

序 言

学习 VxWorks 的几年来，大部分的精力都花在 BSP 的配置和驱动程序的编写上，期间遇到过很多问题，也得到过一些经验。在不断深入的学习过程中，总是希望能有一本中文的资料可以参考，这个想法最终促成了这本书的诞生。

这是一本合作的书籍，周启平在其中花费了大量的心血。大部分的联络工作和大部分章节的编写都是由他完成的。由于工作中，我更多地涉及驱动程序的编写，于是参与了第 6 章、第 7 章、第 8 章和第 10 章的编写以及全书各章节的修改、校对工作。其余工作，包括与编辑的联系、其他章节的编写、修改等，都是由周启平完成的。

同时要感谢的是关心和支持我们的人：我们的家人、我们的直系领导、同事们和出版社的编辑们，感谢他们为这本书所付出的艰辛劳动。

本书注重实际的应用和经验，同时也非常关注基本的原理。全书共分为两个部分：第一部分是驱动程序的编写。驱动程序的编写涉及到很多的内容，从设备的访问方法到设备工作的方式等分别在各章节有所讨论。在总体概论之后，分章节讨论了各种类型设备驱动程序的编写方法并给出了相应的模板。最后为了让读者有更深入的了解，在第 10 章简单介绍了 PCI 设备、ISA 设备和文件系统的编写，这也是编写设备驱动程序中典型的应用。

第二部分以 x86 架构为例，基本上对 BSP 的各个部分都有所描述：描述了预内核的初始化过程、BSP 工作流程、BSP 相关组件、BSP 开发等。最后描述了不同体系结构上需要注意的问题。

在编写过程中，我们参考了一些公司的讲义、Wind River 公司的培训资料以及新上市的部分书籍。我们尽量使文章有自己的特色，但也难免会重复一些他人的东西。在修改过程中，我们尽量使语言更平易近人、更本地化，但水平有限，可能会弄巧成拙。总之，希望读者可以对此多包涵，也希望读者能给我们提出反馈意见，使我们的工作可以得到改进。

由于知识有限、时间仓促，不免会在编写的过程中出现这样或者那样的错误，还要请读者不吝赐教。下面是我们的电子邮箱：

张 杨：zhangyang@first.com.cn

周启平：qpzhou@sina.com

编 者

目 录

序 言

第 1 章 外部设备及设备驱动程序概述	1
1.1 外部设备	1
1.2 外部设备的分类	2
1.3 I/O 设备的数据传送方式	2
1.4 设备驱动程序	3
1.5 设备驱动程序的主要功能	4
1.6 设备驱动程序的组成部分	5
1.7 设备驱动程序的相关概念	5
小结	7
第 2 章 VxWorks 下设备及设备驱动程序	8
2.1 VxWorks 下的设备	8
2.2 VxWorks 下设备驱动程序	11
2.3 VxWorks 下常用设备驱动程序简要描述	13
小结	20
第 3 章 VxWorks 下设备驱动程序的分析	21
3.1 VxWorks 下设备驱动程序在系统中的层次	21
3.2 VxWorks 的设备驱动程序表	22
3.3 VxWorks 下常用设备的驱动程序源文件	22
3.4 VxWorks 下设备驱动程序的配置	23
3.5 VxWorks 下设备驱动程序常用函数	28
小结	30
第 4 章 驱动程序的轮询和中断处理	31
4.1 概述	31
4.2 轮询	31
4.3 中断处理	36
4.4 其他设计需要考虑的事项	43
小结	49

第 5 章 VxWorks 下编写设备驱动程序的方法	50
5.1 概述	50
5.2 开发前资料的收集	50
5.3 宏定义及 C 语言可以调用的汇编函数	51
5.4 存储映射 I/O 与端口 I/O	55
5.5 设备驱动程序错误处理	57
5.6 VxWorks 下设备驱动程序编程规范	60
5.7 高速缓存的一致性问题的	61
小结	62
第 6 章 编写字符设备驱动程序	63
6.1 I/O 系统回顾	63
6.2 一步步编写字符型设备的驱动程序	70
6.3 字符设备驱动程序的完整模板	80
小结	85
第 7 章 编写串行设备驱动程序	86
7.1 串行设备概述	86
7.2 虚拟设备 ttyDrv	87
7.3 编写串行设备驱动程序	90
7.4 串行设备的安装	97
7.5 串行设备驱动程序完整的模板	98
小结	103
第 8 章 编写块设备驱动程序	104
8.1 VxWorks 下的文件系统	104
8.2 一步步编写块设备驱动程序	107
8.3 完整的块设备驱动程序流程	115
小结	120
第 9 章 编写网络设备驱动程序	121
9.1 概述	121
9.2 END 设备驱动程序装载过程	123
9.3 网络设备与系统的数据交换	127
9.4 一步步编写网络设备驱动程序	134
小结	159
第 10 章 接触实际设备	160
10.1 PCI 设备驱动程序	160

10.2	ISA 设备驱动程序.....	169
10.3	原始文件系统分析.....	171
	小结.....	173
第 11 章	板级支持包.....	174
11.1	板级支持包.....	174
11.2	BSP 的职责.....	175
11.3	BSP 的组成.....	177
11.4	BSP 的开发.....	187
11.5	Tornado 的目录结构.....	189
11.6	BSP 的约定和有效性.....	191
	小结.....	192
第 12 章	VxWorks 映像及启动顺序.....	193
12.1	VxWorks 映像类型.....	193
12.2	VxWorks 启动顺序简述.....	194
12.3	VxWorks 映像启动顺序.....	194
12.4	构造 VxWorks 映像.....	196
	小结.....	203
第 13 章	VxWorks 预内核初始化.....	204
13.1	预内核的初始化概述.....	204
13.2	预内核的特殊初始化函数.....	206
13.3	预内核的通用初始化代码.....	225
13.4	预内核初始化的调试.....	233
	小结.....	241
第 14 章	BSP 的配置.....	242
14.1	BSP 的基本配置.....	242
14.2	VxWorks 的基本配置.....	248
14.3	双硬盘配置.....	253
14.4	双网卡配置.....	254
	小结.....	257
第 15 章	板级支持包开发.....	258
15.1	板级支持包开发周期概述.....	258
15.2	板级支持包的开发环境.....	260
	小结.....	263
第 16 章	创建一个新的 BSP.....	264

16.1	概述.....	264
16.2	建立开发环境.....	265
16.3	编写内核启动前 BSP 的初始化代码	266
16.4	使用最小的内核	274
16.5	目标机代理和 Tornado.....	275
16.6	最后的修整工作	276
	小结.....	279
第 17 章	体系结构	280
17.1	Power PC.....	280
17.2	Intel x86	286
17.3	MIPS.....	290
17.4	ARM.....	294
17.5	Motorola 68K	298
	小结.....	302

第 1 章 外部设备及设备驱动程序概述

在嵌入式系统开发过程中，对于编写设备驱动程序的人员来说，每写一个设备驱动程序就代表一个挑战。因为不仅需要开发人员具备软件编程能力和编写设备驱动程序的基本知识，同时还需要了解硬件设备的工作原理以及熟悉整个系统的体系结构。快速而有效地开发设备驱动程序，经验是一方面，背景知识和相关概念也是必不可少的。但是所有设备驱动程序的基本原理和功能都有相通的一面，有其规律可循。

刚接触编写设备驱动程序时，不要急于编写程序，而应该先熟悉硬件设备、了解它的工作原理及功能。然后从设备共性上来看待设备结构和功能，即所谓透过设备的表面现象看“本质”，以数据流为中心，了解设备与 CPU 及外界通信的方式。

本章内容不涉及到具体某类设备和编程，主要讨论一些概念、方法、技巧等知识，希望为读者实际编写设备驱动程序打下一定基础。

1.1 外部设备

在广泛使用的微型计算机系统中，外部设备就是以实现人机交互和机间通信为目的的一些机电设备。在对外部设备的控制过程中，系统不可避免地，有时候甚至要频繁地对设备接口进行控制和访问，因此，在编写设备驱动前对硬件设备本身的认识是非常必要的。

输入/输出设备都要通过一个硬件接口或控制器与 CPU 相连，例如终端显示器通过显卡与 CPU 连接起来；网络设备通过网卡与 CPU 连接起来；软盘驱动器通过软盘控制器与 CPU 连接起来。CPU 与接口板及设备之间的关系如图 1.1 所示。

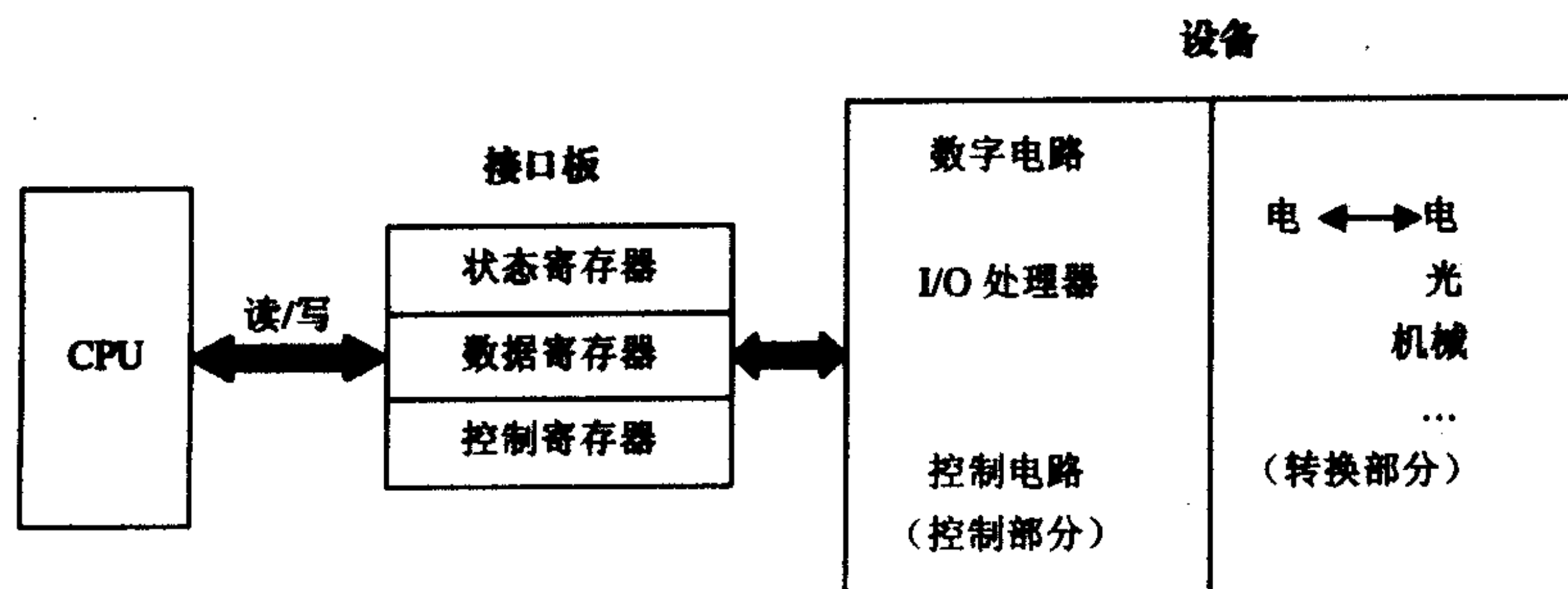


图 1.1 CPU 与接口板及设备之间的关系

通常接触到的外部设备有显示器、键盘、鼠标、硬盘等。设备具有以下共性：

- (1) 是一个硬件；

- (2) 具有输入/输出功能;
- (3) 通常是核心处理器的扩展部分;
- (4) 与 CPU 进行数据交换通常是靠设备接口板完成;
- (5) 系统为每一个连接到 I/O 总线上的设备分配各自的 I/O 地址。

1.2 外部设备的分类

在微型计算机系统中外部设备通常被划分成三种类型：字符设备、块设备和网络设备。下面分别讨论三种类型的设备。

1. 字符设备 (character device)

字符设备是能够像字节流一样被访问的设备。在数据传输中以字符为单位进行传输，由字符设备驱动程序实现这种特性。字符设备驱动程序通常要实现打开、关闭、读和写系统调用。像字符终端、串口、鼠标、打印机等就是字符设备的代表。字符设备可以通过文件方式（如 `/tyCo/0`）来访问，它和普通文件之间的惟一差别在于，对普通文件的访问可以前后移动访问指针，而多数字符设备只能顺序访问数据通道。

字符设备通常具有如下的特点：

(1) 在数据传输中可以传输任意大小的字符数据。像串口既可以一次传输一个字节，也可以通过数据流的方式来控制传输数据的大小。

(2) 通常传输一组字符数据。

2. 块设备 (block device)

块设备是指以“块”为单位对数据进行存取的设备。和字符设备一样，系统通过设备表访问块设备，但块设备上能够容纳文件系统。块设备包含整数个“块”，而每块包含 1KB 或 2 的多次幂字节的数据，它与字符设备的区别主要是内部数据管理的方式不同。块设备除了给操作系统提供接口以外，还要提供专门面向块设备的接口。如硬盘、光驱、软驱、磁带等就是块设备的代表。

块设备通常具有如下特点：

(1) 在数据传输中传输固定大小的数据块。

(2) 可以随机访问块设备中所存放的数据块。

3. 网络设备 (network device)

任何网络通信都要经过网络设备，即能够和其他主机进行数据交换的设备。网络设备由网络子系统驱动程序负责发送和接收数据包，它与普通 I/O 设备不同，没有对应的设备文件。所以应用程序和网络接口之间的数据通信不是基于标准的 I/O 系统接口，而是基于 `socket()`、`bind()`、`listen()`、`accept()`、`send()`、`connect()` 等系统调用。

1.3 I/O 设备的数据传送方式

控制 I/O 设备与 CPU 相连的接口和控制器都能支持 I/O 指令与外部设备交换信息。这些信息包括控制、状态和数据三种不同性质的信息，必须为它们分配不同的端口地址分别进行传送。

控制信息输出到 I/O 接口，告诉接口和设备要进行什么处理；从接口获得的状态信息表示 I/O 设备当前的状态；数据信息则是 I/O 设备和 CPU 真正要交换的信息。不同的 I/O 设备要求传送的数据类型是不同的，例如与字符型终端交换的数据必须是 ASCII 码，而不是二进制表示的数。

输入/输出方式通常包括三种方式。

- (1) 直接控制 I/O 方式：通过指令直接对端口进行输入输出操作称之为直接控制 I/O 方式。
- (2) 中断方式：采用中断实现数据的输入/输出。
- (3) DMA 方式：采用 DMA 控制器实现数据传输。

在外部设备输入/输出方式选择中，通常依据硬件本身而采用不同的方式。例如字符设备通常采用中断方式；而对于一些传输大量数据的 I/O 设备（如磁盘、网卡等设备），通常采用 DMA 方式，以减少 CPU 资源占用。程序直接控制输入/输出方式和中断传送方式在后面的第 4 章中将会专门进行描述，而 DMA 方式主要是由硬件 DMA 控制器来实现数据的传送功能，所以仅在这一小节里简单介绍一下。

DMA 方式即直接存储器访问（Direct Memory Access）方式，也称为成组数据传送方式。DMA 方式主要适合于一些传输大量数据的 I/O 设备，例如磁盘、数模转换器等设备。由于这些设备本身传输字节或字的速度非常快，通过执行输入/输出指令的方法或中断方法来传输数据，将会造成这些数据的丢失。而 DMA 方式能使 I/O 设备直接和存储器进行成批数据的快速传输。数据一旦到达端口，就直接由 DMA 控制器送到存储器；同样，DMA 控制器也能直接从存储器取出数据并把它们送到 I/O 设备中去。

DMA 控制器或接口一般包括四个寄存器：状态控制寄存器、数据寄存器、地址寄存器和字节计数器。这些寄存器在数据传送前要进行初始化操作；每个字节传送后，地址寄存器加 1，字节计数器减 1。

系统完成 DMA 传送的步骤如下所示：

- (1) DMA 控制器向 CPU 发出 HOLD 信号请求使用总线；
- (2) CPU 发出响应信号 HLDA 给 DMA 控制器，并将总线让出，之后 CPU 不再使用总线，而 DMA 控制器获得总线控制权；
- (3) 传输数据的存储器地址（在地址寄存器中）由 DMA 控制器通过地址总线发出；
- (4) 传输的数据字节通过数据总线进行传送；
- (5) 地址寄存器加 1；
- (6) 字节计数器减 1；
- (7) 如字节计数器非 0，转向步骤（3）；
- (8) 否则，DMA 控制器撤消总线请求信号 HOLD，数据传送结束。

1.4 设备驱动程序

设备驱动程序是直接控制设备操作的那部分程序，也是设备上层的一个软件接口。设备驱动程序的功能是对 I/O 进行操作，实际上从软件角度来说就是对 I/O 端口地址进行读写操作。只要系统访问设备就会调用驱动。从这一点可以看出，驱动程序不能自动执行，只能被系统或应用程序调用。

在嵌入式系统中调用设备驱动通常有三种方式：应用程序直接调用、应用程序通过操作系统内核调用、应用程序通过操作系统的扩展模块进行调用。如图 1.2 所示为应用程序调用设备驱动。

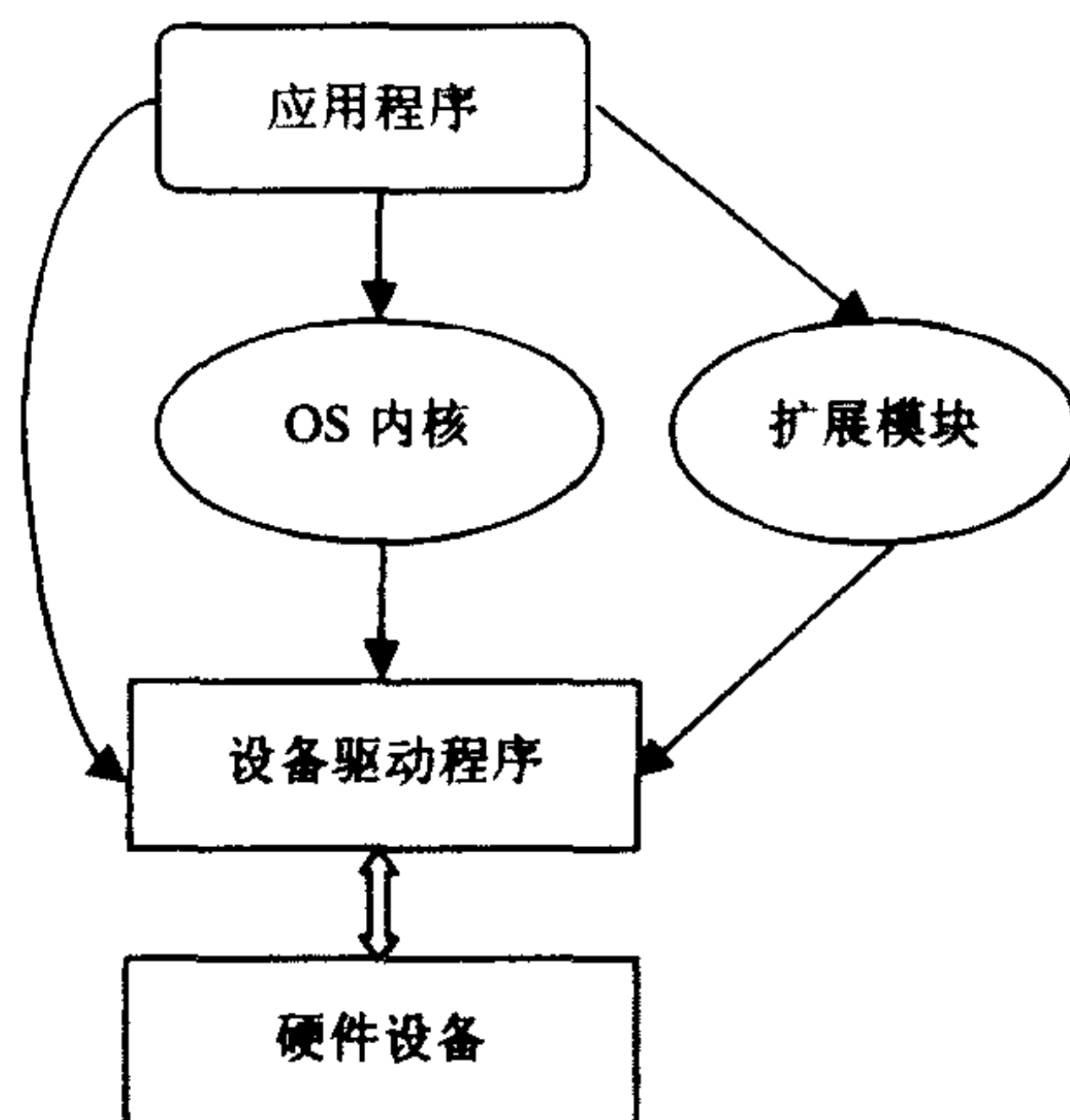


图 1.2 应用程序调用设备驱动

不同的调用方式各有自己的优缺点，下面分别描述它们的优缺点。

(1) 应用程序直接调用设备驱动程序。

优点：可以高效访问设备（因为绕过操作系统，而直接访问设备，减少了系统开销时间），减少了系统代码开销，适合简单的嵌入式系统。

缺点：所编写的应用程序移植性差（因为应用程序直接与设备驱动关联），且用户自己来管理设备，不适合复杂的嵌入式系统。

(2) 应用程序通过操作系统内核调用设备驱动程序。

优点：所编写的应用程序移植性好（统一的标准接口函数），且设备由操作系统管理，适合复杂的嵌入式系统。

缺点：操作系统管理设备，增加了系统开销。

(3) 应用程序通过操作系统的扩展模块调用设备驱动程序。

扩展模块是嵌入式操作系统可供用户选择的模块。当用户需要某个扩展模块时才把该模块加入到系统中，如网络模块、文件模块等。它的优缺点类似第二种调用方式，但这些扩展模块依赖于操作系统内核。

至于采用哪种方式，视用户系统的复杂性及系统本身要求而定。在嵌入式应用开发中，用户通常采用第二种和第三种方式。

1.5 设备驱动程序的主要功能

设备驱动程序通常包含六个主要功能。

(1) 对设备进行初始化。初始化的目的是使设备处于某种工作状态，以使用户程序访问该设备。譬如串口初始化包括设置串口波特率、数据位、奇偶校验位、停止位等。

(2) 打开设备操作。打开设备操作实际上是查询用户指定的设备, 并查看用户是否可以使用该设备。因为设备是共享资源, 当设备正在被使用时, 系统要对它进行保护, 禁止其他任务对设备进行操作, 直到设备资源被释放。

(3) 关闭设备操作。关闭设备操作就是释放设备资源。任务对设备完成操作后, 必须进行关闭设备操作, 否则设备总是处于被占用状态, 其他任务无法使用。与打开设备操作相对应, 有打开操作就应该有关闭操作。

(4) 从设备上接收数据并提交给系统。这项功能通常就是所说的读操作, 接收外部传输来的数据。接收数据采用的方式有查询方式、中断方式和 DMA 方式。

(5) 把数据从主机上发送给设备。这项功能对应通常的写操作, 把主机上的数据传送给外界。通常系统主动调用该操作进行数据发送, 有时也采取中断方式发送数据。

(6) 对设备进行控制操作。在使用设备过程中, 有时根据应用的需要对设备进行控制 (例如改变设备某个状态), 而控制操作就能提供这种功能。例如, 在使用串口过程中, 有时需要改变串口波特率的设置, 这就需要用到这种控制操作。

1.6 设备驱动程序的组成部分

根据设备驱动的主要功能, 设备驱动程序大致由下面几部分组成。

(1) 设备驱动程序的注册函数。驱动程序开发人员需要编写一个函数把相关的驱动程序 (打开、关闭、读/写等) 注册到系统设备驱动程序表中, 以便系统对其管理及使用。

(2) 设备驱动程序的卸载函数。当系统不需要使用某设备时, 为了节省资源开销, 需要从驱动程序表中卸载该设备驱动程序并释放设备占用的资源。这些操作由设备驱动程序的卸载函数完成。

(3) 设备的打开和关闭函数。这两个函数的功能在设备驱动程序主要功能部分已经作过描述, 需要注意的是有打开设备操作, 必须相应地有关闭设备操作。

(4) 设备的读/写操作函数。设备与外界的通信主要由这两个函数完成, 主要功能是完成设备与 CPU 之间的数据传输。

(5) 设备控制函数。在对设备的操作中, 用户有时根据需要对设备进行控制。如寄存器设置、设备相关操作等。

(6) 中断服务函数。该函数通常在对设备进行读/写操作时使用, 当设备上接收到数据或数据发送结束时, 通过触发硬件中断信号, 向系统报告这一状态, 系统便执行中断服务函数进行相应的处理。

1.7 设备驱动程序的相关概念

1.7.1 设备驱动程序表

应用程序对设备进行操作是通过操作系统的 I/O 管理子系统来进行的。而 I/O 管理子系统对设备的管理在系统中是通过一张表来操作, 通常把这张表称为“设备驱动程序表”。设备驱动程序表实际上是一个数组, 它的每一个数组项是一个设备驱动程序地址表结构体, 这个结构体包含初始化、打开、关闭、读、写、控制等操作函数的入口地址。如果哪一个操作函数空缺, 那么它的入口地址

为 NULL。

下面是一个设备驱动程序表的示例。

```
/* 设备驱动程序结构 */
typedef struct {
    xx_device_driver_entry initialization; /* 初始化程序 */
    xx_device_driver_entry open;         /* 打开设备处理程序 */
    xx_device_driver_entry close;        /* 关闭设备处理程序 */
    xx_device_driver_entry read;         /* 读处理程序 */
    xx_device_driver_entry write;        /* 写处理程序 */
    xx_device_driver_entry control;      /* 特殊功能处理程序 */
} xx_driver_address_table;

/* 系统设备驱动表 */
xx_driver_address_table Device_drivers[5] =
{
    /* 控制台驱动 */
    { console_initialize, console_open, console_close,
      console_read, console_write, console_control },
    /* 时钟驱动 */
    { Clock_initialize, NULL, NULL, NULL, NULL, Clock_control },
    ...
};
```

1.7.2 设备名称

操作系统通常提供了给设备命名的功能，系统可以调用相关的系统服务给设备注册一个名字。应用程序可以根据这个名字获得对应设备的主从设备号以及对设备进行打开、关闭、读、写等操作。不过系统分配给设备的设备号是一个整数，这对用户而言非常不直观明了。而设备名克服了这方面的缺点，一看名称一般就知道使用的是什么设备，否则用户在应用程序中使用设备时会带来诸多不便。

下面是一个注册设备名的函数示例。

```
status xx_io_register_name(char *name, int major, int minor);
```

1.7.3 设备驱动接口

当应用程序调用一个操作系统的 I/O 管理子系统所提供的服务时，操作系统将调用对应的设备驱动程序，应用程序传输给 I/O 管理子系统的信息将被传输给对应的设备驱动程序。

通常，操作系统 I/O 管理子系统调用的设备驱动程序具有如下的类似原型：

```
xx_device_driver io_xx(
    xx_device_major_number    major,      /* 主设备号 */
    xx_device_minor_number    minor,      /* 从设备号 */
    void                      *argument   /* 传递给设备驱动程序的参数 */
);
```

值得注意的是“argument”参数通常传送给设备驱动程序，是设备驱动程序的输入参数项。因此，该参数的格式和内容与设备驱动程序有关。

1.7.4 设备驱动程序的运行环境

应用程序开发者和驱动程序开发者必须注意以下与操作系统 I/O 管理子系统相关的几点内容。

(1) 设备驱动程序在调用它的任务上下文中执行，因此如果驱动程序阻塞了，那么调用它的任务也就阻塞了。

(2) 尽管驱动程序会把调用它们的任务恢复到调用前的原始状态，但设备驱动程序还是可以自由地改变调用它的任务的模式。

(3) 设备驱动程序可能被中断服务程序调用。

(4) I/O 管理通常只能调用本地的设备驱动程序。

(5) 设备驱动程序可以调用所有的对本地或全局对象操作的系统服务，包括 I/O 管理系统的服务。

(6) I/O 管理只提供了一个设备驱动程序的框架，而不关心设备驱动的结构和具体的操作。

小 结

这一章主要从总体上对外部设备和设备驱动程序进行了描述，其中包括设备的分类、设备的数据传送方式等。在设备驱动程序方面，则主要描述了驱动程序的主要功能、驱动程序的组成以及相关的概念（例如设备驱动程序表、设备名称等）。这些描述不是针对某种操作系统或某种设备，而是从宏观上来说明设备及设备驱动程序，以便读者能够对其有所了解。

第 2 章 VxWorks 下设备及设备驱动程序

前一章的内容,从宏观的角度讨论了硬件设备结构和设备驱动程序的一些知识,这为在 VxWorks 操作系统下编写设备驱动程序奠定了基础。VxWorks 操作系统下的设备驱动程序同样遵循了基本设备驱动程序的原理、功能以及组成部分,同时它也具有自己的特色和特点。下面将一一讨论。

2.1 VxWorks 下的设备

2.1.1 VxWorks 下设备的分类

在 VxWorks 下,设备分为如下几类:

- 字符设备,大多数的传输设备以及显示终端;
- 随机存储块设备,这种设备主要是指磁盘;
- 虚拟设备,比如管道、套接字都属于这种设备;
- 控制监视设备,一般是指用于控制数/模变换的 I/O 设备;
- 网络设备,那些与网络协议挂接的通信设备。

这些设备通常包括网络设备、磁盘设备、串并口、PCI 桥、VME 桥等,如图 2.1 所示。

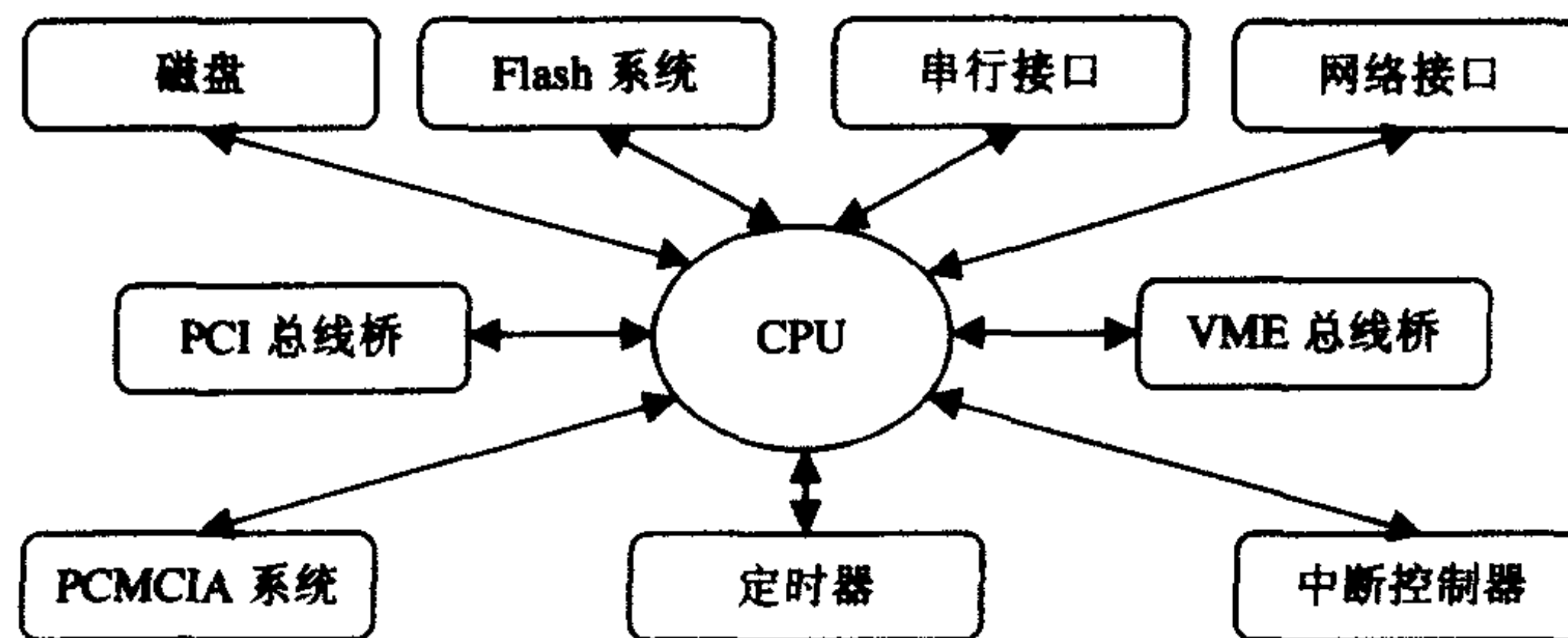


图 2.1 VxWorks 下的设备

根据操作方式的不同, VxWorks 操作系统中 I/O 又分为基本 I/O 系统和缓冲 I/O 系统, 操作系统为两种不同的 I/O 操作方式分别提供了不同的 C 语言函数库。基本 I/O 系统使用了与 UNIX 标准相兼容的 C 语言函数库, 而缓冲 I/O 系统则使用了与 ANSI-C 标准相兼容的 C 语言函数库。为了获得更快的速度和更好的兼容性, VxWorks 系统中的 I/O 都是经过优化设计的, 而速度和兼容性对于 VxWorks 这样的实时系统来说都是至关重要的。

还有一些特殊类型的设备(如网络设备), 由于其自身的特性, 虽然不是通过标准 I/O 来进行存取, 但是也都有它们各自相关的规范。

2.1.2 VxWorks 下设备的命名

在 VxWorks 系统中，设备是被当作文件来管理，而这种机制实际上就是仿照 UNIX 操作系统的设备管理机制。打开设备操作，是通过打开指定的文件来操作 I/O 设备。这个指定的文件可以表示：一个非结构化的原始设备或者是具有文件系统的结构化的随机存储设备上的一个逻辑文件。在对设备命名时通常有一些约定的标准格式，大部分的设备名会以 “/” 开头，并使用 “/” 将设备名和文件名分开表示；但是不基于 NFS 的网络设备会采用远程主机名加冒号的形式，例如 host:download；而使用 dosFs 格式的文件系统设备名经常以大写字母、数字的组合形式加一个冒号构成，例如 DEV1:。

在表 2-1 中列举了 VxWorks 系统中可能出现的设备名。

表 2-1 VxWorks 常用设备说明

设 备 名	设 备
/tyCo/0	串口设备
/fd0	软盘驱动器
/ide0	IDE 接口设备
/ata0	ATA 接口设备
/pipe/0	管道设备
/sd0	SCSI 设备
/tffs0	Trueffs 闪存设备
/pcmcia0	PC 存储卡块设备



注意

不能以单独的 “/” 作为设备名，否则将会引起系统的混乱。

2.1.3 VxWorks 下设备相关概念

1. 设备数据结构（DEV_HDR）及设备描述符

大部分的设备驱动程序都可以为同类型的设备的多个实例提供服务，例如系统中串口设备的驱动程序可以为不同的通道服务。

在 VxWorks I/O 系统中，设备驱动程序需要定义一个被称为设备头（DEV_HDR）的数据结构。在这个结构中包括了设备名称和为设备所服务的驱动程序索引号。I/O 系统中的所有设备的设备头都保存在一个叫做设备列表的链表结构中。实际中，每一个设备都会有更多的数据存储更大的数据结构中，设备头就是这个数据结构的起始部分。而这个更大的数据结构则称为设备描述符，如图 2.2 所示。它包括了很多和设备相关联的数据。

设备头数据结构——DEV_HDR 原型如下：

```
typedef struct          /* DEV_HDR - 所有设备结构的设备头 */
```



```

{
    DL_NODE    node;          /* 设备连接的列表节点 */
    short drvNum;             /* 设备的驱动程序索引号 */
    char *     name;          /* 设备名称 */
} DEV_HDR;

```

设备描述符示例如下：

```

typedef struct    xxDEV
{
    DEV_HDR devHdr;          /* 设备结构 */
    UINT32  RegBase;         /* 本地寄存器基地址 */
    UINT32  UserBase;        /* 板卡内存基地址 */
    BOOL isCreate;           /* 通道创建标志 */
    BOOL isOpen;             /* 通道打开标志 */
    BOOL isWriteNotSend;     /* 只写缓冲标志 */
    BOOL    myDrvDataAvailable; /* 设备可读标志 */
    BOOL    myDrvRdyForWriting; /* 设备可写标志 */
} xxDEV;

```

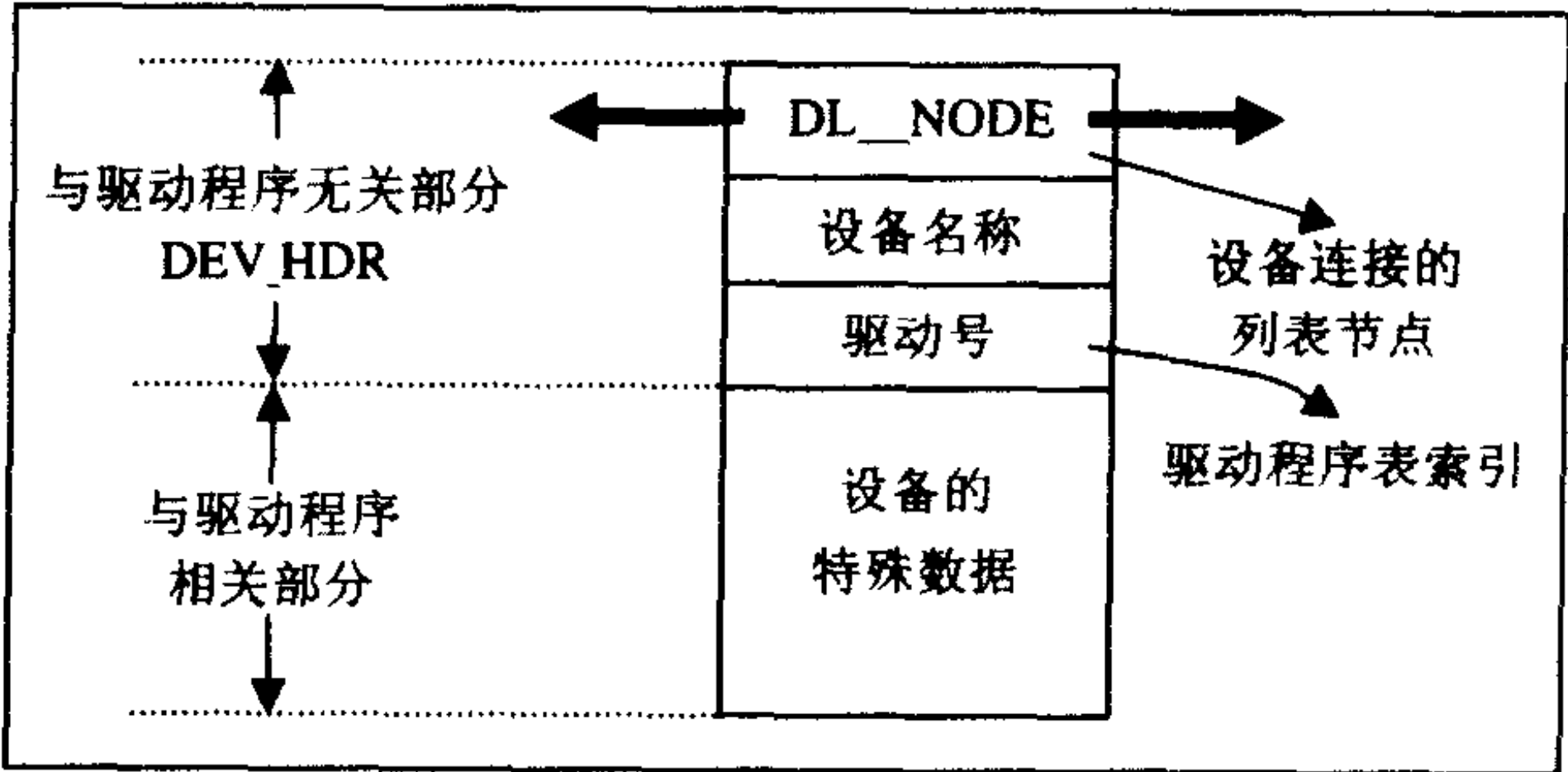


图 2.2 设备描述符结构

2. 设备列表

设备列表如图 2.3 所示，是一个链表，它的作用是将设备的有关信息组织起来。前面提到的设备描述符是一个存储了设备相关信息的数据结构，而且所有设备描述符都是以设备头数据结构开始的，这样可以将这设备描述符组成一个链表形成设备列表。

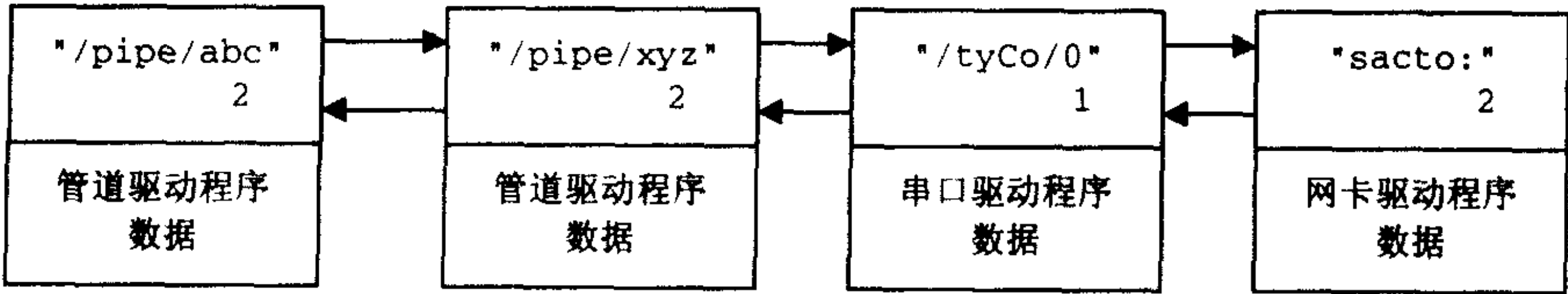


图 2.3 设备列表

用户可以通过调用 devs()函数查看系统中的设备列表。例如在 shell 工具下执行 devs 操作可以查看目标机上所配置的设备，如图 2.4 所示。

2.2 VxWorks 下设备驱动程序

2.2.1 设备驱动程序概述

在 VxWorks 系统中输入/输出设备从宏观上被分为 3 种类型：字符设备、块设备和网络设备。这种分类的方法是根据设备硬件本身特性来决定的。依据设备类型，VxWorks 下设备驱动程序的管理也被划分成三种模块：字符设备驱动程序模块、块设备驱动程序模块、网络设备驱动程序模块。每个模块对应一种设备类型，而每个模块中不同的设备包含的功能不一样，用户可以根据自己的需要在 VxWorks 下，创建不同的功能模块，实现系统的高性能和可裁剪性。

在 VxWorks 应用程序中，系统访问设备是通过 VxWorks 的 I/O 子系统操作的。对于字符设备和块设备 VxWorks 的 I/O 系统提供一些标准的 I/O 接口函数，而网络设备则提供另一套接口函数。这样设计的优点是应用程序开发人员在编写应用程序时不必关心底层设备硬件，就是通常所说的屏蔽底层硬件。在 VxWorks 的 I/O 系统中包含三张表：即文件描述符表、设备列表和设备驱动程序表。通过 VxWorks 提供的函数，用户可以在后两张表中注册和卸载设备及设备驱动程序。这些特性，使得 VxWorks 的 I/O 系统快速、灵活、高效，这些特点对于实时系统来说是非常重要的。

对于一个字符设备来说驱动程序包含了 7 种基本 I/O 操作函数：creat()、remove()、open()、close()、read()、write() 和 ioctl()。当然有些设备并不支持其中的某些操作，可以在实现的时候将其省略。

在 ioctl() 的实现中，可以根据设备的要求实现 select() 功能。这个功能的实现涉及一些驱动编写的知识，将在第 6 章中详述。

块存储设备的驱动不是直接与 I/O 系统相连的，如图 2.5 所示。块存储设备是以文件系统作为接口，由文件系统执行大多数的 I/O 操作，设备驱动程序只需要执行一些读写、复位设备、检查设备状态等操作。

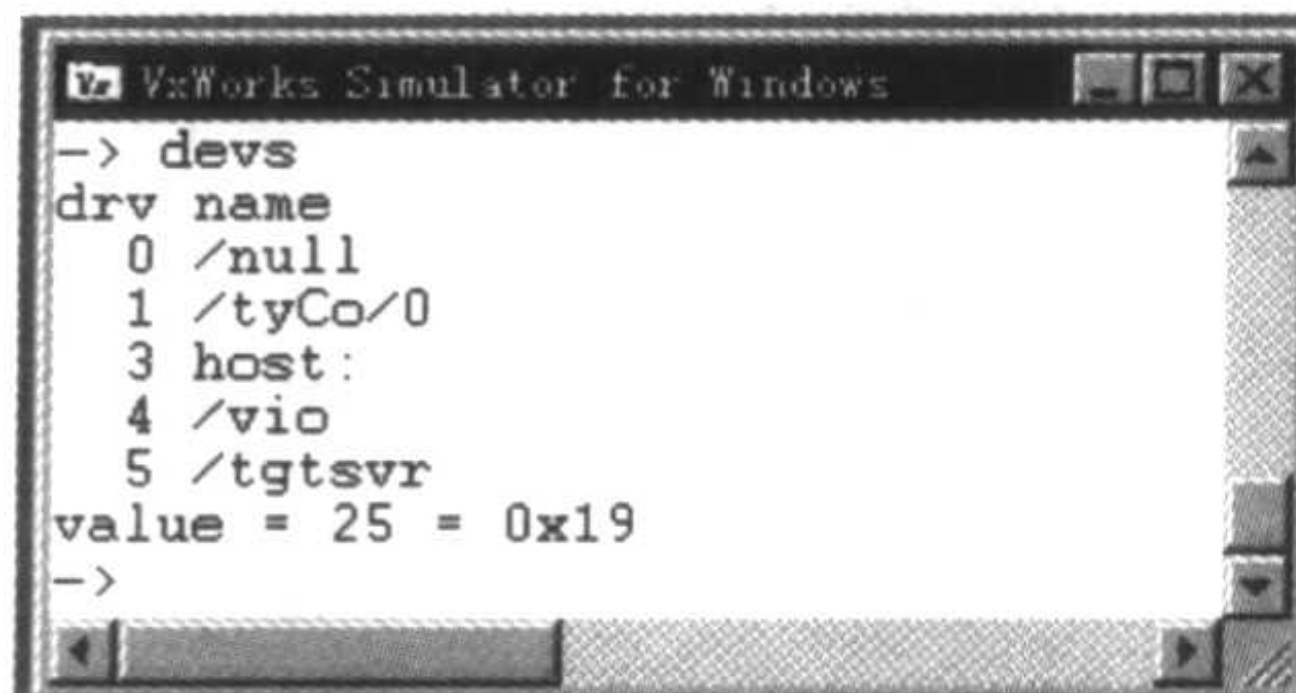


图 2.4 devs 调用

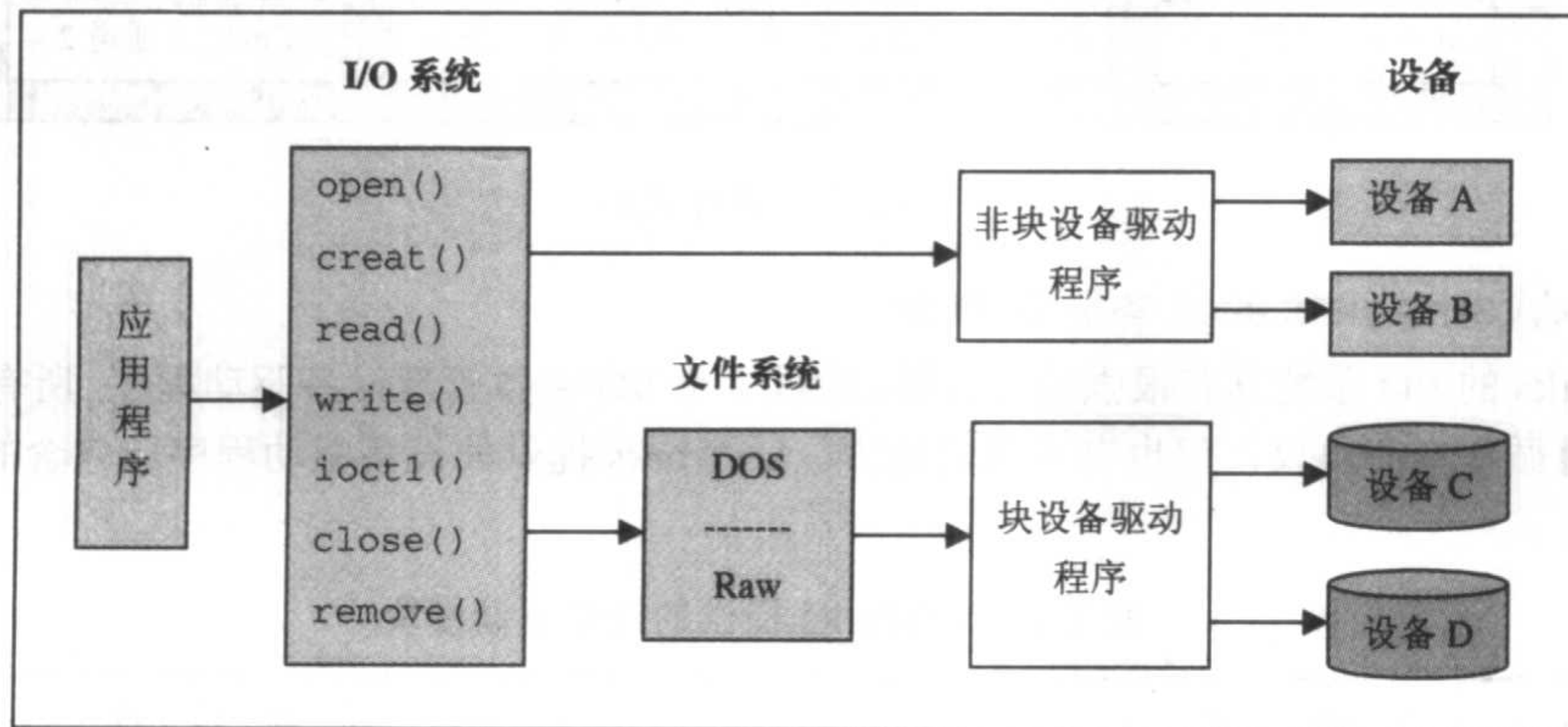


图 2.5 VxWorks 的 I/O 系统与设备驱动程序

某些设备（如网络设备、串行设备），由于其自身的特性，不是通过标准 I/O 来进行存取的，但

是也都有它们各自相关的规范，如图 2.6 和图 2.7 所示。

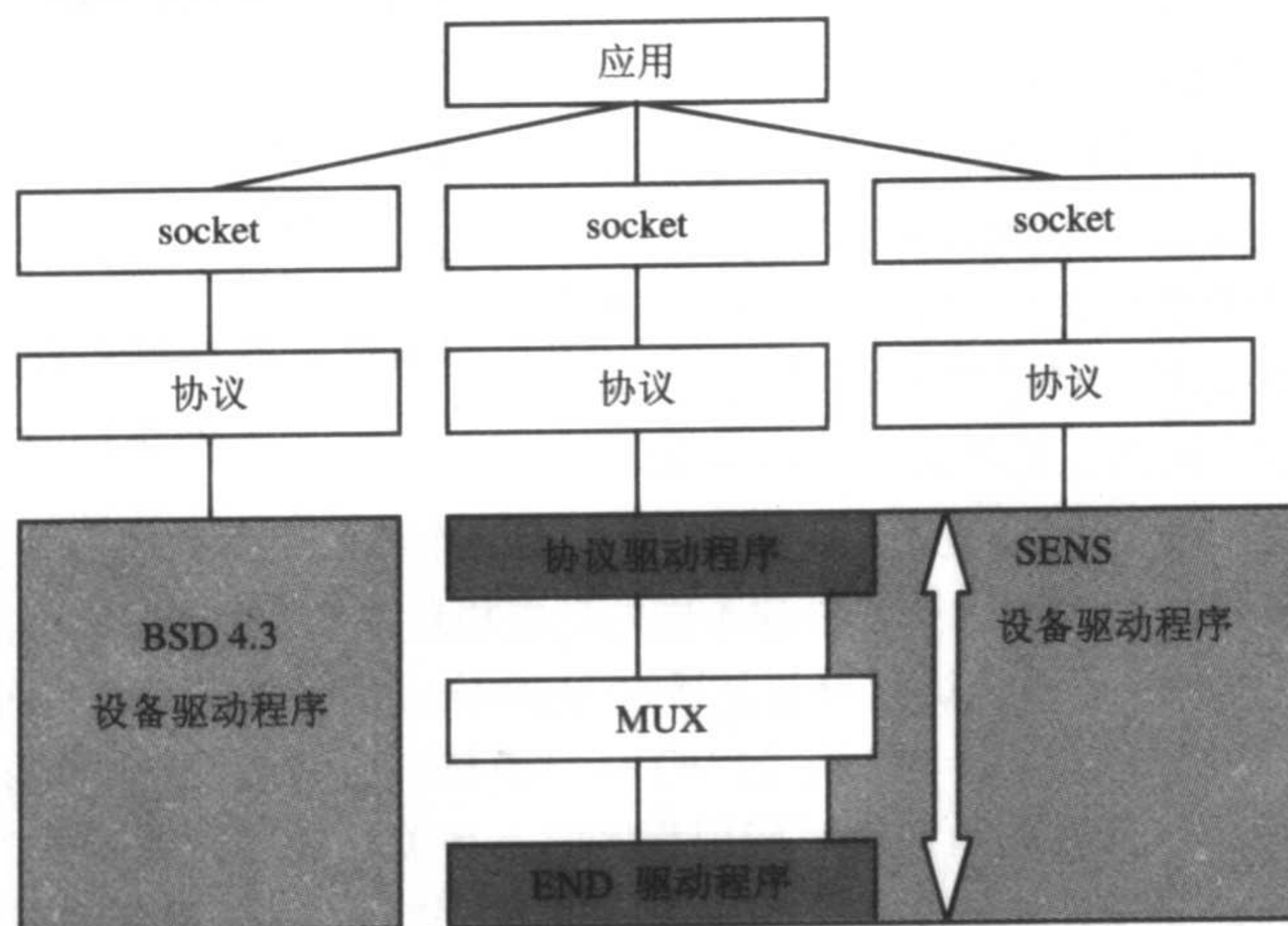


图 2.6 网络接口

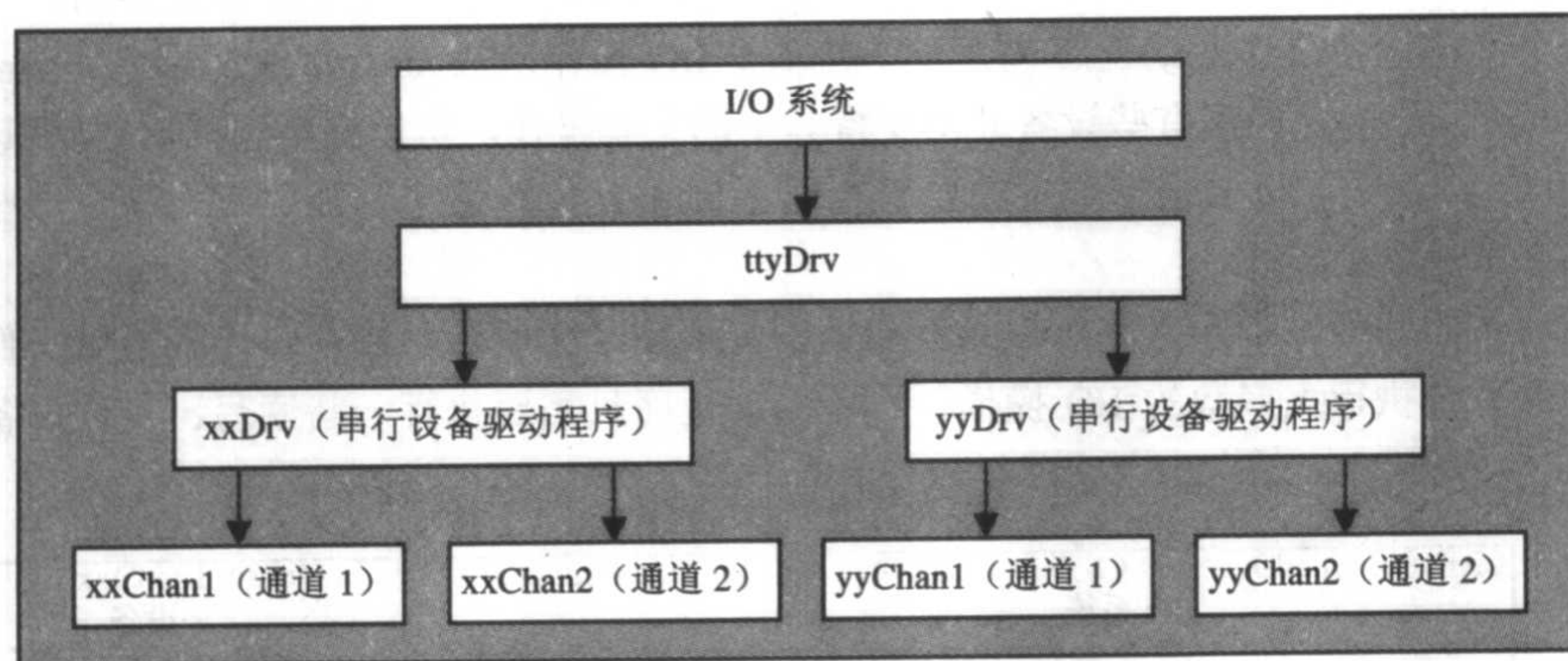


图 2.7 串行接口

2.2.2 VxWorks 提供的设备驱动模块

VxWorks 的 I/O 系统具有很强的灵活性，通过 7 个 I/O 函数管理设备驱动程序。所有的 VxWorks 设备驱动遵循基本的协议，但也有不同的地方。VxWorks 提供的设备驱动程序所包含的主要模块，如表 2-2 所示。

表 2-2 VxWorks 提供的设备驱动模块

模 块	驱 动 描 述
ttyDrv	终端设备驱动程序
ptyDrv	虚拟终端设备驱动程序

续表

模 块	驱 动 描 述
pipeDrv	管道设备驱动程序
memDrv	虚拟存储设备驱动程序
nfsDrv	网络文件系统设备驱动程序
netDrv	远程文件访问网络驱动
ramDrv	RAM 磁盘驱动
scsiLib	SCSI 接口库
—	其他特殊硬件设备驱动程序

2.3 VxWorks 下常用设备驱动程序简要描述

VxWorks 的最大优点就是提供对很多设备的支持，这也是为什么至今 VxWorks 比其他嵌入式操作系统更受欢迎的一个原因。下面对 VxWorks 提供的常用设备驱动程序进行说明，以便大家更清楚地了解 VxWorks 对设备的管理。

2.3.1 串行 I/O 设备

VxWorks 提供多种终端 (tty) 和虚拟终端 (pty) 设备驱动程序。tty 驱动程序用于实际的终端设备，而虚拟终端 pty 驱动程序用于仿真终端设备。虚拟终端对于应用程序非常重要，最常见的用途如远程登录应用。

VxWorks 的串行 I/O 设备属于一种缓冲型的连续字节流设备，每一个设备都有一个环行缓冲区用于设备的输入和输出操作，从 tty 设备中读/写数据实际上是从输入/输出环形缓冲区中读/写数据，每个环行缓冲区的大小由用户在系统初始化过程中创建设备时指定。

1. tty 设备的功能选项

tty 设备具有很多功能选项，而这些选项影响设备的运行。功能选项是通过设置设备选项字中的控制位来选择的，可以通过标准的 ioctl() 函数的 FIOSETOPTIONS 命令来对 tty 设备进行控制，下面是 tty 设备除 OPT_MON_TRAP 功能外的所有功能的设置：

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

表 2-3 中列出了 tty 设备所有的功能选项，在 ioLib.h 文件中定义了这些选项的名称。有关详细的信息，请查阅 tyLib 库相关文件内容。

表 2-3 tty 设备功能选项

选 项	描 述
OPT_LINE	选择线性传输模式
OPT_ECHO	向同一输出通道回应输入字符

续表

选 项	描 述
OPT_CRMOD	把输入的 RETURN 字符转换成 NEWLINE; 把输出的 NEWLINE 转换成 RETURN-LINEFEED
OPT_TANDEM	响应 X-on/X-off 协议 (Ctrl+Q 和 Ctrl+S)
OPT_7_BIT	从所有输入字节中提取奇偶校验位
OPT_MON_TRAP	使特殊的 ROM 监控程序软中断字符有效, 默认为 Ctrl+X
OPT_ABORT	使特殊的目标机 shell 异常中断有效, 默认为 Ctrl+C
OPT_TERMINAL	设置上述的所有选项位
OPT_RAW	设置上述的所有选项位一个也没有

2. 原始模式和线性模式

所有的 tty 设备工作在两种模式中的一种模式下: 原始模式 (非缓冲) 或线性模式。原始模式是系统默认模式, 线性模式的选择是通过设置设备选项字的 OPT_LINE 位来进行控制的。

在原始模式下, 每一个输入字符一旦输入到设备, 系统马上从设备中读取该字符。在原始模式下从 tty 设备读取字符操作时, 可以从输入环行缓冲区中读到尽可能多的字符, 而用户的读缓冲区的大小, 输入时不可以更改且只能由 tty 选项位来改变。

在线性模式下, 所有输入的字符先保存在缓冲区中, 直到 NEWLINE 字符输入后, 整行字符 (包括 NEWLINE 字符) 才被导入环行缓冲区。

3. tty 设备特殊字符

只有 tty 设备工作在线性模式下时, 表 2-4 中的特殊字符才会起作用。

表 2-4 tty 设备特殊字符

特 殊 字 符	描 述	修 改
Ctrl+H	退格 (字符删除)	tyBackspaceSet()
Ctrl+U	删除一行字符	tyDeleteLineSet()
Ctrl+D	EOF (文件结束)	tyEOFSet()
Ctrl+C	终止目标机 shell	tyAbortSet()
Ctrl+X	软中断引导 ROM 监控程序	tyMonitorTrapSet()
Ctrl+S	悬挂输出	无
Ctrl+Q	重新开始输出	无

4. I/O 控制功能

下面的表 2-5 中列出了 tty 设备的控制功能, 在 ioLib.h 文件中定义了这些功能选项。有关详细信息参考 tyLib 库、ttyDrv 驱动程序和 ioctl()函数的相关内容。

表 2-5 tty 设备 I/O 控制功能

功 能	描 述
FIOBAUDRATE	根据指定的参数设置波特率
FIOCANCEL	取消读或写
FIOFLUSH	丢弃输入和输出缓冲区中的所有字节
FIOGETNAME	获得文件描述符对应的文件名
FIOGETOPTIONS	返回当前设备的选项字
FIONREAD	获得输入缓冲区中未读字节数
FIONWRITE	获得输出缓冲区中字节数
FIOSETOPTIONS	设置设备选项字

注意

通过调用 `ioctl()` 函数的 `SIO_HW_OPTS_SET` 功能, 可以改变驱动程序的硬件选项 (例如停止位或奇偶校验位)。因为大多数设备不执行该命令, 所以可能需要把它增加到自定义的 BSP 串行设备驱动程序中, 驱动程序的源代码在 `../target/src/drv/sio` 目录下。如何执行这个命令取决于目标机上的串行芯片。在 `../target/h/sioLib.h` 文件中定义了常量 `SIO_HW_OPTS_SET`。

2.3.2 管道设备

管道是利用 VxWorks 的 I/O 系统进行任务间通信的一种虚拟设备。任务将消息写到管道上, 使这些消息可以由其他任务读出。管道设备由 `pipeDrv` 模块来管理, 并使用核心消息队列承担实际的消息传输操作。

1. 创建管道设备

可以通过调用 `pipeDevCreate()` 函数来创建管道, 调用如下所示:

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

新建的管道设备最大消息数由参数 `maxMsgs` 给定, 每个消息的最大长度由参数 `maxLength` 给定。当任务试图向一个已满的管道设备写数据时, 系统将挂起该任务直到有消息出列。消息的最大长度由参数 `maxLength` 决定, 当试图写入一条大于最大长度的消息时将会产生错误。

2. 在中断服务程序中写管道设备

VxWorks 的管道设备被设计成允许中断服务程序 (ISR) 像任务级代码一样对管道设备执行写操作。许多 VxWorks 的函数不可以在中断服务程序中调用, 包括除管道设备外的其他 I/O 设备都不允许中断服务程序执行写操作。然而, 中断服务程序可以使用管道设备在任务间进行通信, 这样中断服务程序可以与相应的任务通信。

中断服务程序通过调用 `write()` 函数来写管道设备。任务和中断服务程序可以写同一个管道设备。若管道设备满了, 由于中断服务程序无法处于挂起状态, 系统将丢弃消息。中断服务程序除了调用 `write()` 函数外, 不可以调用其他管道设备的 I/O 接口函数。

3. I/O 控制功能

管道设备的 `ioctl()` 控制命令如表 2-6 所示，这些功能定义在 `ioLib.h` 头文件中。其他详细信息，请参考 `pipeDrv` 设备驱动程序模块和 `ioLib` 库函数 `ioctl()`。

表 2-6 管道设备 I/O 控制功能

功 能	描 述
FIOFLUSH	丢弃管道中所有消息
FIOGETNAME	获得文件描述符对应的管道设备名
FIONMSG	获得管道中的剩余消息数
FIONREAD	获得管道中的第一条消息的字节数

2.3.3 虚拟存储设备

`memDrv` 驱动程序允许 I/O 系统把存储设备当作虚拟 I/O 设备一样直接执行访问操作。在设备创建时，需要指定存储位置和存储大小。当必须在 VxWorks 的引导程序之间保存数据或在 CPU 之间共享数据时，把存储设备当作虚拟存储设备的做法非常有效。`memDrv` 驱动程序不像 `ramDrv` 驱动程序，它不需要文件系统的支持。`ramDrv` 驱动程序必须依靠文件系统对存储区进行控制，而 `memDrv` 驱动程序则通过 I/O 调用提供更高级别的读和写的方法在绝对存储区执行读写操作。

1. 安装虚拟存储设备驱动程序

虚拟存储设备驱动程序首先调用 `memDrv()` 进行驱动程序初始化操作，然后再调用 `memDevCreate()` 创建虚拟存储设备。这两个函数的原型如下：

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

虚拟存储设备从存储设备的绝对地址“base”处开始存储数据，参数 `length` 指出了存储区的大小。有关存储设备驱动程序的其他信息，请参考 `memDrv` 库、`memDevCreate()` 和 `memDrv()` 的有关内容。

2. I/O 控制功能

虚拟存储设备的 `ioctl()` 控制命令如表 2-7 所示，这些功能定义在 `ioLib.h` 头文件中。其他详细的信息，请参考 `memDrv` 设备驱动程序模块和 `ioLib` 库函数 `ioctl()`。

表 2-7 存储设备 I/O 控制功能

功 能	描 述
FIOSEEK	设置当前字节在文件中的偏移量
FIOWHERE	返回当前字节在文件中的位置

2.3.4 网络文件系统（NFS）设备

网络文件系统设备允许使用网络文件系统（NFS）协议来存取远程主机上的文件。NFS 协议既指明了客户端软件，通过它从指定远程机器上读文件；也指明了服务器端软件，通过它输出文件到

登录的远程机器上。

VxWorks 下的 NFS 设备驱动程序称为 `nfsDrv`，它作为一个 NFS 客户端来存取网络上的任何 NFS 服务器上的文件。VxWorks 也允许用户运行一个 NFS 服务器来输出文件到其他远程机器上。使用 NFS 设备可以创建、打开和存取远程文件，就像存取本地硬盘上的文件系统一样。通常称这种操作为网络透明性。

1. 安装远程 NFS 文件系统

VxWorks 的网络系统屏蔽了底层的网络驱动程序。存取远程文件系统的方法是通过调用 `nfsMount()` 函数在本地安装文件系统并为它创建一个 I/O 设备。`nfsMount()` 函数带有三个参数，分别为 NFS 服务器的主机名、远程主机文件系统名、本地文件系统名。

下面示例说明了在主机 “rainbow” 上安装了名为 “/usr” 的主机文件系统，本地文件系统名为 “/vxusr”：

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

`nfsMount()` 函数根据指定的本地文件系统名（在上述示例中为 “/vxusr”）创建一个 VxWorks I/O 设备。如果本地文件名指定为 NULL，则本地文件系统名与远程文件系统名同名。

当远程文件系统安装以后，访问远程文件系统（服务端）就像访问本地文件系统一样。这样，执行上述示例的操作后，打开文件 “/vxusr/foo” 就是打开主机 “rainbow” 上的文件 “/usr/foo”。

远程文件系统必须由其实驻留的系统输出。也就是说，NFS 服务器仅可以输出本地文件系统。用户可以在服务器上使用适当的命令查看哪些是本地文件系统。NFS 系统需要认证参数来识别用户的远程访问操作。为了设置这些参数，需要使用 `nfsAuthUnixSet()` 和 `nfsAuthUnixPrompt()` 函数。

在工程的 VxWorks 配置窗口中通过选择 INCLUDE_NFS 组件，这样可以在 VxWorks 里包含 NFS 客户端的支持。有关详细配置和操作参考其他 VxWorks 的书籍。

在 VxWorks 的有关网络编程的书籍中，对 NFS 文件系统的输出和安装以及访问认证进行了详细的描述。读者也可以参考有关 `nfsLib` 库内容、`nfsDrv` 驱动程序和 SUN 系统公司的 NFS 文档。

2. NFS 客户端的 I/O 控制功能

NFS 客户端的 `ioctl()` 控制命令如表 2-8 所示，这些功能定义在 `ioLib.h` 头文件中。更多的其他信息，请参考 `nfsDrv` 设备驱动程序模块和 `ioLib` 库函数 `ioctl()`。

表 2-8 NFS 客户端 I/O 控制功能

功 能	描 述
FIOFSTATGET	获得文件状态信息（目录入口数据）
FIOGETNAME	获得文件描述符对应的文件名
FIONREAD	获得文件里未读取的字节数
FIOREADDIR	读取下一个目录入口
FIOSEEK	设置当前字节在文件中的偏移量
FIOSYNC	对远程 NFS 文件进行数据刷新
FIOWHERE	返回当前字节在文件中的位置

2.3.5 非 NFS 网络设备

VxWorks 对于非 NFS 的网络设备提供 RSH(Remote Shell Protocol)和 FTP(File Transfer Protocol) 协议访问远程主机上的文件。这些网络设备的执行需要使用 netDrv 驱动程序。当通过 RSH 或 FTP 协议打开一个远程文件时, 驱动程序将把整个文件拷贝到本地存储区中。这种操作的结果是打开的最大文件的大小受本地可用存储区的限制, 所以在使用 RSH 和 FTP 打开一个远程文件时要注意文件是否超大而无法将其拷贝到本地存储区中。一旦把文件拷贝到存储区中, 则读写操作只是对本地存储区进行操作。当关闭文件时, 如果文件被修改过, 则驱动程序会将本地存储区中的文件拷贝到远程主机中。

一般而言, NFS 设备比使用 RSH 和 FTP 设备的性能和灵活性更好, 因为 NFS 不把文件拷贝到本地存储区中。但不是所有的主机系统都支持 NFS。

1. 创建网络设备

使用 RSH 和 FTP 访问远程主机上的文件, 必须先调用 `netDevCreate()` 函数创建一个非 NFS 的网络设备。该函数所包含的三个参数是设备名、所访问的远程主机设备名、协议选择 (0 代表 RSH 协议, 1 代表 FTP 协议)。

在下面的示例中, 创建一个称作为 “rainbow” 的 RSH 设备, 该设备将访问远程主机 “rainbow”。按照一般惯例, 网络设备名是由远程主机名后加上一个冒号 (:) 所组成, 示例如下:

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

用户可以对网络设备上的文件执行创建、打开等操作, 就像在本地磁盘上一样。这样, 打开文件 “rainbow:/usr/foo” 实际上就是打开远程主机 “rainbow” 上的文件 “/usr/foo”。

对于通过使用 RSH 或 FTP 协议访问远程主机上的文件, 必须在远程和本地系统上创建许可和用户认证机制。有关创建和配置网络设备的详细信息请参考 VxWorks 的网络编程指导书籍和 netDrv 驱动程序的内容。

由于 RSH 和 FTP 的出现使存取网络文件如此方便, 在许多的 iPAS 等项目中都使用了 FTP 的方式来存取远程主机的文件, 这证明了屏蔽底层细节在开发较大系统中的优越性。



注意

所创建的网络设备允许访问远程系统上的任何文件或系统, 而安装 NFS 文件系统只允许访问指定的文件系统。

2. I/O 控制功能

除 FIOSYNC 和 FIOREADDIR 功能外, RSH 和 FTP 设备的 `ioctl()` 控制命令与 NFS 设备的控制功能一样。这些功能的定义在 `ioLib.h` 头文件中。其他详细的信息, 请参考 netDrv 设备驱动程序模块和 ioLib 库函数 `ioctl()`。

2.3.6 块设备

块设备是由一系列按顺序排列的一个一个单独的可访问的数据块所组成的设备。最常见的块设备就是磁盘。在 VxWorks 系统中, 术语 “块” 表示块设备上可寻址的最小地址单元。对于大多数磁

盘设备，VxWorks 所描述的“块”对应常说的术语“扇区”。

VxWorks 系统中，块设备接口与其他 I/O 设备接口稍微有些差别。块设备不是直接与 I/O 系统连接，而是需要一组底层驱动程序和文件系统的支持。底层驱动程序与文件系统连接，而文件系统与 VxWorks 的 I/O 系统连接。这样的设计允许底层驱动程序可以被各种不同的文件系统所使用，减少了驱动程序必须支持的 I/O 函数的个数。块设备底层驱动程序的内部执行详细过程请阅读第 8 章的块设备驱动程序的编写。

1. 文件系统

对于块设备的使用，VxWorks 提供了与 MS-DOS (dosFs) 和 RT-11 (rt11Fs) 文件系统兼容的文件系统库。另外，还提供了一个简单原始文件系统 (rawFs) 库，它处理整个磁盘就像处理一个大文件一样；也提供一个支持 SCSI 磁带设备的文件系统，它的序列化便于一个一个单独地数据块按顺序进行读写操作；同时还提供了一个支持 CD-ROM 设备的文件系统。有关这些文件的详细描述，请参考相关书籍。也可以阅读 dosFsLib、rt11FsLib、rawFsLib、tapeFsLib 和 cdromFsLib 库文件内容。

2. RAM 磁盘驱动程序

VxWorks 的 RAM 磁盘是将内存模拟为磁盘，将所有数据保存在内存中，内存的位置和“磁盘”的大小依赖于调用 *ramDevCreate()* 函数所创建的结果，可以通过多次调用这个函数来创建多个 RAM 磁盘。RAM 磁盘驱动程序由 ramDrv 驱动程序模块所提供。

RAM 磁盘所使用的内存可以预先分配并将地址作为参数传给 *ramDevCreate()* 函数，或通过调用 *malloc()* 函数从系统内存池中自动分配内存。

设备创建后，必须使用文件系统的设备初始化函数或文件系统的创建函数分配设备名和文件系统 (dosFs、rt11Fs 或 rawFs)。例如 *dosFsDevInit()* 或 *dosFsMkfs()* 函数。有关 RAM 设备的描述信息通过 BLK_DEV 结构传递给文件系统。RAM 磁盘创建函数将会返回指向这个结构的指针。

下面的示例创建了一个自动分配 200KB 内存的 RAM 磁盘，扇区大小为 512 字节，一条磁道，没有扇区偏移。设备分配的设备名为“DEV1:”，使用的文件系统是 dosFs。具体调用和配置如下：

```
BLK_DEV      *pBlkDev;
DOS_VOL_DESC *pVolDesc;
pBlkDev = ramDevCreate (0, 512, 400, 400, 0);
pVolDesc = dosFsMkfs ("DEV1:", pBlkDev);
```

dosFsMkfs() 函数以默认参数形式调用 *dosFsDevInit()* 函数并通过选择控制功能 FIODISKINIT 命令调用 *ioctl()* 函数初始化磁盘上的文件系统。

如果 RAM 磁盘内存已经包含了一个磁盘映像，而且 *ramDevCreate()* 函数的第一个参数就是内存中 RAM 磁盘的地址。当映像被创建时，格式化参数必须与已有的磁盘映像格式一致。如下所示：

```
pBlkDev = ramDevCreate (0xc0000, 512, 400, 400, 0);
pVolDesc = dosFsDevInit ("DEV1:", pBlkDev, NULL);
```

在这种情况下，必须使用 *dosFsDevInit()*，因为 RAM 磁盘中已经存有文件系统并且不需要重新初始化。如果创建一个 RAM 磁盘的地址与 VxWorks 上一次引导时使用的 RAM 磁盘地址相同，这种做法是非常有用的，系统保留了原有 RAM 磁盘中的内容。创建一个 rt11Fs 文件系统的 RAM 磁盘，需要使用 *rt11FsMkfs()* 和 *rt11FsDevInit()* 函数，其他操作与上述过程一样。有关 RAM 磁盘驱动程序

的详细信息，请参考 ramDrv 模块；有关文件系统的详细信息，请参考相关书籍的文件系统章节。

3. SCSI 驱动程序

SCSI 是一个标准外设接口，它允许连接各种硬盘、光驱、软驱、磁带设备和 CD-ROM 设备。SCSI 块设备驱动程序与 dosFs 和 nt11Fs 库兼容，对目标机的配置提供如下特点：

- 非网络环境下的本地大量数据存储；
- 比以太网快的 I/O 吞吐量。

VxWorks 的 SCSI-2 取代了原有的 SCSI 的支持，尽管如此，原有的 SCSI 提供了可选配置功能，现在称它为 SCSI-1。通过使用 SCSI-2，VxWorks 环境依然可以执行 SCSI-1 应用，例如在 SCSI-1 下创建的文件系统。但是，如果应用程序直接使用定义在 scsiLib.h 文件中的 SCSI-1 数据结构，则可能需要对其修改和重新编辑以兼容 SCSI-2。

VxWorks 的 SCSI 支持包含两种模块：一种支持与设备无关的 SCSI 接口，另一种支持特殊的 SCSI 控制器。scsiLib 库提供与设备无关的 SCSI 接口的支持；特殊设备库提供配置程序支持特殊 SCSI 控制器（例如 wd33c93Lib 库支持 WD33C93 设备、mb87030Lib 库支持富士通公司的 MB87030 设备）。对于个别目标机，文件 sysLib.c 中包含了额外的程序支持目标机系统。

由于 SCSI 技术本身比其他设备要复杂，而且在当前的项目开发中使用较少，所以本书中不对它做详细的分析，请读者参考其他资料。

小 结

对硬件设备结构和设备的驱动程序有了一定的认识后，再来认识 VxWorks 下的设备及设备驱动程序会有“似曾相识”的感觉。在这一章中，我们集中讨论了 VxWorks 操作系统下的设备及设备驱动程序，其中包括设备的分类、设备的命名、设备驱动程序概述、VxWorks 本身包含的设备驱动程序模块以及相应的概念（例如设备描述符、设备列表等）。

第 3 章 VxWorks 下设备驱动程序的分析

VxWorks 下的设备驱动程序在设计上继承了通常设备驱动程序的设计，同时也包含了自己的独到之处。例如 MUX 技术（网络驱动程序的一种技术，有关网络驱动将在第 9 章中介绍）的设计。VxWorks 下的设备驱动程序都采用模块化设计，各模块之间功能明确、接口规范，将在后面的具体设备驱动程序编写章节中详细描述，而这一章主要对设备驱动程序的总体结构及设计进行分析。

3.1 VxWorks 下设备驱动程序在系统中的层次

VxWorks 设备驱动程序在系统中的层次如图 3.1 所示。

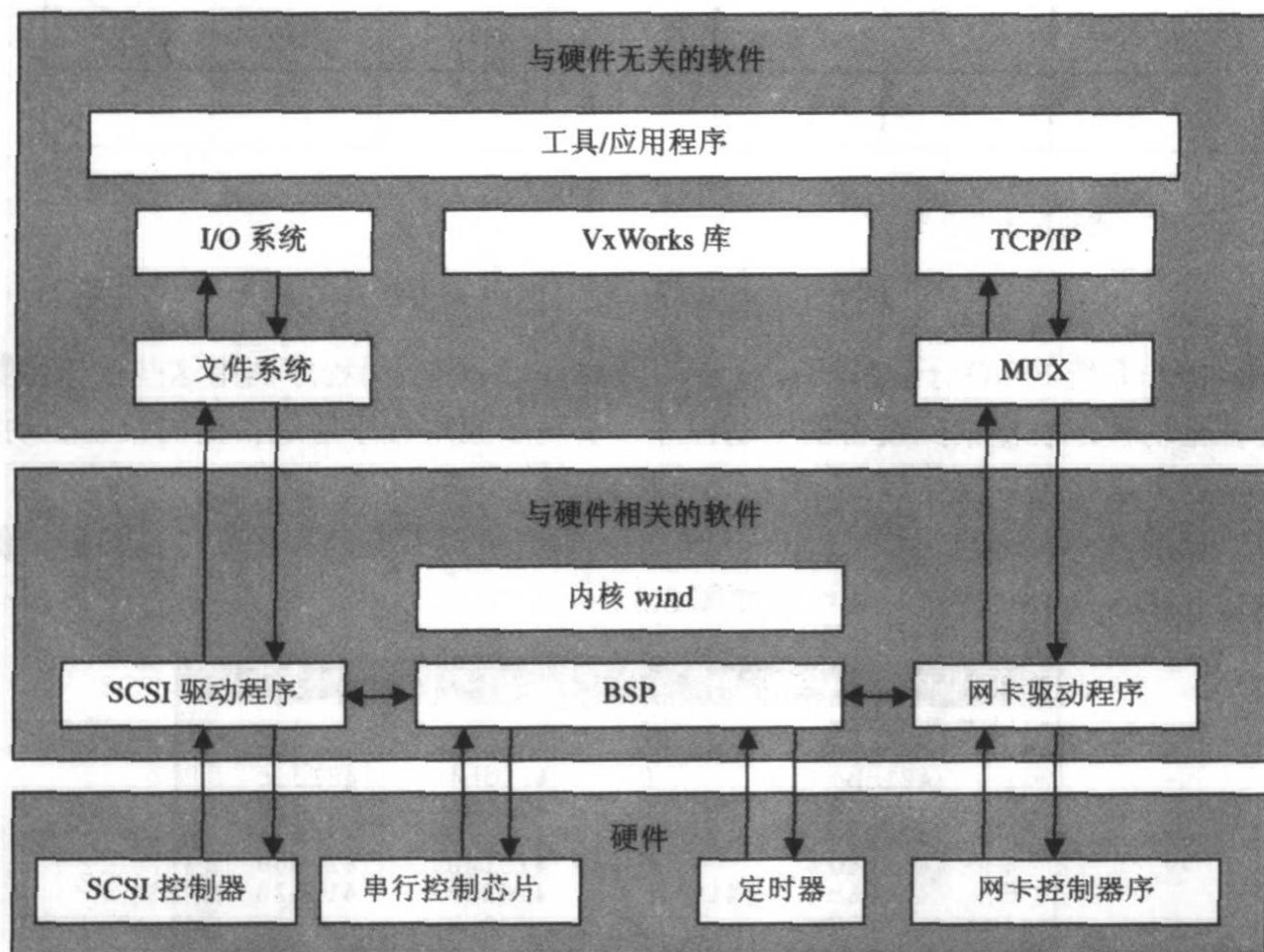


图 3.1 BSP 和设备驱动程序

从图 3.1 中可以看到，VxWorks 设备驱动程序是直接控制下层硬件设备的一个上层软件接口，同时对上层应用程序提供接口函数。

这样，编写出来的应用程序不依赖于硬件，增强了应用程的可维护性和可移植性等。随着系统越来越复杂，程序分工也越来越明细，系统层次化成为了一种必然。在实际项目的开发过程中，精

明的程序员们通常都会找一些类似的程序来进行移植或参考，更希望以前开发的程序不用修改就能重新用到新的项目中。尤其是那些经典的老程序已经通过严格测试，并经过了实践检验。用于移植既减少了开发成本又降低了开发风险。

3.2 VxWorks 的设备驱动程序表

在软件开发过程中经常使用索引表，通过它来检索和管理资料可以提高效率。同样在 VxWorks 应用中，系统访问设备是通过文件描述符表、设备列表和设备驱动程序表来进行，它们主要起索引和管理设备及设备驱动程序的功能。与设备驱动程序直接相关的是设备驱动程序表，如图 3.2 所示。

	creat	delete	open	close	read	write	ioctl
0							
1							
2	myOpen	NULL	myOpen	myClose	myRead	myWrite	myIoctl
3							

驱动程序索引号

驱动程序入口点

图 3.2 VxWorks 设备驱动程序表

VxWorks 操作系统的 I/O 子系统提供标准的函数接口。当应用程序调用这些接口函数时，I/O 子系统通过设备驱动程序表检索到设备的驱动程序，实现了应用程序通过指定的设备驱动程序访问设备、进行数据的发送和接收操作等。

用户可以调用 VxWorks 提供的 iosDrvShow() 函数查看设备驱动程序表中的内容。例如在 Shell 工具下，执行 iosDrvShow() 操作，其结果如图 3.3 所示。

```

Shell vxsim@rainbow
-> iosDrvShow
drv  create      delete      open      close
1    421db4      0          421db4    421ddc
2    0           0          424fd4    0
3    426e88      426f04     427170    426e34
4    415f40      0          415f40    416000
5    41626c      41642c     416288    416520
value = 0 = 0x0
  
```

图 3.3 iosDrvShow 调用

3.3 VxWorks 下常用设备的驱动程序源文件

在 VxWorks 操作系统下，设备驱动程序通常是以库文件 (*.a) 或对象文件 (*.o) 加上头文件的形式提供给用户的。库文件或对象文件是驱动程序的二进制码，而头文件是驱动程序函数的声明

及一些端口地址或相关宏定义。如果需要设备驱动程序的源代码,则需要购买 VxWorks 的 BSP 开发包。在 Tornado 环境下,成功安装 BSP 开发包后,可以在../target/src/drv 目录下找到一些设备驱动程序的源代码文件,如图 3.4 所示的设备驱动程序源代码目录。这些设备驱动程序所涵盖的设备包括:DMA 设备、网卡、串口、块设备、SCSI 等。

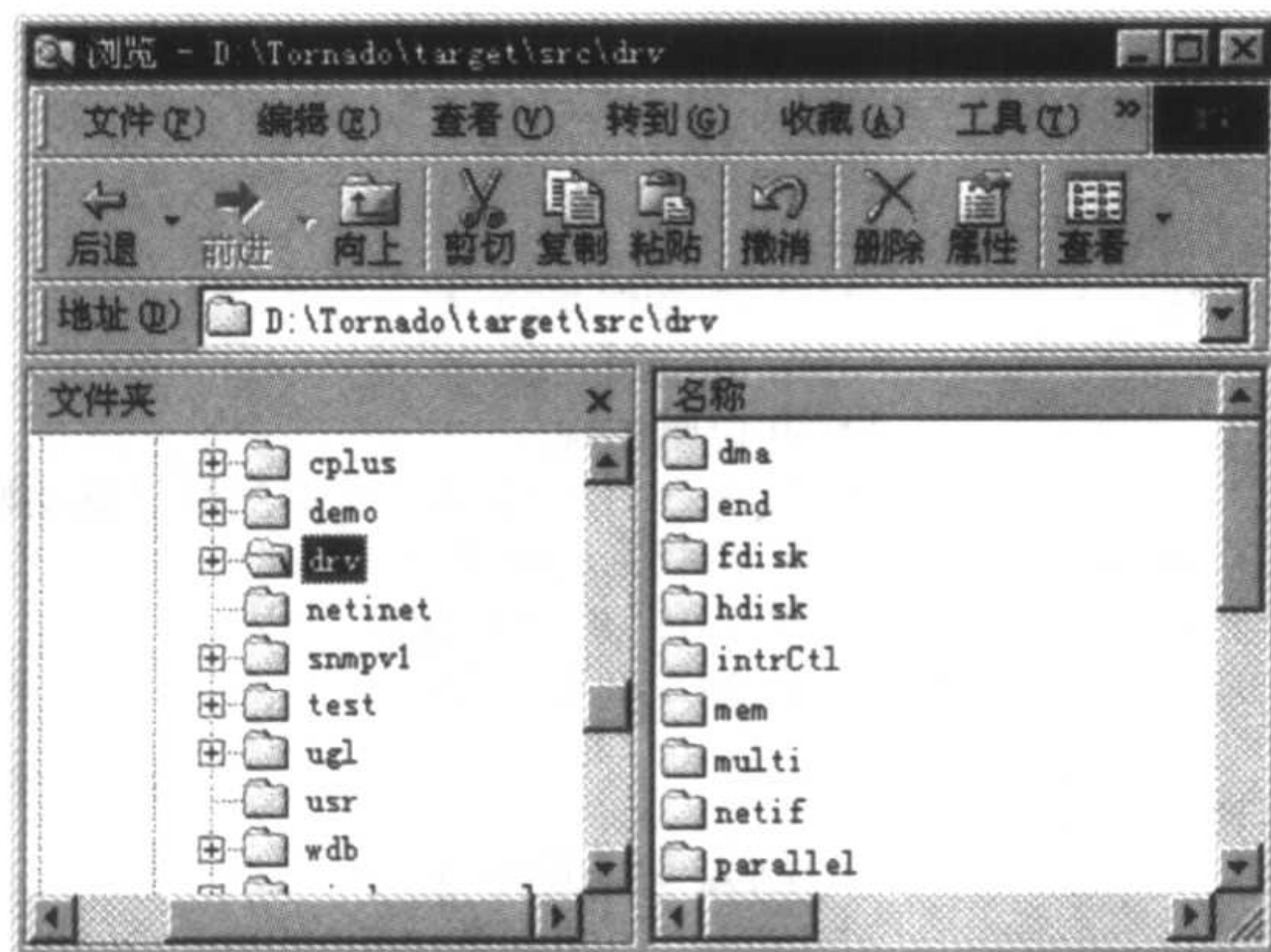


图 3.4 设备驱动程序源代码目录

设备驱动程序的源代码对用户来说是非常重要的,无论是用于学习还是开发设备驱动程序,拥有它都会带来“事半功倍”的效果。好比用一些实例手把手教用户如何编写 VxWorks 的设备驱动程序并在开发中提供宝贵的资源。这样的好处是可以缩短产品的开发周期和提高产品的开发效率,把产品快速地推向市场。

3.4 VxWorks 下设备驱动程序的配置

3.4.1 VxWorks 自带设备驱动程序的配置

配置 VxWorks 下自带的设备驱动程序是一件非常简单的事情,只要在相关配置文件中定义驱动程序相应的宏就可以把设备驱动程序添加到系统中。下面将列举这些设备驱动程序的宏定义所在的文件。

在..\Tornado\target\config\\${bspname} 目录下,可以找到 config.h 文件,在这个文件里包含了通用设备驱动程序的宏定义,譬如网卡驱动程序、硬盘驱动程序、时钟驱动程序、软盘驱动器驱动程序等。打开这个文件并按需求编辑,就可以方便地配置设备驱动程序。下面是...\config\pcPentium\目录下 config.h 文件的一部分内容说明。

```
=====
Config.h 头文件部分内容
=====
...
#ifndef INCconfigh
#define INCconfigh
```



```

/* BSP 版本说明 */
#define BSP_VER_1_1 1 /* 1.2 版向下兼容 1.1 */
#define BSP_VER_1_2 1
#define BSP_VERSION "1.2"
#define BSP_REV "0" /* 0 指第一修订版 */

#include "configAll.h"
#include "pc.h"
...
/* 设备驱动程序和文件系统的选择 */
#define INCLUDE_DOSFS /* DOS 文件系统 */
#define INCLUDE_FD /* 软盘控制器驱动程序 */
#define INCLUDE_LPT /* 并口驱动程序 */
#define INCLUDE_ATA /* IDE/EIDE(ATA) 接口的硬盘驱动程序 */
#undef INCLUDE_TIMESTAMP /* 时间戳驱动程序 */
#undef INCLUDE_TFFS /* TrueFFS 驱动程序 */
#undef INCLUDE_PCMCIA /* PCMCIA 驱动程序 */

#ifdef INCLUDE_PCMCIA
#define INCLUDE_ATA /* ATA 驱动程序 */
#define INCLUDE_SRAM /* SRAM 驱动程序 */
#undef INCLUDE_TFFS /* TFFS 驱动程序 */
#ifdef INCLUDE_NETWORK
#define INCLUDE_ELT /* 3COM EtherLink III 驱动程序 */
#endif
#endif
...
/* 网络设备驱动程序选择 */

#define INCLUDE_END /* 使用增强型的网络驱动 */

#undef INCLUDE_ULTRA /* 包含 SMC Elitel6 Ultra 接口 */
#undef INCLUDE_ENE /* 包含 Eagle/Novell NE2000 接口 */
#undef INCLUDE_ELT /* 包含 3COM EtherLink III 接口 */
#undef INCLUDE_ESMC /* 包含 SMC 91c9x Ethernet 接口 */
#define INCLUDE_FEI /* 包含 Intel Ether Express PRO100B PCI */
#define INCLUDE_SLIP /* 包含串行线 IP 接口 */
#define SLIP_TTY 1 /* 串行线 IP 使用通道 COM2 */
#undef INCLUDE_ELC /* 包含 SMC Elitel6 接口 */
#undef INCLUDE_EEX /* 包含 INTEL EtherExpress 接口 */
#undef INCLUDE_EEX32 /* 包含 INTEL EtherExpress flash 32 */
#undef INCLUDE_EX /* 包含 Excelan Ethernet 接口 */
#undef INCLUDE_ENP /* 包含 CMC Ethernet 接口 */
#undef INCLUDE_SM_NET /* 包含背板网络接口 */
#undef INCLUDE_SM_SEQ_ADDR /* 共享内存网络自动地址设置 */
#undef INCLUDE_EL_3C90X_END /* 3com fast etherLink XL PCI */
#undef INCLUDE_LN_97X_END /* AMD 79C972 END 驱动程序 */
...
/* (系统时钟和辅助时钟)
 * SYS_CLK_RATE_MAX 依赖于 CPU 的能力以及应用程序负担。
 * 使用这个值可以通过内部的测试, 但最大值可以到 PIT_CLOCK 所设的值。
 */
#define SYS_CLK_RATE_MIN 19 /* 最小系统时钟每秒 tick 数 */

```



```

#define SYS_CLK_RATE_MAX (PIT_CLOCK/256) /* 最大系统时钟每秒 tick 数 */
#define AUX_CLK_RATE_MIN 2 /* 最小辅助时钟每秒 tick 数 */
#define AUX_CLK_RATE_MAX 8192 /* 最大辅助时钟每秒 tick 数 */

/* PC 机终端的定义 */

#if TRUE
#define INCLUDE_PC_CONSOLE /* 包含键盘和 VGA */
#endif /* TRUE/FALSE */

#ifdef INCLUDE_PC_CONSOLE

#define PC_CONSOLE 0 /* 终端号 */
#define N_VIRTUAL_CONSOLES 2 /* shell / 应用程序 */

#endif /* INCLUDE_PC_CONSOLE */

#undef NUM_TTY
#define NUM_TTY (N_UART_CHANNELS)

/* 定义键盘类型。默认为 PS/2 接口的标准 101 键盘 */
#define PC_KBD_TYPE PC_PS2_101_KBD
#if FALSE
#define PC_KBD_TYPE PC_XT_83_KBD
#endif /* FALSE */
...
#endif /* INCconfig */

#if defined(PRJ_BUILD)
#include "prjParams.h"
#endif

```

注意

大部分常用设备驱动程序可以在 config.h 文件中配置，但有些可能需要在 configAll.h 中配置。然而，推荐的方式是把 configAll.h 中的设备驱动程序宏定义重新在 config.h 文件中进行配置，保证 configAll.h 文件不被更改。因为 configAll.h 是一个公共文件，其他项目配置也需要它。

3.4.2 设备厂商提供的 VxWorks 下设备驱动程序的配置

在产品开发过程中，开发者所使用的设备可能并没有在 VxWorks 的 BSP 开发包中提供驱动程序。这时通常需要咨询设备厂商是否在 VxWorks 操作系统下提供该设备的驱动程序，如果不能提供只有自己开发或寻求他人开发。

这一小节主要描述设备厂商提供的 VxWorks 下设备驱动程序如何配置，并说明相关信息。关于如何开发设备驱动程序将在后面的章节中详细描述，其中包括字符设备、块设备、串行设备、网络设备。

设备厂商提供的设备驱动程序通常以库文件或对象文件，加上头文件的形式发布给用户。其中，

还会包含自述文件 (readme 文件), 有些厂商甚至可能还会提供一个常见问题解答文件 (FAQ 文件)。在 readme 文件里描述了如何把该设备驱动程序加到 VxWorks 系统中。这个文件通常还包含了驱动程序版本信息、编写厂商、介绍、系统需求及驱动程序使用说明等。下面是一个 VxWorks 下 ATI 显卡驱动程序的 readme 文件。

=====

ATI 显卡驱动程序的 readme 文件

=====

JUMPttec VxWorks ATI Rage Display Driver
Programmer's Manual /* VxWorks 下 ATI 显卡驱动程序程序员指南 */

Version 1.1 /* 版本、创建日期、厂商有关信息 */
Date: 2000.06.27
Copyright 2000 JUMPttec Industrielle Computertechnik AG

Introduction /* 介绍 */

The ATI RAGE Display Driver adds display support for JUMPttec boards that are equipped with ATI RAGE Mobility M graphics chips under VxWorks. The driver supports 640*480 with 256 colors. Additional resolutions are available on request.

Requirements /* 系统需求 */

To run the JUMPttec ATI RAGE display driver you must install the following products first:

- Wind River Tornado 2.0 (VxWorks 5.4, UGL 1.2)
- BSP Pentium

Current Restrictions /* 当前限制 */

Currently the ATI RAGE display driver has only been tested with the above version of the OS.

Tests have been done with CRT and the LCD panel Sharp LQ0DAS9010. Other LCD panels should work but have not been tested.

Please make sure that the panel displays a proper picture during the boot phase. If this is not the case then an updated BIOS is required before the VxWorks driver can be used.

Installation /* 安装说明 */

Please make sure that Tornado has been installed properly.

All paths are relative to the Tornado root that is C:\Tornado by default.

Board Support Package (BSP) /* 板级支持包说明 */

First of all it is recommended that you create a new BSP based on the PENTIUM BSP. The location of the new BSP will be referred to as target\config\<BSP> in this documentation.

Driver Files /* 设备驱动程序库文件及头文件等 */

Copy these files to your Tornado directory

- target\lib\objPENTIUMgnuvx\JATI.a
- target\h\ugl\driver\output\jati\udjati.h
- target\h\ugl\config\jaticfg.h

target\config\<BSP>\config.h /* config.h 文件的修改 */

In your new BSP you must make the following additions at the end of config.h:

#define INCLUDE_PCI

#define UGL_CONFIG_JATI

You need to make sure that VGA or any other display driver is excluded.

target\config\<BSP>\sysLib.c /* sysLib.c 文件的修改 */

The ATI Rage graphics chip is a PCI device and needs to dynamically map memory into the processors address space. Since the base VxWorks package does not support dynamic virtual address management, the function sysATIPciInit() must be called before the driver is actually initialized by UGL or an application.

Add the following **bold** lines:

void sysATIPciInit(void);

void sysHwInit (void)

{

. . .

#endif /* INCLUDE_SCSI */

#ifdef UGL_CONFIG_JATI

sysATIPciInit();

#endif

#endif /* INCLUDE_PCI */

. . .

}

target\h\ugl\config\uglcfg.h /* uglcfg.h 文件的修改 */

The recommended way is to let UGL bind and initialize the graphics driver. For this to work you need to make changes to the uglcfg.h. The supplied file target\h\ugl\config\uglcfg.h contains these changes and has been provided for your convenience. If it is compatible with your environment then you can replace this file.

If you prefer to make the changes manually then after the UGL_CONFIG_VGA section add:

/* add ATI Rage driver if selected */

#ifdef UGL_CONFIG_JATI

#include <ugl/config/jaticfg.h>

#else

#define UGL_JATI_DEVICES_LIST

#endif

If you prefer to initialize the graphics driver in your application then include:

#include <ugl/driver/output/jati/udjati.h>

And call:

uglATI8BitDevCreate(&uglATI8BitModeInitData);

Using the Driver **/* 驱动程序的使用*/**

After that you are ready to create a new Tornado project based on the new BSP. Please add the **UGL components** and make certain that NO other graphics driver is selected. Not even VGA.

There are a number of ways to include the jati.a display driver library to your project. One way of doing that is to add the name jati.a to the **EXTRA_MODULES** or **LIB** macro in the projects settings.

阅读上面的 readme 文件能体会到它的关键作用。所以，开发者要认真、仔细地阅读驱动程序的 readme 文件。另一方面，上述的 readme 文件结构清晰、内容完整，是开发者编写驱动程序说明文

3.5 VxWorks 下设备驱动程序常用函数

3.5.1 端口读/写用函数

VxWorks 系统中提供了常用的端口读/写函数,在表 3-1 中列出了 VxWorks 设备驱动程序常用的端口读/写函数。

函 数	描 述
sysInByte()	从端口里读一个字节
sysOutByte()	往端口写一个字节
sysInWord()	从端口里读一个字
sysOutWord()	往端口写一个字
sysInLong()	从端口里读一个双字
sysOutLong()	往端口写一个双字
sysInWordString()	从端口里读一串字
sysOutWordString()	往端口里写一串字
sysInLongString()	从端口里读一串双字
sysOutLongString()	往端口里写一串双字

3.5.2 中断管理函数

28

表 3-2 VxWorks 设备驱动程序常用中断函数

函 数	描 述
IntConnect()	将中断服务程序和中断向量连接
SysIntEnable()	打开总线中断
SysIntDisable()	关闭总线中断
SysBusIntGen()	产生总线中断
sysBusIntAck()	应答总线中断
SysIntEnablePIC()	打开 8259A PIC 中断
SysIntDisablePIC()	关闭 8259A PIC 中断
SysClkConnect()	将中断服务程序和系统时钟中断关联
SysClkDisable()	关闭系统时钟中断
SysClkEnable()	打开系统时钟中断

在 intArchLib 库中包含了这些资源，函数声明在...\target\h 目录下 intlib.h 文件中。

3.5.3 驱动程序注册和卸载等函数

为了合理、有效地管理 VxWorks 下的设备及设备驱动程序，VxWorks 提供了设备注册和卸载功能、设备驱动程序的注册和卸载功能以及显示系统中设备和设备驱动程序相关信息的功能。这些常用函数如表 3-3 所示。

表 3-3 VxWorks 常用注册和卸载设备及设备驱动程序函数

函 数	描 述
IosDrvInstall()	注册设备驱动程序
IosDrvRemove()	卸载设备驱动程序
IosDrvShow()	显示系统中所有设备驱动程序
IosDevAdd()	注册一个设备
IosDevDelete()	卸载一个设备
IosDevFind()	查找一个设备
IosDevShow()	显示系统中所有设备

在 ioslib 库里包含了这些资源，而函数声明在...\target\h 目录下 ioslib.h 文件中。

3.5.4 PCI 总线设备相关函数

在目前的嵌入式系统中，硬件系统的设计经常会用到 PCI 总线，而且许多外围设备使用了 PCI 总线接口，例如网卡、显卡、声卡等接口板。对于 PCI 总线设备的访问，VxWorks 提供了一些相关函数。表 3-4 中列出了 VxWorks 下 PCI 总线设备驱动程序常用的函数。

表 3-4 VxWorks 下 PCI 总线设备驱动程序常用函数

函 数	描 述
PciFindDevice	查找 PCI 设备
pciFindDeviceShow	查找并打印有关 PCI 设备信息
pciConfigInByte	从 PCI 配置空间读一个字节
pciConfigOutByte	往 PCI 配置空间写一个字节
pciConfigInWord	从 PCI 配置空间读一个字
pciConfigOutWord	往 PCI 配置空间写一个字
pciConfigInLong	从 PCI 配置空间读一个双字
pciConfigOutLong	往 PCI 配置空间写一个双字
pciDeviceShow	打印有关 PCI 设备信息
pciIntConnect	中断服务程序与 PCI 设备中断关联
pciIntDisconnect	中断服务程序与 PCI 设备中断拆开

小 结

本书的前半部分重点描述如何在 VxWorks 下编写设备驱动程序，所以从本章开始将集中对设备驱动程序的编写进行描述。本章对驱动程序在系统中的层次、设备驱动程序表、常用设备驱动程序文件、设备驱动程序的配置和设备驱动程序常用函数进行了详细介绍，这些内容涉及到了系统与设备驱动程序的交互接口。讲述这一部分内容的目的是让读者对 VxWorks 下的设备驱动程序有更进一步的认识，有助于读者开展实际的设备驱动程序的开发工作。

第 4 章 驱动程序的轮询和中断处理

4.1 概述

设备驱动程序是控制硬件运转的软件解决办法。它直接控制设备，能设置并测试设备有关的状态位。当硬件发生事件或状态改变时，软件必须知道这些情况以便应用程序可以做出相应的反应。例如下面的状态改变和事件发生：

- 软盘驱动器门打开；
- DMA 操作结束；
- 数据已经到达；
- 数据发送结束；
- 设备已经被初始化，并可以使用。

设备驱动程序通常采用两种方式来获得这些状态的改变或事件的发生：

(1) 轮询 (polling) 方式。所谓轮询，是指软件周期性查询硬件事件的发生。实际上是通过读取设备状态寄存器来获得设备状态的改变或事件的发生，然后进行相应处理。例如外部设备处于准备好状态或者空闲状态，则 CPU 可以执行输入指令或输出指令与外部设备交换信息。

(2) 中断处理方式。所谓中断处理，是指事件的发生导致软件执行顺序改变。也就是说当事件发生或状态改变时，硬件产生中断，然后执行相应的中断服务程序进行处理。

用轮询方式时，如果一个系统有多个外部设备，那么 CPU 只能轮流对每个外部设备进行查询，而这些外部设备的速度往往并不相同。这时 CPU 显然不能很好满足各个外部设备随机性的状态改变或事件的发生，所以，这种方式的实时性比较差。

为了提高 CPU 的使用效率并使系统具有高实时性能，在实时系统以及多个外部设备的系统中，通常采用中断处理方式来通知系统外部设备的事件发生或状态改变。因为在中断处理方式下，外部设备具有申请 CPU 服务的主动权，当设备的事件发生或状态改变时，便可以向 CPU 发出中断请求，使 CPU 暂时停止当前的工作而处理外部设备的请求。在这种方式下，CPU 不必花费大量时间查询外部设备的工作状态，显然提高了系统的实时性能。

4.2 轮询

当使用不支持中断方式的设备时，可以通过轮询方式对设备进行管理。而轮询方式可以采用下面几种实现方式：

- 驱动程序任务在系统空闲时进行设备轮询；

- 驱动程序任务周期性进行设备轮询；
- 驱动程序通过看门狗定时器进行设备轮询；
- 驱动程序通过辅助时钟进行设备轮询。

具体采用哪种方法，用户应根据系统来做决定，其中需要考虑硬件平台的支持和应用程序的要求。

4.2.1 驱动程序轮询访问设备的方法

1. 驱动程序直接访问设备

在 VxWorks 中，可以由驱动程序直接轮询访问设备，如图 4.1 所示。这种操作有如下特点：

- (1) 驱动程序在应用任务的上下文里执行；
- (2) 所有的 I/O 是同步的。

2. 驱动程序使用轮询任务访问设备

使用这种方式，应用程序调用驱动程序，驱动程序与驱动轮询任务进行通信，由驱动轮询任务来访问设备，如图 4.2 所示。

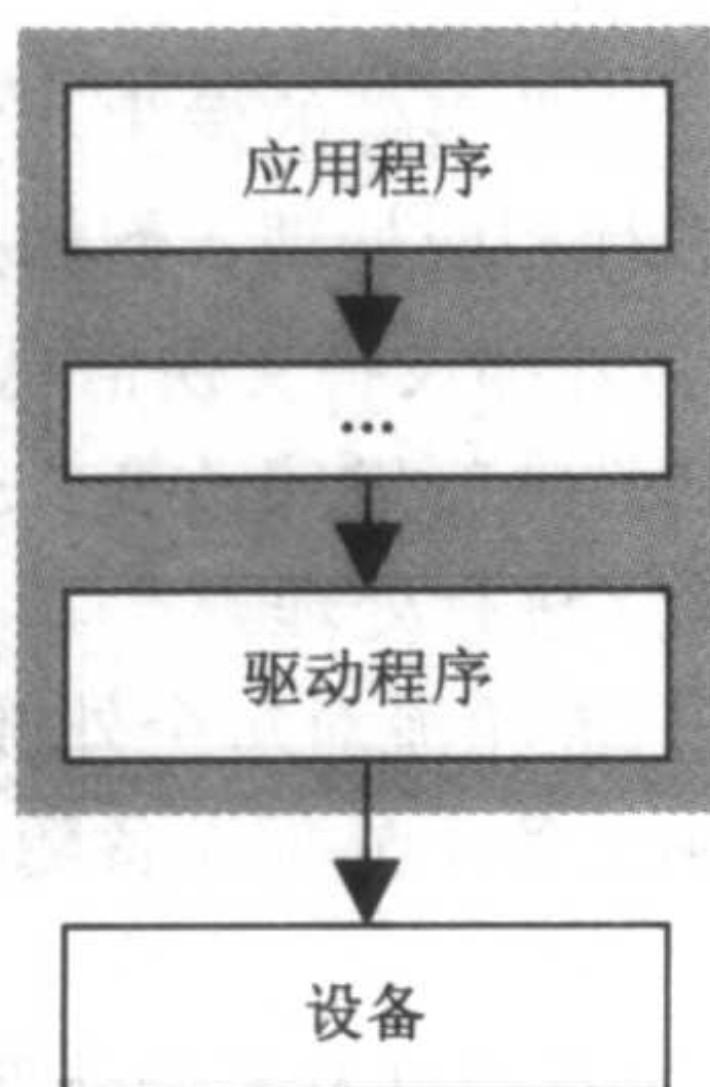


图 4.1 驱动程序直接轮询访问设备

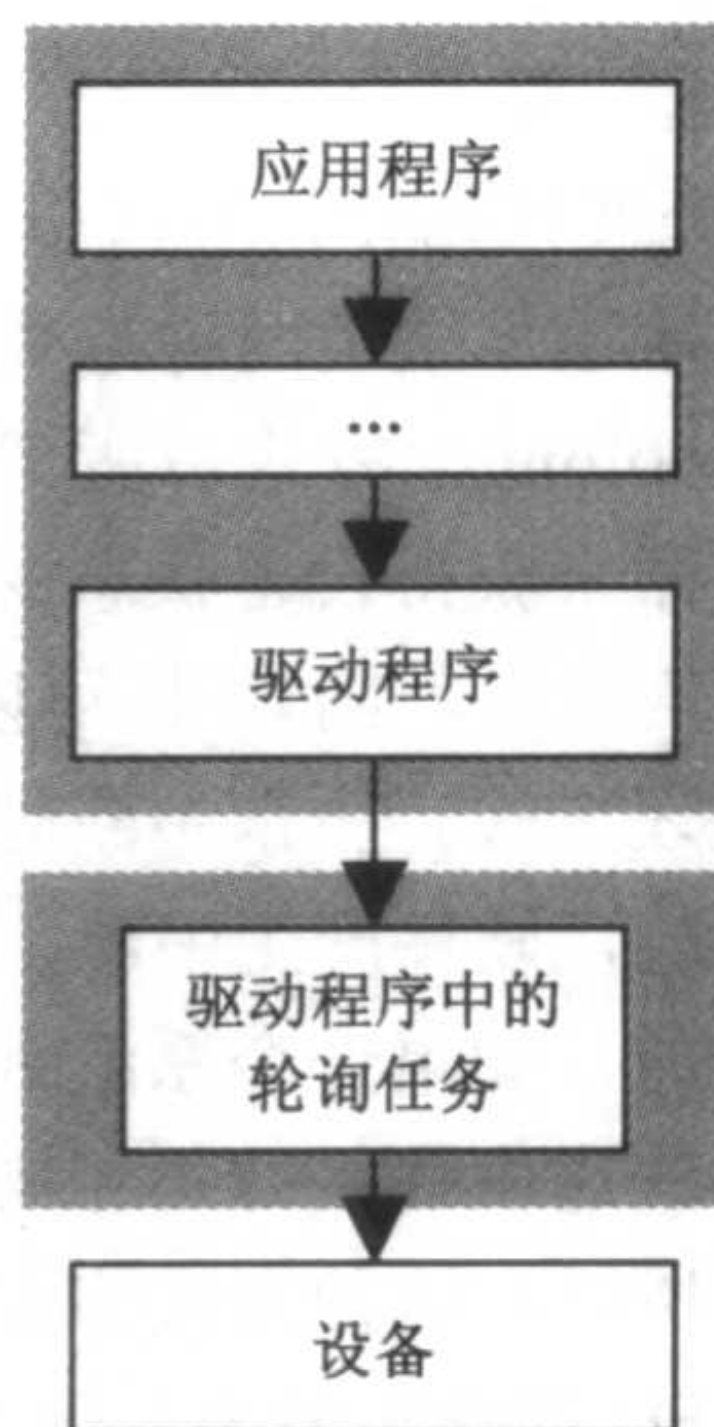


图 4.2 驱动程序中轮询任务访问设备

4.2.2 在系统空闲时轮询设备

Windows NT 中硬盘会在系统空闲时响个不停，实际上是查毒软件在检查病毒。而对于实时系统，同样可以把任务设计成系统空闲时执行。系统通过调度任务，轮询访问设备的状态并进行相应的处理。这种轮询方式需要：

- 创建一个低优先级的任务；
- 在无限循环里轮询访问设备。

轮询操作如下示例所示：

```

FOREVER /* 无限循环 */
{

```



```

status = sysInByte(ioadd + sta_reg);
if((status & SEND_OK)
{
    printf("Start send!\n");
    sysOutByte(ioaddr, *buf);
}
...
}

```

4.2.3 周期性轮询设备

对于嵌入式实时系统而言，一般很难找到合适的空闲时间来查询设备状态。同时这种空闲时间的出现也是不可确定的，将会导致轮询设备的任务处于不可预测状态。而采用周期性轮询设备就避免了这种情况，这种方式是任务按一定的时间间隔轮询设备状态。轮询设备的任务流程是查询设备状态并进行相应处理，然后调用延迟操作将自身挂起。这种轮询方式需要：

- 创建一个中或高优先级任务；
- 在无限循环里轮询设备，并调用 taskDelay()函数。

如下示例所示的轮询操作：

```

FOREVER
{
    status = sysInByte(ioadd + sta_reg);
    if((status & SEND_OK)
    {
        printf("Start send!\n");
        sysOutByte(ioaddr, *buf);
    }
    ...
    taskDelay (POLL_RATE);    /* 任务睡眠 */
}

```

4.2.4 看门狗定时器同步设备轮询任务

使用这种方法，任务轮询设备状态是通过看门狗定时器来同步进行的。系统通过看门狗定时器同步控制轮询设备的任务，轮询设备的任务只有在获得二值信号量（如果没有得到信号量则永远等待）时，才查询设备的状态，至于看门狗定时器所需要实现的操作就是释放二值信号量并定时启动自身。这种轮询方式需要：

- 创建一个轮询任务；
- 创建一个看门狗定时器；
- 创建一个二值信号量。

下面是一个应用示例：

```

LOCAL SEM_ID PollSemId;          /* 信号量 ID */
LOCAL WDOG_ID PollWdId;          /* 看门狗定时器 ID */
#define POLL_RATE 100            /* 轮询访问设备周期 */

/* 创建信号量、轮询任务、看门狗定时器的函数。 */
STATUS PollInit (void)

```



```
{
/* 创建二值信号量和轮询任务 */
PollSemId = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY);
taskSpawn (... , PollDevice,...);

/* 创建看门狗定时器并启动 */
PollWdId = wdCreate ();
wdStart (PollWdId, POLL_RATE, PollWdFunc, param);
}

/* 设备驱动看门狗定时器处理函数 */
LOCAL void PollWdFunc (int param)
{
/* 释放二值信号量 */
semGive (PollSemId);

/* 启动看门狗定时器 */
wdStart (PollWdId, POLL_RATE, PollWdFunc, param);
}

/* 轮询访问设备的任务函数 */
LOCAL void PollDevice (void)
{
/* 获得二值信号量 (如果没有得到信号量且永远等待) */
semTake (PollSemId, WAIT_FOREVER);

/* 查询设备状态并进行相应处理 */
...
status = *pDeviceStaAdr;
...
}
}
```

4.2.5 看门狗定时器调用设备轮询函数

使用这种方法, 查询设备的状态只是通过看门狗定时器调用函数来执行。看门狗定时器所需要进行的操作就是定时查询设备状态进行相应处理并启动自身。这种轮询方式需要创建一个看门狗定时器。

下面是该方法的一个应用示例:

```
#define POLL_RATE 100      /* 轮询访问设备周期 */
LOCAL WDOG_ID PollWdId;    /* 看门狗定时器 ID */

/* 创建看门狗定时器并启动 */
STATUS PollInit (void)
{
PollWdId = wdCreate();
wdStart (PollWdId, POLL_RATE, pollFunc, param);
}

/* 看门狗定时器轮询处理函数 */
LOCAL void pollFunc (int param)
{
}
```



```

/* 查询设备状态并进行相应处理 */
...
status = *pDeviceStaAdr;
...
/* 启动看门狗定时器 */ wdStart (PollWdId, POLL_RATE, pollFunc, param);
}

```

4.2.6 使用辅助时钟高速轮询设备

使用辅助时钟轮询设备的方式是一个高速轮询设备的过程。在实现上，辅助时钟产生中断，由该时钟中断服务程序来查询设备的状态。采用这种方法所要进行的操作有：注册查询设备的时钟中断服务程序、设置辅助时钟中断 tick 数、打开/关闭辅助时钟中断等。这种方式需要：

- 目标机上有辅助时钟支持；
- 查询设备的时钟中断服务程序。

其中，在 config.h 文件中设置辅助时钟 tick 数的最小 (AUX_CLK_RATE_MIN) 和最大 (AUX_CLK_RATE_MAX) 值。

下面是该方法的应用示例：

```

/* 设备中断服务程序的注册及辅助时钟 tick 数的设置 */
void AuxPollStart (int rate)
{
/* 查询设备中断服务程序与辅助时钟中断连接 */
sysAuxClkConnect (AuxClkPoll, param);

/* 设置辅助时钟中断 tick 数，并打开辅助时钟中断 */
sysAuxClkRateSet (rate);
sysAuxClkEnable ();
}

/* 查询设备中断服务程序 */
LOCAL void AuxClkPoll (int param)
{
/* 查询设备状态并进行相应处理 */
status = *pDeviceStaAdr;
...
}

/* 关闭辅助时钟中断 */
void AuxPollStop (void)
{
sysAuxClkDisable ();
}

```

4.2.7 轮询方法的比较

轮询方式访问设备，可以通过任务、看门狗定时、辅助时钟定时等方式来进行。不同的方式各有自己的优缺点，下面是几种方法的比较。

- (1) 利用一个低优先级任务轮询设备，系统需要创建该任务。
- (2) 如果任务优先级不够高，则任务处于饥饿状态。
- (3) 若任务优先级太高，任务将总是访问设备，浪费了 CPU 资源。
- (4) 一个高优先级轮询任务可能阻塞其他任务。

(5) 利用看门狗定时器可以确保周期性访问设备，但增加了系统开销。

(6) 为了避免上述情况，应当使用辅助时钟高速轮询设备方式。

4.3 中断处理

在微处理器的中断机制中，包含两类中断源：一类是外部中断源，它是通过外部设备接口向 CPU 的中断请求引脚“INTR”和“NMI”发出中断请求信号引起中断的，这类中断称为硬中断（或称外部中断）。另一类是内部中断（软中断），内部中断是通过 CPU 执行中断指令所产生的，或者是 CPU 在执行程序时遇到了一些异常情况产生的，像除法出错，溢出中断等都是产生内部中断的中断源。

在嵌入式系统中，硬件中断处理是实时系统的重要组成部分，系统通常都是通过中断与外部世界进行交互的。为了尽可能快地响应中断，VxWorks 中的中断服务程序运行在一个不同于任何任务的上下文中。因此中断的处理不涉及到任务上下文的切换。

在中断处理操作中，需要了解下面的一些知识：

- 中断服务程序（ISR）的安装；
- 中断服务程序可以进行什么操作，不可以进行什么操作；
- 中断服务程序的调试。

实际上，中断是 CPU 和外部设备进行通信的有效方法。这种通信方式一直被大多数计算机系统所采用，它可以避免因反复查询外部设备的状态而浪费 CPU 时间，从而提高了 CPU 的工作效率。

4.3.1 中断服务程序

中断服务程序是用户定义的一个程序，只有当中断产生时它才执行，该程序拥有独立于任务的上下文。

在中断服务程序里，不能执行过多的 I/O 操作，不可以获取信号量，不可以分配内存，不可以申请任务控制块。像 `printf()`、`malloc()`、`semTake()`、`taskIdSelf()` 函数都是不可以使用的，但是可以使用 `semGive()`、`logMsg()`、`msgQSend()` 以及 `bcopy()`。在 4.3.6 节中将详细讨论中断服务程序中的限制。

下面是一个中断服务程序的简单例子：

```
LOCAL SEM_ID SemId; /* 信号量 ID */
...
LOCAL void ISRFunc (void)
{
    /* 释放二值信号量 */
    semGive (SemId);
}
```

用户在使用中断服务程序之前，必须采用一定的方法，将中断服务程序的入口地址放置在中断向量表上与中断类型号相应的位置中。当中断到来时，CPU 会自动地取出中断向量表中相应的内容送入 IP 寄存器和 CS 寄存器，从而改变程序执行的方向，转入中断服务程序的执行。

4.3.2 处理中断

事实中的中断处理并不是简简单单地执行 C 语言函数，它需要保存当前系统寄存器状态以及正在执行的代码，然后装载中断堆栈，最后调用连接在中断向量上的中断服务程序。中断处理完成后，

从中断服务程序函数返回时，系统需要恢复寄存器和堆栈，并退出中断。这个中断处理的过程如下：

- （1）当设备产生 IRQ（中断请求）时，中断当前任务并取中断类型号；
- （2）在中断应答期间，系统决定哪个设备需要服务（计算中断向量地址）；
- （3）为设备调用适当的函数（系统注册的设备中断服务程序）；
- （4）在 VxWorks 定义的中断包裹函数里，保存寄存器值及调用用户定义的中断服务程序；
- （5）中断处理结束后，从中断服务程序返回并恢复寄存器值；
- （6）如果执行中断服务程序后，没有高优先级的任务运行，则恢复被中断任务的执行。

上述处理中断过程，可以通过图 4.3 所示更形象地描述中断处理过程。

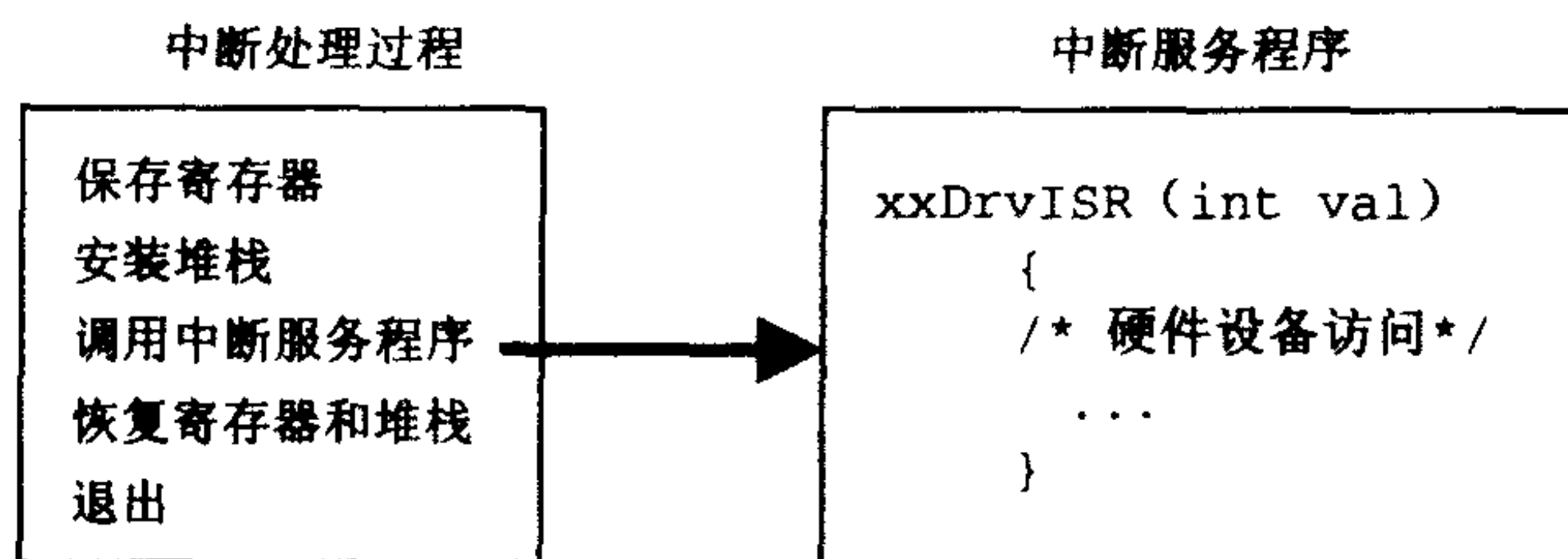


图 4.3 中断处理过程

注意

VxWorks 定义的中断包裹函数并不自动进行保存和恢复浮点寄存器操作。

4.3.3 决定中断优先级

当系统中有多个设备用中断方式和 CPU 进行数据传输时，就有一个中断优先级处理问题。通常在嵌入式系统对中断优先级采用以下三种方法来解决：

（1）软件查询方法。当硬件系统无法确定中断处理优先级时，可以通过软件来决定硬件设备的中断优先级。这种方法的优点是节省硬件，不需要由判断优先级的硬件排队电路，而是用程序的优先级来确定设备中断的优先级。这种方法的缺点是设备发出中断请求信号到 CPU 转入相应的服务程序入口的时间比较长。

（2）简单的硬件方法——菊花链法。这是解决中断优先级的一个简单硬件方法。其做法是在每个外设对应的接口上连接一个逻辑电路，这些逻辑电路构成一个称为菊花链的结构。由菊花链来控制中断回答信号的通路。有了菊花链以后，各个外设接口就不会竞争中断回答信号，因为菊花链已经从硬件的角度根据设备在链中的位置决定了它们的优先级，越靠近 CPU 的设备，优先级越高。

（3）专用硬件方法。在嵌入式系统中，解决中断优先级管理的常用办法是采用可编程的中断控制器。可编程中断控制器中的中断类型寄存器、屏蔽寄存器都是可编程的，而且优先级排列也是按指令来设置的，所以可编程控制器使用起来很灵活方便。

4.3.4 中断号与中断向量的转换

嵌入式系统给每种中断都安排一个中断类型号（通常称中断号）。每种类型的中断都由相应的中

断服务程序来处理，而中断向量表用来存放中断服务程序的入口地址。

例如在 PC 系统中，存储器的低 1.5KB 字节，即地址从 0~5FFH 为系统所占用，其中最低的 1KB 字节，地址从 0~3FFH 存放中断向量表。中断向量表中的 256 项中断向量对应 256 种中断类型，每项占四个字节（如图 4.4 所示的 x86 体系结构的中断向量表），所以每类中断向量的地址可以由中断类型号乘以 4 计算处理。

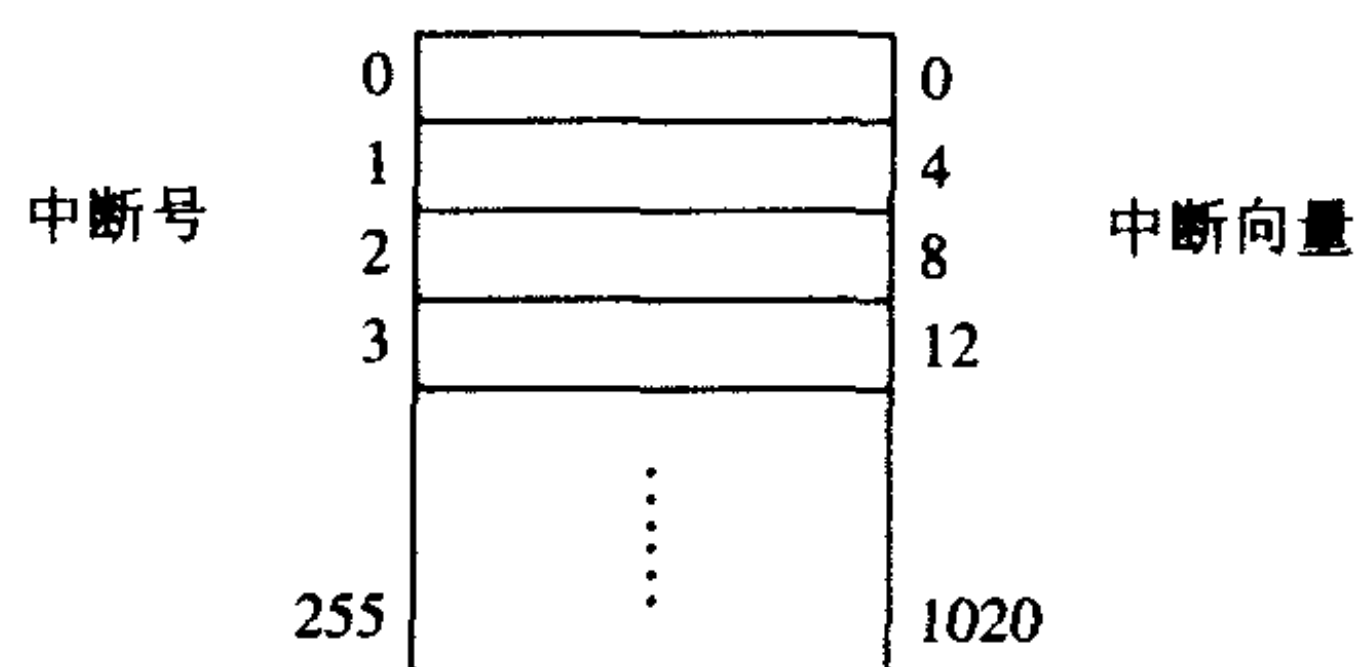


图 4.4 x86 体系结构的中断向量表

在 VxWorks 系统中，中断号与中断向量地址通过函数 INUM_TO_IVEC (intNum) 来转换。示例如下：

```
UINT32 uiIntVec;          /* 中断向量 */
BYTE irqNum = 0x9;        /* 中断号 */
UiIntVec = (UINT32)(INUM_TO_IVEC(irqNum)); /* 转换操作 */
```

采用中断向量的方法，大大加快了中断处理的速度，因为计算机可直接通过中断向量表跳转到相应的中断服务程序，而不需要 CPU 去逐个检测和确定中断原因。

4.3.5 安装中断服务程序

用户可以用指定的程序代替 VxWorks 中的中断服务程序。VxWorks 提供了 intConnect() 函数实现这个功能，通过这个函数可以将用户函数与中断向量相关联。该函数的参数为：需要与服务程序关联的中断向量、需要连接的 C 函数的地址以及传递给该函数的一个参数。对应的中断源产生中断时，系统就会使用指定的参数执行这个被连接到中断的函数。当中断处理完成后，这个被连接到中断的函数返回。使用这种方法连接到中断向量上的 C 函数就是中断服务程序。

安装中断服务程序的 intConnect() 函数原型如下：

```
STATUS intConnect (VOIDFUNCPTR *vector, VOIDFUNCPTR routine, int parameter);
```

下面是一个 intconnect() 的示例程序。

```
=====
setupINT_demo.c——中断注册演示程序
=====
#include <iv.h>
...

/* 用户定义的中断服务程序 */
void BoardISR(void)
{
    /* 用户中断处理操作 */
    ...
}
```



```

/* 安装用户中断程序 */
STATUS xxISRinstall(void)
{
    UINT32 uiIntVec;          /* 中断向量 */
    BYTE irqNum = 0x9;        /* 中断号 */

    /* 根据中断号获得中断向量 */
    UiIntVec = (UINT32)(INUM_TO_IVEC(irqNum));

    /* 安装中断服务程序 */
    if(intConnect((VOIDFUNCPTR*)uiIntVec, (VOIDFUNCPTR) BoardISR, 0) != OK)
        return(ERROR);
    ...
    return(OK);
}

```

注意

vector 参数不是中断类型号，它是与中断类型号关联的中断向量表起始位置的偏移地址。Parameter 参数是传递给中断服务程序的一个参数。例如当系统有多块同类型的设备时，通过这个参数可以指出安装哪块设备的中断服务程序，从而避免编写两个一样的中断服务程序。使用示例如下：

```

/* 安装第一块板子的中断服务程序 */
intConnect (INUM_TO_IVEC(intNumberBoard0), BoardISR, 0);
/* 安装第二块板子的中断服务程序 */
intConnect (INUM_TO_IVEC(intNumberBoard1), BoardISR, 1);
...
void BoardISR (int boardNumber)
...

```

4.3.6 中断服务程序的约束操作

许多 VxWorks 函数在中断服务程序中是禁止使用的。引起这些限制是因为中断服务程序不在一个固定的任务上下文中执行，而且没有任务控制块，所有中断服务程序必须共享一个单独的堆栈。由于这些原因，中断服务程序中禁止使用会引起阻塞的函数。假设在中断服务程序中试图访问一个 I/O 设备，但是这个 I/O 设备却不可用，系统内核将把调用者切换到挂起状态，这样将引起中断的挂起，而中断是没有固定的上下文的，从而导致系统崩溃。但是中断服务程序可以使用那些不会引起阻塞的操作。

在 VxWorks 中存储管理函数，malloc()和 free()都要求获得信号量，而获取信号量的操作将会引起阻塞，所以在中断服务程序中不可以使用 malloc()和 free()函数，也不能使用其他调用 malloc()和 free()的函数。这个限制将使中断服务程序不能调用任何用于创建或删除的函数。

在默认的情况下，中断服务程序的另一个限制是不允许使用调用浮点协处理器的函数。这是因为在 VxWorks 中，intConnect()建立的中断服务程序不能保存并恢复浮点寄存器。如果一定要在中断服务程序中使用浮点运算的话，必须使用 fppArchLib 库中的函数显式的保存和恢复浮点协处理器的寄存器。

如上所述，这个服务程序不能执行 I/O 操作，虽然在 I/O 系统里没有内在的约束，但是大多数

的 I/O 设备驱动程序将会阻塞等待设备的调用者，这将使用到任务上下文，而中断服务程序没有固定的上下文，最终将会导致系统的崩溃。

总之，中断服务程序的约束主要在下面几个方面：

(1) 不可以调用 I/O 系统调用。不可以调用 I/O 系统调用函数（例如 `printf()`）是因为 I/O 设备驱动程序会阻塞等待设备的调用者。

(2) 不可以进行阻塞操作。不可以获得信号量操作或调用任何可能需要获得信号量的函数，例如 `semTake()`、`free()`、`malloc()`。同时，也不允许调用 `taskDelay()` 函数。

(3) 不可以做任何请求任务上下文的操作。不可以进行任何请求任务上下文操作，包括不能够使用 0 作为任务 ID（例如 `taskSuspend(0)`），不能够使用互斥信号量（因为其中使用了 `taskSafe()` 和 `taskUnsafe()` 机制）。

上述列出的一些限制希望读者有所注意，这些内容看似简单，却容易在实际工作中忽略。例如在中断服务程序中调用 `printf()` 函数，系统执行这个中断服务程序就会引起死机。为了方便中断服务程序输出信息，VxWorks 提供了 `logMsg()` 函数，通过该函数可以在中断服务程序中显示信息。如下所示：

```
logMsg ("muxLoad succeeded!\n", 0, 0, 0, 0, 0, 0);
```



注意

I/O 系统调用还包括 `read()` 和 `write()` 等，通常 I/O 设备驱动程序会阻塞等待设备的调用者，直到这个设备满足请求的操作。

4.3.7 中断服务程序可以进行的操作

由于中断服务程序的一些限制，VxWorks 专门为中断服务程序设计了一些其他机制，例如管道设备，在中断服务程序中可以执行管道设备的写操作。

为了给中断服务程序提供输出功能，VxWorks 中提供了 `logLib` 库，这个库中的函数允许系统任务打印文本信息，它是中断服务程序中最常用的输出手段。

为了在中断服务程序中使用浮点运算，可以使用 `fppArchLib` 库中的函数显式地保存和恢复浮点协处理器的寄存器。

除了上一节提到的限制之外，所有其他的 VxWorks 库都可以使用。全局变量 `errno` 也会在保存和恢复时同时也被保护起来。

通过分析和总结，中断服务程序可以进行的操作主要在下面几个方面：

- 对于二值或计数信号量可以调用 `semGive()` 或 `semFlush()` 操作；
- 对内存进行读、写操作，其中包括映射内存的访问；
- 非阻塞操作的管道或消息队列的写操作；
- 调用 `logMsg()`，在中断服务程序中显示有关的错误信息；
- 调用任何 `lslLib` 或 `bLib` 库提供的函数；
- 调用除 `rngCreate()` 函数外的其他所有 `rngLib` 库提供的函数；
- 调用 `wdStart()` 和 `wdCancel()` 函数；

- 调用 taskSuspend()、taskResume()或 taskPrioritySet()函数。

4.3.8 中断服务程序的规范

中断服务程序主要处理一些高优先级的操作，不可以在中断服务程序里进行等待、大量的运算等操作。在上面的小节里明确描述了在中断服务程序里可以进行、不可以进行的操作。下面为读者给出一些如何编写中断服务程序的规范，以便在项目开发中能够起到帮助作用。

- (1) 保持中断服务程序精简（无关紧要的操作尽量放在中断服务程序之外）。
- (2) 通常的规则是“尽可能的少做”。
- (3) 中断服务程序过长将耽误低或同等优先级的中断响应。
- (4) 在中断服务程序里避免进行浮点操作。
- (5) 中断服务程序越短越容易调试。
- (6) 如果中断复用，中断服务程序应判断相应设备是否产生中断。

注意

如果需要在中断服务程序里进行浮点操作，必须使用 fppArchLib 中的函数来保存和恢复浮点协处理器的寄存器。例如 fppSave()、fppRestore()函数。

4.3.9 中断的打开和关闭

在带有外部、共享总线的系统中，必须对总线桥芯片进行编程处理，从而使中断经过总线传递到微处理器。

例如 VME 总线的桥芯片具有这个能力。对于使用 VME 总线的目标机，BSP 提供了两个标准函数用于控制 VME 总线中断，即打开总线中断函数 sysIntEnable()和关闭总线中断函数 sysIntDisable()。

在一个 VME 总线上，有 7 个级别的中断，即 1~7。可以通过上面两个总线中断管理函数指定打开或关闭哪个中断。在所有 BSP（板级支持包）中，并不是所有的中断都可以使用。有一些 BSP 是多路 VME 总线中断，例如 VME 总线中断级别 3 和级别 4 可能被复叠成中断级别 3。有一些 BSP 需要人为处理某些中断级别的中断响应周期，这需要阅读有关目标机特殊的文档资料。也有一些目标机可能需要一个硬件跳转转变。

下面是一个初始化同步 VME 总线中断的示例。

```
=====
InitVME_demo.c——VEM 总线中断初始化演示程序
=====
#define INTLEVEL 1 /* 定义总线中断级别 */
LOCAL void VMEintInit (void)
{
    /* 为同步操作安装信号量 */
    if ((semId = semBCreate (SEM_Q_PRIORITY, SEM_EMPTY)) == NULL)
    {
        perror ("intInit: semBCreate failed");
        return;
    }

    /* 打开 VME 总线中断 interrupts */
}
```



```

    if (sysIntEnable (INTLEVEL) == ERROR)
    {
        printf ("intInit: Error in enabling interrupts\n");
        return;
    }

    /* 安装中断服务程序, 并使信号量与硬件中断关联 */
    #if (CPU_FAMILY == I960) /* 判断 CPU 类型 */
    if (intConnect ((VOIDFUNCPTR *) INUM_TO_IVEC (INTNUM),
                    intHdlr, semId) == ERROR)
    {
        perror ("intInit: Error in connecting to the ISR");
        return;
    }
    #endif

    #if (CPU_FAMILY == MC680X0)
    if (intConnect ((VOIDFUNCPTR *) INUM_TO_IVEC (INTNUM),
                    (VOIDFUNCPTR)semGive, (int)semId) == ERROR)
    {
        perror ("intInit: Error in connecting to the ISR");
        return;
    }
    #endif

    #if (CPU_FAMILY == SPARC)
    if (intConnect ((VOIDFUNCPTR *) INUM_TO_IVEC (INT_NUM_INTSYNC),
                    semGive, semId) == ERROR)
    {
        perror ("intInit: Error in connecting to the ISR");
        return;
    }
    #endif
}
-----

```

4.3.10 调试中断服务程序

中断服务程序的调试不像应用程序的调试, 当调试应用程序时, 可以采用任务级或系统级调试。而中断, 特别是外部硬件中断, 它是由外部事件触发的。换句话说, 外部中断在调试时是不受控制的。所以, 对于中断服务程序的调试比调试应用程序要麻烦得多。下面给出了调试中断服务程序的一些经验。

(1) 只要可能, 应当从简单开始, 采取逐步调试的过程。实际上, 对于中断服务程序的调试, 可以一次打开一个中断, 然后逐个调试相应的中断服务程序。并且在中断服务程序调试前期, 应当使中断服务程序的功能非常简单, 然后逐渐增加中断服务程序需要完成的功能。

(2) 用 logMsg() 函数把信息显示到控制台上。在中断服务程序规范中, 已描述过不能调用 printf() 函数打印信息, 但可以调用 logMsg() 函数显示信息。

(3) 使用 sprintf() 函数把日志数据写到内存中。在中断服务程序中, 不可以调用 printf() 函数, 需要显示可以调用 logMsg() 函数, 而记录则可以调用 sprintf() 函数。

sprintf() 的使用示例如下:

```

Char * ptr1;
...

```



```
ptr1 = (char *)malloc(SIZE_OF_BUFFER);
...
void demoISR (void)
{
    ...
    logMsg("\n Begin to log data!\n",0,0,0,0,0,0);
    ptr1 = ptr1 + sprintf(ptr1, "%d", data);
}
```

(4) 把数据写到内存特定区域中。

把数据写到目标机内存某个特定区域，在系统重新启动后数据也不会被清除。当调试中断时，可以采用这种方式保存大量有用的信息。

注意

通常的做法是在中断向量（典型地址范围 0~0x400）表末尾地址和 vxWork（典型装载地址在 0x1000）开始地址之间查找存放数据的位置。例如从 0x400~0x1000 地址之间。也可以通过定义用户内存池保存信息。

4.4 其他设计需要考虑的事项

在设备驱动程序实际开发过程中，编写设备驱动程序前应该从宏观上对设备驱动程序的设计进行考虑，这些事项包括驱动轮询任务的同步问题、数据缓冲问题等。在这一章中对这些问题进行了描述，请读者在驱动程序的设计过程中注意这些问题，避免设计中“潜伏”的问题，以至于影响驱动程序的开发工作。

4.4.1 任务同步问题

当应用任务访问设备时，如果无论系统设备状态是否改变都对设备进行轮询操作，这种设计必然会增加系统开销，降低系统整体性能。实际上可以采取同步设计解决这类问题。

4.4.1.1 应用任务与设备驱动任务的同步

任务间的同步设计如图 4.5 所示的应用任务与设备驱动任务同步设计。

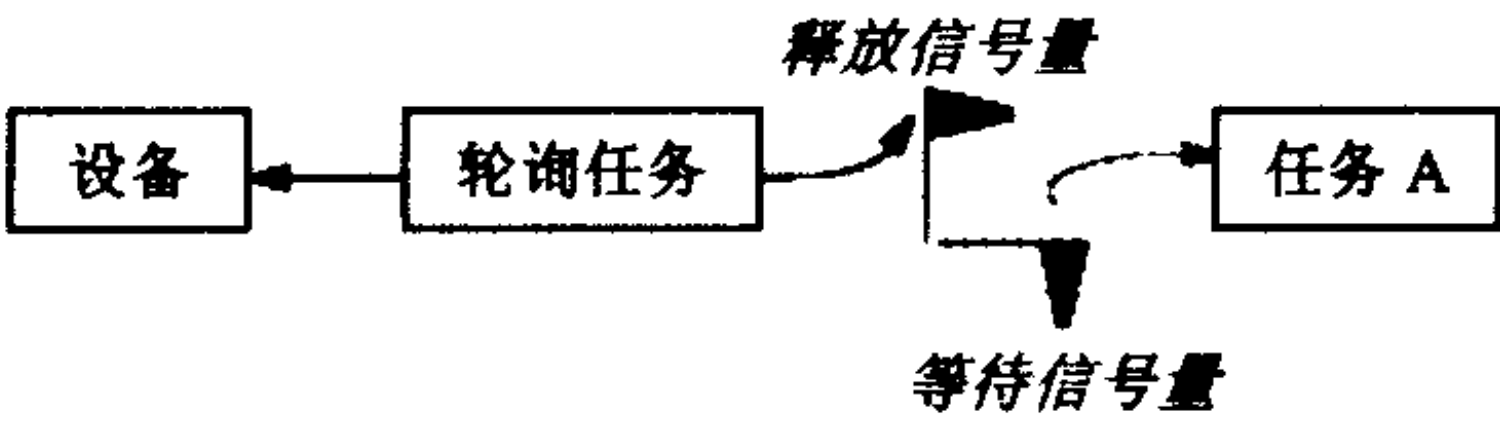


图 4.5 应用任务与设备驱动任务同步设计

这种设计的具体操作过程如下。

- (1) 应用任务（任务 A）通过调用 semTake()函数，因等待信号量而处于阻塞状态，直到设备发生一个事件。
- (2) 当驱动程序的轮询设备任务探测到这个事件时，调用 semGive()函数释放信号量。
- (3) 应用任务（任务 A）从 semTake()中返回。

在系统设备中所发生的“事件”主要指如下信息：

- 设备某个状态的改变；
- 设备接收到数据；
- 设备处于可写状态；
- 错误状态。

注意

如果多个应用任务等待设备状态的发生，在驱动程序的轮询任务中可以使用 `semFlush()` 函数替代 `semGive()` 函数调用。一个演示程序如下。

```
=====
graphControl.c——图形管理任务
=====
#include "vxWorks.h"
...
SEM_ID      nodeSyncSem;      /* 用于节点的同步信号量 */
SEM_ID      graphSem;         /* 用于图形的互斥信号量 */
void graphControl (void)
{
    int      ix;               /* 节点索引 */
    /* 同步二值信号量避免节点不断继续运行 */
    nodeSyncSem = semBCreate (0, SEM_EMPTY);
    /* 二值信号量提供图形的互斥保护 */
    graphSem = semBCreate (0, SEM_FULL);
    FOREVER
    {
        /* 访问数据库临界资源 */
        semTake (graphSem, -1);
        /* 解锁所有等待信号量 nodeSyncSem 的任务 */
        semFlush (nodeSyncSem);
        ...
        /* 任务延迟操作 */
        taskDelay (STEP_TICKS);
        /* 恢复数据库访问 */
        semGive (graphSem);
    }
}
```

4.4.1.2 驱动程序任务与设备中断服务程序的同步

在访问设备的同步问题上，可以采用应用任务与驱动程序任务之间的同步操作。同时，也可以采用驱动程序任务与设备中断服务程序的同步设计，来实现同步操作，如图 4.6 所示。



图 4.6 任务与设备中断服务程序同步设计

这种设计的具体操作过程如下。

- (1) 驱动程序任务执行 `semTake()` 操作, 当没有获得信号量时任务处于阻塞状态。
- (2) 当设备接收到数据时, 产生中断。
- (3) 中断服务程序调用 `semGive()` 函数, 释放信号量。
- (4) 驱动程序任务获得信号量并从 `semTake()` 中返回。

4.4.2 数据缓冲问题

4.4.2.1 设备驱动任务与应用任务的数据缓冲

任务间的数据缓冲如图 4.7 所示。



图 4.7 应用任务与设备任务数据缓冲作用

从图 4.7 中可以发现, 任务之间通过一个共享缓冲区进行数据的发送和接收。这种设计在总体上提高了系统的性能。对这种设计进行如下描述。

1. 数据接收

数据的接收主要经过如下操作。

(1) 设备驱动程序中轮询任务的数据存放操作。当设备上有数据到来时, 轮询任务读取数据并把它存放到共享内存、环行缓冲区、链表、管道或消息队列中。

(2) 应用任务的数据读取操作。协同操作任务 (应用任务) 从共享缓冲区中读取设备轮询任务所存放的数据。

2. 数据发送

数据发送处理与数据接收处理相反, 操作过程如下。

(1) 应用任务层数据的发送。这个处理过程中, 首先的操作是查看共享缓冲区是否有空间存放要发送的数据。如果有空间, 则把要发送的数据写到缓冲区。反之, 应用任务通常因等待缓冲区而被阻塞。

(2) 设备驱动层数据的发送。轮询任务查看设备状态, 当设备的发送寄存器或内存为空时, 轮询任务从共享缓冲区中把数据“拷贝”给设备。

3. 管道或消息队列的采用

在共享缓冲区问题上, 系统通常采用管道或消息队列作为数据发送和接收的中转站。因为它们本身具有一些很明显的优势。

(1) 同步优势。管道或消息队列提供任务间的同步操作。当管道或消息队列中无数据时, 则任务处于悬置态, 一旦有数据任务马上处于就绪态。

(2) 防止优先级反转。管道或消息队列具备优先级继承功能, 采用优先级继承方法, 解决了任务因等待共享资源而发生优先级反转问题。

注意

只有当创建管道或消息队列的选项参数为按优先级排列时, 优先级继承才起效。如果按先进先出排列时, 则防止优先级反转策略视为无效。

4. 数据缓冲注意事项

如果采用共享内存、环行缓冲区或链表，驱动程序必须注意如下事项：

- (1) 如果有必要的话，确保临界资源（如共享内存）互斥现象。
- (2) 自身提供任务的同步操作。

4.4.2.2 应用任务与设备中断服务程序数据缓冲

应用任务与设备中断服务程序的数据缓冲如图 4.8 所示。

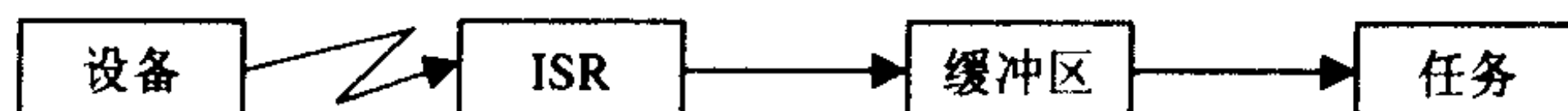


图 4.8 应用任务与设备中断服务程序数据缓冲

从图 4.8 中可以发现，应用任务与中断服务程序之间也是通过一个共享缓冲区进行数据的发送和接收。对于这种设计它的流程主要如下：

- (1) 中断产生。当设备上有数据到来时，设备产生一个中断。
- (2) 中断服务程序的数据读取操作。在中断服务程序里，从设备上读取数据并把它存放到共享环行缓冲区、管道或消息队列中。
- (3) 应用任务数据读取操作。应用任务读取设备所接收到的数据实际上是从缓冲区获得数据。



注意

管道和消息队列具有提供任务同步和防止优先级反转的功能，而共享内存或环行缓冲区不具备这两个功能。所以推荐用户使用管道或消息队列来传递数据。

4.4.3 设计问题

1. 速度与健壮性

(1) 确定设计的选择：

- 是否有多 I/O 板，是否总是使用相同跳线设置；
- 是否总使用同一设备连接，事件发生后是否要求操作。

(2) 所确定的因素越多，开发驱动就越容易。

(3) 设备本身可能限定了一个性能级别（例如时钟限制）。

(4) 当拿不定主意时，不要对系统进行假设。

2. “事件”处理

(1) 典型的是驱动程序捕捉处理事件，由应用程序处理。

(2) 有时，“事件”的组成并非显而易见，这就需要驱动程序将其分解。

(3) 尽管不尽相同，但典型设计需要同步和数据通信机制。

总之，同步和数据通信机制的选择取决于速度与驱动程序健壮方面的考虑。例如使用信号量作为同步机制，虽然处理上非常快，但是并不提供任何数据的传送。使用消息队列，则既提供了同步机制也提供了数据通信机制。至于管道，它提供了消息队列所不具备的优势，那就是标准 I/O 系统接口功能（应用程序调用标准 I/O 函数访问管道）。

4.4.4 设备驱动程序注意事项

1. 互斥操作

当系统对临界资源进行访问时，应对该资源采取保护措施，其保护方式可以通过互斥操作来实现。有关互斥操作的描述如下：

(1) 多个任务同时修改同一数据结构时必须使用互斥操作。当多个任务对同一数据结构进行访问时，应该对该数据区采取保护措施，否则会引起混乱。例如有一个任务正在读取共享数据区的数据时，而另一个任务却要往这片数据区写数据，显然读取的数据可能会发生错误。采用互斥操作可以使共享数据区得到很好保护，即同一时刻有且仅有一个任务在访问共享数据区。

(2) 多个任务试图同时对设备进行访问时必须使用互斥操作。当多个任务访问同一设备时，应该对该设备进行保护，否则将引起系统混乱。

(3) 通过互斥信号量可以实现互斥操作。VxWorks 提供了互斥信号量功能，通过该功能只有获得该信号量的任务才可以对设备或共享数据区进行读写等操作。

2. 竞态条件

所谓竞态条件是一种状态，在这种状态下两个实体（例如两个处理任务）对同一资源进行竞争时，系统没有一种机制来决定把资源首先分配给哪一个实体，因此由于系统不能保证数据的正确处理，其结果是不可预测的。

可以通过如下方式避免竞态条件现象的发生。

(1) 通过互斥信号量或任务锁（`taskLock()`）可以避免竞态条件的发生。互斥信号量的功能决定只有获得该信号量的任务才可以进行数据处理，而任务锁使得数据在处理时不被其他任务打断。

(2) 通过对驱动程序的测试和设置避免竞态条件的发生。当通过一个可抢占的任务改变测试环境时，可能需要重新对驱动程序进行测试和一些参数的设置。

注意

在驱动程序总体设计上，应设法调整设计，避免任务/中断冲突。

4.4.5 任务锁和中断锁

1. 任务锁

在系统默认情况下，VxWorks 的任务调度策略采用的是优先级抢占方式，即高优先级的任务可以打断低优先级任务的执行。然而，由于应用系统从安全性、可靠性出发，对某些资源（例如共享变量或共享数组等）进行读写时需要保护该资源，保证任务对这个资源进行读写时，任何其他任务都不能打断正在对这个资源进行读写的任务，这种现象就是所谓的禁止任务抢占调度。VxWorks 禁止任务抢占调度的设计是通过任务锁——`taskLock()`函数来实现，用户可以根据自己的需要对任务里的任何一段临界区域进行任务锁操作。

VxWorks 的任务锁具有如下特性。

(1) `taskLock()`函数禁止任务抢占调度，直到调用 `taskUnlock()`函数之后。

(2) 不能屏蔽中断，即对中断不起作用。

(3) 调用多少个 `taskLock()`操作，就要调用相同个数的 `taskUnlock()`操作，否则任务可抢占调度

依然被禁止。

(4) 执行任何的操作都会发生阻塞现象，直到开启任务可抢占功能。

下面是一个禁止任务抢占调度的示例。

```
=====
taskLock_demo.c——任务锁演示程序
=====
void taskA(void)
{
    int count=0;
    ...
    FOREVER
    {
        ...
        taskLock();
        count ++; /* 临界资源:禁止任务抢占 */
        taskUnlock();
        ...
    }
}
-----
```

2. 中断锁

对于任务，可以通过调用 taskLock()函数禁止任务抢占调度操作。然而该操作对中断不起作用，也就是说中断依然可以打断任务的执行。可是在系统中，当任务对临界资源进行操作时，要求任何中断都不能打断任务对这个资源的操作，这就要求系统提供锁中断功能。VxWorks 锁中断的设计是通过中断锁——intLock()函数来实现，可以根据自己的需要对任务里的任何一段临界区域进行中断锁操作。

VxWorks 的中断锁具有如下特性。

- (1) intLock()函数禁止中断，直到调用 intUnlock()函数之后。
- (2) 调用 intLock()函数将返回一个整数，而这个整数必须传递给 intUnlock()函数。
- (3) 不能禁止任务抢占，即对高优先级的任务不起作用。
- (4) 在中断时可以被调用。

下面是一个禁止中断的示例。

```
=====
intLock_demo.c——中断锁演示程序
=====
void taskFunc(void)
{
    int level=0, count=0;
    ...
    FOREVER
    {
        ...
        level = intLock();
        count ++; /* 临界资源:禁止中断 */
        intUnlock(level);
        ...
    }
}
-----
```


在 VxWorks 系统中,上下文切换的时候进行开中断操作。用户可以调用 `intLock()` 函数屏蔽中断,但并不能提升中断的级别。如果用户需要提升中断的级别,则需要调用 VxWorks 提供的 `intLevelSet()` 函数。通过该函数,用户可以根据自己的需要设置中断级别。

小 结

这一章中描述了驱动程序采用轮询和中断方式获得设备状态等事件的发生。在内容上,对于轮询方式访问设备可能采用的方法进行了介绍;对于中断方式,介绍了中断服务程序的安装、编写中断服务程序的约束操作、中断服务程序的规范以及中断服务程序的调试。最后还对驱动程序设计时需要考虑的问题进行了描述,例如同步问题、数据缓冲问题、设计问题和驱动程序注意事项。

第 5 章 VxWorks 下编写设备驱动程序的方法

实际上，经过前面的讨论，读者已经可以开始学习具体的驱动程序开发了。但是在开发过程中可能会遇到这样或者那样的问题，为了帮助读者在设备驱动程序开发过程中避免这些问题，这里将介绍一些驱动开发过程中需要注意的细节。根据这些内容，读者也可以解决开发过程中遇到的一些其他问题，从而高效率地完成工作。

5.1 概述

在 VxWorks 下开发设备驱动程序需要掌握一些必备的知识，包括了解设备硬件有关接口、软件访问物理设备的方法、开发设备驱动程序的基本过程、设备驱动程序设计中扩展性和可读性问题以及开发环境和产品运行的环境等问题。

对于硬件和软件问题，需要了解硬件的基本组成部分、工作的流程；对于总线访问问题，需要知道通过存储映射访问还是端口访问设备等，而软件方面需要了解对设备进行初始化操作和读、写、控制等操作的方法。

对于设备驱动程序的设计需要编写设计文档、描述设备的特性、保证代码设计的可维护性和扩展性，编程的一些规范化、驱动程序库的提供等。至于环境问题主要包括宏定义、错误处理等。

总之，掌握一些驱动程序开发方法，对驱动程序开发者来说可以提高工作效率，减少重复工作。

5.2 开发前资料的收集

在开发任何设备驱动程序前，准备工作是不能省的。这部分工作主要为开发者在驱动程序开发过程中需要的资源提供保证。

1. 设备的硬件说明手册

这本手册一般包含设备的组成部分、实现功能、设备特点等说明。这些说明对开发者了解设备的工作流程有所帮助。例如数据的发送、接收工作流程等。

2. 设备的软件开发手册

这本书主要包括设备内部数据寄存器、状态寄存器、控制寄存器等各寄存器说明。简单地说这本书就是把设备的组成抽象成驱动开发程序员所关心的“软件”（例如各寄存器地址分配）。这本书对设备驱动程序开发至关重要，是开发设备驱动程序的基础。

3. 设备驱动程序源代码模板

设备上市后，硬件开发厂商通常会公布该设备所支持的操作系统，有些设备驱动程序还会公开源代码。这样可以基于现有的资源开发符合自己的设备驱动程序，没有必要从零开始编写。如果没有，也应尽量找类似的设备驱动程序源代码做参考。

5.3 宏定义及 C 语言可以调用的汇编函数

在设备驱动程序中宏定义和汇编函数的调用是常见的问题，因为宏定义使程序灵活性得到了发挥，而汇编函数调用提高了驱动程序执行效率。

5.3.1 驱动程序宏定义

写出一个好的设备驱动程序并不是一件容易的事情，功能实现是基础，高效率和兼容性也要考虑。

将端口地址、寄存器位定义为宏、系统函数调用时通过指针获得函数地址及需要的结构，这些都是保证程序高效而又兼容的基本操作。下面进行实例说明。

1. 兼容性设计

下面是一个宏定义的兼容性设计示例。

(1) 一个好的设计，代码如下所示：

```
#ifndef INCLUDE_GO
# define THE_GO myGo
#else
# define THE_GO yourGo
#endif
fooReset (&THE_GO, arg1, arg2, arg3);
```

(2) 一个让人费解的设计，代码如下所示：

```
#ifndef INCLUDE_GO
fooReset (&myGo,
#else
fooReset (&yourGo,
#endif
arg1, arg2, arg3);
```

后者的设计看似精简，但是它的可读性和维护性很差，增加了后期维护的工作量、不易理解。

2. 高效代码设计

结构在代码编译后固定了所需的分配空间而且是不可精简的，但是通过高效的设计可以减少代码占用的空间。下面是一个设备驱动程序发送函数示例。

在设备驱动程序头文件中定义如下：

```
typedef struct { /* I16552_DEV */
char CSR;
char DATA;
char MBR;
} I16552_DEV;
```

在 BSP 头文件中定义如下：

```
#define I16552_BASE 0x....
```

在 BSP 初始化中代码如下：

```
I16552 *port = I16552_BASE;
```

在发送函数中使用如下语句：

```
port->DATA = ...;
```


实际上下面的设计才是高效的。

在驱动程序头文件中定义如下：

```
#define I16552_ADRS(reg) \
(unsigned char *) (I16552_BASE + (reg) * I16552_OFFSET)
#define I16552_CSR I16552_ADRS(0)
#define I16552_DATA I16552_ADRS(1)
#define I16552_MBR I16552_ADRS(2)
```

在 BSP 头文件中定义如下：

```
#define I16552_BASE ...
#define I16552_OFFSET 1
```

在发送函数中使用如下语句：

```
*I16552_DATA = ...;
```

从上述两者设计中可以发现，后者设计更能减少系统的开销，从而达到高效设计的目的。

5.3.2 C 语言调用汇编函数

C 语言仅仅支持标准存储器直接访问功能，对设备驱动程序端口操作需要使用特殊的汇编指令。如果在设备驱动程序中需要访问端口，直接嵌入汇编语言是一种方式，但一般更倾向于编写 C 语言能调用的汇编函数。下面就这个问题进行说明。


```

/* 发送 st 缓冲区的字符 */
while(*st)
{
    status = sysOutByte(port,*st);
    if(status==ERROR)
        printf("Send faulted!\n");
    st++;
}

```

2. VxWorks 中汇编语言的使用

VxWorks 的开发中所使用的汇编语法是 AT&T 语法，它和 Intel 语法有所不同。下面简单的介绍实际应用中常用的部分汇编指令及格式等内容，详细的说明可以参考“GUN ToolsKit”一书的“as”部分和“gcc/g++”部分。

(1) 立即数。AT&T 的语法中，立即数全部是以“\$”开头，例如“\$6”。

(2) 寄存器。AT&T 的语法中，寄存器标识则是以“%”开头，例如“%eax”。

(3) 语法顺序。AT&T 的语法与宏汇编同样有着差异，可以说正好相反。例如宏汇编的一条“mov eax,10”指令，则在 AT&T 中就会变为“mov \$10,%eax”。

(4) 操作码的助记符。宏汇编中所使用的“byte ptr”、“word ptr”、“dword ptr”（即字节、字、双字），在 AT&T 语法中分别变为操作字后所跟的“b”、“w”、“l”。例如宏汇编中“add”就有可能是在 AT&T 的语法“addb”，“addw”，“addl”中的任意一个。

从上述的一些描述中，可以看到在汇编指令大部分的语法上，AT&T 的汇编还是和宏汇编有相似之处的。

在 VxWorks 应用开发过程中，如果希望在 C 语言中嵌入汇编，应使用编译控制指令“__asm__”。具体的语法及使用如下所示：

```
__asm__ ("addl $6,%eax");
```

从上述的示例中可以看到，在 C 语言中嵌入汇编指令就如同在写一行 C 语言程序一样，但也可以使用其他方法，例如__asm__("int \$45")。但是推荐使用“__asm__”编译控制指令。

3. C 语言与汇编语言之间的数据传递

下面描述在 VxWorks 开发中，C 语言和汇编语言之间的数据传递。

利用嵌入汇编的办法编程，会碰到如下的难题：如何将寄存器中的值读回到变量中来。

通常使用的有两种办法，这两种方法的使用范围都是规模比较小的汇编，也就是常会用到的、只有几条语句的程序段。

(1) 在汇编中使用变量。可以使用如下的方法来定义汇编中使用的变量：

```
int foo __asm__ ("myfoo")=2;
```

在汇编中可以使用的变量 myfoo 就是在 C 语言中使用的 foo。

(2) 定义寄存器变量。定义寄存器变量的语法如下：

```
register int *foo __asm__ ("eax");
```

定义的 foo 是指向 eax 的寄存器指针，它的值随着寄存器中值的变化而变化。在使用它时只需要像平常变量一样使用即可。

4. 参考示例

(1) C 语言里嵌入汇编指令参考示例。下面是在 embedS.c 文件中，嵌入汇编语言。

```
=====
embedS.c 文件，作用是读取 CPU ID。
=====
/* 定义寄存器变量*/
register int *VEAX __asm__ ("eax");
register int *VEBX __asm__ ("ebx");
register int *VECX __asm__ ("ecx");
register int *VEDX __asm__ ("edx");
int a,b,c,d;

/* 保存现场 */
__asm__("push %eax");
__asm__("push %ebx");
__asm__("push %ecx");
__asm__("push %edx");

__asm__("mov $0,%eax");
__asm__("cpuid");
a=VEAX;
b=VEBX;
c=VECX;
d=VEDX;

/* 恢复现场 */
__asm__("pop %eax");
__asm__("pop %ebx");
__asm__("pop %ecx");
__asm__("pop %edx");
printf("%x,%x,%x,%x",a,b,c,d);
if((b==0x756e6547)&&(d==0x49656e69)&&(c==0x6c65746e))
{
    sprintf(CPUVendor,"%s","Genuin Intel");
}
=====
```

(2) C 语言中调用汇编函数参考示例。下面是在 C 代码中调用汇编函数的一个参考示例。

```
=====
test()[test.c]——作用是测试系统执行一段 FOR 程序所需要的时间。
=====
#include <stdio.h>
extern long long CPUTicks(); /* 调用外部汇编函数声明 */

int test(void)
{
    long long t1,t2;
    double t;
    int i,n;
    t1 = CPUTicks(); /* 获得当前 CPU ticks 数*/
    for(i=1;i<100;i++)
```



```

n=n+1;
t2 = CPUticks(); /* 获得当前 CPU ticks 数*/
t=(double)((t2-t1)/333.0e6);
printf("time=%f\n",t);
}

```

读取当前 CPU ticks 数的函数:

=====
CPUticks() [CPUticks.s]——读取当前 CPU ticks 数
=====

```

/* read CPU(x86) ticks counter */
.file "CPUticks.s"
.text
.align 16
.globl _iclock
_CPUticks:
    .byte 0x0f,0x31
    ret

```



注意

可能遇到如下一些问题:

(1) 为什么在改变了寄存器的值时, 打印出来不对?

在各个函数的调用中, 会改变某些寄存器的值, 所以在调用之前需要对其值进行保存。

(2) 为什么调用汇编时导致了程序出错?

在调用汇编之前, 最好做好环境的保护, 事先保存寄存器的值。

5.4 存储映射 I/O 与端口 I/O

在 I/O 机制中, 分存储映射 I/O 机制和端口 I/O 机制, 如图 5.1 所示。

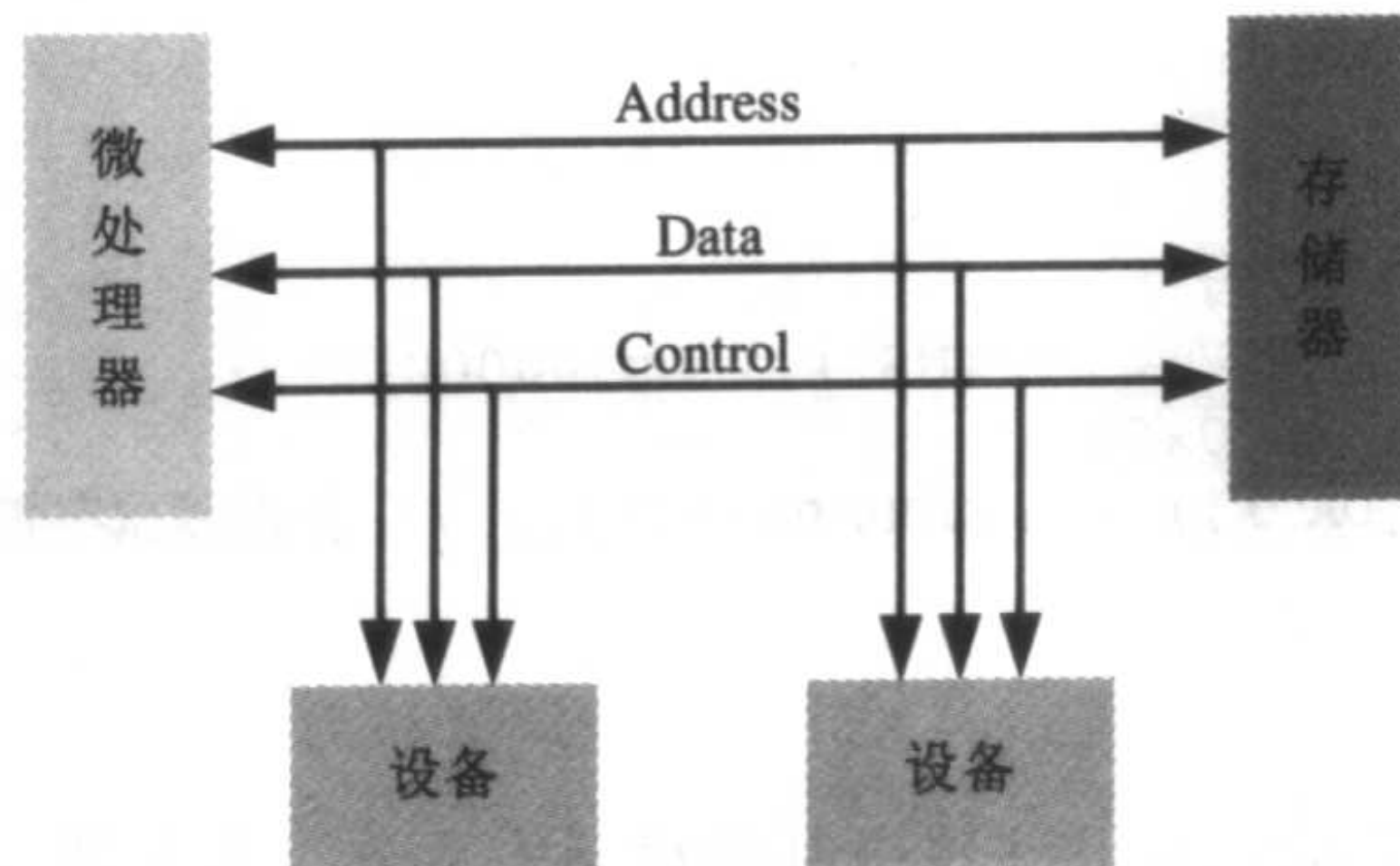


图 5.1 I/O 机制

下面将对这两种机制进行描述。

5.4.1 存储映射 I/O

存储映射 I/O 机制如图 5.2 所示。在这种机制中设备的寄存器与存储区仅仅通过地址来区分，因为系统将设备寄存器映射到系统的存储空间区。所有体系结构的目标机都支持存储映射 I/O，在这种 I/O 机制中用 C 语言访问设备的寄存器就像访问存储区一样方便。但无论选择哪种 I/O 机制，处理器与存储器及设备之间都是通过总线来连接。惟一不同的只是在控制线的使用上有所区别。

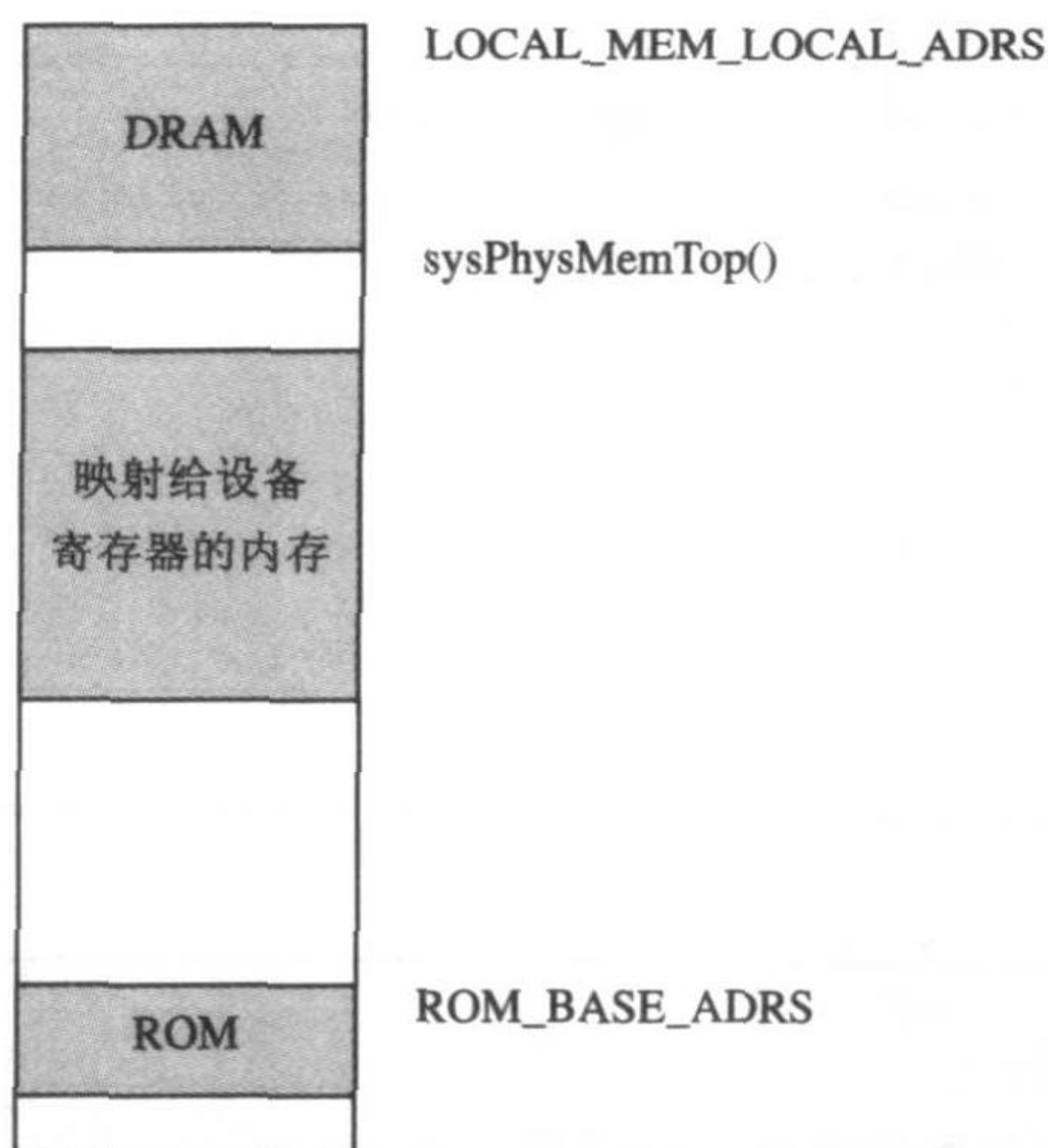


图 5.2 存储映射 I/O 机制

在存储映射 I/O 机制中，用 C 语言访问某设备寄存器的示例如下：

```
...
#define CPU_PCI_ISA_IO_ADRS 0xfc40          /* PCI I/O 空间基地址 */
#define PORTA_DATA (CPU_PCI_ISA_IO_ADRS + 0x0) /* 数据寄存器地址 */
#define PORTA_STATUS (CPU_PCI_ISA_IO_ADRS + 0x5) /* 状态寄存器地址 */

void send_Test(void)
{
    UCHAR    status;
    ...

    /* 读状态寄存器,并发送测试字符 */
    stauts = (UCHAR) ((* (PORTA_STATUS)) & 0x000000ff)
    if ( (status & 0x61) == 0x60 )
        * ((UCHAR *) (PORTA_DATA)) = ((UCHAR)('T')); /* 发送测试字符 'T' */
}
```

5.4.2 端口 I/O 机制

端口 I/O 机制如图 5.3 所示。对于这种机制的设备寄存器，系统通过特殊的总线周期来访问它，但体系结构必须提供控制线，通常需要汇编语言支持，并且需要特殊的存取指令。因为这种机制下处理器使用独立的 I/O 地址空间，而 C 语言没有提供一种用来表明一个地址是存储地址还是 I/O 地

址的方法，所以需要特殊的汇编指令来访问 I/O 地址空间的数据。

值得庆幸的是，可以使用汇编语言编写访问 I/O 地址空间数据的全局函数。然后在 C 代码中调用这些函数来访问设备寄存器，这样便可以非常容易地操作映射到 I/O 空间的设备。这种操作方法在前面的一些章节中已经进行了描述，在这里不再赘述。

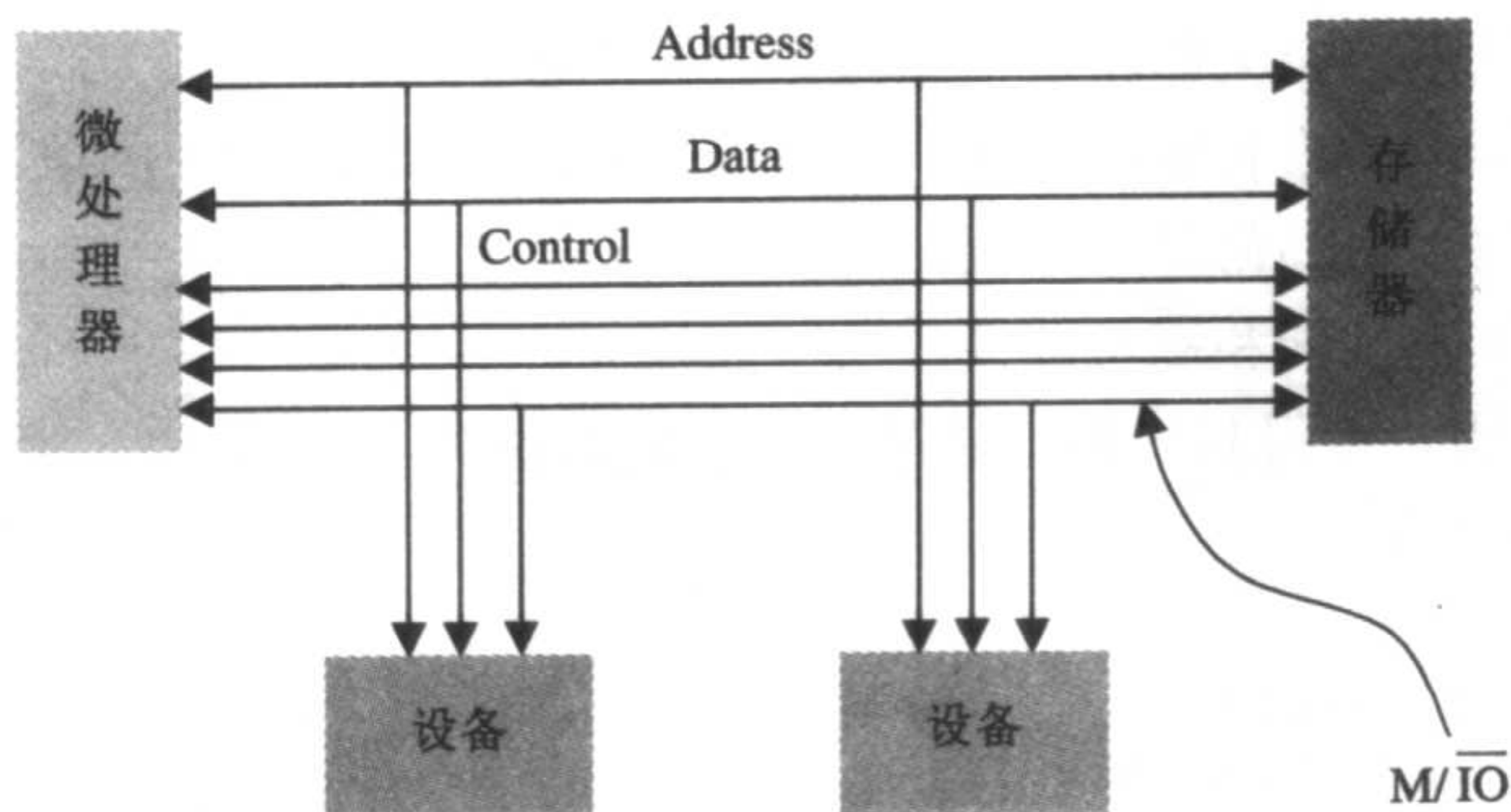


图 5.3 端口 I/O 机制

在端口 I/O 机制中，C 语言通过调用汇编函数访问设备寄存器的示例如下：

```
#define CPU_PCI_ISA_IO_ADRS 0xfc40      /* ISA I/O 空间基地址 */
#define PORTA_DATA          (CPU_PCI_ISA_IO_ADRS + 0x0) /* 数据寄存器地址 */
#define PORTA_STATUS        (CPU_PCI_ISA_IO_ADRS + 0x5) /* 状态寄存器地址 */

/* 下面是引用两个汇编函数，它们通过特殊的 inb 和 outb 指令实现读写操作 */
extern UCHAR sysInByte (int port);      /* 从端口里读一个字节 */
extern void sysOutByte (int port, char data); /* 往端口里写一个字节 */

void send_Test(void)
{
    UCHAR status;
    ...
    /* 读状态寄存器 */
    status = (UCHAR) (sysInByte(PORTA_STATUS) & 0x000000ff)
    if ( (status & 0x61) == 0x60 )
        sysOutByte(PORTA_DATA, (UCHAR)('T')); /* 发送测试字符 'T' */
}
```

5.5 设备驱动程序错误处理

通常情况下，设备驱动程序形成产品后，不应该在控制台上显示错误信息，而应该：

- 设置错误号 (errno) 和返回错误标志 (典型为 ERROR)；
- 让应用程序执行错误处理。

另一方面，当调试某个设备驱动程序时，如果出现错误则把错误信息显示在控制台上，这种做法在这个阶段是一种非常有效的方法，而且对开发者来说错误的定位变得一目了然。

在 VxWorks 系统中，通常是调用 logMsg() 函数打印错误信息。这个函数的优点是既可以在任务

中调用它也可以在中断服务程序中调用它。

本节主要是对设备驱动程序在开发阶段和形成产品后，如何设计错误处理进行描述。

5.5.1 错误处理需要考虑的事项

在理想情况下，处理错误的方式如下：

- 在开发阶段，把用户定义的错误信息打印到控制台上；
- 对于最终发布版本的设备驱动程序，仅仅设置错误号和返回 ERROR；
- 避免系统运行时检查如何处理错误；
- 错误处理代码的高性能和高可读性。

为了使上述的问题进一步得到认识，通过一个示例来说明。

1) 在开发阶段，希望有：

```
->dac 10, 1.0
Error in daLib.c at line21:
S_daLib_INVALID_CHANNEL_NUMBER
value = -1 = 0xffffffff
```

上述示例的操作在 shell 工具下执行，错误信息打印在控制台上，显示了源代码模块名、出错行号、设置的错误号为“S_daLib_INVALID_CHANNEL_NUMBER”和错误值（通道 10 不存在）。

2) 对于最终的产品，希望有：

```
->dac 10, 1.0
value = -1 = 0xffffffff
->printErrno(-1)
S_daLib_INVALID_CHANNEL_NUMBER
Value = 31 = 0X1f
```

上述的操作也在 shell 工具下执行，程序调用失败只返回错误值。通过调用 printErrno() 函数才可以显示错误号。

5.5.2 调试阶段错误处理设计

1. 错误处理宏定义

在设备驱动程序的源代码中，错误处理通过传递一个宏给设备驱动程序特殊的错误代码。例如：

```
if (channel < 0 || channel >= DA_MAX_CHANNELS)
{
    DRV_ERR (S_daLib_INVALID_CHANNEL_NUMBER);
    return (ERROR);
}
```

其中，DRV_ERR() 函数是一个错误处理例子宏，不是 VxWorks 的一部分。下面是这个错误处理宏的原型定义：

```
#ifdef DEBUG
#define DRV_ERR(err) \
drvError (__FILE__, __LINE__, (err))
#else
#define DRV_ERR(err) (errno = (err))
#endif
```


在开发阶段时编译器功能选项需要增加“-DDEBUG”选项时这段代码才有效。典型情况是把这段代码定义在头文件中，例如上述宏定义在 daadioLib.h 文件中。

值得说明的是上述的“_FILE_”和“_LINE_”被创建在 ANSI 定义的符号表常量中，通过带有文件名和当前行号的 C 预处理程序替代它们。例如上述代码可能通过如下预处理程序替代：

```
drvError ("daLib.c", 21, (S_daLib_INVALID_CHANNEL_NUMBER));
```

在开发阶段，当驱动程序任何时候出现错误时，为了能够停止程序执行。可以在错误函数（例如 drvError()）中设置断点。

2. 错误函数

错误函数通过调用 logMsg() 打印错误信息，类似 printf() 函数。但它是通过发送消息到日志任务的消息队列中来显示信息，所以在 ISR 中可以调用该函数。

下面是一个错误函数的示例：

```
#include "errno.h"
...
void drvError (char * file, int line, int error)
{
    if (intContext() == TRUE) /* Interrupt time */
        logMsg ("Errno %#x in %s at line %d\n", error, file, line);
    else /* Task time */
    {
        logMsg ("Error in %s at line %d: ", 10, file, line);
        perror("drvError");
    }
    errno = error;
}
```

5.5.3 发布版本错误处理设计

发布版本错误处理设计与调试错误处理设计不一样，错误代码在系统错误表中，只有通过 printErrno() 函数才可以显示错误代码。这就需要：

- (1) 创建用户头文件目录。例如..\target\config\bspName\h。
- (2) 在用户头文件目录下创建一个 xxModNum.h 文件，该文件包括模块数。
- (3) 连接（或拷贝）用户驱动程序特殊错误代码文件（例如 daadioErr.h）到用户头文件目录下。

同时，用户还需要把驱动程序特殊错误代码文件（daadioErr.h）包含在驱动程序代码中。有关上述的一些操作在下面进行描述。

1. 定义错误代码

这一步操作主要如下：

- (1) 创建用户头文件目录。例如..\target\config\bspName\h。
- (2) 在用户头文件目录下创建一个 xxModNum.h 文件，在该文件包括用户驱动程序模块定义。

例如：

```
#define M_daLib (512<<16)
```

- (3) 在用户头文件目录下的头文件中（例如 daadioErr.h）定义错误宏。例如：

```
#include "xxModNum.h"
```



```
/* daLib errors */
#define S_daLib_DAADIO_NOT_INITIALIZED(M_daLib | 1)
#define S_daLib_INVALID_CHANNEL_NUMBER(M_daLib | 2)
#define S_daLib_INVALID_VOLTAGE(M_daLib | 3)
```

需要说明的是最初的 500 个模块数是 Wind River 系统保留的, 并且错误宏必须定义成 S_xxx 的格式。

2. 重新编译 VxWorks

错误代码定义完成后, 需要重新编译 VxWorks 则错误处理才有效。其操作步骤如下:

- (1) 路径切换到目标机 BSP 目录下, 例如..\target\config\pcpentium。
- (2) 创建 statTbl.c 文件。

如果 Tornado 安装在 UNIX 操作系统上, 则在 BSP 目录下执行如下命令:

```
makeStatTbl ../../h 用户头文件目录 > statTbl.c
```

如果 Tornado 安装在 Windows 操作系统上, 则在 BSP 目录下执行如下命令:

```
makeStatTbl ..\..\h 用户头文件目录
```

- (3) 编辑 config.h 文件, 包含 INCLUDE_STA_SYM_TBL 选项。
- (4) 重新编译 VxWorks 映像。其执行命令如下:

```
make ADDED_MODULES=statTbl.o EXTRA_INCLUDE=-I 用户头文件子目录
```

5.6 VxWorks 下设备驱动程序编程规范

设备驱动程序不但要实现操作设备的功能, 同时在编写设备驱动源程序时也有一些规则和约定。这对程序的可读性、维护性等很有帮助, 否则编写的程序只有自己能看懂, 说不定时间长了连自己都看不懂, 那是一件多么失败的事情。

在 VxWorks 下编写设备驱动程序函数命名和参数命名一些规范如下。

- (1) 打开设备函数, 其规范如下:

```
xxOpen(XX_DEV *pDevHdr, char * pName, int mode)
```

- (2) 关闭设备函数, 其规范如下:

```
xxClose(XX_DEV *pDevHdr)
```

- (3) 从设备读一串数据函数, 其规范如下:

```
xxRead(XX_DEV *pDevHdr, char *rBuf, int nBytes)
```

- (4) 往设备写一串数据函数, 其规范如下:

```
xxWrite(XX_DEV pDevHdr, char *wBuf, int nBytes)
```

- (5) 设备控制函数, 其规范如下:

```
xxIoctl(XX_DEV pDevHdr, int cmd, int arg)
```

- (6) 安装设备驱动程序函数, 其规范如下:

```
xxDrvInstall(int args, ...)
```


该函数通过调用 `iosDrvInstall (xxCreat, xxDelete, xxOpen, xxClose, xxRead, xxWrite, xxIoctl)` 函数, 往 I/O 系统驱动程序表添加设备驱动程序及完成驱动程序的初始化。在系统设备初始化阶段调用 `xxDrv()` 函数, 而且只能调用一次。

(7) 卸载设备驱动程序函数, 其规范如下:

```
xxDrvUninstall (int args, ...)
```

该函数通过调用 `iosDrvRemove (int drvnum, BOOL forceClose)` 函数, 从设备驱动程序表里卸载指定的设备驱动程序并删除与之关联的设备。

(8) 创建设备函数, 其规范如下:

```
xxDevCreate(char *devName, int arg, ...)
```

该函数为设备描述结构分配内存, 执行设备相关的初始化, 调用 `iosDevAdd( XX_DEV pDevHdr, char *devName, int drvNum)` 往设备列表内添加设备描述符, 并用设备名字 (`drvName`) 和驱动程序号 (`drvNum`) 初始化 `XX_DEV` 结构。

(9) 删除设备函数, 其规范如下:

```
xxDevRemove(char *devName, int arg, ...)
```

该函数通过调用 `iosDevDelete(DEV_HDR *pdevHdr)` 函数, 根据 `pdevHdr` 从设备驱动程序表里卸载设备描述符地址。

有关具体内容参考后面的第 6 章的内容。

5.7 高速缓存的一致性问题的

很多时候在开发板卡驱动程序的过程中, 程序员对分配的内存带的高速缓存会产生迷惑, 在何种情况下应该分配带高速缓存的内存、有高速缓存会有什么好处、注意事项有哪些、有可能会出现哪些问题等太多的不明确问题, 所以在这一小节里简单讨论一下有关的高速缓存问题。

首先高速缓存的最大好处是解决 CPU 与内存之间的瓶颈问题。现在使用的 SDRAM 的速度一般都在 $3\text{ns} \sim 10\text{ns}$ 左右, 而 CPU 的执行速度已经到了 1GB , 甚至更高, 很明显这样的速度差异会导致 CPU 在存取内存时经常要等待数据在内存上的操作, 这将致命地影响系统的性能, 所以人们引进了高速缓存机制来缓解瓶颈问题。由于高速缓存的造价及物理上的限制, 高速缓存通常都不大, 只有 256KB 或 512KB , 但是速度非常快, 它的速度值一般在 CPU 与内存之间, 它有一系列的快速算法来保证下次 CPU 存取内存中的数据时, 保证它存放在高速缓存里的数据的高机率性。这样 CPU 在存取数据时, 如果数据在高速缓存内则直接对高速缓存进行存取操作而无须与缓慢的内存打交道。

为板卡开发的驱动程序必须保证高速缓存的一致性, 即高速缓存中的数据必须与内存中的数据保持同步, 因为在使用异步方式读取内存时有可能产生内存与高速缓存失步。数据高速缓存是为了减少存取内存的次数, 而加快系统的执行性能。数据高速缓存通常有两种方式: “写通” 和 “写回”。写通是指在写内存的同时往高速缓存中写。保证高速缓存在数据输出上的一致性, 但不保证在输入时的一致性。写回仅将数据写到高速缓存中, 它保证数据在输入和输出时都是一致的。

当 CPU 写一个 DMA 设备上的内存时, 首先把数据写到高速缓存中, 当 DMA 设备从 RAM 中传输数据时, 不保证内存中的数据与高速缓存中的数据一致。这样输出到设备的数据也许不是最新的,

最新的数据应该在高速缓存内。可以在数据传输到 DMA 设备之前将高速缓存数据刷新到内存中。

当 CPU 读一个 DMA 设备上的内存时，读到的数据可以来自高速缓存的缓冲区而不是从设备传输到内存中的数据，可以确保高速缓存缓冲区已经被标记（屏蔽）来解决读出的数据来自内存而非高速缓存。

设备驱动程序可以通过分配无高速缓存缓冲区（使用 `cacheDmaMalloc()`）或刷新和使高速缓存无效来解决 CPU 与设备之间数据传输的高速缓存一致性问题。分配无高速缓存缓冲区常用于分配静态缓冲区，它需要 MMU 的支持。无高速缓存缓冲区频繁地被动态分配和释放将导致大量的内存被标记为无高速缓存。一个折中的方法是使用动态无高速缓存缓冲区的方式人为地刷新（使用 `cacheFlush()` 函数）和屏蔽高速缓存（使用 `cacheInvalidate()`）。



注意

在设备读数据之前用 `cacheFlush()` 保证数据一致，而在设备写内存后用 `cacheInvalidate()` 将保证高速缓存和内存数据的一致。

小 结

对于一件事情的处理，好的方法会带来“事半功倍”的效果。同样，在 VxWorks 下开发设备驱动程序时，掌握一些方法也是必要的。这些方法包括开发前资料的收集、宏定义的使用、I/O 地址的映射、错误处理设计以及高速缓存问题等，这些问题是本章内容的核心。本章对它们进行了详细描述以供读者参考和借鉴。

第 6 章 编写字符设备驱动程序

接下来的几章中，将分别讨论一下字符设备、串行设备、块设备和网络设备驱动程序的编写方法。在讨论所有这些设备的编写方法之前，将会简要地回顾一下 VxWorks 的 I/O 系统的内部结构以及应用程序、驱动程序、设备之间的关系。

6.1 I/O 系统回顾

6.1.1 驱动程序接口

在 VxWorks 中，I/O 系统提供了与设备无关的用户接口，应用程序可以通过标准的接口（`open()`、`creat()`、`read()`、`write()`、`ioctl()`、`close()`、`remove()`）访问设备驱动程序，而不用关心驱动程序是如何实现的。对于一般的设备来说，`remove()`接口是不需要实现的。`remove()`接口的含义并不是将设备从系统中卸载或是将驱动程序从系统中移除，而是指删除一个文件。因此只有两种设备的驱动程序是必须实现 `remove()`接口的，一个是文件系统驱动程序，这个函数将引起文件的删除事件；另一个是网络设备驱动程序，调用这个接口将删除远程主机上的文件。表 6-1 中是七个驱动程序接口函数的简单描述。

表 6-1 I/O 接口函数表

接口名称	函数原型	描述
open	<code>open(filename, flags, mode)</code>	打开文件
creat	<code>creat(filename, flags)</code>	创建文件
read	<code>read(fd, &buf, nBytes)</code>	从文件中读取
write	<code>write(fd, &buf, nBytes)</code>	向文件中写入
ioctl	<code>ioctl(fd, command, arg)</code>	其他控制命令
close	<code>close(fd)</code>	关闭文件
remove	<code>remove(filename)</code>	移除文件

可以看出，`open()`、`creat()`、`remove()`这三个函数在参数中使用“filename”来指定一个文件，而 `read()`、`write()`、`ioctl()`、`close()`则是使用“fd”指示一个文件。这个不同点将在下面的驱动程序运作过程中得到解释。

由于 I/O 系统提供了与用户无关的应用程序接口。驱动程序的编写者就面临着是否使用标准接口来编写驱动程序的问题。那么什么时候应该使用标准接口，而什么时候又使用非标准的接口呢？

标准的接口提供了与系统相关的标准 I/O 功能，诸如 `select()`函数的支持、I/O 重定向、流设备的支持。应用程序在以相同的接口访问多种设备或者需要使用系统相关的 I/O 功能时就应该使用标

准接口来设计设备驱动程序。

而在另一种情况中，如果设备不需要那些 I/O 系统所提供的系统相关功能，或者设备访问速度在使用中是关键因素的时候，就不应使用标准的用户接口。有时设备会执行一些有别于七个标准结构的其他操作，这时也可以使用非标准的接口。

一般来说，标准接口的驱动程序可以用在下面所列出的设备中：

(1) 块设备驱动程序。基于文件系统、可随机访问、使用多字节块为单位传输的设备，如软盘、硬盘等。这里所说的块设备是指那些用来支持文件系统的设备。VxWorks 中提供了三种主要的文件系统：MS-DOS 兼容的文件系统 dosFs、原始文件系统 rawFs 以及磁带文件系统 tapeFs。块设备接口提供了文件系统对硬件访问的能力。

(2) 大部分的非块设备驱动程序。不同于块设备的其他设备，如鼠标等串行输入设备。

6.1.2 I/O 系统结构

VxWorks 中 I/O 系统与设备驱动程序之间的层次关系非常明显，如图 6.1 所示。

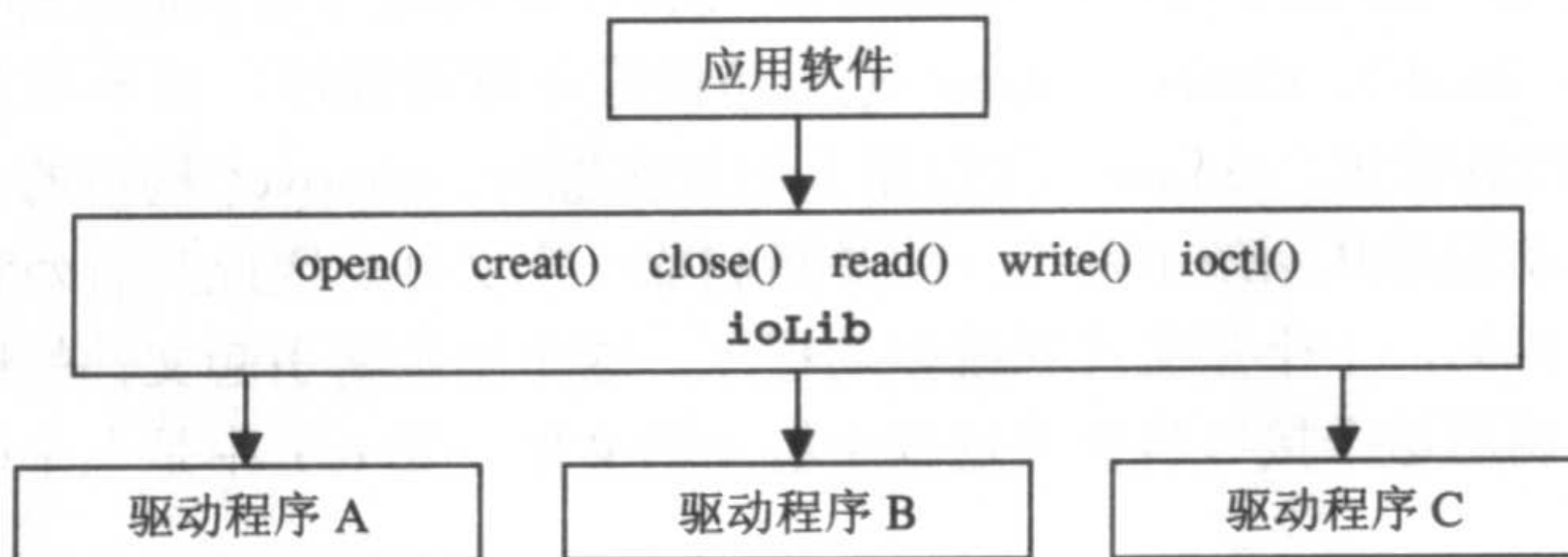


图 6.1 I/O 系统结构

对于块设备来说，应用程序通过 I/O 系统访问文件系统，而后由文件系统调用驱动程序访问设备。而其他的非块设备则是 I/O 系统直接调用驱动程序访问设备。应用程序所看见的设备都是以文件名或文件描述符的形式存在，而设备驱动程序则与这两者无关。设备驱动程序与 I/O 系统的通信都是通过设备描述符和设备 ID 来实现的。图 6.2 显示了应用程序、I/O 系统和设备驱动程序的关系。

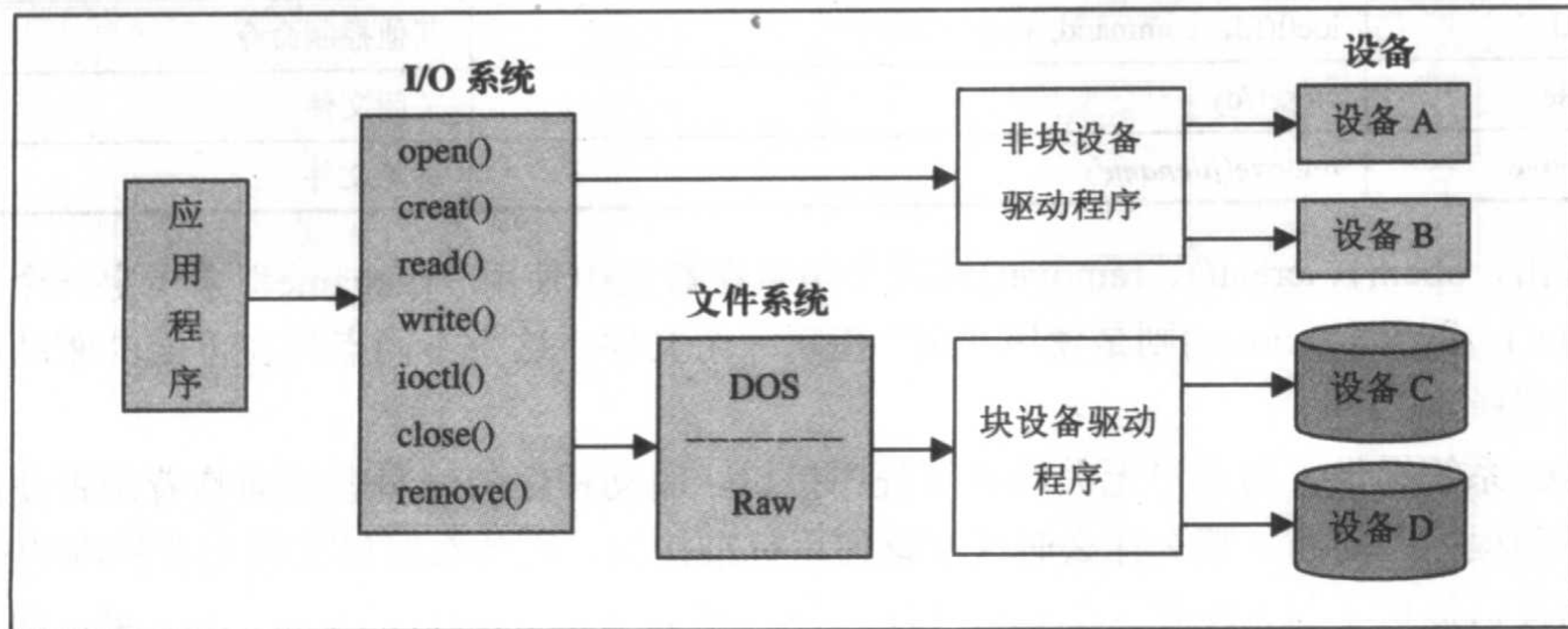


图 6.2 应用程序、I/O 系统和设备驱动程序的关系

实现上面的关系是通过设备列表、驱动程序描述表、文件描述符表来完成的。这三张表构成了应用程序与设备之间的桥梁，在后面的章节中将进行详细描述。

6.1.3 与设备驱动程序关联的表

1. 设备列表

设备列表是一个双向链表。每当添加设备时，系统都会向链表中添加新的环节。与其他两个表不同的是，设备列表的大小是动态的，可以任意向其中添加新设备。而不论是否访问同一个驱动程序，每一个设备都会占用链表的一个环节。设备列表如图 6.3 所示。

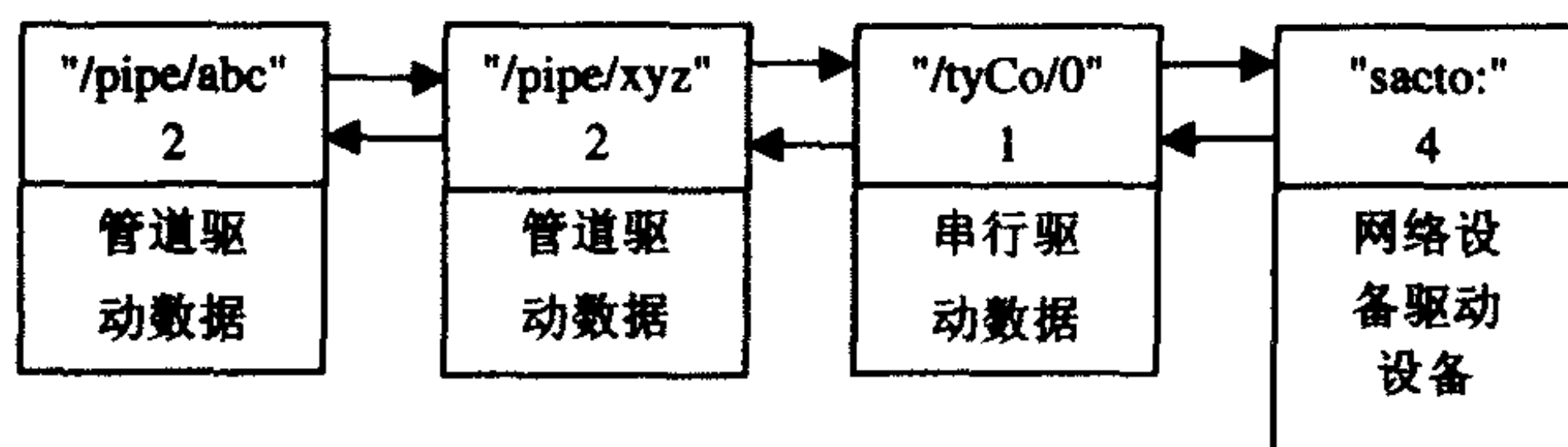


图 6.3 设备列表

设备列表的作用是为 `open()`、`creat()`、`remove()` 这三个函数提供文件与设备的连接。当应用程序开始执行这三个函数中的一个时，I/O 系统会将与文件名参数最匹配的设备返回，用于应用程序对文件的其他操作。I/O 系统在查找文件名时可以不完全匹配，它总会返回在名称上最匹配的那个设备，例如打开设备 `/pipe/xy`，由于在上表中没有完全匹配的文件名，则会返回 `/pipe/xyz` 设备。而如果查到两个设备都符合要求，则会返回最先查到的设备。

在系统运行时，可以在 Shell 或是应用程序中使用 `devs` 或 `sioDevShow` 来查看当前系统中所存在的设备，如下面的例子：

```
-> iosDevShow
Drv      Name
0        /null
1        /tyCo/0
3        host:
4        /vio
5        /tgtsvr
```

在上述系统中建立了 5 个设备，0、1 设备是标准输入输出文件，是由系统创建的；3 号设备是由网络任务建立的；4 号设备是虚拟 I/O 设备由 WindSh 创建；5 号设备则是由 target agent 建立的，用于主机与目标机的通信。

2. 驱动程序描述表

驱动程序描述表的结构如图 6.4 所示。

VxWorks 的 I/O 系统通过这个表存储驱动程序各个函数的入口点。在系统中安装的所有驱动程序都会在这个表中有相应的入口点。这个表的大小是固定的，所以在系统的一次运行中只能安装有限的驱动程序。用户可以通过修改 BSP 的 `configAll.h` 中的宏定义 `NUM_DRIVERS` 来确定这

张表的大小。正如上面所示的那样，通常 `creat()` 和 `open()` 的实现函数是相同的，而 `remove()` 函数不必实现。

驱动程序索引号	creat	remove	open	close	read	write	ioctl
0	...	...	...	...	...	...	...
1	...	...	...	...	...	...	...
2	xxOpen	NULL	xxOpen	xxClose	xxRead	xxWrite	xxIoctl
3	...	...	...	...	...	...	...

图 6.4 驱动程序描述表

值得说明的是，用户可以在系统运行中使用 `iosDrvShow()` 函数来显示驱动程序描述表，该函数示例如下：

-> `iosDrvShow`

Drv	create	remove	open	close	read	write	ioctl
1	421db4	0	421db4	421ddc	42b76c	42b69c	421e08
2	0	0	424fd4	0	425004	425044	425130
3	426e88	426f04	427170	426e34	427228	427254	426ffc
4	415f40	0	415f40	416000	42b76c	42b69c	416074
5	41626c	41642c	416288	416520	4165d0	416670	416710

可以看到，1、2、4 号驱动程序将 `NULL` 作为 `remove()` 函数的入口点，而 2 号驱动程序并未实现 `create`、`remove`、`close` 接口。

3. 文件描述符表

文件描述符表是 I/O 系统将文件描述符与驱动程序、设备对应起来的手段。文件描述符表如图 6.5 所示。

文件描述符	驱动程序索引号	驱动程序指定的设备号
3	2	0
4	5	0
5	5	1

图 6.5 文件描述符表

文件描述符的索引是由 `open()` 或 `creat()` 返回的文件描述符。`open()` 调用如下所示：

```
fd = open("/MOXA/01", O_RDWR, 0);
```

在系统的运行过程中可以使用 `iosFdShow` 来查看已经分配的文件描述符，如下所示：

-> `iosFdShow`

fd	Name	Drv
3	/tyCo/0	1
4	/vio/1	4

在这个系统中，已经由驱动程序 1、4 分别创建了两个文件/tyCo/0 和/vio/1。

6.1.4 安装驱动程序

在用户使用设备驱动程序之前，必须把其挂接在系统中。在打开设备之前，也必须已经创建该设备。VxWorks 系统是通过 iosDrvInstall()函数及 iosDevAdd()函数来完成这两个操作的。

函数 iosDrvInstall()的作用是挂接驱动程序，也就是将驱动程序的入口点填入到驱动程序表中。iosDevAdd()函数会将设备加入到设备列表中。

用户必须在驱动程序中实现包含有这两个函数的接口，一般会用接口 xxDrv()包含 iosDrvInstall()函数；用接口 xxDevCreate()包含 iosDevAdd()函数。关于这两个接口将在下一节中讨论。

xxDrv()和 xxDevCreate()这两个函数不是由 I/O 系统访问的，它们分别用于安装驱动程序和创建设备。通常的情况下，用户会在引导的时候调用这两个函数。



注意

在以后的讨论中，会假定用 xxOpen()或 myOpen()之类的名称来编写驱动程序。

6.1.5 I/O 系统的工作流程

1. 打开设备

用户在使用设备之前，将会使用 open()函数（或 creat()）函数来打开（或创建）文件。通常的驱动程序都会为 open()与 creat()使用相同的函数入口点，所以下面的讨论中都使用 open()函数来打开设备。

假定用户使用下面的语句打开一个设备：

```
fd = open("/Dev/0", O_READ, 0);
```

在这里 fd 是 open()函数返回的文件描述符，/Dev/0 是文件名，O_READ 是文件打开方式，而 0 则是文件打开模式。那么系统是怎样与设备驱动程序联系起来的呢？

首先，用户使用 open()函数之后，I/O 系统开始在设备列表中查找最匹配文件名“/Dev/0”的设备。找到“/Dev/0”设备之后，I/O 系统将通过设备描述结构的 drvNum 字段中存储的驱动程序索引号在驱动程序描述表中查找相应的驱动程序入口。最终，系统将找到相应设备的驱动程序。这一过程如图 6.6 所示。

用户在执行 creat()函数之后，I/O 系统的工作流程也与执行 open()的时候相同。

当得到文件描述符 fd 之后，I/O 系统将会把 fd 对应的驱动程序索引号和设备号加入到文件描述符表中以便应用程序后续的访问，如图 6.7 所示的创建设备操作描述了这个过程。

2. 读写及控制功能操作

用户获得了设备的文件描述符之后就可以将其用于读写和控制功能的操作。由于这三项操作的大体步骤都是相同的，这里仅仅讨论一下读操作。

经过打开操作，系统已将文件描述符、驱动程序索引号、设备号添加到文件描述符表中。在执行读操作的时候，I/O 系统以 fd 参数作为索引，在驱动程序描述符表中直接找到相应程序的入口点，调用设备对应的驱动程序。

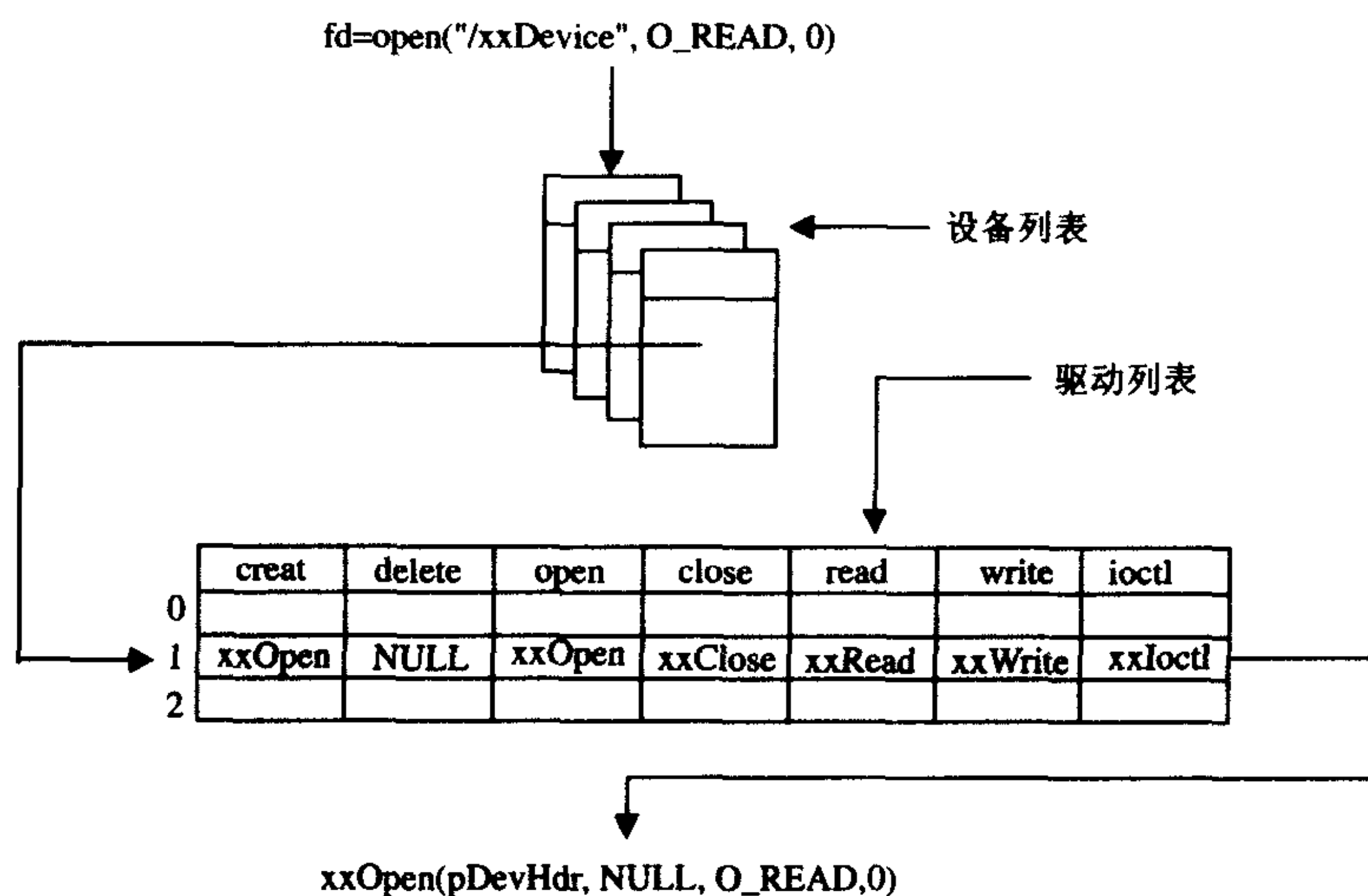


图 6.6 打开设备操作

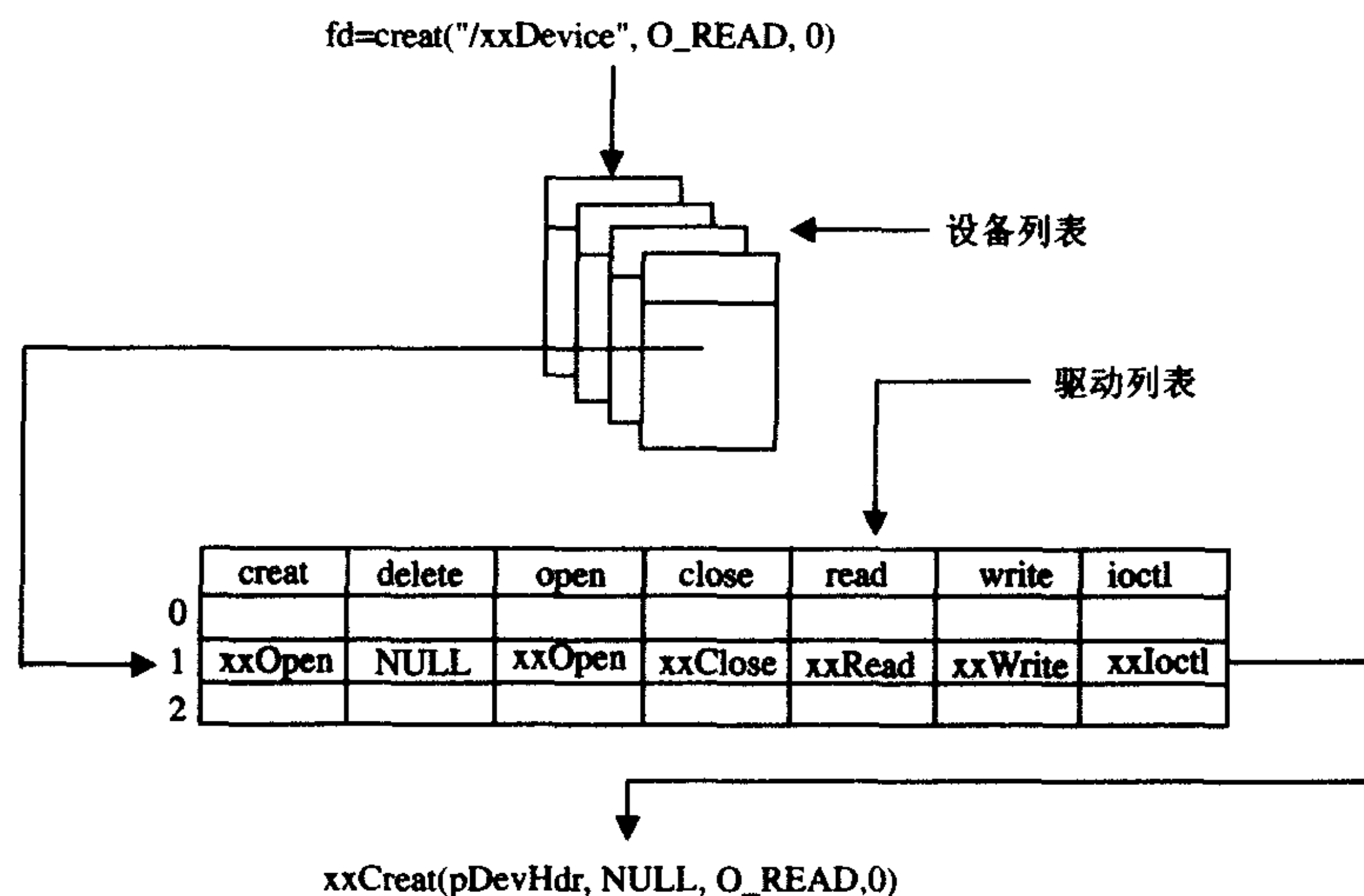


图 6.7 创建设备操作

下面的三个图分别示例了执行设备读（如图 6.8 所示）、写（如图 6.9 所示）及 I/O 控制操作（如图 6.10 所示）的工作流程。

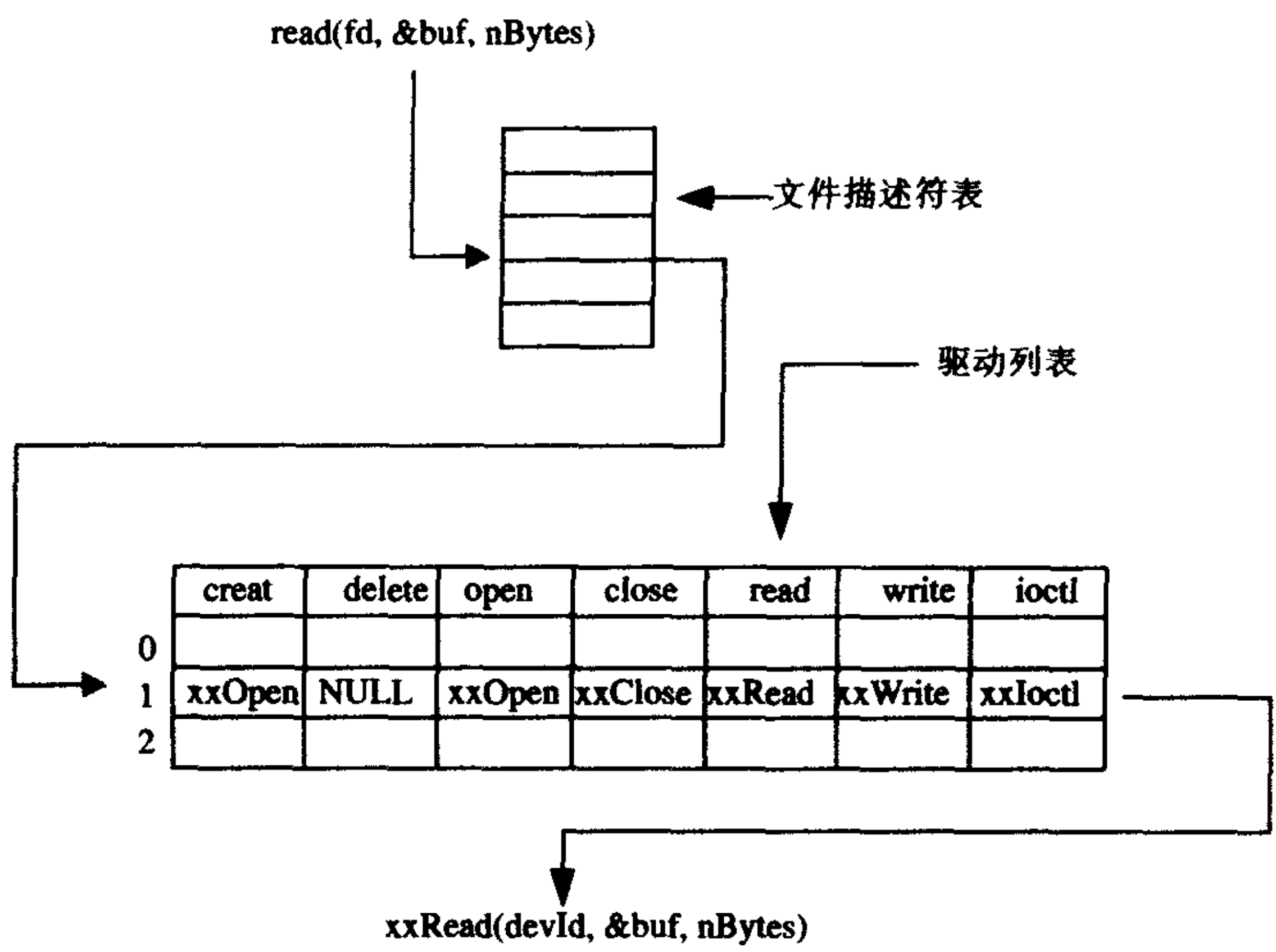


图 6.8 设备读操作

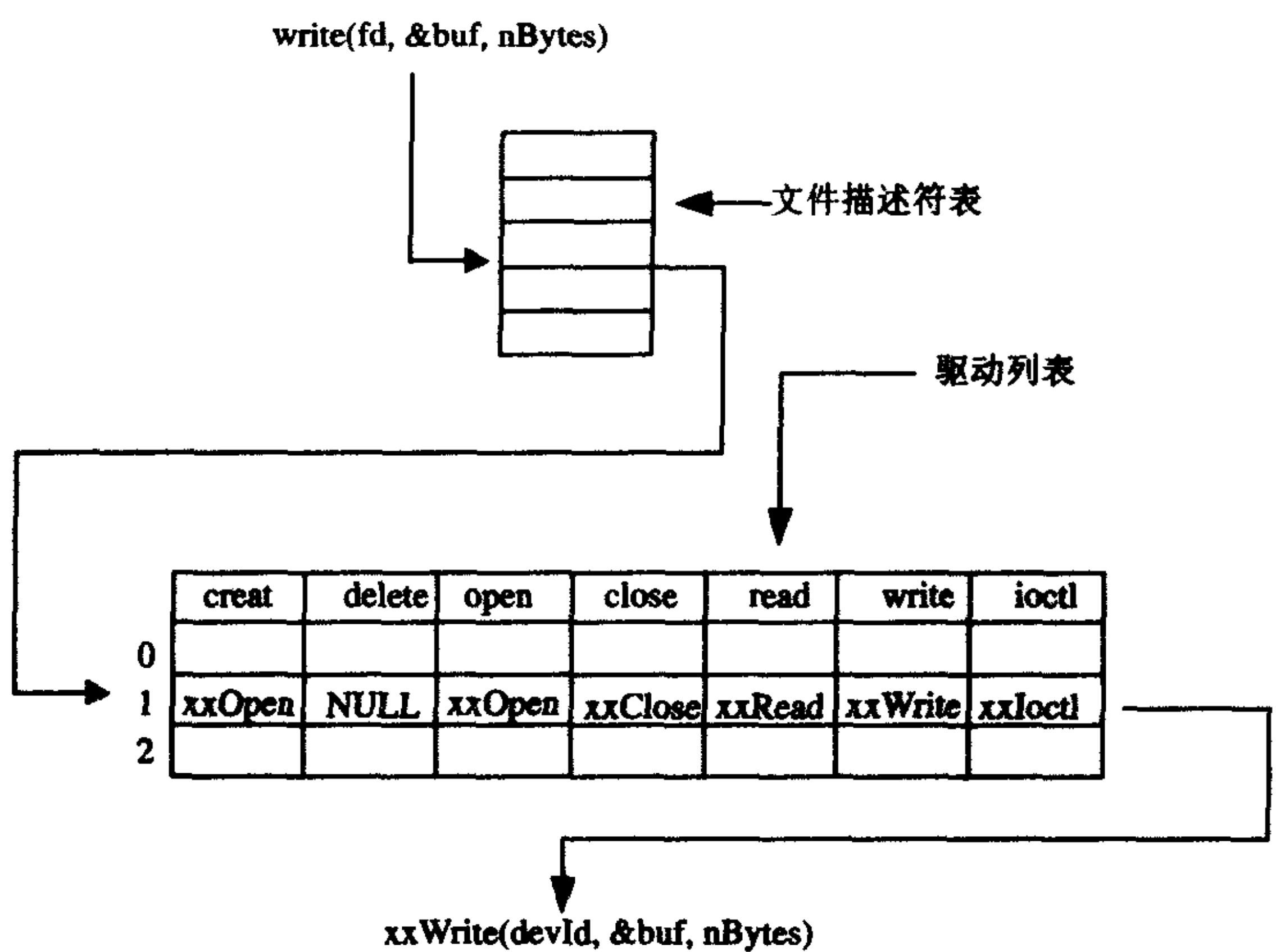


图 6.9 设备写操作

经过上面的回顾，读者可能对设备驱动程序的工作方式有了一些回忆。下面就讨论一下字符型设备的驱动程序并在本章的最后写出一个例子模板。

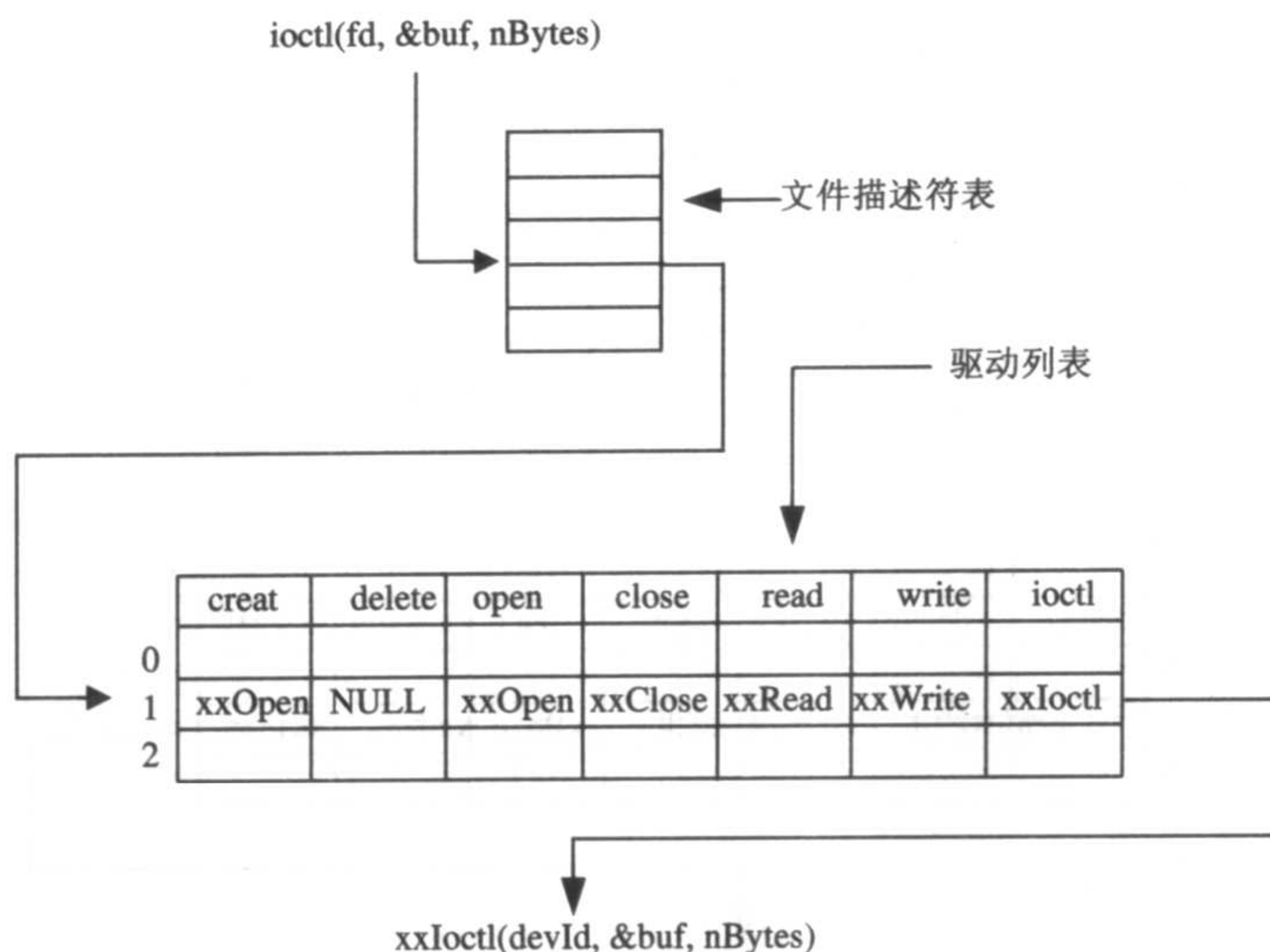


图 6.10 设备 I/O 控制操作

6.2 一步步编写字符型设备的驱动程序

在编写设备驱动程序前，首先看一下应用程序、系统、驱动程序的相互调用关系，如图 6.11 所示。

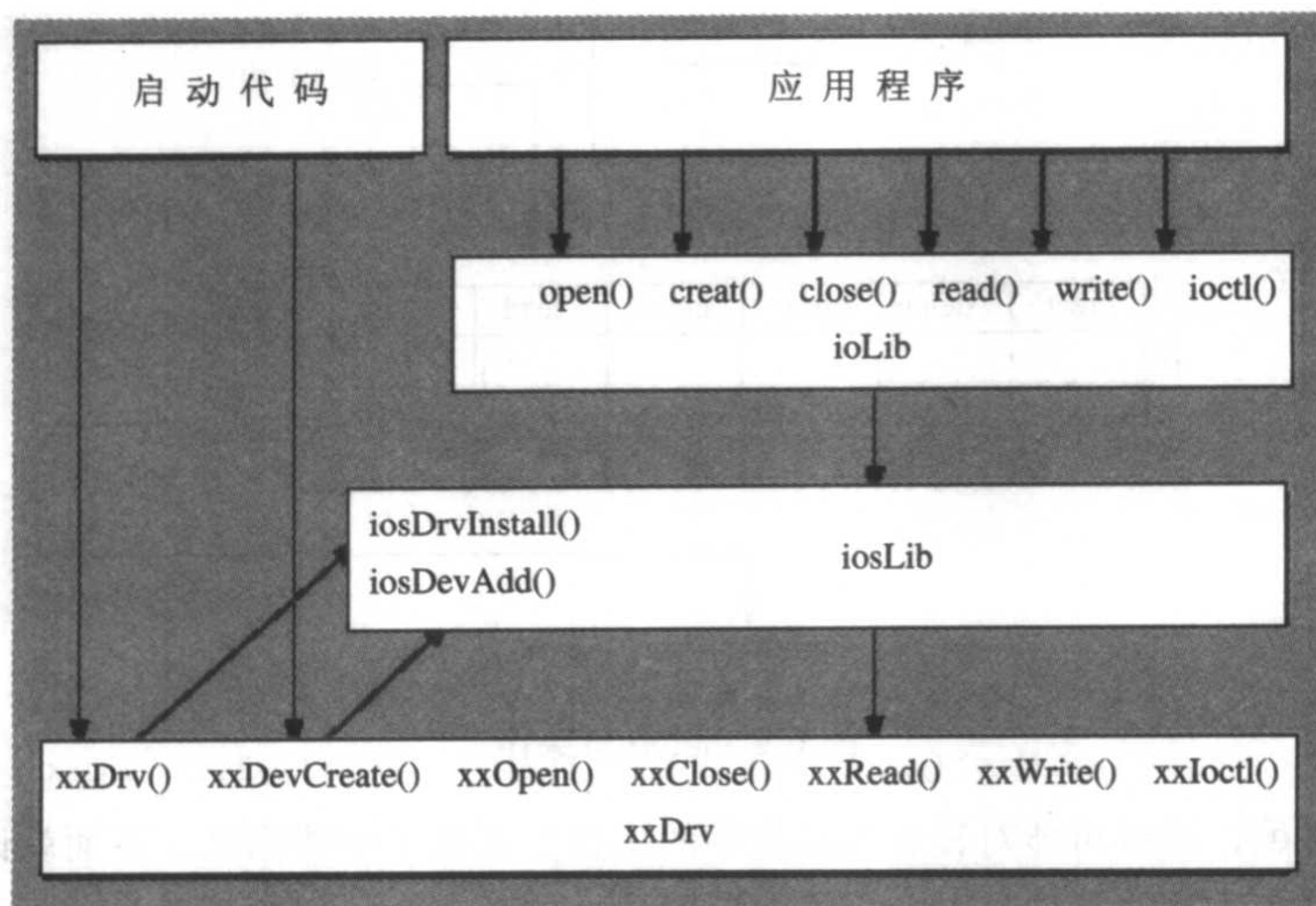


图 6.11 应用程序、I/O 系统及设备驱动程序之间的详细关系

通过图 6.11 可以知道字符设备驱动程序主要由 7 个函数完成驱动程序与系统的衔接, 即:

- `xxDrv()`: 驱动程序安装函数;
- `xxDevCreate()`: 设备创建函数;
- `xxOpen()`: 打开函数;
- `xxClose()`: 关闭函数;
- `xxRead()`: 读函数;
- `xxWrite()`: 写函数;
- `xxIoctl()`: I/O 控制函数。

首先系统启动代码调用驱动程序安装函数, 该函数通过调用 `iosDrvInstall()` 函数安装驱动程序; 然后启动代码将调用设备创建函数创建设备, 该函数通过调用 `iosDevAdd()` 函数为系统创建设备。经过上述操作后, 用户可以调用 VxWorks 的标准 I/O 接口函数对设备进行操作, 例如打开、关闭、读、写等, 不过这些操作最终是由驱动程序来完成。

6.2.1 驱动程序安装函数

驱动程序安装函数的目的是将驱动程序的各个函数入口点添加到系统的驱动程序表中, 提供给 I/O 系统访问。通常的实现方式如下:

```
STATUS xxDrv(int args...);
```

在 `xxDrv` 接口中, 要做下面的事情:

- 调用 `iosDrvInstall()` 函数将驱动程序入口点添加到系统的驱动程序表中;
- 做一些必要的初始化。

根据上面所要进行的操作, 驱动程序编写者可以自行决定该函数的参数。这个程序必须在访问驱动程序之前调用, 为驱动程序的访问做好准备。一般来说这个函数只应执行一次。下面是驱动程序安装函数的一个例子程序:

```
LOCAL int xxDrvNum=0; /* 驱动程序索引号 */
STATUS xxDrv()
{
    /* 首先判断驱动程序是否已经安装 */
    if(xxDrvNum>0)
        return(OK);

    ... /* 在这里添加驱动程序的初始化部分 */

    /* 将驱动程序添加到驱动程序描述表中 */
    if((xxDrvNum = iosDrvInstall(xxOpen, NULL, xxOpen,
        xxClose, xxRead, xxWrite, xxIoctl))==ERROR)
        return(ERROR);
    return(OK);
}
```

这里要注意三点。

- (1) 几乎所有的函数入口点都可以用 `NULL` 来代替, 这取决于驱动程序的编写。
- (2) `iosDrvInstall()` 函数执行错误会返回 `ERROR`, 而执行正确则返回一个驱动程序索引号。

(3) iosDrvInstall()所返回的驱动程序索引号是用来创建设备的,所以一定要注意保存,最好使用一个全局变量。

6.2.2 设备创建函数

安装完驱动程序后的下一个步骤就是创建设备。创建设备由设备创建函数来完成,通常的实现方式如下:

```
STATUS xxDevCreate(char * devName, int arg...);
```

其处理过程如下。

(1) 判断驱动程序是否已经安装,如果未安装则应返回错误并且设置错误号为“S_ioLib_NO_DRIVER”。

(2) 为设备描述符分配空间并将其初始化。

(3) 调用 iosDevAdd()函数将设备添加到系统设备列表中。

(4) 执行一些设备相关的初始化代码。

该函数的参数也是根据驱动程序的编写需要以及硬件需要而定的。如果将设备添加到设备列表的过程中,系统发现了重复设备则会返回错误,但驱动程序编写者可以不处理这个错误。

下面是创建设备函数的一个例子:

```
STATUS xxDevCreate(char * devName, int arg...)
{
    xx_DEV *pxxDev;    /* 设备描述符 */

    /* 检查驱动程序是否安装 */
    if(xxDrvNum<1)
    {
        errno=S_ioLib_NO_DRIVER;
        return(ERROR);
    }

    /* 分配并初始化设备结构 */
    if((pxxDev=(xx_DEV *)malloc(sizeof(xx_DEV)))==NULL)
        return(ERROR);
    bzero(pxxDev, sizeof(xx_DEV));

    ... /* 初始化必要的设备描述结构 pxxDev */
    ... /* 执行设备的初始化 */

    /* 将设备添加到设备列表 */
    if(iosDevAdd(&pxxDev->devHdr, devName, xxDrvNum)==ERROR)
    {
        free((char *)pxxDev); /* 释放设备所占的资源 */
        return(ERROR);
    }
    return(OK);
}
```

当添加好设备之后,就可以通过 open()或 creat()函数来打开设备,进行操作了。

6.2.3 打开设备函数

对于字符型设备来说，打开设备操作与创建设备操作通常使用相同的函数，所以这里仅仅讨论打开操作。打开操作使用下面的接口：

```
int xxOpen( pDevHdr, name, flags, mode)
```

其中，pDevHdr 是指向设备描述符的指针；name 是指向设备名称的指针；flags 是由用户在调用 open 时传入的标记，通常是不使用的，但可以根据设备和驱动程序的需要传递一些其他内容；mode 参数通常用于其他类型的驱动程序，对于字符型设备来说既可以使用，也可以不使用。

打开操作通常会执行一些设备的初始化或者适当的设置。如果成功执行，那么就会返回设备的标识，这个标识通常就是 pDevHdr；如果操作失败，则会返回 ERROR。

xxOpen()函数的返回值会被 I/O 系统添加到文件描述符表中，而 I/O 系统会将其对应的文件描述符返回给用户。如果返回值是 ERROR 则说明设备打开失败，不能进行后续的操作。当然，根据设备的需要，xxOpen()函数也可以返回一些设备相关的内容。比如返回设备自有结构的指针。如果设备开发者想确认一下 xxOpen()函数所返回的内容已经被 I/O 系统添加到了文件描述符表中，可以用 iosFdValue()函数来查询，下面是查询的例子：

```
-> iosFdValue(fd)
value = 176659240 = 0xa879b28
```

执行之后，该函数会返回设备描述符所在的地址。

传入 xxOpen()的 name 参数是包括了设备名的字符串，也可以通过该参数为驱动程序传入必要的选项。假设 xxDevCreate()函数在系统中创建了这样的设备“/xxDev”，在这个参数中就可以使用“/xxDev/4”来指定要打开的设备和其他相关参数。这个例子的“/xxDev/4”中指定了设备“/xxDev”同时传递了“/4”参数，当然也可以直接使用“/xxDev”来打开设备。

下面是一个打开设备函数的模板：

```
int xxOpen(DEV_HDR *pxxDevHdr, char *name, int flags, int mode)
{
    /* 获得设备描述符 */
    xx_DEV* pxxDev=(xx_DEV *)pxxDevHdr;
    if(pxxDevHdr==NULL)
    {
        errnoSet(S_xx_NODEV);
        return(ERROR);
    };

    /* 确定设备是否已经打开 */
    if(pxxDev->Opened)
    {
        errnoSet(S_xx_DEVOPENED);
        return(ERROR);
    }
    pxxDev->Opened=TRUE;

    ... /* 在这里作必要的初始化 */
```



```
return((int)pxxDevHdr);    /* 返回设备标识符 */
}
```

有时在 `xxOpen()` 中需要执行一些其他的操作，比如创建信号量、消息队列，那么可以在函数返回前做这些操作。

6.2.4 设备读/写函数

设备的读操作和写操作是两个相反的动作，一个是向设备发送数据，一个是从设备接收数据。这两个操作非常相似，其实现接口如下：

```
int xxWrite( deviceId, pBuf, nBytes);
int xxRead( deviceId, pBuf, nBytes);
```

参数 `deviceId` 是从 `xxOpen()` 返回的值，可以是指向 `DevHdr` 的指针，也可以是指向用户所指定的结构体的指针，详细的内容请参考上一小节中的叙述。`pBuf` 是指向数据缓冲区的指针，对于读操作来说指向了接收数据的缓冲区；而对于写操作来说指向了发送数据的缓冲区。`nBytes` 是用户所期望的读或写的字节数目，同时指定了缓冲区的大小。读操作或写操作通常有三种返回：其一是实际中读或写的字节数；其二是在出现错误时返回 `ERROR`；最后，对于某些设备来说，将存在返回值 0，以指示设备的某种状态。

读操作和写操作在实现上与硬件的相关性比较大，下面是读操作函数的一个示例模板（写操作也比较类似，这里就不再列出）：

```
int xxRead(int xxDevId, char * pBuf, int nBytes)
{
    /* 对一些变量进行初始化操作 */
    int ReadLength=ERROR;
    BOOL FoundError;

    /* 获得设备描述符并判断 */
    xx_DEV* pxxDev=(xx_DEV*)xxDevId;
    if (pxxDev==(xx_DEV*)NULL)
    {
        errnoSet(S_xx_NODEV);
        return(ERROR);
    }

    /* 从设备接收 */
    if(pxxDev->CanRead)
    {
        ReadLength=0;
        while(ReadLength<nBytes)
        {
            /* 接收 */
            pBuf[i]=rec_buffer[i];
            i++;
            ReadLength++;
        }
        ... /* 判断状态寄存器、接收到的字符数目 */
        ... /* 根据数据返回状态 */
    }
}
```



```

        if(FoundError) /*非正常接收*/
            return(ERROR);
        return(ReadLength);
    }
    return(ReadLength);
}
-----

```

6.2.5 I/O 控制函数

为了适应不同的设备，VxWorks 的 I/O 系统还提供了 ioctl()接口，用户可以通过这个接口实现与设备相关的其他功能。函数 xxioctl()的形式如下：

```
int xxioctl (deviceId, cmd, arg);
```

deviceId 和读写操作中所使用的参数相同。cmd 是一组整型的值，这组值通常是在驱动程序所使用的头文件中定义。I/O 系统也定义了一些相应的宏，这些宏定义可以在 ioLib.h 中找到。对于不同类型的 I/O 设备，I/O 系统提供了不同的宏定义。arg 参数是最复杂的参数，也是一个整型值，可以根据设备的需要传递整型值或者是指向结构、缓冲区的指针，也可以根据函数所执行的不同功能而决定是否使用这个参数。

下面是一个 I/O 控制 xxioctl()函数实现的一个模板：

```

int xxioctl(int xxDevId, int cmd, int arg)
{
    int status;
    ...
    switch(cmd)
    {
        case xx_STATUS_GET: /* 获得设备状态 */
            status=xxStatusGet(&arg);
            break;
        ...
        case xx_CONTROL_SET: /* 控制设置 */
            status=xxCMDSet(arg);
            break;
        default:
            errno=S_ioLib_UNKNOWN_REQUEST;
            status=ERROR;
    }
    return(status);
}
-----

```



注意

在这个模板中仅仅使用了分支判断，相应的功能还需要用户根据不同的情况实现。

6.2.6 关闭设备函数

关闭操作是打开操作的逆过程，在 xxOpen()中所做的操作都应该在这里进行相应的处理。在关

闭操作中，应该释放那些在打开操作中申请的内存、信号量、消息队列等。但是根据不同的设备，应视情况而定，有些设备是不需要关闭操作的。函数 `xxClose()` 的形式如下：

```
STATUS xxClose (devId);
```

其中，参数 `devId` 是 `open()` 操作所返回的设备标识符。

下面给出了关闭操作函数的一个模板：

```
int xxClose(int xxDevId)
{
    /* 获得设备描述符 */
    xx_DEV* pxxDev = (xx_DEV*)xxDevId;
    if (pxxDev == (xx_DEV *)NULL)
    {
        errnoSet(S_xx_NOMEM);
        return(ERROR);
    }

    ... /*设备相关操作*/
    ... /*释放相关资源*/
    free(pxxDev); /*释放设备描述符指针*/
}
```

6.2.7 VxWorks 系统中支持设备驱动程序的其他函数

除了上面讨论到的函数，VxWorks 还提供了其他的一些函数辅助用户开发，使用这些函数可以执行一些重要的操作。

1. 卸载设备函数

卸载设备是安装设备的逆过程。它所作的操作即是将设备的描述结构从设备列表中删除。卸载设备的系统函数是 `iosDevDelete()`，它是 `iosDevAdd()` 函数的逆向操作。执行了设备卸载之后，所有打开该设备的操作都将被禁止，但是已经打开了的文件则不受任何影响。用户卸载设备函数 `xxDevRemove()` 的形式如下：

```
STATUS xxDevRemove(int arg...);
```

下面是卸载设备函数的一个例子：

```
STATUS xxDevRemove(int chno)
{
    /* 通过 chno 参数，获得设备描述符 */
    xxDEV *pDev = &xxDev[chno];
    pDev->isCreate = FALSE;

    /* 释放信号量等相关资源 */
    semDelete(pDev->IntSemID);

    /* 卸载设备 */
    if (iosDevDelete(&pDev->devHdr) != ERROR)
```



```
return(OK);
else return(ERROR);
}
```

2. 卸载驱动程序函数

卸载驱动程序 `iosDrvRemove(drvNum, forceClose)` 函数的功能是将驱动程序接口从驱动程序表中移除并将设备列表中的相关设备删除。参数 `xxDrvNum` 是被卸载驱动程序的驱动程序索引号。参数 `forceClose` 是一个布尔类型的值，当其取值为 `TRUE` 的时候，该函数将会调用 `xxClose()` 函数关闭所有与该驱动程序相关的已打开的文件；当已经存在打开的文件，而 `forceClose` 又设置为 `FALSE` 的时候，该函数将会返回 `ERROR`。用户卸载驱动程序函数 `xxDevRemove()` 的形式如下：

```
STATUS xxDrvUninstall(int xxDrvNum, BOOL forceClose);
```

下面是卸载驱动程序函数的一个例子：

```
STATUS xxDrvUninstall()
{
    if (xxDrvNum != ERROR) /* 设备驱动程序索引号 */
    {
        /* 根据设备驱动程序索引号卸载驱动程序 */
        if (iosDrvRemove(xxDrvNum, forceClose) == ERROR)
        {
            return(ERROR);
        }
        xxDrvNum = ERROR;
    }
    return (OK);
}
```

注意

`iosDrvRemove()` 函数并不释放分配给设备相关的内存。

3. 获取设备描述符

获得设备描述符函数原型如下：

```
DEV_HDR * iosDevFind( name, pNameTail);
```

参数 `name` 是指设备名；`pNameTail` 是指向字符串的指针，在函数执行后，该参数的值保存了找到的设备名与 `name` 指定的设备名不同的地方。`pNameTail` 的返回值可以是指向字符串的指针，也可以是 `NULL`（空字符），表明找到的设备与指定的名称相吻合；如果返回的指针指向了 `name`，则表示未找到指定设备。

函数 `iosDevFind` 将返回一个指向指定设备的设备描述符头结构（`DEV_HDR`）指针；如果找到的设备的名称仅仅与指定设备名称部分匹配，该函数将会返回找到的设备描述符头结构（`DEV_HDR`）指针并且将参数 `pNameTail` 的内容置为找到的设备名与指定的设备名称上不匹配的部

分；如果系统没有找到指定的设备，也找不到部分匹配的设备，该函数将会返回 NULL，同时将参数 pNameTail 置为指定的设备名。在表 6-2 中列出了这三种情况。

表 6-2 iosDevFind 返回状态表

返回情况	返回值	pNameTail 值
找到指定设备	指定设备的头指针结构	NULL
找到名称部分匹配的设备	找到设备的头指针结构	name 与找到设备名称不吻合的部分
未找到设备	NULL	name

下面是调用 iosDevFind()函数的一个示例：

```
#include "vxWorks.h"
#include "iosLib.h"
...
STATUS FDev (char * devName)
{
    DEV_HDR *pDevHdr;
    char * pNameTail;

    /* 查找设备头数据结构 */
    pDevHdr = iosDevFind (xxdevName, &pNameTail);
    if (pDevHdr == NULL || *pNameTail != '\0')
        return (ERROR);

    /* 删除设备 */
    iosDevDelete (pDevHdr);
    return (OK);
}
```

4. 文件描述符转换成设备 ID

有些情况下，用户会希望由文件描述符获取设备的 ID 以确认该文件所使用的设备及其驱动程序。这时可以使用下面的函数：

```
int iosFdValue( fd);
```

该函数会由文件描述符 (fd) 查找相应的设备 ID。这个 ID 通常是由 xxOpen()所返回的。通过这个 ID 可以访问设备相关的结构。如果 fd 是无效的将会返回 ERROR。相应的例子参见 6.2.3 节。

6.2.8 使字符型设备支持 select 功能

对于字符型设备，函数 select()是一个非常有效的工具。select()函数允许任务同时等待多个文件的操作。任务使用了 select()功能后，如果多个设备中没有一个是可读或可写的，则任务阻塞；但是如果任何一个设备可以操作了，任务就会为其服务。

select()函数使用了一个 fd_set 的结构实现多文件描述符的处理，这个结构看起来像一个由很多位 (bit) 组成的数组，fd_set 结构如图 6.12 所示。

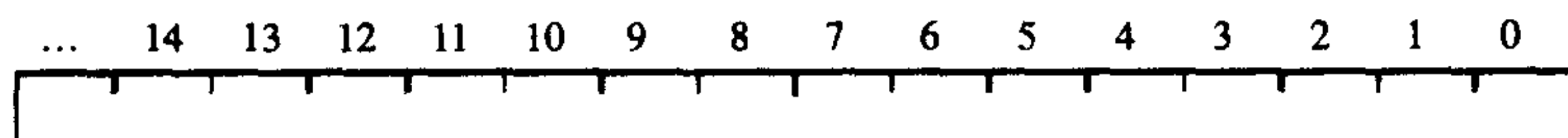


图 6.12 fd_set 结构

fd_set 结构中的每一位对应一个文件描述符。使用 FD_SET、FD_CLR、FD_ZERO、FD_ISSET 等宏定义可以对该结构进行操作。select()与驱动程序的结合如图 6.13 所示。

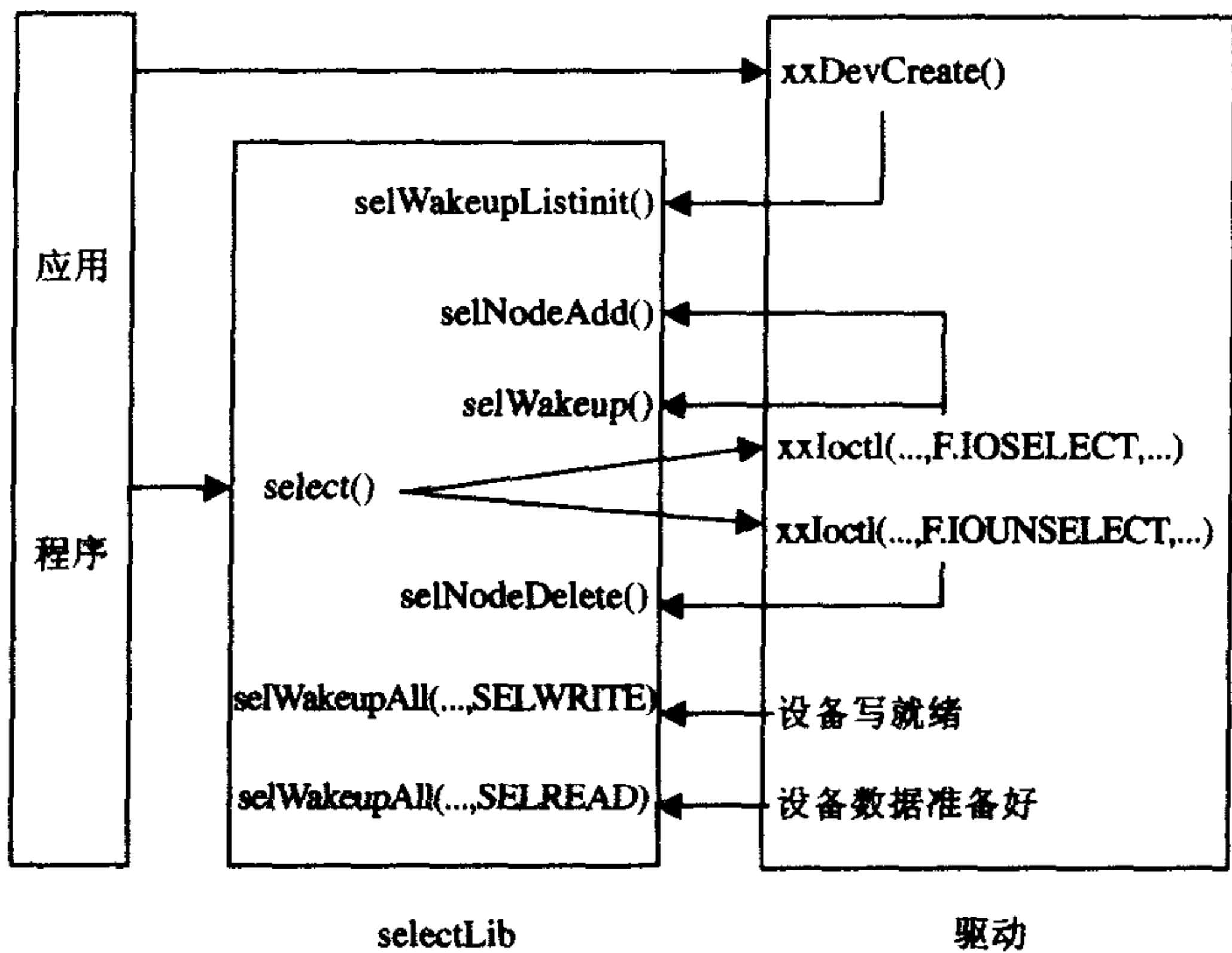


图 6.13 select()与应用及驱动程序的关系

根据图 6.13 给出的形式编程就可以使驱动程序支持 select()功能。

1. select()的初始化

在驱动程序支持 select()函数之前，应在驱动程序中声明 select()函数的结点并将其初始化。首先应在驱动程序中声明一个 SEL_WAKEUP_LIST 的变量，如下所示：

```
SEL_WAKEUP_LIST *pWakeupList;
```

其次，应在设备创建时，使用 selWakeupListInit()执行结点的初始化。如下：

```
selWakeupListInit(&pxxDev->selWakeupList );
```

初始化的过程使系统可以对相应的设备使用 select 功能。经过初始化，驱动程序可以在设备就绪的时候通知系统。

下面是 select()支持的初始化例子：

```
/* 设备数据结构 */
typedef structure
{
    DEV_HRD devHdr;
    ... /*其他必要的代码*/
    SEL_WAKEUP_LIST selWakeupList;
}
```



```

... /*其他必要的代码*/
}xx_DEV;

STATUS xxDevCreate(char * devName)
{
    ...
    if((pxxDev=(xx_DEV *)malloc(sizeof(xx_DEV)))==NULL)
        return(ERROR);
    bzero(pxxDev, sizeof(xx_DEV));
    selWakeupListInit(&pxxDev->selWakeupList);
    ...
}
-----

```

2. 读写响应

经过初始化之后，所需做的步骤即是在设备就绪时通知系统将任务恢复到原来状态。在驱动程序中可以使用 `selWakeupAll()` 函数来将设备就绪的信息通知系统。如下调用：

```

selWakeAll( pWakeupList, SELWRITE);
selWakeAll( pWakeupList, SELREAD);

```

函数 `selWakeupAll()` 有两个参数，第一个用于指定所操作的 `select()` 节点，第二个参数指定设备是读就绪或是写就绪状态。

3. 其他必要的代码

根据前面看到的 `select()` 功能与驱动程序的结合图示，在调用 `select()` 函数时，`select()` 函数会调用设备驱动程序中的 `ioctl()` 函数执行必要的命令。实际中，为了使用 `select` 功能，驱动程序的 `xxioctl()` 函数应实现两个命令：`FIOSELECT` 和 `FIOUNSELECT`。这两个功能分别实现了将设备加入 `select()` 等待列表和将设备从等待列表中删除，这两个功能的示例如下：

```

case FIOSELECT:
    selNodeAdd(&pxxDev->selWakeupList, (SEL_WAKEUP_NODE *)arg);
    if((selWakeupType((SEL_WAKEUP_NODE)arg)==SELREAD)&&(&pxxDev->ReadyToRead))
        selWakeup((SEL_WAKEUP_NODE *)arg);
    if((selWakeupType((SEL_WAKEUP_NODE)arg)==SELWRITE)&&(&pxxDev->ReadyToWrite))
        selWakeup((SEL_WAKEUP_NODE *)arg);
    break;
case FIOUNSELECT:
    selNodeDelete(&pxxDev->selWakeupList, (SEL_WAKEUP_NODE *)arg);
    break;
-----

```

6.3 字符设备驱动程序的完整模板

经过前面的描述，读者可能对 VxWorks 中字符型设备的驱动程序编写有了一定的了解。在这一节里将总结前面的例子给出一个字符型设备通用的模板。根据这个模板，驱动程序的编写者可以方便地写出一个字符型设备的驱动程序。

1. 设备描述符结构

```

LOCAL int xxDrvNum=0; /* 驱动程序索引号 */

```



```
typedef struct
{
    DEV_HRD devHdr;          /* 设备头数据结构 */
    BOOL      isCreate;      /* 设备创建标志 */
    BOOL      isOpen;        /* 设备打开标志 */
    UINT32     RegMEMBase;    /* 设备存储器基地址 */
    UINT32     ioAddr;        /* 设备 I/O 基地址 */
    ...
    SEL_WAKEUP_LIST selWakeupList; /* select() 功能 */
    BOOL ReadyToRead;          /* 设备读就绪 */
    BOOL ReadyToWrite;         /* 设备写就绪 */
} xx_DEV;
```

2. 设备驱动程序装载函数

```
STATUS xxDrv()
{
    /* 首先判断驱动程序是否已经安装 */
    if (xxDrvNum > 0)
        return (OK);

    ... /* 在这里添加驱动程序的初始化部分 */

    /* 将驱动程序添加到驱动程序描述表中 */
    if ((xxDrvNum = iosDrvInstall(xxOpen, NULL, xxOpen,
        xxClose, xxRead, xxWrite, xxIoctl)) == ERROR)
    {
        return (ERROR);
    }
    return (OK);
}
```

3. 设备创建函数

```
STATUS xxDevCreate(char * devName)
{
    xx_DEV *pxxDev;

    /* 检查驱动程序是否安装 */
    if (xxDrvNum < 1)
    {
        errno = S_ioLib_NO_DRIVER;
        return (ERROR);
    }

    /* 分配内存并初始化 */
    if ((pxxDev = (xx_DEV *) malloc(sizeof(xx_DEV))) == NULL)
        return (ERROR);
    bzero(pxxDev, sizeof(xx_DEV));
    selWakeupListInit(&pxxDev->selWakeupList);

    ... /* 初始化必要的描述结构 pxxDev */
}
```



```
... /*执行设备硬件的初始化,例如: I/O 地址、设备本地内存地址等 */
```

```
/*将设备添加到设备列表*/
```

```
if(iosDevAdd(&pxxDev->devHdr, devName, xxDrvNum)==ERROR)
{
    free((char *)pxxDev); /* 释放设备描述符 */
    return(ERROR);
}
return(OK);
}
```

4. 打开函数

```
int xxOpen(DEV_HDR *pxxDevHdr, int option, int flags)
{
    xx_DEV* pxxDev=(xx_DEV *)pxxDevHdr;
    if(pxxDevHdr==NULL)
    {
        errnoSet(S_xx_NODEV);
        return(ERROR);
    };

    /* 判断设备是否打开 */
    if(pxxDev->Opened)
    {
        errnoSet(S_xx_DEVOPENED);
        return(ERROR);
    }
    pxxDev->Opened=TRUE;

    ... /*在这里作必要的初始化*/
    return((int)pxxDevHdr);
}
```

5. 读函数

```
int xxRead(int xxDevId, char * pBuf, int nBytes)
{
    xx_DEV* pxxDev=(xx_DEV*)xxDevId;
    int ReadLength=ERROR;
    BOOL FoundError;

    /*设备描述符是否为空 */
    if (pxxDev==(xx_DEV*)NULL)
    {
        errnoSet(S_xx_NODEV);
        return(ERROR);
    }

    /* 判断设备是否就绪 */
    if(pxxDev->ReadyToRead)
    {
        ReadLength=0;
    }
}
```



```

while(ReadLength<nBytes)
{
    /*从设备接收*/
    ReadLength++;
}
... /*判断状态寄存器、接收到的字符数目*/
... /*根据数据返回状态*/
if(FoundError)/*非正常接收*/
    return(ERROR);
return(ReadLength);
}
return(ReadLength);
}

```

6. 写函数

```

int xxWrite(int xxDevId, char * pBuf, int nBytes)
{
    xx_DEV* pxxDev=(xx_DEV *)xxDevId;
    BOOL FoundError;
    int WriteLength=0;

    /*设备描述符是否为空 */
    if (pxxDev==(xx_DEV *)NULL)
    {
        errnoSet(S_xx_NODEV);
        return(ERROR);
    }

    /* 设备写就绪 */
    if(pxxDev->ReadyToWrite)
    {
        pxxDev->ReadyToWrite=FALSE;
        ... /* 向设备发送处理 */
        ... /* 判断相关状态 */
    }
    pxxDev->ReadyToWrite=TRUE;
    if(FoundError)
        return(ERROR);
    return(WriteLength); /* 返回发送字符长度 */
}

```

7. I/O 控制函数

```

int xxIoctl(int xxDevId, int cmd, int arg)
{
    int status;
    xx_DEV *pxxDev=(xx_DEV *)xxDevId;

    switch(cmd)
    {
        case FIOSELECT:
            selNodeAdd(&pxxDev->selWakeupList, (SEL_WAKEUP_NODE *)arg);

```



```

if((selWakeupType((SEL_WAKEUP_NODE *)arg)==SELREAD)&&(&pxxDev->ReadyToRead))
    selWakeup((SEL_WAKEUP_NODE *)arg);
if((selWakeupType((SEL_WAKEUP_NODE *)arg)==SELWRITE)&&(&pxxDev->ReadyToWrite))
    selWakeup((SEL_WAKEUP_NODE *)arg);
    break;
case FIOUNSELECT:
    selNodeDelete(&pxxDev->selWakeupList, (SEL_WAKEUP_NODE *)arg);
    break;
case xx_STATUS_GET:
    status=xxStatusGet(&arg);
    break;
case xx_CONTROL_SET:
    status=xxCMDSet(arg);
    break;
.../*其他的命令实现*/
default:
    errno=S_ioLib_UNKNOWN_REQUEST;
    status=ERROR;
}
return(status);
}

```

8. 设备关闭函数

```

int xxClose(int xxDevId)
{
    xx_DEV* pxxDev=(xx_DEV*)xxDevId;
    if(pxxDev==(xx_DEV *)NULL)
    {
        errnoSet(S_xx_NOMEM);
        return(ERROR);
    }
    ... /*设备相关操作*/
    ... /*释放相关资源*/
    free(pxxDev); /* 释放资源*/
}

```

9. 卸载设备函数

```

STATUS xxDelete(char *devName)
{
    DEV_HDR *pDevHdr;
    char * pNameTail;
    /* 查找设备 */
    pDevHdr=iosDevFind(devName, &pNameTail);
    if(pDevHdr==NULL||*pNameTail!='\0')
        return(ERROR);

    ... /* 释放设备所占用的资源, 例如: 信号量、唤醒等待该设备的任务等 */
}

```



```
/* 卸载设备 */
iosDevDelete(pDevHdr);
return(OK);
}
```

10. 设备中断管理函数

```
LOCAL ULONG xxIntHandler(int xxDevId)
{
    xx_DEV* pxxDev=(xx_DEV*)xxDevId;

    ... /*读取中断状态*/

    /*如果可以接收*/
    pxxDev->ReadyToRead=TRUE;

    /*如果可以发送*/
    pxxDev->ReadyToWrite=FALSE;
    ... /*清除相应中断*/
}
```

小 结

这一章首先回顾了一下 VxWorks 中的 I/O 系统与设备驱动程序及文件的关系，并讨论了 VxWorks 中字符型设备驱动程序的编写，使读者对这一类型驱动程序的编写有一个较深入的了解。最后给出了字符型设备的驱动程序模板，希望能对开发者有所帮助，使驱动程序开发人员可以将精力集中在设备的操作上。

第 7 章 编写串行设备驱动程序

在 VxWorks 中，串行设备是一种特殊的字符型设备。与字符型设备不同的是，串行设备的驱动程序并不是直接挂在 I/O 系统中，而是由虚拟设备 ttyDrv 来使用。在这一章中，我们将讨论串行设备的特点、功能以及如何编写串行设备驱动程序。最后，依照第 6 章，将给出一个串行设备驱动程序的模板，通过模板可以方便的创建各种串行设备驱动程序。

7.1 串行设备概述

在 VxWorks 中，串行设备是一种特殊的设备。它既支持 VxWorks 下的 I/O 系统，同时又支持目标机代理（Target Agent）的接口；它必须既可以以中断方式工作，又可以在轮询模式下工作。除此之外，串行驱动程序还要实现一些与设备无关的功能：将<CR>的输出映射成<CR><LF>输出；提供命令行编辑能力；对输入、输出的数据进行缓冲等。因为支持 Target Agent 接口，所以可以使用串行设备实现 Tornado 的目标机服务器（Target Server）与 VxWorks 的 Target Agent 之间的通信。如图 7.1 所示的串行设备驱动程序的工作模式。

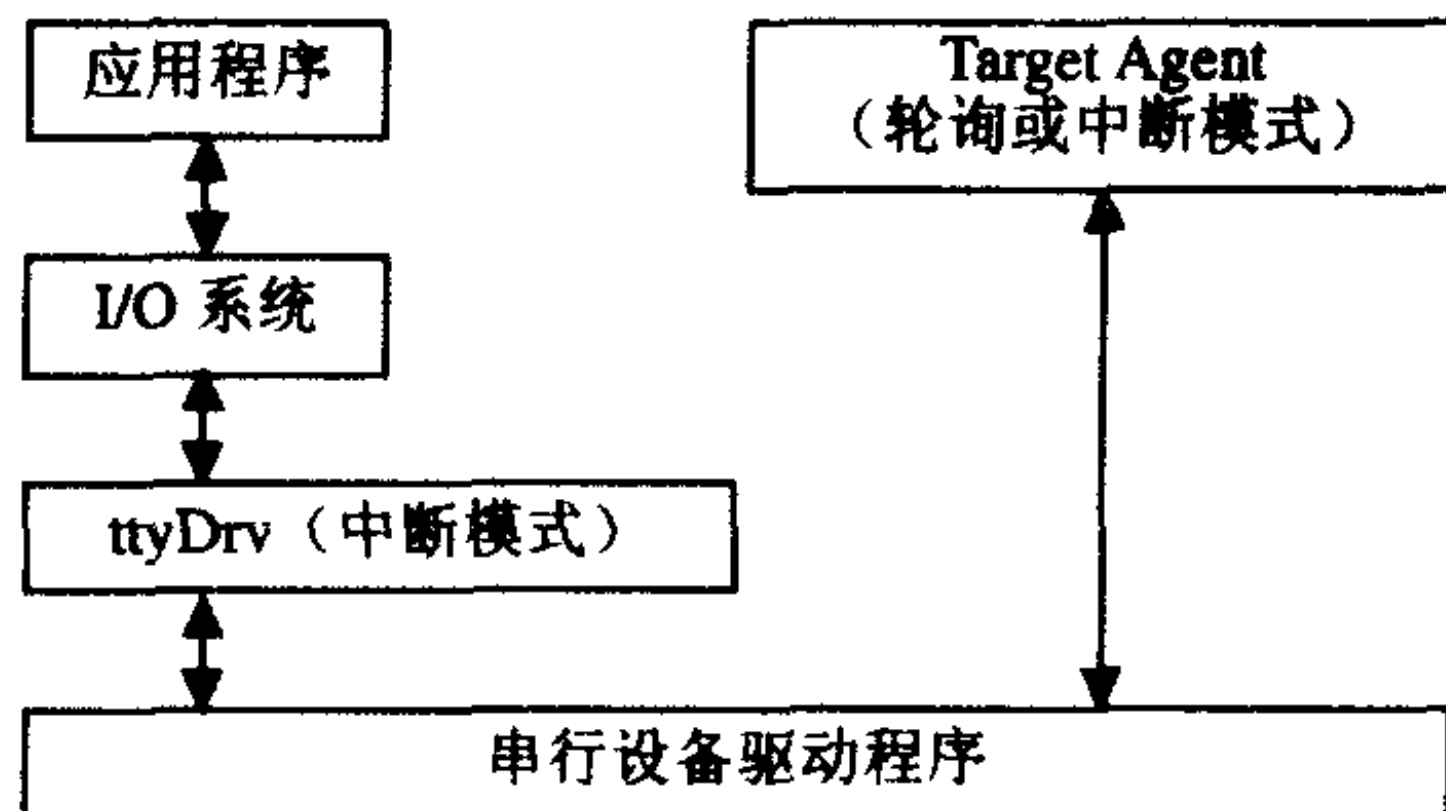


图 7.1 串行设备驱动程序的工作模式

幸运的是，与设备无关的部分都已经由 Wind River 在 VxWorks 的虚拟设备 ttyDrv 中实现，驱动程序的编写者要做的仅仅是根据系统给出的接口实现驱动程序并将其挂接到虚拟设备 ttyDrv 上。



注意

虚拟设备 ttyDrv 在串行设备驱动程序中充当了重要的角色。ttyDrv 处在 I/O 系统和真实驱动程序之间，形成了一个转换层，ttyDrv 的作用如图 7.2 所示。ttyDrv 为系统提供了标准的 I/O 接口，同时提供了那些与设备无关的功能。作为一个虚拟设备 ttyDrv 可以管理多个串行设备驱动程序。有关于 ttyDrv 设备，将在下面一节中详细讨论。

图 7.2 中的 `xxDrv`、`yyDrv` 是真实的串行设备驱动程序。与字符型设备相同，它提供了标准的 I/O 接口，与此同时，它还提供了其他的功能，与字符型设备有很多不同之处。它提供了目标机代理 (Target Agent) 的接口，可以用于目标机与主机之间的通信，同时它必须可以在中断和轮询两种模式下工作。

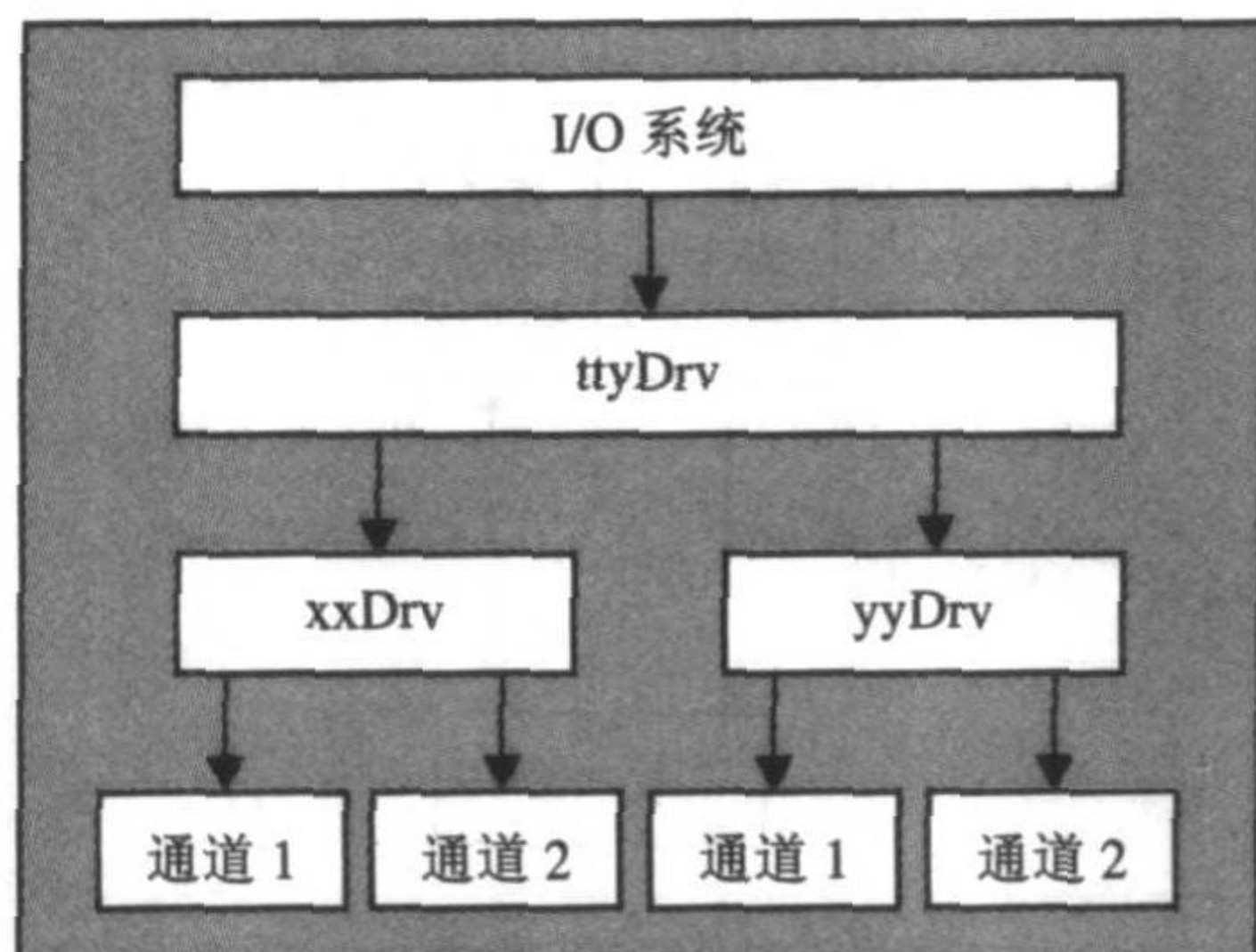


图 7.2 ttyDrv 的作用

串行设备驱动程序的入口点与字符型设备不同，它的驱动程序函数不是安装在驱动程序表中，而是通过回调方式挂在虚拟设备 `ttyDrv` 上。由于驱动程序的安装方式不同，系统也就不会将串行设备驱动程序的设备描述符挂在设备列表上，也就是说，I/O 系统是看不到这个设备的。虽然有这么多特殊的地方，在实现的过程中串行设备还是与字符型设备有很多相似之处：串行设备可以支持 `select` 功能，可以实现 `FIOSETBAUD` 等功能，实现数据缓冲等。



注意

`xxOpen()`、`xxRead()` 等接口也是需要实现的。

7.2 虚拟设备 ttyDrv

虚拟设备 `ttyDrv` 管理着 I/O 系统和真实驱动程序之间的通信。`ttyDrv` 的职责如图 7.3 所示。

在 I/O 系统方面，虚拟设备 `ttyDrv` 作为一个字符型设备存在，它将自身的入口点函数挂在 I/O 系统上（参考本书第 6 章），创建设备描述符并将其加入到设备列表中。当用户有 I/O 请求包到达 I/O 系统中时，I/O 系统会调用 `ttyDrv` 相应的函数响应请求。`ttyDrv` 对于 `select()` 函数也有相应的处理。同时，`ttyDrv` 管理了缓冲区的互斥，提供了任务的同步操作。在 I/O 系统看来，与之打交道的是设备 `ttyDrv`。

另一方面，`ttyDrv` 负责与实际的设备驱动程序交换信息。通过设备驱动程序提供的回调函数及必要的数据结构，`ttyDrv` 将系统的 I/O 请求作相应的处理后，传递给设备驱动程序，由设备驱动程

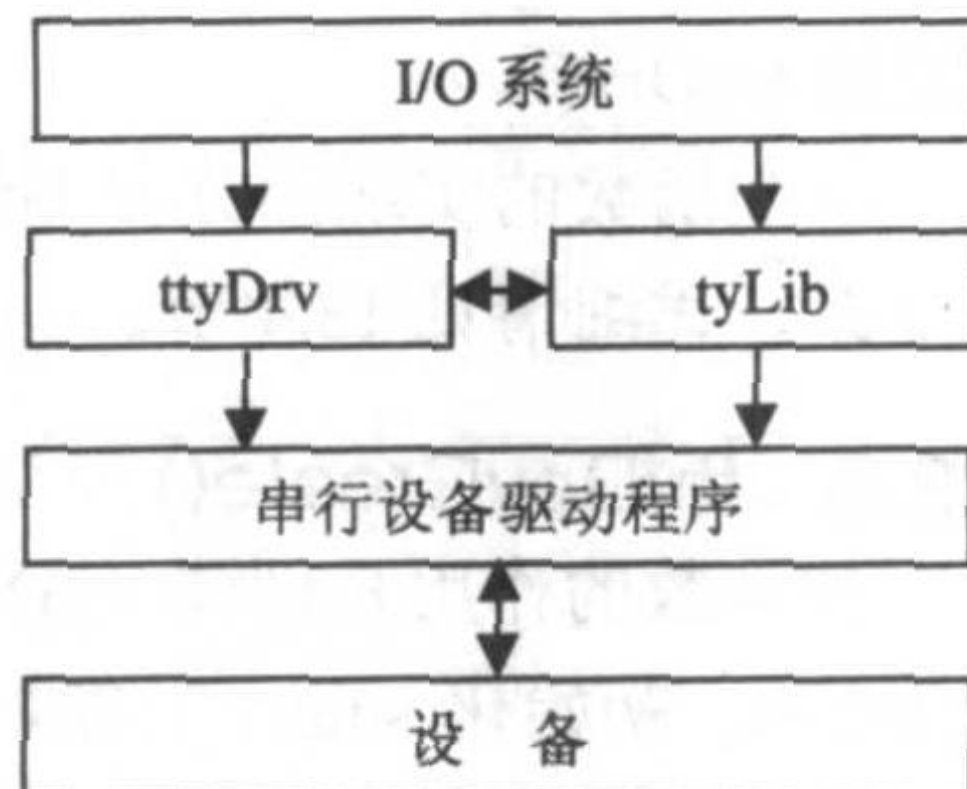


图 7.3 ttyDrv 的职责

序来完成实际的 I/O 操作。图 7.4 的 ttyDrv 数据流中给出了这一过程中数据流向。

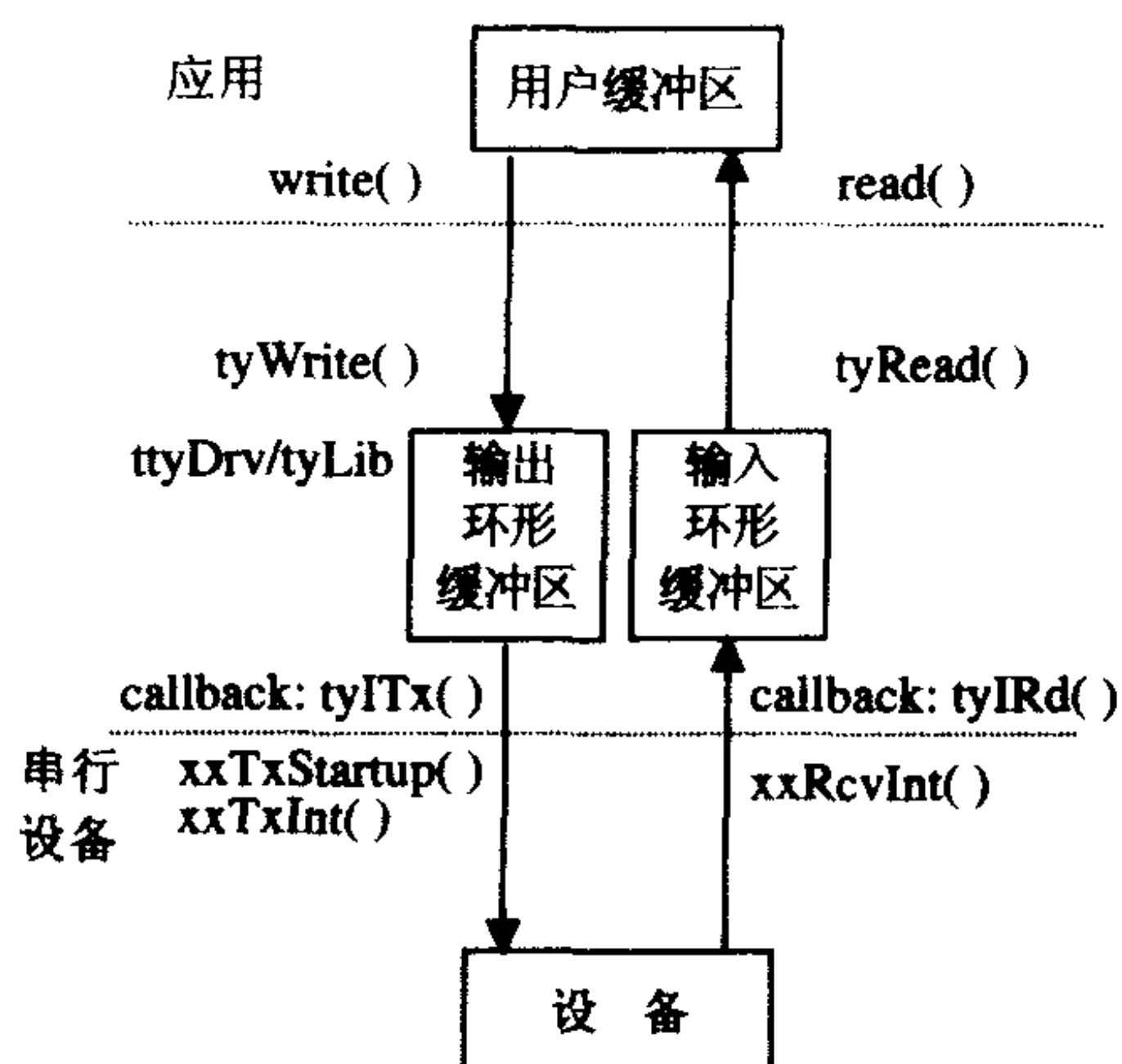


图 7.4 ttyDrv 数据流

7.2.1 ttyDrv 的实现

作为一个字符型设备，ttyDrv 使用了与字符设备相同的实现：通过 ttyDrv()调用 iosDrvInstall()函数安装驱动程序；通过调用 ttyDevCreate()创建设备；使用 tyRead()和 tyWrite()实现设备的读写操作；使用 ttyIoctl()完成其他功能。下面将分别讨论这五个函数。

注意

VxWorks 系统在 `..target\config\all\usrConfig.c` 文件中根据条件调用 ttyDrv()和 ttyDevCreate()。

7.2.2 ttyDrv()函数

ttyDrv()函数的目的是将驱动程序函数入口点加入到驱动程序表中。该函数通过调用 iosDrvInstall()将 ttyOpen()、ttyIoctl()、tyRead()、tyWrite()安装到系统驱动程序表中，供 I/O 系统调用。如果在 BSP 中未定义 INCLUDE_TYCODRV_5_2，系统将在 usrConfig.c 中的 usrRoot 任务中自动调用 ttyDrv()。

这个函数的实现部分在 Tornado/target/config/targetBSP/sysSerial.c 中，如果有特殊的需要可以对其更改，详细请见 13.4 节。

7.2.3 ttyDevCreate()

该函数的作用是创建 ttyDrv 设备。首先它会初始化设备描述符，接着调用 tyDevInit()函数初始化 tyLib、初始化 select 功能、创建信号量和输入输出缓冲区。之后，调用 iosDevAdd()函数将设备添加到设备列表中并将设备置为中断工作模式。

ttyDevCreate()函数也是由 usrConfig.c 的 usrRoot 任务调用的。根据宏 NUM_TTY 的值，usrRoot()函数调用 ttyDevCreate()函数创建相应数量的设备。与 ttyDrv()函数相同，ttyDevCreate()函数也是在 Tornado/target/config/targetBSP/sysSerial.c 中实现的。

7.2.4 tyWrite()和 tyRead()函数

细心的读者可能已经注意到，前面两个函数的前缀都是 tty，而 tyWrite()和 tyRead()函数的前缀是 ty，那是因为 tyWrite()和 tyRead()函数都是 tyLib 库中的一部分。在这里 ttyDrv()函数将 tyWrite()和 tyRead()函数作为 ttyDrv 的驱动程序函数入口。

tyWrite()函数控制着一个环形缓冲区。调用时，tyWrite()函数会将数据从用户缓冲区复制到该缓冲区。如果环形缓冲区已经填满，该函数将导致调用它的任务阻塞。I/O 系统串行设备写过程如图 7.5 所示。

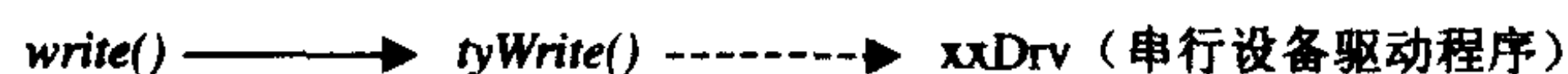


图 7.5 串行设备写操作流程

tyRead()函数同样控制着一个缓冲区，如果缓冲区空，函数会导致调用它的任务阻塞；如果缓冲区满，函数会将等待的任务唤醒。同时，tyRead()函数会处理 Xon/Xoff。I/O 系统串行设备读过程如图 7.6 所示。



图 7.6 串行设备读操作流程

7.2.5 ttyIoctl()函数

ttyIoctl()函数响应 I/O 系统 ioctl()的调用。当调用 ttyIoctl()函数时，这个函数首先访问实际驱动程序所提供的 xxIoctl()函数，如果所需的功能在 xxIoctl()函数中未提供，ttyIoctl()函数还会继续调用 tyIoctl()函数，如果 tyIoctl()函数执行也失败，ttyIoctl()最终会返回失败。I/O 系统串行设备控制过程如图 7.7 所示。

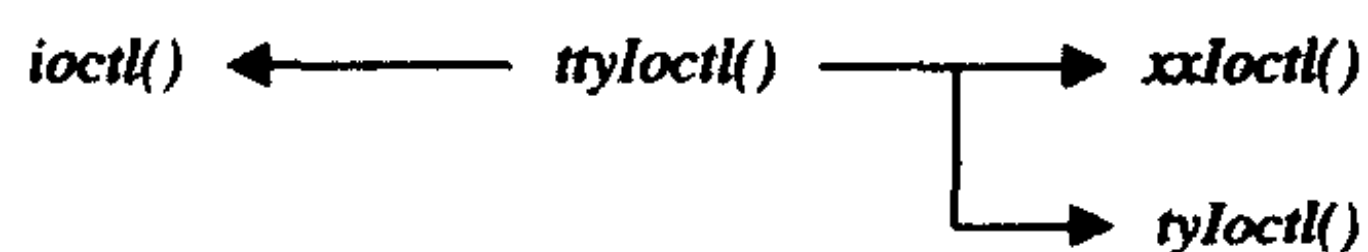


图 7.7 串行设备控制操作流程

下面列出了 tyIoctl()所支持的功能：

- FIONREAD，获取缓冲区中可读取的字符数；
- FIONWRITE，获取缓冲区中将要写入的字符数；
- FIOFLUSH，清除所有缓冲区中所有字符；
- FIOWFLUSH，清除写缓冲区中所有字符；
- FIORFLUSH，清除读缓冲区中所有字符；
- FIOSETOPTIONS，设置串行设备相关选项；
- FIOGETOPTIONS，获取串行设备相关选项；
- FIOCANCEL，取消对设备的读写操作；
- FIOISATTY，是否是 tty 设备；
- FIOSELECT，select 功能所指定的命令；
- FIOUNSELECT，select 功能所指定的命令。

上面简要地介绍了 `ttyDrv`, 同时也带出了一个 `ttyDrv` 和 `tyLib` 之间的关系。图 7.8 中显示了 `ttyDrv`、`tyLib`、I/O 系统和实际驱动程序之间的关系。

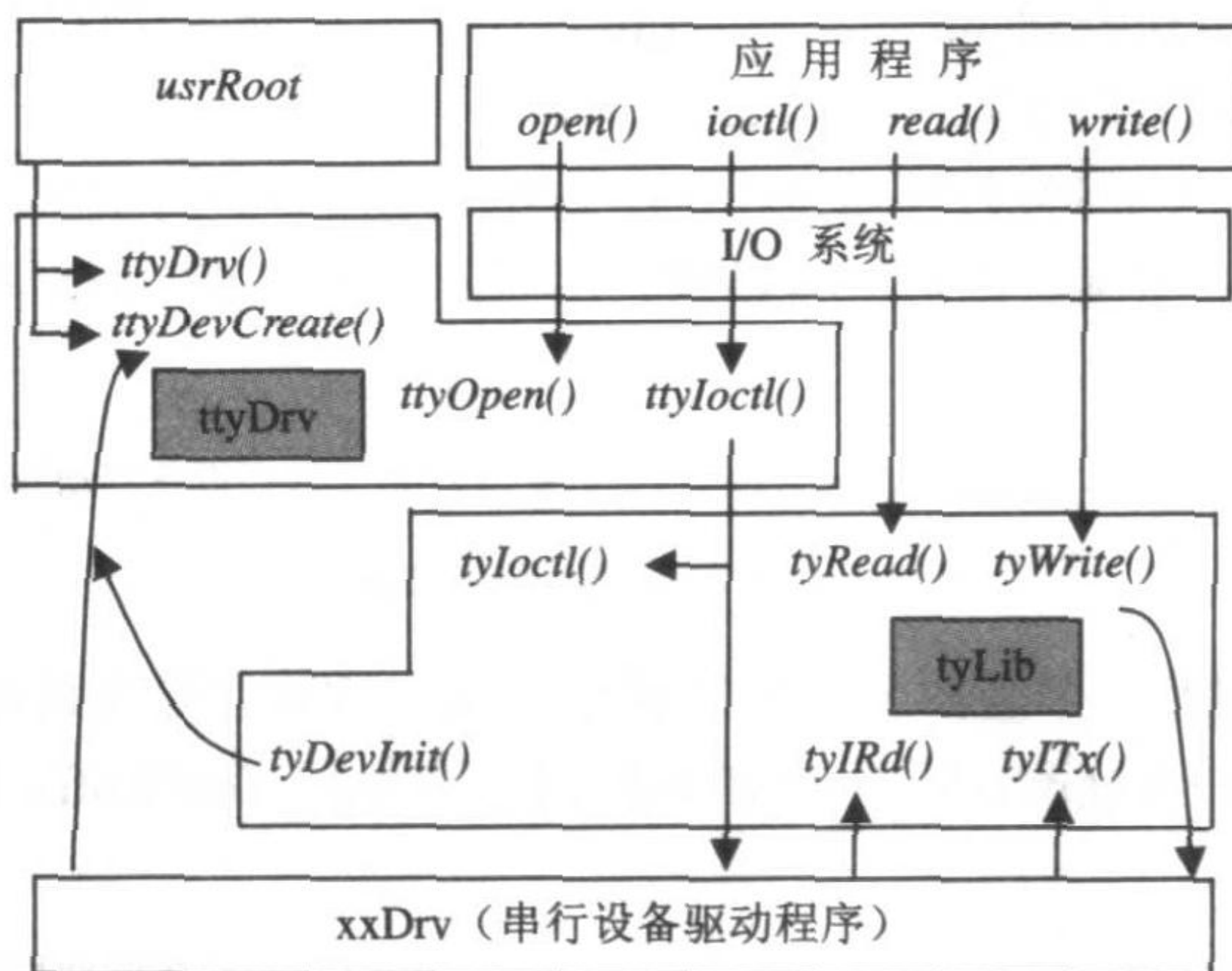


图 7.8 ttyDrv 与 tyLib 之间的关系

7.3 编写串行设备驱动程序

通过前面的描述，我们知道串行设备驱动程序不但要支持标准的 I/O 系统接口，还要支持目标机代理 (Target Agent) 的接口；既要支持中断模式，也要支持轮询模式；需要为虚拟设备 ttyDrv 提供回调函数。总的来说写一个串行设备驱动程序比较繁杂，所以在 VxWorks 中提供了一个模板 (Tornado/target/src/drv/sio/templateSio.c)，用户可以根据需要更改模板，支持新的设备。

编写一个串行设备驱动程序通常要实现初始化、回调函数及其所需要的结构体、回调安装函数、发送接收函数、I/O 控制函数和中断服务程序，下面将分别讨论这些函数，一步步地实现串行设备驱动程序。

7.3.1 初始化代码

串行设备驱动程序的初始化是在系统引导代码中执行的，VxWorks 系统启动代码中串行驱动程序的初始化如图 7.9 所示。

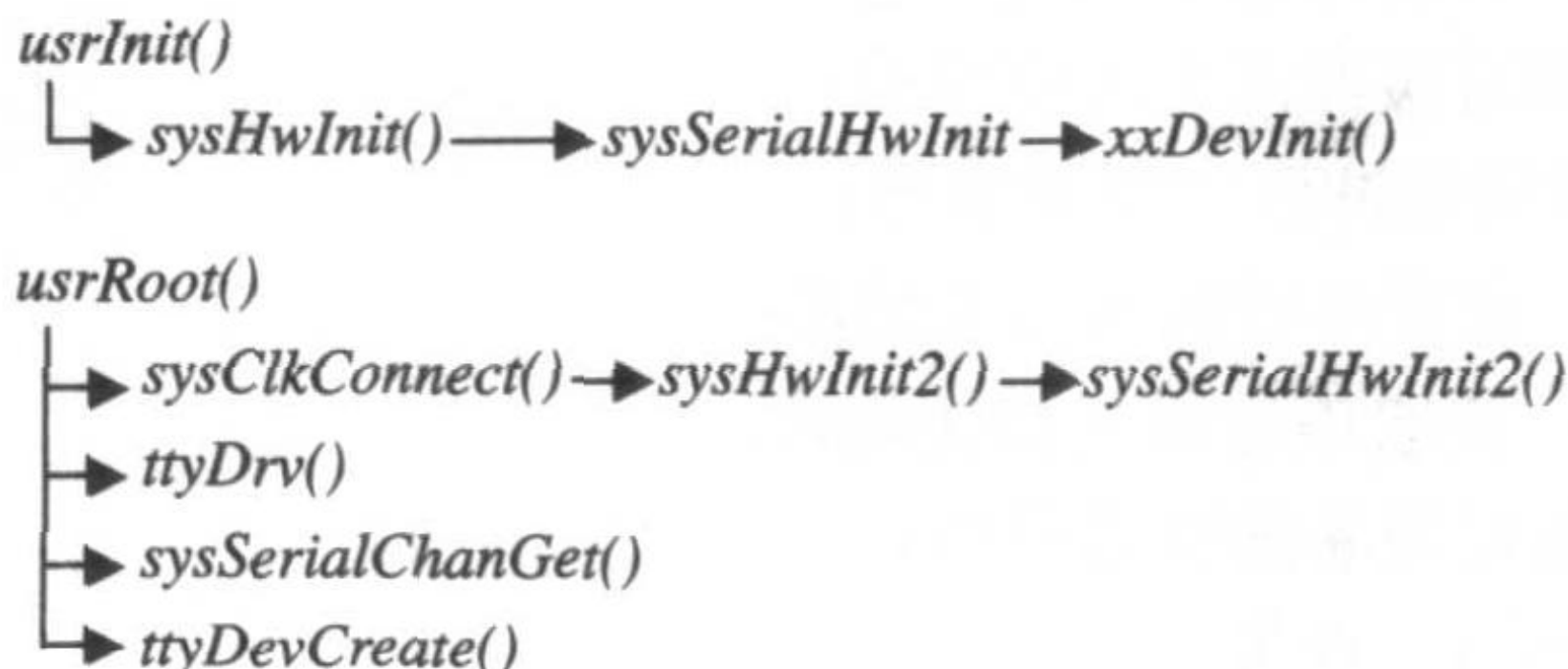


图 7.9 VxWorks 系统启动代码中串行驱动程序的初始化

串行设备的初始化代码在 `usrInit()` 函数中, 由函数 `sysHwInit()` 通过调用 `sysSerialHwInit()` 函数来调用它。在初始化中应定义设备通道数目, 初始化设备描述符并执行设备相关的初始化代码。

与字符型设备相似, 串行设备驱动程序也有一个重要且必要的结构。在这个结构 (`SIO_DRV_FUNCS`) 中指出了驱动程序函数的入口点, 它的定义如下:

```
typedef struct sio_drv_funcs SIO_DRV_FUNCS;

struct sio_drv_funcs          /* 驱动程序所需的函数 */
{
    int      (*ioctl)(SIO_CHAN * pSioChan, int cmd, void * arg );
    int      (*txStartup)(SIO_CHAN * pSioChan);
    int      (*callbackInstall)( SIO_CHAN * pSioChan, int callbackType,
        STATUS (*callback)(), void * callbackArg );
    int      (*pollInput)(SIO_CHAN * pSioChan, char * inChar);
    int      (*pollOutput)(SIO_CHAN * pSioChan, char outChar );
};
```

在串行设备的驱动程序中, 必须要有一个包含指向 `SIO_DRV_FUNCS` 指针的结构 (`xx_CHAN`)。这个结构中可以包含设备的其他信息以及提供给高层协议的回调函数。下面是 `XX_CHAN` 的一个例子:

```
typedef struct          /* xx_CHAN */
{
    SIO_DRV_FUNCS * pDrvFuncs;          /* 驱动程序所需的函数 */
    STATUS      (*getTxChar) ();          /* 传送函数指针 */
    STATUS      (*putRcvChar) ();          /* 接收函数指针 */
    void *      getTxArg;
    void *      putRcvArg;
    UINT16      int_vec;                  /* 中断向量 */
    UINT16      channelMode;              /* 模式中断或轮询 */
    UCHAR      (*inByte) (int);           /* 从硬件读取一个字节函数指针 */
    void      (*outByte) (int, char);      /* 向硬件写入一个字节函数指针 */
    ... /* 必要的寄存器描述及结构 */
} xx_CHAN;
```

所有在 `xx_CHAN` 中定义的成员使串行设备驱动程序和高层协议间的信息传输成为可能, 但是系统并不知道用户所使用的数据结构。因此, `VxWorks` 系统为用户提供了另一个结构——`SIO_CHAN`。

```
typedef struct sio_chan          /* 串行通道数据结构 */
{
    SIO_DRV_FUNCS * pDrvFuncs;
    /* device data */
} SIO_CHAN;
```

该结构原型在 `../target/h/sioLib.h` 文件中。

在系统的定义中 `SIO_CHAN` 结构仅包含了一个指向结构 `SIO_DRV_FUNCS` 的指针, 所以在使用中可以进行如下代换:


```
SIO_CHAN * pSioChan;  
xx_CHAN *pxxChan=(xx_CHAN *)pSioChan;
```

通过这样的代换，系统可以在不修改的情况下使用不同的设备驱动程序。

在实际运行中，虚拟设备 ttyDrv 通过 xxDrv 提供的 SIO_DRV_FUNCS 中的函数入口点请求服务，而 xxDrv 则通过回调函数向 ttyDrv 请求服务。

串行设备驱动程序的初始化函数应该包含下面的内容：

- 包含一个指向 xx_DRV 结构的指针；
- 初始化 xx_CHAN；
- 初始化驱动程序必要的内容；
- 重新设置芯片。

下面是一个串行设备驱动程序的初始化模板：

```
static SIO_DRV_FUNCS xxSioDrvFuncs;  
  
void xxDevInit  
(  
    xx_CHAN * pChan      /* 设备描述符指针 */  
)  
{  
    /* 初始化设备驱动程序数据结构 */  
    if (xxSioDrvFuncs.ioctl == NULL)  
    {  
        xxSioDrvFuncs.ioctl      = (int (*)( ))xxIoctl;  
        xxSioDrvFuncs.txStartup   = (int (*)( ))xxStartup;  
        xxSioDrvFuncs.callbackInstall = xxCallbackInstall;  
        xxSioDrvFuncs.pollInput   = (int (*)( ))xxPRxChar;  
        xxSioDrvFuncs.pollOutput  = (int (*)(SIO_CHAN *,char))xxPTxChar;  
    }  
    pChan->pDrvFuncs = &xxSioDrvFuncs;  
  
    ... /* 必要的硬件初始化 */  
}
```

7.3.2 回调安装函数

通过阅读 SIO_DRV_FUNCS 结构，用户需要提供表 7-1 中所列举的函数入口点。

表 7-1 SIO_DRV_FUNCS 结构包含的驱动程序入口点函数

函 数	描 述
xxCallBackInstall()	回调安装函数
xxPollOutput()	轮询模式输出函数
xxPollInput()	轮询模式输入函数
xxIoctl()	设备特殊控制函数
xxTxStartup()	启动一个发送周期函数

根据上表，在串行设备驱动程序中应该提供一个回调安装函数，该函数的原型如下：

```
int xxCallbackInstall( pSioChan, callbackType, callback, callbackArg);
```

其中，pSioChan 是一个指向 xx_CHAN 结构的指针，callbackType 的取值是 SIO_CALLBACK_GET_TX_CHAR 和 SIO_CALLBACK_PUT_RCV_CHAR，用于指出安装传送回调函数还是安装接收回调函数。callback 是指向回调函数的指针，callbackArg 是回调函数所需的参数。

这个函数初始化 SIO_CHAN 中的部分成员，下面是该函数的一个实现模板：

```
static int xxCallbackInstall
(
    SIO_CHAN * pSioChan,
    int         callbackType,
    STATUS      (*callback)(),
    void *      callbackArg
)
{
    xx_CHAN * pChan = (xx_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);
    }
}
```

7.3.3 启动一个发送周期

当应用使用 ttyDrv 进行写操作时，I/O 请求包将会通过 I/O 系统传给 ttyDrv，使 tyWrite() 开始运行。tyWrite() 开始执行后，将会把用户数据拷贝到 ttyDrv 的环形缓冲区中，并调用 xxTxStartup() 开始一个发送周期。通常这个周期使用中断方式。每一次中断，设备都会通过中断服务程序发送一个或多个字节。

xxTxStartup() 的任务包括初始化发送循环，开启中断准备发送。该函数原型如下：

```
void xxTxStartup (pSioChan);
```

下面是该函数的模板，在模板中作了一个判断，即是否有字符从高层协议传来，如果有则开始传送并使能相应的中断。

```
LOCAL int xxTxStartup(SIO_CHAN * pChan)
{
    xx_CHAN * pxxChan = (xx_CHAN *)pChan;
    char ier = xx_IER_RXRDY;
```



```

char status;

if (pxxChan->channelMode == SIO_MODE_INT)
{
    /*根据硬件寄存器的值判断状态*/
    status = ((*pxxChan->inByte) (pxxChan->sta));

    /*如果状态正确，打开中断*/
    if(status==xx_OK)
        (*pxxChan->outByte) (pxxChan->ier, ier);
}
return (OK);
}

```

在这里有必要提到另一个函数，因为这个函数也是发送循环的重要组成部分，这就是发送中断服务程序 `xxIntTx()`。其函数原型如下：

```
void xxIntTx (pxxChan);
```

`xxIntTx()`负责从高层协议取出字符并将其传送出去同时清除相应中断，如果此时高层协议不再传送数据，则重置传输中断，完成一个传送周期，等待 `xxTxStartup()`开启另一个传送循环。

下面是 `xxIntTx()`函数的一个例子：

```

void xxIntTx (xx_CHAN * pxxChan)
{
    char outChar;

    if ((*pxxChan->getTxChar) (pxxChan->getTxArg, &outchar) != ERROR)
        *pxxChan->dr = outChar;          /* 发送数据 */
    else
        *pxxChan->cr = xx_RESET_TX;      /*复位中断（如果有必要的话）*/

    /* 应答中断（如果有必要的话） */
    *pxxChan->cr = xx_RESET_INT;
}

```

7.3.4 串行设备接收 ISR

`xxRcvInt()`是串行设备的接收中断服务程序，它从设备接收数据，并通过回调函数 `putRcvChar()`将其传给高层协议。在通常的驱动程序中 `xxIntTx()`和 `xxIntRcv()`是在硬件的中断服务程序中同时实现的。`xxIntRcv`的函数原型如下：

```
void xxIntRcv (pSioChan);
```

下面是 `xxIntRcv()`函数的一个例子：

```

void xxIntRcv (xx_CHAN * pxxChan)
{
    char inChar;

    /* 接收数据 */
    inChar = *pxxChan->dr;
}

```



```
/* 中断应答（如果需要的话）*/
*pxxChan->cr = xx_RESET_INT

/* 向高层协议传送接收数据 */
(*pxxChan->putRcvChar)
(pxxChan->putRcvArg, inchar);
}
```

7.3.5 控制函数——xxioctl()

如前所述，当用户使用 ioctl()函数访问虚拟设备 ttyDrv 时，I/O 系统会调用 ttyIoctl()函数完成请求，而 ttyIoctl()函数将分别访问 xxIoctl()和 tyIoctl()函数以完成一次操作。由于前面已经讨论过这个过程，这里就不再赘述。驱动程序的 Ioctl 控制功能通常要实现表 7-2 中所列举的功能。

表 7-2 标准 ioctl 命令

命 令	描 述
SIO_BAUD_SET	设置波特率
SIO_BAUD_GET	获取当前波特率
SIO_MODE_SET	设置中断/轮询模式
SIO_MODE_GET	获取当前模式
SIO_AVAIL_MODES_GET	获取有效的模式
SIO_HW_OPTS_SET	设置硬件
SIO_HW_OPTS_GET	获取硬件设置

值得说明的是在 ttyIoctl()函数中将 FIOBAUDRATE 替换为 SIO_BAUD_SET，所以用户使用 FIOBAUDRATE 命令时，相当于使用了驱动程序的 xxIoctl()函数中的 SIO_BAUD_SET 命令。

 注意

如果某个命令在 xxioctl()中未处理，则 xxioctl()返回 ENOSYS。

7.3.6 轮询处理函数

为了支持 WDB 的系统级调试模式，串行设备必须支持轮询工作模式。这时就需要实现轮询输入函数 xxPollInput()和轮询输出函数 xxPollOutput()。

1. 轮询输入函数

xxPollInput()函数执行的功能相当简单：查询设备中是否有数据输入，如果有则将数据填入到缓冲区中，如果没有则返回 EAGAIN。该函数原型如下所示：

```
int xxPollInput (pSioChan, pChar)
```

下面是轮询输入函数 xxPollInput()的一个模板：

```
LOCAL int xxPollInput(SIO_CHAN * pSioChan, char * pInputChar)
{
```



```

xx_CHAN *pxxChan=(xx_CHAN *)pSioChan;

/*sr:状态寄存器, dt:数据发送寄存器*/
if ((*pxxChan->sr&xx_RX_AVAIL)==0)
    /* 无有效数据 */
    return(EAGAIN);

/* 存放有效数据 */
*pInputChar=(*pxxChan->inByte) (pxxChan->dr);
return(OK);
}
-----

```

2. 轮询输出函数

xxPollOutput()函数执行的功能也相当简单：确定设备处于可写状态，将字符写入并返回 OK，否则返回 EAGAIN。该函数原型如下所示：

```
int xxPollOutput(pSioChan, outchar);
```

下面是轮询输出函数 xxPollOutput()的模板：

```

LOCAL int xxPollOutput(SIO_CHAN *pSioChan, char outchar)
{
    xx_CHAN * pxxChan=(xx_CHAN *)pSioChan;

    /*sr:状态寄存器, dt:数据发送寄存器*/
    if ((*pxxChan->sr &xx_TX_READY)==0)
        return(EAGAIN); /*不能发送*/

    /* 发送数据*/
    (*pxxChan->outByte) (pxxChan->dt, outchar);
    return(OK);
}
-----

```

7.3.7 支持 WDB 通信

串行设备可以实现三种工作模式：中断模式，轮询模式及可在中断轮询之间切换的双模式。如果仅仅支持中断模式，可以不实现前面所提到的 xxPollInput()和 xxPollOutput()函数。如果仅仅支持轮询模式就可以不实现 xxStartup()及其他相关函数。但是，如果要支持双模式，不但要支持上面的所有接口，还要考虑模式间切换的问题。

当然，支持和不支持轮询模式在实现上还是有一点差别。不支持轮询模式时，必须实现一个空的返回 ERROR 的函数，并使 xxPollInput()和 xxPollOutput()的入口点指向该函数。在实现 ioctl()的 SIO_MODE_GET 和 SIO_MODE_SET 命令时也要注意。执行 SIO_MODE_GET 命令应将返回参数 arg 置为 SIO_MODE_INT；执行 SIO_MODE_SET 命令时，若 arg 为 SIO_MODE_POLL 则返回 EIO。

支持轮询模式时，不但要实现 xxPollInput()和 xxPollOutput()两函数，还要相应地实现 SIO_MODE_SET、SIO_MODE_GET 和 SIO_AVAIL_MODES_GET 功能。

支持轮询模式是为了支持 WDB 的系统级调试，通常目标机代理 (Target Agent) 会做如下操作：

- 关闭中断；

- 切换到轮询模式;
- 使用轮询模式;
- 切换到中断模式;
- 打开中断。



注意

在切回到中断模式时，一定要确定设备处于中断状态，并且中断服务程序会适当的响应中断。

7.4 串行设备的安装

串行设备驱动程序不像字符型设备通过简单的 `DrvInstall()` 及 `DevCreate()` 就可以安装到系统中。由于要支持 WDB 等特性，应在系统初始化时安装。通过修改 `sysSerial.c` 中的 `sysSerialHwInit()`、`sysSerialHwInit2()`、`sysSerialChanGet()` 可以将设备安装到系统中。

7.4.1 串行设备的初始化

函数 `sysSerialHwInit()` 函数执行设备的初始化操作，由 `usrInit()` 函数调用。在 `sysSerialHwInit()` 中初始化了驱动程序相关的 `xx_CHAN` 结构，调用 `xxDevInit()` 函数初始化硬件，设置中断未连接的标志。这个函数是在内核初始化之前完成的。下面是该函数的例子（使用 8250 芯片的串行设备驱动程序）：

```
void sysSerialHwInit (void)
{
    int i;

    for (i=0; i<N_UART_CHANNELS; i++)
    {
        i8250Chan[i].int_vec = devParas[i].vector;
        i8250Chan[i].channelMode = 0;

        /* 初始化各寄存器 */
        i8250Chan[i].lcr = UART_REG(UART_LCR, i);
        i8250Chan[i].data = UART_REG(UART_RDR, i);
        i8250Chan[i].brdl = UART_REG(UART_BRDL, i);
        i8250Chan[i].brdh = UART_REG(UART_BRDH, i);
        i8250Chan[i].ier = UART_REG(UART_IER, i);
        i8250Chan[i].iid = UART_REG(UART_IID, i);
        i8250Chan[i].mdc = UART_REG(UART_MDC, i);
        i8250Chan[i].lst = UART_REG(UART_LST, i);
        i8250Chan[i].msr = UART_REG(UART_MSR, i);

        i8250Chan[i].outByte = sysOutByte;
        i8250Chan[i].inByte = sysInByte;

        /* 硬件初始化 */
        i8250HrdInit(&i8250Chan[i]);
    }
}
```


可以看到，这个函数的流程如下：

```
void sysSerialHwInit (void)
{
    ... /*初始化相关数据结构*/
    ... /*初始化硬件*/
}
```

7.4.2 安装串行设备的 ISR

函数 sysSerialHwInit2() 由 usrRoot 任务中的 sysClkConnect() 函数调用，安装了设备的中断服务程序。这个函数是在系统初始化之后执行，此时系统已经可以执行中断连接了。下面是这个函数的例子（使用 8250 芯片的串行设备驱动程序）：

```
void sysSerialHwInit2 (void)
{
    int i;

    /* 连接串行设备中断 */
    for (i=0; i<N_UART_CHANNELS; i++)
        if (i8250Chan[i].int_vec)
        {
            (void) intConnect (INUM_TO_IVEC (i8250Chan[i].int_vec),
                                i8250Int, (int)&i8250Chan[i] );
            sysIntEnablePIC (devParas[i].intLevel);
        }
}
```

在这个过程中，安装了串行设备相应的中断并将打开中断。

7.4.3 串行设备描述符与通道号的转换

函数 sysSerialChanGet() 的作用是将串行设备通道的索引转换为该通道的设备描述符。通常目标机代理会使用这个函数以获得通道的指针，usrRoot 任务也会用到该函数。下面是这个函数的例子（使用 8250 芯片的串行设备驱动程序）：

```
SIO_CHAN * sysSerialChanGet
(
    int channel      /* 串行通道 */
)
{
    return ( (SIO_CHAN *)&i8250Chan[channel] );
}
```

通过适当的修改，用户就可以将自己的串行设备驱动程序安装到系统中，并用来进行调试和通信。

7.5 串行设备驱动程序完整的模板

下面将总结这一章所给出的模板，为读者提供一个完整的模板。

1. 串行通道数据结构

```
typedef struct      /* xx_CHAN */
```



```

{
    SIO_DRV_FUNCS * pDrvFuncs;          /* 驱动程序所需的函数 */
    STATUS (*getTxChar) ();              /* 传送函数指针 */
    STATUS (*putRcvChar) ();            /* 接受函数指针 */
    void * getTxArg;
    void * putRcvArg;
    UINT16 int_vec;                     /* 中断向量 */
    UINT16 channelMode;                 /* 模式中断或轮询 */
    UCHAR (*inByte) (int);              /* 从硬件读取一个字节函数指针 */
    void (*outByte) (int, char);        /* 向硬件写入一个字节函数指针 */

    ... /*必要的寄存器描述及结构*/
} xx_CHAN;

```

2. 串行设备驱动程序结构初始化函数

```

void xxDevInit
(
    xx_CHAN * pChan          /* 指向设备的指针 */
)
{
    SIO_DRV_FUNCS xxSioDrvFuncs;

    /* 驱动程序结构初始化*/
    if (xxSioDrvFuncs.ioctl == NULL)
    {
        xxSioDrvFuncs.ioctl      = (int (*)( ))xxIoctl;
        xxSioDrvFuncs.txStartup  = (int (*)( ))xxStartup;
        xxSioDrvFuncs.callbackInstall = xxCallbackInstall;
        xxSioDrvFuncs.pollInput  = (int (*)( ))xxPollInput;
        xxSioDrvFuncs.pollOutput = (int (*)(SIO_CHAN *, char))xxPollOutput;
    }

    pChan->pDrvFuncs = &xxSioDrvFuncs;

    ... /*必要的硬件初始化*/

}

```

3. 回调安装函数

```

static int xxCallbackInstall
(
    SIO_CHAN * pSioChan,
    int callbackType,
    STATUS (*callback) (),
    void * callbackArg
)
{
    xx_CHAN * pChan = (xx_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);
    }
}
```

4. 启动一次发送周期函数

```
LOCAL int xxTxStartup(SIO_CHAN * pChan)
{
    xx_CHAN * pxxChan = (xx_CHAN *)pChan;
    char ier = xx_IER_RXRDY;
    char status;

    if (pxxChan->channelMode == SIO_MODE_INT)
    {
        /*根据硬件寄存器的值判断状态*/
        status = ((*pxxChan->inByte) (pxxChan->sta));

        /*如果状态正确，打开中断*/
        if(status==xx_OK)
            (*pxxChan->outByte) (pxxChan->ier, ier); /* ier: 打开中断寄存器 */
    }
    return (OK);
}
```

5. 轮询接收函数

```
LOCAL int xxPollInput(SIO_CHAN * pSioChan, char * pInputChar)
{
    xx_CHAN *pxxChan=(xx_CHAN *)pSioChan;

    /* sr: 状态寄存器, dr: 数据接收寄存器 */
    /* 读状态寄存器, 是否接收到数据*/
    if ((*pxxChan->sr&xx_RX_AVAIL)==0)
        return(EAGAIN);

    /* 获得数据 */
    *pInputChar=(*pxxChan->inByte) (pxxChan->dr);

    return (OK);
}
```

6. 轮询发送函数

```

LOCAL int xxPollOutput(SIO_CHAN *pSioChan, char outchar)
{
    xx_CHAN * pxxChan=(xx_CHAN *)pSioChan;

    /*sr: 状态寄存器, dt: 数据发送寄存器 */
    /* 读状态寄存器, 设备发送是否准备好 */
    if ((*pxxChan->sr &xx_TX_READY)!=0)
        return(EGAIN); /*不能发送*/

    /* 发送数据 */
    (*pxxChan->outByte) (pxxChan->dt, outchar);

    return (OK);
}

```

7. 串行设备初始化、中断注册等函数

```

void sysSerialHwInit (void)
{
    ...    /* 初始化相关数据结构 */
    ...    /* 初始化硬件 */
}

void sysSerialHwInit2 (void)
{
    /* 连接串行设备中断 */
    /*使能中断*/
}

SIO_CHAN * sysSerialChanGet
(
    int channel    /* 串行通道 */
)
{
    return ( (SIO_CHAN *)&xxChan[channel] ); /*返回相应的结构*/
}

```

有关上述这些函数, 详细处理请参考 7.4 节内容。

8. 中断管理函数

```

LOCAL ULONG xxIntHandler(int xxDevId)
{
    xx_CHAN * pxxChan=(xx_CHAN*)&xxChan[xxDevId];

    /* lr: 中断状态寄存器, dr: 数据接收寄存器, dt: 数据发送寄存器*/
    /* 读中断状态寄存器 */
    if ((*pxxChan->lr &xx_TX_INT)!=0)    /* 是否为发送中断*/

```



```
(*pxxChan->outByte) (pxxChan->dt, outchar); /* 发送数据*/

/* 读中断状态寄存器 */
if ((*pxxChan->lr&xx_RX_INT) != 0) /* 是否为接收中断*/
    *pInputChar = (*pxxChan->inByte) (pxxChan->dr); /* 接收数据 */
... /* 清除相应中断 */
}
```

9. 控制函数

```
static int xxIoctl
(
    xx_CHAN * pChan,          /* 指向控制的设备 */
    int cmd,                  /* 请求代码 */
    int arg                    /* 参数 */
)
{
    int ix;
    int oldlevel;
    int status = OK;

    switch (cmd)
    {
        case SIO_BAUD_SET:      /* 设置波特率 */
            status = (xxBaudSet(pChan, arg) == OK) ? OK : EIO;
            break;
        case SIO_BAUD_GET:      /* 获得波特率 */
            oldlevel = intLock();
            status = EIO;
            status = (xxBaudGet(pChan, *arg) == OK) ? OK : EIO;
            intUnlock(oldlevel);
            break;
        case SIO_MODE_SET:      /* 工作模式设置 */
            status = (xxModeSet(pChan, arg) == OK) ? OK : EIO;
            break;
        case SIO_MODE_GET:      /* 获得工作模式 */
            *(int *)arg = pChan->channelMode;
            return (OK);
        case SIO_AVAIL_MODES_GET: /* 获取有效的模式 */
            *(int *)arg = SIO_MODE_INT | SIO_MODE_POLL;
            return (OK);
        case SIO_HW_OPTS_SET:    /* 设置硬件 */
            status = (xxOptsSet(pChan, arg) == OK) ? OK : EIO;
            break;
        case SIO_HW_OPTS_GET:    /* 获得硬件设置 */
            *(int *)arg = pChan->options;
            break;
        case SIO_OPEN:          /* 检查 hupcl 选项是否打开 */
            if (pChan->options & HUPCL)
                status = xxOpen(pChan);
            break;

        default:
```



```
        status = ENOSYS;  
        break;  
    }  
    return (status);  
}
```

小 结

这一章中讨论了 VxWorks 的串行设备驱动程序的开发。串行设备拥有独特的初始化过程，与系统之间的关系也比较特殊。它支持目标机代理与主机之间的通信，支持中断、轮询双工作模式，通过回调函数支持不同的高层协议，适合于各种串行设备驱动程序的实现。

第 8 章 编写块设备驱动程序

块设备是 VxWorks 系统中重要的一种设备。这类设备都有下列的特性：

- 支持文件结构；
- 随机访问能力；
- 使用多字节的数据块传输数据。

块设备的驱动程序不是直接挂接在 VxWorks 的 I/O 系统中。由于在 VxWorks 中，文件系统是作为驱动程序挂接到 I/O 系统中，而用户通过访问文件系统访问块设备，所以块设备是挂接在文件系统上的，VxWorks 的 I/O 系统、文件系统及块设备驱动程序之间的关系如图 8.1 所示。

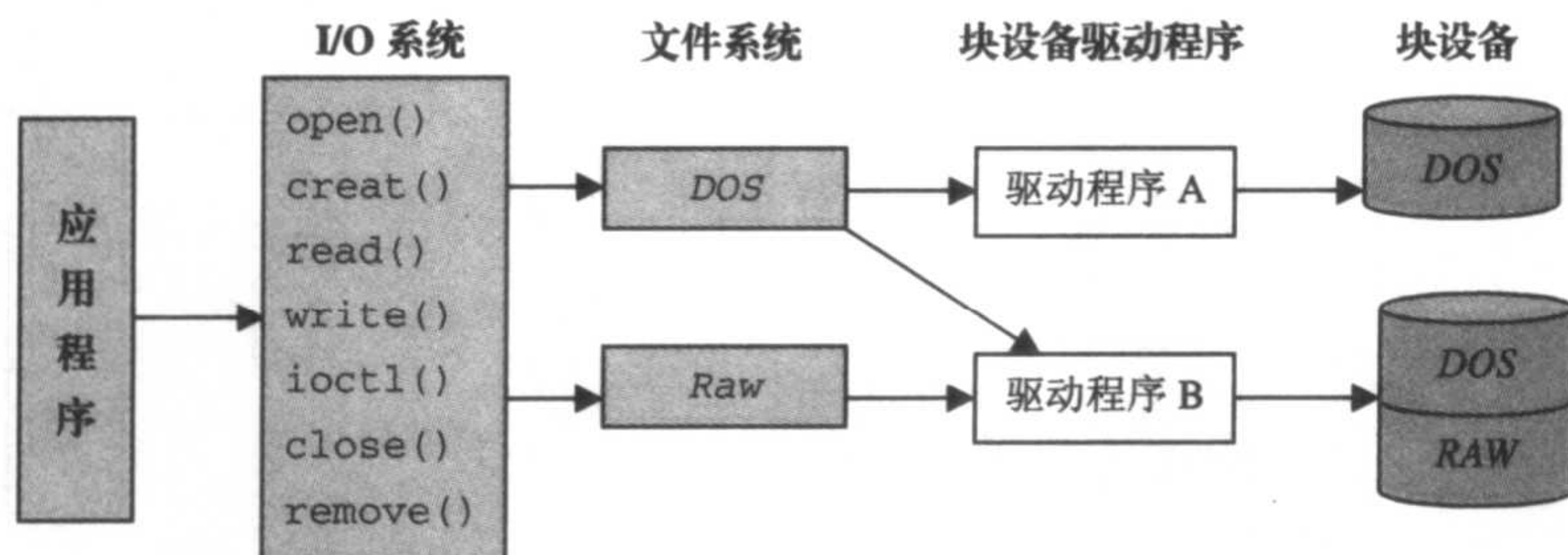


图 8.1 VxWorks 的 I/O 系统、文件系统及块设备驱动程序之间的关系

在这一章中，将首先回顾一下 VxWorks 的文件系统 dosFs 和 rawFs 文件系统的配置及其对块设备的访问，然后一步步讨论块设备驱动程序的编写。

8.1 VxWorks 下的文件系统

VxWorks 支持多种文件系统，与 DOS 文件系统兼容的 dosFs、原始文件系统 rawFs、磁带文件系统 tapeFs、CDROM 文件系统 cdromFs、网络文件系统 NFS 及目标机服务器文件系统 TSFS 等。

8.1.1 VxWorks 支持的文件系统

1. 与 MS-DOS 文件系统兼容的 dosFs

dosFs 有诸多优点，它兼容于 MS-DOS5.0 文件系统，可以有效地利用磁盘空间，同时又保证文件访问的效率。与 MS-DOS 文件系统相同，dosFs 支持目录分级结构，在目录结构中可以包含

更深的目录结构。出于访问效率的考虑，Wind River 还对其进行了优化，dosFs 允许用户指定文件是否连续，从而加快文件访问速度。有关于 dosFs 的详细信息，请参阅本书参考书目中的相关内容。

2. 原始文件系统 rawFs

原始文件系统将块设备当作一个单独的文件，不支持目录结构和多文档结构，但原始文件系统使访问效率得到最大的提高。原始文件系统为用户提供了有效的磁盘管理办法，但是如果对访问没有什么特殊要求首选还是 dosFs。关于 rawFs 的详细信息，请参阅本书参考书目中的相关内容。

3. 其他文件系统

VxWorks 同时还支持其他的文件系统：专门面向磁带系统的 tapeFs、面向 CDROM 的文件系统 cdromFs、网络文件系统 NFS、目标机服务器文件系统 TSFS 等，每一种文件系统都可以和其他文件系统并存，而每一种块设备驱动程序既可以支持其中的一种也可以支持其中的多种格式。详细内容请参阅本书参考书目中的文件系统部分。

8.1.2 文件系统的配置

文件系统在使用前应进行适当的初始化和配置，这里以 dosFs 及 rawFs 为例讨论一下初始化及配置的过程。

不论对于何种文件系统，在使用前都要初始化文件系统、初始化块设备驱动程序并将其挂接到文件系统中。用户可以使用文件系统的指定函数初始化文件系统，使用 xxDrv() 安装块设备驱动程序，使用 xxDevCreate() 创建并初始化设备。步骤如下。

(1) 调用 xxDrv() 函数安装块设备驱动程序。

(2) 执行 pBlkDev = xxDevCreate() 操作，创建设备。在这里 pBlkDev 是由 xxDevCreate() 返回的指向 BLK_DEV 的指针，BLK_DEV 结构将会在后面讨论到。

(3) 接下来就可以用 pBlkDev 作为参数初始化文件系统了。对于原始文件系统来说，初始化的过程比较简单，示例如下：

```
rawFsDevInit(fsName, pBlkDev);
```

而 dosFs 的初始化就比较复杂，初始化新的文件系统时有两种方法：使用 dosFs 库中的初始化函数或者使用 I/O 系统的接口。

使用 dosFs 库中的函数的例子如下：

```
dosFsMkfsOptionSet(option);          /*可选的*/
dosFsMkfs(fsName, pBlkDev);
```

在这个过程中 dosFsMkfsOptionSet 是可选的。dosFsMkfs() 函数会依照默认的参数初始化文件系统；如果使用了 dosFsMkfsOption() 则可以按照用户的设置初始化文件系统。

用户也可以通过 I/O 系统初始化 dosFs 文件系统，其过程如下：

```
dosFsConfigInit(pConfig, ...);
dosFsDevInit(fsName, pBlkDev, pConfig);
```


随后用户就可以用通常的办法访问文件了。

```
fd = open( fsName, O_RDWR, 0);
ioctl( fd, DISKINIT, 0);
```

8.1.3 文件系统对块设备的访问

对于 I/O 系统，块设备的驱动程序是不可见的。用户对文件的 I/O 请求由 I/O 系统传递给文件系统，文件系统根据请求作出相应的处理，或者设置相关参数，或者访问块设备驱动程序。文件系统对 I/O 请求的处理通常分为两类，一类是由文件系统自身来处理的，另一类需要通过块设备驱动程序来处理。其中 open()、close()等操作属于第一类，而 read()、write()、creat()、remove()等操作则是属于第二类，ioctl()操作比较特殊，它执行的命令中既有属于第一类操作的命令，也有属于第二类操作的命令。

open()操作是第一类操作中的典型例子，它的操作流程如图 8.2 所示。当用户执行 open()操作之后，I/O 系统会通过设备列表找到相应的设备（文件系统），并在驱动程序表中找到相应驱动程序函数的入口点（文件系统的驱动程序入口点 dosFsOpen 函数）。这个驱动程序函数将会执行文件的打开操作。

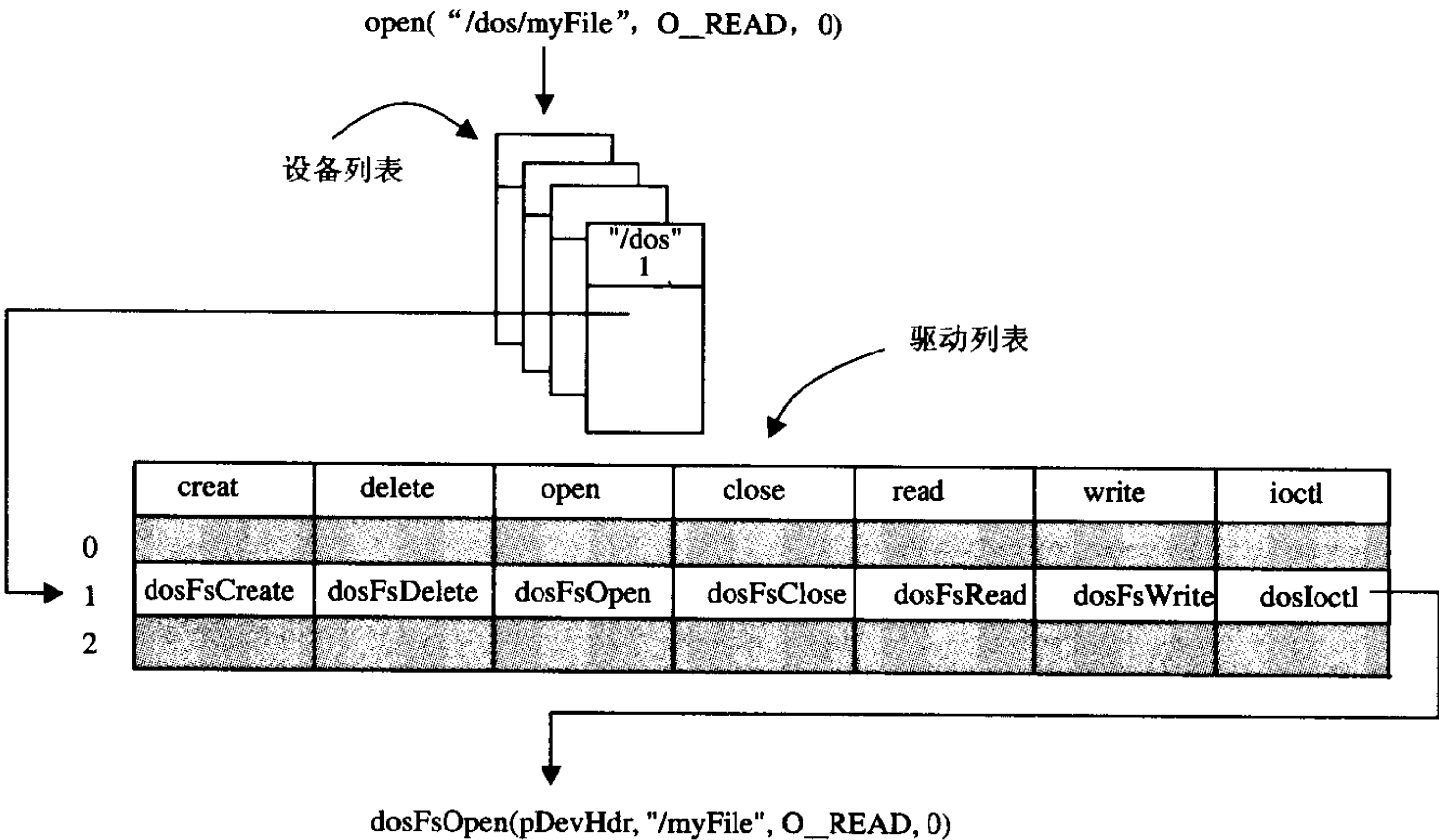


图 8.2 块设备 open()操作

第二类操作与硬件设备有关。在图 8.3 所示的块设备 read()操作中示例了用户执行 read()操作的过程。I/O 系统通过文件描述符表找到驱动程序表中的文件系统的驱动程序函数入口点并执行它时，文件系统的驱动程序函数将会调用真正的块设备的驱动程序函数，完成一次操作。

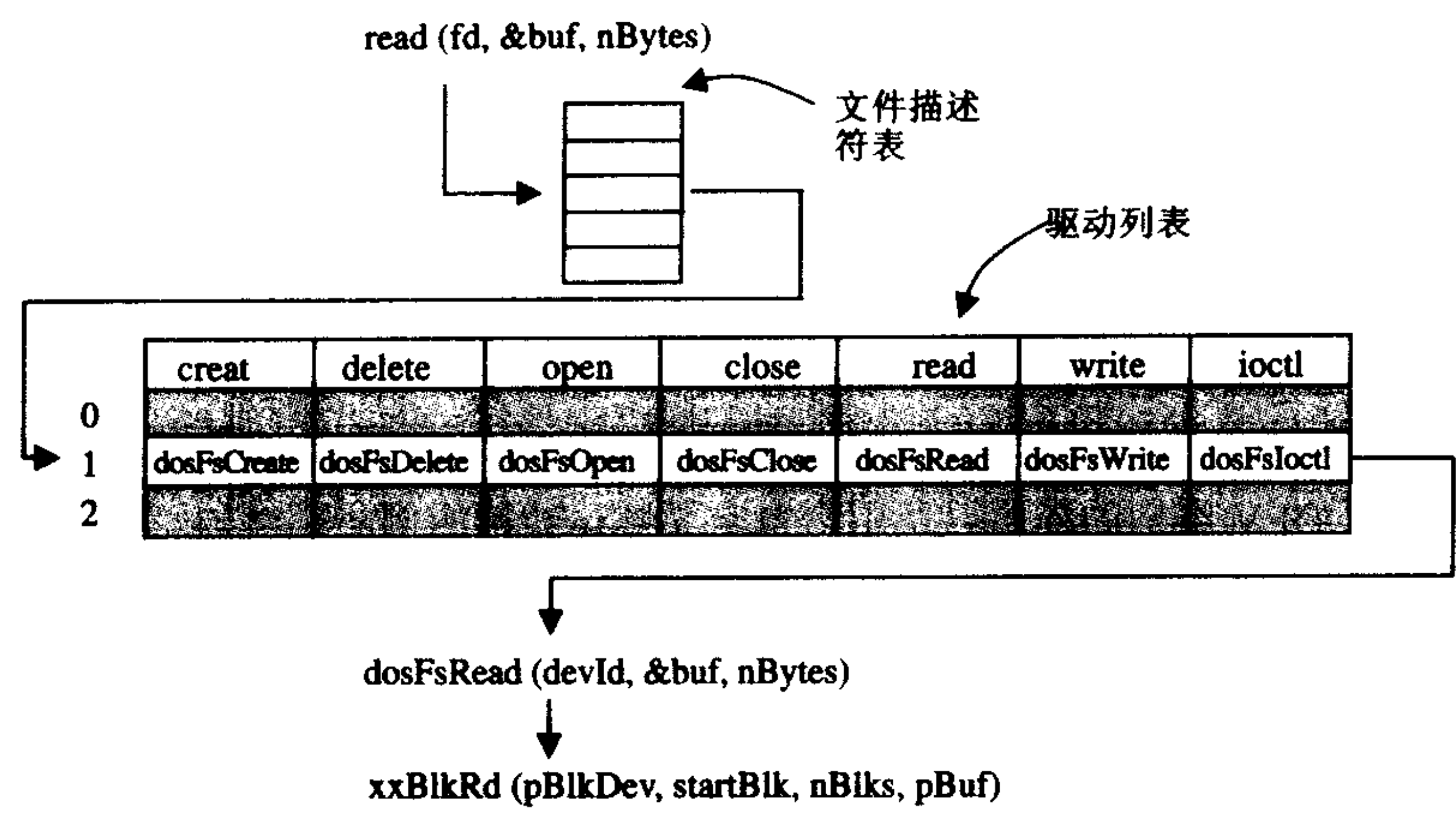


图 8.3 块设备 read()操作

8.2 一步步编写块设备驱动程序

在这一节中，将会讨论到块设备的特点，块设备中特定的数据结构以及块设备驱动程序的编写方法。

8.2.1 块设备驱动程序的特点

块设备与前面讨论的字符设备不同。VxWorks 的 I/O 系统是通过文件系统访问块设备的，所以块设备的驱动程序函数的入口点不会安装到驱动程序表中，块设备的描述符也不会链接到系统的设备列表中。块设备驱动程序是由文件系统来访问的，所以支持另外的一套驱动程序表。块设备是以块为单位进行数据传输的，而字符设备则是以字节为单位进行数据传输，这使块设备驱动程序的功能也大为不同：同一个设备驱动程序应该同时支持不同的文件系统。尽管块设备驱动程序可以支持多种文件系统，但对于块设备本身来说，是不用关心文件系统的存在的。

8.2.2 块设备中重要的数据结构

为了与文件系统交换信息，块设备的驱动程序同样需要声明一个特定的数据结构，这就是 BLK_DEV 结构。BLK_DEV 包含如下成员。

- bd_blkRd: 驱动程序读函数的入口地址。
- bd_blkWrt: 驱动程序写函数的入口地址。
- bd_ioctl: 驱动程序控制功能函数的入口地址。
- bd_reset: 驱动程序重置函数的入口地址。
- bd_statusChk: 驱动程序检查磁盘状态的函数的入口地址，这个成员是可选的。
- bd_removeable: 这是一个布尔型变量，取 TRUE 时表明当前设备是可移动的媒介，而取 FALSE 则表示设备不可移动。
- bd_nBlocks: 指明分区（磁盘）块数的成员变量。

- **bd_bytesPerBlk**: 指明分块中包含的字节数。
- **bd_blksPerTrack**: 指明每一个磁道包含的块的数目。
- **bd_nHeads**: 指明磁头数。
- **bd_retry**: 指明遇到 I/O 错误后重试的次数。
- **bd_mode**: 指明设备模式为 O_RDONLY 或者 O_RDWR。
- **bd_readyChanged**: 这个值总是为 TRUE。

这个结构包含在 blkIo.h 中, 其中 bd_blkRd、bd_blkWrt 和 bd_ioctl 是一个设备所必需的。下面是一个块设备 BLK_DEV 结构的模块以及用户自定义数据结构的例子:

```
typedef struct          /* BLK_DEV */
{
    FUNCPTR      bd_blkRd;          /* 驱动程序读函数的入口地址 */
    FUNCPTR      bd_blkWrt;         /* 驱动程序写函数的入口地址 */
    FUNCPTR      bd_ioctl;          /* 驱动程序其他功能函数的入口地址 */
    FUNCPTR      bd_reset;          /* 驱动程序重置函数的入口地址 */
    FUNCPTR      bd_statusChk;      /* 驱动程序检查磁盘状态的函数的入口地址 */
    BOOL         bd_removable;       /* 可移动媒介标志 */
    ULONG        bd_nBlocks;         /* 分区(磁盘)块数 */
    ULONG        bd_bytesPerBlk;     /* 每块中包含的字节数 */
    ULONG        bd_blksPerTrack;    /* 每一个磁道包含的块的数目 */
    ULONG        bd_nHeads;          /* 磁头数 */
    int           bd_retry;           /* 遇到 I/O 错误后重试的次数 */
    int           bd_mode;            /* O_RDONLY | O_WRONLY | O_RDWR */
    BOOL         bd_readyChanged;     /* 设备变更准备好 */
} BLK_DEV;

/* 块设备描述符 */
typedef struct ideDev
{
    BLK_DEV blkDev;          /* 必须在成员第一位 */
    int drive;               /* 设备号 0 - 1 */
    int blkOffset;           /* 偏移量 */
} xxx_DEV;
```

在用户自定义数据结构中使用了 blkOffset 参数, 这个参数可以指定逻辑块设备的在物理块设备上的起始位置。使用这个参数可以支持磁盘系统的多分区模式。



注意

BLK_DEV 结构必须是块设备描述符第一个成员。

8.2.3 驱动程序安装函数

和其他类型的设备驱动程序一样, 块设备也需要一个初始化程序, 这个初始化程序仅执行一些必要的初始化操作(例如初始化驱动程序数据结构和分配内存等), 但不会将驱动程序安装在 I/O 系统中或文件系统中。根据不同的设备, 初始化将执行不同的操作。该函数原型如下:

```
STATUS xxDrv (args...);
```


下面是 xxDrv()函数的流程:

```

STATUS xxDrv(
int  ctrl,          /* 控制器号 */
int  drives,        /* 驱动程序索引号 */
int  vector,        /* 中断向量 */
int  level,         /* 中断级 */
BOOL configType,    /* 配置类型 */
...
)
{
    int drive;
    ... /* 输入参数检查 */

    if (!xxxDrvInstalled)
    {
        ... /* 初始化设备必要的数据结构 */
    }
    /*安装中断*/
    (void) intConnect (...);

    /*打开中断*/
    sysIntEnablePIC (level);

    /*设备初始化*/
    if (xxInit (ctrl) != OK)
    {
        semGive (&pCtrl->muteSem);
        return (ERROR);
    }

    /*对多分区处理*/
    for (drive = 0; drive < xxx_MAX_DRIVES; drive++)
    {
        ... /*面向多分区的处理, 例如:扇区、磁道等*/
    }

    ...
    xxxDrvInstalled = TRUE;
}
    return (OK);
}

```

8.2.4 设备创建函数

设备创建函数 xxDevCreate()的主要作用是创建逻辑设备。逻辑设备既可以是整个物理设备也可以是物理设备的一部分。在创建逻辑设备的过程中, 应该初始化相应的数据结构, 并且所返回的数据结构必须以 BLK_DEV 结构作为第一个成员。文件系统使用这个返回值将文件系统本身安装到设备上。该函数所使用的参数根据设备的不同而不同。该函数原型如下:

```
BLK_DEV * xxDevCreate (args...)
```

下面是 xxDevCreate()函数的流程:

```

BLK_DEV *xxDevCreate
(
    int ctrl,      /* 控制器号 */

```



```

int drive,          /* 设备号 */
int nBlocks, /* 设备所包含的块数*/
int blkOffset /* 开始块相对于设备起始的偏移量*/
)
{
    xx_CTRL *pCtrl    = &xxCtrl[ctrl];      /* 获得控制器结构 */
    xx_TYPE *pType    = &xxTypes[ctrl][drive]; /* 获得配置类型 */
    xx_DRIVE *pDrive  = &pCtrl->drive[drive]; /* 获得驱动程序结构 */
    xx_DEV *pDev;      /* 设备描述符 */
    BLK_DEV *pBlkdev;  /* 块设备结构 */
    int maxBlks;       /* 最大块的个数*/

    /* 输入参数检查及驱动程序安装判断 */
    ...

    /* 分配空间 */
    if ((pDev = (xx_DEV *)malloc(sizeof (xx_DEV))) == NULL)
        return (NULL);

    /* 获得块设备结构 */
    pBlkdev = &pDev->blkDev;

    pType = &xxTypes[drive];
    ...
    maxBlks = (xxLbaTotalSecs[ctrl][drive]) - blkOffset;
    if (nBlocks == 0)
        nBlocks = maxBlks;

    if (nBlocks > maxBlks)
        nBlocks = maxBlks;

    /* 数据结构初始化*/
    ...
    pBlkDev->bd_blkRd    = xxBlkRd;      /* 设备读函数 */
    pBlkDev->bd_blkWrt    = xxBlkWrt;    /* 设备写函数 */
    pBlkDev->bd_ioctl    = xxIoctl;      /* 设备控制函数 */
    pBlkDev->bd_reset    = xxReset;      /* 设备重新设置函数 */
    pBlkDev->bd_statusChk = NULL;
    ...
    return (&pxxxDev->blkDev);
}

```

8.2.5 块设备的读/写函数

块设备的读写函数分别实现了块设备的读、写操作，对于块设备来说，这两个函数都是必须实现的。即便是只读或者只写设备的驱动程序，这两个函数也必须实现。对于只读设备，xxBlkWrt()将执行空操作并返回 ERROR；对于只写操作，xxBlkRd()函数也同样执行空操作并返回 ERROR。这两个函数原型如下：

```

STATUS xxBlkRd( pDev, startBlk, nBlks, pBuf); /* 块设备驱动程序读函数*/
STATUS xxBlkWrt(pDev, startBlk, nBlks, pBuf); /* 块设备驱动程序写函数*/

```

其中，pDev 是指向块设备数据结构的指针，startBlk 是读或写操作的起始块，nBlks 指明读或写的块数，pBuf 是读写所使用的缓冲区。根据不同的设备，驱动程序的读写方式是不同的，但是读写

所操作的数据都是以块为单位的。下面是读函数的流程（由于写函数使用类似的结构，这里就不再列出）。

```

LOCAL STATUS xxBlkRd (
    xx_DEV *pDev,
    int startBlk,
    int nBlks
    char *pBuf
)
{
    return (xxBlkRW (pDev, startBlk, nBlks, pBuf, O_RDONLY));
}

-----
LOCAL STATUS xxBlkRW
(
    xx_DEV *pDev,
    int startBlk,
    int nBlks,
    char *pBuf,
    int direction
)
{
    xx_CTRL *pCtrl    = &xxCtrl[pDev->ctrl];
    xx_DRIVE *pDrive  = &pCtrl->drive[pDev->drive];
    BLK_DEV *pBlkdev  = &pDev->blkDev;
    xx_TYPE *pType    = &xxTypes[pDev->ctrl][pDev->drive];
    int status        = ERROR;
    int retryRW0 = 0;
    int retryRW1 = 0;
    int retrySeek    = 0;
    int cylinder;
    int head;
    int sector;
    int nSecs;
    int ix;
    int endBlk=startBlk+nBlks+pDev->blkOffset;

    while(startBlk<= endBlk)
    {
        /* 完整检查 */
        ...

        nSecs = pBlkdev->bd_nBlocks;
        if ((startBlk + nBlks) > nSecs)
        {
#ifdef    xx_DEBUG
            printErr ("startBlk=%d nBlks=%d: 0 - %d\n", startBlk, nBlks, nSecs);
#endif    /* xx_DEBUG */
            return (ERROR);
        }

        startBlk += pDev->blkOffset;

        /* 获得读写权限 */
        semTake (&pCtrl->muteSem, WAIT_FOREVER);
    }
}

```



```

...
    /* 获得最小读写扇区数 */
    nSecs = min (nBlks - ix, xx_MAX_RW_SECTORS);

    /* 设备相关的读写操作 */
    while (xxRW(pDev->ctrl, pDev->drive, cylinder, head, sector,
        pBuf, nSecs, direction) != OK)
    {
        ...
    }

    startBlk += nSecs;
    pBuf += pBlkdev->bd_bytesPerBlk * nSecs;
}

status = OK;

done:
    if (status == ERROR)
        (void)errnoSet (S_ioLib_DEVICE_ERROR);
    semGive (&pCtrl->muteSem);
    return (status);
}

```

8.2.6 块设备控制函数

块设备控制函数 `xxIoctl()` 的原型如下：

```
int xxIoctl( pBlkDev, cmd, arg);
```

块设备的 `xxIoctl()` 是由文件系统访问的。用户访问 `ioctl()` 接口函数时，I/O 系统会根据文件描述符表找到文件系统的 `ioctl()` 函数（如 `dosFsIoctl()`），而后文件系统的 `ioctl()` 函数再根据需要访问驱动程序程序的 `xxIoctl()` 函数。

在实现上，块设备驱动程序的 `xxIoctl()` 与字符型设备的 `xxIoctl()` 没有本质上的区别，结构上也非常相似，但是实现块设备驱动程序的 `xxIoctl()` 还需要注意几个细节问题：用户所使用的 `cmd` 参数应取那些大于 0 小于 1000 的值，避免与文件系统所响应的命令冲突（文件系统所响应的命令可以由帮助文件获得，表 8-1 列出了 `dosFs` 所支持的控制命令）；在遇到未知的命令时，应将错误号置为 `S_ioLib_UNKNOWN_REQUEST` 并返回 `ERROR`。

表 8-1 dosFs 所支持的 ioctl 命令

命令值	说明
FIODISKFORMAT	驱动程序提供的功能，格式化磁盘
FIODISKINIT	在设备上初始化 dosFs 文件系统
FIODISKCHANGE	声明媒介更换
FIOUNMOUNT	卸载一个磁盘卷
FIOGETNAME	获取指定文件描述符对应的文件名
FIORENAME	为文件或目录重命名

续表

命 令 值	说 明
FIOSEEK	设置文件中的偏移字节
FIOWHERE	返回文件当前偏移指针位置
FIOFLUSH	丢弃文件输出缓冲区中的数据
FIOSYNC	执行 FIOFLUSH 的功能并重新从文件中读取数据
FIOTRUNC	将指定的文件截取到指定的长度
FIONREAD	获取文件未读取的字节数
FIONFREE	获取磁盘中的剩余空间大小
FIONMKDIR	创建新目录
FIONRMDIR	删除指定目录
FIOLABELGET	获取磁盘卷标
FIOLABELSET	设置磁盘卷标
FIOATTRIBSET	设置文件属性
FIOCONTIG	为文件或目录分配连续的空间
FIONCONTIG	获取可分配的最大连续空间
FIOREADDIR	获取下一个目录入口点
FIOFSTATGET	获取文件状态信息

xxIoctl()也是块设备驱动程序中必不可少的一个接口函数，如果这个接口函数什么都不实现，也应该返回 ERROR，并将任务错误号置为 S_ioLib_UNKNOWN_REQUEST。下面是 xxIoctl()的实现流程：

```
LOCAL STATUS xxIoctl(XXX_DEV *pXXXDev, int cmd, int arg)
{
    FAST int status = ERROR;
    int retryCount;
    XXX_TYPE *pType = &xxxTypes[pXXXDev->drive];

    switch (cmd)
    {
        case FIODISKFORMAT:
            .../*格式化磁盘*/

            if (++retryCount > xxxRetry)/*判断累计出错次数*/
            {
                (void) errnoSet (S_ioLib_DEVICE_ERROR);
                return(ERROR);
            }
            status = OK;
            break;
    }
}
```



```

        default:
            (void) errnoSet (S_ioLib_UNKNOWN_REQUEST);
    }
    return (status);
}
-----

```

8.2.7 块设备复位函数

块设备复位函数的用途是重置硬件状态，其原型如下：

```
STATUS xxReset(pBlkDev );
```

这个接口函数不是必须实现的驱动程序部分。但是如果驱动程序提供了这个接口，文件系统将会在两种情况下调用这个接口。当文件系统第一次加载到一个块设备上时文件系统会调用该函数做一些设备的初始化操作；当文件系统读写设备失败之后并且在重置之前，文件系统会试图调用该函数以复位设备。由于面向的设备不同，xxReset()函数所执行的操作也是不同的，有时需要对系统复位，有时什么都不需要做，下面是块设备复位函数的一个流程。

```

LOCAL STATUS xxReset
(
    xx_DEV *pDev
)
{
    xx_CTRL *pCtrl    = &xxCtrl[pDev->ctrl];

    /* 块设备安装了吗? */
    if (!pCtrl->installed)
        return (ERROR);

    semTake (&pCtrl->muteSem, WAIT_FOREVER);

    /* 复位块设备 */
    (void) xxInit (pDev->ctrl);

    semGive (&pCtrl->muteSem);

    return (OK);
}
-----

```

8.2.8 状态检查函数

状态检查函数也是可选的接口函数，其原型如下：

```
STATUS xxStatusChk( pBlkDev);
```

该接口函数是由文件系统在每一次执行文件 open()或者 creat()时调用的。调用这个函数的目的是处理那些可移动存储设备的状态变化或者设备的写保护状态。如果所检查的媒介状态已经变更，应该将 BLK_DEV 的成员变量 bd_readyChanged 置为 TRUE；如果所检查的媒介处于写保护状态，应将 BLK_DEV 的成员变量 bd_mode 置为 O_RDONLY。

在返回 ERROR 状态时，调用该函数的 open()或 creat()函数将会产生创建错误并返回 xxStatusChk

所设置的错误号。下面是该函数的流程：

```

STATUS xxStatusChk (xx_DEV *pxxxDev)
{
    BOOL MediaChanged;

    /*检查硬件是否更换*/
    MediaChanged=IfChanged(xx_DEV *pxxxDev);
    if(MediaChanged)
    {
        /*进行相应的处理*/
        pxxxDev->blkDev->bd_readyChanged=TRUE;
        return(OK);
    }
    else
    {
        pxxxDev->blkDev->bd_readyChanged=FALSE;
        return(ERROR);
    }
}

```

虽然通过这个函数可以检查可移动媒介的变化状态，但是可移动设备的驱动程序应该在媒介发生变化的时候立即更改 BLK_DEV 的成员参数 bd_readyChanged，以便文件系统可以重新加载到新的媒介上，避免出现不应有的错误。这一工作可以在中断中完成。

8.3 完整的块设备驱动程序流程

下面是一个完整的块设备驱动程序流程，供读者参考。

```

/* 变量声明 */
BOOL xxxDrvInstalled=FALSE;
int xxxDrives=2;

```

1. 块设备描述符

```

typedef struct ideDev
{
    BLK_DEV blkDev;
    int drive;          /* 设备索引 0 - 1 */
    int blkOffset;      /* 扇区偏移量 */
} xxx_DEV;

```

2. 块设备驱动程序安装函数

```

STATUS xxDrv()
{
    int drive;
    ...

    /* 输入参数检查 */
    if ((ctrl >= xx_MAX_CTRLIS) || (drives >xx_MAX_DRIVES))
        return (ERROR);
}

```



```

    if (!xxxDrvInstalled)
    {
        ... /* 初始化设备必要的数据结构 */
        /*安装中断*/
        (void) intConnect ((VOIDFUNCPTR *)INUM_TO_IVEC (vector),
                          (VOIDFUNCPTR)xxIntr, ctrl);
        /*使能中断*/
        sysIntEnablePIC (level);

        /*设备初始化*/
        if (xxInit (ctrl) != OK)
        {
            semGive (&pCtrl->muteSem);
            return (ERROR);
        }

        /*对多分区处理*/
        for (drive = 0; drive < xxx_MAX_DRIVES; drive++)
        {
            ... /*面向多分区的处理, 例如:扇区、磁道等*/
        }

        ...
        /* 调用系统初始化程序允许安装 PIO 模式 */
        SYS_xx_INIT_RTN (ctrl);

        xxxDrvInstalled = TRUE;
    }
    return (OK);
}

```

3. 块设备读写操作函数

```

LOCAL STATUS xxBlkRW
(
    xx_DEV *pDev,
    int startBlk,
    int nBlks,
    char *pBuf,
    int direction
)
{
    xx_CTRL *pCtrl = &xxCtrl[pDev->ctrl];
    xx_DRIVE *pDrive = &pCtrl->drive[pDev->drive];
    BLK_DEV *pBlkdev = &pDev->blkDev;
    xx_TYPE *pType = &xxTypes[pDev->ctrl][pDev->drive];
    int status = ERROR;
    int retryRW0 = 0;
    int retryRW1 = 0;
    int retrySeek = 0;
    int cylinder;
    int head;
}

```



```

int sector;
int nSecs;
int ix;

/* 完整检查 */
if (!pCtrl->installed)
    return (ERROR);

nSecs = pBlkdev->bd_nBlocks;
if ((startBlk + nBlks) > nSecs)
{
#ifdef    xx_DEBUG
    printErr ("startBlk=%d nBlks=%d: 0 - %d\n", startBlk, nBlks, nSecs);
#endif    /* xx_DEBUG */
    return (ERROR);
}

startBlk += pDev->blkOffset;

/* 获得读写权限 */
semTake (&pCtrl->muteSem, WAIT_FOREVER);
...
/* 获得最小读写扇区数*/
nSecs    = min (nBlks - ix, xx_MAX_RW_SECTORS);

/* 读写操作 */
while (xxRW(pDev->ctrl, pDev->drive, cylinder, head, sector,
    pBuf, nSecs, direction) != OK)
{
    if (++retryRW0 > xxRetry)
    {
        (void)xxCmd (pDev->ctrl, pDev->drive, xx_CMD_RECALIB, NULL,
            NULL);
        if (++retryRW1 > xxRetry)
            goto done;
        retrySeek = 0;
        while (xxCmd (pDev->ctrl, pDev->drive, xx_CMD_SEEK, cylinder,
            head) != OK)
            if (++retrySeek > xxRetry)
                goto done;
        retryRW0 = 0;
    }
}

startBlk += nSecs;
pBuf += pBlkdev->bd_bytesPerBlk * nSecs;
}

status = OK;

```



```

done:
    if (status == ERROR)
        (void)errnoSet (S_ioLib_DEVICE_ERROR);

    semGive (&pCtrl->muteSem);
    return (status);
}

```

4. 块设备控制函数

```

LOCAL STATUS xxIoctl (xxx_DEV *pxxxDev, int cmd, int arg)
{
    FAST int status = ERROR;
    int retryCount;
    xxx_TYPE *pType = &xxxTypes[pxxxDev->drive];

    switch (cmd)
    {
        case FIODISKFORMAT: /*格式化磁盘*/
            if (++retryCount > xxxRetry) /*判断累计出错次数*/
            {
                (void) errnoSet (S_ioLib_DEVICE_ERROR);
                return(ERROR);
            }
            status = OK;
            break;
        default:
            (void) errnoSet (S_ioLib_UNKNOWN_REQUEST);
    }

    return (status);
}

```

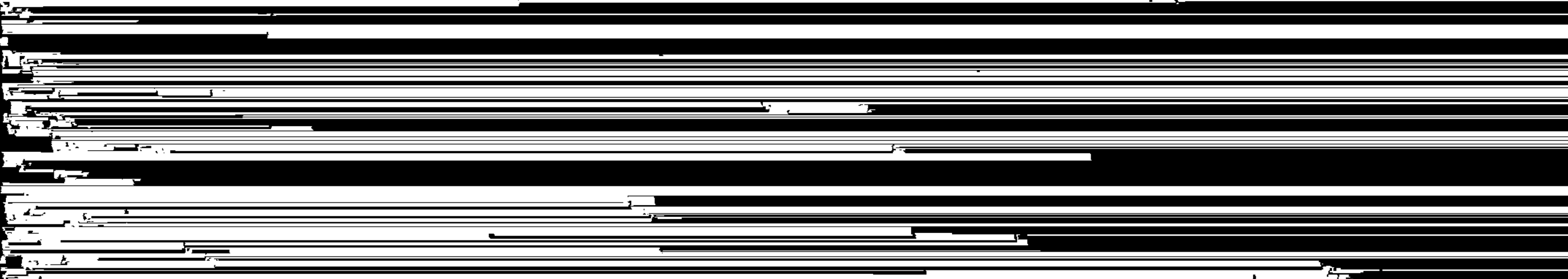
5. 创建块设备

```

BLK_DEV *xxDevCreate
(
    int ctrl, /* 控制器号 */
    int drive, /* 设备号 */
    int nBlocks, /* 设备所包含的块数 */
    int blkOffset /* 开始块相对于设备起始的偏移量 */
)
{
    xx_CTRL *pCtrl = &xxCtrl[ctrl]; /* 获得控制器结构 */
    xx_TYPE *pType = &xxTypes[ctrl][drive]; /* 获得配置类型 */
    xx_DRIVE *pDrive = &pCtrl->drive[drive]; /* 获得驱动程序结构 */
    xx_DEV *pDev; /* 设备描述符 */
    BLK_DEV *pBlkdev; /* 块设备结构 */
    int maxBlks; /* 最大块的个数 */

    /* 输入参数检查及驱动程序安装判断 */
    if ((ctrl >= xx_MAX_CTRL) || (drive >= xx_MAX_DRIVES) ||
        !xxDrvInstalled || !pCtrl->installed)
        return (NULL);
}

```


```
(  
  xxx_DEV *pxxxDev,  
  int startBlk,  
  int nBlks,  
  char *pBuf  
)  
{  
  return (xxxBlkRW (pxxxDev, startBlk, nBlks, pBuf, O_WRONLY));  
}
```

小 结

在这一章里我们讨论了块设备的驱动程序与 VxWorks 文件系统之间的关系、块设备驱动程序的编写并给出了块设备驱动程序的简单流程。但是完成块设备驱动程序还会涉及到一些更实际、更深层次的问题，由于本章面向的不是实际的块设备，这些问题将不会讨论到。按照 VxWorks 中规定的接口完成驱动程序将会使块设备程序设计的大部分问题变得简单，可以参考相关的书籍或者硬件的数据手册解决这些问题。

第 9 章 编写网络设备驱动程序

网络设备是一组特殊的字符设备。网络设备并不直接与用户打交道，而是通过多层协议与应用程序交换信息。应用程序所使用的是协议的接口，通常是套接字（Socket）。网络设备驱动程序是挂在协议最底层直接与硬件打交道的程序。这一章将集中讨论在 VxWorks 下网络设备驱动程序的编写。

9.1 概述

1. 数据交换

图 9.1 所示的 VxWorks 网络模块指出了 VxWorks 应用与网络设备驱动程序之间的各层协议。

网络系统各层之间不断进行数据的交换，要做到有条不紊地交换数据，各层之间都建立了协议，明确规定了所交换的数据的格式等。在数据发送过程中，网络传输协议层通过应用程序接口（例如套接字）获得来自应用程序层的数据，并将其传送给网络协议层（例如 IP 协议等），最后由网络接口层利用物理层（硬件）将数据传出去。而数据接收过程是发送过程的反过程，数据的流向与发送相反。这里的网络接口层实际上就是指网络设备驱动程序。

2. 网络接口驱动程序

网络数据链路层通过网络接口驱动程序，负责在两个相邻节点间的线路上无差错地进行数据的发送和接收。而传输的数据是以帧为单位，每一帧包括一定数量的数据和一些必要的控制信息。在传送数据时，如果接收方检测到所传数据中有差错，则通知发送方重发这一帧，直到这一帧正确无误地到达接收方为止。在每一帧所包括的控制信息中，有同步信息、地址信息、差错控制以及流量控制信息等。

VxWorks 中支持了两种类型的网络设备驱动程序——VxWorks BSD 4.3 网络驱动程序和 VxWorks 可裁减的增强型网络堆栈（SENS, Scalable Enhanced Networks Stack）。VxWorks BSD 4.3 协议栈符合 BSD 4.3 标准，提供了网络设备驱动程序与 IP 协议的紧密结合；VxWorks SENS 协议栈提供了可替换的网络设备驱动程序，由此 Wind River 把专为使用这个网络堆栈而编写的网络接口设备驱动程序称为增强的网络驱动程序（END, Enhanced Network Driver）。

两种不同模式的网络设备驱动程序模型的结构也不相同。这里主要讨论一下 SENS 模型的特点。VxWorks 的网络系统如图 9.2 所示，SENS 模型包含三个部分：协议驱动程序，多元接口（MUX，



图 9.1 VxWorks 网络模块

multiplexer) 层, 增强的网络驱动程序 (END driver)。

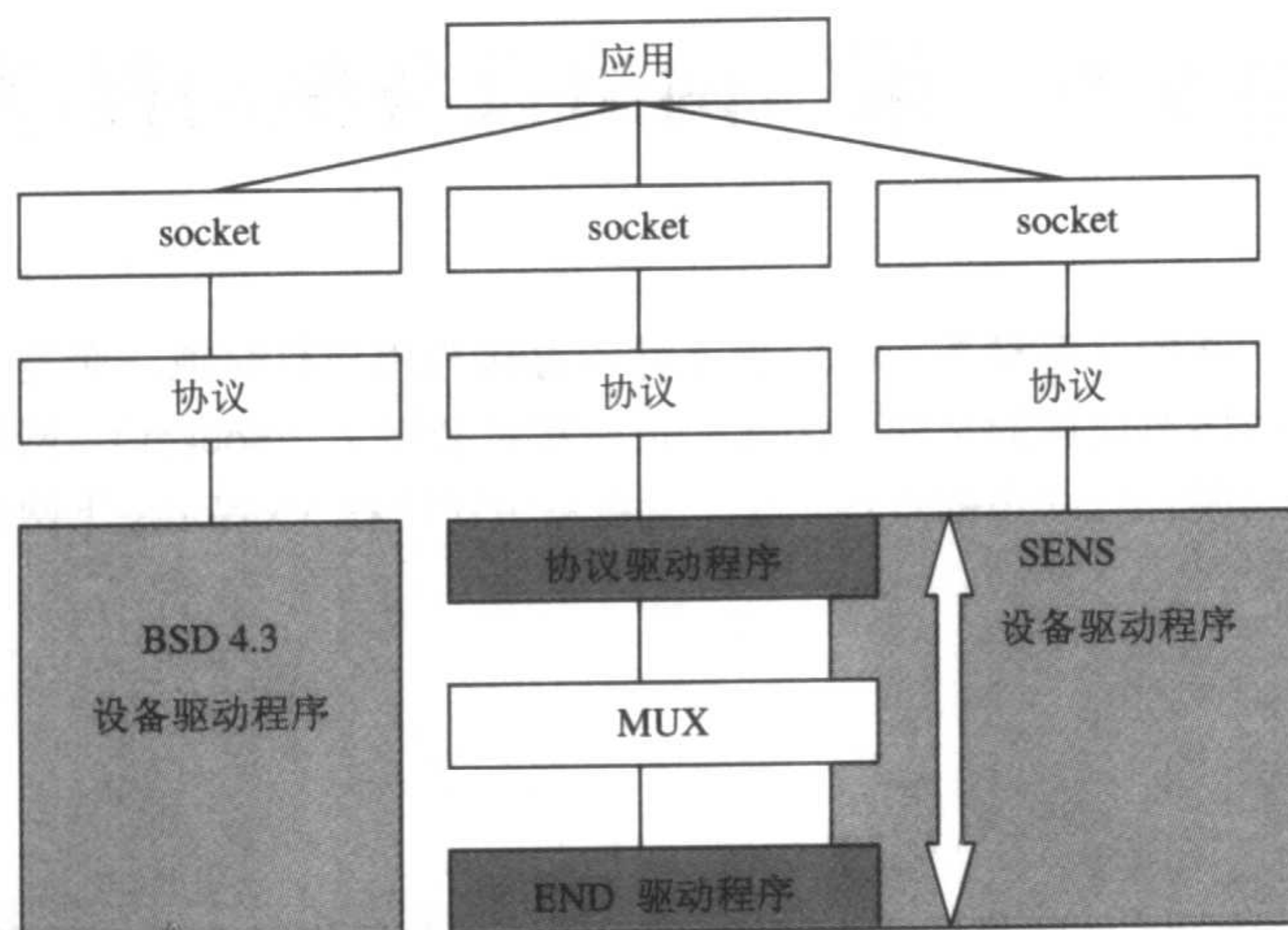


图 9.2 VxWorks 的网络系统

SENS 模型独立于硬件设备接口, 它将网络设备驱动程序细化, 使开发者可以专注于驱动程序 (END 驱动程序) 本身的发展。在驱动程序与上层协议 (例如 TCP/IP) 之间, SENS 模型提供了协议驱动层。在协议层与驱动程序之间, SENS 模型提供了 MUX 层。MUX 层直接与 END 驱动程序相交互, 其应用程序接口提供了独立于网络协议的驱动程序接口, 可以与多个独立的 END 驱动程序同时交互, 如图 9.3 所示。



图 9.3 MUX 的作用

MUX 层管理着网络协议接口和底层驱动程序之间的通信, 使发送、接收数据的过程变得简单, 而不需要再通过挂接钩子程序的办法解决。

SENS 支持在 END 驱动程序中实现 4.4 BSD IP 协议, 同时还允许驱动程序包含诸如多播、轮询模式、零拷贝缓冲等特性。关于网络应用的详细内容请参考相关书籍。

9.2 END 设备驱动程序装载过程

END 设备驱动程序装载过程实际上是分三个步骤来完成的。即指定 END 设备、装载 END 设备和启动 END 设备。

9.2.1 指定 END 设备

与其他类型设备的驱动程序安装方式都不同，网络设备的驱动程序安装时通过修改相应的 BSP 实现。在修改 BSP 之前，首先看一下 BSP 中的 END 设备驱动程序入口表 END_TBL_ENTRY 的结构，其结构原型如下：

```
typedef struct end_tbl_entry
{
    int unit;                /* 设备单元 */
    END_OBJ* (*endLoadFunc) (char*, void*); /* 设置装载函数 */
    char* endLoadString;     /* 装载字符串，作为装载函数的输入参数 */
    BOOL endLoan;            /* 是否借出缓冲区 */
    void* pBSP;              /* BSP 私有信息 */
    BOOL processed;          /* 此设备单元是否已经被处理 */
} END_TBL_ENTRY;
```

在 BSP 的 configNet.h 文件中定义着 END_TBL_ENTRY 结构的数组 endDevTbl[]，其结构原型如下所示：

```
END_TBL_ENTRY endDevTbl [] =
{
    ...
#ifdef INCLUDE_FEI_END
    { 0, FEI82557_LOAD_FUNC, FEI82557_LOAD_STRING, FEI82557_BUFF_LOAN,
      NULL, FALSE},
#endif /* INCLUDE_FEI_END */

#ifdef INCLUDE_ELT_3C509_END
    {0, END_3C509_LOAD_FUNC, END_3C509_LOAD_STRING, END_3C509_BUFF_LOAN,
      NULL, FALSE},
#endif /* INCLUDE_ELT_3C509_END */
    ...
};
```

该数组描述了系统中的所有网络设备的装载函数入口点及其相关参数。将网络设备驱动程序的装载函数（其函数命名格式为 xxLoad()）入口点及相关参数添加到网络设备表 endDevTbl[] 中，而后在 config.h 中加入 “#define INCLUDE_END” 以使系统准备初始化 MUX 并通过 MUX 装载网络设备驱动程序。

在 VxWorks 网络驱动程序的编程规范中，为了系统的高移植性、兼容性和可裁剪性等，通常在 configNet.h 文件中对系统支持的网络设备（以 Intel 82557 网络设备为例）定义一些宏，其编写规范如下所示：

```
#ifdef INCLUDE_FEI_END
#define FEI82557_LOAD_FUNC    fei82557EndLoad /* 设备驱动程序装载函数 */
#define FEI82557_BUFF_LOAN    1               /* 允许缓冲借出 */
```



```

/*
 * fei82557End 初始化字符串格式:
 * <memBase>:<memSize>:<nCFDs>:<nRFDs>:<userFlags>
 */

#define FEI82557_LOAD_STRING "-1:0x00:0x20:0x20:0x00"
IMPORT END_OBJ* FEI82557_LOAD_FUNC (char*, void*);
#endif /* INCLUDE_FEI_END */

```

注意

对于包含 END 设备的 config.h 文件, BSP 默认定义了 INCLUDE_END.

9.2.2 装载及启动 END 设备

这一节主要讨论 END 设备的装载和启动以及标准的 END 入口点。当需要设计和测试网络设备驱动程序时, 需要了解相关的内容。其内容概括如下:

- 装载和启动驱动程序的任务;
- 为驱动程序注册中断处理程序的任务;
- 使用驱动程序处理数据包的任务。

9.2.2.1 装载及启动 END 设备的系统函数

在阅读本章有关内容前, 需要了解如下几个系统函数。

1. muxDevLoad()

该函数装载指定设备的驱动程序装载函数。如果要装载 END 设备, 系统必须调用 muxDevLoad() 函数。其函数原型如下:

```

END_OBJ* muxDevLoad ()
{
    ...
    bzero((char *) devName, END_NAME_MAX);
    if (endLoad((char *) devName, NULL) != 0)
        goto muxLoadErr;
    if (endFindByName((char *) devName, unit) != NULL)
        goto muxLoadErr;
    ... /* 在 endDevTbl[]表中查寻设备 */

    sprintf((char *) initString, "%d:%s", unit, pInitString);
    pNew = (END_OBJ *) endLoad((char *) initString, pBSP);
    if (pNew == NULL) goto muxLoadErr;
    ...
    return (pNew);
muxLoadErr: /* 出错返回 */
    ...
    return (NULL);
}

```


2. muxDevStart()

启动 END 设备函数。该函数调用示例如下：

```
for (count = 0, pDevTbl = endDevTbl; pDevTbl->endLoadFunc != END_TBL_END;
    pDevTbl++, count++)
{
    /* 确认 WDB 调试代理没有安装这个设备。 */
    if (!pDevTbl->processed)
    {
        /* 安装设备驱动程序 */
        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;
        }
    }
}

/* 启动 END 设备 */
if (muxDevStart(pCookie) == ERROR)
{
    printf("muxDevStart failed for device entry %d!\n", count);
}
}
```

3. muxBind()

将协议绑定到指定的 END 设备上。其调用过程是系统调用 ipAttach() 函数，而该函数调用 muxBind() 函数，绑定协议堆栈到 MUX 上的一个指定的网络接口。当一个网络接口被关闭时，ipDetach() 函数将释放网络接口所关联的 TCP/IP 堆栈模块。

9.2.2.2 装载及启动 END 设备驱动程序

使用指定的 END 设备驱动程序的 BSP 引导 VxWorks 系统时，在引导过程中，系统将执行任务 tUsrRoot 来完成如下各项：

- 初始化网络任务的工作队列；
- 创建 tNetTask 任务来处理网络任务工作队列中的条目；
- 调用 muxDevLoad() 装载指定的网络驱动程序；
- 调用 muxDevStart() 启动指定的网络驱动程序。

1. 装载驱动程序到 MUX 中

tUsrRoot 任务调用 MUX 设备装载函数 muxDevLoad() 和设备启动函数 muxDevStart()，通过这两个函数来装载和启动设备驱动程序。

MuxDevLoad() 函数会根据网络设备表 endDevTbl[] 中的定义逐一调用各网络设备的装载函数

(例如 Intel 82557 网络接口的设备装载函数 `fei82557EndLoad()`), 其中这个函数是 `muxDevLoad()` 的一个输入参数。`muxDevStart()` 函数也会根据 `endDevTbl[]` 逐一调用网络设备所定义的设备开始函数。至此 MUX 的初始化以及 END 驱动程序的初始化工作已经完成。

但是网络设备还是不可以使用, 因为还需要将协议绑定到指定的 END 指定的设备上, 这一步骤需要调用 `muxBind()` 实现。

在 VxWorks 中, 系统通过 `usrRoot()` 函数调用 `usrNetInit()` 函数完成 MUX 的初始化、装载网络设备表 `endDevTbl[]` 中描述的所有设备并将 IP 协议绑定到网络引导设备上等工作。

用户和驱动程序的编程者都可以不关心 `muxDevLoad()` 到底做了什么, 但它确实做了一件重要的事情, 就是将网络设备驱动程序所创建的 END 设备添加到 END 设备链表中。

网络设备驱动程序的安装过程即图 9.4 所示的网络初始化顺序。

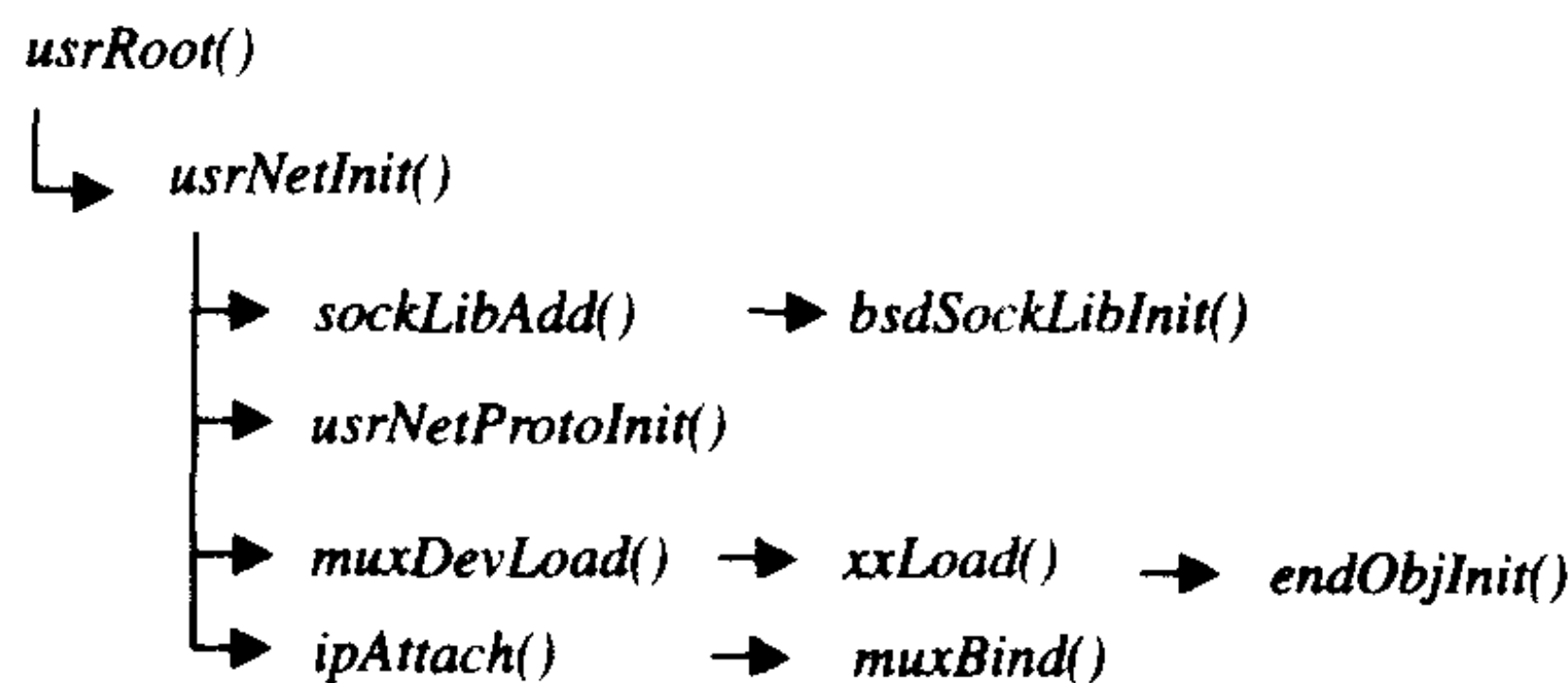


图 9.4 网络初始化顺序

2. 注册驱动程序的中断服务程序

为了注册驱动程序的中断服务程序, 驱动程序需要一些函数的支持, 而在 VxWorks 系统中对于网络设备通常是通过宏来调用这些函数, 这些宏包括 `SYS_INT_CONNECT`、`SYS_INT_ENABLE`, 采用这种方式的好处是便于驱动程序将来的移植和兼容性。

宏 `SYS_INT_CONNECT` 被用来注册中断服务程序, 其宏定义的模板如下:

```

#ifndef SYS_INT_CONNECT
FUNCPTR templateIntConnectRtn = intConnect;
#define SYS_INT_CONNECT(pDrvCtrl,rtn,arg,pResult) \
do { \
    *pResult = (*templateIntConnectRtn) ((VOIDFUNCPTR *) \
        INUM_TO_IVEC (pDrvCtrl->ivec), rtn, (int)arg); \
} while (0)
#endif

```

宏 `SYS_INT_ENABLE` 被用来打开网络设备的中断, 其宏定义的模板如下:

```

#ifndef SYS_INT_ENABLE
STATUS (*templateIntEnableRtn)(int level) = intEnable;
#define SYS_INT_ENABLE(pDrvCtrl) \
do { \
    (*templateIntEnableRtn) (pDrvCtrl->ilevel); \
} while (0)
#endif

```


从上述宏定义中可以发现，注册中断和打开中断宏实际上可以调用 `intConnect()` 和 `intEnable()` 函数来实现，具体调用在网络设备驱动程序的启动函数里执行。当 `muxDevLoad()` 装载驱动程序时，它调用 `muxDevStart()` 函数，通过这个函数系统调用驱动程序启动函数。

9.3 网络设备与系统的数据交换

网络设备与系统的数据交换实际上就是数据的发送和接收问题，数据的发送和接收通常有两种方式，即中断方式和轮询方式。这一部分主要对采用中断方式进行数据发送和接收进行详细描述，而轮询方式在 9.4.3 节中进行详细描述。

9.3.1 驱动程序中断处理的概述

当网络接口产生中断时，系统调用用户驱动程序注册的中断服务程序（中断服务程序）。中断服务程序应该做尽可能少的工作，以完成将数据包从本地网络设备送出的操作。为了将中断阻塞减到最小，网络驱动程序的中断服务程序应当只是处理那些只需要最小执行时间的工作，例如错误或状态改变。对于那些所有耗时处理的任务，网络驱动程序的中断服务程序只是把它们排列到网络任务的工作队列中即可。

为了排列任务级的包接收处理工作，网络驱动程序中断服务程序必须调用 `netJobAdd()` 函数。作为输入，这个函数认可一个函数指针和最多 5 个附加参数（实际上，这 5 个参数是提供给函数指针指向的函数，且是该函数的输入参数）。

`netJobAdd()` 函数原型如下：

```
STATUS netJobAdd (FUNCPTR routine, int arg1, ..., int arg5);
```

在调用 `netJobAdd()` 函数时，应当指定任务级处理数据包的驱动程序入口。然后由 `netJobAdd()` 把函数指派到网络系统任务——`tNetTask` 的工作队列中。`VxWorks` 通过 `tNetTask` 处理任务级的网络处理。

`tNetTask` 调用队列中处理程序如下所示。

- 包接收程序：把接收到的数据包上传到网络缓冲区的堆栈中，通过一个调用上传给 MUX。
- 释放所有发送帧程序：程序调用 `netClFree()` 函数释放发送缓冲区中的所有已经发送的数据帧。

`tNetTask` 的睡眠依附于它的工作队列。`netJobAdd()` 调用把指定的任务级处理程序添加到 `tNetTask` 所依附的工作队列，并自动产生适当的信号量来激活 `tNetTask` 任务。

`tNetTask` 被激活后，`tNetTask` 从它的工作队列中列出函数和分配的参数，然后在它的上下文中执行这些函数。只要队列中有函数，`tNetTask` 就一直工作。当队列为空，所有的数据包已经成功上传到 MUX 中时，`tNetTask` 回到睡眠态。



注意

谨慎使用 `netJobAdd()` 函数。`netJobAdd()` 是能够被网络堆栈同时使用的有限资源，如果其溢出将会导致通用网络堆栈的瘫痪。

9.3.2 END 的中断服务程序: xxInt()

xxInt()函数在网络设备的数据包接收和发送中扮演着重要角色。xxInt()负责处理发送中断和接收中断,其函数格式如下:

```
void xxInt    (xxEND_DEVICE *pDrvCtrl);
```

其中,输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针,在 9.4.2 小节中进行了描述。

xxInt()函数执行了几个必要的操作:读取设备状态、进行发送处理或接收处理。其中发送处理和接收处理通过任务来实现。

下面是网络驱动程序中断服务程序 xxInt()函数的模板:

```
=====
templateInt——网络驱动程序中断服务程序函数模板
=====
```

```
void templateInt
(
    xxEND_DEVICE *pDrvCtrl /* interrupting device */
)
{
    UCHAR stat;

    DRV_LOG (DRV_DEBUG_INT, "Got an interrupt!\n", 1, 2, 3, 4, 5, 6);

    /* 读设备状态寄存器 */

    stat = templateStatusRead (pDrvCtrl);

    /* 如果是错误的中断,返回。 */

    if (!(stat & TEMPLATE_VALID_INT)) /* 测试有效中断 */
    {
        return; /* 立即返回,不带错误消息 */
    }

    /*
     * 打开中断,清除接收和/或发送中断,清除任何可能设置的错误
     */

    ... /* 这个位置需要增加错误检测代码 */

    /* netTask 处理任何输入的数据包 */

    if ((stat & TEMPLATE_RINT) && (stat & TEMPLATE_RXON))
    {
        if (!(pDrvCtrl->rxHandling))
        {
            pDrvCtrl->rxHandling = TRUE;
            netJobAdd ((FUNCPTR)templateHandleRcvInt, (int)pDrvCtrl,
                      0, 0, 0, 0);
        }
    }
}
```



```

/* 处理发送中断 */
if ((stat & TEMPLATE_TINT) && (stat & TEMPLATE_TXON))
{
    if (!(pDrvCtrl->txHandling))
    {
        pDrvCtrl->txHandling = TRUE;
        netJobAdd ((FUNCPTR)templateHandleTxRingFree, (int)pDrvCtrl,
                    0,0,0,0);
    }
}
}

```

其中, DRV_LOG 宏定义是:

```

#define DRV_LOG(FLG, X0, X1, X2, X3, X4, X5, X6)
if (xxDebug & FLG)
    logMsg(X0, X1, X2, X3, X4, X5, X6);

```

9.3.3 驱动程序和协议层之间共享缓冲区

网络设备使用环行缓冲描述符。在装载设备驱动程序时,设备驱动程序装载函数分配了接收和发送缓冲区。当接收数据包时,首先把数据存放在设备接收缓冲区中,然后把指向接收缓冲区的指针传递给 MUX。当发送数据包时,首先把发送缓冲区中的数据包拷贝到设备发送缓冲区中,然后把缓冲区放置到发送队列中并启动发送器,最后把已经发送的数据包缓冲区归还给内存池。

VxWorks 网络系统使用 mBlk 和 cBlk 结构来跟踪必要的信息以便管理簇中的数据。簇包含 mBlk 和 cBlk 结构的数据描述。用户主要通过 mBlk 结构来访问和传递由 netPoolInit()建立的内存池中存放的数据。由于 mBlk 仅访问数据,这使得网络层不需要拷贝数据便可以进行数据交换。mBlk 结构的另一个特征是它的链接性,这使得用户可以通过 mBlk 的链头传递任意大的数据。如图 9.5 所示是两个网络数据包。

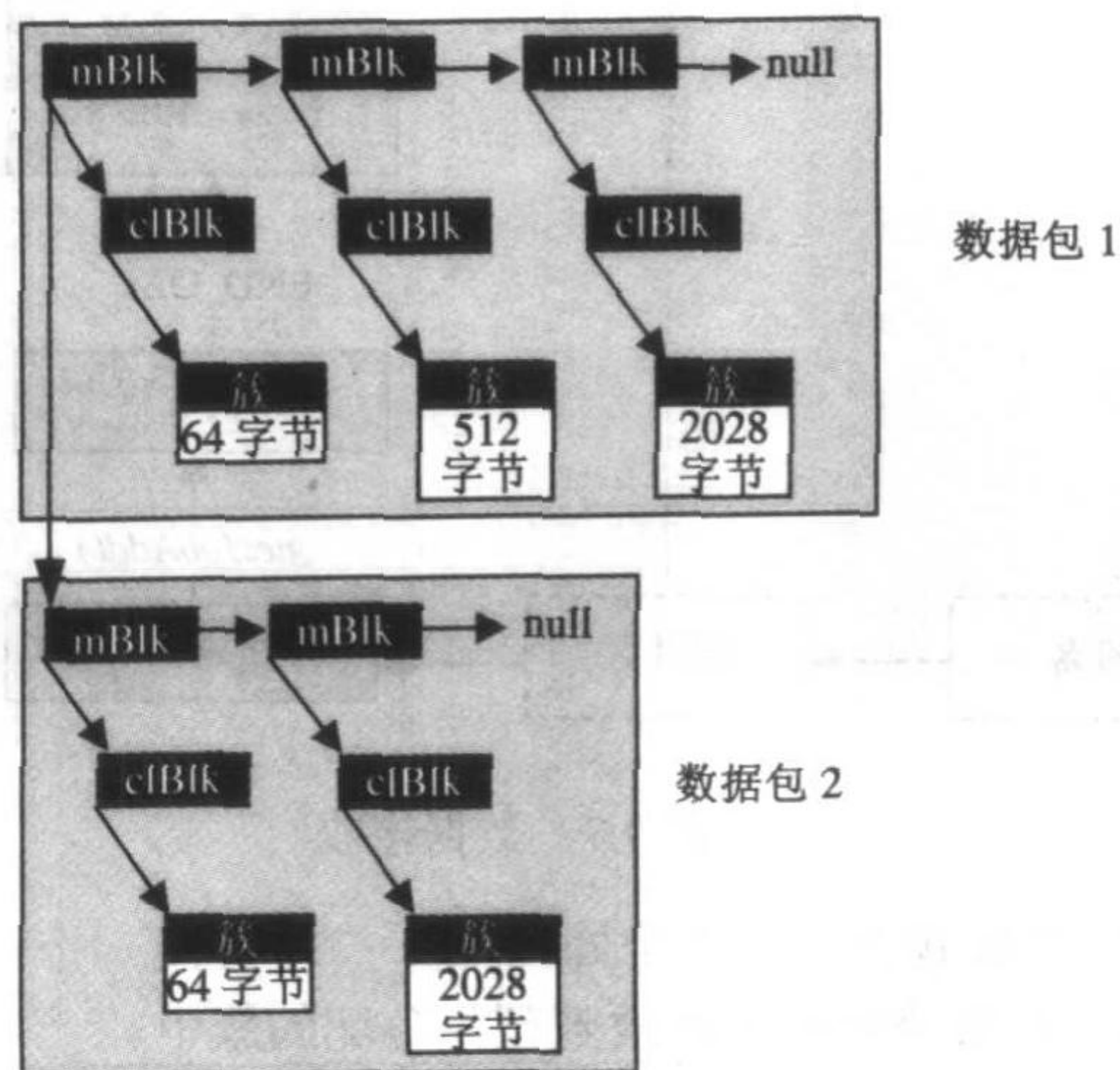


图 9.5 两个网络数据包

另外, mBlk 结构通过 cBlk 结构访问数据, 这使得多个 mBlk 结构很容易共享同一个簇。当用户需要拷贝 mBlk 而不是拷贝它所访问的数据时, 此功能非常有效。例如图 9.6 所示的数据包共享簇。

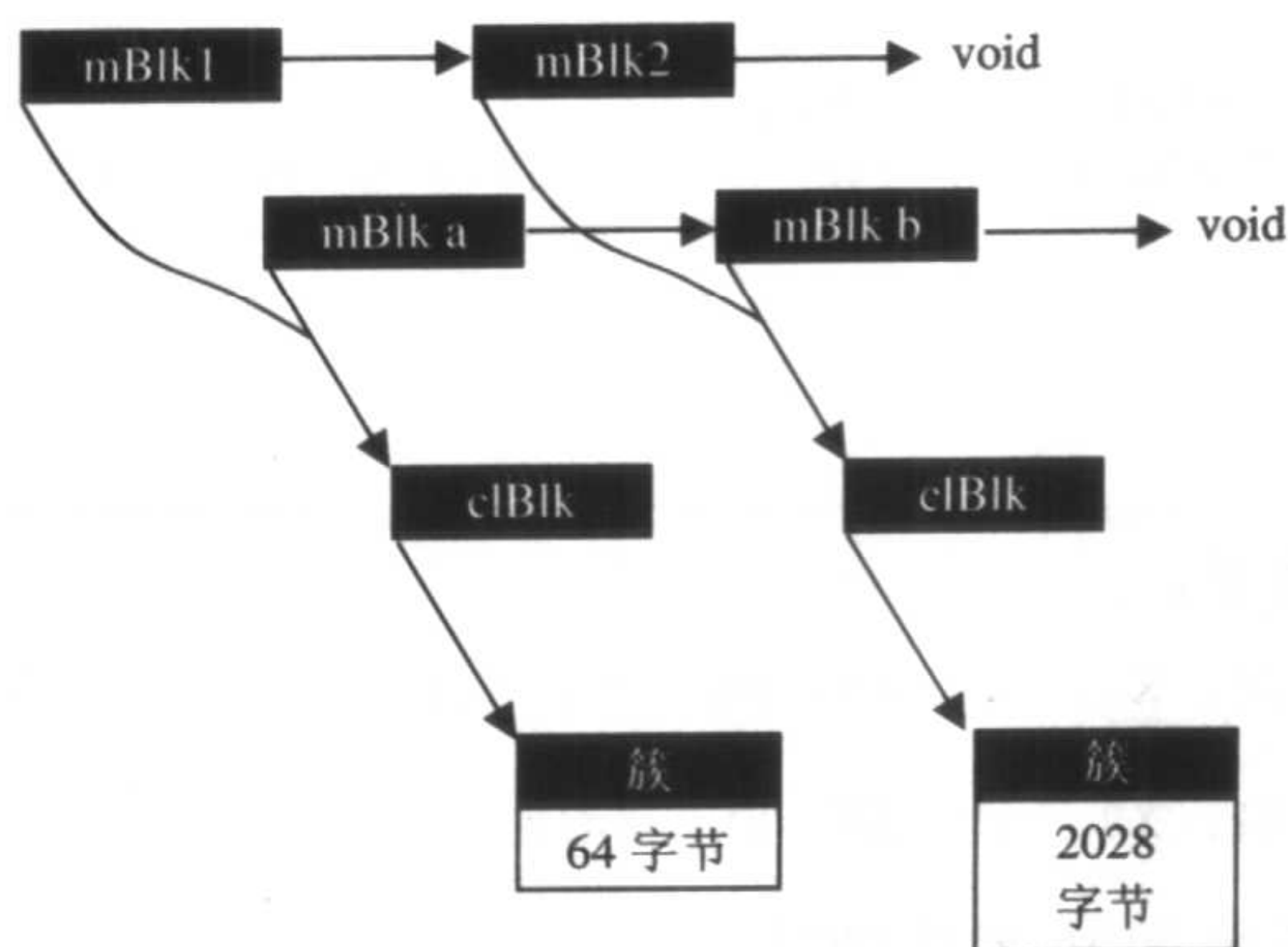


图 9.6 数据包共享簇

9.3.4 接收数据

1. 接收数据概述

在 VxWorks 网络系统中, 接收数据的流程如图 9.7 所示。

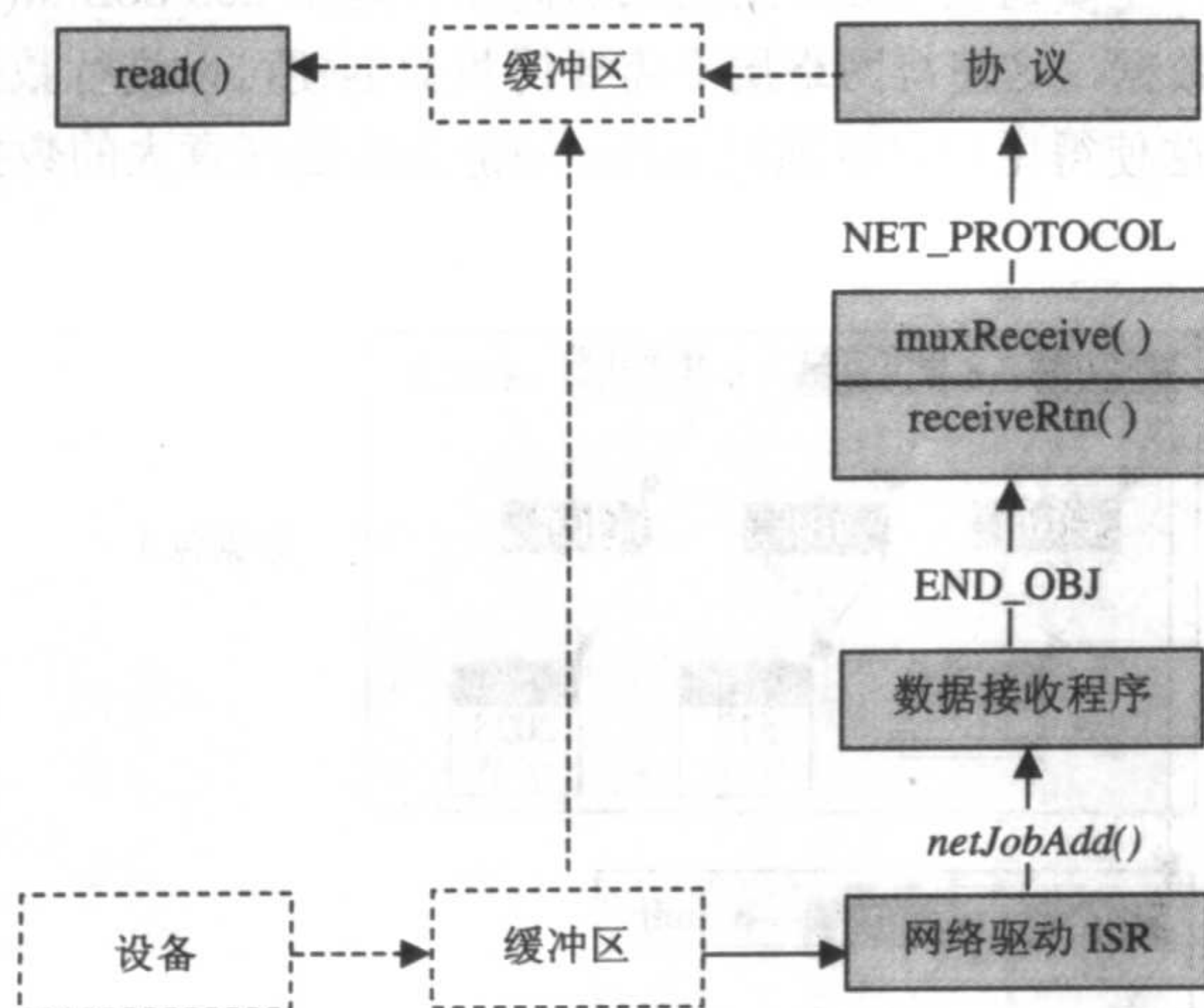


图 9.7 数据接收

对图 9.7 进行分析, 它主要包含以下几个处理。

- (1) 设备接收到数据包后直接把数据存放到预先分配的簇中。
- (2) 当接收到中断时, 驱动程序的中断服务程序调度任务级接收程序进行如下操作。

- cBlk 结构和簇连接；mBlk 和 cBlk 连接；最后构成缓冲区。

在这个操作步骤中调用了如下几个函数。

netClusterGet(): 预定一块簇缓冲区。

netClBlkGet(): 预定一个 cBlk 结构。

netMblkGet(): 预定一个 mBlk 结构。

netClBlkJoin: 把簇添加到 cBlk 结构中。

netMblkClJoin: 把 cBlk 结构添加到 mBlk 结构中。

- 通过调用 receiveRtn()函数，把缓冲区传递给更高级别的协议。

(3) muxReceive()调用协议的 stackRcvRtn()函数，把成列的缓冲区传递给应用。

(4) 用户使用 read()函数，通过套接字访问网络中的成列缓冲区。

2. 任务级接收程序

任务级接收程序主要是接收数据包并进行包检测。其包接收过程首先是申请一块簇缓冲区，接着把接收到的数据包存放到这个簇缓冲区中，然后把簇添加到 cBlk 结构中，再把 cBlk 结构添加到 mBlk 结构中。接下来把数据打包在一起，调用上层接收程序向上层提交数据包，最后释放已经上传的数据包所占用的空间。

(1) 接收数据包中断服务程序。下面是一个任务级接收数据包中断服务程序的一个模板函数。

```
=====
templateHandleRcvInt——任务级接收数据包中断服务程序
=====
/* 中断服务程序在任务级间接调用这个程序，处理接收到的任何消息。
 * 程序中采用双循环保护竞态条件，在该函数里中断代码把 rxHandling 当作 TRUE，但
 * 除任务代码改变 rxHandling 值之外。
 * 循环空转并不是致命的，当下一个数据包到来之前该处理处于挂起状态。一旦接收到数据
 * 包，系统立即触发 netTask 再次调用这个程序。
 *
 * 返回值：空。
 */

void templateHandleRcvInt
(
    xxEND_DEVICE *pDrvCtrl /* 中断设备 */
)
{
    PKT* pPkt;

    do
    {
        pDrvCtrl->rxHandling = TRUE;

        while ((pPkt = templatePacketGet (pDrvCtrl)) != NULL)
            templateRecv (pDrvCtrl, pPkt); /* 处理接收数据包 */

        pDrvCtrl->rxHandling = FALSE;
    }
    while (templatePacketGet (pDrvCtrl) != NULL);
}
```


(2) 处理接收到的数据包。下面是处理接收数据包的一个模板函数。

```

=====
templateRecv——处理接收到的数据包
=====
/* 处理一个输入的数据包。这个包需要错误检测。
 *
 * 返回值：空。
 */

LOCAL STATUS templateRecv
(
    xxEND_DEVICE *pDrvCtrl,      /* 设备驱动程序控制结构 */
    PKT* pPkt                    /* 被处理的数据包 */
)
{
    int len;
    M_BLK_ID pMblk;
    char* pCluster;
    char* pNewCluster;
    CL_BLK_ID pClBlk;

    ... /* 在这个位置必须增加包错误检测代码 */

    END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_UCAST, +1);

    /*
     * 预定一块簇缓冲区，如果必须执行拷贝动作则这个步骤可以跳过，但数据被递交
     * 给协议之前，必须先进行拷贝。
     */

    pNewCluster = netClusterGet (pDrvCtrl->end.pNetPool, pDrvCtrl->pClPoolId);

    if (pNewCluster == NULL)
    {
        DRV_LOG (DRV_DEBUG_RX, "Cannot loan!\n", 1, 2, 3, 4, 5, 6);
        END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
        goto cleanRXD;
    }

    /* 现在需要在网络环行缓冲区中预定一个 clBlk 结构 */

    if ((pClBlk = netClBlkGet (pDrvCtrl->end.pNetPool, M_DONTWAIT)) == NULL)
    {
        netClFree (pDrvCtrl->end.pNetPool, pNewCluster);
        DRV_LOG (DRV_DEBUG_RX, "Out of Cluster Blocks!\n", 1, 2, 3, 4, 5, 6);
        END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
        goto cleanRXD;
    }

    /*
     * 已经获得了备用空间，现在需要在网络环行缓冲区中预定一个 mBlk 结构
     */

```



```

    if ((pMblk = netMblkGet (pDrvCtrl->end.pNetPool, M_DONTWAIT, MT_DATA)) == NULL)
    {
        netClBlkFree (pDrvCtrl->end.pNetPool, pClBlk);
        netClFree (pDrvCtrl->end.pNetPool, pNewCluster);
        DRV_LOG (DRV_DEBUG_RX, "Out of M Blocks!\n", 1, 2, 3, 4, 5, 6);
        END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_ERRS, +1);
        goto cleanRXD;
    }

    END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_UCAST, +1);
    len = TEMPLATE_PKT_LEN_GET (pPkt);
    pCluster = TEMPLATE_PKT_VIRT_GET (pPkt);

    /* 把簇添加到 clBlk 结构中。 */
    netClBlkJoin (pClBlk, pCluster, len, NULL, 0, 0, 0);

    /* 把 clBlk 结构添加到 mBlk 结构中。 */
    netMblkClJoin (pMblk, pClBlk);

    pMblk->mBlkHdr.mLen = len;
    pMblk->mBlkHdr.mFlags |= M_PKTHDR;
    pMblk->mBlkPktHdr.len = len;

    /* 把包数据粘在一起 */

    END_CACHE_INVALIDATE (pMblk->mBlkHdr.mData, len);

    DRV_LOG (DRV_DEBUG_RX, "Calling upper layer!\n", 1, 2, 3, 4, 5, 6);

    /* 调用上层接收程序 */
    END_RCV_RTN_CALL (&pDrvCtrl->end, pMblk);

    ... /* 在这个位置需要增加释放已经上传的数据包所占用的空间代码 */

cleanRXD:

    return (OK);
}

```

9.3.5 发送数据

在 VxWorks 网络系统中，发送数据的流程如图 9.8 所示。

对图 9.8 进行分析，主要包含以下几个处理。

- (1) 用户调用 write() 函数，通过套接字访问网络。
- (2) 网络协议拷贝需要发送的数据到网络缓冲区中，并调用协议驱动程序的发送程序。
- (3) 协议驱动程序调用 muxSend() 启动发送循环。
- (4) muxSend() 通过调用 send 回调函数，把缓冲区传递给 END。
- (5) 数据发送程序把数据拷贝到设备缓冲区中，并把它放置到设备的发送队列中。

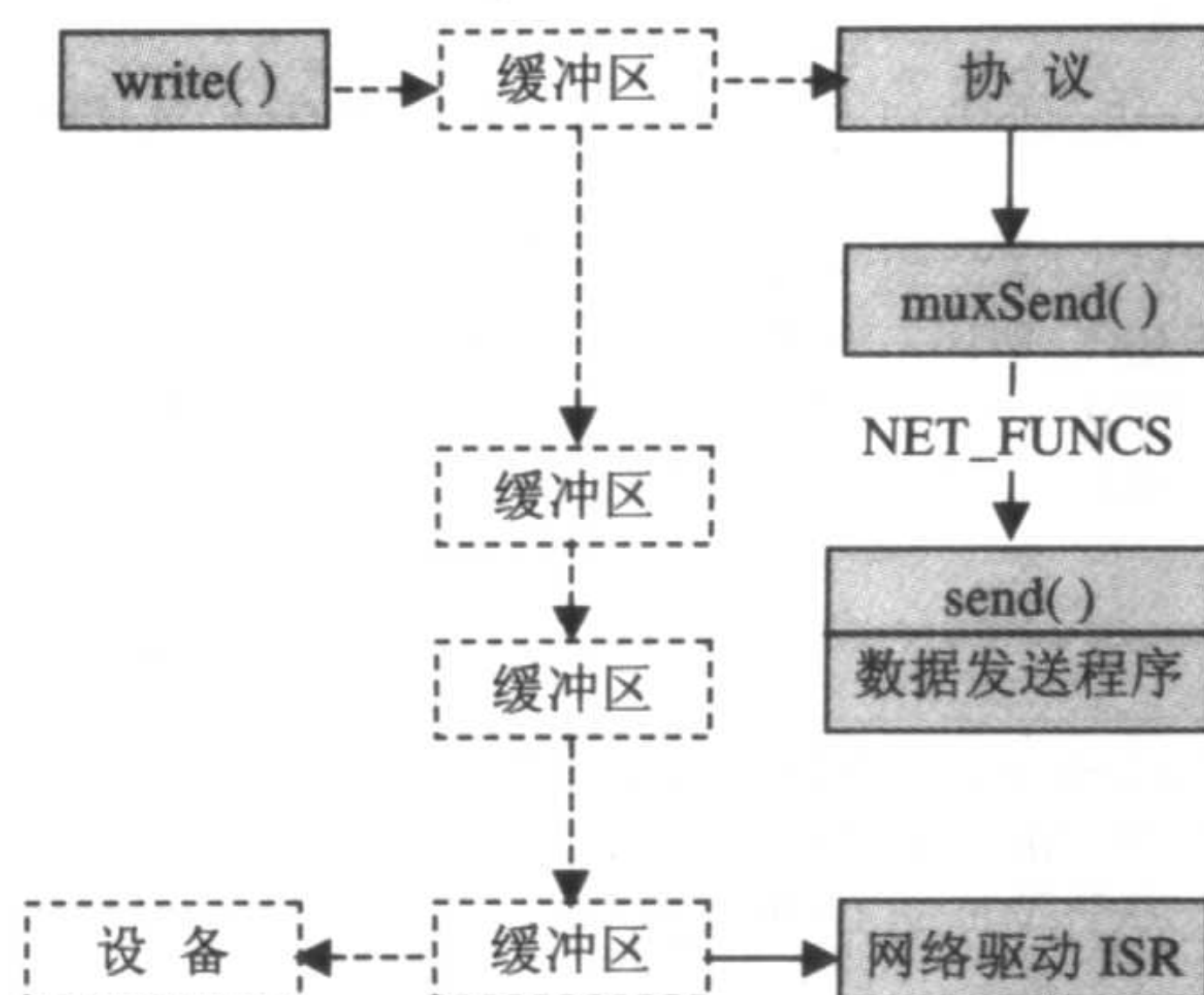


图 9.8 数据发送

(6) 当产生发送中断时，驱动程序的中断服务程序调度程序丢弃已经发送的数据包，彻底清理发送队列。

9.4 一步步编写网络设备驱动程序

9.4.1 END 驱动程序的重要数据结构

网络接口作为一个设备对象，和其他类型设备的驱动程序一样，其网络设备的 END 驱动程序也有相应的数据结构，系统在访问设备时把设备作为一个对象来看待。这些内容将在这一部分进行详细描述。

1. 基本 END 结构：END_OBJ

END_OBJ 结构中定义了所有网络相关的部分，提供了一个独立于设备的数据结构。用户在定义驱动程序控制数据结构（例如 xxEND_DEVICE）时，应包含 END_OBJ 结构，并且必须将其放置在其他结构成员的第一位，以避免不必要的麻烦。

END_OBJ 中包含了两个比较重要的结构，DEV_OBJ 和 NET_FUNCS，这两个结构会在后面讨论。END_OBJ 中同时还包含了接收过程的回调函数指针以及与协议相关的数据，这些数据都是为 MUX 层提供的，用户可以不必理会。

END_OBJ 结构原型如下：

```

typedef struct end_object
{
    NODE node; /* 设备的根节点 */
    DEV_OBJ devObject; /* 被访问设备的控制结构 */
    STATUS (*receiveRtn) (); /* 接收程序 */
    struct net_protocol *outputFilter; /* 可选输出包过滤程序 */
    void* pOutputFilterSpare; /* 输出过滤的备用指针 */
    BOOL attached; /* 单元绑上表明 */
    SEM_ID txSem; /* 发送数据信号量 */
    long flags; /* 多用途标识 */
    struct net_funcs *pFuncTable; /* 设备驱动程序函数表 */
}
  
```



```

M2_INTERFACETBL mib2Tbl;      /* MIBII 计数器 */
LIST multiList;               /* 多点传送地址表头 */
int nMulti;                   /* 多点传送地址表单元数 */
LIST protocols;               /* 协议节点列表 */
int snarfCount;               /* 表头协议数 */
NET_POOL_ID pNetPool;         /* 被 MUX 缓冲区使用的内存 */
M2_ID * pMib2Tbl;             /* RFC 2233 MIB 对象 */
} END_OBJ;

```

2. 设备控制结构: DEV_OBJ

DEV_OBJ 包括了设备名、设备描述等信息, 用户驱动程序使用该结构来控制设备。其中最重要的是它还包含了一个指向用户定义的驱动程序控制结构的指针——pDevice。

DEV_OBJ 的结构原型如下:

```

typedef struct dev_obj
{
    char name[END_NAME_MAX];      /* 设备名 */
    int unit;                     /* 设备单元 */
    char description[END_DESC_MAX]; /* 设备描述 */
    void *pDevice;                /* 指向设备驱动程序控制结构的一个指针 */
} DEV_OBJ;

```

3. 网络设备驱动程序函数表: NET_FUNCS

END_OBJ 还包含了指向结构 NET_FUNCS 的一个指针, 这个结构包含了设备驱动程序各函数的入口点。

NET_FUNCS 的结构原型如下:

```

typedef struct net_funcs
{
    STATUS (*start) (END_OBJ*);      /* 驱动程序的启动函数 */
    STATUS (*stop) (END_OBJ*);       /* 驱动程序的停止函数 */
    STATUS (*unload) (END_OBJ*);     /* 驱动程序的卸载函数 */
    int (*ioctl) (END_OBJ*, int, caddr_t); /* 驱动程序的控函数 */
    STATUS (*send) (END_OBJ*, M_BLK_ID); /* 驱动程序的发送函数 */
    STATUS (*mCastAddrAdd) (END_OBJ*, char*); /* 驱动程序的添加多点传送地址函数 */
    STATUS (*mCastAddrDel) (END_OBJ*, char*); /* 驱动程序的删除多点传送地址函数 */
    STATUS (*mCastAddrGet) (END_OBJ*, MULTI_TABLE*); /* 驱动程序的获得多点传送地址表函数 */
    STATUS (*pollSend) (END_OBJ*, M_BLK_ID); /* 驱动程序的轮询发送函数 */
    STATUS (*pollRcv) (END_OBJ*, M_BLK_ID); /* 驱动程序的轮询接收函数 */
    M_BLK_ID (*formAddress) (M_BLK_ID, M_BLK_ID, M_BLK_ID, BOOL); /* 驱动程序的地址组合函数 */
    STATUS (*packetDataGet) (M_BLK_ID, LL_HDR_INFO*); /* 驱动程序的包数据获得函数 */
    STATUS (*addrGet) (M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID, M_BLK_ID); /* 驱动程序的包地址获得函数 */
    int (*endBind) (void*, void*, void*, long type); /* 在网络服务和网络驱动程序之间进行信息交换的函数 */
} NET_FUNCS;

```

在这个结构中定义的驱动程序函数包括: 开始和停止设备的函数, 发送网络数据包的函数, 设备控制函数, 增加、删除、获取多播地址的函数, 轮询模式下发送和接收的函数, 网络缓冲的操作

函数以及驱动程序资源释放函数，这些函数都将由用户定义，将在后面一一讨论。

4. 跟踪驱动程序和协议之间传送的数据：M_BLK

使用 M_BLK 结构作为一个传送载体，在驱动程序和协议层之间进行数据包的传送。在 netBufLib.h 文件中定义了该结构，DEV_OBJ 的结构原型如下：

```
typedef struct mBlk
{
    M_BLK_HDR    mBlkHdr;           /* mBlk 首部信息 */
    M_PKT_HDR    mBlkPktHdr;        /* 指向 pktHdr 结构的指针 */
    CL_BLK * pClBlk;                /* 一个指向 clBlk 结构的指针 */
} M_BLK;
```

5. 跟踪连接级信息：LL_HDR_INFO

LL_HDR_INFO 数据结构定义的信息只是与堆栈接收程序有关。该结构原型定义如下：

```
typedef struct llHdrInfo
{
    int    destAddrOffset;          /* mBlk 结构中目的地址偏移量 */
    int    destSize;                /* 目的地址大小 */
    int    srcAddrOffset;           /* mBlk 结构中源地址偏移量 */
    int    srcSize;                 /* 源地址大小 */
    int    ctrlAddrOffset;          /* mBlk 结构中控制信息偏移量 */
    int    ctrlSize;                /* 控制信息大小 */
    int    pktType;                 /* 数据包类型 */
    int    dataOffset;              /* mBlk 结构中数据偏移量 */
} LL_HDR_INFO;
```

MUX 使用 LL_HDR_INFO 结构来保存连接级首部信息，并联合从 END 传送到 MUX 的数据包向协议层提交。LL_HDR_INFO 结构作为一个参数传递给所有堆栈接收程序。

9.4.2 用户自定义结构和一些结构的初始化

1. 驱动程序控制结构

不同网络设备其本身资源、驱动程序等方面各不相同。因此，在编写驱动程序前需要为网络设备定义一个驱动程序控制结构。驱动程序通过这个结构获得所需要的资源并进行相应的处理。其中，这个结构必须包含 END_OBJ 结构，并放在该结构的第一项。但是该结构中的内容是根据网络设备和用户的需要而设计的。

下面是驱动程序控制结构的一个模板：

```
typedef struct end_device
{
    END_OBJ    end;                 /* 继承的 END_OBJ 结构 */
    int        unit;                /* 设备单元数 */
    int        ivec;                /* 中断向量 */
    int        ilevel;              /* 中断级 */
    char*      pShMem;              /* 一个指向共享缓冲区的指针 */
    long       flags;               /* 一个标识 */
    UCHAR      enetAddr[6];         /* 以太网地址 */
    CACHE_FUNCS cacheFuncs;         /* 高速缓存功能函数指针 */
    FUNCPTR    freeRtn[128];        /* 释放程序数组 */
    struct free_args freeData[128]; /* 释放程序的释放参数数组 */
}
```



```

CL_POOL_ID    pClPoolId;          /* 簇缓冲池 */
BOOL          rxHandling;          /* 接收任务调度标识 */
} xxEND_DEVICE;

```

2. 初始化设备驱动程序表

在装载设备驱动程序前，需要对设备驱动程序表进行初始化操作。

下面是初始化设备驱动程序表的一个模板：

```

LOCAL NET_FUNCS templateFuncTable =
{
    (FUNCPTR) xxStart,          /* 设备启动函数 */
    (FUNCPTR) xxStop,          /* 设备停止函数 */
    (FUNCPTR) xxUnload,        /* 驱动程序的卸载函数 */
    (FUNCPTR) xxIoctl,         /* 驱动程序的控制函数 */
    (FUNCPTR) xxSend,          /* 驱动程序的数据发送函数 */
    (FUNCPTR) xxMCastAddrAdd,  /* 驱动程序的添加多点传送地址函数 */
    (FUNCPTR) xxMCastAddrDel,  /* 驱动程序的删除多点传送地址函数 */
    (FUNCPTR) xxMCastAddrGet,  /* 驱动程序的获得多点传送地址表函数 */
    (FUNCPTR) xxPollSend,      /* 轮询发送函数 */
    (FUNCPTR) xxPollRcv,       /* 轮询接收函数 */
    endEtherAddressForm,       /* 组合地址信息到 NET_BUFFER 里 */
    endEtherPacketDataGet,     /* 获得 NET_BUFFER 里的数据 */
    endEtherPacketAddrGet      /* 数据包地址获得函数 */
};

```

3. 配置网络缓冲区

(1) 初始化网络 mbuf 配置表。

```

/*
 * 这个结构是为了配置 mBlk 数量和簇数据块数量，在网络缓冲池中分配该结构。
 * 每一个数据簇都有一个相应的簇数据块来管理它。一个 mBlk 管理一个簇数据块和数据
 * 簇。网络缓冲池的配置应该能够存放所累积的簇数据块和数据簇。
 * 对于 TCP 应用程序，最好的方式就是每一个簇数据块有两个 mBlk。
 */

```

```

M_CL_CONFIG templateMclBlkConfig =
{
    /*
    mBlk 个数          clBlk 个数          预先分配的存储区          预先分配的存储区大小
    -----
    */
    0,                  0,                  NULL,                  0
};

```

(2) 初始化网络簇池配置表。

```

/*
 * 这个结构定义了网络代码利用簇的规模。
 * 在 usrNetwork.c 文件中填充网络堆栈数据簇池配置表 clDescTbl[]。
 */

```

```

CL_DESC templateClDescTbl [] =
{
    /*

```


9.4.3 网络设备驱动程序函数

表 9-1 驱动程序的函数

9.4.3.1 设备装载程序: xxLoad()

```
END_OBJ *xxLoad(char *initString);
```

xxLoad()函数中执行了几个必要的操作：初始化 END_OBJ 结构、初始化网络缓冲内存、初始化 MIB、设置网络准备好标志。VxWorks 为网络设备驱动程序的开发提供了一些宏定义简化开发

工作。其中初始化 END_OBJ 可以使用宏定义 END_OBJ_INIT, 这个宏定义指向函数 endObjInit(), 执行了初始化工作; 初始化 MIB 可以使用 END_MIB_INIT, 该宏定义指向函数 mib2Init(), 执行了 MIB 的初始化; 宏定义 END_OBJ_READY 指向函数 endObjFlagsSet(), 执行了设置“网络设备准备好”标记的功能。

下面是 xxLoad() 函数的一个模板。

```

=====
templateLoad() —— 驱动程序装载函数模板
=====
/*****
 *
 * templateLoad - 初始化驱动程序和设备
 *
 * 这个函数初始化驱动程序和设备到可操作的状态。
 * 设备的所有特殊参数通过 initString 进行传递。这个字符串包含如下所示的特殊参数:
 * 设备寄存器基地址、中断向量、中断级、共享缓冲区地址、共享缓冲区大小等。
 *
 * 返回值: 一个指向 END 对象指针或一个 NULL 错误状态。
 */

END_OBJ* templateLoad
(
    char* initString          /* 传递给驱动程序的字符串 */
)
{
    xxEND_DEVICE *pDrvCtrl;   /* 驱动程序控制数据结构 */

    DRV_LOG (DRV_DEBUG_LOAD, "Loading template...\n", 1, 2, 3, 4, 5, 6);

    /* 分配设备驱动程序结构 */

    pDrvCtrl = (xxEND_DEVICE *)calloc (sizeof (xxEND_DEVICE), 1);
    if (pDrvCtrl == NULL)
        goto errorExit;

    /* 分析初始化字符串, 并填充到设备驱动程序结构中 */

    if (templateParse (pDrvCtrl, initString) == ERROR)
        goto errorExit;

    /* 询问 BSP 提供以太网地址 */
    SYS_ENET_ADDR_GET(pDrvCtrl);

    /* 初始化 END 和 MIB2 部分结构 */

    /*
     * M2 元素必须来自于 m2Lib.h。
     * 这个模板安装一个 DIX 类型的以太网设备。
     */

    if (END_OBJ_INIT (&pDrvCtrl->end, (DEV_OBJ *)pDrvCtrl, "template",

```



```

        pDrvCtrl->unit, &templateFuncTable,
        "END Template Driver.") == ERROR
    || END_MIB_INIT (&pDrvCtrl->end, M2_ifType_ethernet_csmacd,
        &pDrvCtrl->enetAddr[0], 6, END_BUFSIZ,
        END_SPEED)
    == ERROR)
goto errorExit;

/* 内存的分配 */
if (templateMemInit (pDrvCtrl) == ERROR)
goto errorExit;

/* 复位及重新配置设备 */
templateReset (pDrvCtrl);
templateConfig (pDrvCtrl);

/* 设置标识, 指出设备准备就绪 */
END_OBJ_READY (&pDrvCtrl->end,
    IFF_NOTRAILERS | IFF_BROADCAST | IFF_MULTICAST);

DRV_LOG (DRV_DEBUG_LOAD, "Done loading Template...", 1, 2, 3, 4, 5, 6);

return (&pDrvCtrl->end);

errorExit:

templateUnload(pDrvCtrl); /* 卸载设备驱动程序 */
free ((char *) pDrvCtrl); /* 释放空间 */
return NULL;
}

```

9.4.3.2 启动设备程序: xxStart()

xxStart()函数主要负责完成设备停止校验操作、注册设备驱动程序的中断服务程序和在中断模式里启动设备, 其函数格式如下:

```
STATUS xxStart(xxEND_DEVICE * pDrvCtrl)
```

其中, 输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针, 参见 9.4.2 小节。此函数返回值为 OK 或 ERROR。

xxStart()函数执行了几个必要的操作: 注册设备驱动程序的中断服务程序、打开设备中断、记下设备启动及启动设备。其中注册设备驱动程序的中断服务程序使用宏定义 SYS_INT_CONNECT, 这个宏定义最终通过调用 intConnect()函数注册中断服务程序; 打开设备中断使用宏定义 SYS_INT_ENABLE, 这个宏定义最终通过调用 intEnable()函数打开中断服务程序中断; 记下设备启动使用宏定义 END_FLAGS_SET, 这个宏定义最终通过调用((pEnd)->flags |= (setBits))语句标志设备启动。

下面是 xxStart()函数的一个模板。


```

=====
templateStart——启动设备函数的一个模板
=====
/*****
 * templateStart - 启动设备
 *
 * 这个函数调用 BSP 的函数连接中断和驱动程序设备，使设备在中断模式下工作。
 *
 * 返回值：OK 或 ERROR
 *
 */

STATUS templateStart
(
    xxEND_DEVICE * pDrvCtrl          /* 驱动程序控制数据结构 */
)
{
    STATUS result;
    DRV_LOG (DRV_DEBUG_START, ("Starting end...\n"), 1, 2, 3, 4, 5, 6);

    /* 驱动程序的中断服务程序与设备中断连接 */
    SYS_INT_CONNECT (pDrvCtrl, templateInt, (int)pDrvCtrl, &result);
    if (result == ERROR)
        return ERROR;

    DRV_LOG (DRV_DEBUG_LOAD, "Interrupt connected.\n", 1, 2, 3, 4, 5, 6);

    /* 打开设备中断 */
    SYS_INT_ENABLE (pDrvCtrl);

    DRV_LOG (DRV_DEBUG_LOAD, "interrupt enabled.\n", 1, 2, 3, 4, 5, 6);

    /* 设置设备启动标识 */
    END_FLAGS_SET (&pDrvCtrl->end, IFF_UP | IFF_RUNNING);

    ... /* 在这个位置增加启动设备代码 */

    return (OK);
}
=====

```

9.4.3.3 设备停止程序：xxStop()

xxStop()函数主要负责完成设备的停止操作。其函数格式如下：

```
STATUS xxStop(xxEND_DEVICE * pDrvCtrl);
```

其中，输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针。此函数返回值为 OK 或 ERROR。

xxStop()函数执行了几个必要的操作：关闭系统中断、记下设备停止、复位设备和卸载驱动程序的中断服务程序。其中关闭系统中断使用宏定义 SYS_INT_DISABLE，这个宏定义最终通过调用 intDisable ()函数关闭驱动程序的中断服务程序中断；记下设备停止使用宏定义 END_FLAGS_CLR，

这个宏定义最终通过调用((pEnd)->flags &= ~(clrBits))语句标志设备停止；设备复位操作是通过调用设备复位函数来实现的；卸载设备驱动程序的中断服务程序使用宏定义 SYS_INT_DISCONNECT，这个宏定义可以为空操作或通过调用 pciIntDisconnect()函数实现卸载操作（如果是 PCI 接口网络设备）。

下面是 xxStop()函数的一个模板。

```

=====
templateStop——设备停止函数的一个模板
=====
/*****
 *
 * templateStop——停止设备
 *
 * 这个函数调用 BSP 的函数关闭中断和停止中断模式下工作的设备。
 *
 * 返回值：OK 或 ERROR。
 */

STATUS templateStop
(
    xxEND_DEVICE *pDrvCtrl /* 被停止的设备驱动程序结构 */
)
{
    STATUS result = OK;

    /* 关闭系统中断 */
    SYS_INT_DISABLE (pDrvCtrl);

    /* 标识设备停止 */
    END_FLAGS_CLR (&pDrvCtrl->end, IFF_UP | IFF_RUNNING);

    /* 复位网络接口芯片 */
    templateReset (pDrvCtrl);

    /* 卸载设备中断服务程序 */
    SYS_INT_DISCONNECT (pDrvCtrl, (FUNCPTR)templateInt, (int)pDrvCtrl, &result);
    if (result == ERROR)
    {
        DRV_LOG (DRV_DEBUG_LOAD, "Could not disconnect interrupt!\n",
            1, 2, 3, 4, 5, 6);
    }

    return (result);
}
=====

```

9.4.3.4 卸载驱动程序：xxUnload()

xxUnload()主要负责完成从系统中卸载驱动程序。其函数格式如下：

```
STATUS xxUnload (xxEND_DEVICE* pDrvCtrl);
```

其中，输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针。此函数返回值为 OK 或

ERROR。

xxUnload()执行了几个必要的操作：释放 END_OBJ 对象、释放任何共享内存。其中释放 END_OBJ 对象使用宏定义 END_OBJECT_UNLOAD，这个宏定义最终通过调用 lstFree()函数释放 END_OBJ 对象；释放共享内存的实现可能调用 cacheDmaFree()函数、cfree()函数。

下面是 xxUnload()函数的一个模板。

```
=====
templateUnload——卸载驱动程序的一个模板
=====
/*****
 *
 * templateUnload - 从系统中卸载设备驱动程序
 *
 * 这个函数首先是停止设备，然后释放驱动程序装载函数中任何分配给驱动程序的资源。
 *
 * 返回值：OK 或 ERROR。
 */

STATUS templateUnload
(
    xxEND_DEVICE* pDrvCtrl      /* 被卸载的设备驱动程序结构 */
)
{
    DRV_LOG (DRV_DEBUG_LOAD, ("Unloading end..."), 1, 2, 3, 4, 5, 6);

    /* 释放设备 */
    END_OBJECT_UNLOAD (&pDrvCtrl->end);

    /* 释放任何共享的 DMA 内存 */
    if (pDrvCtrl->flags & xx_FLG_MEM_ALLOC_FLAG)
        cacheDmaFree (pDrvCtrl->pShMem);

    ... /* 如果有必要，在这个位置增加释放所有分配的内存空间 */

    return (OK);
}
=====
```

9.4.3.5 处理 I/O 控制请求程序：xxIoctl()

xxIoctl()主要负责处理 I/O 控制请求。其函数格式如下：

```
int xxIoctl (xxEND_DEVICE* pDrvCtrl, int cmd, caddr_t data);
```

其中，输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针；cmd 是一个 I/O 控制命令代码；data 是一个命令参数。此函数返回值为 OK 或 ERROR。

xxIoctl()函数执行了几个必要的 I/O 控制操作：设置 MAC 地址、获得 MAC 地址、设置或清除标志、获得标志、开始和结束轮询操作、返回 MIB 信息、返回链的最初的最小缓冲区。具体实现过程请参考相关的函数。

下面是 xxIoctl()函数的一个模板。


```
=====
templateIoctl——处理 I/O 控制请求函数模板
=====
```

```
/*
 *
 * templateIoctl - 驱动程序 I/O 控制函数
 *
 * 处理 I/O 控制请求。
 *
 * 返回值：命令特殊响应，通常是 OK 或 ERROR。
 */
```

```
int templateIoctl
(
    xxEND_DEVICE * pDrvCtrl,    /* 接收控制命令的设备 */
    int cmd,                    /* I/O 控制命令码 */
    caddr_t data                /* 命令参数 */
)
{
    int error = 0;
    long value;

    switch (cmd)
    {
        case EIOCSADDR:        /* 设置 MAC 地址 */
            if (data == NULL)
                return (EINVAL);
            bcopy ((char *)data, (char *)END_HADDR(&pDrvCtrl->end),
                END_HADDR_LEN(&pDrvCtrl->end));
            break;

        case EIOCGADDR:        /* 获得 MAC 地址 */
            if (data == NULL)
                return (EINVAL);
            bcopy ((char *)END_HADDR(&pDrvCtrl->end), (char *)data,
                END_HADDR_LEN(&pDrvCtrl->end));
            break;

        case EIOCSFLAGS:       /* 设置(或清除)标识 */
            value = (long)data;
            if (value < 0)
            {
                value = -value;
                value--;
                END_FLAGS_CLR (&pDrvCtrl->end, value);
            }
            else
            {
                END_FLAGS_SET (&pDrvCtrl->end, value);
            }
            templateConfig (pDrvCtrl);
            break;
```



```

        case EIOCGFLAGS:      /* 获得标识 */
            *(int *)data = END_FLAGS_GET(&pDrvCtrl->end);
            break;

        case EIOCPOLLSTART:    /* 开始轮询操作 */
            templatePollStart (pDrvCtrl);
            break;

        case EIOCPOLLSTOP:     /* 结束轮询操作 */
            templatePollStop (pDrvCtrl);
            break;

        case EIOCGMIB2:        /* 返回 MIB 信息 */
            if (data == NULL)
                return (EINVAL);
            bcopy((char *)&pDrvCtrl->end.mib2Tbl, (char *)data,
                sizeof(pDrvCtrl->end.mib2Tbl));
            break;

        case EIOCGFBUF:        /* 返回链表的第一个最小缓冲区 */
            if (data == NULL)
                return (EINVAL);
            *(int *)data = TEMPLATE_MIN_FBUF;
            break;

        default:               /* 未知请求 */
            error = EINVAL;
    }

    return (error);
}

```

9.4.3.6 发送数据包程序: xxSend()

xxSend()函数主要负责发送数据包的发送。其函数格式如下:

```
STATUS xxSend (xxEND_DEVICE* pDrvCtrl, M_BLK_ID pBuf);
```

其中, 输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针; pMblk 是要发送的数据包。此函数返回值为 OK 或 ERROR、或 END_ERR_BLOCK。

xxSend()函数执行了几个必要的操作: 获得发送器的使用权、查询发送缓冲区、启动设备发送数据、统计 MIB-II 错误数和释放已经发送的数据包。其中获得发送器的使用权使用宏定义 END_TX_SEM_TAKE, 这个宏定义最终通过调用 semTake()函数获得发送权; 统计 MIB-II 错误数是使用宏 END_ERR_ADD, 这个宏定义最终通过调用 mib2ErrorAdd()函数统计 MIB-II 错误数; 释放已经发送的数据包是通过调用 netMblkClChainFree()函数来实现的。

下面是 xxSend()函数的一个模板。

```

=====
templateSend——发送数据包函数模板
=====
/*****
* templateSend - 数据包发送函数

```



```
*
* 这个程序在发送数据缓冲区 M_BLK_ID 中取出一个数据包并发送。
* 在安装驱动程序时，必须给出这个缓冲区的地址信息。当设备成功发送数据后，
* 调用释放函数释放已发送数据包的缓冲区。
*
* 返回值：OK，ERROR，或 END_ERR_BLOCK。
*/

STATUS templateSend
(
    xxEND_DEVICE * pDrvCtrl,    /* 发送设备 */
    M_BLK_ID      pMblk         /* 要发送的数据 */
)
{
    int          oldLevel;
    BOOL         freeNow = TRUE;

    /*
     * 获得惟一的发送权。这个处理是必要的，因为可能有多个任务同时对一个堆
     * 栈进行处理。
     */

    if (!(pDrvCtrl->flags & TEMPLATE_POLLING))
        END_TX_SEM_TAKE (&pDrvCtrl->end, WAIT_FOREVER);

    /* 这是一个公用的缓冲区，所有的数据在一个 M_BLK_ID 中 */

    /* 在本地结构中设置一个指针指向发送数据缓冲区 */

    /*
     * 如果没有可用的缓冲区，释放信号量并返回 END_ERR_BLOCK。
     * 但是，不要释放数据包。
     */

    /* 给与发送请求 */

    if (!(pDrvCtrl->flags & TEMPLATE_POLLING))
        oldLevel = intLock (); /* 保护中断服务程序 */

    ... /* 在这个位置添加启动设备发送代码 */

    /* 前进管理索引，释放信号量 */

    if (!(pDrvCtrl->flags & TEMPLATE_POLLING))
        END_TX_SEM_GIVE (&pDrvCtrl->end);

    if (!(pDrvCtrl->flags & TEMPLATE_POLLING))
        intUnlock (oldLevel);

    /* 对发送数据包进行统计 */

    END_ERR_ADD (&pDrvCtrl->end, MIB2_OUT_UCAST, +1);
}
```



```

/*
 * 发送缓冲区清除。当发送中断产生后，这个驱动程序必须现在释放发送数据包或
 * 装载到一个结构中以便将来可以释放。
 */

if (freeNow)
    netMblkClChainFree (pMblk);

return (OK);
}

```

9.4.3.7 添加多点传送地址程序: xxMCastAddrAdd ()

xxMCastAddrAdd()函数主要为设备添加一个指定的多点传送地址。其函数格式如下:

```
STATUS xxMCastAddrAdd(xxEND_DEVICE* pDrvCtrl, char* pAddress);
```

其中，输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针；pAddress 是一个指向用户想要添加的地址指针。此函数返回值为 OK 或 ERROR。

xxMCastAddrAdd()函数执行了几个必要的操作：添加多点传送地址、重新配置设备接口。其中添加多点传送地址是通过调用 etherMultiAdd()函数来实现。

下面是 xxMCastAdd()函数的一个模板：

```

=====
templateMCastAddrAdd——添加多点传送地址函数模板
=====
/*****
 *
 * templateMCastAddrAdd - 为设备添加一个指定的多点传送地址
 *
 * 这个函数为任何处于倾听的设备添加一个指定的多点传送地址，然后为多点传送地址重新安装过滤器。
 *
 * 返回值：OK 或 ERROR。
 */

STATUS templateMCastAddrAdd
(
    xxEND_DEVICE *pDrvCtrl,          /* 添加多点传送地址的设备 */
    char* pAddress                    /* 要添加的地址 */
)
{
    int error;

    DRV_LOG (DRV_DEBUG_IOCTL, ("MCastAddrAdd\n"), 0, 0, 0, 0, 0, 0);

    if ((error = etherMultiAdd (&pDrvCtrl->end.multiList,
        pAddress)) == ENETRESET)
        templateConfig (pDrvCtrl); /* 重新配置接口函数，为多点传送地址安装过滤器。 */

    return (OK);
}

```


9.4.3.8 删除多点传送地址程序: xxMCastAddrDel()

xxMCastAddrDel()函数主要是从设备所存储的多点传送地址表中删除一个指定的多点传送地址。其函数格式如下:

```
STATUS xxMCastAddrDel (xxEND_DEVICE* pDrvCtrl, char* pAddress);
```

其中, 输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针; pAddress 是一个指向用户想要删除的地址指针。此函数返回值为 OK 或 ERROR。

xxMCastAddrDel()函数执行了几个必要的操作: 删除多点传送地址、重新配置设备接口。其中添加多点传送地址是通过调用 etherMultiDel()函数来实现。

下面是 xxMCastAddrDel()函数的一个模板。

```
=====
templateMCastAddrDel——删除多点传送地址函数模板
=====
/*****
*
* templateMCastAddrDel - 删除多点传送地址
*
* 这个函数为任何处于倾听的设备驱动程序删除一个指定的多点传送地址, 然后为多点传送地址
* 重新安装过滤器。
*
* 返回值: OK 或 ERROR。
*/

STATUS templateMCastAddrDel
(
    xxEND_DEVICE *pDrvCtrl,      /* 删除多点传送地址的设备 */
    char* pAddress                /* 要删除的多点传送地址 */
)
{
    int error;

    if ((error = etherMultiDel (&pDrvCtrl->end.multiList,
        (char *)pAddress)) == ENETRESET)
        templateConfig (pDrvCtrl); /* 重新配置接口函数, 为多点传送地址安装过滤器 */

    return (OK);
}
=====
```

9.4.3.9 获得多点传送地址表程序: xxMCastAddrGet()

xxMCastAddrGet()函数主要是获得设备的多点传送地址表。其函数格式如下:

```
STATUS xxMCastAddrGet (xxEND_DEVICE* pDrvCtrl, MULTI_TABLE* pTable);
```

其中, 输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针; pTable 是指向地址表结构的一个指针。此函数返回值为 OK 或 ERROR。

xxMCastAddrGet()函数通过调用 etherMultiGet()函数, 获得设备多点传送地址表, 而这个表存放在 pTable 指向的地址表结构中。

下面是 xxMCastAddrGet()函数的一个模板。

```

=====
templateMCastAddrGet——获得多点传送地址表的函数模板
=====
/*****
*
* templateMCastAddrGet - 获得多点传送地址表
*
* 这个函数为任何处于倾听的驱动程序获得设备的多点传送地址。
*
* RETURNS: OK or ERROR.
*/

LOCAL STATUS templateMCastAddrGet
(
    xxEND_DEVICE *pDrvCtrl,          /* 从该设备中获得多点传送地址表 */
    MULTI_TABLE* pTable              /* 存放多点传送地址表 */
)
{
    DRV_LOG (DRV_DEBUG_IOCTL, ("templateMCastAddrGet\n"), 0, 0, 0, 0, 0, 0);

    return (etherMultiGet (&pDrvCtrl->end.multiList, pTable));
}
=====

```

9.4.3.10 轮询模式发送数据包程序: xxPollSend ()

xxPollSend()函数主要是采用轮询模式发送数据包。其函数格式如下:

STATUS xxPollSend (xxEND_DEVICE* pDrvCtrl, M_BLK_ID pMblk);

其中, 输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针; pMblk 是要发送的数据包。此函数返回值为 OK 或 EAGAIN。

xxPollSend()函数执行了几个必要的操作: 查看发送器是否忙、把发送数据放到设备发送数据包里、发送数据包和统计 MIB-II 错误数。

下面是 xxPollSend()函数的一个模板。

```

=====
templatePollSend——轮询发送数据包的函数模板
=====
/*****
*
* templatePollSend - 在轮询模式下发送数据包
*
* 用户调用这个函数尝试在轮询模式下在设备上发送数据包。
*
* 返回值: OK, 如果设备忙则为 EAGAIN.
*/

LOCAL STATUS templatePollSend
(
    xxEND_DEVICE* pDrvCtrl,          /* 轮询发送数据包的设备 */
    M_BLK_ID      pMblk              /* 要发送的数据包 */
)
{

```



```

    )
    {
        int          len;
        u_short      stat;

        DRV_LOG (DRV_DEBUG_POLL_TX, "templatePollSend\n", 1, 2, 3, 4, 5, 6);

        ... /* 在这个位置处增加设备测试代码, 以便检测发送器是否忙 */

        stat = templateStatusRead (pDrvCtrl);    /* 获得设备当前状态 */
        if ((stat & TEMPLATE_TINT) == 0)
            return ((STATUS) EAGAIN);

        ... /* 在这个位置处增加把网络缓冲区的数据导入到设备发送包里的代码 */

        len = max (ETHERSMALL, pMblk->m_len);

        ... /* 在这个位置处增加发送数据包代码 */

        /* 对发送的数据进行统计 */

        END_ERR_ADD (&pDrvCtrl->end, MIB2_OUT_UCAST, +1);

        DRV_LOG (DRV_DEBUG_POLL_TX, "leaving templatePollSend\n", 1, 2, 3, 4, 5, 6);

        return (OK);
    }
}
-----

```

9.4.3.11 轮询模式接收数据包程序: xxPollRcv()

xxPollRcv()函数主要是采用轮询模式接收数据包。其函数格式如下:

```
STATUS xxPollRcv (xxEND_DEVICE* pDrvCtrl, M_BLK_ID pMblk);
```

其中, 输入参数* pDrvCtrl 是指向驱动程序控制结构的一个指针; pMblk 是存放接收到的数据包。此函数返回值为 OK 或 EAGAIN。

xxPollRcv()函数执行了几个必要的操作: 查看是否接收到数据包、从设备缓冲区/描述符获得数据包和数据包长度、查看上层是否有存放数据包的空间、向上层提交数据包发送数据包。

下面是 xxPollRcv()函数的一个模板。

```

=====
templatePollRcv——轮询接收数据包的函数模板
=====
/*****
* templatePollRcv - 在轮询模式下接收数据
*
* 用户调用这个函数尝试在轮询模式下在设备上进行数据包接收。
*
* 返回值: OK, 当没有接收到数据包时, 返回 EAGAIN.
*/

LOCAL STATUS templatePollRcv

```



```

(
xxEND_DEVICE * pDrvCtrl,    /* 轮询接收数据包的设备 */
M_BLK_ID      pMblk         /* 接收缓冲区 */
)
{
u_short stat;
char* pPacket;
int len;

DRV_LOG (DRV_DEBUG_POLL_RX, "templatePollRcv\n", 1, 2, 3, 4, 5, 6);

stat = templateStatusRead (pDrvCtrl);    /* 获得设备当前状态 */

... /* 在这个位置增加如果没有接收到数据包立即返回的代码 */

if (!(stat & TEMPLATE_RINT))
{
DRV_LOG (DRV_DEBUG_POLL_RX, "templatePollRcv no data\n", 1,
2, 3, 4, 5, 6);
return (EAGAIN);
}

/* 从设备缓冲区/描述符中获得数据包及包长度 */
pPacket = NULL;          /* 虚构的代码 */
len = 64;                /* 虚构的代码 */

/* 上层必须提供一个可用缓冲区 */
if ((pMblk->mBlkHdr.mLen < len) || (!(pMblk->mBlkHdr.mFlags & M_EXT)))
{
DRV_LOG (DRV_DEBUG_POLL_RX, "PRX bad mblk\n", 1, 2, 3, 4, 5, 6);
return (EAGAIN);
}

... /* 在这个位置增加清除任何可能设置的状态位的代码 */

... /* 在这个位置增加包和设备错误检测代码 */

END_ERR_ADD (&pDrvCtrl->end, MIB2_IN_UCAST, +1);

... /* 在这个位置增加把数据包导入到网络缓冲区的代码 */

bcopy (pPacket, pMblk->m_data, len);
pMblk->mBlkHdr.mFlags |= M_PKTHDR;    /* 设置包头 */
pMblk->mBlkHdr.mLen = len;            /* 设置数据长度 */
pMblk->mBlkPktHdr.len = len;          /* 设置数据总长度 */

... /* 在这个位置增加清除已经上传的数据包缓冲区的代码 */

DRV_LOG (DRV_DEBUG_POLL_RX, "templatePollRcv OK\n", 1, 2, 3, 4, 5, 6);

return (OK);
}

```


9.4.3.12 组合以太网地址到发送数据包中: endEtherAddressForm()

endEtherAddressForm()函数主要是组合以太网地址到发送数据包中。其函数原型如下:

```
M_BLK_ID endEtherAddressForm
(
    M_BLK_ID pMblk,
    M_BLK_ID pSrcAddr,
    M_BLK_ID pDstAddr
)
```

其中, 输入参数 pMblk 是指向数据包 mBlk 的一个指针; pSrcAddr 是指向数据包源地址的一个指针; pDstAddr 是指向数据包目的地址的一个指针。此函数返回值为 M_BLK_ID 或 NULL。

endEtherAddressForm()函数执行了几个必要的操作: 预先分配以太网地址空间、组合以太网地址到发送数据包中。

下面是 endEtherAddressForm()函数的源代码。

```

/*****
 *
 * endEtherAddressForm - 组合一个以太网地址到发送数据包中
 *
 * 这个函数的输入参数<pSrcAddr>和<pDstAddr>分别是源地址信息和目的地址信息,
 * 并返回一个组合地址信息的 mBlk 数据包 'M_BLK_ID'。
 *
 * 返回值: M_BLK_ID 或 NULL。
 */

M_BLK_ID endEtherAddressForm
(
    M_BLK_ID pMblk,      /* 一个指向 mBlk 数据包的指针 */
    M_BLK_ID pSrcAddr,   /* 一个指向源地址的指针 */
    M_BLK_ID pDstAddr    /* 一个指向目的地址的指针 */
)
{
    M_PREPEND(pMblk, ENET_HDR_REAL_SIZ, M_DONTWAIT);

    if (pMblk != NULL)
    {
        bcopy (pDstAddr->m_data, pMblk->m_data, 6);
        bcopy (pSrcAddr->m_data, pMblk->m_data + 6, 6);
        ((ENET_HDR *)pMblk->m_data)->type = pDstAddr->mBlkHdr.reserved;
    }
    return (pMblk);
}
-----

```

9.4.3.13 从数据包里获得连接级首部信息: endEtherPacketDataGet()

endEtherPacketDataGet() 函数主要是从数据包里获得连接级首部信息, 并存放在一个 LL_HDR_INFO 数据结构中。其函数原型如下:

```
STATUS endEtherPacketDataGet
(
```



```

M_BLK_ID      pMblk,
LL_HDR_INFO * pLinkHdrInfo
)

```

其中，输入参数 pMblk 是指向数据包的一个指针；pLinkHdrInfo 是指向存放连接级首部信息空间的一个指针。此函数返回值为 OK 或 ERROR。

endEtherPacketDataGet()函数执行了几个必要的操作：确认第一个 M_BLK 数据包中完整的以太网首部、依据 802.3 进行寻址处理。

下面是 endEtherPacketDataGet()函数的源代码。

```

/*****
 *
 * endEtherPacketDataGet - 返回数据包的首部信息
 *
 * 这个函数获得数据包 pMblk 的首部信息，并存放到 pLinkHdrInfo 中。
 *
 * 返回值：OK 或 ERROR。
 */

STATUS endEtherPacketDataGet
(
    M_BLK_ID      pMblk,
    LL_HDR_INFO * pLinkHdrInfo
)
{
    ENET_HDR      *pEnetHdr;
    ENET_HDR      enetHdr;

    struct llc     *pLLCHdr;
    struct llc     llcHdr;

    USHORT        etherType;

    pLinkHdrInfo->destAddrOffset = 0;
    pLinkHdrInfo->destSize       = 6;
    pLinkHdrInfo->srcAddrOffset  = 6;
    pLinkHdrInfo->srcSize        = 6;

    /*
     * 确认整个 ether_header 在第一个 M_BLK,
     * 如果不在则拷贝这个数据包到一个临时缓冲区。
     */

    if (pMblk->mBlkHdr.mLen < ENET_HDR_REAL_SIZ)
    {
        pEnetHdr = &enetHdr;
        if (netMblkOffsetToBufCopy (pMblk, 0, (char *) pEnetHdr,
                                    ENET_HDR_REAL_SIZ, (FUNCPTR) bcopy)
            < ENET_HDR_REAL_SIZ)
        {
            return (ERROR);
        }
    }
}

```



```
    }
}
else
pEnetHdr = (ENET_HDR *)pMblk->mBlkHdr.mData;

etherType = ntohs(pEnetHdr->type);

/* 处理 802.3 寻址 */

/* 这是一个运算法则。 */
/* 如果 etherType 小于 MTU, 则要识别它。 */
/* 这是来自于 RFC 1700 的 802.x 地址。 */
if (etherType < ETHERMTU)
{
/*
* 确认整个 ether_header + llc_hdr 是在第一个 M_BLK 中,
* 如果不在则拷贝这个数据包到一个临时缓冲区。
*/

if (pMblk->mBlkHdr.mLen < ENET_HDR_REAL_SIZ + LLC_SNAP_FRAMELEN)
{
pLLCHdr = &llcHdr;
if (netMblkOffsetToBufCopy (pMblk, ENET_HDR_REAL_SIZ,
(char *) pLLCHdr, LLC_SNAP_FRAMELEN,
(FUNCPTR) bcopy)
< LLC_SNAP_FRAMELEN)
{
return (ERROR);
}
}
else
pLLCHdr = (struct llc *)((char *)pEnetHdr + ENET_HDR_REAL_SIZ);

/* 现在可能是越过 802.x 到达 IP, 所以检测是否目的地址 SAP 是 IP。 */

/* 如果没有到达 IP, 则使用 DSAP 作为 etherType */

if (pLLCHdr->llc_dsap == LLC_SNAP_LSAP)
{
etherType = ntohs(pLLCHdr->llc_un.type_snap.ether_type);
pLinkHdrInfo->dataOffset = ENET_HDR_REAL_SIZ + 8;
}
else
{ /* 非 SNAP 头 */
pLinkHdrInfo->dataOffset = ENET_HDR_REAL_SIZ + 3;
etherType = pLLCHdr->llc_dsap;
}
}
else
{
pLinkHdrInfo->dataOffset = ENET_HDR_REAL_SIZ;
}
```



```

    pLinkHdrInfo->pktType      = etherType;

    return (OK);
}
-----

```

9.4.3.14 从数据包里获得地址信息: endEtherPacketAddrGet ()

endEtherPacketAddrGet()函数主要是从数据包里获得地址信息,并存放在一个 LL_HDR_INFO 数据结构中。其函数原型如下:

```

STATUS endEtherPacketAddrGet
(
    M_BLK_ID pMblk,
    M_BLK_ID pSrc,
    M_BLK_ID pDst,
    M_BLK_ID pESrc,
    M_BLK_ID pEDst
)

```

其中,输入参数 pMblk 是指向 mBlk 结构的一个指针,从此结构中获得地址信息; pSrc 是指向 mBlk 结构的一个指针或为 NULL,获得的数据包的本地源地址将存入该结构中; pDst 是指向 mBlk 结构的一个指针或为 NULL,获得的数据包的本地目的地址将存入该结构中; pESrc 是指向 mBlk 结构的一个指针或为 NULL,获得的数据包的末端源地址将存入该结构中; pEDst 是指向 mBlk 结构的一个指针或为 NULL,获得的数据包的末端目的地址将存入该结构中。此函数返回值为 OK 或 ERROR。

endEtherPacketAddrGet()函数执行了几个必要的操作:获得数据包的本地源地址、获得数据包的本地目的地址、获得数据包的末端源地址、获得数据包的末端目的地址。

下面是 endEtherPacketAddrGet()函数的源代码。

```

/*****
 *
 * endEtherPacketAddrGet - 在数据包中查找地址信息
 *
 * 这个函数取出一个数据包'M_BLK_ID',查找地址信息,并调整 M_BLK_ID 的结构。
 * 地址信息并没有拷贝,所有'mBlk'结构共享同一个簇。
 *
 * 返回值: OK 或 ERROR.
 */

STATUS endEtherPacketAddrGet
(
    M_BLK_ID pMblk,      /* 指向数据包的一个指针 */
    M_BLK_ID pSrc,       /* 指向本地源地址的一个指针 */
    M_BLK_ID pDst,       /* 指向本地目的地址的一个指针 */
    M_BLK_ID pESrc,      /* 指向远程源地址的一个指针 */
    M_BLK_ID pEDst       /* 指向远程目的地址的一个指针 */
)
{
    if (pSrc != NULL)

```



```

    {
        pSrc = netMblkDup (pMblk, pSrc);
        pSrc->mBlkHdr.mData += 6;
        pSrc->mBlkHdr.mLen = 6;
    }
    if (pDst != NULL)
    {
        pDst = netMblkDup (pMblk, pDst);
        pDst->mBlkHdr.mLen = 6;
    }
    if (pESrc != NULL)
    {
        pESrc = netMblkDup (pMblk, pESrc);
        pESrc->mBlkHdr.mData += 6;
        pESrc->mBlkHdr.mLen = 6;
    }
    if (pEDst != NULL)
    {
        pEDst = netMblkDup (pMblk, pEDst);
        pEDst->mBlkHdr.mLen = 6;
    }

    return (OK);
}

```

9.4.3.15 其他辅助函数

在前面描述的设备驱动程序中，调用了一些辅助函数，例如轮询模式的启动和停止、设备复位操作等，这些辅助函数结构及处理描述如下。

1. 启动轮询模式: xxPollStart()

下面是 xxPollStart 函数的一个模板。

```

/*****
 *
 * templatePollStart - 启动轮询模式操作
 *
 * 返回值: OK 或 ERROR.
 */

LOCAL STATUS templatePollStart
(
    xxEND_DEVICE * pDrvCtrl          /* 轮询设备 */
)
{
    int      oldLevel;
    oldLevel = intLock ();           /* 关闭中断 */

    ... /* 在这个位置增加关闭中断的代码 */

    pDrvCtrl->flags |= TEMPLATE_POLLING;

    intUnlock (oldLevel);
}

```



```

    DRV_LOG (DRV_DEBUG_POLL, "STARTED\n", 1, 2, 3, 4, 5, 6);

    templateConfig (pDrvCtrl); /* 重新配置设备 */

    return (OK);
}
-----

```

2. 停止轮询模式: xxPollStop()

下面是 xxPollStop 函数的一个模板。

```

/*****
 *
 * templatePollStop - 停止轮询模式操作
 *
 * 这个函数终止轮询模式操作。设备返回到中断模式下。
 *
 * 设备中断打开，当前模式标志切换到中断模式，并且设备重新装配中断操作。
 *
 * 返回值: OK 或 ERROR。
 */

LOCAL STATUS templatePollStop
(
    xxEND_DEVICE * pDrvCtrl          /* 轮询操作的设备 */
)
{
    int          oldLevel;

    oldLevel = intLock (); /* 在寄存器刷新期间关闭中断 */

    ... /* 在这个位置增加打开中断代码 */

    pDrvCtrl->flags &= ~TEMPLATE_POLLING;

    intUnlock (oldLevel);

    templateConfig (pDrvCtrl);

    DRV_LOG (DRV_DEBUG_POLL, "STOPPED\n", 1, 2, 3, 4, 5, 6);

    return (OK);
}
-----

```

3. 复位设备: xxReset()

下面是 xxReset 函数的一个模板。

```

/*****
 *
 * templateReset - 复位设备
 *
 * 返回值: N/A。
 */

```



```

LOCAL void templateReset
(
    xxEND_DEVICE *pDrvCtrl
)
{
    ... /* 在这个位置增加复位设备控制器的代码 */
}

```

4. 获得当前设备状态: xxStatusRead()

下面是 xxStatusRead 函数的一个模板。

```

/*****
 *
 * templateStatusRead - 获得设备当前状态。
 *
 * 返回值: 状态位。
 */

LOCAL UINT templateStatusRead
(
    xxEND_DEVICE *pDrvCtrl
)
{
    ... /* 在这个位置增加读并返回状态位/寄存器值的代码 */
    return (0);
}

```

5. 重新配置设备接口: xxConfig()

下面是 xxConfig 函数的一个模板。

```

/*****
 *
 * templateConfig - 重新配置设备接口。
 *
 * 重新配置设备接口, 安装不加选择的模式, 并替换设备多点传送地址表。
 *
 * 返回值: N/A。
 */

LOCAL void templateConfig
(
    xxEND_DEVICE *pDrvCtrl /* 重新配置的设备 */
)
{
    /* 如果没有要求, 安装不加选择的模式 */

    if (END_FLAGS_GET(&pDrvCtrl->end) & IFF_PROMISC)
    {
        DRV_LOG (DRV_DEBUG_IOCTL, "Setting promiscuous mode on!\n",
            1, 2, 3, 4, 5, 6);
    }
}

```



```
else
{
DRV_LOG (DRV_DEBUG_IOCTL, "Setting promiscuous mode off!\n",
1, 2, 3, 4, 5, 6);
}

/* 为多点传送地址安装过滤器 */

if (END_MULTI_LST_CNT(&pDrvCtrl->end) > 0)
{
templateAddrFilterSet (pDrvCtrl);
}

... /* 在这个位置增加彻底关闭设备的代码 */

... /* 在这个位置增加复位设备所有计数器/指示器等代码 */

... /* 在这个位置增加根据标识初始化设备硬件 */

return;
}
```

小 结

编写网络接口驱动程序是一件比较复杂的工作。本章首先从宏观上描述网络系统的数据交换及网络接口驱动程序，然后描述了 VxWorks 下 END 设备驱动程序的装载过程、设备与系统的数据交换过程，最后详细给出了编写 END 网络设备驱动程序的步骤，并给出了驱动程序的一些模板函数。

第 10 章 接触实际设备

这一章将讨论一些实际设备驱动程序的编写，虽然这里不会完整地介绍设备驱动程序的编写过程，这里还是会涉及到很多实际编写驱动中需要考虑的问题。这一章将分为三部分，第一部分讨论编写 PCI 设备驱动程序所需要注意的内容；第二部分将会讨论驱动程序如何访问 ISA 设备；第三部分将简单分析 VxWorks 下原始文件系统（rawFs）的编写方法以便于用户可以更深入理解块设备驱动程序。

10.1 PCI 设备驱动程序

这一节将简单讨论 VxWorks 对 PCI 总线的支持。为了便于读者理解，首先会简要地涉及到一些 PCI 规范上的知识，然后讨论 VxWorks 支持 PCI 设备所需要注意的问题，最后分析 VxWorks 对 PCI 设备的支持。这里将以常见的 PCI 桥 PLX9052 为例，介绍一下对 PCI 桥的编程。

10.1.1 PCI 配置空间

根据 PCI 规范，每一个 PCI 设备都要有一个 PCI 配置空间，容量最大为 256 字节，称为配置寄存器。配置寄存器是 PCI 设备的硬件与 PCI 设备的初始化软件及错误处理软件之间的信息交接区，以便软件对 PCI 设备进行识别和控制以及 PCI 设备向软件反映设备状态和要求。

10.1.1.1 配置空间的组织结构

256 字节的配置空间分为头标区（Header Region）和设备关联区（Device Dependent Region）两部分，如图 10.1 所示。

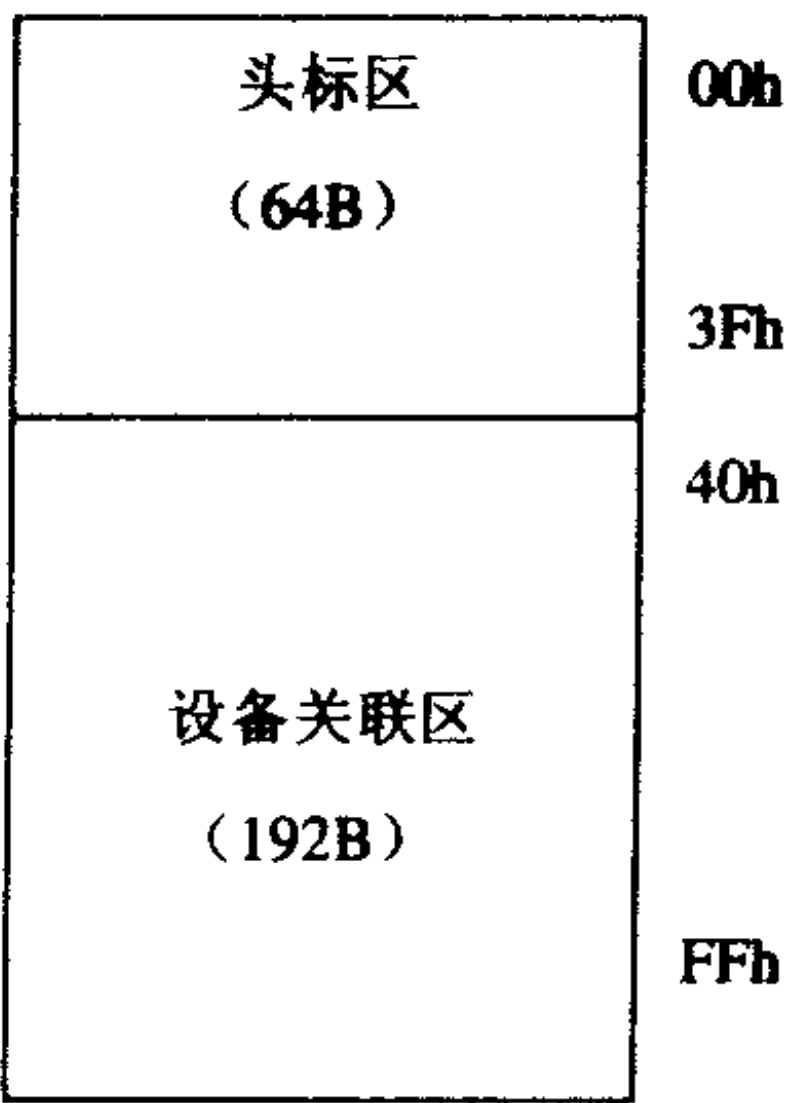


图 10.1 PCI 总线配置空间的组织结构

图 10.1 中 64 个字节的头标区是每个 PCI 设备的配置空间里都具有的区域，对所有的 PCI 设备

来说头标区中的寄存器布局和格式大致相同。而是否具有 192 字节设备关联区及区内寄存器的设置与布局则完全取决于该 PCI 设备本身的需要，一个 PCI 设备因本身的特殊功能所需设置的专用寄存器只能放在这个区域中。

配置空间里头标区寄存器的布局如图 10.2 所示。头标区又分为两部分，头 16 个字节的定义对所有类型的 PCI 设备都是相同的，余下的 48 字节根据设备的功能类别有不同的布局。位于 0Eh 的头标类型（Header Type）字段规定了后 48 字节所用的布局类型。目前规范只规定了 00h 和 01h 两个头标类型：01h 类型的布局规定用于 PCI-to-PCI 桥；00h 类型的布局用于其余 PCI 设备。

31	16		15	0
设备标识		厂商标识		00h
状态字		命令字		04h
分类代码			修订版本标识	08h
内置自测试	头标类型	等待时间定时器	Cache 行长	0Ch
基地址寄存器（6 个）				10h
				14h
				18h
				1Ch
				20h
				24h
Card bus 卡的卡信息结构指针				28h
子系统标识		子系统供应商标识		2Ch
扩充 ROM 基地址				30h
（保留）				34h
（保留）				38h
最长等待时间	最短获准时间	中断引脚	中断线	3Ch

图 10.2 PCI 总线配置空间里头标区寄存器的布局

10.1.1.2 配置空间的功能

类型 0 配置空间头标区里的寄存器实现了很多功能，这里将介绍一些与软件编程相关的几类。

1. 设备识别

在头标区内有五个寄存器是与一个 PCI 设备的识别有关的，配置软件通过访问这些寄存器很容易确定在该 PCI 总线上有什么样的 PCI 设备。这五个寄存器都是只读的。

(1) 厂商标识（Vendor ID）。

标识该设备的制造厂商，这个标识符由 PCI 规范的权威组织 SIG 统一分配，以保证惟一性，例如分配给 Intel 公司的标识码为“8086h”。“FFFFh”是一个无效的标识。

(2) 设备标识（Device ID）。

标识某一种设备，这个标识符由制造厂商分配。

(3) 修订版本标识 (Revision ID)。

标识设备具体的修订版本，由制造厂商选择。修订版本标识可以看作是对设备标识的扩展。

(4) 头标类型 (Head Type)。

表明头标区中 10h 到 3Fh 字节空间的布局类型和该设备是否是多功能设备。位 7 用来标识一个多功能设备，为 1 则该设备是一个多功能设备；为 0 该设备为单一功能设备。位 6 到 0 规定头标区中 10h 到 3Fh 字节的布局类型，如前所述，目前只有两种类型：00h 和 01h，此处讨论的是 00h 类型。

(5) 分类代码 (Class Code)

该寄存器分为三个字节段：高字节 (0Bh) 是基本分类码，粗略地对设备的功能进行分类；中字节 (0Ah) 是一个子分类码，标识设备更具体的功能；低字节 (09h) 标识所用的、寄存器一级的编程接口。

2. 设备控制

位于配置空间 04h 偏移地址处的命令寄存器是控制一个 PCI 设备产生和响应 PCI 周期的能力。命令寄存器是一个读/写寄存器，各位的定义和作用如下（这里只描述软件常用的位）。

(1) I/O 空间使能位。

控制设备对 I/O 空间访问的响应，为 1 时，允许该设备对 I/O 空间的访问进行响应，为 0 时禁止。如果该设备内有属于 PCI I/O 空间的单元并允许 PCI 主设备访问时，此位应置为 1。

(2) 存储器空间使能位。

控制本设备对存储器空间访问的响应，为 1 时允许该设备对存储器空间的访问进行响应，为 0 时禁止。如果该设备内有属于 PCI 存储空间的单元并允许 PCI 主设备访问时，此位应置为 1。

3. 基地址

在头标区中有 6 个双字的基地址寄存器，在上电时，这 6 个基地址寄存器中的内容向 POST 软件反映该设备需要多少存储器空间和 I/O 空间。POST 软件通过各设备内的基地址寄存器了解到 PCI 总线上有哪一些存储器设备和 I/O 设备，根据它们的容量需求把它们映射定位到适当的存储器地址和 I/O 地址，并把起始地址再写入基地址寄存器，然后引导系统。

(1) 基地址寄存器。

每一个基地址寄存器对应该设备里的一片存储器或 I/O 空间。由于有 6 个基地址寄存器，故一个设备可以有不止一片存储器空间或 I/O 空间。基地址寄存器的位 0 都是只读的，设备用这一位来表示这个寄存器所申请的空间是存储器空间还是 I/O 空间。

如果某一基地址寄存器的位 0 为 1，则表示该寄存器相关联的空间是要映射到 I/O 空间的，该寄存器是用于 I/O 空间映射的基地址寄存器。位 1 为保留位，位 2~31 用来把设备上的这片空间映射到系统 I/O 空间，POST 软件最后要把分配给它的系统 I/O 空间基地址再写回到该寄存器的 2~31 位。

如果某一基地址寄存器的位 0 为 0，则表示该寄存器相关联的空间是要映射到存储器空间的，该寄存器是用于存储器空间映射的基地址寄存器。该寄存器的位 1~3 也是只读的，其中位 1~2 是类型编码，表示该存储空间的映射要求，在表 10-1 中列出了这些类型编码。

表 10-1 基地址寄存器位 2、位 1 的含义

位	含 意
00	基地址寄存器为 32 位宽，该存储器可以映射于 32 位存储空间的任何地方
01	基地址寄存器为 32 位宽，该存储器必须映射于 1MB 以下的存储空间里
10	基地址寄存器为 64 位宽，该存储器可以映射于 64 位存储器空间的任何地方
11	保留

如果基地址寄存器为 64 位，那么该寄存器就占据两个双字的位置，它后面的基地址寄存器的偏移地址就要向后推一个双字，但基地址寄存器的总容量不变（保持 6 个双字的大小）。

（2）地址映射。

POST 软件对设备里的存储器和 I/O 机型地址映射是这样做的：首先读取各设备中的基地址寄存器，查明各设备所想要的空间和容量，经统一协调和分配后，向各寄存器的高位字段写入所分配空间的基地址，把寄存器所对应的存储器或 I/O 映射到系统的物理地址空间中去。这个方法分为下面两个步骤。

首先是确定空间类别和容量要求。空间类别从基地址寄存器的最低位很容易判断。基地址寄存器表达所需空间容量的办法是把寄存器中的基地址字段的若干个低位从硬件上连到地。POST 软件从 PCI 总线先向该基地址寄存器全部写 1，然后读回其值。由于硬件连到了地，在读回的值中，从 b4 位（对于存储器类型）或从 b2 位（对于 I/O 类型）开始到最高位所返回的零的个数反映了设备所需要的空间。按照这种设计方法，设备利用基地址寄存器可以申请比实际耗用更多的空间，对于需要量少于 4KB 的存储器，建议占到 4KB 以便减少地址译码器的位数。对于 I/O 空间，把控制功能映射到 I/O 空间的设备，每一个 I/O 基地址寄存器最多申请 256 字节的 I/O 空间。

其次是分配基地址。基地址寄存器的容量按上面的方法确定以后，POST 软件就可以在系统物理空间中为这片空间分配一个基地址，即起始地址，并把这个基地址写回该基地址寄存器的基地址字段。可以看出，软件可编程的高位地址就是所占容量的对界地址。

10.1.2 PCI 设备的中断复用

根据 PCI 的规范，所有 PCI 设备都将复用 4 个中断级别，也就是常说的 PCI 的 A、B、C、D 中断。想要设备都实现复用中断不但要在硬件上使用电平触发机制，还要在软件上相应处理。下面将分两个方面讨论中断复用的软件处理方式，其一是单一设备的中断复用，其二是多设备的中断复用。

软件处理单一设备的中断复用主要由中断服务程序完成。单一设备复用中断，驱动程序将挂接在指定的中断号上，这时在该中断号上所出现的中断一定是属于该类设备的。基于这样一个原因，用户可以在中断服务程序中检查所有该类型的设备，以确认中断的来源并对产生中断的设备进行服务。

另一方面，对于不同的设备复用同一中断的情况，处理上就要复杂得多。

在这种情况下，如果每一个中断服务程序都使用相同的中断向量，那么后接入的中断服务程序将会覆盖先前所使用中断向量，等挂接多个中断服务程序到相同的中断号上后，只有最后一个中断服务程序是起作用的，无法达到中断复用的目的。

基于这种原因，在实际的中断复用过程中，系统使用了另一种办法。在 VxWorks 系统内部，对


```

VOID pciInt
(
    int irq          /* PCI 中断所使用的 IRQ */
)
{
    PCI_INT_RTN *pRtn;

    for (pRtn = (PCI_INT_RTN *)DLL_FIRST (&pciIntList[irq]); pRtn != NULL;
        pRtn = (PCI_INT_RTN *)DLL_NEXT (&pRtn->node))
        (* pRtn->routine) (pRtn->parameter);
}

```

10.1.3 VxWorks 对 PCI 设备的支持

如果为系统定义了 PCI 设备支持, 在 BSP 的文件 sysLib.c 中将会有 PCI 设备支持库的相关初始化内容, 如下所示:

系统首先初始化配置支持库，随后初始化中断支持库。这两个库提供了对 PCI 设备配置空间的访问和 PCI 中断复用功能的实现。

在 PCI 配置支持库中，VxWorks 系统提供了表 10-2 所列出的函数。

表 10-2 PCI 总线配置函数

函 数	描 述
pciConfigLibInit	初始化 PCI 配置支持库
pciFindDevice	寻找以厂商标识、设备标识指定的 PCI 设备并返回对应的总线号、设备号、功能号
pciFindClass	寻找以类别指定的 PCI 设备并返回总线号、设备号、功能号
pciDevConfig	配置指定的设备
pciConfigBdfPack	返回组合为整型数的总线号、设备号、功能号
pciConfigInByte	以字节为单位读入指定 PCI 设备配置空间的配置

续表

函 数	描 述
pciConfigInWord	以字为单位读入指定 PCI 设备配置空间的配置
pciConfigInLong	以双字为单位读入指定 PCI 设备配置空间的配置
pciConfigOutByte	以字节为单位写指定 PCI 设备配置空间的配置
pciConfigOutWord	以字为单位写指定 PCI 设备配置空间的配置
pciConfigOutLong	以双字为单位写指定 PCI 设备配置空间的配置
pciConfigModifyLong	以双字为单位更新指定 PCI 设备配置空间的配置
pciConfigModifyWord	以字为单位更新指定 PCI 设备配置空间的配置
pciConfigModifyByte	以字节为单位更新指定 PCI 设备配置空间的配置
pciSpecialCycle	产生一个携带信息的特殊周期

使用这些函数可以寻找指定 PCI 设备，读取、修改指定 PCI 设备的配置空间内容。这些函数在 PCI 地址映射过程中都是非常有用的，详细的例子参见 10.1.4 节。

在 PCI 中断复用支持库中，提供了表 10-3 所列出的函数。

表 10-3 PCI 总线中断复用函数

函 数	描 述
pciIntLibInit	初始化 PCI 中断复用支持库
pciInt	PCI 中断服务程序，用于管理、调用 PCI 设备中断
pciIntConnect	挂接 PCI 中断服务程序
pciIntDisconnect	解除中断服务程序的挂接

使用这些函数可以实现 PCI 设备的中断复用功能，使用过程同样请参考 10.1.4 节的例子。

10.1.4 PCI 设备在 VxWorks 中的地址映射

VxWorks 操作系统在上电的过程中也会对 PCI 设备执行地址映射的操作，以便应用程序可以正确的访问 PCI 设备。在使用了 MMU 管理组件时，这个操作步骤是由系统自动完成的，但是如果系统只包含了基本的 MMU 功能时就必须由用户修改 BSP 来配置了。

由于只包含了基本 MMU 的 VxWorks 不会对 PCI 设备进行自动配置，用户可以有两种方法来配置 PCI 设备所占用的内存映射空间（I/O 空间不需要映射）。这两种办法分别是手动配置和半自动配置。

首先我们将简单描述一下手动配置的方法和这种方法涉及到的内存映射表。在 sysLib.c 中有一张表 sysPhysMemDesc 描述了系统物理内存的分配。如下所示：

```

PHYS_MEM_DESC sysPhysMemDesc [] =
{
    /* 地址和长度参数必须是按页对齐的（4KB/4MB 的倍数） */

```



```
/* 低端内存 */
{
    (void *) LOCAL_MEM_LOCAL_ADRS,
    (void *) LOCAL_MEM_LOCAL_ADRS,
    0xa0000,
    VM_STATE_MASK_FOR_ALL,
    VM_STATE_FOR_MEM_OS
},

/* 显示内存等 */
{
    (void *) 0xa0000,
    (void *) 0xa0000,
    0x60000,
    VM_STATE_MASK_FOR_ALL,
    VM_STATE_FOR_IO
},

/* 为操作系统保留的高端内存 */
{
    (void *) 0x100000,
    (void *) 0x100000,
    0x080000,
    VM_STATE_MASK_FOR_ALL,
    VM_STATE_FOR_MEM_OS
},

/* 为应用程序保留的高端内存 */
{
    (void *) 0x180000,
    (void *) 0x180000,
    LOCAL_MEM_SIZE - 0x180000, /* 在 sys Mem Top() 中会改变 */
    VM_STATE_MASK_FOR_ALL,
    VM_STATE_FOR_MEM_APPLICATION
},

/* 用于动态映射的条目 */

DUMMY_MMU_ENTRY,
DUMMY_MMU_ENTRY,
};
```

在这张表中描述了系统物理内存存在逻辑内存中所分配的空间。系统在引导的时候会根据这里的描述管理内存空间。

所谓的手动配置 PCI 设备的方法就是根据 PCI 基地址寄存器中的值修改这张表。但是通常是不推荐这样修改的，这样修改将会无法适应硬件环境的变化局限了 BSP 的使用。

通常使用的方法是半自动配置办法，下面我们针对 PLX9052 芯片举例分析。PLX9052 是一个 PCI 桥设备，它的作用是为本地总线提供 PCI 的桥接功能，有关 PLX9052 的详细信息请参考 PLX9052 数据手册。

所谓半自动的方法是相对于包含完整 MMU 的 VxWorks 所采用的自动化办法。自动化的配置办

法将自动发现 PCI 设备并根据内存空间的使用情况自动为 PCI 设备分配所需要的空间。但是仅包含了基本 MMU 的 VxWorks 并不会自动完成这些配置，还需要其他的一些辅助步骤帮助内存映射。

首先是为所有设备分配不冲突的内存地址。

其次是编写程序使系统可以找到指定的 PCI 设备并为其分配空间。下面是这一过程的例子，注意这是个通用的简化例子，在改写的时候应根据实际情况替换厂商标识、设备标识等。

```

=====
PCI 总线设备初始化
=====
void xxxPciInit (void)
{
    xxx_RESOURCE *pReso;
    int pciBus;
    int pciDevice;
    int pciFunc;
    int unit;
    int iCommand;
    BOOL duplicate;
    UINT32 membaseCsr;
    UINT32 iobaseCsr;
    UINT32 membaseFlash;
    char irq;
    int ix;

    for (unit = 0; unit < xxx_MAX_UNITS; unit++)
    {
        if (pciFindDevice (0x10B5, 0x9050, unit,
                           &pciBus, &pciDevice, &pciFunc) != OK)
            break;

        /* 检查是否重复 */

        pReso = &xxxResources [0];
        duplicate = FALSE;

        for (ix = 0; ix < xxx_MAX_UNITS; ix++, pReso++)
        {
            if ((ix != unit) && (pReso->pciBus == pciBus) &&
                (pReso->pciDevice == pciDevice) && (pReso->pciFunc == pciFunc))
                duplicate = TRUE;
        }

        if (duplicate) continue;

        /* 找到了正确的设备 */

        pReso = &xxxResources [unit];
        pReso->pciBus = pciBus;
        pReso->pciDevice = pciDevice;
        pReso->pciFunc = pciFunc;
    }
}

```



```

/* PCI_CFG_TYPE 可以定义为 PCI_CFG_NONE, PC_CFG_AUTO, or PCI_CFG_FORCE. */
/* BIOS 可以自动配置内存基地址和 I/O 基地址, 这里不需要做任何事情, 但也可以根据需要进行配置 */
if (PCI_CFG_TYPE == PCI_CFG_FORCE) /*PCI_CFG_TYPE 在 pc.h 中定义为 PCI_CFG_NONE*/
{ /* 设置 PCI 配置空间*/
    pciConfigOutLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
        PCI_CFG_BASE_ADDRESS_0, pReso->membaseCsr);
    pciConfigOutLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
        PCI_CFG_BASE_ADDRESS_2, pReso->membaseFlash);
    pciConfigOutByte (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
        PCI_CFG_DEV_INT_LINE, pReso->irq);
    pciConfigInLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
        PCI_CFG_COMMAND, &iCommand);
    pciConfigOutLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
        PCI_CFG_COMMAND, iCommand | CSR_MEM_EN | CSR_BM_EN );
}
pciConfigInLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_BASE_ADDRESS_0, &membaseCsr); /*读内存基地址 1 寄存器*/
pciConfigInLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_BASE_ADDRESS_2, &membaseFlash); /*读内存基地址 2 寄存器*/

pciConfigInLong (pReso->pciBus, pReso->pciDevice,
    pReso->pciFunc, PCI_CFG_COMMAND, &iCommand);
pciConfigOutLong (pReso->pciBus, pReso->pciDevice,
    pReso->pciFunc, PCI_CFG_COMMAND,
    iCommand | CSR_MEM_EN | CSR_BM_EN );

xPLxAddress[unit]=membaseCsr;
xxxxAddress[unit]=membaseFlash;

/*检查所需的内存地址和 I/O 地址范围 */
pciConfigOutLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_BASE_ADDRESS_0, 0xffffffff);
pciConfigOutLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_BASE_ADDRESS_2, 0xffffffff);

pciConfigInLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_BASE_ADDRESS_0, &r_membaseCsr[unit]);
pciConfigInLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_BASE_ADDRESS_2, &r_membaseFlash[unit]);

pciConfigOutLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_BASE_ADDRESS_0, xPLxAddress[unit]);
pciConfigOutLong (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_BASE_ADDRESS_2, xxxxAddress[unit]);

pciConfigInByte (pReso->pciBus, pReso->pciDevice, pReso->pciFunc,
    PCI_CFG_DEV_INT_LINE, &irq); /*从寄存器中获取中断线 */
if ((pciIntConnect (INUM_TO_IVEC (((int) irq) + INT_NUM_IRQ0),
    IntFunc, (int) ix)) == ERROR)
{
    logMsg ("Failed to connect interrupt\n",
        0, 0, 0, 0, 0, 0);
}

```



```

return (ERROR);
    }
sysIntDisablePIC((int)irq) ;

    membaseCsr    &= PCI_MEMBASE_MASK;
    iobaseCsr     &= PCI_IOBASE_MASK;
    membaseFlash &= PCI_MEMBASE_MASK;

    /* 将内存映射加入到 sysPhysMemDesc[] 中*/

    if (sysMmuMapAdd((void *)membaseCsr, pReso->memLength,
                    pReso->initialStateMask,
                    pReso->initialState) == ERROR)
        break;

    if (sysMmuMapAdd((void *)membaseFlash, pReso->memLength,
                    pReso->initialStateMask,
                    pReso->initialState) == ERROR)
        break;

    /* 保存相关内容 */

    pReso->membaseCsr    = membaseCsr;
    pReso->iobaseCsr     = 0xe000; /* iobaseCsr; */
    pReso->membaseFlash = membaseFlash;
    pReso->irq           = irq;

    xxxUnits++;
}
}

```

10.2 ISA 设备驱动程序

相对于 PCI 设备，编写 ISA 设备驱动程序就要简单得多。编写 ISA 设备驱动程序不需要太多的硬件知识，而且在操作上也不用考虑过多的硬件配置。在这一节里将首先讨论一下 ISA 设备的硬件特征，随后简要描述程序对 ISA 设备的操作，最后将着重讨论为复用中断的 ISA 设备编写中断服务程序。

10.2.1 ISA 设备

ISA 总线在 PC 的发展过程中是比较成功的一种总线结构，特别适合于控制外设和数据通信。在目前的 PC 中，ISA 总线已经很少用到，但是在工业控制上还在很大范围中应用着 ISA 总线。每个 ISA 设备都会占用指定的内存空间、端口号和中断号。对于相同的设备，可以复用相同的中断号或相同的内存空间。这些都是由硬件决定的。

10.2.2 访问 ISA 设备

ISA 设备所占用的空间和中断号全部是由硬件决定的。在这方面，软件只能根据硬件的相关说明来操作。访问 ISA 设备包括了端口访问和内存访问两方面。

访问 ISA 设备的端口就如同访问其他设备的端口。对于 VxWorks 来说可以使用与架构相关的 `sysInByte()` 和 `sysOutByte()` 函数。VxWorks 在 BSP 的 `sysALib.s` 中实现了相关的函数。

访问 ISA 设备的内存空间与访问通常所指的内存空间没有差别，但是一定要确认已经正确配置的硬件。

在 ISA 类型硬件设计的过程中已经决定了该设备所使用的内存空间、I/O 空间和中断号，但是在普通的 x86 架构下，系统为 ISA 设备分配的内存空间是有限的，整个系统的终端号数目也是有限的，如果一个系统中有很多 ISA 设备，就必须使这些资源可以复用。这样的资源复用首先要在硬件上支持，这里不深入讨论。下面是中断复用后，设备事件触发与软件处理的流程：

首先，某个设备产生了中断，系统响应中断进入中断服务程序。这是软件开始对每一个支持的设备进行查询以找到产生中断的设备。当找到产生中断的设备后，中断服务程序对其进行处理。

中断服务程序会一直重复这个行为直到处理完每一个可能产生中断的设备。由于 ISA 总线通常使用在工业控制中，所以中断复用对于使用多个相同类型的板卡是很有意义的。

10.2.3 ISA 设备的中断复用

前面曾经提到 ISA 设备的中断是可以复用的，这首先要在硬件上支持电平触发的模式，同时软件上也要做相应的处理。这里不讨论硬件上的实现，仅仅讨论一下软件的实现。

在软件上，中断服务程序应该支持设备的中断复用。在中断到来时，相应的中断服务程序应该查询每一个可能产生中断的设备，以确定中断的来源。完成处理后，中断服务程序应将相应设备的中断清除，等待下一个触发。下面的例子实现了上述的情况：

```
void IntHandle(At96TX_Dev *pAt96TXDEV)
{
    int BoardNum=0;
    int ChannelNum=0;
    UINT8 *NUM8;
    int BaseAddr;
    BOOL IntIsOver=TRUE;
    unsigned char IntState;
    UINT8 Flag;

    for(BoardNum=0; BoardNum<At96TXUnitNum; BoardNum++)
    {
        /*清除中断位*/
        IntState=sysInByte(FirstBasePort+BoardNum*0x10);
        IntState|=0x01;
        sysOutByte (FirstBasePort +BoardNum*0x10,IntState);
        sysOutByte (FirstBasePort +BoardNum*0x10,(IntState&0xFE));
        /*-----*/
        sysOutByte ( FirstBasePort+BoardNum*0x10+0x03,0xd0+4*BoardNum);
        sysOutByte ( FirstBasePort+BoardNum*0x10+0x04,4*0);
        /*访问中断标志位*/
        BaseAddr=BaseAddr0+BoardNum*0x4000;
        NUM8=(UINT8 *) (BaseAddr+OffInterruptF);
        Flag=*NUM8;
```



```

*NUM8=0x00;
.../*相应的处理*/
}
}

```

这个中断处理程序中，既有设备端口的访问又有设备内存空间的访问。同时在循环的过程中，中断服务程序查询了每一块设备的状态。

10.3 原始文件系统分析

虽然前面描述过块设备驱动程序的编写，同时也简要说明了文件系统的工作情况，在这里还是要讨论一下原始文件系统的实现原理（真正的过程还是有区别的），使读者可以对文件系统的工作流程及其与块设备的信息交换有更好的理解。

10.3.1 原始文件系统与系统的关系

对于 VxWorks 系统来说，文件系统是以设备的形式存在的。它是 VxWorks I/O 系统与块设备驱动程序的接口。三者的关系如图 10.3 所示。

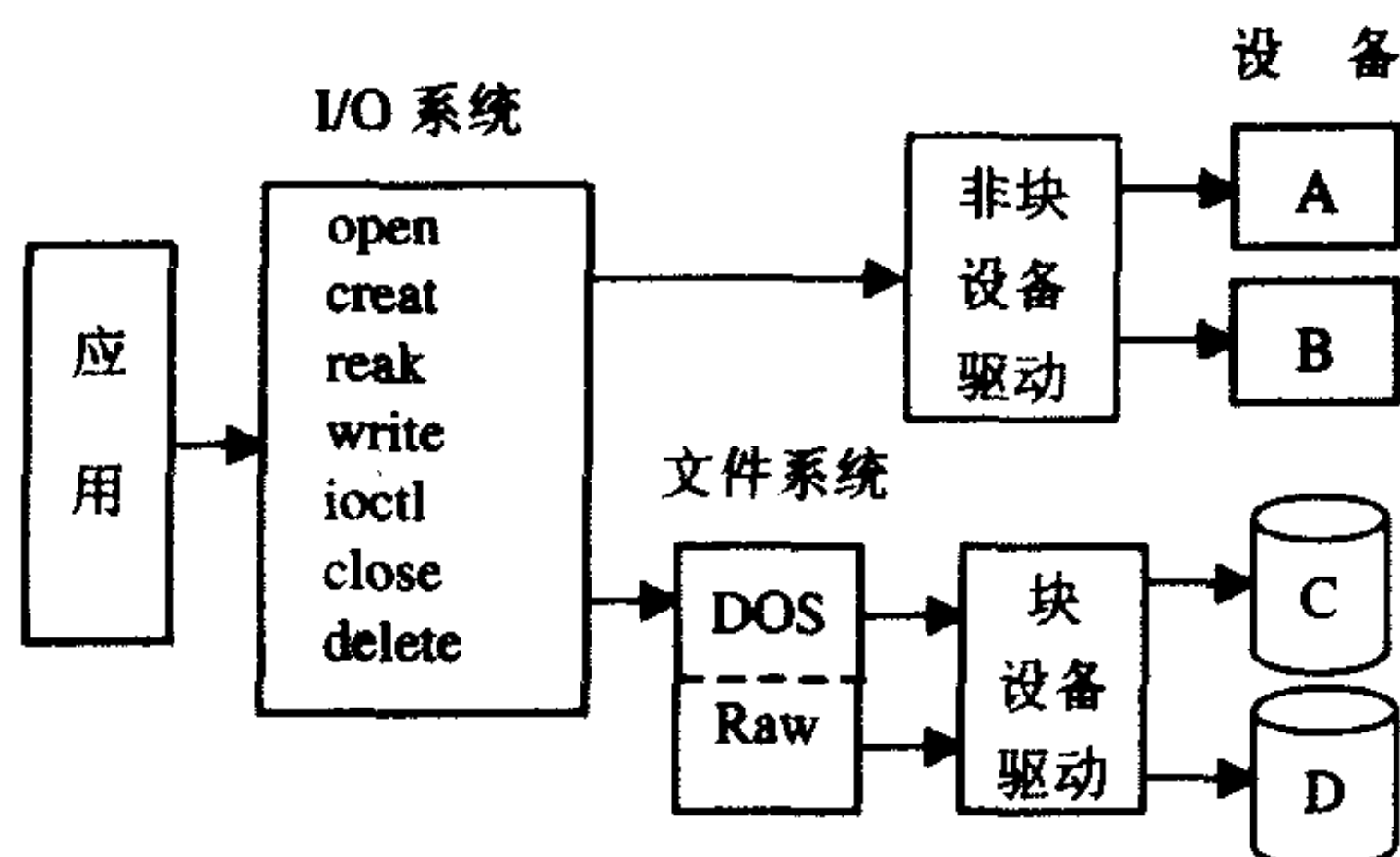


图 10.3 VxWorks I/O 系统、文件系统及块设备驱动程序之间的关系

第 8 章曾经提到，文件系统在使用之前必须初始化。文件系统初始化的过程不仅是初始化自身数据结构，还要执行下面的操作：

创建一个文件系统驱动程序条目；

创建一个指定设备名的设备；

将块设备的驱动函数挂接到相应的数据结构中。

以原始文件系统为例，系统首先会执行 `rawFsInit()` 操作，在该操作中，文件系统会将其自身初始化、设置适当的变量和参数并在 I/O 系统的驱动程序表中创建一个指向自身驱动程序的条目。

初始化块设备驱动程序之后，用户使用 `rawFsDevInit()` 函数将设备与原始文件系统相关联。在执行这一步的操作中，原始文件系统会使用传入的设备名创建一个指向自身驱动程序的设备。

用户在访问块设备的时候，I/O 系统将会首先访问文件系统，而后通过文件系统访问块设备驱动程序。以 `read()` 操作为例，过程如图 10.4 所示。

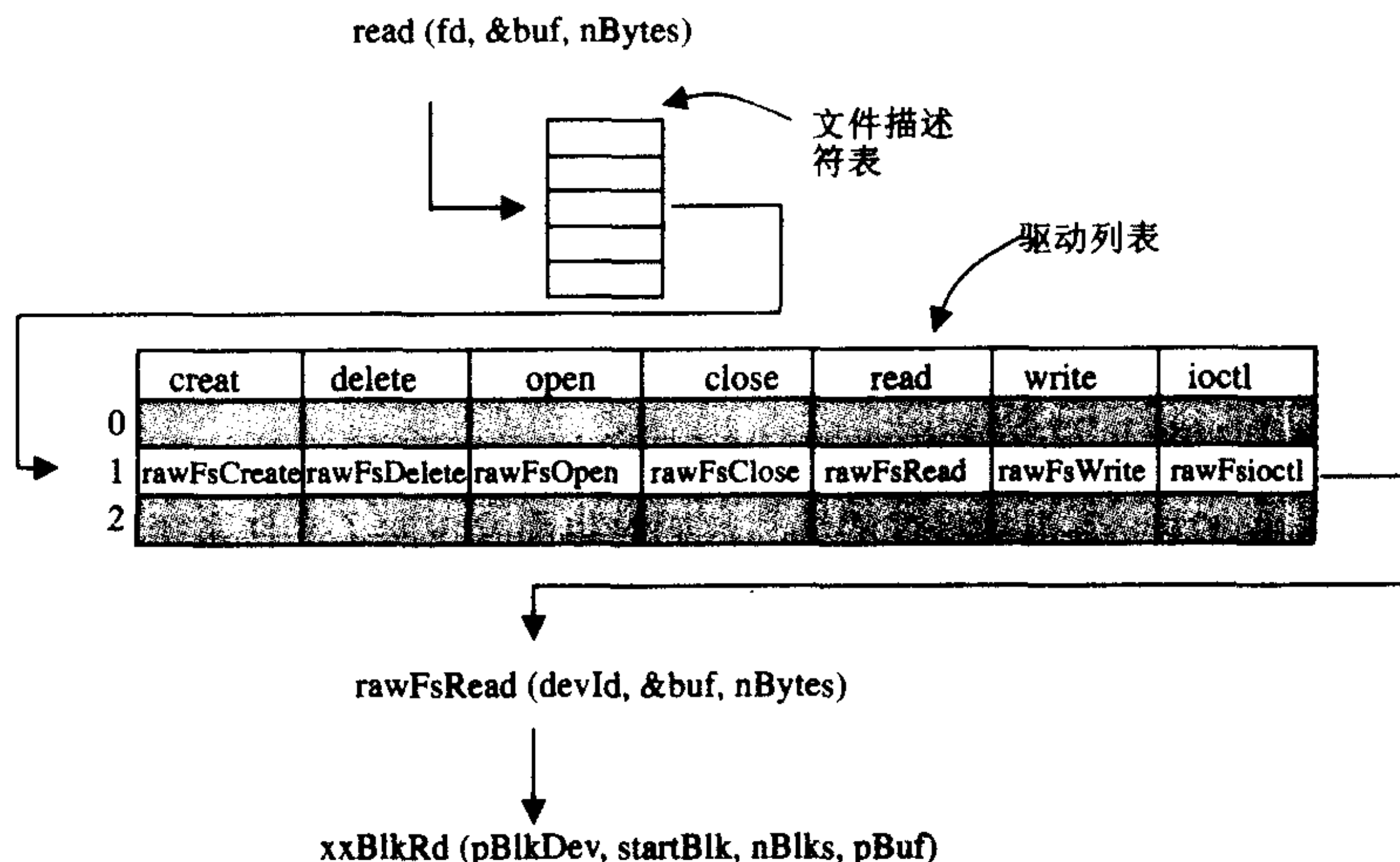


图 10.4 原始文件系统读操作流程

当用户执行 read() 操作时，I/O 系统会将用户的 I/O 请求包发送给文件系统的相关函数，然后由文件系统访问驱动程序的操作函数完成一次用户请求。

当用户执行 creat()、read()、write()、remove() 或者 ioctl() 操作时，文件系统的 I/O 服务函数不但会执行一些特定的操作，而且会调用块设备的处理函数对硬件操作（raw 文件系统没有创建和删除操作）。

用户使用完文件后，会调用 close() 发送 I/O 请求，I/O 系统将会调用文件系统的关闭函数执行操作，此时也不会访问硬件。

10.3.2 原始文件系统与块设备的通信

由上面的讨论可以看出，在执行创建、读取数据、写入数据、删除文件及特殊控制等操作（raw 文件系统没有创建和删除操作）时，文件系统 I/O 服务函数将会和块设备驱动程序打交道。在这些访问中，文件系统实际所作的操作分为两步，首先根据文件的组织结构确定需要操作的起始磁盘扇区、磁盘扇区数，然后使驱动程序根据计算出来的磁盘扇区和磁盘数操作硬件。

由于原始文件系统对块设备不采用任何组织结构，而是将其看成一个单一的文件，起始扇区和磁盘数的操作计算起来就非常容易。首先用户在操作时会指定文件的起始位置。根据这个起始位置，原始文件系统将会计算出文件的起始扇区。访问设备时，用户又会指定文件操作的偏移量。根据这个偏移量，原始文件系统就可以算出起始扇区了，如下所示：

```
rawStartBlk = fileStartBlk + fileOffset;
```

原始文件系统会将该起始扇区传给驱动程序完成操作。

经过上面的讨论，读者可能已经明白了原始文件的工作原理。对于其他的文件系统来说也是一样的，只是有些文件系统复杂一些，而原始文件系统简单一些。这些复杂的文件系统在访问硬件的时候可能要多次访问硬件设备或者需要计算出文件所分布的位置等信息。但是在实际的操作原理上，

都是大同小异的。就拿 dosFs 来说，在一次读操作上，首先需要计算出文件所在的起始扇区，然后根据文件分配表找到文件的相应位置，最后计算出扇区的起始地址读取数据，在读取的过程中可能还要多次计算数据所在的其他扇区位置。但是，最终文件系统都会计算出起始地址并将其传递给设备驱动程序完成硬件操作。

小 结

本章对实际的设备做了比较浅显的讨论，本章的三个部分分别面向了常用的设备接口和文件系统的工作原理，虽然只是点到为止，却也包括了其中的精髓。希望这些经验之谈可以对读者的开发有帮助。

第 11 章 板级支持包

从本章起,将讨论 VxWorks 的一个重要组成部分——板级支持包(BSP, Board Support Package)。在目标机系统加电后,首先执行的代码就是板级支持包。由此可以想像在开发过程中 BSP 在整个系统软件中的重要性,如果没有正确的 BSP 软件,整个系统的启动和运行就不会正常工作。另外需要强调的是: BSP 的概念只是针对嵌入式操作系统而言的,而像 DOS、Windows、UNIX 等依赖 BIOS 引导的操作系统是无 BSP 可言的。对同一目标机而言,不同操作系统的 BSP 在本质上是一样的,但是不同的操作系统将提供不同的支持库,另外在 BSP 结构上也可能会有所不同。

11.1 板级支持包

11.1.1 板级支持包简述

所谓 BSP 通常是指针对具体的硬件平台,用户所编写的启动代码和部分设备驱动程序的集合。它所实现的功能包括初始化、驱动部分设备。最基本的 BSP 仅需要支持处理机复位、初始化、驱动串口和必要的时钟处理。不过到目前为止还没有对 BSP 的明确定义。例如在 pSOS 操作系统中,对 BSP 的描述是系统用来管理外部设备的部分,其主要由两部分组成:初始化代码、硬件驱动程序。

在 VxWorks 系统中,对 BSP 的简单描述为介于底层硬件环境和 VxWorks 之间的一个软件接口,它的主要功能是系统加电后初始化目标机硬件、初始化操作系统及提供部分硬件的驱动程序,具体功能包括。

(1) 初始化。所谓初始化是指从系统上电复位开始直到 VxWorks 开始初始化用户应用时(即系统执行到 `usrAppInit` 函数处)的一段时间内系统所执行的过程。这个过程主要包括三个部分的工作。

- CPU 初始化。初始化 CPU 的内部寄存器(如状态寄存器、控制寄存器、高速缓存等)。
- 目标机初始化。初始化控制芯片的寄存器(如 BUS、DMA、DRAM)、I/O 设备寄存器(驱动各设备),为整个软件系统提供底层硬件环境的支持。
- 系统资源初始化。为操作系统及系统的正常运行做准备,进行资源初始化(如操作系统初始化、空间分配等)。

(2) 使 VxWorks 能够访问硬件驱动程序。这主要是指 BSP 包含部分必要的设备驱动程序和相关设备的初始化操作。

(3) 在 VxWorks 系统中,集成了与硬件相关(Hardware-dependent)的软件和部分硬件无关(Hardware-independent)的软件。

在以后所讨论的 BSP 中,如果未加说明都是指 VxWorks 所包含的 BSP。

11.1.2 板级支持包与 VxWorks

图 11.1 所示的 BSP 在系统中的层次清楚地展现了 BSP 与 VxWorks 之间的具体关系以及在目标机应用系统中 BSP 所处的地位。

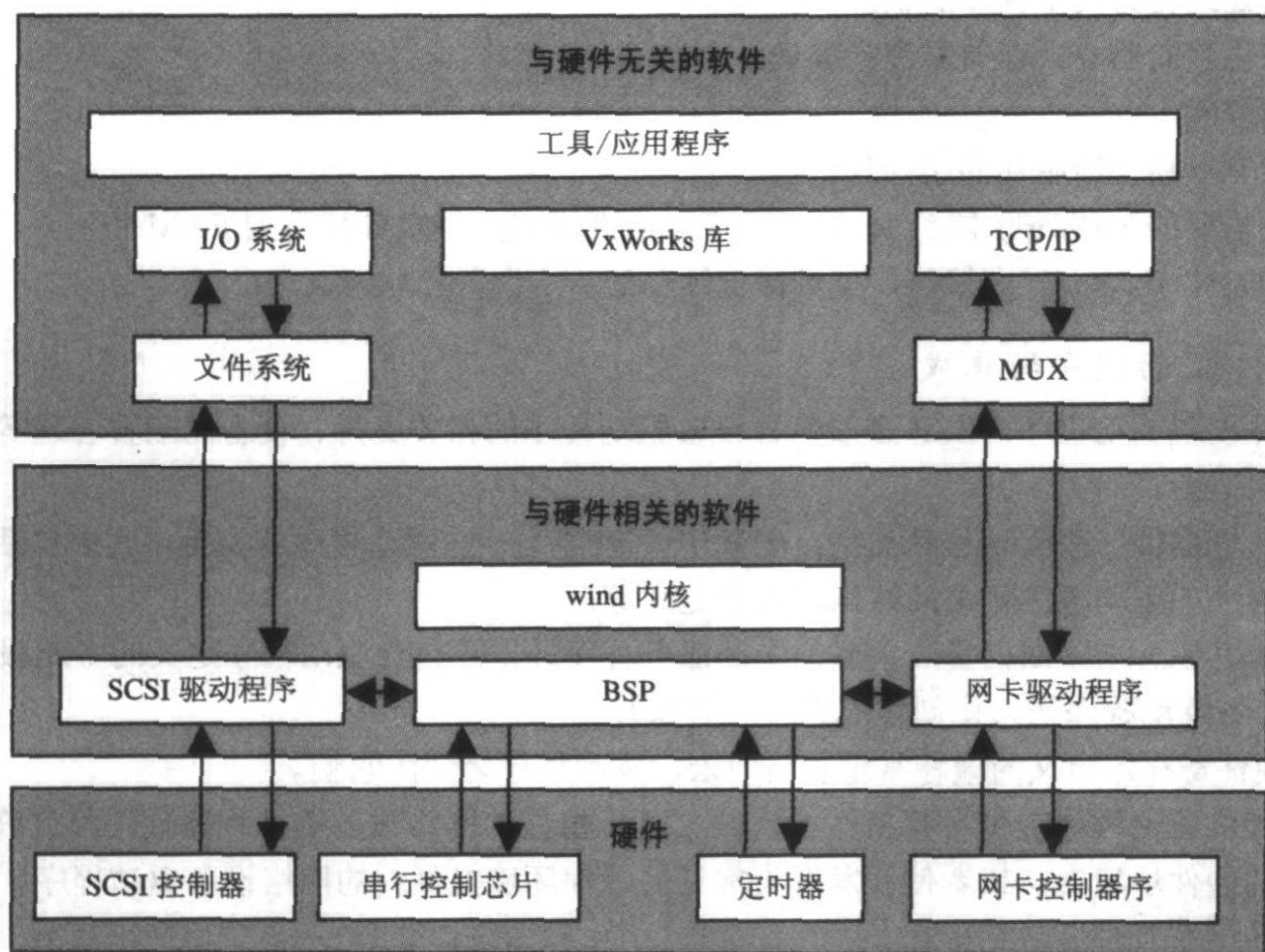


图 11.1 BSP 在系统中的层次

从图 11.1 中可以发现 BSP 不是一个设备驱动程序，因为设备驱动程序能够访问硬件设备（例如网络设备驱动程序、SCSI 设备驱动程序），同时相同类型设备的驱动程序，可以不作修改从一个目标环境中移植到另一个目标环境中（例如网卡驱动程序）；而 BSP 则只能运行在指定设备的硬件环境中。

就 BSP 开发而言，简单地说 BSP 开发人员需要完成两方面的工作：

- (1) 完全支持 BSP 管理的设备；
- (2) 通用设备驱动程序的集成工作。

11.2 BSP 的职责

无论在什么操作系统下，开发目标机的 BSP 时，对于 BSP 的职责应该在宏观上有所了解。概括起来，主要有下面几个要点。

11.2.1 目标机硬件环境的初始化

VxWorks 的引导顺序具体过程将随着处理器和硬件环境而变化，目标机通用初始化的必要条件

是:

- 在存储器里的特殊地址处提供一段代码, 当复位或加电目标机时处理器将立即执行这段代码;
- 设置处理器为某个具体的状态;
- 初始化存储器并对存储器进行编址;
- 关闭中断;
- 将控制权移交给附加的引导代码;
- 装载需要的 VxWorks 段到 RAM 中 (例如把数据段从 ROM 拷贝到 RAM 中);
- 在初始化 VxWorks 内核前, 使目标机硬件处于一个静止状态。

11.2.2 硬件驱动程序的集成

访问硬件设备驱动程序, BSP 必须包含设备驱动程序的相关支持、设备的配置管理等。这主要是指下面几个方面。

(1) BSP 里提供一些驱动程序支持。在 BSP 里包含了一些驱动程序的支持, 其中包括驱动程序的中断服务程序、驱动程序相关的结构以及宏定义等:

- 除 BSP 包含与中断向量表关联的中断服务程序外, 还包含驱动程序定义的中断服务程序;
- BSP 为驱动程序初始化创建的结构 (对象);
- 为硬件寄存器定义的偏移常量和访问宏, 驱动程序将使用它们。

(2) 为硬件驱动程序提供可移植性的代码。可移植性的网络设备驱动程序代码将允许开发者把它移植到新的硬件环境上, 或者使开发者为常见的硬件环境提供新的网络设备驱动程序。

(3) 设备配置管理。包括:

- 可使用设备的全部特点;
- 将产品的开发与配置分开。

11.2.3 与硬件相关软件的集成

一个好的 BSP 不但功能齐全, 它的灵活性、可读性、可移植性等方面也是需要考虑的问题。

1. 提供代码的适应性和可移植性。适应性和可移植性在这里是指:

- 编译时的适应性;
- 执行时的可移植性。

2. 编译时的适应性。编译时的适应性主要从两个方面来说明。

(1) 使用预处理宏定制系统。使用预处理宏, 用户可以根据自己的需要对 BSP 系统进行定制。

例如:

```
#define INCLUDE_ATA      /* 包含 IDE/EIDE(ATA) 硬盘驱动程序 */
#define INCLUDE_FD       /* 包含软盘驱动程序 */
```

(2) 在不改变源程序的情况下, 提供优化目标模块的能力。这主要是指程序可根据用户的配置, 进行目标模块的优化操作, 如下所示:

```
void usrRoot
(
```



```

    char *      pMemPoolStart,      /* 系统内存块起始位置 */
    unsigned    memPoolSize        /* 初始化内存池大小 */
)
{
    ...
#ifdef INCLUDE_WDB
    wdbConfig();                    /* 配置和初始化目标机调试代理——WDB */
    vxBootFile[0] = NULL;          /* 清除引导行 */
    ...
}

```

3. 执行时的可移植性

执行时的可移植性主要包括两个方面。

- (1) 使用指针访问程序。使用指针访问程序，大大提高了程序的执行速度，减少了系统开销。
- (2) 提供编译对象模块的可移植性。

11.3 BSP 的组成

BSP 包含的程序是提供 VxWorks 访问目标机硬件环境的主要接口。BSP 主要由下面几类文件组成。

1. 源文件

主要是用 C 语言编写的代码所组成的文件。同时，有一部分文件是由体系结构相关的且执行最优化的汇编语言编写的代码组成。例如 C 语言编写的 bootConfig.c、usrConfig.c，汇编语言编写的 romInit.s、sysALib.s 文件等。

2. 头文件

包含针对 CPU 板的硬件定义及内存定位定义的文件。例如 config.h、configAll.h 等文件。

3. make file 文件

控制构造所有类型映像的文件。例如 make file 文件。

4. 派生文件

由源文件、头文件等其他文件衍生而成的文件和 VxWorks 存档库模块。这些文件分为下面的几类：

- 硬件初始化对象模块；
- VxWorks 引导对象模块；
- VxWorks 映像；
- VxWorks 二进制符号表。

BSP 的组成文件主要包含在以下四个目录里。

- ..\Tornado\target\config\all
- ..\Tornado\target\config\comps\vxWorks
- ..\Tornado\target\config\comps\src
- ..\Tornado\target\config\bspname

这些目录下的文件将在后面的小节里详细描述。

11.3.1 BSP 的源文件

这一节详细描述了 BSP 下的一些源文件。

11.3.1.1 ..\target\config\all 目录下的文件

这个目录下的文件作为 VxWorks 体系结构产品的一部分而提交给用户。如果不是特殊需要的话没有必要更改这些文件。其中，比较特殊的是 configAll.h 文件。这个文件设置了所有类型 VxWorks 映像的默认配置。在这个文件里定义的宏是一些通用设置，如果所需的 VxWorks 映像不同于默认配置，可以使用 config\bspname 目录下的 config.h 文件中所定义（#define 或 #undef）的宏来替换 configAll.h 的默认配置。

通常，构造工程并不直接使用..\config\all 目录下的文件，只有新建一个可引导型工程时才使用它们。用户创建工程后，config\all 目录下文件的任何修改都不会影响工程。如果希望改变，则必须在工程区域直接进行修改。

简单的说 config\all 下的文件并不是特定 BSP 的一部分，而是所有 BSP 的公共部分。该目录主要由下面的几个文件组成：

1. bootConfig.c

bootConfig.c 是所有引导 ROM 映像的主要初始化和控制文件。它包含完整的引导 ROM 外壳任务、一个网络设备初始化表（NETIF），是文件 usrConfig.c 的一个子模块。引导 ROM 映像不包含完整 VxWorks 映像的所有可选特性。值得注意的是引导映像不使用 MMU 库（SPARC 体系结构除外）。

在 Tornado 2.0 开发环境下，没有创建和修改引导 ROM 工程的功能，只在 Build 菜单中提供了执行构造引导 ROM 映像的命令（Build boot ROM...）。执行此命令后会弹出如图 11.2 所示的构造窗口。

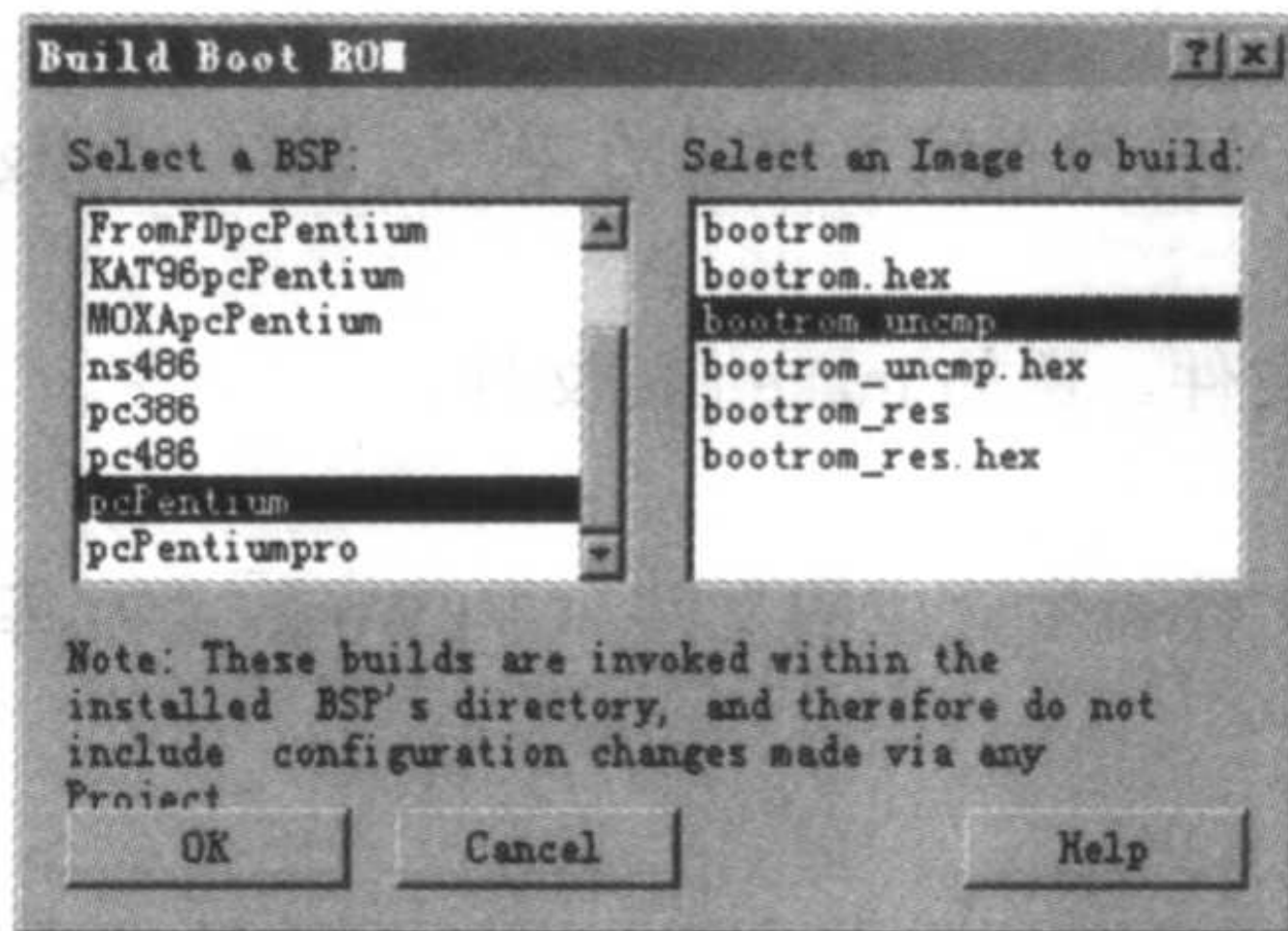


图 11.2 构造引导 ROM 的窗口



注意

在 Tornado 开发环境下修改工程的有关配置信息并不会影响引导映像的构造。只有直接修改 config.h、configAll.h、bootConfig.c 和 bootInit.c 文件时才影响引导映像的构造。

2. bootInit.c

bootInit.c 包含引导 ROM 映像第二阶段的初始化代码。该文件定义了 romStart() 函数，romInit.s 文件中的 romInit() 函数执行完后将会跳转到 romStart() 函数。该函数执行必要的代码压缩和 ROM 型

映像的重定位，即首先把代码段和数据段从 ROM 中拷贝到 RAM 中，然后初始化主 RAM 中不使用的部分，最后进行映像压缩部分的解压操作。映像配置的不同选择将会执行不同操作。

在 Tornado 2.0 开发环境下创建的可引导映像工程中的 `romStart.c` 文件替代了 `bootInit.c` 的功能（这是对基于 ROM 的 VxWorks 映像而言，而不是指引导 ROM 映像）。

3. dataSegPad.c

只有在 VxVMI 对代码段进行写保护时才使用 `dataSegPad.c` 文件。它保证了代码段和数据段不会共享一个 MMU 页。

4. usrConfig.c

`usrConfig.c` 文件包含了 VxWorks 映像的主要初始化代码。不同于 `bootConfig.c` 文件，它是非常完整的，包含了 `..\target\src\config\usrExtra.c` 文件以及其他提供子系统初始化和配置的文件。

这个文件只有当使用命令行方式编译时才使用，就像在 Tornado 1.0 环境所使用的。当使用 Tornado 2.0 开发环境创建工程时，工程初始化组合决定了模块初始化顺序。每一个模块都有一系列属性，这些属性确定了初始化组合。

5. version.c

`version.c` 文件提供了 VxWorks 版本号、构造映像时的日期和时间。这些信息总是在执行 VxWorks 映像时显示，例如：

```
...
VxWorks: 5.4.2
  BSP version: 1.2
Creation date: Oct 4 2003
```

6. configAll.h

`configAll.h` 文件设置了所有 VxWorks 映像的默认配置。在这个文件里定义的变量是一些通用设置，如果需要的 VxWorks 映像不同于默认配置，使用 `config\bspname` 目录下的 `config.h` 文件中所定义（`#define` 或 `#undef`）的宏来替换 `configAll.h` 的默认宏。



注意

更改 `configAll.h` 文件将会影响通过开发环境新创建的可引导型映像工程。所以一般情况下，在 `config.h` 文件中根据工程需要定义宏来替换 `configAll.h` 的默认宏定义，而不是直接修改 `configAll.h` 文件。

11.3.1.2 ..\target\config\bspname 目录下的文件

这个子目录包含系统或硬件相关的 BSP 文件。主要由下面的一些文件组成。

1. Readme

`Readme` 文件包含了 BSP 发布记录。它记录了每次发布版本、修订版本和在新版本上所做的工作。下面是某个 BSP 的 `Readme` 文件。

```
=====
Readme - BSP 自述文件
=====
README: PC-386/486/Pentium/PentiumPro
```


This file contains board-specific information for a generic PC-386/486/Pentium/PentiumPro target board.
 This file documents BSP interface changes from previous software or hardware versions, and notifies the user of serious caveats that must be noted before using this BSP. Additionally, the target board's manual page entry (e.g., man pc386, man pc486, man pcPentium, man pcPentiumpro) provides board-specific information necessary to run VxWorks. The target manual page should be read fully before this BSP is used.

-----RELEASE 1.2/0

First release for Tornado 2.0

RELEASE 1.1/4

This release includes Pentium/PentiumPro support.

...

2. makefile 和 depend.bspname 文件

makefile 文件控制使用命令行方式构造映像。从 VxWorks 5.2 开始, 标准的 make 工具是 GNU make, 并且当前的 make 技术继承了许多 GNU make 的优点。BSP make 系统包含了位于 target\h\make 目录下的 make 子文件。如果某个相关性文件不存在, make 将自动产生 depend.bspname 文件, 它可以用来追踪所有模块的相关性。depend.bspname 相关性文件并不包含在 BSP 中, 它是一个自动产生的文件。

在 makefile 文件里, 必须定义下面的宏。

- (1) CPU, 目标机 CPU 体系结构 (例如 CPU = PENTIUM)。
- (2) TOOL, 宿主机工具链 (例如 TOOL = gnu)。
- (3) TARGET_DIR, 目标目录名 (指 BSP 目录, 例如 TARGET_DIR = \$(WIND_BASE)/target)。
- (4) VENDOR, 目标机制造商的名字 (例如 VENDOR = IBM)。
- (5) BOARD, 目标机的名字 (例如 BOARD = PC Pentium)。
- (6) ROM_TEXT_ADRS, 引导 ROM 的入口地址 (以 16 进制表示, 例如 ROM_TEXT_ADRS = fff20000)。对于多数目标板, 这是 ROM 地址区的起始地址设置。
- (7) ROM_WARM_ADRS, 热启动入口, 指向引导 ROM 代码。
- (8) ROM_SIZE, ROM 空间的大小 (以 16 进制表示, 例如 ROM_SIZE = 0007fe00)。
- (9) RAM_LOW_ADRS, 装载 VxWorks 映像的地址 (以 16 进制表示, 例如 ROM_LOW_ADRS = 00108000)。
- (10) RAM_HIGH_ADRS, 拷贝引导 ROM 映像到 RAM 中的目标地址 (以 16 进制表示, 例如 RAM_HIGH_ADRS = 00008000)。

3. config.h 文件

config.h 文件包含了所有头文件和与 CPU 板相关的特殊定义。标准的 config.h 包含下面的内容。

(1) BSP 版本号和修订版本号:

```
#define BSP_VERSION    "1.2"
#define BSP_REV        "/0" /* 0 为第一次修订版本 */
```

(2) 包含 configAll.h 文件:

```
#include "configAll.h"
```


(3) 高速缓冲存储器和 MMU 的配置:

```
#define USER_I_CACHE_ENABLE
#define USER_D_CACHE_ENABLE
#define USER_B_CACHE_ENABLE
#define USER_I_MMU_ENABLE
#define USER_D_MMU_ENABLE
```

(4) 共享内存网络定义:

```
#define INCLUDE_SM_SEQ_ADDR
```

(5) 目标板上的内存地址和大小:

```
#define LOCAL_MEM_LOCAL_ADRS 0x00000000
#define LOCAL_MEM_BUS_ADRS 0x00000000
#define LOCAL_MEM_SIZE 0x00800000 /* 8MB 或更低内存 */
```

(6) ROM 地址和大小:

```
#define ROM_BASE_ADRS 0xffff20000 /* ROM 的基地址 */
#define ROM_SIZE 0x0007fe00 /* ROM 大小 */
```

(7) 非易失性 RAM (NVRAM, non-volatile memory) 参数。例如:

```
#define NV_RAM_SIZE NONE /* 没有 NVRAM */
```

(8) 默认引导参数:

```
#define DEFAULT_BOOT_LINE \
"fd=0,0(0,0)host:/fd0/vxWorks h=90.0.0.3 e=90.0.0.50 u=target"
```

(9) 时间戳 (timestamp) 驱动程序支持:

```
#define INCLUDE_TIMESTAMP /* 包含时间戳定时器, 提供给 Wind View 工具 */
```

(10) 外部总线地址映射。

(11) 网络设备和中断向量:

```
#define INCLUDE_END /* 使用增强型网络驱动 */
#define INCLUDE_FEI /* 包含 Intel Ether Express PRO100B PCI */
#define NUMBER_OF_IRQS 24
#define INT_NUM_IRQ0 0xe0 /* IRQ0 的向量号 */
```

(12) 包含 *bspname.h* 文件:

```
#include "pc.h"
```

 注意

config.h 文件应当包含 configAll.h 和 *bspname.h* 文件。

4. romInit.s 文件

该文件包含引导 ROM 和基于 ROM 的 VxWorks 映像的入口初始化汇编代码。入口点为 romInit() 函数, 是系统加电启动后首先执行的代码。同时它也设置了启动类型参数 BOOT_COLD, 并把该参数传递给通用的 romStart() 函数。如果硬件需要立即内存映射或设置特殊的寄存器, 则在这里处理。大部分硬件初始化操作在 sysLib.c 文件中的 sysHwInit() 函数里进行。

在 romInit()函数里, 有三个必须执行的主要功能:

- 屏蔽中断, 初始化 CPU;
- 初始化内存系统, 通常包括关闭高速缓存和初始化内存控制器。对于 SPARC 体系结构的目标机, 必须打开 MMU;
- 初始化堆栈指针和其他寄存器以便能够执行 C 代码, 然后调用 romStart()函数。

值得注意的是 romInit()函数必须设计成与地址无关的代码 (PIC, Position-Independent Code)。它需要支持综合的 VxWorks 引导策略。如果必须使用绝对地址, 通常会通过 ROM_ADRS(x)宏对地址重新计算。该宏定义如下:

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

一个普遍的错误是 BSP 开发人员企图在 romInit.s 里进行过多的初始化操作。实际上大部分其他初始化功能应推迟到 sysHwInit()函数中进行。另外, romInit.s 中的代码不应当被其他模块或函数调用。

另一个普遍的错误是 BSP 开发人员认为在 romInit.s 里初始化过的设备不需要在 sysALib.s 或 sysLib.c 中重新初始化。实际上 VxWorks 映像并不认为它是由引导映像程序引导的。因此, 它必须重新设置和重新初始化所有它自己需要使用的设备。一般情况下, VxWorks 的引导映像主要用于 VxWorks 映像的调试和开发阶段。当 VxWorks 映像开发结束后, VxWorks 的引导映像也就完成了它的使命。



注意

产品中的 VxWorks 映像尽量不要依赖于引导 ROM 映像。

5. sysALib.s

这个文件包含目标机体系结构决定的、与系统有关的汇编程序。所有 BSP 程序应当用 C 语言编写, 然而如果由于编译等原因需要用汇编语言编写程序, 可以把所编写的汇编程序放到 sysALib.s 文件里。

sysALib.s 文件里的 sysInit()函数是 VxWorks 映像的入口点。系统引导后, 首先执行该段代码。sysALib.s 所执行的所有功能类似于 romInit.s 的工作, 但它不执行系统内存初始化操作。系统应当在程序启动阶段初始化内存系统。

与 romInit.s 不同的是不需要把 sysALib.s 中的代码编写成与 PIC 无关的代码或使用 ROM_ADRS 宏重新映射绝对地址。它可以调用其他模块和程序。

6. sysLib.c

这个文件包含目标机体系结构决定的、与系统有关的 C 程序。这些程序提供板级接口, 基于这些板级接口, VxWorks 和应用程序的构造与系统无关。sysLib.c 文件可以包含 src/drv 目录下的驱动程序。

驱动程序的安装和初始化通常在子文件中完成, 在 sysLib.c 中包含了这些操作。这些子文件通常的命名是 sysSerial.c、sysScsi.c、sysNet.c 等。使用子文件的目的是为了保持设备驱动程序的安装和初始化不随结构的变化而改变。一块目标板上的设备安装子文件应该能被另一块板上同样的设备

所使用。

7. sysSerial.c

这是一个可选文件，包含所有的 SIO 设备串行 I/O 驱动程序的安装和初始化。系统没有单独编译它，而是将其包含在 sysLib.c 文件中。把串行设备初始化从 sysLib.c 中独立出来形成一个模块，则允许在 BSP 之间重复使用这个代码。

8. sysScsi.c

这是一个可选文件，包含 SCSI-2 设备驱动程序的安装和初始化。像 sysSerial.c 文件一样包含在 sysLib.c 文件中。

9. sysNet.c

可选文件，包含网络接口驱动程序的安装和初始化。

10. bspname.h

包含与目标板有关的特殊信息定义。例如串行接口、时钟、I/O 设备。许多相关信息包含在 target/h/drv 目录下，这些 VxWorks 所要使用的目标板包含设备的驱动程序头文件应当包含在 bspname.h 文件的开始。

这个文件预定的常量信息不应该影响用户的配置。如果定义在 bspname.h 文件里的任何宏或值可以改变系统的定制，那么应该由定义在 config.h 文件中的同样的宏或值所替代。

使用一个样本头文件是非常有用的，因为大多数已经被定义的常量名、基本设备地址等已经定义只需要根据实际情况修改。bspname.h 文件中应定义下面的信息：

- 中断向量、中断级别；
- I/O 设备地址；
- 设备寄存器位的含义；
- 系统时钟和辅助时钟参数（最大和最小速率）。

同样，使用一个目标板描述来说明目标板所需要的特殊宏定义（例如以“PCI_”为开头定义相应的设备）。

11. target.nr

这个文件包含了运行 VxWorks 必需需要的目标板特殊信息。对于 Tornado 2.0，target.nr 文件并不可以作为 UNIX 类型参考页或 Windows 兼容的文本页来处理，而是把它作为一个 HTML 文件提交给用户。

12. configNet.h

网络配置头文件。主要包含 END 网络设备驱动程序的配置。

11.3.1.3 ..\target\config\comps\vxWorks 目录下的文件

这个目录下的文件是 VxWorks 实时内核基本模块描述文件（CDF 文件）。有关详细信息参考有关模块描述语言（CDL, Component Description Language）的相关书籍。

11.3.1.4 ..\target\config\comps\src 目录下的文件

这个目录下的文件是 VxWorks 实时内核模块配置文件。这些文件提供的代码被 usrConfig.c 文件所使用。

11.3.2 派生文件

所谓派生文件是指从源文件、系统头文件、驱动程序源文件和 VxWorks 文档库中的模块衍生而来的文件。这些文件仅仅为示范目的而衍生出来的文件，当配置一个系统时，可以重新创建这些文件中的部分文件或所有文件。

通过 Tornado 2.0 开发环境创建工程时，这些派生文件被放置在工程目录下。只有通过命令行方式构造工程时，系统才会使用在 BSP 目录下的派生文件副本。

下面是一些派生文件。

(1) bootInit.o、bootInit_res.o、bootInit_uncmp.o。ROM 型映像有三个引导策略，分别是：压缩的 ROM 型、非压缩的 ROM 型、ROM 驻留型。这三个引导策略可以被引导 ROM 映像或 ROM 型 VxWorks 映像所使用。模块的名字指出了所使用的引导策略。

(2) bootrom，是一个包含二进制的 VxWorks 引导 ROM 的对象文件。

(3) bootrom.hex，是一个包含 VxWorks 引导 ROM 代码的 ASCII 文件。适合于通过串行连接 PROM 编程器来下载该文件。默认的编码使用 Motorola S 记录文件格式。

(4) bootrom.Z.s 和 bootrom.Z.o，是压缩型 bootrom.o 映像模块的源程序和对象文件。

(5) ctdt.o，是管理 C++ 构造/析构函数的模块。

(6) dataSegPad.o，是保证代码段和数据段不会共享同一个内存页的模块。使用了 VxVMI 模块时，这个模块强制数据段和代码段处于不同的 MMU 页中。

(7) symTbl.c 和 sysALib.o，是内置符号表的源程序和对象文件。

(8) sysLib.o 和 sysALib.o，这两个模块总是一起成对出现。它们是 sysLib.c 和 sysALib.s 文件的输出模块。

(9) romInit.o，这个模块是引导映像或 VxWorks 映像的启动代码。对于 Tornado 2.0 而言，它联合 romStart.o 模块一起工作，替代了 bootInit.o。

(10) vxWorks 和 vxWorks.sym，vxWorks 文件是链接好的、完整的 VxWorks 二进制文件，可以在目标机上运行。vxWorks.sym 是 vxWorks 二进制文件的符号表文件。系统通过配置文件创建这两个文件。

(11) vxWorks.st，是一个链接好的、完整的、内置符号表的 VxWorks 二进制文件。

(12) target/proj/bspname_vx/*.*, 这个目录包含 BSP 工程文件和默认构造输出。有两种方式创建一个新的工程：一种是使用工程工具；一种是通过命令行的方式执行 make 来实现。

(13) doc/vxworks/bsp/bspname/*.*, 这个目录包含了用户帮助系统的 HTML 文档文件。在命令行方式下执行 make man 命令可以产生这些文档。

11.3.3 BSP 里的函数

BSP 里包括下面几类函数。

1. 初始化函数

通过初始化函数进行三类操作：

- 初始化硬件到已知状态；
- 标识系统；

- 初始化设备，如 SCSI 或常规设备。

2. 内存/地址空间函数

通过内存/地址空间函数主要获得下面的信息：

- 得到板上内存大小；
- 总线地址空间；
- 设定/获得非易失性 RAM (NVRAM)；
- 定义板的内存位图（可选）；
- 为有 MMU 的处理器定义虚拟内存到物理内存的映射。

3. 总线中断函数

总线中断函数主要实现下述功能：

- 打开/关闭总线中断；
- 产生总线中断。

4. 时钟/定时器函数

时钟/定时器函数主要对时钟/定时器进行两步操作：

- 打开/关闭定时中断；
- 设置定时器的周期性。

5. 邮箱/位置监视函数（可选）

- 打开邮箱/位置监视中断。

这些函数原型及说明在后面的小节中进行了简单描述。

11.3.3.1 必需的函数

表 11-1 中的函数是在 sysLib.o 模块中必须包含的函数。如果没有发现其中任何一个函数，在链接的时候将会出现一个全局错误。

表 11-1 BSP 中必须需要的函数

函 数	描 述
sysBspRev()	返回 BSP 版本和修订号
sysClkConnect()	把中断服务程序连接到系统时钟中断上
sysClkDisable()	关闭系统时钟中断
sysClkEnable()	打开系统时钟中断
sysClkInt()	处理系统时钟中断
sysClkRateGet()	获得系统时钟频率
sysClkRateSet()	设置系统时钟频率
sysHwInit()	初始化系统硬件
sysHwInit2()	初始化附加的系统硬件
sysMemTop()	获得逻辑存储的顶端地址
sysModel()	返回 CPU 板的模块名

续表

函 数	描 述
sysNvRamGet()	获得 NVRAM 中的内容
sysNvRamSet()	向 NVRAM 中写数据
sysSerialHwInit()	初始化 BSP 中的串行设备到静止态
sysSerialHwInit2()	连接 BSP 中的串行设备中断
sysSerialChantGet()	获得与 SIO_CHAN 设备关联的串行通道
sysToMonitor()	把 CPU 控制权转交给 ROM 监控程序

11.3.3.2 可选函数

表 11-2 中的函数通常在所有的目标机上出现。这些函数作为 BSP 内部使用，应用程序不调用它们。

表 11-2 BSP 中可选函数

函 数	描 述
sysAbortInt()	处理 ABORT 按钮中断
sysAuxClkConnect()	连接中断服务程序到辅助时钟中断上
sysAuxClkDisable()	关闭辅助时钟中断
sysAuxClkEnable()	打开辅助时钟中断
sysAuxClkInt()	处理辅助时钟中断
sysAuxClkRateGet()	获得辅助时钟频率
sysAuxClkRateSet()	设置辅助时钟频率
sysPhysMemTop()	获得物理存储顶端地址

而表 11-3 中的函数通常只是在具有总线接口的目标机上出现。

表 11-3 具有总线接口的 BSP 所具有的函数

函 数	描 述
sysProcNumGet()	获得处理器号
sysProcNumSet()	设置处理器号
sysBusIntAck()	应答总线中断
sysBusIntGen()	产生总线中断
sysBusTas()	测试并设置交叉总线位
sysBusTasClear()	清除由 sysBusTas()设置的总线位
sysBusToLocalAdrs()	转换总线地址为本地地址

续表

函 数	描 述
sysIntDisable()	关闭总线中断
sysIntEnable()	打开总线中断
sysLocalToBusAdrs()	转换本地地址为总线地址
sysMailboxConnect()	把中断服务程序连接到邮箱中断上
sysMailboxEnable()	打开邮箱中断
sysMailboxInt()	处理邮箱中断
sysNanoDelay()	设置校准延迟（自旋循环）

11.4 BSP 的开发

BSP 的开发对于系统开发人员来说是一件比较“棘手”的问题，如何快速而有效的对一目标板进行 BSP 的开发，也是系统开发人员经常谈论的问题。下面就 BSP 的开发作一些宏观上的描述，而在第 16 章中将详细介绍。

1. BSP 的开发步骤应当是一个逐步增加功能的过程

BSP 开发过程基本步骤如下：

- (1) 首先设置系统环境（例如下载方式、调试策略等）；
- (2) 编写预内核初始化代码；
- (3) 选择目标机调试代理 WDB 与开发环境下 Tornado 工具的连接方式（例如使用轮询串口或以太网接口）；
- (4) 启动最小的 VxWorks 内核加载系统时钟支持，安装中断；
- (5) 完成 BSP 提供的所有必须的硬件环境支持（例如网络支持等）；
- (6) 整理、测试 BSP 和编写相应文档。

2. 开发过程中的软、硬件保障

BSP 的开发不同于应用程序的开发，它的操作直接与硬件相关，所以在开发过程中会借助一些硬件和软件工具来辅助开发。例如 ICE（在线仿真器）、BDM 等。

3. 开发效率的提高

在 BSP 开发过程中，如何缩短开发周期，提高开发效率也是所必须关心的问题。下面为提高开发效率提供了一些方法。

(1) **BSP 开发包**。为了满足一些用户的需求，Wind River 公司为系统开发人员准备了 BSP 开发包。在这个“包”里包括：

- 有效性测试套件（VTS, Validation Test Suite）；
- 所有体系结构的 BSP 模板（例如 x86、PPC、68K、MIPS 等）；
- 设备驱动程序模板（例如硬盘、网络接口、SCSI 等）。

(2) **参考的 BSP**。根据自己的目标机硬件环境，参考一个与硬件环境最接近的 BSP。

注意

当购买开发环境 Tornado 时，包含了参考 BSP。但是没有包含一些特殊的设备驱动程序。

4. BSP 调试方法

对于 BSP 的调试，通常有下面三种方法。

(1) ICE 调试法。ICE 调试法在开发 BSP 过程中是最常用的一种方法。其过程是仿真头插在用户板的 CPU 位置上，被调试的 VxWorks 映像下载到仿真器 RAM 中或目标机 RAM 中，由仿真器的 CPU 在用户的硬件环境下运行来调试软件，如图 11.3 所示。

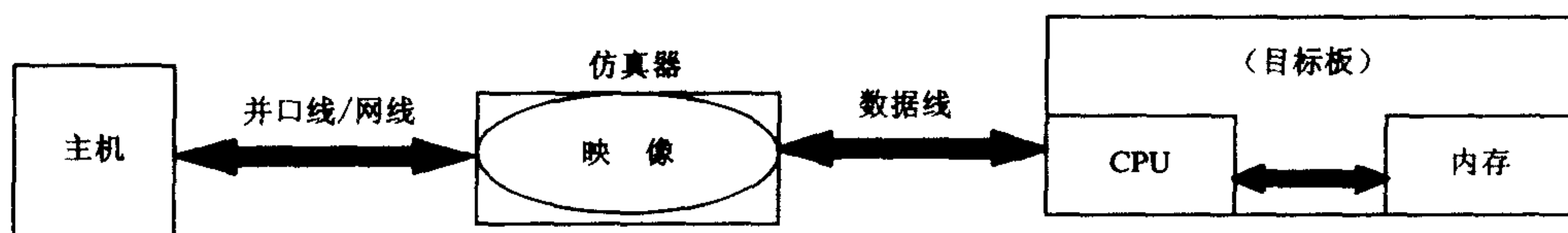


图 11.3 ICE 调试法

这种调试方法的缺点在于 ICE 价格/使用率的比率太高，并且不同类型的 CPU 需要购买不同的 ICE。

(2) BDM 调试法。宿主机通过并口与 BDM 适配器连接，而 BDM 适配器通过自带的数据线连接到目标板上 CPU 的 BDM 接口上，且用户板上的 CPU 支持一种断点逻辑。通过这种硬件调试代理，进行映像的调试，如图 11.4 所示。

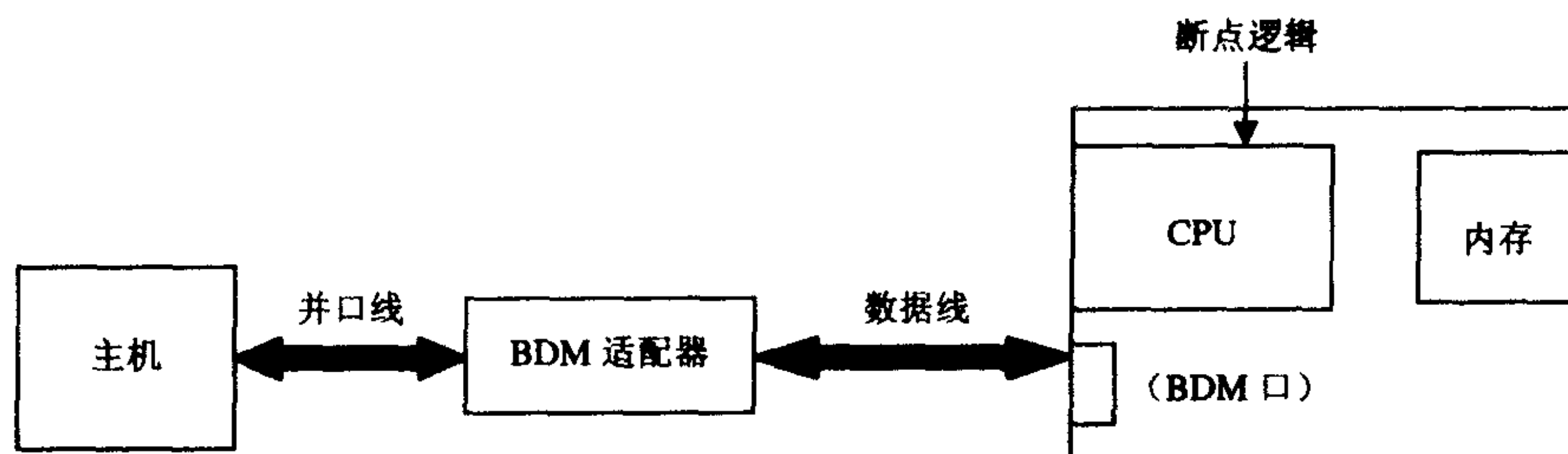


图 11.4 BDM 调试法

这种调试方法的缺点在于 CPU 必须支持 BDM 方式。其步骤是首先通过 BDM 适配器下载激活目标板 CPU 的程序，然后才把被调试的 VxWorks 映像下载到目标机 RAM 中。

(3) “黑”调试法。黑调法是通过点灯、用示波器测片选等手段调试 BSP，类似在调试应用程序时使用 printf() 函数打印程序执行位置。目的是使 BSP 可以驱动目标板串口，实现宿主机与目标机之间的通信。

黑调法工程步骤如图 11.5 所示。

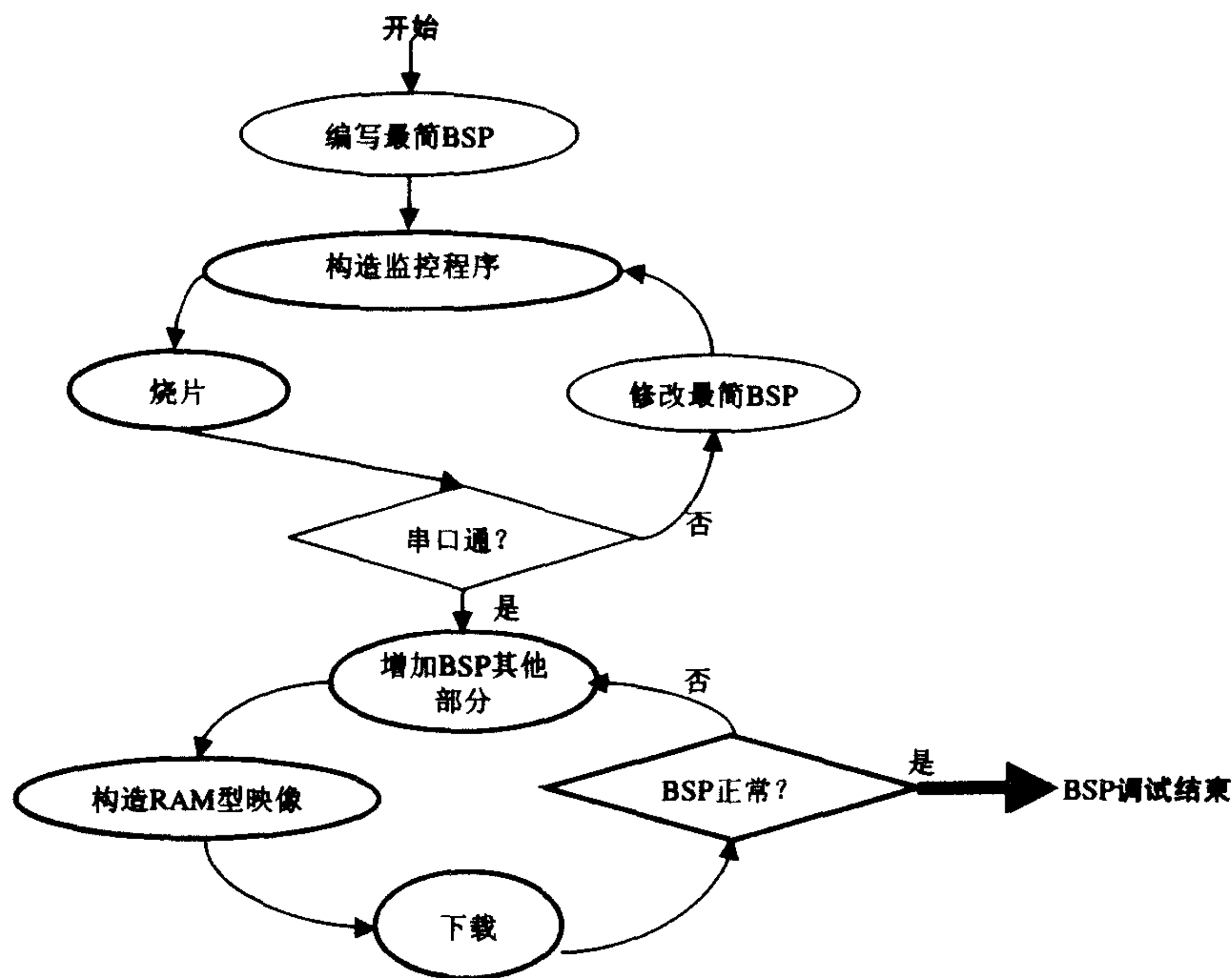


图 11.5 “黑”调试法工程步骤

11.5 Tornado 的目录结构

这一节主要介绍 Tornado 的模块组成及目录树。这对 BSP 开发人员来说是最基本的知识，也是必须了解的内容。

11.5.1 Tornado 模块性

Tornado 开发系统包含三个高度集成的部分：

- 运行在宿主机和目标机上功能强大的交叉开发工具和实用程序；
- 运行在目标机上的高性能、可裁剪(用户根据自己需要配置系统)的实时操作系统 VxWorks；
- 连接宿主机和目标机的多种通讯方式，如以太网、串口线、ICE (在线仿真器)、ROM 仿真器或用户定制的连接方式等。

而这些部分主要由下面模块组成。

- (1) 宿主机支持的包 (HSP, Host Support Package)。
- (2) 通用 (与目标机无关) 的 VxWorks。
- (3) 体系结构模块 (与目标机有关)。
- (4) 板级支持包——BSP。
- (5) 目标机调试代理 (WDB, Wind Debug Agent)。

这些模块在移植性、裁剪性、维护等方面非常方便，它们之间的层次如图 11.6 所示。

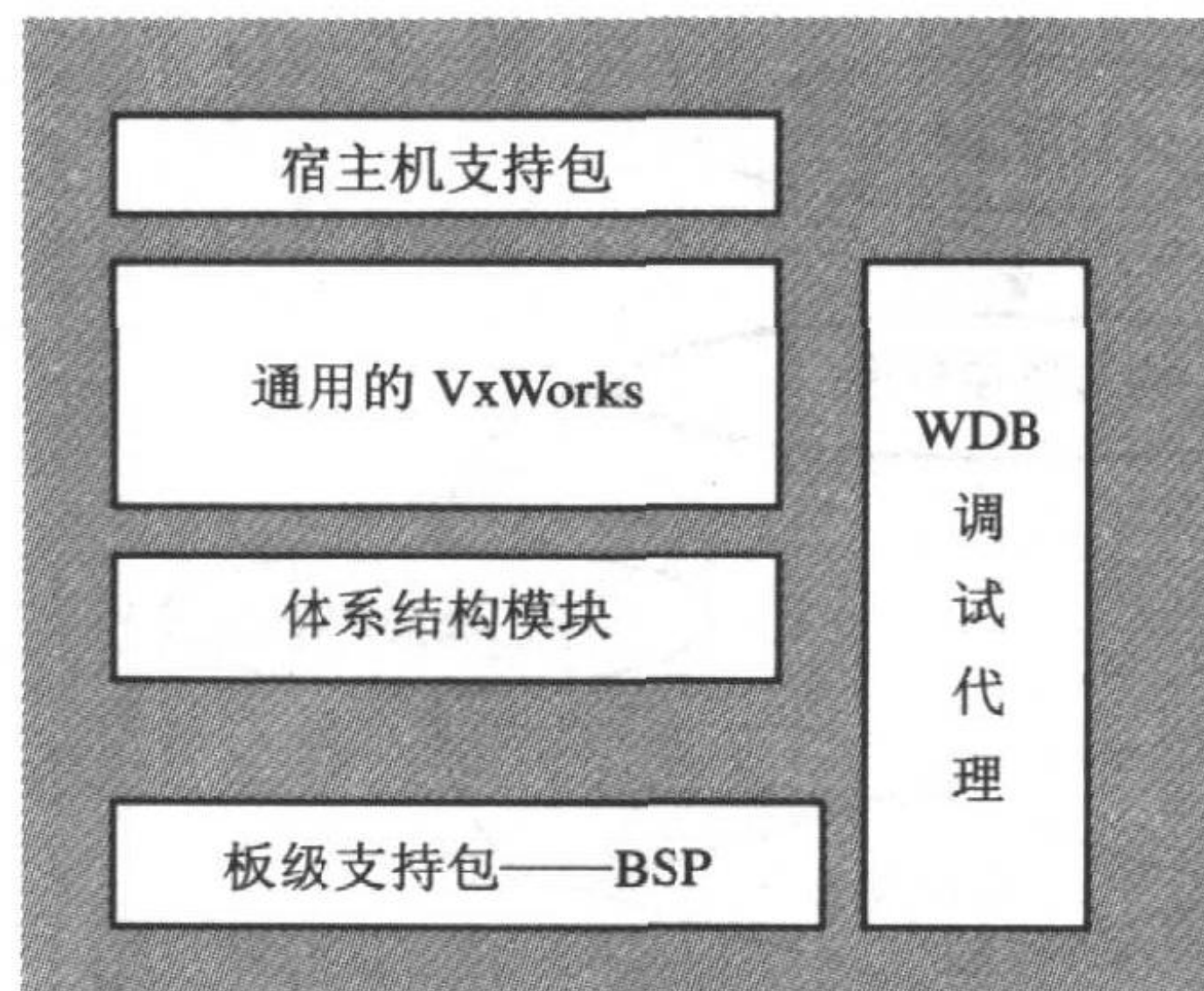


图 11.6 Tornado 模块

11.5.2 Tornado 目录树

Tornado 的组成模块是以文件的形式存放在 Tornado 目录下，各个目录包含不同的模块。Tornado 下的主要目录如图 11.7 所示。

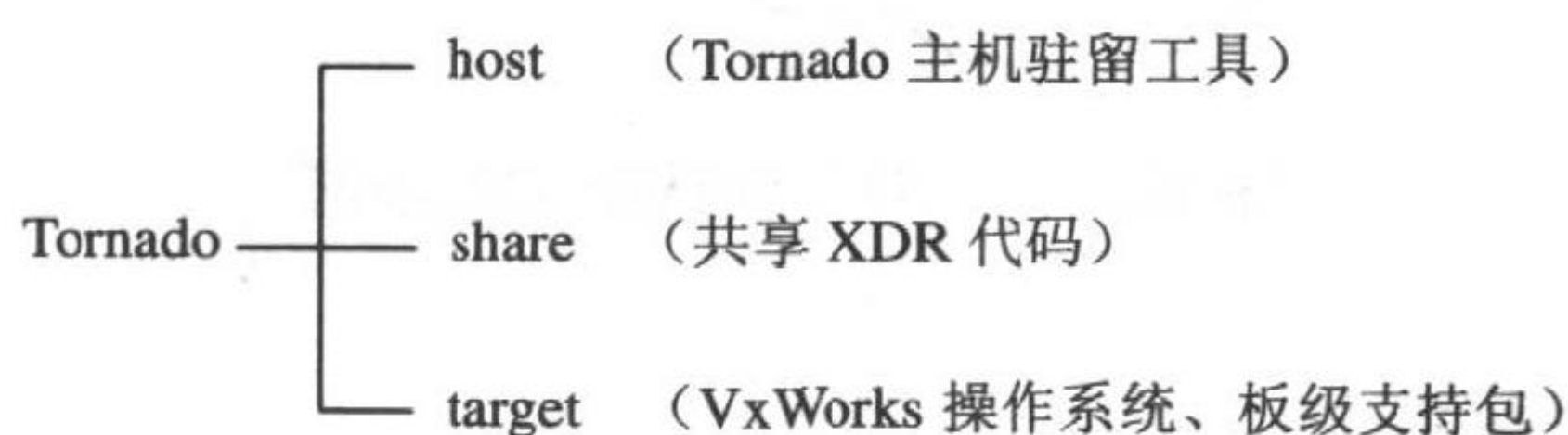


图 11.7 Tornado 目录树

11.5.3 Host 目录树

Host 目录下主要包含一些开发工具，例如 make、编辑器、调试器、文件格式转换工具等，如图 11.8 所示。

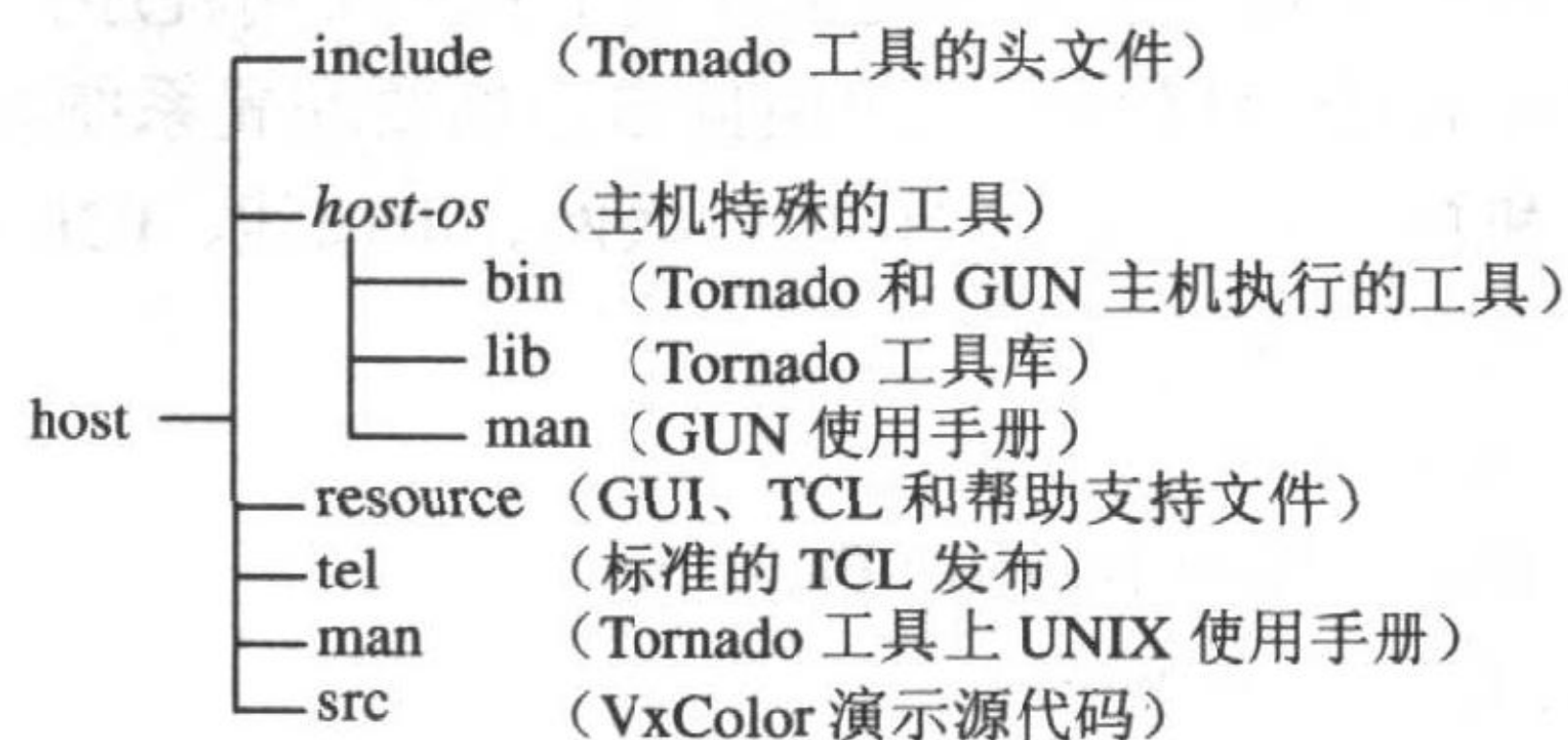


图 11.8 Host 目录树

11.5.4 Target 目录树

Target 目录下主要包含 BSP、VxWorks 头文件、VxWorks 库文件以及 VxWorks 部分源文件等，如图 11.9 所示。

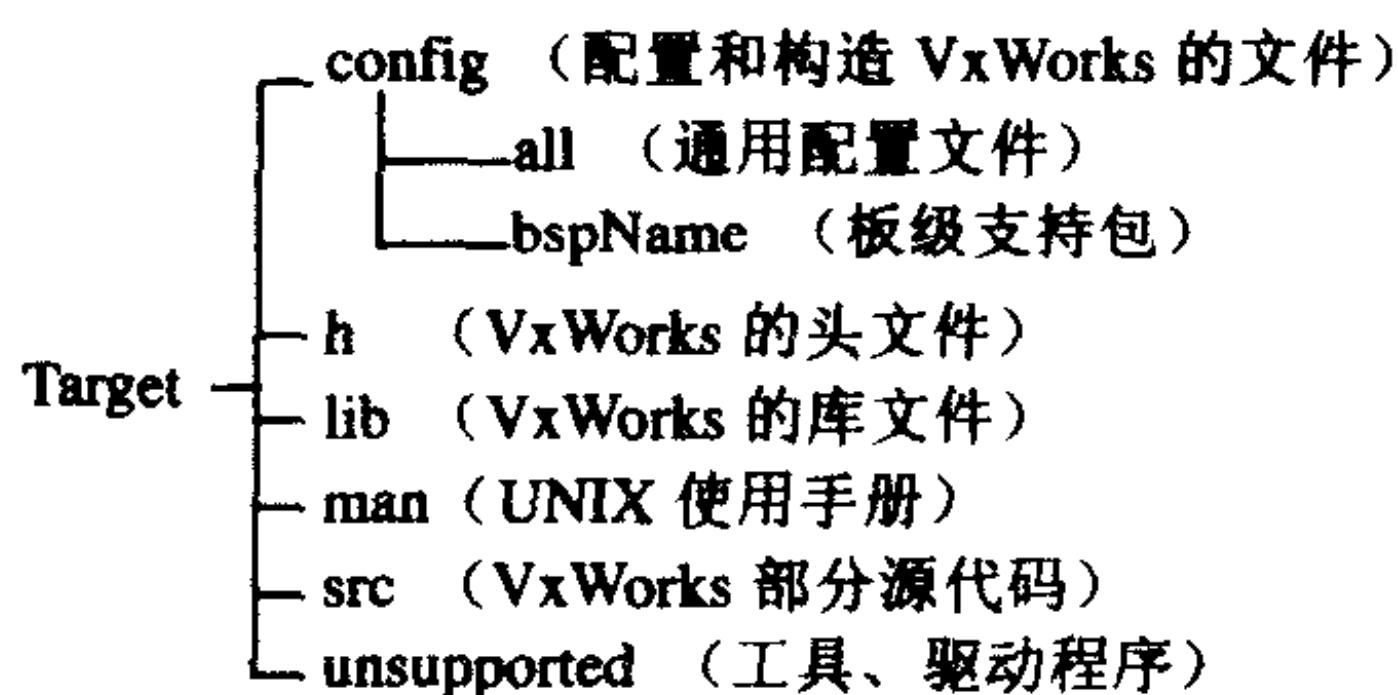


图 11.9 Target 目录树

11.6 BSP 的约定和有效性

当许多人共同设计一个由许多模块组成的系统时，不同时间内会出现不同的版本，不同的人会有不同的编程风格及文档编写风格。这就要求在项目开发前应该有一些约定，来约束设计和编程的规范化。同样，BSP 的开发也要求规范化操作。

1. BSP 约定

BSP 约定主要包括下面的内容：

(1) 编码约定。编码约定是 BSP 代码编写前所必须进行的约定，这样便于在开发过程中进行技术交流和后期代码的移植等工作。编码约定通常包括源程序文件的命令约定、函数命令约定、变量命名约定等，通常以文字加示例提供给开发人员。

(2) 文档编写指南。文档编写指南主要为 BSP 开发人员文档的编写格式和所具备的内容等提供帮助，使文档规范化，通常要求以文字加模板形式提供给开发人员。这些文档包括概要设计文档、详细设计文档、使用手册等。

(3) BSP 打包。BSP 的各部分编写完成后，要对 BSP 进行打包，以完整的形式提供。

(4) 驱动程序编写指南。

2. BSP 的有效性测试

BSP 编写完成后，要对它进行有效性测试。这是 BSP 开发过程中必须要进行的步骤，并且测试人员和编写人员要相互独立。所要进行的操作是：

(1) BSP 包完整性测试。这主要是指根据 BSP 开发人员提供的 readme 文档，检查所提供的 BSP 文件是否完整。

(2) BSP 安装测试。按 BSP 开发人员的要求，进行 BSP 的安装工作，检查是否能够成功安装。

(3) BSP 功能测试。BSP 的功能测试是测试过程中的重点和难点，测试人员通常采用“黑盒”法测试技术对 BSP 的功能进行测试。对于这一步测试，通常需要设计和编写测试用例。

(4) 代码复查和最终确认。

总之，BSP 的协定和有效性测试过程可以帮助 BSP 开发人员保证 BSP 设计的完整性。

小 结

BSP 在系统中的地位是不言而喻的, 那么 BSP 具体是什么、它包含哪些操作内容、VxWorks 下的 BSP 由哪些文件组成等, 都是需要了解的问题。而这一章对上述问题一一做了回答。在内容上包括 BSP 的概述、BSP 的职责、BSP 的组成、BSP 的开发以及 BSP 的协定和确认等方面。从这一章开始, 后面所有章节的内容都将围绕 BSP 展开描述。

第 12 章 VxWorks 映像及启动顺序

这一章详细描述 VxWorks 映像启动情况。从根本上看，在初始化和装载 VxWorks 映像的过程中，处理器所执行的步骤在逻辑上是一样的。对于有些处理器可能需要增加一些额外的步骤，而另一些处理器可能会省略掉某些步骤。本章将一步一步描述这些步骤，但是有些处理器将不会需要某些步骤。

最小的初始化处理是由放置在目标机内存特殊位置的一部分代码和可能的一些表(中断描述表、全局描述表)组成。当目标机复位或加电启动后，处理器总是跳转到这个位置。这部分代码初始化处理器到某一指定状态、初始化内存和内存地址、关中断，然后把控制权转交给附加引导带代码。

对于各种各样的配置和体系结构，这段代码包含的引导顺序是按部就班的。在某些情况下，有关引导顺序的一些描述最好提供一个大致指导，因为对于一些特殊的芯片可能只有惟一的变化，例如 M68302 嵌入式控制器。

12.1 VxWorks 映像类型

当构造 VxWorks 映像时，可以根据需要构造不同类型的映像。系统把这些映像划分成三种类型，各类型映像内部又包含不同子类型的映像。下面是这三种映像类型的描述。

1. 可加载 (Loadable) 的映像类型

可加载型映像的执行需要通过引导代码把它装载到目标机 RAM 中，然后才开始执行。而引导代码分为两种：

- 引导代码固化在 ROM 或 Flash 中；
- 引导代码是一个独立的 VxWorks 应用。

引导代码通常也是一种 VxWorks 映像，有时称之为引导映像。它的设计目的就是把另一个包含应用的 VxWorks 映像装入到 RAM 中。引导映像在运行时，可能在 ROM/Flash 中执行（例如 ROM 驻留型引导映像）；也可能在 RAM 中执行。

2. 基于 ROM (ROM-based) 的映像类型

基于 ROM 型映像在执行前首先把自己从 ROM 或 Flash 中装载到 RAM 中。这种类型的映像通常在启动阶段比较慢，但在执行阶段比 ROM 驻留型映像要快。

3. ROM 驻留 (ROM-Resident) 的映像类型

ROM 驻留型映像在启动时把数据段拷贝到目标机 RAM 中。这种类型的映像启动阶段比较快，当目标机的 RAM 空间比较小的时候通常使用它。在嵌入式应用中通常会使用该类型的映像，然而，它在目标机上执行的速度比其他类型的映像要慢，原因是 CPU 访问 ROM 比访问 RAM 要慢。

194

- 非压缩型引导代码，这种类型的引导代码直接进行拷贝操作即可；
- 驻留型引导代码，这种类型的引导代码在拷贝时，仅仅拷贝它的数据段。

(2) VxWorks 映像的装入。引导代码执行后，把 VxWorks 映像装入到 RAM 中，然后跳转到 VxWorks 映像装入点。

(3) 系统初始化。这步操作由静态链接在 VxWorks 映像里的系统初始化代码来执行，最终完成系统初始化操作（例如操作系统初始化）。

2. 基于 ROM 的 VxWorks 映像启动顺序

基于 ROM 的 VxWorks 映像启动顺序如图 12.2 所示。

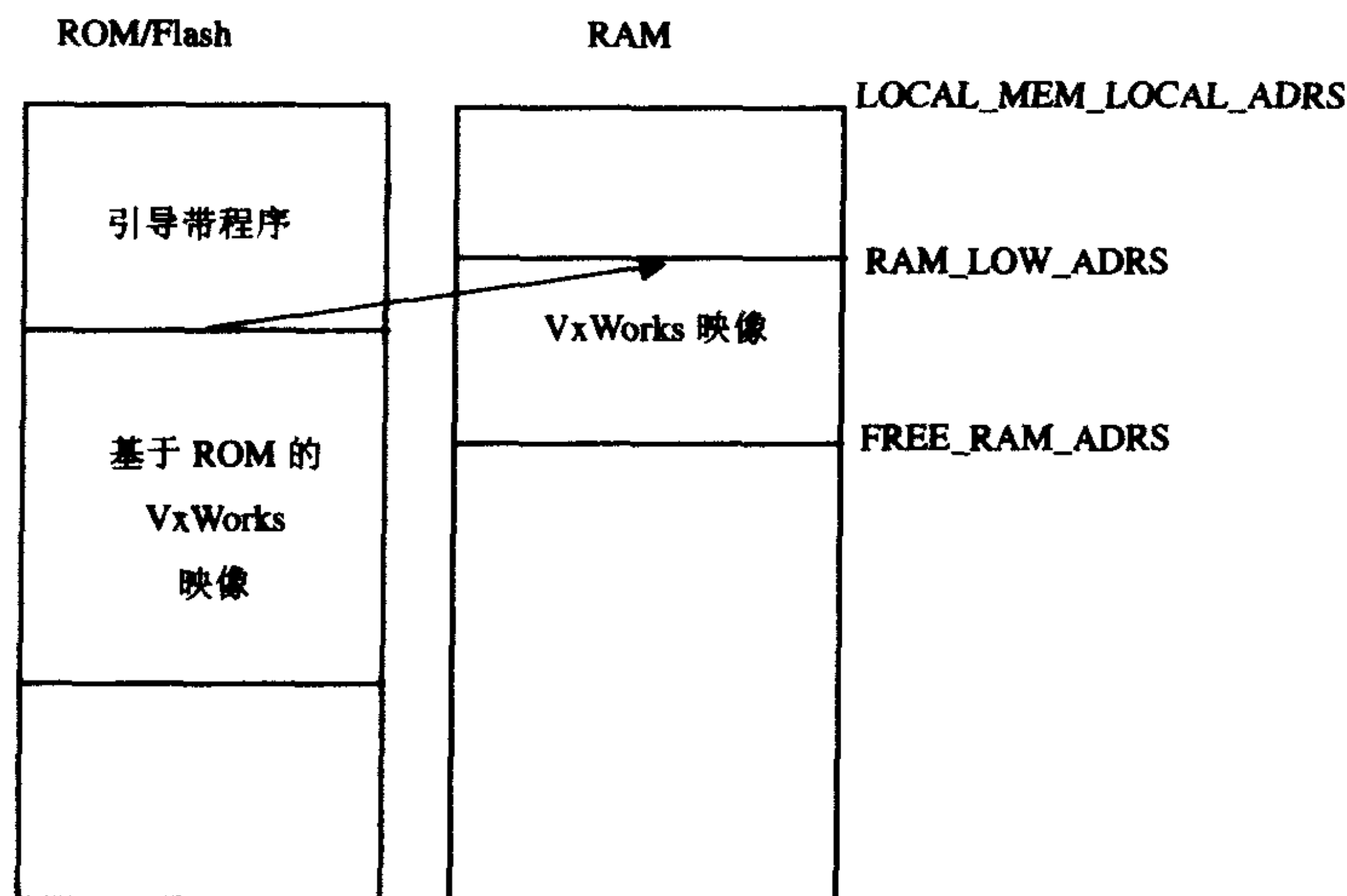


图 12.2 基于 ROM 的 VxWorks 映像

基于 ROM 的 VxWorks 映像启动详细过程如下。

(1) VxWorks 映像的装入。系统加电后引导带代码执行，首先把 VxWorks 映像的代码段和数据段从 ROM 或 Flash 里装入到 RAM 中，此时需要考虑下列情况：

- 压缩型 VxWorks 映像，这种类型的 VxWorks 映像 在拷贝时，需要对它执行解压操作；
- 非压缩型 VxWorks 映像，这种类型的 VxWorks 映像直接执行拷贝操作。

(2) 控制权转移。VxWorks 映像被装入后，系统控制权转移给 RAM 中 VxWorks 映像的初始化代码。

(3) 系统初始化。静态链接在 VxWorks 映像里的系统初始化代码在 RAM 中执行这步操作，最终完成系统初始化操作（例如操作系统初始化）。

3. ROM 驻留型 VxWorks 映像启动顺序

ROM 驻留型 VxWorks 映像启动顺序如图 12.3 所示。

ROM 驻留的 VxWorks 映像启动详细过程如下：

(1) VxWorks 映像的数据段装入。系统加电后引导带代码执行，首先把 VxWorks 映像的数据段

从 ROM 或 Flash 里装入到 RAM 中。

(2) 控制权转移。VxWorks 映像数据段被装入到 RAM 中后, 系统控制权转移给 ROM/Flash 里 VxWorks 映像初始化代码。

(3) 系统初始化。静态链接在 VxWorks 映像里的系统初始化代码在 ROM/Flash 中执行该操作, 最终完成系统初始化操作 (例如操作系统初始化)。

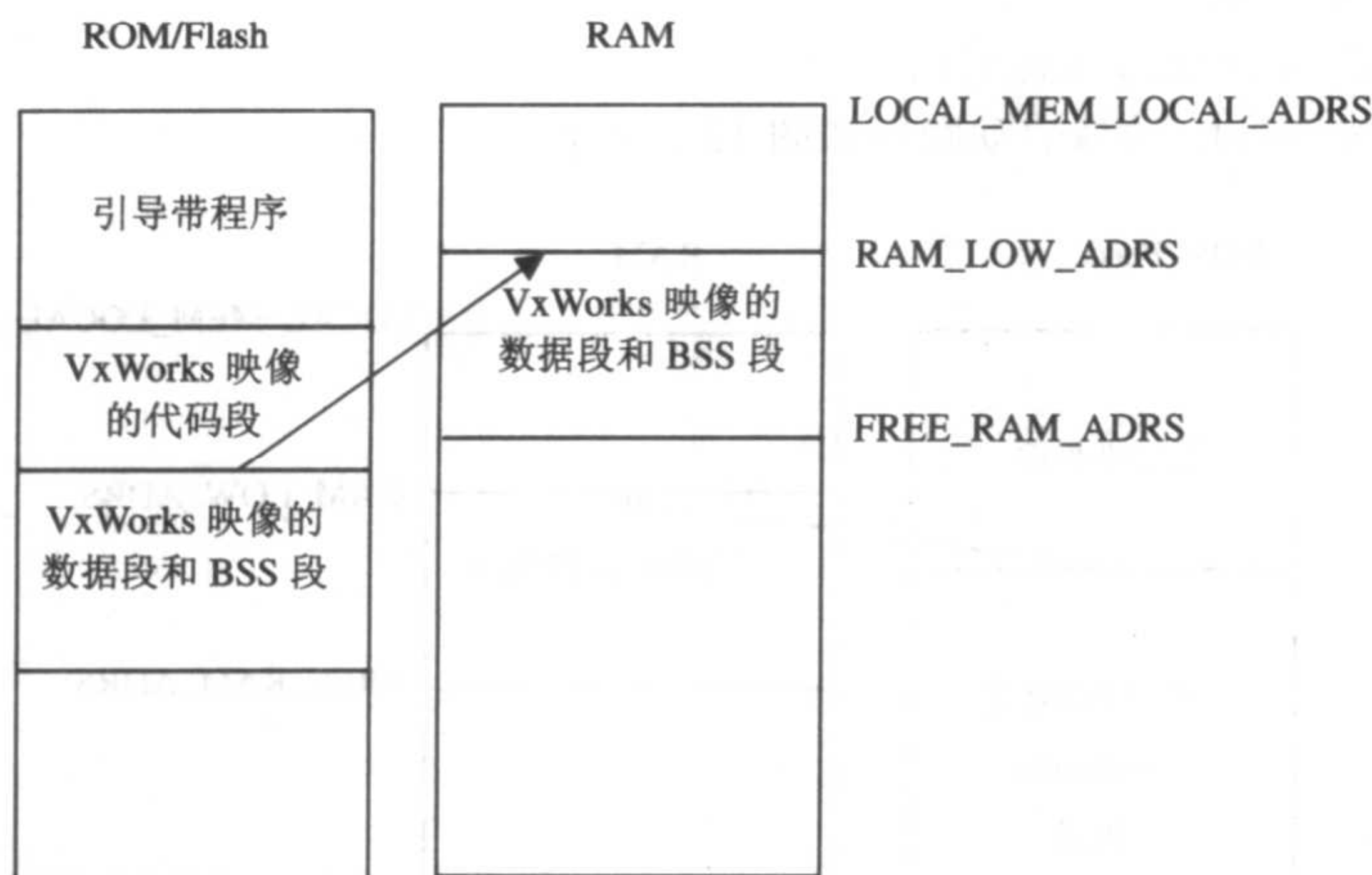


图 12.3 ROM 驻留型 VxWorks 映像

12.4 构造 VxWorks 映像

12.4.1 VxWorks 映像的构造

在 Tornado 2.0 开发环境下 VxWorks 映像的构造是通过 Build 菜单及 Build 属性窗口选项来辅助完成用户所需要构造的 VxWorks 映像的类型, 如图 12.4 和图 12.5 所示。

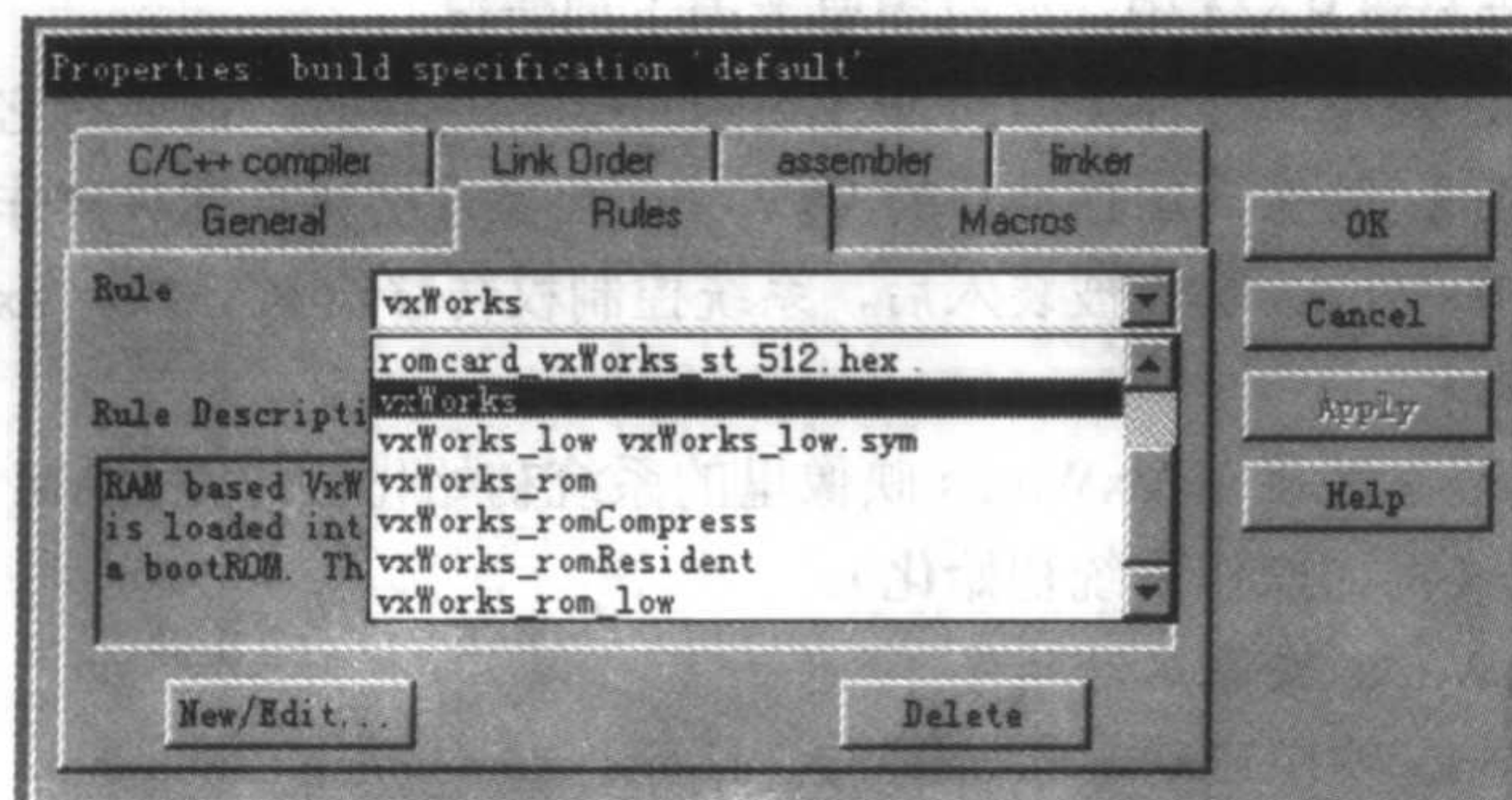


图 12.4 构造属性窗口

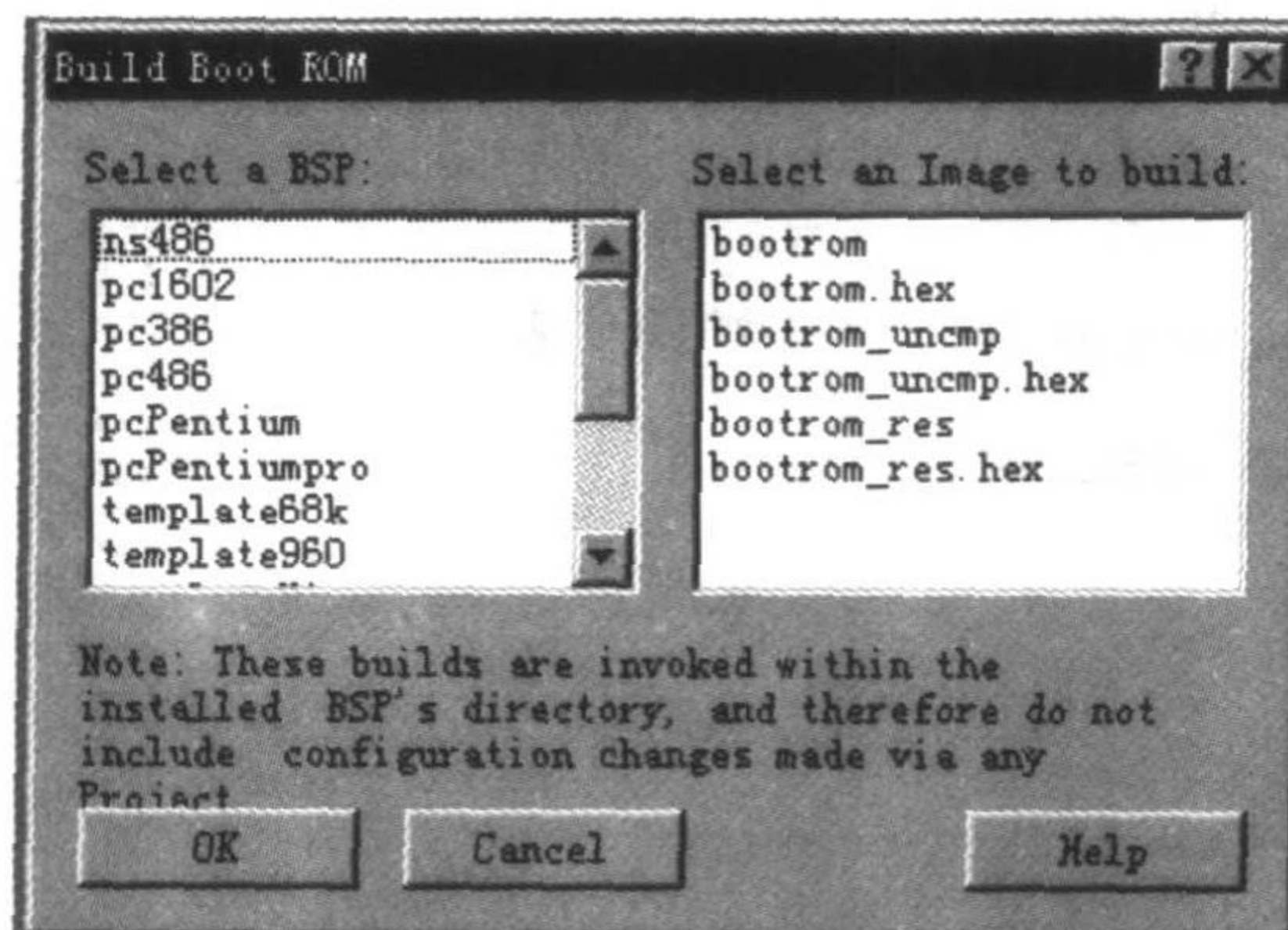


图 12.5 Build Boot ROM 窗口

而实际上，VxWorks 映像的构造最终靠 make 工具和构造控制文件来完成。其中有两个主要的构造控制文件。

1. makefile 文件

这个文件在../target/config/<bspname>目录下，类似 DOS 批处理文件。文件内容及结构类似下面的文件。

```
=====
makefile
=====
# Makefile generated by the project manager
#
# GENERATED: Mon Dec 01 11:11:14 中国标准时间 2003
# DO NOT EDIT - file is regenerated whenever the project changes

## core information (核心信息)

PRJ_DIR      = $(WIND_BASE)/target/proj/Project17
PRJ_FILE     = Project17.wpj
PRJ_TYPE     = vxWorks
PRJ_OBJS     = sysALib.o sysLib.o usrAppInit.o prjConfig.o linkSyms.o
BOOT_OBJS    = romInit.o romStart.o
BUILD_SPEC   = default
BSP_DIR      = $(WIND_BASE)/target/config/pcPentium
TGT_DIR      = $(WIND_BASE)/target

## set searching directories for dependencies (设置相关的搜寻目录)

vpath %.c $(BSP_DIR)
vpath %.cpp $(BSP_DIR)
vpath %.cxx $(BSP_DIR)

## build-configuration info (构造配置信息)

ifeq ($(BUILD_SPEC),default)
```



```

CPU            = PENTIUM
TOOL           = gnu
DEFAULT_RULE   = vxWorks
endif

## component-configuration info (模块配置信息)

COMPONENTS = INCLUDE_ANSI_ASSERT \
INCLUDE_ANSI_CTYPE \
...
INCLUDE_WDB_VIO_LIB
COMPONENT_LIBS =

include $(TGT_DIR)/h/make/defs.project

## build-configuration info (构造配置信息)

ifeq ($(BUILD_SPEC),default)
AR            = ar386
AS            = cc386
CC            = cc386
CFLAGS        = -g -mpentium -ansi -nostdinc -DRW_MULTI_THREAD -D_REENTRANT
-fvolatile -nostdlib -fno-builtin -fno-defer-pop -I$(PRJ_DIR) -I$(WIND_BASE)/target/
config/pcPentium -I$(WIND_BASE)/target/h -I$(WIND_BASE)/target/config/comps/src -I$
(WIND_BASE)/target/src/drv -DCPU=PENTIUM -DPRJ_BUILD -DCPU_VARIANT=PENTIUM
CFLAGS_AS     = -g -mpentium -ansi -nostdinc -fvolatile -nostdlib -fno-builtin -fno-
defer-pop -P -x assembler-with-cpp -I$(PRJ_DIR) -I$(WIND_BASE)/target/config/pcPentium
-I$(WIND_BASE)/target/h -I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/
src/drv -DCPU=PENTIUM -DPRJ_BUILD -DCPU_VARIANT=PENTIUM
CPP           = cc386 -B$(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib/ -E -P -xc
EXTRA_MODULES =
LD            = ld386
LDDEPS        =
LD_FLAGS      = -X -N
LD_PARTIAL    = cc386 -B$(WIND_BASE)/host/$(WIND_HOST_TYPE)/lib/gcc-lib/ -nostdlib
-r -Wl, -X
LD_PARTIAL_FLAGS = -X -r
LIBS          = $(WIND_BASE)/target/lib/libPENTIUMgnuvx.a
NM            = nm386 -g
OPTION_DEFINE_MACRO = -D
OPTION_INCLUDE_DIR = -I
POST_BUILD_RULE =
RAM_HIGH_ADRS = 00008000      # 引导映像入口点
RAM_LOW_ADRS  = 00108000      # VxWorks 映像入口点
SIZE          = size386
endif

# override make definitions only above this line (包含 make 定义文件)

include $(TGT_DIR)/h/make/rules.project

## build-configuration info (构造配置信息)

```



```

ifeq ($(BUILD_SPEC),default)

sysALib.o:
$(AS) -g -mpentium -ansi -nostdinc -fvolatile -nostdlib -fno-builtin -fno-defer-pop
-P -x assembler-with-cpp -I$(PRJ_DIR) -I$(WIND_BASE)/target/config/pcPentium -I$
(WIND_BASE)/target/h -I$(WIND_BASE)/target/config/comps/src -I$(WIND_BASE)/target/src
/drv -DCPU=PENTIUM -DPRJ_BUILD -DCPU_VARIANT=PENTIUM -c $(WIND_BASE)/target/config/
pcPentium/sysALib.s -o sysALib.o
...
endif
## dependencies (从属关系)

sysALib.o: $(WIND_BASE)/target/config/pcPentium/sysALib.s \
$(PRJ_DIR)/prjComps.h \
$(PRJ_DIR)/prjParams.h
...
linkSyms.o: $(PRJ_DIR)/linkSyms.c

## user defined rules (用户定义规格)

bootrom_high :
$(CC) -c $(CFLAGS) -DRAM_DST_ADRS=0x$(RAM_LOW_ADRS) $(BOOTINIT)
- @ $(RM) $(WIND_BASE)/target/config/pcPentium/bootrom.Z.s $@
$(MAKE) -f $(PRJ_DIR)/Makefile LD_HIGH_FLAGS="-Ttext $(RAM_LOW_ADRS)" $(WIND_BASE)/
target/config/pcPentium/bootrom.Z.s
$(MAKE) -f $(PRJ_DIR)/Makefile bootrom LD_LOW_FLAGS="$(LD_HIGH_FLAGS)"
$(CP) bootrom $@
- @ $(RM) bootrom.Z.* bootrom bootInit.o
...
-----

```

2. rules.bsp 文件

这个文件在../target/h/make 目录下，通过它构造指定的 VxWorks 映像。文件内容及结构类似下面的文件。

```

=====
rules.bsp
=====
# THIS FILE IS GOING AWAY.
# PLEASE MERGE ANY CHANGES INTO rules.project
#
# rules.bsp - makefile target rules for BSP (BSP 的 makefile 目标规则)
#
...
# DESCRIPTION      (描述)
# 这个文件包含构造目标文件所使用的规则。
#
# VARIABLES      (参数说明)
# 下面的变量必须被定义:
#
#   ROM_SIZE      config.h 中定义的 ROM 空间大小
#   LOCAL_MEM_SIZE config.h 中定义的目标机 RAM 空间大小
#   BINHEX        binary->hex 文件转换程序
#   HEX_FLAGS     BINHEX 程序选项

```



```

# LD_LOW_FLAGS      在低内存区链接代码选项
# LD_HIGH_FLAGS     在高内存区链接代码选项
# LD_PARTIAL_FLAGS  链接所有对象代码的链接标志
# LDDEPS            主机必须预先具备的链接条件
# LDOUT_CONV        格式转换等工具
# COMPILE_SYMTBL    符号表隐含标志
# ROM_ENTRY         ROM 入口点(典型为 &romInit)
# SYS_ENTRY         RAM 型映像系统入口点(典型为 &sysInit)
# USR_ENTRY         压缩系统入口点(典型为 &usrInit)
#
# 下面的变量为可选定义:
#
# LDFLAGS           vxWorks & ROM 标准链接标志
# ROM_LDFLAGS       ROM 型映像的特殊链接标志
# MACH_EXTRA        所有引导/vxWorks 映像 s 的扩展模块
# BOOT_EXTRA        压缩型映像的扩展模块
# RELEASE           映像发布
#
# 常量:
# CPU               cpu 类型
# */
#####
# 构造目标文件
## make will default to "exe"
exe : vxWorks
release : $(RELEASE) bsp2prj
bsp2prj:
$(TCL) $(WIND_BASE)/host/src/hutils/bsp2prj.tcl
clean :
- $(RM) *.o
- $(RM) *.rpo
- $(RM) vxWorks*
- $(RM) bootrom*
- $(RM) ctdt.c
- $(RM) symTbl.c
- $(RM) depend.$(TARGET_DIR)

ifeq ($(IMPLICIT_RULE_S_O),)
.s.o :
@ $(RM) $@
$(CC) $(CFLAGS_AS) -c -o $@ $<
endif

ifeq ($(IMPLICIT_RULE_C_O),)
.c.o :
@ $(RM) $@
$(CC) $(CFLAGS) -c $<
endif

ifeq ($(IMPLICIT_RULE_CXX_O),)
# C++ 编译规则
.cpp.o :
@ $(RM) $@

```



```

$(CXX) $(C++FLAGS) -c $<
...
.cpp.out :
@ $(RM) $@
...
endif

#####
# 与目标机相关的对象

romInit_res.o: depend.$(BSP_NAME) romInit.s romInit.o
- @ $(RM) $@
$(CC) $(CFLAGS_AS) -DROM_RESIDENT -c -o $@ romInit.s
...
##### vxWorks #####
# vxWorks      -标准的 vxWorks 系统
# vxWorks.sym - vxWorks 符号表

vxWorks vxWorks.sym : depend.$(BSP_NAME) usrConfig.o dataSegPad.o \
    $(MACH_DEP) $(LDDEPS) $(LIBS)
    $(BINXSYM) < vxWorks > vxWorks.sym
...
$(LDOUT_SYMS) vxWorks.sym
    $(VXSIZEPROG) -v $(RAM_HIGH_ADRS) $(RAM_LOW_ADRS) vxWorks

# vxWorks_rom - ROM 型 VxWorks (不包含符号表)
# 这是一个典型的 ROM 型映像,它不包括 shell 和符号表
vxWorks_rom : depend.$(BSP_NAME) bootInit_uncmp.o romInit.o \
    dataSegPad.o usrConfig.o $(MACH_DEP) $(LDDEPS) $(LIBS)
...
##### bootrom #####
# bootrom      - 压缩 ROM 型引导对象模块
# bootrom.hex   - 压缩十六进制格式的 bootrom
# bootrom_uncmp - 非压缩 ROM 型引导对象模块
# bootrom_uncmp.hex - 非压缩十六进制格式的 bootrom
#
# Bootroms 通常不包含静态 C++ 对象
...
bootrom_uncmp : depend.$(BSP_NAME) bootInit_uncmp.o romInit.o \
    bootConfig.o $(MACH_DEP) $(LDDEPS) $(LIBS)
...
#####
# headerMacros - 所有宏输出变量
ifeq ($(CPP_PRINT_MACROS),)
CPP_PRINT_MACROS = $(CPP) -dM -E -pipe $(CC_INCLUDE) $(CC_DEFINES)
endif

headerMacros:
@ $(CPP_PRINT_MACROS) sysLib.c

#####
# makeMacros - 输出 makefile 宏相关联的工具链设置

```



```
makeMacros:
$(NOP) CPU = $(CPU)
...
$(NOP) MACH_DEP = $(MACH_DEP)
ifneq ($(DEPEND_GEN_UTIL),)
$(NOP) DEPEND_GEN_UTIL = $(DEPEND_GEN_UTIL)
endif
-----
```



注意

可以通过命令行方式构造 VxWorks 映像，可以如下操作：

- (1) 在..\Tornado\host\x86-win32\bin 目录下执行 torVars.bat 批处理；
- (2) 在构造工程的工程目录下执行 make 命令，例如 make vxWorks。

12.4.2 典型的 VxWorks 映像

VxWorks 构造规则将创建下面列出的一些典型 VxWorks 映像。

1. vxWorks

这是基于 RAM 的映像，需要通过一个 VxWorks 引导 ROM 把它下载到目标机内存中才能执行。在开发环境中这是一个默认的选项，主要在调试阶段使用。使用宿主机上的 WindSh 工具和符号表 (vxWorks.sym)。

2. vxWorks.st

这也是基于 RAM 的映像，需要通过一个 VxWorks 引导 ROM 把它下载到目标机内存中才能执行。但该对象文件内置符号表。

3. vxWorks_rom

这是非压缩、基于 ROM 的映像。在这个对象文件执行前把自己拷贝到目标机 RAM 中。这种类型的映像通常在启动阶段比较慢，但在执行阶段比 ROM 驻留型的映像要快。

4. vxWorks.st_rom

这是基于 ROM 压缩的 vxWorks.st 映像。它在执行前把自己解压并拷贝到目标机 RAM 中执行。

5. vxWorks.res_rom

这是 ROM 驻留的非压缩 vxWorks.st 的映像。它在执行前把自己数据段拷贝到目标机 RAM 中。

6. vxWorks.res_rom_nosym

vxWorks 的 ROM 驻留型映像，仅在启动时把数据段拷贝到目标机 RAM 中。这种类型的映像启动阶段比较快，通常在 RAM 空间比较小的目标机上使用这种类型的映像。它是嵌入式系统中的典型应用，然而它在目标机上执行的速度比其他 ROM 类型的映像要慢，原因是 CPU 访问 ROM 比访问 RAM 要慢。

7. bootrom

基于 ROM 压缩型的引导映像。

8. bootrom_uncmp

基于 ROM 非压缩型的引导映像。

9. bootrom_res

ROM 驻留型的引导映像。可以“烧”到 ROM 或 flash 中，目标机加电后直接执行该映像。

另外，可以构造以“.hex”结尾的 ROM 型 VxWorks 映像，表示该对象文件为 Motorola S-Record 格式。其中所有 ROM 型 VxWorks 映像可以被固化到 PROM 或 Flash 中。

小 结

BSP 的启动流程归根到底是 VxWorks 映像的启动流程。无论哪种类型的映像，启动阶段所执行的处理在逻辑上是一样的。这一章通过文字加图片的形式对各种类型映像的启动情况进行了描述，其中还包括对常用映像的介绍等内容。

第 13 章 VxWorks 预内核初始化

预内核的初始化主要是指 CPU 的初始化、向量表的安装、全局描述表的安装、目标机上硬件设备的初始化、操作系统的初始化等操作。不同类型的 VxWorks 映像其初始化流程有所不同，在这一章中主要对 ROM 型和 RAM 型映像的预内核的初始化进行了详细描述。

13.1 预内核的初始化概述

13.1.1 下载型 VxWorks 映像预内核的初始化顺序

1. 初始化简述

目标机系统加电启动后，首先执行 VxWorks 的引导程序，由引导程序把 VxWorks 映像装载到目标机的 RAM 中。当把 VxWorks 映像装入到 RAM 中后，处理器跳转到映像装入地址处。接下来首先执行 `sysInit()` 函数，然后执行 `usrInit()` 函数，由 `usrInit()` 函数彻底完成预内核的初始化。整个初始化过程如图 13.1 所示。

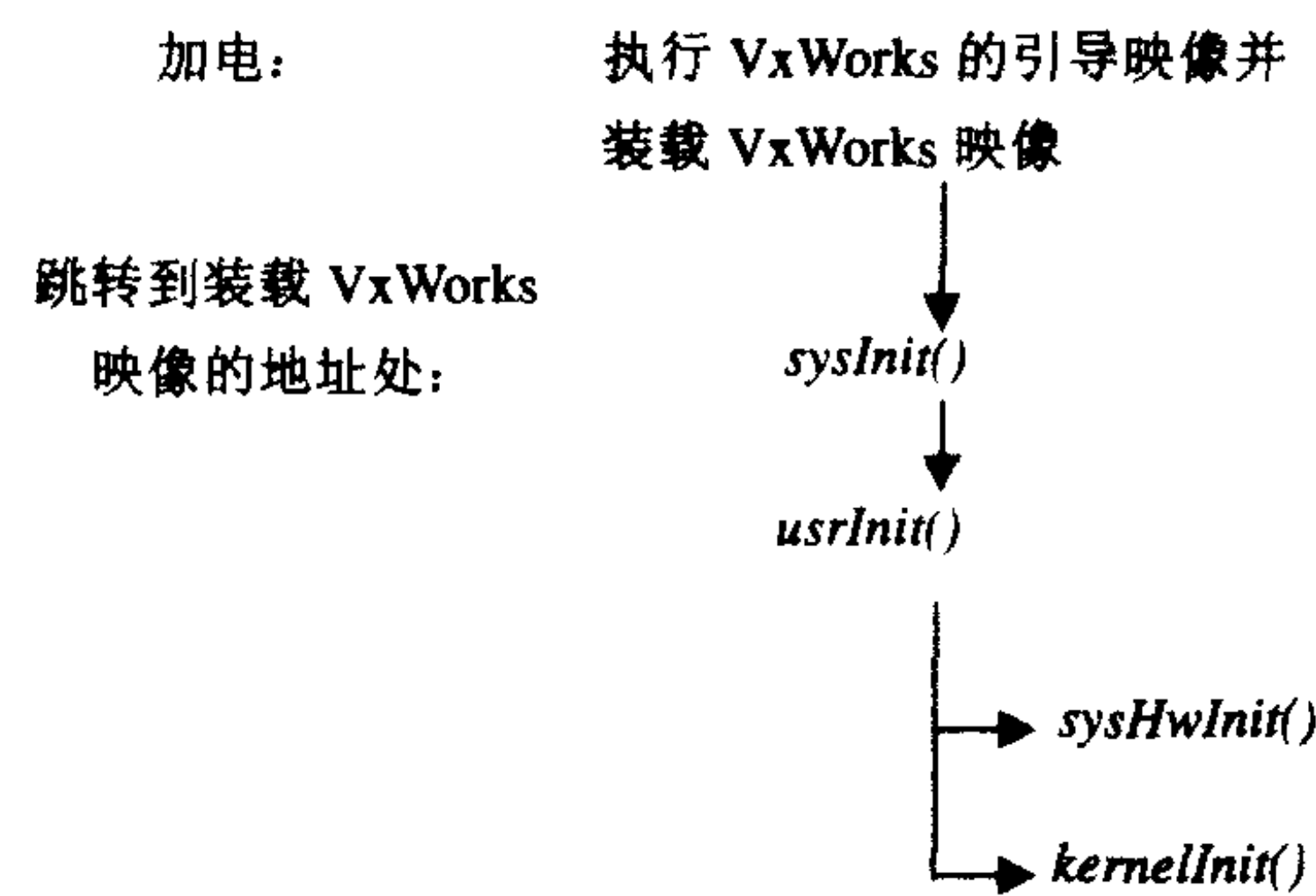


图 13.1 装载型映像预内核的初始化

2. 初始化详细过程

初始化详细流程如表 13-1 所示。

13.1.2 ROM 型 VxWorks 映像预内核的初始化顺序

1. 初始化简述

目标机系统加电启动后，首先执行 `romInit()` 函数其功能主要是对 CPU 进行初始化，然后执行 `romstart()` 函数进行映像段的拷贝及解压工作，接下来执行 `usrInit()` 函数，由 `usrInit()` 函数彻底完成预

内核的初始化。整个初始化过程如图 13.2 所示。

表 13-1 装载型 VxWorks 系统初始化详细流程

函 数	函 数 功 能	所在文件
sysInit()	(a) 锁住中断 (b) 禁用缓冲 (c) 用缺省值初始化系统中断表 (仅 i960) (d) 用缺省值初始化系统错误表 (仅 i960) (e) 初始化处理器寄存器到一缺省值 (f) 使回溯失效 (g) 清除所有悬置中断 (h) 激活 usrInit(), 指明启动类型	sysALib.s
UsrInit()	(a) 对 bss 赋零 (b) 把 bootType 值赋给 sysStartType (c) 调用 excVecInit(), 初始化所有系统和缺省中断向量 (d) 调用 sysHwInit(), 初始化与目标板相关的硬件 (e) 调用 usrKernelInit(), 配置 VxWork 内核 (f) 调用 kernelInit(), 初始化并启动内核	usrConfig.c
UsrKernelInit()	如果系统配置了下面的模块, 则依次调用 classLibInit(), taskLibInit(), taskHookInit(), semBLibInit(), semMLibInit(), semCLibInit(), semOLibInit(), wdLibInit(), msgQLibInit(), qlnit(), workQInit()	usrKernel.c
KernelInit()	初始化并启动内核 (a) 激活 intLockLevelSet() (b) 从内存池顶部创建根堆栈和 TCB (c) 调用 taskInit(), taskActivate(), 用于 usrRoot() (d) 调用 usrRoot()	kernelLib.c
UsrRoot()	初始化 I/O 系统, 安装设备驱动程序, 创建设备 (在 configAll.h 和 config.h 中指定) 等 (a) 调用 sysClkConnect(), sysClkRateSet(), iosInit() 安装系统时钟驱动程序 (b) 调用 ttyDrv() (可选项) 安装控制台驱动程序 (c) 调用 excInit(), logInit(), sigInit() 初始化异常管理 (d) 调用 pipeDrv() 初始化管道驱动程序 (e) 调用 stdioInit() 初始化标准的 I/O 库 (f) 调用 mathSoftInit() 或 mathHardInit() 支持浮点运算 (g) 调用 wdbConfig() 配置并初始化目标代理机 (h) 如果系统配置了 target shell, 则启动它 (i) 启动用户应用程序 USER_APPL_INIT	usrConfig.c

2. 初始化详细流程

ROM 型 VxWorks 系统初始化详细流程如表 13-2 所示。

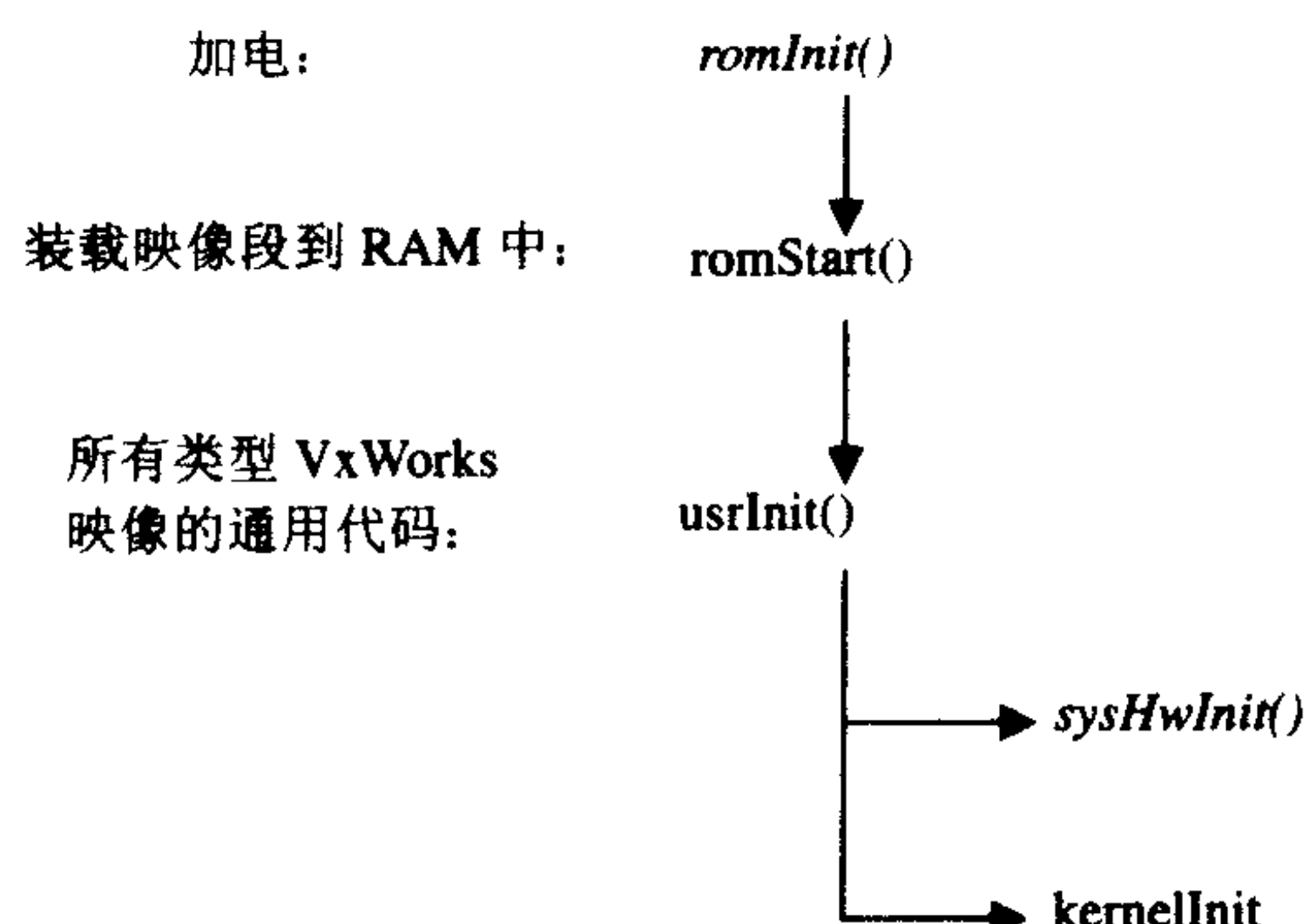


图 13.2 ROM 型映像预内核的初始化

表 13-2 ROM 型 VxWorks 系统初始化详细流程

函 数	函 数 功 能	所在文件
<code>romInit()</code>	(a) 禁止中断 (b) 保存启动类型 (冷/热启动) (c) 硬件相关的初始化 (d) 调用 <code>romStart()</code>	<code>romInit.s</code>
<code>romStart()</code>	(a) 将数据段从 ROM 拷贝到 RAM, 清内存 (b) 将代码段从 ROM 拷贝到 RAM, 有必要的话需解压缩 (c) 调用 <code>usrInit()</code>	<code>bootInit.c</code>
<code>usrInit()</code>	系统初始化程序	<code>usrConfig.c</code>
<code>usrKernelInit()</code>	配置内核数据结构	<code>usrKernel.c</code>
<code>kernelInit()</code>	初始化并启动内核	<code>kernelLib.c</code>
<code>usrRoot()</code>	初始化 I/O 系统, 安装设备驱动程序, 创建设备等, 最后启动用户应用程序	
<code>usrAppInit()</code>	应用程序入口函数	<code>usrAppInit.c</code>

13.2 预内核的特殊初始化函数

VxWorks 映像典型特殊函数如下:

- `romInit()` 函数
- `romStart()` 函数
- `sysInit()` 函数

`romInit()` 和 `romStart()` 函数是所有固化在 ROM/Flash 里 VxWorks 映像所必须执行的函数, 而 `sysInit()` 函数仅仅只是所有装载型 VxWorks 映像执行的函数。实际上 `romInit()` 和 `sysInit()` 是两个非常相似的函数, 有一点值得注意的是 `romInit()` 函数初始化目标机内存, 而 `sysInit()` 函数不进行这步

操作。

上述这些函数将在下面进行详细描述。

13.2.1 romInit()函数

romInit()函数是所有固化在 ROM/Flash 里 VxWorks 映像首先执行的函数,对该函数主要从如下几个方面进行描述。

1. romInit()函数的功能简述

romInit()函数包含在 romInit.s 文件中,且用汇编语言编写。它是系统加电后首先执行的代码,也是所有 VxWorks ROM 型映像的入口点。它执行目标机最小的初始化操作及调用 romStart()函数,硬件初始化操作推迟到预内核通用初始化代码中进行。romInit()函数必须包含下列功能:

- 屏蔽处理器中断及处理器复位;
- 初始化内存系统;
- 初始化堆栈指针和其他寄存器,开始执行 romStart()函数及传递启动类型。

2. 目标机体系结构和 BSP 特殊问题

romInit()函数所需要进行的处理大部分由处理器指定,而这些处理可以从参考 BSP 中拷贝获得。这些处理包括:

- (1) 屏蔽处理器中断。
- (2) 初始化处理器高速缓存。
- (3) 初始化堆栈指针。

至于非处理器指定的初始化包括 DRAM 和将需要指定的硬件环境,如下所示:

- 等待状态;
- 刷新频率;
- 芯片选择(总线桥/总线/存储器控制器等);
- 屏蔽二级高速缓存(如果需要)。

3. 冷启动和热启动

VxWorks 映像启动类型分冷启动和热启动两种:

(1) 冷启动。所谓冷启动是指硬件环境通过加电启动。在 romInit()函数中需要保存系统启动类型,启动类型的宏定义为 BOOT_COLD。

(2) 热启动。所谓热启动是通过调用 reboot()、Ctrl+X 或异常中断重新启动目标机系统。实际上这些操作是把控制权传递给 ROM 中的监控函数——sysToMonitor(),这个函数包含在 sysLib.c 文件中。如执行 sysToMonitor(2),则系统执行热启动。



注意

启动类型必须保存在一个寄存器中,并传递给 romStart()函数。例如

```
movl    $ BOOT_COLD,%ebx    /* %ebx 存放启动类型 */
```

4. 堆栈指针的初始化

STACK_ADRS 宏定义配置了堆栈开始地址,该宏定义包含在 configAll.h 文件中。

(1) ROM 驻留型映像堆栈开始地址。

- 堆栈向下增长的处理器，其堆栈开始地址定义如下：

```
#define STACK_ADRS STACK_RESIDENT
```

- 堆栈向上增长的处理器，其堆栈开始地址定义如下：

```
#define STACK_ADRS (STACK_RESIDENT-STACK_SAVE)
```

(2) 非 ROM 驻留型映像堆栈开始地址。

- 堆栈向下增长的处理器，其堆栈开始地址定义如下：

```
#define STACK_ADRS _romInit
```

- 堆栈向上增长的处理器，其堆栈开始地址定义如下：

```
#define STACK_ADRS (_romInit-STACK_SAVE)
```

(3) configAll.h 文件中堆栈部分代码如下：

```
/* ROM 驻留型映像常量*/
#if (CPU_FAMILY==I960)
#define STACK_SAVE 512 /* 预留堆栈最大尺寸 */
#else
/* sparc 或其他 */
#if ((CPU_FAMILY==SPARC) || (CPU_FAMILY==PPC))
#define STACK_SAVE 0x1000
#else
/* 所有其他的体系结构 */
#define STACK_SAVE 0x40 /* 预留堆栈最大尺寸 */
#endif
/* mips cpp 非 elif */
#endif
...
#if ((CPU_FAMILY == MIPS) || (CPU_FAMILY == PPC))
#define STACK_RESIDENT RAM_DST_ADRS
#else
#define STACK_RESIDENT _sdata
#endif

#if (_STACK_DIR == _STACK_GROWS_DOWN)

#ifdef ROM_RESIDENT
#define STACK_ADRS STACK_RESIDENT
#else
#define STACK_ADRS _romInit
#endif /* ROM_RESIDENT (ROM 驻留型) */

#else /* _STACK_DIR == _STACK_GROWS_UP */

#ifdef ROM_RESIDENT
#define STACK_ADRS (STACK_RESIDENT-STACK_SAVE)
#else
#define STACK_ADRS (_romInit-STACK_SAVE)
#endif /* ROM_RESIDENT (ROM 驻留型) */

#endif /* _STACK_DIR == _STACK_GROWS_UP */
-----
```


5. romInit()函数示例

下面是一个 x86 体系结构的 romInit()函数模板。

```

=====
romInit.s
=====
/* romInit.s - x86 ROM 初始化模块 */

/* 版权 1984-1997 Wind River Systems, Inc. */
.data
.globl  _copyright_wind_river
.long   _copyright_wind_river
/*
修改记录
-----
(每次修改文件需要在该位置添加修改记录, 修订版本也随之增加)
01b,27aug97,dar  code review comments, added x86 reset notes.
01a,27jan97,dar  written (from pc386/romInit.s, ver 011).
*/
/*

```

描述:

这个模块包含从 ROM 启动运行的 VxWorks 映像的入口代码, 例如 “bootrom” 和 “VxWorks_rom” 映像。

入口点 romInit()函数, 它是系统加电后首先执行的代码。同时它也设置了启动类型参数 BOOT_COLD, 该参数将被传递给通用的 romStart()函数以及执行最小的必要安装操作以便调用通用 C 语言函数 romStart()。

RomInit()函数的典型操作是屏蔽处理器中断, 初始化 CPU; 设置堆栈指针的初始值 STACK_ADRS (在 configAll.h 文件中定义了该常量) 和初始化其他寄存器以便能够执行 C 代码。如果有必要, 通过配置 DRAM 控制器准备好系统内存。其他大部分硬件初始化操作将在 sysLib.c 文件中的 sysHwInit()函数里进行。

romInit.s 里的另一个入口点是 romInitWarm()。由文件 sysLib.c 中的 sysToMonitor()函数调用 romInitWarm()函数, 执行热启动操作。

热启动入口点必须作为一个参数存放在堆栈里, 并传递给 romStart()函数。对于 x86 处理器这种做法不需要, 系统可以通过调用函数 sysReboot()强制处理器重新启动。



警告

romInit()函数必须设计成与地址无关的代码 (PIC, Position Independent Code)。

它需要支持综合的 VxWorks 引导策略, 意味着该函数不应当包含任何绝对地址。如果必须使用一个绝对地址, 必须通过 ROM_ADRS(x)宏对其重新计算。由这个宏把绝对地址转换成 ROM 空间中适当的引用地址。其计算公式是:

$$\text{ROM\_ADRS}(x) = ((x) - \_romInit + \text{ROM\_TEXT\_ADRS})$$

另外, 其他模块或函数不应当调用 romInit.s 中的代码, 特别是 sysLib.o 和 sysALib.o 模块。如果需要一外部模块, 则把这个模块附加到 makefile 文件里的变量 BOOT_EXTRA 中即可, 这样这个模块将被链接到系统里。如果同一个模块被其他 BSP 代码所引用, 然而这个模块必须被附加到 MACH_EXTRA 中。值得注意的是, 一些 C 编译器产生带有绝对地址的代码。这样的代码不应该从这个模块中调用。如果绝对地址不可避免, 仅仅只是从该模块中产生 ROM 驻留代码。如果绝对地址没有经过宏 ROM_ADRS 对其进行处理, 则压缩型和未压缩型的 bootrom 或 VxWorks

映像将不会工作。



警告

一个普遍的错误是 BSP 开发人员想在 `romInit.s` 里进行过多的初始化操作。这个文件不是硬件初始化的主要程序，仅仅只是做一些必要的初始化以便获得内存功能。设备初始化的大部分操作被推迟到 `sysLib.c` 文件中的 `sysHwInit()` 函数中进行。

与其他实时操作系统不一样，VxWorks 不是使用单直线性的设备初始化，而是分阶段进行。对于缺乏经验的 BSP 开发者来说，最普遍的做法是从其他实时操作系统下获得一个 BSP，提取用汇编语言编写的硬件安装代码并粘贴到这个文件中。由于 VxWorks 提供 3 种不同内存配置：压缩型、非压缩型和 ROM 驻留型，采用上述策略通常不会成功工作。



警告

另一个普遍的错误是 BSP 开发人员认为在 `romInit.s` 里初始化过的设备不需要在 `sysALib.s` 或 `sysLib.c` 重新初始化。实际上 VxWorks 映像并假设引导映像能进行一些处理，只认为 VxWorks bootrom 程序仅仅只是引导 VxWorks 映像。因此，它必须重新设置和重新初始化所有它自己需要使用的设备。一般情况下，引导映像主要被用于 VxWorks 映像的调试和开发阶段。当 VxWorks 映像开发结束后，VxWorks bootrom 也就完成了它的使命。

x86 CPU 复位注意事项：

80x86 在实模式下冷启动。在这种模式下，80486 控制 20 根地址线，A19 到 A0 (1MB)。

80486 CS:EIP 包含 `f000:0000ffff000` 之前的初始化，这是一个线性地址 `0xfffff0`。重新设置前，处理器指令指示器指向 1MB 的最后 16 个字节。

然而，重新设置后 80486 处理器实际上拥有物理地址线，A31 到 A20，不过需要执行一条高级别的跳转指令。这是因为段描述高速缓存寄存器中代码段的基地址在处理器重新设置为 `0xfffff0000` 后，拥有 A31 到 A20 有效逻辑的 `0xfffff0000` 将一切应用于本地总线。这意味着处理器复位向量 `0xfffffffff0` 是物理（总线）地址。

重新设置处理器后，处理器将使用 32 位地址空间（4GB）的最高 64KB 空间中的地址，即从 `0xfffff0000` 到 `0xfffffffffff`，直到使用段间跳转为止。

另一个值得注意的是既然物理复位向量是 `0xfffffffff0`，物理地址空间的最后 16 个字节的第一条指令需要是一个跳转到 `0xfffff0000` 上方任何点的跳转指令。

```
*/
```

```
#define _ASMLANGUAGE
#include "vxWorks.h"
#include "sysLib.h"
#include "asm.h"
#include "config.h"
```

```
/* 转换绝对地址到 ROM 相等的地址 */
```



```

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

/* 段内指令 */

.globl  _romInit      /* 系统代码的起始点 */
.globl  _sdata        /* 数据段的起始点 */

_sdata:
.asciz  "start of data"

.text
.align 4

/*****
*
* romInit - ROM 中 VxWorks 的入口点
*
* romInit (startType)
*   int startType;   /* 启动类型 */
*/

/*
* 注意: 在 0xffffffff0 处的代码做一个跳转跳到这里(_romInit)
*
* 这是一个 16 位实模式代码, 它必须在同一个 64KB 空间里被当作复位代码 (例如一
* 个 jump 指令)
*/

_romInit:
/* 下面的指令在 16 位模式下执行 */

cli          /* 关中断 */
jmp cold

.align 4,0x90
_romWarmHigh:
cli          /* 关中断 */
movl  SP_ARG1(%esp),%ebx /* %ebx 包含启动类型——startType */
jmp warm

.align 4,0x90
_romWarmLow:
cli          /* 关中断 */
cld          /* 拷贝自己到入口点 */
movl $ RAM_LOW_ADRS,%esi
movl $ ROM_TEXT_ADRS,%edi
movl $ ROM_SIZE,%ecx
shrl $2,%ecx
rep
movsl
movl  SP_ARG1(%esp),%ebx /* %ebx 包含启动类型——startType */
jmp warm

/* 版权横幅发布在 ROM 开始处(代码段) */

```



```
.ascii "Copyright 1984-1997 Wind River Systems, Inc."
.align 4

/* 装载临时 IDT */
cold:
    aword          /* 下一个指令是 32 位地址 */
    word           /* 下一个指令是 32 位操作 */
    lidt %cs:ROM_IDTR

/* 装载临时 GDT */

    aword          /* 下一个指令是 32 位地址 */
    word           /* 下一个指令是 32 位操作 */
    lgdt %cs:ROM_GDTR

/* 切换到保护模式 */

    mov %cr0,%eax  /* 把 cr0 搬移到 eax */
    .byte 0x66      /* 下一个指令是 32 位操作 */
    or $0x00000001,%eax /* 设置 PE 位 */
    mov %eax,%cr0   /* 把 eax 搬移到 cr0 */
    jmp romInit1 /* 执行 jump 指令刷新指令队列 */

romInit1:
    .byte 0x66      /* 下一个指令是 32 位操作 */
    mov $0x0010,%eax /* 选择器 0x0010 是第三个描述符 */
    mov %ax,%ds      /* 装载到 DS */
    mov %ax,%es      /* 装载到 ES */
    mov %ax,%fs      /* 装载到 FS */
    mov %ax,%gs      /* 装载到 GS */
    mov %ax,%ss      /* 装载到 SS */

    .byte 0x66      /* 下一个指令是 32 位操作 */
    mov $ ROM_STACK,%esp /* 设置堆栈指针 */

    aword          /* 下一个指令是 32 位地址 */
    word           /* 下一个指令是 32 位操作 */

/*
 * 注意：这个跳转必须是段间跳转指令(例如 'far' jump)
 * 为了触发 A31-A20 地址线
 */

    ljmp $0x08, $ ROM_TEXT_ADRS + ROM_INIT2 /* 段间跳转 */

_romIdtr:
    .word 0x0000      /* 大小: 0 */
    .long 0x00000000 /* 地址: 0 */

_romGdtr:
    .word 0x0027      /* 大小 : 39(8 * 5 - 1) 字节 */
    .long ROM_TEXT_ADRS + ROM_GDT /* 地址: romGdt */
```



```

.align 4,0x90
_romGdt:
/* 0(selector=0x0000): 空描述符 */
.word    0x0000
.word    0x0000
.byte    0x00
.byte    0x00
.byte    0x00
.byte    0x00

/* 1(selector=0x0008): 代码描述符 */
.word    0xffff          /* 界限: 0xffff */
.word    0x0000          /* 基数: xxxx0000 */
.byte    0x00            /* 基数: xx00xxxx */
.byte    0x9a            /* 代码 e/r, Present, DPL0 */
.byte    0xcf            /* 界限: fxxxx, Page Gra, 32 位 */
.byte    0x00            /* 基数: 00xxxxxx */

/* 2(selector=0x0010): 数据描述符 */
.word    0xffff          /* 界限: 0xffff */
.word    0x0000          /* 基数: xxxx0000 */
.byte    0x00            /* 基数: xx00xxxx */
.byte    0x92            /* 代码 r/w, Present, DPL0 */
.byte    0xcf            /* 界限: fxxxx, Page Gra, 32bit */
.byte    0x00            /* 基数: 00xxxxxx */

/* 3(selector=0x0018): 代码描述符, 用于嵌套中断 */
.word    0xffff          /* 界限: 0xffff */
.word    0x0000          /* 基数: xxxx0000 */
.byte    0x00            /* 基数: xx00xxxx */
.byte    0x9a            /* 代码 e/r, Present, DPL0 */
.byte    0xcf            /* 界限: fxxxx, Page Gra, 32bit */
.byte    0x00            /* 基数: 00xxxxxx */

/* 4(selector=0x0020): 代码描述符, 用于嵌套中断 */
.word    0xffff          /* 界限: 0xffff */
.word    0x0000          /* 基数: xxxx0000 */
.byte    0x00            /* 基数: xx00xxxx */
.byte    0x9a            /* 代码 e/r, Present, DPL0 */
.byte    0xcf            /* 界限: fxxxx, Page Gra, 32bit */
.byte    0x00            /* 基数: 00xxxxxx */

/* 下面的指令在 32 位模式下执行 */

.align 4,0x90
_romInit2:
cli                /* 关中断 */
mov $ ROM_STACK,%esp /* 设置堆栈指针 */
call _romA20on      /* 打开 A20 */
movl $ BOOT_COLD,%ebx /* %ebx 包含启动类型——startType */

warm:
movl $_romGdtr,%eax /* 装载原始 GDT */
subl $_romInit,%eax

```



```

addl $ ROM_TEXT_ADRS, %eax
pushl  %eax
call _romLoadGdt

movl $ STACK_ADRS, %esp      /* 初始化堆栈指针 */
movl $0, %ebp                /* 初始化结构 (frame) 指针 */
pushl  $0                    /* 初始化%eflags */
popfl

pushl  %ebx                  /* startType 入栈 */
cld                          /* 拷贝自己到入口点 */
movl   $ ROM_TEXT_ADRS, %esi
movl   $_romInit, %edi
movl   $_end, %ecx
subl   %edi, %ecx
shrl   $2, %ecx
rep
movsl
movl $_romStart, %eax        /* 跳转到 romStart 处 */
call *%eax

```

/*万一usrInit 存在问题, 则执行下面代码 */

```

_romInitHlt:
hlt

```

下面是 romInit()函数中所调用的一些相关函数。

```

/*****
*
* romA20on - 打开 A20
*
* 打开 A20
*
* 返回: N/A
* void romA20on (void)
*/

.align 4,0x90

_romA20on:

... /* 如果目标板需要打开 A20, 在这里添加代码 */
ret

.align 4,0x90

_romA20Result:
.long 0x00000000 /* 0 表示 A20 打开, -1 表示未打开 */

/*****
*
* romLoadGdt - 装载全局描述符号表
*

```



```

* 返回: N/A
*
* void romLoadGdt (char *romGdtr)
*/

        .align 4,0x90
_romLoadGdt:
movl 4(%esp),%eax
lgdt (%eax)
movw $0x0010,%ax      /* selector 0x0010 是第 3 个描述符 */
movw %ax,%ds
movw %ax,%es
movw %ax,%fs
movw %ax,%gs
movw %ax,%ss
ret
-----

```

13.2.2 romStart()函数

romStart()函数是所有固化在 ROM/Flash 里 VxWorks 映像第一个执行的 C 语言函数, 对该函数主要从如下几个方面进行描述。

1. romStart()函数的功能简述

romStart()函数包含在../all/bootInit.c 文件中, 且用 C 语言编写。它是系统最先执行的一个 C 语言程序, 为 ROM 映像执行必要的代码重定位、解压和 RAM 初始化操作。具体操作如下:

- 拷贝适当的 ROM 映像段到 RAM 中;
- 清理没有被使用的那部分内存 (冷启动时);
- 执行解压操作 (如果需要);
- 调用预内核的通用初始化程序——usrInit()。

2. romStart()的修改

在 BSP 开发过程中, 可能需要修改 bootInit.c 文件。这个文件是 BSP 的一个公用文件, 所以对于该文件的修改, 情况比较特殊, 具体操作步骤如下:

(1) 拷贝 bootInit.c 文件。bootInit.c 文件是一个公用文件, 所以不可以直接修改../config/all 目录下的 bootInit.c 文件。而是把该文件拷贝到 BSP 目录下 (例如 config\pcPentium 目录下), 然后修改 BSP 目录下的 bootInit.c 文件。

(2) 修改 makefile 文件。为了链接拷贝的 bootInit.c 文件而不是原始的文件, 需要修改 BSP 目录下的 makefile 文件。在 HEX_FLAGS 宏定义后面增加下面一行宏定义:

```
BOOTINIT = bootInit.c
```

在构造 VxWorks 映像的控制文件 rules.bsp 中, 系统通过使用 BOOTINIT 宏定义来访问 bootInit.c 文件。这个宏定义的默认值访问的是../config/all/bootInit.c 文件。

3. romStart()配置宏

配置宏控制着 romStart()的行为。在 config.h、makefile、configAll.h 及 bootInit.c 中定义了这些宏。而 BSP 开发人员的职责需要配置 config.h、<bsp>.h、makefile 文件中的宏定义, 但不应当修改

bootInit.c 中的宏。

(1) config.h 文件中的宏定义。

config.h 文件中的宏定义如表 13-3 所示。

表 13-3 config.h 文件中的宏定义

宏 定 义	描 述
LOCAL_MEM_LOCAL_ADRS	目标板本地 RAM 起始地址
LOCAL_MEM_SIZE	本地 RAM 大小
USER_RESERVED_MEM	从 RAM 顶端起用户保留字节数
RAM_HIGH_ADRS	拷贝引导映像的入口地址
RAM_LOW_ADRS	拷贝 VxWorks 映像入口地址
ROM_TEXT_ADRS	引导 ROM 的入口地址
ROM_SIZE	ROM 空间大小
ROM_BASE_ADRS	ROM 基地址

(2) makefile 文件中的配置宏。makefile 文件中的配置宏如表 13-4 所示。

表 13-4 makefile 文件中的配置宏

宏 定 义	描 述
RAM_HIGH_ADRS	引导映像入口地址
RAM_LOW_ADRS	装载 VxWorks 映像的地址
ROM_TEXT_ADRS	引导 ROM 的入口地址
ROM_SIZE	ROM 空间大小

(3) configAll.h 文件中的宏定义。configAll.h 文件中的宏定义如表 13-5 所示。

表 13-5 configAll.h 文件中的宏定义

宏 定 义	描 述
RESERVED	从 RAM 底部起用户保留字节数
STACK_SAVE	为 RomStart()准备的堆栈大小

(4) bootInit.c 文件中的宏定义。bootInit.c 文件中的宏定义如表 13-6 所示。

表 13-6 bootInit.c 文件中的宏定义

宏 定 义	描 述
USER_RESERVED_MEM	从 RAM 顶端起用户保留字节数
SYS_MEM_BOTTOM	系统存储区底部地址
SYS_MEM_TOP	系统存储区顶端地址

续表

宏 定 义	描 述
UNCMP_RTN	解压函数名
ROM_OFFSET	重新计算绝对地址为非 PIC 的宏
RAM_DST_ADRS	压缩映像的最终重新定位的地址
RESIDENT_DATA	体系结构特殊信息
ROM_COPY_SIZE	非压缩 ROM 驻留型映像的重新定位大小
ROM_BASE_ADRS	ROM 基地址
binArrayStart	压缩型二进制映像的起始地址
binArrayEnd	压缩型二进制映像的结束地址

(5) romStart()函数的可选宏定义。romStart()函数可选的宏定义如表 13-7 所示。

表 13-7 romStart()函数可选的宏定义

宏 定 义	描 述
BOOTCODE_IN_RAM	在系统冷启动时不清 RAM 区
UNCOMPRESS	编译时产生非压缩型映像
ROM_RESIDENT	编译时产生 ROM 驻留型映像

4. ROM 和 RAM 的布局

(1) ROM 的布局如图 13.3 所示。

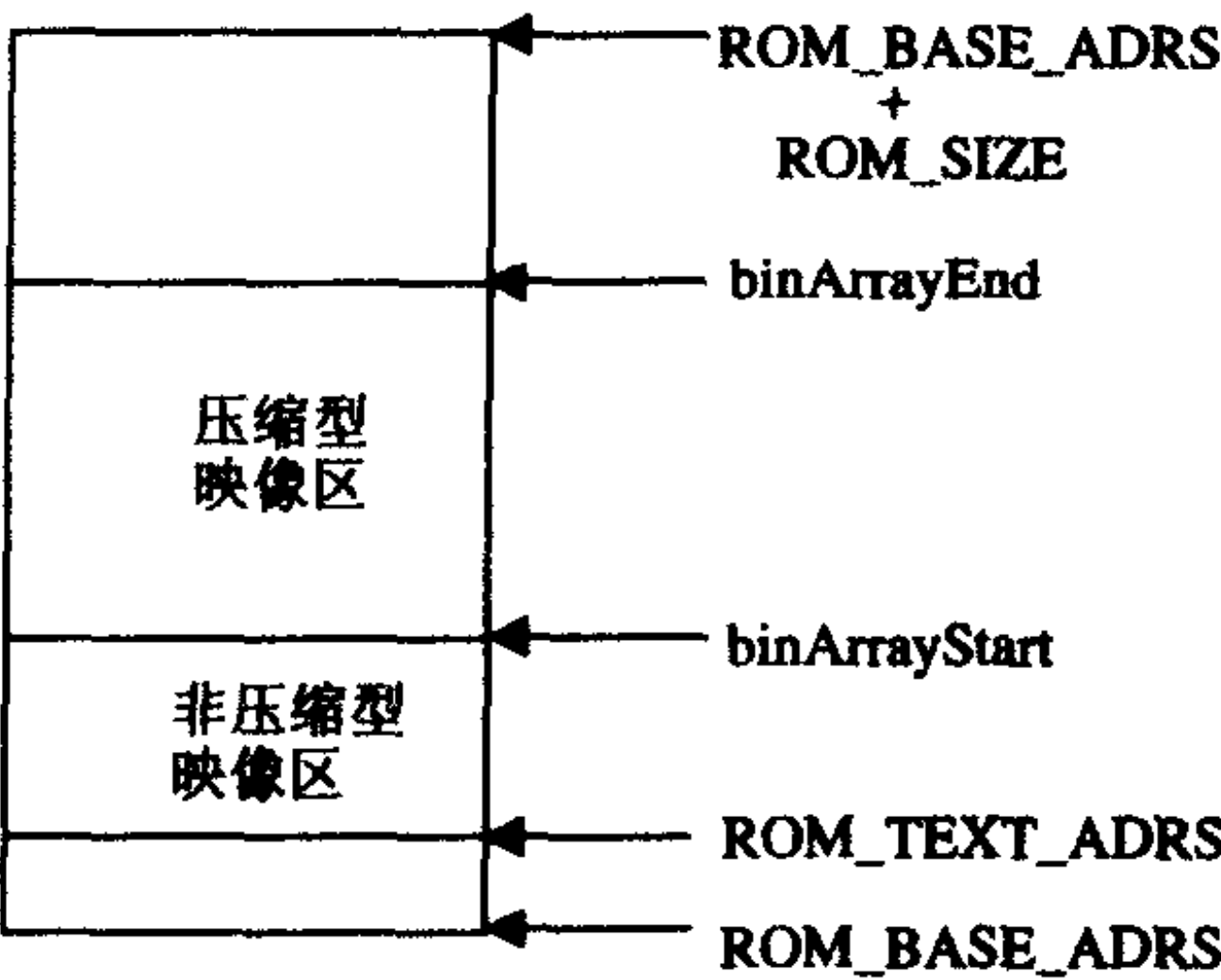


图 13.3 ROM 布局

从图 13.3 中可以发现压缩型映像和非压缩型映像处于不同的 ROM 空间。

(2) RAM 的布局如图 13.4 所示。

从图 13.4 中可以发现用户保留区处于 RAM 空间的顶端和底部，映像重新定位区位于保留堆栈区的上面。

5. romStart()函数示例

下面是 romStart()函数源代码。

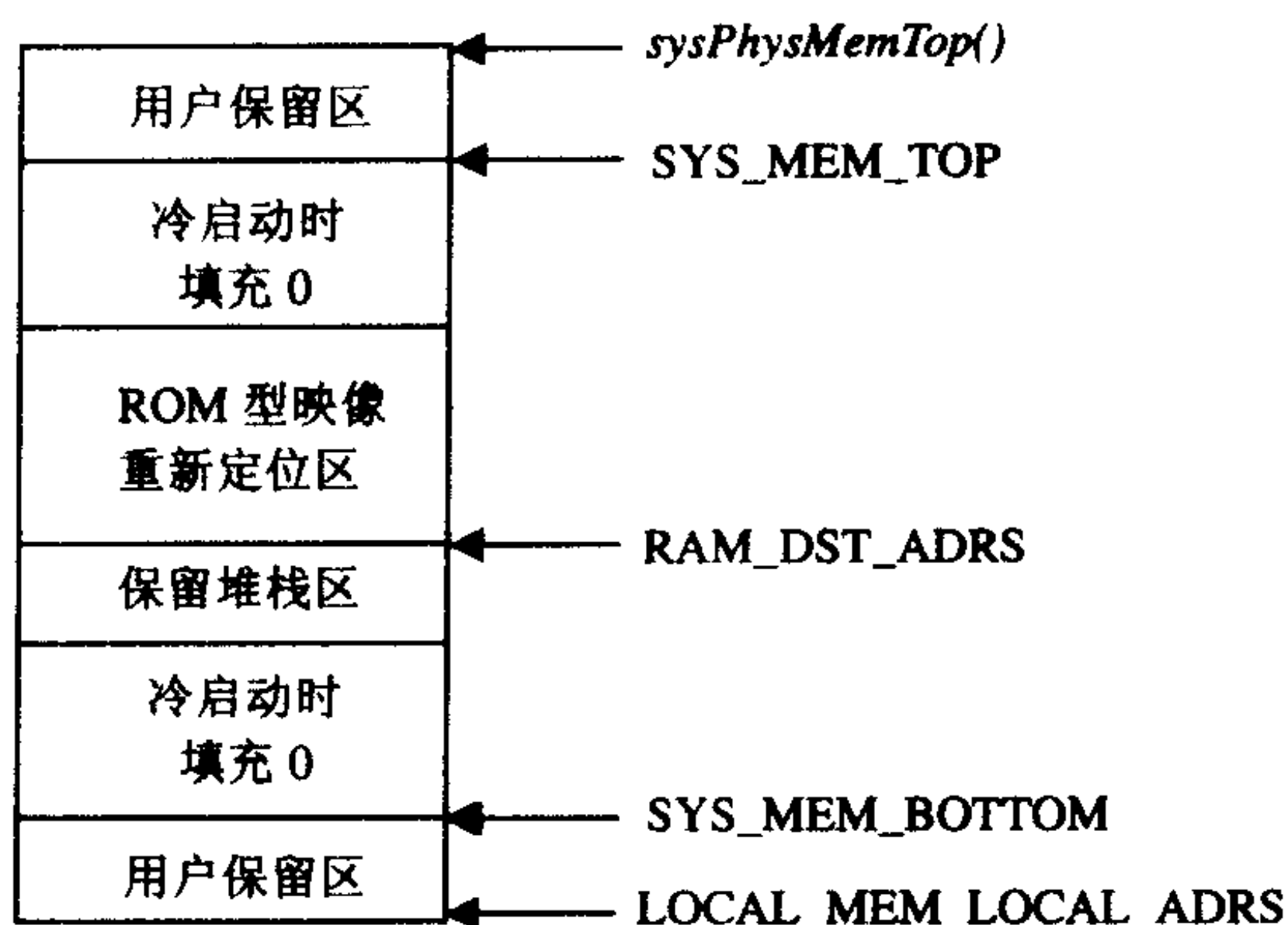


图 13.4 RAM 布局

```

=====
bootInit.c——包含 romInit()函数的文件
=====
/* bootInit.c - ROM 初始化模块 */

/* 版权 1989-1998 Wind River Systems, Inc. */
#include "copyright_wrs.h"

/*
修改记录:
...
03e,09dec94,caf undid mod 03a, use sdata for resident roms (SPR #3856).
*/

/*

```

描述:

该函数提供一个通用的引导 ROM 功能，是引导 ROM 第二阶段的初始化。该文件定义了 `romStart()` 函数，当 `romInit.s` 文件中的 `romInit()` 函数执行完最小目标板初始化后跳转到 `romStart()` 函数。这个程序执行必要的代码压缩和 ROM 映像的重定位。首先把代码段 (text) 和数据段 (data) 从 ROM 里拷贝到 RAM 中，然后初始化主 RAM 中其他不用的部分，最后进行映像压缩部分的解压操作。不同的配置选择将会执行不同的操作。

参考函数:

inflate()、romInit()等。

著者:

```
...
*/

#include "vxWorks.h"
#include "sysLib.h"
#include "config.h"
#include "errno.h"
#include "sioLib.h"
```



```

#define UNCMP_RTN    inflate

#ifndef USER_RESERVED_MEM
#define USER_RESERVED_MEM 0
#endif

/*
 * 如果需要清除内存，系统将从 SYS_MEM_BOTTOM 到 SYS_MEM_TOP 进行内存清除操作，但
 * 除代码段和数据段之外。
 * 用户保留区不可以清除。
 */

#define SYS_MEM_TOP \
    (LOCAL_MEM_LOCAL_ADRS + LOCAL_MEM_SIZE - USER_RESERVED_MEM)

#define SYS_MEM_BOTTOM \
    (LOCAL_MEM_LOCAL_ADRS + RESERVED)

IMPORT void    romInit ();
IMPORT STATUS UNCMP_RTN ();
IMPORT void    usrInit ();
IMPORT void    sysInitAlt ();
IMPORT void    start ();

#if ((CPU_FAMILY == MIPS) || (CPU_FAMILY == PPC))
#define binArrayStart    _binArrayStart
#define binArrayEnd      _binArrayEnd
#define RESIDENT_DATA    RAM_DST_ADRS
#else
#define RESIDENT_DATA    (&sdata)
IMPORT char    sdata;      /* 在 romInit.s 中定义该参数 */
#endif

IMPORT UCHAR binArrayStart []; /* 压缩型二进制映像 */
IMPORT UCHAR binArrayEnd;      /* 压缩型二进制映像的结束处 */
IMPORT char  etext [];
IMPORT char  edata [];
IMPORT char  end [];

/* 压缩映像的最终重新定位的地址 */
#ifndef RAM_DST_ADRS      /* 默认压缩映像的最终重新定位的地址 */
#define RAM_DST_ADRS    RAM_HIGH_ADRS
#endif

/* 如果引导代码已经初始化了 RAM 区，对 RAM 区不需要进行重新清除操作。宏 BOOTCODE_IN_RAM 是不需要
清除的 RAM 区 */
#ifdef BOOTCODE_IN_RAM    /* 不需要清除的 RAM 区 */
#undef ROM_TEXT_ADRS
#undef ROM_BASE_ADRS
#define ROM_TEXT_ADRS    ((UINT)romInit)
#define ROM_BASE_ADRS    ((UINT)romInit)
#endif /* BOOTCODE_IN_RAM */

#if defined (UNCOMPRESS) || defined (ROM_RESIDENT)
#ifndef ROM_COPY_SIZE
#define ROM_COPY_SIZE    (ROM_SIZE - (ROM_TEXT_ADRS - ROM_BASE_ADRS))

```



```

#endif
#endif /* UNCOMPRESS */

#define ROM_OFFSET(adr) (((UINT)adr - (UINT)romInit) + ROM_TEXT_ADRS)

/* 声明 */

LOCAL void copyLongs (FAST UINT *source, FAST UINT *destination, UINT nlongs);
#ifdef BOOTCODE_IN_RAM
LOCAL void fillLongs (FAST UINT *buf, UINT nlongs, FAST UINT val);
#endif /* BOOTCODE_IN_RAM */

/*****
*
* romStart - 通用 ROM 初始化
*
* 系统复位后, 首先执行的 C 代码。
*
* romInit() 汇编函数执行完最小目标板初始化后调用 romStart() 函数。
* 它清内存, 拷贝 ROM 中的相关段到 RAM 中, 可能需要进行解压操作。
* 最后跳转到非压缩对象代码的入口点处。
*
* 返回: N/A
*/

void romStart
(
    FAST int startType /* 启动类型 */
)
{
    #if ((CPU_FAMILY==SPARC) || (CPU_FAMILY==MIPS) || (CPU_FAMILY==I80x86) || \
        (CPU_FAMILY==PPC) || (CPU_FAMILY==ARM))
        volatile /* 强制绝对地址 */
    #endif /* (CPU_FAMILY==SPARC) */
    FUNCPTR absEntry; /* 转移子程序 */
    #if (CPU_FAMILY==ARM) && (!defined(ROM_RESIDENT))
        VOIDFUNCPTR ramfillLongs = fillLongs; /* 强制访问 RAM */
    #define fillLongs(a,b,c) ramfillLongs(a,b,c)
    #endif /* (CPU_FAMILY==ARM) */

    /*
    * 把相关段从 ROM 中拷贝到 RAM 中。
    * 如果压缩引导 ROM 依赖 binArray 存放在数据段最后处, 则减去压缩映像。
    */

    #ifdef ROM_RESIDENT
        /* 如果是 ROM 驻留代码, 然而仅仅只是把数据段从 ROM 中拷贝到 RAM 中,
        * 然后初始化内存并跳转到 usrInit。
        */

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

```



```

#endif
    ((UINT) edata - (UINT) RESIDENT_DATA) / sizeof (long));

#else    /* ROM_RESIDENT */

#ifdef UNCOMPRESS

#if (CPU_FAMILY == MIPS)
    /*
     * 为了避免出问题，代码段回拷到高速缓存里。
     */
    ((FUNCPTR)ROM_OFFSET(copyLongs)) (ROM_TEXT_ADRS, (UINT)K0_TO_K1(romInit),
        ROM_COPY_SIZE / sizeof (long));
#else    /* CPU_FAMILY == MIPS */
    ((FUNCPTR)ROM_OFFSET(copyLongs)) (ROM_TEXT_ADRS, (UINT)romInit,
        ROM_COPY_SIZE / sizeof (long));
#endif    /* CPU_FAMILY == MIPS */

#else    /* UNCOMPRESS */

#if (CPU_FAMILY == MIPS)
    /*
     * 为了避免出问题，代码段回拷到高速缓存里。
     */

    ((FUNCPTR)ROM_OFFSET(copyLongs)) (ROM_TEXT_ADRS, (UINT)K0_TO_K1(romInit),
        ((UINT)edata - (UINT)romInit) / sizeof (long));
#else    /* CPU_FAMILY == MIPS */
    ((FUNCPTR)ROM_OFFSET(copyLongs)) (ROM_TEXT_ADRS, (UINT)romInit,
        ((UINT)binArrayStart - (UINT)romInit) / sizeof (long));
#endif    /* CPU_FAMILY == MIPS */

#endif    /* UNCOMPRESS */
#endif    /* ROM_RESIDENT */

#if (CPU_FAMILY != MIPS) && (!defined (BOOTCODE_IN_RAM))

/* 如果是冷引导清除所有内存 */

    if (startType & BOOT_CLEAR)
    {
#ifdef ROM_RESIDENT

/* 清除没有装载代码段和数据段的内存。
 * 要非常小心初始化出现奇偶错的所有内存(除 STACK_SAVE 字节外)。
 */

fillLongs ((UINT *)SYS_MEM_BOTTOM,
    ((UINT) RESIDENT_DATA - STACK_SAVE - (UINT)SYS_MEM_BOTTOM)
    / sizeof(long), 0);
fillLongs ((UINT *) edata,
    ((UINT)SYS_MEM_TOP - ((UINT) edata)) / sizeof(long), 0);

#else    /* ROM_RESIDENT */
fillLongs ((UINT *) (SYS_MEM_BOTTOM),

```



```
((UINT)romInit - STACK_SAVE - (UINT)SYS_MEM_BOTTOM) /
sizeof(long), 0);

#if defined (UNCOMPRESS)
fillLongs ((UINT *)((UINT)romInit + ROM_COPY_SIZE),
            ((UINT)SYS_MEM_TOP - ((UINT)romInit + ROM_COPY_SIZE))
            / sizeof(long), 0);
#else
fillLongs ((UINT *)binArrayStart,
            ((UINT)SYS_MEM_TOP - (UINT)binArrayStart) / sizeof (long), 0);
#endif /* UNCOMPRESS */
#endif /* ROM_RESIDENT */

/*
 * 保证引导行为空。对于那些拒绝接纳引导行的目标机来说是必要的。
 */

*(BOOT_LINE_ADRS) = EOS;
}

#endif /* (CPU_FAMILY != MIPS) && (!defined (BOOTCODE_IN_RAM)) */

/* jump to VxWorks entry point (after uncompressing) */

#if defined (UNCOMPRESS) || defined (ROM_RESIDENT)
#if (CPU_FAMILY == I960)
    absEntry = (FUNCPTR)sysInitAlt;          /* 重新初始化系统 */
#else
    absEntry = (FUNCPTR)usrInit;             /* 系统初始化 */
#endif /* CPU_FAMILY == I960 */

#else
{
    #if (CPU_FAMILY == MIPS)
        volatile FUNCPTR absUncompress = (FUNCPTR) UNCMP_RTN;
        if ((absUncompress) ((UCHAR *)ROM_OFFSET(binArrayStart),
                                (UCHAR *)K0_TO_K1(RAM_DST_ADRS),
                                (int)((UINT)&binArrayEnd - (UINT)binArrayStart)) != OK)
    #elif (CPU_FAMILY == I80x86) || (CPU_FAMILY == ARM)
        volatile FUNCPTR absUncompress = (FUNCPTR) UNCMP_RTN;
        if ((absUncompress) ((UCHAR *)ROM_OFFSET(binArrayStart),
                                (UCHAR *)RAM_DST_ADRS, &binArrayEnd - binArrayStart) != OK)
    #else
        if (UNCMP_RTN ((UCHAR *)ROM_OFFSET(binArrayStart),
                                (UCHAR *)RAM_DST_ADRS, &binArrayEnd - binArrayStart) != OK)
    #endif /* (CPU_FAMILY == MIPS) */
    return; /* 如果返回, 则 ROM 将停机 */

    absEntry = (FUNCPTR)RAM_DST_ADRS;        /* compressedEntry() 函数 */
}
#endif /* defined UNCOMPRESS || defined ROM_RESIDENT */

#if (CPU == ARM7TDMI_T)
    absEntry = (FUNCPTR)((UINT32)absEntry | 1);
#endif
```



```
#endif /* (CPU == ARM7TDMI_T) */
```

```
(absEntry) (startType);
}
```

```
#if (CPU_FAMILY==ARM) && (!defined(ROM_RESIDENT))
#undef fillLongs
#endif /* (CPU_FAMILY==ARM) */
```

下面是 romStart()函数中所调用的相关函数。

```

/*****
*
* copyLongs -缓冲区拷贝函数。
*
* 这个函数从源缓冲区<source> 中拷贝<nlongs>长度的数据到目的缓冲区
* <destination>中。
*/

LOCAL void copyLongs (source, destination, nlongs)
    FAST UINT *source;      /* 指向源缓冲区的指针 */
    FAST UINT *destination; /* 指向目的缓冲区的指针 */
    UINT nlongs;            /* 拷贝数据大小 */

{
    FAST UINT *dstend = destination + nlongs;
    while (destination < dstend)
        *destination++ = *source++;
}

#ifndef BOOTCODE_IN_RAM
/*****
* fillLongs -缓冲区填充函数
*
* 这个函数用变量<val>填充<nlongs>长度的缓冲区。
*/

LOCAL void fillLongs (buf, nlongs, val)
    FAST UINT *buf;      /* 指向填充缓冲区的一个指针 */
    UINT nlongs;          /* 填充长度 */
    FAST UINT val;        /* 填充缓冲区的字符 */

{
    FAST UINT *bufend = buf + nlongs;
    while (buf < bufend)
        *buf++ = val;
}
#endif /* BOOTCODE_IN_RAM */

```

13.2.3 sysInit()函数

1. sysInit()函数的功能简述

sysInit()函数是装载型 VxWorks 映像的入口点。当映像被装载到 RAM 中以后，处理器将跳转到 sysInit()函数处并执行。该函数原型在 sysALib.s 中，是用汇编语言编写的。

当映像被装载到 RAM 中, sysInit()函数驻留在装载型 VxWorks 映像的装载地址 RAM_LOW_ADRS 处。它执行必须的最小初始化操作及调用 usrInit()函数, 而硬件初始化推迟到预内核通用初始化代码中进行。除不执行内存系统初始化操作外, 其他所有功能与 romInit()函数一样。

sysInit()函数必须包含下列功能:

- 屏蔽处理器中断和复位处理器操作;
- 初始化堆栈指针和其他寄存器, 并调用 usrInit()函数及传递启动类型。

2. sysInit()函数代码的获得

sysInit()函数代码可以通过修改 romInit()函数而获得, 仅仅需要作一些修改便可以创建 sysInit()函数代码。具体修改内容如下:

- 删除内存初始化代码;
- 把调用 romStart()函数改为调用 usrInit()函数;
- sysInit()在 RAM 中执行, 所以不需要设计成与地址无关的代码。

3. RAM 布局

装载型 VxWorks 映像 in RAM 中的布局如图 13.5 所示。从图 13.5 中可以看到, VxWorks 映像 in 目标机 RAM 中的起始地址为 ROM_LOW_ADRS。实际上这就是 sysInit()函数入口地址。

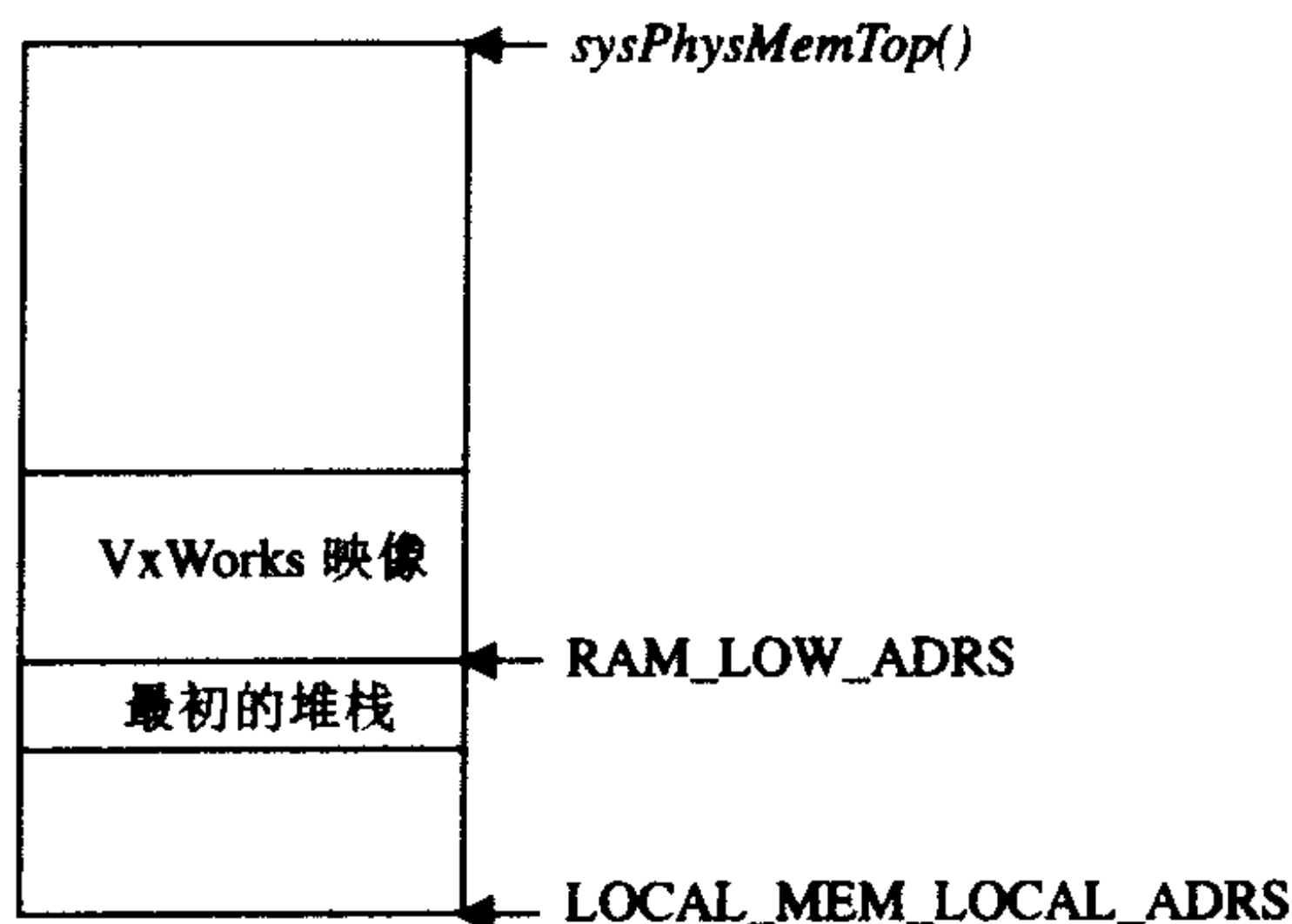


图 13.5 装载型映像的 RAM 布局

4. sysInit()函数示例

下面是一个 x86 体系结构的 sysInit()函数模板。

```
=====
sysInit() 函数
=====
/*****
 *
 * sysInit - 引导后装载型 VxWorks 映像的启动函数。
 *
 * 这个函数是 RAM 中 VxWorks 映像的系统启动入口点, 系统引导后首先执行的代码。
 * 它屏蔽中断, 安装堆栈, 然后调用 usrConfig.c 文件中的 C 函数 usrInit()。
 *
 * 最初的(initial)堆栈设置在 sysInit()的地址下面。仅仅只是 usrInit()使用该堆栈,
 * 并永远不会再一次使用该堆栈。当决定系统装载地址时, 必须计算堆栈内存区。
 *****/
```



```

*
* 注意：用户不应当调用这个函数。
*
* 返回：N/A

* sysInit ()          /@ 用户不应当调用的函数@/

*/

_sysInit:
cli                /* 关中断 */
movl $BOOT_WARM_AUTOBOOT,%ebx /* 启动类型 startType 保存到%ebx 中 */

movl $_sysInit,%esp /* 初始化堆栈指针 */
movl $0,%ebp        /* 初始化帧 (frame) 指针 */
pushl $0            /* 初始化%eflags */
popfl

    pushl $_sysGdtr /* 装载 GDT */
    call _sysLoadGdt

    pushl %ebx      /* 启动类型 startType 入栈 */
    movl $_usrInit,%eax
    movl $_sysInit,%edx
    pushl %edx      /* %edx, _sysInit 入栈 */
    pushl $0        /* EFLAGS0 入栈 */
    pushl $0x0008
    pushl %eax      /* EIP, _usrInit 入栈 */
    iret           /* 返回 */
-----

```

13.3 预内核的通用初始化代码

13.3.1 通用初始化代码概述

1. 通用初始化代码职责

(1) 环境的提供。通用初始化代码必须为 VxWorks 内核提供一个环境，以便激活 VxWorks 内核。

(2) 内核激活前系统情况。在激活内核前系统没有初始化内存池。这样隐含下列情况：

- 没有多任务环境；
- 没有中断管理；
- 不可以访问 I/O 硬件设备；
- 不可以访问网络接口。

(3) 内核激活后系统情况。内核代码执行后，初始化系统所需要的系统内存空间或多任务环境。

2. 通用初始化

预内核的通用初始化操作是通过 `usrInit()` 函数来执行，该函数是用 C 语言编写，原型包含在 `usrConfig.c` 文件里。当构造 VxWorks 映像时，把它静态的链接到映像中。系统调用该函数同时也

激活了 VxWorks 内核。

而初始化硬件设备的操作主要是通过调用 sysHwInit()函数来实现, 该函数主要功能如下:

- 屏蔽所有硬件设备的中断;
- 初始化硬件设备到可知的静止状态。

总之, usrInit()执行最小必要的初始化操作及激活内核。

13.3.2 通用初始化详细描述

通用初始化实际上是由 usrInit()函数来实现, 下面是 usrInit()函数原代码, 有关通用初始化详细描述请参考函数的详细内部说明。

```

=====
usrInit() 函数
=====
/*****
 *
 * usrInit - 用户定义的系统初始化函数
 *
 * 这个函数是系统引导后首先执行的 C 代码函数。在关闭中断情况下调用该函数, 而在这
 * 之前, 内核处于非多任务环境下。
 *
 * 该函数首先对 bss 赋零; 这样把所有的变量初始化成 0, 通过调用 excVecInit() 函数
 * 初始化所有系统和缺省中断向量; 通过调用 sysHwInit(), 初始化与目标板相关的硬件;
 * 安装中断/异常向量; 通过调用 usrKernelInit(), 配置 VxWork 内核; 调用 kernelInit()
 * 初始化并启动内核; usrRoot() 作为系统的根任务。
 *
 * 返回: N/A
 *
 * 参考: kernelLib
 *
 */

void usrInit
(
    int startType /* 启动类型 */
)
{
    #if (CPU_FAMILY == SPARC)
        excWindowInit ();                /* SPARC 窗口管理 */
    #endif

    /* 配置数据和指令高速缓存 */

    #ifdef INCLUDE_CACHE_SUPPORT
        cacheLibInit (USER_I_CACHE_MODE, USER_D_CACHE_MODE);
    #endif /* INCLUDE_CACHE_SUPPORT */

    #if CPU_FAMILY!=SIMNT && CPU_FAMILY!=SIMSPARCSUNOS && CPU_FAMILY!=SIMHPPA &&
    CPU_FAMILY!=SIMSPARCSOLARIS

        /* 该调用前, 不要假设变量已赋零 */
        bzero (edata, end - edata);      /* 对 BSS 进行初始化 */
    #endif
}

```



```

#endif /* CPU_FAMILY!=SIMNT && CPU_FAMILY!=SIMPARCSUNOS && CPU_FAMILY!=SIMHPPA &&
CPU_FAMILY!=SIMPARCSOLARIS */

    sysStartType = startType; /* 保存系统启动类型 */

    intVecBaseSet ((FUNCPTR *) VEC_BASE_ADRS); /* 设置向量表基地址 */

    #if (CPU_FAMILY == AM29XXX)
        excSpillFillInit (); /* am29k 堆栈高速缓存管理 */
    #endif

    #ifdef INCLUDE_EXC_HANDLING
        excVecInit (); /* 安装异常向量 */
    #endif /* INCLUDE_EXC_HANDLING */

    sysHwInit (); /* 初始化系统硬件 */

    usrKernelInit (); /* 配置 Wind 内核 */

    #ifdef INCLUDE_CACHE_SUPPORT
    #ifdef USER_I_CACHE_ENABLE
        cacheEnable (INSTRUCTION_CACHE); /* 打开指令高速缓存 */
    #endif /* USER_I_CACHE_ENABLE */

    #ifdef USER_D_CACHE_ENABLE
        cacheEnable (DATA_CACHE); /* 打开数据高速缓存 */
    #endif /* USER_D_CACHE_ENABLE */

    #if (CPU == MC68060)
    #ifdef USER_B_CACHE_ENABLE
        cacheEnable (BRANCH_CACHE); /* 打开分支 (branch) 高速缓存 */
    #endif /* USER_B_CACHE_ENABLE */
    #endif /* (CPU == MC68060) */
    #endif /* INCLUDE_CACHE_SUPPORT */

    /* 启动内核, 指定 usrRoot 为根任务 */
    kernelInit ((FUNCPTR) usrRoot, ROOT_STACK_SIZE,
#ifdef INCLUDE_WDB
        (char *) FREE_RAM_ADRS + WDB_POOL_SIZE,
#else
        (char *) FREE_RAM_ADRS,
#endif
    sysMemTop (), ISR_STACK_SIZE, INT_LOCK_LEVEL);
}
-----

```

13.3.3 系统硬件初始化——sysHwInit()

sysHwInit()函数的功能是初始化系统硬件, 该函数包含在 sysLib.c 文件中。从如下几个方面对它进行详细描述。

1. sysHwInit()函数职责

(1) 在预内核中, 设置硬件环境到一静止态。其中包括如下处理:

- 初始化硬件环境;

- 屏蔽硬件中断;
- 初始化设备控制/状态寄存器。
- (2) 初始化设备所需要的寄存器和 BSP 指定的环境参数。
- (3) 必须屏蔽所有设备的硬件中断, 其对中断控制过程如下:
 - 调用中断控制器初始化函数 (例如 `sysIntInitPIC ()`) 初始化并屏蔽所有硬件中断;
 - 当调用 `kernelInit()` 函数启动内核时, 中断被打开;
 - 不可以安装 ISR, 直到启动内核。

2. 环境、设备和中断

(1) 环境和设备的初始化首先处理的是屏蔽中断。

(2) 通用环境的配置。在明确设备可以被访问前, 必须对通用环境进行部分配置, 需要进行的处理如下:

- 初始化存储控制器;
- 配置访问设备的总线。
- (3) 目标机可能包含一个中断控制器 (例如 8259A)。
- (4) 通用设备的初始化。在屏蔽硬件中断前, 必须对通用设备进行部分初始化操作。其中包括:
 - 配置设备, 允许中断控制器访问;
 - 可能需要初始化中断控制器。

3. 环境的初始化

目标机系统访问外围设备时, 必须为其提供一个环境。其中包括:

(1) CPU 访问系统总线。CPU 访问系统总线是通过下面操作来实现:

- 执行 `romInit()` 或 `sysInit()` 函数实现局部总线的访问;
- 执行 `sysHwInit()` 函数实现局部总线与其他系统总线的连接。

(2) 总线设备的访问。执行 `sysHwInit()` 函数提供访问总线设备的环境。

4. 设备的初始化。

(1) 通用设备的初始化。在 `sysHwInit()` 函数中包含了通用设备初始化操作, 这些设备通常包括:

- 可编程中断控制器;
- 总线控制器的初始化;
- 串口的初始化。

(2) I/O 设备的初始化。在 `UsrRoot()` 函数中包含了 I/O 设备的初始化操作。

5. BSP 下设备控制

(1) 通用设备的驱动程序。通用设备驱动程序控制着设备硬件, 这些程序能够被多个 BSP 所使用, 其源代码在 `../target/src/drv` 和 `../target/h/drv` 目录下。

(2) 特殊设备驱动程序。对于惟一的设备和 Wind River 系统不支持的设备, 这些设备驱动程序通常驻留在 BSP 目录下。

6. 串口控制器的初始化

(1) 初始化。初始化串行控制器为轮询方式, 这样, 允许在预内核支持系统级调试。

(2) 初始化程序。在 `sysHwInit()` 函数中调用串口初始化程序 (例如 `sysSerialHwInit ()`), 其源代

码在 sysSerial.c 文件中。

(3) sysSerialHwInit() 函数。该函数所进行的处理如下：

- 初始化每个通道的串行控制器的数据结构；
- 调用驱动程序初始化代码，并屏蔽中断。

7. 调试 sysHwInit() 函数

如果内核启动后，系统没有启动初始化任务 tUsrRoot，则在 sysHwInit() 里进行的设备初始化操作可能会出现问题。例如可能有设备产生中断，那么必须找到中断源。其调试的手段如下：

- 修改 sysHwInit() 函数，把调试程序与可疑设备中断关联；
- 通过 ICE 在中断向量表里设置断点；
- 通过逻辑分析仪分析访问中断向量表的指令。

8. 重启

如果 sysHwInit() 遇到错误，为了阻止继续执行初始化操作，它将重新启动系统。而重启操作实际上是调用如下函数来实现的：

```
STATUS sysToMonitor (int startType )
```

这个函数将把 CPU 控制权转交给 BSP 指定的 ROM 监控程序。输入参数 startType 为重启类型参数，其定义在 ../target/h/sysLib.h 中，宏定义说明如下：

```
#define BOOT_NORMAL          0x00 /* 通常的倒数计秒启动方式 */
#define BOOT_NO_AUTOBOOT     0x01 /* 非自动启动方式 */
#define BOOT_CLEAR           0x02 /* 清除内存启动方式 */
#define BOOT_QUICK_AUTOBOOT  0x04 /* 快速自动启动方式 */
```

9. sysHwInit() 函数原型

下面是一个 x86 体系结构的 sysHwInit() 函数原型，其功能详细描述见函数内部说明。

```

/*****
 *
 * sysHwInit - 初始化系统硬件
 *
 * 该函数初始化 i386/i486 目标板上各种各样的硬件。
 * usrConfig.c 文件中的 usrInit() 函数调用该函数。
 *
 * 注意：用户应用程序不可以直接调用该函数。
 *
 * 返回：N/A
 */

void sysHwInit (void)
{
    PHYS_MEM_DESC *pMmu;
    int ix = 0;

#ifdef (CPU == PENTIUM)

    /* 打开 MTRR (Memory Type Range Registers) */

    if ((sysCpuId.feature & CPUID_MTRR) == CPUID_MTRR)

```



```
{
    pentiumMtrrDisable ();          /* 关闭 MTRR */
#ifdef INCLUDE_MTRR_GET
    (void) pentiumMtrrGet (&sysMtrr); /* 通过 BIOS 获得 MTRR 初始值 */
#else
    (void) pentiumMtrrSet (&sysMtrr); /* 设置自己的 MTRR */
#endif /* INCLUDE_MTRR_GET */
    pentiumMtrrEnable ();          /* 打开 MTRR */
}

#ifdef INCLUDE_PMC
    /* 打开 PMC (Performance Monitoring Counters) */

    if ((sysProcessor == x86CPU_PENTIUMPRO) &&
        ((sysCpuId.feature & CPUID_MSR) == CPUID_MSR))
    {
        pentiumPmcStop ();          /* 停止 PMC0 和 PMC1 */
        pentiumPmcReset ();         /* 复位 PMC0 和 PMC1 */

        /*
         * 选择你关心的事件, 例如:
         * PMC_HW_INT_RX          - 接收到的硬件中断数
         * PMC_MISALIGN_MEM_REF - 可参考的不重合数据内存数
         */

        (void) pentiumPmcStart (PMC_EN | PMC_OS | PMC_UMASK_00 | PMC_HW_INT_RX,
                                PMC_EN | PMC_OS | PMC_UMASK_00 | PMC_MISALIGN_MEM_REF);
    }
#endif /* INCLUDE_PMC */

    /* 打开 MCA (Machine Check Architecture) */

    if ((sysCpuId.feature & CPUID_MCE) == CPUID_MCE)
    {
#ifdef INCLUDE_SHOW_ROUTINES
        IMPORT FUNCPTR excMcaInfoShow;

        /*
         * 如果 excMcaInfoShow 不为空, 当自动校验发生异常时, 将调用默认异常处理程序。
         */

        excMcaInfoShow = (FUNCPTR) pentiumMcaShow;
#endif /* INCLUDE_SHOW_ROUTINES */

        if ((sysCpuId.feature & CPUID_MCA) == CPUID_MCA)
        {
            UINT32 zero[2] = {0x00000000, 0x00000000};
            UINT32 one[2] = {0xffffffff, 0xffffffff};
            UINT32 cap[2];
            int mcaBanks;
            int ix;

            /* 如果 MCG_CTL 寄存器出现, 打开所有 MCA 特性 */

```



```
pentiumMsrGet (MSR_MCG_CAP, (long long int *)&cap);
if (cap[0] & MCG_CTL_P)
    pentiumMsrSet (MSR_MCG_CTL, (long long int *)&one);

mcaBanks = cap[0] & MCG_COUNT;

/* 除 MCO_CTL 寄存器以外, 打开所有错误记录 */

for (ix = 1; ix < mcaBanks; ix++)
    pentiumMsrSet (MSR_MCO_CTL+(ix * 4), (long long int *)&one);

/* 清除所有错误 */

for (ix = 0; ix < mcaBanks; ix++)
    pentiumMsrSet (MSR_MCO_STATUS+(ix * 4), (long long int *)&zero);
}

pentiumCr4Set (pentiumCr4Get () | CR4_MCE); /* 打开 MCE 异常 */
}

#endif /* (CPU == PENTIUM) */

/* 初始化可用映射数(sysPhysMemDescNumEnt) */

pMmu = &sysPhysMemDesc[0];

for (ix = 0; ix < NELEMENTS (sysPhysMemDesc); ix++)
    if (pMmu->virtualAddr != (void *)DUMMY_VIRT_ADDR)
        pMmu++;
    else
        break;

sysPhysMemDescNumEnt = ix;

/* 初始化 PIC (Programmable Interrupt Controller) */

sysIntInitPIC ();
intEoiGet = sysIntEoiGet;

/* 初始化 PCI 和关联设备 */

#ifdef INCLUDE_PCI

pciConfigLibInit (PCI_MECHANISM_1, PCI_CONFIG_ADDR, PCI_CONFIG_DATA, NONE);
pciIntLibInit ();

/*
 * 如果存在 PCI-to-PCI 桥, 应该在这里对它们进行初始化。
 * 对于 Intel 430HX PCISSET 没必要进行该步骤, 它按如下操作分布扩展内存:
 *   - Flash BIOS 区域从 4GByte 到 (4GB - 512KB)
 *   - DRAM 内存从 1MB 到最大为 512MB
 */
```



```

    * - PCI 内存空间从 DRAM 顶端到 (4GB - 512KB)
    */

    #if defined (INCLUDE_FEI) || defined (INCLUDE_FEI_END)
        sys557PciInit ();          /* 初始化 Intel 82557 网卡上的 PCI 控制器 */
    #endif /* INCLUDE_FEI || INCLUDE_FEI_END */

    #ifdef INCLUDE_LN_97X_END
        sysLan97xPciInit ();
    #endif /* INCLUDE_LN_97X_END */

    #ifdef INCLUDE_EL_3C90X_END
        sysEl3c90xPciInit ();
    #endif /* INCLUDE_EL_3C90X_END */

    #ifdef INCLUDE_SCSI
        #ifdef INCLUDE_AIC_7880
            sysAic7880PciInit ();
        #endif /* INCLUDE_AIC_7880 */
    #endif /* INCLUDE_SCSI */

    #endif /* INCLUDE_PCI */

    /* 初始化串行设备 */
    sysSerialHwInit ();          /* 初始化串行设备的数据结构 */
}
-----

```

13.3.4 启动内核

1. 内核启动概述

内核启动过程分两个步骤进行，其过程如下：

(1) 内核数据结构的配置。内核启动前需要对内核数据结构进行配置。而配置操作是由 `usrKernellInit()` 函数来实现。调用该函数则在 `sysHwInit()` 函数之后进行。有关信息如下：

- 源代码在 `../target/src/config/usrKernel.c` 文件中；
- 通过 `configAll.h` 文件中的配置宏进行库的初始化；
- 数据结构包括：二值信号量、看门狗定时器、任务对象、消息队列等。

(2) 内核启动。通过调用 `kernellInit()` 函数初始化和启动 VxWorks 内核，其过程如下：

- 调用 `intLockLevelSet()` 函数，设置当前的中断级别；
- 如果需要中断堆栈，在内存池底部创建中断堆栈；
- 从内存池顶部创建根堆栈和 TCB（任务控制块）；
- 调用 `taskInit()` 函数，初始化 `tRootTask` 任务；
- 调用 `taskActivate()` 函数，启动 `usrRoot` 任务；
- 执行 `usrRoot()`。

2. 系统内存块

系统内存块的顶部和底部地址作为参数传递给 `kernellInit()` 函数。

(1) 系统内存块的底部地址。其该地址变量值如下:

- 如果系统没有定义 `INCLUDE_WDB` 宏, 则地址为 `FREE_RAM_ADRS` 宏变量值;
- 如果系统定义了 `INCLUDE_WDB` 宏, 则地址为 `FREE_RAM_ADRS + WDB_POOL_SIZE`。

(2) 系统内存块的顶部地址。系统内存块的顶部地址值是通过调用 `sysMemTop()` 函数而返回的值。

(3) 初始化系统内存块。系统内存块的初始化操作在 `tUsrRoot` 任务里进行, 其中包括:

- 调用 `memInit()` 函数进行内存池的初始化;
- 如果包含虚拟内存, 则调用 `usrMmuInit()` 函数进行虚拟内存的初始化。

3. `tUsrRoot` 任务

`tUsrRoot` 任务是启动系统内核后第一个运行的任务, 它执行 I/O 系统的初始化操作, 设备驱动程序的安装, 创建设备 (在 `configAll.h` 和 `config.h` 中指定) 以及安装网络等操作。它相关的信息如下:

- 该任务优先级为 0;
- 任务堆栈为 `ROOT_STACK_SIZE`, 且从系统内存块中分配堆栈。

`tUsrRoot` 具体处理过程如下:

(1) 调用 `memInit()` 函数初始化内存池。

(2) 初始化及安装系统时钟, 其系统调用过程如下:

```
sysClkConnect ((FUNCPTR) usrClock, 0);    /* 连接时钟 ISR */
sysClkRateSet (60);                       /* 设置系统时钟 tick 数 */
sysClkEnable ();                          /* 启动时钟中断 */
```

(3) 初始化 I/O 系统 (可选项)。

(4) 创建设备 (可选项)。

(5) 配置网络 (可选项)。

(6) 启动 WDB 调试代理 (可选项), 源代码选项如下:

```
#ifdef INCLUDE_WDB
    wdbConfig();
```

(7) 启动应用程序。

13.4 预内核初始化的调试

这一小节主要基于 Tornado 开发环境, 如何进行预内核初始化的调试工作。

13.4.1 概述

基于 Tornado 下的工具调试预内核的初始化代码, 这种手段必须在系统硬件初始化函数 `sysHwinit()` 执行后才可以进行。同时需要 WDB 调试代理的支持, 且调试代理在系统模式下运行。

1. Tornado 工具

开发环境 Tornado 包含的基本工具如下:

- 交叉调试器——CrossWind;

- shell 工具——WindSh;
- 配置工具——WindConfig;
- 对象装载/卸载器;
- 符号表管理;
- 用户自定义工具。

可以通过 TCL (Tool Command Language) 语言对这些工具进行修改和功能扩充, 而这些工具并不是直接与目标机系统通信, 它们通过宿主机上的目标机服务器 (target server) 平台访问目标机系统。

当调试目标机系统时, 目标机与目标机服务器的通讯通常是靠目标机代理 (WDB 调试代理) 来实现, 如图 13.6 所示的主机与目标机的通讯。此时的 WDB 调试代理作为一个 VxWorks 系统任务运行, 并独立于 VxWorks 内核。

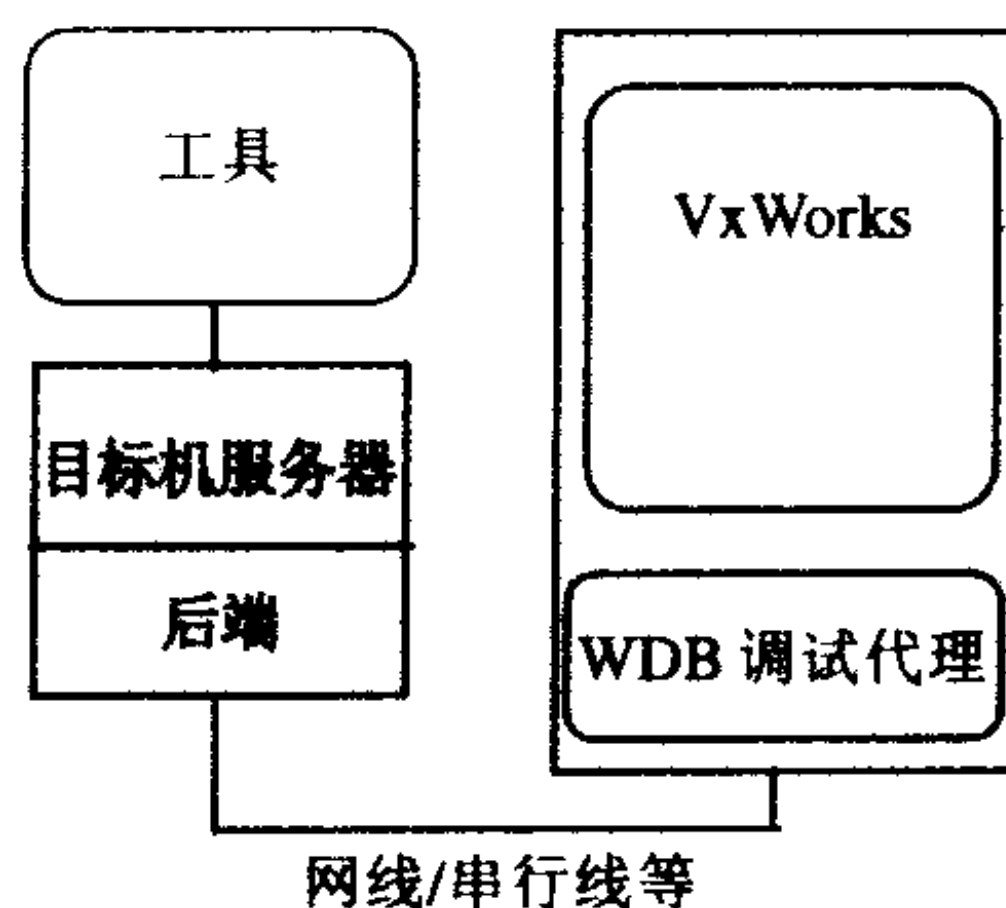


图 13.6 主机与目标机的通讯

2. 启动 WDB 调试代理

在内核启动前, Tornado 工具可能需要借助于 WDB 调试代理进行 BSP 初始化代码的调试。这时, 必须要启动 WDB 调试代理, 并且 WDB 调试代理要在系统模式下执行。这就要求系统在这个阶段满足下列条件:

(1) 在 sysHwInit() 执行后启动 WDB 调试代理。WDB 调试代理与宿主机之间进行通讯, 需要一个通讯环境的支持。而 sysHwInit() 正好执行了系统必要设备初始化操作和硬件中断屏蔽, 为 WDB 调试代理提供了这个环境。

(2) 系统模式调试的支持。由于打开硬件中断是在内核启动后才进行, 这就要求系统此时的设备驱动程序为轮询方式。而 sysHwInit() 执行后, 提供了轮询访问设备的驱动程序。

这时, 系统调试模式将允许 BSP 开发人员通过交叉调试工具 CrossWind 进行下列代码的调试:

- 中断处理程序的调试;
- 预内核设备初始化的调试。

13.4.2 使用 WDB 调试代理

1. 概述

在内核启动前, 如果开发环境支持 Tornado 工具访问目标机系统, 则需要下列其中一个条件的

支持:

- NetROM;
- 用户自配置的 ICE;
- 串行接口。

在上面的条件中, 最容易满足且又节省开发成本的就是串行接口。所以无论是购买的开发板还是自己设计的开发板通常会带有串口。

WDB 调试代理必须根据宿主机与目标机之间的通讯硬件环境进行相应的配置, 如果使用串行接口, 需要进行如下操作:

- 串口必须配置成轮询 (poll) 方式访问串口设备;
- 修改 `usrInit()` 函数, 增加初始化 WDB 和停止 WDB 执行操作。

2. 配置 WDB 调试代理

使用 WDB 调试代理进行代码调试需要对 WDB 进行配置, 有关详细信息如下:

(1) 背景的详细说明。配置 WDB 调试代理需要进行下列详细说明。

- 宿主机与目标机连接方式 (例如串口连接)。
- 代理执行模式 (系统级模式还是任务级模式)。

(2) `config.h` 文件。配置 WDB 调试代理主要是通过修改 `config.h` 文件来进行。在这个文件里定义了配置 WDB 的一些宏定义, 例如通讯类型宏定义 `WDB_COMM_TYPE`。

(3) 宿主机与目标机通讯方式。宿主机与目标机通讯方式的宏定义包含在 `configAll.h` 文件中, 如下所示:

```
#define WDB_COMM_NETWORK      0    /* 网络接口 */
#define WDB_COMM_SERIAL      1    /* 串口 */
#define WDB_COMM_TYCODRV_5_2  2    /* 老版本串口驱动程序连接 */
#define WDB_COMM_ULIP        3    /* vxSim 设备 */
#define WDB_COMM_NETROM      4    /* NetROM 接口 */
#define WDB_COMM_CUSTOM      5    /* 自定义设备接口 (例如 ICE) */
#define WDB_COMM_END         6    /* END 网络接口 */
#define WDB_COMM_PIPE        7    /* 管道设备 */
```

其中, `WDB_COMM_SERIAL`、`WDB_COMM_NETROM`、`WDB_COMM_CUSTOM` 可用于内核启动前宿主机与目标机通讯连接。配置过程则需要在 `config.h` 文件中增加通讯方式宏定义, 宏定义如下 (以串行接口为例):

```
#define WDB_COMM_TYPE WDB_COMM_SERIAL /* 连接方式为串行线 */
```

(4) 调试模式选择。WDB 调试代理调试模式分为任务模式、系统模式和双重模式。这些宏定义包含在 `../target/h/wdb/wdb.h` 文件中, 如下所示:

```
#define WDB_MODE_TASK      1    /* 任务模式 */
#define WDB_MODE_EXTERN    2    /* 系统模式 */
#define WDB_MODE_DUAL (WDB_MODE_TASK | WDB_MODE_EXTERN) /* 双重模式 */
```

如果要调试预内核的初始化代码, WDB 调试代理的调试模式必须配置成系统模式。具体配置在 `config.h` 文件中增加调试模式宏定义, 宏定义如下:


```
#define WDB_MODE          WDB_MODE_EXTERN      /* 调试模式的宏定义为系统模式*/
```

(5) 运行时 WDB 调试代理的配置。运行时 WDB 调试代理的配置是通过调用 wdbConfig()函数来完成。

3. 启动 WDB 调试代理

配置完 WDB 调试代理后,要启动调试代理。下面介绍了启动 WDB 的时机以及启动和配置 WDB 调试代理的函数:

(1) WDB 调试代理必须在 usrInit()函数中进行配置和启动,且在 sysHwInit()函数执行后才可以进行。

(2) 配置和启动函数。配置和启动 WDB 调试代理需要调用下面两个函数。

- wdbConfig(), 配置 WDB 调试代理的函数。
- wdbSuspendSystemHere(), 启动 WDB 调试代理的函数。

(3) wdbConfig()函数。wdbConfig()函数主要是建立 WDB 调试代理的上下文并返回 (OK 或 ERROR)。

(4) wdbSuspendSystemHere()函数。wdbSuspendSystemHere()函数启动 WDB 调试代理,实际上是把 CPU 的控制权转交给 WDB 调试代理。

一旦启动 WDB 调试代理后, Tornado 下的工具可以访问目标机,进行代码调试或系统测试等操作。

4. usrInit()函数的修改

在 usrConfig.c 文件中包含了 usrInit()函数源代码,需要在 usrInit()函数中插入一些代码进行 WDB 调试代理的配置和启动,如下代码所示:

```
=====
...
    sysHwInit(); /* 系统硬件初始化操作 */

/* 配置和启动 WDB 调试代理 */
    wdbConfig();
    wdbSuspendSystemHere(NULL, 0);

    usrKernelInit(); /* 配置 Wind 内核 */
    ...
-----
```

修改 usrInit()函数后,需要注释 usrConfig.c 文件中 usrRoot()函数里的 wdbConfig()函数调用,如下所示:

```
...
/* 初始化 WDB 调试代理 */
/*
#ifdef INCLUDE_WDB
    wdbConfig();
*/
...
-----
```


13.4.3 基于串口的 WDB 调试代理

1. 基于串行接口后端的系统级调试

(1) WDB 调试代理的代码调用。对于在系统模式下的串行接口后台操作，WDB 调试代理程序将调用如下函数：

- 轮询串口驱动程序的输入/输出函数；
- 访问串行接口设备的配置函数。

(2) 如果目标机的 BSP 包含了串行接口设备的支持，仅仅只要在 config.h 文件中进行如下配置：

```
#define WDB_COMM_TYPE WDB_COMM_SERIAL      /* 连接方式为串口 */
#define WDB_MODE      WDB_MODE_EXTERN      /* 调试模式宏定义为系统模式 */
#undef INCLUDE_WDB_VIO                      /* 虚拟 I/O */
```

(3) 如果目标机的 BSP 不包含串行接口设备的支持，则需要开发串行接口驱动程序。

2. WDB 调试代理与串行接口驱动程序

如果在内核启动前，串行接口支持 WDB 调试代理在系统模式下调试初始化代码，则需要进行下列操作：

- 初始化串口 I/O 控制结构；
- 调用 ioctl() 函数设置轮询方式访问串口；
- 调用轮询输入/输出函数；
- 确定用于调试的串口通道（指 WDB 使用的通道）。

为了满足上面的一些操作和条件，需要对一些文件进行修改，这些文件包括：

- ../target/src/drv/sio/templateSio.c
- ../target/config/bspname/sysSerial.c
- ../target/h/drv/sio/templateSio.h

而支持 WDB 调试代理基于串口进行代码调试的支撑函数将被 sysSerialHwInit()、wdbConfig()、wdbSuspendSystemHere() 函数所调用。

3. 串口驱动程序的结构

每一个串口设备都有一个控制结构，下面是一个串口驱动程序结构的示例：

```
typedef struct          /* I8250_CHAN */
{
    SIO_DRV_FUNCS * pDrvFuncs;      /* 驱动程序函数 */

    STATUS          (*getTxChar) (); /* 指向发送函数的指针 */
    STATUS          (*putRcvChar) (); /* 指向接收函数的指针 */
    void *          getTxArg;
    void *          putRcvArg;

    UINT16 int_vec;                  /* 中断向量号 */
    UINT16 channelMode;              /* SIO_MODE_[INT | POLL] (通道模式) */
    UCHAR  (*inByte) (int);          /* 从寄存器读一个字节的函数 */
    void   (*outByte) (int, char);    /* 往寄存器写一个字节的函数 */

    ULONG lcr;                       /* UART 线控制寄存器 */
    ULONG lst;                       /* UART 线状态寄存器 */
}
```



```

        ULONG    mdc;                /* UART modem 控制寄存器 */
        ULONG    msr;                /* UART modem 状态寄存器 */
        ULONG    ier;                /* UART 中断开启寄存器 */
        ULONG    iid;                /* UART 中断 ID 寄存器 */
        ULONG    brdl;               /* UART 波特率寄存器 */
        ULONG    brdh;               /* UART 波特率寄存器 */
        ULONG    data;               /* UART 数据寄存器 */
        ULONG    options;            /* UART 硬件选择 */
    } I8250_CHAN;

```

在通道控制结构中，都会有一个 SIO_DRV_FUNCS（或 SIO_CHAN）结构作为通道控制结构的第一个成员。这个结构将在下面进行描述。

4. 设备控制结构

每一类设备都有它自己的设备结构，下面是包含两个串口通道（port0 和 port1）的设备结构示例：

```

typedef struct
{
    I8250_CHAN    port0;              /* I8250 的通道 */
    I8250_CHAN    port1;
} I8250_DUSART;

```

5. SIO_DRV_FUNCS 结构

SIO_CHAN 实际上是一个指向 SIO_DRV_FUNCS 结构的指针。SIO_DRV_FUNCS 结构有 5 个成员，这些成员进行串口设备的管理。WDB 调试代理将会调用这些成员。

下面是 SIO_DRV_FUNCS 的结构：

```

struct sio_drv_funcs                /* 驱动程序函数 */
{
    int (*ioctl1)(SIO_CHAN *pSioChan, int cmd, void * arg);

    int (*txStartup)(SIO_CHAN *pSioChan);

    int (*callbackInstall)(SIO_CHAN * pSioChan, int callbackType,
        STATUS (*callback)(), void * callbackArg );

    int (*pollInput)( SIO_CHAN *    pSioChan,char *    inChar);

    int (*pollOutput)(SIO_CHAN *    pSioChan, char outChar);
};

```

6. 串行接口设备的初始化

在使用串口设备前，需要对串口设备控制结构和串口设备进行初始化操作。初始化过程是调用 sysSerialHwInit() 函数来实现。如下所示（以 8250 串行控制芯片为例）：

```

=====
sysSerialHwInit() 函数
=====
/*****
*

```



```

* sysSerialHwInit - 初始化 BSP 中的串行设备到一静止状态。
*
* 该函数初始化 BSP 中的串行设备描述符，并设置设备到一静止状态。
* 处于关中断情况下，在 sysHwInit() 函数中调用该函数。
*
* 返回: N/A
*
* 参考: sysHwInit()
*/

```

```

void sysSerialHwInit (void)
{
    int i;

    /* 串口驱动程序结构项的初始化 */
    for (i=0; i<N_UART_CHANNELS; i++)
    {
        i8250Chan[i].int_vec = devParas[i].vector;
        i8250Chan[i].channelMode = 0;
        i8250Chan[i].lcr = UART_REG(UART_LCR, i);
        i8250Chan[i].data = UART_REG(UART_RDR, i);
        i8250Chan[i].brdl = UART_REG(UART_BRDL, i);
        i8250Chan[i].brdh = UART_REG(UART_BRDH, i);
        i8250Chan[i].ier = UART_REG(UART_IER, i);
        i8250Chan[i].iid = UART_REG(UART_IID, i);
        i8250Chan[i].mdc = UART_REG(UART_MDC, i);
        i8250Chan[i].lst = UART_REG(UART_LST, i);
        i8250Chan[i].msr = UART_REG(UART_MSR, i);

        i8250Chan[i].outByte = sysOutByte;
        i8250Chan[i].inByte = sysInByte;

        /* 控制结构和串口芯片的初始化 */
        i8250HrdInit(&i8250Chan[i]);
    }
}

```

注意

在 sysSerial.c 文件中，需要进行串口驱动程序的结构声明。例如：

```
static I8250_CHAN i8250Chan[N_UART_CHANNELS];
```

7. 控制结构和串行接口芯片的初始化

控制结构的初始化是指驱动程序结构中第一项 SIO_DRV_FUNCS 结构的初始化，而芯片的初始化指串行接口寄存器的初始化。

下面是一个控制结构和串行接口芯片的初始化示例：

```

=====
i8250HrdInit()
=====

```



```

static SIO_DRV_FUNCS i8250SioDrvFuncs;
...
void i8250HrdInit
(
    I8250_CHAN * pChan      /* 指向串行设备的指针 */
)
{
    if (i8250SioDrvFuncs.ioctl == NULL)
    {
        i8250SioDrvFuncs.ioctl      = (int (*)(int))i8250Ioctl;
        i8250SioDrvFuncs.txStartup  = (int (*)(int))i8250Startup;
        i8250SioDrvFuncs.callbackInstall = i8250CallbackInstall;
        i8250SioDrvFuncs.pollInput  = (int (*)(int))i8250PRxChar;
        i8250SioDrvFuncs.pollOutput = (int (*)(int, char))i8250PTxChar;
    }

    /* 初始化控制结构 */
    pChan->pDrvFuncs = &i8250SioDrvFuncs;

    /* 初始化串口芯片 */
    i8250InitChannel(pChan);

    /* 设置串口通讯方式为轮询模式 */
    i8250Ioctl (pChan, SIO_MODE_SET, SIO_MODE_POLL);
}

```

8. 串口设备的设置

在 VxWorks 中，串口设备的设置通常包括波特率和模式的设置，可以调用 `ioctl()` 函数进行串口设备的设置。而事实上，`ioctl()` 函数最终是调用 `SIO_DRV_FUNCS` 结构中的 `int (*ioctl)` 函数（例如 `i8250Ioctl`）。设置命令如下：

- `SIO_BAUD_SET`，波特率设置命令；
- `SIO_MODE_SET`，模式设置命令。

其中模式设置包含两种情况，一种是轮询模式 `SIO_MODE_POLL`，另一种为中断模式 `SIO_MODE_INT`。

9. 通道访问

初始化串口设备后，可以对串口设备进行访问。但在访问前，需要对系统进行一些配置和处理。这些配置和处理如下所示。

(1) 确定通道个数。在 `config.h` 文件中包含了串口通道的个数配置宏，如下所示：

```
#define NUM_TTY      2      /* Sio 串口通道个数 */
```

(2) 确定 WDB 调试代理使用的串口通道。在系统中使用配置宏 `WDB_TTY_CHANNEL` 确定 WDB 调试代理使用的串口通道。系统默认值为 1，如下所示：

```
#define WDB_TTY_CHANNEL 1      /* 调试使用的默认 Sio 串行通道 */
```

(3) 获得 `SIO_CHAN` 结构。配置一个串行通道给 WDB 调试代理，必须首先获得这个通道的 `SIO_CHAN` 结构。这个结构可以通过调用 `sysSerialChanGet()` 函数来获得。在 `wdbConfig()` 函数中调用了这个函数，其中通道参数为 `WDB_TTY_CHANNEL`。

小 结

在 BSP 的职责中，初始化过程极为重要，毫无疑问也是 BSP 开发人员最“棘手”的地方。针对这一情况，在本章中，详细介绍了 VxWorks 预内核的初始化。其内容包括初始化概述、特殊初始化函数（例如 romInit()、romstart()、sysInit()）的详细分析、通用初始化代码的分析、RAM 和 ROM 的布局、系统硬件初始化以及预内核的初始化调试和内核启动等。这些内容可以说是 BSP 代码中最重要的部分，希望读者能够仔细阅读从而对 BSP 的开发能够达到运用自如的地步。

第 14 章 BSP 的配置

用户在开发目标机的 BSP 之前, 必须了解在 VxWorks 操作系统下 BSP 的配置, 只有在熟悉 BSP 的配置后才可以正确对 BSP 进行移植和修改工作, 并根据自己的需求成功配置一个 BSP。

14.1 BSP 的基本配置

这一部分主要对目标机硬件环境及硬件相关的参数进行配置。

14.1.1 ROM 和 RAM 的布局

用户可以根据目标机的具体情况对 ROM 和 RAM 的空间布局进行配置, 配置过程主要是通过修改地址宏定义的值而达到目的, 在 config.h 文件中定义了这些宏, 主要有 ROM_TEXT_ADRS、ROM_SIZE、RAM_LOW_ADRS、RAM_HIGH_ADRS 等, 具体配置如下所示:

```
/*
 * 在 Makefile 文件中定义了如下一些参数。
 * 这些参数在 config.h 中的定义值必须与 Makefile 中定义的值相一致。
 */

#ifdef BOOTCODE_IN_RAM
#define ROM_BASE_ADRS      0x00008000      /* ROM 基地址 */
#define ROM_TEXT_ADRS      (ROM_BASE_ADRS) /* 启动点 */
#define ROM_SIZE           0x00090000      /* ROM 空间大小 */
#else
#define ROM_BASE_ADRS      0xfff20000      /* ROM 基地址 */
#define ROM_TEXT_ADRS      (ROM_BASE_ADRS) /* EPROM 启动点 */
#define ROM_SIZE           0x0007fe00      /* ROM 空间大小 */
#endif

#define RAM_LOW_ADRS        0x00108000      /* VxWorks 映像入口点 */
#define RAM_HIGH_ADRS       0x00008000      /* 引导映像入口点 */
-----
```

14.1.2 内存配置

内存的配置包括用户驻留程序的内存基地址、局部总线内存基地址、内存基地址以及内存大小等, 具体配置如下所示:

```
/* 内存地址 */

/* 用户保留内存。 阅读 sysMemTop() */

#define USER_RESERVED_MEM 0

/*
```



```

* 本地到总线内存地址常量:
* 本地内存地址总是为 0:
* 它并非双口内存。
*/

#define LOCAL_MEM_LOCAL_ADRS    0x00000000    /* 目标机本地 RAM 起始地址 */
#define LOCAL_MEM_BUS_ADRS      0x00000000    /* 总线本地内存起始地址 */
#define LOCAL_MEM_SIZE          0x00800000    /* 本地 RAM 大小为 8MB */

/*
* 当定义这个选项时, 支持自动分配内存大小。在这种情况下系统忽视 LOCAL_MEM_SIZE。
* 相关信息请参考 sysyPhysMemTop() 函数。
*/

#define LOCAL_MEM_AUTOSIZE
-----

```

14.1.3 启动设备及参数配置

启动设备及参数配置主要包含引导参数的配置, 即从哪里下载 VxWorks 映像、系统热启动后从什么设备上启动系统以及设备相关的其他参数配置等。如下所示:

1. 默认引导参数配置

下面以目标机通过 Intel Ether Express PRO100B PCI 接口的网卡, 从宿主机 “rainbow” 上下载 vxWorks 映像为例进行默认引导参数配置说明。有关该参数的详细说明请参考相关书籍。

```

#define DEFAULT_BOOT_LINE \
"fei(0,0)rainbow:vxWorks h=10.17.13.119 e=10.17.13.120 u=user pw=user tn=user"

```

2. 热启动设备和参数配置

/* 热启动(重新启动)设备和参数 */

```

#define SYS_WARM_BIOS            0            /* 从 BIOS 热启动 */
#define SYS_WARM_FD              1            /* 从软盘热启动 */
#define SYS_WARM_ATA             2            /* 从 ATA 硬盘热启动 */
#define SYS_WARM_TFFS            3            /* 从片上系统 (DiskOnChip) 热启动 */

#define SYS_WARM_TYPE            SYS_WARM_FD  /* 热启动设备 */
#define SYS_WARM_FD_DRIVE        0            /* 0 = 驱动器 a:, 1 = b: */
#define SYS_WARM_FD_TYPE         0            /* 0 = 3.5" 2HD, 1 = 5.25" 2HD */
#define SYS_WARM_ATA_CTRL        0            /* 控制器 0 */
#define SYS_WARM_ATA_DRIVE       0            /* 0 = c:, 1 = d: */
#define SYS_WARM_TFFS_DRIVE      0            /* 0 = c: (DOC) */

```

14.1.4 外围设备驱动程序及相关参数配置

外围设备驱动程序配置和相关参数配置描述了在配置文件中如何配置外围设备, 这些设备包括软盘驱动器、硬盘、并行端口、时钟、协处理器等驱动程序的配置。其中还包含一些相关参数的配置, 例如设备寄存器基地址、中断级等。具体一些配置如下:

1. 配置 TrueFFS 驱动程序

```

#undef INCLUDE_TFFS /* 包含 TrueFFS 驱动程序 */

```


2. 配置软盘驱动程序

```
#define INCLUDE_FD          /* 包含软盘驱动程序 */

/* 软盘驱动器 (FD) 相关参数宏定义, 见文件 pc.h */

#define FD_INT_LVL          0x06      /* 中断号 */
#define FD_DMA_BUF_ADDR    0x2000    /* 软驱 DMA 缓冲区地址 */
#define FD_DMA_BUF_SIZE    0x3000    /* 软驱 DMA 缓冲区大小 */
```

3. 并行端口驱动程序配置

```
#define INCLUDE_LPT          /* 包含并行端口驱动程序 */

/* 并行端口 (LPT) 相关参数宏定义 */

#define LPT0_BASE_ADRS     0x3bc     /* 并行端口 0 寄存器基地址 */
#define LPT1_BASE_ADRS     0x378     /* 并行端口 1 寄存器基地址 */
#define LPT2_BASE_ADRS     0x278     /* 并行端口 2 寄存器基地址 */
#define LPT_INT_LVL        0x07     /* 中断号 */
#define LPT_CHANNELS        1        /* 并行通道 */
```

4. 时钟配置

```
/*
 * SYS_CLK_RATE_MAX 依赖于 CPU 的功率, 应用程序可以对它进行控制。
 */
```

(1) 配置辅助时钟。

```
#define INCLUDE_AUX_CLK
#define AUX_CLK_RATE_MIN 2          /* 辅助时钟最小 tick 数 */
#define AUX_CLK_RATE_MAX 8192      /* 辅助时钟最大 tick 数 */
```

(2) 配置系统时钟。

```
#define INCLUDE_SYSCLK_INIT
#define SYS_CLK_RATE_MIN 19          /* 系统时钟最小 tick 数 */
#define SYS_CLK_RATE_MAX (PIT_CLOCK/256) /* 系统时钟最大 tick 数 */
#define SYS_CLK_RATE 60              /* 系统时钟 tick 数 */
```

(3) 配置时间戳。

```
#undef INCLUDE_TIMESTAMP /* 包含时间戳定时器驱动程序, 以便提供给 Wind View */
```

(4) 时钟相关地址定义。

```
/* 定时器 (PIT) */

#define PIT_BASE_ADR      0x40      /* 定时器寄存器基地址 */
#define PIT0_INT_LVL      0x00      /* 中断级 */
#define PIT_REG_ADDR_INTERVAL 1
#define PIT_CLOCK          1193180 /* 定时器频率 */

/* 实时时钟 (RTC) */

#define RTC_INDEX          0x70      /* 实时时钟控制寄存器地址 */
#define RTC_DATA           0x71      /* 实时时钟数据寄存器地址 */
```



```
#define RTC_INT_LVL      0x08      /* 中断级 */
```

5. 浮点处理支持

(1) 硬浮点支持。

```
#if      (CPU_FAMILY==I80x86)
#define INCLUDE_I80387      /* I80387 浮点协处理器 */
#define INCLUDE_HW_FP      /* 硬浮点处理器支持 */
#endif /* CPU_FAMILY==I80x86 */
```

(2) 软浮点支持。

```
#define INCLUDE_SW_FP      /* 软件浮点仿真 */
```

6. 硬盘配置

通过下面的宏定义可以配置系统支持 ATA/IDE 接口的硬盘。

```
#define INCLUDE_ATA      /* 包括 IDE/EIDE(ATA) 接口硬盘驱动程序 */
/* 硬盘(IDE)相关参数定义 */
```

```
#define IDE_CONFIG      0x0      /* 1: 使用 IDE 类型 */
#define IDE_INT_LVL      0x0e      /* 中断级 */
```

```
/* 硬盘(ATA)相关参数定义 */
```

```
#define ATA0_IO_START0      0x1f0      /* ATA0 的 IO 地址 */
#define ATA0_IO_STOP0      0x1f7
#define ATA0_IO_START1      0x3f6
#define ATA0_IO_STOP1      0x3f7
#define ATA0_INT_LVL      0x0e
#define ATA0_CONFIG      (ATA_GEO_CURRENT | ATA_PIO_AUTO | \
ATA_BITS_16 | ATA_PIO_MULTI)

#define ATA1_IO_START0      0x170      /* ATA1 的 IO 地址 */
#define ATA1_IO_STOP0      0x177
#define ATA1_IO_START1      0x376
#define ATA1_IO_STOP1      0x377
#define ATA1_INT_LVL      0x09 /* 9 分配给 PCIC, 5 分配给 TCIC */
#define ATA1_CONFIG      (ATA_GEO_CURRENT | ATA_PIO_AUTO | \
ATA_BITS_16 | ATA_PIO_MULTI)

#define ATA_SEM_TIMEOUT      5      /* ATA 同步信号超时 */
#define ATA_WDG_TIMEOUT      5      /* ATA 看门狗定时器超时 */
```

7. 串行控制器支持

```
#define INCLUDE_SIO
#define NUM_TTY      2      /* 串行通道数 */
#define CONSOLE_TTY      0      /* 显控台通道 */
#define CONSOLE_BAUD_RATE      9600 /* 显控台(console)波特率 */
```

```
/* 串行端口(COM1,COM2) */
```

```
#define COM1_BASE_ADR      0x3f8      /* 串行端口 COM1 寄存器基地址 */
#define COM2_BASE_ADR      0x2f8      /* 串行端口 COM2 寄存器基地址 */
#define COM1_INT_LVL      0x04      /* 串行端口 COM1 中断级 */
#define COM2_INT_LVL      0x03      /* 串行端口 COM2 中断级 */
```



```
#define UART_REG_ADDR_INTERVAL 1
#define N_UART_CHANNELS 2
```

14.1.5 文件系统配置

VxWorks 系统中的文件系统是作为设备的驱动程序安装在系统中的，它使用与设备驱动程序相同的标准 I/O 但是同时它又是个独立的系统。这样的设计使得一个 VxWorks 应用可以使用多种文件系统而不发生冲突。同时根据这些标准接口协议，用户还可以根据自己的需要编写文件系统，并将文件系统与相关的驱动程序相结合。它包含与 MS-DOS 系统兼容的文件系统，dosFs 文件系统，原始文件系统，rawFsLib 文件系统，磁带文件系统，tapeFs 文件系统等。配置如下所示：

```
#define INCLUDE_DOSFS /* 包含 dosFs 文件系统 */
#undef INCLUDE_CDROMFS /* 包含 cdromFs 文件系统 */
#define INCLUDE_TAPEFS /* 包含 tapeFs 文件系统 */
```

14.1.6 网络驱动程序配置

在 VxWorks 网络系统运转中，以太网是其中的一个介质，也是众多厂商所支持的一个局域网规范。它是大多数 VxWorks 网络应用中理想的选择，但是某些协议除外，譬如 BOOTP 和 DHCP，在 VxWorks 或主机网络系统中它们固有地约束了以太网。

至于以太网驱动程序的配置，Wind River 提供了一些厂商的以太网驱动程序，其中包括 3COM、AMD、Intel 等公司的网卡设备驱动程序，通常以库文件或源代码的形式安装在 Tornado 目录下，用户通过配置后可以把它们包含到系统中去，一些驱动程序配置如下所示：

```
/* 网络驱动程序的选择 */
#define INCLUDE_END /* 使用增强型网络驱动程序 */

#undef INCLUDE_ULTRA /* 包含 SMC Elite16 Ultra 接口驱动程序 */
#undef INCLUDE_ENE /* 包含 Eagle/Novell NE2000 接口驱动程序 */
#undef INCLUDE_ELT /* 包含 3COM EtherLink III 接口驱动程序 */
#undef INCLUDE_ESMC /* 包含 SMC 91c9x 以太网接口驱动程序 */
#define INCLUDE_FEI /* 包含 Intel Ether Express PRO100B PCI 接口驱动程序 */
#define INCLUDE_SLIP /* 包含 SLIP 接口驱动程序 */
#define SLIP_TTY 1 /* SLIP 通道为 COM2 */
#undef INCLUDE_ELC /* 包含 SMC Elite16 接口驱动程序 */
#undef INCLUDE_EEX /* 包含 INTEL EtherExpress 接口驱动程序 */
#undef INCLUDE_EEX32 /* 包含 INTEL EtherExpress flash 32 接口驱动程序 */
#undef INCLUDE_EX /* 包含 Excelan 以太网接口驱动程序 */
#undef INCLUDE_ENP /* 包含 CMC 以太网接口驱动程序 */
#undef INCLUDE_SM_NET /* 包含底板 (backplane) 网络接口 */
#undef INCLUDE_SM_SEQ_ADDR /* 网络共享内存地址 */
#undef INCLUDE_EL_3C90X_END /* 3com 快速以太网连接器 XL PCI */
#undef INCLUDE_LN_97X_END /* AMD 79C972 END 驱动程序 */
```

14.1.7 高速缓存配置

如果硬件支持且系统又需要使用高速缓存，则需要对高速缓存进行配置。配置过程及相关要求、注意事项等说明如下：

```
/*
 * 高速缓存配置
```



```

*
* 所有的高速缓存首先是关闭的。在 BSP 的基本功能完成后，才激活高速缓存。
* 高速缓存打开的步骤是首先打开指令高速缓存，接下来才打开数据高速缓存。
* 高速缓存最先处于非监听工作下；打开它们后，如果目标板支持才可以监听。
*
* 注意：当 MMU 打开时，通过 sysPhysMemDesc[] 中的 MMU 表入口控制高速缓存，
* 而不是通过下面的高速缓存模式宏定义来控制。
* /

```

与高速缓存相关的配置宏如下：

```

#define USER_I_CACHE_MODE  CACHE_WRITETHROUGH  /* 指令高速缓存模式 */
#define USER_D_CACHE_MODE  CACHE_WRITETHROUGH  /* 数据高速缓存模式 */
#define USER_I_CACHE_ENABLE  /* 打开指令高速缓存 */
#define USER_D_CACHE_ENABLE  /* 打开数据高速缓存 */
#define USER_B_CACHE_ENABLE  /* 打开分支(branch) 高速缓存 */
#define USER_I_MMU_ENABLE    /* 选择指令 MMU */
#define USER_D_MMU_ENABLE    /* 选择数据 MMU */
-----

```

14.1.8 堆栈配置

在 BSP 的配置过程中，堆栈的配置也是应关心的问题。因为系统需要使用堆栈，而用户有时也需要保留一些堆栈来存放一些重要信息等。所以要对 VxWorks 系统如何布局堆栈要有所了解。相关的配置信息如下：

```

#if (CPU_FAMILY==I960)
#define STACK_SAVE      512      /* 预留的最大堆栈数 */
#else /* sparc 或其他体系结构 */
#if ((CPU_FAMILY==SPARC) || (CPU_FAMILY==PPC))
#define STACK_SAVE      0x1000
#else /* 所有其他体系结构 */
#define STACK_SAVE      0x40      /* 预留的最大堆栈数 */
#endif
#endif

#if ((CPU_FAMILY == MIPS) || (CPU_FAMILY == PPC))
#define STACK_RESIDENT   RAM_DST_ADRS
#else
#define STACK_RESIDENT   _sdata
#endif

#if (_STACK_DIR == _STACK_GROWS_DOWN)

#ifdef ROM_RESIDENT
#define STACK_ADRS       STACK_RESIDENT
#else
#define STACK_ADRS       _romInit
#endif /* ROM_RESIDENT */

#else /* _STACK_DIR == _STACK_GROWS_UP */

#ifdef ROM_RESIDENT
#define STACK_ADRS       (STACK_RESIDENT-STACK_SAVE)
#else

```



```
#define STACK_ADRS      (_romInit-STACK_SAVE)
#endif    /* ROM_RESIDENT */

#endif    /* _STACK_DIR == _STACK_GROWS_UP */
```

14.1.9 目标机通用配置

```
/* 设备控制器 I/O 地址 */
```

```
#define IO_ADRS_EI      ((char *) 0x000fff00)    /* 32A,32D i82596CA 以太网 */
```

14.2 VxWorks 的基本配置

BSP 的职责包含了初始化操作，在初始化过程中包含了操作系统 VxWorks 的初始化。这一部分初始化主要是 VxWorks 系统资源的数据结构初始化，例如包含类的初始化、任务对象的初始化、信号量的初始化、消息队列的初始化等。

而 VxWorks 系统资源的数据结构初始化有相当多的一部分资源是可选配置，可根据用户的需要配置 VxWorks 系统。VxWorks 的配置头文件为 config/all/configAll.h 和 config/bspname/config.h。当配置 VxWorks 的初始化时，程序 usrConfig.c、bootConfig.c、bootInit.c 调用了这些文件。在开发环境中，用户可能要测试几种不同的配置，或者用户想在不同情况下指明不同的目标代码。为了编译 VxWorks 满足不同情况，用户必须调整配置。

14.2.1 内核软件配置

内核软件配置包括符号表、空闲 RAM 起始地址、异步 I/O 任务、最大驱动程序个数、最大打开文件数、串行通过数、内核和任务系统参数等配置，配置宏详细描述如下：

```
#define SYM_TBL_HASH_SIZE_LOG2      8    /* 符号哈希表的记录数为 256 */
#define STAT_TBL_HASH_SIZE_LOG2     6    /* 状态哈希表的记录数为 64 */
#define MQ_HASH_SIZE                 0    /* POSIX 消息队列哈希表的大小，0 为默认值 */
#define NUM_SIGNAL_QUEUES            16   /* POSIX 排队信号数 */

#if CPU==SIMNT
#define FREE_RAM_ADRS simMemBlock
#else
#define FREE_RAM_ADRS (end)    /* 空闲 RAM 起始地址在 VxWorks 的 BSS 段结束处 */
#endif

/* I/O 系统参数 */

#define NUM_DRIVERS      20    /* 驱动程序表中最大驱动个数 */
#define NUM_FILES        50    /* 同时打开最大文件个数 */
#define NUM_DOSFS_FILES  20    /* 打开最大 dosFs 文件个数 */
#define NUM_RAWFS_FILES   5    /* 打开最大 rawFs 文件个数 */
#define NUM_RT11FS_FILES  5    /* 打开最大 rt11Fs 文件个数 */
#define MAX_LOG_MSGS      50   /* 最大日志记录个数 */

#define NUM_TTY           2    /* 串行通道个数 */
#define CONSOLE_TTY       0    /* 显控台通道 */
```



```

#define CONSOLE_BAUD_RATE 9600      /* 显控台波特率 */

#ifdef INCLUDE_SLIP
#define SLIP_TTY 1 /* 串口线接口协议通道 */
#endif /* INCLUDE_SLIP */

#define MAX_LIO_CALLS 0 /* 最大异步 I/O 个数, 默认值为 0 */
#define MAX_AIO_SYS_TASKS 0 /* 最大异步 I/O 系统任务数, 默认值为 0 */
#define AIO_TASK_PRIORITY 0 /* 异步 I/O 系统任务优先级, 默认值为 0 */
#define AIO_TASK_STACK_SIZE 0 /* 异步 I/O 任务堆栈大小, 默认值为 0 */

/* 体系结构的内核和系统任务参数 */

#if CPU_FAMILY==MC680X0
#define INT_LOCK_LEVEL 0x7 /* 68k 中断屏蔽码 */
#define ROOT_STACK_SIZE 10000 /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE 10000 /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE 0x1000 /* WDB 调试代理堆栈大小 (字节) */
#define ISR_STACK_SIZE 1000 /* ISR 堆栈大小 (字节) */
#define TRAP_DEBUG 2 /* trap 2 - 断点 trap */
#define VEC_BASE_ADRS ((char *) LOCAL_MEM_LOCAL_ADRS) /* 向量表基地址 */
#endif /* CPU_FAMILY==MC680X0 */

#if CPU_FAMILY==SPARC
#define INT_LOCK_LEVEL 15 /* SPARC 中断屏蔽级 */
#define ROOT_STACK_SIZE 10000 /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE 50000 /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE 0x2000 /* WDB 调试代理堆栈大小 (字节) */
#define ISR_STACK_SIZE 10000 /* ISR 堆栈大小 (字节) */
#define VEC_BASE (LOCAL_MEM_LOCAL_ADRS + 0x1000)
#define VEC_BASE_ADRS ((char *) VEC_BASE) /* 向量表基地址 */
#endif /* CPU_FAMILY==SPARC */

#if CPU_FAMILY==SIMNT || CPU_FAMILY==SIMSPARCSUNOS || CPU_FAMILY==SIMHPPA ||
CPU_FAMILY==SIMSPARCSOLARIS
#define INT_LOCK_LEVEL 0x1 /* 中断屏蔽码 */
#define ROOT_STACK_SIZE 20000 /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE 50000 /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE 0x2000 /* WDB 调试代理堆栈大小 (字节) */
#define ISR_STACK_SIZE 50000 /* ISR 堆栈大小 (字节) */
#define VEC_BASE_ADRS 0 /* 虚拟的 */
#endif /* CPU_FAMILY == SIMNT || CPU_FAMILY==SIMSPARCSUNOS || CPU_FAMILY==SIMHPPA
|| CPU_FAMILY==SIMSPARCSOLARIS */

#if CPU_FAMILY==I960
#define INT_LOCK_LEVEL 0x1f /* i960 中断屏蔽码 */
#define ROOT_STACK_SIZE 20000 /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE 40000 /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE 0x2000 /* WDB 调试代理堆栈大小 (字节) */
#define ISR_STACK_SIZE 1000 /* ISR 堆栈大小 (字节) */
#define TRAP_DEBUG 0 /* 80960 为空 */
#define VEC_BASE_ADRS NONE /* 向量表基地址 */
#endif /* CPU_FAMILY==I960 */

```



```

#if CPU_FAMILY==MIPS
#define INT_LOCK_LEVEL      0x1          /* R3K 中断屏蔽码 */
#define ROOT_STACK_SIZE    (20000)      /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE   (20000)      /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE     (0x2000)     /* WDB 调试代理堆栈大小 (字节) */
#define ISR_STACK_SIZE     (5000)       /* ISR 堆栈大小 (字节) */
#define VEC_BASE_ADRS      ((char *) 0x0) /* 对于 R3k 无意义 */
#define VME_VECTORED       FALSE        /* VME 中断向量 */
#define TRAP_DEBUG         0            /* trap 0 - 断点 trap */
#endif /* CPU_FAMILY==MIPS */

#if CPU_FAMILY==PPC
#define INT_LOCK_LEVEL      0x0          /* 没有使用 */
#define ROOT_STACK_SIZE    (20000)      /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE   (20000)      /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE     0x2000      /* WDB 调试代理堆栈大小 (字节) */
#define ISR_STACK_SIZE     (5000)       /* ISR 堆栈大小 (字节) */
#define VEC_BASE_ADRS      ((char *) 0x0) /* 向量表基地址 */
#endif /* CPU_FAMILY==PPC */

#if CPU_FAMILY==I80x86
#define INT_LOCK_LEVEL      0x0          /* 80x86 中断屏蔽码 */
#define ROOT_STACK_SIZE    10000        /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE   10000        /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE     0x1000      /* WDB 调试代理堆栈大小 (字节) */
#define ISR_STACK_SIZE     1000        /* ISR 堆栈大小 (字节) */
#define TRAP_DEBUG         0            /* 没有使用 */
#define VEC_BASE_ADRS      ((char *) LOCAL_MEM_LOCAL_ADRS)
#endif /* CPU_FAMILY==I80x86 */

#if CPU_FAMILY==AM29XXX
#define INT_LOCK_LEVEL      0x0001      /* 29k 所有中断屏蔽码 */
#define ROOT_STACK_SIZE    10000        /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE   40000        /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE     0x2000      /* WDB 调试代理堆栈大小 (字节) */
#define ISR_STACK_SIZE     10000        /* ISR 堆栈大小 (字节) */
#define TRAP_DEBUG         15          /* trap 2 - 断点 trap */
#define VEC_BASE_ADRS      ((char *) LOCAL_MEM_LOCAL_ADRS)
#endif /* CPU_FAMILY==AM29XXX */

#if CPU_FAMILY==ARM
#define INT_LOCK_LEVEL      0            /* 中断屏蔽码- 没有使用 */
#define ROOT_STACK_SIZE    0x4000      /* 根任务堆栈大小 (字节) */
#define SHELL_STACK_SIZE   0x10000     /* shell 堆栈大小 (字节) */
#define WDB_STACK_SIZE     0x2000      /* WDB 调试代理堆栈大小 (字节) */
#define TRAP_DEBUG         0            /* 没有使用 */
#define VEC_BASE_ADRS      ((char *) LOCAL_MEM_LOCAL_ADRS)
/*
 * 注意: ISR_STACK_SIZE 在 config.h 中定义, 而不是在这里, 因为由 BSP 中断结构决定。
 */
#endif /* CPU_FAMILY==ARM */

```


14.2.2 内核软件工具配置

内核软件工具的配置包括语言的支持、任务的同步与通讯、网络系统、调试代理、时钟管理等配置，有关内核软件工具配置宏描述如下：

```
#define INCLUDE_CACHE_SUPPORT      /* 包含 cache 支持包 */
#define INCLUDE_CONSTANT_RDY_Q     /* 固定插入就绪队列时间 */
#define INCLUDE_CPLUS              /* 包含 C++ 支持 */
#define INCLUDE_CPLUS_IOSTREAMS    /* 包含基本 I/O 流 */
#define INCLUDE_CPLUS_STL          /* 包含标准模板核心库 */
#define INCLUDE_ENV_VARS           /* unix 兼容性环境变量 */
#define INCLUDE_EXC_HANDLING       /* 包含基本异常处理 */
#define INCLUDE_EXC_TASK           /* 异常任务支持 */
#define INCLUDE_FLOATING_POINT     /* 浮点 I/O 支持 */
#define INCLUDE_FORMATTED_IO       /* 磁盘 I/O 支持 */
#define INCLUDE_GCC_FP             /* gcc 浮点支持库 */
#define INCLUDE_IO_SYSTEM          /* 包含 I/O 系统 */
#define INCLUDE_LOGGING            /* 包含日志任务 - logTask */
#define INCLUDE_MEM_MGR_FULL       /* 完成内存管理 */
#define INCLUDE_MSG_Q              /* 包含消息队列 */
#define INCLUDE_NETWORK            /* 网络子系统支持 */
#define INCLUDE_PIPES              /* 包含管道驱动程序 */
#define INCLUDE_SELECT             /* select() 功能的支持 */
#define INCLUDE_SEM_BINARY         /* 包含二值信号量 */
#define INCLUDE_SEM_MUTEX          /* 包含互斥信号量 */
#define INCLUDE_SEM_COUNTING       /* 包含计数信号量 */
#define INCLUDE_SIGNALS            /* 软信号库支持 */
#define INCLUDE_STDIO              /* 标准 I/O 支持 */
#define INCLUDE_TASK_HOOKS         /* 任务钩子函数的支持 */
#define INCLUDE_TASK_VARS          /* 任务变量包 */
#define INCLUDE_TIMEX              /* 支持操作计时工具 */
#define INCLUDE_TTY_DEV            /* 绑定串行端口驱动程序 */
#define INCLUDE_WATCHDOGS          /* 包含看门狗定时器 */
#define INCLUDE_WDB                /* WDB 调试代理支持 */
#define INCLUDE_ANSI_ALL           /* 包含完整的 ANSI C 库函数 */
#define INCLUDE_ANSI_ASSERT        /* ANSI-C 声明库函数 */
#define INCLUDE_ANSI_CTYPE         /* ANSI-C ctype 库函数 */
#define INCLUDE_ANSI_LOCALE        /* ANSI-C locale 库函数 */
#define INCLUDE_ANSI_MATH          /* ANSI-C math 库函数 */
#define INCLUDE_ANSI_STDIO         /* ANSI-C stdio 库函数 */
#define INCLUDE_ANSI_STDLIB        /* ANSI-C stdlib 库函数 */
#define INCLUDE_ANSI_STRING        /* ANSI-C string 库函数 */
#define INCLUDE_ANSI_TIME          /* ANSI-C time 库函数 */
```

14.2.3 内核软件扩展工具配置

内核软件扩展工具的配置包括测试工具 CodeTest、调试工具、Java 虚拟机、HTML、对象模块装载器等，有关内核软件扩展工具配置宏描述如下：

```
#define INCLUDE_CODETEST           /* CodeTEST 目标机应用库 */
#define INCLUDE_CPLUS_VXW         /* 包含 VxWorks wrapper 类 */
#define INCLUDE_CPLUS_TOOLS       /* 包含工具类库 */
#define INCLUDE_CPLUS_STRING      /* 包含字符串类 */
```



```

#define INCLUDE_CPLUS_STRING_IO      /* 包含字符串 I/O */
#define INCLUDE_CPLUS_COMPLEX        /* 包含复数类 */
#define INCLUDE_CPLUS_COMPLEX_IO     /* 包含复数 I/O */
#define INCLUDE_CPLUS_IOSTREAMS_FULL /* 包含所有 I/O 流 */
#define INCLUDE_CONFIGURATION_5_2    /* tornado 之前的工具 */
#define INCLUDE_DEBUG                 /* tornado 之前的调试 */
#define INCLUDE_DEMO                  /* 包含 shell 的简单演示 */
#define INCLUDE_DOSFS                 /* dosFs 文件系统 */
#define INCLUDE_HTML                  /* 包含 HTML 支持 */
#define INCLUDE_JAVA                  /* Java 虚拟机支持 */
#define INCLUDE_LOADER                /* 对象模块装载支持 */

/* INCLUDE_MMU_BASIC 在 config.h 文件中定义。如 INCLUDE_MMU_FULL 在 configAll.h
 * 文件中定义且 INCLUDE_MMU_BASIC 在 config.h 文件中定义，则 INCLUDE_MMU_FULL 优先。
 */
#define INCLUDE_MMU_BASIC             /* 捆绑基本 MMU 支持 */
#define INCLUDE_MMU_FULL              /* (没有捆绑) 完整 MMU 支持 */

#define INCLUDE_SYM_TBL_SYNC          /* 主机和目标机符号表同步 */
#define INCLUDE_POSIX_AIO             /* POSIX 异步 I/O 支持 */
#define INCLUDE_POSIX_AIO_SYSDRV      /* POSIX 异步 I/O 系统驱动程序 */
#define INCLUDE_POSIX_FTRUNC          /* POSIX ftruncate 函数支持 */
#define INCLUDE_POSIX_MEM             /* POSIX 内存锁支持 */
#define INCLUDE_POSIX_MQ              /* POSIX 消息队列支持 */
#define INCLUDE_POSIX_SCHED           /* POSIX 调度支持 */
#define INCLUDE_POSIX_SEM             /* POSIX 信号量 */
#define INCLUDE_POSIX_SIGNALS         /* POSIX 排队等待信号 */
#define INCLUDE_POSIX_TIMERS          /* POSIX 定时器 */
#define INCLUDE_PROTECT_TEXT          /* 正文段写保护 (没有捆绑) */
#define INCLUDE_PROTECT_VEC_TABLE     /* 向量表写保护 (没有捆绑) */
#define INCLUDE_RAMDRV               /* 原始磁盘驱动程序 */
#define INCLUDE_RAWFS                 /* rawFs 文件系统 */
#define INCLUDE_RBUFF                 /* 环行缓冲区库支持 */
#define INCLUDE_RT11FS                /* rt11Fs 文件系统 */
#define INCLUDE_SECURITY              /* shell 安全网络访问 */
#define INCLUDE_SHELL                 /* 交互式的 C 表达解释程序 */
#define INCLUDE_SHOW_ROUTINES         /* 系统工具显示程序 */
#define INCLUDE_SM_OBJ               /* 共享内存对象 (没有捆绑) */
#define INCLUDE_SPY                   /* 任务监控监视库支持 */
#define INCLUDE_STANDALONE_SYM_TBL    /* 经过编译的对象文件包含符号表 */
#define INCLUDE_STARTUP_SCRIPT        /* 启动脚本有效 */
#define INCLUDE_STAT_SYM_TBL          /* 构造用户易读的错误状态 */
#define INCLUDE_SW_FP                 /* 软浮点仿真 */
#define INCLUDE_SYM_TBL               /* 符号表包 */
#define INCLUDE_UGL                   /* 包含通用图形库支持 */
#define INCLUDE_UNLOADER              /* 对象模块卸载支持 */
#define INCLUDE_WILLOWS_RT            /* 包含 Willows RT 库 */
#define INCLUDE_WDB_TSFS              /* target-server 文件系统支持 */
#define INCLUDE_VXFUSION              /* WindMP (没有捆绑) 支持 */
#define INCLUDE_WINDVIEW              /* WindView 目标机应用库 */
#define INCLUDE_WVUPLOAD_ALL          /* 包含所有的 Windview 上载路径类型 */
#define INCLUDE_WVUPLOAD_FILE         /* Windview 上载路径为文件方式 */
#define INCLUDE_WVUPLOAD_SOCKET       /* Windview 上载路径为 socket 方式 */

```



```

#define INCLUDE_WVUPLOAD_TSFS SOCK /* Windview 上载路径为 TSFS sock 方式 */

#define INCLUDE_ANSI_5_0 /* 包含仅仅只是 5.0 版本的 ANSI 支持 */
#define INCLUDE_DELETE_5_0 /* 定义与 VxWorks 5.0 里一样的 delete() 功能 */

#define INCLUDE_TSFS_BOOT /* 通过 TSFS(Target Server File System)方式引导 */
#define INCLUDE_TSFS_BOOT_VIO_CONSOLE /* TSFS 引导介质是串行端口 */
-----

```

14.3 双硬盘配置

在 VxWorks 的 BSP 默认配置中, 硬盘控制器和硬盘驱动程序最大数量默认配置分别为 2。具体宏定义在文件 `ataDrv.h` 中, 如下所示:

```

#define ATA_MAX_CTRLIS 2 /* ATA 控制器最大数量 */
#define ATA_MAX_DRIVES 2 /* ATA 驱动程序最大数量 */

```

如果由于目标机系统的需要, 要求在目标机上“挂”两块 ATA 接口的硬盘, 且都挂在 0 号硬盘控制器上, 同时在硬盘上安装 dosFs 文件系统。其中一块硬盘 (硬盘 A) 逻辑驱动号为“C:”; 另一块硬盘 (硬盘 B) 逻辑驱动号为“D:”。

1. 资源的重新配置

在 BSP 默认配置中, 一个硬盘控制器默认情况下只支持一个驱动号。而系统需要在 0 号硬盘控制器上支持两个驱动号, 所以需要对硬盘所配置的资源进行重新配置。其重新配置过程只要对 `sysLib.c` 文件中的 `ataResources[]` 结构进行修改即可。如下所示 (修改的地方用粗体+阴影字表示):

```

ATA_RESOURCE ataResources[ATA_MAX_CTRLIS] =
{
    {
        {
            5, 0,
            {ATA0_IO_START0, ATA0_IO_START1}, {ATA0_IO_STOP0, ATA0_IO_STOP1}, 0,
            0, 0, 0, 0, 0
        },
        IDE_LOCAL, ATA0_INT_VEC, ATA0_INT_LVL, ATA0_CONFIG,
        ATA_SEM_TIMEOUT, ATA_WDG_TIMEOUT, 0, 0
    }, /* ctrl 0 (0 号硬盘控制器) */
    {
        {
            5, 0,
            {ATA1_IO_START0, ATA1_IO_START1}, {ATA1_IO_STOP0, ATA1_IO_STOP1}, 0,
            0, 0, 0, 0, 0
        },
        ATA_PCMCIA, 1, ATA1_INT_VEC, ATA1_INT_LVL, ATA1_CONFIG,
        ATA_SEM_TIMEOUT, ATA_WDG_TIMEOUT, 0, 0
    } /* ctrl 1 (1 号硬盘控制器) */
};
-----

```


**注意**

有关 ATA_RESOURCE 数据结构原型，其定义的原型在..\target\h\drv\pcmcia\pccareLib.h 文件中。

2. 文件系统的安装及硬盘驱动号的分配

对资源进行重新配置后，接下来需要在硬盘上安装 DOS 文件系统及硬盘驱动号的分配。而这一过程是通过调用 usrAtaConfig() 函数来实现。其具体过程如下：

(1) 硬盘 A 驱动号的分配及文件系统的安装。通过执行 usrAtaConfig(0,0,"C:") 操作，给硬盘 A 分配驱动号为 "C:"，并安装文件系统。

(2) 硬盘 B 驱动号的分配及文件系统的安装。通过执行 usrAtaConfig(0,1,"D:") 操作，给硬盘 B 分配驱动号为 "D:"，并安装文件系统。

经过上述配置后系统支持两块硬盘，但是上述只是简单给出了配置步骤，有关上面的一些数据结构和函数原型请读者阅读相应的源代码和数据结构定义。

14.4 双网卡配置

14.4.1 ISA 总线网卡配置

若目标系统需要使用两块同类型的 ISA 总线网卡，需要设置网卡的 I/O 地址、中断级等。以 NE2000 网卡为例，配置目标机系统支持两块 NE2000 网卡，其中一块网卡的 I/O 基地址为 0x320、中断级为 0x05；另一块网卡的 I/O 基地址为 0x300、中断级为 0x08。具体操作过程如下。

1. 网卡的 I/O 地址、中断级的分配

对于 ISA 总线的网卡需要指定它的 I/O 地址和中断级，其宏定义在 config.h 文件中。以 NE2000 网卡为例，对 config.h 文件内容修改如下：

```
...
#define IO_ADRS_ENE          0x320
#define INT_LVL_ENE          0x05
#define INT_VEC_ENE          (INT_NUM_IRQ0 + INT_LVL_ENE)

#define IO_ADRS_ENE1         0x300
#define INT_LVL_ENE1         0x08
#define INT_VEC_ENE1         (INT_NUM_IRQ0 + INT_LVL_ENE1)
...
```

2. 网络设备表 endDevTbl[] 的修改

与其他类型设备驱动的安装方式不一样，网络设备驱动的安装是根据 confignet.h 文件里的网络设备表 (endDevTbl[]) 中的定义，系统逐一调用各网络设备的装载函数来实现网络设备驱动程序的安装。以 NE2000 网卡为例，对 endDevTbl[] 表的修改如下：

```
END_TBL_ENTRY endDevTbl [] =
{
```



```

...
#ifdef INCLUDE_ENE_END
{0, END_ENE_LOAD_FUNC, END_ENE_LOAD_STRING, END_ENE_BUFF_LOAD, NULL, FALSE},
{1, END_ENE_LOAD_FUNC, END_ENE_LOAD_STRING, END_ENE_BUFF_LOAD, NULL, FALSE},
#endif /* INCLUDE_ENE_END */
...
}

```

经过上述修改后，表示系统支持两块 NE2000 网卡。



注意

有关 END_TBL_ENTRY 数据结构，其定义原型在..\target\h\end.h 中。

3. 网卡设备装载函数的修改

以 NE2000 网卡装载函数为例，需要修改 sysNe2000End.c 文件中的设备装载函数 sysNe2000EndLoad()。修改如下：

```

END_OBJ * sysNe2000EndLoad (char * pParamStr, void * unused )
{
    ...
    if (strlen (pParamStr) == 0)
    {
        pEnd = ne2000EndLoad (pParamStr);
    }
    else
    {
        ...
        cp += strlen (paramStr);
        /* finish off the initialization parameter string */
        if (pParamStr[0] == '0')
            sprintf (cp, ne2000ParamTemplate,
                    IO_ADRS_ENE,
                    INT_VEC_ENE,
                    INT_LVL_ENE,
                    ENE_BYTE_ACCESS,
                    ENE_USE_ENET_PROM,
                    ENE_OFFSET);
        else if (pParamStr[0] == '1')
            sprintf (cp, ne2000ParamTemplate,
                    IO_ADRS_ENE1,
                    INT_VEC_ENE1,
                    INT_LVL_ENE1,
                    ENE_BYTE_ACCESS,
                    ENE_USE_ENET_PROM,
                    ENE_OFFSET);
        if ((pEnd = ne2000EndLoad (paramStr)) == (END_OBJ *)ERROR)
        { printf ("Error: NE2000 device failed ne2000EndLoad routine.\n"); }
    }
    return (pEnd);
}

```


4. IP 堆栈、IP 地址及接口数量配置

(1) 接口数量配置。VxWorks 系统对网络接口最大数量默认值为 1，而目标机系统如果使用两块网卡，则需要对这个参数进行配置，即在 VxWorks 的配置窗口上将 IP_MAX_UNITS 设置为 2。

(2) 给网卡分配 IP 堆栈。通过调用 ipAttach() 函数给网卡分配 TCP/IP 堆栈。例如调用 ipAttach(1, "ene"), 表示给第二块网卡分配一个 IP 堆栈。

(3) 给网卡分配 IP 地址。通过调用 usrNetIfConfig() 函数给网卡分配 IP 地址和主机名。例如调用 usrNetIfConfig ("ene", 1, "192.192.192.3", "rainbow", 0), 表示给第二块网卡分配 IP 地址和主机名。

经过上述配置后系统支持两块网卡，但是上述只是简单给出了配置步骤，用户可以根据上述配置构造一个演示程序来验证系统是否成功配置。

14.4.2 PCI 总线双网卡配置

若目标系统需要支持两块同类型的 PCI 总线网卡，则需要对系统进行配置，其配置比 ISA 总线网卡要简单。以 Realtek 81x9 以太网卡为例，具体操作过程如下。

1. 有关源文件

首先需要获得 RTL81x9 END 驱动程序的源文件，然后根据源文件的自述文件 readme 将文件夹 config、h、src 下的文件拷贝到指定的目录下，且覆盖掉原有文件。

2. 网络设备表 endDevTbl[] 的修改

对 confignet.h 文件中的 endDevTbl[] 表的修改如下：

```
END_TBL_ENTRY endDevTbl [] =
{
    ...
    #ifdef INCLUDE_RTL_81X9_END
    {0, RTL_81X9_LOAD_FUNC, RTL_81X9_LOAD_STR_0, TRUE, NULL, FALSE},
    {1, RTL_81X9_LOAD_FUNC, RTL_81X9_LOAD_STR_0, TRUE, NULL, FALSE},
    #endif /* INCLUDE_RTL_81X9_END */
    ...
}
```

经过上述修改后，表示系统支持两块 RTL81x9 网卡。

3. IP 堆栈、IP 地址及接口数量配置

(1) 接口数量配置。VxWorks 系统对网络接口最大数量默认值为 1，而使用两块网卡。所以需要配置这个参数，即在 VxWorks 的配置窗口上将 IP_MAX_UNITS 设置为 2。

(2) 给网卡分配 IP 堆栈。通过调用 ipAttach() 函数给网卡分配 TCP/IP 堆栈。例如调用 ipAttach(1, "rtl"), 表示给第二块网卡分配一个 IP 堆栈。

(3) 给网卡分配 IP 地址。通过调用 usrNetIfConfig() 函数给网卡分配 IP 地址和主机名。例如调用 usrNetIfConfig ("rtl", 1, "192.192.192.3", "rainbow", 0), 表示给第二块网卡分配 IP 地址和主机名。

经过上述配置后系统支持两块 RTL81x9 网卡，但是上述只是简单给出了配置步骤，用户可以根据上述配置构造一个演示程序，然后在主机下执行 ping 192.192.192.3 来验证系统是否成功配置。

小 结

配置 BSP 所面临的难度虽然不及开发一个 BSP 所遇到的问题，但了解如何在 VxWorks 操作系统下进行 BSP 的配置也是开发人员不能忽视的方面，只有熟悉 BSP 的配置后才可以正确移植和修改 BSP 以及根据自己的需求成功配置一个 BSP。在这一章里，从四个方面进行配置说明，即 BSP 的基本配置、VxWorks 的基本配置，和提高篇的双硬盘及双网卡的配置。

第 15 章 板级支持包开发

在开发目标机的板级支持包之前，开发者从总体上应对板级支持包的开发流程有所了解，同时在开发过程中所应具备的开发环境和所需要的辅助工具等也应该有所准备。这样的目的有助于在开发板级支持包时做到有计划、有步骤的进行，缩短开发周期、提高开发效率等。对于这些问题在本章中进行描述。

15.1 板级支持包开发周期概述

板级支持包的开发过程是阶段性的，每下一个阶段的开始都依赖于前一阶段的开发情况。总体上而言，板级支持包的开发过程大致如下：

- (1) 尽量获得一个与目标板硬件环境相近的参考板级支持包和相关代码模板。
- (2) 开发环境的准备。
- (3) 编写 VxWorks 预内核的初始化代码。
- (4) 提供一个轮询访问串口的驱动程序，以便 Tornado 可以访问目标机。
- (5) 一旦启动 VxWorks 内核，则需要连接系统中断。
- (6) 启动系统时钟。

(7) 根据目标板硬件环境，完成板级支持包包含的驱动程序及设备初始化，从而完善板级支持包的开发。

- (8) 板级支持包的测试和文档的编写（例如使用手册）。

至于板级支持包开发的有关详细资料参考板级支持包的开发环境及目标板本身的一些资料和需求。

1. 参考板级支持包及其模板

在进行板级支持包开发时，首先的工作是寻找参考的板级支持包。寻找的途径有下列几种方法：

- 向 Wind River 公司咨询。随着嵌入式系统在各行各业的普及以及产品更新换代的加速，产品研发周期必然缩短。为了满足产品开发商的需求，Wind River 公司通常会根据用户目标板的体系结构为用户提供一个参考板级支持包。
- 向 CPU 生产厂商咨询。一些 CPU 生产厂商为了展示自己的 CPU 处理能力，有时会以该 CPU 为核心设计一款目标板，这款目标板包含了 CPU 所有能够处理的能力。通常把这款目标板称为开发板。用户可以从厂商那里购买自己所需要的开发板（带有板级支持包），进行用户自己的产品设计和开发工作。
- 通过 Internet，寻找所需要的板级支持包。当通过上述两种方法不能获得需要的参考板级支

持包时，用户可以通过 Internet 来搜寻需要的资料。

总之，尽可能的找到一些可参考的资料，来辅助板级支持包的开发。

(1) 参考板级支持包。VxWorks 之所以能够在实时操作系统市场上占有绝对的优势，其中有一点就是 Wind River 公司提供了许多参考板级支持包供用户选择。有了参考的板级支持包可以减少产品开发时间，提供开发效率。当用户根据宿主机上的开发环境和自己的目标机体系结构（指 CPU）购买 Tornado 和 VxWorks 时，实际上就已经获得了参考板级支持包。因为参考板级支持包自带在 Tornado 开发环境里。但是，通用的设备驱动程序（例如串口、网络接口等）可能不包含。

(2) 板级支持包的模板文件。当用户自己开发板级支持包时，可以借助板级支持包模板文件来辅助用户自己的板级支持包开发。用户可以通过购买 Wind River 公司的板级支持包开发包来获得模板文件。

在板级支持包开发包里不仅包含所有体系结构的板级支持包（例如 x86、Mips、68k、i960、PPC、Sparc 等体系结构的板级支持包），而且包含了许多通用设备驱动程序（例如 DMA、网络接口、PCI、串口、并口、中断控制器、定时器等）的源程序。

2. 板级支持包的开发环境

开发环境的优势也决定了板级支持包开发的进度。开发环境在总体上主要由下面几个部分组成：

- 下载代码到目标机上的装备；
- 测试和调试代码的装备。

况且，板级支持包处于不同的开发阶段，用户需要选择不同的开发工具。主要分为两个阶段：

- VxWorks 内核启动前，初始化阶段需要的开发环境；
- VxWorks 内核启动后，将需要访问 Tornado 的一些开发工具。

至于板级支持包的开发环境这一部分，将在后面章节中进行详细描述。其内容也包括板级支持包开发过程中需要的硬件设备和软件工具等。

3. 预内核的开发环境

内核启动前这一阶段的板级支持包开发是整个板级支持包开发过程中一个难点，也是一个重点。开发者通常需要借助一些工具来辅助板级支持包的开发。而且，这些工具在价格上都比较贵。从这一点上就不难理解内核启动前板级支持包开发的重要性。这些辅助开发工具通常包括以下内容。

(1) 通用的装载工具，如下所示：

- 目标机厂家的调试 ROM；
- ROM 仿真器；
- 在线仿真器（ICE, In-Circuit Emulator）。

(2) 通用的调试工具，如下所示：

- 目标机厂家的调试 ROM；
- ICE；
- 逻辑分析仪；
- 发光二极管 LED；
- Tornado 下的一些工具。

15.2 板级支持包的开发环境

上述一些小节简单描述了开发环境的重要性，在本节里将详细描述开发环境具体包含的内容。

在内核启动前，对于这一阶段的板级支持包开发必须搭建一个开发环境，通常这一阶段的开发工具提供下载和调试功能，例如 ICE、目标机厂商调试 ROM。有时，也需要一些配套工具辅助开发者的开发，例如 ROM 仿真器，且需要单独提供调试工具。同时还有其他一些工具，例如逻辑分析仪、目标机状态指示灯（LED）等。

1. 在线仿真器 ICE

在板级支持包开发阶段，ICE 是一个非常好的辅助开发工具，它具有如下一些特性。

(1) 替代目标机处理器（指 CPU）。使用 ICE 时，仿真头替代目标机处理器。且仿真头与仿真器是通过一根数据线相连接。仿真器包含如下的仿真功能：

- 仿真指定的体系结构；
- 仿真处理器本质的功能。

(2) 监控或信号注入。通过监控或注入信号到目标系统，可以访问处理器总线。而监控或信号注入可以通过如下单元实现：

- 仿真器的处理器；
- 仿真器的电路。

(3) 代码执行及追踪。仿真器允许代码在 CPU 的全速下执行，同时也允许追踪竞态条件（不可预知的情况）。

(4) 调试功能。ICE 提供调试功能，其功能包括：

- 源码级的调试；
- 中断仿真。

(5) ICE 调试扩展特性。ICE 的调试功能包括断点设置、单步执行、内存查看、变量查看等一系列功能。

(6) 存储映射。大多数 ICE 包含存储映射功能，允许把目标机的 ROM 和 RAM 映射到 ICE 上。

(7) 代码下载。ICE 系统通常支持通过并口或网络接口下载映像到目标机的 RAM 中或 Flash 中。

(8) 其他工具的捆绑。ICE 为了增强自己的功能，通常捆绑一些工具。

- 逻辑状态分析仪；
- 逻辑时间分析仪；
- 脉冲和图形发生器；
- 多路追踪功能；
- 友好的用户图形接口；
- 调试器的集成；
- 总线访问功能；
- 事件 ID 和隔离功能；

- 实时追踪和过滤。

(9) 主要问题。ICE 功能非常强大, 对于板级支持包开发人员来说, 它主要存在如下两个问题:

1) 仿真器必须与目标机体系结构一致。不同的目标机体系结构所使用的 ICE 必须与它相匹配。例如目标机是 8086 体系结构, 所使用的 ICE 必须也是 8086 结构的, 不能使用 80386 结构的 ICE。

2) 价格昂贵。ICE 的价格一般比较昂贵, 动辄就是几十万。对于小项目的开发, 不是一笔小数目, 毫无疑问增加了开发成本。

(10) 开发环境 Tornado 与 ICE。在 Tornado 开发环境下, 集成了两种 ICE。

- vision ICE;
- Trace32 ICE。

这两种产品提供了以太网下载功能, 支持几种通用的体系结构, 并且为实时系统提供了一些特性。为了能够很好的使用 ICE 功能, 目标机服务器后端为 ICE 提供了能够在内核启动前使用 Tornado 的一些工具的功能。

2. ROM 仿真器

ROM 仿真器提供了宿主机通过 ROM 插座 (socket) 访问目标机的功能。它具有如下一些特性:

(1) 替换目标机 ROM。在使用 ROM 仿真器时, 首先撤走目标机的 ROM, 然后把仿真器的夹子插到目标机 ROM 插座上。且仿真器与宿主机通过串口或网络进行连接。

(2) 通用性。ROM 仿真器并不依赖于目标机的体系结构, 只要目标机有 ROM 插座, 就可以使用 ROM 仿真器。

(3) 存储功能。ROM 仿真器含有存储器, 它可以替代目标机的 ROM。这样, 可以把宿主机上的代码下载到仿真器的存储器中, 同时仿真器的存储器也是目标机存储器的一部分。

(4) 代码执行。VxWorks 映像可以在 ROM 仿真器中执行, 通常是 ROM 驻留型映像, 在 ROM 仿真器中执行, 而基于 ROM 的映像, 在目标机内存中执行。

(5) 不包含调试功能。ROM 仿真器提供代码下载和执行功能, 但不包含调试工具。而代码的调试且通过逻辑分析仪或代码中的调试代理来实现。

(6) 调试信息传递。大多数 ROM 仿真器提供通信路径和协议, 以便在目标机和宿主机之间传递调试信息。

(7) 开发环境 Tornado 与 ROM 仿真器。在 Tornado 开发环境下, 用户可以使用 NetROM 仿真器。它本身的特性如下:

- 1) 把带有 ROM 的目标机转换成网络接点。
- 2) 1MB 的存储空间。
- 3) 通过 TFTP 或 TCP 协议下载代码。
- 4) 2KB 的双端口 RAM 作为调试通信通道。

3. 厂家调试 ROM

在一些目标板上, 厂家提供了一个调试 ROM。它具有如下一些特性。

(1) 开发软件的固化。目标机调试 ROM 固化了开发软件, 它包含如下一些功能:

- 动态装载代码;

- 提供下载路径;
- 跳转到指定的地址和执行功能;
- 调试工具的集成;
- 目标机诊断功能。

(2) 跳线功能。目标机环境可能带有跳线, 允许目标机从 ROM 或 NVRAM 启动。

(3) 代码下载。调试 ROM 中的开发环境提供了下载代码功能, 下载方式和存放地点如下:

- 通过串口或网络接口下载;
- 把下载型 VxWorks 映像装载到 RAM 中;
- 把 ROM 型的 VxWorks 映像通常装载到 Flash 中。

(4) 通讯设备驱动程序的提供。调试 ROM 具有下载功能, 那么调试 ROM 里的代码必然包含宿主机与目标板之间的通讯接口设备驱动程序 (例如串口或网络接口)。

(5) 调试/诊断工具。调试 ROM 里的开发环境包含如下一些调试和诊断工具。

- 断点设置和单步执行;
- 内存检查和修改;
- 环境参数设置 (总线时钟速度等);
- 目标机自动检测。

4. WDB 调试代理和 Tornado 工具

在板级支持包开发阶段, 也可以借助 WDB 调试代理和 Tornado 工具辅助板级支持包开发。WDB 与 Tornado 工具之间的通讯如图 15.1 所示。

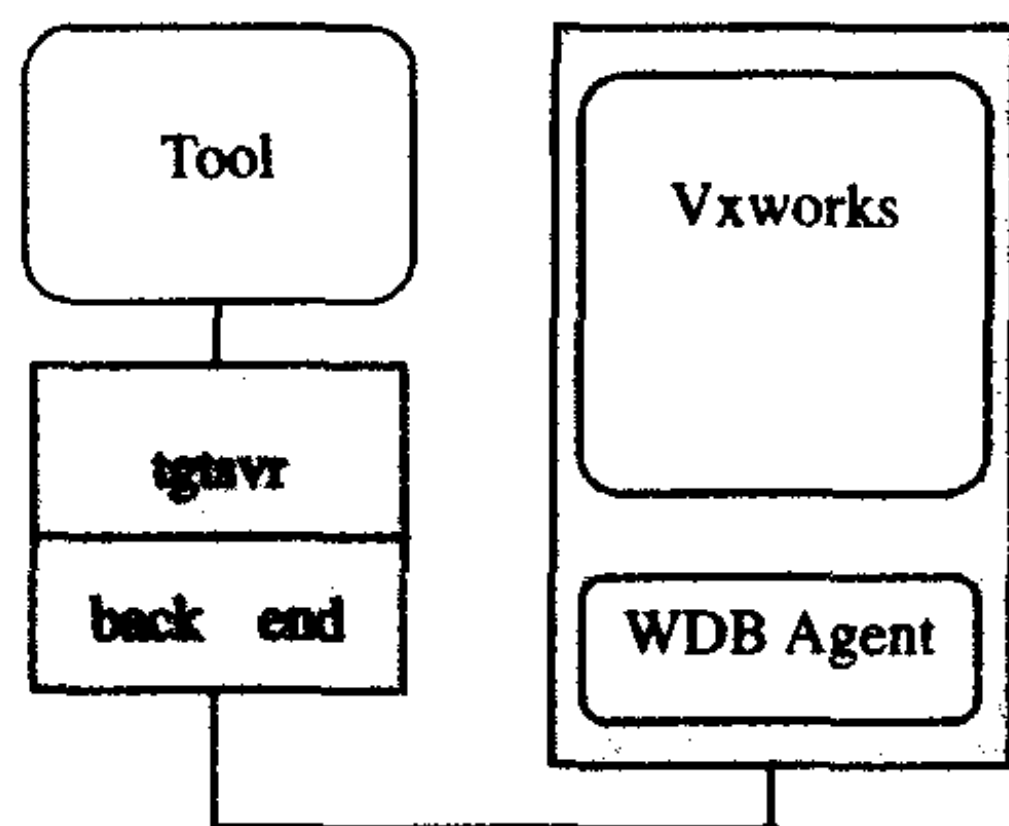


图 15.1 主机与目标机的通讯

(1) WDB 调试代理。WDB 调试代理是 VxWorks 的一个扩展模块。不仅可以用于板级支持包开发阶段, 而且在应用程序调试阶段也是一个主要调试工具模块。用户可以根据自己的需要把它配置到自己的 VxWorks 映像中, 它具有一些特性:

- 代理和操作系统的一系列初始化;
- 作为操作系统的共享库, 以便操作系统减少代码空间;
- 整个调试环境的支持;
- 可以在任务级 (内核启动后) 或系统级下执行。

(2) target server。在内核启动前使用下面的一些开发工具, 作为板级支持包的开发环境且需要 target server 的支持。

- NetROM;
- 串行线;
- ICE。

小 结

在这一章中，从宏观的角度描述板级支持包的开发流程和所需要的开发环境以及开发过程中可能使用的辅助工具等。对于这些问题的描述，目的是为了读者在开发板级支持包之前对板级支持包的开发步骤有所了解，和开发过程中需要配备哪些辅助开发工具，缩短开发周期、提高开发效率。

第 16 章 创建一个新的 BSP

16.1 概述

VxWorks 虽然为许多体系结构的目标机提供了 BSP 模板,但是这毕竟是一个模板,用户还需根据自己具体的目标机硬件环境编写一个新的 BSP。而新建 BSP 的过程可以基于开发环境 Tornado 下的一些辅助开发工具,按照一定的步骤分阶段的一步一步地进行开发,但是每一步的开发都是建立在前一步的开发基础上,步骤如下:

- (1) 安装一个基本的 BSP 开发环境。
- (2) 编写 VxWorks 内核启动前的初始化代码。
- (3) 启动一个最小的 VxWorks 内核,然后向其逐个添加中断控制器、串口设备、定时器等基本驱动程序。
- (4) 启动目标机代理(指 WDB),并与 Tornado 开发工具连接。
- (5) 完善新的 BSP 开发。主要包括网络接口、引导 ROMS、SCSI、cache、MMU 和 DMA 的初始化。
- (6) 用户基于新的 BSP 新建一个默认工程。

本章将对上述列出的每一个开发步骤进行详细描述。在这一章里,目标不仅是指导用户如何创建一个新的 BSP,而且还要让用户在最短的时间里,了解和使用 Tornado 下的一些开发工具,特别是目标机调试代理——WDB。因为 WDB 调试代理是内核的一个扩展模块,可以把它连接到内核中,与内核共享初始化代码。这样,当初始化代码执行后,可以启动 WDB 调试代理,也可以启动 VxWorks 内核,或者两者都启动。例如基于 WDB 调试代理调试内核启动前的初始化代码时,只启动 WDB 调试代理;而在调试应用程序时,既启动了 WDB 调试代理也启动了 VxWorks 内核。

使用 WDB 调试代理,既可以调试内核启动前的初始化代码,也可以调试包含 WDB 调试代理的 VxWorks 映像。与传统的 ROM-monitor 不同,ROM-monitor 首先需要在目标板上固化一个 ROM-monitor,然后才装载操作系统。

与传统的 ROM-monitor 相比,在 VxWorks 映像中嵌入 WDB 调试代理具有如下优势:

- 仅仅只需编写一个初始化代码模块。至于传统的 ROM-monitor,必须编写两个初始化代码模块:一个是为固化在 ROM-monitor 里的监控编写,另一个是为操作系统。另外,传统的方法需要进行不规范的修改,以便解决监控与操作系统之间相互争抢 MMU、向量表、设备初始化的问题。
- 代码尺寸小。因为 VxWorks 和目标机调试代理可以共享通用库程序,例如 memcpy()函数。
- 传统的 ROM 监控器仅仅只支持“系统模式”调试。把整个操作系统当作一个进程而对它进

行调试。WDB 调试代理不仅支持系统模式调试，而且还提供了一个完整的 VxWorks-aware 任务模式调试。这种模式允许选择操作系统中的一部分（例如一个对立的任务）进行调试，而不影响系统的其他部分。

如何下载 WDB 调试代理和 VxWorks 内核取决于所处的开发阶段。当正处于编写和调试目标板初始化代码时，必须建立自己的下载方式。这一点上与传统的 ROM-monitor 方法相比既不占优势也没有劣势，但 ROM-monitor 的监控器需要为自己建立一个下载路径。一旦目标机初始化代码正常工作后，接下来如何做，取决于建立的下载路径的速度。

如果拥有一条高速的下载路径（例如网络接口），可以在以后的内核开发中继续使用它。这比传统的只能基于串口下载的 ROM 监控器更具有优势。如果只拥有一条低速的下载路径（例如串口），应当将内核、调试代理程序以及尽可能多的通用 VxWorks 代码集成在一起而“烧”进 ROM 里。

因为 Tornado 开发环境下的工具允许动态下载和执行代码，可以下载扩充程序（例如应用程序代码、新的设备驱动程序、附加的硬件初始化代码等）并使用 WDB 调试代理对这些扩充代码进行调试。这一点与传统的调试方法相比具有优势，因为传统的调试方法每次需要下载整个内核，而不考虑代码作了多大的修改。

16.2 建立开发环境

建立开发环境就意味着建立一个下载代码到目标机，然后检验下载代码的机制。通常，从一些简单测试代码映像开始入手，比从一个完整的、固化 VxWorks 映像入手要好得多。查阅参考 BSP 和构造 VxWorks（一个 RAM 型 VxWorks 映像）或 vxWorks_resrom_nosym（一个 ROM 型 VxWorks 映像）的构造文件，可以找出所需要的编译器和连接器的标志。

1. 下载方式的选择

下面是一些把代码下载到目标机上所采用的常用技术：

（1）目标板厂商的调试 ROM。有些目标板厂商在其目标板上嵌入了一个调试 ROM，可以使用调试 ROM 提供的下载协议（例如 FTP 协议）下载目标代码。这种下载方式的缺点是下载速度一般比较慢，但优点是容易建立下载环境。

（2）ROM 仿真器。使用 ROM 仿真器（例如 NetROM）建立下载路径。这种下载方式的缺点是需要花一些时间去学习如何使用它。至于它的优点包括下载速度快、与多数目标板兼容，通讯协议允许通过 ROM 插座把调试信息从目标机上传递给宿主机。

（3）在线仿真器 ICE。使用在线仿真器 ICE。这种下载方式的缺点是设备本身价格昂贵，且不同的体系结构需要不同的 ICE。但优点是允许单步执行初始化代码。

当下载代码到目标机上后，需要检查内存和 ROM，以确认代码被装载在正确的位置。Wind River 公司提供的 GNU 工具 nm（列出对象文件符号表）和 objdump（显示对象文件）可以用于编译好的映像文件，查看目标机内存里应有的内容。特别是代码段和数据段的开始地址是值得特别注意的地方。

2. 调试方法的选择

代码下载到目标机上后，下一步是建立一个调试机制，以便跟踪初始化顺序的进展情况。下面

是一些常用的方法:

- (1) 使用 ICE, 例如 visionICE 或 Trace32 ICE。
- (2) 使用目标板上本身的调试 ROM (如果它支持断点调试)。
- (3) 使用逻辑分析仪跟踪处理器的地址线。
- (4) 编写一个调试库做一些类似如下的处理:
 - 点亮 LED;
 - 永久性存储器 (本身的、目标板以外的或带电池的存储器) 的写入;
 - 在轮询模式下通过串口线发送一个字符;
 - 通过 NetROM 调试端口发送数据。

一旦选择了一个调试机制后, 检查可以跳转到代码的入口点并成功运行, 测试代码应当设置所需要的目标板内存, 然后通过对这片内存进行读写操作来证明它正在正常工作。测试程序将可能使用与 VxWorks 相同的初始化代码。有关详细的内容在 romInit.s 和 sysALib.s 文件里, 请参考 10.3.1 节。



注意

如果测试基于一个 ROM-based VxWorks 映像中的调试库, 要注意的是数据段和 bss 段还没有创建。特别是依赖于全局变量正确初始化的库仍不可以工作。

16.3 编写内核启动前 BSP 的初始化代码

这一部分描述了如何编写 BSP 文件, 构造和下载 VxWorks, 调试内核启动前的初始化代码以及在内核启动前何时、又如何启动 WDB 调试代理进行 BSP 的调试。

16.3.1 编写 BSP 文件

在 ../target/config 目录下新建一个 BSP 子目录, 并把 ../target/config/templateCPU 目录下的模板 BSP 文件拷贝到新建的 BSP 目录下。下面将逐个描述当修改这些文件时, 需要考虑和注意的事项。

1. Makefile 文件

Makefile 文件是一个构造 VxWorks 映像的批处理文件。在这个文件中, 至少必须定义如下一些宏:

- CPU: 目标机的 CPU (例如 PENTIUM), 必须与参考的 BSP 一致;
- TOOL: 宿主机的工具链 (例如 gnu), 也必须与参考的 BSP 一致;
- TGT_DIR: target 目录的路径。默认值为 \$(WIND_BASE)/target;
- TARGET_DIR: BSP 目录名, 例如 pcPentium;
- VENDOR: 目标板生产商的名字, 例如 intel;
- BOARD: 目标板的名字, 例如 PC Pentium;
- ROM_TEXT_ADRS: 十六进制表示的引导 ROM 入口地址。

对于多数目标板, 把它设置成 ROM 区的起始地址。然而, 可能有一些硬件结构使用 ROM 中的起始区域作为向量复位区; 这样, 它应当做一些偏移, 而偏移量典型依赖于体系结构。所以, 这

个宏的低字节可以从参考 BSP 中拷贝获得。

- **ROM_WARM_ADRS**: 十六进制表示的引导 ROM 热启动入口地址;
这个地址通常是冷启动入口点 **ROM_TEXT_ADRS** 偏移 8 个字节的一个组合地址, 例如 (**ROM_TEXT_ADRS**+8)。当硬件 ROM 程序需要一个开关量时, **sysToMonitor()** 中的代码应当直接跳转到 **ROM_WARM_ADRS**。
- **ROM_SIZE**: 十六进制表示的 ROM 区大小;
- **RAM_LOW_ADRS**: 装载 VxWorks 映像的地址;
- **RAM_HIGH_ADRS**: 将引导 ROM 映像拷贝到 RAM 中的地址;
- **HEX_FLAGS**: 体系结构特殊标志, 用于 Wind River 应用程序 (指 objcopy) 产生 S-record 文件, 必须与参考的 BSP 一致;
- **MACH_EXTRA**: 任何与目标机系统相关的扩展文件 (例如第三方的设备驱动程序库文件), 先把它作为一个空声明。



注意

上述的十六进制地址不应当以 0x 开头, 如下所示:

```
ROM_TEXT_ADRS      = fff20000
```

2. bspname.h 文件

通常在 **config.h** 所有选项选择后包含这个文件。允许在 **bspname.h** 文件里嵌入测试代码来检查正确的选择组合。例如在一块目标板上有两块不同的以太网卡, 在使用时仅仅只是可以选择它们中的一块网卡。假如这样的话, 则在 **bspname.h** 文件将检查网卡选择情况, 如果系统选择了两块网卡, 则产生错误信息。

为了进一步说明这个文件, 再附加一个例子, 例如用户可以选择 NFS 支持, 且在 **bspname.h** 文件里将增加 **RPC** 和网络支持作为支持 NFS 的必要条件。正好 **usrExtra.c** 所做的工作就是 **usrConfig.c** 所作的。当然, 这是一个假设的例子, 并不是一个实际例子。

3. config.h 文件

config.h 文件在 10.3.1 节里作了基本的描述, 但是作为新建的一个 BSP 需要作一些修改。

首先, 确定下面的代码段应该在 “**#include "configAll.h"**” 代码行之前。

```
#define BSP_VER_1_1      1          /* 版本 1.2 向后兼容版本 1.1 */
#define BSP_VER_1_2      1
#define BSP_VERSION      "1.2"
#define BSP_REV          "/0"      /* 0 是起始修订版本 */
```

在正式开始编写 **config.h** 文件前, 把下列几行代码增加到头文件 **config.h** 接近结尾的地方:

```
#undef INCLUDE_MMU_BASIC      /* 捆绑基本型 MMU 支持 */
#undef INCLUDE_CACHE_SUPPORT  /* 包括 cache 支持包 */
```

然后确认 **config.h** 文件中所定义的 **ROM_TEXT_ADRS**、**ROM_SIZE** 和 **RAM_LOW_ADRS** 的值与 **Makefile** 文件中定义的值一致。确认 **LOCAL_MEM_LOCAL_ADRS** 和 **LOCAL_MEM_SIZE** 是否正确定义。

4. sysLib.c 文件

到目前为止 sysLib.c 文件是最大的 BSP 文件，在这个阶段应当仅仅只是执行一些基本的函数：sysModel()、sysBspRev()、sysHwInit()、sysHwInit2()和 sysMemTop()函数。与 VxWorks 5.3 版本相比，sysBspRev()是一个新的函数，这个函数原形如下：

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

同时，sysLib.c 文件也应当包含下面的驱动程序段：

```
#include "mem/nullNvram.c"    /* 不可使用的 NVRAM 库 */
#include "vme/nullVme.c"      /* 不可使用的 VMEbus 库 */
```

文件中的 sysHwInit()函数是 sysLib.c 文件中的核心部分，大部分工作将放在这个函数上。这个函数复位所有的设备到一静止态，以便当中断打开后设备也不会产生中断。

5. romInit.s 文件

在目标机系统加电（冷启动）后，处理器首先执行 romInit()函数，这个函数必须是 romInit.s 文件中代码段的第一段程序。至于热启动，处理器在 romInit()地址加上一个小偏移量处开始执行（见 sysLib.c 中的 sysToMonitor()函数）。硬件和设备初始化操作将在后续的初始化序列中执行，具体由 sysLib.c 文件中的 sysHwInit()函数实现。romInit()函数的工作是将系统控制权转交给 romStart()函数（原型见 bootInit.c 文件）所需要执行的最小设置。具体职责如下所示：

（1）处理器的初始化（这些代码专门用于处理器而非目标板，且可以从参考 BSP 中拷贝获得）。其过程如下：

- 屏蔽处理器中断；
 - 设置堆栈指针初始化值为 STACK_ADRS（在 configAll.h 中定义）；
 - 关闭处理器高速缓存（cache）。
- （2）初始化目标机 DRAM（这些代码专门用于目标板）。为了下面的需要：
- 等待状态；
 - 刷新频率；
 - 芯片选择；
 - 关闭二级（L2）高速缓存（如果需要）。

最后，romInit()函数调用 bootInit.c 文件中的 romStart()函数，并把启动类型传递给它。当冷启动时，启动类型参数应当是 BOOT_COLD；当热启动时，启动类型参数将由 sysToMonitor()函数传递过来。

romInit.s 文件也必须包含一个数据变量“sdata”，用于标记数据段起始地址。这个变量典型存储一串字符，例如“start of data”。

有关详细信息，请参考产品随带的模板 BSP 中的 romInit.s 模板文件。

6. sysALib.s 文件

sysALib.s 文件包含了基于 RAM 型映像（例如 vxWorks）的入口点。其入口点为 sysInit()函数，它执行了将系统控制权转交给 usrInit()函数（原型见 usrConfig.c 文件）所需要的最小设置。这个函

数类似于 romInit()。sysInit()函数对于每一种体系结构的目标机在很大程度上都一样。同样，编写这段代码最好的开始办法就是从模板 BSP 中拷贝获得。sysInit()函数典型处理仅仅屏蔽处理器中断和设置堆栈指针。对于堆栈向下增长的处理器，堆栈指针设置成从 sysInit()开始；对于堆栈向上增长的处理器，在 sysInit()下面数百个字节处设置了堆栈指针，然后调用 usrInit()函数，并把 BOOT_WARM_AUTOBOOT 参数传递给它。硬件和设备初始化操作将在后续的初始化序列中执行，具体由 sysLib.c 文件中的 sysHwInit()函数实现。

如果可能，BSP 程序应该使用 C 语言编写，但是，如果由于编译技术原因而采取汇编语言编写 BSP 程序，然而这些程序应该放在 sysALib.s 文件中。

7. config/all/usrConfig.c 文件

在开发过程中，希望增加调试代码和其他操作，例如对通用（非 BSP）配置文件进行修改。这时，在 BSP 目录下创建一个本地备份优于直接修改通用代码，如下所示。

(1) 首先把../config/all 目录下的 usrConfig.c 和 bootInit.c 文件拷贝到 BSP 目录下。然后，在新建 BSP 目录下的 Makefile 文件中，并确定在 HEX_FLAGS 定义之后，增加如下代码行：

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

当构造 VxWorks 映像时，这将导致 Makefile 使用拷贝的通用配置文件。

(2) 接下来修改拷贝的 usrConfig.c 文件。通过使用“#if FALSE/#endif”语句删除所有其他除内核启动前的初始化代码之外的代码。这样，导致 usrExtra.c 文件也不包含在内。并且 usrRoot()函数内部为空。具体修改如下所示：

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

void usrRoot(char *    pMemPoolStart,unsigned memPoolSize )
{
    #if FALSE
    ...
    #endif
}
```

usrExtra.c 文件中的#include 连接了 src/config 目录下的其他配置文件。而现在并不需要这些文件，但是当将来需要增加一些特性时需要使用它们。例如当准备开始启动 VxWorks 内核和目标机代理时，将需要连接如下所示的配置代码：

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


16.3.2 构造和下载 VxWorks 映像

装载到目标机上的 VxWorks 映像取决于所使用的下载方式。主要的映像如下所示:

(1) vxWorks。基于 RAM 型映像, 需要通过一个引导程序 (例如, 目标板自带的调试 ROM) 把它下载到目标机内存中才可以执行。在开发环境中是一个默认选项, 主要在调试阶段使用。

(2) vxWorks_rom。基于 ROM 型映像, 在执行前把自己拷贝到目标机 RAM 中。这种类型的映像通常在启动阶段比较慢, 但在执行阶段比 vxWorks_romResident 要快。

(3) vxWorks_resrom_nosym。ROM 驻留型映像, 仅仅在启动时把数据段拷贝到目标机 RAM 中。这种类型的映像启动阶段比较快, 通常被用到 RAM 空间比较小的目标机上。在嵌入式应用中该类型的映像作为典型, 然而, 它在目标机上执行的速度比其他 ROM 类型的映像要慢, 原因是 CPU 访问 ROM 比访问 RAM 要慢。

如果下载路径把映像放在 RAM 中 (例如当使用厂家的调试 ROM 时), 使用 vxWorks。如果下载路径把映像放在 ROM 中 (例如当使用 NetROM 时), 使用 vxWorks_rom 或 vxWorks_resrom_nosym。vxWorks_rom 的优点是它更容易调试, 因为软件的断点只能设置在 RAM 的地址上。vxWorks_resrom_nosym 的优点是占用很少的目标机内存。这两个 ROM 映像的 Makefile 文件允许指定一个可选者的 .hex 文件扩展名 (例如 vxWorks_rom.hex), 产生一个 S-record 格式的文件, 而不是对象文件。

有一个被称为 depend.cputool 的文件, 它包含了 Makefile 使用的附属文件规则。如果这个文件不存在, Makefile 文件将自动产生这个附属文件。如果增加了新的文件到 BSP 中, 删除这个附属文件, 让 Makefile 文件重新产生这个附属文件。

当下载代码到目标机上后, 检查内存和 (或) ROM, 确认代码被装载到正确的位置。对于编译好的映像文件, 使用 nm 和 objdump 工具, 查看目标机内存里应有的内容, 尤其是代码段和数据段的开始地址值得特别注意。

16.3.3 调试初始化代码

ROM 和 RAM 的初始化序列在开始时是不同的, 但是在其他方面是一样的。每一个 BSP 需要处理的过程在前面章节中已经详细作了描述。这一节将重新回顾初始化序列的步骤, 并提供了当某一步发生错误时如何来检查的一些技巧。

1. ROM 初始化

这一部分描述了基于 ROM 型 VxWorks 映像: vxWorks_rom 和 vxWorks_resrom_nosym 的初始化序列。

(1) romInit.s 文件中的 romInit() 函数。在目标机系统加电 (冷启动) 后, 处理器首先执行 romInit() 函数, romInit() 函数的工作是将系统控制权转交给 romStart() (函数原型见 bootInit.c 文件) 所需要执行的最小设置。更多的硬件和设备初始化操作将在后续的初始化序列中执行, 具体由 sysLib.c 文件中的 sysHwInit() 函数实现。

(2) bootInit.c 文件中的 romStart() 函数。代码段和数据段采用下述的一种方法把它们从目标机 ROM 中拷贝到 RAM 中:

- 对于 vxWorks_rom, 把数据段和代码段都拷贝到 RAM 中;

- 对于 vxWorks_resrom_nosym, 仅仅只是把数据段拷贝到 RMA 中。拷贝后, 检查数据段已经完整地初始化。例如:

```
int taskval = 68;    /* 某个数据段的变量 */
...
if(taskval != 68)
ThereIsWrong();      /* 自定义的一个错误显示函数 */
```

如果出现了错误, 检查 RAM 是否可以正确地访问。例如:

```
int temp;
...
temp = 68;
if(temp != 68)
ThereIsWrong();      /* 自定义的一个错误显示函数 */
```

如果 RAM 能够正常工作, 检查是否在正确的偏移位置上拷贝数据段到内存中, 这种问题仅仅只是 vxWorks_resrom_nosym 可能发生。romStart() 函数假设数据放置在 ROM 代码段结束加上体系结构指定偏移量的这个位置处。下面是来自于 bootInit.c 文件中的一段代码:

```
#if (CPU_FAMILY == SPARC)
    copyLongs ((UINT *) (etext + 8), (UINT *) RESIDENT_DATA,
#elif ((CPU_FAMILY == MIPS) || (CPU_FAMILY == PPC))
    copyLongs ((UINT *) (etext + 0), (UINT *) RESIDENT_DATA,
#else
    copyLongs ((UINT *) (etext + 4), (UINT *) RESIDENT_DATA,
#endif
    ((UINT) edata - (UINT) RESIDENT_DATA) / sizeof (long));
```

然而, 如果使用可选的工具构造 ROM 映像, 那么这个偏移量会不一样。在这种情况下, 可能需要相应的调整偏移量。

romStart() 函数最后的事情是调用 usrConfig.c 文件中的通用初始化程序 usrInit()。i960 是一个例外, 它首先跳转到 sysInitAlt() 函数 (原型在 sysALib.s 文件中), 是为了在跳转到 usrInit() 函数之前重新装载处理器平台。剩下的初始化序列在后面的通用初始化部分描述。

2. RAM 初始化

这一部分描述了基于 RAM 型 VxWorks 映像的初始化程序。

sysALib.s 文件中的 sysInit() 函数。这个文件包含了基于 RAM 型映像 (例如 vxWorks) 的入口点 sysInit() 函数。它的工作是执行将系统控制权转交给 usrInit() (函数原型见 usrConfig.c 文件) 所需要的最小设置。这个函数类似于 romInit() 函数。更多的硬件和设备初始化操作将在后续的初始化序列中执行, 具体是由 sysLib.c 文件中的 sysHwInit() 函数实现。

3. 通用初始化

对于基于 ROM 型和基于 RAM 型映像, 其余的初始化代码是通用的。对这一部分初始化代码将在此处进行描述。

(1) usrConfig.c 文件中的 usrInit() 函数。从一个 BSP 开发者的角度来看, usrInit() 函数主要的工作是清除 bss 段 (未初始化的 C 语言全局变量现在都为 0), 然后调用 sysHwInit() 函数 (在 sysLib.c 文件中) 初始化硬件。如果已经正确设置了内存, 在这个程序里很少会出现错误。

(2) sysLib.c 文件中的 sysHwInit() 函数。这是 BSP 初始化代码的核心程序。它必须重新设置硬

件环境到一静止态，这样以后打开中断的时候，也不会产生未初始化的中断。

(3) `usrConfig.c` 文件中的 `usrInit()` 函数。`sysHwInit()` 函数执行完后，控制权转交给 `usrInit()` 函数。`usrInit()` 函数最后的工作是调用 `kernelInit()` 函数启动 VxWorks 内核。此时内核启动前的初始化代码到此结束，而 `kernelInit()` 函数不会返回。然而，它启动内核并把 `usrRoot()` 作为第一个创建的任务。在 `kernelInit()` 里中断被打开，如果在 `sysHwInit()` 函数里没有关闭和清除中断源，将会产生严重的混乱现象。

如果希望在目标机代理控制下启动 VxWorks 内核，必须在 `kernelInit()` 前启动目标机代理。这是一个可选的步骤，并不是一个典型的做法。有关详细信息，见 16.3.4 节，在内核启动前启动 WDB 调试代理。

(4) `usrConfig.c` 文件中的 `usrRoot()` 函数。内核已经启动后，其余的 VxWorks 初始化在 `usrRoot()` 函数里进行，详细的初始化步骤见前面的章节。在这个阶段，如果 `usrRoot()` 函数能够证实用户正确地编写了 `sysHwInit()` 函数，这就已经足够了。

如果调用了 `kernelInit()` 函数，但是还没有开始执行 `usrRoot()` 就出现了错误导致执行失败。错误可能是下面两个错误中的一个，即 `sysMemTop()` 函数返回一个错误的地址或更多类似的设备没有被复位而产生了中断。对于第二种情况，必须修改 `sysHwInit()` 函数复位产生中断的设备。为了找到中断源，首先确定产生中断的中断向量，可以应用下面的一些技术来实现：

- 使用逻辑分析仪，查找访问中断向量表的指令；
- 使用 ICE，在中断向量表中设置断点；
- 修改 `sysHwInit()` 函数，通过中断控制器屏蔽可疑的中断向量；
- 修改 `sysHwInit()` 函数，使用 `intVecSet()` 函数把调试程序连接到可疑的中断向量上（不可以使用 `intConnect()` 函数，因为它调用了 `malloc()` 函数，而此时 VxWorks 存储分配器并没有初始化）。

在这个时候，拥有一个可工作的 VxWorks 内核，但是并没有设备驱动程序。VxWorks 仅仅需要一个定时器和一个中断控制器（如果可能的话）驱动程序。大多数 BSP 也拥有串口设备驱动程序。有关添加基本的驱动程序来支持 VxWorks 的工作，将在后面使用最小内核部分进行描述。

16.3.4 在内核启动前启动 WDB 调试代理

注意

这一部分仅仅应用于通过命令行方式构造 VxWorks 映像。

这是一个可选步骤，当新建一个 BSP 时几乎不进行该步操作。在内核启动前启动 WDB 调试代理的缺点是：

- 一旦编写完硬件初始化代码，启动内核比启动 WDB 调试代理所花的时间要少。因为大多数开发人员认为一个正常工作的内核是一个更重要的里程碑，因此从内核开始。
- 在内核启动前启动 WDB 调试代理并不会给基本内核的正常工作带来帮助。这是因为基本内核添加很少的内容到已经编写好的代码中；只是一个定时器和一个中断控制器（如果可

能的话) 驱动程序, 它们都是一些简单的设备。

- 在内核启动前启动 WDB 调试代理限制只能使用系统模式调试; 早期启动 WDB 调试代理惟一的原因是下载的环境速度太慢。在这种情况下, 可能希望尽早把所有的内容放置到 ROM 中, 以节省下载周期。

在内核启动前启动 WDB 调试代理详细描述在 12.4.2 节, 使用 WDB 调试代理。下面的两个部分重要描述了其他信息。

1. 警告

因为虚拟 I/O 驱动程序需要使用 VxWorks 内核, 所以把下面代码行添加到 config.h 文件中:

```
#undef INCLUDE_WDB_VIO
```

如果计划使用目标机代理的串口线与宿主机通讯, 这是一个非常重要的警告。当内核一旦启动, 处理器中的中断被打开, 但是设备驱动程序中断处理程序依然没有连接。必须小心确认用于 WDB 调试代理通讯所使用的串口设备并没有产生中断。如果目标板有一个中断控制器, 在 sysHwInit() 中通过中断控制器屏蔽串口中断。谨防目标机代理企图在 “interrupt on first packet” 模式里使用所有的驱动程序。所以, 需要修改串口驱动程序非中断模式, 即使目标机代理需要使用也不例外。

2. 系统模式调试技术

WDB 调试代理正常工作后, 将希望使用它调试与它连接在一起的 VxWorks 映像。为了节省下载时间, 应当把希望测试的代码与 VxWorks 连接在一起, 构造成一个 VxWorks 映像, 并固化到目标板的 ROM 中。特别是, 使用最小的内核, 请阅读 16.4 节。在这部分内容里, 如何把驱动程序代码与 VxWorks 连接在一起是更值得学习的内容。为了避免重新擦写 ROM, 考虑在 BSP 和驱动程序中添加钩子 (hook) 程序。如下所示:

```
void (*TestsysHwInitHook)(void);          /* 定义一个钩子函数 */
...
void sysHwInit2(void)
{
    if (TestsysHwInitHook != NULL)        /* 条件调用 */
    {
        TestsysHwInitHook();
        return;
    }
    ... /* 默认代码 */
}
```

这种设计允许通过调试器动态的替换程序。例如, 为了覆盖上述的 sysHwInit2() 函数, 仅仅只是创建一个新的版本, 即 sysHwInit2() 函数调用 myLib.o 模块中的 myHwInitHook() 函数。操作如下:

```
(gdb) load myLib.o
(gdb) set TestsysHwInitHook = myHwInitHook
(gdb) break myHwInitHook
(gdb) continue
```

然而, 如果在内核启动前启动目标机代理, 必须在 sysHwInit() 函数调用之后启动它。这样, 不可以覆盖 sysHwInit() 函数。另一方面, 可能希望在内核启动前额外增加一些初始化硬件的代码。在这种情况下, 可以在调用 kernellnit() 函数前增加一个钩子函数。作为一个可选的钩子, 可以通过使

用 GDB 的 call 程序从调试器中调用程序。例如：

```
(gdb) call myHwInitHook
```

使用钩子函数替代 call 机制的优势是：

- 钩子可以避免执行原始的代码；
- 钩子执行速度更快。

如果目标板有一个“异常结束”按钮，考虑把一个调试程序连接到这个异常结束中断上。这样，可以通过调试器在中断处理上设置一个断点。如果目标系统出现死机情况，使用这种方法可以为获得系统的控制权。在这种情况下，最好是将异常结束开关连接到一个非屏蔽中断上（NMI，Non-Maskable Interrupt）。



注意

只是致命的中断（例如异常结束）可以连接到 NMI，如果设备中断连接到 NMI 上，则内核将不能正常工作。

16.4 使用最小的内核

一个最小的内核包括只是添加一些简单的设备驱动程序。因为这些驱动程序包括中断服务程序的连接（ISR），所以这一部分内容首先就是从 ISR 指南开始描述。

1. 中断服务程序指南

内核启动后，可以使用 intConnect() 函数和 INUM_TO_IVEC 宏，把中断服务程序连接到中断向量表中。例如：

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

其中，intVec 参数是中断向量（通常称中断号），proc 是中断产生时调用的 C 程序，param 是传递给 proc 程序的一个参数。有关更多的信息，参考 intConnect() 函数。

在编写 ISR 时，通常的错误就是疏忽清除中断源的中断应答。不幸的是这种疏忽的结果将导致当 ISR 完成后，立刻产生另一个中断。

非屏蔽中断不可以与多数 VxWorks 库函数关联使用。必须非常小心的使用和执行它们。有关 ISR 详细信息，参考 5.3 节。

2. 必要的驱动程序

VxWorks 必要的驱动程序仅仅只是系统时钟，其他某些体系结构，例如 80x86 和 Power pc 同时还需要中断控制器驱动程序。执行系统时钟驱动程序具有代表性地就是使用 ../target/src/drv/timer 目录下已经存在的驱动程序（中断控制器驱动程序在 src/drv/intrCtl 目录下）。如果重新使用一个已经存在的驱动程序，需要在 sysHwInit() 函数中执行目标板特定的硬件初始化，并且在 sysHwInit2() 函数中调用 intConnect() 函数连接中断。

定时器驱动程序是一个简单的设备，可以通过修改 usrRoot() 函数执行一些周期性地动作来测试它，例如点亮一个 LED。测试代码如下：

```
void Testtimer(void)
```



```

{
while(1)
{
    taskDelay(5*sysClockRateGet());    /* 延迟 5 秒钟 */
    sysFlashled();                      /* 点亮 LED */
}
}

```

3. 串口驱动程序

尽管串口驱动程序不是必要的，但是大多数 BSP 包括串口驱动程序。随着 Tornado 的引进，串口驱动程序已经发生了重大的改变（实际上，它是 BSP 的主要改变）。新的串口驱动程序放置在 `../target/src/drv/sio` 目录下。在这些驱动程序中总会有一个驱动程序满足需求。如果没有，使用 `../target/src/drv/sio/templateSio.c` 模块驱动程序作为编写自己串口驱动程序的基础。

早期，通过使用 `#if FALSE/#endif` 删除 `usrRoot()` 函数体的大部分程序。而现在将需要把 `#if FALSE` 语句往下移，放到串口设备初始化之后。

应该修改 `usrRoot()` 函数产生一些应用测试代码对驱动程序进行测试。例如，可以周期性的往控制台打印一条信息：

```

void Testderial (void)
{
extern int vxTicks;
char * message = "Serial is working....";
while(1)
{
    taskDelay(5*sysClockRateGet());    /* 延迟 5 秒钟 */
    write(1, message, strlen(message)); /* 打印信息 */
}
}

```

16.5 目标机代理和 Tornado

借助目标机调试代理 —— WDB，可以调试内核启动前的初始化代码。这需要通过调用 `wdbConfig()` 函数初始化目标机代理。默认情况下，调用发生在 `usrRoot()` 函数的结尾处。然而，在前面修改 `usrRoot()` 函数使它函数体为空（见 16.3.1 节）。现在重新恢复目标机代理，在串口设备初始化代码之后调用 `wdbConfig()` 函数。目标机代理默认配置是使用网络与宿主机进行通讯。不幸地是，网络现在依然不可用。这样，必须使用一个可供选择的连接方式，建立目标机与宿主机之间的通讯。

下面是一个目标机与宿主机通过串口建立连接方式的修改示例：

在 `config.h` 文件中加入以下宏定义：

```

...
#undef WDB_COMM_TYPE
#define WDB_COMM_TYPE      WDB_COMM_SERIAL    /* 定义通信方式为串口连接 */
#undef CONSOLE_TTY
#define CONSOLE_TTY        0                  /* 目标机控制台通讯通道号为 COM1 */

```



```
#undef WDB_TTY_CHANNEL
#define WDB_TTY_CHANNEL 1          /* WDB 调试代理通讯通道号为 COM2 */
#define WDB_TTY_BAUD 38400        /* 串口通讯速率, 可设置至 38400 */
```

16.6 最后的修整工作

这一部分对各种各样的工作所需要最后的修整进行了概括。包括注释清除、添加定时器、网络等其他问题的讨论。

1. 注释清除

到目前为止, 已经使用了自己私有的 `usrConfig.c` 和 `bootInit.c` 拷贝文件。BSP 应该与通用版本一起工作。为了重新安装通用版本, 从 BSP 的 Makefile 文件中删除下面代码行:

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

然后执行 `make clean` 操作。

在这之前, 使用 `#if FALSE/#endif` 语句屏蔽了不需要的配置代码。现在, 需要一个更标准的方法来删除不需要的代码。也就是说, 必须在 `config.h` 文件中通过使用 `#undef` 宏来进行。必须记住 `../target/src/config/usrDepend.c` 文件中所列出的宏的从属性。

2. NVRAM

VxWorks 为读写持久稳固的存储区定义了一个接口。称这个接口为非易失性内存库。一些通用的驱动程序包含在 `../target/src/drv/mem` 目录下。可以根据自己的需要把该模块配置到系统中。

3. 添加其他定时器

驱动程序可能包括一个辅助时钟驱动程序, 也同时可能包括一个高性能的时间戳驱动程序。

辅助时钟通常被 VxWorks 的 `spy` 工具所使用, 同时也被宿主机上 Tornado 下的 `browser` 工具所使用。有关辅助时钟接口的更多信息, 参考 `sysAuxClk()` 函数。

高性能时间戳驱动程序当前仅仅只是被 `WindView` 工具所使用, 但是编写这个驱动程序有益于将来的调试工作。可以在 `../target/h/drv/timer/timestampDev.h` 头文件中找到这些接口。

4. 网络

这个指南描述了如何使用一个“独立的” VxWorks 映像——`vxWorks.st`。这个映像包含两个特性:

- 它内部连接了一个 VxWorks 符号表。这个特性不再需要, 因为 Tornado 使用基于主机的符号表作为调试信息;
- 它连接了网络系统, 但是并没有初始化。

继续使用 `vxWorks` 或 `vxWorks_rom`, 替代 `vxWorks.st` 映像的使用。在其内部连接未初始化的网络系统, 需要在 `usrConfig.c` 文件中的 `usrNetInit()` 函数调用之前增加一个虚拟的条件。例如:

```
if(FALSE)          /* 增加虚拟条件 */
    usrNetInit (BOOT_LINE_ADRS); /* 为了不调用该函数 */
```

然而, 可以在后面调用 `usrNetInit()` 函数, 并使用交叉调试器 `CrossWind` 调试网络驱动程序。在关闭高速缓冲器和 MMU 的方式下, 这是使得网络驱动程序首先工作的最好办法。

下面的表在../target/src/config/usrNetwork.c 文件中，通过它来配置 BSD 4.4 版本的网络设备：

```

#ifdef NETIF_USR_DECL /* Additional declarations, from BSP */
    NETIF_USR_DECL
#endif
...
NETIF usrNetIfTbl [] = /* network interfaces */
{
#ifdef NETIF_USR_ENTRIES /* Additional entries, from BSP */
    NETIF_USR_ENTRIES
#endif

#ifdef INCLUDE_LNEBSA
    { "lnebsa", lnebsaattach, (char*)IO_ADRS_LNEBSA, INT_VEC_LNEBSA,
      INT_LVL_LNEBSA, LNEBSA_POOL_ADRS, LNEBSA_POOL_SIZE, LNEBSA_DATA_WIDTH,
      LNEBSA_MODE, LNEBSA_DMA_CHAN },
#endif /* INCLUDE_LNEBSA */
#ifdef INCLUDE_OLI
    { "oli", oliattach, (char*)IO_ADRS_NISA_BASE,
      IO_ADRS_NISA_PCMCIA, IO_ADRS_NISA_PCMEM,
      INT_VEC_PCMCIA_A, INT_LVL_PCMCIA_A,
      INT_VEC_PCMCIA_B, INT_LVL_PCMCIA_B },
#endif /* INCLUDE_OLI */
#ifdef INCLUDE_DC
    { "dc", dcattach, (char*)IO_ADRS_DC, INT_VEC_DC, INT_LVL_DC,
      DC_POOL_ADRS, DC_POOL_SIZE, DC_DATA_WIDTH, DC_RAM_PCI_ADRS,
      DC_MODE },
#endif /* INCLUDE_DC */
#ifdef INCLUDE_EGL
    { "egl", eglattach, (char*)IO_ADRS_EGL, INT_VEC_EGL, INT_LVL_EGL },
#endif /* INCLUDE_EGL */
    ...
}

```

对于 END 驱动程序，网络设备初始化是通过 configNet.h 文件中的一张表来进行，这张表的名字是 endDevTbl。如下所示：

```

#define FEI82557_LOAD_FUNC    fei82557EndLoad    /* 驱动程序外部接口 */
#define FEI82557_BUFF_LOAN    1                  /* 可借出的缓冲区 */

/*
 * The fei82557End initialization string format is:
 *
 * <memBase>:<memSize>:<nCFDs>:<nRFDs>:<userFlags>
 */

#define FEI82557_LOAD_STRING "-1:0x00:0x20:0x20:0x00"

IMPORT END_OBJ* FEI82557_LOAD_FUNC (char*, void*);
...
END_TBL_ENTRY endDevTbl [] =
{
    ...
}

```



```
#ifdef INCLUDE_FEI_END
    { 0, FEI82557_LOAD_FUNC, FEI82557_LOAD_STRING, FEI82557_BUFF_LOAN,
      NULL, FALSE},
    ...
    { 0, END_TBL_END, NULL, 0, NULL, FALSE},
    };
-----
```

就像看到的那样，驱动程序的初始化是通过一系列宏来控制的。在 BSP 中添加一个已经存在的 VxWorks 网络驱动程序，需要修改 config.h 文件，定义相应的宏和 I/O 参数。

在 BSP 中添加自定义的 BSD 驱动程序，需要定义 NETIF_USR_ENTRIES 和 NETIF_USR_DECL 宏。至于 END 驱动程序，用户仅仅只需要在 endDevTbl 表中增加一个适当的入口，包含这个设备驱动程序。有关驱动程序初始化字符串，参考驱动程序具体的格式和语法文档。

在 Tornado 2.0 标准的产品中，基于 VME 总线的独立网卡，EXOS-202 和 CMC ENP-10 已经取消支持。

5. cache、MMU、DMA 设备

接下来的工作是解决 BSP 在打开 cache 和 MMU 的情况下能够正常工作。这一部分内容由于本身的特殊性，本书不详细介绍，请读者参考其他书籍。



注意

cache 和 MMU 的配置会严重影响 DMA 设备的运行，例如网络驱动程序。

6. 引导 ROM

引导 ROM 也使用 VxWorks 内核。在 bootrom 映像和 vxWorks 映像之间，两个主要的不同点是：

- bootrom 映像使用 ../target/config/all/bootConfig.c 文件替代 ../target/config/all/usrConfig.c 文件。这两个文件比较类似，所以引导 ROM 的工作流程与前面所描述的在本质上是一样的。
- 默认 bootrom 映像是压缩型的。在 bootInit.c 文件中把它解压并拷贝到 RAM 中。

7. SCSI

SCSI 是一个标准外设接口，它允许连接各种硬盘、光驱、软驱、磁带设备和 CD-ROM 设备。SCSI 块设备驱动程序与 dosFs 和 rt11Fs 库兼容。VxWorks 的 SCSI-2 取代了原有的 SCSI 的支持，尽管原有的 SCSI 提供可选配置功能，则现在称它为 SCSI-1。通过使用 SCSI-2，VxWorks 环境依然可以执行 SCSI-1 应用，例如在 SCSI-1 下创建的文件系统。但是，应用程序直接使用定义在 scsiLib.h 文件中的 SCSI-1 数据结构，可能需要修改和重新编辑以便兼容 SCSI-2。

由于 SCSI 技术本身比其他设备要复杂，而且在项目中基本上用不到它，所以在本书中对它不做详细的分析，请读者参考其他资料。

8. 工程

BSP 最终是为用户的 VxWorks 应用工程服务。BSP 开发结束后，用户应该基于新的 BSP 新建一个默认工程。而对于 VxWorks 应用工程方面的问题，Tornado 文档包含了必要的信息。例如有关创建、配置和构造工程等。这些操作都可以通过开发环境 Tornado 的图形用户界面（GUI）来完成。有关详细信息请读者参考有关 Tornado 方面的书籍。

小 结

前面几章的内容对 BSP 开发者来说可以算作“养精蓄锐”，因为只是为 BSP 的开发作铺垫，没有联系到具体的开发步骤。而这一章内容主要指导用户根据自己具体的目标机硬件环境如何编写一个新的 BSP。通常新建 BSP 的过程可以借助开发环境 Tornado 下的一些辅助开发工具，按照一定的步骤分阶段的一步一步地进行开发，不过每一步的开发都是建立在前一步的开发基础上。这些步骤包括建立开发环境、编写内核启动前 BSP 的初始化代码、构造 VxWorks 映像、调试 BSP、使用最小内核、配置 WDB 通过串口使目标机与主机通讯等。

第 17 章 体 系 结 构

在开发过程中，BSP 开发人员应当参考一些有关体系结构的特殊书籍。这一章的焦点集中在系统的体系结构和在 VxWorks 系统中如何访问它们的一些相关问题。因此，在这一章中并不讨论编译选项和其他有关工具影响产生代码的问题。

17.1 Power PC

VxWorks 支持的 PowerPC 微处理器种类包括：

- PowerPC 603/604;
- Power PC 821/860;
- Power PC 403 (GA、GC 和 GCX 版本)。

尽管这些微处理器中的一些是 64 位体系结构，但 VxWorks 仅仅只是在 32 位模式下使用它们。PowerPC 存储顺序可以是**大端模式或小端模式**，然而 VxWorks 和 Tornado 工具仅仅支持大端模式。

17.1.1 26 位寻址

为了遵照 PowerPC EABI (Embedded Application Binary Interface) 标准，PowerPC 体系结构限制了它为带符号的 26 位寻址偏移。通过使用 PowerPC 分支指令 (例如 bl)，GNU 编译器编码直接进行 C 函数调用。系统接受子程序的调用相对于当前地址最大偏移量是 $\pm 32\text{MB}$ 。带符号的偏移是 24 位内部分支指令编码。除地址偏移位总是假定为零外，这两种很少具有特殊意义的地址偏移位并没有被编码。

这意味着 RAM/ROM 的映射和 RAM 大小受到很少限制。使用一个函数指针调用 C 程序并不会有地址限制，因为调用的函数地址是通过一个 32 位寄存器访问。也就是说可以访问 4GB 的存储空间。

如果下载一个 VxWorks 映像到目标机 RAM 中，然后下载一个对象模块到目标机系统内存中，要做到 VxWorks 映像和下载的对象模块不会相隔很远。另外，会存在交叉引用问题。如果使用驻留在宿主机上的装载器，应该小心自动处理，因为所有的对象模块是被装载到 WDB 内存池中，在默认情况下，WDB 内存池链接在 VxWorks 映像结尾处。

然而，如果 WDB 内存池在存储空间外运行，内存将从系统内存顶端分配。如果系统内存池大于 32MB 这将是一个问题。当使用 VxWorks 映像自带的驻留在目标机上的装载器，且 VxWorks 映像的系统内存池已经被初始化并大于 32MB 时，一个类似的问题也将出现。问题的起因是因为驻留在目标机上的装载器把下载的模块放在系统内存块结尾处。

经过对上述总结，所有 VxWorks 映像 (包括任何其他模块) 应该包含在同一个 32MB RAM 块中，而 RAM 块的起始为 0x0 (异常和中断部分位于 RAM 第一页)。

当使用 ROM 驻留型 VxWorks 映像时，出现了其他问题。通过使用 PowerPC 有关的分支指令确定可能存在的回跳，和自从 RAM 到 ROM 以后没有 26 位带符号的偏移相冲突问题。因为 RAM 不

得不被映射在物理地址 0x0 开头处，ROM 起始地址不应该少于 0xFC000000。使用 VxWorks 动态装载器和一个 ROM 驻留型映像需要预防任何可能出现的问题，也包括中断和异常管理问题。

17.1.2 MMU 的支持

VxWorks 支持 PPC603、PPC604、PPC821 和 PPC860 上的 MMU 芯片，有关详细信息可以参考 VxWorks 的虚拟内存接口。但对于这些 PPC 微处理器只是提供基本虚拟内存库的支持，并不支持可选产品 VxVMI。

1. 指令和数据 MMU

对于 PowerPC 指令 MMU 和数据 MMU，允许各自独立被打开和关闭。在 VxWorks 系统默认配置中，定义了参数 USER_I_MMU_ENABLE 和 USER_D_MMU_ENABLE。为了打开/关闭一个或两个 MMU，选择相应的参数并撤消定义，例如：

```
#define USER_I_MMU_ENABLE      /* 打开指令 MMU */
#undef  USER_D_MMU_ENABLE     /* 关闭数据 MMU */
```

2. 60x 存储映射

PowerPC 603 和 604 MMU 支持两种存储映射模式。第一种是块地址转换（BAT，Block Address Translation）模式，允许存储块（块大小为 128~256MB）映射到一个 BAT 寄存器里。第二种是段模式，把内存映射成 4KB 大小的一个个页面。针对 PowerPC 的 VxWorks 支持两种内存模式。

（1）603/604 块地址转换模式。BAT 寄存器的尺寸是两个 32 位的字。对于 PPC603 和 PPC604，包含 8 个 BAT 寄存器：给指令 MMU 和给数据 MMU 各分配 4 个。

定义在 sysLib.c 文件中的数据结构 sysBatDesc[]，控制 BAT 寄存器的配置。这些寄存器将被 MMU 库中的初始化软件进行设置。默认情况下，这些寄存器被清除并设置为 0。

（2）603/604 段模式。这种模式对每个内存页指定配置。在数据结构 sysPhysMemDesc[]中，描述了整个物理内存，且在 sysLib.c 文件中定义了该数据结构。数据结构由页或页组配置常量组成。

页表大小取决于总的内存映射。映射的内存加大，页表大小也将加大。VxWorks 段模式的实现推荐根据目标板编程环境而定。在 MMU 库初始化期间，计算总的映射内存大小，同时允许动态确定页表大小。在表 17-1 中显示了总映射内存和页表尺寸。

表 17-1 页表尺寸

总映射内存	页表尺寸
8MB 或更少	64KB
16MB	128KB
32MB	256KB
64MB	512KB
128MB	1MB
256MB	2MB
512MB	4MB

续表

总映射内存	页表尺寸
1GB	8MB
2GB	16MB
4GB	32MB

17.1.3 高速缓存问题

PowerPC 处理器包括一个指令高速缓存和一个数据高速缓存。在默认配置情况下, VxWorks 打开了这两个高速缓存。在 config.h 文件中, 通过取消 USER_I_CACHE_ENABLE 和 USER_D_CACHE_ENABLE 宏的定义, 分别屏蔽指令高速缓存和数据高速缓存。

1. PowerPC 60x 和 860

对于大多数 PowerPC 处理器 (PPC603、PPC604、PPC821 和 PPC860), 使用高速缓存和 MMU 功能解决高速缓存一致性问题。每页的页描述符选择了高速缓存模式。通过填充 sysPhysMemDesc[] 数据结构配置页描述符, 而这个数据结构在 sysLib.c 文件中被定义。有关高速缓存一致性的更多信息, 请参考 cacheLib。有关 MMU 和 VxWorks 虚拟内存的更多信息参考 VxWorks 的虚拟内存接口。有关 PowerPC 处理器的 MMU 信息见 17.1.2 小节。

数据高速缓存和指令高速缓存的状态是通过保留在 BAT 或段描述符里的 WIMG 信息来进行控制。因为不支持默认高速缓存状态, 相应的 MMU 被打开后, 每一个高速缓存可以被独立地打开, 有关这方面的信息参考 Motorola 和 IBM 出版的有关 PowerPC 微处理器的开发环境。



注意

在 Tornado 2.0 里提供了 PowerPC 860 的高速缓存库。高速缓存可以被 CPU 的修订版本为 A.1 或更高的任何一款 CPU 所使用。由于芯片硬件问题, VxWorks 运行在比 CPU 的修订版本 A.1 还早的任何一款处理器上时, 必须关闭处理器上的高速缓存。

2. PowerPC 403

在 Tornado 1.0.1 版本里, 依然没有为 PPC403GC 和 403GCX 处理器 (403GA 没有任何 MMU) 提供 MMU 的支持。这样, VxWorks 运行在所有 403 系列的实模式下。然而, 使用寄存器 DCCR (Data Cache Cacheability Register) 和 ICCR (Instruction Cache Cacheability Register), 定义在 128MB 区域的数据和指令可以被访问。这两个寄存器分别通过 ppc403DccrVal 和 ppc403IccrVal 变量进行设置, 默认情况下, 这两个变量的初始化值为:

```
PPC403_ICCR_DEFAULT_VAL (0x80000001)
PPC403_DCCR_DEFAULT_VAL (0x80000001)
```

换一句话说, 默认情况下, 地址范围 0x0~0x07ffffff (通常是 RAM) 和 0xf8000000~0xffffffff (通常是 ROM) 是作为数据和指令访问的高速缓存而被设置的。为了符合目标板内存/设备的布置, 需要更改这个默认设置。具体的操作是编辑 sysLib.c 文件中的 sysHwInit() 函数, 更改这两个变量值。

对于 PPC403GC 和 GCX 的目标板，必须初始化 DCWR（Data Cache Write-through Register）为 0x0，因为当在实模式下使用这个数据高速缓存时，它不支持 write-through 模式。

另一方面，在目标板初始化期间，指令和数据高速缓存必须重新复位。

17.1.4 中断管理

PowerPC 核心只是接受一个外部中断异常（中断向量为 0x500），异步中断输入的控制必须通过一个外部中断控制芯片（PPC603 和 PPC604）或一个片上（on-chip）中断控制器（PPC403 或 PPC860）。这样，中断驱动程序必须提供通常包含在 intArchLib 库里的大部分功能。而与 PowerPC 体系结构有关的代码，如表 17-2 中所列举的函数。

表 17-2 中断函数表

函 数	描 述
intLock()	关中断
intUnlock()	开中断

表 17-3 中的函数必须作为中断驱动程序的一部分。

表 17-3 中断驱动程序常用中断函数表

函 数	描 述
intConnect()	将中断服务程序和中断向量连接
intVecSet()	设置 CPU 中断向量
IntVecGet()	获得中断向量
IntVecBaseSet()	设置中断向量基地址
IntVecBaseGet()	获得中断向量基地址
IntlevelSet()	设置中断级
IntEnable()	打开中断
IntDisable()	关闭中断

为了缚上这些函数到 intArchLib 库中，需要使用下面的函数指针，而这些指针定义在../target/h/arch/ppc/ivPpc.h 文件中。

- (1) _func_intConnectRtn
- (2) _func_intVecSetRtn
- (3) _func_intVecGetRtn
- (4) _func_intVecBaseSet
- (5) _func_intVecBaseGet
- (6) _func_intLevelSet
- (7) _func_intEnable

(8) _func_intDisable

中断驱动初始化程序应该设置上述的函数指针与它们的驱动程序相对应, 例如:

```
func_intConnectRtn = sysXXXIntConnect;
```

这样, 当 intConnect() 函数被执行时, 保证了调用指定的驱动程序, 且其他的驱动程序也类似。

另外, 中断驱动程序必须提供一个中断多路 (demultiplex) 程序 (例如 sysXXXIntHandler() 函数), 它可以根据系统产生的中断源, 调用正确的 ISR (中断服务程序)。这个多路程序应该被缚到 PowerPC 外部中断向量 (0x500) 上, 类似代码如下:

```
excIntConnect ((VOIDFUNCPTR *) _EXE_OFF_INTR, sysXXXIntHandler);
```

1. PowerPC 60x

PPC603 和 PPC604 没有片上中断控制器。因此, 如果需要前面描述的函数性, 驱动程序必须自己履行这些函数性。

2. PowerPC 403

PPC403 系列处理器包含了一个片上中断控制器。这个控制器结合了异步中断输入和把它们作为单个中断信号提交给处理器核心部分。异步中断源是外部信号、DMA 通道、串行端口和 JTAG/调试设备。

中断驱动程序 ppc403Intr.c 文件提供了管理外部中断所需要的程序。这些现有的设备驱动程序应该是通用的程序, 可以应用于所有以 PPC403 为基础的 BSP。

3. PowerPC 821 和 860

PowerPC 800 系列的处理器由三个单元组成: PowerPC 核心单元、系统接口单元、通讯处理机单元。PowerPC 核心单元只是接受一个外部中断异常 (向量为 0x500)。系统接口单元提供一个中断控制器, 它提供 32 级中断, 但只是 16 级中断被使用。这个中断控制器连接 PowerPC 核心单元的外部中断。通讯处理机单元也提供了一个中断控制器, 它可以连接到系统接口单元中断控制器的一个入口上。现有的中断驱动程序 ppc860Intr.c 应该是通用的程序, 可以应用于所有以 PPC800 为基础的目标板上, 提供管理外部中断所需要的程序。

17.1.5 浮点支持

这一部分描述了 PPC403、PPC860、PPC60X 的浮点支持。

1. PowerPC 403 和 860

PowerPC 403 和 860 并不支持硬件浮点指令。然而, VxWorks 提供了一个浮点库, 它模拟了这些数学上的函数。使用来自于美国的浮点库, 所有 ANSI 浮点函数已经被优化。

2. PowerPC 60x

使用来自于 Motorola 的浮点库, 一部分 ANSI 函数被优化, 如下所示:

- acos()
- asin()
- atan()
- atan2()
- cos()
- exp()
- log()
- log10()
- pow()
- sin()
- sqrt()

其余的 ANSI 浮点函数由 VxWorks 的 ansiMath 库提供。

对于 PowerPC 60x 处理器, VxWorks 系统支持浮点异常管理。通过选择 VX_FP_TASK 选项,

系统创建一个浮点异常管理任务。下面是默认浮点异常环境：

- 用零除、溢出和下溢异常被打开；
- 舍入模式是“就近舍入”；
- 选择浮点异常模式是“不精确的和不可恢复的”。

如果一个指定的任务需要更改默认环境，在这个任务代码开始处修改机器状态寄存器（MSR，Machine State Register）、浮点状态和控制寄存器（FPSCR）的值。

- MSR 的 FE0 和 FE1 位选择浮点异常模式；
- FPSCR 的 VE、OE、UE、ZE、XE、NI 和 RN 位打开或关闭相应的浮点异常和舍入模式（相关宏定义参考 archPpc.h 文件）。

通过 vxMsrGet()、vxMsrSet()、vxFpscrGet()和 vxFpscrSet()函数，可以访问寄存器的变量值。这些函数的原型在../target/h/arch/ppc/vxPpcLib.h 文件中被定义。

17.1.6 存储配置

对于所有的 PowerPC 处理器，VxWorks 存储配置是一样的，如图 17.1 所示。

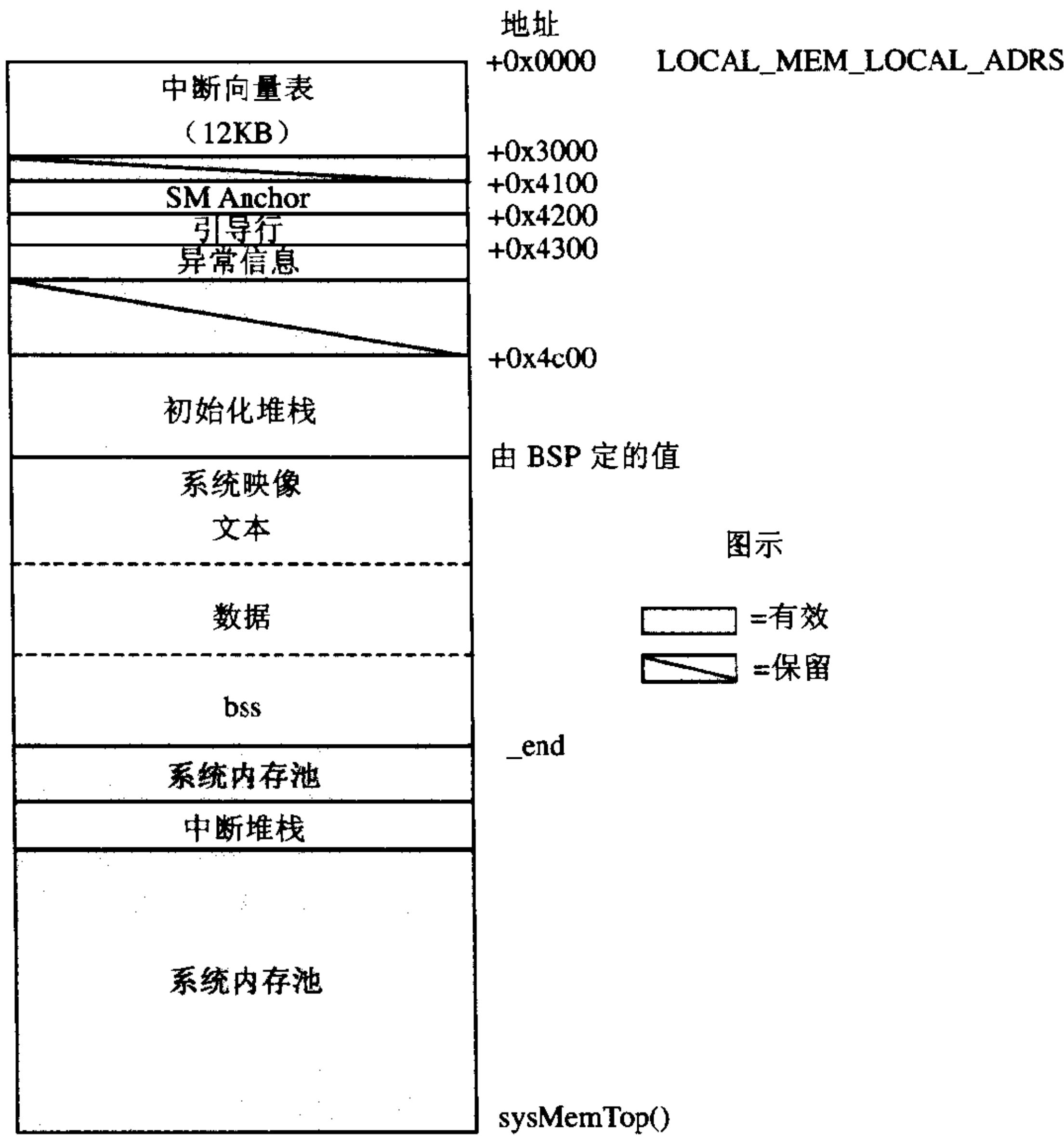


图 17.1 VxWorks 系统内存配置（PowerPC）

在图 17.1 中，所有显示的地址是相对于特定目标板的存储起点而言。对于每一个目标机，在包

含存储配置 INCLUDE_MEMORY_CONFIG 之下，内存起始地址宏定义为 LOCAL_MEM_LOCAL_ADRS（在图 17.1 中对应为 0x0）。

17.2 Intel x86

VxWorks 支持 32 位 Intel x86 体系结构的处理器，支持的 CPU 包括：

- i386DX、i386SX、i386EX
- i486DX、i486SX、i486DX2、i486DX4
- NS486SXF
- Pentium
- Pentiumpro

同时也支持与上述处理器协助工作的数学协处理器——片上 FPU（Floating Point Unit）并提供软件模拟浮点库。

i386DX 是 Intel 公司最先推出的 32 位 CPU，它内部结构具有 32 位的数据总线和 32 位的地址总线，且寻址空间为 4GB。i386SX 内部结构具有 16 位的数据总线和 24 位的地址总线，且寻址空间为 16MB。i386EX 是在 i386SX 基础上集成了 PC 兼容的外部设备，它内部结构具有 16 位的数据总线和 26 位的地址总线，且寻址空间为 64MB。i386DX 和 i386SX 使用的数学协处理器分别为 386DX 和 386SX。

i486DX 是 Intel 公司推出的第二代 32 位 CPU，它内部结构具有一个 i387 兼容的片上 FPU 和一个 8KB 统一标准的高速缓存。i486SX 没有片上 FPU，但它有一个 486SX 数学协处理器。i486DX2 使用一个时钟倍频器（doubler）提高内部时钟。

Pentium 是 Intel 公司推出的第三代 32 位 CPU，它内部结构具有 64 位的数据总线和 32 位的地址总线，且寻址空间为 4GB。它的一些新特点是独立的 8KB 指令高速缓存和 8KB 数据高速缓存，一个极佳的计数器，分支预测，执行的流水线技术和一个后台写 MESI 数据高速缓存协议。

NS486SXF 是一个嵌入式 i486 类型的 32 位 CPU。它具有一个 i486 兼容的指令装置（仅在保护模式下），集成了 PC 兼容的外部设备（包括 PCMCIA 接口）以及一个 1KB 指令高速缓存。它淘汰了实地址模式、虚拟 8086（virtual-8086）模式、MMU 和 FPU。

x86 体系结构支持三种操作模式：保护（protect）模式、实地址（real-address）模式和虚拟 8086（virtual-8086）模式。保护模式是 32 位处理器本身的操作模式，在这种模式下，所有的指令和结构特点发挥最高的性能和兼容性。实地址模式提供了 Intel 8086 处理器的编程环境。虚拟 8086 模式让处理器在保护模式下执行 8086 软件及多任务环境。VxWorks 使用 32 位保护模式。

17.2.1 MMU 支持

1. 总体概述

x86 体系结构的存储管理包括段式存储管理和页式存储管理。段式存储管理为每一个程序或任务分配独立的代码、数据、堆栈模块。页式存储管理支持一个常规的页需求（demand-paged）虚拟存储系统。在保护模式下，处理器经过两个阶段的地址转换到达物理地址：通过段式存储管理，逻辑地址到线性地址转换；通过页式存储管理，线性地址到物理地址转换。两者转换在 VxWorks 里是

一目了然的。

全局描述表（GDT, Global Descriptor Table）被段式存储管理所使用。页目录和页表被 MMU 所使用。VxWorks 使用平坦（flat）段模式。代码和数据段被映射到整个 32 位 4GB 线性地址空间。在 GDT 里，两者的段描述使用同样的基地址 0 和同样的段界限 4GB。GDT 在 sysALib.s 文件中被描述，且复制到描述地址（LOCAL_MEM_LOCAL_ADRS + GDT_BASE_OFFSET）处。

当使用页式存储管理时，线性地址空间被划分成定长页面（默认配置为 4KB）。在页目录入口点对应页表，在页表入口点对应物理存储页。线性地址的第 22 位到第 31 位提供页目录入口的偏移量。线性地址的第 12 位到第 21 位提供选择页表入口的偏移量。线性地址的第 0 位到第 11 位提供物理地址页的偏移量。页大小在 x86 BSP 的 config.h 文件中被定义为 4KB。sysPhysMemDesc[]表包含了线性空间到物理地址映射的一些信息。PCI 存储映射 I/O 空间也在这个结构中被描述。如果系统使用页式存储管理，PCI 存储映射 I/O 空间在 MMU 初始化后可以使用。



注意

i386 MMU 没有写保护能力。

2. PentiumPro MMU

PentiumProde 的增强型 MMU 支持两个额外的页属性位：全局位和页级别直写/回写（write-through/write-back）位。

当设置时，全局位标志一个全局页。当页被标志成全局页时，在寄存器 CR4 中页全局使能位被设置，ISR 的地址必须在字边界上开始。这样，允许处理器假定最低两位是 0。当寄存器 CR3 被装载或任务切换发生时，在 TLB 里的页入口起效。这一位的提供是为了防止频繁地使用 TLB 中已经刷新的页。

页级别直写/回写位（PWT）控制着单独页或页表的直写或回写式高速缓存方式。当设置 PWT 位时，打开关联页或页表的直写高速缓存。当清除这一位时，打开关联页或页表的回写高速缓存。

在表 17-4 中的宏描述了这些属性位，具体位置在 sysLib.c 文件里的物理地址描述表 sysPhysMemDesc[]中。

表 17-4 属性位宏定义描述

宏 定 义	描 述
VM_STATE_WBACK	页使用 write-back 高速缓存方式
VM_STATE_WBACK_NOT	页使用 write-through 高速缓存方式
VM_STATE_GLOBAL	设置页全局位
VM_STATE_GLOBAL_NOT	不设置页全局位

系统支持两种页大小：4KB 和 4MB。4KB 页的线性地址划分成三个部分。

- 页目录入口：第 22 位到第 31 位；
- 页表入口：第 12 位到第 21 位；

- 页偏移量：第 0 位到第 11 位。
- 4MB 页的线性地址划分成两个部分：
- 页目录入口：第 22 位到第 31 位；
 - 页偏移量：第 0 位到第 21 位。

通过使用 VM_PAGE_SIZE 宏配置页大小，系统默认配置是 4KB。为了重新配置 4MB 页，我们必须更改 config.h 文件中的 VM_PAGE_SIZE 变量值。

17.2.2 高速缓存

i386 系列的 CPU 没有片上高速缓存。其余 x86 CPU 使用 VxWorks 高速缓存库进行高速缓存管理，这个库为 cacheLib。如果页被打开，每页的高速缓存属性在 sysPhusMemDesc[] 表里进行描述。

i486 系列的 CPU 有一个 8KB 统一标准的片上高速缓存，是为了存储指令和数据。这个片上高速缓存是一个 write-through 高速缓存。CRO 里的 CD 和 NW 标志控制着全部的系统存储的高速缓冲存储。在页目录或页表入口上，PCD 和 PWT 控制着页级别高速缓冲存储。在 cacheLib 库里，WBINVD 指令被用于刷新高速缓存。

Pentium 系列的 CPU 有独立的指令和数据高速缓存，每一个高速缓存大小为 8KB。数据高速缓存支持 write-throug 和 write-back 更新方式。在页表入口里的 PWT 标志控制 write-back 方式。Pentium 的 MESI 数据高速缓存协议维持其他处理器的高速缓存和外部 cache 更新的一致性。对软件而言，MESI 协议的操作是透明的。

在 PC386 和 PC486 BSP 里，write-throug 更新方式是作为默认配置而被使用。在 config.h 文件中，如果 USER_D_CACHE_MODE 是 CACHE_COPYBACK，则系统使用 write-back 更新方式。

17.2.3 中断管理

异常和 NMI 中断向量在 0~31 范围里分配。在这个范围里没有分配的向量被保留，是为了将来使用。在 32~255 范围里的向量，提供给可屏蔽中断。对于 x86 体系结构，没有独立的中断堆栈，ISR 使用被中断任务的堆栈。这样，堆栈必须足够大，以便管理最坏情况下的嵌套中断。

x86 体系结构通过使用 EFLAGS 寄存器的 IF 标志打开或关闭全部的可屏蔽中断。一个外部中断控制器管理多级次序中断，最流行的中断控制器是 Intel 8259 PIC (Programmable Interrupt Controller)。VxWorks 支持 8259 PIC，将其当做一个中断控制器驱动程序来看待。

intLock() 和 intUnlock() 函数控制着 IF 标志。sysIntEnablePIC() 和 sysIntDisablePIC() 函数控制着指定的 PIC 中断级。sysLib.c 文件里的全局变量 sysVectorIRQ0 是 IRQ0 的中断向量号。



注意

中断控制器设备的新标准已经建立。将来，现有的 API 将被新标准接口所替代。

17.2.4 中断描述表

中断描述表 (IDT, Interrupt Descriptor Table) 占用的地址范围是从 0x0 到 0x800，它也被称为

中断向量表，如图 17.2 所示的 VxWorks 系统内存配置（x86）。默认异常处理器处理从 0x0 到 0x1F 的向量号，默认中断处理器处理从 0x20 到 0xFF 的向量号。

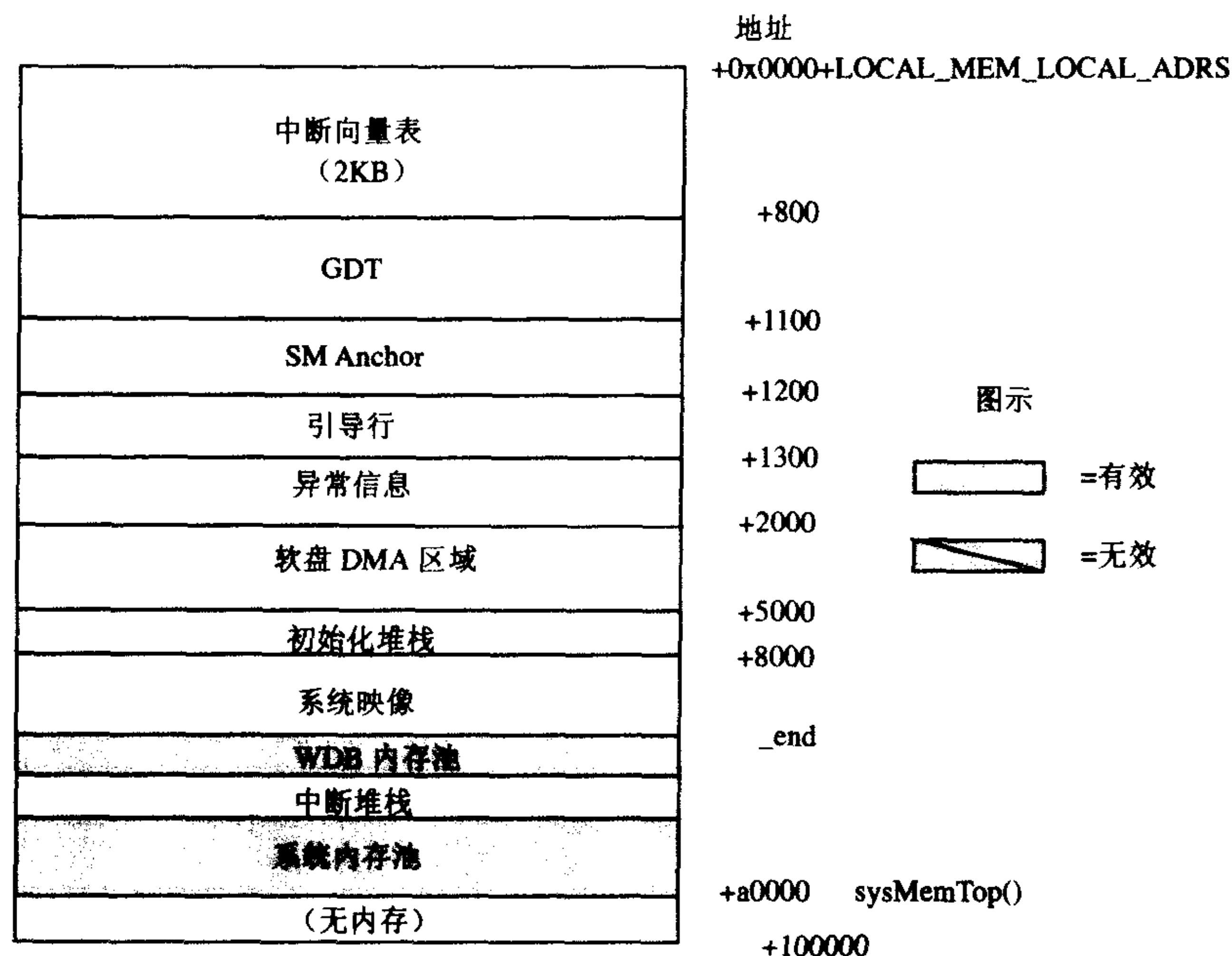


图 17.2 VxWorks 系统内存配置（x86）

默认情况下，向量号 0x20 到 0x2f 被映射给 IRQ 0~15 级别的中断。为了重新定义这个基地址，编辑 sysLib.c 文件中的 sysVectorIRQ0。

对于向量号 0x0 到 0x11，非任务门使用，而仅仅只是中断门。默认情况下，向量号 0x12 到 0xff 是 trap 门，但这些可以通过重新定义全局变量 sysIntIdtType 来改变。

在中断门和 trap 门之间的差别是它对 IF 标志有效：使用中断门清除 IF 标志，它防止来自于当前中断处理器的其他中断。

每一个 IDT 向量包含下面信息：

- offset，中断处理器的偏移量；
- selector 0x0018，在异常 GDT 里的第三个描述符（代码）；
0x0020，在中断 GDT 里的第四个描述符（代码）；
- descriptor3，特权级别；
- descriptor1，现场位。

中断处理器调用 intEnt()函数和保存易失寄存器的值（eax、edx 和 ecx），然后调用 ISR，而 ISR 通常用 C 语言编写。最后，处理器恢复保存寄存器的值和调用 intExit()函数。

x86 没有设计中断堆栈，中断的堆栈结构创建在被中断的任务上。这样，每一个任务需要为中

断嵌套准备额外堆栈空间；额外空间总计大小与 ISR 和潜在的嵌套级别有关。

有一些设备驱动（与厂商、配置等有关）在 IRQ7 上产生一个偶然的中断，但 IRQ7 被并行接口使用。每产生一次中断，全局变量 sysStrayIntCount 就增加，一个虚拟的 ISR 被连接来处理这些中断。

芯片在两种格式里产生异常堆栈结构，有关异常类型为：(EIP + CS + EFLAGS) 或 (ERROR + EIP + CS + EFLAGS)。

17.2.5 浮点支持

VxWorks 支持数学协处理器和片上 FPU，具体设置是通过 INCLUDE_HW_FP 配置宏来实现。同时，通过定义 INCLUDE_HW_FP 和 INCLUDE_SW_FP，系统也支持软浮点仿真。系统调用 fppProbe() 函数实现浮点协处理器的初始化和安装操作。当浮点协处理器异常时，系统使用软浮点仿真 FPU 指令。全局变量 sysCoprocesor 保存数学协处理器或 FPU 的类型。

17.3 MIPS

MIPS 类型的 RISC 处理器可以运行在小模式或大模式下。然而，VxWorks 内核仅仅只是支持大模式操作。这一部分主要从下面几个方面描述 MIPS 结构：

- MMU 支持；
- 高速缓存问题；
- 中断管理；
- 浮点支持。

17.3.1 MMU 支持

MIPS 处理器包含一个最小的存储管理单元，通常作为后备缓冲器而被提到。

VxWorks 不支持这个后备缓冲器。MIPS 处理器提供三种不同操作模式：用户模式、核心模式、超级用户模式。VxWorks 内核总是运行在核心模式下。

17.3.2 高速缓存

MIPS R3000 微处理器对于所有目标板用户，提供一个标准的高速缓冲存储接口。R3000 包括一个可变大小的指令高速缓存和一个 write-through 模式的数据高速缓存。在数据存取上，R3000 执行块重新装满操作；在指令访问上，执行指令流操作。通过 CP0 配置寄存器或其他 CP0 寄存器配置块的大小，且与指令高速缓存和数据高速缓存的大小一致，同时也取决于处理器本身。

R4000 微处理器提供片上指令和数据高速缓存系统及可选的二级高速缓存。高速缓存的大小可以是固定的或可变的，具体取决于目标板的配置。VxWorks 在默认情况下，对于高速缓存的操作模式是回写模式。然而，可以通过配置 CP0 配置寄存器使高速缓存在直写模式下运行。

R4650/R4640 微处理器，并没有像其他以 R4000 为基础的处理器一样具有同样的高速缓存配置。R4650/R4640 处理器像 R4000 一样提供片上高速缓存系统；然而，R4650 使用 CP0 寄存器配置高速缓存，并且设置数据高速缓存和指令高速缓存的大小为 8KB。对于 R4650/R4640 微处理器，默认情况下，VxWorks 在 write-back 模式下使用这些高速缓存。

17.3.3 中断管理

MIPS 处理器包含六个外部硬中断和两个软中断输入。在许多情况下，六个外部中断不能满足需要，于是目标板制造商在一个或多个输入线上实现多个中断。例如 MIPS R3010 浮点单元，有六个不同的异常条件可以导致 MIPS 处理器产生一个浮点中断，但只是使用一个中断输入线。软件的工作就是通过使用浮点控制和状态寄存器，深一层的解码这个中断。

不像 Motorola 680x0 系列处理器，MIPS 处理器相等地对待异常和中断。当两者中某个条件发生时，微处理器转移到两个存储驻留向量（常规异常向量或 UTLB 错过的异常向量），并提供状态和原因寄存器允许软件决定哪一种情形的芯片在里面。MIPS 硬件并不切换到中断堆栈或异常堆栈，也不产生一个 IACK 周期。所有这些功能必须在软件或板级硬件（这样，一个 IACK 周期是一个板级硬件功能）里执行。

1. VxWorks 的执行

尽管 MIPS 处理器缺乏许多类似于 Motorola 680x0 类型处理器的中断处理特点，但对于 VxWorks 来说，两个体系结构的中断处理非常相似。针对 MIPS 处理器的 VxWorks，用户可以绑定它们的中断处理到 256 个入口向量表上。这个静态的数据结构，并不需要用户再定位操作。这样，对 `intVecBaseSet()` 函数来说是无效的。

与 Motorola 680x0 处理器的 VxWorks 一样，函数 `intConnect()` 和 `intVecSet()` 缚上中断处理到任何给定的向量。任何没有在 `../h/arch/mips/ivMips.h` 定义的向量是不可用的。向量号应该定义在目标板指定的头文件中。

数据结构 `intPrioTable` 在 `sysLib.c` 文件中可以找到，它是与目标板相关的数据结构，支持 MIPS 八个中断条件的处理。它被 `excLib` 库中的异常和中断处理程序调用用户缚上的中断处理所使用。现在的搜寻运算法则依赖于 MIPS 处理器中断线和向量表入口的一一对应的映射。例如在一个 MIPS 处理器的 VME 总线板上，VME 总线中断在 3 号中断线上进入，于是 VME 总线字段在表的入口 3 里。

结构中的每一个入口具有 4 个字段：中断 ID、向量号、掩码字段和复用字段。现在的搜寻运算法则使用字段 2 和字段 4，但是依然要求每一个入口具有 4 个 32 位字的宽度。当中断产生时，处理器映射待解决的处理线到相应的表入口。下一步，读取复用字段。如果这个字段是 0，在 BSR 表里字段 2 作为中断向量号被取出。否则，认为是作为复用函数被调用，且字段 4 作为一个参数传递给这个函数。复用程序的工作是计算正确的中断向量号，并返回给处理器。处理器可以调用用户通过 `intConnect()` 或 `intVecSet()` 已经安装的程序。VME 总线中断是一个非常好的复用中断的示例。

VME 总线有 7 个中断级（1~7）。在大部分 VME 总线板上，VME 总线中断级 1~7 是绑在一个中断线上。这需要软件在一个悬而未决的中断级上来判断 VME 总线中断和多路中断条件。根据用户定义的数据结构 `intPrioTable`，低级别中断处理可以实现这个目标。

函数 `excIntStub()` 判断 VME 总线中断和读取 `intPrioTable` 里相应的入口。这个入口提供了字段 4 里的一个基本向量和字段 2 里的一个复用函数。函数 `excIntStub()` 读取字段 4。如果不为空，取出字段 2 里的数据（中断复用程序的地址）。函数 `excIntStub()` 然后调用这个程序，把字段 4 里的内容传递给它，中断复用程序返回待解决中断的中断号。然而，如果字段 4 为空，这个中断线只有一个中

断与它联合。这样，用户可以通过编辑数据结构 `intPrioTable` 完全地配置中断系统。所有的中断复用函数都应用汇编语言来编写，以便提高执行速度。

2. VME 总线特性

只是在用户没有应答不得不进行处理中断条件下，VME 总线中断处理才执行。在 `config.h` 文件中，如果用户定义条件宏 `VME_VECTORED` 为 `TRUE`，所有的 VME 总线中断通过 VME 总线复用程序应答，且 `VxWorks` 返回的中断向量与 VME 总线通过 `IACK` 周期返回的中断向量一致。具体参考 `sysLib.c` 文件和 `intPrioTable` 数据结构。

中断处理调度允许多个 VME 总线板出现在同一的 VME 总线中断级上，且在用户处理级上不需要深一层的掩码。没有打开向量中断，用户可能依然绑定 VME 总线中断，但是现在的 VME 总线应答职责在用户装载的中断处理里（见 `sysBusIntAck()` 函数）。VME 总线向量不在与 `IACK` 周期返回的中断向量一致；替代它的一致性为 VME 总线中断级的向量号，且向量号被用户定义在目标板指定的头文件中。除 VME 总线中断外，向量中断的规格不改变任何中断的处理。所有必要的中断应答程序需要在 `sysLib.c` 或 `sysALib.s` 文件中提供。参见 `../target/config/templateMips` 目录下的模板文件。

3. 中断应答

因为 MIPS 处理器不提供一个 `IACK` 周期，缚上中断处理器来应答（清除）中断是用户的职责。函数 `sysAutoAck()` 必须作为任何可能中断的默认处理被提供。如果一个伪造的中断产生，`sysAutoAck()` 应答这个中断是它的职责。如果一个中断没有被应答，`VxWorks` 企图连续处理这个中断。如果这种情况发生，热复位未能自动启动目标机，因为 `VxWorks` 环境变量已经被中断堆栈溢出而被破坏。于是需要冷启动，从而拷贝这个变量到内存中。针对 MIPS 处理器的 `VxWorks`，它有独立的中断堆栈和用户任务堆栈。

下面是 `intPrioTable` 数据结构的原型：

```
typedef struct
{
    ULONG intCause;           /* 中断原因 */
    ULONG bsrTableOffset;     /* BSR 表索引 */
    ULONG statusReg;          /* 中断级别 */
    ULONG pad;
} PRIO_TABLE;
PRIO_TABLE intPrioTable[8] =
{
    {CAUSE_SW1, (ULONG) IV_SWTRAP0_VEC, 0x0100, 0}, /* sw trap 0 */
    {CAUSE_SW2, (ULONG) IV_SWTRAP1_VEC, 0x0200, 0}, /* sw trap 1 */
    {CAUSE_IP3, (ULONG) IV_SPURIOUS_VEC, 0x0400, 0},
    {CAUSE_IP4, (ULONG) IV_SPURIOUS_VEC, 0x0800, 0},
    {CAUSE_IP5, (ULONG) IV_SPURIOUS_VEC, 0x0400, 0},
    {CAUSE_IP6, (ULONG) sysFpaDeMux, 0x2000,
    IV_FPA_BASE_VEC}, /* FPA muxed */
    {CAUSE_IP7, (ULONG) IV_SPURIOUS_VEC, 0x4000, 0},
    {CAUSE_IP8, (ULONG) IV_UART_VEC, 0x8000, 0} /* bus error */
};
```


17.3.4 浮点支持

早期 MIPS 类型的处理器与浮点元件连接在一起。R3000 微处理器是一个与 R3010 浮点单元连接在一起的处理器。R3000 微处理器通过执行协处理器指令访问 R3010 FPU。R3010 有 32 个单精度通用目的寄存器，也可以作为 16 个双精度浮点寄存器使用。

R4000 微处理器也封装了一个片上浮点元件。像 R3000 一样，R4000 处理器通过执行协处理器指令访问 FPU。由 CPO 状态寄存器的 FR 位而定，R4000 可以有 32 个 32 位单精度浮点寄存器（FR 位为 0），也可以作为双精度浮点寄存器成对使用。或它可以有 32 个 64 位浮点寄存器（FR 位为 1），它可以支持单或双精度变量。VxWorks 使用 32 个 32 位单精度浮点寄存器配置。

R4650 微处理器与 R4000 和 R3000 稍微有一些不同。R4650 处理器也封装了一个浮点元件；然而，这种处理器只支持单精度硬浮点。处理器可以有 16 个浮点寄存器（状态寄存器 FR=0）或 32 个浮点寄存器（状态寄存器 FR=1）。VxWorks 使用 16 个浮点寄存器配置。对于 R4650 处理器的 VxWorks 使用 GNU 软件数学库，支持双精度浮点程序。例如 `acos()`、`sin()`。

对于大多数 MIPS 处理器，芯片包括一个浮点控制和状态寄存器及一个执行修正寄存器。5 个浮点异常条件和未生效的异常条件可以通过使用浮点控制和状态寄存器打开和屏蔽它们。可是，像这样的异常操作对 VxWorks 没有影响，因为浮点异常被 VxWorks 内核忽略不计。

对于任务使用浮点，5 个浮点异常可以在任务前基于在浮点任务里通过改变浮点控制和状态寄存器打开这些异常。但是，用户必须提供操作浮点控制和状态寄存器的功能。对于浮点任务，在上下文切换时所有的寄存器被保存和恢复。这样，用户在任务基础上不需要担心存储和恢复浮点寄存器。

VxWorks 提供一系列优化转换函数，它们可以被任何带有浮点支持的任务所使用。因为所有的浮点异常对 MIPS 处理器是不与理睬的，所有单个受约束的浮点条件永远不会执行。同时，对于浮点指令仿真当前系统不支持。

对于处理器，它没有浮点单元。VxWorks 对各种数学函数提供软件仿真，例如函数 `log()`、`sin()` 等，这些数学函数通过 GNU 数学库提供。

注意

为了访问浮点单元，必须在 MIPS CPO 状态寄存器里设置，可用协处理器位（Cu1）。但是，这个操作由内核实现而用户不需要知道具体位置。

17.3.5 存储配置

图 17.3 显示了 MIPS 存储配置。在图 17.3 中，所有显示的地址是相对于特定目标板的存储起点而言。对于每一个目标机，在包含存储配置 `INCLUDE_MEMORY_CONFIG` 之下，内存起始地址宏定义为 `LOCAL_MEM_LOCAL_ADRS`。

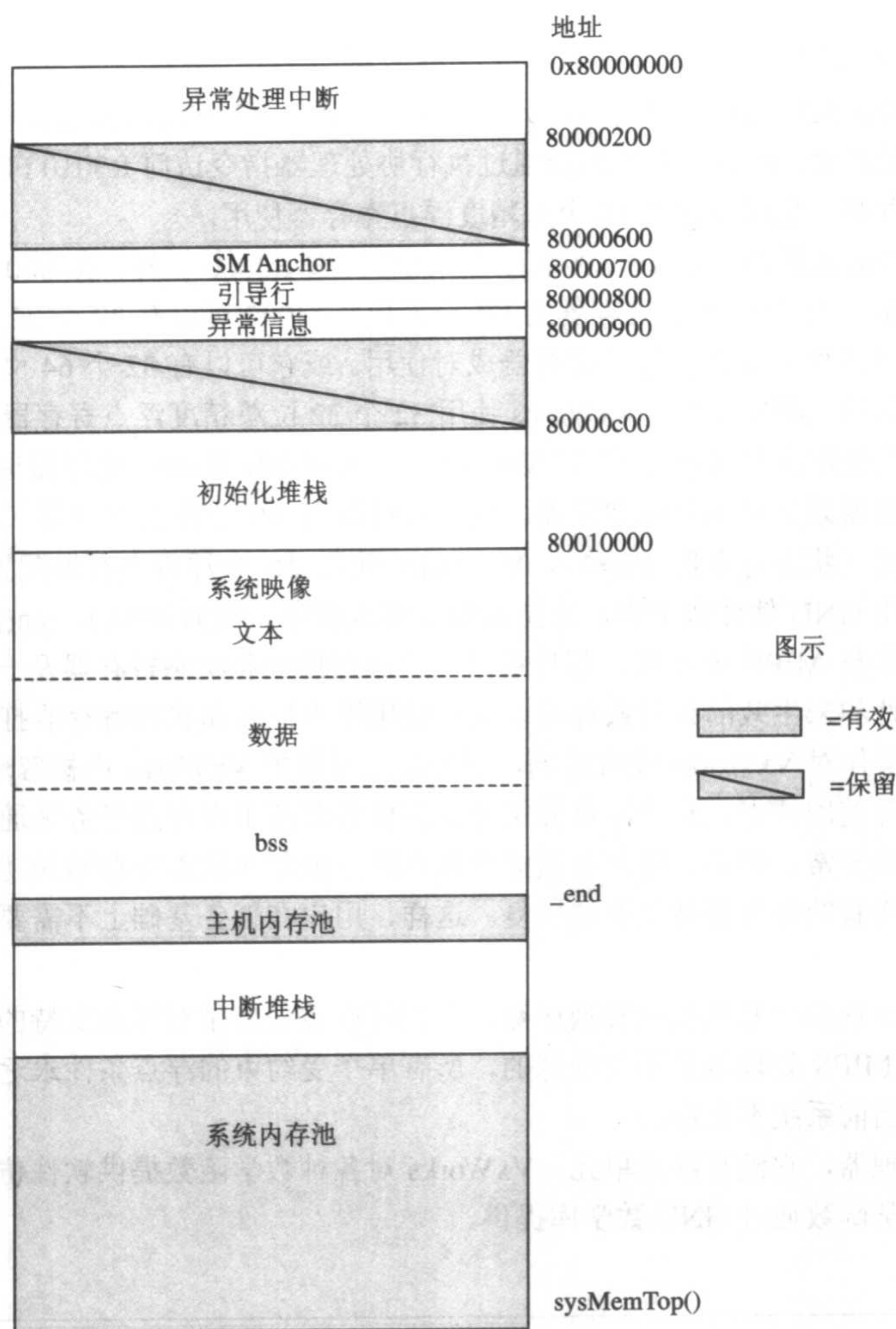


图 17.3 VxWorks 系统存储配置 (MIPS)

17.4 ARM

这一部分描述了 ARM 处理器的一些特征，当编写 VxWorks 应用时，可能需要牢记这些特征：

- 处理器模式和字节对齐；
- 中断和异常；
- 浮点支持；
- 高速缓存；
- 存储管理单元；
- WindView；

- 存储布局。

对于 ARM 体系结构的全面文档和特殊的处理器，可以参考《ARM 体系结构参考手册》和处理器相应的信息图表。

17.4.1 处理器模式和字节对齐

对于 ARM 的 VxWorks 主要执行在 32 位超级用户模式下 (SVC32)。当异常发生时，这将会导致 CPU 进入其他模式，对于大多数处理器内核发生 SVC32 模式的切换。没有代码可以执行在用户模式里。不支持 26 位模式，因为它已经被废弃。

ARM CPU 包括小模式和大模式字节对齐的支持。发布的版本仅仅只支持小模式字节对齐，但是网络应用必须转换一些数据到标准的网络对齐，因为它是大模式对齐。特别是网络应用确认通过使用 htons() 转换端口号到网络字节对齐。

有关宏和函数更多的转换字节对齐信息，请参考相关的 VxWorks 书籍。

17.4.2 中断和异常

当 ARM 中断或异常发生时，CPU 切换到异常模式中的某个模式下，而每一种模式都有一些专用寄存器。为了使处理是可重入的，VxWorks 把桩 (stub) 程序安装到 trap 中断和从异常模式到 SVC 模式的异常切换中；当在异常模式里执行时，处理程序是不可重入的。因为重入操作将破坏连接寄存器。当一个异常或基本级别的中断处理程序通过 VxWorks 的一个调用安装时，处理程序的地址被存储以便在模式切换完成后让 stub 程序使用。处理器返回到 stub 程序恢复处理器到异常发生前的状态。异常处理可以修改状态，而恢复操作是通过转换寄存器内容设置传给处理。

通常，ARM 处理器没有一个片上中断控制器。除快速中断之外，所有的中断是在 IRQ 针上实现。这样，必须在 BSP 里提供打开和屏蔽指定设备中断的程序，安装指定设备的 ISR 程序，确定中断的原因和分派正确处理程序的程序。这些程序通过设置函数指针被安装。例如在 ../target/src/drv/intrCtl 目录下的中断控制模块。设备驱动程序是通过调用函数 intConnect() 安装 ISR。

异常不同于中断，它的处理类似形成是：异常 stub 切换到 SVC 模式，然后调用任何安装的处理程序。处理程序的安装是通过调用 exeVecSet() 函数来实现，并且可以调用 exeVecGet() 函数读取处理程序的安装地址。

1. 中断堆栈

针对于 ARM 的 VxWorks 使用独立的中断堆栈，以避免从任务堆栈中分配足够大的中断堆栈来满足中断处理程序。对于 IRQ 中断模式，ARM 体系结构有一个专门的堆栈指针。但是，因为低级别中断处理代码必须是可重入的，IRQ 模式只是在处理程序的入口和出口时使用；中断破坏了 IRQ 模式连接寄存器。中断处理程序主要运行在 SVC 模式下的专门 SVC 模式中断堆栈里。

2. 快速中断

VxWorks 并不处理快速中断 (FIQ)。在 BSP 需要的时候它可以使用 FIQ，但是 FIQ 处理程序不应当调用 VxWorks 代码。如果这个功能需要，推荐机制是通过软件把 FIQ 降低到 IRQ。IRQ 处理程序可以调用 VxWorks 函数。

17.4.3 浮点支持

Tornado 2.0 不支持浮点协处理器。浮点运算作为 GNU ARM 的一部分被自由软件组织所提供，这些在 libgcc.a 库文件里。

在 ARM 上，当双精度变量是单个字时，被作为小模式保存，大部分有意义的字（包含符号位的字）放在低地址处。既不支持纯粹的大模式（68K 处理器的操作）也不支持纯粹的小模式（x86 处理器的操作）。Cygnus 增加了一个新的二进制浮点格式，littlebyte_bigword，更换 GDB 让 ARM 使用这个格式；这个作法被 ARM BSP 和 CrossWind 采用。

这个格式的选择要与 ARM 硬浮点操作相一致。然而，既然大部分 IEEE 浮点操作是纯粹的大端或小端模式，共享内存双精度变量的交换是复杂的。

17.4.4 高速缓存

ARM-7TDMI 处理器没有高速缓存或 MMU。ARM SA-110 处理器有一个 16KB 的指令高速缓存、一个 16KB 的数据高速缓存、一个写缓冲器及一个 MMU 在这个芯片上。ARM SA-1100 有一个 16KB 的指令高速缓存、一个 8KB 的数据高速缓存和一个 512 个字节的袖珍型高速缓存。ARM-710A 和 ARM-810 处理器都有一个 8KB 混合的指令和数据高速缓存，写缓冲器和一个 MMU。

对于所有 ARM 的高速缓存，高速缓存的性能必须与 MMU 一起使用，来解决高速缓存一致性问题。当 MMU 被打开时，每页的页描述选择了高速缓存模式，它可以是高速缓冲存储或非高速缓冲存储。页描述通过填充在 sysPhysMemDesc[] 结构中的定义来进行配置，该结构在 ../target/config/bspname/sysLib.c 文件中。

ARM 类型的 VxWorks 不支持 ARM 片上高速缓存的解锁和锁操作，因为不是所有的 ARM 片上高速缓存支持 cache 锁和解锁。这样，cacheLock() 和 cacheUnlock() 函数对 ARM 目标机是无效的操作，并且返回值总是 ERROR。

函数 cacheClear() 和 cacheInvalidate() 的作用取决于 CPU 类型和上面的高速缓存。这些高速缓存包括：

- SA-110 和 SA-1100 高速缓存；
- ARM-710A 高速缓存；
- ARM-810 高速缓存。

17.4.5 存储管理单元 (MMU)

VxWork 提供两种级别的虚拟存储管理。基本级别与 VxWorks 捆绑在一起，完全级别需要可选产品 VxVMI。ARM SA-110、SA-1100、ARM-710A 和 ARM-810 处理器两者都支持；ARM-7TDMI 因为没有 MMU，两者都不支持。

17.4.6 存储布局

对于所有 ARM 处理器，VxWorks 存储布局是一样的。图 17.4 展示了存储布局。

对于 ARM 目标板，图 17.4 中所有展示的地址相对于它的起始内存而言。在包含 INCLUDE_MEMORY_CONFIG 配置下，内存起始地址宏定义为 LOCAL_MEM_LOCAL_ADRS。

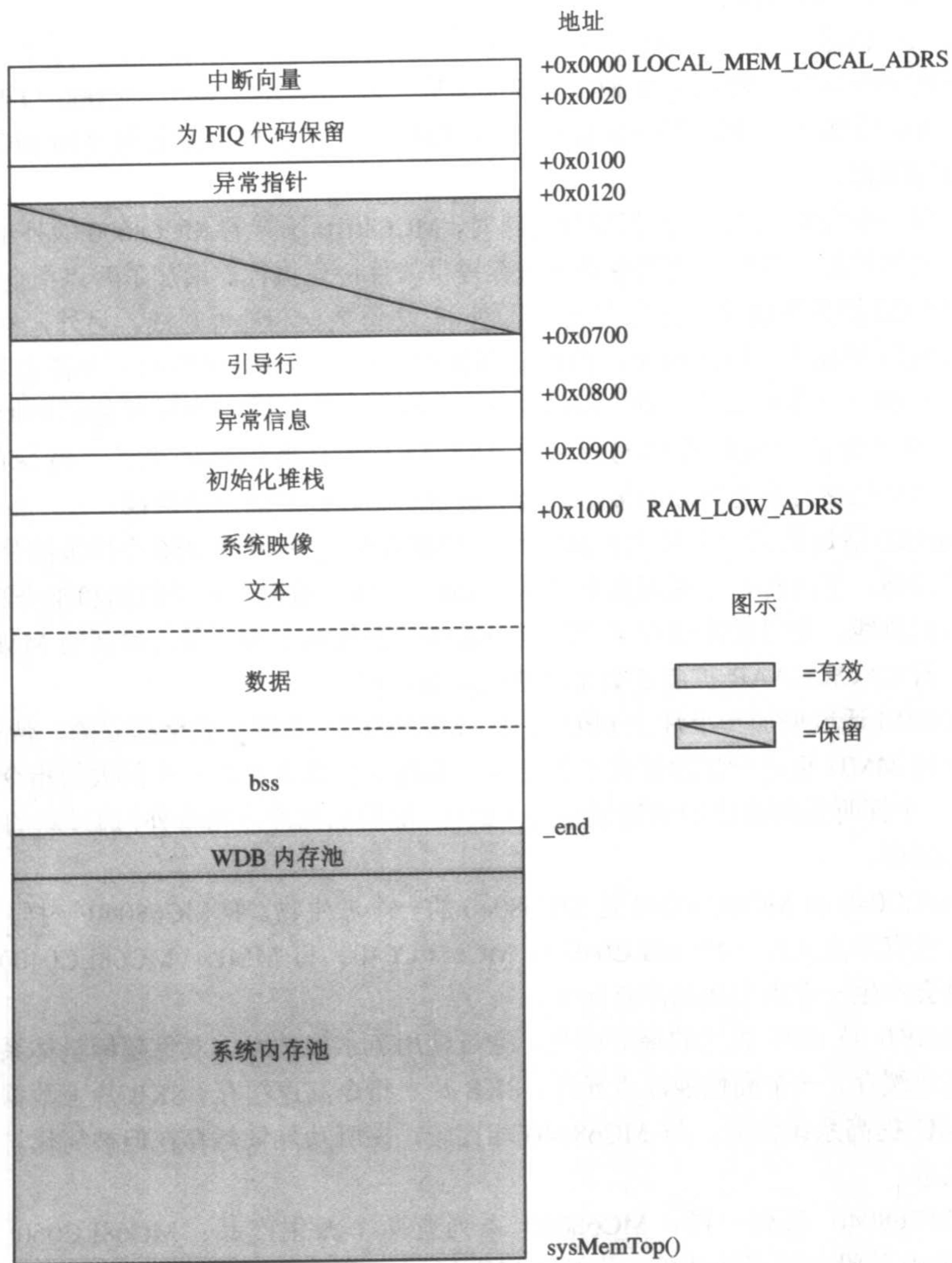


图 17.4 VxWorks 系统存储布局 (ARM)



注意

初期的堆栈和系统映像地址在 BSP 里配置。

17.5 Motorola 68K

68K 类型的最初微处理器是 MC68000, 它有一个 16 位数据通路和一个 24 位地址总线, 提供 16MB 地址空间访问。除只需要 MC6800 十分之一的功率外, MC68HC000 与 MC6800 完全兼容。MC68HC001 增加了一个静态的 8 位或 16 位数据通路的选择。完全兼容的 MC68008 仅仅只是一个 8 位数据通路。

当采用一个完整的 32 位内部数据通路时, MC68010 维持着 MC68000 的外部总线宽度。异常堆栈结构格式被扩展, 循环模式指令高速缓存提供软件快速执行。增加了断点指令 BKPT。

MC68020 把数据通路改变成完全内部/外部 32 位数据和地址总线。另外, 一个 256 个字节的片上指令高速缓存减少了执行时间。协处理器接口允许扩展, 其中包括一个浮点元件 (MC68881 和 MC68882) 和一个存储管理元件 (MC68851)。值得注意的是指令设置包括增加位字段操作、十进制填充、不可分割的读修改到写指令 CAS/CAS2 和众多的地址模式。特殊的高速缓存寄存器 CACR/CAAR 包括对高速缓存的维护。处理器自动管理独立的中断堆栈。

MC68030 通过增加一个片上 MMU 元件 (MC68851) 和一个 256 个字节的片上数据高速缓存扩展了它的线路。在内部, 它采用真实的 harvard-bus 体系结构, 对于数据和指令流使用独立的 32 位地址和数据总线。分段高速缓存填充和同步总线性能提高了吞吐量。除新的 MMU 指令之外, 高速缓存寄存器 CACR/CAAR 扩展了数据高速缓存的维护。

MC68040 通过增加一个片上 FPU 元件 (MC68882) 和两个完全独立的、带有 4KB 指令和数据高速缓存的 MMU 更进一步地优化系统性能。多路执行单元允许一个巨大的指令平行度。在多任务应用里, 一个探听控制器维护高速缓存的一致性。除附加的浮点指令外, 通过新的指令 CINV/CPUSH 管理高速缓存。

MC68LC040 和 MC68EC040 是 MC68040 的一个派生物。像 MC68040 一样, 它们是一个整体单元。但是没有浮点元件 (MC68LC040 和 MC68EC040) 和 MMU (MC68EC040)。万一执行了浮点指令, 将会产生一个未生效的浮点异常。

MC68060 是 68K 类型的最后产品。通过使用流水线技术、双重超标量体系结构执行元件、一个分支高速缓存、一个高性能浮点元件、8KB 片上指令高速缓存、8KB 片上数据高速缓存以及双重片上 MMU 提高系统性能。与 MC68040 相比较, 探听功能依然存在但被简化。仅仅使用一根线控制探听状态。

像 MC68040 系列一样, MC68060 系列有两个派生产品: MC68LC060 和 MC68EC060。MC68LC060 除没有浮点元件外, 其他与 MC68060 一样具有同样的执行元件和 MMU。MC68EC060 除没有浮点元件和 MMU 外, 其他与 MC68060 一样具有同样的执行元件。

MC68302 是把智能的外围设备和 MC68000/MC68008 处理器核心集成在一起的一个芯片。其他 MC683xx 系列, 包括 MC68330、MC68331、MC68332 和 MC68340, 使用 32 位的处理器核心。这个 32 位的处理器核心是 MC68020, 没有字段操作、高速缓存、独立的中断堆栈、协处理器支持、十进制填充、不能分割的 CAS/CAS2 指令及特殊的地址模式。取消了低能量指令, 增加了表管理。

这个部分余下的内容主要从下面几个方面来描述 68K 体系结构:

- MMU 支持;

- 高速缓存问题;
- 中断管理;
- 浮点支持。

17.5.1 MMU 支持

VxWorks 支持 MC68030、MC68040 和 MC68060 的片上 MMU。回顾 VxWorks 虚拟内存, 包括基本虚拟内存库——vmBaseLib 和可选产品 VxVMI 库——vmLib。其中 vmBaseLib 库与 VxWorks 捆绑在一起, 是 BSP 的一个默认配置。如果 VxVMI 已经安装, 只需要在 config.h 文件中把 INCLUDE_MMU_BASIC 改变为 INCLUDE_MMU_FULL。

在 MMU 应用过程中, 可能需要更改 sysLib.c 文件和 config.h 文件。在 sysLib.c 文件中, 为了支持虚拟内存需要增加头文件和数据结构。如下所示:

```
/* 增加头文件 */
#include "private/vmLibP.h"
/* 增加全局数据结构 */
PHYS_MEM_DESC sysPhysMemDesc [] =
{
    ...
}
```

具体信息参考 sysLib.c 文件。至于 config.h 文件, 需要定义虚拟内存的支持。如下所示:

```
#define INCLUDE_MMU_BASIC /* 支持基本级别的 MMU */
```



注意

由于 VxWorks 的局限性, 当使用 68K 的 MMU 时, 虚拟地址必须与物理地址相等。

17.5.2 高速缓存问题

MC68000、MC68010 和 MC683xx 类型的处理器不提供片上高速缓存。其他 68K 类型的处理器使用 VxWorks 的高速缓存库——cacheLib 进行高速缓存的管理。

不像 MC68030, MC68040 提供总线探听以保证高速缓存的一致性。总线发现外部访问内存, 刷新或使内部高速缓存相应的线无效。对于数据高速缓存, 总线探听允许使用 chapbook 模式, 其结果是提高了性能。但是, 为了保证总线探听工作的正确, 所有设备 (包括任何总线接口) 直接 (也就是说设备带有 DMA 功能) 刷新内存必须有硬件支持总线探听。

在 MC68040 上的探听功能是通过两个信号来控制: SC1 和 SC2。VxWorks 不支持 10 模式。01 模式提供完整的探听功能, 当需要探听功能时, 在 VxWork 中推荐使用这种模式。

在 MC68060 上, 硬件支持探听功能。但在处理器和二级总线主控制器之间共享的内存应当是 write-through 模式。对于该处理器, cacheDmaMallo() 函数总是返回一个内存块指针, 而高速缓存处于 write-through 模式下。

17.5.3 中断管理

系统硬件中断的中断向量在 2~255 范围内。由于考虑来自于总线错误的异常，所以系统保留/重新定义 64 之前的向量。其余向量是用户可定义的，典型情况是板上设备和非板上设备使用这些中断向量。

与每一个外部中断联合的是一个 1~7 的中断级。中断级的目的是把中断区分优先次序。例如非屏蔽中断是最高级（7），其他（例如串行芯片接收到字符）是在次要级上。设计者应当把中断级别 7 保留给灾难的事件，例如系统复位或系统失效。在非屏蔽中断接口中，不应当使用 VxWorks 的互锁临界数据结构。

对于 MC68000，向量表出现在内存的底部，即地址为 0x0 处。对于 MC68010 和 MC68010 以上的处理器，可以使用 `intVectBaseSet()` 函数替换这个表，但在 VxWorks 实际应用中，这种做法很少被采用。

系统硬件中断不同于应用开发者使用 VxWorks 函数。VxWorks 支持 `intConnect()` 函数，它允许 C 函数与任何中断关联。`intConnect()` 的第一个参数是连接中断向量的偏移量，第二个参数是连接到中断向量上的 C 函数的地址，第三个参数是传递给 C 函数的一个参数。函数原型如下：

```
intConnect (VOIDFUNCPTR *vector, VOIDFUNCPTR routine, int parameter);
```

当中断发生时，调用与中断关联的 C 函数，然后从中断中简单地返回。通常情况下，把这个与中断关联的 C 函数称为 ISR（Interrupt Service Routine）。

事实上，中断并不是直接调用 C 函数。代而替之的是 `intConnect()` 增加了保存必要的寄存器代码，安装堆栈和传递参数，然后调用连接的 C 函数。从这个函数返回的时候，它恢复保存的寄存器，堆栈和退出中断。

中断产生后，处理器从数据总线读取向量偏移量或在自动向量导引情况，基于中断级选择向量偏移量。自动向量导引需要处理器额外的一些支持，但设备产生的向量偏移量作为标准操作。对于 MC68000/MC68302 处理器，偏移量来自逻辑地址 0x0。对于 MC68010 和 MC68010 以上的处理器，偏移量来自向量表底部，而这个地址通过 `intVectBaseSet()` 函数来定义。传统上，应用程序使用内存的底部存放向量表。

许多情况下，通过一个可编程的中断控制器来管理各种扩展外围设备的中断级和向量。当提供多种用途的中断结构时，这种弹性方式是值得做的。

在 MC68000、MC68010 和 MC68060 处理器上，不支持独立的中断堆栈。在这些处理器上的 ISR 使用被中断任务的堆栈。对于 MC68020 和 MC68020 以上的处理器，所有的 ISR 使用同一个中断堆栈，根据 `kernelInit()` 函数指定的参数，在系统启动时分配和初始化这个堆栈。这个堆栈应该足够大，以便满足最坏情况下可能出现的嵌套中断。

当 ISR 直接使用 MC68881/2 浮点协处理器时，必须明确地使用 `fppALib` 提供的函数保留和恢复协处理器寄存器。这是因为通过 `intConnect()` 函数创建的中断驱动程序代码并没有保存和恢复浮点寄存器。

当任务导致硬件异常时（例如非法的指令或总线错误），仅仅只是该任务被挂起，而系统继续运行。但是，当 ISR 导致异常时，对于系统异常没有请求安全保护。这样，系统将会重新启动。

17.5.4 浮点支持

在 MC68040 之前, Motorola 68K 类型的处理器是使用 MC68881/2 浮点协处理器来支持硬浮点操作。这两个协处理器的功能几乎是同等的, 只是上下文保存/恢复有少许差异, 而对内部操作系统而言通常是重要的一点。对于 MC68000/MC68010 处理器, 尽管一些设计者已经发现通过使用存储映射 I/O 或 CPU 空间仿真的方式与它们相接合, 但事实上浮点协处理器并不容易与它们相接合。

解决方案是特别的, 然而 VxWorks 并不支持。通过调用 `fppProbe()` 函数, VxWorks 可以发现浮点协处理器是否存在。除 MC68040 外, 对于“透明的”浮点协处理器的软件仿真, 在 VxWorks 里不提供软件仿真库。

MC68040 包括一个内部浮点单元, 除控制、状态和数据寄存器编程模式外, 它提供有效的 MC68881/2 指令集合的子集。除高级别抽象功能 (例如三角学、对数等) 不包括外, 提供基本浮点算法和处理功能。这些软件的仿真是由 MC68040 浮点软件包所提供, 且软件包的版本号为 2.1 并来自于 Motorola 公司, 在 VxWorks 里已经包含了这个浮点软件包。这个浮点软件包使用了 MC68040 硬件所提供的全部浮点支持。

当执行未支持的指令时, 指定的异常处理被调用, 而异常处理将调用浮点软件包。尽管上述的异常是值得注意的, 但依然允许执行 MC68881/2 指令。异常处理也提供给其他浮点异常 (例如浮点除零、溢出和下溢)。

通过调用 `mathHardInit()` 函数, 安装 MC68040 异常处理。当在 `configAll.h` 文件中定义 `INCLUDE_MC68881` 时, `usrConfig.c` 文件将调用该函数。

如同 MC68040 一样, MC68060 仅支持 MC68881/2 指令集合的子集。而子集与 MC68040 所支持的有一点差异。这些不支持的浮点指令的仿真被 MC68060 软件包所提供。这个软件包包含了不支持的整数和浮点指令的仿真, 所有这些都被集成到 VxWorks 里。

17.5.5 存储配置

除 MC68060 处理器没有中断堆栈外, 所有的 MC680x0 处理器的 VxWorks 存储配置是一样的。图 17.5 显示了存储配置。

在图 17.5 中, 所有显示的地址是相对于特定目标板的存储起点而言。对于每一个目标机, 在包含存储配置 `INCLUDE_MEMORY_CONFIG` 之下, 内存起始地址宏定义为 `LOCAL_MEM_LOCAL_ADRS` (在这个存储配置图中对应为 0x0)。

对于 MC680x0 目标板, 图 17.5 中所有展示的地址相对于它的起始内存而言。在包含 `INCLUDE_MEMORY_CONFIG` 配置下, 内存起始地址宏定义为 `LOCAL_MEM_LOCAL_ADRS`。

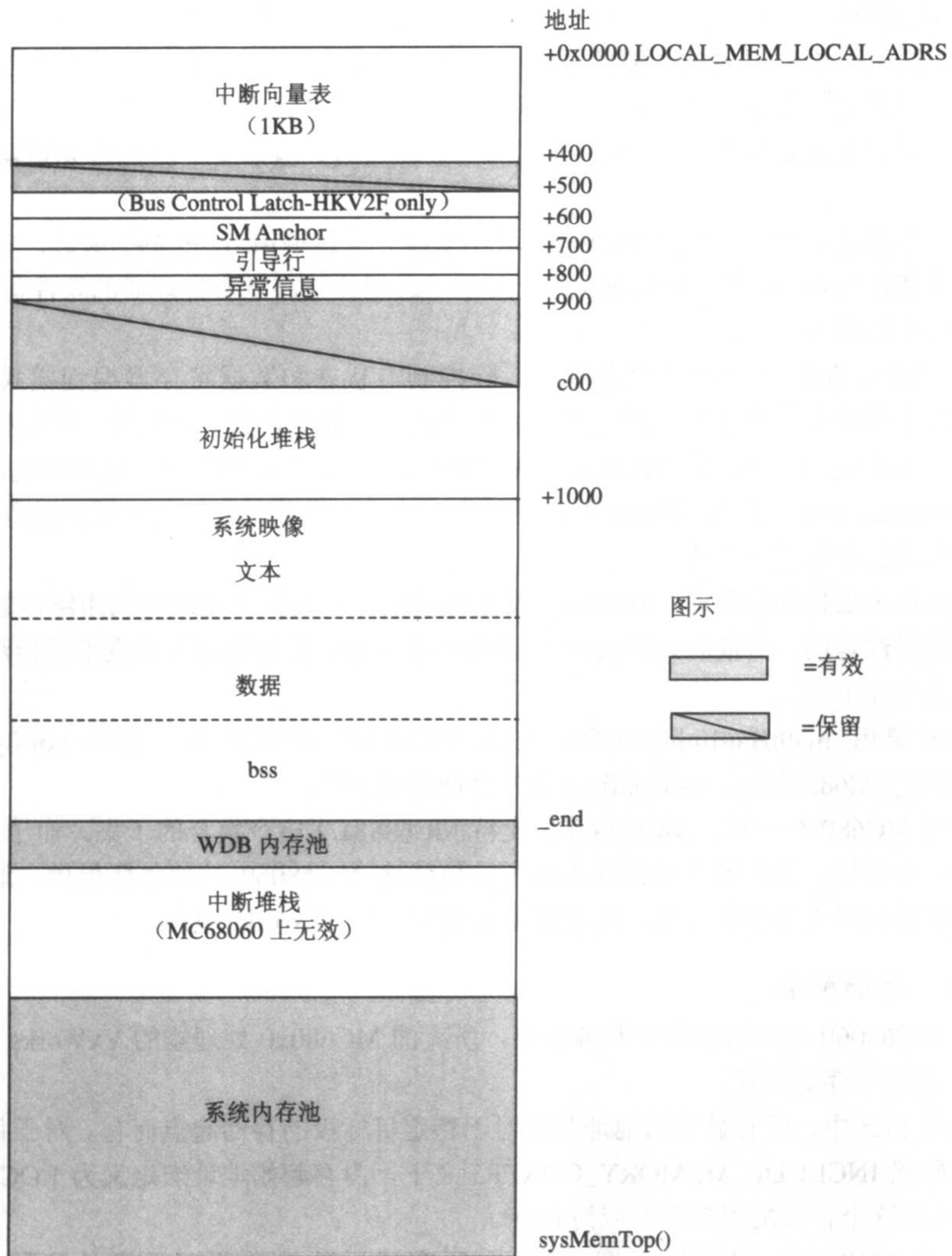


图 17.5 VxWorks 系统存储配置 (MC680x0)

小 结

这一章集中对如何在 VxWorks 系统中访问 Power PC、Intel x86、MIPS、ARM、Motorola 68K 的体系结构的有关问题进行了描述。这些问题包括 MMU 的支持、高速缓存问题、中断管理、浮点支持和存储配置等。

[G e n e r a l I n f o r m a t i o n]
书名= V x W o r k s 下设备驱动程序及 B S P 开发指南
作者= 周启平 张杨编著
页数= 3 0 2
S S 号= 1 1 2 8 2 5 7 2
出版日期= 2 0 0 4 年 0 9 月第 1 版

目录
序言

第1章外部设备及设备驱动程序概述

- 1.1 外部设备
- 1.2 外部设备的分类
- 1.3 I / O设备的数据传送方式
- 1.4 设备驱动程序
- 1.5 设备驱动程序的主要功能
- 1.6 设备驱动程序的组成部分
- 1.7 设备驱动程序的相关概念

小结

第2章VxWorks下设备及设备驱动程序

- 2.1 VxWorks下的设备
- 2.2 VxWorks下设备驱动程序
- 2.3 VxWorks下常用设备驱动程序简要描述

小结

第3章VxWorks下设备驱动程序的分析

- 3.1 VxWorks下设备驱动程序在系统中的层次
- 3.2 VxWorks的设备驱动程序表
- 3.3 VxWorks下常用设备的驱动程序源文件
- 3.4 VxWorks下设备驱动程序的配置
- 3.5 VxWorks下设备驱动程序常用函数

小结

第4章驱动程序的轮询和中断处理

- 4.1 概述
- 4.2 轮询
- 4.3 中断处理
- 4.4 其他设计需要考虑的事项

小结

第5章VxWorks下编写设备驱动程序的方法

- 5.1 概述
- 5.2 开发前资料的收集
- 5.3 宏定义及C语言可以调用的汇编函数
- 5.4 存储映射I / O与端口I / O
- 5.5 设备驱动程序错误处理
- 5.6 VxWorks下设备驱动程序编程规范
- 5.7 高速缓存的一致性问题

小结

第6章编写字符设备驱动程序

- 6.1 I / O系统回顾
- 6.2 一步步编写字符型设备的驱动程序
- 6.3 字符设备驱动程序的完整模板

小结

第7章编写串行设备驱动程序

- 7.1 串行设备概述
- 7.2 虚拟设备ttyDrv
- 7.3 编写串行设备驱动程序
- 7.4 串行设备的安装
- 7.5 串行设备驱动程序完整的模板

小结

第8章编写块设备驱动程序

8 . 1	V x W o r k s 下的文件系统
8 . 2	一步步编写块设备驱动程序
8 . 3	完整的块设备驱动程序流程
	小结
第 9 章	编写网络设备驱动程序
9 . 1	概述
9 . 2	E N D 设备驱动程序装载过程
9 . 3	网络设备与系统的数据交换
9 . 4	一步步编写网络设备驱动程序
	小结
第 1 0 章	接触实际设备
1 0 . 1	P C I 设备驱动程序
1 0 . 2	I S A 设备驱动程序
1 0 . 3	原始文件系统分析
	小结
第 1 1 章	板级支持包
1 1 . 1	板级支持包
1 1 . 2	B S P 的职责
1 1 . 3	B S P 的组成
1 1 . 4	B S P 的开发
1 1 . 5	T o r n a d o 的目录结构
1 1 . 6	B S P 的约定和有效性
	小结
第 1 2 章	V x W o r k s 映像及启动顺序
1 2 . 1	V x W o r k s 映像类型
1 2 . 2	V x W o r k s 启动顺序简述
1 2 . 3	V x W o r k s 映像启动顺序
1 2 . 4	构造 V x W o r k s 映像
	小结
第 1 3 章	V x W o r k s 预内核初始化
1 3 . 1	预内核的初始化概述
1 3 . 2	预内核的特殊初始化函数
1 3 . 3	预内核的通用初始化代码
1 3 . 4	预内核初始化的调试
	小结
第 1 4 章	B S P 的配置
1 4 . 1	B S P 的基本配置
1 4 . 2	V x W o r k s 的基本配置
1 4 . 3	双硬盘配置
1 4 . 4	双网卡配置
	小结
第 1 5 章	板级支持包开发
1 5 . 1	板级支持包开发周期概述
1 5 . 2	板级支持包的开发环境
	小结
第 1 6 章	创建一个新的 B S P
1 6 . 1	概述
1 6 . 2	建立开发环境
1 6 . 3	编写内核启动前 B S P 的初始化代码
1 6 . 4	使用最小的内核
1 6 . 5	目标机代理和 T o r n a d o
1 6 . 6	最后的修整工作
	小结

第1 7章体系结构

1 7 . 1 P o w e r P C

1 7 . 2 I n t e l x 8 6

1 7 . 3 M I P S

1 7 . 4 A R M

1 7 . 5 M o t o r o l a 6 8 K

小结