIoctl
(
    SIO_CHAN * pSioChan,            /* device to control */
    int        request,         /* request code */
    void *     someArg              /* some argument */
)
{
    ARM926_CHAN *pChan = ( ARM926_CHAN * )pSioChan;
    int        oldlevel;        /* current interrupt level mask */
    int        arg = (int)someArg;

    switch (request){
    case SIO_BAUD_SET:
        /*
         * like unix, a baud request for 0 is really a request to
         * hangup.
         */
        if (arg == 0)
            return arm926UartHup (pChan);

        /*
         * Set the baud rate. Return EIO for an invalid baud rate, or
         * OK on success.
         */
        if (arg < ARM926UART_BAUD_MIN || arg > ARM926UART_BAUD_MAX ){
            return (EIO);
```

```
        }

        /* Check the baud rate constant for the new baud rate */
        switch(arg){
        case 1200:
        case 2400:
        case 4800:
        case 9600:
        case 19200:
        case 38400:
        case 57600:
        case 115200:
        case 230400:
        case 460800:
              return ( arm926UartSetNewBaudRate( pChan, arg ) );

        default:
        }

        return(OK);

case SIO_BAUD_GET:
        /* Get the baud rate and return OK */
        *(int *)arg = pChan->baudRate;
        return (OK);

case SIO_MODE_SET:
        /*
          * Set the mode (e.g., to interrupt or polled). Return OK
          * or EIO for an unknown or unsupported mode.
          */
        return ( arm926UartModeSet (pChan, arg) );

case SIO_MODE_GET:
        /* Get the current mode and return OK.    */
        if( pChan ->intrmode )
              *(int *)arg = SIO_MODE_INT;
        else
              *(int *)arg = SIO_MODE_POLL;
        return (OK);

case SIO_AVAIL_MODES_GET:
        /* Get the available modes and return OK.    */
        *(int *)arg = SIO_MODE_INT | SIO_MODE_POLL;
```

```
            return (OK);


        case SIO_HW_OPTS_SET:
        case SIO_HW_OPTS_GET:
        case SIO_OPEN:
        case SIO_HUP:
        default:
            return (ENOSYS);
        }
        return (ENOSYS);
}
```

arm926UartIoctl 具体实现中对 SIO_BAUD_SET，SIO_BAUD_GET，SIO_MODE_SET，SIO_MODE_GET，SIO_AVAIL_MODES_GET 五个选项进行了响应，这五个选项涉及波特率设置和获取，模式设置和获取，设备支持模式查询。对于 SIO_HW_OPTS_SET，SIO_HW_OPTS_GET 表示的硬件设备选项返回 ENOSYS，表示不支持硬件设备选项设置和获取。SIO_OPEN 选项对应用户串口打开操作（open 函数调用），TTY 中间层使用该选项调用底层驱动 ioctl 函数，此处底层驱动实现返回 ENOSYS，注意返回 ENOSYS 并不表示串口设备不支持打开操作，实际上对于 SIO_OPEN 选项，TTY 中间层忽略底层驱动返回的任何参数，换句说话，用户打开串口设备操作总是成功的。SIO_HUP 选项对应用户关闭串口操作（close 函数调用），TTY 中间层同样也不对底层驱动返回值进行检查，也即用户关闭串口操作总是成功的。如果底层驱动需要实现 SIO_OPEN 以及 SIO_HUP 的功能，则必须进行针对性的响应。不过由于串口设备比较特殊，一般在初始化过后，就已经进入正常工作状态，而且基于串口设备的简单性和常用型，一般底层实现上，也不专门将串口设备打开和关闭作为子操作进行封装。所以用户层打开和关闭串口设备更多的意义是获取一个读写串口的文件描述符，而非需要底层驱动一定进行一个打开和关闭的具体响应。

arm926UartIoctl 实现代码比较简单，对于其他平台串口驱动可以复制该函数实现，修改具体底层响应函数即可，这写具体响应的函数实现与平台相关，一般都是配置串口的相关寄存器，修改相关结构参数配合着管理。

对于一个串口驱动，以上控制函数的实现可以说是一种最简实现，当然也可以将其实现为一个空函数，即不支持任何选项的配置，而中断所有的使能工作在 sysSerialHwInit2 函数中完成。此种方式串口也可以正常工作，但是如此编写代码“是无法见人的”，当然如果永远“不见外人”，怎么着都可以，只要能正常工作。事实上，对于内核级编程，如果不要求规范性，要达到同一个效果可以有非常多的实现方式。

7.5.4 arm926UartInt 函数实现

arm926UartInt 是串口驱动中的中断响应函数，可以说是整个驱动的灵魂。这种说法适用于其他所有的驱动。驱动简单的说就是一套函数集合。这套函数只有被执行才能有意义。函数被调用的方式通常有两个途径：（1）用户发出一个请求，这个请求经过内核层层传递，最后到达底层驱动，即驱动中的某个函数被调用执行，完成用户的请求；（2）中断；一般而言，用户请求触发的函数执行都是单次的，但是很多请求并不是单次执行一个函数就可以得到满

足。如用户读取数据操作，通常用户一次性读取多个字节，很少有一次只读取一个字节的情景，对于多个字节的情况，驱动则必须多次调用硬件操作函数才能收齐用户请求的数据量。那么如何完成硬件操作函数的多次调用？答案只能是中断。轮询不可以，轮询是一种盲目的消耗资源方式。中断相比之下是事件通知机制，即发生一个中断时，一定是某个条件得到满足。而发生一次轮询时，可以什么事都没有。对于串口设备（以及其他大多数设备）而言，中断通常有三种来源：（1）设备接收到新数据；（2）设备已发送完需发送的数据；（3）设备在数据收发过程中发生了错误。
对于设备接收到新数据发出的中断，其意义在于及时同时驱动将这个数据从设备内部的硬件缓冲区中（对于串口而言，其内部硬件缓冲区就是一个 8 比特的寄存器）读取出来，以便腾出空间接收下一个新数据，这是设备接收数据的主要和标准工作方式。当然数据的接收也可以使用轮询，但是轮询方式的使用对于设备的工作方式是有条件的，即数据的接收一定是对发送的数据的响应，简单的说，就是两个通信的设备之间具有严格的主从关系，而不是变动的主从关系。这种设备有 I2C 设备，SPI 设备，这些设备间的通信无论是数据发送和接收都是由一端主动发起，而另一端被动响应。此种方式下，在发送一个数据后，由于明确的知道在一定时间内，一定有数据传送过来，而且数据一般都具有确定的数目，故可以不断的轮询读取数据。对于像网口之类的设备，数据的接收并非一定是对发送数据的响应，故很难使用轮询工作方式接收数据。而串口设备则介于两者之间，需要根据具体使用情景确定。不过一般而言，为了提供数据收发效率，串口大多（而且必须）使用中断进行数据的接收，而对于数据的发送则也可以使用非中断方式，此时可能比中断方式具有更高的效率（见下文讨论）。

前文中不断提到发送触发函数，针对我们的例子，对应 arm926UartTxStartup 函数，该函数被 TTY 中间层调用，用以触发一个发送过程。其确切的含义当用户写入一批数据，内核写缓冲区由空变为非空时，TTY 中间层将调用 arm926UartTxStartup 函数。该函数并不能一次性将所有的数据都发送出去，基于串口每次最多只能操作一个字节的数据的前提，用户所有数据的发送将分为多次进行。这个多次发送操作的实现可以有两种：中断方式和非中断方式。

中断方式即将从内核写缓冲区读取一个字节，操作串口寄存器将这个字节发送出去，当串口硬件完成该字节的发送后，其发出一个中断，中断响应函数 arm926UartInt 监测到此次中断来源是一个数据发送完毕中断，这就表示可以进行下一个字节的发送了，此时中断响应函数将再从内核写缓冲区中读取下一个字节，将这个字节写入串口硬件数据寄存器后返回。串口硬件操作这个字节，当发送完毕后，其再次发出一个中断，如此循环往复，直到内核写缓冲区清空。中断方式下，串口硬件设备每发送完毕一个字节就发出一次中断，对系统的整体性能将造成不利的影响，因为每次中断响应过程都消耗不小的内核资源，而每个中断仅仅处理一个字节实在是“大材小用”。所以对于串口数据的发送，大多并不采用中断方式。当然，考虑到这一点，现在的串口设备很多内部集成一个 FIFO（一般为 16 字节），这就相当于从原来一个字节的硬件缓冲区变为现在的 16 字节硬件缓冲区（读写缓冲区是分开的，每个都是 16 字节），此种方式可以通过配置相关寄存器对中断触发条件进行控制，如每次发送 8 个字节才发出一个中断（事实上，串口设备是等到写 FIFO 中累积到 8 个字节才真正启动发送动作）。这种方式部分的缓解了频繁中断给整个系统带来的不利影响，但是引起的一个问题是，如果一个用户单次只写入 4 个字节或者 5 个字节，总之是小于 8 个字节，那么这 4 个或 5 个字节就会有不确定时间的延迟，必须等待用户后续字节的“推动”才能将这 4 个或 5 个字节发送出去，对于这种不确定的延迟，某些情况下是不可容忍的。基于以上这些原因，虽然现在串口设备都内部集成有 FIFO，但是实际中很少使用。而对于串口数据的发送，则采用轮询发送方式，即当 TTY 中间层调用 arm926UartTxStartup 函数时，该函数进入一个

while 循环，每次循环从内核写缓冲区中读取一个字节的数据，操作串口将其发送出去，并等待其发送动作的完成，此后再次从内核写缓冲区中读取一个字节，再次操作串口将其发送出去，直到 while 循环期间内核写缓冲器清空。
而中断方式则只用于数据的接收和错误响应。对于数据的接收，FIFO 缓冲区的使用也有类似于发送的问题，此时 FIFO 中必须累积到一定数量的字节，才会触发中断，将这些数据传送给用户，如果外界只发送一个字节的命令，其后就等待回应，那么由于这边串口接收设备需要 FIFO 中累积到比如说 8 个字节，才给出中断，那么这一个字节将无限期等待，两边陷入死锁。所以串口设备接收一般也很少使用 FIFO，但是必须使用中断，虽然还是每次只能读取一个字节，但这是没有办法的事，如果使用轮询方式接收数据，基于串口数据接收时机的不确定性，其消耗的资源更是无法容忍。

综上所述，arm926UartInt 函数实现将只对数据接收中断和收发错误中断进行响应。发送中断从硬件角度被禁止使用，即不会有发送中断产生。

arm926UartInt 函数实现如下。

```
void arm926UartInt
(
    ARM926_CHAN * pChan          /* channel generating the interrupt */
)
{

    char        rByte;
    UINT32    status;

    CSL_UART_S* udev=pChan->regs;

    status = udev->LSR;
    //错误中断
    if(status&(CSL_UART_LSR_RXFIFOE_MASK| CSL_UART_LSR_FE_MASK |
            CSL_UART_LSR_OE_MASK | CSL_UART_LSR_PE_MASK)) //error interrupt
    {

        pChan->errcount ++;
        CSL_uartReset(pChan);
        CSL_uartConfig(pChan);
        CSL_uartEnable(pChan);
    }

    //数据接收中断
    while(status&(CSL_UART_LSR_DR_MASK)) //receive interrupt
    {

        CSL_FINSR(uartRegs->LCR, 7, 7, 0);
        rByte = CSL_FEXT(uartRegs->RBR, UART_RBR_DATA);
```

```
            (*pChan->putRcvChar) (pChan->putRcvArg, rByte);
            status = udev->LSR; //get status again to check
        }

}
```

arm926UartInt 函数首先根据设备结构获取对应设备寄存器基地址，而后读取中断状态寄存器，检查中断来源：（1）是一个错误中断，则复位串口，重新配置串口工作;（2）是一个数据接收中断，读取串口数据寄存器，调用内核读缓冲区写入函数，将这个字节传递给 TTY 中间层。注意我们对于数据接收中断的响应代码使用的是 while 语句，而不是 if 语句，因为在读取数据之后，将数据传递给内核 TTY 层的过程中，串口可能又完成下一个字节的接收，此时由于处于中断处理函数中，中断被屏蔽，故中断没有给出，但是中断状态寄存器已经显示了更新的信息，所以我们将当前这个字节传递给 TTY 中间层之后，再次读取中断状态寄存器，查看串口是否在此期间已经完成对下一个字节的接收。

基于上文中的讨论，中断响应函数中，没有对发送中断的响应代码，发送中断从设备硬件的角度被禁止使用，所以不会有发送中断产生。

7.5.5 arm926UartTxStartup 函数实现

该函数在底层串口驱动中起着重要的作用：完成数据的发送。由于 TTY 中间层的存在，应用层通过串口发送的数据首先被 TTY 中间层缓存，这些缓存的数据最终需要通过底层驱动程序驱动串口硬件设备发送出去。那么 arm926UartTxStartup 函数就是 TTY 中间层向底层串口驱动提交发送数据的媒介。

诚如前文中一再声明的，该函数并非每次用户发送数据时都被调用，该函数的调用条件是：TTY 中间层维护的内核写缓冲区由空变为非空。从前文 TTY 中间层初始化代码可以看出，在调用 ttyDevCreate 函数时，我们传入了两个参数，指定了内核读写缓冲区的大小（512 字节），其中写缓冲区用以缓存用户写入的数据，写缓冲区在管理上作为一个环形存储区，如果用户写入的频率较高，那么在缓冲区中已有数据被发送之前，又有用户数据到来。即在底层串口一个字节一个字节的发送数据时，用户可以并行的写入数据到缓冲区中，此时由于内核写缓冲区中已经存在数据，即缓冲区非空，此时 TTY 中间层并不调用 arm926UartTxStartup 函数，因为该函数的所有功能就是触发底层串口启动一个发送过程，而当缓冲区非空时，则表示底层驱动已然正在进行着发送过程。换句话说，底层驱动的 arm926UartTxStartup 函数实现必须启动一个动作，这个动作将不断的读取内核写缓冲区的数据发送，直到缓冲区变为空。这一点非常重要，否则串口驱动与 TTY 中间层将无法协调工作。

由于内核写缓冲区空间有限，如果用户写入数据的频率过高，造成写缓冲区变满，此时上层用户任务将被设置为挂起等待状态，等待缓冲区变为非空。用户任务被挂起以及唤醒的操作由 TTY 中间层负责完成。

上一节中，我们讨论到串口收发数据的两种基本工作方式：中断和轮询。对于串口数据的接收，通常不可避免的要使用中断，而数据的发送则通常使用轮询。实际上这要比使用中断发送数据要有效地多。基于这一点，arm926UartTxStartup 函数实现将启动一个 while 循环，采用轮询方式将内核写缓冲区中的当前数据清空。该函数的基本实现代码如下。

```
LOCAL int arm926UartTxStartup
```

```
(
    SIO_CHAN * pSioChan       /* channel to start */
)
{
    ARM926_CHAN * pChan = (ARM926_CHAN *)pSioChan;
    char outchar;
    if( pChan ->intrmode == TRUE )
    {

        while( (*pChan->getTxChar) (pChan->getTxArg, &outchar) != ERROR )
        {
            DAVINCIEVM_UART_putChar(outchar);
        }
        return OK;
    }
    else
        return ENOSYS;
}
```

注意：我们只是在发送数据时采用轮询方式，实际上我们还是采用串口中断工作方式。这与设备的轮询工作模式是两个概念。下文中将介绍用于系统调试目的的串口轮询工作模式。arm926UartTxStartup 函数实现代码比较简单，但是完成了与 TTY 中间层之间的协调工作。即当 arm926UartTxStartup 函数被调用，无论其继续调用其他函数，还是自身实现，必须清空当前内核写缓冲区的所有数据，才能终止本次发送过程，这即是前文中说到的“启动一个发送过程的”含义。DAVINCIEVM_UART_putChar 函数是底层发送数据实现函数，该函数每次接收一个字节，操作串口寄存器将数据发送出去，注意 DAVINCIEVM_UART_putChar 函数每次发送字节前，都检查发送就绪位，只当发送就绪位表示串口可接收下一个字节时，才将要发送的这个字节写入相关（数据）寄存器。所以 DAVINCIEVM_UART_putChar 是一个阻塞函数，而 arm926UartTxStartup 函数也是一个阻塞函数。也就是说，如果用户写入频率不高，那么每次写入时内核写缓冲区都将为空，此时 arm926UartTxStartup 函数将被 TTY 层调用，而用户任务将等待底层 arm926UartTxStartup 函数执行完毕，即用户层调用 write 函数将阻塞于该函数直到写入的数据真正的从串口设备中发送出去。这对于某些仅仅想通过串口输出信息的任务而言可能无法容忍，此时可以使用 logMsg 函数通过默认的串口通道打印，而不需要专门通过 open，write 等标准函数打开一个非标准通道进行信息的输出。如果非要这么做，而用户任务不能容忍等待，那么可以创建一个后台任务，模拟 tLogTask 对信息通过非标准串口通道进行输出。

有些读者可能想到使用中断发送数据，实际上，频繁中断造成的任务等待比以上的等待时间可能更长，而且每次中断的响应都要消耗不小的资源，对于 Vxworks 操作系统而言，要经过多次转移（关于 Vxworks 下中断的详细内容，请参见本书前文相关章节）。笔者始终坚持对于串口这样的每次只能发送一个字节数据的设备，不建议采用中断，以上 arm926UartTxStartup 函数实现是具体项目中使用的代码，此种工作方式并没有对系统造成影响，而且工作的很好。

最后再次提醒，虽然我们使用查询就绪状态位的方式进行数据的发送，而不是采用中断，但

是整个设备依然是工作在中断模式下，数据接收和错误报告都是通过中断通知。这与轮询工作模式是两回事，不可简单的认为由于发送数据采用查询方式，就是轮询工作模式。实际上轮询工作模式在调用的函数上就与以上中断工作模式不同。轮询工作模式下，arm926UartTxStartup 不会被 TTY 中间层使用，串口设备所有的设备被禁止。TTY 中间层通过如下两个函数完成与底层串口驱动以及串口硬件设备的数据交互：arm926UartPollInput，arm926UartPollOutupt。

7.5.6 串口轮询工作模式函数实现

在网口驱动尚未开发或者调试成功的情况下，串口通道一般都暂时被用于 WDB（Wind DeBug）调试通道。在被用作调试通过的情况下，串口设备将工作在轮询模式，此时 TTY 中间层与底层驱动和串口设备之间的数据交互将不再使用上文介绍的 arm926UartTxStartup 函数以及中断，而是使用如下两个函数：arm926UartPollInput，arm926UartPollOutupt。当然如果串口设备从不被用作 WDB 调试通道，则这两个函数设计为空实现，为了串口驱动结构的完整性，建议依然保留这两个函数的定义。
如下是在 config.h 文件中的宏定义，表示使用串口通道 1 作为 WDB 调试通道。

```
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE            WDB_COMM_SERIAL
#undef WDB_TTY_CHANNEL
#define WDB_TTY_CHANNEL        1     /*default Sio SERIAL channel */
#undef WDB_TTY_BAUD
#define    WDB_TTY_BAUD              115200
```

为了支持轮询工作模式，除了实现以上两个函数外，对于前文中已经介绍的 arm926UartIoctl 函数实现也提出了一些要求，即必须响应如下选项：SIO_MODE_SET，SIO_MODE_GET，SIO_AVAIL_MODES_GET。前文中已经给出了对这些选项的标准响应代码。

（1） 数据发送函数 arm926UartPollOutput 实现
该函数在 WDB 调试模式下被上层调用完成数据的发送，实现思想非常简单，即操作具体的串口设备将作为参数传入的字节发送出去。该函数基本实现代码如下。

```
/*******************************************************************************
* sndsPollOutput - output a character in polled mode
*
* RETURNS: OK if a character arrived, EIO on device error, EAGAIN
* if the output buffer if full. ENOSYS if the device is
* interrupt-only.
*/
LOCAL int arm926UartPollOutput
(
    SIO_CHAN * pSioChan,
    char outChar
)
{
```

```
    ARM926_CHAN * pChan = (ARM926_CHAN *)pSioChan;
    CSL_UART_S* udev;
    UINT32  status;

    udev = pChan ->regs;

    /* is the transmitter ready to accept a character? */
    status = udev->LSR;
    if(status&( CSL_UART_LSR_RXFIFOE_MASK| CSL_UART_LSR_FE_MASK |
                CSL_UART_LSR_OE_MASK | CSL_UART_LSR_PE_MASK))
    {
        pChan->errcount ++;
        CSL_uartReset(pChan);
        CSL_uartConfig(pChan);
    }

    if((status)&( CSL_UART_LSR_THRE_MASK))
    {
        CSL_FINSR(udev->LCR, 7, 7, 0);
        udev->THR = (ARMREG)(outChar);
        return (OK);
    }
    else
        return (EAGAIN);
}
```

函数前注释给出了函数实现上的基本处理方式，如果串口设备不支持轮询工作模式，这个函数实现是简单返回 ENOSYS 即可。
arm926UartPollOutput 具体实现上比较简单，从函数的调用原型可以看出该函数的使用环境，每发送一个字节，就调用该函数一次。

（2） 数据接收函数 arm926UartPollInput 实现
该函数被调用每次从串口设备接收一个字节，注意这个函数并非被底层驱动自身调用进行串口读取操作，而是提供给 TTY 中间层的函数。
该函数基本实现如下。

```
/*******************************************************************************
* sndsPollInput - poll the device for input
*
* RETURNS: OK if a character arrived, EIO on device error, EAGAIN
* if the input buffer if empty, ENOSYS if the device is
* interrupt-only.
*/

LOCAL int arm926UartPollInput
```

```
(
    SIO_CHAN * pSioChan,
    char *      thisChar
)
{
    ARM926_CHAN * pChan = (ARM926_CHAN *)pSioChan;
    CSL_UART_S* udev;
    UINT32  status;
    char rByte;

    udev = pChan ->regs;

    status = udev->LSR;
    if( status&(CSL_UART_LSR_RXFIFOE_MASK| CSL_UART_LSR_FE_MASK |
                    CSL_UART_LSR_OE_MASK | CSL_UART_LSR_PE_MASK))
    {
        pChan->errcount ++;
        CSL_uartReset(pChan);
        CSL_uartConfig(pChan);
    }
    if( status&(CSL_UART_LSR_DR_MASK))
    {

        CSL_FINSR(udev->LCR, 7, 7, 0);
        rByte = CSL_FEXT(udev->RBR, UART_RBR_DATA);
        (*thisChar) = rByte;
        return (OK);
    }
    else
        return (EAGAIN);
}
```

注意：以上两个函数只当串口通道被用作调试通道时才被使用，更上一层即只当串口工作于轮询模式下的时才被使用。在串口被用作正常的信息输入输出通道时，其与上层 TTY 中间层之间的数据交互通道完全不同。所以如果串口设备不支持轮询工作模式（即不支持被用作 WDB 系统调试通道），则可以简单在以上两个函数实现中返回一个 ENOSYS 即可，如下代码所示。

```
LOCAL int arm926UartPollOutput
(
    SIO_CHAN * pSioChan,
    char outChar
)
{
```

```
    return ENOSYS;
}

LOCAL int arm926UartPollInput
(
    SIO_CHAN * pSioChan,
    char *     thisChar
)
{
    return ENOSYS;
}
```

在不支持调试模式的情况下，除了将以上两个函数实现为返回 ENOSYS 外，在 arm926UartIoctl 函数中对于 SIO_AVAIL_MODES_GET 选项的实现也有讲究，此时不可如前文实现中返回（SIO_MODE_INT | SIO_MODE_POLL），而是只返回 SIO_MODE_INT。同时如果在使用 SIO_MODE_SET 设置工作模式时，如果传入的 arg 参数为 SIO_MODE_POLL，则 arm926UartIoctl 需要返回 EIO。经过以上这些的处理，串口设备将只支持正常中断工作模式，不可被用作 WDB 系统调试通道。
那么反过来，串口可不可以设计为只被用作调试通道，理论上可以，但是实际工作中如此使用方式是十分罕见的。笔者没有尝试过，感兴趣读者可以试一试。

底层串口驱动组成中的最后一个部分就是具体操作串口寄存器的函数集合，这些函数平台相关，但是由于串口设备的常用性，基本上当前所有的串口设备都采用相同的寄存器映射方式，操作这些寄存器进行串口数据的收发代码很容易从网上下载或者直接使用开源的 Linux 串口设备的寄存器操作代码，只需进行简单修改即可使用到其他任何操作系统下，因为这部分硬件操作代码是与操作系统本身无关的，故本书不再列举串口设备寄存器操作函数的实现代码。

至此，我们完成底层串口驱动所有代码的介绍。串口驱动底层寄存器操作代码已经非常标准，所以当前串口驱动主要涉及的内容就是如何协调好与特定操作系统的接口层的交互。对于 Vxworks 操作系统而言，就是处理与 TTY 中间层的交互工作。无论最终采用何种机制，其基本目的都是一致的：以一种有效地方式完成操作系统与底层驱动之间的数据交互。在这一点上，Vxworks 操作系统较为出色的完成了这个目的，而其中最为关键的内核组件就是 TTY 驱动中间层，其具体负责与底层串口驱动之间的数据交互，本章前文中对于 TTY 中间层与底层驱动之间的关系已经作为较为详细的分析和说明，此处不再讨论。
上文中使用一节内容对 TTY 中间层的初始化作了一个较为浅显的说明，其后就一直使用 TTY 中间层一个广泛的用词包括整个串口驱动内核接口层，下面我们将深入 TTY 中间层内核组件，较为详细的介绍一些 TTY 中间层的内部组成以及阐述一个用户读写请求是如何通过 IO 子系统，TTY 中间层最终到达底层串口驱动的，我们着重关注一下在 TTY 中间层所做的处理。

7.6 深入 TTY 中间层

在“TTY 中间层初始化”一节中我们介绍了两个重要函数：ttyDrv，ttyDevCreate，其中 ttyDrv 完成 TTY 中间驱动层向 IO 子系统的驱动注册，即通过 iosDrvInstall 函数向 IO 子系统注册 TTY 中间层驱动函数。自此所有的串口设备请求将通过 IO 子系统传递给 TTY 中间层驱动函数。TTY 中间层注册的驱动函数为：ttyOpen，ttyClose，tyRead，tyWrite，ttyIoctl。ttyDevCreate 函数完成 TTY 中间层与底层串口驱动之间的相互注册，使能串口中断，使用传入的字符串创建一个串口设备节点并通过调用 iosDevAdd 函数将其添加到系统设备列表中。自此用户就可以通过诸如 open 之类的标准接口函数对串口设备进行打开，读写，关闭，控制等操作。所有的串口设备中，一般第一个注册到 TTY 中间层的串口设备被用作标准输入输出，此时对于该串口的输入输出不用之前调用 open 打开串口设备，Vxworks 操作系统启动过程中已经使用 open 调用打开了这个设备，返回的文件描述符被复制为标准输入输出，错误输出，即 0，1，2 三个系统文件描述符都指向了这个串口设备。Vxworks 操作系统内运行的所有任务都可以通过标准打印语句 printf，logMsg 通过这个串口设备输出信息或者读取信息。除了这个被用作标准输入输出的串口设备外，其他串口设备的操作必须遵循标准的设备操作流程，即打开设备，操作设备，关闭设备。虽然串口设备的打开操作只是上层软件上的一种操作，而不会对底层串口硬件施加行为，但是打开操作后将返回一个重要的参数：文件描述符。用户层此后所有的操作都必须基于这个文件描述符。

从 ttyDrv 注册的五个函数中我们看到 ttyOpen，ttyClose 就是 TTY 中间层对用户层打开和关闭串口设备的响应函数。而 ttyIoctl 函数则是用户层 ioctl 的 TTY 层响应函数。虽然 open，close，ioctl 标准用户层调用在 TTY 层实现为对应的三个响应函数，但是从前文中底层串口驱动代码的分析中我们看到底层驱动中并没有对应这三个函数的实现，实际上在 TTY 层就进行了转移，即 ttyOpen，ttyClose，ttyIoclt 函数最后都调用了底层串口驱动 ioctl 函数实现，对应上文的例子就是 arm926UartIoctl 函数。ttyOpen 将以 SIO_OPEN 选项调用 arm926UartIoctl，而 ttyClose 将以 SIO_HUP 选项调用 arm926UartIoctl，而 arm926UartIoclt 则是 ttyIoctl 的底层标准实现。

TTY 中间层提供的用户层 read，write 读写响应函数是 tyRead，tyWrite。细心的读者可能发现这两个函数的命名方式不同于以上介绍的其他三个函数，即这两个函数不是命名为 ttyRead，ttyWrite。这正是区别所在，实际上 tyRead，tyWrite 由 tyLib 库提供而 ttyOpen，ttyClose，ttyIoctl 包括 ttyDevCreate 函数以及 ttyDrv 函数则是由 ttyDrv 库提供。前文中我们分析出 TTY 中间层需要向底层驱动提供两个函数：内核写缓冲区读取函数，内核读缓冲区写入函数，实际上这两个函数也是由 tyLib 库提供，其中内核写缓冲区读取函数为 tyITx，内核读缓冲区写入函数为 tyIRd 函数，这两个函数都定义在 tyLib 库中，ttyDevCreate 函数只是“代为转交”。

由此可见，TTY 中间层具体有两个部分组成：ttyDrv 库和 tyLib 库。这两个内核组件共同组成 TTY 中间层。二者之间并无上下层关系，而是平行关系。从如上 ttyDrv 函数中调用 iosDrvInstall 函数时提供的函数即可看出，ttyDrv 和 tyLib 都贡献了一部分，不过从实现上来看，tyLib 将完成 TTY 中间层基本所有的实质性工作，包括与底层串口驱动的具体的数据交互。除了 ttyDrv 库提供的 ttyIoctl 函数外，tyLib 也提供了一个 tyIoctl 函数，加上底层串口提供的 arm926UartIoctl 函数，这三个函数之间的调用关系是：用户层调用 ioctl 对设备进行配置和控制，这个请求通过 IO 子系统首先传递给 ttyIoctl 函数，ttyIoctl 函数本身并不完成任何实质性工作，其调用其他函数完成用户请求：首先调用底层驱动提供的 arm926UartIoctl 函数，如果 arm926UartIoctl 返回值不是 ENOSYS，则直接返回；如果 arm926UartIoctl 返回

ENOSYS，即表示底层驱动不支持对应请求，此时 ttyIoctl 将继续调用 tyIoctl 完成用户请求，所以 tyIoctl 作为最终的所有底层未处理请求的“终结地”。

如下图 7-2 所示详细了给出了 TTY 中间层与其上下层之间的关系以及 TTY 中间层内部关系。

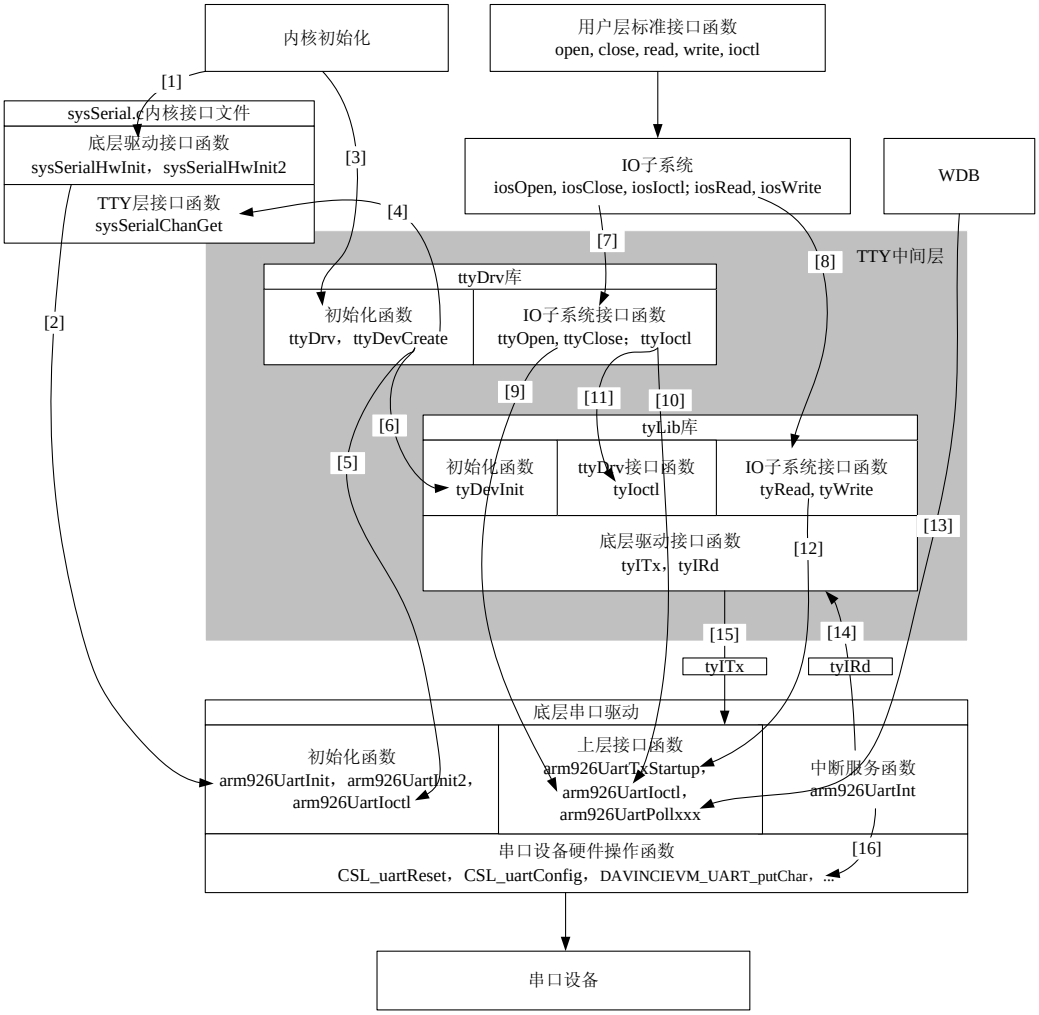


图 7- 2 TTY 中间层与上下层及其内部函数调用关系

[1] Vxworks 内核启动过程中通过调用串口内核接口文件 sysSerial.c 中定义的函数 sysSerialHwInit，sysSerialHwInit2 完成对底层串口驱动的初始化。

[2] sysSerialHwInit，sysSerialHwInit2 通过调用底层驱动中定义的初始化函数 arm926UartInit，arm926UartInit2 完成具体的初始化。

[3] Vxworks 内核启动过程中调用 ttyDrv 完成 TTY 中间层向 IO 子系统的注册，并调用 ttyDevCreate 函数完成 TTY 中间层与底层驱动之间的注册，串口设备中断使能，串口设备创建。

[4] ttyDevCreate 调用 sysSerial.c 中 sysSerialChanGet 函数获取对应串口设备的结构指针。

[5] ttyDevCreate 调用底层驱动 arm926UartIoctl 函数使能串口设备中断。

[6] ttyDevCreate 调用 tyLib 库初始化函数 tyDevInit 对 tyLib 库进行初始化，其中重要的一个初始化工作就是创建内核读写缓冲区，所以 TTY 中间层中串口设备内核读写缓冲区实际上是由 ttyLib 进行维护的，这也解释了为何 TTY 中间层提供给底层驱动的两个内核读写缓冲区操作函数（tyITx，tyIRd）最终是由 tyLib 提供的，ttyDrv 只是（通过 ttyDevCreate 函数）“代为转交”。同时 ttyDevCreate 函数还将底层驱动 arm926UartTxStartup 函数地址提供给

tyLib，所以对于 arm926UartTxStartup 的调用是由 tyLib 进行的。
[7] 对于串口设备 open，close，ioclt 的请求将通过 IO 子系统传递给 ttyDrv 库提供的 ttyOpen，ttyClose，ttyIoctl 函数进行处理。
[8] 对于 read，write 的请求将通过 IO 子系统传递给 tyLib 库提供的 tyRead，tyWrite 函数进行处理。
[9] ttyOpen，ttyClose 继续将用户请求传递给底层驱动 arm926UartIoctl 函数进行处理，并由该函数完成最终的请求服务。
[10] ttyIoctl 函数首先调用底层驱动 arm926UartIoctl 函数对用户设备控制和配置操作进行响应。
[11] 对于底层驱动不支持的选项，ttyIoctl 最终调用 tyLib 库提供的 tyIoctl 函数进行响应。
[12] tyWrite 函数完成用户的写数据请求，将用户数据暂存入内核写缓冲区中，此后根据缓冲区的状态，该函数可以进一步调用底层串口驱动中 arm926UartTxStartup 函数启动一个发送过程将用户数据发送出去。
[13] 对于支持轮询工作模式的串口设备，WDB 内核组件将通过 TTY 中间层调用底层驱动 arm926UartIoctl 以及 arm926UartPollInput，arm926UartPollOutput 函数使用串口设备进行调试。
[14] 底层驱动中断服务函数全权负责串口设备的数据接收，其通过调用 tyIRd 函数将数据写入 TTY 中间层维护的内核读缓冲区中，供 tyRead 函数进行读取。
[15] 底层驱动 arm926UartTxStartup 函数将使用 tyITx 函数从内核写缓冲区读取数据，最终操作串口设备将数据发送出去。
[16] 底层驱动中断服务函数，arm926UartTxStartup 函数等等与上层接口的函数最终都需要调用串口硬件寄存器操作函数控制串口设备完成数据的收发工作。

7.7 本章小结

本章我们首先从底层串口驱动与内核组件之间的关系出发，讨论了要完成底层驱动与操作系统之间的数据交互，二者之间必须相互提供哪些功能函数，继而讨论了 Vxworks 下基本的串口驱动内核接口设计：TTY 中间层。此后较为简洁的介绍了 TTY 中间层的初始化过程。在完成对 TTY 中间层的初始化讨论后，我们对底层串口驱动内核接口文件 sysSerial.c 进行了分析，并详细介绍了其代码实现，该文件可以视作为串口驱动的一个固有组成部分。之后我们总结出了底层串口驱动的组成并进而对其实现以实例代码的方式进行了详细的介绍。最后，对 TTY 中间层的内部组成以及其与上下层之间的函数调用关系进行了较为简洁的说明。串口驱动是平台外设驱动中较为简单的一类，其简单性主要表现在硬件操作的简单上，其内核接口由于在 IO 子系统之下介入了 TTY 中间层，相比普通字符设备驱动，变得有些繁琐。但是从串口设备的使用效率来看，这也是值得的，而且在理解了 TTY 中间层与底层驱动之间的关系后，底层串口驱动的设计也变得简单。读者在阅读本章时，首先应在把握串口驱动总体结构的基础上，结合实际代码进行理解。

第八章 块设备驱动

到目前为止，我们对于驱动的讨论仅限于字符设备。本书前文中将设备总体上分为了三类：字符设备，块设备以及网络设备。块设备是非常重要的一类设备，基本上所有通用操作系统都需要块设备的配合才能使得系统本身正常运行或者更确切的说，块设备的存在才让操作系统的功能变得强大，其保存操作系统启动参数，提供给进程运行所需的数据以及永久保存进程生成的或从外界读取的数据，块设备的存在使得大多数进程的工作方才显得有意义。对于应用层而言，块设备以文件系统的形式存在，用户以操作文件和目录的方式访问块设备。
我们将块设备定义为每次只能以数据块的方式进行数据写入和读取的设备，块设备最大的优势就是容量大，可以存储大量的数据。但是操作时间长。比较常见的块设备就是硬盘设备，这是使用最为广泛的一类块设备。硬件设备每次只能以一个扇区（512 字节）为单位进行读写。可以对同一数据进行反复读写，这一点是块设备区别于一般字符设备和网络设备的特点之一。正因如此，块设备被大量使用在需要永久存储某些数据的场合。现在的 PC 机都配备有大容量的硬盘块设备，用以保存用户日常操作数据。没有这些块设备作为数据存储，操作系统本身的价值将大打折扣。
由于硬盘设备体积大，虽然在通用 PC 上使用非常广泛，但是在嵌入式平台上一般很少使用，这类平台上大多使用 FLASH 存储介质。FLASH 设备本质上也是一类块设备，但是其操作方式与硬盘设备有些差别，对于 FLASH 设备操作方式及其驱动编写，在本书后面将有专门章节进行介绍。本章将着重于硬盘这一类块设备驱动的设计和实现。

操作系统下块设备的使用方式与字符设备和网络接口设备有很大的不同，不同于字符设备和网络接口设备主要用以与外界进行数据交互，块设备的基本用途是保存数据，这些数据可以是操作系统运行本身所需的关键参数，任务运行所需参数，任务运行中产生的数据，以及字符设备和网络接口设备从外界接收的数据等等。块设备在操作系统管理下就是一个数据“仓库”。为了对这些数据进行分类管理以及便于用户使用和操作，块设备总是以文件系统的形式提供给应用层以及操作系统上层。即操作系统内核驱动接口层加上块设备底层驱动共同完成块设备的模拟工作，将块设备封装成一个具有文件和目录结构的层次性设备提供给操作系统其他组件以及应用层用户程序。所以块设备总是受操作系统文件系统中间层的管理。Vxworks 下也不列外。从本书前文分析中，我们知道 Vxworks 下所有类型设备的驱动都由 IO 子系统直接或者间接的管理。本书第 4 章中讨论的普通字符设备驱动受 IO 子系统的直接管理，第 5 章介绍的串口设备驱动则受 IO 子系统下 TTY 中间驱动层管理，TTY 中间层则直接受 IO 子系统的管理。对于本章将要讨论的块设备驱动，其则直接受文件系统中间层管理，而毫无疑问文件系统中间层将向 IO 子系统直接注册，受 IO 子系统的直接管理。那么对于块设备的内核驱动层次就很明显了，如下图 8-1 所示。

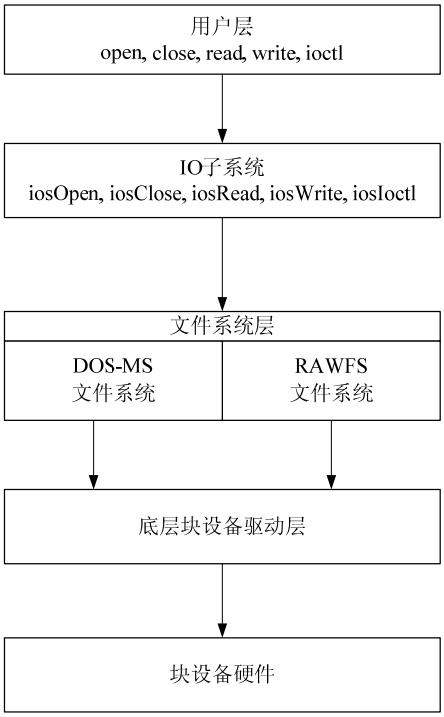


图 8- 1 Vxworks 下块设备驱动内核层次

对于一类块设备只需一个驱动，块设备对于上层的不同视图将由块设备驱动层之上的文件系统层决定。例如对于一个硬盘块设备，我们既可以使用 FAT32 文件格式（Windows 下常用文件系统）对其进行格式化，也可以使用 Ext3 文件格式（Linux 下常用文件系统）对其进行格式化，但是一般而言，底层块设备驱动代码都是相同的。不同的只是上层文件系统封装层实现，使得上层操作变得不同。

Vxworks 较为常见的文件系统有两种，这两种文件系统在 Vxworks 操作系统早期版本中就开始进行支持：MS-DOS 兼容型文件系统 dosFs 与 Raw 原始文件系统 rawFs。MS-DOS 兼容型文件系统使用 FAT12，FAT16 或者 FAT32 文件格式格式化硬盘设备，此时整个硬盘块设备将以文件和目录的视图行式对应用层和操作系统其他组件（文件系统组件之外的组件）可见，这也是我们通常常见的视图形式。Vxworks 下常见的另一类文件系统是 Raw 原始文件系统，当 Vxworks 操作系统无法解读一个块设备上建立的文件系统格式时，其默认的将这个块设备封装成一个原始文件系统，即整个块设备将被当做一个单一的文件使用。Vxworks 也支持其他类型的文件系统，如支持 CD-ROM 数据读取的 cdrom 文件系统，支持磁带机读写的 tapeFs 文件系统，还有提供数据一致性高可靠性的 HRFS 文件系统，这些文件系统类型早期 Vxworks 操作系统版本并不支持，其次也没有 dosFs 以及 Raw 两种文件系统常用。本章将只以 dosFs 和 rawFs 为例介绍底层块设备驱动的设计和实现。

8.1 rawFs 文件系统

Vxworks 为只需基本硬盘 IO 读写操作的平台提供了 rawFs 文件系统，该文件系统由 rawFsLib 内核组件支持。rawFs 文件系统将整个硬盘设备当做一个单一的文件进行读写。虽然 dosFs 文件系统从一定程度上也支持这个功能，但是由于 rawFs 文件系统本身从基本功能实现的角

度作了很多的优化，故相比较 dosFs 文件系统提供的类似功能具有更大的优势。rawFs 文件系统不对硬盘块设备中存储的内容做任何分类或者管理，也不对硬盘内部区域进行划分，这个硬盘设备空间被作为一个“平坦”的区域，没有任何层次性划分。当用户需要对基于 rawFs 的硬盘块设备进行操作时，其首先使用 open 函数打开这个硬盘设备，open 函数返回一个文件描述符，这个文件描述符将整个硬盘作为一个单一文件进行操作。所有使用 open 函数打开这个基于 rawFs 的硬盘设备的任务都对硬盘设备内的任何一个区域具有读写权利。数据在这个硬盘设备内的组织和安排完全由各任务之间进行协调，rawFs 不对这些进行负责。其仅仅在底层硬盘块设备驱动之上提供了一个中间层，对用户读写的数据进行中间缓冲，维护内核文件系统组织结构的统一性。用户以及内核其他组件对于基于 rawFs 的块设备进行 open 调用时传入的路径名直接就是块设备节点名，其后不附加任何文件或者目录名。因为此时块设备整体就被单作一个单一文件，没有文件和目录层次结构。对于多个任务同时写硬盘设备的操作，每个任务一般都需要指定其写入的偏移位置，使得各任务写入的数据不至于相互覆盖。

由于 rawFs 文件系统将整个硬盘块设备当做一个单一的文件进行操作，所以该文件系统并没有如同 dosFs 文件系统中提供的格式化之类的函数。rawFs 文件系统并不查看经过其传递的数据格式，数据格式的约定和解释由写入和读取数据的任务负责。因而 rawFs 没有 dosFs 文件系统中诸如超级快，文件节点位图等结构概念。当 Vxworks 操作系统对一个挂载的块设备无法辨别其现有文件系统类型时，默认的将这个块设备以 rawFs 文件系统进行挂载。所以对于一个已创建有文件系统的块设备而言，如果在 Vxworks 下无法识别文件系统类型，则不可以对该块设备进行读写操作，否则会破坏原有文件系统的内部数据格式，造成原有文件系统的不可恢复性损害，此时这个块设备文件系统将真正无法被任何操作系统识别了。

rawFs 文件系统由 rawFsLib 库实现，这个库可在 Vxworks 操作系统启动过程中被自动的初始化。当 INCLUDE_RAWFS 宏被定义时，usrRoot 函数将调用 rawFsInit 函数初始化 rawFs 文件系统内核组件。rawFsInit 调用原型如下。

```
STATUS rawFsInit
    (
    int    maxFiles    /* max no. of simultaneously opened file descriptors */
    );
```

注意参数表示同时打开基于 rawFs 文件系统的块设备进行操作的文件描述符数量。由于整个块设备此时被当做一个单一文件，所以参数 maxFiles 并非是指最大打开的文件数量。

rawFsInit 函数完成：（1）rawFs 文件系统中间层向 IO 子系统的注册：调用 iosDrvInstall 函数注册 rawFs 中间层驱动到系统驱动表中；（2）创建 rawFs 文件系统中间层管理组件。

rawFs 向 IO 子系统注册的驱动函数如下。

creat	delete	open	close	read	write	ioctl
rawFsCreate	NULL	rawFsOpen	rawFsClose	rawFsRead	rawFsWrite	rawFsIoctl

注意：并无 rawFsDelete 函数，因为 rawFs 文件系统对整个块设备本身进行操作，而并非块设备中某个文件进行操作，故其不提供文件或者目录删除功能。这一点也可以从普通字符设备和 TTY 中间层注册的驱动函数看出。creat 和 delete 用于具有文件和目录结构层次的文件系统中，用以在块设备中创建和删除一个文件或者目录，如果设备本身内部不区分层次（如字符设备以及 rawFs 管理下的块设备），则 creat 一般就直接实现为 open 打开操作，而 delete 则为空实现。从下一节 dosFs 文件系统注册的驱动函数可以看出，dosFs 文件系统提供独立的 creat 和 delete 函数实现。

rawFsInit 只是完成 rawFs 文件系统中间层向 IO 子系统的注册，即此时只是建立了 rawFs 中间层与上层的联系，使得 IO 子系统可以将用户请求传递给 rawFs 文件系统中间层。但是还存在两个最为关键的工作尚未完成：（1）块设备节点创建，即将块设备添加到系统设备列表中，使得用户可以使用这个设备；（2）完成 rawFs 文件系统中间层与底层块设备驱动之间的联系，从而使得用户的请求最终可以传递给底层块设备驱动，驱动块设备进行数据的交互。这两个工作的完成将由 rawFsDevInit 函数完成。rawFsDevInit 函数调用原型如下。

```
RAW_VOL_DESC *rawFsDevInit
    (
    char        * pVolName,         /* volume name to be used with iosDevAdd */
    BLK_DEV     * pDevice           /* a pointer to a BLK_DEV or a CBIO_DEV_ID */
    );
```

该函数返回一个 RAW_VOL_DESC 结构指针，该结构定义在 h/rawFsLib.h 头文件中，如下所示。

```
/*target/h/rawFsLib.h*/
typedef struct      /* RAW_VOL_DESC */
{
    DEV_HDR     rawVdDevHdr;            /* std. I/O system device header */
    int         rawVdStatus;        /* (OK | ERROR) */
    SEM_ID      rawVdSemId;         /* volume descriptor semaphore id */
    CBIO_DEV_ID     rawVdCbio;          /* CBIO handle */
    int         rawVdState;         /* state of volume (see below) */
    int         rawVdRetry;         /* current retry count for I/O errors */
} RAW_VOL_DESC;
```

rawFsDevInit 需要两个参数：

参数 1：块设备节点名称，这个名称将在 open 调用中被用作路径名。

参数 2：BLK_DEV 结构指针，这个结构由底层块设备驱动初始化，其中包含着块设备关键信息以及由底层驱动实现的块设备读写函数地址。这个 BLK_DEV 结构是文件系统中间层与底层块设备驱动层之间联系的“桥梁”，其功能类似于 IO 子系统使用的 DEV_HDR 结构。我们从以上 RAW_VOL_DESC 结构定义可以看出，其第一个成员即是 DEV_HDR 结构，实际上最后注册到系统设备列表中的就是一个 RAW_VOL_DESC 结构，IO 子系统只使用其第一个成员变量 DEV_HDR 结构，其他成员变量由 rawFs 中间层自身使用。而 BLK_DEV 结构是由文件系统层提供，用以获取底层块设备驱动信息的“媒介”。

上文中刚刚提到为了将一个用户层请求转换成对底层块设备的具体操作，上文中图 1 所示的内核所有层次之间必须建立联系，rawFsInit 函数完成 IO 子系统与 rawFs 中间层之间的联系，其联系建立方式是调用 IO 子系统提供的注册函数 iosDrvInstall，将 rawFs 中间层驱动函数提供给 IO 子系统层；而此处的 rawFsDevInit 函数则完成 rawFs 中间层与底层块设备驱动之间的联系，其联系建立方式是底层块设备驱动将 rawFs 所需的信息填入 BLK_DEV 结构中，并以参数形式调用 rawFsDevInit 函数提供给 rawFs 中间层，从而完成 rawFs 中间层与底层块设备驱动之间的联系。传入的 BLK_DEV 结构在 rawFsDevInit 函数中将被封装成 CBIO_DEV_ID 结构被保存到 RAW_VOL_DESC 结构的 rawVdCbio 字段中。除此之外，

rawFsDevInit 还使用参数 1 指定的名称创建一个块设备，并将其添加到系统设备列表中。rawFsDevInit 调用完毕后，底层块设备就处于可用状态，用户可以使用 open 打开这个块设备并对其进行读写操作了。BLK_DEV 结构的具体定义在下文中将有专门一节进行介绍。

由于块设备本身特殊的驱动方式，块设备初始化完成后一般就处于可用状态，所以用户打开块设备操作请求的大部分工作都已在 rawFs 中间层完成（分配资源，返回文件描述符），底层块设备驱动 ioctl 函数仅被调用检查块设备的就绪状态（这是还是针对软盘块设备而言的，硬盘块设备平时都是在就绪状态）。
下面我们以一个用户层块设备打开 open 函数调用为例，介绍这个设备打开请求是如何层层传递到底层块设备驱动层的。
当然为了使得应用层能够对块设备进行打开操作，首先必须创建一个块设备节点，并将其添加到系统设备列表中。
从上文介绍中，为了使用基于 rawFs 文件系统的块设备，我们首先必须调用底层块设备驱动程序初始化一个 BLK_DEV 结构，然后使用该结构作为参数调用 rawFsDevInit 函数。所以我们首先定义一个 BLK_DEV 结构指针，指向底层块设备驱动返回的这个 BLK_DEV 结构。
BLK_DEV *pBlkDev;
其次 rawFsDevInit 函数返回一个 RAW_VOL_DESC 结构指针，虽然程序中暂时不使用该结构指针，但是我们还是对其进行保存，以便之后可能的使用。
RAW_VOL_DESC *pRawDesc;

另外为了使用 rawFs 文件系统组件，必须在 Vxworks 操作系统启动过程调用 rawFsInit 函数对 rawFs 进行初始化，即需要定义 INCLUDE_RAWFS 宏。我们假设 rawFsInit 函数在操作系统启动过程中已经被调用对 rawFs 文件系统中间层进行了初始化。

令底层块设备驱动程序定义了 xxxDevCreate 函数，创建一个 BLK_DEV 结构（从下文可见，这实际上是一个驱动自定义结构，只是 BLK_DEV 作为其第一个成员变量），对其进行初始化，而后返回该结构地址。
现在我们首先调用底层块设备驱动中定义的 xxxDevCreate，完成 BLK_DEV 结构的创建和初始化，这个结构在块设备驱动中非常重要，类似于串口驱动中的 SIO_CHAN，完成上层与底层驱动之间的关键信息传递。
pBlkDev=xxxDevCreate(…);
我们暂时忽略 xxxDevCreate 调用所需的参数。
现在我们从底层块设备驱动中获得了 rawFs 所需的信息，可以调用 rawFsDevInit 函数完成 rawFs 文件系统中间层与底层块设备之间的“衔接”了，同时完成块设备节点的创建和添加。
pRawDesc=rawFsDevInit（”/rawBlk”, pBlkDev）;
rawFsDevInit 的第一个参数表示创建块设备节点时使用的节点名称，这个名称将作为 open 函数调用时的路径名；第二个参数指向了一个 BLK_DEV 结构，该结构中包含了文件系统中间层所需的底层块设备驱动的关键信息（如块设备读写函数），这些信息将被 rawFs 文件系统中间层保存，在之后用户层对块设备硬件发起操作时，使用这些信息与底层块设备驱动进行交互。
rawFsDevInit 成功返回后，系统设备列表中将多出一个设备节点，这个设备节点的名称就是“/rawBlk”，确切的说，现在我们的块设备已经可以被用户使用了。
我们将以上使得块设备对用户可用的代码集中到一个函数中，如下：
STATUS createRawBlkDev(){

```
    BLK_DEV *pBlkDev;   //定义一个 BLK_DEV 结构指针。
    RAW_VOL_DESC *pRawDesc;   //定义一个 RAW_VOL_DESC 结构指针。
    pBlkDev=xxxDevCreate(…);   //调用底层块设备驱动函数初始化一个 BLK_DEV 结构。
    //将底层驱动返回的 BLK_DEV 结构作为参数调用 rawFsDevInit 创建块设备节点，
    //以及完成底层驱动与 rawFs 文件系统中间层之间的“衔接”。
    pRawDesc=rawFsDevInit(“/rawBlk”, pBlkDev);

    return OK;
}
```

应用层程序可以使用如下语句对以上创建的基于 rawFs 文件系统的块设备进行操作。

```
void main(void){
    int fd;
    char rdback[256];
    char *msg=”hello, guy.”;
    int msglen=strlen(msg);

    //打开基于 rawFs 的块设备，路径名是调用 rawFsDevInit 时使用的第一个参数。
    if((fd=open(“/rawBlk”, O_RDWR, 0)<0){
        printErr(“Cannot open device.\n”);
        return;
    }
    ioctl(fd, FIOSEEK, 0x100); //将文件指针偏移 0x100 个字节。
    write(fd, msg, msglen); //在偏移块设备起始处 0x100 的地址处写入一些数据。
    close(fd);
    //以下代码将以上写入的数据读回，进行比较。
    if((fd=open(“/rawBlk”, O_RDONLY, 0)<0){
        printErr(“Cannot open device.\n”);
        return;
    }
    ioctl(fd, FIOSEEK, 0x100);
    read(fd, rdback, msglen);
    close(fd);
    printf(“write msg:%s\n, read back:%s\n”, msg, rdback);

}
```

继续上文的讨论，我们深入以上 open 函数调用，查看其调用路径。用户程序下第一层是 IO 子系统层，对于 open 函数的响应函数是 iosOpen 函数，rawFsInit 函数调用时注册 rawFs 文件系统中间层驱动函数到 IO 子系统中，所以 IO 子系统根据系统设备列表匹配到“/rawBlk”字符串后，得知其下一层被调用的函数是 rawFsOpen。rawFsOpen 完成了块设备打开操作所需完成的大部分工作：分配所需资源，获取一个可用的文件描述符句柄，之后调用在 rawFsDevInit 函数调用时传入的 BLK_DEV 结构中指向的底层块设备函数，进行底层块设备驱动中所需完成的打开设备操作，事实上是调用底层块设备驱动 ioctl 函数检查块设备是否

就绪。这个操作主要针对软盘块设备，对于硬盘设备，则始终是就绪的。以上的请求传递中主要经过了两个请求转移，第一次转移发生在 IO 子系统向 rawFs 文件系统中间层之间，此时 IO 子系统将使用路径名匹配系统设备列表中设备名称，由于 rawFsDevInit 函数在创建块设备节点时，使用 rawFsInit 调用时返回的驱动号注册到设备节点中，所以 IO 子系统将调用 rawFs 文件系统注册的 rawFsOpen 函数对用户打开设备操作进行响应，完成请求的第一次转移；第二次转移发生在 rawFs 文件系统中间层与底层块设备驱动之间，这时起着关键作用的就是 BLK_DEV 结构，该结构由底层块设备驱动完成初始化，并提供给 rawFs 文件系统中间层进行管理，BLK_DEV 结构中保存了文件系统中间层所需的所有信息，包括由底层块设备驱动实现的块设备读写，控制函数以及块设备本身的一些参数。所以通过 BLK_DEV 结构，rawFs 文件系统中间层根据请求的类型调用该结构中相关字段指向的底层块设备驱动函数完成应用层请求的最终处理。

如下图 8-2 所示，具体显示了基于 rawFs 文件系统中间层的块设备驱动内核层次结构。

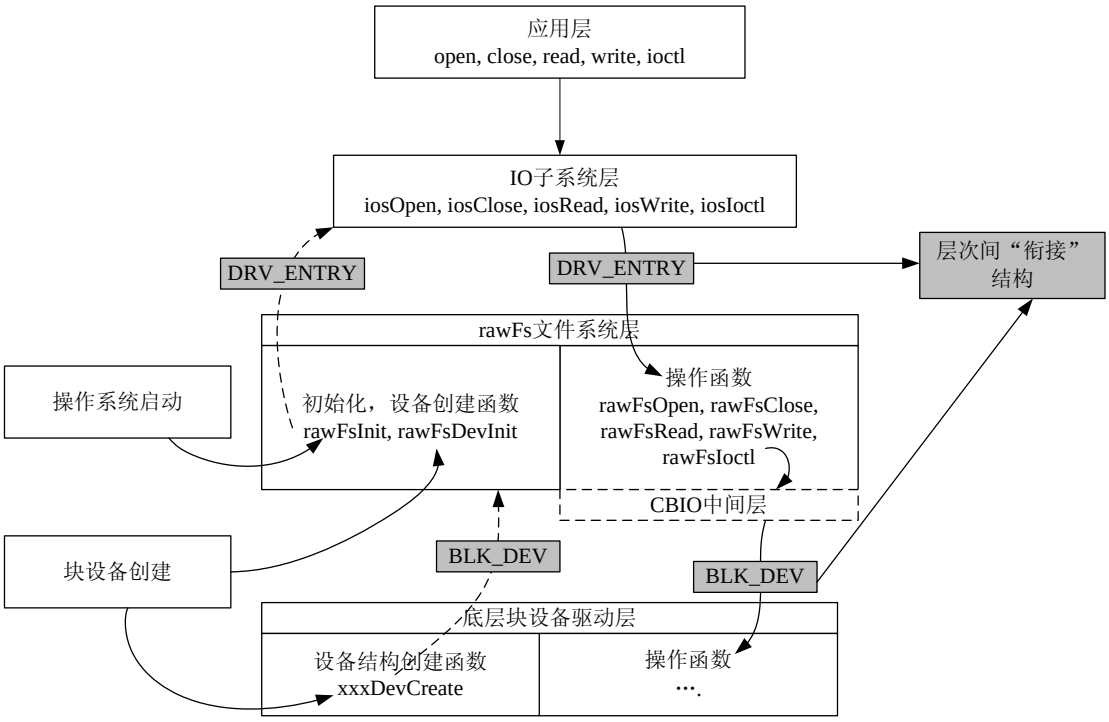


图 8- 2 基于 rawFs 中间层的内核块设备驱动层次结构

图 8-2 中，在 rawFs 文件系统与底层块设备驱动之间显示了一个 CBIO 中间层，这是基于块设备读写操作时间相对较长而在内核所作的缓冲中间层，实际上 Vxworks 内核分为几个 CBIO 中间层，一层套一层完成控制操作和数据的缓冲（通常所说的硬盘缓冲区）。

我们知道，硬盘并不能单个字节的进行数据读写，每次必须以一个扇区为单位进行数据的读写。实际上底层块设备驱动直接打交道的是硬盘控制器，硬盘控制器之下是硬盘驱动器，硬盘驱动器才真正完成数据在物理存储介质上的读写。硬盘驱动实际上驱动的硬盘控制器，与硬盘控制器之间进行数据的交互，交互的通信手段就是中断。对数据读操作为例，当底层驱动有一批数据需要写入硬盘设备时，其首先需要检查硬盘控制器现在是否空闲，即并不在处理其他命令，如果硬盘控制器空闲（或者更确切的说是就绪），那么底层硬盘驱动将根据数据读取的起始扇区，总共要读取的扇区数，驱动器号，磁道号，磁头号写入相关硬盘控制器

寄存器中，启动一个数据读取过程。从该读取命令发送到硬盘控制器，硬盘控制器将驱动硬盘驱动器转动磁盘片，定位磁头，寻址到对应的扇区，读出一个扇区（512 字节）的数据后，发出一个中断，表示已经完成一个扇区的读取操作，底层块设备中断响应函数从硬盘控制器缓冲区中将这一个扇区的数据读取出来后返回。但是事情还没有，当硬盘控制器内部缓冲区中数据被中断处理程序读取后，其继续下一个扇区的读取操作，完成读取后，再次发出中断，让底层驱动再次读取控制器缓冲区中刚才读取的这一个扇区数据，清空缓冲区后，再次读取下一个扇区，知道规定的扇区数被读取完毕。此处读者可以看到，硬盘控制器内部缓冲区每次只能缓冲一个扇区的数据，所以每次从盘片读取一个扇区的数据后，其就发出一个中断，让底层驱动中断处理函数清空其内部缓冲区，继而进行下一个扇区的读取工作，直到要求的扇区数被读取完毕。当然并不是每次读取都会成功，如果读取失败也会发出中断，此时就由底层驱动中断处理函数进行对应的处理，如复位硬盘控制器，重新校准磁头位置。当然现在的硬盘控制器可以有一个很大的内部数据缓冲区，此时无需每次读取一个扇区就发出一个中断，可以将驱动要求的所有扇区读取完毕后，一次性给出中断让底层驱动中断处理函数读取所有的数据。但是无论如何，从驱动发出一个读取数据请求，到所有数据从硬盘控制器传递给底层驱动中间有一个较大的延迟（ms 量级），所以一般而言，读取硬盘数据的任务都要被挂起等待。

由于从硬盘读取数据相当“不易”，所以这些已经读取出来的数据就变得非常“可贵”，如果一个程序将数据从硬盘设备中读取出来使用完毕后直接释放，那么如果另一个程序恰好也要使用相同的数据，那么又要“劳民伤财”的从硬盘设备读取，这对于整个系统的性能极为不利，故对于像硬盘这样的块设备（当然也包括 FLASH 设备），操作系统都会在内存中创建一个硬盘缓冲区，统一管理从硬盘读取的数据以及要写入硬盘的数据，平时一个任务需要使用硬盘中的数据时，先从硬盘缓冲区中查找，如果查找不到，才发起硬盘读取操作，如果正好数据已然存在于硬盘缓冲区中，则直接使用这些数据，而无须动用大量资源从硬盘读取数据。数据的写入也是如此，任务写入的数据实际上是写到硬盘缓冲区中，这些硬盘缓冲区中改变的内容将由内核专门后台进程负责向硬盘设备进行刷新。

Vxworks 操作系统在文件系统之下也提供了硬盘缓冲区，注意这些硬盘缓冲区管理组件是独立于文件系统本身的，任何文件系统类型都可以使用这些下层的硬盘缓冲区，实际上是在文件系统层与底层块设备之间又插入了一个中间层，我们称这个中间层为 CBIO 中间层，实际上 CBIO 中间层内部又分为三个子层次，如上文介绍的 rawFs 文件系统，在调用 rawFsDevInit 函数完成 rawFs 文件系统中间层与底层块设备之间的“衔接”时，rawFsDevInit 函数实现实际上自动在其与底层块设备之间插入了一个 CBIO 基本功能层（CBIO 三个子层次中的最低层），其调用 CBIO 中间层接口函数将底层驱动提供的 BLK_DEV 结构提供给 CBIO 基本功能层，而其（rawFs 中间层）本身则直接使用 CBIO 基本功能层提供的函数。所以应用层请求到达 rawFs 中间层后，其实际上调用 CBIO 中间层中的基本功能层函数，再由这个 CBIO 基本功能层函数根据 BLK_DEV 结构中信息调用底层块设备驱动函数。CBIO 中间层中的其他两个层次-分区层和数据缓冲层必须明确进行相关代码调用才被使用。注意硬盘缓冲区功能由 CBIO 中间层中的数据缓冲层完成，如果只使用 CBIO 基本功能层，则没有硬盘缓冲区，此时将对整个系统的性能造成极为不利的影响，所以对于块设备，一般都需要使用内存硬盘缓冲区。在上文的代码示例中，我们没有使用硬盘缓冲区，下文中我们将给出使用 CBIO 中间层中的数据缓冲层的块设备创建代码示例。

CBIO 中间层本身并非一个单一的层次，其内部继续进行分层，分为三个 CBIO 子层次，每个 CBIO 子层次建立在下一个 CBIO 子层次之上，如下图 8-3 所示。

位于底层块设备之上的 CBIO 子层次被称为 CBIO 基本层，其完成 BLK_DEV 结构的封装，实际上也是这个基本层直接与底层块设备驱动进行数据交互，但是底层驱动总是处于被动调用，其并不知道 CBIO 中间层的存在性，事实上，从以上基于 rawFs 文件系统的块设备使用来看，底层驱动直接返回一个 BLK_DEV 传递给 rawFs 文件系统接口函数，至于这个 BLK_DEV 结构中的信息如何被内核使用则是操作系统本身的事，对于底层块设备驱动而言，其认为与其直接交互就是文件系统中间层，在底层块设备驱动设计上这不会造成任何问题。

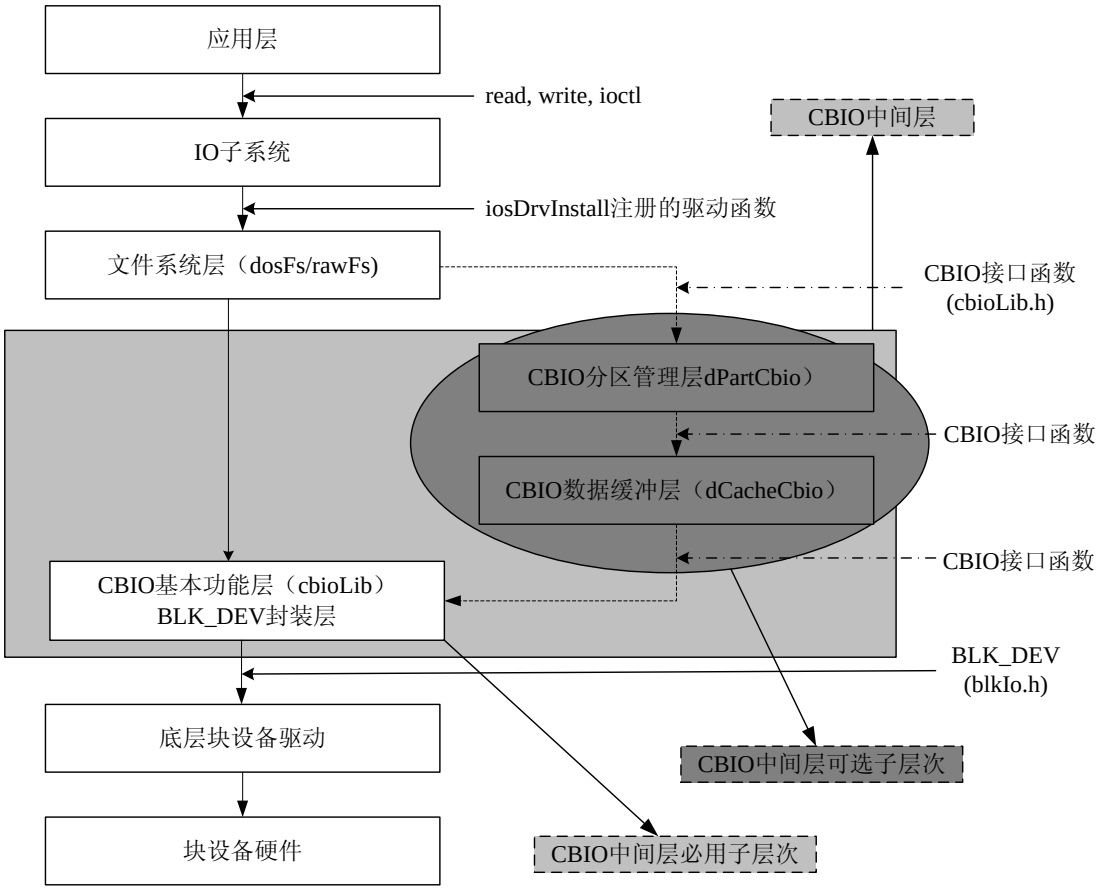


图 8- 3 CBIO 中间层上下关系及其内部结构

而对于使用 CBIO 数据缓冲层和分区管理层的情况，在进行文件系统接口函数调用之前，必须调用两个层次提供的接口函数对底层块设备驱动返回的 BLK_DEV 接口进行封装，添加各自层次信息，而后将这两个层次返回的 CBIO_DEV_ID 结构传递给文件系统接口函数，完成数据缓冲层和分区管理层的添加。

注意：即使是使用 CBIO 中另外两个层次，这两个层次也是建立在 CBIO 基本功能层之上，所以此时与底层块设备驱动直接交互的仍是 CBIO 基本功能层，只是在基本功能层与文件系统层之间又插入的两个层次，这些对于底层块设备驱动都是不可见的，事实上，基本功能层对于底层块设备驱动而言也是不可见的，这些层次的使用完全在块设备实现之外。所以对于底层块设备实现而言，其只对 BLK_DEV 结构负责，无需对 CBIO 中间层负责，这大大简化了底层块设备驱动的设计和实现。

Vxworks 下通常分区管理层使用较少，而数据缓冲层一般都需要使用以提供系统性能（不过

需要消耗部分内存空间)，数据缓冲层提供 dcacheDevCreate 接口函数对底层块设备驱动提供的 BLK_DEV 结构进行封装，其返回一个 CBIO_DEV_ID 类型（cbioDev 结构指针）。dcacheDevCreate 实际上进行两次封装，首先调用 CBIO 基本功能层对底层驱动进行封装，而后其自身对 CBIO 基本功能层进行封装。dcacheDevCreate 函数调用原型如下。

```
CBIO_DEV_ID dcacheDevCreate
    (
    CBIO_DEV_ID    subDev,    /* block device handle */
    char    *      pRamAddr,      /* where it is in memory (NULL = KHEAP_ALLOC)    */
    int            memSize,       /* amount of memory to use */
    char *         pDesc          /* device description string */
    );
```

参数 1：CBIO_DEV_ID 结构，实际上在调用该函数时通常传入的是底层块设备驱动初始化函数返回的 BLK_DEV 结构，由 CBIO 数据缓冲层自己完成 CBIO 基本功能层的封装。
参数 2：数据缓冲区内存地址，即用作硬盘数据缓冲区的内存起始地址，一般传递 NULL，由内核自己分配。
参数 3：硬盘内存缓冲区大小。如果传递 0，则内核将使用默认大小。
参数 4：描述字符串，信息显示时用，对该参数没有特殊要求。

dcacheDevCreate 函数返回一个 CBIO_DEV_ID 结构，这个结构将被用于调用 rawFsDevInit 函数，从而完成文件系统层与 CBIO 中间层的衔接。此时文件系统层将直接与 CBIO 中间层中的数据缓冲层进行交互，而由数据缓冲层与 CBIO 基本功能层之间进行交互，最终 CBIO 基本功能层与底层块设备驱动之间进行交互。具体关系如下图 8-4 所示。

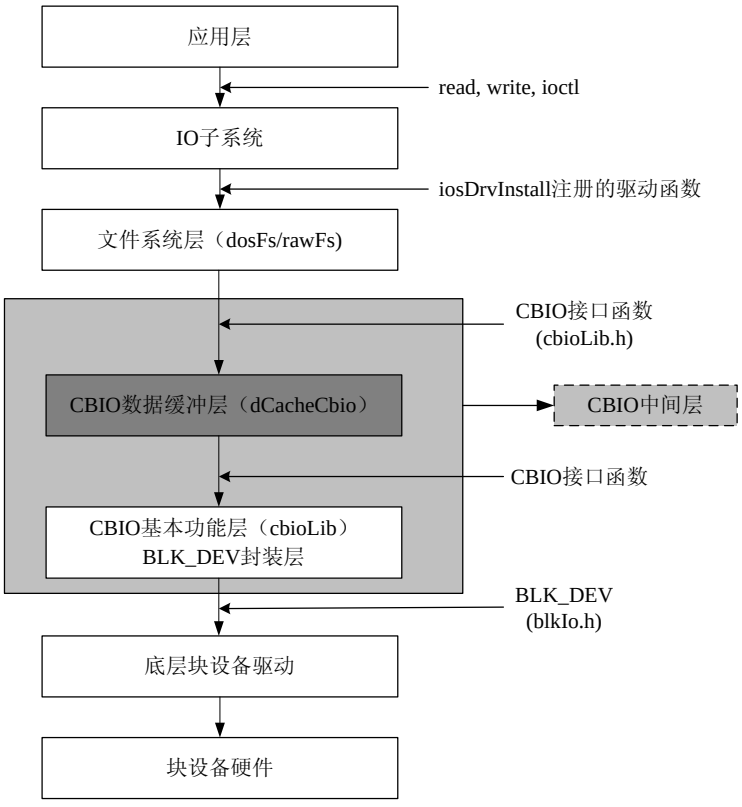


图 8- 4 使用 CBIO 数据缓冲层时块设备内核驱动层次

使用 CBIO 数据缓冲层时块设备创建代码如下。

```
STATUS createRawFsBlkDev(){
    BLK_DEV *pBlkDev;
    //typedef struct cbioDev *CBIO_DEV_ID;
    //cbioDev 结构定义在 h/private/cbioLibP.h 内核头文件中。
    CBIO_DEV_ID pCbioDev;
    RAW_VOL_DESC *pRawDesc;

    pBlkDev=xxxDevCreate(…);
    //调用 dcacheDevCreate 函数进行封装，添加 CBIO 数据缓冲层，缓冲区大小为 128KB。
    pCbioDev=dcacheDevCreate(pBlkDev, NULL, 128*1024, “ata hard disk cache”);
    pRawDesc=rawFsDevInit(“/rawBlk”, pCbioDev);
    return OK;
}
```

应用层对于块设备的使用并不会因为 CBIO 中间层的变化而变化，所以上文中给出的块设备使用示例依然有效。基于 rawFs 文件系统的块设备将整个块设备作为单一文件进行操作，其底层并无文件系统数据结构支持，故也就不存在分区表的概念，所以 CBIO 分区管理层不可使用在 rawFs 文件系统中。我们将在下文 dosFs 文件系统的介绍简单讲述 CBIO 分区管理层的使用及其基本功能。

8.2 dosFs 文件系统

rawFs 只是一种非常简单的文件系统类型，其将整个块设备作为单一的文件进行访问，没有文件和目录层次，整个块设备是一个平坦的区域。dosFs 文件系统提供一种文件和目录的层次性视图，使得应用层对块设备的操作转换成存储在块设备之上的文件和目录的操作。这种文件和目录层次的划分对数据的分类管理起着重要的作用，也是块设备常见的使用方式。块设备的文件和目录层次视图完全由文件系统本身提供，从底层块设备驱动以及块设备本身的角度而言，其只是“看到”数据流，并不对数据本身的含义进行解释。所以当底层块设备驱动被要求向块设备写入一批数据时，这些数据从文件系统角度而言，可能是一个文件的内容，可能是一个目录的内容，也可能是文件系统管理之用的超级块内容，换句话说，这些写入的数据每个都是对文件系统本身有确切的含义的，但是对于底层块设备驱动而言，其只是“看到”有一批数据需要写入到块设备中，至于这批数据是何用途，其一概不过问。这极大的简化了底层块设备驱动代码设计的复杂性。实际上，相比较文件系统本身及其辅助组件的设计，底层块设备驱动设计要简单的多得多。

为了在块设备之上提供文件和目录视图，dosFs 文件系统使用一系列数据结构对数据进行划分。所以使用 dosFs 文件系统的块设备在初次使用前要经过一个格式化的过程，格式化简单的说，就是在块设备之上创建 dosFs 文件系统管理用的数据结构数据，有些场合将这些数据称为元数据，即本身占用一定的块设备存储空间，但是不是用户使用的数据，而是文件系统为管理用户数据而使用的辅助数据，如超级块，超级块实际上是一个数据结构，其占用一个块（如 1KB，两个扇区）的块设备存储容量，存储文件系统总体信息，文件系统的挂载就

是根据超级块中的信息进行的。除了超级块之外，为了建立一个文件系统，还必须有节点位图，逻辑块位图等等文件系统关键数据。此处我们不对文件系统本身实现进行讨论，感兴趣读者可参考开源操作系统 Linux 下的相关代码实现。

类似于上一节介绍的 rawFs 文件系统，dosFs 文件系统在使用前也必须经过初始化，这个初始化也是在 Vxworks 启动过程中完成的。基于 dosFs 文件系统的复杂性，其初始化包括几个方面：（1）dosFs 主模块初始化（dosFsLibInit，属于 dosFsLib 库），对应 INCLUDE_DOSFS_MAIN 宏定义，该宏总体上决定了是否包含 dosFs 文件系统组件；（2）dosFs 格式化功能模块初始化（dosFsFmtLibInit，属于 dosFsFmtLib 库），对应 INCLUDE_DOSFS_FMT 宏定义，注意必须预先定义 INCLUDE_DOSFS_MAIN，这个宏才有意义；（3）FAT 文件分配表管理模块初始化（dosFsFatInit，属于 dosFsFat 库），对应 INCLUDE_DOSFS_MAIN 宏定义；（4）dosFs 一致性检查功能模块初始化（dosChkLibInit，属于 dosChkLib 库），对应 INCLUDE_DOSFS_CHKDSK 宏定义，必须预先定义 INCLUDE_DOSFS_MAIN，这个宏才有意义；（5）dosFs 长文件名处理模块初始化（dosVDirLibInit，属于 dosVDirLib 库），对应 INCLUDE_DOSFS_DIR_FIXED，必须预先定义 INCLUDE_DOSFS_MAIN，这个宏才有意义。

其中 INCLUDE_DOSFS_MAIN 总体上控制是否包含 dosFs 文件系统，在定义 INCLUDE_DOSFS_MAIN 的情况下，dosFsLib 和 dosFsFatLib 将自动被包含，这两个模块定义了 dosFs 文件系统的核心工作模块，以上介绍的其他模块具有子宏进行控制，即这些模块都是可选的。在这些可选的模块中，格式化模块一般需要包含，故需要定义 INCLUDE_DOSFS_FMT 宏。
在所有 dosFs 包含的模块中，由 dosFsLib 完成 dosFs 文件系统中间层与 IO 子系统中间的衔接以及 dosFs 与底层块设备驱动之间的衔接。dosFsFmtLib 用于初次挂载块设备后对其进行格式化，建立文件和目录层次所需的基本元数据。以下我们将从实际块设备使用及底层块设备驱动设计的角度讨论 dosFs 文件系统中间层的初始化以及其与上下文之间的衔接工作。

在包含相关宏定义的情况下，dosFs 文件系统各组成部分组件初始化函数将在 usrRoot 函数被调用，涉及 dosFs 所有相关代码如下所示。

```
#ifdef INCLUDE_DOSFS_MAIN    /* dosFs2 file system initialization */
    //hash 表被用于对硬件缓冲区进行管理。
    hashLibInit ();              /* initialize hash table package */

    /* First initialize the main dosFs module */
    dosFsLibInit( 0 ); //dosFs 注册函数

    /* Initialize sub-modules */

    /* ensure that at least one directory handler is defined */
#   if ((!defined INCLUDE_DOSFS_DIR_VFAT) && \
        (!defined INCLUDE_DOSFS_DIR_FIXED))
#       define INCLUDE_DOSFS_DIR_VFAT
#   endif
```

```
        /* init VFAT (MS long file names) module */
#       ifdef INCLUDE_DOSFS_DIR_VFAT
        /* Sub-module: VFAT Directory Handler */

        dosVDirLibInit();

#       endif /* INCLUDE_DOSFS_DIR_VFAT */

        /* init strict 8.3 and vxLongNames handler */

#       ifdef INCLUDE_DOSFS_DIR_FIXED
        /* Sub-module: Vintage 8.3 and VxLong Directory Handler */

        dosDirOldLibInit();

#       endif /* INCLUDE_DOSFS_DIR_FIXED */

        /* Sub-module: FAT12/FAT16/FAT32 FAT Handler */
        dosFsFatInit();

#       ifdef INCLUDE_DOSFS_CHKDSK

        /* Sub-module: Consistency check handler */
        dosChkLibInit();

#       endif /* INCLUDE_DOSFS_CHKDSK */

#       ifdef INCLUDE_DOSFS_FMT

        /* Sub-module: Formatter */
        dosFsFmtLibInit();      /* init dosFs scalable formatter */   //格式化模块初始化

#       endif /* INCLUDE_DOSFS_FMT */

#endif /* INCLUDE_DOSFS_MAIN */
```

此处我们只关注其中的两个函数：dosFsLibInit， dosFsFmtLibInit，这两个函数完成的工作与底层块设备驱动直接相关。其中 dosFsLibInit 完成 dosFs 文件系统与上层 IO 子系统层之间的衔接工作，而 dosFsFmtLibInit 则完成 dosFs 文件系统格式化组件的初始化工作。

从本章开始部分给出的块设备内核驱动层次（如前文图 1 所示）可见，dosFs 文件系统对上由 IO 子系统进行管理，对下管理底层块设备驱动，为了完成与上下层之间的衔接工作，其（1）首先必须通过调用 iosDrvInstall 函数向 IO 子系统注册 dosFs 中间层驱动函数；（2）从底层块设备驱动获取 BLK_DEV 结构。

dosFs 文件系统中间层与上层 IO 子系统层之间的衔接由 dosFsLibInit 函数完成，该函数调用原型如下。

```
STATUS dosFsLibInit
    (
    int    ignored
    );
```

参数暂时没有被使用，故 usrRoot 函数中对该函数进行调用简单的传递 0 作为参数。

dosFsLibInit 函数底层实现使用 iosDrvInstall 函数将 dosFs 中间层驱动函数注册到系统驱动表，完成与上层 IO 子系统的衔接。其注册的驱动函数如下图 8-5 所示。

creat	delete	open	close	read	write	ioctl
dosFsCreate	dosFsDelete	dosFsOpen	dosFsClose	dosFsRead	dosFsWrite	dosFsIoctl

图 8- 5 dosFs 向 IO 子系统注册驱动函数集合

可以看到，dosFs 文件系统提供了所有函数的底层实现，包括 creat，delete。creat 函数用于创建一个新的文件或目录，而 delete 则用于删除一个文件或者目录。因为 dosFs 将块设备封装成具有文件和目录层次性的视图，所以 creat 和 delete 实际上只是对存储在块设备中的一部分数据的操作，并非对块设备本身的操作，所以这两个函数实现才有意义。对于 rawFs 下的块设备以及普通字符设备，creat 和 delete 没有实现的意义，故此时 creat 直接实现为 open，而 delete 则为空。

作为中间层，dosFs 文件系统将拦截所有对底层块设备的操作，当用户对一个基于 dosFs 文件系统的块设备发出操作请求时，IO 子系统将首先根据系统驱动表寻址到此处 dosFs 注册的函数，调用对应的响应函数如打开请求操作对应的 dosFsOpen，由这些函数进行响应。当然作为中间层，只是起着过渡的作用，最终要读取或者写入数据，还必须调用底层块设备驱动函数。那么 dosFs 文件系统必须首先建立起与底层块设备驱动之间的衔接，这个衔接的工作也是由 dosFsLib 库完成，该库除了定义有 dosFsLibInit 函数外，以上注册的 dosFsxxx 函数都是定义在该库中，另外还有一个关键的函数 dosFsDevCreate。dosFsDevCreate 的作用类似于 rawFsDevInit 函数，我们首先从 dosFsDevCreate 调用原型出发看其实现的功能。

```
STATUS dosFsDevCreate
    (
    char *      pDevName,     /* device name */
    CBIO_DEV_ID     cbio,       /* CBIO or cast blkIo device */
    u_int       maxFiles,/* max no. of simultaneously open files */
    u_int       autoChkLevel /* automate volume integrity */
                        /* check level via mounting */
                        /* 0 - default: */
                        /* DOS_CHK_REPAIR | DOS_CHK_VERB_1 */
    );
```

参数 1：块设备节点名称。类似 rawFsDevInit 函数，可以确知 dosFsDevCreate 函数将完成块设备节点的创建并将其添加到系统设备列表中的工作。

参数 2：CBIO 结构指针，注意 CBIO_DEV_ID 实际上是一个 cbioDev 结构指针。虽然此处 dosFsDevCreate 函数要求第二个参数是一个 CBIO 结构指针，但是实际上我们在调用该函数

时，总是传递一个 BLK_DEV 结构的指针，dosFsDevCreate 内部将完成将 BLK_DEV 结构封装成 CBIO 结构的工作。从上文对 rawFs 的讨论中，我们得知，在文件系统层与底层块设备驱动之间实际上还存在一个 CBIO 中间层，这个 CBIO 中间层由文件系统层内部使用，并不对外可见，底层块设备驱动完全可以忽略 CBIO 中间层的存在而不会造成任何问题。从内核提供的接口函数来看，CBIO 层也并不对底层块设备驱动开放接口函数。

参数 3：dosFs 文件系统中同时使用的最多文件句柄数，当该参数为 0 时，dosFs 将使用其默认值 20。

参数 4：文件系统一致性检查时的处理级别，不同的级别将完成不同程度的工作：如仅仅查询错误或者尽量更正错误。可使用的一致性检查级别定义在 h/dosFsLib.h 头文件中，如下所示。通常使用 DOS_CHK_REPAIR。

```
#define DOS_CHK_ONLY    1
#define DOS_CHK_REPAIR 2
#define DOS_CHK_VERB_0       (0xff<<8)
#define DOS_CHK_VERB_SILENT    DOS_CHK_VERB_0
#define DOS_CHK_VERB_1       (1<<8)
#define DOS_CHK_VERB_2       (2<<8)
```

我们总结 dosFsDevCreate 函数实现功能如下：（1）完成 dosFs 文件系统与底层块设备驱动之间的衔接工作，具体上是通过传递一个由底层驱动初始化的 BLK_DEV 结构给 dosFs 中间层完成的，正如前文对 rawFs 的讨论，BLK_DEV 结构包含着底层块设备驱动的关键信息（如块设备读写函数地址），该结构将在下一节进行详细讨论；（2）完成块设备节点的创建和向系统设备列表注册的工作，这个工作完成后，底层块设备就对用户可见了，用户可以使用标准接口函数对块设备进行操作，如创建一个新的文件，写入文件数据等等；（3）对 dosFs 文件系统本身的一些参数进行初始化，如最大文件句柄数，文件系统一致性检查级别。

dosFsDevCreate 函数调用完成后，应用层程序就可以对底层块设备进行操作了。但是在操作方式上不同于字符设备和基于 rawFs 的块设备，字符设备和基于 rawFs 的块设备是将整个设备作为一个单一的个体被读写和控制，而基于 dosFs 的块设备将以文件和目录层次性的视图提供给用户层，所以用户只是对块设备中一个区域进行操作，从上层来看，就是对某个文件或者目录的操作。此时用户程序不再是打开整个设备，而是打开存储在底层块设备上的某个文件或者目录。

dosFsDevCreate 函数中创建的块设备节点表示的是整个块设备，如“/dosFsBlk”，以一个用户 open 函数调用为例，如下代码所示。

```
int fd;
fd=open(“/dosFsBlk/greet.txt”, O_RDWR, 0);
```

与 rawFs 比较，rawFsDevInit 函数中创建的块设备节点表示的也是整个块设备，如“/rawFsBlk”，也是以用户 open 函数调用为例，如下代码所示。

```
int fd;
fd=open(“/rawFsBlk”, O_RDWD, 0);
```

此处可以看出不同之处，对于基于 dosFs 文件系统的块设备的打开操作，在块设备节点名称之后一般都有一个子字符串，表示某个文件名或者目录名，底层上表示在块设备之上的某个区域，用户只能对这个区域进行读写，不可以超出这个区域，这个区域对应应用层视图中的

某个文件或者目录在块设备中的数据占用空间，而对基于 rawFs 文件系统的块设备打开操作，文件路径名总是块设备节点名，即打开整个块设备区域，此时块设备上所有的区域都可读写，对于 rawFs，不需要在块设备节点名之后添加子字符串，而且也没有任何意义，rawFs 文件系统将忽略子字符串。

那么是否可以使用如下代码打开一个基于 dosFs 文件的块设备？

```
fd=open("/dosFsBlk", O_RDWR, 0);
```

从内核代码实现来看，这并不会造成什么问题，但是用户必须知道文件系统在块设备内的数据组织，对于一个普通用户而言，这一般是不大可能的，此时用户一旦进行一个误写入操作，将会破坏文件系统的完整性，其严重性可能就是所有数据的丢失。而且这也不是 dosFs 文件系统在应用层正常的使用方式，如果需要将整个块设备作为一个文件进行访问，那么还是使用 rawFs 文件系统比较好，rawFs 文件系统由于只需提供基本的功能，故内部做了很多的优化，对块设备具有更高的访问效率。

以上对基于 dosFs 文件系统的整个块设备的打开操作被 dosFs 文件系统本身使用，用以检查 dosFs 文件系统的一致性。

对于一个初始被使用的块设备，其内部并未建立文件系统所需的元数据，这些元数据将用以表示整个硬盘资源信息，从而便于 dosFs 文件系统对底层硬盘块设备进行管理。这个创建文件系统元数据的过程我们称之为格式化。dosFs 文件系统下专门有 dosFsFmtLib 模块提供格式化功能。具体完成格式化的函数是 dosFsVolFormat，该函数调用原型如下。

```
STATUS dosFsVolFormat
    (
    void *      device,         /* device name or volume or CBIO pointer */
    int         opt,        /* bit-wise or'ed options */
    FUNCPTR     pPromptFunc  /* interactive parameter change callback */
    );
```

参数 1：指定被格式化的快设备节点名。

参数 2：格式化选项。

参数 3：格式化参数获取函数，当格式化过程中 dosFs 文件系统需要征询一些参数值时，将调用该参数指向的函数，如果为 NULL，则将使用 dosFs 默认参数。通常该参数都是 NULL。

对于 dosFsVolFormat 可使用的选项都定义在 h/dosFsLib.h 头文件中，如下所示。

```
#define DOS_OPT_DEFAULT             0x0000    /* format with default options */
#define DOS_OPT_PRESERVE            0x0001    /* preserve boot block if possible */
#define DOS_OPT_BLANK               0x0002    /* create a clean boot block */
#define DOS_OPT_QUIET               0x0004    /* dont produce any output */
#define DOS_OPT_FAT16               0x0010    /* format with FAT16 file system */
#define DOS_OPT_FAT32               0x0020    /* format with FAT32 file system */
#define DOS_OPT_VXLONGNAMES         0x0100    /* format with VxLong file names */
```

对于小于 2GB 的块设备，dosFs 默认使用 FAT16 进行格式化，大于等于 2GB 的块设备默认使用 FAT32 进行格式化。用户可以强制指定使用何种分配表进行格式化，而不论实际块设

备的容量大小。
以上选项中还有专门针对如何处理引导块的选项，这些选项有：DOS_OPT_PRESERVE，DOS_OPT_BLANK。其中 DOS_OPT_BLANK 表示创建一个空的引导块，而 DOS_OPT_PRESERVE 保留硬盘上原有的引导块内容。所有的硬盘设备第 1，2 个扇区都被预留做引导块（一个引导块一般占用两个扇区），这个引导块一般存储操作系统启动最初的一小段代码以及分区表信息。对于不作为引导设备的块设备而言，也要预留引导块的空间。引导块之后才是文件系统本身的信息以及用户数据。如下图 8-6 所示。

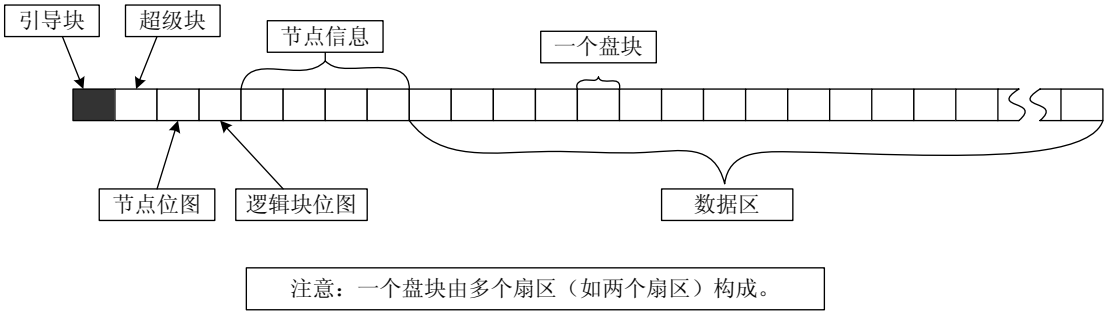


图 8-6 硬盘块设备空间使用

引导块之后的超级块，节点位图，逻辑块位图，节点信息都是文件系统本身的信息。这是管理块设备，维护块设备文件和目录结构视图的基本文件系统元数据。虽然不同文件系统底层管理用数据结构定义有别，但是基本原理都是一致的：建立一种机制对块设备空间进行有序的管理，为应用层提供文件和目录结构视图。具体文件系统基本实现代码，读者可参考开源操作系统 Linux 中的相关代码实现。

使用 dosFsVolFormat 完成块设备的初始格式化后，此后就可以在该设备上进行文件和目录创建，如同普通 PC 下对文件和目录的操作。
注意：dosFsVolFormat 一般只调用一次，在设备最初使用时进行格式化，格式化过程即重构块设备空间，所以块设备中已有用户数据将全部丢失，这一点用户在进行块设备格式化时必须明白。

我们总结一下使得块设备可用的操作流程。
A. 在 Vxworks 启动过程中，完成 dosFs 文件系统内核组件的初始化，完成 dosFs 与 IO 子系统之间的衔接。
B. 调用底层块设备驱动函数 xxxDevCreate 初始化一个 BLK_DEV 结构。
C. 调用 dosFsDevCreate 函数创建设备并将其添加到系统设备列表中，使得用户可以访问设备；使用底层驱动初始化的 BLK_DEV 结构完成 dosFs 与底层块设备驱动之间的衔接。
D. 对于初次使用的块设备，调用 dosFsVolFormat 函数进行格式化，在块设备上创建 dosFs 文件系统基本元数据。

相关代码如下：

```
void creatDosFsBlkDev(){
    BLK_DEV *pBlkDev;

    //调用底层块设备驱动函数初始化 BLK_DEV 结构。
```

```
    pBlkDev=xxxDevCreate(…);

    //调用 dosFsDevCreate 创建块设备节点并添加到系统能够设备列表，
    //完成 dosFs 与底层块设备驱动之间的衔接。
    dosFsDevCreate(“/dosFsBlk”, pBlkDev, 0, DOS_CHK_REPAIR);

    //调用 dosFsVolFormat 对初次使用的块设备进行格式化，使用默认选项和默认参数。
    //注意：如果设备已经进行过格式化，则无需调用 dosFsVolFormat 函数。
    //dosFsVolFormat 将使得原有的用户数据全部丢失，这一点在使用该函数时必须明白。
    dosFsVolFormat(“/dosFsBlk”, 0, NULL);

}
```

在完成对 createDosFsBlkDev 函数的调用后，块设备就对用户可用，如下代码所示为使用以上代码创建的块设备，我们创建一个新的文件，并写入一些数据，而后读取出来进行对比显示。

```
int main(void){
    int fd;
    char rdback[256];
    char *msg=”some messge, but who cares.”;
    int msglen=strlen(msg);

    fd=open(“/dosFsBlk/test”, O_CREAT|O_RDWR, 0);
    if(fd<0){
        printErr(“Cannot create file on device.\n”);
        return -1;
    }

    if(write(fd, msg, msglen)!=msglen){
        printErr(“cannot write designated bytes to device.\n”);
        close(fd);
        return -1;
    }
    close(fd);

    if((fd=open(“/dosFdBlk/test”, O_RDONLY, 0)<0){
        printErr(“Cannot open file on device.\n”);
        return -1;
    }

    if(read(fd, rdback, msglen)!=msglen){
        printErr(“Cannot read designated bytes from device.\n”);
        close(fd);
```

```
        return -1;
    }
    close(fd);

    printf("write msg:%s\n, read back:%s\n", msg, rdback);
    return 0;
}
```

诚如前一节中对 rawFs 文件系统的论述，以上代码中我们只是使用了 CBIO 基本功能层，没有使用 CBIO 数据缓冲层以及 CBIO 分区管理层，对于块设备而言，CBIO 数据缓冲层一般是必须使用的，而 CBIO 分区管理层在底层块设备被分为多个分区使用时采用。值得注意的是，CBIO 分区管理层在块设备只能创建主分区（即最多支持 4 个分区），不支持扩展分区的创建，而且由 CBIO 分区管理层创建的分区只能使用在 Vxworks 下，与其他操作系统不兼容，即使是使用 dosFs 文件系统。所以要使得 Vxworks 块设备文件 dosFs 系统系统能够被 Dos 系统访问，则不要使用 CBIO 分区管理层，即整个块设备只作为一个分区进行使用。对于 CBIO 分区管理层的详细情况，我们将在下文中块设备驱动具体实现一节中进行介绍。

如下所示为使用 CBIO 数据缓冲层的块设备创建代码。此时对应的块设备内核驱动层次如上文图 4 所示。

```
void creatDosFsBlkDev(){
    BLK_DEV *pBlkDev;
    //typedef struct cbioDev *CBIO_DEV_ID;
    //cbioDev 结构定义在 h/private/cbioLibP.h 内核头文件中。
    CBIO_DEV_ID pCbioDev;

    pBlkDev=xxxDevCreate(…);
    pCbioDev=dcacheDevCreate(pBlkDev, NULL, 128*1024, "ata hard disk cache");
    dosFsDevCreate("/dosFsBlk", pCbioDev, 0, DOS_CHK_REPAIR);

    //调用 dosFsVolFormat 对初次使用的块设备进行格式化，使用默认选项和默认参数。
    //注意：如果设备已经进行过格式化，则无需调用 dosFsVolFormat 函数。
    //dosFsVolFormat 将使得原有的用户数据全部丢失，这一点在使用该函数时必须明白。
    dosFsVolFormat("/dosFsBlk", 0, NULL);

}
```

从此处介绍的使得块设备对用户可用的初始化过程以及前一小节中 rawFs 文件系统下使得块设备可用的初始化过程可以看出，实际上这个过程都相当的简单，只需调用一两个文件系统接口函数，即可使得块设备被正常的使用。其中文件系统本身完成了大量的工作。但是无论文件系统做了多少过渡性和管理型工作，最终将数据写入块设备或者从块设备中读取用户数据或者文件系统本身的元数据，都需要底层块设备驱动协助。在前文对 rawFs 以及 dosFs 的介绍中，无论是 rawFsDevInit 函数还是 dosFsDevCreate 函数，都需要一个经过底层块设备驱动初始化后的 BLK_DEV 结构完成文件系统层与底层块设备驱动之间的衔接工作，使得文件系统层可以调用底层块设备驱动的相关函数读写，控制块设备。所以 BLK_DEV 结

构在块设备驱动层次中起着至关重要的作用，下一节我们就专门对 BLK_DEV 结构进行介绍。

8.3 BLK_DEV 结构

在介绍 BLK_DEV 结构具体定义之前，我们先试着分析从实际使用的角度，该结构应包含哪些信息。BLK_DEV 结构是文件系统层与底层块设备驱动层之间信息交互的唯一“媒介“，即 BLK_DEV 结构必须包含文件系统层需要底层块设备驱动提供的所有信息。从块设备实际使用的角度来看，这些信息必然包含：（1）块设备数据写入函数；（2）块设备数据读取函数；（3）块设备控制函数（如复位块设备，重新定位磁头位置）；（4）块设备本身关键参数信息，如总扇区数（或总块数），磁道数，磁头数，每磁道扇区数，对于非硬盘块设备，还需提供每个块的字节数（对于硬盘块设备，一个块通常被称为扇区，固定为 512 字节）。

BLK_DEV 结构定义如下，该结构定义在 h/blkIo.h 头文件中。

```
typedef struct      /* BLK_DEV */
{
    FUNCPTR         bd_blkRd;           /* function to read blocks */
    FUNCPTR         bd_blkWrt;          /* function to write blocks */
    FUNCPTR         bd_ioctl;       /* function to ioctl device */
    FUNCPTR         bd_reset;       /* function to reset device */
    FUNCPTR         bd_statusChk;       /* function to check status */
    BOOL            bd_removable;       /* removable medium flag */
    ULONG       bd_nBlocks;         /* number of blocks on device */
    ULONG       bd_bytesPerBlk;         /* number of bytes per block */
    ULONG       bd_blksPerTrack;    /* number of blocks per track */
    ULONG       bd_nHeads;          /* number of heads */
    int         bd_retry;       /* retry count for I/O errors */
    int         bd_mode;        /* O_RDONLY |O_WRONLY| O_RDWR */
    BOOL        bd_readyChanged;  /* dev ready status changed */
} BLK_DEV;
```

bd_blkRd：
块设备数据块读取函数。注意块设备每次只能以一个块的方式进行读取或者写入，不支持单个字节的读写方式。文件系统中间层将完成文件或者目录向块设备中对应块号的转换工作，当调用 bd_blkRd 指向的函数时，传递给它是文件或目录对应的初始化块号，需要读取的数据块数，以及数据缓冲区基地址。底层块设备读取函数必须操作块设备控制器读取指定起始块开始的数据块数，将这些数据从设备控制器内部缓冲区复制到文件系统层提供的缓冲区中，完成块设备数据读取工作。

bd_blkWrt：
块设备数据块写入函数。如同 bd_blkRd 函数，文件系统层在调用 bd_blkWrt 指向的函数之前已经完成文件或者目录向对应块号的转换工作，所以传递给块设备写入函数的参数将是文

件或目录对应的起始块号，需要写入的数据块数，被写入数据的缓冲区基地址。底层块设备写入函数需要复制文件系统传入的数据缓冲区中数据到块设备控制器内部缓冲区中，操作块设备控制器完成数据块的写入工作。

bd_ioctl：
块设备控制函数。文件系统提供了一系列标准控制选项对块设备进行控制。底层块设备可根据需要对这些控制选项进行响应。所有可用控制选项都定义在 h/ioLib.h 头文件中，部分选项如下图 8-7 所示。

选项	说明
FIODISKFORMAT	块设备格式化
FIODISKCHANGE	设置设备更换标识
FIODISKINIT	初始化块设备目录
FIOUNMOUNT	卸载块设备
FIOSEEK	设置文件偏移指针
FIORENAME	重命名文件或目录
FIOGETNAME	由文件句柄获取文件名称
FIOWHERE	获取当前文件偏移指针
FIOFLUSH	清空缓冲区中数据
FIOSYNC	同步数据，即将缓冲区中数据写入设备
FIONREAD	缓冲区中可读取字节数据
FIOFSTATGET	获取文件状态信息（如文件创建，修改时间等）

图 8- 7 部分可用选项列表

实际上，从图 8-7 中大部分选项的含义可以看出，很多选项的响应根本不需要底层块设备驱动进行，而只能由文件系统本身完成，而很多选项的最终效果到达底层块设备驱动时都是相同的操作，如块设备格式化，初始化设备目录，创建目录，同步数据都是需要向块设备写入数据，而设置文件偏移指针，获取当前文件偏移指针，由文件句柄获取文件名则完全由文件系统本身完成，这些请求根本不会到达底层块设备驱动。

bd_reset：
复位设备函数。该函数用于在发生设备读写错误时，复位设备重新开始操作。底层驱动对应实现函数必须完成设备复位以及使设备重新就绪的工作。

bd_statusChk：
查询设备相关状态函数。主要是查询块设备控制器是否已经完成上一个命令的执行，当前是否可以继续接收命令。

bd_removable：
该变量表示设备是否是可移动的，即指定块设备是一个硬盘设备还是一个软盘设备。对于硬盘设备由于无需每次写入前检查设备是否存在，故可以极大的提供效率。

bd_nBlocks：底层块设备总块数
bd_bytesPerBlk：每块字节数

bd_blksPerTrack：每磁道块数
bd_nHeads：磁头数
以上这四个字段表示了底层块设备本身的容量信息。对于磁盘块设备而言，每块字节数（bd_bytesPerBlk）即一个扇区的大小，为 512 字节。bd_nBlocks 与 bd_bytesPerBlk 的乘积表示了块设备的总容量大小。

bd_retry：
该字段是底层驱动提供给文件系统层一个错误发生时的尝试次数。并不是每次读写块设备都可以成功，当失败时，我们需要重新发起操作，如果接连失败，那么达到一定次数后，我们就放弃，这个一定的次数就是由 bd_retry 字段指定。

bd_mode：
底层块设备的操作模式。共有三种可用操作模式：只读，只写，读写。一般而言，只写模式较少使用，因为要写入数据，必须先读取数据。只读模式如对于 CD-ROM 块设备，其数据的特殊存储方式就不支持写，而只能从中读取已有的数据。读写模式是通常块设备常用的模式，如硬盘设备一般都是工作在读写模式。
注意：这个字段是对整个块设备工作模式的指定，不同于 open 调用打开一个文件或目录时指定的模式，这个模式只对被打开的文件或者目录有效，而且是一种软件层的模拟，并非底层块设备本身的限制。而 bd_mode 则表示底层块设备由于其特殊的存储数据方式（如 CD-ROM 的打孔方式）物理上的一种模式限制。

bd_readyChanged：
该字段被底层块设备设置用以提醒文件系统层底层块设备的就绪状态已经发生改变。这个字段被用于软盘等可移动的设备。当软件被从驱动器中移除后，底层块设备字段将设置该字段为 TRUE，表示底层块设备已经被移除，此时文件系统层的相关操作将以 DISK_NOT_PRESENT 错误返回。

BLK_DEV 是文件系统层与底层块设备驱动之间信息交互的唯一方式。这个数据结构完全由底层块设备驱动进行初始化，被传递给文件系统层使用，为了完成最终对底层块设备硬件的操作，这个结构包括了文件系统层操作块设备的所有关键信息，如读写函数，就绪状态检测函数，设备复位函数以及块设备容量信息。其中最为关键的就是读写函数。无论文件系统层如何对用户操作进行封装，最终还是需要读取或者写入数据到块设备中，而底层块设备驱动提供的读写函数将完成读写的具体工作。
诚如上文所述，块设备的文件和目录层次结构视图是由文件系统层向上层（用户层）提供的，对于底层块设备驱动而言，其没有这些视图，其视图形式是起始数据块号以及需读写的总数据块数。对于一个硬盘设备而言，物理上一个扇区就是一个块，硬盘设备的总扇区数由 BLK_DEV 结构中的 bd_nBlocks 字段指定。文件系统层将块的定义稍微做了些改变，这是从文件系统层本身数据管理的效率上考虑的，我们将文件系统层使用的块称为逻辑块，而将底层块设备本身的固定数据块大小（如硬盘的扇区）称为物理块。一般而言，多个物理块构成一个逻辑块，通常物理块数是逻辑块数的 2 的幂次方，如 2 个或 4 个或 8 个物理块构成一个逻辑块。对于硬盘设备而言，物理块大小为 512 字节，而逻辑块大小一般设置为两个物理块的大小，即为 1024 字节。
当用户对一个打开的文件进行读写操作时，文件系统层将完成文件到底层块设备中物理块号

的映射，所以最终调用底层块设备驱动读写函数时，使用的直接就是文件对应的块设备中起始物理块号以及需读写的物理块数。注意：底层块设备驱动只对物理块进行操作，逻辑块由文件系统自身（以及硬盘缓冲区管理组件 CBIO 中间层）进行使用，不对底层块设备驱动可见，所以底层块设备驱动初始化 BLK_DEV 结构时也是基于物理块的概念，即对于硬盘设备而言，bd_nBlocks 表示的是硬盘总扇区数，而 bd_bytesPerBlk 就是 512（单位：字节），这两个字段的乘积即表示整个硬盘设备的容量（单位：字节）。
bd_blksPerTrack 以及 bd_nHeads 这两个字段主要由底层块设备本身进行使用，用以在操作硬盘控制器时将起始块号转换成磁头号，磁道号，以及磁道中块号，硬盘控制器需要这些参数定位磁头。从这一点来看，实际上文件系统在调用底层块设备读写函数时传入的起始块号是一个绝对的块号，即将整个块设备的块从 0 开始计数时的块号，对于硬盘控制器而言，其则使用磁头号，磁道号以及磁道内扇区号进行操作，所以底层驱动必须将绝对块号转换成硬盘控制器使用的磁头号，磁道号以及磁道内扇区号，此时 bd_blksPerTrack，bd_nHeads 就起着关键作用。
我们令绝对块号为 block（以 0 开始计数），那么用 block 除以 bd_blksPerTrack 得到的余数就是磁道内块号；然后我们再用商除以 bd_nHeads，此时余数就是磁头号，商就是磁道号。注意：硬盘设备是以磁道构成的柱面为单位进行数据读写的，而不是以盘片为单位进行数据读写的。另外我们还需要将以上计算后得到的磁道内块号做加 1 处理，因为绝对块号是以 0 为基数的，即第一个块记为 0，而磁道内块号是以 1 为基数的，即第一个块记为 1.在得到磁头号，磁道号，磁道内块号后，我们就可以使用这些参数配置硬盘控制器的相关寄存器，发起硬盘块设备的读写操作。

作为底层驱动设计的惯例，我们需要一个驱动自定义数据结构，用以保存一些驱动相关信息。同时为了应对内核接口要求，通常我们将内核结构作为驱动自定义接口的第一个成员。这个成员由底层驱动初始化后提供给上层内核接口层，而之后内核接口层对底层驱动的调用也总是使用这个结构，所有我们将驱动信息与内核结构作为一个整体进行定义，可以同时访问到自定义信息。当然也可以单独定义一个驱动结构，与内核结构分离，作为全部变量在底层驱动中使用，但是这将增加很多限制，远不如作为一个整体时来的灵活，而且对于驱动多个相同设备时，也不如将二者作为一个结构简便。

对于块设备驱动而言，BLK_DEV 已然提供了很多关键信息，当然还是有一些信息必须由底层驱动自己保存，如硬盘控制器的配置寄存器基地址（虽然已众所周知），硬盘控制器中断号。另外为了表示某次中断是针对写操作还是读操作还是复位操作或者是磁头重定位操作，我们还需要一个变量指定当前是哪种操作类型（读，写，复位，磁头重定位）。为了提供更大的灵活性，我们使用函数指针替代这个变量，如此在硬盘控制器中断总入口函数中，我们将直接调用这个函数指针指向的函数，而不用通过检查变量的值作出决定。此时在发起某个操作前，只需对这个函数指针重新初始化指向对应响应函数即可。如本次将发起一个读硬盘操作，那么首先将函数指针初始化为指向一个读响应函数（如 read_intr），而后配置硬盘控制器进行读操作，如此硬盘控制器完成读操作并发出中断后，首先进入底层驱动中断总入口函数，而后该函数直接调用驱动自定义结构中函数指针指向的函数（即 read_intr），完成读操作中断的具体响应。

注意：从下一节的讨论中，我们将看到 Vxworks 下块设备驱动一般不使用中断工作方式，所以下面定义的这个函数指针以及中断号实际上都不会被使用。

综上所述，驱动自定义结构如下。

```
typedef struct  _blk_dev_struct{
    BLK_DEV     blkDev; //必须是第一个成员。
    UINT32      regBase; //硬盘控制器寄存器基地址，是否必要还值得考虑。
    FUNCPTR     intHandler; //硬盘控制器中断具体响应函数指针。
    UINT8       intLevel; //硬盘控制器中断号。

}ATA_BLK_DEV;
```

驱动自定义结构 ATA_BLK_DEV 中，最后两个成员是 drive 和 partition，分别表示驱动器号以及分区号。通常一个块设备对应驱动器，而一个块设备可以分为多个分区，可在不同分区中使用不同的文件系统。

前文中说到要使得块设备可用，底层驱动必须初始化一个 BLK_DEV 结构，传递给文件系统层相关接口函数（如 rawFsDevInit，dosFsDevCreate），事实上底层驱动 BLK_DEV 结构初始化函数并不单是对 BLK_DEV 结构进行初始化，其完成底层驱动所需的所有初始化工作，只当所有初始化工作都成功完成后，其才返回一个 BLK_DEV 结构，表示底层驱动已处于可用状态。因为文件系统接口函数（rawFsDevInit，dosFsDevCreate）一旦完成设备创建，可能尚未返回时，块设备就已被使用，即底层块设备驱动就已经要工作了，所以底层驱动初始化函数必须完成自身以及块设备控制器的所有初始化工作，使得块设备驱动本身以及块设备硬件处于正常工作状态后，才能返回一个 BLK_DEV 结构。我们将底层块设备中完成初始化的这个函数通常命名为 xxxDevCreate 或 xxxBlkDevCreate。在下文中具体块设备驱动介绍中，我们将使用 ataBlkDevCreate 函数名。

8.4 块设备驱动结构

块设备驱动工作模式可分为两种：（1）阻塞模式，即轮询工作方式；（2）非阻塞模式，即中断工作方式。通常而言，对于像块设备这样读写时间较长的设备，一般都采用非阻塞工作模式，也即中断工作方式。但是中断工作方式下底层驱动与内核耦合较紧密。一般通用操作系统如 Linux 块设备驱动作为操作系统固定组成的一部分，所以采用中断工作方式提高系统整体性能。但是嵌入式 Vxworks 操作系统由于其特殊的工作环境，其必须提供块设备驱动接口，即必须尽量去除操作系统本身以及底层块设备驱动之间的耦合性，这就造成了 Vxworks 下块设备的工作方式必须是阻塞模式，即轮询方式。

我们知道，块设备驱动直接驱动的是块设备控制器，发出一个读命令后，块设备控制器需要较长的时间才能将数据准备好，中断方式下，我们可以在发出读命令后立刻返回，而后在中断处理程序中将数据拷贝给内核，此种方式正如刚才所述，底层驱动必须能够访问内核组件的大量资源（不仅仅是一个内核缓冲区），造成底层块设备驱动与内核的紧密联系，这就要求底层块设备驱动程序员必须对操作系统本身十分了解。而这对于 Vxworks 而言是不太可能的，Vxworks 主要使用在嵌入式平台上，所以 Vxworks 主要提供了一个最小内核，外设驱动都作为 BSP 的一部分，这就要求 Vxworks 最小内核与底层驱动（包括块设备驱动）之间的接口关系必须是非紧密耦合的，即脱离的，所以 Vxworks 块设备读写函数实现将采用阻塞实现方式，即向块设备控制器发出读写请求后，必须阻塞等待，直到块设备控制器将数据准备好或者将数据写入设备中才可返回，Vxworks 上层（文件系统层）依赖于底层块设备

驱动的这一点，所以在实现底层块设备读写函数时必须十分注意，不可借鉴其他通常操作系统（如 Linux）的中断工作方式。

基于 CBIO 中间层的块设备驱动内核结构层次如下所述。
CBIO 中间层对每个块设备都在内存中分配一块专门的缓冲区进行数据缓冲。以一个块设备数据读取操作为例，文件系统层调用 CBIO 中间层函数发出块设备数据读取请求，CBIO 中间层将首先从内存缓冲区中进行查找，如果找到对应块，则直接从内存缓冲区中拷贝数据给文件系统层，如果没有找到，则创建一个新的块，而后调用底层驱动读设备函数，将数据从块设备中读取到这个新创建的块中，而后从这个块中将数据拷贝给文件系统层。在调用底层驱动读设备函数期间，整个系统都阻塞在这个读设备函数中，直到该函数返回。所以在初次读取一个块时，将存在较大的延迟，此后由于内存缓冲区已经保留了数据，所以此后的读取操作都不需要经过底层块设备读函数，效率将大大提高。
对于用户写入操作，同样文件系统层也通过 CBIO 中间层进行操作，CBIO 中间层也是从块设备内存缓冲区查找对应块，如果找到对应块，则直接将数据从文件系统层拷贝到这个内存缓冲区的对应块中，如果没有找到，则首先创建一个新的缓冲区，而后发起一个读设备操作，将对应块中原有数据从块设备中读到内存缓冲区，而后根据文件系统传入的数据对这个内存缓冲区的块进行修改。
对于内存缓冲区中所有被改写的块（操作系统使用一个 dirty 位表示块是否被改写），Vxworks 专门创建 tDcacheUpd 后台任务完成这些块的刷新，即调用底层块设备写函数将改写后数据写入块设备中，同样写函数的调用也是阻塞的。
这种读写函数阻塞实现方式虽然不如非阻塞方式效率高，但是接口简单，也符合 Vxworks 使用的嵌入式环境。最关键的是将 BSP 开发人员与操作系统开发人员完全隔离，编写块设备驱动的程序员无需对内核实现细节完全了解即可进行块设备驱动的开发，而这是 Vxworks 这个非开源操作系统必须要提供的。

注意：BSP 开发人员完全可以自己在底层驱动读写函数实现中挂起当前任务，使用中断工作方式驱动块设备，但这并非 Vxworks 下提供的块设备驱动开发机制的初衷，而且这种工作方式对整个系统的性能究竟有何种影响需要经过仔细的测试。笔者下文中将只对阻塞模式进行讨论，对于自实现的非阻塞模式的具体实现，感兴趣读者可与笔者私下讨论。

基于阻塞实现方式下的块设备驱动结构要比非阻塞模式简单的多。下面我们从 BLK_DEV 结构以及块设备操作需求出发介绍 Vxworks 下块设备驱动的基本结构。

（1） 初始化函数
按照 Vxworks 下驱动开发惯例，我们将初始化函数分为两个部分实现：xxxDrv 完成设备初始化；xxxDevCreate 函数完成上层结构初始化以及相关资源分配工作。此处对于我们的块设备驱动，定义如下两个函数：ataBlkDrv 完成块设备控制器初始化工作；ataBlkDevCreate 完成 BLK_DEV 结构初始化以及所有其他准备工作。当 ataBlkDevCreate 函数返回后，底层块设备驱动以及块设备本身将处于正常工作状态，等待上层设备操作请求。这两个函数只需被调用一次。

（2） 设备读写函数
设备读写函数是块设备驱动中最为关键的函数，也是被上层调用最多的函数。这两个函数的地址将被用以初始化 BLK_DEV 结构中的 bd_blkRd，bd_blkWrt 字段，从而传递给文件系统

层用以在之后的操作中访问块设备。对于下文介绍的块设备驱动，读函数定义为 ataBlkRd，写函数定义为 ataBlkWrt。

（3） 设备控制，状态查询函数

设备控制主要包括复位设备，配置设备，对应 BLK_DEV 结构，主要实现为两个函数：reset 函数以及 ioctl 函数，分别用以初始化 BLK_DEV 结构中的 bd_reset，bd_ioctl 字段。设备查询函数用以初始化 BLK_DEV 结构中 bd_statusChk 字段。该函数被文件系统层使用用以查询设备的就绪状态，是否可接收下一个请求，该函数完成的功能如同 BLK_DEV 结构中的 bd_readyChanged 字段，事实上，文件系统层同时使用 bd_statusChk 函数以及 bd_readyChanged 字段确保底层块设备可以对下一个请求进行响应。下文中将设备复位函数定义为 ataBlkReset，控制函数定义为 ataBlkIoctl，将状态查询函数定义为 ataBlkStatusChk。

事实上，设备状态查询函数主要使用于软盘这一类可移动设备，每次文件系统层对软件设备进行操作时，都必须调用设备状态检测函数查看软盘设备是否还在驱动器中，如果已被移除，则放弃所有操作，设置 DISK_NOT_PRESENT 错误。对于硬盘设备而言，可以将 BLK_DEV 结构中 bd_statusChk 字段设置为 NULL，表示非可移除设备。

BLK_DEV 结构中的 bd_statusChk 字段与 bd_readyChanged 字段的功能相同，bd_readyChanged 更多的是由底层块设备驱动进行设置，直接提供给文件系统层使用，而 bd_statusChk 则指向一个函数用以被文件系统自身使用“亲自“检查设备状态。在底层块设备实现上，可以将设置 bd_readyChanged 字段的函数直接对 bd_statusChk 字段进行初始化。

（4） 块设备控制器硬件操作函数

这些函数被底层块设备自身使用用以操作设备控制器，如读写控制器相关寄存器，这些代码与具体的设备控制器相关。

8.5 块设备驱动具体实现

上一节中我们总结了块设备驱动的基本构成，本节我们将以实际代码为例介绍各个组成部分的基本实现。注意这些实现并没有给出某些实现细节，因为不同嵌入式平台块设备接口有所不同，读者需要关注这些函数实现的基本思想，在实际应用中，按部就班的进行即可。底层驱动设计主要包括两个方面的内容：与内核接口部分的设计以及硬件配置设计。对于块设备驱动而言，内核接口部分的设计较为简单，相反硬件驱动部分倒是比较复杂，需要配置块设备控制器各个寄存器完成各项操作，这需要对被驱动的块设备（控制器）有比较深入的理解。对于通用 PC 下常用硬盘块设备驱动实例代码，读者可参阅 Linux 实现下的相关源代码。

8.5.1 ATA（IDE）硬盘结构

为了帮助对下文中相关代码进行理解，我们在介绍代码前先对当前比较常用的硬盘设备进行简单的介绍。在前文中我们不加区别的使用硬盘和磁盘两个概念，实际上，磁盘包括硬盘和软盘。软盘在上个世纪 90 年代以及本世纪初使用比较普遍，最常见的就是 1.44MB，3.5 英寸的软盘，随着 USB 技术的发展和普及，软盘现在已经完全被 U 盘取代，当前计算机上也

已经没有软盘驱动器了。所以现在讲到磁盘也可以理解是硬盘。诚如本章前面部分所述，底层硬盘驱动并不直接驱动硬盘块设备，而是驱动硬盘控制器，实际上，以 PC 下普遍使用的 ATA（IDE）硬盘为例，其由三个部分构成：（1）硬盘控制器（电子控制电路部分），通过 IDE 接口（40 针）完成与主机（板）之间的数据通信，即进行数据和命令的收发，其内部集成有数据缓冲区，在硬盘读操作中存储从硬盘读取的数据，再通过 IDE 接口传递给主机，在硬盘写操作中用以存储从主机读取的数据，再驱动硬盘驱动器将数据写入硬盘对应扇区；（2）硬盘驱动器（机械控制部分），完成磁头等机械部分的驱动，具体完成的硬盘的读写；（3）用作物理存储介质的磁盘片，对于硬盘设备而言，将由多个盘片构成。如下图 8-8 所示普通 IDE 硬盘内部结构简图。图 8-9 所示为 IDE 硬盘与主板之间的接口，包括 4 针电源接口以及 40 针通信接口。

图 8- 8 IDE（ATA）硬盘内部结构简图

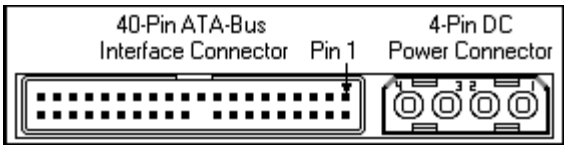


图 8- 9 IDE 硬盘接口

将硬盘控制部分分为控制器和驱动器两个部分，可以使得一个控制器可以控制多个驱动器进行工作，不过通常使用上一个控制器只对应一个驱动器，二者与物理存储盘片做成一个整体，这就是上图 8-8 中常见的硬盘设备。在现在的硬盘设备中，通常都集成了硬盘控制器，硬盘驱动器和物理存储盘片。其中硬盘控制器完全负责与 CPU 之间的通信，所以底层硬盘驱动实际上驱动也就是硬盘控制器，硬盘驱动器将由硬盘控制器进行驱动，根据 CPU 发出的硬盘操作命令驱动硬盘驱动器定位磁头，读写扇区数据。具体的，以一个硬盘读操作为例，在硬盘系统内部，其服务过程将由如下四个动作组成：（1）移动磁头到指定磁道，即寻道；（2）等待指定扇区转动到磁头下，注意硬盘盘片总是处于转动状态的，转动速率是硬盘设备本身一个重要的技术指标，通常转速越快，数据读写速度越快，当然要求的技术越高；（3）读取扇区数据到硬盘内部数据缓冲区；（4）发出中断，通知 CPU 读取硬盘内部数据缓冲区中数据。故一次硬盘读写操作内部所用时间由寻道时间，等待时间（又称延迟时间），传送时间组成。其中寻道时间相对较长，所以大多数操作系统实现对于硬盘读写操作将进行“电梯算

法”的排序工作，尽量减小操作间磁头的移动距离。
除了 IDE（ATA）接口外，硬盘设备常用的接口类型还有 SCSI，光纤，SATA（Serial ATA），此处不再对这些接口类型进行讨论，感兴趣读者可查阅相关资料。

8.5.2 硬盘分区

作为系统启动盘的硬盘设备在整个硬盘设备的第一个分区中存储着关键信息，一个分区只有 512 个字节。通用 PC 机上电执行后，首先执行固化到高端地址处的 ROM 中程序，完成整个系统的自检测试，而后从启动盘设备中读取第一个扇区的代码到系统 RAM 中，并跳转到这段代码处执行。这个扇区的开始部分存储着启动代码，代码之后就是硬盘分区表，这个分区表中最多有四个表项，即表示最多支持四个基本分区，可以使用扩展分区进行分区的扩展。我们通常将这个存放系统最初执行代码以及分区表的扇区称为 MBR（主引导记录：Master Boot Record）。

如果一个硬盘设备不作为系统启动盘，其第一个扇区依然被保留，作为分区表的存放地。对于一个作为多个分区使用的硬盘设备，其支持在每个分区上创建不同的文件系统，大大加强了系统灵活性。现在单个硬盘设备动辄几百 GB，一般都要进行分区，除了四个基本分区，还可以使用多层扩展分区，将一个几百 GB 的硬盘设备分为多个逻辑盘使用。对于分区使用的硬盘，无论是作为操作系统启动盘还是非启动盘，都必须使用主引导扇区作为分区表存储介质，反过来说，分区使用的硬盘设备必须构建分区表内容，并将其写入主引导扇区中。Vxworks 专门提供了这样一个库 usrFdiskPartLib 按照用户要求构建分区表，并将其写入主引导扇区中。
usrFdiskPartLib 提供如下两个接口函数在块设备创建过程中使用。
（1） usrFdiskPartCreate
该函数根据传入的参数创建分区表，并将分区表写入硬盘设备的主引导扇区（即整个硬盘设备的第一个扇区）。该函数调用原型如下。

```
STATUS usrFdiskPartCreate
    (
    CBIO_DEV_ID cDev,   /* device representing the entire disk */
    int         nPart,      /* how many partitions needed, default=1, max=4 */
    int         size1,      /* space percentage for second partition */
    int         size2,      /* space percentage for third partition */
    int         size3       /* space percentage for fourth partition */
    );
```

参数 1（cDev）：这是一个 CBIO_DEV_ID 结构，一般是 dcacheDevCreate 函数返回的值，即封装后的块设备结构。
参数 2（nPart）：表示将硬盘设备分为几个分区，最大支持 4 个分区，Vxworks 不支持扩展分区。
参数 3，4，5：第二，三，四个分区分别所占整个硬盘设备的百分比，余下的就是第一个分区的小。如果只有一个分区，那么这三个参数都是 0；如果分为两个分区，则第三，四个参数为 0，第二个参数为第二个分区的百分比；如果分为三个分区，则第四个参数为 0，第二，三个参数分别为第二，三个分区的百分比。
如下代码调用 usrFdiskPartCreate 将硬盘设备分为两个分区，第二个分区占 50%的硬盘容量。

usrFdiskPartCreate (pCbioCache, 2, 50, 0, 0);

（2） usrFdiskPartRead

这个函数将在调用 CBIO 分区管理层接口函数 dpartDevCreate 时作为参数传入，对 usrFdiskPartCreate 函数创建的分区表进行解析。该函数原型如下。

```
STATUS usrFdiskPartRead
    (
    CBIO_DEV_ID cDev,              /* device from which to read blocks */
    PART_TABLE_ENTRY *pPartTab, /* table where to fill results */
    int nPart                      /* # of entries in <pPartTable> */
    );
```

参数 1：通常是 dcacheDevCreate 的返回值，表示底层硬盘设备。usrFdiskPartRead 将通过该结构中封装的 BLK_DEV 结构，使用底层块设备驱动读写设备中存储的分区表信息。

参数 2：这个结构传入时是一个空结构，将由 usrFdiskPartRead 函数完成初始化，dpartDevCreate 函数调用 usrFdiskPartRead 函数的目的也正是如此。

参数 3：表示设备分区表中共有几个有效分区信息，最多支持 4 个分区。

除了以上两个功能函数外，usrFdiskPartLib 还提供了一个信息显示函数 usrFdiskPartShow，用户在调用 usrFdiskPartCreate 函数创建分区表后，可以通过 usrFdiskPartShow 函数打印分区表信息。usrFdiskPartShow 调用原型如下。

```
STATUS usrFdiskPartShow
    (
    CBIO_DEV_ID cbio
    );
```

参数是表示硬盘设备在 CBIO 层封装结构，实际上直接传入 BLK_DEV 结构即可，因为 usrFdiskPartShow 只是通过该结构使用底层块设备读取主引导扇区信息，解析其中存储的分区表，打印出其信息而已，所以具体是 CBIO 层封装结构还是原始的 BLK_DEV 结构关系不大。

8.5.3 CBIO 分区管理层

usrFdiskPartLib 只是创建了一个分区表，换句说话，其并不提供分区管理功能，为了在每个分区上可以使用不同的文件系统，还必须借助于分区管理组件，这就是 CBIO 分区管理层完成的功能，即 dpartCbio 库。如同 CBIO 数据缓冲层，CBIO 分区管理层也提供了接口函数用于对下层 CBIO 层（即 CBIO 数据缓冲层）进行封装，即 dpartDevCreate，该函数使用 usrFdiskPartLib 库提供的 usrFdiskPartRead 函数解析分区表，在内存构建每个分区的软件表示，对每个分区提供文件系统支持。dpartDevCreate 函数调用原型如下。

```
CBIO_DEV_ID dpartDevCreate
    (
    CBIO_DEV_ID     subDev,         /* lower level CBIO device */
    int             nPart,          /* # of partitions */
    FUNCPTR         pPartDecodeFunc   /* function to decode partition table */
    );
```

参数 1：CBIO 下层块设备结构封装结构，通常是 CBIO 数据缓冲层设备创建函数 dcacheDevCreate 函数的返回值。
参数 2：表示底层块设备共分为几个分区使用。
参数 3：分区表解析函数，即 usrFdiskPartRead。

正如刚才所述，dpartDevCreate 函数使用 usrFdiskPartRead 读取分区表信息，对每个分区创建内核相关结构进行分别管理，对于上层文件系统层而言，每个分区都是一个独立的块设备，文件系统层可以在每个分区上创建不同的文件系统进行块设备的使用。为了获取 dpartDevCreate 创建的每个分区，dpartCbio 提供 dpartPartGet 函数，该函数被文件系统层调用获取其中一个分区，在其上创建文件系统。dpartPartGet 函数调用原型如下。

```
CBIO_DEV_ID dpartPartGet
    (
    CBIO_DEV_ID masterHandle,     /* CBIO handle of the master partition */
    int partNum                   /* partition number from 0 to nPart */
    );
```

参数 1：dpartDevCreate 函数返回值。
参数 2：指定获取第一个分区的分区信息。
注意：dpartPartGet 函数返回一个 CBIO_DEV_ID 结构，表示 CBIO 分区管理层将每个分区都作为一个独立的块设备提供给文件系统层使用。文件系统层将“不知道”其使用的只是整个块设备的一个部分还是整个块设备，当然其也不需要知道。

如此一来，虽然底层只有一个块设备，但是经过 CBIO 分区管理层的封装，从文件系统层的角度而言，就好像有多个块设备可被使用。文件系统层对每个分区内部的操作，将经过 CBIO 分区管理层的转换，转换为对应底层块设备具体扇区的操作。例如分配第二个分区在整个块设备中的起始扇区号为 5000，则对第二个分区第四个扇区的操作，在底层块设备驱动来看，即是对底层块设备硬件的第 5004 个扇区的操作。换句话说，CBIO 分区管理层将每个分区内部的操作经过其维护的分区信息转换成对整个底层块设备的对应区域操作，所以对于底层块设备驱动而言，其将“看不到”分区的划分，对其而言，底层块设备还是一个单一的设备。所以底层块设备驱动根本不需要管理分区信息，其只需将整个硬盘块设备当成一个由许多扇区组成的一维数组即可，数组中每个元素就是一个扇区，上层到达底层块设备驱动的所有读写操作都是以底层块设备绝对扇区号进行的。

如下所示是一个使用了 CBIO 所有三个内部子层次的块设备创建代码，我们首先调用底层块设备驱动初始化 BLK_DEV 结构，而后以这个 BLK_DEV 结构调用 dcacheDevCreate 函数完成 CBIO 基本功能层和 CBIO 数据缓冲层的添加，之后使用 dcacheDevCreate 返回的 CBIO_DEV_ID 结构调用 usrFdiskPartLib 库提供的 usrFdiskPartCreate 函数创建两个分区，并将分区表信息写入块设备主引导扇区中，再之后调用 dpartDevCreate 函数添加 CBIO 分区管理层，dpartDevCreate 将读取块设备主引导扇区中分区表信息，构建各分区信息（如各分区起始扇区号，各分区扇区总数），之后调用文件系统层接口 dosFsDevCreate 函数在每个分区上构建 dosFs 文件系统，在这完成之后，应用层可以独立使用各个分区，就好像底层有多个块设备存在一样。

```
STATUS createFsOnPartition(){
    BLK_DEV *pBlkDev;
    CBIO_DEV_ID pCbioCache;
```

```
    CBIO_DEV_ID pCbioParts;

    //调用底层块设备驱动函数初始化 BLK_DEV 结构。
    pBlkDev=xxxDevCreate(…);
    //调用 dcacheDevCreate 函数添加 CBIO 基本功能层和数据缓冲层。
    pCbioCache=dcacheDevCreate(pBlkDev, NULL，0，”ATA Hard Disk Cache”);

    //调用 usrFdiskPartCreate 创建分区表，并将分区表写入底层块设备主引导扇区。
    //此处我们创建了两个扇区，各占用整个底层块设备容量的一半。
    usrFdiskPartCreate(pCbioCache, 2, 50, 0, 0);
    //调用 dpartDevCreate 添加 CBIO 分区管理层。该函数使用 usrFdiskPartRead 函数
    //解析分区表，创建各分区信息。
    pCbioParts=dpartDevCreate(pCbioCache, 2, usrFdiskPartRead);

    //调用 dosFsDevCreate 函数在每个分区上创建 dosFs 文件系统。
    //注意我们使用 dpartPartGet 函数返回一个块设备结构，本质上返回的是一个分区。
    //但是对于文件系统层而言，一个分区和一整个块设备之间没有任何区别，只不过分区
    //的起始扇区编号不是 0，仅此而已，而这一点区别也被 CBIO 分区管理层屏蔽。
    dosFsDevCreate(“/dosA”, dpartPartGet(pCbioParts, 0), 16, 0);
    dosFsDevCreate(“/dosB”, dpartPartGet(pCbioParts, 1), 16, 0);

    //对于初次使用的设备，我们使用 dosFsVolFormat 进行格式化。
    dosFsVolFormat(“/dosA”, 0, 0);
    dosFsVolFormat(“/dosB”, 0, 0);
    return OK;
}
```

之后对于“/dosA”，“/dosB”的使用就如同普通块设备一样，前文中已给出例子，此处不再论述。

由此可见，借助于 CBIO 分区管理层以及 usrFdiskPartLib，底层块设备实现并不会因为需要支持多个分区而变得更加复杂，实际上作为多个分区使用的块设备底层驱动与作为单个分区使用的块设备驱动实现之间没有任何差别。这就大大简化了底层块设备驱动的设计。我们不用关心一个硬盘块设备究竟被分为几个扇区使用，涉及分区创建以及分区管理的工作已然由内核其他组件负责。对于底层块设备驱动而言，仅仅需要关心底层有几个硬盘块设备，对于多个实际块设备的情况，必须指定控制器号。虽然 Vxworks 下使用硬盘的场合不常见，更不用说使用多个硬盘设备的情景，但是底层驱动并不能对此进行假设，而必须对这种情况进行考虑。为了简化设计复杂度以及从通常情况考虑，我们令每个硬盘控制器只驱动一个硬盘驱动器，此时只需根据控制器号即可判断是哪个硬盘设备，通常主硬盘设备被作为操作系统启动设备，控制器编号为 0，从硬盘即一般数据存储设备，控制器编号为 1.

基于对多个硬盘设备支持的考虑，我们对上文中给出的驱动自定义结构 ATA_BLK_DEV 做些调整，增加一个控制器号字段，表示对应哪个块设备（主，从或更多），我们需要对每个块设备分配一个 BLK_DEV 结构。此外为了阻止并发关键资源访问，还需要定义一个互斥

信号量（注意：系统中的每个任务都可能在使用底层块设备代码读写设备）。调整后的ATA_BLK_DEV 结构如下。

```
typedef struct  _blk_dev_struct{
    BLK_DEV     blkDev; //必须是第一个成员。
    UINT32      regBase; //硬盘控制器寄存器基地址，是否必要还值得考虑。
    FUNCPTR     intHandler; //硬盘控制器中断具体响应函数指针。
    UINT8       intLevel; //硬盘控制器中断号。

    UINT8       controller; //控制器号
    SEMAPHORE       muteSem;       /*  互斥信号量*/
}ATA_BLK_DEV;
```

8.5.4 初始化函数实现

遵循 Vxworks 驱动初始化的惯例，我们将块设备初始化分为两个函数进行实现：（1）ataBlkDrv 函数，完成底层块设备控制器寄存器的配置工作，复位块设备，校正磁头位置等等；（2）ataBlkDevCreate 函数，完成驱动自定义结构的分配，初始化以及底层所需其他所有资源的分配工作，该函数执行完毕后，底层块设备驱动以及块设备本身将处于正常工作状态。

实际上块设备初始化与底层块设备硬件的构成以及每个块设备的分区数有很大关系，我们在下面的例子中，假设平台有多个块设备构成（虽然一般不可能，尤其是对于嵌入式平台），每个块设备分成多个分区使用，这样单个块设备可以支持不同的文件系统，虽然 Vxworks 极不“建议”单个块设备使用多个文件系统的做法。我们采用最复杂的组成方式在于对读者介绍此种情况下块设备的组织形式，这样对于单个块设备，单个分区就可以做到得心应手。

基于以上的假设，ataBlkDrv 函数原型如下。

```
STATUS ataBlkDrv(int controller, UINT32 regBase, int intLevel);
```

对于存在多个块设备的情况，对于每个块设备都必须调用 ataBlkDrv 函数一次，每次必须给出块设备控制器编号，块设备控制器寄存器基地址，中断号。

ataBlkDrv 函数实现如下。

```
#define MAX_ATA_DISKS   4 //最多同时支持四个硬盘块设备
LOCAL BOOL ataDrvInstalled = FALSE;
LOCAL ATA_BLK_DEV       ataBlkDev[MAX_ATA_DISKS];

STATUS ataBlkDrv(int controller, UINT32 regBase, int intLevel){
    int ctrl;
    ATA_BLK_DEV *pAtaDev

    if(ataDrvInstalled==FALSE){ //只在初次调用时执行如下代码。
        for(ctrl=0;ctrl<MAX_ATA_DISKS;ctrl++)
            ataBlkDev[ctrl].controller=-1; //-1 表示该项空闲，可使用。
```

```
        ataDrvInstalled=TRUE;
    }
    if(controller>MAX_ATA_DISKS-1){
        printErr("controller %d out of range.\n", controller);
        return -1;
    }
    pAtaDev=&ataBlkDev[controller];
    if(pAtaDev->controller!=-1){
        printErr("controller %d has been used.\n", controller);
        return -1;
    }
    //save parameters
    pAtaDev->controller=controller;
    pAtaDev->regBase=regBase;
    pAtaDev->intLevel=intLevel;
    pAtaDev->intHandler=dummyHandler; //dummyHandler 实现为直接返回。
    semMInit (&pAtaDev->muteSem, SEM_Q_PRIORITY | SEM_DELETE_SAFE |
                        SEM_INVERSION_SAFE);
    ataReset(controller, regBase);
    ataRecalibrate(controller, regBase);

    return OK;
}
```

实际上对于各个硬盘控制器对应的寄存器基地址以及中断号，也完全可以在底层块设备驱动中进行定义，而后根据传入的控制器号对 ATA_BLK_DEV 结构中字段进行初始化。

注意：硬盘控制器各寄存器将以基地址为基准做固定偏移，通常不同偏移处的寄存器对于所有的硬盘控制器都具有相同的功能，对于嵌入式平台下硬盘设备，可查阅硬盘控制器手册获取各偏移地址处寄存器的功能。硬盘控制器初始化主要包括复位和磁头重新校准两个步骤，并没有诸如网卡设备需要配置一大堆寄存器的繁琐步骤。应该说复位和重新校准只是确保一个最初的预知状态，实际上硬盘控制器上电后就可以使用。复位和重新校准并不单是最初初始化时被调用，实际上每当读写硬盘设备发生连续错误时，都需要进行复位和重新校准。复位函数还需要通过 BLK_DEV 结构提供给上层使用。

ataBlkDrv 函数只是对 ATA_BLK_DEV 结构中自定义字段进行了初始化，尚未完成 BLK_DEV 这个提供给上层使用的关键结构成员的初始化，这个初始化工作将由 ataBlkDevCreate 函数完成，该函数将返回一个 BLK_DEV 结构。从本章前面章节的相关代码示例中，读者也可看出，ataBlkDevCreate 函数用于创建一个块设备的过程，其返回的 BLK_DEV 将被用于完成底层块设备驱动与文件系统层（实际上是 CBIO 中间层）的衔接。

ataBlkDevCreate 函数定义原型如下。

```
BLK_DEV *ataBlkDevCreate(int controller);
```

我们需要参数指定返回哪个块设备对应的 BLK_DEV 结构。注意在调用 ataBlkDevCreate 函数前必须调用 ataBlkDrv 函数完成驱动自定义结构 ATA_BLK_DEV 的最初初始化。

ataBlkDevCreate 函数具体实现代码如下。

```
 /* ATA_TYPE 定义平台上硬盘设备的关键参数：
 * 硬盘总扇区数；柱面数，磁头数，每磁道扇区数，每扇区字节数，写预补偿控制字节。
 */
typedef struct ataType
    {
    int nTsectors;   /*total number of sectors*/
    int cylinders;       /* number of cylinders */
    int heads;           /* number of heads */
    int sectors;         /* number of sectors per track */
    int bytes;           /* number of bytes per sector */
    int precomp;         /* precompensation cylinder */
    } ATA_TYPE;
//ataType 数组定义了平台上所有硬盘设备的参数，此处我们以一主一从两个硬盘设备为例。
//如下常量将根据具体硬盘手册给出的数据进行定义。
ATA_TYPE ataTypes[]={
    {MS_TSECTORS, MS_CYLINDERS, MS_HEADS, MS_SECTORS,
            MS_BYTES, MS_PRECOMP},
    {SL_TSECTORS, SL_CYLINDERS, SL_HEADS, SL_SECTORS,
            SL_BYTES, SL_PRECOMP},
    {0, 0, 0, 0, 0,0}
};

BLK_DEV *ataBlkDevCreate(int controller){
    ATA_BLK_DEV *pAtaDev;
    ATA_TYPE *pType;
    BLK_DEV *pBlkDev;

    pAtaDev=&ataBlkDev[controller];
    if(pAtaDev ->controller!=controller){
        printErr("ataBlkDrv should be called first.\n");
        return NULL;
    }
    if(ataType[controller].nTsectors==0){
        printErr("controller %d corresponds no hard disk device.\n", controller);
        return NULL;
    }

    pType=&ataTypes[controller];
    pBlkDev=&pAtaDev->blkDev;

    //初始化 BLK_DEV 结构
    pBlkdev->bd_nBlocks         = pType->nTsectors;
    pBlkdev->bd_bytesPerBlk     = pType->bytes;
```

```
    pBlkdev->bd_blksPerTrack   = pType->sectors;
    pBlkdev->bd_nHeads         = pType->heads;
    pBlkdev->bd_removable      = FALSE; //不可移除介质
    pBlkdev->bd_retry          = 1;
    pBlkdev->bd_mode           = O_RDWR;
    pBlkdev->bd_readyChanged   = FALSE; //硬盘设备总是处于就绪状态
    pBlkdev->bd_blkRd          = ataBlkRd;
    pBlkdev->bd_blkWrt         = ataBlkWrt;
    pBlkdev->bd_ioctl          = ataIoctl;
    pBlkdev->bd_reset          = ataReset;
    pBlkdev->bd_statusChk      = ataStatus;


    //此语句表示实际返回的是 ATA_BLK_DEV 结构首地址。
    //但是是作为 BLK_DEV 结构返回的。
    return &pAtaDev->blkDev;
}
```

ataBlkDevCreate 完成了最重要的一个工作：即对 BLK_DEV 结构进行初始化，该结构将作为 ataBlkDevCreate 函数返回值返回，提供给上层（CBIO 中间层）使用，此后上层对于底层块设备的所有操作（读写，控制，状态查询，复位）都将通过 BLK_DEV 结构中的信息完成。

注意：虽然我们在 ataBlkDrv 函数中要求用户传入一个硬盘控制器使用的中断号，但是在 ataBlkDevCreate 函数并没有使用中断，没有调用 intConnect 函数注册中断。这个原因在前文中已经提到，Vxworks 下块设备读写函数都必须实现为阻塞方式，即读写函数返回时，其缓冲区中的数据必须得到处理，对于读操作函数，缓冲区中必须是要求读取的扇区数据，对于写操作而言，缓冲区中数据必须已经写入设备中对应扇区。所以读写函数实现上，在配置硬盘控制器进行读写后，一直在不断检查操作完成状态位，直到状态位显示硬盘控制器已经完成块设备的数据读写，才返回。基于 Vxworks 上层要求的这种特殊工作方式，对于 Vxworks 下块设备驱动并不使用中断，这一点与通用操作系统完全不同。

8.5.5 读设备函数实现

按照内核接口要求，底层块设备驱动中读设备函数必须具有如下原型：

```
STATUS ataBlkRd
(
    BLK_DEV *pDev,
    int     startBlk,
    int     nBlks,
    char    *pBuf
);
```

参数 1（pDev）：块设备结构指针，该结构本质上是一个驱动自定义结构，在 ataBlkRd 函数开始处一般都经过一个强制转换，转成驱动自定义结构，此时除了可以使用 BLK_DEV 结

构中的字段外，也可以使用驱动自定义结构中的字段，如块设备控制器寄存器基地址等。
参数 2（startBlk）：文件系统层需要读取的块设备中数据的起始块号。注意这是一个绝对块号，ataBlkRd 函数需要将其转换成块设备控制器所需的磁头号，磁道号，磁道内扇区号。
参数 3（nBlks）：文件系统层需要读取的总块数。注意参数 2 和参数 3 中的块指的是块设备本身使用的物理块，文件系统层在调用底层驱动读设备函数前，已经完成逻辑块到物理块的转换。
参数 4（pBuf）：存放被读取数据的缓冲区。

ataBlkRd 函数实现如下。

```
STATUS ataBlkRd
    (
    BLK_DEV *pDev,
    int     startBlk,
    int     nBlks,
    char    *pBuf
    )
{
    return (ataBlkRW (pDev, startBlk, nBlks, pBuf, O_RDONLY));
}
```

由于硬盘设备读写函数在硬盘控制器的配置上差别很小，故我们将块设备读写函数集成在一个底层函数中实现，这个函数就是 ataBlkRW，该函数具体实现将在下文给出。

8.5.6 写设备函数实现

写设备函数与读设备函数具有相同的原型，如下所示。

```
STATUS ataBlkWrt
(
    BLK_DEV *pDev,
    int     startBlk,
    int     nBlks,
    char    *pBuf
);
```

各参数的含义如同读设备函数，不同的是现在是将 pBuf 中的数据写入块设备中。
ataBlkWrt 函数实现如下。

```
STATUS ataBlkWrt
    (
    BLK_DEV *pDev,
    int     startBlk,
    int     nBlks,
    char    *pBuf
    )
{
    return (ataBlkRW (pDev, startBlk, nBlks, pBuf, O_WRONLY));
```

```
}
```

ataBlkWrt 实现上也是调用更底层的 ataBlkRW 函数。
ataBlkRW 函数实现代码如下。注意我们在参数传递过程中，完成由 BLK_DEV 结构向 ATA_BLK_DEV 结构的强制类型转换。

```
LOCAL int ataRetry=3;
TATUS ataBlkRW
    (
    ATA_BLK_DEV *pDev,
    int         startBlk,
    int         nBlks,
    char        *pBuf,
    int         direction    /*read or write*/
    )
{
    BLK_DEV     *pBlkdev= &pDev->blkDev;
    ATA_TYPE    *pType    = &ataTypes[pDev->controller];
    nSecs = pBlkdev->bd_nBlocks;
    int             status    = ERROR;
    int             retryRW0     = 0;
    int             retryRW1     = 0;
    int             retrySeek    = 0;
    int             cylinder;
    int             head;
    int             sector;
    int             nSecs;
    int             ix;

    if ((startBlk + nBlks) > nSecs){
        logMsg("startBlk=%d nBlks=%d: 0 - %d\n", startBlk, nBlks, nSecs, 0, 0, 0);
        return (ERROR);
    }

    semTake (&pDev->muteSem, WAIT_FOREVER);

    for (ix = 0; ix < nBlks; ix += nSecs){
        //由绝对扇区号计算磁道号（cylinder），磁道内扇区号（sector），磁头号（head）。
        cylinder = startBlk / (pType->sectors * pType->heads);
        sector     = startBlk % (pType->sectors * pType->heads);
        head        = sector / pType->sectors;
        sector     = sector % pType->sectors + 1;
        nSecs      = min (nBlks - ix, ATA_MAX_RW_SECTORS);

        //retryRW1 表示不做任何其他操作，在某次操作失败后，
```

```
            //连续尝试的次数（最多 ataRetry 次）。
            //retryRW0 则表示某次操作连续失败了 ataRetry 次后，重新校正一次磁头位置，
            //在重新校正了 ataRetry 次后仍然失败，则直接退出本次操作。
            retryRW0 = 0; //表示操作失败后，连续试验的次数。
            retryRW1 = 0; //表示操作失败后，重新校正的次数。

            //ataRW 函数将完成具体块设备数据读取的工作。
            //ataCmd 则控制硬盘控制器进行非数据读写之外的操作，如校正磁头，寻道，复位。
            while (ataRW(pDev, cylinder, head, sector, pBuf, nSecs, direction) != OK){
                if (++retryRW0 > ataRetry){
                    (void)ataCmd (pDev, ATA_CMD_RECALIB, 0, 0);
                    if (++retryRW1 > ataRetry)
                        goto done;

                    etrySeek = 0;
                    while (ataCmd (pDev, ATA_CMD_SEEK, cylinder,head) != OK)
                        if (++retrySeek > ataRetry)
                            goto done;

                    retryRW0 = 0;
                }
            }
            startBlk += nSecs;
            pBuf += pBlkdev->bd_bytesPerBlk * nSecs;
        }
        status = OK;

done:
        if (status == ERROR)
            (void)errnoSet (S_ioLib_DEVICE_ERROR);
        semGive (&pDev->muteSem);
        return (status);
}
```

我们将不再给出 ataRW，ataCmd 函数的具体实现，对于硬盘控制器底层驱动代码，请读者参看开源操作系统 Linux 下的具体实现或者参考文献[7]中块设备驱动一章相关内容。

8.5.7 设备控制函数实现

虽然对于文件和目录，文件系统层提供了很多控制选项，但是对于底层块设备驱动而言，其需要支持的选项很少，就以 FIODISKFORMAT 选项为例，这是要求格式设备的一个选项，事实上这个选项不会直接通过调用底层驱动中的 ataIoctl 函数实现，因为文件系统的格式化必须定义文件系统元数据，而这对于底层块设备驱动而言，是不可能实现的，所以事实上，

对于 FIODISKFORMAT 选项的响应将由文件系统层本身负责，最后传递给底层块设备驱动时，调用的其实是设备读写函数，即将文件系统元数据写入块设备开始几个扇区中。
内核头文件 h/ioLib.h 中定义了所有可用选项，这些选项都是通过文件系统进行处理的，直接传递给 ataIoclt 函数进行处理的基本没有，所以我们的 ataIoctl 实现也非常简单，如下所示。
注意：如下所示的 ataIoctl 实现是一个实际中可用的实现方式，并非是一个简单示例，虽然我们没有对任何选项进行实现。

```
/*******************************************************************************
* ataIoctl - do device specific control function
*
* This routine is called when the file system cannot handle an ioctl() function.
*
* RETURNS:    OK or ERROR.
*/

STATUS ataIoctl
      (
      ATA_BLK_DEV    *pDev,
      int            function,
      int            arg
      )
{
      int status  = ERROR;
      semTake (&pDev->muteSem, WAIT_FOREVER);

      switch (function)
      {
            case FIODISKFORMAT:
                  (void) errnoSet (S_ioLib_UNKNOWN_REQUEST);
                  break;
            default:
                  (void) errnoSet (S_ioLib_UNKNOWN_REQUEST);
      }
      semGive (&pDev->muteSem);
      return (status);
}
```

8.5.8 设备状态查询函数实现

设备状态查询函数被上层调用检查底层块设备是否还在，对于硬盘设备而言，其实完全可以直接将 BLK_DEV 结构中的 bd_statusChk 初始化为 NULL，即表示设备始终存在，这个函数主要针对可移除软盘设备使用。

```
/*******************************************************************************
* ataStatus - check status of a ATA/IDE disk controller
* This routine check status of a ATA/IDE disk controller.
* RETURNS: OK, ERROR if the card is removed.
*/

STATUS ataStatus (ATA_BLK_DEV    *pDev)
{
    BLK_DEV *pBlkDev=pDev->blkDev;
    pBlkDev->bd_readyChanged=FALSE; //硬盘设备总是在那儿，不会中途被移除。
    return (OK);
}
```

8.5.9 设备复位函数实现

设备复位函数在发生严重操作错误时用以复位设备。该函数具体实现代码如下。

```
STATUS ataReset(ATA_BLK_DEV    *pDev){
    int retrySeek=0;
    int status=OK;

    while (ataCmd (pDev, ATA_CMD_RESET, 0, 0) != OK)
        if (++retrySeek > ataRetry){
            status=ERROR;
        }

    return status;
}
```

ataReset 实现调用 ataCmd 函数完成复位命令的发送，并等待复位是否成功，如果没有成功，则连续尝试 3 次（ataRetry=3），如果仍然失败，则返回 ERROR，表示复位失败。

至此，我们完成 Vxworks 块设备驱动基本所有函数的实现，当然对于硬盘设备相关代码，我们没有给出。Vxworks 下块设备驱动由于上层组件特殊的工作方式，使得块设备驱动只能使用查询工作方式，所有函数均实现为阻塞模式，即函数不完成所要求的功能绝不返回。这种工作方式效率上差于中断方式，但是基于 CBIO 数据缓冲层以及 Vxworks 特殊的使用环境，这种效率上的差异基本可以不计（也无法计较）。

8.6 本章小结

本章首先从文件系统层出发，详解介绍了 rawFs 以及 dosFs 两种文件系统的接口函数，以及以实际代码为例介绍了在两种文件系统下如何初始化块设备，使得块设备对用户层可用。而后从块设备驱动内核层次出发详细介绍了块设备驱动中内核不同层次间如何衔接，着重对底层块设备驱动层与上层之间衔接起着关键作用的 BLK_DEV 结构作了比较详细的分析。而后，从 BLK_DEV 结构以及上层工作方式出发，我们总结出了 Vxworks 下块设备驱动的基本结构，最后实际代码为例详细介绍了底层块设备驱动几个关键函数的实现方式。编写一个设备驱动必须弄清楚两个方面的内容：其一是与内核接口关系；其二就是理解被驱动设备的工作流程。Vxworks 块设备驱动与内核之间的接口都体现在 BLK_DEV 结构中，换句话说，内核接口比较简单，所以实际上块设备驱动设计的关键是对被驱动块设备的工作方式和流程的理解，这与网口驱动是一样的。而这一点又是设备相关的，所以读者应在充分理解内核接口的基础上，仔细研读块设备手册，并按照本章给出的块设备驱动基本结构进行代码编写，应不难应对实际中的任务。

事实上，Vxworks 下很少场合会涉及到硬盘块设备驱动的设计，更不用说多个硬盘块设备的驱动设计， Vxworks 下使用较多块设备主要还是 FLASH 设备。基于这一点以及 FLASH 本身的特殊结构和读写方式，Vxworks 在 CBIO 中间层之下又插入了一个中间层-TFFS 中间层来完成与底层 FLASH 驱动的接口，下一章中我们将详细介绍 Vxworks 下 FLASH 设备的驱动设计。

第九章 FLASH 设备驱动

Vxworks 操作系统通常使用在嵌入式平台上，上一章我们对硬盘块设备驱动进行详细分析，但是对于嵌入式平台很少有使用硬盘设备的，绝大多数都是使用 FLASH 设备，本章将详细介绍 Vxworks 下 FLASH 设备驱动的设计和实现。

9.1 FLASH 设备

9.1.1 综述

FLASH 设备总体上分为两类：NorFlash 和 NandFlash，二者具有完全不同的数据访问方式，但是二者也具有很多相同的特点，如基于块的擦除。NOR 和 NAND 是现在市场上两种主要的非易失闪存技术。Intel 于 1988 年首先开发出 NOR flash 技术，彻底改变了原先由 EPROM 和 EEPROM 一统天下的局面。紧接着，1989 年，东芝公司发表了 NAND flash 结构，强调降低每比特的成本，更高的性能，并且象磁盘一样可以通过接口轻松升级。

大多数情况下闪存只是用来存储少量的代码，这时 NOR 闪存更适合一些。而 NAND 则是高数据存储密度的理想解决方案。NOR 的特点是芯片内执行(XIP, eXecute In Place)，这样应用程序可以直接在 flash 闪存内运行，不必再把代码读到系统 RAM 中。NOR 的传输效率很高，在 1～4MB 的小容量时具有很高的成本效益，但是很低的写入和擦除速度大大影响了它的性能。NAND 结构能提供极高的单元密度，可以达到高存储密度，并且写入和擦除的速度也很快。应用 NAND 的困难在于 flash 的管理和需要特殊的系统接口。

性能比较

FLASH 闪存是非易失存储器，可以对称为块的存储器单元块进行擦写和再编程。任何 FLASH 器件的写入操作只能在空或已擦除的单元内进行，所以大多数情况下，在进行写入操作之前必须先执行擦除，擦除的结果是将块内所有的位都置为 1。向 FLASH 写入数据的过程就是将相应位设置为 0. 无论 Nand 还是 Nor Flash 擦除的单元都是一个块，通常每个块的大小为 128KB。而写入操作上二者存在较大的差别，对于 NorFlash 可以对单个字节进行写入，而不对周围其他字节造成任何影响，而对于 NandFalsh 则不然，NandFlash 不支持单个字节的写入操作，只能采取面向一个页面的写入操作，每个页面通常为 2112 个字节（2048 个数据字节+64 个空闲字节）。换句话说，在 NandFlash 中要修改一个字节的内容，则首先必须将这个字节所在的页面全部读出来，在内存中修改这个字节，而后将修改后整个页面再写入 NandFlash 中。如下图 9-1 所示为 Nand 和 Nor 二者之间总体参数比较。

NOR-NAND主要参数比较		
参数	NOR	NAND
容量	1MB~16MB	8MB~128MB
XIP功能	支持	不支持
擦除时间	很慢（~5s）	快（3ms）
读时间	很快	快
写时间	慢	快
擦除次数	10,000~100,000	100,000~1,000,000
接口	类似SRAM接口	信号线复用
访问方式	随机	面向页
价格	较高	低

图 9- 1 Nand 和 Nor Flash 总体参数比较

9.1.2 芯片硬件接口差别

NOR flash 带有 SRAM 接口，有独立的地址线来寻址，可以很容易地存取其内部的每一个字节，具有 EIP 在芯片内执行功能。NAND 器件使用同一的数据线来进行操作，每次数据读写操作，都要分为几个周期使用这一组相同的信号线进行命令，地址，数据的传输。如下图 9-2 所示总结了两种 Flash 间的接口差别。

Required Hardware Pins

NAND Flash: 24 Pins (x16)		**NOR Flash: 41 Pins**	
I/O device-type interface, composed of:		Random-access interface, typically composed of:	
CE#	Chip enable	CE#	Chip enable
WE#	Write enable	WE#	Write enable
RE#	Read enable	OE#	Output enable
CLE	Command latch enable	D[15:0]	Data bus
ALE	Address latch enable	A[20:0]	Address bus
I/O[7:0]	Data bus (I/O[15:0] for x16 parts)	WP#	Write protect
WP#	Write protect		
R/B#	Ready/busy		
PRE	Power-on read enable (useful for system boot)		

图 9- 2 Nand 和 Nor Flash 接口信号线对比

9.1.3 容量和成本

如下图 9-3 所示，NAND flash 的每个存储单元尺寸几乎是 NOR 器件的一半，由于生产过程更为简单，NAND 结构可以在给定的模具尺寸内提供更高的容量，也就相应地降低了价格。NOR flash 占据了容量为 1～16MB 闪存市场的大部分，而 NAND flash 只是用在 8～128MB 的产品当中，这也说明 NOR 主要应用在代码存储介质中，NAND 适合于数据存储，NAND 在 CompactFlash、Secure Digital、PC Cards 和 MMC 存储卡市场上所占份额最大。

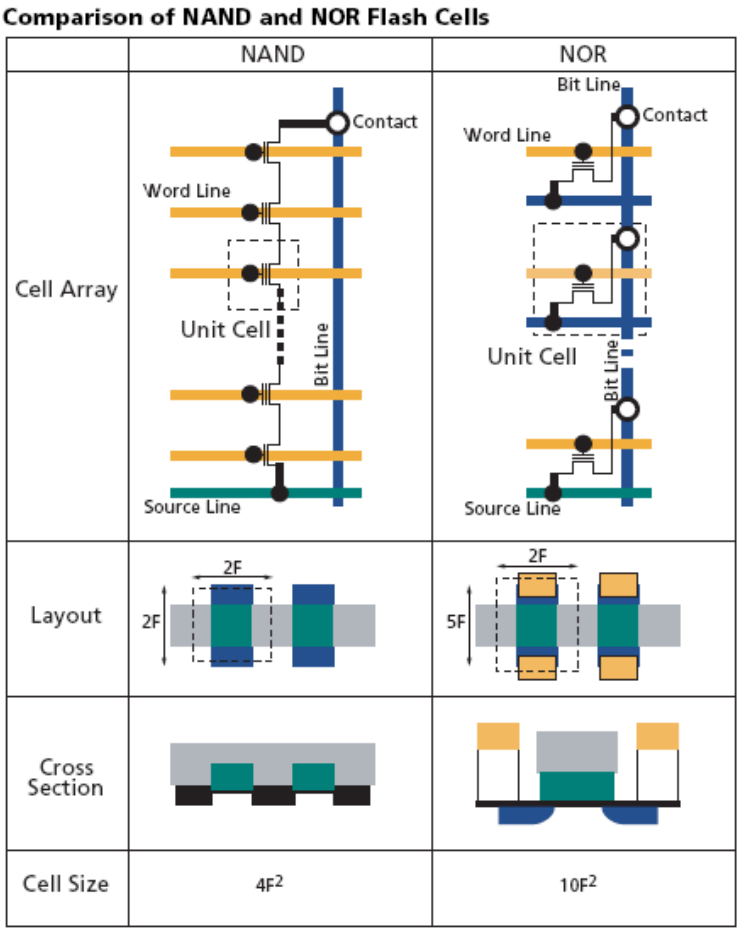


图 9- 3 Nand 和 Nor Flash 存储单元比较

9.1.4 可靠性和耐用性

采用 flahs 介质时一个需要重点考虑的问题是可靠性。对于需要扩展 MTBF 的系统来说，Flash 是非常合适的存储方案。可以从擦除次数、位交换和坏块处理三个方面来比较 NOR 和 NAND 的可靠性：（1）NAND 闪存每个块的最大擦写次数是一百万次，而 NOR 的擦写次数是十万次；（2）所有 flash 器件都受位交换现象的困扰。在某些情况下(很少见，NAND 发生的次数要比 NOR 多)，一个比特位会发生反转或被报告反转了。一位的变化可能不很明显，但是如果发生在一个关键文件上，这个小小的故障可能导致系统停机。如果只是报告有问题，多读几次就可能解决了。当然，如果这个位真的改变了，就必须采用错误探测/错误更正(EDC/ECC)算法。位反转的问题更多见于 NAND 闪存，NOR 很少有这个问题，NAND 的供应商建议使用 NAND 闪存的时候，同时使用 EDC/ECC 算法。这个问题对于用 NAND 存储多媒体信息时倒不是致命的。当然，如果用本地存储设备来存储操作系统、配置文件或其他敏感信息时，必须使用 EDC/ECC 系统以确保可靠性；（3）NAND 器件中的坏块是随机分布的。以前也曾有过消除坏块的努力，但发现成品率太低，代价太高，根本不划算。NAND 器件需要对介质进行初始化扫描以发现坏块，并将坏块标记为不可用。在已制成的器件中，如果通过可靠的方法不能进行这项处理，将导致高故障率。

9.1.5 易于使用

由于 NorFlash 提供类似 RAM 的接口，所以可以非常直接地使用基于 NOR 的闪存，可以像其他存储器那样连接，并可以在上面直接运行代码。由于需要复用同一组信号线，NAND 要复杂得多。各种 NAND 器件的存取方法因厂家而异。向 NAND 器件写入信息需要相当的技巧，因为设计师绝不能向坏块写入，这就意味着在 NAND 器件上自始至终都必须进行虚拟映射。

9.1.6 软件支持

当讨论软件支持的时候，应该区别基本的读/写/擦操作和高一级的用于磁盘仿真和闪存管理算法的软件，包括性能优化。在 NOR 器件上运行代码不需要任何的软件支持，在 NAND 器件上进行同样操作时，通常需要驱动程序，也就是内存技术驱动程序(MTD)，NAND 和 NOR 器件在进行写入和擦除操作时都需要 MTD。使用 NOR 器件时所需要的 MTD 要相对少一些，许多厂商都提供用于 NOR 器件的更高级软件，这其中包括 M-System 的 TrueFFS 驱动，该驱动被 Wind River System、Microsoft、QNX Software System、Symbian 和 Intel 等厂商所采用。驱动还用于对 DiskOnChip 产品进行仿真和 NAND 闪存的管理，包括纠错、坏块处理和损耗平衡。

9.2 深入 NANDFLASH

上一节中我们总体上对 Nand 和 Nor Flash 进行了介绍，实际上，还有很多内容没有提到，本节将深入介绍 NandFlash 相关内容。诚如上一节所述，NandFlash 设备复用一组信号线，如下图 9-4 所示为 NandFlash 设备的结构框图，图 9-5 所示为 NandFlash 接口信号线定义。

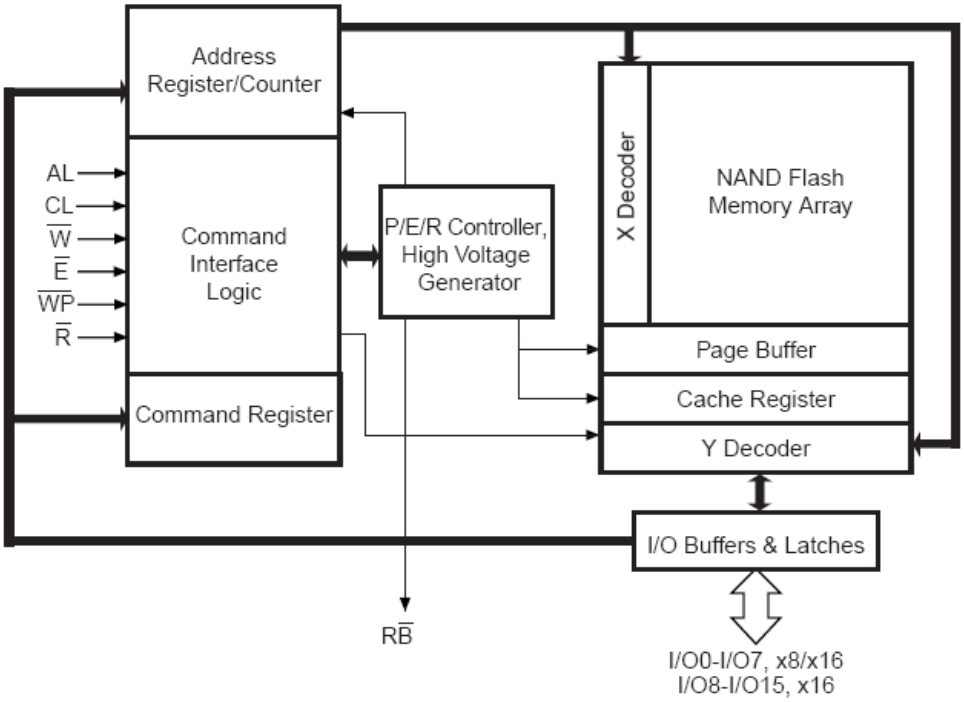


图 9- 4 NandFlash 结构框图

I/O8-15	Data Input/Outputs for x16 devices
I/O0-7	Data Input/Outputs, Address Inputs, or Command Inputs for x8 and x16 devices
AL	Address Latch Enable
CL	Command Latch Enable
$\overline{E}$	Chip Enable
$\overline{R}$	Read Enable
$R\overline{B}$	Ready/Busy (open-drain output)
$\overline{W}$	Write Enable
$\overline{WP}$	Write Protect

图 9- 5 NandFlash 接口信号线定义

NandFlash 设备只有一组数据线（图 4 中 IO0~IO7，16 位宽的 NandFlash 还包括 IO8~IO15），这组数据线需要传递命令，地址，用户数据，为了完成命令，地址和用户数据在同一组数据线上传递，NandFlash 使用 AL，CL，W，R 等控制信号线来决定当前数据线上传递的内容。由于数据线复用，一个读写或控制操作的发出往往需要经过几个周期的总线操作。这一点与 NorFlash 具有独立的地址和数据线完全不同。

NandFlash 总体上分为两种内部结构，这主要是根据块的大小进行区分的：上文介绍中我们以大块结构，每个块 128KB 作为示例，小块结构中每个块只有 16KB；NandFlash 的读写都是面向页的，而擦除则是面向块的，每个块由多个页面构成，如大块 Flash 结构下，每个块由 64 个页面组成，每个页面大小为 2112（2048+64）字节；小块 Flash 结构下，每个块由 32 个页面组成，每个页面大小为 528（512+16）字节，这每个页面中多出来的 64 或者 16 字节通常专门用于 ECC 校验字节。大小块 NandFlash 的读写方式存在较大的差异，小块结构的 NandFlash 对每个页面中的 528 个字节划分为三个区域：（1）0~255 字节为第一区域；

（2）256~511 字节为第二区域；（3）512~527 字节为第三区域，每个区域读写时使用不同的命令。而对于大块结构的 NandFlash，将提供足够的地址线访问每个页面内 2112 个字节，所以一个页面所有的字节读写方式都是一致的。另外大小块不同结构下的 NandFlash 的地址构成方式也有差别。读者编写自己的 NandFlash 驱动时必须注意到大小块结构的这些区别。有关 NandFlash 地址的问题下文中还将有讨论。
通常大于 1Gb 的 NandFlash 采用大块结构，小于 1Gb 的 NandFlash 采用小块结构，1Gb 的 NandFlash 大小块结构都有。如下图 9-6 所示为一个 2Gbit NandFlash 内部结构层次。

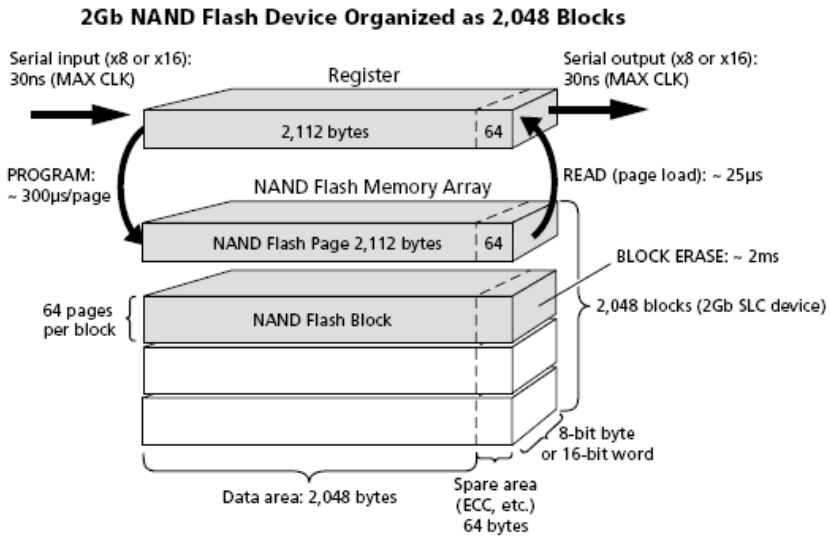


图 9- 6 NandFlash 内部结构层次图

从上图 9-6 中可以看出，每次对于 NandFlash 读写的数据首先转移到一个与页面相同大小的寄存器中，而后由 NandFlash 完成内部寄存器到对应页面的数据读写操作。这种工作方式有些类似于硬盘控制器。不过硬盘结构层次比 FLASH 要简单的多，其基本操作单元是扇区，不再有其他层次。

9.3 深入 NORFLASH

9.3.1 NorFlash 存储器特点

NorFlash 做为一种非易失性存储器，在原理、技术和结构上，与 ROM、PROM、EPROM 和 EEPROM 存储器有显著的不同。它是一种可快速擦除可现场编程的快擦写存储器。这种特性决定了 Flash 做为 BIOS、在线擦写、掉电保护数据以及分区保护数据等场合有着广泛的应用。 由于其内部结构的特殊性，Flash 存储器最主要的特点在于其内部状态机（Internal State Machine）及指令序列（Command Sequences）的操作模式，因此 Flash 需要较复杂的软件支持。除了读取数据（Read Array Data）以外的其它所有操作，如生产商 ID（Manufacture ID）及器件 ID（Device ID）识别，分区写保护（Sector Protection）/去写保护（Sect

or Unprotection），擦除（chip erase/sector erase），写数据/校验，复位等操作都需要通过器件内部的指令寄存器（Command Register）启动内嵌算法（embeded algorithms）来完成。

9.3.2 NorFlash 命令集 BCS / SCS

Intel 从早期的 8 位闪存芯片 28F008 开始，就为闪存芯片的操作定义了操作规程和一套相应的命令，一般就称为“28F008”或者“008”。后来正式命名为闪存操作基本命令集，即 BCS，其要点如下：闪存芯片平时处于随机读出状态，此时的闪存芯片就像普通的 ROM 芯片，访问地址决定了具体的存储单元；除非芯片处于写入状态，往芯片任意一个单元写都意味着向芯片发命令，因此是对其控制器的访问，而不是对其存储单元的访问；根据命令代码的不同，闪存芯片内部的状态机进入不同的状态，从而可以进行不同的操作。BCS 中有以下几条命令：（1）随机读出（Read Array）。向芯片的任意地址写 0xff，使芯片“复位”进入“读存储阵列” 状态，在此状态下 CPU 可以想访问普通内存那样从闪存芯片随机读出。闪存芯片加电后的初始状态就是随机读出状态。(2)读状态寄存器（Read Status Register）。 向芯片的任意地址写 0x70，表示要求从芯片读出其状态寄存器，此后可以从芯片的任意地址读出状态寄存器的内容。(3)写入（Program）,要求写入芯片的若干存储单元。往目标块起始地址写 0x40，表示要求进入写模式，或者称为“编程模式”。进入写模式后，就可以对目标块进行随机写入，就像写静态 RAM 一样。对每个单元只能写一次（要再次对某单元写入，需要先对该单元所在的块进行擦除操作后，才能再进行）。(4)清除状态寄存器（Clear Status Register）。向芯片的任意地址写 0x50，表示要求清除状态寄存器。

单字节或者字写入。先向需要写入的单元写入 0x40 或 0x10，然后写入具体数据。(5)成块擦除（Block Erase）。向目标块内任意地址写 0x20（或者 0x28），表示要求擦除芯片内一个块的所有内容。(6)暂停擦除（Erase Suspends）。向目标块内任意地址写 0xb0，表示要求暂停擦除操作。需要时，可以通过往目标块内的任意地址写 0xd0 来恢复擦除成块擦除确认，或者继续擦除。往目标块内的任意地址写 0xd0，表示确认要求擦除芯片内一个块的所有内容，或者在暂停擦除后恢复擦除操作。每当向芯片发出命令时，状态寄存器的最高位（bit 7）就会变成 0，当操作完成后，该 bit 会被硬件置为 1。

随着闪存技术的发展，Intel 后来对 BCS 进行了扩展，称为 SCS（Scaleable Command Set），即“可扩充命令集”。其中增加了芯片加锁（Set/Clear Lock Bit）、缓冲写入。芯片配置、整片擦除操作等命令功能。上面 BCS 和 SCS 描述的操作只适用于 Intel 的闪存芯片，或者使用 Intel BCS/SCS 规范的芯片。

9.3.3 NorFlash 接口访问标准

（1）JEDEC ID

在闪存技术发展的初期，只有 Intel 一家生产闪存芯片，产品单一。后来慢慢多起来，新投入闪存芯片生产的厂商往往使自己的产品与 Intel 兼容，所以 Intel 的 BCS 就成为了事实上的标准。但是，随着闪存芯片生产厂商的增加，闪存芯片种类和规格也越来越多，一方面导致 BCS 不够用了，另一方面就是有的厂商使用了不同于 BCS 的规程。例如，同样是读取芯片 ID，由 AMD 生产的芯片则需要向片上地址 0 写入 0xAA5。实际上，INTEL 和 AMD 就是最主要的闪存芯片厂商。在这种情况下，特别是对于通用计算机系统，在对闪存芯片进行操作之前，首先要搞清这是由哪个厂商生产的什么芯片。为此，就需要有一个“中立”、受所有闪存芯片厂商支持的手段，使得计算机软件可以从芯片中读取这些信息。于是，便有了所谓的“Jedec ID”以及相应当命令：读芯片 ID（Read ID）。往芯片任何地址写 0x90，表示要求从芯片读出固化在芯片内的身份信息，如由谁生产，什么型号等等，称为“Jedec ID”。这些信息并不占用闪存芯片在内存空间的地址，而是存储在内部控制器中。此后，CPU 可以从片上地址 0 开始逐个字节读出这些信息。

（2）CFI

由于 Jedec ID 所提供的信息实在是太少了（只提供了生产商，产品型号），因此，“公共闪存接口”（Common Flash Interface）,即 CFI，便应运而生。这是由存储芯片工业界定义的一种获取闪存芯片物理会和结构参数的操作规程和标准。CFI 规定，各厂商可以使用自己的操作规程和命令集，但是必须有统一的编号，并且在软件到查询下提供这个编号。而且，闪存芯片可以同时支持两套操作规程和命令集，让软件选择使用。同时采用不同的实现思想，又规定闪存芯片必须提供更多的信息，包括：芯片的电源电压是多少，擦除/写入操作是否需要额外的电源，如果需要是多少，芯片的容量有多大，分成几个擦除块，各种操作所需时间的典型值和最大值，等等。为此，还定义了一个用来提供详细信息的标准数据结构，凡是支持 CFI 的闪存芯片在收到查询时都应该能够按照这个数据结构的格式提供信息，即 CFI 信息块。此外，还可以与所支持的操作规程和命令集配套提供附加的参数和信息。CFI 查询的规程和命令如下：向芯片地址 0x55 写入 0x98，然后从片上地址 0x10 开始读出数据。如果读出的内容依次为：“Q”、“R”、“Y”，则该芯片支持 CFI。然后从芯片上连续的地址 0x13 ～ 0x30 读出 CFI 信息块的固定部分。除了固定部分以外，后面还有一个无符号整数数组，其大小取决于芯片上的存储区间划分成几个擦除区间，数组中的每一个无符号整数都描述一个擦除区间。一个擦除区间中可以有若干个大小相同的擦除块。如果从地址 0x15 和 0x16 读出的 16 位无符号短整数 P 非 0，则在片上地址 P 处还有一个“主算法扩充表”。这就是与“主算法”（即芯片中主要的操作规程和命令集）配套的参数和其他信息。如果从地址 0x19 和 0x20 读出的 16 位无符号短整数 T 非 0，则在片上地址 T 处还有一个“次算法扩充表”。这就是与“次算法”（即芯片所支持的另外一套操作规程和命令集）配套的参数和其他信息。

9.4 FLASH 地址问题

Nand 或者 Nor Flash 中数据（特指出现在数据线的数据）写入时地址（特指出现在地址线上

的数据）的确定需要根据 Flash 芯片手册以及实际硬件连线决定，二者缺一不可。

对于 Nand Flash，由于其地址数据线复用，故命令，地址，数据的写入都是通过同一组总线写入的（姑且称这组总线为数据线）。对于当时写入的究竟是命令，地址还是数据，由 ALE，CLE 两个信号线的电平决定。当 ALE=1，CLE=0 时，表示数据线上当前放置的是地址；而当 ALE=0，CLE=1 时，则表示的是命令；当 ALE=0，CLE=0 时，则表示的是数据；ALE=1，CLE=1 位无效组合。这是单从 NandFlash 角度来看的。从 CPU 的角度而言，向某个外设写入数据，必是以*（Type *）addr=data 的形式，换句话说，在当以 Nandflash 的角度来看问题时，我们一直在说得是 data，对于系统地址总线上的数据则没有考虑（注意 Nandflash I/O 线连接到系统数据线）。实际上，我们通过变换系统地址总线来控制 Nandflash 的 ALE，CLE。首先要访问 Nandflash，我们必须知道 Nandflash 在系统地址空间的基地址和地址范围，这样在系统地址总线上放入数据时，我们必须首先确定该地址数据 Nandflash 的地址空间，否则数据将写入到其他外设上。其次在了解到 Nandflash 基地址后，我们需要从电路实际硬件连线确知 Nandflash 的 ALE，CLE 被连接到了系统地址总线的哪些位上。注意 Nandflash 的 ALE，CLE 是通过变换地址总线位来控制的，没有专门的信号线进行控制，这一点与同属于 Nandflash 的 WEn，RD 信号线不同。此处为便于说明问题我们假设 Nandflash 基地址为 0x42000000，而 ALE 被连接到系统总线的 bit18，CLE 则连接到 bit19。则当我们对 Nandflash 进行编程时，首先写入的是命令，那么我们将在系统数据总线上放置要写入的命令，如 Reset 命令 0xF0，现在我们要在确定系统地址总线上的数据。要向 Nandflash 写入命令，则 CLE=1，ALE=0，那么 bit19=1，bit18=0；而为了将数据写入 Flash，地址必须落入到 Nandflash 地址空间中，为简便起见，我们直接加上 Nandflash 的基地址，如此就得到系统地址总线上应该放置的数据为 0x42080000。即需要向 Nandflash 写入 Reset 命令时，我们在程序中进行如下的编码：*（unsigned char *）0x42080000=0xF0.

注意以上用语中的系统地址总线，而非地址总线，因为很多 Nandflash 接口总线相对于系统总线作了偏移，而程序中发出的地址则是系统地址总线上的数据。这一点需要特别注意，这需要查看电路原理图方能确知 Nandflash 的 ALE，CLE 到底连接到系统总线的哪些位上，而不是 Nandflash 接口总线的哪些位上。

对于 Norflash 的情况，有关地址的情况比较简单，Norflash 具有其自身的地址总线，但是很多情况下 Norflash 接口地址总线相对与系统地址总线也是做了一个偏移。故要向 0x55 写入一个命令时，而如果 Norflash 接口地址总线相对与系统地址总线偏移为 1，则程序中应该向 0xAA 写入一个命令。

一般而言，对于 Nandflash 和 Norflash，其直接接口地址总线相对于系统地址总线都会存在一个偏移。就笔者目前经验来看，基本所有的 Flash 都有偏移存在，尤其当 Flash I/O 线为 16 位宽时。

9.5 Vxworks 下 Flash 设备驱动内核层次

FLASH 是嵌入式系统下非常常见和常用的一类存储设备，为此，Vxworks 为 Flash 设备驱动在 CBIO 中间层下又提供了一个 TFFS 中间层来直接负责与底层 FLASH 设备的交互。TFFS

中间层存在的原因在于 FLASH 不同于硬盘设备的特殊操作方式：Flash 设备写入前必须进行擦除，而且擦除是面向块的，即每次只能擦除一个块，对于 NandFlash 而言，每个块包括多个页面，NandFlash 的数据写入是基于页面的，即每次只能写入一整页的数据，实际上读取也是按一整页进行的，只不过 NandFlash 内部集成有缓冲区，每次 NandFlash 内部控制器将数据从 Flash 存储介质上读取一整页的数据到内部缓冲区中，而后可以字节的方式访问内部缓冲区中的数据，本质上，对于 NandFlash 而言，读写都是面向页面的。而且 Flash 的页面大小与硬盘设备的基本操作单元-扇区的大小完全不同，故不可直接在 CBIO 中间层对 Flash 设备驱动进行管理，而需要插入一个新的层次，将 Flash 中页面映射为文件系统中使用的扇区，对 Flash 中坏块进行管理，对块使用进行均衡（因为 Flash 设备中每个块的擦除次数都是有限的，为了尽量增加 Flash 设备的使用寿命，必须使用均衡算法对 Flash 设备中每个块都等概率的使用到）等等，这些都由这个新插入的中间层负责，这个中间层就是 Vxworks 提供的 TFFS 中间层。TFFS 是 True Flash File System 的缩写形式，也称为 TrueFFS 中间层。TrueFFS 中间层对 Flash 设备进行封装，提供一个普通硬盘设备接口给上层使用，即通过 TrueFFS 中间层，文件系统层（CBIO 中间层）就像操作一个硬盘设备一样操作底层 Flash 设备。如同 CBIO 中间层，TrueFFS 中间层也不是一个单一的层次，其内部还分为几个子层次，完成 Flash 设备向硬盘设备模拟实现的各个功能。

Flash 设备驱动内核层次如下图 9-7 所示。

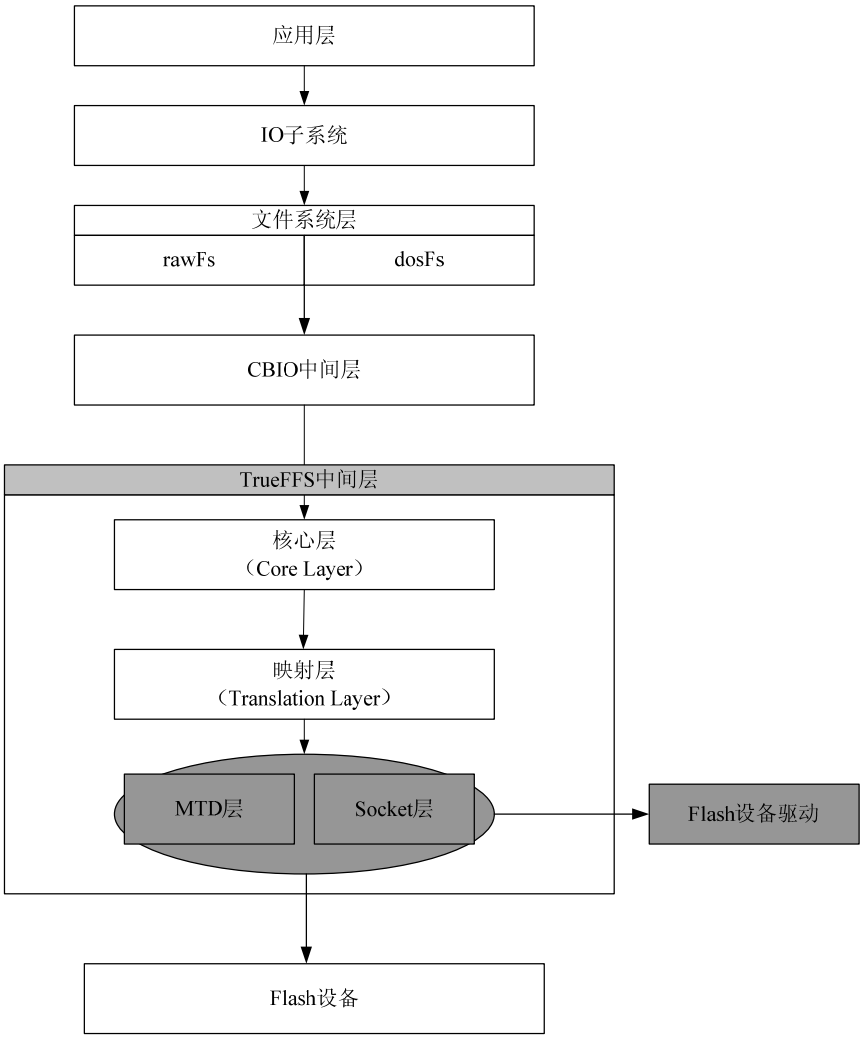


图 9- 7 Flash 设备驱动内核层次

图 9-7 中，TrueFFS 中间层位于 CBIO 中间层与 Flash 设备之间，由 TrueFFS 中核心层完成与 CBIO 中间层的交互，而由 TrueFFS 中 MTD 层和 Socket 层共同完成底层 Flash 设备的驱动。值得注意的是，在 TrueFFS 中间层之下直接就是 Flash 硬件设备，并不存在一个 Flash 设备驱动层，换句话说，Flash 设备驱动将包含在 TrueFFS 中间层实现中，具体的由其中的 MTD 层和 Socket 层两个子层次实现，这两个层次从上下层关系来看是一种平行关系，各自实现 Flash 设备驱动中所需的不同功能，具体的，MTD 层完成主要驱动函数（设备读写，擦除）的实现，而 Socket 层则完成设备电源管理，设备探测以及设备容量探测函数的实现。TrueFFS 中间层四个子层次中，其中：

（1） 核心层

该层将 TrueFFS 中其他三个层次衔接在一起，起着一个 TrueFFS 中间层总管理中枢的作用，其负责接收从上层发送的设备操作请求，调用其他三个子层次提供的功能共同完成这个请求，并处理一个全局资源的分配和释放问题。

（2） 映射层

该层的作用相当关键，映射层提供 Flash 设备的硬盘设备视图，其完成文件系统层中使用的块到 Flash 设备中实际块的映射。具体的，映射层使用块分配映射表完成文件系统中使用块到 Flash 设备中实际块的映射过程，这个映射表还负责错误校正和块使用均衡的工作，这对于 Flash 设备的使用寿命起着决定性作用。由于 NorFlash 和 NandFlash 在数据读写方式上的差别（NorFlash 是面向字节读写的，而 NandFlash 是面向页读写的），所以映射层对于 NorFlash 和 NandFlash 分别提供了两种不同的实现。

（3） MTD 层

该层提供 Flash 设备识别，读写，擦除，地址映射函数的具体实现。该层实现由 Flash 驱动开发人员完成。该层将实现在 BSP 下 tffsConfig.c 以及 tffsXXX.c 文件中。其中 tffsConfig.c 定义有 MTD 层一个重要的数组 mtdTable，该数组中保存了 MTD 层实现的所有 Flash 设备的识别函数，这些识别函数将完成 MTD 层实现向上层的注册。

（4） Socket 层

该层提供底层块设备硬件的控制函数实现，这些函数包括电源管理函数，Flash 设备探测函数，Flash 设备容量（又称窗口）设置函数，以及 Socket 层注册函数。Socket 注册函数将以上定义的这些设备控制函数注册给 TrueFFS 其他子层次使用。该层实现由 Flash 驱动开发人员完成。该层将实现在 BSP 目录下 sysTffs.c 文件中。注意此处的 sysTffs.c 文件名以及以上的 usrConfig.c 文件名都是系统要求的，必可随意更改，而 MTD 层具体实现文件 tffsXXX.c 的文件名则可自定义，如定义为 tffsMtd.c。

可以看出，TrueFFS 中间层的功能有些特殊，其实现的一部分由内核本身提供，而另一部分将由 Flash 设备驱动开发人员提供，换句话说，Flash 设备驱动实际上是作为 TrueFFS 中间层的一部分实现的，而不是完全在 TrueFFS 中间层之下的一个新的层次。在 TrueFFS 中间层的内部四个层次中，核心层和映射层由 Vxworks 操作系统实现，而 MTD 层和 Socket 层则由 Flash 驱动开发人员实现。当然 Vxworks 操作系统本身在 MTD 层也提供了一些典型 Flash 设备的读写和擦除函数实现，但是一般很少有机会使用这些现成的代码，而需要针对嵌入式平台下使用的特定 Flash 设备完成对应的读写以及擦除函数实现，即实现 TrueFFS 中的 MTD 子层。TrueFFS 中间层中的 Socket 子层次实现起来相对比较简单，实际上，从下文中的分析来看，直接使用 BSP 提供的模板实现（当然需要修改其中的一些参数）即可。Socket 子层

次只在一些非常特殊的情况下才需要关注，如功耗要求非常严格的场合。

9.6 TrueFFS 初始化

作为中间层和驱动本身，其初始化必须完成三个方面的工作：（1）内部初始化，包括各子层次所需资源分配；（2）MTD 层和 Socket 层初始化，这两个子层次实现为 Flash 设备驱动本身，必须完成这两个子层次向 TrueFFS 其他子层次（核心层）的注册；（3）TrueFFS 中间层向其上层（CBIO 中间层）的注册。

（1） 内部初始化

内部初始化由 ttyDrv 函数完成，该函数由 ttyDrv 库提供，将在 Vxworks 启动过程中被调用。具体的，在 usrRoot 函数调用，代码如下。

```
#ifdef     INCLUDE_TFFS
    tffsDrv ();                    /* it should be after pcmciaInit() */
#endif     /* INCLUDE_TFFS */
```

必须进行 INCLUDE_TFFS 宏定义方可包括 TFFS 中间层。

tffsDrv 函数初始化 TrueFFS 相关结构，Vxworks 系统最多支持 5 个 Flash 驱动，即同时可以向 TrueFFS 中间层注册 5 个 MTD 层和 Socket 层的实现。Vxworks 维护一个内核数组对 Flash 驱动进行管理，这个数组将在 tffsDrv 中进行初始化。tffsDrv 完成自身初始化后，最后将调用 tffsConfig.c 文件中实现的 flRegisterComponents 函数进行 MTD 层和 Socket 层的初始化-即完成 MTD 层和 Socket 层驱动向 TrueFFS 中其他层（核心层）的注册。

（2） MTD 层和 Socket 层初始化

MTD 层和 Socket 层具体完成底层 Flash 设备的驱动，其中 MTD 层必须实现 Flash 设备识别，读写，擦除，地址映射函数，Socket 层必须实现 Flash 设备电源管理和设备探测等函数。MTD 层和 Socket 层共同组成 Flash 设备驱动，同时也隶属于 TrueFFS 中间层。

Socket 层的初始化在 tffsDrv 中触发，tffsDrv 将在完成 TrueFFS 中其他层次资源的初始化后，调用 flRegisterComponents 函数对 MTD 层和 Socket 层进行初始化。flRegisterComponents 函数定义在 tffsConfig.c 文件中，该函数实现非常简单，其并不完成任何具体初始化工作，而是直接调用 sysTffs.c 文件中定义的 sysTffsInit 函数具体完成 Socket 层驱动的初始化工作：向 TrueFFS 核心层注册 Socket 层驱动。sysTffsInit 函数完成所有 Socket 层驱动实现的注册工作，Vxworks 最多支持 5 个驱动实现。sysTffsInit 对各 Socket 层驱动初始化函数的调用顺序将决定各驱动对应的驱动号，第一个被初始化的 Socket 层驱动对应的驱动号为 0，依次加 1，直到 4.这个驱动号非常重要，将在 MTD 层初始化过程中完成 Socket 层和 MTD 层驱动的衔接，Socket 层和 MTD 层共同构成 Flash 设备驱动，所以必须在驱动工作之前完成二者之间的联系。例如某个平台下同时具有 NorFlash 和 NandFlash，此时需要实现两个 Flash 驱动，即两个 MTD 层实现和两个 Socket 层实现。基于这个前提，我们的 sysTffsInit 函数实现如下。在下文 Flash 驱动的所有代码示例中，我们都将基于这个前提，即基于 TrueFFS 中间层我们将同时实现对 NorFlash 和 NandFlash 的驱动。

```
/*******************************************************************************
* sysTffsInit - board-level initialization for TrueFFS
*
* This routine calls the socket registration routines for the socket component
```

```
* drivers that will be used with this BSP. The order of registration determines
* the logical drive number given to the drive associated with the socket.
*
* RETURNS: N/A
*/

LOCAL void sysTffsInit (void)
{
    norRegister (); //NorFlash 驱动号为 0.
    nandRegister(); //NandFlash 驱动号为 1.
}
```

NorRegister 完成 NorFlash Socket 层驱动的初始化工作，NandRegister 则完成 NandFlash Socket 层驱动的初始化工作。这两个函数的具体实现代码我们将在下文“Flash 驱动 Socket 层实现”一节给出。

在 Socket 层驱动注册中起着衔接作用的 flRegisterComponents 函数定义在 tffsConfig.c 文件中，该文件是 BSP 组成的一部分，是专用于 TrueFFS 中间层的配置文件，其中除了定义有 flRegisterComponents 函数外，还定义有 MTD 层一个关键数组 mtdTable，TrueFFS 中间层将调用该数组中元素（函数指针类型）指向的函数完成对底层 Flash 设备的识别，所以在 mtdTable 数组中添加设备识别函数也是 MTD 层实现的一个部分。

Vxworks 操作系统本身已然提供了一些典型 Flash 设备的驱动实现，所以 mtdTable 数组中已经存在这些典型设备的识别函数，对于一个平台特定 Flash 设备，在自实现了 MTD 层本身之后，还必须将 MTD 层中实现的识别函数注册到 mtdTable 数组中。MTD 层实现的识别函数将完成设备读写和擦除以及地址映射函数向 TrueFFS 核心层的注册。而识别函数自身将在 TrueFFS 首次使用设备时被调用，具体的将在 Flash 设备创建函数 tffsDevCreate 中（下议）中被调用。

MTD 层的初始化（注册）总结：MTD 层注册工作在 MTD 层实现的 Flash 设备识别函数中完成。当 TrueFFS 中间层初次访问一个 Flash 设备时（调用 tffsDevCreate 创建 Flash 设备之时），其将遍历 mtdTable 数组中每个元素指向的 Flash 设备识别函数对底层 Flash 硬件设备进行识别，直到一个函数返回 OK，才停止对数组中余下函数的调用，所以识别函数在 mtdTable 数组中是位置敏感的。Flash 识别函数通过 Flash 标准接口（如 NorFlash 的 CFI 接口）读取 Flash 信息，查看 Flash 设备类型，从而决定底层 Flash 驱动是否针对该设备，如果可以识别（即 MTD 层驱动可用），则将 MTD 层实现的设备读写，擦除，地址映射函数注册到 TrueFFS 中间层中，供上层之后使用对底层 Flash 设备进行操作。

（3） TrueFFS 中间层向上层的注册

如同普通块设备，TrueFFS 中间层向上层的注册工作将推迟到 Flash 设备创建过程中。ttyDrv 库提供 tffsDevCreate 函数完成 Flash 设备的创建工作。该函数调用原型如下：

```
BLK_DEV * tffsDevCreate
    (
    int tffsDriveNo,                /* TFFS drive number (0 - DRIVES-1) */
    int removableMediaFlag          /* 0 - nonremovable flash media */
    );
```

参数 1（tffsDriveNo）：Flash 设备对应的驱动号，这个驱动号在 sysTffsInit 函数中分配，一

个驱动号对应一种 Socket 层实现，这个驱动号将被用以在 tffsDevCreate 中完成 MTD 层驱动和 Socket 层驱动的衔接，二者结合共同完成对底层 Flash 设备的驱动。
参数 2：设备可移除标志位，通常都是 0，表示不可移除。

tffsDevCreate 函数最终将调用 MTD 层提供的 Flash 设别函数，对底层 Flash 设备进行辨别，注册 MTD 层驱动实现，此时 MTD 层和 Socket 层实现将完成衔接，共同驱动底层 Flash 设备工作。tffsDevCreate 函数对 Flash 识别函数调用过程如下图 9-8 所示。

注意 tffsDevCreate 函数的返回值是一个 BLK_DEV 结构指针，这个返回值将被 CBIO 层或文件系统层使用，完成 TrueFFS 中间层与其上层的衔接。BLK_DEV 结构的使用此时就如同块设备驱动初始化函数中返回的 BLK_DEV 结构。

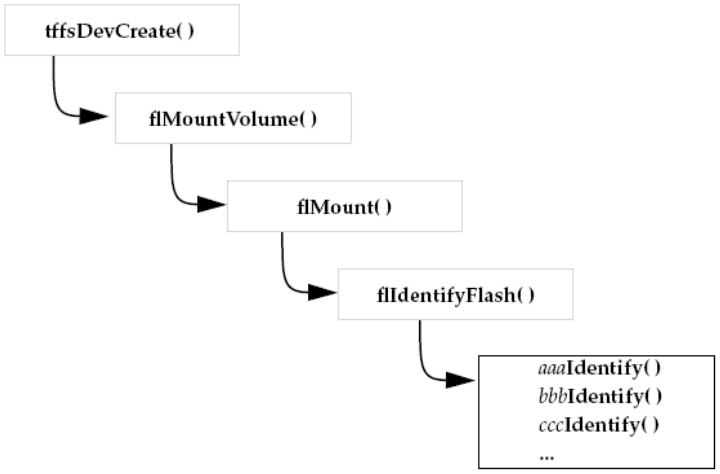


图 9- 8 tffsDevCreate 调用 Flash 识别函数流程

图 9-8 中诸如 xxxIdentify 函数由 MTD 层实现。该函数完成 MTD 层实现的设备读写，擦除和地址映射函数向 TrueFFS 核心层的注册。

TrueFFS 初始化过程总结：

（1） TrueFFS 中间层中核心层，映射层，Socket 层驱动在 Vxworks 操作系统启动过程中完成初始化工作。

（2） TrueFFS 中 MTD 层在调用 tffsDevCreate 函数进行 Flash 设备创建时完成初始化，并同时完成与 Socket 层驱动的对接，二者共同完成 Flash 设备的驱动。

9.7 Flash 设备创建和使用

要使得 Flash 设备可以文件和目录方式被用户访问，必须在系统设备列表中创建一个 Flash 设备节点，完成前文图 7 中各个层次之间的衔接。

（1） 文件系统层与 IO 子系统之间的衔接，由文件系统层初始化函数（如 dosFsLibInit，rawFsInit）在 Vxworks 启动过程中完成。

（2） TrueFFS 中间层内部子层次之间的衔接分为两个阶段：其一是 Socket 子层次与核心层和映射层之间的衔接将在 Vxworks 启动过程中完成；其二 MTD 子层次与核心层

和映射层以及和 Socket 子层次之间的衔接将在 Flash 设备创建时完成。

（3） TrueFFS 中间层与 CBIO 层之间的衔接将在 Flash 设备创建时完成。

（4） 文件系统层和 CBIO 层之间的衔接将在调用文件系统层函数对 CBIO 中间层封装的 CBIO_DEV_ID 结构进行封装时衔接。

以上四个步骤中，在 Vxworks 操作系统启动过程中完成工作上文中已经做了较为详细的介绍，现在我们以启动后的初始化工作为依据，讨论如何进行后续初始化使得 Flash 设备可被用户层访问。

与普通磁盘块设备不同，Flash 设备必须工作在 TrueFFS 中间层下，为了可被 TrueFFS 中间层可用，Flash 设备必须按照 TrueFFS 要求的特定格式进行预格式化。预格式化化的目的是对 Flash 块进行标识和管理。注意 Flash 设备直接归 TrueFFS 中间件管理，故其格式化首先将根据 TrueFFS 的要求进行，而不是直接使用文件系统层提供的格式化组件对 Flash 设备进行格式化。TrueFFS 中映射层将完成文件系统层中设备使用模式向 Flash 设备使用模式的转换。总而言之，对 Flash 设备的使用必须使用 TrueFFS 中间层提供的格式化工具进行预格式化，不可跳过这一步直接使用文件系统层提供的格式化工具（如 dosFsVolFormat），这一点要特别注意。tffsDrv 库中提供了一个 tffsDevFormat 函数专门用于对 Flash 设备进行预格式化，即在 Flash 设备上建立元数据，这些数据将被 TrueFFS 中间层使用对 Flash 设备中各个块进行定位和管理，并完成文件系统层块到 Flash 块的映射。

tffsDevFormat 函数调用原型如下。

```
STATUS tffsDevFormat
    (
    int tffsDriveNo,            /* TrueFFS drive number (0 - DRIVES-1) */
    int arg                 /* pointer to tffsDevFormatParams structure */
    );
```

参数 1：驱动号。该驱动号用以寻址 Flash 设备在 MTD 层和 Socket 层对应的驱动。注意 tffsDevFormat 一般在 tffsDevCreate 函数尚未调用时被用以格式化 Flash 设备，换句话说，此时尚未完成 MTD 层的初始化工作以及 MTD 层与 Socket 层的衔接，所以 tffsDevFormat 函数底层实现上实际上自己调用了 MTD 层提供的 Flash 设备识别函数完成 MTD 层驱动函数的注册以及 MTD 层和 Socket 层的衔接工作，因为格式化过程中必须向 Flash 设备写入和读取数据，故必须完成 MTD 层的注册才能使用其中的函数对 Flash 设备进行读写操作。

对于 Flash 设备，Vxworks 最多支持 5 个驱动号，即同时系统内可以 5 套 MTD 层和 Socket 层的不同实现用以驱动 5 种不同的 Flash 设备。驱动号的分配在 sysTffsInit 函数完成，该函数将对每种 Socket 层驱动初始化函数进行调用，每次调用分配一个驱动号，而 Socket 层与 MTD 层的接合将由调用 tffsDevFormat 或 tffsDevCreate 函数传入的驱动号完成。这两个函数底层实现上都将调用 MTD 层中实现的 Flash 设备识别函数，这个传入的驱动号将被作为 MTD 层查找对应 Socket 层的依据，二者共同完成同一个 Flash 设备的驱动，如下图 9-9 所示。

实际上当 MTD 层根据驱动号查找到对应 Socket 层时，两个层之间将使用数据结构指针互相引用，从而便于相互间的访问。

驱动号	MTD层实现	Socket层实现	被驱动设备
0	MTD层实现0	Socket层实现0	Flash设备0
1	MTD层实现1	Socket层实现1	Flash设备1
2	MTD层实现2	Socket层实现2	Flash设备2
3	MTD层实现3	Socket层实现3	Flash设备3
4	MTD层实现4	Socket层实现4	Flash设备4

图 9- 9 MTD 层和 Socket 层由驱动号进行衔接

tffsDevFormat 的第二个参数是一个指针，指向一个 tffsDevFormatParams 结构，该结构定义在 h/tffs/tffsDrv.h 内核头文件中，如下所示。

```
typedef struct
    {
    tffsFormatParams    formatParams;
    unsigned                formatFlags;
    } tffsDevFormatParams;
```

该结构用于指定 Flash 设备格式化的参数和控制位。对该结构本书不再做讨论，读者可直接查看 tffsDrv.h 头文件内容，获取结构的详细定义。由于 tffsDevFormatParams 结构比较复杂，故 Vxworks 提供一个默认参数和控制位定义，当以 0 作为参数传入 tffsDevFormat 函数时，内核将使用默认参数和控制位对底层 Flash 设备进行格式化。注意：默认参数将整个 Flash 设备置于 TrueFFS 中间层的管理之下，如果用户在 Flash 的开始处预留一段空间作其他使用，必须使用自定义参数。实际上我们可以将 Flash 设备在系统地址空间中的基地址做一个偏移即可达到预留 Flash 空间的作用，同时使用系统默认参数对余下的 Flash 空间进行格式化。完成对底层 Flash 块设备的格式化后，下面我们就需要挂载该 Flash 设备，使得 Flash 设备可被用户使用。

要挂载 Flash 设备，首先我们必须调用前文中介绍的 tffsDevCreate 函数创建一个设备，之后使用 tffsDevCreate 函数返回的 BLK_DEV 结构，我们调用 CBIO 缓冲层接口函数 dcacheDevCreate 进行封装，之后使用 dcacheDevCreate 返回的 CBIO_DEV_ID 结构调用 dosFs 文件系统接口函数 dosFsDevCreate 最终完成设备的创建工作，dosFsDevCreate 调用后即完成 Flash 设备驱动内核所有层次的衔接工作，此时 Flash 设备已然被添加到系统设备列表中，对用户可用了。

Flash 设备创建代码如下所示。

```
STATUS createFlashDev(int drive, char *devName){
    BLK_DEV *pBlkDev;
    CBIO_DEV_ID pCbioDev;
```

```
    //使用默认参数格式化 Flash 设备。
    //完成 MTD 层和 Socket 层的衔接，该函数调用完成后，底层 Flash 驱动已完全就绪。
    tffsDevFormat(drive, 0);

    //调用 TrueFFS 中间层设备创建函数创建一个 Flash 设备。
    pBlkDev=tffsDevCreate(drive, 0/*removable*/);
    //添加 CBIO 基本功能层和数据缓冲层。
    pCbioDev=dcacheDevCreate(pBlkDev, NULL, dosFsCacheSizeDefault, devName);
    //完成 CBIO 中间层与文件系统层的衔接，将 Flash 设备添加到系统设备列表中。
    //dosFsDevCreate 完成调用后，用户即可使用 devName 指定的设备名访问 Flash 设备。
    //访问方式如同一般的硬盘块设备，可创建文件和目录。
    dosFsDevCreate(devName, pCbioDev, 0, NONE);

    return OK;
}
```

由于 Flash 的设备使用之前都需要经过挂载，故 Vxworks 内核提供了一个更高层次的封装函数 usrTffsConfig，该函数完成对以上代码中 tffsDevCreate，dcacheDevCreate，dosFsDevCreate 的封装，usrTffsConfig 函数调用原型如下。

```
STATUS usrTffsConfig
    (
    int         drive, /* drive number of TFFS */
    int         removable,      /* 0 - nonremovable flash media */
    char *    fileName   /* mount point */
    );
```

故我们 createFlashDev 函数实现就变得十分简单，如下所示。

```
STATUS createFlashDev(int drive, char *devName){
    dosFsDevFormat(drive, 0);
    usrTffsConfig(drive, 0, devName);
    return OK;
}
```

注意：dosFsDevFormat 函数将删除 Flash 设备上已有的所有数据，故对于一个已经格式化过的 Flash 设备而言，只需进行挂载即可，无需进行重新格式化，所以大多数时候 usrTffsConfig 函数就已经完成使 Flash 设备可用的目的。对于初次使用的 Flash 设备，我们进行如下调用：createFlashDev(0, “/tffs0”);

而后，就可以使用标准的应用层接口函数对 Flash 设备进行文件创建，读写等操作了。如下代码所示。

```
int main(void){
    int fd;
    char buf[256];
    char *msg=”who cares.”.
```

```
    int msglen=strlen(msg);

    if((fd=open("/tffs0/test.txt", O_CREAT|O_RDWR, 0)<0){
        printErr("Cannot create file on flash device.\n");
        return -1;
    }
    write(fd, msg, msglen);
    close(fd);

    if((fd=open("/tffs0/test.txt", O_RDONLY, 0)<0){
        printErr("Cannot open file on flash device.\n");
        return -1;
    }
    read(fd, buf, msglen);
    close(fd);

    printf("write msg:%s\nread back:%s\n", msg, buf);
    return 0;
}
```

9.8 TrueFFS 核心层和映射层再议

TrueFFS 中间层中的 MTD 层和 Socket 层构成底层 Flash 驱动，这两个层次将由 Flash 驱动开发人员实现。TrueFFS 中间层中的其他两个关键层：核心层和映射层完成底层 Flash 驱动（即 MTD 层和 Socket 层）与上层（CBIO 中间层）之间的请求转换和传递的工作，这两个层次将由 Vxworks 操作系统自身实现。用户必须定义相关宏对内核实现的这两个层次的功能进行包括，其中核心层功能的包括由 INCLUDE_TFFS 宏控制，而基于 NorFlash 和 NandFlash 之间的不同工作方式，实现有两个不同映射层，如果平台只使用 NorFlash，则可以只包括 NorFlash 映射层实现，即只需定义 INCLUDE_TL_FTL 宏，不需要包括 NandFlash 映射层实现，同样的道理，如果平台上只使用 NandFlash，而没有使用 NorFlash，则只需包括 NandFlash 映射层实现，即此时只需定义 INCLUDE_TL_SSFDC 宏。如果平台上同时使用了 NorFlash 和 NandFlash，而且都工作在 TrueFFS 中间层下，则必须同时进行 INCLUDE_TL_FTL 和 INCLUDE_TL_SSFDC 宏的定义。

对于 TrueFFS 核心层和映射层功能的包括，只需在 config.h 文件中进行以上所述的相关宏定义即可，无需任何编码工作，而 TrueFFS 中 MTD 层和 Socket 层的实现则通常需要 Flash 驱动开发人员编写所有的代码，不过 Socket 层实现相对要简单的多，我们可以拷贝一个模板，进行简单修改即可，而 MTD 层的实现如果底层 Flash 设备是内核已提供支持的设备，则直接使用内核提供的代码即可，但是通常而言，针对平台特定 Flash 设备，都需要编写 MTD 层实现的所有代码，即设备识别函数，设备读写函数，设备擦除函数，设备地址映射函数。下两个小节我们将详细介绍构成底层 Flash 驱动的 MTD 层和 Socket 层的实现。

9.9 Flash 驱动 Socket 层实现

Flash 驱动有两个层次需要自实现：MTD 层和 Socket 层，这两个层次都属于 TrueFFS 中间层的一部分，但是必须由 Flash 驱动开发人员完成具体底层实现，TrueFFS 中核心层和映射层由 Vxworks 操作系统本身实现。按照约定，Socket 层实现代码必须定义在 sysTffs.c 文件中，该文件位于 BSP 目录下。

sysTffs.c 文件中必须实现如下函数：（1）sysTffsInit 初始化函数，这个函数调用所有 Socket 层注册函数，对每种 Socket 层实现分配一个驱动号；（2）xxxRegister 注册函数（如上文的 NorRegister 和 NandRegister），这个函数将 Socket 层实现的驱动函数注册到 TrueFFS 核心层，以便于之后操作 Flash 设备时使用；（3）Socket 层驱动具体实现函数，这些函数包括：Flash 设备电源管理函数，设备探测函数和设备容量设置函数等；（4）控制映射层和 MTD 层内核实现代码是否包含的宏定义，Vxworks 自身提供映射层两种不同实现方式，由相关宏定义进行控制，其次 Vxworks 同时提供一些典型 Flash 设备的 MTD 层实现，这些实现也由相关宏进行控制，如果平台上使用的正好使这些 Flash 设备，则可以直接包含这些内核实现代码，而无须自己实现 MTD 层了。虽然这些控制宏的定义也可以在 config.h 文件中完成，但是 Wind River 推荐将这些控制宏定义在 sysTffs.c 文件中。

再次提请注意：sysTffs.c 文件名是指定的，必须位于 BSP 目录下，不可更改文件名，该文件指定为 TrueFFS 中 Socket 层实现代码。对于存在多个 Flash 的 Socket 层实现的情况，可以对每个 Flash 设备提供一个文件，具体实现 Socket 层，而后通过 include 语句将这些文件统一包括到 sysTffs.c 文件中，此种情况下，sysTffs.c 中只需定义 sysTffsInit 函数和相关控制宏，而 xxxRegister 函数以及每个 Flash 设备的 Socket 层具体实现则可以放入各自对应的文件中。

在下面的示例中，我们同时驱动两块 Flash 设备：其一是一块 NorFlash 设备，其二是一块 NandFlash 设备，故我们独立出来两个文件，对应完成各自的 Socket 层实现，NorFlash 对应的 Socket 层实现代码定义在 norTffs.c 文件中，而 NandFlash 对应的 Socket 层实现代码定义在 nandTffs.c 文件中，而后我们在 sysTffs.c 文件中，加入如下所示两个 include 语句即可：

```
#include “norTffs.c”
#include “nandTffs.c”
```

首先让我们实现 Socket 层初始化总入口文件 sysTffs.c 的实现。此时该文件中需要完成 sysTffsInit 函数实现和相关控制宏的定义。

在 TrueFFS 初始化过程中，初始化总入口函数 tffsDrv 最终将调用 tffsConfig.c 中定义的 flRegisterComponents 函数完成 Socket 层的初始化工作。flRegisterComponents 直接调用 sysTffs.c 中定义的 sysTffsInit 函数完成具体的工作。sysTffsInit 必须对所有底层 Flash 设备对应的 Socket 层驱动进行初始化，将 Socket 层实现的驱动函数注册给 TrueFFS 中上层（映射层和核心层）使用。每完成一个 Flash 设备 Socket 层的注册，即分配一个驱动号，起始驱动号为 0，最大为 4，即系统同时支持 5 种不同的 Socket 层实现，可以同时驱动 5 个 Flash 设备。每个 Flash 设备对应的 Socket 层驱动号与在 sysTffsInit 中其注册函数的调用顺序相关，这一点特别需要注意，在使用 tffsDevFormat 以及 tffsDevCreate 函数时，都需要传递一个驱动号，这个驱动号就是在 sysTffsInit 中分配的驱动号，故必须注意各个 Flash 设备 Socket 层 xxxRegister 函数的调用顺序。

在我们的示例中，我们同时驱动 NorFlash 和 NandFlash，即需要两套 Socket 层实现，也就需

要两个驱动号，在本书前文中我们已经给出了 sysTffsInit 函数的实现代码，为便于读者查看，我们重新给出，如下所示。

```
/*******************************************************************************
* sysTffsInit - board-level initialization for TrueFFS
*
* This routine calls the socket registration routines for the socket component
* drivers that will be used with this BSP. The order of registration determines
* the logical drive number given to the drive associated with the socket.
*
* RETURNS: N/A
*/

LOCAL void sysTffsInit (void)
{
    norRegister ();
    nandRegister();
}
```

实现中 norRegister 函数首先被调用，这就表示 NorFlash 使用驱动号 0，则 NandFlash 则使用驱动号 1，这在格式化（tffsDevFormat）或者创建 Flash 设备（tffsDevCreate）时必须注意，如果要对 NorFlash 进行操作，则必须传递 0 作为驱动号；而要对 NandFlash 进行操作，则必须传递 1 作为驱动号。

norRegister 函数将完成 NorFlash Socket 层实现向 TrueFFS 上层的注册，该函数实现如下。注意 norRegister 函数定义在 norTffs.c 文件中。

```
/*******************************************************************************
* norRegister - registration routine for the NorFlash Socket Driver
*
* This routine populates the 'vol' structure for a logical drive with the
* socket component driver routines for the NorFlash. All
* socket routines are referenced through the 'vol' structure and never
* from here directly
*
* RETURNS: flOK, or flTooManyComponents if there're too many drives
*/
//定义在 norTffs.c 文件中。
LOCAL FLStatus norRegister (void)
{
    //noOfDrives 是一个内核全局变量，用以表示当前已分配的驱动号数目。
    //如 noOfDrives=2，则表示当前已分配两个驱动号，即系统同时具有两套 Socket 层实现，
    //驱动两个底层 Flash 设备。
    FLSocket vol = flSocketOf (noOfDrives);

    if (noOfDrives >= DRIVES) //DRIVES=5
```

```
        return (flTooManyComponents);

    tffsSocket[noOfDrives] = "NOR";
    vol.serialNo = noOfDrives;
    noOfDrives++;

    /* fill in function pointers */
    vol.cardDetected        = rfaCardDetected;
    vol.VccOn               = rfaVccOn;
    vol.VccOff              = rfaVccOff;
#ifdef SOCKET_12_VOLTS
    vol.VppOn               = rfaVppOn;
    vol.VppOff              = rfaVppOff;
#endif
    vol.initSocket          = norSocketInit;
    vol.setWindow           = norSetWindow;
    vol.setMappingContext = rfaSetMappingContext;
    vol.getAndClearCardChangeIndicator = rfaGetAndClearCardChangeIndicator;
    vol.writeProtected      = rfaWriteProtected;

    return (flOK);
}
```

nandRegister 函数完成 NandFlash Socket 层实现的驱动函数向 TrueFFS 上层的注册，该函数实现非常类似于 norRegister，我们一并给出其实现代码，注意 nandRegister 函数定义在 nandTffs.c 文件中。

在我们的实现中，NorFlash 和 NandFlash 各自对应的 Socket 实现中很多函数都是共同的，只是 initSocket 和 setWindows 函数实现上各自不同，通常都是如此，因为对于其他驱动函数，基于实际情况，基本上所有的 Flash 设备 Socket 层驱动都可以使用同一套函数。

```
//定义在 nandTffs.c 文件中。
LOCAL FLStatus nandRegister (void)
{
    FLSocket vol = flSocketOf (noOfDrives);

    if (noOfDrives >= DRIVES) //DRIVES=5
        return (flTooManyComponents);

    tffsSocket[noOfDrives] = "NAND";
    vol.serialNo = noOfDrives;
    noOfDrives++;

    /* fill in function pointers */
    vol.cardDetected        = rfaCardDetected;
```

```
    vol.VccOn                   = rfaVccOn;
    vol.VccOff                  = rfaVccOff;
#ifdef SOCKET_12_VOLTS
    vol.VppOn                   = rfaVppOn;
    vol.VppOff                  = rfaVppOff;
#endif
    vol.initSocket          =   nandSocketInit;
    vol.setWindow           =   nandSetWindow;
    vol.setMappingContext   =   rfaSetMappingContext;
    vol.getAndClearCardChangeIndicator = rfaGetAndClearCardChangeIndicator;
    vol.writeProtected      =   rfaWriteProtected;

    return (flOK);
}
```

TrueFFS 内核实现维护一个 FLSocket 结构数组，对每种 Socket 层实现维护一个 FLSocket 结构，该结构定义在 h/tffs/flsocket.h 内核头文件中。该结构的作用如同 BLK_DEV，DEV_HDR 等结构，是内核提供的，由驱动初始化的传递信息给内核的一种“媒介”。此处 FLSocket 就起着将 Socket 层实现信息提供给 TrueFFS 上层的作用，这些信息包括底层 Flash 设备的基地址，容量，Socket 层实现的驱动函数。

内核维护的 FLSocket 结构数组共有 5 个元素，即内核最多支持 5 种不同的 Socket 层实现，可同时驱动 5 个不同的 Flash 设备。在我们的例子中，我们使用了 FLSocket 结构数组中的前两个元素。flSocketOf 函数即返回 FLSocket 结构数组中的对应元素，其中驱动号作为元素索引。所以在 MTD 层初始化中，其将根据驱动号查询到对应的 Socket 层 FLSocket 结构，完成二者之间的衔接。
Socket 层实现将围绕 FLSocket 结构的定义来完成，所以下面我们将从 FLSocket 结构出发，探讨 Socket 层需要实现的具体的驱动函数。FLSocket 结构定义如下。

```
/*h/tffs/flsocket.h*/
/* Socket state variables */
typedef struct tSocket FLSocket;   /* Forward definition */

struct tSocket {
   unsigned    volNo;          /* Volume no. of socket */
   unsigned    serialNo;       /* Serial no. of socket on controller */

   FLBoolean  cardChanged;  /* need media change notification */

   int            VccUsers;      /* No. of current VCC users */
   int            VppUsers;      /* No. of current VPP users */

   PowerState  VccState;/* Actual VCC state */
   PowerState  VppState;/* Actual VPP state */
```

```
    FLBoolean  remapped;            /* set to TRUE whenever the socket window is moved */

    void           (*powerOnCallback)(void *flash); /* Notification routine for Vcc on */
    void *  flash;              /* Flash object for callback */

    struct {                                /* Window state */
        unsigned int baseAddress;      /* Physical base as a 4K page */
        unsigned int currentPage;      /* Our current window page mapping */
        void FAR0 *    base;             /* Pointer to window base */
        long int    size;                 /* Window size (must by power of 2) */
        unsigned  speed;            /* in nsec. */
        unsigned        busWidth;       /* 8 or 16 bits */
    } window;

    FLBoolean (*cardDetected)(FLSocket vol);
    void (*VccOn)(FLSocket vol);
    void (*VccOff)(FLSocket vol);

#ifdef SOCKET_12_VOLTS

    FLStatus (*VppOn)(FLSocket vol);
    void (*VppOff)(FLSocket vol);

#endif      /* SOCKET_12_VOLTS */


    FLStatus (*initSocket)(FLSocket vol);
    void (*setWindow)(FLSocket vol);
    void (*setMappingContext)(FLSocket vol, unsigned page);
    FLBoolean (*getAndClearCardChangeIndicator)(FLSocket vol);
    FLBoolean (*writeProtected)(FLSocket vol);

#ifdef EXIT
    void (*freeSocket)(FLSocket vol);
#endif
};
```

volNo：
serialNo：
这两个字段表示 Socket 层驱动号，其中 volNo 被内核使用，而 serialNo 则被 Socket 层驱动自身使用。这个驱动号将用于索引内核维护的 FLSocket 结构数组，返回对应的 Socket 层 FLSocket 结构。serialNo 初始化工作在 xxxRegister 中完成。

cardChanged：

底层 Flash 设备移除状态是否改变，对于 Flash 设备而言，这个字段总是 FALSE，表示不可移除。这个成员由 Socket 层和内核共同使用。这个字段的赋值在 cardDetected 函数中完成。

VccUsers：
VppUsers：
内核使用，用以维护当前对 Flash 设备进行电源控制的使用者数量。例如如果由多个用户打开了电源，那么只当最后一个用户做关闭电源操作后，电源才能真正被关闭，否则 Flash 设备电源依然是开启的。这个字段只供内核使用。

remapped：
该字段表示 Flash 设备地址空间是否进行了改变。通常该字段总是为 FALSE。该字段一般由 Socket 层使用，用以通知上层 Flash 设备地址空间是否发生改变。

powerOnCallBack：
Flash 设备电源开启时回调函数，这是一个钩子函数，用以让 Socket 层驱动在内核开启 Flash 设备电源做一些辅助性的工作。开启电源，就表示将要使用 Flash 设备，而由于硬盘的特殊设计方式，要使用 Flash 设备，可能需要设置某个 GPIO 管脚功能，此时就可以在此处完成。该字段由 Socket 层实现进行初始化，内核负责调用。在我们的示例中，我们不使用这个字段。

flash：
这是一个指向 FLFLASH 结构的指针，FLFLASH 结构是 MTD 层表示结构，这个成员以及 FLFLASH 结构中的 socket 成员用于 Socket 层和 MTD 层之间的相互引用。注意 Socket 层和 MTD 层共同组成底层 Flash 驱动。该字段由内核进行初始化。

window：
这个成员本身是一个结构，用以表示底层 Flash 设备的基地址，容量，读写速率，数据线宽度等信息。其中基地址是以 4K 大小为单位，所以上文中我们将基地址做右移 12 处理。读写速率以 ns 为单位，这个用于 TrueFFS 内部定时器，因为 Flash 擦除时间较长，内核可能基于此做相关优化处理以提供系统整体性能。容量必须是 2 的幂次方。注意：内核通常将 Flash 设备在系统地址空间中占据的空间称为一个窗口，窗口实际上表示了 Flash 设备基地址和大小。
在我们的示例中，我们将在 xxxSetWindow 函数中，完成各自 Flash 设备窗口的初始化。

cardDetected：
函数指针，该函数用以探测底层 Flash 设备是否存在：0 表示不存在，其他表示存在。通常 Flash 设备是物理焊接到板子上，不会如同 U 盘等存储介质或 PCI 设备可插拔，故 cardDetected 函数实现都是返回 OK。无需做实际探测工作。该函数由 Socket 层实现。
在我们的示例中，对于 NorFlash 和 NandFlash，我们采用相同的实现，所以该函数定义在 sysTffs.c 文件中，代码如下：

```
/*******************************************************************************
* sysTffs.c
* rfaCardDetected - detect if a card is present (inserted)
```

```
* This routine detects if a card is present (inserted).
*
* RETURNS: TRUE, or FALSE if the card is not present.
*/
LOCAL FLBoolean rfaCardDetected (FLSocket vol)
{
    return (TRUE);
}
```

VccOn：
VccOff：
Flash 设备供电电源开启和关闭函数。通常板子上 Flash 设备是持续供电的，即 Flash 设备一直处于供电状态，没有对电源的控制，所以这两个函数一般实现为空。这两个函数由 Socket 层实现。
有时，我们将 VccOn 实现为使能 Flash 设备，即 Flash 设备可被写入，而将 VccOff 实现为保护 Flash 设备，即此时 Flash 被锁定（锁定是通过控制 Flash 的 writeProtect 管脚电平完成的），不可进行写入操作。
对于我们的示例，NorFlash 和 NandFlash 将采用相同的实现，这两个函数具体实现代码如下。

```
/*******************************************************************************
* sysTffs.c
* rfaVccOn - turn on Vcc (3.3/5 Volts)
* This routine turns on Vcc (3.3/5 Volts).    Vcc must be known to be good
* on exit.
*
* RETURNS: N/A
*/
LOCAL void rfaVccOn (FLSocket vol)
{
    rfaWriteEnable ();
}

/*******************************************************************************
* sysTffs.c
* rfaWriteEnable - enable write access to the RFA
* This routine enables write access to the RFA.
*
* RETURNS: N/A
*/
LOCAL void    rfaWriteEnable (void)
{
   //设置平台特定（GPIO）功能管脚，使能 Flash 设备写入操作。
```

```
}

/*******************************************************************************
* sysTffs.c
* rfaVccOff - turn off Vcc (3.3/5 Volts)
* This routine turns off Vcc (3.3/5 Volts).
*
* RETURNS: N/A
*/
LOCAL void rfaVccOff (FLSocket vol)
{
      rfaWriteProtect ();
}

/*******************************************************************************
* sysTffs.c
* rfaWriteProtect - disable write access to the RFA
* This routine disables write access to the RFA.
*
* RETURNS: N/A
*/
LOCAL void rfaWriteProtect (void)
{
    //设置平台特定（GPIO）管脚，禁止 Flash 设备写入操作。
}
```

VppOn：
VppOff：
有些 Flash 设备擦除操作需要 12V 电压，这两个函数完成的功能即开启和关闭 12V 供电电源。同样的道理，通常板子上 Flash 设备电源都是持续供应的，所以这两个函数一般也是实现为空。对于我们的示例，NorFlash 和 NandFlash 将采用相同的实现，如下。

```
#ifdef SOCKET_12_VOLTS
/*******************************************************************************
* sysTffs.c
* rfaVppOn - turns on Vpp (12 Volts)
* This routine turns on Vpp (12 Volts). Vpp must be known to be good on exit.
*
* RETURNS: flOK always
*/
LOCAL FLStatus rfaVppOn(FLSocket vol /* pointer identifying drive */)
{
      return (flOK);
}
```

```
/*******************************************************************************
* sysTffs.c
* rfaVppOff - turns off Vpp (12 Volts)
* This routine turns off Vpp (12 Volts).
*
* RETURNS: N/A
*/
LOCAL void rfaVppOff (FLSocket vol   /* pointer identifying drive */)
{
      //nothing
}
#endif     /* SOCKET_12_VOLTS */
```

initSocket：

Socket 层 Flash 设备初始化函数，该函数可以完成使能 Flash 设备，设置 Flash 设备为就绪（即将 cardChanged 设置为 FALSE），调用 setWindow 设置 Flash 设备大小（注意在我们的 xxxRegister 函数中只进行了 Flash 基地址的初始化）等工作。该函数返回 0 表示成功，否则失败。

对于我们的示例，NorFlash 对应实现函数为 norSocketInit，定义在 norTffs.c 文件中，NandFlash 对应实现函数为 nandSocketInit，定义在 nandTffs.c 文件中，这两个函数实现如下。

```
/*******************************************************************************
* norTffs.c
* norSocketInit - perform all necessary initializations of the socket
* This routine performs all necessary initializations of the socket.
*
* RETURNS: flOK always
*/
LOCAL FLStatus norSocketInit(FLSocket vol)
{
      rfaWriteEnable ();
      vol.cardChanged = FALSE;

      /* enable memory window and map it at address 0 */
      norSetWindow (&vol);

      return (flOK);
}

/*******************************************************************************
* nandTffs.c
* nandSocketInit - perform all necessary initializations of the socket
* This routine performs all necessary initializations of the socket.
```

```
*
* RETURNS: flOK always
*/
LOCAL FLStatus nandSocketInit(FLSocket vol)
{
    rfaWriteEnable ();
    vol.cardChanged = FALSE;

    /* enable memory window and map it at address 0 */
    nandSetWindow (&vol);

    return (flOK);
}
```

setWindow：

Flash 设备窗口设置函数。通常这个函数完成 Flash 设备基地址，大小，速率和数据线宽的设置，即初始化 FLSocket 中 window 字段。

对于我们的示例，NorFlash 对应 norSetWindow 函数，定义在 norTffs.c 文件中，NandFlash 对应 nandSetWindow 函数，定义在 nandTffs.c 文件中，这两个函数的实现如下。

```
/*******************************************************************************
* norTffs.c
* norSetWindow - set current window attributes, Base address, size, etc
* This routine sets current window hardware attributes: Base address, size,
* speed and bus width.    The requested settings are given in the 'vol.window'
* structure.    If it is not possible to set the window size requested in
* 'vol.window.size', the window size should be set to a larger value,
* if possible. In any case, 'vol.window.size' should contain the
* actual window size (in 4 KB units) on exit.
*
* RETURNS: N/A
*/
LOCAL void norSetWindow (FLSocket vol)
{
    /* Physical base as a 4K page */
    vol.window.baseAddress = NOR_FLASH_BASE_ADRS >> 12;
    flSetWindowSize (&vol, NOR_FLASH_SIZE >> 12); //window size as a 4K page.
    flSetWindowBusWidth(pVol->socket, 16);/* use 16-bits */
    flSetWindowSpeed(pVol->socket, 120);    /* 120 nsec. */
}

/*******************************************************************************
* nandTffs.c
```

```
* nandSetWindow - set current window attributes, Base address, size, etc
* This routine sets current window hardware attributes: Base address, size,
* speed and bus width.    The requested settings are given in the 'vol.window'
* structure.    If it is not possible to set the window size requested in
* 'vol.window.size', the window size should be set to a larger value,
* if possible. In any case, 'vol.window.size' should contain the
* actual window size (in 4 KB units) on exit.
*
* RETURNS: N/A
*/
LOCAL void nandSetWindow (FLSocket vol)
{
      /* Physical base as a 4K page */
      vol.window.baseAddress = NAND_FLASH_BASE_ADRS >> 12;
      flSetWindowSize (&vol, NAND_FLASH_SIZE >> 12); //window size as a 4K page.
      flSetWindowBusWidth(pVol->socket, 16);/* use 16-bits */
      flSetWindowSpeed(pVol->socket, 120);    /* 120 nsec. */
}
```

setMappingContext：

窗口映射寄存器设置函数，在实际应用中，我们通常将这个 Flash 设备映射到系统地址空间中，而不是限于地址空间不够用，每次只能映射一部分，所以不需要使用这个函数，我们实现了一个空函数来初始化该字段。

在我们的示例中，NorFlash 和 NandFlash 采用相同的实现，如下。

```
/*******************************************************************************
* sysTffs.c
* rfaSetMappingContext - sets the window mapping register to a card address
* This routine sets the window mapping register to a card address.
* The window should be set to the value of 'vol.window.currentPage',
* which is the card address divided by 4 KB. An address over 128MB,
* (page over 32K) specifies an attribute-space address. On entry to this
* routine vol.window.currentPage is the page already mapped into the window.
* (In otherwords the page that was mapped by the last call to this routine.)
*
* The page to map is guaranteed to be on a full window-size boundary.
*
* RETURNS: N/A
*/
LOCAL void rfaSetMappingContext
      (
      FLSocket vol,        /* pointer identifying drive */
      unsigned page        /* page to be mapped */
      )
```

```
{
    //nothing, but works OK.
}
```

getAndClearCardChangeIndicator：
Flash 设备存在状态查询函数。该函数被上层调用用以检测底层 Flash 设备是否存在。基于实际使用环境，Flash 设备是物理焊接到板上的，即始终是存在的，cardChanged 字段始终为 FALSE，故该函数实现将直接返回内核所需的状态，无需任何实质性工作。
我们的示例中，NorFlash 和 NandFlash 将采用相同的实现，代码如下。

```
/*******************************************************************************
* sysTffs.c
* rfaGetAndClearCardChangeIndicator - return the hardware card-change indicator
* This routine returns the hardware card-change indicator and clears it if set.
*
* RETURNS: FALSE, or TRUE if the card has been changed
*/

LOCAL FLBoolean rfaGetAndClearCardChangeIndicator (FLSocket vol)
{
    return (FALSE);
}
```

writeProtected：
该函数用以被内核调用查询 Flash 的写保护状态。Flash 设备都有一个 writeProtect 管脚，用以锁定 Flash 设备，禁止写入操作。可以使用这种功能，也可以不使用，在我们的示例中，将不使用写保护功能，故实现为空，NorFlash 和 NandFlash 将采用相同的函数，代码如下。

```
/*******************************************************************************
* sysTffs.c
* rfaWriteProtected - return the write-protect state of the media
* This routine returns the write-protect state of the media
*
* RETURNS: FALSE, or TRUE if the card is write-protected
*/
LOCAL FLBoolean rfaWriteProtected (FLSocket vol)
{
    return (FALSE);
}
```

freeSocket：
这是 FLSocket 中定义的最后一个字段，用以释放 Socket 层分配的资源，这个函数将在卸载 Socket 层驱动时使用，不过实际中通常很少使用这个函数，因为 Flash 设备一般是作为支持

操作系统运行的文件系统使用的，在系统运行期间，将一直被使用，直到关闭系统。
在我们的示例中，将不使用该函数。

至此，我们完成 FLSocket 中需要 Socket 层实现的所有功能的代码实现，这些代码包括在三个文件中：（1）sysTffs.c 文件，该文件定义有 Socket 层共用函数，sysTffsInit 初始化函数，相关控制宏定义；（2）norTffs.c 文件，该文件定义有 NorFlash 特定 Socket 层函数实现；（3）nandTffs.c 文件，该文件定义有 NandFlash 特定 Socket 层函数实现。
在以上的介绍中，我们基本完成了所有的操作，还遗留一个控制宏的定义，这些宏用以控制 TrueFFS 映射层实现类型，内核 MTD 层相关实现。在我们的例子中，我们同时需要驱动 NorFlash 和 NandFlash，故需要在 TrueFFS 映射层同时包括对 NorFlash 和 NandFlash 的支持，即必须同时定义 INCLUDE_TL_SSFDC 以及 INCLUDE_TL_FTL。另外由于我们将自实现 Flash 对应的 MTD 层驱动（下一节内容），故不需要任何内核 MTD 层实现代码，所以必须 undef 所有内核提供的 MTD 层实现，这也是通过 undef 相关宏进行的。具体代码如下。

```
/*sysTffs.c*/
/* defines */
#undef    INCLUDE_MTD_I28F016        /* Intel: 28f016 */
#undef    INCLUDE_MTD_I28F008        /* Intel: 28f008 */
#undef    INCLUDE_MTD_AMD            /* AMD, Fujitsu: 29f0{40,80,16} 8bit */
#undef    INCLUDE_MTD_CDSN           /* Toshiba, Samsung: NAND, CDSN */
#undef    INCLUDE_MTD_DOC2           /* Toshiba, Samsung: NAND, DOC */
#undef    INCLUDE_MTD_CFISCS         /* CFI/SCS */
#undef    INCLUDE_MTD_WAMD           /* AMD, Fujitsu: 29f0{40,80,16} 16bit */
#define   INCLUDE_TL_FTL             /* FTL translation layer */
#define   INCLUDE_TL_SSFDC           /* SSFDC translation layer */
```

Socket 层实现小结
我们的示例中，同时对 NorFlash 和 NandFlash 设备进行驱动，故我们从大体上实现有两套 Socket 实现，基于实际应用环境，两套 Socket 中，大部分函数都是共用的。涉及 Socket 层实现文件有三个：sysTffs.c, norTffs.c, nandTffs.c，前文中我们基于 FLSocket 结构分别实现了每个文件中的相关函数，此处我们将各自文件代码集中在一起进行显示，便于读者进行查看和理解。
（1） sysTffs.c 文件实现

```
#include "copyright_wrs.h"
#include "vxWorks.h"
#include "config.h"
#include "tffs/flsocket.h"
#include "tffs/tffsDrv.h"

/* defines */
#undef    INCLUDE_MTD_I28F016        /* Intel: 28f016 */
#undef    INCLUDE_MTD_I28F008        /* Intel: 28f008 */
#undef    INCLUDE_MTD_AMD            /* AMD, Fujitsu: 29f0{40,80,16} 8bit */
```

```
#undef    INCLUDE_MTD_CDSN          /* Toshiba, Samsung: NAND, CDSN */
#undef    INCLUDE_MTD_DOC2          /* Toshiba, Samsung: NAND, DOC */
#undef    INCLUDE_MTD_CFISCS        /* CFI/SCS */
#undef    INCLUDE_MTD_WAMD          /* AMD, Fujitsu: 29f0{40,80,16} 16bit */
#define   INCLUDE_TL_FTL            /* FTL translation layer */
#define   INCLUDE_TL_SSFDC          /* SSFDC translation layer */

/* forward declarations */
LOCAL void        rfaWriteProtect (void);
LOCAL void        rfaWriteEnable (void);
LOCAL FLBoolean           rfaCardDetected (FLSocket vol);
LOCAL void        rfaVccOn (FLSocket vol);
LOCAL void        rfaVccOff (FLSocket vol);
#ifdef    SOCKET_12_VOLTS
LOCAL FLStatus        rfaVppOn (FLSocket vol);
LOCAL void        rfaVppOff (FLSocket vol);
#endif    /* SOCKET_12_VOLTS */
LOCAL FLBoolean           rfaGetAndClearCardChangeIndicator (FLSocket vol);
LOCAL FLBoolean           rfaWriteProtected (FLSocket vol);
LOCAL void        rfaSetMappingContext (FLSocket vol, unsigned page);

#include "norTffs.c"
#include "nandTffs.c"

/*******************************************************************************
* sysTffsInit - board-level initialization for TrueFFS
* This routine calls the socket registration routines for the socket component
* drivers that will be used with this BSP. The order of registration determines
* the logical drive number given to the drive associated with the socket.
*
* RETURNS: N/A
*/
LOCAL void sysTffsInit (void)
{
    norRegister ();
    nandRegister();
}

/*******************************************************************************
* sysTffs.c
* rfaCardDetected - detect if a card is present (inserted)
* This routine detects if a card is present (inserted).
*
* RETURNS: TRUE, or FALSE if the card is not present.
```

```
*/
LOCAL FLBoolean rfaCardDetected (FLSocket vol)
{
    return (TRUE);
}

/*******************************************************************************
* sysTffs.c
* rfaVccOn - turn on Vcc (3.3/5 Volts)
* This routine turns on Vcc (3.3/5 Volts).    Vcc must be known to be good
* on exit.
*
* RETURNS: N/A
*/
LOCAL void rfaVccOn (FLSocket vol)
{
    rfaWriteEnable ();
}

/*******************************************************************************
* sysTffs.c
* rfaWriteEnable - enable write access to the RFA
* This routine enables write access to the RFA.
*
* RETURNS: N/A
*/
LOCAL void    rfaWriteEnable (void)
{
   //设置平台特定（GPIO）功能管脚，使能 Flash 设备写入操作。
}

/*******************************************************************************
* sysTffs.c
* rfaVccOff - turn off Vcc (3.3/5 Volts)
* This routine turns off Vcc (3.3/5 Volts).
*
* RETURNS: N/A
*/
LOCAL void rfaVccOff (FLSocket vol)
{
    rfaWriteProtect ();
}

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

```
* sysTffs.c
* rfaWriteProtect - disable write access to the RFA
* This routine disables write access to the RFA.
*
* RETURNS: N/A
*/
LOCAL void rfaWriteProtect (void)
{
    //设置平台特定（GPIO）管脚，禁止 Flash 设备写入操作。
}

#ifdef SOCKET_12_VOLTS
/*******************************************************************************
* sysTffs.c
* rfaVppOn - turns on Vpp (12 Volts)
* This routine turns on Vpp (12 Volts). Vpp must be known to be good on exit.
*
* RETURNS: flOK always
*/
LOCAL FLStatus rfaVppOn(FLSocket vol /* pointer identifying drive */)
{
      return (flOK);
}

/*******************************************************************************
* sysTffs.c
* rfaVppOff - turns off Vpp (12 Volts)
* This routine turns off Vpp (12 Volts).
*
* RETURNS: N/A
*/
LOCAL void rfaVppOff (FLSocket vol   /* pointer identifying drive */)
{
      //nothing
}
#endif     /* SOCKET_12_VOLTS */

/*******************************************************************************
* sysTffs.c
* rfaSetMappingContext - sets the window mapping register to a card address
* This routine sets the window mapping register to a card address.
* The window should be set to the value of 'vol.window.currentPage',
* which is the card address divided by 4 KB. An address over 128MB,
* (page over 32K) specifies an attribute-space address. On entry to this
```

```
* routine vol.window.currentPage is the page already mapped into the window.
* (In otherwords the page that was mapped by the last call to this routine.)
*
* The page to map is guaranteed to be on a full window-size boundary.
*
* RETURNS: N/A
*/
LOCAL void rfaSetMappingContext
    (
    FLSocket vol,       /* pointer identifying drive */
    unsigned page       /* page to be mapped */
    )
{
    //我们将整个 Flash 设备都进行了映射，无需使用该函数，故实现为空。
}

/*****************************************************************************
* sysTffs.c
* rfaGetAndClearCardChangeIndicator - return the hardware card-change indicator
* This routine returns the hardware card-change indicator and clears it if set.
*
* RETURNS: FALSE, or TRUE if the card has been changed
*/

LOCAL FLBoolean rfaGetAndClearCardChangeIndicator (FLSocket vol)
{
    return (FALSE);
}

/*****************************************************************************
* sysTffs.c
* rfaWriteProtected - return the write-protect state of the media
* This routine returns the write-protect state of the media
*
* RETURNS: FALSE, or TRUE if the card is write-protected
*/
LOCAL FLBoolean rfaWriteProtected (FLSocket vol)
{
    return (FALSE);
}
```

（2） norTffs.c 文件实现

```
#include "copyright_wrs.h"
```

```
#include "vxWorks.h"
#include "config.h"
#include "tffs/flsocket.h"
#include "tffs/tffsDrv.h"

/*******************************************************************************
* norRegister - registration routine for the NorFlash Socket Driver
* This routine populates the 'vol' structure for a logical drive with the
* socket component driver routines for the NorFlash. All
* socket routines are referenced through the 'vol' structure and never
* from here directly
*
* RETURNS: flOK, or flTooManyComponents if there're too many drives
*/
LOCAL FLStatus norRegister (void)
{
    //noOfDrives 是一个内核全局变量，用以表示当前已分配的驱动号数目。
    //如 noOfDrives=2，则表示当前已分配两个驱动号，即系统同时具有两套 Socket 层实现，
    //驱动两个底层 Flash 设备。
    FLSocket vol = flSocketOf (noOfDrives);

    if (noOfDrives >= DRIVES) //DRIVES=5
        return (flTooManyComponents);

    tffsSocket[noOfDrives] = "NOR";
    vol.serialNo = noOfDrives;
    noOfDrives++;

    /* fill in function pointers */
    vol.cardDetected          = rfaCardDetected;
    vol.VccOn                 = rfaVccOn;
    vol.VccOff                = rfaVccOff;
#ifdef SOCKET_12_VOLTS
    vol.VppOn                 = rfaVppOn;
    vol.VppOff                = rfaVppOff;
#endif
    vol.initSocket            = norSocketInit;
    vol.setWindow             = norSetWindow;
    vol.setMappingContext = rfaSetMappingContext;
    vol.getAndClearCardChangeIndicator = rfaGetAndClearCardChangeIndicator;
    vol.writeProtected        = rfaWriteProtected;

    return (flOK);
}
```

```
/*******************************************************************************
* norTffs.c
* norSocketInit - perform all necessary initializations of the socket
* This routine performs all necessary initializations of the socket.
*
* RETURNS: flOK always
*/
LOCAL FLStatus norSocketInit(FLSocket vol)
{
    rfaWriteEnable ();
    vol.cardChanged = FALSE;

    /* enable memory window and map it at address 0 */
    norSetWindow (&vol);

    return (flOK);
}

/*******************************************************************************
* norTffs.c
* norSetWindow - set current window attributes, Base address, size, etc
* This routine sets current window hardware attributes: Base address, size,
* speed and bus width.  The requested settings are given in the 'vol.window'
* structure.  If it is not possible to set the window size requested in
* 'vol.window.size', the window size should be set to a larger value,
* if possible. In any case, 'vol.window.size' should contain the
* actual window size (in 4 KB units) on exit.
*
* RETURNS: N/A
*/
LOCAL void norSetWindow (FLSocket vol)
{
    /* Physical base as a 4K page */
    vol.window.baseAddress = NOR_FLASH_BASE_ADRS >> 12;
    flSetWindowSize (&vol, NOR_FLASH_SIZE >> 12); //window size as a 4K page.
    flSetWindowBusWidth(pVol->socket, 16);/* use 16-bits */
    flSetWindowSpeed(pVol->socket, 120);  /* 120 nsec. */
}
```

（3） nandTffs.c 文件实现

```
#include "copyright_wrs.h"
#include "vxWorks.h"
```

```
#include "config.h"
#include "tffs/flsocket.h"
#include "tffs/tffsDrv.h"

/*******************************************************************************
* nandRegister - registration routine for the NandFlash Socket Driver
* This routine populates the 'vol' structure for a logical drive with the
* socket component driver routines for the NandFlash. All
* socket routines are referenced through the 'vol' structure and never
* from here directly
*
* RETURNS: flOK, or flTooManyComponents if there're too many drives
*/
LOCAL FLStatus nandRegister (void)
{
	FLSocket vol = flSocketOf (noOfDrives);

	if (noOfDrives >= DRIVES) //DRIVES=5
		return (flTooManyComponents);

	tffsSocket[noOfDrives] = "NAND";
	vol.serialNo = noOfDrives;
	noOfDrives++;

	/* fill in function pointers */
	vol.cardDetected		= rfaCardDetected;
	vol.VccOn			= rfaVccOn;
	vol.VccOff			= rfaVccOff;
#ifdef SOCKET_12_VOLTS
	vol.VppOn			= rfaVppOn;
	vol.VppOff			= rfaVppOff;
#endif
	vol.initSocket		=	nandSocketInit;
	vol.setWindow		=	nandSetWindow;
	vol.setMappingContext	=	rfaSetMappingContext;
	vol.getAndClearCardChangeIndicator = rfaGetAndClearCardChangeIndicator;
	vol.writeProtected	=	rfaWriteProtected;

	return (flOK);
}

/*******************************************************************************
* nandTffs.c
* nandSocketInit - perform all necessary initializations of the socket
```

```
* This routine performs all necessary initializations of the socket.
*
* RETURNS: flOK always
*/
LOCAL FLStatus nandSocketInit(FLSocket vol)
{
    rfaWriteEnable ();
    vol.cardChanged = FALSE;

    /* enable memory window and map it at address 0 */
    nandSetWindow (&vol);

    return (flOK);
}

/*******************************************************************************
* nandTffs.c
* nandSetWindow - set current window attributes, Base address, size, etc
* This routine sets current window hardware attributes: Base address, size,
* speed and bus width.    The requested settings are given in the 'vol.window'
* structure.    If it is not possible to set the window size requested in
* 'vol.window.size', the window size should be set to a larger value,
* if possible. In any case, 'vol.window.size' should contain the
* actual window size (in 4 KB units) on exit.
*
* RETURNS: N/A
*/
LOCAL void nandSetWindow (FLSocket vol)
{
    /* Physical base as a 4K page */
    vol.window.baseAddress = NAND_FLASH_BASE_ADRS >> 12;
    flSetWindowSize (&vol, NAND_FLASH_SIZE >> 12); //window size as a 4K page.
    flSetWindowBusWidth(pVol->socket, 16);/* use 16-bits */
    flSetWindowSpeed(pVol->socket, 120);    /* 120 nsec. */
}
```

9.10 Flash 驱动 MTD 层实现

MTD 层实现涉及到两个文件，这两个文件都位于 BSP 目录下：（1）tffsConfig.c 文件，该文件定义有一个重要的数组：mtdTable，该数组中包含了所有的 Flash 识别函数，系统将依据该数组对底层 Flash 设备进行识别，故 MTD 层实现所有函数后，必须将其设备识别函数添

加到 mtdTbale 数组中，供内核使用；注意 tffsConfig.c 文件名是事先约定好的，不可更改文件名；事实上除了 mtdTable 数组外，还有另一个主要的数组也定义在该文件中：tlTable，该数组将被 TrueFFS 映射层使用，根据底层 Flash 设备类型（Nor 或 Nand）调用不同的函数挂载和格式化设备，NorFlash 对应 moutFTL，formatFTL 函数，NandFlash 对应 mountSSFDC 和 formatSSFDC 函数；（2）tffsMtd.c 文件，该文件完成 MTD 层所有函数的实现，这些函数包括：设备识别函数，设备读写函数，设备擦除函数，设备地址映射函数；注意 tffsMtd.c 文件名是自定义的，开发人员可取任何有效的名称。

9.10.1 tffsConfig.c 文件实现

tffsConfig.c 文件定义有两种重要数组和一个 Socket 层初始化入口函数。这两个重要数组分别是 Flash 设备识别函数数组和 Flash 设备挂载和格式化函数数组，Socket 层初始化总入口函数为 flRegisterComponents，该函数将被 tffsDrv 调用对 Socket 层驱动进行初始化，flRegisterComponents 进一步调用 sysTffsInit 函数，这已在前一节中进行了介绍。故本小节主要对两个数组进行介绍。

Vxworks 在首次操作 Flash 设备之前必须探测 Flash 设备，这个 Flash 设备探测的工作由 MTD 层实现，MTD 层必须提供一个识别其驱动的 Flash 设备的函数，完成平台 Flash 的识别工作。Vxworks 提供了一个数组 mtdTable 将所有 Flash 设备识别函数集合在一起进行管理，内核需要对一个底层 Flash 设备进行识别时，其将直接从数组中第一个元素开始依次调用 Flash 设备识别函数，直到一个函数返回 OK，才停止对数组中余下函数的调用。故 MTD 层实现代码除了完成设备设别，设备读写，擦除，地址映射等函数的实现外，还必须更改 tffsConfig.c 文件中的 mtdTable 数组，将自己实现的设备设别函数添加进去，使得可被 TrueFFS 中间层调用，对自己驱动的 Flash 设备进行识别。mtdTable 数组定义如下。

```
/* externs */
#define INCLUDE_MTD_USR
#ifdef INCLUDE_MTD_USR
FLStatus    norFlashMTDIdentify (FLFlash vol);
FLStatus    nandFlashMTDIdentify (FLFlash vol);
#endif /* INCLUDE_MTD_USR */

/* globals */
MTDidentifyRoutine mtdTable[] =       /* MTD tables */
{
#ifdef    INCLUDE_MTD_CFIAMD
      cfiAmdIdentify,
#endif    /* INCLUDE_MTD_CFIAMD */

#ifdef      INCLUDE_MTD_CFISCS
      cfiscsIdentify,
#endif      /* INCLUDE_MTD_CFISCS */

#ifdef      INCLUDE_MTD_I28F016
      i28f016Identify,
```

```
#endif     /* INCLUDE_MTD_I28F016 */

#ifdef     INCLUDE_MTD_I28F008
      i28f008Identify,
#endif     /* INCLUDE_MTD_I28F008 */

#ifdef     INCLUDE_MTD_I28F008_BAJA
      i28f008BajaIdentify,
#endif     /* INCLUDE_MTD_I28F008_BAJA */

#ifdef     INCLUDE_MTD_AMD
      amdMTDIdentify,
#endif     /* INCLUDE_MTD_AMD */

#ifdef INCLUDE_MTD_USR
      norFlashMTDIdentify,
      nandFlashMTDIdentify,
#endif    /* INCLUDE_MTD_USR */

#ifdef     INCLUDE_MTD_WAMD
      flwAmdMTDIdentify,
#endif     /* INCLUDE_MTD_WAMD */
};
int noOfMTDs = NELEMENTS (mtdTable);/* number of MTDs */
```

TrueFFS 对该数组的使用代码示例如下。

```
/* Attempt all MTD's */
for (iMTD = 0; iMTD < noOfMTDs && status != flOK; iMTD++)
      status = mtdTable[iMTD](&vol);
```

mtdTable 和 noOfMTDs 都是全局变量，将直接被内核使用。以上 mtdTable 数组中每个元素都由一个宏定义进行控制，其中 INCLUDE_MTD_USR 宏是专门提供给用户使用的，用以添加自实现 MTD 层设备识别函数。在我们示例中，我们利用该宏添加了两个识别函数，分别对我们驱动的底层 NorFlash 和 NandFlash 进行识别。

注意：mtdTable 数组已定义的其他设备识别函数是内核提供的一些典型 Flash 设备的 MTD 层实现设备识别函数，当包括这些 MTD 层的内核实现时，这些识别函数也同时被包括在 mtdTable 数组中，提供给 TrueFFS 中内核上层组件（核心层和映射层）使用。我们在上一节 sysTffs.c 文件分析中，将所有的内核 MTD 层实现都删除（undef 语句），故预处理后的 mtdTable 数组将如下所示。此时将只有两个自实现的 Flash 设备识别函数存在数组中，分别对我们驱动的底层 NorFlash 和 NandFlash 进行识别。

```
MTDidentifyRoutine mtdTable[] =       /* MTD tables */
{
      norFlashMTDIdentify,
      nandFlashMTDIdentify,
}
```

tffsConfig.c 文件中定义的另一个重要数组就是 tlTable。该数组定义如下。

```
TLentry tlTable[] =                    /* TL tables */
{
#ifdef     INCLUDE_TL_FTL
#ifdef     FORMAT_VOLUME
      {mountFTL, formatFTL},
#else
      mountFTL,
#endif     /* FORMAT_VOLUME */
#endif     /* INCLUDE_TL_FTL */

#ifdef     INCLUDE_TL_SSFDC
#ifdef     FORMAT_VOLUME
      {mountSSFDC, formatSSFDC},
#else
      mountSSFDC,
#endif     /* FORMAT_VOLUME */
#endif     /* INCLUDE_TL_SSFDC */
};
int noOfTLs = NELEMENTS (tlTable);/* number of TLs */
```

tlTable 实际上也是一个函数指针数组，指向映射层实现的几个 Flash 设备挂载和格式化函数。前文中我们已经提及内核提供的映射层对 NorFlash 和 NandFlash 将具有不同的实现，这同时也表示对于 NorFlash 和 NandFlash 将具有不同的挂载和格式化实现，所以 tlTable 数组需要根据底层 Flash 设备的类型决定其要包括哪些挂载和格式化函数，在我们的示例中，我们需要同时驱动 NorFlash 和 NandFlash 设备，故两种挂载和格式化函数都要包括，经过预处理后的 tlTable 数组如下。

```
TLentry tlTable[] =                    /* TL tables */
{
      {mountFTL, formatFTL}, //NorFlash 挂载和格式化函数。
      {mountSSFDC, formatSSFDC}, //NandFlash 挂载和格式化函数。
};
```

内核对 tlTable 的访问代码如下示例。

```
for (iTL = 0; iTL < noOfTLs && status == flUnknownMedia; iTL++)
      status = tlTable[iTL].formatRoutine(&flash,formatParams);
```

以设备格式化为例，对于一个底层 Flash 设备，内核将依次调用 tlTable 数组中每个格式化函数对设备进行格式化。当底层是一个 NorFlash 时，formatFTL 将成功返回，此时 status=flOK，内核将停止对 tlTable 中其他格式化函数（此处指 formatSSFDC）的调用。如果底层是一个 NandFlash 设备时，formatFTL 将无法进行格式化，其返回 flUnknownMedia，此时内核将继续调用 formatSSFDC 函数，该函数是专门针对 NandFlash 设备的格式化函数，故其将返回 flOK，表示格式化成功。Flash 设备无外乎 Nor 和 Nand，故 formatFTL 和 formatSSFDC 两

个函数已经足以应对所有的 Flash 设备的格式化要求。

除了以上两个涉及 MTD 层实现的主要数组定义外，tffsConfig.c 文件还实现有 flRegisterComponents 函数，作为 Socket 层初始化过渡函数，该函数被 tffsDrv 调用，其进一步调用 sysTffsInit 完成 Socket 层驱动的初始化工作。flRegisterComponents 函数实现代码如下。

```
/*****************************************************************************
* tffsConfig.c
* flRegisterComponents - register MTD and translation layer components for use
* This routine registers MTD and translation layer components for use.
* This function is called by FLite once only, at initialization of the
* FLite system.
*
* RETURNS: N/A
*/
void flRegisterComponents (void)
{
    sysTffsInit (); /*sysTffs.c*/
}
```

tffsConfig.c 文件中还定义 TrueFFS 中间层某些信息显示函数，以及一个通过预留 Flash 空间存放 Vxworks Image 的 Image 写入函数，实际上 Image 的写入不需要经过 Flash 驱动这一层次，我们可以直接使用开发平台提供的 Flash 写入工具将 Image 写入 Flash 中，而后将这部分空间“切掉”，只将余下的部分作为 TrueFFS 管理的空间，如此比在 TrueFFS 预留一段空间要简便的多，而且不易出错。对于信息显示函数（tffsShow）以及 Image 写入函数（tffsBootImagePut）此处不作讨论，感兴趣读者可查看 Tornado 开发环境提供的 BSP 下 tffsConfig.c 源文件。

9.10.2 tffsMtd.c 文件实现

在 tffsConfig.c 文件中定义的 mtdTable 中，我们添加了两个 Flash 设备函数：norFlashMTDIdentify，nandFlashMTDIdentify。这两个函数由各自 Flash 驱动的 MTD 层实现。Flash 设备驱动的 MTD 层实现代码定义在诸如 tffsXXX.c 之类的文件，对于我们的示例，我们可以定义一个 tffsNorMTD.c 实现 NorFlash 的 MTD 层实现，一个 tffsNandMTD.c 实现 NandFlash 的 MTD 层实现，不过为了便于比较和节省篇幅，我们将 NorFlash 和 NandFlash 的 MTD 层都实现在一个文件中-tffsMTD.c 文件。这个文件将完成 NorFlash 和 NandFlash 的设备识别，设备读写，设备擦除，地址映射函数的具体实现。

9.10.2.1 Flash 设备识别函数实现

虽然被称为设备识别函数，但是其完成的功能并不是去识别一个底层 Flash 设备。通常在嵌入式系统下，我们并不需要编写一个“放之四海而皆准”的 Flash 驱动，换句话说，在进行函数实现时，我们已经确知被驱动的 Flash 各项参数，所谓识别函数只是从内核调用该函数的时机来说的，设备识别函数本质上就是 Flash MTD 层驱动的初始化和注册函数。该函数中完成所有 MTD 层所需资源的分配并通过 MTD 层专用结构 FLFlash 将 MTD 层实现的驱动函数提供给 TrueFFS 核心层和映射层使用，或者说将 MTD 层实现的驱动函数向 TrueFFS 其他层进行注册。正如 Socket 层实现围绕着 FLSocket 结构，MTD 层实现将围绕着 FLFlash 结构完成。FLFlash 结构定义在 h/tffs/flflash.h 内核头文件中，如下所示。

```
typedef unsigned short FlashType;       /* JEDEC id */
typedef struct tFlash FLFlash;          /* Forward definition */
struct tFlash {
   FlashType   type;          /* Flash device type (JEDEC id) */
   long int    erasableBlockSize;
   long int    chipSize;      /* chip size */
   int         noOfChips;         /* no of chips in array */
   int         interleaving;
   unsigned    flags;
   void *  mtdVars;
   FLSocket *  socket;            /* Socket of this drive */

   void FAR0*(*map)(FLFlash *, CardAddress, int);
   FLStatus    (*read)(FLFlash *, CardAddress, void FAR1 *, int, int);
   FLStatus    (*write)(FLFlash *, CardAddress, const void FAR1 *, int, int);
   FLStatus    (*erase)(FLFlash *, int, int);

   void    (*setPowerOnCallback)(FLFlash *);
};
```

type：
Flash 设备类型：JEDEC ID，即生产商 ID 和设备 ID，嵌入式平台下通常无需使用这个字段，直接初始化为 1 即可。

erasableBlockSize：
Flash 擦除块的大小，通常为 128KB（对于小块结构的 NandFlash，则为 16KB）。
chipSize：
Flash 设备单个芯片的大小。

noOfChips：
使用多个 Flash 设备（以增加数据线宽）时的芯片数。

interleaving：

芯片交织数。例如平台上共有 4 个同类型 Flash 设备，每个线宽为 8-bit，使用时每两两构成一组，形成 16-bit 线宽的“单个”Flash 设备被使用，那么此时 chipSize 是物理上单个芯片（此时共有 4 个芯片）的容量大小，noOfChips 为 4，而 interleaving 则为 2.
通常而言，平台将只有一片 Flash 芯片，故 noOfChips 为 1，interleaving 也为 1，而 chipSize 就是整个 Flash 芯片的容量大小。我们的示例中将假设单片 Flash 设备，数据线宽为 16-bit。

flags：
存储 Flash 的相关选项。这些选项被映射层或 MTD 层使用用以控制某些操作行为。

mtdVars：
MTD 层使用以指向一些自定义数据。

socket：
指向对应 Socket 层 FLSocket 结构，MTD 层和对应 Socket 层共同构成 Flash 底层驱动，故两个层次之间互用结构指针指向各自层次使用的结构便于相互间的访问。FLSocket 结构中也有一个 flash 成员指向 MTD 层中使用的 FLFlash 结构。

map：
Flash 设备地址映射函数。这个函数的实现的功能是将片上地址（Card Address）转换为 Flash 在全局地址空间中的地址。注意 TrueFFS 中核心层和映射层在调用 MTD 层实现函数时都是使用片上地址进行调用的，而在实际操作 Flash 设备时，必须先将其转换为全局地址，map 指向的函数就完成这个工作。该函数实现较为简单，只需对片上地址加上一个 Flash 在全局地址空间中的基地址即可。

read，write，erase：
Flash 设备读写，擦除函数。

setPowerOnCallback：
Flash 开启电源回调函数，这个函数的功能如同 FLSocket 结构对应字段 powerOnCallback 的功能，让 MTD 层在电源开启时做一些其认为必要的工作。该字段一般不被使用。

从 FLFlash 结构定义来看，MTD 层必须实现 map，read，write，erase 函数。而设备识别函数根据底层 Flash 设备参数以及使用 MTD 层实现的函数完成 FLFlash 结构中所有字段的初始化工作，从而完成 MTD 层实现向 TrueFFS 核心层和映射层的注册。

（1） norFlashMTDIdentify 函数实现

```
/*******************************************************************************
* tffsMtd.c
* norFlashMTDIdentify - MTD identify routine for our NorFlash
* RETURNS: FLStatus
*/

FLStatus norFlashMTDIdentify (FLFlash* pVol)
```

```
{
    if(pVol->socket.serialNo!=0){
        return flUnknownMedia;
    }
    pVol->type = 0x1;
    pVol->erasableBlockSize = (NOR_FLASH_ERASE_SIZE);
    pVol->chipSize = NOR_FLASH_SIZE;
    pVol->noOfChips = 1;
    pVol->interleaving = 1;
    pVol->flags |= SUSPEND_FOR_WRITE;
    pVol->write = norWrite;
    pVol->erase = norErase;
    pVol->map = norMap;

    return(flOK);
}
```

注意 norFlashMTDIdentify 开始处的 if 语句，我们通过 FLFlash 设备获取其对应的 FLSocket 结构，通过检查 FLSocket 结构中 serialNo 表示的驱动号，来确定当前需要识别的底层 Flash 设备是 NorFlash 还是 NandFlash。注意 norFlashMTDIdentify 函数将被 tffsDevCreate 或者 tffsDevFormat 函数调用，而 tffsDevFormat 和 tffsDevCreate 函数被调用时都传入了一个驱动号，这个驱动号在 sysTffsInit 函数中被分配，注意前文中，我们首先调用 norRegister 函数，后调用 nandRegister，故 NorFlash 对应的驱动号为 0，NandFlash 对应的驱动号为 1. norFlashMTDIdentify 开始处的 if 语句正是检查驱动号是否对应，从而完成 MTD 层和对应 Socket 层的衔接。
注意：内核在调用 norFlashMTDIdentify 函数前，已经根据驱动号获得对应的 FLSocket 结构并用以初始化 FLFlash 中的 socket 字段。

在 norFlashMTDIdentify 函数具体实现中，我们基本完成对传入的 FLFlash 结构中所有字段的初始化工作，只有一个关键字段没有进行初始化：read 函数。因为 NorFlash 提供类似于 SRAM 方式的接口，故对于 NorFlash 的数据读取操作，使用通常的 memcpy 函数即可。此处我们没有对 read 字段进行初始化，内核将使用默认的读取函数，即 tffscpy（h/tffs/flsystem.h），也即 memcpy。
NorFlash 的写入操作类似于 NandFlash，在写入之前必须进行擦除操作，故需要提供 write 函数实现。

（2） nandFlashMTDIdentify 函数实现

```
/*****************************************************************************
* tffsMtd.c
* nandFlashMTDIdentify - MTD identify routine for our NandFlash
* RETURNS: FLStatus
*/

FLStatus nandFlashMTDIdentify (FLFlash* pVol)
```

```
{
    if(pVol->socket.serialNo!=1){
        return flUnknownMedia;
    }
    pVol->type = 0x1;
    pVol->erasableBlockSize = (NAND_FLASH_ERASE_SIZE);
    pVol->chipSize = NAND_FLASH_SIZE;
    pVol->noOfChips = 1;
    pVol->interleaving = 1;
    pVol->flags |= SUSPEND_FOR_WRITE;
    pVol->read = nandRead;
    pVol->write = nandWrite;
    pVol->erase = nandErase;
    pVol->map = nandMap;

    return(flOK);
}
```

由于 NandFlash 接口数据线复用，故必须提供所有函数的实现，包括 read 函数实现。其他代码的含义如同 norFlashMTDIdentify 函数。

9.10.2.2 Flash 设备读函数实现

由于 NorFlash 提供类似于 SRAM 的接口，故 NorFlash 的读取函数直接使用内存读取函数 memcpy 即可，无需特别实现，在上一节 norFlashMTDIdentify 函数，我们没有初始化 FLFlash 结构的 read 字段，内核会自动使用 memcpy 函数作为读取函数。而对于 NandFlash 设备则不然，其接口不提供独立的地址线，而是地址，数据，命令线复用，故必须提供专门的 NandFlash 读取函数。另外 NandFlash 的读写都是面向页面的，即每次只能以页面为单位进行读写。NandFlash 读取函数实现代码如下。

```
char tmpbuf[NAND_PAGE_SIZE];
FLStatus  nandRead(FLFlash *pVol, CardAddress address, void FAR1 *buffer,
                                                int length, int mode)
{
    Uint32 startPage;
    Uint32 pageCnt;
    Uint32 pageOft;
    int i;
    int nread;

    startPage=address/NAND_PAGE_SIZE;
    pageOft=address%NAND_PAGE_SIZE;
    pageCnt=(length+NAND_PAGE_SIZE-1)/NAND_PAGE_SIZE;
```

```
	nread=0;
	if(nandReadPage(startPage, tmpbuf, 1)>0){
		memcpy(buffer, tmpbuf+pageOft, NAND_PAGE_SIZE-pageOft);
	}
	else{
		return -1;
	}
	nread+=NAND_PAGE_SIZE-pageOft;
	if(nandReadPage(startPage+1, &buffer[nread], pageCnt-2)>0){
		nread+=(pageCnt-2)*NAND_PAGE_SIZE;
	}
	else{
		return nread;
	}
	if(nandReadPage(startPage+pageCnt-1, tmpbuf, 1)>0){
		memcpy(buffer+nread, tmpbuf, length-nread);
	}
	else{
		return nread;
	}
	return flOK;
}

Uint32 nandReadPage(Uint32 start_page, Uint32 *rx, Uint32 page_count) {
	Uint32 i;
	Uint32 page_addr;
	Uint32 retcode = 0;

	for ( i = start_page ; i < start_page + page_count ; i++ )
	{
		/* Fix for large page addressing */
		page_addr = (i * 2048 * 2);

		/* Read a single page */
		NAND_CMD( NAND_LO_PAGE );		// Read page
		NAND_PAGE_ADDR( page_addr );		// Page address

		/* Use Read confirm? */
		if ( 1 )
			NAND_CMD( NAND_READ_30H );	// Read Confirm

		if ( NAND_busywait( 0x00100000 ) )
			return -1;
```

```
        /* Read page in 16 bit mode */
        retcode += NAND_read16( (Uint32) rx, 0 );
    }

    return retcode;
}
```

由于 NandFlash 必须面向页面进行读取操作，故我们实现一个 nandReadPage 底层函数读取 Flash 页，而后根据页内偏移进行数据复制。此处我们不在给出更底层函数的实现。

9.10.2.3 Flash 设备写函数实现

Flash 设备写入操作有一个特点，即必须在写入新的数据之前对要写入的区域进行擦除。故设备写必须在写入数据之前调用设备块擦除函数将写入区域对应的块擦除干净，即将所有的位置 1.由于每次只是写入块中的一个区域，故还必须对块中其他区域的数据进行保存，即写入之前还需要一个读取操作，即调用 Flash 设备读函数先将对应块中内容读出来，在内存中进行修改，再将这个块擦除干净，而后将修改后的内容再写回去。读者可以看到，如果每次写入一个字节，所作的工作量有多大，正因如此，TrueFFS 核心层和映射层做了很多的优化，如累积写操作，尽量减少 Flash 块的擦除次数，增加 Flash 设备的使用寿命。

（1） norWrite 函数实现

```
FLStatus eraseBeforeWrite
    (
    FLFlash*              pVol,
    CardAddress           address,
    const void FAR1*    buffer,
    int                   length,
    FLBoolean             overwrite
    )
{
    volatile char* pFlash;
    char* pBuffer;
    STATUS rc = OK;
    BOOL doFree = FALSE;
    INT32 overwritelen;
    INT32 eraselen;
    int sector;
    int offset;

    sector = address / FS_FLASH_SECTOR_SIZE;
    offset = address % FS_FLASH_SECTOR_SIZE;
```

```
overwritelen = offset+length;
eraselen=overwritelen&(~(UINT32)(FS_FLASH_SECTOR_SIZE-1));

if(overwritelen!=eraselen)
{
	eraselen = (eraselen + FS_FLASH_SECTOR_SIZE);
}

pBuffer = (UINT32*) malloc(eraselen);

if (pBuffer == 0)
{
	printf("%s:fail to allocate memory!\n",__func__);
	return(flBadParameter);
}
doFree = TRUE;

/* Get a pointer to the flash sector */
pFlash = (volatile char*) pVol->map(pVol,sector * FS_FLASH_SECTOR_SIZE,
                                    FS_FLASH_SECTOR_SIZE);
/* Copy the sector from flash to memory */
memcpy(((UINT8*) pBuffer), (void*) pFlash, offset);

/* Overwrite the sector in memory */
memcpy(((UINT8*) pBuffer) + offset, (char*)buffer, length);
memcpy(((UINT8*) pBuffer) + offset+length,
		(void*) pFlash+ offset+length,eraselen-(offset+length));

/* Erase sector */
rc=norflash_erase( FS_FLASH_BASE_ADRS+
			sector * FS_FLASH_SECTOR_SIZE,eraselen );
if (rc != OK)
{
	printf("%s:erase failed.\n",__func__);
	free(pBuffer);
	return(rc);
}

rc = norflash_write( pBuffer, pFlash, eraselen);
if (doFree)
{
	free(pBuffer);
}
```

```
    return rc;

}

FLStatus norWrite
    (
    FLFlash*                pVol,
    CardAddress             address,
    const void FAR1*    buffer,
    int                         length,
    FLBoolean               overwrite
    )
{
    volatile char* pFlash;
    char* pBuffer;
    STATUS rc = OK;
    INT32 eraselen;


    if (flWriteProtected(vol.socket))
    {
        return(flWriteProtect);
    }
    if(length==0)
        return (flOK);

    pBuffer = (char*) buffer;
    pFlash = (volatile char*) pVol->map(pVol, address, length);
    eraselen = length;

    rc = norflash_write( pBuffer, pFlash, eraselen);
    if(rc!=OK)
    {
        if(overwrite){
            rc=eraseBeforeWrite(pVol,address,buffer,length,overwrite);
        }
    }

    return((rc == OK) ? flOK : flTimedOut);
}
```

norWrite 函数首先直接进行写入操作，如果对应块是干净的，那么写入操作就成功返回；如果对应块已有数据，则调用 eraseBeforeWrite 函数，先读取块中数据，再擦除块，再写入修改后数据。

norWrite 以及 eraseBeforeWrite 中调用的 norflash_erase，norflash_write 是底层 NorFlash 块擦除和写入实现函数，此处不再给出代码。

（2） nandWrite 函数实现

```
FLStatus nandWrite
    (
    FLFlash*              pVol,
    CardAddress           address,
    const void FAR1*      buffer,
    int                   length,
    FLBoolean             overwrite
    )
{
    …

    return flOK;
}
```

此处我们不再给出 nandWrite 实现的详细代码，一方面由于代码比较繁琐，另一方面代码很难具有代表性，也不易进行分析。对于 NandFlash，读者可以参考开源操作系统 Linux 下的相关驱动源代码，虽然与 Vxworks 下接口不同，但是底层 Flash 设备直接驱动代码都是通用的，例如 nandWrite 的实现代码，直接可以进行借鉴（只需对一些参数进行转换即可直接使用）。

9.10.2.4 Flash 设备块擦除函数实现

Flash 设备擦除是面向块进行的，即至少擦除一个块的内容。Flash 设备一个块的大小通常为 128KB。擦除操作将块内所有位设置为 1，因为 Flash 设备写入操作只能将 1 反转为 0，达到数据的写入目的，故在写入新的数据之前，如果对应区域已有数据，则必须首先将这个区域的数据全部擦除（即将所有位重新置 1），才能写入新的数据。Flash 设备每个块的擦除次数都是有限的，NorFlash 大概在 10 万次左右，而 NandFlash 则在 100 万次左右，所以对于 Flash 设备一般都要采用均衡算法，合理使用设备内每个块，不能偏好于某些块的使用，从而造成一些块过早的衰老，而另一些则基本未用的局面。

对于块擦除函数实现，我们只给出模板函数，不再具体区分 NorFlash 和 NandFlash，擦除操作相对比较简单，而且当前有大量代码可供借鉴，故不再详细讨论。块设备擦除模板函数 xxxErase 代码如下。

```
FLStatus xxxErase
(
    FLFlash vol,
    int firstBlock,
    int numOfBlocks
)
{
```

```
    volatile UINT32 * flashPtr;
    int iBlock;
    if (flWriteProtected(vol.socket))
        return flWriteProtected;
    for (iBlock = firstBlock; iBlock < iBlock + numOfBlocks; Iblock++)
    {
        flashPtr = vol.map (&vol, iBlock * vol.erasableBlockSize, 0);
        /*  进行擦除操作  */
        …
        /*  检查擦除操作是否成功  */
        …
        /*  如果擦除操作失败，则返回 flWriteFault */
        retrun flWriteFault;
    }
    return flOK;
}
```

注意 xxxErase 的参数指定了要擦除的起始块号以及要擦除的总块数。

9.10.2.5 Flash 设备地址映射函数实现

Flash 地址映射函数完成片上地址到全局地址的转换。所谓片上地址即以 0 为基地址计算的 Flash 地址，而全局地址即 Flash 在系统全局地址空间中的地址，二者之间相差一个偏移，这个偏移就是 Flash 设备在系统全局地址空间中的基地址。如下 xxxMap 函数实现代码所示。

```
void FAR0* xxxMap
    (
    FLFlash* pVol,
    CardAddress address,
    int length
    )
{
    UINT32 flashBaseAddr = (pVol->socket->window.baseAddress << 12);
    void FAR0* pFlash = (void FAR0*) (flashBaseAddr + address);
    return(pFlash);
}
```

对于我们的 NorFlash 以及 NandFlash 的地址映射函数都将采用如上 xxxMap 函数的实现代码。

至此我们完成 MTD 层所有驱动函数的实现，这些函数都是围绕 FLFlash 结构进行实现的，以提供给 TrueFFS 核心层和映射层所需的功能，对于 Flash 驱动而言，这主要涉及到设备读写，擦除和地址映射函数。此外 MTD 层实现的设备识别函数完成 MTD 层驱动函数的注册。

9.11 本章小结

Flash 设备是嵌入式平台下最常使用的一类设备，提供嵌入式操作系统文件系统支持以及作为操作系统映像本身的存储介质。本章首先简单的对 Flash 设备类型进行了介绍，而后从 Flash 设备驱动内核层次出发分析了构成 Flash 设备驱动的各中间层及其之间的关系。对于 Flash 设备驱动，Vxworks 操作系统在 CBIO 中间层下提供 TrueFFS 中间层专门对 Flash 设备进行管理。TrueFFS 中间层下直接就是底层 Flash 设备硬件，换句话说，Flash 设备驱动包括在 TrueFFS 中间层中，具体的由 TrueFFS 中 MTD 层和 Socket 层组成。这与 Vxworks 提供的其他内核中间层有些区别。此时 TrueFFS 中核心层和映射层由 Vxworks 内核实现，而 MTD 层和 Socket 层将由 Flash 驱动开发人员提供实现代码。当然 Vxworks 操作系统也提供了一些常见 Flash 设备的 MTD 层实现代码，这些代码的包括与否将由相关宏定义进行控制，而针对 Socket 层实现，Tornado 开发环境也提供了模板实现文件，这个模板文件只需做简单修改即可用作实际 Flash 设备的 Socket 层驱动实现代码。不过对于嵌入式平台而言，Vxworks 提供的 MTD 层实现大多无法满足需求，故通常需要自实现 MTD 层，本章较为详细了介绍了 MTD 层和 Socket 层的初始化过程以及实现接口，两个层次各自涉及的实现文件，并以同时驱动 NorFlash 和 NandFlash 的情况为例，介绍了如何在 TrueFFS 下实现多个 Flash 设备的驱动。

读者在阅读本章时，应着重掌握 Flash 驱动的组成以及涉及到的相关接口函数和数据结构，对于 Flash 底层操作代码实现可以参考当前许多开源系统提供的源代码，二者结合，应不难应对 Vxworks 下 Flash 设备驱动。

第十章 网络设备驱动

网络设备是一类非常特殊的设备，其没有普通文件接口，不属于 IO 子系统管理。网络设备用户层接口函数采用与其他设备（字符，块设备）完全不同的操作函数集合。在完成对字符设备和块设备驱动设计的介绍后，本章将着重介绍第三类设备-网络设备的驱动设计和实现。

作为驱动程序的共同点，网络设备驱动也需要向内核接口层注册驱动函数集合，从而使得内核可以调用网口驱动进行数据的收发。但是网络设备在数据的收发有其自身的特点，以块设备作为比较，当我们完成一个块设备的创建后，用户即可使用 open 函数打开这个块设备，并使用 read，write 对块设备进行读写操作。对于网络设备而言，首先网络设备不存在对应的设备节点，其不属于 IO 子系统管理，不向 IO 子系统注册驱动和设备，当一个网络设备驱动完成向其内核接口层的注册后，用户即可对网络设备进行操作。但是这种操作是间接的，因为没有设备节点，故网络设备不可以使用路径名之类的机制进行直接打开，当用户需要使用网络设备进行数据的收发时，其首先调用 socket 函数创建一个本地套接字通道，而后调用 connect 函数完成与远端套接字通道的对接，此后用户也可以调用 read，write 函数进行数据读取和写入，但是数据读取和写入的方式与普通块设备的读取和写入底层实现机制上差别较大，尤其是数据读取操作。对于块设备而言，数据的读取和写入都是被动的，即必须经过用户层的请求后才能触发，如果平时用户层没有任何块设备读写请求，那么这个块设备将一直处于空闲状态，在其上没有任何数据的交互行为，然而对于网络设备则不然，网络设备对外界数据的接收是异步的，即当用户层没有任何的数据读取请求时，其也在不断的从外界接收数据，缓存到内核专门分配的缓冲区中，实际上所有用户层将来读取的数据都是从这个内核缓冲区中获得的。对于数据的写入，用户写入的数据也是预先写入到内核缓冲区中，而后由内核决定在合适的时机将这些数据通过网络设备发送出去，这一点实际上与普通块设备类似。不过网络设备发送的数据并不都是用户层写入的数据，其同时也发送网络栈协议实现代码所写入的特殊数据，如 ICMP 应答报文数据，ARP 应答报文数据等等。

网络设备本质上与串口设备具有很大的相似性，二者本身并不作为数据存储介质（这一点不同于块设备），而仅仅是一个数据收发的接口，用于本地机器与外界的数据的交互。网络设备与串口设备最大的不同在于工作于网络设备之上的协议要比串口设备复杂的多，数据量和数据率也要大得多，这就要求网络设备驱动内核接口要更有效率，要更便于对大量数据的处理。另外一个不同点是串口设备一般用于点对点之间的数据通信，而网络设备则用于点阵列之间的通信，这就决定了串口设备收发的数据内部不需要任何协议的封装，两端直接传送纯数据即可，因为确知这些数据的来源；而网络设备则不然，网络设备收发的数据内部具有特定的格式，即纯数据被按特定格式进行了封装，在纯数据之外添加了很多用以确定数据来源和去向的标识数据，通常我们将这些标识数据称为头部。除了使用额外的数据确定数据来源和去向外，我们还使用其他一些列额外的数据用以确定网络数据收发的可靠性，以及更进一步确保网络数据的服务质量，这些都需要在发送的纯数据之外添加额外的数据。为了对所有这些额外添加的数据进行解析以及便于解析，需要一个称为协议栈的内核模块，这个模块专门解析这些额外添加的数据，网络设备本身并不对收发的数据进行任何更改，以上所述的这些用以确定数据来源和去向，确保数据可靠性和服务质量等等方面所添加的额外数据都是由被称为协议栈的内核模块自行添加的。而实际上之所以称为协议栈，是因为为了便于后期的扩展，我们将这些添加的额外的数据分成了多个层次，一层套一层，每层使用特定的头部数

据承担一定的功能，这样一个结构称为栈，而这些特定的头部数据则称为协议。因为数据是网络介质传递的，所以协议栈有时也称为网络栈。

网络栈的实现是一项非常复杂的工程，这是操作系统实现的一部分，对于我们的网络设备驱动而言，我们不涉及到这一块，网络设备驱动从协议栈接收一个已经封装的数据帧，而后只需控制网络设备原封不动的将这个帧发送到网络介质上即可。而对于网络介质本身的共用以及冲突解决都是由网络硬件设备自动进行处理，无需网络设备驱动负责。所以网络设备的功能就相对比较简单了：（1）其从网络栈接收一个数据帧，操作网络设备，将这个帧发送出去；（2）其从网络设备接收一个帧，通过调用网络栈提供的接口函数，将这个帧传递给网络栈。网络数据的接收是异步于用户请求的，网络设备一直都处于数据接收的状态，当其发现网络介质上一个帧标志为发向本地的，其就接收这个数据帧，给出中断，网络设备驱动中断响应函数从网络设备内部硬件缓冲区中读取这个帧，并调用网络栈提供的接口函数，将这个帧传递给网络栈处理，网络栈将对这个帧的头部进行解析，根据头部信息将这个帧中的纯数据缓存到特定内核区域，等待用户层读取。

对于用户数据的发送，首先也是经过网络栈，网络栈根据用户数据的目的地对这些用户写入的纯数据添加头部，完成所有头部的创建后，其调用网络设备注册的发送函数将这个数据帧传递给网络设备驱动，网络设备驱动中实现的这个发送函数操作网络设备，将这个帧发送到网络介质上即可，通常只需将数据复制到网络设备内部硬件缓冲区即可，至于发送的具体细节将由网络设备本身处理，无需驱动干预。由于网络设备发送速率有限且网络介质共同造成的速率浪费，当用户层写入数据较快时，将由网络栈负责对这些写入的用户数据进行缓存。但是内核缓冲区总是有限的，当内核缓冲区满后，用户的写入操作将返回错误（非阻塞），或者写入的任务被挂起（阻塞）。

类比于 IO 子系统，对于网络设备而言，我们可以将网络栈等效看做是网络设备的 IO 子系统，其管理着网络设备驱动。网络设备驱动初始化过程中将向网络栈注册其实现的相关函数，供网络栈调用进行数据的发送以及对网络设备的配置和控制。对于数据的接收，网络栈也提供一个接口函数直接供网络设备驱动进行调用，将接收的数据帧传递给网络栈。这种网络栈与底层网络设备直接的交互方式是早期 Vxworks 的基本工作方式，我们称之为 4.3BSD 网络驱动类型，这种方式下，底层驱动与网络栈的耦合性比较紧密，这一点对于底层驱动开发比较不利，尤其对于像 Vxworks 这种非开源操作系统而言。所以 Vxworks 在后来版本中（如 5.5）提供了另一种驱动开发类型，即基于可裁剪的增强型网络栈（SENS：Scalable Enhanced Networks Stack）实现的底层驱动开发类型，Wind River 将使用这个 SENS 而编写的网络设备驱动称为增强型网络驱动，即 END-Enhanced Network Driver。当前 Vxworks 下网络设备驱动通常都使用 END 接口，故本章将只对 END 网络设备驱动进行分析。END 网络设备驱动内核层次如下图 10-1 所示。

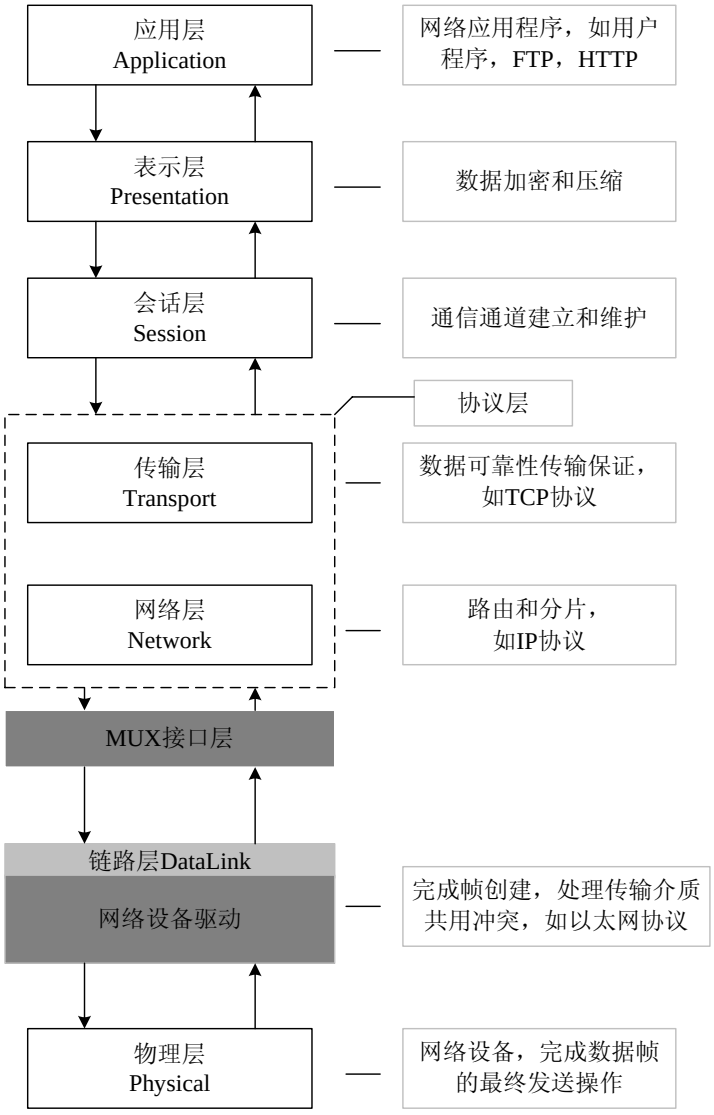


图 10- 1 END 网络设备内核驱动层次

为了简化网络设备驱动设计，Vxworks 内核提供了一个 MUX 中间层，这个中间层对上与网络栈核心实现交互，对下与底层网络设备驱动交互，管理着底层网络设备驱动。在图 1 中，我们将链路层置于了 MUX 中间层之下，同时在底层网络设备驱动之上，实际上，链路层实现与底层驱动更多的是一种平行关系，链路层实现主要完成数据帧的最后一个头部的添加工作，完成数据帧的最终创建，这个头部的添加工作可以由底层网络设备驱动自行完成，也可以直接使用内核链路层实现的函数，通常对于以太网链路层协议，我们将直接使用内核链路层实现函数。使用方式上即将内核链路层相关实现函数与底层网络设备驱动相关实现函数一起注册给 MUX 中间层使用。

基于 MUX 中间层，核心网络栈实现与底层网络设备驱动之间完全被隔离，底层网络设备不再直接与网络栈之间有任何关系，而是直接与 MUX 中间层进行通信，MUX 中间层也使得协议层实现更具有灵活性，如下图 10-2 所示为基于 MUX 中间层的上下层实现。

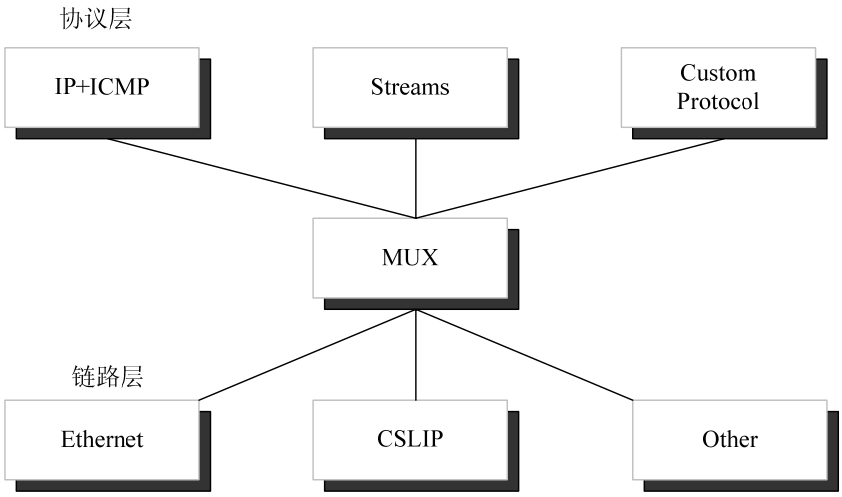


图 10- 2 基于 MUX 中间层的上下层实现

从图 10-2 可见，基于 MUX 中间层，上下层各自的实现具有了更大的灵活性。例如在协议层，我们可以实现一个自定义协议，注册到 MUX 中间层中，MUX 下所有链路层实现都可以使用我们自定义协议与用户层进行通信。

如下图 10-3 所示为协议层，MUX 中间层，END 网络设备驱动层（包括链路层）之间的详细接口。协议层必须实现如下接口函数，完成与 MUX 中间层的通信通道创建。

（1） stackShutDownRtn
（2） stackError
（3） stackRcvRtn
（4） stackTxRestartRtn

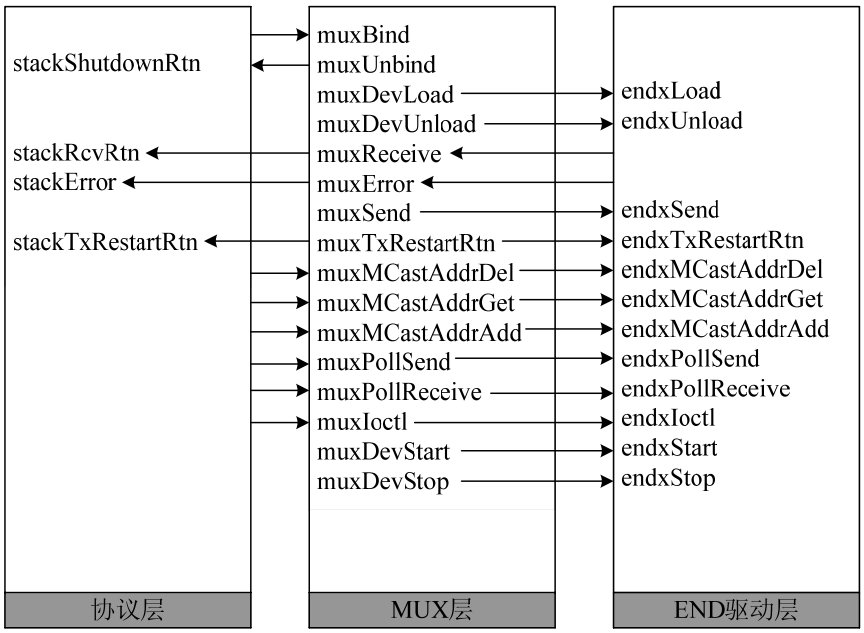


图 10- 3 协议层，MUX 层，END 驱动层之间详细接口关系

本书不涉及网络协议层的实现，下面我们主要关注与 End 驱动层需要实现的接口函数。按

照驱动程序的管理，内核需要使用底层驱动的函数时，会提供一个内核数据结构，这个结构中将包括所有内核层需要底层驱动递交的信息，底层驱动在完成这个内核结构的初始化后，调用内核接口函数将这个初始化后的内核结构提供给内核即可。对于 END 网络设备驱动也不例外。

10.1 内核数据结构

几乎所有的底层驱动都需要自定义一个数据结构用以保存底层硬件设备的关键信息，这个自定义数据结构通常都继承自一个内核结构，从而达到同一个结构既可以被内核使用也可以被驱动本身使用的目的。例如对于普通字符设备，内核提供 DEV_HDR 结构，对于串口设备，内核提供 SIO_CHAN 结构，对于块设备，内核提供 BLK_DEV 结构。对于网络设备，内核同样也提供了这样一个结构：END_OBJ 结构。该结构定义在 h/end.h 内核头文件中，如下所示。

```
/*
 * h/end.h
 * endObject - the basic end object that everyone derives from
 *
 * This data structure defines a device independant amount of state
 * that is maintained by all drivers/devices.
 * Each specific device (Lance, Intel, etc.) derives from this object
 * first and then incorporates it's own data structures after it.
 *
 */
// END_OBJ 结构成员在文献 vxworks bsp developer’s guide 中有相关解释。
typedef struct end_object
{
    NODE node;
    DEV_OBJ devObject;          /* Root of the device heirarchy. */
    STATUS (*receiveRtn) ();    /* Routine to call on reception. */
    struct net_protocol *outputFilter;    /* Optional output filter routine. */
    void* pOutputFilterSpare;     /* Output filter's spare pointer */
    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. */
    LIST multiList;             /* Head of the multicast address list */
    int    nMulti;              /* Number of elements in the list. */
    LIST protocols;             /* Protocol node list. */
    int snarfCount;                 /* Number of snarf protocols at head of list */
    NET_POOL_ID pNetPool;           /* Memory cookie used by MUX buffering. */
    M2_ID * pMib2Tbl;               /* RFC 2233 MIB objects */
} END_OBJ;
```

END_OBJ 中绝大部分字段的初始化由内核本身负责，不过有一个字段必须由底层驱动提供，这个字段就是 pFuncTable，该字段也是一个结构类型 net_funcs，该结构也定义在 h/end.h 头文件中，如下所示。

```
/*
 * h/end.h
 * NET_FUNCS - driver function table
 *
 * This is a table that is created, one per driver, to hold all the
 * function pointers for that driver.    In this way we can have only one
 * instance to this structer, but one pointer per netDevice structure.
 */
typedef struct net_funcs
{
    STATUS (*start) (END_OBJ*);              /* Driver's start func. */
    STATUS (*stop) (END_OBJ*);               /* Driver's stop func. */
    STATUS (*unload) (END_OBJ*);             /* Driver's unload func. */
    int (*ioctl) (END_OBJ*, int, caddr_t);       /* Driver's ioctl func. */
    STATUS (*send) (END_OBJ* , M_BLK_ID);          /* Driver's send func. */
    STATUS (*mCastAddrAdd) (END_OBJ*, char*); /* Driver's mcast add func. */
    STATUS (*mCastAddrDel) (END_OBJ*, char*); /* Driver's mcast delete func. */
    STATUS (*mCastAddrGet) (END_OBJ*, MULTI_TABLE*); /* Driver's mcast get fun. */
    STATUS (*pollSend) (END_OBJ*, M_BLK_ID);    /* Driver's polling send func. */
    STATUS (*pollRcv) (END_OBJ*, M_BLK_ID);     /* Driver's polling receive func. */
    M_BLK_ID (*formAddress) (M_BLK_ID, M_BLK_ID, M_BLK_ID, BOOL);
                                                /* Driver's addr formation func. */
    STATUS (*packetDataGet) (M_BLK_ID, LL_HDR_INFO *);
                                                 /* Driver's packet data get func. */
    STATUS (*addrGet) (M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID);
                                                /* Driver's packet addr get func. */
    int (*endBind) (void*, void*, void*, long type);
                        /* information exchange between */
                        /* network service and network driver */
} NET_FUNCS;
```

对于 END_OBJ 结构中定义的其他结构类型（如 DEV_OBJ，net_protocol）字段，读者可查阅 h/end.h 头文件。

END_OBJ 中 receiveRtn 函数指针将由协议层进行初始化，指向协议层实现的数据帧接收函数。底层驱动中调用 END_RCV_RTN_CALL 宏将数据帧传递给 MUX 层，MUX 层即调用 END_OBJ 结构中 receiveRtn 指向的函数将数据帧传递给协议层（网络栈）进行处理。

10.2 使用示例

本章讨论中我们将以 TMS320DM6446 开发板上 EMAC 网络接口驱动源码为例，介绍 Vxworks 下 END 网口驱动程序设计的各个方面。

EMAC(Ethernet Media Access Controller)是 DM6446 开发板上一个网络接口，支持 10Mbps 和 100Mbps 传输速率，其驱动一个物理层芯片与外界进行数据交互，如下图 10-4 所示。

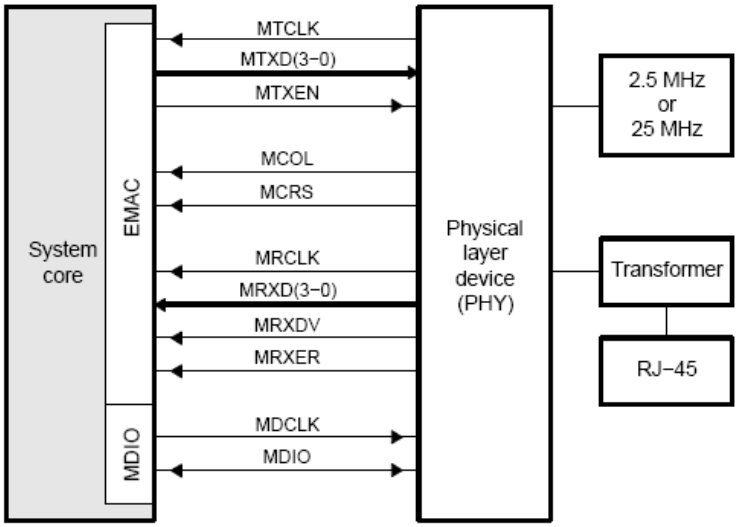


图 10- 4 DM6446 开发板 EMAC 接口

图 10-4 中 MDIO 接口采用两根信号线对物理层芯片进行配置，EMAC 接口则完成正常的网络数据的收发工作。DM6446 开发板（ARM926EJS）处理器并不直接控制 EMAC 和 MDIO 模块，而是通过 EMAC Control 模块驱动 EMAC 和 MDIO，实际上网口驱动就是驱动 EMAC Control 模块工作，完成正常的网络数据的收发工作。EMAC Control 模块与 EMAC 以及 MDIO 之间的关系如下图 10-5 所示。

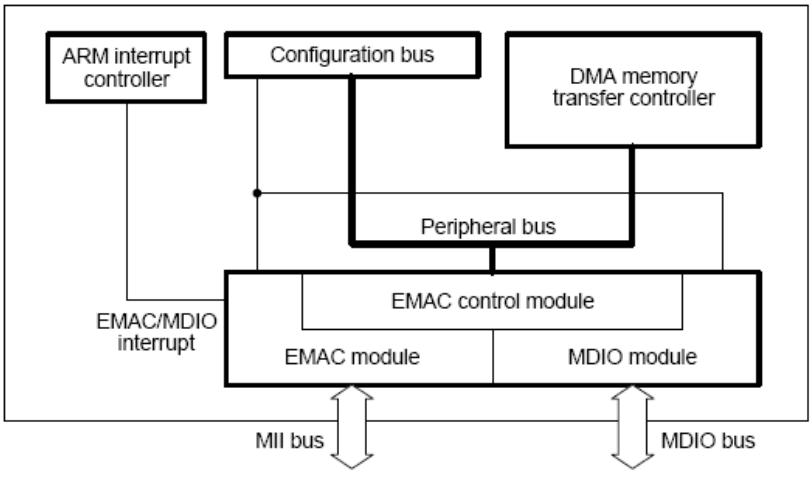


图 10- 5 EMAC Control 模块控制 EMAC 和 MDIO 两个模块工作

DM6446 开发板对 EMAC 接口专门提供有 8KB 内部硬件缓冲区用以保存网络数据。要驱动 EMAC Control 进行数据的收发，底层驱动程序必须按照特定格式将网络数据帧进行封装，需要使用一个称为描述符的数据结构。注意这个结构是网络接口硬件操作必须的，而非软件

上对数据帧的封装。这个配合硬件工作的描述符结构如下。

```
typedef struct _EMAC_Desc {
    struct _EMAC_Desc *pNext; /* Pointer to next descriptor in chain */
    char *pBuffer; /* Pointer to data buffer */
    volatile UINT32 pBufOffLen; /* Buffer Offset(MSW) and Length(LSW) */
    volatile UINT32 pktFlgLen; /* Packet Flags(MSW) and Length(LSW) */
} EMAC_DESCU;
```

pNext 字段用于将多个 EMAC_DESCU 结构串接起来，构成一个链表，从而 EMAC Control 模块可以自行遍历链表，对其中尚未处理的 EMAC_DESCU 结构进行处理；对于网络数据的发送，pBuffer 指向要发送的数据帧所在的缓冲区；pBufOffLen 表示 pBuffer 指向的缓冲区中数据帧的偏移，即数据帧的第一个字节并非是缓冲区的第一个字节，而是偏移了一段空间，同时这个字段还表示了 pBuffer 指向的缓冲区中有效数据字节数；一个数据帧可能由多个 EMAC_DESCU 结构表示，此时多个 EMAC_DESCU 指向的缓冲区中内容才构成一个实际要发送的数据帧，pktFlgLen 表示这个数据帧的总长度以及各个 EMAC_DESCU 之间的关系。如下图 10-6 所示。

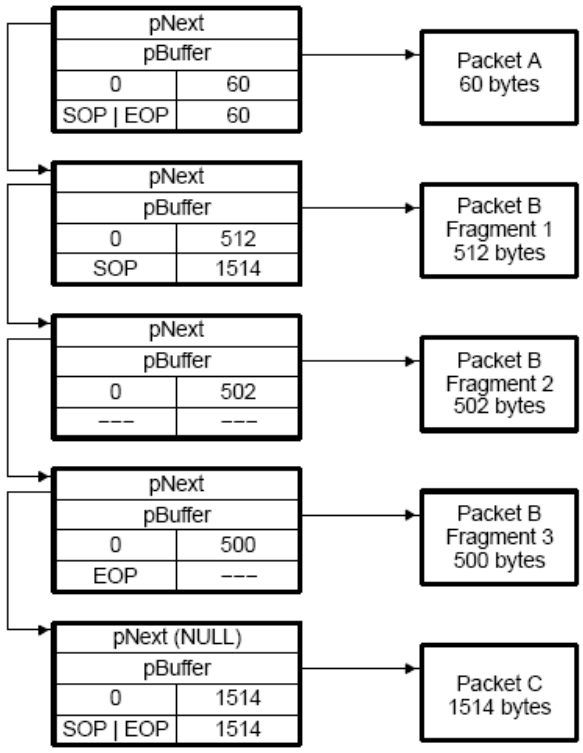


图 10- 6 EMAC_DESCU 结构关系示例

如上图 10-6 所示，当前链表中共有 5 个 EMAC_DESCU 结构，其中第一个结构表示一个独立的数据帧，这个帧的总长度为 60 个字节；第二，三，四个结构共同组成一个数据帧，这个数据帧的总长度为 1514 个字节，由第二个结构中 pktFlgLen 字段表示，各个结构中 pBufOffLen 字段表示各自结构中存储的部分帧长度，如第二个结构中存储了 512 个字节，第三个结构中存储了 502 个字节等，第二个结构中设置了 SOP 标志位标识帧的开始，而第四个结构中通过设置 EOP 标志位标识帧的结束。第五个结构也表示一个独立的帧，这个帧的总长度为 1514 个字节，注意第一，五个结构都是对应一个独立的帧，此时 SOP，EOP 标志位同时被设置。

在 DM6446 平台上，要利用 EMAC 接口收发网络报文，底层驱动必须初始化一系列这样的 EMAC_DESCU 结构，对于数据的发送而言，每个结构中 pBuffer 字段指向要发送的数据，其他字段指明了数据帧的长度；对于数据的接收而言，每个结构中 pBuffer 字段指向了数据缓冲区，其他字段则指明了缓冲区的长度。由于网络报文最大长度不会超过 1518 个字节，故我们将用于发送和接收的每个缓冲区的大小都分配为 1518 个字节，此时每个 EMAC_DESCU 结构必将对应一个独立的网络数据帧，不会存在多个 EMAC_DESCU 结构对应一个数据帧的情况。

具体的，针对数据发送的情况，我们将每个要发送的数据帧通过 EMAC_DESCU 结构维护成一个数组的形式，每当上层传递一个新的数据帧时，我们将取用数组中下一个可用的 EMAC_DESCU 结构封装这个新的数据帧。在最初的一个数据帧发送时，我们将数组第一个元素的地址写入到 EMAC Control 模块中一个特殊的寄存器中，启动数据帧的发送，EMAC Control 模块将自动按 EMAC_DESCU 结构进行解析，将 pBuffer 指向的数据帧通过 EMAC 和 PHY 接口发送出去。对于每个经过 EMAC Control 模块处理过的 EMAC_DESCU 结构，其都会将结构中 OWNER 标志位清除，表示该结构对应的数据帧已经处理完毕（即已发送），而底层驱动在插入一个新的数据帧时，会将对应 EMAC_DESCU 结构中的 OWNER 标志位设置为 1，而后在需要使用新的 EMAC_DESCU 结构时，通过检查 OWNER 标志位是否被清除来决定这个结构是否已得到处理，可被重新使用。

每个 EMAC_DESCU 结构占用 16 字节的空间，8KB 内部硬件缓冲区可以分配 512 个结构，在使用中我们将前 254 个结构用作数据接收，空出中间 4 个结构作为隔离，将后 254 个结构作为数据发送。EMAC_DESCU 结构本身并不保存数据，所以网络数据的保存需要从系统内存中分配空间，我们使用 cacheDmaMalloc 函数从系统内存中分配网络数据内存空间，在初始化过程中，对于数据接收 EMAC_DESCU 结构队列，我们初始化每个结构中的 pBuffer 字段，使其指向一个 1520 字节长的空闲缓冲区，并初始化结构中相关标志位（主要是设置 OWNER 标志位）。当 EMAC 接口从外界接收到网络数据帧时，其使用一个 EMAC_DESCU 结构，将数据帧拷贝到该结构 pBuffer 字段指向的内核缓冲区中，并清除 OWNER 标志位，发出一个中断。底层驱动中断处理程序检测到中断来自一个数据帧的接收时，其检查 EMAC_DESCU 结构接收队列，查看当前结构的 OWNER 标志位，确定该结构中是否已被 EMAC Control 使用，存储着网络数据帧，如果 OWNER 标志位被清除，中断处理程序将 pBuffer 指向的数据帧封装成网络栈需要的形式，通过调用 MUX 层接口函数将这个数据帧传递给上层网络栈进行处理，完成一个网络数据帧的完全接收，同时移动到队列中下一个 EMAC_DESCU 结构进行如上相同的处理。如下图 10-7，示例了 EMAC_DESCU 结构接收队列的使用情况。

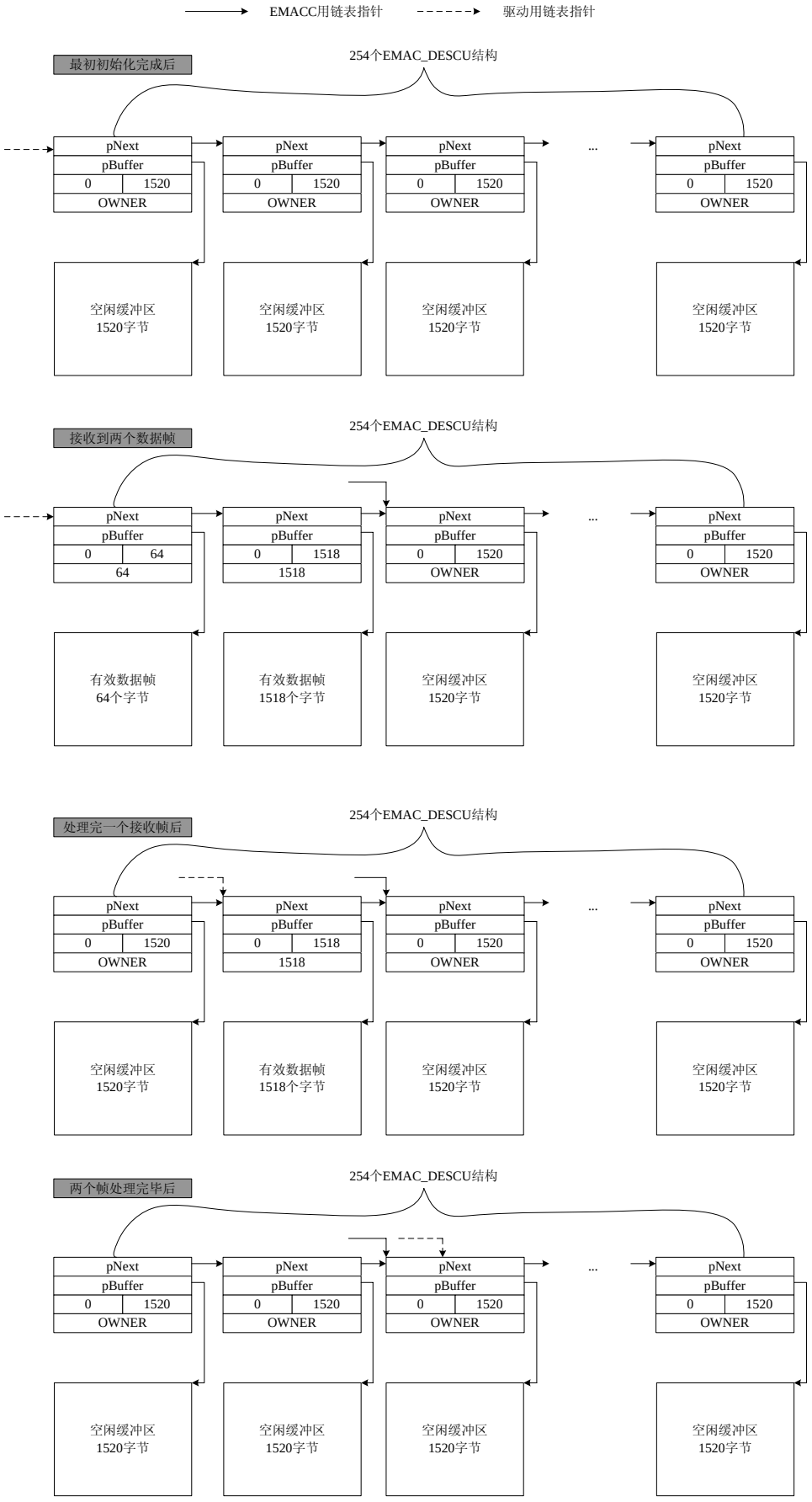


图 10- 7 EMAC_DESCU 结构接收队列使用示例

对于数据发送，我们也有 254 个 EMAC_DESCU 结构，每次 MUX 层传递一个新的数据帧给底层驱动时，其从 EMAC_DESCU 结构发送队列中，取出当前空闲的一个结构，将内核传递的数据帧拷贝到 pBuffer 指向的缓冲区中，同时设置 SOP，EOP，OWNER 标志位，即完成数据帧的发送行为，具体的底层数据发送将由 EMAC Control 模块自行处理，EMAC Control 模块完成一个结构的处理后，将清除 OWNER 标志位，以便对应的 EMAC_DESCU 结构能够被底层驱动下次使用，而底层驱动针对一个新的数据帧在设置完一个 EMAC_DESCU 结构后，将移动到发送队列中下一个 EMAC_DESCU 结构，准备从 MUX 层接收下一个要发送的数据帧，从而再次设置一个 EMAC_DESCU 结构。

由于每个缓冲区的大小为 1520 个字节，足够存储网络发送的最长帧，故每个 EMAC_DESCU 结构都最多对应一个数据帧，这虽然损失了一定的内存使用效率，但是极大的方便了底层驱动代码设计。对于内存资源有限的平台，可以减少 EMAC_DESC 结构的个数。对于 DM6446 平台，其系统 RAM 资源可达 256MB，故足以应对 508 个 EMAC_DESCU 所需的内存缓冲区空间。对于数据接收的情况，我们将所有的 EMAC_DESCU 结构串联成一个链表的形式，便于 EMAC Control 模块的连续接收数据帧。而对于数据帧发送的情况，由于每次 MUX 层只能传递一个数据帧给底层驱动进行处理，故我们将发送 EMAC_DESCU 结构组织成一个环形数组的形式，而非链表的形式，每次从数组中取一个 EMAC_DESCU 结构对当前帧进行封装，提供给 EMAC Control 模块进行发送。如下图 10-8 所示为 EMAC_DESCU 结构发送数组使用示意图。

有关 DM6446 平台下 EMAC 接口收发网络数据帧的更多细节请参考文献[7]。

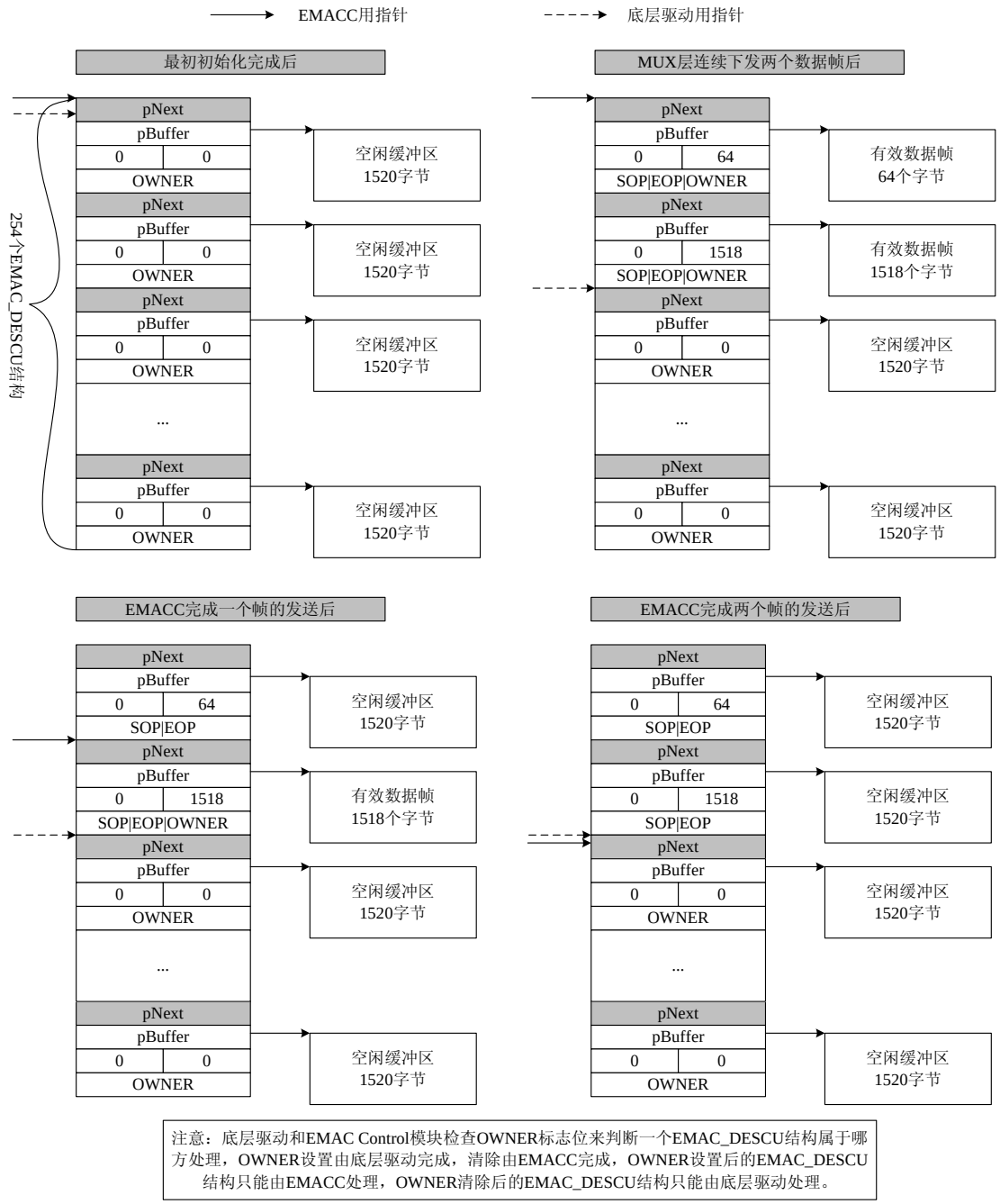


图 10- 8 EMAC_DESCU 结构发送数组使用示例

10.3 驱动自定义结构

对于我们的网口驱动，我们需要基于 END_OBJ 内核结构定义一个底层结构，供驱动本身以及内核使用，该结构定义如下所示。

```
typedef    EMAC_DESCU    ARM_END_RECV_FD;
typedef    EMAC_DESCU    ARM_END_TRAN_FD;
/*Driver Control Structure*/
```

```
typedef struct    _arm_end_device
{
    END_OBJ       end;       /* The class we inherit from.  必须是第一个成员*/
    SEM_ID        txLock;    /* I do not trust with the kernel lock,so define it myself.*/
    int           unit;          /* unit number */
    int           level;         /* interrupt vector */
    long          flags;             /* Our local flags. */
    CL_POOL_ID    pClPoolId;  /*数据缓冲区*/
    /*rcv data buffer pointer array, it is defined just for fast access*/
    /*接收缓冲区指针，254 个 EMAC_DESCU 结构对应 254 个缓冲区指针，方便访问*/
    UCHAR*            clRxPointBuf[ RX_FD_NUM ];
    /*EMAC_DESCU 结构接收链表，底层是一个数组*/
    volatile ARM_END_RECV_FD* pRecvBufDesc;
    /*驱动用当前 EMAC_DESCU 索引*/
    volatile UINT32           rxDescIdx;

    /*发送数据缓冲区首地址，每个 EMAC_DESCU 指向的缓冲区在内存中都是连续的*/
    UCHAR*            tranBuf;
    /*接收数据缓冲区首地址*/
    UCHAR*            recvBuf;
    UINT32        tranPkgNumber; /*initiated, but not used*/
    UINT32        tranPkgSize; /*EMAC_DESCU 结构中 pBuffer 指向的缓冲区大小 1520*/
    /*EMAC_DESCU 结构发送数组*/
    volatile ARM_END_TRAN_FD* pTxBufDesc;
    volatile UINT32 txDescPreSendIdx;
    volatile UINT32 txDescSendIdx;        /* point to desc sending */
    volatile UINT32 txDescWriteIdx;  /* point to desc writing */
    /*发送数组使用情况*/
    volatile UINT32 txBufStatus;
    /*netJobAdd 函数已被调用*/
    volatile BOOL     netJobScheduled;
    volatile BOOL     txHandling;
    volatile BOOL      firstCall;  /* first time calling armTransPacket? */
    /*相关参数定义*/
    UCHAR        enetAddr[6];   /* ethernet address */
    UCHAR        netSpeed;      /* 10 or 100 ,NOT USED*/
    UCHAR        duplexMode;        /* HDX = 0. FDX = 1 */
    UCHAR        autoNeg;       /* 1 = autoneg enabled,not used */
    ETHER_STATISTICS    statistics; /* Ethernet statistics counters */
    CACHE_FUNCS* pCacheFuncs;
    UINT32   mcastAddrCount;   /* Number of valid multicast addresses,NOT USED*/
} ARM_END_DEVICE;
```

10.4 驱动初始化：configNet.h

网络栈所有内核组件（包括网络设备 END 底层驱动）的初始化都在 usrNetInit 函数中完成，usrNetInit 函数在 Vxworks 启动过程中被 usrRoot 函数调用完成网络层的所有初始化工作。usrNetInit 函数定义在 target/src/config/usrNetwork.c 文件中，其中涉及到网络设备底层驱动的初始化代码如下所示。

```
    /* Add in mux ENDs. */
    for (count = 0, pDevTbl = endDevTbl; pDevTbl->endLoadFunc != END_TBL_END;
            pDevTbl++, count++)
    {
        /* Make sure that WDB has not already installed the device. */
        if (!pDevTbl->processed)
        {
            pCookie = muxDevLoad(pDevTbl->unit,
                                 pDevTbl->endLoadFunc,
                                 pDevTbl->endLoadString,
                                 pDevTbl->endLoan, pDevTbl->pBSP);
            if (pCookie == NULL)
            {
                printf("muxDevLoad failed for device entry %d!\n", count);
            }
            else
            {
                pDevTbl->processed = TRUE;
                if (muxDevStart(pCookie) == ERROR)
                {
                    printf("muxDevStart failed for device entry %d!\n", count);
                }
            }
        }//if (!pDevTbl->processed)
    }
```

总体上对于网络设备底层驱动的初始化包括两个步骤：（1）调用 muxDevLoad 函数载入底层驱动，此时底层驱动必须完成除中断外所有资源的分配；（2）调用 muxDevStart 正式启动网络设备工作，进行网络数据包的收发工作，此时底层驱动必须注册中断，使能底层设备，启动设备工作。

这两个函数在一个 for 循环语句中被调用，for 循环语句遍历 endDevTbl 数组中每个元素，对每个元素中注册的函数进行调用。网络设备底层驱动必须将其初始化函数添加到 endDevTbl 数组中。endDevTbl 数组类似于 Flash 驱动中的 mtdTable 数组。endDevTbl 数组定义在 configNet.h 文件中，该文件实现在 BSP 目录下。endDevTbl 数组中每个元素都是一个 END_TBL_ENTRY 结构，该结构定义在 h/end.h 内核头文件中，如下所示。

```
/* This is the structure that is used by the BSP to build up a table
```

```
 * of END devices to be started at boot time.
 */
typedef struct end_tbl_entry
{
    int unit;                                   /* This device's unit # */
    END_OBJ* (*endLoadFunc) (char*, void*);     /* The Load function. */
    char* endLoadString;                        /* The load string. */
    BOOL endLoan;                               /* Do we loan buffers? */
    void* pBSP;                                 /* BSP private */
    BOOL processed;                             /* Has this been processed? */
} END_TBL_ENTRY;
#define END_TBL_END NULL
```

unit：
设备号，被底层驱动使用用以区分多个设备的情况。

endLoadFunc：
底层驱动提供的初始化函数，在 muxDevLoad 函数中被调用，该函数返回一个经初始化后的 END_OBJ 结构，这个结构将被 MUX 层保存，用于此后对底层驱动函数的调用。

endLoadString：
驱动自定义参数字符串。这个字符串完全由底层驱动人员定制，其解析也是由 endLoadFunc 函数本身负责，故可以传递任何底层驱动需要的参数字符串。

endLoan：
该字段含义不明，底层驱动不使用该字段，内核目前也未使用。

pBsp：
底层驱动私有数据指针，一般不使用。

processed：
初始化标志位。起始值为 FALSE，当内核调用了 endLoadFunc 函数后，将被重新赋值为 TRUE。该字段用以避免对网络驱动进行多次初始化。网络设备驱动只需一次初始化即可。这一点从以上给出的代码中也可看出，当完成对 muxDevLoad 的成功调用后（即底层初始化函数成功返回后），该字段被设置为 TRUE。

网络设备驱动开发人员必须修改 configNet.h 文件中定义的 endDevTbl 数组，添加自实现驱动中的初始化函数到 endDevTbl 数组中，以便内核在初始化网络栈过程中对我们自实现的底层网络设备驱动进行初始化，包括对底层网络设备的初始化。
针对我们的示例，configNet.h 文件实现如下。

```
/*configNet.h*/
#ifndef _CONFIGNET_H
#define _CONFIGNET_H
```

```
#ifdef __cplusplus
extern "C" {
#endif

#ifdef INCLUDE_END /* ignore everything if NETWORK not included */

#include "vxWorks.h"
#include "end.h"

/* This template presumes the template END driver */

#define ARM_LOAD_FUNC       armLoad
#define ARM_BUFF_LOAN       0



/* From the light of armParse routine's accomplishment, this string is self-defined.
 * so i can define params meaningful to myself as the following format:
 *<unit>: <ether_speed>:<duplex>:<auto_nego>
 */
#define    ARM_LOAD_STRING "0:100:1:1"

IMPORT END_OBJ * ARM_LOAD_FUNC (char *,void *);

END_TBL_ENTRY endDevTbl [] ={
   { 0, ARM_LOAD_FUNC, ARM_LOAD_STRING, ARM_BUFF_LOAN, NULL,FALSE},
   { 0, END_TBL_END, NULL, 0, NULL,FALSE},
};

#endif /* INCLUDE_END */

#ifdef __cplusplus
}
#endif

#endif /* _CONFIGNET_H */
```

此处我们注册的初始化函数为 armLoad，自定义参数字符串为“100:1:1”。注意 processed 必须设置初始值为 FALSE，否则即便在 endDevTbl 数组添加了初始化函数，也不会被内核调用。另外自定义参数字符串中各个参数之间必须使用某个固定符号作为隔离符，通常使用冒号，各个参数的含义由底层驱动自定义，内核不使用该字符串，而是原封不动的作为参数传递给底层驱动初始化函数 armLoad。

armLoad 原型如下。

```
END_OBJ* armLoad(char* initString,void *pBSP);
```

其两个参数都是由底层驱动开发人员在操作 endDevTbl 数组时进行指定的，对应 END_TBL_ENTRY 结构中的 endLoadString 和 pBSP 字段。在以上的示例中，endLoadString 为“100:1:1”，而 pBSP 为 NULL。

底层驱动初始化函数 armLoad 的实现必须注意到一个特点，实际上 muxDevLoad 函数将两次调用底层驱动初始化函数 armLoad。第一次时传递的 initString 为一个空字符串（initString[0]=0），此时 armLoad 必须驱动名称拷贝到 initString 中就直接返回；第二次传递个 initString 才是驱动开发人员在 endDevTbl 数组中指定的参数字符串，此时 armLoad 才进行真正的初始化工作。这种两次调用工作方式是内核与底层驱动约定的，当以一个空字符串调用底层驱动初始化函数时，底层驱动必须返回驱动名称，内核将使用该字符串来标识这个驱动。

armLoad 作为底层驱动的初始化函数，将完成：（1）自定义结构的分配；（2）END_OBJ 结构的初始化；（3）底层驱动所需资源的分配；（4）网络设备硬件的初始化。

armLoad 具体实现代码如下。

```
END_OBJ* armLoad(char* initString,void *pBSP){
    ARM_END_DEVICE    *pDrvCtrl;
    if (initString == NULL){
        DRV_LOG (DRV_DEBUG_LOAD, "armLoad: NULL initStr\r\n",0,0,0,0,0,0);
        return NULL;
    }
    //当 initString 为一个空字符串时，直接返回驱动名称，此时并不进行真正的初始化。
    if (initString[0] == 0){
        strcpy(initString,DRV_NAME);
        return (END_OBJ *)NULL;
    }

    /* else initString is not blank, pass two ... */
    /* allocate the device structure */
    pDrvCtrl = (ARM_END_DEVICE *)malloc (sizeof (ARM_END_DEVICE));
    if (pDrvCtrl == NULL)
        goto outer;
    bzero((void *)pDrvCtrl,sizeof(ARM_END_DEVICE));

    /* parse the init string, filling in the device structure */
    if (armParse (pDrvCtrl, initString) == ERROR)
        goto outer;

    pDrvCtrl->level=INT_EMACINT;

    /* Ask the BSP for the ethernet address. */
    armGetMacAddr();/*get mac address out of default boot line parameter.*/
    SYS_ENET_ADDR_GET(pDrvCtrl);

    /* initialize the END and MIB2 parts of the structure. */
```

```
        strcpy (pDrvCtrl->end.devObject.name, DRV_NAME);
        pDrvCtrl->end.devObject.unit=pDrvCtrl->unit;
        strcpy (pDrvCtrl->end.devObject.description, "DM644X EMAC END Driver");

        /*
          * The M2 element must come from m2Lib.h
          * This arm is set up for a DIX type ethernet device.
          */
        if(END_OBJ_INIT(&(pDrvCtrl->end),(DEV_OBJ *)(pDrvCtrl),DRV_NAME,
                    pDrvCtrl->unit, &armFuncTable,"DM644X EMAC END Driver")!=OK){
            DRV_LOG(DRV_DEBUG_LOAD,"armLoad-endObjInit call failed!\n",1,2,3,4,5,6);
            goto outer;
        }

        if(END_MIB_INIT(&pDrvCtrl->end, M2_ifType_ethernet_csmacd,
                        &pDrvCtrl->enetAddr[0], 6, ETHERMTU, END_SPEED)!=OK){
            DRV_LOG(DRV_DEBUG_LOAD,"armLoad-mib2Init call failed!\n",1,2,3,4,5,6);
                    goto outer;
        }

        /* Perform memory allocation/distribution */
        if (armMemInit (pDrvCtrl) == ERROR)
                goto outer;

        /* reset and configure the device */
        armReset (pDrvCtrl);
        armConfig (pDrvCtrl);

        /* set the flags to indicate readiness */
        pDrvCtrl->flags=0;

        /* initialize my txlock.*/
        pDrvCtrl->txLock=semMCreate(SEM_Q_FIFO|SEM_DELETE_SAFE);
        if(pDrvCtrl->txLock==NULL){
                goto outer;
        }

        endObjFlagSet(&pDrvCtrl->end,
                        IFF_NOTRAILERS | IFF_BROADCAST | IFF_MULTICAST
#ifdef MIB2233RFC
        | END_MIB_2233
#endif
        );
```

```
    DRV_LOG (DRV_DEBUG_LOAD,
             "armLoad-Done loading ARM926EJS END Driver...\n", 1, 2, 3, 4, 5, 6);
    return (&pDrvCtrl->end);

outer:
    DRV_LOG (DRV_DEBUG_LOAD,
             "armLoad-Fail loading ARM926EJS END Driver...\n", 1, 2, 3, 4, 5, 6);

    if (pDrvCtrl != NULL){
        armMemUninit(pDrvCtrl);
        free ((char *)pDrvCtrl);
    }

    return NULL;
}
```

armLoad 代码对于 END_OBJ 结构的初始化代码具有通用性，对于任何网络设备的初始化函数都可以使用相同的 END_OBJ 结构初始化代码。其中最重要的一个语句如下：

```
    if(END_OBJ_INIT(&(pDrvCtrl->end),(DEV_OBJ *)(pDrvCtrl),DRV_NAME,
             pDrvCtrl->unit, &armFuncTable,"DM644X EMAC END Driver")!=OK){
        DRV_LOG(DRV_DEBUG_LOAD,"armLoad-endObjInit call failed!\n",1,2,3,4,5,6);
        goto outer;
    }
```

该语句将底层驱动函数集合安装到 END_OBJ 结构中，供 MUX 中间层使用。此处我们给出的驱动函数集合由 armFuncTable 表示，该变量作为一个全局变量定义在底层驱动中，如下所示。

```
/*
 * Declare our function table.    This is static across all device instances.
 */
LOCAL NET_FUNCS armFuncTable ={
    (STATUS (*) (END_OBJ*))armStart,        /* Function to start the device. */
    (STATUS (*) (END_OBJ*))armStop,         /* Function to stop the device. */
    (STATUS (*) (END_OBJ*))armUnload,           /* Unloading function for the driver. */
    (int (*) (END_OBJ*, int, caddr_t))armIoctl,     /* Ioctl function for the driver. */
    (STATUS (*) (END_OBJ* , M_BLK_ID))armSend,   /* Send function for the driver. */
    (STATUS (*) (END_OBJ*, char*))armMCastAdd,/* Multicast add function for the driver. */
    (STATUS (*) (END_OBJ*, char*))armMCastDel,/*Multicast delete function for thedriver. */
    /* Multicast retrieve function for the driver. */
    (STATUS (*) (END_OBJ*, MULTI_TABLE*))armMCastGet,
    (STATUS (*) (END_OBJ*, M_BLK_ID))armPollSend,           /* Polling send function */
    (STATUS (*) (END_OBJ*, M_BLK_ID))armPollRcv,      /* Polling receive function */
    endEtherAddressForm,        /* put address info into a NET_BUFFER */
    endEtherPacketDataGet,      /* get pointer to data in NET_BUFFER */
```

```
    endEtherPacketAddrGet           /* Get packet addresses. */
};
```

armFuncTable 是一个 NET_FUNCS 结构类型，该结构类型在 7.1 节中已经给出，定义了 END 驱动层必须实现的操作函数集合。armFuncTable 将被用以初始化 END_OBJ 结构中的 pFuncTable 字段。此后 MUX 中间层所有对底层驱动实现函数的调用都将通过该字段进行。

armLoad 函数最后返回一个 END_OBJ 结构，该结构中 pFuncTable 字段已经包含了底层驱动实现的所有 MUX 中间层要求的关键函数。
现在我们回过头来对 NET_FUNCS 结构中定义的函数指针进行分析，根据底层驱动和 MUX 中间层通信的需要来讨论这些函数的必要性以及是否还存在一些函数没有定义。在前文中我们已经给出 NET_FUNCS 结构的定义，此处我们对结构中每个成员做一一分析介绍。
NET_FUNCS 结构中的所有成员都是函数指针类型，指向实现相应功能的底层驱动函数。

start
设备启动函数，该函数将由 muxDevStart 函数调用，完成中断注册和使能，以及启动网络设备工作所需要的所有配置工作。我们的驱动示例中对应实现函数为 armStart。

stop
设备暂停函数，该函数将由 muxDevStop 函数调用，完成中断禁止，以及停止网络设备工作所需要完成的所有配置工作。我们的驱动示例中对应实现函数为 armStop。

unload
设备卸载函数，该函数必须取消 load 函数（注册在 endDevTbl 数组中）完成的所有工作：（1）释放 load 函数中分配的所有资源；（2）配置网络设备使其处于“安静”状态；（3）释放驱动自定义结构。我们的驱动示例中对应实现函数为 armUnload。

ioctl
设备控制，信息获取函数。如获取链路层头部长度，获取网络数据收发统计信息。我们的驱动示例中对应实现函数为 armIoctl。

send
网络数据帧发送函数，该函数被 MUX 层调用用以发送一个网络数据帧。我们的驱动示例中对应实现函数为 armSend。

mCastAddrAdd
mCastAddrDel
mCastAddrGet
多播地址添加，删除，获取函数。当用户需要接收某些多播数据帧，其将通过接口函数添加多播地址，这些地址最终必须通过底层驱动配置到网络设备的寄存器中，从而使得网络设备能够接收这些多播报文。通常网络设备只接收链路层地址是本地的数据报文，不接收多播报文，如果需要接收这些报文，则必须配置网络设备相关的寄存器。我们的驱动示例中对应的实现函数分别为 armMCastAdd，armMCastDel，armMCastGet。

pollSend
pollRcv
查询方式网络数据帧发送和接收函数。当使用网络通道作为系统调试通道时，这两个函数将被使用，在网络设备通常工作状态下，将使用中断方式进行数据帧的收发。我们的驱动示例中对应的实现函数分别为 armPollSend，armPollRcv。

formAddress
链路层头部创建函数。该函数将被 MUX 层调用用以创建数据帧中链路层头部。对于常用的以太网而言，内核提供了一个实现函数 endEtherAddressForm，底层驱动的网络设备如果工作在以太网下，则可以直接使用 endEtherAddressForm 函数。我们的驱动中直接使用内核提供的 endEtherAddressForm 函数。

packetDataGet
数据帧中链路层头部获取函数。该数据被调用用以对底层驱动接收的数据帧进行分析，从中分离出链路层头部，填充到 LL_HDR_INFO 结构参数中（该函数的第二个参数）。对于以太网，内核提供了 endEtherPacketDataGet 函数供底层驱动使用。我们的驱动示例中直接使用内核提供的 endEtherPacketDataGet 函数。

addrGet
链路层地址获取函数。该函数将通过解析网络数据帧，对传入的 pSrc，pDst，pESrc，pEDst 参数进行初始化，使其指向数据帧的不同位置， pSrc，pESrc 指向数据帧发送端 MAC 地址，pDst，pEDst 指向目的端 MAC 地址。对于以太网，内核提供 endEtherPacketAddrGet 函数供底层驱动使用。我们的驱动示例中直接使用内核提供的 endEtherPacketAddrGet 函数。

endBind
该字段未被使用，含义不明。在我们的驱动示例中没有对该字段进行初始化，这不会对网络设备的实际工作造成任何影响。

通过对 NET_FUNCS 结构的分析，我们可以看出 MUX 层可以使用该结构中字段指向的底层驱动函数启动一个网络设备，停止一个网络设备，卸载一个网络设备，使用网络设备发送报文，控制配置以及获取底层驱动和网络设备本身的信息等等，但是惟独缺了一个网络数据接收函数。事实上，经过本书前面几个章节的介绍，读者应不难理解，数据的发送通常是由上层主动调用底层驱动函数完成，但是数据的接收通常是上层提供一个数据接收接口函数，由底层驱动调用该函数，从而将数据传递给上层。例如串口驱动中，TTY 中间层通过回调的方式提供给底层串口驱动两个函数，供底层串口驱动将接收的数据存储到内核缓冲区中。对于网络设备驱动，同样是相同的机制：由 MUX 层提供数据接收接口函数，底层驱动调用该函数将接收的数据传递给上层，这个接收函数通过 END_OBJ 结构中的 receiveRtn 成员提供。例如对于我们的驱动示例，armRcv 完成具体的数据帧的接收工作，其完成上层所需的数据帧的数据结构封装后（将数据帧封装在一个 M_BLK_ID 结构中），将调用 END_RCV_RTN_CALL 宏将封装后的数据帧传递给上层，END_RCV_RTN_CALL 定义在 h/endLib.h 内核头文件中，如下所示。其中 pEnd 是一个指向 END_OBJ 结构的指针。

```
#define END_RCV_RTN_CALL(pEnd,pMblk) \
        if ((pEnd)->receiveRtn) \
```

```
{ \
    (pEnd)->receiveRtn ((pEnd), pMblk,NULL,NULL,NULL,NULL); \
}
```

可以看到，END_RCV_RTN_CALL 实际上是调用 END_OBJ 结构中 receiveRtn 字段指向的函数，该字段实际上在 muxDevLoad 函数中被初始化为 muxReceive 函数。即 muxDevLoad 函数一方面调用 armLoad 完成底层驱动初始化（其中包括对 END_OBJ 结构的部分初始化），另一方面也对 END_OBJ 结构中内核负责的部分字段进行初始化中，其中就包括对 receiveRtn 字段的初始化。

网络设备驱动通常采用中断驱动的方式进行数据帧的接收，每当网络设备接收到一个网络数据帧时，其触发一个中断，底层驱动中断函数将完成数据帧的封装，而后调用 muxReceive 函数将数据帧传递给 MUX 层，MUX 层进一步将数据帧传递给网络栈进行处理。也就是说 DEV_OBJ 结构中 receiveRtn 字段以及 pFuncTable 字段完成 MUX 层与底层网络设备驱动之间通信所需的所有信息。由底层驱动完成 pFuncTable 字段的初始化，由 MUX 层自身完成 receiveRtn 字段的初始化。当 MUX 层需要调用底层驱动函数时，其通过 END_OBJ 结构中的 pFuncTable 字段进行，而当底层驱动需要向上传递接收的数据时，其通过调用 END_OBJ 结构中的 receiveRtn 指向的函数。所以 END_OBJ 结构在 MUX 层和底层网络设备驱动之间起着承上启下的作用，MUX 层和底层驱动都维护着对 END_OBJ 结构的引用，其中底层驱动通过自定义结构引用，而 MUX 层通过保存底层驱动初始化函数（armLoad）返回的 END_OBJ 结构指针来引用，二者实际上使用的是相同的一段内存区域，只不过底层驱动在 END_OBJ 结构之后还保存着一些驱动特有的参数。

在 usrNetInit 函数中，通过 muxDevLoad 和 muxDevStart 函数的调用后，底层驱动和网络设备本身都进入了正常的工作状态，完成着网络数据帧的收发工作。用户可以使用 socket 接口函数与远端主机之间建立网络通信通道，进行数据的发送和接收。muxDevLoad 和 muxDevStart 分别调用底层驱动相关函数，在我们的驱动示例中，即调用 armLoad，armStart 函数完成以上的功能。armLoad 实现代码前文已经给出，此处我们给出 armStart 函数的实现代码。

```
/*******************************************************************************
* armStart - start the device
*
* This function calls BSP functions to connect interrupts and start the
* device running in interrupt mode.
*
* RETURNS: OK or ERROR
*/
LOCAL STATUS armStart(ARM_END_DEVICE * pDrvCtrl){
    STATUS result=OK;
    UINT32 val;

    //注册 armInit 函数为中断响应函数。
    result=intConnect ( (VOIDFUNCPTR *)INUM_TO_IVEC ( pDrvCtrl->level ),
            armInt, (UINT32) pDrvCtrl );
    if (result == ERROR){
```

```
            DRV_LOG(DRV_DEBUG_LOAD,
                    "armStart-cannot connect interrupt handler!\n",1,2,3,4,5,6);
            return ERROR;
        }
        DRV_LOG (DRV_DEBUG_LOAD, "armStart-Interrupt connected.\n", 1, 2, 3, 4, 5, 6);
        //使能中断
        intEnable(pDrvCtrl->level);

        armEnable(pDrvCtrl); //使能设备工作
        armFlgChange(pDrvCtrl,1);
        //设置网络设备状态为正常工作状态。
        END_FLAGS_SET (&pDrvCtrl->end, IFF_UP | IFF_RUNNING);
        //将 EMAC_DESCU 结构接收队列首地址写入网络设备特殊寄存器，启动数据帧接收。
        REG_WRITE(EMAC_CTRL_REG_RX1HDP,(UINT32)(pDrvCtrl->pRecvBufDesc));

        DRV_LOG(DRV_DEBUG_LOAD,"armStart-Done starting driver!\n",1,2,3,4,5,6);
        return (OK);
}
```

armStart 函数的实现思想非常简单：（1）注册和使能中断；（2）使能设备工作；（3）设置 END_OBJ 结构中相关标志位，标识设备已处于正常工作状态。注意对于标志位的设置必须完成，因为驱动上层（MUX 层或网络栈）将通过检查这些标志位来确定网络设备是否已经正常运作，如果不对这些标志位进行正确设置，那么在上层使用底层驱动时将发生问题。

10.5 后台处理：netJobAdd

对于一个嵌入式系统而言，通常需要较快的响应时间。如果一个中断处理函数执行的时间较长，那么其他设备的中断将得不到响应。故无论是 Vxworks 还是一般的通用操作系统（如 Linux）都要求尽量减少中断上下文的持续时间。所以通常中断函数被人为的分为两个部分：上半部分和下半部分。在中断上下文中只完成上半部分的工作，下半部分的工作将安排在一个任务上下文中进行。由于网络设备中断处理函数需要完成整个数据帧的复制和封装，执行时间较长，故 Vxworks 操作系统专门提供了一个后台任务完成中断的“下半部分”。这个后台任务就是 tNetTask。tNetTask 维护一个工作队列，只要工作队列非空，就执行工作队列的每个元素中包括的函数。所以对于一个网络设备中断响应函数而言，其完成中断所需的关键的上半部分工作，如检查中断来源，写相关硬件寄存器清除中断，而具体的数据帧的处理则通过在 tNetTask 维护的工作队列中插入一个元素，交由 tNetTask 在任务上下文中完成。注意：在任务上下文中这一点非常重要，由于执行在任务上下文，所以可以调用任何内核函数，可以等待资源（如内核缓冲区，下议）。而基于中断上下文则不可调用一切可引起阻塞睡眠的函数，这一点在进行数据帧的复制时可能会造成问题。

为了将中断的“下半部分”工作交由 tNetTask 任务完成，我们必须在中断的“上半部分”中将这个工作插入到 tNetTask 维护的工作队列中。Vxworks 提供 netJobAdd 函数完成这个工作向工作队列添加的功能。netJobAdd 函数调用原型如下。

```
STATUS netJobAdd
```

```
(
FUNCPTR routine,
int param1,
int param2,
int param3,
int param4,
int param5
);
```

参数 1 表示在任务上下文中被执行的“下半部分”入口函数。这个函数可以携带五个参数，param1~param5 就是 tNetTask 调用 routine 时传入的五个参数，完全由底层驱动在调用 netJobAdd 函数时指定。

netJobAdd 函数不可调用过于频繁，不可在每次中断的“上半部分”中都进行调用，因为 tNetTask 维护的内核工作队列“槽位”有限，同时只能安排有限个“下半部分”。而网络设备数据帧接收非常频繁，如果每次中断都调用 netJobAdd 函数，不可避免的会造成 tNetTask 维护的工作队列的溢出，此时将无法继续安排“下半部分”。
事实上，我们使用 netJobAdd 函数的目的在于进行网络数据帧的收发，而基于上文中的介绍，我们得知实际上我们将 EMAC_DESCU 结构维护成一个链表（接收）或者数组（发送）的形式，所以通过 EMAC_DESCU 中 OWNER 标志位的检查，我们可以在“下半部分”中进行连续的处理，而不是每次只处理一个 EMAC_DESCU 结构，所以只要“下半部分”按这种方式（连续处理方式）进行实现，而不是每次只处理一个数据帧，则只要有一个“下半部分”被安排，就可以完成数据帧接收的任务，而不用安排多个“下半部分”。所以在驱动自定义结构中，我们定义了一个 netJobScheduled 成员，该成员表示内核工作队列中是否已经安排了一个“下半部分”，如果 netJobScheduled 为 TRUE，则不用再次调用 netJobAdd 函数。我们将在“下半部分”函数最后退出之时重新将这个成员设置为 FALSE，如此本次“下半部分”完成所有数据帧的处理后，通知“下半部分”下一次有数据帧到来时，可以使用 netJobAdd 函数继续安排一个“下半部分”。

netJobAdd 主要在中断处理函数中调用，在 armStart 函数中，我们通过 intConnect 函数将 armInt 注册为底层驱动中断处理函数，该函数实现如下。

```
LOCAL void armInt(ARM_END_DEVICE   *pDrvCtrl){
    UINT32 val;
    UINT32 value;
    UINT32 valcp;

    /*disable interrupt temporarily. just make sure.*/
    REG_READ(EMAC_CTRL_MODULE_REG_EWCTRL,value);
    value&=~EMAC_GLOBAL_INT_ENABLE_BIT;
    REG_WRITE(EMAC_CTRL_MODULE_REG_EWCTRL,value);

    REG_READ(EMAC_CTRL_REG_MACINVECTOR,val);
    if((val&EMAC_MACINTVECTOR_VALID_INT)==0){ //无有效中断，则退出。
        /* return immediately, no error message */
        goto ena_int;
```

```
}
/* TODO - Check for statistics int*/
if(val&EMAC_MACINTVECTOR_STATINT){
	/* update all statistics and then clear all statistics registers.
	* for GMIIEN bit is 1, so write 0xffffffff to clear them.*/
	    netJobAdd ((FUNCPTR)armUpdateStatistics, (int)pDrvCtrl,0,0,0,0);
}
/* TODO - Check for errors
 * The error bit can only be cleared by resetting the EMAC module in hardware.
 */
if(val&EMAC_MACINTVECTOR_HOSTINT){
	DRV_LOG(DRV_DEBUG_RX,
		"armInt-host error interrupt received,restart driver!\n",1,2,3,4,5,6);
	armEndRestart(pDrvCtrl);
}

/* Have netTask handle any input packets
 * For i now use only channel 7, so just check channel7's int.
 */
if (val&EMAC_MACINTVECTOR_RXPEND){
	if (pDrvCtrl->netJobScheduled == FALSE){
	    pDrvCtrl->netJobScheduled = TRUE;
	    netJobAdd ((FUNCPTR)armHandleRcvInt, (int)pDrvCtrl,0,0,0,0);
	}
}

/* TODO - handle transmit interrupts */
if(val&EMAC_MACINTVECTOR_TXPEND){
	/* we have completed sending a packet.
	 * just acknownledge whatever is transmitted.
	 */
	/* Because i transmit packet in armTransPacket seriesly, so here acknownledging
	 * transmission-complete interrupt is just read the corresponding register and
	 * write the value back.
	 */
	REG_READ(EMAC_CTRL_REG_TX0CP,valcp);
	REG_WRITE(EMAC_CTRL_REG_TX0CP,valcp);
}
/* Note i put this code outside at the end to make it be called whenever an interrupt occured.
 * It is safe for armTransPacket will decide whether or not to transmit a packet.
 */

if(pDrvCtrl->txBufStatus!=TXBUFFEMPTY&&pDrvCtrl->txHandling==FALSE){
	/*armTransPacket(pDrvCtrl);*/
```

```
            pDrvCtrl->txHandling=TRUE;
            netJobAdd((FUNCPTR)armHandleTxInt, (int)pDrvCtrl,0,0,0,0);
        }

ena_int:
        /*reenable interrupt.*/
        REG_WRITE(EMAC_CTRL_MODULE_REG_EWCTRL,
                value|EMAC_GLOBAL_INT_ENABLE_BIT);
}
```

在 armInt 函数实现中，我们在三个地方使用了 netJobAdd 函数：（1）信息统计，由于需要读取大量寄存器的值，故我们将这个工作交由任务上下文中完成，我们没有使用 netJobScheduled 之类的变量来确保每次只有一个信息统计函数在内核工作队列中，是因为信息统计函数被调用的频率非常低，大量中断将来自于数据收发；（2）数据帧接收，我们使用 netJobScheduled 成员来确保每次 tNetTask 维护的内核工作队列中只有一个数据帧接收“下半部分”执行函数，这个执行函数将连续处理当前所有接受的数据帧，直到处理完毕，此时其将 netJobScheduled 重新设置为 FALSE，当下一次一个数据帧的接收触发中断时，armInt 再次调用 netJobAdd 函数安排一个数据帧接收“下半部分”；（3）数据帧发送，每当网络设备成功发送一个数据帧时，其也发出一个中断。事实上，我们并没有使用中断来触发数据帧的发送，而是一直使用 tNetTask 任务进行数据帧的发送。如同数据帧接收，我们使用 txHandling 成员来确保内核工作队列中同时只有一个数据帧发送“下半部分”。在前文介绍中，我们将数据发送使用的 EMAC_DESCU 结构维护成一个环形数组的形式，此时底层驱动使用 EMAC_DESCU 安排每个要发送的数据帧，数据帧发送“下半部分”则检查 EMAC_DESCU 是否已包含有效的数据帧，如果是，则安排网络设备将其发送出去，所以数据帧发送“下半部分”将一直依次检查每个 EMAC_DESCU 结构，如果检查到该结构已经包含有效的数据帧（即 EMAC_DESCU 结构中 OWNER 位被设置），则操作网络设备相关寄存器，将这个 EMAC_DESCU 对应的数据帧发送出去。发送完毕后，将触发一个发送完毕中断，armInt 中断处理函数简单清除发送中断，而后检查相关变量，决定是否需要再次安排一个数据帧发送“下半部分”。这种工作方式下，只要有数据帧需要发送，则内核工作队列中将有且仅有一个数据帧“下半部分”一直处于活跃状态。

10.6 数据帧接收函数

10.6.1 数据帧接收“下半部分”入口函数

底层驱动中断处理函数中，我们使用 netJobAdd 在 tNetTask 维护的内核工作队列中添加一个“下半部分”专门用于数据帧的具体接收工作，通过 netJobAdd 注册的函数是 armHandleRcvInt，该函数将完成数据帧的具体接收工作。基于前文中对 DM6446 平台下网络接口工作方式的介绍，armHandleRcvInt 函数将通过检查 EMAC_DESCU 结构队列来进行数据帧的接收工作，该函数具体实现代码如下所示。

```
LOCAL void armHandleRcvInt(ARM_END_DEVICE *pDrvCtrl){
```

```
int count=0;
UINT32    idx=pDrvCtrl->rxDescIdx;
volatile ARM_END_RECV_FD *pRxDesc;
//pRxDesc 是一个 EMAC_DESCU 结构指针，此处将其初始化
//为当前需要处理接收链表中的的 EMAC_DESCU 结构。
pRxDesc=pDrvCtrl->pRecvBufDesc+pDrvCtrl->rxDescIdx;
//如果 EMAC_DESCU 结构中 OWNER 标志位被清除，则表示该结构对应一个
//有效的数据帧。
//MAX_PACKETS_DEAL 表示连续处理的数据帧个数（=10），为了留出时间给
//内核工作队列中的其他“下半部分”执行，故在连续处理了 MAX_PACKETS_DEAL
//个数据帧，我们退出此次“下半部分”执行过程，余下的数据帧由下一次被安排的
//“下半部分”完成接收。
while((pRxDesc->pktFlgLen&EMAC_DSC_FLAG_OWNER)==0&&
            (++count<=MAX_PACKETS_DEAL)){
      //调用 armRecv 完成一个数据帧的接收，该函数实现下议。
      armRecv(pDrvCtrl,pDrvCtrl->rxDescIdx);

      //已到接收链表尾部，重新初始化网络设备寄存器，使其重新指向链表头部，
      //开始新的一次的 EMAC_DESCU 结构链表使用。
      if(pRxDesc->pktFlgLen&EMAC_DSC_FLAG_EOQ){
            REG_WRITE(EMAC_CTRL_REG_RX1HDP,
                  (UINT32)(pDrvCtrl->pRecvBufDesc));
            goto outer_fal;
      }

      idx=pDrvCtrl->rxDescIdx;
      pDrvCtrl->rxDescIdx++;
      pRxDesc=&((pDrvCtrl->pRecvBufDesc)[pDrvCtrl->rxDescIdx]);
}

//写网络设备寄存器，清除数据帧接收中断状态位。
pRxDesc=&((pDrvCtrl->pRecvBufDesc)[idx]);
REG_WRITE(EMAC_CTRL_REG_RX1CP,(UINT32)pRxDesc);

//此次没有处理完所有的数据帧接收，继续安排下一个数据帧接收“下半部分“并推出
//本次“下半部分”执行，以便工作队列中其他“下半部分”得到及时执行。
if(count>MAX_PACKETS_DEAL){
      DRV_LOG(DRV_DEBUG_RX,
            "armHandleRcvInt-too many input packets, reschedule again!\n",1,2,3,4,5,6);
      pDrvCtrl->netJobScheduled = TRUE;
      netJobAdd ((FUNCPTR)armHandleRcvInt, (int)pDrvCtrl,0,0,0,0);
      goto outer;
}
```

```
outer_fal:
    //已处理完当前所有数据帧的接收，重新设置 netJobScheduled 为 FALSE，当下一个
    //数据帧到来时，将触发一个中断，我们将在中断处理函数（armInt）中重新安排一个
    //新的“下半部分”进行数据帧的接收工作。
    pDrvCtrl->netJobScheduled=FALSE;
outer:
    /*DRV_LOG((DRV_DEBUG_TX|DRV_DEBUG_RX),
            "armHandleRcvInt-Done handle interrupt!\n",1,2,3,4,5,6);*/

}
```

10.6.2 内核数据帧封装要求

内核数据帧封装要求是指为了便于内核代码对数据帧的解析，在底层驱动将数据帧传递给上层之前，必须将数据帧封装在一个内核提供的数据结构中，例如 Linux 下的 sk_buff 结构，对于 Vxworks，则必须封装在一个 M_BLK_ID 结构中。M_BLK_ID 及其相关结构均定义在 h/netBufLib.h 内核头文件中，如下所示。

```
/* header at beginning of each mBlk */
typedef struct mHdr
{
    struct mBlk *  mNext;       /* next buffer in chain */
    struct mBlk *  mNextPkt;    /* next chain in queue/record */
    char *         mData;       /* location of data */
    int            mLen;        /* amount of data in this mBlk */
    UCHAR          mType;       /* type of data in this mBlk */
    UCHAR          mFlags;      /* flags; see below */
    USHORT         reserved;
} M_BLK_HDR;

/* record/packet header in first mBlk of chain; valid if M_PKTHDR set */
typedef struct  pktHdr
{
    struct ifnet *  rcvif;          /* rcv interface */
    int             len;        /* total packet length */
} M_PKT_HDR;

typedef union clBlkList
{
    struct clBlk *  pClBlkNext;  /* pointer to the next clBlk */
    char *          pClBuf;      /* pointer to the buffer */
} CL_BLK_LIST;
```

```
/* description of external storage mapped into mBlk, valid if M_EXT set */
typedef struct clBlk
{
    CL_BLK_LIST     clNode;         /* union of next clBlk, buffer ptr */
    UINT            clSize;         /* cluster size */
    int             clRefCnt;/* reference count of the cluster */
    FUNCPTR         pClFreeRtn;     /* pointer to cluster free routine */
    int             clFreeArg1;     /* free routine arg 1 */
    int             clFreeArg2;     /* free routine arg 2 */
    int             clFreeArg3;     /* free routine arg 3 */
    struct netPool *    pNetPool;       /* pointer to the netPool */
} CL_BLK;

/* mBlk statistics used to show data pool by show routines */
typedef struct mBlk
{
    M_BLK_HDR       mBlkHdr;            /* header */
    M_PKT_HDR mBlkPktHdr;       /* pkthdr */
    CL_BLK *    pClBlk;             /* pointer to cluster blk */
} M_BLK;

typedef M_BLK *  M_BLK_ID;
```

mBlk 与 clBlk 的使用及其之间的关系如下图 10-9 所示。

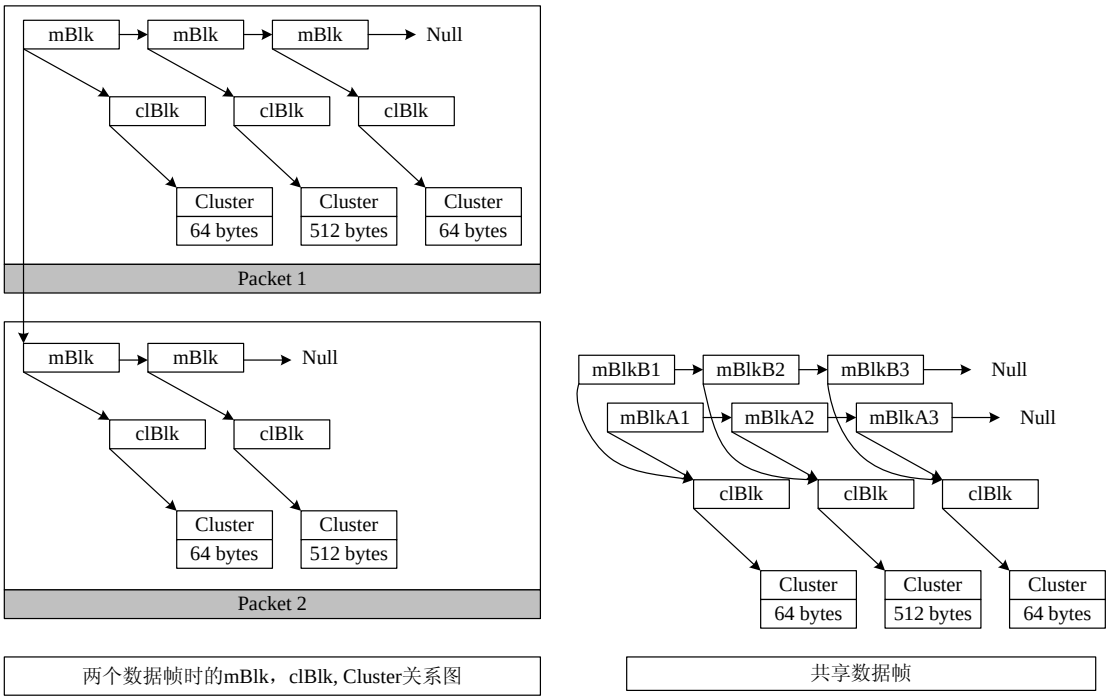


图 10- 9 mBlk，clBlk 以及底层缓冲区 Cluster 之间的关系

从图 10-9 中以及如上结构的定义中，我们可以得到如下的结论。

（1） 每个 clBlk 对应一个底层缓冲区，这个缓冲区 clBlk 结构中 clNode.pClBuf 指向。
（2） mBlk 中 mBlkHdr.mNext 指针用于同一个数据帧中 mBlk 之间的链接。
（3） mBlk 中 mBlkHdr.mNextPkt 指针用于数据帧间的 mBlk 之间的链接。
（4） mBlk 中 pClBlk 用以指向该 mBlk 结构对应的底层数据缓冲区。
（5） mBlk 中 mBlkPktHdr 字段由网络栈使用，不用于 mBlk，clBlk 以及更底层缓冲区之间的链接。
（6） 一个 clBlk 可以同时被多个 mBlk 共享，此种方式避免了内核数据复制的麻烦，但是如果有一个使用者需要修改这些数据，则必须进行 clBlk 的分离。
（7） mBlk 中 mBlkHdr.mData 以及 mBlkHdr.mLen 将根据数据帧到达网络栈的不同层次而相应进行调整，如当被网络层处理时，mData 将指向数据帧中网络层协议头部，mLen 对应的指网络层数据帧的长度（即已减去链路层头部长度），当被传输层处理时，mData 将指向数据帧中传输层协议头部，而 mLen 对应的指传输层数据帧的长度（即已减去链路层，网络层头部的长度）。mData 始终指向的是 Cluster 区域不同位置处，因为数据帧就存储在 Cluster 中。

底层驱动必须将数据帧封装成 mBlk，clBlk，Cluster 的层次结构。为此底层驱动必须预先分配一定数量的 mBlk，clBlk 以及 Cluster。通常 clBlk 与 Cluster 的数量相同，而 mBlk 数量一般为 clBlk 的两倍。在图 9 中，一个数据帧有多个 mBlk 组成，这是为了提供内存使用效率，将底层 Cluster 缓冲区分为多种大小不同区域，在我们的驱动示例中，我们只使用一种 Cluster 缓冲区类型，大小设置为 1520，如此第一个数据帧将对应一个 Cluster，一个 clBlk，一个 mBlk。即便如此，在分配 mBlk 数量时，依然应该分配为 clBlk 的两倍，因为驱动上层会使用多个 mBlk 共享一个 clBlk，方便数据的处理，提高数据处理效率。

为便于底层驱动维护 mBlk，clBlk 以及底层数据帧缓冲区 Cluster，内核提供两个结构和一个“池”：（1）两个结构，M_CL_CONFIG，CL_DESC；（2）一个池，netPool 结构。以上结构均定义在 h/netBufLib.h 内核头文件中。

M_CL_CONFIG 结构用以表示 mBlk，clBlk 的分配数量以及分配的内存区域和大小，该结构定义如下。

```
typedef struct
{
    int         mBlkNum;        /* number of mBlks */
    int         clBlkNum;       /* number of cluster Blocks */
    char *      memArea;        /* pre allocated memory area */
    int         memSize;    /* pre allocated memory size */
} M_CL_CONFIG;
```

在我们的底层驱动中，我们定义了一个全局 M_CL_CONFIG 结构变量 armMclBlkConfig，如下所示。

```
M_CL_CONFIG armMclBlkConfig = {
    /*
    no. mBlks       no. clBlks          memArea     memSize
    ----------      ----------          -------     -------
    */
```

```
    0,                  0,              NULL,       0
};
```

armMclBlkConfig变量尚未进行初始化，这个初始化的工作将在armMemInit函数（在armLoad函数中被调用）完成，该函数具体实现稍后给出。

CL_DESC 则表示底层数据帧缓冲区的类型和大小以及分配内存区域首地址和内存区域总容量。为了简化代码设计，我们只使用了一种缓冲区类型，每个缓冲区大小为 1520 字节，此时一个缓冲区就可以完全存放一个完整的帧，也就只需一个 clBlk 结构，一个 mBlk 结构对应。但是这种情况下对于内存的使用效率较低，因为网络数据帧最大 1518 个字节，通常的帧长度可能只有 64 个字节或者几百字节，这就表示一个缓冲区中大部分空间都是空闲的。对于内存资源紧张的平台，可以使用多种缓冲区类型，如使用两种缓冲区类型：缓冲区类型 1 采用 64 个字节，缓冲区类型 2 采用 1520 个字节。多个缓冲区类型的使用请读者参考文献 vxworks bsp developer’s guide。
在我们的驱动示例中，我们定义了一个全局 CL_DESC 结构变量 armClDescTbl，如下所示。

```
CL_DESC armClDescTbl [] = {
    /*In fact, 1514 is ok for the cluster size not include crc, but for ip header aligning on 16-byte
      *boundary edge, we need 2-byte offset here.
      */
    /*
    clusterSize                 num         memArea     memSize
    -----------                 ----        -------     -------
    */
    /* I just define a single cluster type whose size is 1520.
      * you can also define cluster buffers whose size is 65535, which is the largest
      * packet length for network protocols, since ip header's field used to indicate
      * the packet size is a 16-bit jamble.
      */
    {ARM_END_CL_SIZE,       0,          NULL,           0}
};
```

注意：以上注释中提到最大缓冲区大小可以定义为 65535 字节，并表示这是网络协议最大包长度，这是从考虑分片的角度以及 IP 协议首部表示包长度的字段是一个 16-bit 字段的角度而言的。在上文介绍中，我们一直说到最大数据帧长度为 1518（包括校验）字节，这两个之间并无矛盾。这要从报文如何进行定义来解释了。65535 是从网络层之上的层次而言的，而网络层及其之下（包括底层驱动，网络设备以及网络介质）一个数据帧的长度都不可超过 1518 字节，具体细节请读者参考网络协议相关资源。

以上定义的两个结构变量都将在 armMemInit 函数中完成最终的初始化。不过在介绍 armMemInit 函数实现之前，还必须对缓冲“池”进行介绍。基于 mBlk，clBlk 以及底层缓冲区 Cluster 之间的紧密关系，内核专门提供 netPool 结构对这些结构进行统一管理。该结构定义如下。

```
/*h/netBufLib.h*/
struct netPool                  /* NET_POOL */
```

```
{
    M_BLK_ID    pmBlkHead;          /* head of mBlks */
    CL_BLK_ID   pClBlkHead;         /* head of cluster Blocks */
    int         mBlkCnt;        /* number of mblks */
    int         mBlkFree;           /* number of free mblks - deprecated,
                                                    use pPoolStat->mTypes[MT_FREE]
                                                    instead */
    int         clMask;             /* cluster availability mask */
    int         clLg2Max;           /* cluster log2 maximum size */
    int         clSizeMax;          /* maximum cluster size */
    int         clLg2Min;           /* cluster log2 minimum size */
    int         clSizeMin;          /* minimum cluster size */
    CL_POOL *   clTbl [CL_TBL_SIZE];  /* pool table */
    M_STAT *    pPoolStat;          /* pool statistics */
    POOL_FUNC *     pFuncTbl;           /* ptr to function ptr table */
};
```

其中有三个成员需要特别关注：pmBlkHead，pClBlkHead, clTbl。这三个字段分配对应着 mBlk 池，clBlk 池以及底层缓冲区 Cluster 池。
END_OBJ 结构必须与驱动自定义结构配合同时使用 netPool 结构中维护的这三个结构池。为此 END_OBJ 结构定义 pNetPool 成员，该成员是一个 netPool 结构指针。驱动自定义结构定义 pClPoolId 成员，该成员是一个 clPool 结构指针。此时由 pNetPool 负责 mBlk，clBlk 结构的分配，而由 pClkPoolId 负责底层缓冲区的分配。

armMemInit 函数将完成 mBlk，clBlk，缓冲区的分配以及 pNetPool 和 pClPoolId 的初始化工作，其实现代码如下。

```
LOCAL STATUS armMemInit(ARM_END_DEVICE *pDrvCtrl){
    int txbufsize,rxbufsize;
     /*
      * This is how we would set up an END netPool using netBufLib(1).
      * This code is pretty generic.
      */
    //分配 netPool 结构。
    if ((pDrvCtrl->end.pNetPool = (NET_POOL_ID)memalign (sizeof(int),
                    sizeof(NET_POOL))) == NULL){
        DRV_LOG(DRV_DEBUG_LOAD,
            "armMemInit-Fail to allocate mem for NET_POOL_ID!\n",1,2,3,4,5,6);
        return (ERROR);
    }

    /*
      * Note, Separating the descriptor from the data buffer can be
      * advantageous for architectures that need cached data buffers but
      * don't have snooping units to make the caches fully coherent.
```

```
 * It is a disadvantage for architectures that do have snooping, becuase
 * they need to do twice the number of netBufLib operations for each
 * data frame.    This arm driver assumes the descriptor and data
 * buffer are contiguous.
 */
//初始化 M_CL_CONFIG 结构各成员。
/* number of driver Descriptor/data buffers */
armClDescTbl[0].clNum = END_CL_NUM;/*number of clusters ： =RX_FD_NUM=254*/

/* Setup mbuf,cluster block pool with    mbufs equals clBlks */
armMclBlkConfig.clBlkNum = armClDescTbl[0].clNum;
armMclBlkConfig.mBlkNum = END_MBLK_NUM; /* END_CL_NUM*2*/

/* Calculate the total memory for all the M-Blks and CL-Blks. */
/* MSIZE is defined in netBufLib.h as sizeof(struct mBlk),
* CL_BLK_SZ is defined in netBufLib.h as sizeof(struct clBlk)
*
* The reason to add sizeof(long) is that the kernel will add 4-byte header for
* each mBlk, clBlk struct, when accessing these structs, the 4-byte header is opaque!
*/
//计算 mBlk，clBlk 结构共需的内存空间大小。
armMclBlkConfig.memSize = (armMclBlkConfig.mBlkNum *(MSIZE + sizeof (long))) +
                          (armMclBlkConfig.clBlkNum *(CL_BLK_SZ + sizeof(long)));

/* allocate mbuf/Cluster blocks from normal memory */
//分配 mBlk，clBlk 所需的内存。
if ((armMclBlkConfig.memArea = (char *) memalign (sizeof(long),
                          armMclBlkConfig.memSize))== NULL){
      DRV_LOG(DRV_DEBUG_LOAD,
            "armMemInit-fail to allocate mem for mBlk,clBlk strutures!\n",1,2,3,4,5,6);
      return (ERROR);
}

//初始化 CL_DESC 结构。
/* Round cluster size up to a multiple of a cache line */
armClDescTbl[0].clSize =ROUND_UP(armClDescTbl[0].clSize,4);

/* Calculate the memory size of all the clusters. */
armClDescTbl[0].memSize =
      ROUND_UP(armClDescTbl[0].clNum * (armClDescTbl[0].clSize+sizeof(long))
            +sizeof(int),_CACHE_ALIGN_SIZE);

/* Allocate the memory for the clusters from cache safe memory. */
/* cacheDmaMalloc( ) – allocate a cache-safe buffer for DMA devices and drivers.*/
```

```
    /* cacheDmaMalloc memory is assumed to be cache line aligned ! */
    armClDescTbl[0].memArea = (char *) cacheDmaMalloc
                    (armClDescTbl[0].memSize+_CACHE_ALIGN_SIZE);

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

    /* use cacheDmaFuncs for cacheDmaMalloc memory */
    pDrvCtrl->pCacheFuncs = &cacheDmaFuncs;

    /* Initialize the memory pool. */
    //初始化 END_OBJ 结构中 pNetPool 成员。
    if (netPoolInit(pDrvCtrl->end.pNetPool, &armMclBlkConfig,&armClDescTbl[0],
            armClDescTblNumEnt,NULL) == ERROR){
        DRV_LOG (DRV_DEBUG_LOAD,
             "armMemInit-Could not init buffering\n",1, 2, 3, 4, 5, 6);
        printf("err=%d\n",errno);
        return (ERROR);
    }

    /*
     * If you need clusters to store received packets into then get them
     * here ahead of time.    This arm driver only uses the cluster pool
     * for receiving.    Here it takes all the clusters out of the pool and
     * attaches them to the device.    The device will fill them with incoming
     * packets and trigger an interrupt.    This will schedule the
     * armHandleRcvInt routine.    It will unlink the cluster and send
     * it to the stack.    When the stack is done with the data, the cluster
     * is freed and returned to the cluster pool to be used by the device
     * again.
     */
    //初始化驱动自定义结构中 pClPoolId 成员。
    if ((pDrvCtrl->pClPoolId =netClPoolIdGet (pDrvCtrl->end.pNetPool,
            armClDescTbl[0].clSize, FALSE)) == NULL){
        DRV_LOG(DRV_DEBUG_LOAD,"armMemInit-fail to create pool id!\n",1,2,3,4,5,6);
        return (ERROR);
    }

    //EMAC_DESCU 结构初始化。
    armQueueInit(pDrvCtrl,(TX_QUEUE_INIT|RX_QUEUE_INIT));
}
```

armMemInit 函数中对于 mBlk，clBlk 以及底层缓冲区和 END_OBJ 结构中 pNetPool 以及驱动自定义结构中 pClPoolId 字段的初始化代码可以被复制到其他 Vxworks 下网络设备驱动代码中，这些代码是通用的，只需修改部分常量的值即可直接使用。

在完成 mBlk，clBlk 以及底层缓冲区池的准备，下面我们介绍 armRecv 函数的实现，具体查看底层驱动是如何使用此处预备的池对数据帧进行封装的。

10.6.3 数据帧处理和上传

在我们的驱动示例中，数据帧的处理和上传由 armRecv 函数完成，该函数将在 tNetTask 任务上下文被调用，完成数据帧的向上（MUX 中间层）递交工作。在递交之前，我们必须按照上层要求将数据帧封装成 mBLk，clBlk 以及 Cluster 形式。
armRecv 函数实现代码如下。

```
LOCAL STATUS armRecv(ARM_END_DEVICE *pDrvCtrl,UINT32 rxIdx){
    int             len = 0;
    M_BLK_ID        pMblk;
    char*           pNewCluster;
    CL_BLK_ID       pClBlk;

    /* TODO - Packet must be checked for errors. */
    flgLen=(pDrvCtrl ->pRecvBufDesc)[rxIdx].pktFlgLen;
    if(flgLen&EMAC_DSC_FLAG_ERROR){
        goto rxerr;
    }
    /* Add one to our unicast data. */
    //更新数据包接收统计信息。
    END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_UCAST, +1);

    /*
     * We implicitly are loaning here, if copying is necessary this
     * step may be skipped, but the data must be copied before being
     * passed up to the protocols.
     */
    //分配一个 Cluster（底层数据帧缓冲区），注意 netClusterGet 需要同时使用 END_OBJ
    //结构中 pNetPool 以及自定义结构中 pClPoolId 成员。
    pNewCluster = netClusterGet (pDrvCtrl->end.pNetPool, pDrvCtrl->pClPoolId);

    if (pNewCluster == NULL){
        END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
        goto rxerr;
    }
```

```
/* Grab a cluster block to marry to the cluster we received. */
//从 clBlk 池中分配一个 clBlk 结构，用以封装底层缓冲区 Cluster。
if ((pClBlk = netClBlkGet (pDrvCtrl->end.pNetPool, M_DONTWAIT)) == NULL){
        netClFree (pDrvCtrl->end.pNetPool, (UCHAR *)pNewCluster);
        END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
        goto rxerr;
}
//从 mBlk 池中分配一个 mBlk 结构，用以封装 clBlk 结构。
if ((pMblk = mBlkGet (pDrvCtrl->end.pNetPool, M_DONTWAIT, MT_DATA)) == NULL){
        netClBlkFree (pDrvCtrl->end.pNetPool, pClBlk);
        netClFree (pDrvCtrl->end.pNetPool, (UCHAR *)pNewCluster);
        END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
        goto rxerr;
}

//完成数据帧向底层 Cluster 缓冲区的拷贝。
len = (flgLen&(0xffff));
/*add 2 to align ip header on 16-byte boundary.*/
bcopy (pDrvCtrl->clRxPointBuf[rxIdx],pNewCluster+2,len);
//将 Cluster，clBlk，mBlk 串接在一起，串接后的情景如下图 10 所示。
/* Join the cluster to the MBlock */
netClBlkJoin (pClBlk, pNewCluster, armClDescTbl[0].clSize, NULL, 0, 0, 0);
netMblkClJoin (pMblk, pClBlk);

END_CACHE_INVALIDATE (pMblk->mBlkHdr.mData, len);
/* The purpose of adding 2 is to align ip header on 16-byte boundary edge.*/
//更新 mBLk 结构中 mBlkHdr 字段中各成员的值。
//mData 指向底层缓冲区数据帧链路层首部。
//mLen 表示整个数据帧的长度，包括链路层，网络层，传输层以及纯数据的长度。
//mFlags 设置 M_PKTHDR 标志位，表示这个 mBlk 表示了一个完整的数据帧。
pMblk->mBlkHdr.mData+=2;
pMblk->mBlkHdr.mLen=len;
pMblk->mBlkHdr.mFlags|=M_PKTHDR;
pMblk->mBlkPktHdr.len=len;

/* Call the upper layer's receive routine. */
//调用 END_OBJ 结构中 receiveRtn 字段指向的函数（muxReceive）将封装后数据帧
//从底层驱动传递给 MUX 中间层进行进行处理，完成传递后，底层驱动数据帧接收
//工作宣告成功结束。
END_RCV_RTN_CALL(&pDrvCtrl->end, pMblk);

//更新驱动使用的 EMAC_DESCU 结构。
/* clear this descriptor*/
pDrvCtrl ->pRecvBufDesc[rxIdx].pBufOffLen=pDrvCtrl->tranPkgSize;
```

```
        pDrvCtrl ->pRecvBufDesc[rxIdx].pktFlgLen=EMAC_DSC_FLAG_OWNER;
        return (OK);

rxerr:
        pDrvCtrl ->pRecvBufDesc[rxIdx].pBufOffLen=pDrvCtrl->tranPkgSize;
        pDrvCtrl ->pRecvBufDesc[rxIdx].pktFlgLen=EMAC_DSC_FLAG_OWNER;
        return (ERROR);
}
```

armRecv 函数的实现非常具有代表性，其演示了 mBlk，clBlk 以及底层缓冲区之间是如何联系在一起的，以及经过以上三个结构封装后的数据帧是如何传递给 MUX 中间层的。这些代码对所有 Vxworks 下网络设备驱动都是通用的，读者完全可以复制到自己的网络设备底层驱动相关函数中进行使用（只需很少的修正）。

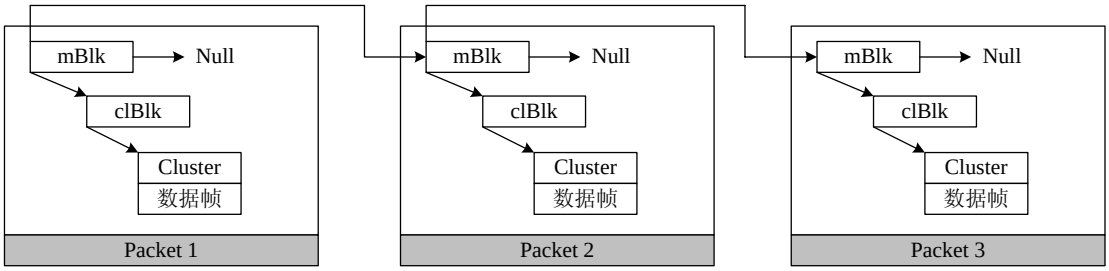


图 10- 10 三个数据帧时 mBlk，clBlk，Cluster 关系示例图

10.6.4 数据帧接收再议

在 armRecv 函数中，我们进行了一次数据帧的复制过程：即将 EMAC_DESCU 结构指向的缓冲区中数据帧复制到 Cluster 指向的缓冲区中。我们区分使用两套不同的缓冲区，一方面需要进行一次数据帧的复制，另一方面也增加了内存使用空间。为了提供内存使用效率，以及数据帧上传的效率，实际上我们可以直接将 EMAC_DESCU 指向的缓冲区设置为 Cluster 缓冲区，即二者使用相同的一套缓冲区，如此只需进行 mBLk，clBlk 结构的分配以及 mBlk，clBLk 以及 Cluster 之间的串接工作即可，省去了数据帧复制的工作，同时也大大减少了内存的使用。

但是这种使用同一套缓冲区的方式必须确保同时只能有一方对缓冲区进行操作。因为内核和网络设备是异步的在使用缓冲区，如果内核尚未完成对缓冲区中数据的处理，而网络设备又要想缓冲区中填入新的数据，则将造成系统混乱，所以使用同一套缓冲区的策略在我们所示的驱动实现中是不可取的。但是这种策略在某些平台下是可以使用的，读者需要根据自身平台下驱动设计的特点决定是否使用同一套缓冲区以提供内存使用效率以及数据帧上传效率。

10.7 数据帧发送函数

从底层驱动的角度看，数据帧发送完全是被动的，即底层驱动提供一个发送函数，当上层MUX 中间层需要发送数据帧，其将这个数据帧作为参数调用底层驱动发送函数完成数据帧的发送。底层驱动数据帧发送函数实现需要考虑一个数据帧缓存的问题，如果 MUX 中间层调用发送函数非产密集，此时网络设备发送数据帧的速率将无法跟上 MUX 层调用发送函数的速率，那么在底层驱动中将积累一些待发送的数据帧，这一点在实现中必须要注意。在我们的驱动示例中，我们最多同时可以缓存 254 个数据帧（我们为发送分配了 254 个EMAC_DESCU 结构，每个结构对应一个数据帧）。如果大于 254 个数据帧，此时底层驱动数据帧发送函数将返回 ERROR，通知 MUX 中间层稍后重新发送数据帧。当然事实上，在底层驱动中也完全可以不使用缓存，每次只发送一个数据帧，直到完成发送，才从上层接收下一个数据帧，这种方式也可以工作，但是效率上比较低。通常网络设备本身都会提供一个内部硬件缓冲区可以缓存多个数据帧。

诚如数据帧接收函数中在将一个数据帧传递给上层之前，必须将数据帧封装成 mBlk，clBlk 以及 Cluster 的形式，数据帧接收函数所接受的从上层下发的数据帧同样也是被封装成 mBlk 形式，故我们需要完成 mBlk 中数据向 EMAC_DESCU 结构的复制，而后释放 mBlk 等封装结构和 Cluster 底层缓冲区，将这些结构和底层缓冲区重新放入“池”中。这将由内核专门函数完成。

数据帧发送函数实现代码如下所示。

```
/*******************************************************************************
* armSend - the driver send routine
*
* This routine takes a M_BLK_ID sends off the data in the M_BLK_ID.
* The buffer must already have the addressing information properly installed
* in it.    This is done by a higher layer.    The last arguments are a free
* routine to be called when the device is done with the buffer and a pointer
* to the argument to pass to the free routine.
*
* RETURNS: OK, ERROR, or END_ERR_BLOCK.
*/
LOCAL STATUS armSend(ARM_END_DEVICE *pDrvCtrl,M_BLK_ID pMblk){
      UINT32   writeIdx;
      UINT32   oldlevel;
      UINT32    len;
      UINT32    status;
      char       *txbuf;

      /*
     * Obtain exclusive access to transmitter.    This is necessary because
     * we might have more than one stack transmitting at once.
     */
      /*END_TX_SEM_TAKE ( &pDrvCtrl->end, WAIT_FOREVER );*/
      semTake(pDrvCtrl->txLock,WAIT_FOREVER); //use ourself's lock
```

```
    if(pDrvCtrl ->txBufStatus == TXBUFFFULL){
        semGive(pDrvCtrl->txLock);
        return ERROR;
    }
    oldlevel = intLock ();
    writeIdx = pDrvCtrl ->txDescWriteIdx;
    intUnlock(oldlevel);

    txbuf = (pDrvCtrl ->pTxBufDesc)[writeIdx].pBuffer;
    len = pMblk->mBlkPktHdr.len&0xffff;
    /* here we need differentiate the packet length greater than 1514 and less than 1514*/
    if(len<=1514){
        if(len<60){/*it is legitimate for packets to be transmitted smaller than 60 bytes.*/
            len=60;
        }
        bzero(txbuf,len);
        (pDrvCtrl ->pTxBufDesc)[writeIdx].pktFlgLen=EMAC_DSC_FLAG_OWNER|
            EMAC_DSC_FLAG_SOP|EMAC_DSC_FLAG_EOP|(len);
        (pDrvCtrl ->pTxBufDesc)[writeIdx].pBufOffLen = len;
        /* Set pointers in local structures to point to data. */
        netMblkToBufCopy(pMblk, (void *)txbuf, NULL) ;
    }
    else{
    /*packet length>1514*/
    /* needs fragmentation，  it is the network stack,not device driver,
    * assumes the responsibility of fragmenting large packet.
      * Return ERROR for such circumstance.
    */
        semGive(pDrvCtrl->txLock);
        return ERROR;
    }

/* place a transmit request */
/* lock task to prevent task scheduling.
  * in fact, itis unreasonable(or necessary) to lock task here,
  * but transmit errors always occure, so everything suspectable should be cleared.
  * i need double surance now.

  * Note that, lock task can not prevent interrupt, so i call intLock to disable
  * disturbance from interrupt, so the code block below is double surly executed quitely.
  */

if(taskLock()==OK){
```

```
        oldlevel = intLock ();
        status=pDrvCtrl->txBufStatus;
        if( pDrvCtrl ->txDescWriteIdx == (TX_FD_NUM-1) )
            pDrvCtrl ->txDescWriteIdx = 0;
        else
            pDrvCtrl ->txDescWriteIdx ++;

        if( pDrvCtrl ->txDescWriteIdx == pDrvCtrl ->txDescSendIdx)
            pDrvCtrl ->txBufStatus = TXBUFFFULL;
        else
            pDrvCtrl ->txBufStatus = TXBUFFNORMAL;

        if(status==TXBUFFEMPTY){
            /*armTransPacket(pDrvCtrl);*/
            netJobAdd((FUNCPTR)armHandleTxInt, (int)pDrvCtrl,0,0,0,0);
        }
        intUnlock (oldlevel);
        taskUnlock();
    }
    else{
        DRV_LOG(DRV_DEBUG_TX,"armSend-Fail to lock task from scheduling!\n",1,2,3,4,5,6);
        semGive(pDrvCtrl->txLock);
        netMblkClChainFree(pMblk);
        goto ret_fal;
    }

        semGive(pDrvCtrl->txLock);
        /*END_TX_SEM_GIVE (&pDrvCtrl->end);*/

    /* Bump the statistic counter. */
    END_ERR_ADD (&pDrvCtrl->end, MIB2_OUT_UCAST, +1);
    netMblkClChainFree (pMblk);
    return (OK);

ret_fal:
        return (ERROR);
}
```

基于 DM6446 平台网口驱动的特点，armSend 函数将上层传入的封装在 mBlk 结构中的数据复制到 EMAC_DESCU 结构对应的缓冲区中，即完成数据帧的第一阶段发送工作。第二阶段的发送工作（即具体操作网络设备的工作）将在 tNetTask 任务上下文中完成，我们提供的执行函数是 armHandleTxInt。armSend 函数的实现代码具有通用性，对于其他网络设备驱动而言，只不过复制的目的地不同而已，其基本实现思想都是一致的。

armSend 函数中涉及到两个内核提供的函数，这两个函数在 Vxworks 下底层网络设备驱动数

据帧发送函数的实现中通常都需要使用，这些函数如下。

（1） netMblkToBufCopy

该函数完成 mBlk 结构中数据向用户指定缓冲区的复制。使用该函数的好处是我们无需剖析 mBlk，clBlk 以及底层 Cluster 缓冲区之间的复杂关系即可完成数据的复制工作。该函数的调用原型如下：

```
int netMblkToBufCopy
    (
    M_BLK_ID        pMblk,          /* pointer to an mBlk */
    char *          pBuf,           /* pointer to the buffer to copy */
    FUNCPTR         pCopyRtn        /* function pointer for copy routine */
    );
```

参数 1 为 MUX 中间层传递的 mBlk 结构，其中封装有需要发送的数据帧。

参数 2 为数据帧复制的目的缓冲区。

参数 3 为复制数据时使用的函数，当以 NULL 传入时，内核默认使用 bcopy 函数，通常都是以 NULL 调用 netMblkToBufCopy。

在 armSend 实现中，netMblkToBufCopy 调用代码如下。

```
netMblkToBufCopy(pMblk, (void *)txbuf, NULL) ;
```

pMblk 为上层传入的 mBlk 结构指针，txbuf 为 EMAC_DESCU 结构对应的缓冲区，缓冲区大小为 1520 字节。我们以 NULL 作为第三个参数传入，此时内核将使用 bcopy 函数具体完成数据的复制工作。

（2） netMblkClChainFree

这个函数完成 mBlk，clBlk 以及底层 Cluster 缓冲区的释放工作，更确切的说应该是回收工作，将 mBlk，clBlk，底层缓冲区分别回收到各自的“池”中，以便于下次被使用。该函数调用原型如下，只有一个参数，即顶层 mBlk 结构。

```
void netMblkClChainFree
    (
    M_BLK_ID        pMblk           /* pointer to the mBlk */
    );
```

在 armSend 函数实现中，我们将 mBlk 结构中封装的数据帧复制到 EMAC_DESCU 结构对应的缓冲区中，这个缓冲区数据的最终发送将由 armHandleTxInt 函数完成，该函数被使用 netJobAdd 函数添加到 tNetTask 维护的内核工作队列中，在 tNetTask 任务上下文执行。在前文中我们已提及，DM6446 平台上的网络接口设备必须使用 EMAC_DESCU 结构进行网络数据帧的收发，对于发送而言，我们维护一个 EMAC_DESCU 环形数组，armSend 函数只是将数据写入这个环形数组中，而环形数组中的数据通过网络设备的发送将由 armHandleTxInt 函数完成，该函数实现如下。

```
LOCAL void armHandleTxInt(ARM_END_DEVICE *pDrvCtrl){
     UINT32 oldlevel;
    /* In fact, armTransPacket routine should defined as return a bool variable, so
     * we can decide whether or not calling this routine from this return value.
     * but as it is defined as return void, so i must differentiate what situation is
```

```
     * myself in every routine calling armTransPacket.
     */
    while(1){
        if(pDrvCtrl->txDescSendIdx==pDrvCtrl->txDescWriteIdx
                ||pDrvCtrl->txBufStatus==TXBUFFEMPTY){
            break;
        }
        oldlevel=intLock();
        armTransPacket(pDrvCtrl);
        intUnlock(oldlevel);
    }

    pDrvCtrl->txHandling=FALSE;
}
```

驱动自定义结构中对于数据帧发送维护了几个成员变量，txDescWriteIdx 是 armSend 函数用以操作 EMAC_DESCU 结构数组的索引，其表示下一个数据帧写入的位置，而 txDescSendIdx 则被网络设备使用，其表示下一个待发送数据帧的位置，当 txDescSendIdx 等于 txDescWriteIdx 时，则表示 EMAC_DESCU 结构数组中所有缓存的数据帧都已被发送。此外 txBufStatus 成员也用以表示 EMAC_DESCU 结构数组的使用状态。armHandleTxInt 的实现思想非常简单，其实现为一个 while 循环，不断的对 EMAC_DESCU 结构数组中待发送的数据帧进行处理，直到当前所有的数据帧被发送完毕，具体的发送工作由 armTransPacket 函数完成。注意 armHandleTxInt 函数最后将 txHandling 设置为 FALSE，这个成员在 armSend 和 armInt 函数中被检查来确定是否需要再次安排一个“下半部分”到 tNetTask 工作队列中。armTransPacket 是具体操作网络设备寄存器发送数据帧的实现函数，此处我们不再给出该函数实现。

10.8 设备控制函数

基本所有的底层驱动都会提供一个 ioctl 函数用以重新配置和获取设备的相关参数或者设备的工作模式等等。在我们的网络设备驱动示例中，这个函数实现为 armIoctl。这个函数也将通过 NET_FUNCS 结构提供给上层使用。

对于网络设备驱动，ioctl 函数可以使用的选项如下图 10-11 所示。

命令	功能	数据类型
EIOCSFLAGS	设置设备标志位	int(END_OBJ.flags)
EIOCGFLAGS	获取设备标志位	int
EIOCSADDR	设置设备MAC地址	char*
EIOCGADDR	获取设备MAC地址	char*
EIOCMULTIADD	添加多播地址	char*
EIOCMULTIDEL	删除多播地址	char*
EIOCMULTIGET	获取多播地址	MULTI_TABLE*
EIOCPOLLSTART	设置设备到查询模式	NULL
EIOCPOLLSTOP	设置设备到中断模式	NULL
EIOCGFBUF	获取最小缓冲区大小	int
EIOCGMIB2	获取MIB-II统计信息	M2_INTERFACETBL*

图 10- 11 网络设备可用选项

armIoctl 函数实现如下。

```
/*******************************************************************************
* armIoctl - the driver I/O control routine
*
* Process an ioctl request.
*
* RETURNS: A command specific response, usually OK or ERROR.
*/
LOCAL int armIoctl(ARM_END_DEVICE * pDrvCtrl,int cmd,caddr_t data){
    int err= 0;
    long value;

    switch ((unsigned)cmd){
      case EIOCSADDR:          /* set MAC address */
            if (data == NULL){
                err=(EINVAL);
                goto outer;
            }
            DRV_LOG(DRV_DEBUG_IOCTL,"armIoctl-EIOCSADDR.\n",1,2,3,4,5,6);
            bcopy ((char *)data, (char *)END_HADDR(&pDrvCtrl->end),
                    END_HADDR_LEN(&pDrvCtrl->end));
            break;

      case EIOCGADDR:          /* get MAC address */
            if (data == NULL){
                err=(EINVAL);
                goto outer;
            }
            DRV_LOG(DRV_DEBUG_IOCTL,"armIoctl-EIOCGADDR.\n",1,2,3,4,5,6);
            bcopy ((char *)END_HADDR(&pDrvCtrl->end),(char *)data,
                    END_HADDR_LEN(&pDrvCtrl->end));
```

```
        break;

case EIOCSFLAGS:   /* set (or clear) flags */
        DRV_LOG(DRV_DEBUG_IOCTL,"armIoctl-EIOCSFLAGS.\n",1,2,3,4,5,6);
        value = (long)data;
        if (value < 0){
              value = -value;
              value--;
              END_FLAGS_CLR (&pDrvCtrl->end, value);
        }
        else{
              END_FLAGS_SET (&pDrvCtrl->end, value);
        }
        armFlgChange (pDrvCtrl,0);
        break;

case EIOCGFLAGS:  /* get flags */
        DRV_LOG(DRV_DEBUG_IOCTL,"armIoctl-EIOCGFLAGS.\n",1,2,3,4,5,6);
        *(int *)data = END_FLAGS_GET(&pDrvCtrl->end);
        break;

   case EIOCPOLLSTART:       /* Begin polled operation */
        armPollStart (pDrvCtrl);
        break;

   case EIOCPOLLSTOP:  /* End polled operation */
        armPollStop (pDrvCtrl);
        break;

case EIOCGMIB2233:
case EIOCGMIB2:           /* return MIB information */
        DRV_LOG(DRV_DEBUG_IOCTL,"armIoctl-EIOCGMIB2TBL.\n",1,2,3,4,5,6);
        if (data == NULL){
          err=(EINVAL);
          goto outer;
        }
        bcopy((char *)&pDrvCtrl->end.mib2Tbl, (char *)data,
                    sizeof(pDrvCtrl->end.mib2Tbl));
        break;

case EIOCGFBUF:           /* return minimum First Buffer for chaining */
        DRV_LOG(DRV_DEBUG_IOCTL,"armIoctl-EIOCGFBUF.\n",1,2,3,4,5,6);
        if (data == NULL){
              err=(EINVAL);
```

```
                goto outer;
            }
            *(int *)data = ARM_MIN_FBUF;
            break;

        case EIOCGHDRLEN:
            DRV_LOG(DRV_DEBUG_IOCTL,"armIoctl-EIOCGHDRLEN.\n",1,2,3,4,5,6);
            if (data == NULL)
                return (EINVAL);
            *(int *)data = EH_SIZE;
            break;

        default:            /* unknown request */
            DRV_LOG(DRV_DEBUG_IOCTL,"armIoctl-DEFAULT.\n",1,2,3,4,5,6);
            err = EINVAL;
    }
outer:
    return (err);
}
```

armIoctl 实现简单直观，此处不再多做介绍。在 armIoctl 函数中，我们调用了 armFlgChange，armPollStart，armPollStop 函数完成具体的功能实现。armFlgChange 根据具体标志位的改变对应的配置网络设备，该函数都是网络设备寄存器配置操作，在下文中介绍多播地址配置时再给出代码；armPollStart 切换网络设备到查询工作模式，关闭中断，armPollStop 则开启中断，重新切换到中断工作模式。这两个函数的代码如下。

```
/*******************************************************************************
* armPollStart - start polled mode operations
*
* RETURNS: OK or ERROR.
*/
LOCAL STATUS armPollStart(ARM_END_DEVICE * pDrvCtrl){
    int oldLevel;

    /* disable ints during update */
    oldLevel = intLock ();
    /* TODO - turn off interrupts */
    pDrvCtrl->flags |= ARM_POLLING;
    armFlgChange(pDrvCtrl,1);
    intUnlock (oldLevel);     /* now armInt won't get confused */

    DRV_LOG (DRV_DEBUG_POLL, "armPollStart-STARTED\n", 1, 2, 3, 4, 5, 6);
    armEndRestart(pDrvCtrl);
    return (OK);
}
```

```
/*******************************************************************************
* armPollStop - stop polled mode operations
*
* This function terminates polled mode operation.    The device returns to
* interrupt mode.
*
* The device interrupts are enabled, the current mode flag is switched
* to indicate interrupt mode and the device is then reconfigured for
* interrupt operation.
*
* RETURNS: OK or ERROR.
*/
LOCAL STATUS armPollStop(ARM_END_DEVICE * pDrvCtrl){
    int    oldLevel;

    oldLevel = intLock ();    /* disable ints during register updates */
    /* TODO - re-enable interrupts */
    pDrvCtrl->flags &= ~ARM_POLLING;
    intUnlock (oldLevel);

    armEndRestart(pDrvCtrl);
    DRV_LOG (DRV_DEBUG_POLL, "armPollStop-STOPPED\n", 1, 2, 3, 4, 5, 6);
    return (OK);
}
```

在armIoctl函数，我们并没有直接实现EIOCMULTIADD，EIOCMULTIDEL，EIOCMULTIGET选项，实际上这些函数的响应将由 MUX 中间层通过调用底层驱动通过 NET_FUNCS 结构注册的 armMCastAdd，armMCastDel，armMCastGet 函数完成。这三个函数分别完成多播地址的添加，删除或获取。网络设备通常工作在单播模式下，只接收发往本地的数据包，对于广播和多播报文的接收都必须通过设置网络设备的相关控制器来完成，一般而言，广播报文都是默认进行接收的（如 ARP 请求报文），而对于多播报文则默认为全部拒绝。如果要接收一个特定地址的多播报文，则必须对进行明确的配置，开启网络设备控制寄存器中的相关位。armMcastAdd，armMcastDel 除了在软件维护的多播地址列表进行修改外，还需要配置网络设备硬件寄存器。对于多播地址列表的维护，MUX 中间层提供了相关接口函数，方便底层驱动的管理，则网络设备硬件寄存器的配置则必须要由底层驱动完成。armMCastAdd，armMCastDel，armMCastGet 函数实现如下。

```
/*******************************************************************************
* armMCastAdd - add a multicast address for the device
*
* This routine adds a multicast address to whatever the driver
* is already listening for.    It then resets the address filter.
*
* RETURNS: OK or ERROR.
```

```
*/
LOCAL STATUS armMCastAdd(ARM_END_DEVICE *pDrvCtrl,char* pAddress){
	int error;
	if ((error = etherMultiAdd (&pDrvCtrl->end.multiList,pAddress)) == ENETRESET){
		armFlgChange(pDrvCtrl,0);
	}
	return (OK);
}

/*******************************************************************************
* armMCastDel - delete a multicast address for the device
*
* This routine removes a multicast address from whatever the driver
* is listening for.    It then resets the address filter.
*
* RETURNS: OK or ERROR.
*/
LOCAL STATUS armMCastDel(ARM_END_DEVICE *pDrvCtrl,char* pAddress){
	int error;
	if ((error = etherMultiDel (&pDrvCtrl->end.multiList,(char *)pAddress)) == ENETRESET){
		armFlgChange(pDrvCtrl,0);
	}
	return (OK);
}

/*******************************************************************************
*
* armMCastGet - get the multicast address list for the device
*
* This routine gets the multicast list of whatever the driver
* is already listening for.
*
* RETURNS: OK or ERROR.
*/
LOCAL STATUS armMCastGet(ARM_END_DEVICE *pDrvCtrl,MULTI_TABLE* pTable   ){
	return (etherMultiGet (&pDrvCtrl->end.multiList, pTable));
}
```

armMCastAdd 首先调用内核接口函数 etherMultiAdd 完成对 END_OBJ 结构中维护的多播地址列表进行更改，而后调用 armFlgChange 函数完成网络设备控制寄存器的配置，使得网络设备对指定多播地址的报文进行接收。

armMCastDel 也是首先调用内核接口函数 etherMultiDel 完成多播地址列表的更改，此时也是通过底层函数 armFlgChange 函数完成硬件寄存器的配置，使得网络设备对指定多播地址

的报文进行过滤，拒绝接收此类多播报文。
armMCastGet 由于只是信息获取，故无需任何底层操作，其直接使用内核接口函数 etherMultiGet 返回多播地址列表，

对于 armIoctl 以及此处调用的 armFlgChange 函数实现代码如下。

```
/*******************************************************************************
 * armFlgChange
 * configure device according to updated flags, such as changing
 * into promiscuous mode.
 *******************************************************************************/
LOCAL void armFlgChange(ARM_END_DEVICE *pDrvCtrl,UINT32 cmd){
    UINT32 val;
    /* Set up address filter for multicasting. */
    if (lstCount(&((&(pDrvCtrl->end))->multiList)) > 0){
        DRV_LOG(DRV_DEBUG_IOCTL,"armFlgChange-set multicast list!\n",1,2,3,4,5,6);
        armAddrFilterSet (pDrvCtrl);
    }
    if(cmd==0){
        return;
    }
    /* TODO - initialise the hardware according to flags */
    /* Set promiscuous mode if it's asked for. */
    if (END_FLAGS_GET(&pDrvCtrl->end) & IFF_PROMISC){
        REG_READ(EMAC_CTRL_REG_RXMBPENABLE,val);
        val|=(EMAC_RXMBPENABLE_RXCAFEN(1)|
                EMAC_RXMBPENABLE_RXPROMIS(1));
        REG_WRITE(EMAC_CTRL_REG_RXMBPENABLE,val);
    }
    else{
        REG_READ(EMAC_CTRL_REG_RXMBPENABLE,val);
        val&=~(EMAC_RXMBPENABLE_RXCAFEN(1));
        REG_WRITE(EMAC_CTRL_REG_RXMBPENABLE,val);
    }


    /* Check if in polling mode, if it is, disable interrupt signal to cpu.*/
    if((pDrvCtrl->flags&ARM_POLLING)){
        REG_READ(EMAC_CTRL_MODULE_REG_EWCTRL,val);
        val&=~EMAC_GLOBAL_INT_ENABLE_BIT;
        REG_WRITE(EMAC_CTRL_MODULE_REG_EWCTRL,val);
        intDisable(pDrvCtrl->level);
    }
    else{
        REG_READ(EMAC_CTRL_MODULE_REG_EWCTRL,val);
```

```
            val|=EMAC_GLOBAL_INT_ENABLE_BIT;
            REG_WRITE(EMAC_CTRL_MODULE_REG_EWCTRL,val);
            intEnable(pDrvCtrl->level);
        }
}
```

armFlgChange 函数进一步调用 armAddrFilterSet 完成多播报文过滤寄存器的配置。armFlgChange 中其他代码是对工作模式的配置。这些代码比较简单，此处不再阐述。

10.9 查询模式函数

查询模式函数有两个：（1）armPollSend 完成一个数据帧的发送，该函数是 armSend 函数的简化版本，无需对数据帧做底层缓存，采用阻塞发送方式，直到数据帧发送完毕才返回。在 armSend 函数，我们经过层层调用，最后又 armTransPacket 函数完成网络设备的控制将数据帧发送出去，在 armPollSend 函数中我们将直接调用 armTransPacket 函数完成一个数据帧的发送；（2）armPollRcv 完成一个数据帧的接收，该函数每次直接收一个数据帧。由于底层网络设备的数据帧接收都是异步的，实际上在查询工作模式下，数据帧的接收依然是异步的，所谓查询工作模式是指此时对于接收的数据帧，底层驱动不再主动的调用上层数据帧接收接口函数（muxReceive）进行上传，而是上层调用 armPollRcv 一次，才上传一个数据帧，即此时数据帧的上传操作也是采用被动方式。
查询模式在网络通道被用作系统调试通道时被使用，我们在串口驱动一章，也曾讨论到使用串口作为系统调试通道，此时串口驱动也需要实现此处类似的两个查询收发函数，要求都是一致的。相对而言，网络通道使用的概率更大一些，因为其数据传输率相比而言要大得多。

armPollSend 函数实现如下。

```
/*******************************************************************************
* armPollSend - routine to send a packet in polled mode.
*
* Polled mode operation takes place without any kernel or other OS
* services available.  Use extreme care to insure that this code does not
* call any kernel services.  Polled mode is only for WDB system mode use.
* Kernel services, semaphores, tasks, etc, are not available during WDB
* system mode.
*
* A typical implementation is to set aside a fixed buffer for polled send
* operation.  Copy the mblk data to the buffer and pass the fixed buffer
* to the device.  Performance is not a consideration for polled operations.
*
* An alternate implementation is a synchronous one.  The routine accepts
* user data but does not return until the data has actually been sent.  If an
* error occurs, the routine returns EAGAIN and the user will retry the request.
*
```

```
* If the device returns OK, then the data has been sent and the user may free
* the associated mblk.    The driver never frees the mblk in polled mode.
* The calling routine will free the mblk upon success.
*
* RETURNS: OK upon success.    EAGAIN if device is busy or no resources.
* A return of ERROR indicates a hardware fault or no support for polled mode
* at all.
*/

LOCAL STATUS armPollSend(ARM_END_DEVICE*      pDrvCtrl,M_BLK_ID pMblk){
      int len;
      UINT32 curTxIdx = pDrvCtrl ->txDescSendIdx;

      len = pMblk->mBlkPktHdr.len;
      /* Set pointers in local structures to point to data. */
      netMblkToBufCopy(pMblk, (void *)pDrvCtrl ->pTxBufDesc[curTxIdx].pBuffer, NULL) ;

      (pDrvCtrl ->pTxBufDesc)[curTxIdx].pktFlgLen =
                                    EMAC_DSC_FLAG_OWNER|EMAC_DSC_FLAG_SOP|
                                    EMAC_DSC_FLAG_EOP|(len & 0xFFFF);
      (pDrvCtrl ->pTxBufDesc)[curTxIdx].pBufOffLen = (len & 0xFFFF);

      /*send it*/
      armTransPacket(pDrvCtrl);

      if(pDrvCtrl ->txDescSendIdx == (TX_FD_NUM - 1)){
            pDrvCtrl ->txDescSendIdx = 0;
      }
      else{
            pDrvCtrl ->txDescSendIdx ++;
      }
      /*update statistic counter. */
      END_ERR_ADD (&pDrvCtrl->end, MIB2_OUT_UCAST, +1);

    /*
     * Cleanup.    The driver must either free the packet now or
     * set up a structure so it can be freed later after a transmit
     * interrupt occurs.
     */
      netMblkClChainFree (pMblk);
      DRV_LOG(DRV_DEBUG_TX,"armPollSend-Done polling Send!\n",1,2,3,4,5,6);
      return (OK);
}
```

armPollSend 函数是 armSend 函数的一个简化版本，此时底层驱动将不存在缓存的数据帧，每次上层调用 armPollSend 函数发送一个数据帧，这个数据帧都是即时的被发送出去。注意 armPollSend 函数直接调用 armTransPacket 函数完成了数据帧的发送，而不是如同 armSend 函数中通过 tNetTask 任务上下文进行处理。

armPollRcv 函数实现如下。

```
/*******************************************************************************
* armPollRcv - routine to receive a packet in polled mode.
*
* Polled mode operation takes place without any kernel or other OS
* services available.  Use extreme care to insure that this code does not
* call any kernel services.  Polled mode is only for WDB system mode use.
* Kernel services, semaphores, tasks, etc, are not available during WDB
* system mode.
*
* The WDB agent polls the device constantly looking for new data.  Typically
* the device has a ring of RFDs to receive incoming packets.  This routine
* examines the ring for any new data and copies it to the provided mblk.
* The concern here is to keep the device supplied with empty buffers at all
* time.
*
* RETURNS: OK upon success.  EAGAIN is returned when no packet is available.
* A return of ERROR indicates a hardware fault or no support for polled mode
* at all.
*/

LOCAL STATUS armPollRcv(ARM_END_DEVICE * pDrvCtrl,M_BLK_ID pMblk){
    int len;
    UINT32 rxptr;
    STATUS retVal = EAGAIN;
    UINT32 status;

    rxptr = pDrvCtrl ->rxDescIdx;
    status = (pDrvCtrl ->pRecvBufDesc[rxptr].pktFlgLen);
    if((status & EMAC_DSC_FLAG_OWNER ))
        return EAGAIN;

    /* Upper layer must provide a valid buffer. */
    len = (pDrvCtrl ->pRecvBufDesc[rxptr].pktFlgLen&(0xffff));

    if ((pMblk->mBlkHdr.mLen < len) || (!(pMblk->mBlkHdr.mFlags & M_EXT))){
        goto outer;
    }
```

```
        END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_UCAST, +1);
        pMblk->m_len = len;

        /* I am not sure the reasonability of adding 2 here.
          * i am not sure if the kernel has offseted to align ip header.
          * Due to different protocols, kernel cannot assume ethernet header is used,
          * so most probobaly it does nothing offsetting.
          */
        pMblk->mBlkHdr.mData += 2;
        pMblk->mBlkHdr.mFlags |= M_PKTHDR;
        pMblk->mBlkPktHdr.len = len;
        bcopy ( pDrvCtrl ->clRxPointBuf[rxptr], pMblk->m_data, len );

        retVal = OK;

outer:
        if( rxptr == (RX_FD_NUM-1) )
                pDrvCtrl ->rxDescIdx = 0;
        else
                pDrvCtrl ->rxDescIdx ++;

        DRV_LOG(DRV_DEBUG_RX,"armPollRcv-Done polling receive!\n",1,2,3,4,5,6);
        return retVal;
}
```

同理，armPollRcv 函数是 armRecv 函数的一个简化版本，每次被动从接收队列中取一个数据帧传递给上层。armPollRcv 函数实现简单，读者对比着 armRecv 函数应不难理解，此处不多做介绍。

10.10 设备停止和卸载函数

我们在 configNet.h 文件的 endDevTbl 数组中通过注册设备加载函数 armLoad，使得底层驱动在系统网络栈以及 MUX 层初始化中一并被初始化。设备加载函数将 MUX 中间层所需的所有函数通过 NET_FUNCS 结构提供给了 MUX 层，这些函数的调用都是被动进行的。设备启动函数 armStart 在 muxDevStart 函数中被调用，这也是在操作系统启动过程中完成的。虽然对于网络设备而言，通常在操作系统整个运行期间都是出于运转状态，很少有停止甚至卸载网络设备的需求，不过作为驱动实现的一部分，这个功能还是必须提供，以应对某些特殊使用环境。在我们的驱动示例中，设备停止函数为 armStop，设备卸载函数为 armUnload。与设备启动和设备加载函数对应，设备停止函数必须取消设备启动函数的所有操作，而设备卸载函数必须取消设备加载函数的所有操作。

设备停止函数 armStop 函数实现如下。

```
/*******************************************************************************
*
*
* armStop - stop the device
*
* This function calls BSP functions to disconnect interrupts and stop
* the device from operating in interrupt mode.
*
* RETURNS: OK or ERROR.
*/

LOCAL STATUS armStop(ARM_END_DEVICE *pDrvCtrl){
    UINT32 val;
    STATUS result = OK;
    intDisable(pDrvCtrl->level);

    END_FLAGS_CLR (&pDrvCtrl->end, IFF_UP | IFF_RUNNING);

    /* TODO - stop/disable the device. */
    REG_READ(EMAC_CTRL_REG_TXCONTROL,val);
    val&=~EMAC_TXCONTROL_TXEN_BIT;
    REG_WRITE(EMAC_CTRL_REG_TXCONTROL,val);

    REG_READ(EMAC_CTRL_REG_RXCONTROL,val);
    val&=~EMAC_RXCONTROL_RXEN_BIT;
    REG_WRITE(EMAC_CTRL_REG_RXCONTROL,val);

    /*set gmiien bit in maccontrol reg*/
    REG_READ(EMAC_CTRL_REG_MACCONTROL,val);
    val&=~(EMAC_MACCONTROL_GMIIEN(1));
    REG_WRITE(EMAC_CTRL_REG_MACCONTROL,val);

    /* disable global interrupt enable bit*/
    REG_READ(EMAC_CTRL_MODULE_REG_EWCTRL,val);
    val&=~EMAC_GLOBAL_INT_ENABLE_BIT;
    REG_WRITE(EMAC_CTRL_MODULE_REG_EWCTRL,val);


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

```
    return (result);
}
```

armStop 函数实现较为简单，其主要完成两个工作：（1）禁止中断；（2）停止设备。停止设备通过配置网络设备控制寄存器的工作位进行的。

设备卸载函数 armUnload 函数实现如下。

```
/*******************************************************************************
*
* armUnload - unload a driver from the system
*
* This function first brings down the device, and then frees any
* stuff that was allocated by the driver in the load function.
*
* RETURNS: OK or ERROR.
*/

LOCAL STATUS armUnload(ARM_END_DEVICE* pDrvCtrl){
        armStop(pDrvCtrl);
        armReset(pDrvCtrl);

        END_OBJECT_UNLOAD (&pDrvCtrl->end);

        /* TODO - Free any special allocated memory */
        armMemUninit(pDrvCtrl);
        semDelete(pDrvCtrl->txLock);
        /* New: free the END_OBJ structure allocated during armLoad() */
        free ((char *)pDrvCtrl);

        return (OK);
}
```

armUnload 将取消 armLoad 函数中的所有操作，其具体完成：（1）停止设备，重新复位设置到预知状态；（2）释放内存资源以及其他资源；（3）释放设备结构。
设备结构的释放表示了设备的消亡，此时系统将无任何手段可以访问到设备，要访问设备，必须重新调用设备加载函数。

10.11 内核接口函数

在进行 NET_FUNCS 结构初始化时，除了底层实现的函数外，我们还使用了内核提供的三个函数提供给 MUX 中间层，这三个函数是 endEtherAddressForm，endEtherPacketDataGet，endEtherPacketAddrGet。

（1） endEtherAddressForm

该函数完成以太网链路层首部的创建工作。函数调用原型如下。

```
M_BLK_ID endEtherAddressForm
    (
    M_BLK_ID pMblk,         /* pointer to packet mBlk */
    M_BLK_ID pSrcAddr,    /* pointer to source address */
    M_BLK_ID pDstAddr,    /* pointer to destination address */
    BOOL bcastFlag          /* use link-level broadcast? */
    );
```

（2） endEtherPacketDataGet

该函数解析接收的数据帧，返回数据帧的链路层头部各字段信息。具体的是通过解析数据帧内容，初始化一个 LL_HDR_INFO 结构，返回给调用该函数的上层使用。

该函数调用原型如下。

```
STATUS endEtherPacketDataGet
    (
    M_BLK_ID                pMblk,
    LL_HDR_INFO *           pLinkHdrInfo
    );
```

参数 1 为需要解析的数据帧，参数 2 则是需要 endEtherPacketDataGet 进行初始化的 LL_HDR_INFO 结构，该结构中存储着链路层首部各字段的信息。LL_HDR_INFO 结构定义在 h/end.h 内核头文件中，如下所示。

```
/*
* h/end.h
* LL_HDR_INFO - link level Header info
*
* This data structure defines information that is only relevant to
* a stack receive routine.
*
*/
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;
```

（3） endEtherPacketAddrGet

该函数被用以定位数据帧中链路层首部中发送端和目的端的 MAC 地址位置。该函数调用原型如下。

```
STATUS endEtherPacketAddrGet
    (
    M_BLK_ID pMblk, /* pointer to packet */
    M_BLK_ID pSrc,   /* pointer to local source address */
    M_BLK_ID pDst,   /* pointer to local destination address */
    M_BLK_ID pESrc, /* pointer to remote source address (if any) */
    M_BLK_ID pEDst   /* pointer to remote destination address (if any) */
    );
```

endEtherPacketAddrGet 通过解析 pMblk 指向的数据帧，初始化余下的四个参数。其中 pSrc，pESrc 指向数据帧源端的 MAC 地址，pDst，pEDst 指向数据帧目的端的 MAC 地址。

对于以太网或者 802.3 链路层协议，都可以使用如上所示的三个内核函数来初始化 NET_FUNCS 结构中的 formAddress，packetDataGet，addrGet 三个字段，对于链路层其他协议，则需要底层驱动根据具体协议的定义来实现这个三个函数。

10.12 底层驱动实现小结

至此，围绕底层驱动初始化以及与 MUX 中间层之间的接口需求，我们已经完成所有相关函数以及数据结构的介绍，涉及的函数如下：

（1） armLoad 函数：底层驱动和设备加载函数，完成底层驱动注册以及网络设备所有初始化工作。

（2） armUnload 函数：驱动和设备卸载函数，取消 armLoad 函数中的所有操作。

（3） armMemInit 函数：mBlk，clBlk 以及底层缓冲区 Cluster 结构池分配函数。

（4） armStart 函数：底层驱动和设备启动函数，完成中断注册和使能以及设备启动的工作，该函数调用后，底层网络设备将进入正常的网络数据帧收发状态。

（5） armStop 函数：底层驱动设备停止函数，完成中断禁止以及设备停止工作的设置，该函数调用后，网络设备将处于“禁止”状态，不进行任何数据帧的收发工作。

（6） armInt 函数：底层驱动中断响应函数，主要完成网络数据帧的异步接收。

（7） armHandleRcvInt 函数：数据帧接收过渡函数，该函数处理接收队列，具体调用 armRecv 函数完成数据帧的接收，该函数工作在 tNetTask 任务上下文中。

（8） armRecv 函数：数据帧上传函数，该函数完成网络数据帧的封装工作，封装成网络栈需要的结构（mBlk）形式，并通过 MUX 中间层提供的接口函数（muxReceive）将数据帧传递给网络栈进行处理。

（9） armSend 函数：数据帧发送函数，该函数将被 MUX 中间层调用用以进行所有数据帧的发送工作。所有网络栈需要发送的网络数据帧都必须通过调用该函数进行发送。该函数具体调用 armHandleTxInt 函数完成底层处理。

（10） armHandleTxInt 函数：该函数在 armSend 中被安排在 tNetTask 任务上下文中运行，处理发送队列，调用 armTransPacket 函数完成一个数据帧的具体发送。

（11） armTransPacket 函数：其接收一个数据帧，操作网络设备，将这个数据帧最终发送到网络介质上。

（12） armIoctl 函数：网络设备参数以及工作状态设置和获取函数。
（13） armPollRcv，armPollSend 函数：查询模式数据帧收发函数，当网络通道被用于系统调试通道时，这两个函数将被使用用作数据帧的接收和发送。
（14） armMCastAdd，armMCastDel，armMCastGet 函数：多播地址配置和获取函数，用以控制网络设备是否接收特定地址的多播报文。
（15） 其他硬件操作函数：这些函数被以上函数调用完成配置网络设备硬件寄存器完成一个具体的功能。

对照以上函数，我们再次给出底层驱动对 NET_FUNCS 结构的初始化，读者对比以上函数定义查看 Vxworks 下底层网络设备驱动的实现基本框架。

```
/*
 * Declare our function table.    This is static across all device instances.
 */
LOCAL NET_FUNCS armFuncTable ={
      (STATUS (*) (END_OBJ*))armStart,          /* Function to start the device. */
      (STATUS (*) (END_OBJ*))armStop,           /* Function to stop the device. */
      (STATUS (*) (END_OBJ*))armUnload,             /* Unloading function for the driver. */
      (int (*) (END_OBJ*, int, caddr_t))armIoctl,       /* Ioctl function for the driver. */
      (STATUS (*) (END_OBJ* , M_BLK_ID))armSend,            /* Send function for the driver. */
      (STATUS (*) (END_OBJ*, char*))armMCastAdd, /* Multicast add function for the driver. */
      (STATUS (*) (END_OBJ*, char*))armMCastDel, /*Multicast delete function. */
      (STATUS (*) (END_OBJ*, MULTI_TABLE*))armMCastGet,/*Multicast retrievefunction. */
      (STATUS (*) (END_OBJ*, M_BLK_ID))armPollSend,             /* Polling send function */
      (STATUS (*) (END_OBJ*, M_BLK_ID))armPollRcv,      /* Polling receive function */
      endEtherAddressForm,          /* put address info into a NET_BUFFER */
      endEtherPacketDataGet,        /* get pointer to data in NET_BUFFER */
      endEtherPacketAddrGet             /* Get packet addresses. */
};
```

10.13 IP 地址和 MAC 地址

对于一个网络设备必须具有确定的 MAC 地址，虽然 MAC 地址并不要求全球唯一性，但是必须在一个局域网内是唯一的。PC 下网络设备的 MAC 地址是固化在设备内的，在设备出厂时就已经确定，一般不允许修改。而对于嵌入式系统而言，网络接口 MAC 地址通常需要由用户指定，这通过配置网络设备 MAC 地址寄存器完成。对于以太网，指定一个六个字节的长的普通数字字符串即可。在我们的 BSP 设计中，我们使用 bootline 传递 MAC 地址，默认 bootline 定义在 config.h 文件中，如下所示。

```
#define DEFAULT_BOOT_LINE "atemac(0,0) vxw:vxWorks h=192.168.1.6 e=192.168.1.254:ffffff00 g=192.168.1.1 u=www pw=www tn=AT91RM9200 o=00:00:0a:0b:0d:00"
```

其中 o=00:00:0a:0b:0d:00 指定的就是网络设备的 MAC 地址。在 Vxworks 启动过程中，将存在一个倒计时的阶段，当用户键入任何键时，其将处理 bootline 配置状态，运行用户即时更改 bootline 函数，包括主机服务器 IP 地址，目标机 IP 地址等等。我们一般通过将 bootline 存储到平台 EEPROM 等 NVRAM 中来保存用户修改后的 bootline 参数，这样下次 Vxworks 内核启动时将使用新的 bootline 参数。
IP 地址也是通过 bootline 参数进行指定的，这是由 bootline 中 e=192.168.1.254:ffffff00 语句指定的，其中包括了 IP 地址及其子网掩码。

读者可以查看前文中 armLoad 函数实现，在该函数中我们调用了一个 armGetMacAddr 函数，从存储在 NVRAM 中的 bootline 中获取 MAC 地址，该函数实现如下。

```
LOCAL void armGetMacAddr(){
    BOOT_PARAMS Params;
    int b1,b2,b3,b4,b5,b6;

    sysinitBootLine();/*self-definded in sysLib.c*/
    bootStringToStruct(sysBootLine, &Params);
    if(sscanf(Params.other, "%x:%x:%x:%x:%x:%x", &b1, &b2, &b3, &b4, &b5, &b6) != 6){
        return;
    }
    armEndEnetAddr[0]=(char)(b1);
    armEndEnetAddr[1]=(char)(b2);
    armEndEnetAddr[2]=(char)(b3);
    armEndEnetAddr[3]=(char)(b4);
    armEndEnetAddr[4]=(char)(b5);
    armEndEnetAddr[5]=(char)(b6);
}
```

其中 sysinitBootLine 从 NVRAM 中读取 bootline，存储到 sysBootLine 全局变量中，armGetMacAddr 使用 bootStringToStruct 内核接口函数对 bootline 行进行解析，将各项参数解析到 BOOT_PARAMS 结构各对应字段中，其中 BOOT_PARAMS 中 other 成员就对应 o=00:00:0a:0b:0d:00 语句，我们使用 sccanf 函数将 bootline 中指定的 MAC 地址读取到底层驱动中定义的 armEndEnetAddr 数组中，而后在 armLoad 函数中使用 SYS_ENET_ADDR_GET 宏将这个 MAC 地址复制到 END_OBJ 结构中，作为网络设备的 MAC 地址使用，这个地址一方面用于配置网络设备 MAC 地址寄存器，另一方面被 MUX 层使用用以创建本地发送的数据帧的链路层首部。
而 bootline 中指定的目标机 IP 地址将被用以绑定网络栈中网络层实现，用于本地发送数据帧的源 IP 地址，与外界进行通信。
BOOT_PARAMS 结构定义在 h/bootLib.h 内核头文件中，如下所示，读者可以对比着 bootline 行的定义，很容易的看出 bootline 行中各参数与 BOOT_PARAMS 结构中各成员之间的对应关系。

```
typedef struct                  /* BOOT_PARAMS */
{
    char bootDev [BOOT_DEV_LEN];      /* boot device code */
    char hostName [BOOT_HOST_LEN];  /* name of host */
```

```
    char targetName [BOOT_HOST_LEN];      /* name of target */
    char ead [BOOT_TARGET_ADDR_LEN];  /* ethernet internet addr */
    char bad [BOOT_TARGET_ADDR_LEN];  /* backplane internet addr */
    char had [BOOT_ADDR_LEN];         /* host internet addr */
    char gad [BOOT_ADDR_LEN];         /* gateway internet addr */
    char bootFile [BOOT_FILE_LEN];    /* name of boot file */
    char startupScript [BOOT_FILE_LEN];      /* name of startup script file */
    char usr [BOOT_USR_LEN];      /* user name */
    char passwd [BOOT_PASSWORD_LEN];   /* password */
    char other [BOOT_OTHER_LEN];      /* available for applications */
    int    procNum;                   /* processor number */
    int    flags;                     /* configuration flags */
    int    unitNum;                              /* network device unit number */
} BOOT_PARAMS;
```

10.14 多网口支持

前文介绍中，我们的底层驱动只能支持一个网络设备，如果需要同时支持多个网络设备，则需要对驱动结构进行部分修改，同时涉及网络设备的相关文件也需要进行更改，下面我们将从三个方面：（1）底层驱动；（2）configNet.h 文件；（3）usrNetInit 函数，进行介绍如何支持多网口。

10.14.1 底层驱动修改

每个网络设备必须对应一个独立的 END_OBJ 结构，也即一个驱动自定义结构。涉及到设备的资源必须进行独立定义，不可直接全局变量的形式定义，如此便于进行代码的设计和管理。基于以上我们的驱动示例，下面几个变量需要从全局转为驱动自定义结构中的一个成员。

（1） M_CL_CONFIG 结构变量
（2） CL_DESC 结构变量
（3） MAC 地址变量

对于单网口驱动，我们在以上示例中，将这三个变量都定义为全局的（虽然 MAC 地址作为了一个结构成员，但是没有使用，而是使用一个全局的变量，具体参考前文中给出的 armGetMacAddr 函数实现）。

为了对多网口进行支持，我们必须重新定义驱动自定义结构，将以上全局变量转为结构成员。另外为了在 armLoad 函数中区分各个网络设备，我们有一个 unit 成员。这个成员的值将通过 endDevTbl 数组 END_TBL_ENTRY 结构成员的 endLoadString（这是一个驱动自定义参数字符串）传入。

更改后 AMR_END_DEVICE 结构如下。

```
typedef struct arm_end_device
{
    END_OBJ         end;            /* The class we inherit from. */
    int             unit;       //网络设备标识成员。
    …
    UCHAR           enetAddr[6];   /* ethernet address */
    M_CL_CONFIG     mclBlkConfig;
    CL_DESC         clDescTbl;
    …
} ARM_END_DEVICE;
```

此时 armMemInit 函数将对结构中 mclBlkConfig 以及 clDescTbl 成员进行初始化，而不是对全局变量 armMclBlkConfig 以及 armClDescTbl 进行初始化；而 armGetMacAddr 函数将对 enetAddr 成员进行初始化，而不是对全局变量 armEndEnetAddr 进行初始化。

对于如何确定多网口的 MAC 地址，可以通过对以上单个网口 MAC 地址做简单的变换算法进行。我们通过 bootline 传递一个 MAC 地址，而后各个网口的 MAC 地址将通过这个 bootline 中指定的 MAC 地址的不同组合得到。
armGetMacAddr 此时的实现代码如下所示（以双网口为例）。

```
LOCAL void armGetMacAddr(ARM_END_DEVICE *pDev){
    BOOT_PARAMS Params;
    int b1,b2,b3,b4,b5,b6;

    sysinitBootLine();/*self-definded in sysLib.c*/
    bootStringToStruct(sysBootLine, &Params);
    if(sscanf(Params.other, "%x:%x:%x:%x:%x:%x", &b1, &b2, &b3, &b4, &b5, &b6) != 6){
        return;
    }
    if(pDev->unit==0){//第一个网口
        armEndEnetAddr[0]=(char)(b1);
        armEndEnetAddr[1]=(char)(b2);
        armEndEnetAddr[2]=(char)(b3);
        armEndEnetAddr[3]=(char)(b4|b3);
        armEndEnetAddr[4]=(char)(b5|b4);
        armEndEnetAddr[5]=(char)(b6|b5);
    }
    else if(pDev->unit==1){//第二个网口
        armEndEnetAddr[0]=(char)(b1);
        armEndEnetAddr[1]=(char)(b2);
        armEndEnetAddr[2]=(char)(b3);
        armEndEnetAddr[3]=(char)(b4^b3);
        armEndEnetAddr[4]=(char)(b5^b4);
```

```
            armEndEnetAddr[5]=(char)(b6^b5);
        }
}
```

以上所示只是一种简单的示例，如果存在两个网口工作于同一个局域网中，必须确保这两个网口的 MAC 地址不同，如果各个网口工作在不同的局域网中，则即便 MAC 地址巧合重复，也不会对实际工作造成任何影响。

除了以上明显的修改外，对于底层驱动实现的所有函数，此时都需要根据设备号的不同操作不同的网络设备硬件，或者对不同设备实现为不同的函数，在进行 NET_FUNCS 结构的初始化时指定不同函数集合即可。

10.14.2 configNet.h 文件修改

configNet.h 文件定义有 endDevTbl 这个重要的数组，我们必须在其中添加网口加载函数，每个网口设备对应一个加载函数，虽然通常对于同一类型的设备我们使用的入口函数，但是其他参数如 unit，endLoadString 等都会不同，如下所示为双网口支持时 endDevTbl 的定义。

```
#ifndef _CONFIGNET_H
#define _CONFIGNET_H

#ifdef __cplusplus
extern "C" {
#endif

#ifdef INCLUDE_END /* ignore everything if NETWORK not included */

#include "vxWorks.h"
#include "end.h"

/* This template presumes the template END driver */

#define ARM_LOAD_FUNC       armLoad
#define ARM_BUFF_LOAN       1

/* From the light of armParse routine's accomplishment, this string is self-defined.
  * so i can define params meaningful to myself as the following format:
  *<unit>: <ether_speed>:<duplex>:<auto_nego>
  */
#define   ARM_LOAD_STRING0      "0:100:1:1"
#define   ARM_LOAD_STRING1      "1:100:1:1"

IMPORT END_OBJ * ARM_LOAD_FUNC (char *,void *);
```

```
/*called in userRoot function of bootConfig.c*/
END_TBL_ENTRY endDevTbl [] ={
   { 0, ARM_LOAD_FUNC, ARM_LOAD_STRING0, ARM_BUFF_LOAN, NULL,FALSE},
   { 1, ARM_LOAD_FUNC, ARM_LOAD_STRING1, ARM_BUFF_LOAN, NULL,FALSE},
   { 0, END_TBL_END, NULL, 0, NULL,FALSE},
};

#undef     IP_MAX_UNITS
#define    IP_MAX_UNITS      2

#endif /* INCLUDE_END */

#ifdef __cplusplus
}
#endif

#endif /* _CONFIGNET_H */
```

注意：endDevTbl 数组中每个元素的第一个成员也是表示网络设备的编号，这个编号将被用于初始化 END_OBJ 结构中的 devObject.unit 成员。实际上这个 devObject.unit 的值与 AMR_END_DEVICE 结构中 unit 的值是相同的。

为了对多网口进行支持，此时在 configNet.h 文件中必须新添加一个宏定义，这个宏为 IP_MAX_UNITS，表示 IP 地址的数目，如果每个网口对应一个 IP 地址，那么这个就定义为网口的数目，通常而言都是如此。由于 IP_MAX_UNITS 已在其他文件（target/src/config/usrNetwork.c）中进行了定义，故为了修改其定义，我们必须先进行 undef，而后进行 define。

10.14.3 usrNetInit 函数修改

使用多个网口时，很多时候并非是为了在两个网口均衡数据包收发，而是每个网口工作在不同的网段上，即具有不同的 IP 地址。此时就必须在 usrNetInit 函数添加语句，对不同的网络设备绑定不同的 IP 地址。由于通过 bootline 只能传递一个 IP 地址，故我们可以在 config.h 文件中以宏定义的形式指定不同网口的 IP 地址，如下所示双网口的情景。

```
/*config.h*/
#define END_DEV0_IP       “192.168.0.2:ffffff00”
#define END_DEV1_IP       “192.168.1.2:ffffff00”
```

除了指定每个网口设备的 IP 地址外，我们还必须将这个 IP 地址绑定到网络栈网络层（IP 协议）实现中，这涉及到如下两个函数的调用。

（1） ipAttach 函数：绑定网络设备到网络层实现。

（2） usrNetIfAttach：绑定 IP 地址到网络层实现。

对于每个网络设备，都必须调用 ipAttach 和 usrNetIfAttach 函数。
ipAttach 函数调用原型如下。

```
int ipAttach
    (
    int unit,                          /* Unit number    */
    char *pDevice       /* Device name (i.e. ln, ei etc.). */
    );
```

参数 1 指定设备号，参数 2 指定设备名，这两个参数合并成一个网络设备名，用以查询底层驱动，底层驱动将根据设备号的不同对不同的设备进行操作。

usrNetIfAttach 函数调用原型如下。

```
STATUS usrNetIfAttach
    (
    char *      devName,
    int    unitNum,
    char *      inetAdrs
    );
```

参数 1 指定设备名，参数 2 指定设备号，二者共同组成网络设备名称，参数 3 指定与设备相绑定的网络层协议如 IP 协议的地址。

10.15 本章小结

本章我们着重介绍了 Vxworks 网络设备驱动的设计和实现，我们以 DM6446 平台下网络接口设备驱动代码为例，详细的介绍了 Vxworks 下 END 网口驱动的各个方面。网络设备作为设备分类中的三大设备之一，与其他两种设备相比，具有完全不同的操作接口和数据接收模式，其内核层次结构也与字符和块设备完全不同。读者在进行网络设备驱动设计时，必须同时从内核接口以及底层硬件操作两个方面入手进行。内核接口主要从 END_OBJ，NET_FUNCS 两个顶层结构以及 mBlk，clBlk，clBuff 缓冲结构出发进行理解，而硬件设备本身则需要读者仔细研读网络设备手册，了解网络设备的内部工作方式，如此方可顺序完成网络设备驱动的开发工作。
在驱动的设计调试中，必须特别注册中断相关的内容，这是最容易引起问题的地方，如果一个驱动一旦被加载，就造成整个系统的死机和崩溃，则出问题的可能性最大的地方就是中断没有处理好。笔者在调试一个网络设备的驱动时，就遇到过此类问题，最后发现将网络设备的中断号弄错了，改正过来了，一切正常。笔者的经验是，不可以想当然的认为某个地方一定不会出问题，如果始终调试不出问题在哪儿，建议从最基本的项开始一项一项进行排查，包括自己起先确定一定不会出现问题的地方。
本章最后，我们简单的介绍了一下对于多网口的支持，一般在交换机之类（通常基于 PowerPC 体系结构，使用 FCC 驱动网口）的设计中，需要使用到多网口，读者可根据本章介绍的内容进行多网口支持代码的设计，对于疑问的地方，读者应仔细分析相关代码进行理解。
在本章的示例中，我们基于的是 TI 公司的 DM6446 平台上的网络接口驱动，本章给出的所有代码都是实际中在工作的代码，是经过了长时间运行验证的，确定没有问题的。其中一些

函数（armMemInit，armRecv，armSend 等等）的实现代码具有通用性，用户可以拷贝这些代码到自己的驱动中，做简单修改后，即可直接使用。

第十一章 USB 设备驱动

在上一章中完成网络设备驱动的介绍后，本章我们将对 USB 设备驱动进行阐述。其实将二者进行一下对比，会发现网络设备和 USB 设备在抽象层次上具有很多的共同点。如下图 11-1 所示二者之间内核层次对比。

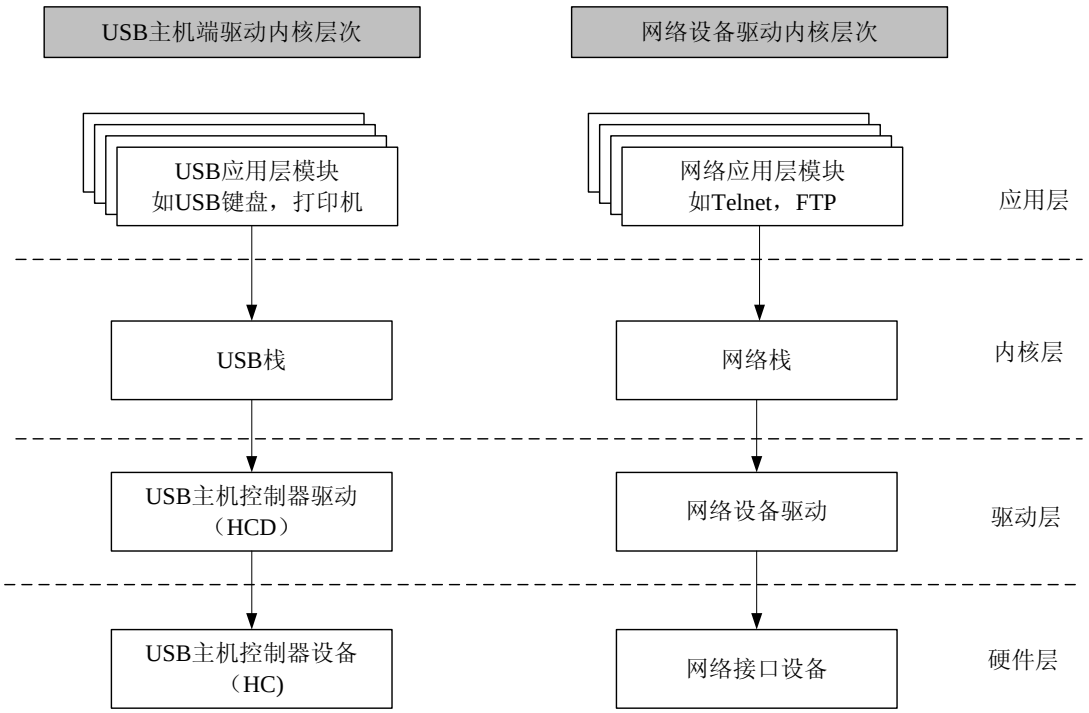


图 11- 1 USB 设备与网络设备内核驱动层次比较

在具体介绍 USB 设备驱动设计之前，有必要先对 USB 本身进行一下介绍，虽然当前有很多涉及 USB 方面的资料，但是如果没有接触过 USB 控制器底层驱动代码的实际编写，对于规范上或者其他资料上给出的很多概念还是无法真正理解。相比网络栈实现，USB 栈实现要简单的多，不过 USB 主机端控制器（HC：Host Controller）驱动的设计和实现要比网络设备驱动复杂的多，其复杂性主要体现在与内核 USB 栈的紧密耦合关系上，故要完成 USB HC 驱动的编写，必须对 USB 栈实现本身有一个比较彻底的了解，Vxworks 下 USB 栈内核实现代码都是以源码方式提供，这就为自实现 HC 驱动提供了可能。

11.1 何为 USB

通用串行总线（USB）仅仅是一个总线接口，定义了一套数据传输的方式和协议。除去 USB 本身，我们可以将 USB 通信双方看作一对 FIFO，USB 规范定义了这一对 FIFO 之间数据传输的所需进行的各种配置和通信方式。从系统软件软件的角度，则只需从 FIFO 的角度进行理解即可。底层驱动（Host Controller 驱动）通过中断被通知 FIFO 中接收到另一端发送的

数据或者被通知 FIFO 中数据已经被发送，需要向 FIFO 中填入数据准备下一次的发送，仅此而已，所有的发送和接收细节由称为 USB 控制器的硬件完成。

对于 USB 规范中定义的需要默认控制端点（EndPoint）响应的各种请求（如 set_configuration, set_interface 等等），则通过 USB 控制器后端的系统软件解析这些请求，并将合适的数据写入发送 FIFO，由 USB 控制器发送到主机完成请求的过程。所以 USB 规范中定义的各种描述符（device，configuration，interface，endpoint）是由系统软件维护的，而不是硬件自动响应的。USB 仅仅是一个接口，用于数据的传送，其并不对数据本身进行解析和处理。通常我们将这些请求以及对这些请求的响应与 USB 本身混为一谈，是因为很多 USB 资料上对此不加区分的进行介绍，造成了很多 USB 新手认为这些请求以及对这些请求的响应是由 USB 本身完成的。而这是大错特错的，如此理解方式人为的复杂化了 USB 协议本身。

对于 USB 规范中定义的各种描述符一般存放在非易失介质上如 EEPROM，当主机被（Hub）通知有一个新的 USB 硬件连接到 USB 总线上时，其发送标准请求到这个新连接的 USB 硬件，读取各种描述符，从而获知该硬件具备的基本功能。这些请求通常为一个长度为 8 字节的内容，USB 硬件中内嵌的系统软件通过对这些内容进行解析，之后从非易失介质上读取相应的描述符，写入 USB 硬件的 FIFO 中，由 USB 硬件中的 USB 控制器将这些描述符通过 USB 总线发送给主机。

下面以最常见的 U 盘为例介绍 USB 相关的各个方面。

一个普通的 U 盘其内部结构一般为 USB 接口，USB 控制器，非易失存储介质，系统软件，微控制器；如下图 11-2 所示（图 11-2 中 USB controller 应包含 USB 控制器和微控制器两部分）。

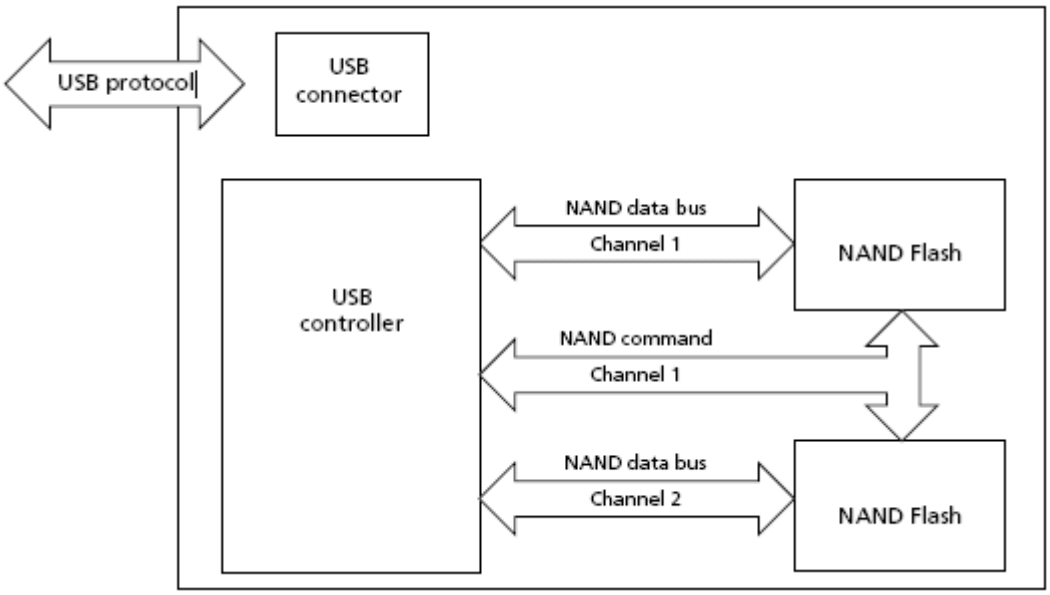


图 11- 2 U 盘内部典型结构

此处系统软件起着至关重要的作用。U 盘对应主机上的驱动一般称为 Mass Storage Class 驱动。U 盘内嵌的系统软件必须与主机上的 Mass Storage Class 驱动协调工作方能达到在 U 盘上进行文件存储的效果，即可以在 U 盘创建文件，像普通文件一样操作 U 盘上的数据。

以 Bulk-Only 协议为例，主机与 Mass Storage Device（MSD）的数据交换必须按照特定的格式，否则无法达到将某个文件写入到 U 盘中特定的位置上。U 盘中的存储介质一般为 NandFlash。对于使用 USB 接口的数据传输，必须在传输的数据中表明每次读写的数据量，Nandflash 起始 Block，此后由内嵌于 U 盘中的 Nandflash 驱动将数据写入或读出。

换句话说，由于 USB 传输数据是以少量 Byte 为单位的，无法像通常对 Nandflash 的读写那样整块的进行，而且对 Nandflash 进行读写，必须指定是从哪个 Block 开始。故为了通过 USB 总线（接口）操作 U 盘，并在文件系统之上进行 U 盘的读写，必须每次读写时指定 U 盘中 Nandflash 的起始 Block 和需要操作的 Block 数目。一般而言，U 盘中文件系统界面是由主机上操作系统维护的，其通过文件系统中间层将 U 盘中 Nandflash 模拟为普通硬盘（二者最大的差别是每个block 的大小差异很大，硬件为 512B/Block，Nandflash 一般为 128KB/Block）。在进行 U 盘数据的读写时，主机文件系统以 Nandflash 中整个 Block 的方式进行读写。但 USB 每个 Packet 大小一般只有几十 Byte，为了解决这之间的差异，此时 U 盘中内嵌的系统软件以及主机中 MSD Client 驱动就起着重要作用。首先，主机上 MSD Client 驱动每次读写一定是以 Nandflash 中整块 Block 进行的，而 U 盘中内嵌的系统软件则通过 USB 总线接收数据，直到接收完整一个或几个 Block 数据，再进行 Nandflash 的写入。此时 USD Client 驱动会通过一个称为“setup”的 USB Packet 通知 Nandflash 写入的起始 Block 号。

本质上，USB 仅仅完成数据的传输，至于数据内容的解释则完全由 USB 外设的功能决定。实现某一功能的 USB 外设和主机上实现相同功能的 Client 驱动都遵循某种协议进行数据的封装和解析，从而共同完成某一功能的实现。而 Mass Storage 则是最为常见的一种功能。

以 OUT 为例，当主机需要向一 Mass Storage USB 外设（如 U 盘）写入数据时，其首先发送一个 CBW（Control Block Wrapper）USB Packet，MSUSB 外设从内部 USB 控制器 FIFO 中读取这个 CBW，以确知主机本次要写入的起始 Block 号和写入的总数据量，此时 MSUSB 外设等待主机发送要写入的数据，由于一般写入的数据较多，故将分为多个 USB Packet 进行发送，MSUSB 外设将每次从内部 USB 控制器 FIFO 中读取一个 Packet 并进行缓存，直到 CBW 中声称的数据量完全接收完毕。而主机在成功发送 CBW 后（接收到外设返回 ACK），将发送要写入 MSUSB 外设的数据，直到发送完其所声称的数据量。此后主机发送一个 CSW Packet，用以询问 MSUSB 外设是否成功接收到所有的数据，MSUSB 相应的返回一个 USB Packet 用以响应，表示其接收数据的最终状态。MSUSB 在接收到所有主机要写入的数据后，通过系统软件 Nandflash 驱动将数据写入之前 CBW Packet 中指定的起始 Block 号开始的块中，从而完成一次 U 盘的写入工作。
对于 IN 操作，同样以一个 CBW Packet 开始，不过此时指定 MSUSB 外设向主机发送数据，MSUSB 系统软件根据 CBW Packet 指定的起始 Block 以及需要读取的数据量，由 Nandflash 驱动读取相应的数据，分多次写入内部 USB 控制器 FIFO 中，启动 USB 控制器通过 USB 纵向分多个 Packet 将数据发送给主机。主机在接收完数据后（或者被中断接收），最后同样发送一个 CSW Packet 询问数据接收的最终状态，以确定本次读取是否成功。

如下是关于 USB 理解关键的几点：

1. USB 仅仅是数据传输的接口，其本身不对数据内容进行解析。数据的解析和应答由 USB 控制器后面的系统软件（驱动）完成。系统软件从 USB 控制器 FIFO 中读取请求和数据进行解释和响应。
2. USB 外设各种功能的实现由主机上称为 USB Client 驱动和外设中内嵌的系统软件配合完成。
3. USB 规范中定义的各种请求以及相关数据存储在 USB 外设非易失介质上，主机对这些数据的请求由 USB 外设系统软件进行应答：从非易失介质中读取合适的数据并通过 USB 控制器发送给主机。
4. 主机和 USB 外设各自具有一个 USB 控制器完成通过 USB 总线的数据交换。主机软件和

USB 外设中内嵌系统软件通过 FIFO 和中断与各自的 USB 控制器进行数据交换。

5. USB 外设一般结构为：USB 接口，USB 控制器，微控制器（如常见的 8051），系统软件（或称固件，firmware），非易失存储介质。通常 USB 控制器和微控制器作为一个部分出现，名称上却仍然叫做 USB 控制器，这容易引起误解。对于 Mass Storage 类 USB 外设，则存在大容量 Nandflash 作为数据存储介质，其他 USB 外设一般为较少量的 EEPROM 存储各种 USB 规范中要求的描述符参数。为了在微控制器（如 8051）中运行系统软件，USB 外设一般包含少量的系统 RAM。

在嵌入式系统下，USB 驱动一般而言是指 USB 控制器驱动，而非 Client 驱动。Client 驱动在 USBD 核心驱动之上，一般操作系统（如 Vxworks）对此的支持比较完善，无须用户作任何更改；而 USB 控制器在嵌入式系统下则五花八门，一般通用的操作系统很难正好包含相应的 HC（Host Controller）驱动，故必须由嵌入式操作系统移植人员完成。HC 驱动在 USBD 核心驱动之下，其驱动 USB 控制器（即 HC）通过 USB 总线进行数据的传输，主要是硬件寄存器的控制和控制器 FIFO 的读写以及中断的响应。

关于几种描述符之间的关系：

1. 每个 USB 外设必须有且只有一个 device descriptor。 device descriptor 描述整个 USB 外设，包含的信息是针对整个 USB 外设，如默认控制管道（EP0）的 Packet 大小。devcie descriptor 的基本结构如下。对于各个字段的解释参见 USB2.0 规范第 9 章。

```
typedef struct usb_device_descr
    {
    UINT8 length;              /* bLength */
    UINT8 descriptorType;      /* bDescriptorType */
    UINT16 bcdUsb;             /* bcdUSB - USB release in BCD */
    UINT8 deviceClass;         /* bDeviceClass */
    UINT8 deviceSubClass;      /* bDeviceSubClass */
    UINT8 deviceProtocol;      /* bDeviceProtocol */
    UINT8 maxPacketSize0;      /* bMaxPacketSize0 */
    UINT16 vendor;             /* idVendor */
    UINT16 product;            /* idProduct */
    UINT16 bcdDevice;          /* bcdDevice - dev release in BCD */
    UINT8 manufacturerIndex;       /* iManufacturer */
    UINT8 productIndex;        /* iProduct */
    UINT8 serialNumberIndex;       /* iSerialNumber */
    UINT8 numConfigurations;       /* bNumConfigurations */
    } WRS_PACK_ALIGN(4) USB_DEVICE_DESCR, *pUSB_DEVICE_DESCR;

 #define USB_DEVICE_DESCR_LEN  18
```

2. 配置描述符（configuration descriptor）描述 USB 外设的实现的功能。一个 USB 设备必须至少有一个配置描述符。主机对配置描述符进行请求时，USB 外设返回配置描述符内容外，还同时返回该配置下的所有 interface 描述符以及 endpoint 描述符。可以认为 configuration 描述符和 interface，endpoint 描述是相邦定的。一种配置体现了 USB 外设一种功能实现方式。如某个外设在一种配置下实现为鼠标，在另一种配置却实现为键盘。

每种配置下可以有多个 interface，每种 interface 实现一种子功能。在一个 interface 下可以有多个 endpoint，表示该子功能下有多个数据传输节点。Endpoint 可以理解为网络中的套接字，它仅仅是主机与 USB 外设之间进行通信的一种手段。我们将主机与 USB 外设某个 endpoint 之间建立的数据传输通道称为管道（Pipe）。USB 外设遵循以 endpoint 进行通信的方式，为每个 endpoint 都分配数据 FIFO 进行数据缓存，或者说使用 FIFO 进行 USB 外设中内部 USB 控制器与内嵌系统软件之间的数据交换。所以 endpoint 仅仅是一个叫法，它表示了一个主机与 USB 外设之间数据通信的节点，每个 endpoint 都有一个地址，USB 外设在接收请求时将检查这个 endpoint 地址，将数据转发到对应的 endpoint 的 FIFO 中，供内嵌的系统软件作相应的处理。

configuration descriptor 结构定义如下。对相关字段的解释请参见 USB2.0 规范第 9 章。

```
typedef struct usb_config_descr
    {
    UINT8 length;           /* bLength */
    UINT8 descriptorType;       /* bDescriptorType */
    UINT16 totalLength;         /* wTotalLength */
    UINT8 numInterfaces;        /* bNumInterfaces */
    UINT8 configurationValue;       /* bConfigurationValue */
    UINT8 configurationIndex;       /* iConfiguration */
    UINT8 attributes;           /* bmAttributes */
    UINT8 maxPower;             /* MaxPower */
    } WRS_PACK_ALIGN(4) USB_CONFIG_DESCR, *pUSB_CONFIG_DESCR;

 #define USB_CONFIG_DESCR_LEN 9
```

3. 接口描述符（interface descriptor）必须作为某个 configuration 描述符的一部分，或者说接口仅仅是 USB 外设某种配置下实现的一个子功能。配置表示了 USB 外设较大方面实现的功能，而接口则表示这个大的功能下实现的某一个小的子功能，而包含在 interface 下的 endpoint 则是数据传输的节点。每个 configuration 可以有多个 interface，即每个实现的大的功能下可以有多个子功能，最少是一个。配置表示的是大的方面，只有到接口这个层次才进行具体功能的描述，故虽然 configuration 描述符表示的是 USB 外设实现的大的方面的功能，这只是为便于理解的一种说法，只有在 interface 描述符中方才对功能类进行描述，如 Mass Storage Class。接口描述符结构定义如下。关于各字段的说明参见 USB2.0 规范第 9 章。

```
typedef struct usb_interface_descr
    {
    UINT8 length;           /* bLength */
    UINT8 descriptorType;       /* bDescriptorType */
    UINT8 interfaceNumber;      /* bInterfaceNumber */
    UINT8 alternateSetting;     /* bAlternateSetting */
    UINT8 numEndpoints;         /* bNumEndpoints */
    UINT8 interfaceClass;       /* bInterfaceClass */
    UINT8 interfaceSubClass;        /* bInterfaceSubClass */
    UINT8 interfaceProtocol;        /* bInterfaceProtocol */
    UINT8 interfaceIndex;       /* iInterface */
```

} USB_INTERFACE_DESCR, *pUSB_INTERFACE_DESCR;

#define USB_INTERFACE_DESCR_LEN 9

4. 端点（或称节点）（endpoint）是 USB 中最终的数据传输节点。主机的数据传输将以 USB 外设中某个 endpoint 作为数据源或者目的地。Endpoint descriptor 相应的描述了一个 endpoint 的地址，属性（如支持的最大传输包大小）等。Endponit 隶属于某个 interface，每个 interface 包括 0 个或多个 endpoint，表示在某种子功能下可以有 0 个或多个数据传输的节点。Endpoint 类似于网络编程中使用的套接字，二者本质功能一样，表述方式不同而已。USB 外设为每个 endpoint 分配 FIFO 作为缓冲区，一般而言发送和接收各自具有缓冲区。USB 外设内嵌系统软件（或称为固件 firmware）通过这些 FIFO 与主机进行数据交互。所以本质上讲，endpoint 只是书面上的一种说法，或者说只是便于数据传输的一种机制。每个 endpoint 都具有一个地址用于数据传输中的寻址从而确定本次数据是发送到哪个 FIFO 的。当主机进行读取操作时，其发送一个数据请求 packet 到某个 endpoint，USB 外设（内部的 USB 控制器）接收到请求后，将请求写入该 endpoint 对应的接收 FIFO，并发出一个中断，系统软件中断处理程序读取状态寄存器，获知此次中断源，从而到相应 endpoint 对应的 FIFO 中读取数据，并对数据进行解析，而后进行适当的响应。如请求为 get_configuration 描述符，则系统软件从 USB 外设中非易失介质中读取 configuration，interface，endpoint 描述符信息，写入原先接收请求的 endpoint 对应的发送 FIFO 中，之后配置 USB 控制器相关寄存器，启动 USB 控制器将这些描述符信息发送给主机，从而完成一次主机请求。对于主机写入数据，原理相同。所以从根本上讲，endpoint 只是一种便于描述的书面上的一种用语，可以简单的将其看作是两个 FIFO：接收 FIFO 和发送 FIFO。使用 endpoint 的目的是为数据传输提供一种统一的机制，如寻址机制。USB 控制器并非如想象中的那么“能干”，它仅仅完成将数据写入相关的 FIFO 或者从 FIFO 中将数据通过 USB 总线发送给对端的 USB 控制器。由此我们可以简单用下图 11-3 描述 USB 实现的功能。

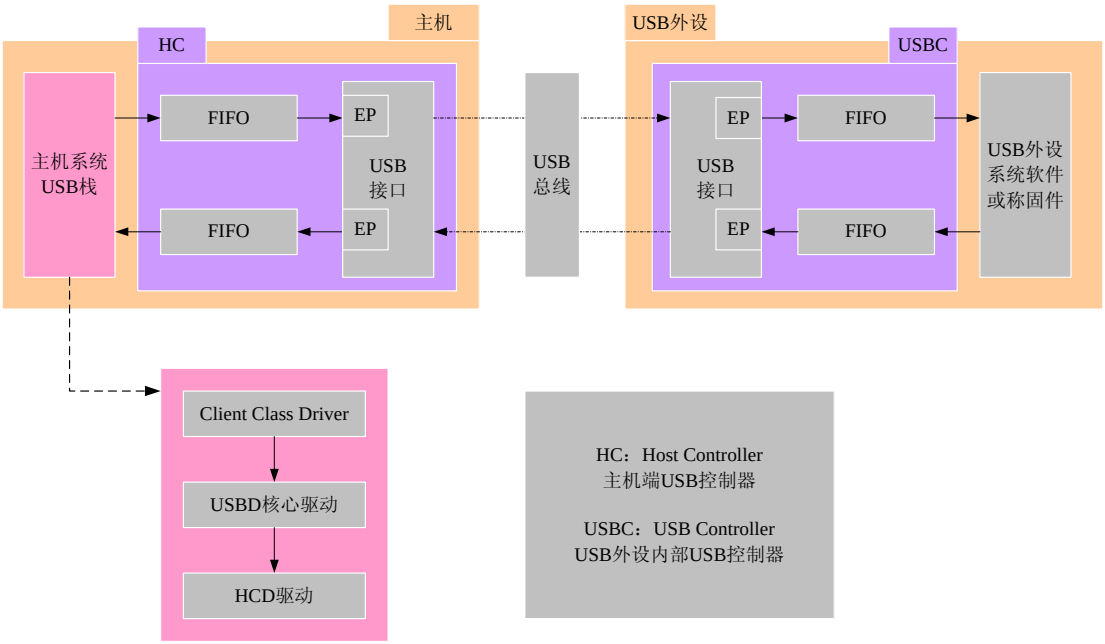


图 11- 3 USB 实现功能简介

如下为 endpoint 描述符定义。各字段的说明参见 USB2.0 规范第 9 章。

```
typedef struct usb_endpoint_descr
    {
    UINT8 length;              /* bLength */
    UINT8 descriptorType;      /* bDescriptorType */
    UINT8 endpointAddress;     /* bEndpointAddress */
    UINT8 attributes;          /* bmAttributes */
    UINT16 maxPacketSize;      /* wMaxPacketSize */
    UINT8 interval;            /* bInterval */
    } WRS_PACK_ALIGN(1) USB_ENDPOINT_DESCR, *pUSB_ENDPOINT_DESCR;

 #define USB_ENDPOINT_DESCR_LEN    7
```

关于 USB 控制器目前主要有两种，即符合 OHCI，UHCI 规范的控制器（EHCI 本质相同）。目前主机 USB 栈最底层的 HC 驱动也是主要针对这两种控制器的，而且绝大部分都是以 PCI 总线作为主机 CPU 和 HC 之间的默认通信总线。这些都不需要我们的更改。而一旦需要我们写相关 USB 方面的驱动时，绝大部分时都是 HC 驱动，而不是上图 3 中 Client Class 驱动，这一点在嵌入式系统下尤其突出。

HC 驱动即驱动 USB 控制器工作的程序，主要涉及对 USB 控制器硬件寄存器的配置。要进行此类驱动编写，一般有如下几点要注意的地方。

1. 嵌入式系统下 HC 控制器一般既不属于 OHCI 范畴，也不属于 UHCI 范畴，而是符合 USB 规范的另一类 HC，这也是构成我们需要重新编写 HC 驱动的基本前提。故首先在了解 OHCI，UHCI 其中之一工作原理的基础上，仔细阅读我们要驱动的 HC 的手册，对比理解要比完全接受一套全新的思想容易的多。
2. 阅读 USB 规范，对基本概念进行掌握，如各类描述符，各类标准请求以及 USB 通信基本的帧序列和格式。弄清这些对于理解哪些是硬件完成的，哪些需要系统软件（即我们的驱动）完成至关重要。
3. 研究主机操作系统 USB 栈实现，尤其是与 HCD 直接交换的 USBD 核心层实现。此外对于 Client Class 驱动层也要做必要的理解。
4. 研究 UHCI 或者 OHCI HCD 驱动的实现，深刻进行理解，从而为实现它类 HC 驱动获取一个基本思路和总体方向。
5. 改代码。注意不是直接从零开始。从一个 UHCI 或者 OHCI HC 驱动直接修改要比从空白开始编写代码要快得多。这可以首先让我们把握整体架构，另外对于保持与系统 USB 栈原有其他部分的兼容性也要方便的多。
6. 代码修改过程中先以与其他部分的衔接为主要方面，具体的 HC 硬件操作可以放在最后，这部分比较简单，复杂的是如何与系统原有 USB 栈进行融合，尤其是与 USBD 核心驱动层的衔接工作，这要求第 3 步中的工作要深入。

11.2 USB 硬件接口

诚如上文所述，USB 接口仅仅是数据传输的一种介质，其采用一对差分线完成数据的最终

传输。如下图 11-4 所示为一嵌入式平台上 USB 控制器的内部结构框图。

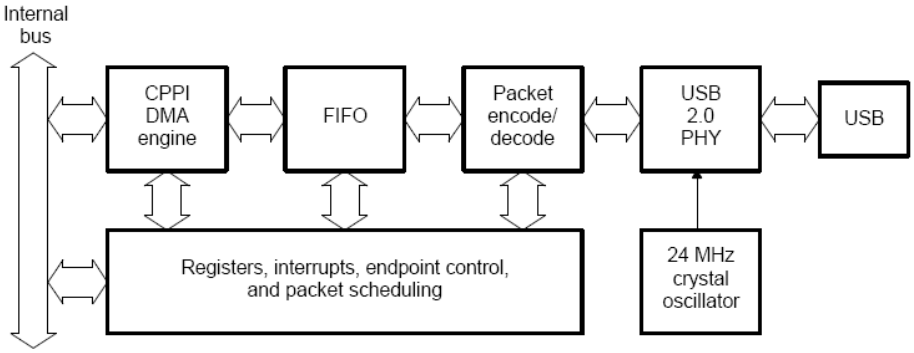


图 11- 4 USB 控制器结构框图

图 11-4 中：

（1） USB 指 USB 总线，USB 传输通常采用 4 线结构：D+，D-，电源线，地线。注意 D+，D-两根数据线通常采用双绞方式，以提高抗噪能力。如下图 11-5 所示，为常用的 USB A 类和 B 类接口类型及其信号线分布。

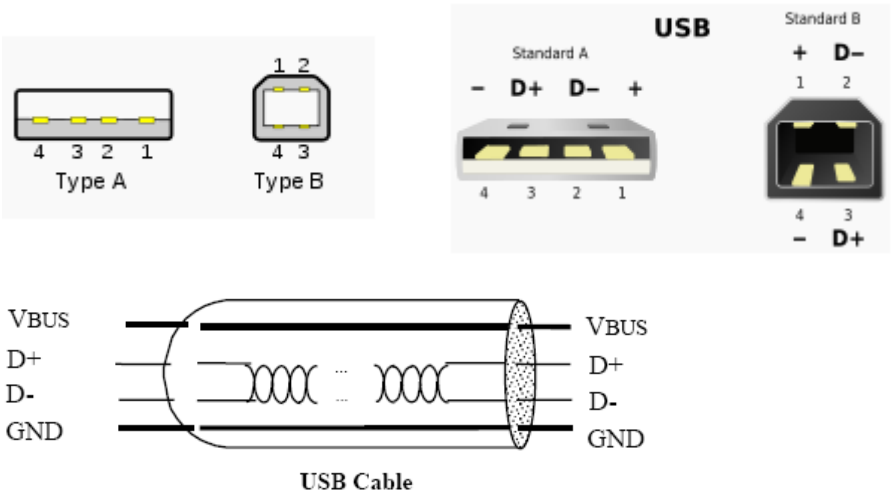


图 11- 5 USB A 类和 B 类接口

（2） PHY 物理层完成差分信号到数字信号的转换。

（3） 编解码模块完成数据的编解码工作。

（4） FIFO 为 USB 控制器中某个 endpoint 对应的数据缓冲区。USB 控制器中每个 endpoint 都对应两个缓冲区：数据发送缓冲区和数据结构缓冲区。FIFO 也是 HC 驱动与 HC 之间数据交互的中间媒介。

USB 支持如下数据传输速率：

（1）1.5Mbps 低速（Low Speed）传输，最早在 USB1.0 规范中提出。

（2）12Mbps 全速（Full Speed）传输，最早在 USB1.1 规范中提出。

（3）480Mbps 高速（High Speed）传输，在 USB2.0 规范中提出。

（4）4.8Gbps 超速（Super Speed）传输，在 USB3.0（2008 年）规范中提出。

目前市面上使用广泛是 USB2.0 规范（支持 1.5Mbsp，12Mbps，480Mbps），USB3.0 规范支持的 USB 设备（2010 年 1 月第一个经过验证的 USB3.0 设备被宣布）也已开发出来，但是到广泛使用还有一个过程。在下文的介绍中，我们将以 USB2.0 规范作为对象进行讲解。

11.3 USB 设备驱动

USB 总线是具有固定主从关系的工作总线，即一端始终是主控制端，总线上所有操作的发起都是由主控制端完成的，从设备一端只能处于被动响应的地位，不可主动发起任何操作（当然唤醒主控制器除外）。通常 PC 机都是作为主设备端工作的，其内部集成一个根 HUB，所有外壳机箱上配置的 USB 接口都是这个根 HUB 分级延伸出来的，换句话说，所有的 USB 接口都竞争使用一根总线进行数据的传输和接收。对于嵌入式系统而言，根据系统的使用场合不同，作为 USB 主从端的可能性都存在。Vxworks 同时提供了主从端的 USB 核心栈的实现，但是作为主设备端的 USB 主机控制器 HC 的驱动以及作为从设备端的 USB 目标机控制器（Target Controller：TC）的驱动支持的都不是很完备，尤其是在嵌入式平台下，HC 通常不是挂接在 PCI 总线上的基于 OHCI 或者 UHCI 规范的设备，而是仅仅实现了 USB 接口的特殊接口，其数据交互方式完全不同于标准的 OHCI 或者 UHCI 设备。这也构成了 Vxworks 下设计和实现 USB 设备驱动的基本前提。

USB 设备驱动具体的来讲应该是 USB 主机控制器驱动或者 USB 目标机控制器驱动，其完成 USB 栈与底层控制器硬件之间的数据传输，数据内容的解释将由位于 USB 栈之上的 USB 类驱动（Class Driver）完成。USB 类驱动是使用 USB 接口作为通信通道的特定应用，如 USB 键盘，USB 打印机，USB 块存储介质等。USB 类驱动通常无需开发者设计，操作系统实现本身一般都包含有比较完备的 USB 类驱动代码，因为其位于 USB 栈之上，做到了底层硬件设备无关，故可以完全由操作系统本身按标准的方式进行实现。如下图 11-6 所示，为 Vxworks 下 USB 内核驱动层次详细示例图。

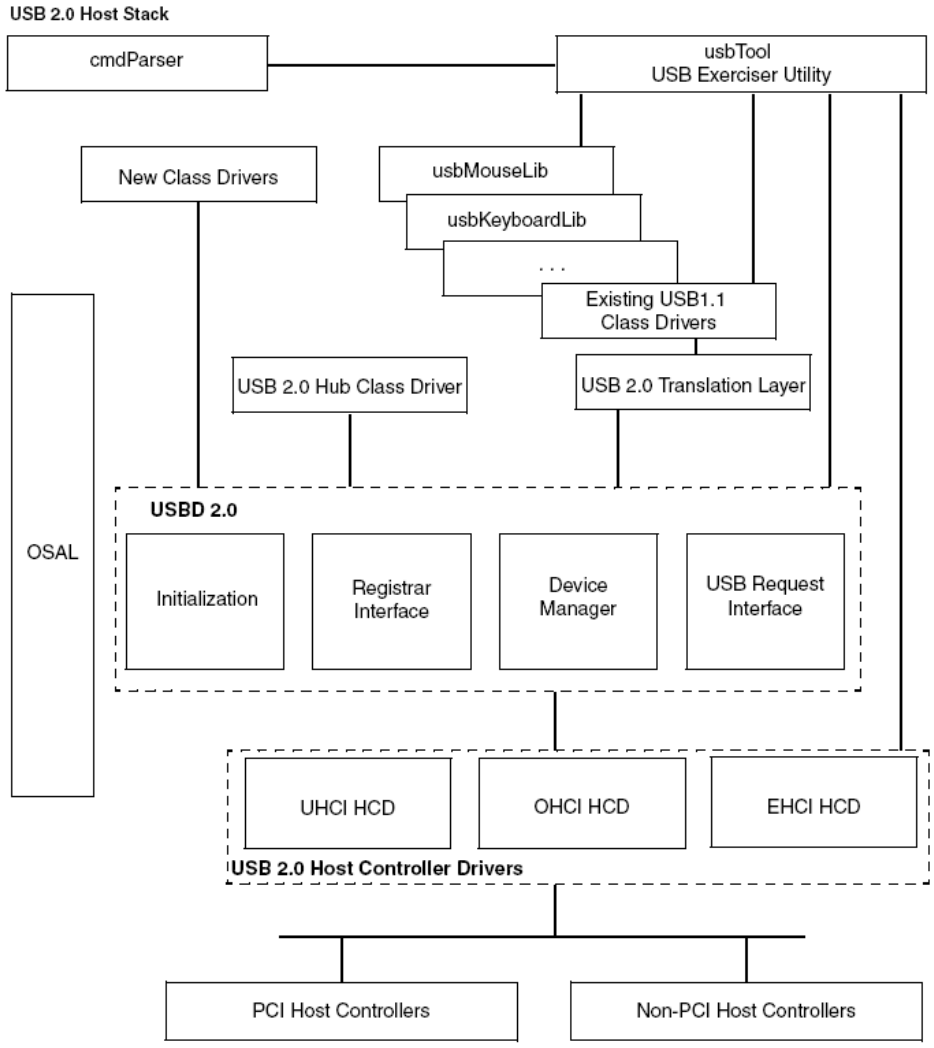


图 11- 6 Vxworks 下 USB 内核驱动层次图

由图 11-6 可见，Vxworks 已经提供对于标准 UHCI，OHCI，EHCI 主机控制器的驱动支持，这些主机控制器一般挂接在 PCI 总线上。但是正如上文所述，Vxworks 较多的使用在嵌入式平台上，而对于这些平台，通常没有工业标准的 PCI 总线支持，而是一些特殊的局部总线。而对于 USB 主机控制器而言，其一般也不是按照 UHCI，OHCI，EHCI 标准设计的，而是使用了自定义的一套数据传输结构，当然在底层上还是使用 USB 接口，而数据操作方式上还是遵循 USB 规范要求。其实从 UHCI，OHCI，EHCI 三者本身即可理解，三者都是使用 USB 接口，符合 USB 规范，换句话说，实际比特位的传输都是使用 USB 接口的 D+,D-信号线，而且数据的解释和响应都遵照 USB 规范中的要求，但是三者却使用不同的方法将 FIFO 中的数据转化为 D+，D-信号线上的信号。嵌入式平台上主机控制器也是同样的道理，其只不过在 OHCI，UHCI 或者 EHCI 之外又实现了一种数据的转换方式，我们可以称之为 PHCI 或者 RHCI 或者其他任何名称，其本质上都没有区别，不同的是 OHCI，UHCI 提出的较早，使用的平台较多，故渐渐成为一种规范，而我们的 PHCI 或者 RHCI 由于使用在特定的嵌入式平台上，使用范围较窄，故没有成为规范而已。

正因如此，Vxworks 只能对 UHCI，OHCI，EHCI 这些使用范围广，有具体规范定义的主机控制器提供驱动支持，而对于嵌入式平台特定的主机控制器则必须由平台操作系统支持人员进行开发。

到此，我们应明白 USB 驱动的开发对象和开发原因。开发对象是平台上支持 USB 主接口的 USB 主机控制器 HC 或者支持 USB 从接口的 USB 目标机控制器 TC。开发原因是在嵌入式平台上的这些 USB 控制器通常都不符合 OHCI，UHCI，EHCI 规范要求，不是使用操作系统提供的控制器驱动代码，必须自行实现这些 USB 控制器的驱动。
开发难度是 USB 控制器驱动本身与内核 USB 栈的耦合性很紧密，必须对 USB 栈的实现有比较深入的了解才能成功完成驱动的开发工作。

注意：基于作者开发环境的限制，本章使用的 USB 代码都是基于 Vxworks 5.5 版本的，不同版本下 USB 驱动开发方式会有所不同，读者在阅读本章时旨在掌握实现的思想和关键概念，在新版本下进行 USB 驱动开发时进行类比即可。

11.4 使用示例

为了实现 USB 主机功能的统一，提高系统的可靠性与可移植性，上游芯片生产厂家在确定 USB 标准的同时，也确定了相应的主机规范。现在用得比较广泛的有三种，其中的用于 USB2.0 高速设备的 EHCI（Enhanced Host Control Interface 增强主机控制接口）规范是 INTEL 用于 USB2.0 高速主机的。而同是 INTEL 推出的 UHCI（Universal Host Control Interface 通用主机）与前 Compaq 、Microsoft 等推出的 OHCI（Open Host Control Interface 开放主机控制接口）可用于全速与低速 USB 系统中，硬件的要求与系统性能、软件复杂的要求相对较低，也能够满足大部分的具有 USB 接口嵌入式系统的要求。而在 UHCI 与 OHCI 的对比中，UHCI 对硬件的要求相对较少，但对系统的处理能力与软件的开发要求相对要高（PC 机就较多地采用了 UHCI）；OHCI 则把较多的功能定义在硬件中，软件需要处理的内容就相对容易，对系统的处理能力和系统资源的要求就低。因此，在嵌入式的 USB HOST 功能中，较多地选用了遵循 OHCI 的规范的硬件，从而简化了系统的设计。

在上一节中我们提到USB驱动开发的原因在于嵌入式平台上USB控制器通常不符合OHCI，UHCI 和 EHCI 规范要求。但是作为驱动示例，我们将采用 UHCI 控制器内核驱动代码作为本章下文中讲解的基础，一方面由于 UHCI 规范相对比较简单，我们不需要花大量时间在规范以及底层硬件的分析上，另一方面 Vxworks 本身提供这些驱动的源代码，读者可在阅读本章的同时查阅相关源代码进行深入理解。关键在于驱动设计思想的分析，对此任何一种 USB 控制区驱动代码都可以达到这个要求，而 UHCI 由于可用资料较多，也较容易获得，也免去笔者花费大量笔墨在控制器硬件本身的描述上。不过接下来，我们还是简单介绍一个符合 UHCI 规范的 USB 主机控制器的基本工作原理。

UHCI 规范将 UHCI 分为两个部分：HCD（Host Controller Driver：主机控制器驱动程序）和 HC（主机控制器）。其中 HCD 负责按照上层数据传输要求在内存中组织 HC 传输所需的数据结构；HC 则根据内存中数据结构的指示在硬件层次上完成数据的具体收发工作。

USB 支持四种数据传输方式：

（1） 同步传输（Isochronous）。这种传输方式在 UHCI 中被优先处理，在每个传输周期（1ms）中 UHCI 控制器件将首先处理同步传输，其次中断传输，其次控制传输，其次块传输。同步传输要求固定持续的传输速率，但对数据传输可靠性不作要求（除了同步传输外，其他三类传输都要求数据可靠性）。UHCI 控制器以及驱动软件都不

保证同步传输的数据一定成功。如 USB 音频编解码器。

（2） 中断传输（Interrupt）。这种传输方式用于小量但要求及时性的数据传输。如 USB 键盘，鼠标等设备都采用这类传输方式。

（3） 控制传输（Control）。这种传输主要用于控制，状态，配置等信息的传输。通常用于 USB 设备自身信息的传输上。

（4） 块传输（Bulk）。这种传输用于大量，可靠的数据传输。如 U 盘等存储设备都采用此类传输方式。

数据传输可靠性从 HC 以及 HCD 两个方面进行保证，首先 HC 会根据数据传输结构中要求的重传次数在传输失败时连续进行尝试，如果在规定次数后，仍然无法成功完成数据的传输，HC 将通知 HCD 驱动程序，由 HCD 完成相关重新配置后，重新安排这些数据传输。

USB 数据传输的基本单元称为帧，每个帧固定占用 1ms（对于低速和全速而言，高速下位 125us）的时间间隔，在每个 1ms 时间间隔内处理一个帧队列。对于 UHCI HC 而言，内存中共维护了 1024 个帧队列，HC 将循环对这 1024 个队列进行处理。在每个 1ms 帧中，同步数据优先处理，其次中断，其次控制，最后是块。USB 规范要求必须为控制传输预留 10% 的传输时间，不保证块传输的时间。如下图 11-7 所示为 1ms 帧内各传输类型的分配方式。

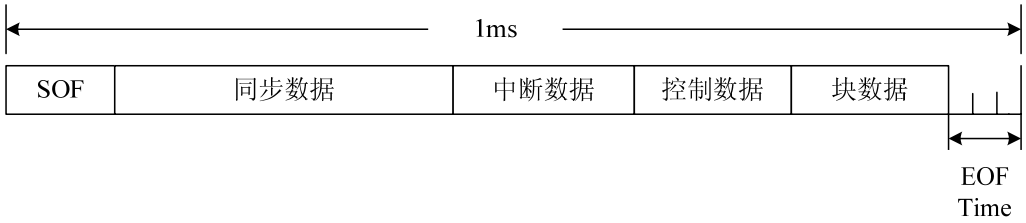


图 11- 7 1ms 帧内传输类型分配方式

每个 1ms 帧由一个称为帧起始（SOF：Start Of Frame）的报文开始。该报文中包含了当前帧的序号，这个帧序号将在每完成一个 1ms 帧后递增 1。在 UHCI 控制器内部，有一个 SOF 计数寄存器完成 1ms 帧的建立，当 SOF 计数寄存器溢出时，就产生一个 1ms 帧。可以 SOF 更改寄存器进行修改，以微调 1ms 帧的时刻点。UHCI 控制器内部还维护一个帧计数寄存器，用以对帧进行计数，帧计数寄存器的最低 11 比特被用帧序号，而最低 10 位同时也被用作索引对内存中 1024 个帧队列进行寻址，如此每个 1ms 帧间隔就处理一个帧队列，完成 USB 数据的传输。基本原理图如下图 11-8 所示。

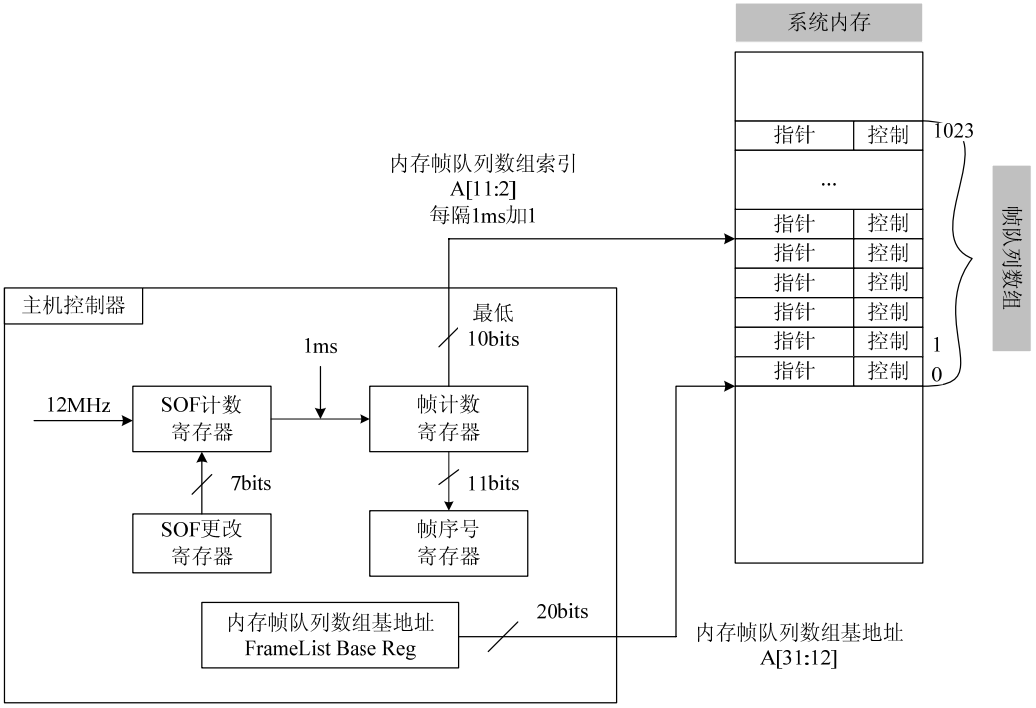


图 11- 8 UHCI 控制器内部寄存器以及内存映射关系

对于 UHCI HC，HCD 在内存中维护 1024 个帧队列，构成一个帧队列数组。数组中每个元素在 1ms 时间间隔内被处理。被处理元素的确定将由内存帧队列数组基地址和帧序号寄存器的最低 10 比特决定。数组中元素本质上都是指针，指向需要进行处理“传输描述符”。内存帧队列数组以及其指向的“传输描述符”阵列如下图 11-9 所示。

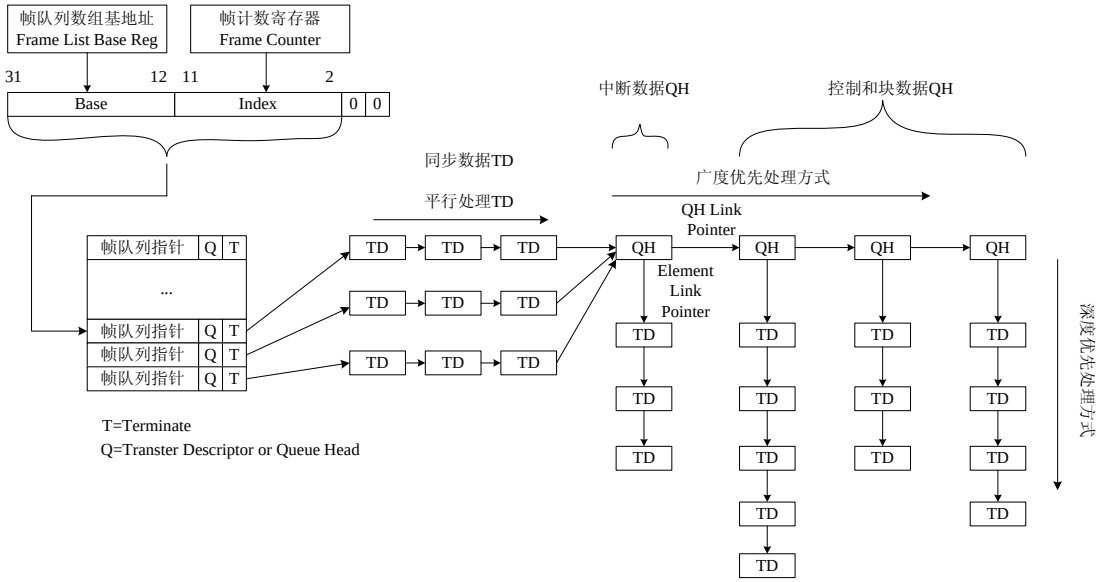


图 11- 9 内存中帧队列数组及“传输描述符”阵列示意图

数组中每个元素的具体格式如下图 11-10 所示。

31	4	3	2	1	0
帧队列指针（Frame List Pointer）	0	0	Q	T	

图 11- 10 帧队列数组中每个元素的格式

其中帧队列指针指向一个“传输描述符”队列，这个队列将被 HC 处理，发起 USB 相关操作。Q 标志位用于标识帧队列指针是指向一个 TD 还是一个 QH。当在一个 1ms 间隔内，有同步数据需要传输时，则 Q 标志位为 0，表示帧队列指针指向一个同步数据 TD 描述符，而当 Q 标志位为 1，则表示帧队列指针指向一个 QH 描述符，该 1ms 帧内没有同步数据需要传输。对于中断数据，控制数据，以及块数据传输方式，将组织成一个独立的队列的方式，这个队列头由一个 Queue Head 结构表示，该结构下将管理一批 Transfer Descriptor（TD）传输描述符。T 标志位表示帧队列指针的有效性，当 T=1，则表示当前 1ms 间隔内，没有数据需要处理，这是一个空帧。

QueueHead 结构共 8 个字节，具体定义如下图 11-11 所示。

31	4	3	2	1	0	
Queue Head Link Pointer	0	0	Q	T		00-03h
Queue Element Link Pointer	0	R	Q	T		04-07h

图 11- 11 Queue Head 结构定义

Queue Head（QH）结构中“Queue Head Link Pointer”用以链接其他 QH 结构（参见图 9）；而“Queue Element Link Pointer”则用于连接此 QH 管理下的 TD 描述符；但是这两个字段在具体应用中到底是指向一个 QH 还是 TD，最终的确定还是由 Q 标志位决定。Q 表示对应的指针字段是指向一个 QH 还是 TD，当 Q=1，则指向一个 QH 结构，否则指向一个 TD 结构。T 标志位表示指针的有效性，当 T=1 时，表示对应的指针无效。

Transfer Descriptor（TD）结构共 32 个字节，其中前 16 个字节具有确定的含义，后 16 个字节由 HCD 驱动程序自定义。该结构定义如下图 11-12 所示。

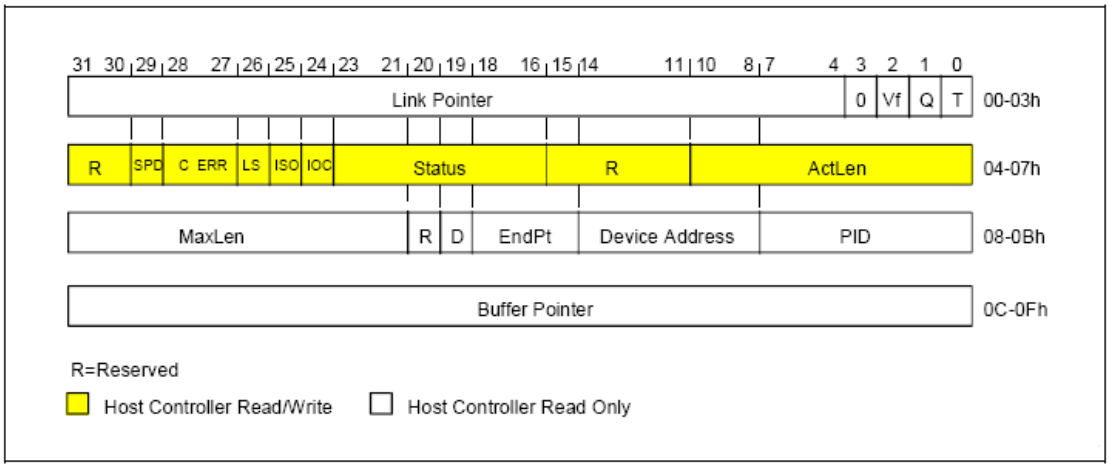


图 11- 12 Transfer Descriptor 结构定义（前 16 字节）

其中，结构中“Link Pointer”用于 TD 间的链接，“Buffer Pointer”指向该 TD 描述符对应的数据缓冲区，其他字段的含义请读者参见 UHCI 规范。

对于 UHCI 控制器驱动，其将根据上层请求在内存中分配 TD 和 QH 结构，并将其挂接到帧队列数组中某个元素指向的帧队列中，控制器硬件本身将自动对这些 TD，QH 结构进行处理，发起 USB 总线操作，完成 TD 中指示的动作，如设备或接口描述符数据获取等。

对于 UHCI 控制器而言，其主要的操作就是在内存中安排相关数据结构，控制器本身的配置只需在初始化过程完成一次即可，其后 USB 操作的发起将完全由 UHCI 控制器本身完成，其在每个 1ms 时间内处理一个帧队列，在其工作期间，往复循环的遍历帧队列数组，对其中可能具有的 TD 描述符进行解析，发起 USB 总线操作，这个解析是硬件自动完成的，无需驱动从中“插手”，UHCI 控制器完成对一个 TD 的处理后，将更新该 TD 中相关字段，发出一个中断，通知 UHCI 控制器驱动进行后处理，如将读取的数据传递给上层使用。驱动以及控制器本身都不对数据内容进行解释，数据内容的解释将由位于 USB 内核栈之上 USB 类驱动（Class Driver）完成，操作系统对类驱动支持比较完备，无需用户进行开发工作。

对于我们的 UHCI 控制器驱动而言，现在的任务已经比较明确了，我们从 USB 内核栈接收用户请求，在内存中安排 QH，TD 等结构，并将其挂接到帧队列数组中，而后等待 UHCI 控制器完成对这些 TD 的处理后发出一个中断，我们将在中断处理程序中完成一些后处理工作，最终完成上层的服务请求。

作为层次性驱动设计的惯例，在设备最终工作前，各层次之间必须完成衔接，即互相注册各自所需的信息。对于 USB 这种遵循严格主从工作模式的设备而言，底层驱动完全处理被动调用的地位，无论是数据的发送还是数据的接收，都是如此，所以在注册关系上比较简单，底层 USB 控制器驱动只需要向 USB 核心层注册接口函数，事实上，只需要注册一个接口函数即可，对于所有的上层请求都将以这个函数作为入口函数进行处理。故上文中我们一再说到 USB 控制器驱动的复杂性，在于与 USB 核心实现层的紧密耦合性，这种紧密耦合并非是说二者之间的函数调用关系复杂，而是二者之间函数调用关系过于简单，使得必须要充分理解 USB 核心层实现之后，才能对底层实现的功能进行掌握。
从 Vxworks USB 栈实现来看，底层 USB 控制器驱动只需要向 USB 栈注册一个总入口函数即可，所有的操作请求，USB 栈都将调用这个函数进行处理，这有些类似于 ioctl 函数的实现，实际上这个总入口函数就是一堆 case 语句对应不同功能的实现，这就要求 USB 控制器驱动开发人员必须充分理解每种请求对应实现的功能以及 USB 栈对请求的封装。USB 控制器驱动实现复杂性的另一个方面是各种数据结构关系错综复杂，基本上在请求传递过程中每调用一个函数就会换一个结构对请求进行封装，传递给底层 USB 控制器驱动时，驱动开发人员必须很清楚其完成请求所需的各种信息如何通过结构间的关系查询到，这就要求驱动开发人员必须清楚请求传递过程中的处理以及其中每种结构的具体含义及其之间的关系，归根结底，还是要充分理解内核 USB 栈的实现。
对内核 USB 栈实现的分析，最好的方式是跟随一个上层请求的传递过程，弄清楚这个请求在 USB 栈中的传递过程，直到到达底层 USB 控制器驱动函数为止，即可基本弄清 USB 栈的实现层次。

下一节中，我们将以一个实现存储功能的类驱动作为最上层应用为例，分析一个上层请求是如何经过内核 USB 栈最终传递到底层驱动的，我们将分析其中涉及到的关键数据结构以及请求传递路径上调用的关键函数。

11.5 USB 操作请求传递过程

本节我们将以一个实现存储功能（mass storage）的类驱动（client class driver）上层应用为例，分析最上层类驱动中发出的一个数据读取请求是如何经过内核 USB 栈最终传递到底层 USB 主机控制器驱动的（实际上在 mass storage client class driver 之上还有文件系统层，IO 子系统层，对于应用层类驱动的详细情况我们将在下一节“应用层类驱动初始化”中介绍）。Mass Storage 类驱动对应 usbBulkDevLib 库（使用 CBI 接口对应的库为 usbCbiUfiDevLib），该库实现了基于 USB 接口的应用层应用，用以进行大量数据的存储，操作类似 U 盘之类的具有 USB 接口的设备。

实际上在 Mass Storage 类驱动之上是文件系统层，读者可以想见，当我们在 PC 下插入一个 U 盘设备时，实际上是以文件和目录的方式对 U 盘中存储的数据进行操作，即在 Mass Storage 类驱动之上还进行了文件系统层的封装，在 Vxworks 下文件系统层由 IO 子系统进行管理，那么我们可以给出如下图 11-13 所示的实现存储功能的 USB 设备的主机端内核驱动层次。

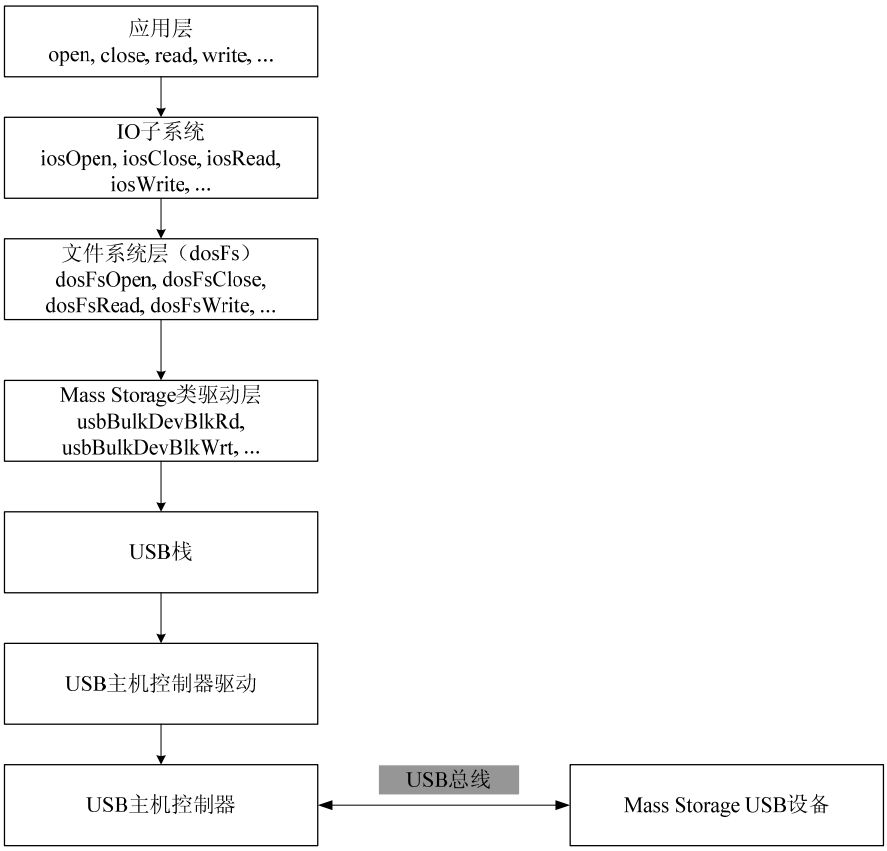


图 11- 13 Mass Storage 设备主机端内核驱动层次

由图 11-13，可以看到 Mass Storage 类驱动对应的 USB 设备读写函数是 usbBulkDevBlkRd 和 usbBulkDevBlkWrt，Mass Storage 类驱动遵循块设备驱动接口，其将向文件系统层通过 BLK_DEV 结构注册相关函数（包括上面两个读写函数），对于 Mass Storage USB 设备的操作最终是通过文件和目录的方式进行的，这对于用户而言，提供了极大的方便性。

以数据读操作为例，经过 IO 子系统和文件系统层的传递，这个请求将到达 Mass Storage 类驱动中的 usbBulkDevBlkRd 函数。下面我们就以该函数为起点，介绍这个数据读取操作是

如何经过 USB 栈最终传递到 USB 主机控制器驱动的，我们将给出在 USB 栈中传递期间调用的关键函数以及数据结构，使得读者对 USB 栈的基本实现框架有一个了解，对于安装了 Tornado 开发环境的读者，建议在一边阅读下面内容的同时，一边查看源代码，我们会在每个函数和结构之前的注释中给出该函数和结构的具体位置，以方便读者查找。

11.5.1 第一层：usbBulkDevBlkRd

用户读数据请求在 Mass Storage 类驱动层响应函数 usbBulkDevBlkRd 函数定义在 usbBulkDevLib.c 文件中，该函数实现如下。

```
/*******************************************************************************
* target/src/drv/usb/usbBulkDevLib.c

* usbBulkDevBlkRd - routine to read one or more blocks from the device.
*
* This routine reads the specified physical sector(s) from a specified
* physical device.    Typically called by file system when data is to be
* read from a particular device.
*
* RETURNS: OK on success, or ERROR if failed to read from device
*/
LOCAL STATUS usbBulkDevBlkRd
    (
    BLK_DEV * pBlkDev,              /* pointer to bulk device     */
    int       blkNum,               /* logical block number       */
    int       numBlks,              /* number of blocks to read */
    char *    pBuf                  /* store for data             */
    )
{
    pUSB_BULK_DEV    pBulkDev = (USB_BULK_DEV *)pBlkDev;
    UINT readType;

    /*    Ensure that the device has not been removed during a transfer */
    if ( pBulkDev->connected == FALSE )
    return ERROR;

    USB_BULK_DEBUG ("usbBulkDevBlkRd: Number of blocks = %d, Starting blk = %d\n",
                    numBlks, blkNum, 0, 0, 0, 0);

    OSS_MUTEX_TAKE (bulkDevMutex, OSS_BLOCK);

    /* intialise the pointer to store bulk in data */
    pBulkDev->bulkInData = (UINT8 *)pBuf ;
```

```
    if (pBulkDev->read10Able)
        readType = USB_SCSI_READ10;
    else
        readType = USB_SCSI_READ6;

    if ( usbBulkFormScsiCmd (pBulkDev,
                             readType,
                             blkNum,
                             numBlks)
                           != OK )
    {
        OSS_MUTEX_RELEASE (bulkDevMutex);
        return (ERROR);
    }

    if ( usbBulkCmdExecute (pBulkDev) != USB_COMMAND_SUCCESS )
    {
        OSS_MUTEX_RELEASE (bulkDevMutex);
        return (ERROR);
    }

    OSS_MUTEX_RELEASE (bulkDevMutex);
    return (OK);

}
```

usbBulkDevBlkRd 作为文件系统层直接管理下的驱动层实现函数，遵循块设备驱动的接口实现方式，从 usbBulkDevBlkRd 函数的原型来看，无论底层实现如何复杂，单从类驱动层而言，USB 设备被简单的看做一个普通块设备进行操作。我们传入块设备对应的 BLK_DEV 结构（实际上是一个 USB_BULK_DEV 结构），数据读取的起始块号，要读取的数据块数以及存放读取数据的缓冲区。对于 USB 设备而言，类驱动层自定义了一个 USB_BULK_DEV 结构用以保存自身关键参数，当然作为块设备驱动的设计规则，BLK_DEV 结构将作为 USB_BULK_DEV 结构的第一个成员。
USB_BULK_DEV 结构定义如下。

```
/* target/src/drv/usb/usbBulkDevLib.c */
/* USB_BULK_DEV Structure - used to describe USB MSC/SCSI/BULK-ONLY device */
typedef struct usbBulkDev
{
    BLK_DEV             blkDev;          /* Vxworks block device structure */
                                         /* Must be the first one          */
    USBD_NODE_ID        bulkDevId;       /* USBD node ID of the device     */
    UINT16              configuration;   /* Configuration value            */
    UINT16              interface;       /* Interface number               */
```

```
    UINT16                  altSetting;         /* Alternate setting of interface */
    UINT16                  outEpAddress;       /* Bulk out EP address                */
    UINT16                  inEpAddress;        /* Bulk in EP address                 */
    USBD_PIPE_HANDLE    outPipeHandle;    /* Pipe handle for Bulk out EP        */
    USBD_PIPE_HANDLE    inPipeHandle;     /* Pipe handle for Bulk in EP         */
    USB_IRP                 inIrp;              /* IRP used for bulk-in data          */
    USB_IRP                 outIrp;             /* IRP used for bulk-out data         */
    USB_IRP                 statusIrp;          /* IRP used for reading status        */
    UINT8                   maxLun;             /* Max. number of LUN supported       */
    USB_BULK_CBW        bulkCbw;            /* Structure for Command block        */
    USB_BULK_CSW        bulkCsw;            /* Structure for Command status       */
    UINT8 *                 bulkInData;         /* Pointer for bulk-in data           */
    UINT8 *                 bulkOutData;        /* Pointer for bulk-out data          */
    UINT32                  numBlks;            /* Number of blocks on device         */
    UINT32                  blkOffset;          /* Offset of the starting block       */
    UINT16                  lockCount;          /* Count of times structure locked*/
    BOOL                    connected;          /* TRUE if USB_BULK device connected */
    LINK                    bulkDevLink;        /* Link to other USB_BULK devices     */
    BOOL                    read10Able;         /* Which read/write command the device    */
                            /* supports.    If TRUE, the device uses    */
                            /* READ10/WRITE10, if FALSE uses READ6 / */
                            /* WRITE6 */
} USB_BULK_DEV, *pUSB_BULK_DEV;
```

USB_BULK_DEV 结构除了定义了对上层的接口结构 BLK_DEV 外，还维护底层 USB 设备一些关键配置信息。每个 USB 设备可以支持多个 configuration，每个 configuration 下可以有多个 interface，每个 interface 一般具有多个 endpoint。其中 interface 定义具体功能，endpoint 作为数据传输节点，完成具体数据的传输，configuration 则表示功能的组合关系。在 USB 设备工作之前，我们必须对 USB 设备的 configuration，interface 进行明确的配置以实现 USB 设备具有的特定功能，如 Mass Storage 设备功能。同时为了完成数据的传输，我们必须创建两个数据通信管道（USB 规范将主机和 USB 设备中节点之间的数据传输通道称为管道），一个管道实现数据的上行（IN）传输，一个管道实现数据的下行（OUT）传输。每个管道由 USB 设备地址（USB 设备被检测是主机端分配的地址，用以分辨不同的 USB 设备），节点号，数据传输类型（如块传输），数据传输方向（IN 或者 OUT）等构成。Vxworks 下主机端创建通信管道后会返回一个管道句柄（一个整型数，类似于文件描述符），此后通过该管道的所有操作，将通过该管道句柄进行。

USB 设备请求使用 USB_IRP 结构（下议）进行封装，对于上行和下行两个不同方向的数据请求，USB_BULK_DEV 中定义了 inIrp 和 outIrp 两个字段进行表示。对于状态的请求，也独立定义了一个 statusIrp 来表示，以区分数据的传输。

结构中 bulkInData 用以指向存储被读取数据的缓冲区，实际上从 usbBulkDevBlkRd 函数实现可以看到，这个字段被初始化直接指向调用 usbBulkDevBlkRd 函数的传入的缓冲区，在 USB_BULK_DEV 结构专门定义字段进行缓冲区的表示，而不是直接使用上层缓冲区地址，在于此后的方便寻址，在之后的函数调用中，只需传递一个结构即可，不用每次都要传递一个缓冲区地址。bulkOutData 字段的使用基本相同，不过此时指向了一个要写入数据对应的

缓冲区。
USB_BULK_CBW 结构类型成员 bulkCbw 专门用于 Mass Storage 设备，用以表示设备读写的地址信息如起始块，数据块数等等。USB_BULK_CSW 结构成员 bulkCsw 功能类似，也是专用于 Mass Storage 设备用以存储返回的操作状态信息。对于这两个结构的确切含义请读者参考“Universal Serial Bus Mass Storage Class”规范。这两个结构均定义在 target/h/drv/usb/usbBulkDevLib.h 内核头文件中。对于 USB_BULK_CBW 的初始化，usbBulkDevBlkRd 将调用 usbBulkFormScsiCmd 函数完成，读者可以具体查看 usbBulkFormScsiCmd 函数实现，了解初始化的细节，此处不作讨论。
USB_BULK_DEV 结构的最后一个成员 read10Able 表示采用何种 SCSI 命令集对设备进行操作，SCSI 为 Small Computer System Interface 的简称，直译为“小型计算机系统接口”，是用于计算机与外设互连和数据传输的一套规则，其定义了一套命令集合用于主机和外设之间的通信。对于 SCSI 命令集的详细情况，读者可参考如下地址给出的网站：http://en.wikipedia.org/wiki/SCSI_command。

usbBulkDevBlkRd 函数实现完成如下任务：（1）初始化 USB_BULK_DEV 结构中涉及到数据读取的各个字段，如存储被读取的数据的缓冲区，读取数据的命令集合（调用 usbBulkFormScsiCmd 函数完成）；（2）调用 usbBulkCmdExecute 传递数据读取请求，注意，此时使用的数据结构从 BLK_DEV 转成了 USB_BULK_DEV。

11.5.2 第二层：usbBulkCmdExecute

主机与 Mass Storage USB 设备之间的数据传输分为三个过程：（1）命令阶段，主机发送命令数据，通知要读写的地址信息，长度信息；（2）数据阶段，对于读操作而言，主机将发送 USB 读数据请求，Mass Storage USB 设备中固件将根据第一阶段的参数将主机所需的数据传送给主机；（3）状态阶段，确认读写操作是否成功完成。
对于第一个阶段命令数据的发送，usbBulkCmdExecute 将初始化一个 USB_IRP 结构，将命令封装在一个 USB_BULK_CBW 结构中，将这个 USB_BULK_CBW 结构通过 USB 接口发送给 Mass Storage USB 设备，即此时 USB_IRP 结构指定的数据缓冲区指向一个初始化的 USB_BULK_CBW 结构，由 Mass Storage USB 设备中固件对 USB_BULK_CBW 结构中命令进行解析，并作出适当响应，如对于我们的读操作请求，设备内固件需要从设备存储介质中（一般是 NandFlash）读取该阶段接收命令数据中指定的块中数据，等待第二阶段将数据通过 USB 接口发送给主机。
第二阶段才是真正的数据读取阶段，这些数据将从 Mass Storage USB 设备传送到主机，Mass Storage USB 设备实际上在第一阶段接收到命令数据时，就在进行其内部的数据读取工作，但是只当 USB 接口有数据请求命令时，才通过 USB 接口将这些数据传递给主机。此时 usbBulkCmdExecute 将初始化一个 USB_IRP 结构，专门用于数据的读取，此时 USB_IRP 结构中指定的缓冲区就是 usbBulkDevBlkRd 中接受的用户缓冲区。读取的数据将被填充到该缓冲区中，传递给上层（文件系统层）使用。
第三阶段是读写操作成功与否状态的读取，对于底层 USB 控制器而言，第三个阶段与第二阶段完全一样，只是对于类驱动层而言，是读取不同的数据，第二阶段读取的是用户要求的数据，而本阶段将读取状态数据，这些数据将被类驱动层使用确认第二阶段操作是否顺利完成。此时 usbBulkCmdExecute 函数也是初始化一个 USB_IRP 结构，用于状态数据的读取，USB_IRP 结构中指定的缓冲区指向一个空白 USB_BULK_CSW 结构，读取的数据将被填入

到该结构中，由 Mass Storage 类驱动层检查 USB_BULK_CSW 结构中各字段值，确认第二阶段的工作是否成功完成。

由此我们可以看到，为了完成对 Mass Storage USB 设备的一次读写操作，需要三次块数据传输过程。这是 Mass Storage USB 设备规范上要求的，包括对 USB_BULK_CBW 以及 USB_BULK_CSW 结构的使用也是规范规定的。换句话说，以上所述的三个阶段是 USB 栈之上的高层协议，对于底层 USB 控制器和 USB 接口而言，每个阶段都对应一次块数据的传输过程，从 USB 协议的角度而言，每次块数据传输又要分为几个阶段完成，从 USB 类驱动角度而言，我们并不关心每次块数据传输在 USB 底层上分为几个阶段完成，USB 接口只是一种传递数据或者命令的方式，其并不对数据内容进行解析，数据内容的解释由主机端 USB 类驱动和设备端系统软件（或称之为固件）完成。以常见的 U 盘设备为例，U 盘设备内容其实也是一个小系统，其具有运行代码的微处理器，对主机端发出的数据读写请求进行解析和响应，USB 接口只是数据传输的一种手段，其并不能完成对数据和命令的解析和响应，这些工作包括 NandFlash 中数据的读取和写入都有 U 盘设备内固件完成。基于存储介质的特点，每次读写操作必须知道地址信息，数据长度信息等等，而这就通过专门的命令数据进行传输，这就是上面第一阶段完成的工作。

注意：每种 USB 类驱动实现都不同，USB 类驱动简单的说就是建立在 USB 接口之上的高层协议，其利用 USB 接口完成实际数据的传输，至于对数据如何使用则是由 USB 类驱动决定，所以 USB 类驱动不同，底层实现的功能也就不同，USB 规范本身并不能定义一个 USB 设备的功能，它只是定义了两个设备之间如何传递数据。对于 USB 规范中定义的对各种描述符的需求，实际上这些描述符也是由位于 USB 协议之上的类驱动进行定义，这些描述符可以是存储在 ROM 中的一些信息，对于功能简单的 USB 设备，类驱动软件完成可以临时创建这些描述符，对描述符请求进行响应。诚如本章开头部分的讨论，读者完全可以将 USB 接口看做是两个 FIFO，USB 规范定义了一系列规则如何在 FIFO 之间交换数据，但是如何对数据进行解析则不是 USB 规范的内容，而是设备自身功能决定的。如对于我们此处介绍的实现数据存储功能的 Mass Storage USB 设备，为了完成一次数据的读写，实际上要使用三次底层 USB 块设备操作，分别传输命令，数据，状态。这些命令，数据，状态是被位于 USB 之上的系统软件使用的，而不是指 USB 内部实现数据传输所用的命令，数据，状态，这一点必须要分清。

如下图 11-14 所示为 Mass Storage USB 设备操作流程。

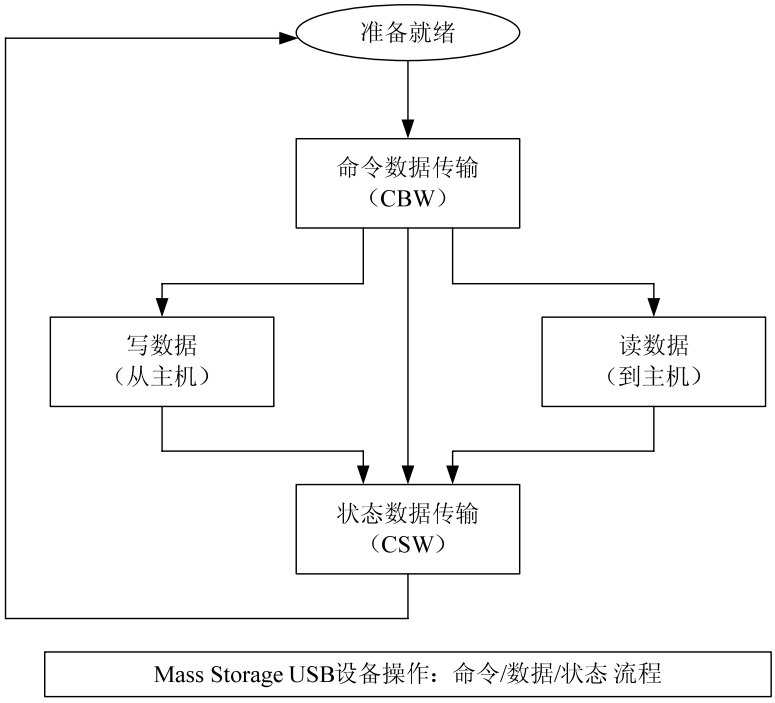


图 11- 14 Mass Storage USB 设备操作流程图

usbBulkCmdExecute 函数也是定义在 usbBulkDevLib.c 文件中，由于实现代码较长，我们只给出用户数据请求的代码，对于命令和状态数据的请求，读者自行查看该函数的实现代码。

```
/*******************************************************************************
* target/src/drv/usb/usbBulkDevLib.c
* usbBulkCmdExecute - Executes a previously formed command block.
*
* This routine transports the CBW across the BULK_OUT endpoint followed by
* a data transfer phase(if required) to/from the device depending on the
* direction bit.    After data transfer, CSW is received from the device.
*
* All transactions to the device are done by forming IRP's initilized with
* command dependent values.
*
* RETURNS: OK on success, or ERROR if failed to execute
*/

LOCAL USB_COMMAND_STATUS usbBulkCmdExecute(pUSB_BULK_DEV pBulkDev)
{
    pUSB_BULK_CSW pCsw ;

    ... //命令数据发送

    /*
     * Check whether any data is to be transferred to/from the device.
     * If yes, form an IRP and submit it.
```

```
 */

if (pBulkDev->bulkCbw.dataXferLength > 0)
{
      if (pBulkDev->bulkCbw.direction == USB_CBW_DIR_IN) //读数据
      {

            /* data is expected from the device. read from BULK_IN pipe */
            memset (&(pBulkDev->inIrp), 0, sizeof (USB_IRP));

            /* form an IRP to read from BULK_IN pipe */
            pBulkDev->inIrp.irpLen                = sizeof(USB_IRP);
            pBulkDev->inIrp.userCallback          = usbBulkIrpCallback;
            pBulkDev->inIrp.timeout               = usbBulkIrpTimeOut;
            pBulkDev->inIrp.transferLen           = USB_BULK_SWAP_32(
                                                      pBulkDev->bulkCbw.dataXferLength);
            pBulkDev->inIrp.bfrCount               = 0x01;
            pBulkDev->inIrp.bfrList[0].pid         = USB_PID_IN;
            pBulkDev->inIrp.bfrList[0].pBfr        = pBulkDev->bulkInData;
            pBulkDev->inIrp.bfrList[0].bfrLen     = USB_BULK_SWAP_32(
                                                       pBulkDev->bulkCbw.dataXferLength;
            pBulkDev->inIrp.userPtr                 = pBulkDev;

            /* Submit IRP */
            if (usbdTransfer (usbdHandle, pBulkDev->inPipeHandle,
                  &(pBulkDev->inIrp))!= OK)
            {
                  USB_BULK_ERR ("usbBulkCmdExecute: Unable to submit IRP for "\
                                "BULK_IN data transfer\n", 0, 0, 0, 0, 0, 0);
                  OSS_SEM_GIVE (bulkIrpSem);
                  return (USB_INTERNAL_ERROR);
            }

            /*
             * wait till the data transfer ends on the bulk in pipe, before
             * reading the command status
             */

            if ( OSS_SEM_TAKE (bulkIrpSem, usbBulkIrpTimeOut + USB_BULK_OFFS)
                        == ERROR )
            {
                  USB_BULK_DEBUG ("usbBulkCmdExecute: Irp time out \n",
                                  0, 0, 0, 0, 0, 0);
                  OSS_SEM_GIVE (bulkIrpSem);
```

```
                return (USB_INTERNAL_ERROR);
            }
        }
        else
        {
            ... //写数据
        }
    }

    ... //状态数据读取

    return (USB_COMMAND_SUCCESS);
}
```

从以上代码可以看，usbBulkCmdExecute 函数初始化一个 USB_IRP 结构，最后调用 usbdTransfer 完成用户读数据请求的传递，此时使用的数据结构从 USB_BULK_DEV 变成了 USB_IRP。USB_IRP 用以表示一个 USB 请求，该结构定义如下。

```
/* target/h/usb/usb.h
 * USB_BFR_LIST
 */
typedef struct usb_bfr_list
{
    UINT16 pid;         /* Specifies packet type as USB_PID_xxxx */
    pUINT8 pBfr;        /* Pointer to bfr */
    UINT32 bfrLen;          /* Length of buffer */
    UINT32 actLen;          /* actual length transferred */
} USB_BFR_LIST, *pUSB_BFR_LIST;
```

USB_BFR_LIST 结构被 USB_IRP 结构使用，表示数据缓冲区地址及其大小，actLen 字段表示缓冲区中实际存储的数据大小。结构中第一个成员表示数据类型，可取值如下：

```
/* USB packet identifiers */
#define USB_PID_SETUP       0x2d //SETUP 数据，只用于 USB 控制传输类型。
#define USB_PID_OUT         0xe1 //USB 写数据请求。
#define USB_PID_IN          0x69 //USB 读数据请求。
```

在我们的例子中，pBfr 指向用户缓冲区（注意 USB_BULK_DEV 结构中的 bulkInData 字段在 usbBulkDevBlkRd 函数被初始化指向用户缓冲区），bfrLen 为需要读取的数据长度，通常表示缓冲区的大小，actLen 将由 USB 栈根据实际读取的数据量进行初始化，pid 被赋值为 USB_PID_IN 表示这是一个 USB 读数据请求。

```
/* target/h/usb/usb.h
 * USB_IRP
 *
 * NOTE: There are certain requirements on the layout of buffers described
```

```
 * in the bfrList[].
 *
 * For control transfers, the first bfrList [] entry must be the Setup packet.
 * If there is a data stage, the bfrList [] entry for the data stage should
 * follow.    Finally, a zero-length bfrList [] entry must follow which serves
 * as a place-holder for the status stage.
 *
 * For isochronous, interrupt, and bulk transfers there may be one or more
 * bfrList[] entries.
 *
 * If there is more than one bfrList[] entry for an isochronous, interrupt,
 * or bulk transfers or more than two bfrList [] entries for control
 * transfers, then each bfrList[].bfrLen (except the last) must be an exact
 * multiple of the maxPacketSize.    The HCD and underlying hardware will
 * make no attempt to gather (during OUT) or scatter (during IN) a single
 * USB packet across multiple bfrList[] entries.
 */

typedef struct usb_irp
{
      LINK usbdLink;          /* Link field used internally by USBD */
      pVOID usbdPtr;          /* Ptr field for use by USBD */
      LINK hcdLink;           /* Link field used internally by USB HCD */
      pVOID hcdPtr;           /* Ptr field for use by USB HCD */
      pVOID userPtr;          /* Ptr field for use by client */
      UINT16 irpLen;          /* Total length of IRP structure */
      int result;             /* IRP completion result: S_usbHcdLib_xxxx */
      IRP_CALLBACK usbdCallback;  /* USBD completion callback routine */
      IRP_CALLBACK userCallback;   /* client's completion callback routine */
      UINT16 dataToggle;          /* IRP should start with DATA0/DATA1. */
      UINT16 flags;       /* Defines other IRP processing options */
      UINT32 timeout;         /* IRP timeout in milliseconds */
      UINT16 startFrame;          /* Start frame for isoch transfer */
      UINT16 dataBlockSize; /* Data granularity for isoch transfer */
      UINT32 transferLen;     /* Total length of data to be transferred */
      UINT16 bfrCount;        /* Indicates count of buffers in BfrList */
      USB_BFR_LIST bfrList [1];
} USB_IRP, *pUSB_IRP;
```

读者可以对比 USB_IRP 结构定义与 usbBulkCmdExecute 函数中的代码，查看 USB_IRP 结构的初始化。USB_IRP 定义的一些指针将在请求传递过程中逐步被初始化，而回调函数指针用以在请求完成时进行相关函数的调用，以进行一些后处理操作。此处不再对结构中各字段进行详细说明，读者可根据这些字段的使用确定其含义。

11.5.3 第三层：usbdTransfer

从这一层开始，请求就已经进入内核 USB 栈进行处理，在第二层入口 usbBulkCmdExecute 函数中，用户读数据请求被封装在一个 USB_IRP 结构中，该结构被作为参数传入 usbdTransfer 函数，完成请求从类驱动层向内核 USB 栈的传递。同时注意 usbdTransfer 函数的第一个参数是一个用户句柄，用于表示特定的类驱动；第二个参数为管道描述符，用于指定数据传输所使用的通道，类似于文件描述符，这个管道在类驱动初始化过程中被创建。usbdTransfer 函数实现代码如下。

```
/*******************************************************************************
* target/src/usb/usbdLib.c
* usbdTransfer - Initiates a transfer on a USB pipe
*
* A client uses this function to initiate an transfer on the pipe indicated
* by <pipeHandle>.    The transfer is described by an IRP, or I/O request
* packet, which must be allocated and initialized by the caller prior to
* invoking usbdTransfer().
*
* RETURNS: OK, or ERROR if unable to submit IRP for transfer.
*/
STATUS usbdTransfer
    (
    USBD_CLIENT_HANDLE clientHandle,    /* Client handle */
    USBD_PIPE_HANDLE pipeHandle,    /* Pipe handle */
    pUSB_IRP pIrp                   /* ptr to I/O request packet */
    )
{
    URB_TRANSFER urb;

    /* Initalize URB */
    urbInit (&urb.header, clientHandle, USBD_FNC_TRANSFER, NULL, NULL,
    sizeof (urb));

    urb.pipeHandle = pipeHandle;
    urb.pIrp = pIrp;

    /* Execute URB */
    return urbExecBlock (&urb.header);
}
```

usbdTranser 函数并不完成具体的功能，其继续对请求进行封装，调用 urbExecBlock 函数传递请求。此时使用的封装结构从 USB_IRP 变成了 URB_TRANSFER，URB_TRANSFER 结构定义如下。

```
/* target/h/usb/usbdCoreLib.h
 * URB_HEADER
```

```
 *
 * The URB_HEADER must be the first field in each URB data structure.    It
 * identifies the USBD function to be executed, the size of the URB, etc.
 */
typedef struct urb_header
{
    pVOID link;          /* n/a      USBD private link ptr */
    USBD_CLIENT_HANDLE handle;     /* I/O      Client's handle with USBD */
    UINT16 function;          /* IN       USBD function code */
    int result;               /* OUT    Final URB result: S_usbdLib_xxxx */
    UINT16 urbLength;              /* IN       Length of the total URB */
    URB_CALLBACK callback; /* IN       Completion callback */
    pVOID userPtr;            /* IN       Generic pointer for client use */
} URB_HEADER, *pURB_HEADER;

/* target/h/usb/usbdCoreLib.h
 * URB_TRANSFER
 */
typedef struct urb_transfer
{
    URB_HEADER header;        /*     URB header */
    USBD_PIPE_HANDLE pipeHandle;/* IN    Pipe handle */
    pUSB_IRP pIrp;            /* IN       ptr to I/O request packet */
} URB_TRANSFER, *pURB_TRANSFER;
```

usbdTransfer 首先调用 urbInit 函数（定义在同一个文件中）初始化 URB_TRANSFER 结构中中 header 成员，这是一个 URB_HEADER 结构。注意 urbInit 调用时传递的功能号为 USBD_FNC_TRANSFER，表示这是一个数据传输请求。读者可对照 urbInit 原型和此处的调用参数。

```
VOID urbInit
    (
    pURB_HEADER pUrb,              /* Clients URB */
    USBD_CLIENT_HANDLE clientHandle,    /* Client handle */
    UINT16 function,               /* USBD function code for header */
    URB_CALLBACK callback,         /* Completion callback routine */
    pVOID userPtr,                 /* User-defined pointer */
    UINT16 totalLen                /* Total len of URB to be allocated */
    );
```

usbdTransfer 函数只对 URB_HEADER 结构中成员进行了部分初始化，其他成员（如 callback，usrPtr）的初始化进一步在 urbExecBlock 函数中完成。我们的读数据请求信息 USB_IRP 结构被封装在 URB_TRANSFER 结构的 pIrp 成员中，进行向下传递。

11.5.4 第四层：urbExecBlock

urbExecBlock 函数被 usbdTransfer 函数调用进一步完成对 URB_HEADER 结构中成员的初始化，继续传递我们的读数据请求。urbExecBlock 函数实现如下。

```
/*******************************************************************************
* target/src/usb/usbdLib.c
* urbExecBlock - Block on the execution of the caller's URB
*
* Execute the <pUrb> passed by the caller and block until execution
* completes.
*
* RETURNS: OK, or ERROR if error detected while trying to execute URB
*/

LOCAL STATUS urbExecBlock (pURB_HEADER pUrb )
{
    USB_MESSAGE msg;

    /* Have we been initialized? */
    if (initCount == 0)
        return ossStatus (S_usbdLib_NOT_INITIALIZED);

    /* Get a semaphore from the pool of synchronization semaphores to be
     * used for this URB.
     */
    if (usbQueueGet (semPoolQueue, &msg, OSS_BLOCK) != OK)
        return ossStatus (S_usbdLib_OUT_OF_RESOURCES);

    /* msg.lParam is an available SEM_HANDLE. */
    pUrb->callback = urbCallback;
    pUrb->userPtr = (pVOID) msg.lParam;

    if (usbdCoreEntry (pUrb) == OK)
    {
        /* wait for the URB to complete */
        OSS_SEM_TAKE ((SEM_HANDLE) msg.lParam, OSS_BLOCK);
    }
    else
    {
    /* If the USBD reported an error, we don't know if the URB
     * callback was invoked or not (depends on how bad an error
     * the USBD detected, e.g., malformed URB).    So, we need to
     * make sure the semaphore is returned to the cleared state
     * before we put it back on the queue.
```

```
         */
            OSS_SEM_TAKE ((SEM_HANDLE) msg.lParam, OSS_DONT_BLOCK);
        }

        usbQueuePut (semPoolQueue, 0, 0, msg.lParam, OSS_BLOCK);
        return (pUrb->result == OK) ? OK : ERROR;
}
```

urbExecBlock 函数初始化 URB_HEADER 结构中回调函数指针，从信号量池中获取一个信号量用于接下来的操作，最后调用 usbdCoreEntry 函数将我们的读数据请求进一步向下传递，并等待请求的完成。注意 urbExecBlock 函数虽然接收的是一个 URB_HEADER 结构类型参数，但是从 usbdTransfer 函数来看，其实际上是一个 URB_TRANSFER 结构类型，该结构中 pIrp 指针指向了我们的请求信息。在 urbExecBlock 函数，这个结构继续被向下传递。

11.5.5 第五层：usbdCoreEntry

usbdCoreEntry 函数实现形式如同一个 ioctl 函数，根据请求功能号调用具体的函数进行处理，对于我们读数据请求，usbdTransfer 函数中将请求功能号设置为 USBD_FNC_TRANSFER。usbdCoreEntry 函数较长，我们只给出相关代码。

```
/*******************************************************************************
* target/src/usb/usbdCoreLib.c
* usbdCoreEntry - Primary entry point
*
* usbdCoreEntry is the primary entry point to the USBD through which callers
* invoke individual URBs (USB Request Blocks).
*
* NOTE: This function is intended exclusively for the use of functions in
* usbdLib.    Clients should not construct URBs manually and invoke this
* function directly.
*
* RETURNS: OK or ERROR
*
* ERRNO:
*    S_usbdLib_BAD_PARAM
*/

STATUS usbdCoreEntry (pURB_HEADER pUrb )
{
        int s;

        /* Validate parameters */
        if (pUrb == NULL || pUrb->urbLength < sizeof (URB_HEADER))
            return ossStatus (S_usbdLib_BAD_PARAM);
```

```
/* Unless this call is for initialization or shutdown, take the
 * structMutex to prevent other threads from tampering with USBD
 * data structures.
 */
if (pUrb->function != USBD_FNC_INITIALIZE &&
        pUrb->function != USBD_FNC_SHUTDOWN)
{
    OSS_MUTEX_TAKE (structMutex, OSS_BLOCK);
}

/* Fan-out to appropriate function processor */
switch (pUrb->function)
{
case USBD_FNC_INITIALIZE:
    s = fncInitialize (pUrb);
    break;

…   //其他请求处理代码。

case USBD_FNC_TRANSFER:
    s = fncTransfer ((pURB_TRANSFER) pUrb);
    break;

case USBD_FNC_TRANSFER_ABORT:
    s = fncTransferAbort ((pURB_TRANSFER) pUrb);
    break;

…   //其他请求处理代码。

default:    /* Bad function code */
    s = S_usbdLib_BAD_PARAM;
    break;
}

/* Store result. */
setUrbResult (pUrb, s);

/* release structMutex */
if (pUrb->function != USBD_FNC_INITIALIZE &&
        pUrb->function != USBD_FNC_SHUTDOWN)
{
    OSS_MUTEX_RELEASE (structMutex);
}
```

```
    /* return status to caller */
    if (s == OK || s == PENDING)
        return OK;
    else
        return ossStatus (s);
}
```

正如该函数名称所指，usbdCoreEntry 作为所有 USB 请求类型的总入口函数。usbdCoreEntry 函数定义在 usbdCoreLib.c 文件中，我们将这个文件理解为内核 USB 栈的核心实现文件。正如底层 HC 驱动程序对上也只有一个总入口函数一样，我们可以想见虽然在 usbdCoreEntry 函数将根据不同的请求分发到不同的函数进行处理，但是从 USB 栈进入到底层 HC 驱动时，又要殊途同归，进入到同一个总入口函数，不过此时的总入口函数是由底层 HC 驱动提供的，其也将根据具体的请求分发到不同的函数进行处理，此时将实现请求的最终服务。而在 usbdCoreLib.c 文件中，不同请求分发到不同的函数进行处理的目的是针对特定的请求初始化一些参数，此时不同请求将从 URB_HEADER 结构退回到原始的数据结构表示各自特殊的请求参数信息，对于我们的读数据请求而言，将退回到 URB_TRANSFER 结构。
在 usbdTransfer 函数中，我们的读数据请求被赋予的功能号为 USBD_FNC_TRANSFER，在 usbdCoreEntry 函数对应的分发函数为 fncTransfer，在进行 fncTransfer 函数的调用时，URB_HEADER 结构被退回到原始的 URB_TRANSFER 结构。

11.5.6 第六层：fncTransfer

fncTransfer 函数处理涉及数据（非描述符数据）传输的所有请求。其完成数据传输的启动工作。对于 USB 设备而言，数据传输通道称为管道，是主机端类驱动与 USB 设备中节点之间的一条数据传输通道，每个管道只能进行单向的数据传输。为了启动传输，我们需要管道信息，这个信息在管道创建时被建立，而管道的创建是在类驱动初始化过程中完成，故此处我们需要根据某种信息获取这个管道信息，这里所指的某些信息就是管道号，这个信息由 usbBulkCmdExecute 传入 usbdTransfer，并保存在 URB_TRANSFER 结构的 pipeHandle 字段中。除了管道信息，我们还需要请求信息，这个信息在 usbBulkCmdExecute 函数中被封装在一个 USB_IRP 结构中，这个结构也被保存在 URB_TRANSFER 结构中，由 pIrp 字段指向。所以在传入 fncTransfer 函数的 URB_TRANSFER 结构中存储着一个传输所需的所有信息，现在 fncTransfer 将取出这些信息，启动一个传输过程。该函数实现如下。

```
/*******************************************************************************
* target/src/usb/usbdCoreLib.c
* fncTransfer - Initiates a transfer through a pipe
*
* RETURNS: S_usbdLib_xxxx
*/
LOCAL int fncTransfer (pURB_TRANSFER pUrb)
{
    pUSBD_PIPE pPipe;
    pUSB_IRP pIrp;
```

```
    /* validate the pipe handle */

    if (!validatePipe (pUrb->pipeHandle, &pPipe))
        return S_usbdLib_BAD_HANDLE;

    /* validate the IRP */
    if ((pIrp = pUrb->pIrp) == NULL || pIrp->userCallback == NULL)
        return S_usbdLib_BAD_PARAM;

    /* Fill in IRP fields */
    pIrp->usbdPtr = pPipe;
    pIrp->usbdCallback = transferIrpCallback;
    pIrp->dataToggle = pPipe->dataToggle;

    /* Set an appropriate timeout value */
    if (pPipe->transferType == USB_XFRTYPE_CONTROL ||
            pPipe->transferType == USB_XFRTYPE_BULK)
    {
        if (pIrp->timeout == 0)
            pIrp->timeout = USB_TIMEOUT_DEFAULT;
    }

    /* Update bus statistics */
    if (pIrp->bfrList [0].pid == USB_PID_IN)
        pPipe->pNode->pBus->stats.totalTransfersIn++;
    else
        pPipe->pNode->pBus->stats.totalTransfersOut++;

    /* Link the IRP to the list of IRPs we're tracking on this pipe. */
    usbListLink (&pPipe->irps, pIrp, &pIrp->usbdLink, LINK_TAIL);

    /* Deliver the IRP to the HCD */
    if (usbHcdIrpSubmit (&pPipe->pNode->pBus->pHcd->nexus,
                pPipe->hcdHandle, pIrp) != OK)
    {
        usbListUnlink (&pIrp->usbdLink);
        return S_usbdLib_HCD_FAULT;
    }

    return OK;
}
```

诚如上文所述，fncTransfer 根据 URB_TRANSFER 结构中存储的信息获取管道对应结构

USBD_PIPE 以及请求对应结构 USB_IRP，对这两个结构中的相关字段进行更新和初始化，最后调用 usbHcdIrpSubmit 函数将用户请求传递给 HCD 接口层。
USBD_PIPE 结构对应一个管道，在管道创建时，将与底层 HCD 驱动进行绑定，USBD_PIPE 结构中保存着一个 HCD 句柄，用以对底层 HCD 驱动进行标识。管道创建时将对各层次结构进行初始化和链接，故如下函数调用中第一个参数最终指向一个 HCD_NEXUS 结构，该结构存储了底层 HCD 驱动总入口函数。
usbHcdIrpSubmit (&pPipe->pNode->pBus->pHcd->nexus, pPipe->hcdHandle, pIrp)
USBD_PIPE 结构及其相关的 USBD_NODE，USBD_BUS，USBD_HCD 结构都定义在 target/h/usb/usbdStructures.h 内核头文件中，此处不再给出这些结构的定义，请读者自行查看。
HCD_NEXUS 结构定义如下。

```
/* target/h/usb/usbHcdLib.h
 * HCD_NEXUS
 *
 * HCD_NEXUS contains the entry point and HCD_CLIENT_HANDLE needed by an
 * HCD caller to invoke an HCD.
 */
typedef struct hcd_nexus
{
    HCD_EXEC_FUNC hcdExecFunc;          /* HCD primary entry point */
    HCD_CLIENT_HANDLE handle;       /* client's handle with HCD */
} HCD_NEXUS, *pHCD_NEXUS;
```

HCD_NEXUS 结构中第一个成员 hcdExecFunc 指向底层 USB 主机控制器驱动注册到内核 USB 栈中的总入口函数，所有的用户以及内核 USB 请求都将经过这个总入口函数进入底层驱动进行最终的处理；第二个成员是一个句柄，标识底层 HCD 驱动，这个句柄被上层使用。

在 fncTransfer 函数实现的最后，我们调用 usbHcdIrpSubmit 函数将请求传递给 HCD 接口层，启动一个 USB 传输过程，我们传递了一个 HCD_NEXUS 结构，提供底层 HCD 总入口函数；一个 HCD_PIPE_HANDLE 结构，标识传输使用的管道；一个 USB_IRP 结构，存储着服务请求信息。usbHcdIrpSubmit 函数启动服务所需的所有信息都已经准备完备，只等最后将这些信息传递给底层 HCD 驱动了，usbHcdIrpSubmit 将完成这个最终传递的工作。

11.5.7 第七层：usbHcdIrpSubmit

usbHcdIrpSubmit 是 HCD 接口层提供的实现。HCD 接口层是内核 USB 栈实现的最底层，其下就是底层 HCD 驱动了。usbHcdIrpSubmit 在将请求传递给底层 HCD 驱动之前将进行最后的封装，此时使用的封装结构是 HRB_IRP_SUBMIT 结构。该结构定义如下。

```
/* target/h/drv/usb/usbHcd.h
 * HRB_HEADER
 *
 * All requests to an HCD begin with an HRB (HCD Request Block) header.
 */
typedef struct hrb_header
```

```
{
    HCD_CLIENT_HANDLE handle;/* I/O      caller's HCD client handle */
    UINT16 function;          /* IN       HCD function code */
    UINT16 hrbLength;             /* IN       Length of the total HRB */
} HRB_HEADER, *pHRB_HEADER;
/* target/h/drv/usb/usbHcd.h
 * HRB_IRP_SUBMIT
 */
typedef struct hrb_irp_submit
{
    HRB_HEADER header;        /*     HRB header */
    HCD_PIPE_HANDLE pipeHandle; /* IN     pipe to which IRP is directed */
    pUSB_IRP pIrp;            /* IN       ptr to IRP */
} HRB_IRP_SUBMIT, *pHRB_IRP_SUBMIT;
```

usbHcdIrpSubmit 函数实现如下。

```
/*******************************************************************************
* target/src/usb/usbHcdLib.c
* usbHcdIrpSubmit - Submits an IRP to the HCD for execution
*
* This function passes the <pIrp> to the HCD for scheduling.    The function
* returns as soon as the HCD has queued/scheduled the IRP.    The <pIrp>
* must include a non-NULL <callback> which will be invoked upon IRP
* completion.
*
* RETURNS: OK, or ERROR if unable to submit IRP for transfer.
*/

STATUS usbHcdIrpSubmit
    (
    pHCD_NEXUS pNexus,                  /* client's nexus */
    HCD_PIPE_HANDLE pipeHandle,         /* pipe to which IRP is directed */
    pUSB_IRP pIrp                       /* IRP to be executed */
    )
{
    HRB_IRP_SUBMIT hrb;

    /* Initialize HRB */
    hrbInit (&hrb.header, pNexus, HCD_FNC_IRP_SUBMIT, sizeof (hrb));

    hrb.pipeHandle = pipeHandle;
    hrb.pIrp = pIrp;

    /* Execute HRB */
```

```
    return (*pNexus->hcdExecFunc) ((pVOID) &hrb);
}
```

usbHcdIrpSubmit 首先调用 hrbInit 初始化 HRB_IRP_SUBMIT 结构中 header 字段，hrbInit 函数实现如下。

```
/*******************************************************************************
* target/src/usb/usbHcdLib.c
* hrbInit - Initialize an HCD request block
*
* RETURNS: N/A
*/
LOCAL VOID hrbInit
    (
    pHRB_HEADER pHrb,
    pHCD_NEXUS pNexus,
    UINT16 function,
    UINT16 totalLen
    )
{
    memset (pHrb, 0, totalLen);

    if (pNexus != NULL)
        pHrb->handle = pNexus->handle;

    pHrb->function = function;
    pHrb->hrbLength = totalLen;
}
```

注意 HRB_HEADER 结构 handle 字段被初始化为 HCD_NEXUS 结构中 handle 字段，这个字段是一个句柄，用以标识底层 HCD 驱动。另外 usbHcdIrpSubmit 使用 HCD_FNC_IRP_SUBMIT 功能号对 HRB_HEADER 结构中 function 字段进行了初始化，这个字段将在底层 HCD 驱动总入口函数中被使用进行具体功能实现函数的调用，类似于 usbdCoreEntry 的实现方式。

在保存了管道号和请求参数信息后，usbHcdIrpSubmit 最后调用底层 HCD 驱动总入口函数，完成请求的最终传递。对于 UHCI HCD 驱动而言，这个总入口函数即为 usbHcdUhciExec。这个函数将在 UHCI HCD 驱动初始化过程中通过调用 usbdHcdAttach 函数提供给内核 USB 栈（USB 栈中 HCD 接口层）使用。

11.5.8 第八层：底层 HCD 总入口函数

对于 UHCI 主机控制器驱动而言，这个 HCD 总入口函数就是 usbHcdUhciExec，该函数实现方式类似于 usbdCoreEntry 函数，将根据功能号分发到具体函数进行处理，在 usbHcdIrpSubmit 函数，使用的是 HCD_FNC_IRP_SUBMIT 功能号，如下是 usbHcdUhciExec 函数的相关处理

代码。

```
/*******************************************************************************
*target/src/drv/usb/usbHcdUhciLib.c
* usbHcdUhciExec - HCD_EXEC_FUNC entry point for UHCI HCD
*
* RETURNS: OK or ERROR
*
* ERRNO:
*    S_usbHcdLib_BAD_PARAM
*    S_usbHcdLib_BAD_HANDLE
*    S_usbHcdLib_SHUTDOWN
*/

STATUS usbHcdUhciExec
      (
      pVOID pHrb                /* HRB to be executed */
      )
{
      pHRB_HEADER pHeader = (pHRB_HEADER) pHrb;
      pHCD_HOST pHost;
      int s;

      …

            /* Fan-out to appropriate function processor */
            switch (pHeader->function)
            {
            case HCD_FNC_ATTACH:
                  s = fncAttach ((pHRB_ATTACH) pHeader);
                  break;
            …

            case HCD_FNC_IRP_SUBMIT:
                  s = fncIrpSubmit ((pHRB_IRP_SUBMIT) pHeader, pHost);
                  break;

            case HCD_FNC_IRP_CANCEL:
                  s = fncIrpCancel ((pHRB_IRP_CANCEL) pHeader, pHost);
                  break;

            …

            default:
                  s = S_usbHcdLib_BAD_PARAM;
```

```
            break;
        }

        …

        /* Return status */
        if (s == OK || s == PENDING)
            return OK;
        else
            return ossStatus (s);
}
```

可以看到，对应用户读数据请求，这个请求最终将由底层 HCD 驱动中 fncIrpSubmit 函数处理。此时数据结构将重新退回到原始的 pHRB_IRP_SUBMIT 结构，这个结构已在 usbHcdIrpSubmit 函数进行了初始化。

11.5.9 请求传递过程总结

从以上的介绍中，我们可以看到用户 USB 请求传递过程中，主要经过两个明显的节点，第一个节点在 USB 栈实现的核心层中，具体对应 usbdCoreLib.c 实现文件，该文件中定义的 usbdCoreEntry 函数将作为所有上层请求的总入口函数，所有的请求都由这个函数根据请求功能号的不同进行分发，此时不同的处理函数将根据各自处理的特定请求分配各自所需的数据结构，对该请求相关的参数和信息进行封装；第二个节点就是底层 USB 控制器驱动总入口函数，如 UHCI 控制器驱动总入口函数 usbHcdUhciExec，所有的 USB 请求经过 usbdCoreEntry 的分发后最终殊途同归，又进入到同一个函数中进行统一处理，虽然进入到同一个函数，但是底层数据结构根据不同的请求各不相同，在驱动总入口函数（如 usbHcdUhciExec）实现中，采用和 usbdCoreEntry 相同的实现方式，也是根据不同的功能号进行分发，此时调用驱动总入口函数时传入的结构将还原到原始的结构，由具体的功能实现函数进行使用，完成请求的最终服务。
以 usbdCoreEntry 函数和底层驱动总入口函数为节点，我们可以简单的对请求的传递过程分为两个阶段，每个阶段中又包括准备和处理两个子阶段。第一阶段从类驱动层到 usbdCoreEntry 函数；第二阶段则从 usbdCoreEntry 中调用的分发函数到底层驱动总入口函数。每个阶段中的准备子阶段主要完成对应特定请求的数据结构的初始化，即请求参数和信息的封装过程，在这个过程中将赋予该请求一个特定的功能号，最终到达 usbdCoreEntry 函数完成一次集中处理；usbdCoreEntry 根据功能号将调用不同的实现函数进一步进行处理，这就进入到第二阶段，这个阶段的准备工作如同第一阶段，进一步进行封装，并在调用底层总入口函数之前再次对请求进行功能号的分配，最终在底层总入口函数中再一次进行集中处理。注意到 usbdCoreEntry 中使用的功能号都是以“USBD_FNC_”开始，而底层驱动总入口函数中使用的功能号都是以“HCD_FNC_”开始。其中 USBD_FNC 表示被 USB 栈使用的功能号，而 HCD_FNC 则表示被底层 HC 驱动使用的功能号，界限分明。

在前文的分析中，我们只是跟随了一个读数据请求的传递过程，并简单介绍了该请求传递过程中调用的函数以及相关数据结构。不同的请求，在传递过程中基本具有相同的函数调用路

径，但是数据结构则完全不同，这从上一段分析中也可看出，usbdCoreEntry 将根据不同的功能号调用不同的函数，使用不同的数据结构，所以总的路径大致相同，但是数据结构则决然不同，这也对理解 USB 栈的实现形成了较大的障碍。在进行底层 HC 驱动设计时，我们必须知道各种请求对应使用的数据结构定义，并清楚相关结构之间的嵌套关系，这样才能查询到我们所需的各种信息。因为只有一个底层总入口函数，所以底层 HC 驱动开发者不可不自行弄清楚每个功能号的确切含义及其使用的所有相关内核结构。从前文代码的分析中，可以看出，底层 HC 驱动和内核 USB 栈实现之间并不区分数据结构的界限，不像其他设备类型驱动，数据结构具有严格的界限，一般内核只提供少数几个(通常一个，如 BLK_DEV，SIO_CHAN，DEV_HDR)数据结构给底层驱动用以向内核层注册时使用，而在底层 HC 驱动中也如同内核 USB 栈中一样随意使用各种内核结构，这共同构成了底层 HC 驱动设计的复杂性。

11.6 应用层类驱动初始化

以 Mass Storage USB 设备为例，其对应的类驱动在主机端实现了对 Mass Storage USB 设备的各项操作接口。Mass Storage 类驱动将 Mass Storage USB 设备封装成一个普通块设备，我们可以在 Mass Storage 类驱动之上架设文件系统层，以文件和目录视图的方式操作 Mass Storage USB 设备。在此我们必须提及，USB 设备硬件结构以及类驱动(包括设备系统软件，我们一般称之为固件)共同组成了一个 USB 设备的具体功能。如 USB 键盘，USB 鼠标，USB 存储设备，其底层都是使用 USB 接口传递数据，然而其外在表项的行为和功能则完全不同，这都是由设备内部硬件构成以及类驱动的不同造成的。对于 USB 存储设备而言（如常见的 U 盘设备），其内部硬件构成上必不可少将要包括数据存储设备（大多使用 NandFlash），运行固件的单片机（通常使用 8051 系列单片机），USB 控制器等，主机端必须配备 Mass Storage 类驱动才能真正实现 USB 存储设备要完成的功能：存储用户数据，缺少其中任何一样都不行。由于 USB 类驱动位于内核 USB 栈之上，与底层硬件相隔离，且对于每种功能的 USB 设备一般都有规范，故操作系统本身对于 USB 类驱动都提供了较为完善的支持，包括 Vxworks。以实现存储功能的 USB 设备为例，其对应“Universal Serial Bus Mass Storage Class”规范。该规范对实现 Mass Storage 功能的设备与主机之间的数据交互有确定的定义，包括为达到块数据读写目的而定义的相关结构（如上文中提到的 USB_BULK_CBW 和 USB_BULK_CSW 结构）和操作流程（如上文中提到的为完成一次数据读写而进行的三阶段操作过程：命令，数据，状态）。

Vxworks 下提供了如下几类 USB 类驱动（USB Class Driver）支持：

（1） 根 HUB 类驱动
（2） 键盘（Keyboard）类驱动
（3） 鼠标（Mouse）类驱动
（4） 打印机（Printer）类驱动
（5） 播音器（Speaker）类驱动
（6） 大容量存储（Mass Storage）类驱动
（7） USB 接口网络通信（Communication）类驱动

本节我们将以 Bulk-Only 大容量存储（Mass Storage）类驱动为例，介绍类驱动的初始化过程。Mass Storage 类驱动将用以存储目的的 USB 外设封装成一个普通的块设备，我们可以

在 Mass Storage 类驱动之上挂载一个文件系统层，以文件和目录的方式在 USB 外设上存储数据。

Bulk-Only Mass Storage 类驱动在 Vxworks 启动过程中进行初始化，这个初始化函数将在 usrRoot 中被调用，如下所示。

```
#ifdef INCLUDE_USB_MS_BULKONLY_INIT
    usrUsbBulkDevInit();    /* Bulk Driver Initialization */
#endif
```

为了对 Mass Storage 类驱动进行初始化，我们包括 INCLUDE_USB_MS_BULKONLY_INIT 宏定义。usrUsbBulkDevInit 是 Bulk-Only Mass-Storage 类驱动初始化入口函数，该函数定义在 target/comps/src/usrUsbBulkDevInit.c 文件中。

usrUsbBulkDevInit 函数完成：（1）调用 usbBulkDevInit 函数完成具体的 Mass Storage 类驱动初始化工作；（2）创建 tBulkClnt 任务对底层驱动的变动作出及时响应；（3）调用 usbBulkDynamicAttachRegister 函数注册回调函数 bulkAttachCallback，这个回调函数将在一个 USB 设备挂接入系统时被调用，这个函数将保存内核挂接 USB 设备时分配的句柄，这个句柄标识着这个 USB 设备，或者称之为 USB 设备地址。

下面我们还将回到 tBulkClnt 任务和此处保存的 USB 设备地址进行讨论。

usbBulkDevInit 完成具体的 Mass Storage 类驱动的初始化工作，该函数定义在 target/src/drv/usb/usbBulkDevLib.c 文件中。该函数完成：（1）调用 usbdClientRegister 向内核 USB 栈注册并获取一个标识该类驱动的句柄；（2）调用 usbdDynamicAttachRegister 注册回调函数，这个回调函数在发生一个 USB 事件时（如插入或者移除一个设备）被调用，在 usbBulkDevInit 中注册的回调函数为 usbBulkDevAttachCallback。

usbBulkDevAttachCallback 是一个回调函数，在发生一个 USB 事件时被调用，该函数完成的工作十分重要，在 USB 外设插入时，将完成类驱动与底层 HCD 连接以及 Mass Storage USB 外设通信管道的创建工作，而当 USB 外设移除时，则释放在该外设插入时分配的所有资源。此处我们对插入一个 Mass Storage USB 设备为例，此时 usbBulkDevAttachCallback 函数将被调用，其将具体调用 usbBulkPhysDevCreate 完成对 Mass Storage USB 外设的配置，创建上行，下行通信管道，创建 USB 设备结构 USB_BULK_DEV 来表示这个外设，并完成与底层 HCD 驱动的衔接。

当一个 Mass Storage USB 设备挂载到系统时，内核 USB 栈实现会分配一个唯一的标识符来标识这个设备，我们通常将这个标识符称为 USB 设备地址，在 Mass Storage 类驱动中，每个 USB 设备对应一个 USB_BULK_DEV 结构，与 USB 栈分配的 USB 设备地址一一对应，实际上 USB_BULK_DEV 的获取就是通过 USB 设备地址进行的，负责这个工作的函数就是 usbBulkDevFind，其接收一个 USBD_NODE_ID 结构类型的参数，返回一个 USB_BULK_DEV 结构。这个结构在 usbBulkPhysDevCreate 函数中被创建和初始化，其中包含了主机端与 Mass Storage USB 外设进行通信的所有信息，包括已经创建通信管道。

在一个 Mass Storage USB 外设挂载入系统后，usbBulkDevAttachCallback 函数被调用，其调用 usbBulkPhysDevCreate 函数完成该 USB 设备对应 USB_BULK_DEV 结构的创建和初始化，同时完成 USB 设备的配置，与 USB 设备通信管道的建立，以及与底层 HCD 驱动的衔接工

作。至此完成所有的初始化工作。

要使得应用层用户可以对这个 Mass Storage USB 外设进行数据读写操作，我们必须在 Mass Storage 类驱动之上架设文件系统层，这个工作的完成由在 usrUsbBulkDevInit 创建的 tBulkClnt 任务完成，该任务通过 bulkAttachCallback 回调函数返回的 USB 设备地址获取设备对应的 USB_BULK_DEV 结构，并进而调用 dcacheDevCreate 和 dosFsDevCreate 函数完成 dosFs 文件系统的架设，至此应用层就可以以文件和目录的方式操作这个挂接的 Mass Storage USB 外设。注意 USB_BULK_DEV 第一个成员就是一个 BLK_DEV 结构类型，所以实际上 Mass Storage 类驱动自身实现上就将一个 Mass Storage USB 外设封装成普通块设备，以便可以使用文件系统对这个设备进行数据读写操作。

读者可以查看 tBulkClnt 任务对应的执行函数 bulkClientThread，该函数在一个 Mass Storage USB 外设挂接入系统后，将调用 bulkMountDrive 函数在主机端 Mass Storage 类驱动层之上架设文件系统层。bulkClientThread 以及 bulkMountDrive 函数均定义在 target/comps/src/usrUsbBulkDevInit.c 文件中，此处我们给出 bulkMountDrive 函数实现以供读者查看，结合本书块设备驱动一章的内容，该函数实现代码应不难理解。

```
/*************************************************************************
* target/config/comps/src/usrUsbBulkDevInit.c
* bulkMountDrive - mounts a drive to the DOSFS.
*
* RETURNS: OK or ERROR
*/

LOCAL STATUS bulkMountDrive (USBD_NODE_ID attachCode)
{
    CBIO_DEV_ID cbio, masterCbio;

    /* Create the block device with in the driver */
    pBulkBlkDev = (BLK_DEV *) usbBulkBlkDevCreate (bulkNodeId,
                                0,
                                0,
                                USB_SCSI_FLAG_READ_WRITE10);

    if (pBulkBlkDev == NULL)
    {
        logMsg ("usbBulkBlkDevCreate() returned ERROR\n", 0, 0, 0, 0, 0, 0);
        return ERROR;
    }

    /* optional dcache */
    cbio = dcacheDevCreate ((CBIO_DEV_ID) pBulkBlkDev, 0, 0, "usbBulkCache");

    if( NULL == cbio )
```

```
    {
        /* insufficient memory, will avoid the cache */
        printf ("WARNING: Failed to create disk cache\n");
    }

    masterCbio = dpartDevCreate (cbio, 1, usrFdiskPartRead);
    if( NULL == masterCbio )
    {
        printf ("Error creating partition manager\n");
        return ERROR;
    }

    /* Mount the drive to DOSFS */
    if (dosFsDevCreate (BULK_DRIVE_NAME,    dpartPartGet(masterCbio, 0),
            0x20,NONE) == ERROR)
    {
            printf ("Error creating dosFs device\n");
            return ERROR;
    }
    return OK;
}
```

在一个 Mass Storage USB 设备被接入系统后，bulkMountDrive 自动被调用，其首先调用 usbBulkBlkDevCreate 函数（下议）创建一个机遇 Mass Storage USB 设备的块设备视图，即对 USB_BULK_DEV 结构中 BLK_DEV 结构成员进行初始化，并返回这个初始化后的 BLK_DEV 结构供外界使用。bulkMountDrive 函数之后使用 usbBulkBlkDevCreate 函数返回的 BLK_DEV 结构连续调用 dcacheDevCreate，dpartDevCreate 添加 CBIO 数据缓冲层，分区管理层，最后调用 dosFsDevCreate 函数完成块设备节点的创建和文件系统层的加载。此后这个 Mass Storage USB 设备就以一个普通块设备对用户可见，用户层可以使用基于文件系统的接口函数以文件和目录的方式读写这个 Mass Storage USB 设备。

usbBulkBlkDevCreate 函数完成的功能其实很简单，其初始化 USB_BULK_DEV 结构中的第一个成员，该成员是一个 BLK_DEV 结构类型的字段，用以提供 USB 设备的块设备视图。我们在上文中提到，每个 Mass Storage USB 外设在 Mass Storage 类驱动中都对应有一个 USB_BULK_DEV 结构，该结构包含了访问这个 USB 外设的所有信息，包括用以上行，下行数据传输的两个管道也已经完成建立，这些工作在这个 USB 外设插入系统后，由回调函数调用 usbBulkPhysDevCreate 函数自动完成的。现在为了对用户层提供读写 USB 外设的手段，Mass Storage 类驱动将 USB 外设封装成一个普通块设备，即返回一个 BLK_DEV 给其他层使用，这就回到本书块设备驱动一章的内容中，我们使用 BLK_DEV 结构构建 CBIO 中间层，继而完成文件系统层的添加，至此这个 Mass Storage USB 设备就以文件和目录的方式对用户层可见。

usbBulkBlkDevCreate 函数初始化 BLK_DEV 结构的相关代码如下。

```
/*******************************************************************************
* target/src/drv/usb/usbBulkDevLib.c
```

```
* usbBulkBlkDevCreate - create a block device.
*
* This routine    initializes a BLK_DEV structure, which describes a
* logical partition on a USB_BULK_DEV device.    A logical partition is an array
* of contiguously addressed blocks; it can be completely described by the number
* of blocks and the address of the first block in the partition.
*
* NOTE:
* If `numBlocks' is 0, the rest of device is used.
*
* This routine supplies an additional parameter called <flags>.    This bitfield
* currently only uses bit 1.    This bit determines whether the driver will use a
* SCSI READ6 or SCSI READ10 for read access.
*
* RETURNS: A pointer to the BLK_DEV, or NULL if parameters exceed
* physical device boundaries, or if no bulk device exists.
*/

BLK_DEV * usbBulkBlkDevCreate
      (
      USBD_NODE_ID nodeId,              /* nodeId of the bulk-only device          */
      UINT32             numBlks,            /* number of logical blocks on device */
      UINT32             blkOffset,         /* offset of the starting block             */
      UINT32             flags           /* optional flags                        */
      )
      {
      UINT8 * pInquiry;                /* store for INQUIRY data    */

      pUSB_BULK_DEV pBulkDev    = usbBulkDevFind (nodeId);
      …
      /*
        * Initialize the standard block device structure for use with file
        * systems.
        */

        pBulkDev->blkDev.bd_blkRd              = (FUNCPTR) usbBulkDevBlkRd;
        pBulkDev->blkDev.bd_blkWrt             = (FUNCPTR) usbBulkDevBlkWrt;
        pBulkDev->blkDev.bd_ioctl              = (FUNCPTR) usbBulkDevIoctl;
        pBulkDev->blkDev.bd_reset              = (FUNCPTR) usbBulkDevReset;
        pBulkDev->blkDev.bd_statusChk       = (FUNCPTR) usbBulkDevStatusChk;
        pBulkDev->blkDev.bd_retry              = 1;
        pBulkDev->blkDev.bd_mode                = O_RDWR;
        pBulkDev->blkDev.bd_readyChanged = TRUE;
```

```
    …
    return (&pBulkDev->blkDev);
}
```

可以看到，在 usbBulkBlkDevCreate 函数，读写函数分别初始化为 usbBulkDevBlkRd 和 usbBulkDevBlkWrt，这正是 9.5 节中我们跟随一个 USB 请求的起点函数，而用户层请求到这两个读写函数的传递过程如下（以读数据操作为例）：read→iosRead→dosFsRead→usbBulkDevBlkRd。

注意：

（1） 在 bulkMountDrive 函数中，调用 usbBulkBlkDevCreate 函数，第二个参数传入的值是 0，这表示将使用整个设备作为单个分区进行文件系统的加载。通常对于实现存储目的的 Mass Storage USB 设备而言，整个设备作为一个分区使用。

（2） 当一个 Mass Storage USB 设备插入系统时，该设备的加载是由内核自动完成的，无需用户层进行任何函数的调用。内核将通过回调函数的方式，完成相关结构的分配和初始化，并在 Mass Storage 类驱动提供的块设备视图之上加载文件系统，创建设备节点。所以用户无需做任何辅助性工作，即可直接使用 open，read，write 等函数打开这个 Mass Storage 设备中的文件，进行读写操作。

（3） USB 设备支持动态插入和移除，即支持热插拔。热插拔首先是从硬件上进行支持的，当发生热插拔事件时，硬件通知给软件，软件通过调用已注册的回调函数进行响应，这构成了（2）中各项的工作会自动完成的前提，而无须用户从中进行任何帮助。

11.7 USB 控制器驱动初始化

以 UHCI 控制器驱动为例，本节将介绍 USB 控制器驱动（Host Controller Driver：HCD）初始化过程。从本章前文内容可知，HCD 只提供一个总入口函数给内核 USB 栈 ，所有经过的内核 USB 栈转发的请求都将通过这个总入口进入到底层 HCD 进行处理，所以 HCD 的初始化主要完成将这个总入口函数注册给内核 USB 栈相关组件使用的工作。在 9.5.7 节中我们看到 usbHcdIrpSubmit 函数最后通过调用 HCD_NEXUS 结构中 hcdExecFunc 字段指向的函数进入到这个底层 HCD 总入口函数，所以在以下的介绍中，我们将着重阐述 HCD 底层驱动总入口函数是如何初始化一个 HCD_NEXUS 结构的。

UHCI HCD 驱动初始化入口函数在 Vxworks 启动过程中在 usrRoot 函数中被调用，如下所示。

```
#ifdef INCLUDE_UHCI_INIT
    usrUsbHcdUhciAttach ();      /* UHCI Initialization */
#endif
```

为了对 UHCI HCD 进行初始化，我们还必须进行 INCLUDE_UHCI_INIT 宏定义。
usrUsbHcdUhciAttach 实现代码如下所示。

```
/*******************************************************************************
* target/config/comps/src/usrUsbHcdUhciInit.c
```

```
* usrUsbHcdUhciAttach - attaches a Host Controller to the USB Stack
*
* This functions searches the pci bus for either a OHCI or UHCI type
* USB Host Controller.    If it finds one, it attaches it to the already
* initialized USB Stack
*
* RETURNS: OK if sucessful or ERROR if failure
*/
LOCAL STATUS usrUsbHcdUhciAttach ()
{
        UINT16 ret;
        UINT8 busNo;
        UINT8 deviceNo;
        UINT8 funcNo;
        PCI_CFG_HEADER pciCfgHdr;
        UINT8 pciClass        = UHCI_CLASS;
        UINT8 pciSubclass = UHCI_SUBCLASS;
        UINT8 pciPgmIf        = UHCI_PGMIF;
        HCD_EXEC_FUNC execFunc = &usbHcdUhciExec;
        GENERIC_HANDLE *pToken = &uhciAttachToken;

        if (*pToken == NULL)
        {
                if (!usbPciClassFind (pciClass, pciSubclass, pciPgmIf, 0,
                                &busNo, &deviceNo, &funcNo))
                {
                        printf ("No UHCI host controller found.\n");
                        return ERROR;
                  }

                printf ("UHCI Controller found.\n");
                printf ("Waiting to attach...");

                usbPciConfigHeaderGet (busNo, deviceNo, funcNo, &pciCfgHdr);

                /* Attach the UHCI HCD to the USBD. */
                ret = usbdHcdAttach (execFunc, &pciCfgHdr, pToken);

                if (ret == OK)
                {
                     printf ("Done, AttachToken = 0x%x\n", (UINT) *pToken);
                }
                else
                {
```

```
            printf ("USB HCD Attached FAILED!\n");
            return ERROR;
        }
    }
    else
    {
        printf("UHCI Already Attached\n");
    }

    return OK;
}
```

usrUsbHcdUhciAttach 首先调用 usbPciClassFind 函数寻找要驱动的 UHCI 主机控制器（Host Controller：HC）设备，而后调用 usbPciConfigHeaderGet 读取 HC 设备 PCI 配置空间头部信息，最后调用 usbdHcdAttach 注册 HCD 总入口函数，对于 UHCI HCD 而言，这个总入口函数就是 usbHcdUhciExec，在 9.5.8 节我们已经给出了该函数的部分代码，该函数是所有 USB 请求的总入口函数，由该函数根据请求号的不同再调用不同的函数进行处理。
此处假设了 HC 设备通过 PCI 总线接入系统，对于嵌入式系统而言，通常没有 PCI 总线，此时我们可以直接构建 HC 设备的信息，再调用 usbdHcdAttach 函数，无需对 usbPciClassFind 和 usbPciConfigHeaderGet 进行调用。

usbdHcdAttach 函数实现如下。

```
/*******************************************************************************
* target/src/usb/usbdLib.c
* usbdHcdAttach - Attaches an HCD to the USBD
*
RETURNS: OK, or ERROR if unable to attach HCD.
*/
STATUS usbdHcdAttach
    (
    HCD_EXEC_FUNC hcdExecFunc,          /* Ptr to HCD 担  primary entry point */
    pVOID param,            /* HCD-specific parameter */
    pGENERIC_HANDLE pAttachToken  /* Token to identify HCD in future */
    )
{
    URB_HCD_ATTACH urb;
    STATUS s;

    /* Initalize URB */
    urbInit (&urb.header, NULL, USBD_FNC_HCD_ATTACH, NULL, NULL,
    sizeof (urb));

    urb.hcdExecFunc = hcdExecFunc;
```

```
    urb.param = param;

    /* Execute URB */
    s = urbExecBlock (&urb.header);

    /* Return result */
    if (pAttachToken != NULL)
        *pAttachToken = urb.attachToken;

    return s;
}
```

对于 HCD 注册请求，usbdHcdAttach 函数将其封装在一个 URB_HCD_ATTACH 结构中，因为所有的请求最终都要经过 usbdCoreEntry 函数进行集中处理，不同请求将使用不同的结构保存各自所需的信息，对于数据传输而言，使用 URB_TRANSFER 结构，而对于 HCD 注册而言，则使用 URB_HCD_ATTACH 结构。URB_HCD_ATTACH 结构定义如下。

```
/* target/h/usb/usbdCoreLib.h
 * URB_HCD_ATTACH
 */
typedef struct urb_hcd_attach
{
    URB_HEADER header;              /*          URB header */
    HCD_EXEC_FUNC hcdExecFunc;          /* IN     HCD primary entry point */
    pVOID param;                    /* IN     HCD-specific parameter */
    GENERIC_HANDLE attachToken;     /* OUT    attach token */
} URB_HCD_ATTACH, *pURB_HCD_ATTACH;
```

URB_HEADER 结构与 URB_HCD_ATTACH 定义在同一个文件中，这是所有请求结构具有的共同的成员，用以存储请求号。此处请求号被设置为 USBD_FNC_HCD_ATTACH，usbdCoreEntry 将使用该请求号调用具体的函数进行处理。

而 usrUsbHcdUhciAttach 传入的 UHCI HCD 总入口函数 usbHcdUhciExec 则被保存到 URB_HCD_ATTACH 结构的 hcdExecFunc 字段中。

之后 usbdHcdAttach 调用 urbExecBlock，最终进入到 usbdCoreEntry 函数进行执行，对应于 USBD_FNC_HCD_ATTACH 请求号，usbdCoreEntry 将调用 fncHcdAttach 函数进行具体的处理。

fncHcdAttach 部分实现代码如下。

```
/*******************************************************************************
* target/src/usb/usbdCoreLib.c
* fncHcdAttach - Attach an HCD to the USBD
*
* RETURNS: S_usbdLib_xxxx
*/
LOCAL int fncHcdAttach
    (
```

```
    pURB_HCD_ATTACH pUrb
    )
{
    pUSBD_HCD pHcd = NULL;
    UINT16 busNo;
    int s;

    /* validate URB */
    if ((s = validateUrb (pUrb, sizeof (*pUrb), NULL)) != OK)
        return s;


    /* Allocate structure for this host controller */
    if ((pHcd = OSS_CALLOC (sizeof (*pHcd))) == NULL)
        return S_usbdLib_OUT_OF_MEMORY;

    /* Issue an attach request to the HCD.  If it succeeds, determine the
     * number of buses managed by the HCD.
     */
    if (usbHcdAttach (pUrb->hcdExecFunc, pUrb->param, hcdMngmtCallback,
            pHcd, &pHcd->nexus, &pHcd->busCount) != OK)
    {
        OSS_FREE (pHcd);
        return S_usbdLib_GENERAL_FAULT;
    }

    …  //其他代码省略。

    return s;
}
```

在 usbdHcdAttach 函数中，我们将 URB_HCD_ATTACH 结构中 hcdExecFunc 初始化为指向 UHCI HCD 总入口函数，param 指向了 HC 设备 PCI 配置空间头部。在 fncHcdAttach 实现中，这两个字段以及和新创建的 USBD_HCD 结构一道被用作参数调用 usbHcdAttach 函数。USBD_HCD 结构定义如下，注意 USBD_HCD 结构中 nexus 字段是一个 HCD_NEXUS 结构类型。

```
/* target/h/usb/usbdStructures.h
 * USBD_HCD
 *
 * The USBD maintains a list of currently attached HCDs.  Each HCD may control
 * one or more buses.
 */
typedef struct usbd_hcd
{
```

```
    LINK hcdLink;                   /* linked list of HCDs */
    GENERIC_HANDLE attachToken;     /* Attach token returned for this HCD */
    HCD_NEXUS nexus;                /* identifies connection to HCD */
    UINT16 busCount;                /* number of buses managed by HCD */
    pUSBD_BUS pBuses;               /* buses managed by HCD */
} USBD_HCD, *pUSBD_HCD;
```

usbHcdAttach 函数实现如下。

```
/*******************************************************************************
* target/src/usb/usbHcdLib.c
* usbHcdAttach - Attach to the HCD
*
* Attempts to connect the caller to the HCD.    The <param> value is HCD-
* implementation-specific.    Returns an HCD_CLIENT_HANDLE if the HCD was
* able to initialize properly.    If <pBusCount> is not NULL, also returns
* number of buses managed through this nexus.
*
* <callback> is an optional pointer to a routine which should be invoked
* if the HCD detects "management events" (e.g., remote wakeup/resume).
* <callbackParam> is a caller-defined parameter which will be passed to
* the <callback> routine each time it is invoked.
*
* RETURNS: OK, or ERROR if unable to initialize HCD.
*/
STATUS usbHcdAttach
    (
    HCD_EXEC_FUNC hcdExecFunc,          /* HCD's primary entry point */
    pVOID param,            /* HCD-specific param ,pci config header*/
    USB_HCD_MNGMT_CALLBACK callback,/* management callback */
    pVOID callbackParam,        /* parameter to management callback */
    pHCD_NEXUS pNexus,              /* nexus will be initialized on return */
    pUINT16 pBusCount
    )
{
    HRB_ATTACH hrb;
    STATUS s;

    /* Initialize HRB */
    hrbInit (&hrb.header, NULL, HCD_FNC_ATTACH, sizeof (hrb));

    hrb.param = param;
    hrb.mngmtCallback = callback;
```

```
    hrb.mngmtCallbackParam = callbackParam;

    /* Execute HRB */
    s = (*hcdExecFunc) ((pVOID) &hrb);

    /* Return results */
    if (pNexus != NULL)
    {
        pNexus->hcdExecFunc = hcdExecFunc;
        pNexus->handle = hrb.header.handle;
    }

    if (pBusCount != NULL)
        *pBusCount = hrb.busCount;

    return s;
}
```

usbHcdAttach 初始化一个 HRB_ATTACH 结构，以此调用底层 HCD 总入口函数，使得底层驱动进行一些操作配合完成其本身的注册过程，如进行 USB 外设的一些配置或者分配其所需的各种资源等等。usbHcdAttach 函数最后将 HCD 驱动总入口函数保存到 HCD_NEXUS 结构中，完成 HCD 的最终注册。至此，底层 HCD 驱动就处于正常工作状态，接收所有经过内核 USB 栈转发的请求，如配置 USB 外设，创建通信管道，读写设备等等。
HRB_ATTACH 结构定义在 target/h/drv/usb/usbHcd.h 内核头文件中，感兴趣读者可自行进行查看。

11.8 USB 控制器驱动结构

USB 控制器工作完全由中断进行驱动，不论是发送还是接收，在完成一次操作后，控制器都将发出一个中断，由系统软件根据中断源进行具体的处理。所以控制器驱动中一个中断处理函数是一个中心点。但是基于中断处理函数执行在中断上下文中，对许多的操作有限制，更重要的是中断处理函数如果执行时间较长，将导致整个系统的性能下降，故实现上采用将中断分为两个部分的策略：“上半部分”进行某些关键操作，并安排“下半部分”完成较具体的工作；“下半部分”则具体完成对中断的响应。“上半部分”仍然执行在中断上下文中，而“下半部分”将执行在任务上下文中。
在 UHCI 控制器驱动实现中，“上半部分”和“下半部分”之间采用信号量进行同步，并专门创建一个后台任务处理“下半部分”，注意这个后台任务是由驱动自身创建的，而非内核提供（这与网络设备驱动中的 tNetTask 任务有区别，tNetTask 是内核创建的）。由 UHCI HC 驱动创建的这个后台任务平时处于阻塞于同步信号量，每当有一个 HC 中断到达，HCD 中断处理函数即释放信号量，此时后台任务将退出阻塞状态，并检查中断源，调用具体的函数对中断进行处理，这些处理主要是对上层请求的服务完成后的一些后处理操作。
而上层请求的服务启动则由 HCD 驱动总入口函数完成，该函数作为与内核 USB 栈之间唯

一的通信通道，将接收经 USB 栈转发的所有请求，其将根据请求号的不同分发到底层驱动中实现的具体函数进行处理，如启动一个传输，当传输完成后，HC 硬件将发出一个中断，由驱动创建的后台任务以及中断处理函数共同完成请求的后处理，如将读取的数据返回给上层使用。

基于以上讨论，我们可以总结出 USB 主机控制器驱动的结构如下。

（1） 后台任务，负责中断的具体处理。

（2） 中断处理函数，负责中断的初级响应，唤醒后台任务进行具体的处理。

（3） 总入口函数，接收所有经 USB 栈转发的请求，根据请求号的不同分发到底层驱动中具体实现函数进行处理。

（4） 底层请求处理函数，这些函数被总入口函数调用，完成对某个请求的实际处理。

（5） 针对实际主机控制器相关数据结构定义。

以上 5 点构成了 HCD 驱动的基本结构，表面上看似比较简单，但是每个函数的实现都比较复杂，这与主机控制器操作方式有关。另外因为只有一个总入口函数，故必须清楚每个请求号对应要完成的具体功能，这需要对 USB 栈实现以及 USB 规范本身都比较了解。

注意：对于 USB 设备操作，由于在中断中需要涉及数据的处理，故通常将这个处理过程放入任务上下文中进行，这对于所有在中断中需要处理大量数据的驱动通常都遵循这个规则，故以上给出的 USB 主机控制器驱动中后台任务的创建是必须的。对于像网络设备之类的驱动，内核提供后台任务，而对于像 USB 设备之类的驱动，后台任务需要底层驱动自行建立。

11.9 USB 控制器驱动实现

USB 控制器驱动的具体实现围绕两个中心函数进行：（1）请求总入口函数，该函数上层所有请求的入口函数，不同的请求将根据请求号分发到具体的底层实现函数进行处理；（2）中断处理函数，本质上中断的实际处理是在后台任务中，故我们将后台任务入口函数作为第二个中心函数。

对应于 UHCI 控制器驱动，第一个中心函数就是 usbHcdOhciExec，第二个中心函数就是 intThread（中断入口函数为 intHandler）。

对于控制器驱动的设计，首先必须把握总入口函数的实现，做好与上层的衔接工作，将上层提出的每个请求都弄清楚，之后根据特定的主机控制器类型实现对应的函数即可。故首先我们分析总入口函数中需要进行响应的请求号以及各请求号对应的确切含义。

11.9.1 总入口函数实现

UHCI 主机控制器驱动总入口函数响应的请求号及其对应响应函数和相关说明如下图 11-15 所示。

请求号	底层响应函数	说明
HCD_FNC_ATTACH	fncAttach	HCD加载
HCD_FNC_DETACH	fncDetach	HCD卸载
HCD_FNC_SET_BUS_STATE	fncSetBusState	设置总线状态（挂起/恢复）
HCD_FNC_SOF_INTERVAL_GET	fncSofIntervalGet	获取当前SOF间隔
HCD_FNC_SOF_INTERVAL_SET	fncSofIntervalSet	设置当前SOF间隔
HCD_FNC_CURRENT_FRAME_GET	fncCurrentFrameGet	获取当前帧序号
HCD_FNC_IRP_SUBMIT	fncIrpSubmit	启动一个传输过程
HCD_FNC_IRP_CANCEL	fncIrpCancel	取消一个已启动的传输过程
HCD_FNC_PIPE_CREATE	fncPipeCreate	创建通信管道
HCD_FNC_PIPE_DESTROY	fncPipeDestroy	销毁通信管道
HCD_FNC_PIPE_MODIFY	fncPipeModify	修改通信管道参数（如最大包长度）

图 11- 15 UHCI HCD 总入口函数请求号及其响应函数

OHCI 主机控制器驱动总入口函数响应的请求号及其对应响应函数和相关说明如下图 11-16 所示。

请求号	底层响应函数	说明
HCD_FNC_ATTACH	fncAttach	HCD加载
HCD_FNC_DETACH	fncDetach	HCD卸载
HCD_FNC_SET_BUS_STATE	fncSetBusState	设置总线状态（挂起/恢复）
HCD_FNC_SOF_INTERVAL_GET	fncSofIntervalGet	获取当前SOF间隔
HCD_FNC_SOF_INTERVAL_SET	fncSofIntervalSet	设置当前SOF间隔
HCD_FNC_CURRENT_FRAME_GET	fncCurrentFrameGet	获取当前帧序号
HCD_FNC_IRP_SUBMIT	fncIrpSubmit	启动一个传输过程
HCD_FNC_IRP_CANCEL	fncIrpCancel	取消一个已启动的传输过程
HCD_FNC_PIPE_CREATE	fncPipeCreate	创建通信管道
HCD_FNC_PIPE_DESTROY	fncPipeDestroy	销毁通信管道
HCD_FNC_PIPE_MODIFY	fncPipeModify	修改通信管道参数（如最大包长度）

图 11- 16 OHCI HCD 总入口函数请求号及其响应函数

读者可能会非常诧异，图 11-15 和图 11-16 显示的内容不是完全一样吗？的确如此。这正是下面要讨论的问题：即对于任何一种主机控制器其响应的请求都是相同的！注意图 11-15 和图 11-16 中对同一个请求号虽然使用了相同的响应函数名称，但是底层实现上却是不同，这将与具体的主机控制器硬件相关。
注意：图 11-15，11-16 中所示请求号均定义在 tartget/h/drv/usb/usbHcd.h 内核头文件中。

对于嵌入式平台上非 UHCI，OHCI 或者 EHCI USB 主机控制器，我们可以借用这个实现框架，可以使用相同的响应函数名称，只是在函数的实现上将根据我们实际驱动的主机控制器进行具体的设计。

对于图 11-15，11-16 中给出的具体底层响应函数，本书将不再作介绍，感兴趣读者可自行对这些代码进行分析，其中 UHCI 对应的实现函数均定义在 target/src/drv/usb/usbHcdUhciLib.c 文件中，OHCI 对应的实现函数均定义在 target/src/drv/usb/usbHcdOhciLib.c 文件中。为了自行实现一个主机控制器的驱动设计，建议读者首先研究一下 UHCI 或者 OHCI 控制器驱动的实现代码，而后在已有函数框架下进行修改。

11.9.2 中断处理函数实现

中断主要用于 USB 数据传输的后处理。当 USB 控制器完成一次总线操作后（即传输一个报文），就会发出一个中断，让中断处理函数对控制器缓冲区中数据进行处理，对于读数据（IN）操作而言，中断处理函数必须读取控制器内部 FIFO 中的数据，以便空出空间接收下一个 USB 数据包，而对于写数据（OUT）操作而言，中断处理函数必须将新的数据写入控制器内部 FIFO 中，以便控制器进行下一个写数据操作。如果在数据传输过程中发生错误，同样也会产生中断，此时中断处理函数必须对这些错误进行处理后，再次启动先前的数据传输操作。中断处理函数的实现代码完全硬件相关，不同的主机控制器中断处理函数的处理流程各不相同。但是实现框架基本一致：即中断处理函数只进行关键性操作，具体的中断响应工作放在后台任务中完成，二者之间使用同步信号量进行同步。
例如 UHCI HCD 中断处理函数实现如下。

```
/*******************************************************************************
* target/src/drv/usb/usbHcdUhciLib.c
* intHandler - hardware interrupt handler
*
* This is the actual routine which receives hardware interrupts from the
* UHC.    This routine immediately reflects the interrupt to the intThread.
* interrupt handlers have execution restrictions which are not imposed
* on normal threads...So, this scheme gives the intThread complete
* flexibility to call other services and libraries while processing the
* interrupt condition.
*
* RETURNS: N/A
*/
LOCAL VOID intHandler (pVOID param)
{
    pHCD_HOST pHost = (pHCD_HOST) param;
    UINT16 intBits;


    /* Is there an interrupt pending in the UHCI status reg? */
    if ((intBits = (UHC_WORD_IN (UHCI_USBSTS) & UHC_INT_PENDING_MASK)) != 0)
    {
        pHost->intCount++;

        /* A USB interrupt is pending. Record the interrupting condition */
        pHost->intBits |= intBits;

    /* Clear the bit(s).
     *
     * NOTE: The UHC hardware generates "short packet detect"
     * repeatedly until the QH causing the interrupt is removed from
```

```
         * the work list.    These extra interrupts are benign and occur at
         * at relatively low frequency.    Under normal operation, the
         * interrupt thread will clean up the condition quickly, and the
         * extra interrupts are tolerable.
         */

            UHC_SET_BITS (UHCI_USBSTS, intBits);

            /* Signal the interrupt thread to process the interrupt. */
            OSS_SEM_GIVE (pHost->intPending);
        }

        if(pHost->hcSyncSem)
            semGive (pHost->hcSyncSem);
}
```

intHandler 是 UHCI 驱动中注册的直接硬件中断响应函数，该函数并不完成任何中断相关的工作，其仅仅记录这个中断，而后释放一个信号量，通知其他函数来处理这个中断，此处的这个“其他函数“就是驱动自行创建的后台任务执行函数 intThread。

intThread 函数实现如下。

```
/*******************************************************************************
* target/src/drv/usb/usbHcdUhciLib.c
* intThread - Thread invoked when hardware interrupts are detected
*
* By convention, the <param> to this thread is a pointer to an HCD_HOST.
* This thread waits on the intPending semaphore in the HCD_HOST which is
* signalled by the actual interrupt handler.    This thread will continue
* to process interrupts until intThreadExitRequest is set to TRUE in the
* HCD_HOST structure.
*
* This thread wakes up every UHC_ROOT_SRVC_INTERVAL milliseconds and
* looks for IRPs on the root IRP list.      If any are present, this thread
* will check the root hub change status.    If there is a pending change,
* the IRP will be completed and its usbdCallback routine invoked.    The
* thread also checks for IRPs which have timed-out during this interval.
*
* RETURNS: N/A
*/
LOCAL VOID intThread (pVOID param)
{
    pHCD_HOST pHost = (pHCD_HOST) param;
    UINT32 lastTime;
    UINT32 now;
```

```
UINT32 interval;
UINT8 hubStatus;
UINT16 irpCount;
pUSB_IRP pIrp;

lastTime = OSS_TIME ();

do
{
    /* Wait for an interrupt to be signalled. */

    now = OSS_TIME ();
    interval = UHC_ROOT_SRVC_INTERVAL -
        min (now - lastTime, UHC_ROOT_SRVC_INTERVAL);

    if (OSS_SEM_TAKE (pHost->intPending, interval) == OK)
    {
        /* semaphore was signalled, int pending */

        if (!pHost->intThreadExitRequest)
        {
            OSS_MUTEX_TAKE (pHost->hostMutex, OSS_BLOCK);
            processUhcInterrupt (pHost);
            OSS_MUTEX_RELEASE (pHost->hostMutex);
        }
    }

    if ((now = OSS_TIME ()) - lastTime >= UHC_ROOT_SRVC_INTERVAL)
    {
        /* time to srvc root */
        lastTime = now;

        /* Take the hostMutex before continuing. */
        OSS_MUTEX_TAKE (pHost->hostMutex, OSS_BLOCK);

        /* Service any IRPs which may be addressed to the root hub */
        /* If there is a pending status, then notify all IRPs
          * currently pending on the root status endpoint.
          *
          * NOTE: The IRP callbacks may re-queue new requests.
          * Those will be added to the end of the list and will
          * not be removed in the following loop.
          */
        if ((hubStatus = isHubStatus (pHost)) != 0)
```

```
                {
                    for (irpCount = pHost->rootIrpCount; irpCount > 0; irpCount--)
                    {
                        pIrp = usbListFirst (&pHost->rootIrps);

                        --pHost->rootIrpCount;
                        usbListUnlink (&pIrp->hcdLink);
                        *(pIrp->bfrList [0].pBfr) = hubStatus;
                        pIrp->bfrList [0].actLen = 1;

                        setIrpResult (pIrp, OK);
                    }
                }

                /* Look for IRPs which may have timed-out. */
                checkIrpTimeout (pHost);

                /* Release the hostMutex */
                OSS_MUTEX_RELEASE (pHost->hostMutex);
            }
        }
        while (!pHost->intThreadExitRequest);

        /* Signal that we've acknowledged the exit request */
        OSS_SEM_GIVE (pHost->intThreadExit);
}
```

注意 intThread 函数平时阻塞于 pHost->intPending 信号量，当一个中断发生时，intHandler 释放该信号量，此时 intThread 方才得以执行，此时 intThread 将调用 processUhcInterrupt 函数完成中断的具体处理，并对相关状态进行更新。在任务上下文中进行中断的处理，一方面不会对整个系统的响应速度造成负面影响，另一方面可以跳出中断上下文的限制，可以调用任何函数，无需对执行时间长度做刻意留意。
对于我们自实现的 HC 驱动，可以借鉴此种实现方式，最好也是使用此种中断处理函数+后台任务的方式。对于 HC 驱动而言，内核不提供后台任务（不像网络设备驱动，内核提供 tNetTask 后台进程），故 HC 驱动必须自行完成后台任务的创建工作。这些工作都将在 fncAttach 函数中完成（包括中断注册，资源分配，后台任务创建等等），fncAttach 可以看作是 HCD 底层初始化函数。

11.9.3 其他函数实现

这里指的的其他函数即两个中心函数（总入口函数和中断函数）中调用的函数，这些函数将完成底层具体某个功能的实现，这些实现与主机控制器硬件直接相关。例如对于 UHCI HC 而言，为了完成数据传输，fncIrpSubmit 函数必须在内存中组织 QH，TD 等数据结构（参见

上文对 UHCI 工作原理的介绍)并在驱动中维护对这些结构的引用,这些结构将自动被 UHCI HC 使用启动一个传输过程，在成功完成或者发生错误时，将产生一个中断，此时中断处理程序必须检查中断源，如果是一个传输中断，则检查 TD 结构中状态字段，判断此次传输是否成功，如果成功，则调用相关结构中函数指针指向的回调函数，通知上层（USB 栈）进行其需要的后处理。
每种请求都对应有一个特定的数据结构来表示请求参数和信息，HC 驱动开发者必须对每种请求对应的数据结构研究清楚，特别对一些函数指针类型的字段，这些字段有些是由内核 USB 栈初始化的，需要在一个操作完成后被回调的函数，这些回调函数的执行非常重要，因为这是将执行结果传递给上层的唯一手段（如将读取的数据传递给上层)。

11.9.4 相关说明

在本节的讨论中，我们并没有对 UHCI 控制器驱动代码做详细的分析，一方面是由于工作量太大，另一方面是某些细节很难讲清楚，故需要对这一方面由需求的读者自行结合 UHCI 规范对底层驱动实现代码进行分析。对于这类较复杂的驱动，读者必须首先对整体框架进行掌握，这即是本章的意图，在理解了整体框架的基础上，再一一对具体功能进行实现，可以尽快进入和掌握 USB 主机控制器驱动的设计和实现。

在本章至此的讨论中，我们都是对主机端控制器驱动进行介绍，实际上 USB 从设备端也存在一个栈层次，Vxworks 也对此提供了支持，由于从设备端驱动设计的概率相对较小，且基本分析方式同主机端,故本章对这部分内容将不再涉及。对于 USB 从设备端栈实现,Vxworks 开发环境 Tornado 也以源代码的方式提供给了有需求的开发者，感兴趣读者可以可自行进行分析。

11.10 本章小结

本章主要对 USB 设备驱动进行了介绍。首先我们对 USB 本身进行了较为详细的讨论，指出了 USB 只是数据传输一种手段，其本身并不对数据内容进行解释，数据的解释由位于 USB 栈之上的 USB 类驱动进行。在底层，我们可以将 USB 主从两端看做是两组 FIFO，USB 主机和目标机控制器负责操作接口向 FIFO 中写入数据或者读取数据，而底层驱动则同样的只对 FIFO 进行数据操作，至于数据如何通过 USB 总线传递到另一端则完全由控制器本身负责。此后我们着重介绍了 UHCI 控制器的基本工作原理，并以类驱动层一个读数据请求为起点，详细跟随了这个请求在内核 USB 栈的传递过程，直至到达底层 HCD 总入口函数，介绍了在这过程中调用的关键函数以及相关结构，读者需要注意一点的是对于不同的请求，虽然调用路径基本相同，但是所使用的数据结构则完全不同，不同的请求将具有特定的数据结构存储该请求特定的参数和信息。接着我们对底层 HCD 驱动结构进行了总结，指出了驱动两个中心函数，HCD 驱动基本围绕这两个中心函数进行展开，最后我们简单介绍了两个中心函数的实现框架。在本章中，我们并无对 UHCI HCD 底层功能实现代码进行分析，原因

已在前文中进行了交代，对于有需要自行 USB 主机控制器驱动需求的读者，请结合本章给出的驱动框架自行对 Vxworks 下 USB 栈以及 UHCI 或者 OHCI 驱动源代码进行分析，着重对相关结构和硬件工作原理了解清楚，应不难应对实际工作中 USB HC 驱动的设计和实现。

Vxworks 对 USB 组件都定义有相关的控制宏，要包括对应的功能，必须首先对这些宏进行定义，这些内容连同 USB 从设备栈层次，读者可以参考文献[15]，[16]，对于本章使用示例中 Mass Storage USB 设备相关规范，读者可以参考文献[18]，[19]，[20]。

参考文献

[1] Wind River, vxworks kernel programmer’s guide， Sep20 03.
[2] Wind River, vxworks device driver developer’s guide, Sep 2003.
[3] Wind River, vxworks bsp developer’s guide, May 2003.
[4] Wind River, vxworks programmers’ guide, Mar 2003.
[5] Wind River, vxworks architecture supllement, Sep 2003.
[6] Wind River, network protocol toolkit user’s guide, Jul 1999.
[7] Wind River, Tornado BSP Training Workshop, Apr 1998.
[8] Wind River, usb vxworks api reference, Nov 2007.
[9] Wind River, Vxworks Reference Manual, May 1999.
[10] Wind River, Vxworks Application API Reference, Oct 2006.
[11] Wind River, Vxworks Drivers API Reference, Oct 2006.
[12] Wind River, Vxworks Kernel API Reference, Oct 2006.
[13] Wind River, Vxworks for arm architecture supplement, Apr 2003.
[14] Wind River, Vxworks for powerpc architecture supplement, Sep 2003.
[15] Wind River, USB Developer’s Kit Programmer’s Guide, Apr 2003.
[16] Wind River, USB Programmer’s Guide, May 2006.
[17] Texas Instruments, TMS320DM644x DMSoC EMACMDIO Module User’s Guide, 2005.
[18] Universal Serial Bus Mass Storage Class Bulk-Only Transport, Sep 1999.
[19] Universal Serial Bus Mass Storage Class Control/Bulk/Interrupt (CBI) Transport, Dec 1998.
[20] Universal Serial Bus Mass Storage Class UFI Command Specification, Dec 1998.
[21] Understanding the bootrom image, AppNote-237, Jan 2003.
[22] Intel, Universal Host Controller Interface(UHCI) Design Guide, Mar 1996.
[23] Compaq etc, Open Host Controller Interface Specification for USB, Sep 1999.
[24] Compaq etc, Universal Serial Bus Specification 2.0, Apr 2000.
[25] Micron, An Introduction to NAND Flash, TN-29-19, 2006.
[26] Micron, Small-Block vs. Large-Block NAND Flash Devices, TN-29-07, 2005.
[27] AMD, Common Flash Memory Interface Specification, Dec 2001.
[28] Eran Gal, Sivan Toledo, Algorithms and Data Structures for Flash Memories, Aug 2005.
[29] SumSung, S3C2440A 32-bit coms Microcontroller User’s Manual, 2004.