编写设备驱动程序

文件号码 E36756–01
2012 年 10 月

ORACLE

121210@25097

目录

图

表

示例

前言

《编写设备驱动程序》提供有关为面向字符的设备、面向块的设备、网络设备、SCSI 目标和 HBA 设备以及 USB 设备开发 Oracle Solaris 操作系统 (Oracle Solaris Operating System, Oracle Solaris OS) 驱动程序的信息。本书讨论了如何为符合 Oracle Solaris OS DDI/DKI（Device Driver Interface/Driver-Kernel Interface，设备驱动程序接口/驱动程序内核接口）的所有体系结构开发多线程可重入设备驱动程序。介绍了一种常用的驱动程序编写方法，该方法允许在编写驱动程序时忽略特定于平台的问题，如字节存储顺序 (endianness) 和数据排序等。

其他主题包括：强化 Oracle Solaris 驱动程序；电源管理；驱动程序自动配置；程控 I/O；直接内存访问 (Direct Memory Access, DMA)；设备上下文管理；编译、安装和测试驱动程序；调试驱动程序以及将 Oracle Solaris 驱动程序移植到 64 位环境。

注 – 此 Oracle Solaris 发行版支持使用 SPARC 和 x86 系列处理器体系结构的系统。支持的系统可以在 Oracle Solaris OS: Hardware Compatibility Lists（Oracle Solaris OS：硬件兼容性列表）中找到。本文档列举了在不同类型的平台上进行实现时的所有差别。

在本文档中，这些与 x86 相关的术语表示以下含义：

- x86 泛指 64 位和 32 位的 x86 兼容产品系列。
- x64 特指 64 位的 x86 兼容 CPU。
- “32 位 x86”指出了有关基于 x86 的系统的特定 32 位信息。

有关支持的系统，请参见Oracle Solaris OS: Hardware Compatibility Lists（Oracle Solaris OS：硬件兼容性列表）。

目标读者

本书是为熟悉 UNIX 设备驱动程序的 UNIX 程序员编写的。虽然本书提供了概述信息，但编写本书的目的不是为了将其作为设备驱动程序的通用教程。

注 – Oracle Solaris 操作系统 (Oracle Solaris operating system, Oracle Solaris OS) 既可在 SPARC 体系结构也可在 x86 体系结构中运行。而且，Oracle Solaris OS 既可在 64 位地址空间也可在 32 位地址空间中运行。除非特别说明，否则本文档中的信息适用于所有的平台和地址空间。

本书的结构

本书分为以下各章：

- 第 1 章，Oracle Solaris 设备驱动程序概述介绍了 Oracle Solaris 平台上的设备驱动程序和关联的入口点。每种类型的设备驱动程序的入口点都列在表中。
- 第 2 章，Oracle Solaris 内核和设备树对 Oracle Solaris 内核进行了概述，并介绍了设备如何表示为设备树中的节点。
- 第 3 章，多线程针对设备驱动程序开发者介绍了 Oracle Solaris 多线程内核的各个方面。
- 第 4 章，属性介绍了一组用于使用设备属性的接口。
- 第 5 章，管理事件和排队任务介绍了设备驱动程序如何记录事件，以及如何使用任务队列在以后执行任务。
- 第 6 章，驱动程序自动配置介绍了驱动程序必须提供的用于自动配置的支持。
- 第 7 章，设备访问：程控 I/O 介绍了驱动程序用来读取或写入设备内存的接口和方法。
- 第 8 章，中断处理程序介绍了用来处理中断的机制。这些机制包括分配、注册、维护和删除中断。
- 第 9 章，直接内存访问 (Direct Memory Access, DMA) 介绍了直接内存访问 (direct memory access, DMA) 和 DMA 接口。
- 第 10 章，映射设备和内核内存介绍了用于管理设备和内核内存的接口。
- 第 11 章，设备上下文管理介绍了一组设备驱动程序用来管理用户对设备的访问的接口。
- 第 12 章，电源管理介绍了用于电源管理（一个用于管理能耗的框架）的接口。
- 第 13 章，强化 Oracle Solaris 驱动程序介绍了如何将故障管理功能集成到 I/O 设备驱动程序中、如何引入防御性编程做法，以及如何使用驱动程序强化测试工具。
- 第 14 章，分层驱动程序接口 (Layered Driver Interface, LDI) 介绍了 LDI，利用 LDI，内核模块可以访问系统中的其他设备。
- 第 15 章，字符设备驱动程序介绍了面向字符的设备的驱动程序。
- 第 16 章，块设备驱动程序介绍了面向块的设备的驱动程序。
- 第 17 章，SCSI 目标驱动程序概述了 Sun 公用 SCSI 体系结构 (Sun Common SCSI Architecture, SCSA) 和对 SCSI 目标驱动程序的要求。

- 第 18 章，SCSI 主机总线适配器驱动程序介绍了如何将 SCSA 应用到 SCSI 主机总线适配器 (Host Bus Adapter, HBA) 驱动程序。
- 第 19 章，网络设备驱动程序介绍了通用 LAN 驱动程序 (Generic LAN driver, GLD)。GLDv3 框架是 MAC 插件和 MAC 驱动程序服务例程与结构的基于函数调用的接口。
- 第 20 章，USB 驱动程序介绍了如何使用 USBA 2.0 框架编写客户机 USB 设备驱动程序。
- 第 21 章，SR-IOV 驱动程序介绍了编写 SR-IOV 设备驱动程序的要求。
- 第 22 章，编译、装入、打包和测试驱动程序提供了有关编译、链接和安装驱动程序的信息。
- 第 23 章，调试、测试和调优设备驱动程序介绍了有关调试、测试和调优驱动程序的技术。
- 第 24 章，推荐的编码方法介绍了推荐的用于编写驱动程序的编码惯例。
- 附录 A，硬件概述介绍了设备驱动程序的多平台硬件问题。
- 附录 B，Oracle Solaris DDI/DKI 服务汇总提供了设备驱动程序的内核函数表。同时指出了过时的函数。
- 附录 C，使设备驱动程序支持 64 位提供了更新设备驱动程序以在 64 位环境中运行的指导原则。
- 附录 D，控制台帧缓存器驱动程序介绍了如何为帧缓存器驱动程序添加必要的接口，以使驱动程序能够与 Oracle Solaris 内核终端仿真器进行交互。

相关书籍和文章

有关设备驱动程序接口的详细参考信息，请参见手册页第 9 节。第 9E 节 `Intro(9E)` 介绍 DDI/DKI（Device Driver Interface/Driver-Kernel Interface，设备驱动程序接口/驱动程序内核接口）驱动程序入口点。第 9F 节 `Intro(9F)` 介绍 DDI/DKI 内核函数。第 9P 节和 9S 节 `Intro(9S)` 介绍 DDI/DKI 属性和数据结构。

有关硬件以及其他与驱动程序相关问题的信息，请参见以下书籍：

- 《Device Driver Tutorial》
- 《Oracle Solaris Modular Debugger Guide》
- 《Oracle Solaris 11.1 Dynamic Tracing Guide》
- 《Multithreaded Programming Guide》
- 《STREAMS Programming Guide》

以下书籍也可能有用：

- SPARC International，《The SPARC Architecture Manual, Version 9》，Prentice Hall，1993，ISBN 978-0130992277

获取 Oracle 支持

Oracle 客户可以通过 My Oracle Support 获取电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info，或访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs（如果您听力受损）。

印刷约定

下表介绍了本书中的印刷约定。

表 P–1 印刷约定

字体或符号	含义	示例
`AaBbCc123`	命令、文件和目录的名称；计算机屏幕输出	编辑 `.login` 文件。 使用 `ls -a` 列出所有文件。 `machine_name% you have mail.`
`AaBbCc123`	用户键入的内容，与计算机屏幕输出的显示不同	`machine_name%` **`su`** `Password:`
aabbcc123	要使用实名或值替换的命令行占位符	删除文件的命令为 `rm` *filename*。
AaBbCc123	保留未译的新词或术语以及要强调的词	这些称为 *Class* 选项。 **注意**：有些强调的项目在联机时以粗体显示。
新词术语强调	新词或术语以及要强调的词	**高速缓存**是存储在本地的副本。 请**勿**保存文件。
《书名》	书名	阅读《用户指南》的第 6 章。

命令中的 shell 提示符示例

下表显示了 Oracle Solaris OS 中包含的缺省 UNIX shell 系统提示符和超级用户提示符。请注意，在命令示例中显示的缺省系统提示符可能会有所不同，具体取决于 Oracle Solaris 发行版。

表 P–2 shell 提示符

shell	提示符
Bash shell、Korn shell 和 Bourne shell	`$`
Bash shell、Korn shell 和 Bourne shell 超级用户	`#`
C shell	`machine_name%`
C shell 超级用户	`machine_name#`

第 1 部分

针对 Oracle Solaris 平台设计设备驱动程序

本手册第一部分提供了针对 Oracle Solaris 平台开发设备驱动程序的一般信息。本部分包括以下各章：

- 第 1 章，Oracle Solaris 设备驱动程序概述介绍了 Oracle Solaris 平台上的设备驱动程序和关联的入口点。每种类型的设备驱动程序的入口点都列在表中。
- 第 2 章，Oracle Solaris 内核和设备树对 Oracle Solaris 内核进行了概述，并介绍了设备如何表示为设备树中的节点。
- 第 3 章，多线程针对设备驱动程序开发者介绍了 Oracle Solaris 多线程内核的各个方面。
- 第 4 章，属性介绍了一组用于使用设备属性的接口。
- 第 5 章，管理事件和排队任务介绍了设备驱动程序如何记录事件，以及如何使用任务队列在以后执行任务。
- 第 6 章，驱动程序自动配置介绍了驱动程序必须提供的用于自动配置的支持。
- 第 7 章，设备访问：程控 I/O 介绍了驱动程序用来读取或写入设备内存的接口和方法。
- 第 8 章，中断处理程序介绍了用来处理中断的机制。这些机制包括分配、注册、维护和删除中断。

- 第 9 章，直接内存访问 (Direct Memory Access, DMA) 介绍了直接内存访问 (direct memory access, DMA) 和 DMA 接口。
- 第 10 章，映射设备和内核内存介绍了用于管理设备和内核内存的接口。
- 第 11 章，设备上下文管理介绍了一组设备驱动程序用来管理用户对设备的访问的接口。
- 第 12 章，电源管理介绍了用于电源管理功能（这是一个用于管理能耗的框架）的接口。
- 第 13 章，强化 Oracle Solaris 驱动程序介绍了如何将故障管理功能集成到 I/O 设备驱动程序中、如何引入防御性编程做法，以及如何使用驱动程序强化测试工具。
- 第 14 章，分层驱动程序接口 (Layered Driver Interface, LDI) 介绍了 LDI，利用 LDI，内核模块可以访问系统中的其他设备。

◆ ◆ ◆ 第 1 章

1 Oracle Solaris 设备驱动程序概述

本章概述了 Oracle Solaris 设备驱动程序。本章提供有关以下主题的信息：

设备驱动程序基础知识

本节介绍 Oracle Solaris 平台上的设备驱动程序及其入口点。

什么是设备驱动程序？

设备驱动程序是一种内核模块，负责管理硬件设备的底层 I/O 操作。设备驱动程序是使用标准接口编写的，内核可通过调用该标准接口与设备进行交互。设备驱动程序也可以是仅针对软件的，即模拟仅存在于软件中的设备，如 RAM 磁盘、总线以及伪终端。

设备驱动程序包含与设备进行通信时所需的所有特定于设备的代码。此代码包括一组用于系统其余部分的标准接口。就像系统调用接口可使应用程序不受平台特定信息影响一样，此接口可保护内核不受设备特定信息的影响。应用程序和内核其余部分需要非常少的特定于设备的代码（如果有）对此设备进行寻址。这样，设备驱动程序使得系统的可移植性更强，并更易于维护。

初始化 Oracle Solaris 操作系统 (Oracle Solaris operating system, Oracle Solaris OS) 后，设备会进行自标识并组织为**设备树**，即设备分层结构。实际上，设备树是内核的硬件模型。单个设备驱动程序表示为树中的一个节点，并且不包含任何子节点。此类型的节点称为**叶驱动程序**。为其他驱动程序提供服务的驱动程序称为**总线结点驱动程序**，并显示为包含子节点的节点。在引导过程中，物理设备会映射到树中的驱动程序，以便可以在需要时找到这些驱动程序。有关 Oracle Solaris OS 如何使用设备的更多信息，请参见第 2 章，Oracle Solaris 内核和设备树。

设备驱动程序按其处理 I/O 的方式进行分类。设备驱动程序分为以下三大类：

- **块设备驱动程序**－适用于可将 I/O 数据作为异步块进行处理的情况。通常，块驱动程序用于管理可物理寻址的存储介质的设备，如磁盘。
- **字符设备驱动程序**－适用于针对连续的字节流执行 I/O 操作的设备。

注 – 如果为文件系统设置了两个不同的接口，则驱动程序可同时为块驱动程序和字符驱动程序。请参见第 57 页中的 “作为特殊文件的设备”。

使用 STREAMS 模型（请参见下文）、程控 I/O、直接内存访问、SCSI 总线、USB 以及其他网络 I/O 的驱动程序都属于字符类别的驱动程序。

- **STREAMS 设备驱动程序**－字符驱动程序的子集，将 `streamio(7I)` 例程集用于内核中的字符 I/O。

什么是设备驱动程序入口点？

入口点是设备驱动程序内的一个函数，外部实体可调用此函数以访问某种驱动程序功能或运行某个设备。每个设备驱动程序都提供一组标准函数作为入口点。有关所有驱动程序类型入口点的完整列表，请参见 `Intro(9E)` 手册页。Oracle Solaris 内核使用入口点执行以下常见任务区域：

- **装入和卸载驱动程序**
- **自动配置设备**－自动配置是将设备驱动程序的代码和静态数据装入内存以在系统内注册此驱动程序的过程。
- **为驱动程序提供 I/O 服务**

根据设备执行的操作类型，不同类型设备的驱动程序具有不同的入口点集。例如，对于内存映射的面向字符的设备，其驱动程序支持 `devmap(9E)` 入口点，而块驱动程序不支持此入口点。

使用基于驱动程序名称的前缀可为驱动程序函数指定唯一的名称。通常，此前缀是驱动程序的名称，例如 `xx_open()` 代表驱动程序 `xx` 的 `open(9E)` 例程。有关更多信息，请参见第 491 页中的 “使用唯一前缀来避免内核符号冲突”。在本书后面的示例中，`xx` 用作驱动程序前缀。

设备驱动程序入口点

本节提供以下类别的入口点列表：

通用于所有驱动程序的入口点

有些操作可由任何类型的驱动程序执行，如装入模块所需的函数以及必需的自动配置入口点所需的函数。本节介绍通用于所有驱动程序的入口点类型。第 42 页中的 “通用入口点汇总”中列出了通用入口点，并包含指向手册页以及其他相关讨论的链接。

设备访问入口点

字符设备和块设备的驱动程序导出 `cb_ops(9S)` 结构，该结构定义用于块设备访问和字符设备访问的驱动程序入口点。这两种类型的驱动程序都需要支持 `open(9E)` 和 `close(9E)` 入口点。块驱动程序需要支持 `strategy(9E)`，而字符驱动程序可选择实现适用于设备类型的 `read(9E)`、`write(9E)`、`ioctl(9E)`、`mmap(9E)` 或 `devmap(9E)` 入口点的任意组合。字符驱动程序还可通过 `chpoll(9E)` 支持轮询接口。块驱动程序以及那些可使用块文件系统和字符文件系统的驱动程序可通过 `aread(9E)` 和 `awrite(9E)` 支持异步 I/O。

可装入模块入口点

所有驱动程序都需要实现可装入模块入口点 `_init(9E)`、`_fini(9E)` 和 `_info(9E)`，以便装入、卸载和报告有关驱动程序模块的信息。

驱动程序应在 `_init(9E)` 中分配并初始化所有全局资源，驱动程序应在 `_fini(9E)` 中释放其资源。

注 – 在 Oracle Solaris OS 中，只有可装入模块例程必须在驱动程序对象模块的外部可见。其他例程可具有存储类 `static`。

自动配置入口点

对于设备自动配置，驱动程序需要实现 `attach(9E)`、`detach(9E)` 和 `getinfo(9E)` 入口点。设备（如 SCSI 目标设备）在引导过程中无法标识自己时，驱动程序还可以实现可选入口点 `probe(9E)`。有关这些例程的更多信息，请参见第 6 章，驱动程序自动配置。

内核统计信息入口点

Oracle Solaris 平台提供了一组丰富的接口来维护和导出内核级统计信息（也称为 *kstat*）。驱动程序可以自由使用这些接口导出驱动程序和设备的统计信息，用户应用程序可使用这些统计信息来查看驱动程序的内部状态。提供了两个入口点来处理内核统计信息：

- ks_snapshot(9E)，可在特定时间捕获 kstat。
- ks_update(9E)，可用于根据需要更新 kstat 数据。在设置设备来跟踪内核数据但是提取该数据很耗时的情况下，ks_update() 非常有用。

有关详细信息，请参见 kstat_create(9F) 和 kstat(9S) 手册页。另请参见第 484 页中的“内核统计信息”。

电源管理入口点

提供电源管理功能的硬件设备的驱动程序可支持可选的 power(9E) 入口点。有关此入口点的详细信息，请参见第 12 章，电源管理。

系统停止入口点

对设备进行管理的驱动程序必须实现 quiesce(9E) 入口点。不管理设备的驱动程序可以将 dev_ops 结构中的 devo_quiesce 字段设置为 ddi_quiesce_not_needed()。只有当处于高 PIL（priority interrupt level，优先中断级别）且禁用了抢占的系统是单线程时，才可以调用 quiesce() 函数。因此，该函数不得阻塞。如果设备有一个已定义的重置状态配置，则作为停止操作的一部分，驱动程序应当将该设备返回为相应的重置状态。此类情况的一个示例是快速重新引导，在这种情况下，当引导到新的操作系统映像时将绕过固件。

通用入口点汇总

下表列出了所有类型驱动程序都可使用的入口点。

表 1–1 用于所有驱动程序类型的入口点

类别/入口点	使用情况	说明
cb_ops 入口点		
open(9E)	必需	获取访问设备的权限。有关其他信息，请参见： ■ 第 248 页中的“open() 入口点（字符驱动程序）” ■ 第 272 页中的“open() 入口点（块驱动程序）”
close(9E)	必需	放弃访问设备的权限。STREAMS 驱动程序的 close() 版本具有不同于字符驱动程序和块驱动程序的签名。有关其他信息，请参见： ■ 第 249 页中的“close() 入口点（字符驱动程序）” ■ 第 274 页中的“close() 入口点（块驱动程序）”
可装入模块入口点		
_init(9E)	必需	初始化可装入模块。有关其他信息，请参见：第 95 页中的“可装入驱动程序接口”
_fini(9E)	必需	准备可装入模块以进行卸载。该入口点是所有驱动程序类型所必需的。有关其他信息，请参见：第 95 页中的“可装入驱动程序接口”

表 1–1 用于所有驱动程序类型的入口点 （续）

类别/入口点	使用情况	说明
_info(9E)	必需	返回有关可装入模块的信息。有关其他信息，请参见：第 95 页中的 “可装入驱动程序接口”
自动配置入口点		
attach(9E)	必需	在初始化过程中向系统添加设备。此外，还用于恢复已暂停的系统。有关其他信息，请参见：第 101 页中的 “attach() 入口点”
detach(9E)	必需	从系统中分离设备。此外，还用于临时暂停设备。有关其他信息，请参见：第 106 页中的 “detach() 入口点”
getinfo(9E)	必需	获取特定于驱动程序的设备信息，如设备编号和相应实例之间的映射。有关其他信息，请参见： ■ 第 107 页中的 “getinfo() 入口点” ■ 第 298 页中的 “getinfo() 入口点（SCSI 目标驱动程序）”
probe(9E)	请参见说明	确定是否存在非自标识设备。对于无法进行自标识的设备，该入口点是必需的。有关其他信息，请参见： ■ 第 99 页中的 “probe() 入口点” ■ 第 294 页中的 “probe() 入口点（SCSI 目标驱动程序）”
内核统计信息入口点		
ks_snapshot(9E)	可选	捕获 kstat(9S) 数据的快照。有关其他信息，请参见：第 484 页中的 “内核统计信息”
ks_update(9E)	可选	动态更新 kstat(9S) 数据。有关其他信息，请参见：第 484 页中的 “内核统计信息”
电源管理入口点		
power(9E)	必需	设置设备的电源级别。如果不使用此入口点，请设置为 NULL。有关其他信息，请参见：第 197 页中的 “power() 入口点”
系统停止入口点		
quiesce(9E)	请参见说明	停止设备以便设备不再生成中断或者修改或访问内存。
其他入口点		
prop_op(9E)	请参见说明	报告驱动程序属性信息。除非替换 ddi_prop_op(9F)，否则此入口点是必需的。有关其他信息，请参见： ■ 第 76 页中的 “创建和更新属性” ■ 第 79 页中的 “prop_op() 入口点”
dump(9E)	请参见说明	系统出现故障时将内存转储到设备。对于出现紧急情况时要用作转储设备的任何设备，该入口点是必需的。有关其他信息，请参见： ■ 第 284 页中的 “dump() 入口点（块驱动程序）” ■ 第 306 页中的 “转储处理”

表 1–1 用于所有驱动程序类型的入口点 （续）

类别/入口点	使用情况	说明
identify(9E)	已过时	请勿使用此入口点。请在 dev_ops 结构中将 nulldev(9F) 指定给此入口点。

用于块设备驱动程序的入口点

支持文件系统的设备称为**块设备**。为这些设备编写的驱动程序称为块设备驱动程序。块设备驱动程序接受 buf(9S) 结构形式的文件系统请求，并向磁盘发出 I/O 操作以传送指定的块。文件系统的主接口为 strategy(9E) 例程。有关更多信息，请参见第 16 章，块设备驱动程序。

块设备驱动程序还可以提供字符驱动程序接口，以使实用程序能够绕过文件系统并直接访问设备。这种设备访问通常称为块设备的**原始**接口。

下表列出了块设备驱动程序可使用的其他入口点。另请参见第 41 页中的 “通用于所有驱动程序的入口点”。

表 1–2 用于块驱动程序的其他入口点

入口点	使用情况	说明
aread(9E)	可选	执行异步读取。不支持 aread() 入口点的驱动程序应使用 nodev(9F) 错误返回函数。有关其他信息，请参见： ■ 第 252 页中的 “同步 I/O 与异步 I/O 之间的差别” ■ 第 255 页中的 “DMA 传输（异步）”
awrite(9E)	可选	执行异步写入。不支持 awrite() 入口点的驱动程序应使用 nodev(9F) 错误返回函数。有关其他信息，请参见： ■ 第 252 页中的 “同步 I/O 与异步 I/O 之间的差别” ■ 第 255 页中的 “DMA 传输（异步）”
print(9E)	必需	在系统控制台上显示驱动程序消息。有关其他信息，请参见：第 284 页中的 “print() 入口点（块驱动程序）”
strategy(9E)	必需	执行块 I/O。有关其他信息，请参见： ■ 第 160 页中的 “取消 DMA 回调” ■ 第 254 页中的 “DMA 传输（同步）” ■ 第 257 页中的 “strategy() 入口点” ■ 第 255 页中的 “DMA 传输（异步）” ■ 第 289 页中的 “常规控制流程” ■ 第 352 页中的 “x86 目标驱动程序配置属性”

用于字符设备驱动程序的入口点

字符设备驱动程序通常以字节流的形式执行 I/O 操作。使用字符驱动程序的设备包括磁带机和串行端口。字符设备驱动程序还可以提供块驱动程序中不存在的其他接口，如 I/O 控制 (`ioctl`) 命令、内存映射以及设备轮询。有关更多信息，请参见第 15 章，字符设备驱动程序。

任何设备驱动程序的主要任务都是执行 I/O，并且许多字符设备驱动程序执行称为**字节流**或**字符**的 I/O。驱动程序可在设备上来回传送数据，而无需使用特定设备地址。此类型的传送与块设备驱动程序中的相反，后者部分文件系统请求会标识设备上的特定位置。

`read(9E)` 和 `write(9E)` 入口点可处理标准字符驱动程序的字节流 I/O。有关更多信息，请参见第 250 页中的 “I/O 请求处理”。

下表列出了字符设备驱动程序可使用的其他入口点。有关其他入口点的信息，请参见第 41 页中的 “通用于所有驱动程序的入口点”。

表 1–3 用于字符驱动程序的其他入口点

入口点	使用情况	说明
`chpoll(9E)`	可选	针对非 STREAMS 字符驱动程序轮询事件。有关其他信息，请参见：第 259 页中的 “对文件描述符执行多路复用 I/O 操作”
`ioctl(9E)`	可选	针对字符驱动程序执行一系列 I/O 命令。`ioctl()` 例程必须确保根据需要显式使用 `copyin(9F)`、`copyout(9F)`、`ddi_copyin(9F)` 和 `ddi_copyout(9F)` 在内核地址空间复制用户数据。有关其他信息，请参见： ■ 第 261 页中的 “`ioctl()` 入口点（字符驱动程序）” ■ 第 553 页中的 “已知的 `ioctl` 接口”
`read(9E)`	必需	从设备读取数据。有关其他信息，请参见： ■ 第 250 页中的 “向量化的 I/O” ■ 第 252 页中的 “同步 I/O 与异步 I/O 之间的差别” ■ 第 253 页中的 “程控 I/O 传输” ■ 第 254 页中的 “DMA 传输（同步）” ■ 第 289 页中的 “常规控制流程”
`segmap(9E)`	可选	将设备内存映射到用户空间。有关其他信息，请参见： ■ 第 167 页中的 “导出映射” ■ 第 172 页中的 “为用户访问分配内核内存” ■ 第 187 页中的 “将用户映射与驱动程序通知关联”

表 1–3 用于字符驱动程序的其他入口点 （续）

入口点	使用情况	说明
write(9E)	必需	将数据写入设备。有关其他信息，请参见： ■ 第 114 页中的 “设备访问函数” ■ 第 250 页中的 “向量化的 I/O” ■ 第 252 页中的 “同步 I/O 与异步 I/O 之间的差别” ■ 第 253 页中的 “程控 I/O 传输” ■ 第 254 页中的 “DMA 传输（同步）” ■ 第 289 页中的 “常规控制流程”

用于 STREAMS 设备驱动程序的入口点

STREAMS 是一个独立的编程模型，用于编写字符驱动程序。异步接收数据的设备（如终端设备和网络设备）适合实现 STREAMS。STREAMS 设备驱动程序必须提供第 6 章，驱动程序自动配置中介绍的装入和自动配置支持。有关如何编写 STREAMS 驱动程序的其他信息，请参见《STREAMS Programming Guide》。

下表列出了 STREAMS 设备驱动程序可使用的其他入口点。有关其他入口点的信息，请参见第 41 页中的 “通用于所有驱动程序的入口点”和第 45 页中的 “用于字符设备驱动程序的入口点”。

表 1–4 用于 STREAMS 驱动程序的入口点

入口点	使用情况	说明
put(9E)	请参见说明	协调以流的形式将消息从一个队列传递到下一个队列。此入口点是必需的，但是读取数据的驱动程序端除外。有关其他信息，请参见：《STREAMS Programming Guide》
srv(9E)	必需	处理队列中的消息。有关其他信息，请参见：《STREAMS Programming Guide》

用于内存映射设备的入口点

对于某些设备（如帧缓存器），提供直接访问设备内存的应用程序比字节流 I/O 更高效。应用程序使用 mmap(2) 系统调用可将设备内存映射到其地址空间。要支持内存映射，设备驱动程序需要实现 segmap(9E) 和 devmap(9E) 入口点。有关 devmap(9E) 的信息，请参见第 10 章，映射设备和内核内存。有关 segmap(9E) 的信息，请参见第 15 章，字符设备驱动程序。

定义 devmap(9E) 入口点的驱动程序通常不会定义 read(9E) 和 write(9E) 入口点，因为应用程序在调用 mmap(2) 之后会直接对设备执行 I/O 操作。

下表列出了使用 devmap 框架执行内存映射的字符设备驱动程序可以使用的其他入口点。有关其他入口点的信息，请参见第 41 页中的 “通用于所有驱动程序的入口点”和第 45 页中的 “用于字符设备驱动程序的入口点”。

表 1–5 使用 devmap 进行内存映射的字符驱动程序的入口点

入口点	使用情况	说明
devmap(9E)	必需	验证和转换内存映射设备的虚拟映射。有关其他信息，请参见：第 167 页中的 “导出映射”
devmap_access(9E)	可选	在访问映射的过程中遇到验证或保护问题时通知驱动程序。有关其他信息，请参见：第 181 页中的 “devmap_access() 入口点”
devmap_contextmgt(9E)	必需	对映射执行设备上下文切换。有关其他信息，请参见：第 182 页中的 “devmap_contextmgt() 入口点”
devmap_dup(9E)	可选	复制设备映射。有关其他信息，请参见：第 184 页中的 “devmap_dup() 入口点”
devmap_map(9E)	可选	创建设备映射。有关其他信息，请参见：第 180 页中的 “devmap_map() 入口点”
devmap_unmap(9E)	可选	取消设备映射。有关其他信息，请参见：第 185 页中的 “devmap_unmap() 入口点”

网络设备驱动程序入口点

有关使用 Generic LAN Driver v3 (GLDv3) 框架的网络设备驱动程序的入口点列表，请参见表 19–1。有关更多信息，请参见第 19 章，网络设备驱动程序中的第 355 页中的 “GLDv3 网络设备驱动程序框架”和第 356 页中的 “GLDv3 MAC 注册函数”。

用于 SCSI HBA 驱动程序的入口点

下表列出了 SCSI HBA 设备驱动程序可使用的其他入口点。有关 SCSI HBA 传输结构的信息，请参见 scsi_hba_tran(9S)。有关其他入口点的信息，请参见第 41 页中的 “通用于所有驱动程序的入口点”和第 45 页中的 “用于字符设备驱动程序的入口点”。

表 1–6 用于 SCSI HBA 驱动程序的其他入口点

入口点	使用情况	说明
tran_abort(9E)	必需	中止已传输到 SCSI 主机总线适配器 (Host Bus Adapter, HBA) 驱动程序的指定 SCSI 命令。有关其他信息，请参见：第 348 页中的 “tran_abort() 入口点”
tran_bus_reset(9E)	可选	重置 SCSI 总线。有关其他信息，请参见：第 349 页中的 “tran_bus_reset() 入口点”

表 1–6 用于 SCSI HBA 驱动程序的其他入口点 （续）

入口点	使用情况	说明
tran_destroy_pkt(9E)	必需	释放已为 SCSI 包分配的资源。有关其他信息，请参见：第 336 页中的 “tran_destroy_pkt() 入口点”
tran_dmafree(9E)	必需	释放已为 SCSI 包分配的 DMA 资源。有关其他信息，请参见：第 337 页中的 “tran_dmafree() 入口点”
tran_getcap(9E)	必需	获取 HBA 驱动程序所提供的特定功能的当前值。有关其他信息，请参见：第 343 页中的 “tran_getcap() 入口点”
tran_init_pkt(9E)	必需	分配和初始化 SCSI 包的资源。有关其他信息，请参见：第 329 页中的 “资源分配”
tran_quiesce(9E)	可选	停止 SCSI 总线上的所有活动（通常是为了进行动态重新配置）。有关其他信息，请参见：第 350 页中的 “动态重新配置”
tran_reset(9E)	必需	重置 SCSI 总线或目标设备。有关其他信息，请参见：第 348 页中的 “tran_reset() 入口点”
tran_reset_notify(9E)	可选	请求通知 SCSI 目标设备进行总线重置。有关其他信息，请参见：第 349 页中的 “tran_reset_notify() 入口点”
tran_setcap(9E)	必需	设置 SCSI HBA 驱动程序所提供的特定功能的值。有关其他信息，请参见：第 345 页中的 “tran_setcap() 入口点”
tran_start(9E)	必需	请求传输 SCSI 命令。有关其他信息，请参见：第 338 页中的 “tran_start() 入口点”
tran_sync_pkt(9E)	必需	按 HBA 驱动程序或设备同步数据视图。有关其他信息，请参见：第 336 页中的 “tran_sync_pkt() 入口点”
tran_tgt_free(9E)	可选	代表目标设备请求释放已分配的 SCSI HBA 资源。有关其他信息，请参见： ■ 第 328 页中的 “tran_tgt_free() 入口点” ■ 第 318 页中的 “传输结构克隆”
tran_tgt_init(9E)	可选	代表目标设备请求初始化 SCSI HBA 资源。有关其他信息，请参见： ■ 第 327 页中的 “tran_tgt_init() 入口点” ■ 第 315 页中的 “scsi_device 结构”
tran_tgt_probe(9E)	可选	探测 SCSI 总线上的指定目标。有关其他信息，请参见：第 328 页中的 “tran_tgt_probe() 入口点”
tran_unquiesce(9E)	可选	调用 tran_quiesce(9E)（通常是为了进行动态重新配置）之后恢复 SCSI 总线上的 I/O 活动。有关其他信息，请参见：第 350 页中的 “动态重新配置”

用于 PC 卡驱动程序的入口点

下表列出了 PC 卡设备驱动程序可使用的其他入口点。有关其他入口点的信息，请参见第 41 页中的 “通用于所有驱动程序的入口点”和第 45 页中的 “用于字符设备驱动程序的入口点”。

表 1–7 仅适用于 PC 卡驱动程序的入口点

入口点	使用情况	说明
csx_event_handler(9E)	必需	处理 PC 卡驱动程序的事件。驱动程序必须显式调用 csx_RegisterClient(9F) 函数来设置入口点，而不是使用类似 cb_ops 的结构字段。

设备驱动程序设计注意事项

从服务的使用方和提供方角度来看，设备驱动程序都必须与 Oracle Solaris OS 兼容。本节所讨论的以下问题是设计设备驱动程序过程中需要考虑的问题：

- 第 49 页中的 “DDI/DKI 功能”
- 第 51 页中的 “驱动程序上下文”
- 第 52 页中的 “返回错误”
- 第 52 页中的 “动态内存分配”
- 第 53 页中的 “热插拔”

DDI/DKI 功能

为了使驱动程序具有可移植性，提供了 Oracle Solaris DDI/DKI 接口。利用 DDI/DKI，开发者可采用标准方式编写驱动程序代码，而不必担心硬件或平台差异。本节介绍 DDI/DKI 接口的各个方面。

设备 ID

利用 DDI 接口，驱动程序可以为设备提供永久、唯一的标识符。可以使用设备 ID 标识或查找设备。此 ID 与设备的名称或编号 (dev_t) 无关。应用程序可以使用 libdevid(3LIB) 中定义的函数来读取和处理由驱动程序注册的设备 ID。

设备属性

设备或设备驱动程序的特性 (attribute) 通过**属性** (property) 指定。属性是一个名称/值对。名称是标识具有关联值的属性的字符串。属性可以由自标识设备的 FCode、硬件配置文件（请参见 driver.conf(4) 手册页）或驱动程序自身使用 ddi_prop_update(9F) 系列例程进行定义。

中断处理

DDI/DKI 解决了设备中断处理的以下方面的问题：

- 向系统注册设备中断
- 删除设备中断
- 向中断处理程序分发中断

设备中断源包含在称为 *interrupt* 的属性中，此属性既可由自标识设备的 PROM（位于硬件配置文件中）提供，也可由 x86 平台上的引导系统提供。

回调函数

某些 DDI 机制提供**回调**机制。DDI 函数提供一种在满足某个条件时调度回调的机制。在以下典型的情况下，可以使用回调函数：

- 传送已完成
- 资源已变得可用
- 超时时间已过期

回调函数在某种程度上与入口点（例如，中断处理程序）类似。允许回调的 DDI 函数期望回调函数执行特定任务。如果使用 DMA 例程，则回调函数必须返回一个值，指示是否需要在出现故障时重新调度此函数。

回调函数作为单独的中断线程执行。回调必须处理所有常见的多线程问题。

注 – 驱动程序在分离设备之前必须先取消所有已调度的回调函数。

软件状态管理

为了帮助设备驱动程序编写人员分配状态结构，DDI/DKI 提供了一组称为**软件状态管理例程**的内存管理例程，也称为**软状态例程**。这些例程可动态分配、检索以及销毁指定大小的内存项，并可隐藏列表管理的详细信息。可使用**实例编号**来标识所需内存项。此编号通常为系统指定的实例编号。

这些例程用于实现以下任务：

- 初始化驱动程序的软状态列表
- 为驱动程序的软状态实例分配空间
- 检索指向驱动程序软状态实例的指针
- 释放驱动程序软状态实例的内存
- 结束使用驱动程序的软状态列表

有关如何使用这些例程的示例，请参见第 95 页中的“可装入驱动程序接口”。

程控 I/O 设备访问

程控 I/O 设备访问是指通过主机 CPU 读/写设备寄存器或设备内存的行为。Oracle Solaris DDI 提供通过内核映射设备寄存器或内存的接口，以及从驱动程序读/写设备内存的接口。使用这些接口，通过自动管理设备和主机字节存储顺序中的任何差异，以及强制执行设备所强加的任何内存存储顺序要求，可以开发与平台和总线无关的驱动程序。

直接内存访问 (Direct Memory Access, DMA)

Oracle Solaris 平台可定义与体系结构无关的高级模型，以支持具备 DMA 功能的设备。Oracle Solaris DDI 可防止驱动程序使用特定于平台的详细信息。使用此概念，通用驱动程序可以在多个平台和体系结构上运行。

分层驱动程序接口

DDI/DKI 提供一组称为分层设备接口 (layered device interface, LDI) 的接口。利用这些接口，可以从 Oracle Solaris 内核中访问设备。该功能使开发者可以编写查看内核设备使用情况的应用程序。例如，`prtconf(1M)` 和 `fuser(1M)` 命令都可以使用 LDI，以便使系统管理员可以跟踪设备使用情况的各方面。第 14 章，分层驱动程序接口 (Layered Driver Interface, LDI) 对 LDI 进行了更详细的说明。

驱动程序上下文

驱动程序上下文是指驱动程序的当前运行环境。上下文会限制驱动程序可执行的操作。驱动程序上下文取决于调用的执行代码。驱动程序代码在以下四种上下文中执行：

- **用户上下文**。用户线程以同步方式调用驱动程序入口点时，此入口点具有**用户上下文**。即，用户线程会等待系统从调用的入口点返回。例如，通过 `read(2)` 系统调用来调用驱动程序的 `read(9E)` 入口点时，此入口点具有用户上下文。在这种情况下，驱动程序可访问用户区域，以在用户线程中复制数据。
- **内核上下文**。通过某部分内核调用驱动程序函数时，此函数具有**内核上下文**。在块设备驱动程序中，可以通过 `pageout` 守护进程来调用 `strategy(9E)` 入口点，以向设备中写入页面。由于页面守护进程与当前用户线程无关，因此在这种情况下 `strategy(9E)` 具有内核上下文。
- **中断上下文** **中断上下文**是一种限制性更强的内核上下文形式。中断上下文是在提供中断服务的情况下调用。驱动程序中断例程在中断上下文中以关联的中断级别运行。回调例程也在中断上下文中运行。有关更多信息，请参见第 8 章，中断处理程序。
- **高级中断上下文**。**高级中断上下文**是一种限制性更强的中断上下文形式。如果 `ddi_intr_hilevel(9F)` 指示某中断为高级中断，则驱动程序中断处理程序将在高级中断上下文中运行。有关更多信息，请参见第 8 章，中断处理程序。

手册页的第 9F 节介绍了每个函数所允许的上下文。例如，在内核上下文中，驱动程序不得调用 copyin(9F)。

返回错误

设备驱动程序通常不输出消息，除了异常错误（如数据损坏）。相反，驱动程序入口点应返回错误代码，以便应用程序可以确定如何处理错误。可以使用 cmn_err(9F) 函数将消息写入随后会在控制台上显示的系统日志中。

由 cmn_err(9F) 解释的格式字符串说明符与 printf(3C) 格式字符串说明符类似，前者还添加了可列显位字段的格式 %b。格式字符串的第一个字符可能具有特殊意义。在对 cmn_err(9F) 的调用中，还会指定消息 *level*，用于指示要列显的严重性标签。有关更多详细信息，请参见 cmn_err(9F) 手册页。

级别 CE_PANIC 具有使系统崩溃的负面影响。仅当系统处于不稳定状态以至继续运行将导致更多问题时，才应使用此级别。此外，调试时可以使用此级别来获取系统核心转储。不应使用 CE_PANIC 生成设备驱动程序。

动态内存分配

必须将设备驱动程序设计为可以同时处理驱动程序声明要驱动的所有连接设备。驱动程序处理的设备数不应受到限制。必须动态分配所有每设备信息。

```
void *kmem_alloc(size_t size, int flag);
```

标准内核内存分配例程为 kmem_alloc(9F)。kmem_alloc() 与 C 库例程 malloc(3C) 类似，前者添加了 flag 参数。flag 参数可以是 KM_SLEEP 或 KM_NOSLEEP，用于指示没有所需大小的内存空间时调用方是否要阻塞。如果设置了 KM_NOSLEEP 并且内存不可用，kmem_alloc(9F) 将返回 NULL。

kmem_zalloc(9F) 与 kmem_alloc(9F) 类似，但前者还可以清除已分配内存的内容。

注 – 内核内存是有限资源，并且不可分页，它还会与用户应用程序和内核其余部分争用物理内存。分配大量内核内存的驱动程序可导致系统性能降低。

```
void kmem_free(void *cp, size_t size);
```

可使用 kmem_free(9F) 将通过 kmem_alloc(9F) 或 kmem_zalloc(9F) 分配的内存返回到系统。kmem_free() 与 C 库例程 free(3C) 类似，但前者添加了 size 参数。驱动程序**必须**跟踪每个已分配对象的大小，以便在以后调用 kmem_free(9F)。

热插拔

本手册没有重点介绍热插拔信息。如果按照本书中介绍的规则和建议编写设备驱动程序，则您的应用程序应该能够处理热插拔。需要特别指出的是，请确保您的驱动程序中的自动配置（请参见第 6 章，驱动程序自动配置）和 detach(9E) 都能正常工作。此外，如果要设计使用电源管理的驱动程序，则应遵循第 12 章，电源管理中介绍的信息。SCSI HBA 驱动程序可能需要向其 dev_ops 结构中添加 cb_ops 结构（请参见第 18 章，SCSI 主机总线适配器驱动程序），以利用热插拔功能。

早期版本的 Oracle Solaris OS 要求可热插拔的驱动程序包括 DT_HOTPLUG 属性，但现在已不再需要该属性。不过，驱动程序编写者可视情况自由加入和使用 DT_HOTPLUG 属性。

◆ ◆ ◆ 第 2 章

Oracle Solaris 内核和设备树

设备驱动程序需要作为操作系统的组成部分透明地工作。理解内核工作方式是了解设备驱动程序的前提条件。本章概述了 Oracle Solaris 内核和设备树。有关设备驱动程序工作方式的概述，请参见第 1 章，Oracle Solaris 设备驱动程序概述。

本章介绍有关以下主题的信息：

什么是内核？

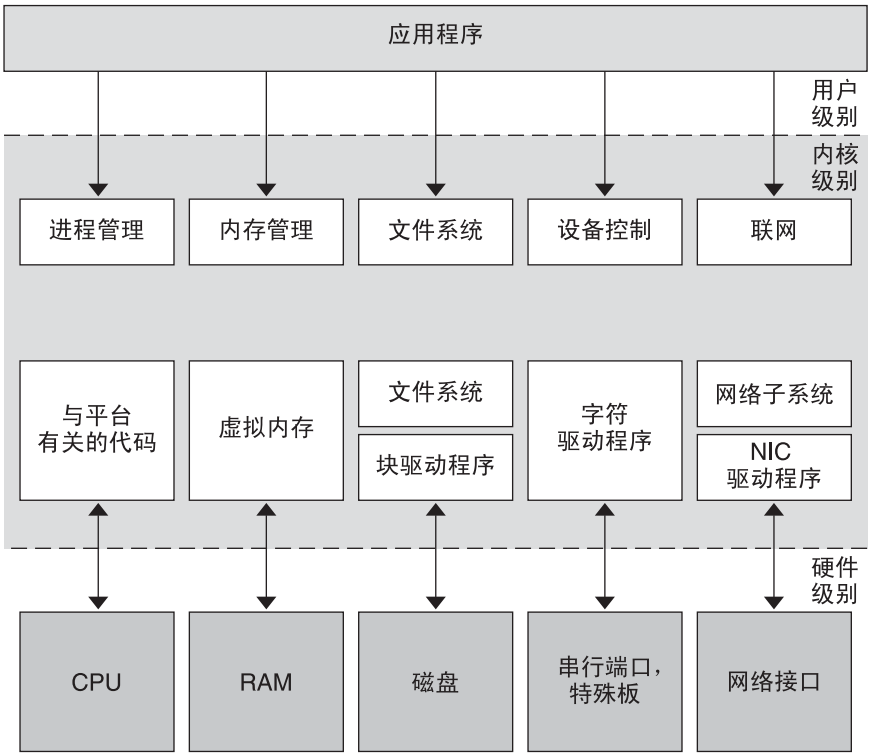
Oracle Solaris 内核是用于管理系统资源的程序。内核将应用程序与系统硬件隔离，并为它们提供基本系统服务，如输入/输出 (input/output, I/O) 管理、虚拟内存和调度。内核由需要时动态装入内存的对象模块组成。

Oracle Solaris 内核在逻辑上可分为两个部分：第一部分称为**内核**，用于管理文件系统、调度和虚拟内存。第二部分称为 *I/O* **子系统**，用于管理物理组件。

内核提供了一组接口，供可通过**系统调用**访问的应用程序使用。Reference Manual Collection 的第 2 部分对系统调用进行了介绍（请参见 `Intro(2)`）。某些系统调用用于调用设备驱动程序以执行 I/O 操作。**设备驱动程序**是可装入的内核模块，用于管理数据传输，同时将内核的其余部分与设备硬件隔离。为了与操作系统兼容，设备驱动程序需要能够提供多线程、虚拟内存寻址以及 32 位和 64 位操作之类的功能。

下图解释了内核的工作机制。内核模块用于处理来自应用程序的系统调用。I/O 模块用于与硬件通信。

图 2–1 Oracle Solaris 内核



内核通过以下功能提供对设备驱动程序的访问：

- **设备至驱动程序映射**。内核将维护**设备树**。树中的每个节点都表示一个虚拟设备或物理设备。内核通过将设备节点名称与系统中安装的驱动程序集进行匹配，从而将每个节点绑定到驱动程序。仅当存在驱动程序绑定时，应用程序才能访问设备。
- **DDI/DKI 接口**。DDI/DKI（Device Driver Interface/Driver-Kernel Interface，设备驱动程序接口/驱动程序内核接口）接口可对驱动程序和内核、设备硬件以及引导/配置软件之间的交互进行标准化。这些接口使驱动程序独立于内核，并且改进了驱动程序在相同体系架构下不同操作系统发行版间的可移植性。
- **LDI**。LDI（Layered Driver Interface，分层驱动程序接口）是 DDI/DKI 的扩展。LDI 允许内核模块访问系统中的其他设备。LDI 还允许确定内核当前使用的设备。请参见第 14 章，分层驱动程序接口 (Layered Driver Interface, LDI)。

多线程执行环境

Oracle Solaris 内核是多线程的。在多处理器计算机上，多个内核线程可以运行内核代码并且可以并发运行。内核线程也可能随时被其他内核线程抢先。

内核的多线程特征对设备驱动程序强加了某些附加限制。有关多线程注意事项的更多信息，请参见第 3 章，多线程。必须对设备驱动程序进行编码，使其在许多不同线程请求时按需运行。对于每个线程，驱动程序必须处理重叠的 I/O 请求的争用问题。

虚拟内存

Oracle Solaris 虚拟内存系统的完整概述超出本书范围，但在讨论设备驱动程序时使用了两个特别重要的虚拟内存术语：虚拟地址和地址空间。

- **虚拟地址**。**虚拟地址**是由内存管理单元 (memory management unit, MMU) 映射到物理硬件地址的地址。驱动程序可直接访问的所有地址都属于内核虚拟地址。内核虚拟地址引用**内核地址空间**。
- **地址空间**。**地址空间**是一组**虚拟地址段**。每个地址段都是一个连续范围的虚拟地址。每个用户进程都拥有一个称为**用户地址空间**的地址空间。内核拥有其自己的地址空间，称为**内核地址空间**。

作为特殊文件的设备

设备在文件系统中表示为**特殊文件**。在 Oracle Solaris OS 中， 这些文件驻留在 `/devices` 目录分层结构中。

特殊文件的类型可以为**块**，也可以为**字符**。该类型表示了设备驱动程序的种类。驱动程序可以实现这两种类型。例如，磁盘驱动程序导出字符接口以供 `fsck(1)` 和 `mkfs(1)` 实用程序使用，导出块接口以供文件系统使用。

每个特殊文件都与一个**设备编号** (`dev_t`) 关联。设备编号由**主设备号**和**次要设备号**组成。**主**设备号标识与特殊文件关联的设备驱动程序。**次要**设备号由设备驱动程序创建，供其用来进一步标识特殊文件。通常，次要设备号是一种编码，用于标识驱动程序应访问的设备实例以及应执行的访问类型。例如，次要设备号可以标识用于备份的磁带设备，并可指定完成备份操作后需要将磁带反绕。

DDI/DKI 接口

在 System V Release 4 (SVR4) 中，设备驱动程序与 UNIX 内核其余部分之间的接口被标准化为 DDI/DKI。DDI/DKI 在 Reference Manual Collection 的第 9 部分中进行介绍。第 9E 节介绍驱动程序入口点，第 9F 节介绍驱动程序可调用的函数，而第 9S 节介绍设备驱动程序使用的内核数据结构。请参见 `Intro(9E)`、`Intro(9F)` 和 `Intro(9S)`。

DDI/DKI 旨在对设备驱动程序与内核其余部分之间的所有接口进行标准化并进行说明。此外，无论处理器体系结构是 SPARC 还是 x86，DDI/DKI 都允许任何运行 Oracle Solaris OS 的计算机的驱动程序的源代码和二进制代码保持兼容。仅使用 DDI/DKI 中包含的内核函数的驱动程序称为**符合 *DDI/DKI* 标准的设备驱动程序**。

DDI/DKI 允许您为运行 Oracle Solaris OS 的任何计算机编写与平台无关的设备驱动程序。通过这些二进制代码兼容的驱动程序，您可以更方便地将第三方硬件和软件集成到运行 Oracle Solaris OS 的任何计算机中。DDI/DKI 与体系结构无关，从而允许同一驱动程序在一组不同的计算机体系结构中工作。

平台无关性是通过在以下方面设计 DDI 实现的：

- 动态装入和卸载模块
- 电源管理
- 中断处理
- 从内核或用户进程访问设备空间，即寄存器映射和内存映射
- 使用 DMA 服务从设备访问内核或用户进程空间
- 管理设备属性

设备树概述

Oracle Solaris OS 中的设备表示为互连的设备信息节点树。设备树描述特定计算机的已装入设备的配置。

设备树组件

系统将会生成树结构，其中包含有关引导时连接到计算机的设备的信息。此外，系统正常运行时也可以动态重新配置设备树。设备树从表示平台的根设备节点开始。

根节点下面是设备树的分支。分支由一个或多个总线结点设备和一个终止叶设备组成。

总线结点设备可为设备树中的从属设备提供总线映射和转换服务。PCI－PCI 网桥、PCMCIA 适配器和 SCSI HBA 都是结点设备的示例。编写结点设备驱动程序的讨论仅限于 SCSI HBA 驱动程序的开发（请参见第 18 章，SCSI 主机总线适配器驱动程序）。

叶设备通常为外围设备，如磁盘、磁带、网络适配器、帧缓存器等。叶设备驱动程序可以导出传统的字符驱动程序接口和块驱动程序接口。通过这些接口，用户进程可在存储设备或通信设备中读取和写入数据。

系统通过以下步骤来生成树：

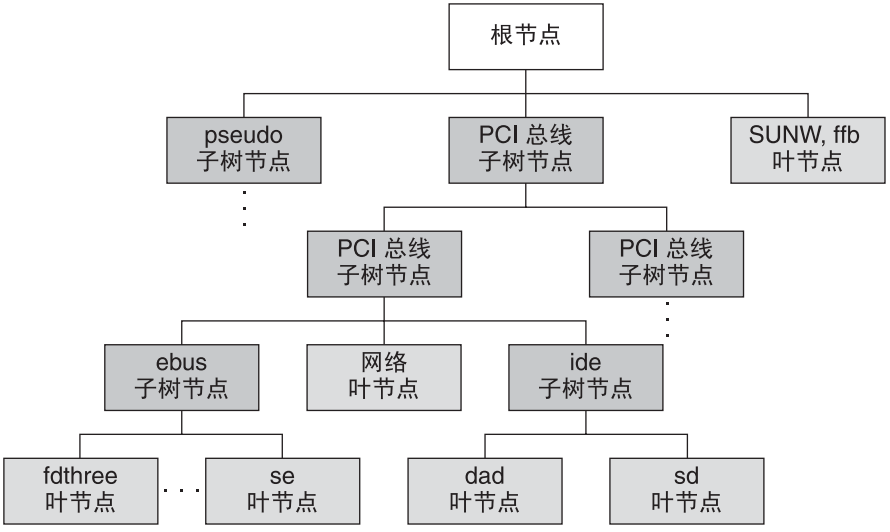
1. CPU 经过初始化后搜索固件。
2. 主要固件（OpenBoot、基本输入/输出系统 (Basic Input/Output System, BIOS) 或 `Bootconf`）初始化并创建包含已知或自标识硬件的设备树。

3. 当主要固件在设备中发现兼容固件时，主要固件将初始化该设备并检索设备属性。
4. 该固件将查找并引导操作系统。
5. 内核从树的根节点开始，搜索匹配的设备驱动程序并将该驱动程序绑定到设备。
6. 如果设备是结点，则内核会查找固件尚未检测到的子设备。内核会将所有子设备都添加到树的子树节点下面。
7. 内核从步骤 5 开始重复该过程，直到无需再创建设备节点。

每个驱动程序都会导出设备操作结构 `dev_ops(9S)`，以定义设备驱动程序可以执行的操作。设备操作结构包含通用操作（如 `attach(9E)`、`detach(9E)` 和 `getinfo(9E)`）的函数指针。该结构同时还包含了一组与特定总线结点驱动程序操作相关的函数指针，以及一组与特定叶结点设备驱动程序操作相关的函数指针。

树结构将在节点之间创建父子关系。此父子关系是体系结构无关性的关键。当叶驱动程序或总线结点驱动程序本质上需要依赖于体系结构的服务时，该驱动程序会请求其父级提供该服务。采用此方法，不管计算机或处理器的体系结构是什么，驱动程序都可以正常运行。下图显示了典型的设备树。

图 2–2 示例设备树



子树节点可以有一个或多个子节点。叶节点表示各个设备。

显示设备树

设备树可以采用以下三种方式显示：

- `libdevinfo` 库提供访问设备树内容的编程接口。
- `prtconf(1M)` 命令显示设备树的完整内容。
- `/devices` 分层结构是设备树的表示形式。使用 `ls(1)` 命令查看该分层结构。

注 – `/devices` 仅显示将驱动程序配置到系统中的设备。`prtconf(1M)` 命令显示所有设备节点，而不管系统中是否存在设备驱动程序。

---

libdevinfo 库

`libdevinfo` 库提供用于访问所有公共设备配置数据的接口。有关接口列表，请参见 `libdevinfo(3LIB)` 手册页。

prtconf 命令

以下摘录的 `prtconf(1M)` 命令示例显示了系统中的所有设备。

```
Memory size: 128 Megabytes
System Peripherals (Software Nodes):

SUNW,Ultra-5_10
    packages (driver not attached)
        terminal-emulator (driver not attached)
        deblocker (driver not attached)
        obp-tftp (driver not attached)
        disk-label (driver not attached)
        SUNW,builtin-drivers (driver not attached)
        sun-keyboard (driver not attached)
        ufs-file-system (driver not attached)
    chosen (driver not attached)
    openprom (driver not attached)
        client-services (driver not attached)
    options, instance #0
    aliases (driver not attached)
    memory (driver not attached)
    virtual-memory (driver not attached)
    pci, instance #0
        pci, instance #0
            ebus, instance #0
                auxio (driver not attached)
                power, instance #0
                SUNW,pll (driver not attached)
                se, instance #0
                su, instance #0
                su, instance #1
                ecpp (driver not attached)
                fdthree, instance #0
                eeprom (driver not attached)
                flashprom (driver not attached)
```

```
                SUNW,CS4231 (driver not attached)
            network, instance #0
            SUNW,m64B (driver not attached)
            ide, instance #0
                disk (driver not attached)
                cdrom (driver not attached)
                dad, instance #0
                sd, instance #15
        pci, instance #1
            pci, instance #0
                pci108e,1000 (driver not attached)
                SUNW,hme, instance #1
                SUNW,isptwo, instance #0
                    sd (driver not attached)
                    st (driver not attached)
                    sd, instance #0 (driver not attached)
                    sd, instance #1 (driver not attached)
                    sd, instance #2 (driver not attached)
                    ...
    SUNW,UltraSPARC-IIi (driver not attached)
    SUNW,ffb, instance #0
    pseudo, instance #0
```

/devices 目录

/devices 分层结构提供了表示设备树的名称空间。下面是 /devices 名称空间的缩写列表。样例输出对应于先前显示的示例设备树和 prtconf(1M) 输出。

```
/devices
/devices/pseudo
/devices/pci@1f,0:devctl
/devices/SUNW,ffb@1e,0:ffb0
/devices/pci@1f,0
/devices/pci@1f,0/pci@1,1
/devices/pci@1f,0/pci@1,1/SUNW,m64B@2:m640
/devices/pci@1f,0/pci@1,1/ide@3:devctl
/devices/pci@1f,0/pci@1,1/ide@3:scsi
/devices/pci@1f,0/pci@1,1/ebus@1
/devices/pci@1f,0/pci@1,1/ebus@1/power@14,724000:power_button
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:a
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:b
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:0,hdlc
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:1,hdlc
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:a,cu
/devices/pci@1f,0/pci@1,1/ebus@1/se@14,400000:b,cu
/devices/pci@1f,0/pci@1,1/ebus@1/ecpp@14,3043bc:ecpp0
/devices/pci@1f,0/pci@1,1/ebus@1/fdthree@14,3023f0:a
/devices/pci@1f,0/pci@1,1/ebus@1/fdthree@14,3023f0:a,raw
/devices/pci@1f,0/pci@1,1/ebus@1/SUNW,CS4231@14,200000:sound,audio
/devices/pci@1f,0/pci@1,1/ebus@1/SUNW,CS4231@14,200000:sound,audioctl
/devices/pci@1f,0/pci@1,1/ide@3
/devices/pci@1f,0/pci@1,1/ide@3/sd@2,0:a
/devices/pci@1f,0/pci@1,1/ide@3/sd@2,0:a,raw
/devices/pci@1f,0/pci@1,1/ide@3/dad@0,0:a
/devices/pci@1f,0/pci@1,1/ide@3/dad@0,0:a,raw
/devices/pci@1f,0/pci@1
/devices/pci@1f,0/pci@1/pci@2
```

```
/devices/pci@1f,0/pci@1/pci@2/SUNW,isptwo@4:devctl
/devices/pci@1f,0/pci@1/pci@2/SUNW,isptwo@4:scsi
```

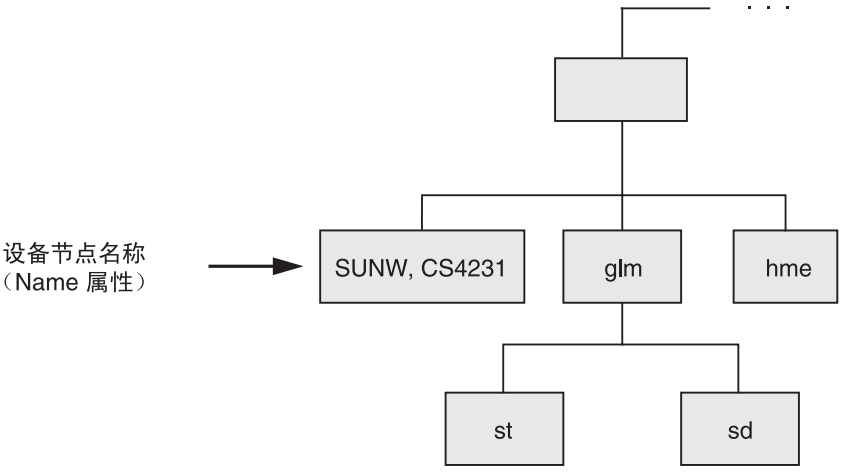
将驱动程序绑定到设备

除了构造设备树之外，内核还可确定用于管理设备的驱动程序。

将驱动程序绑定到设备指的是系统选择用于管理特定设备的驱动程序的过程。绑定名称是将驱动程序与设备信息树连接在一起的唯一设备结点名称。对于设备树中的每个设备，系统都会尝试从已安装的驱动程序列表中选择一个驱动程序。

每个设备节点都有关联的 *name* 属性。可以在系统引导期间通过外部代理（如 PROM）或通过 `driver.conf` 配置文件指定此属性。无论在哪种情况下，`name` 属性都表示指定给设备树中的设备的**节点名称**。**节点名称**是在 `/devices` 中可见并列在 `prtconf(1M)` 输出中的名称。

图 2–3 设备节点名称



设备节点也可以有关联的 *compatible* 属性。*compatible* 属性包含设备的一个或多个可能的驱动程序名称或驱动程序别名的有序列表。

系统使用 *compatible* 属性和 *name* 属性来为设备选择驱动程序。如果 *compatible* 属性存在，则系统会首先尝试将 *compatible* 属性的内容与系统中的驱动程序匹配。系统将从 *compatible* 属性列表的第一个驱动程序名称开始，尝试将该驱动程序名称与系统中的已知驱动程序匹配。系统将会处理该列表中的每一项，直到找到匹配项或者到达列表结尾。

如果 *name* 属性或 *compatible* 属性的内容与系统中的某个驱动程序匹配，则将该驱动程序绑定到设备节点。如果未找到匹配项，则不会将任何驱动程序绑定到设备节点。

通用设备名称

某些设备将**通用**设备名称指定为 *name* 属性的值。通用设备名称用于描述设备的功能，不实际标识设备的特定驱动程序。例如，SCSI 主机总线适配器可能具有通用设备名称 `scsi`。以太网设备可能具有通用设备名称 `ethernet`。

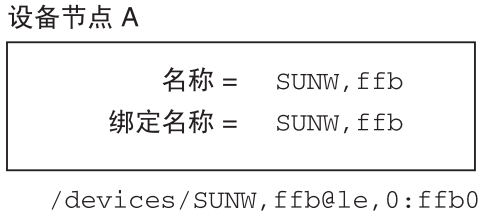
通过 *compatible* 属性，系统可以确定具有通用设备名称的设备的备用驱动程序名称，例如，`glm` 对应于 `scsi` HBA 设备驱动程序，`hme` 对应于 `ethernet` 设备驱动程序。

具有通用设备名称的设备需要提供 *compatible* 属性。

注 – 有关**通用设备名称**的完整说明，请参见 IEEE 1275 Open Firmware Boot Standard。

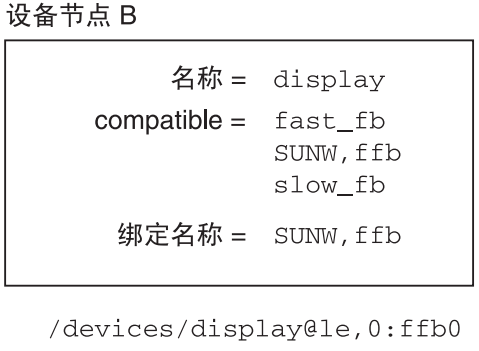
下图显示了具有特定设备名称的设备节点。驱动程序绑定名称 `SUNW,ffb` 与设备节点名称同名。

图 2–4 特定驱动程序节点绑定



下图显示了具有通用设备名称 `display` 的设备节点。驱动程序绑定名称 `SUNW,ffb` 是 *compatible* 属性驱动程序列表中与系统驱动程序列表中的驱动程序匹配的第一个名称。在这种情况下，`display` 是帧缓存器的通用设备名称。

图 2–5 通用驱动程序节点绑定



◆ ◆ ◆ 第 3 章

多线程

本章介绍 Oracle Solaris 多线程内核的锁定原语和线程同步机制。设计设备驱动程序时应当充分利用多线程的特性。本章介绍有关以下主题的信息：

锁定原语

在传统 UNIX 系统中，可以通过显式调用 `sleep(1)` 以放弃处理器或通过硬件中断来终止内核代码的每个部分。Oracle Solaris OS 的运行方式与此不同。系统可随时抢占内核线程以运行其他线程。由于所有内核线程共享内核地址空间，并且通常需要读取和修改相同的数据，因此内核提供了大量的锁定原语以防止线程损坏共享数据。这些机制包括**互斥锁**、读取器/写入器锁以及信号量。

驱动程序数据的存储类

数据的存储类用于指示驱动程序是否需要采取显式步骤控制对数据的访问。共有三个数据存储类：

- **自动（栈）数据**。每个线程有一个专用栈，因此驱动程序永远无需锁定自动变量。
- **全局静态数据**。全局静态数据可由驱动程序中任意数量的线程共享。驱动程序有时可能需要锁定此类型的数据。
- **内核堆数据**。驱动程序中任意数量的线程均可共享内核堆数据，如 `kmem_alloc(9F)` 分配的数据。驱动程序需要随时保护共享数据。

互斥锁

互斥锁 (*mutex*) 通常与一组数据关联，并控制对这些数据的访问。互斥锁提供了一种一次仅允许一个线程访问这些数据的方法。互斥锁函数包括：

mutex_destroy(9F)	释放任何关联的存储空间。
mutex_enter(9F)	获取互斥锁。
mutex_exit(9F)	释放互斥锁。
mutex_init(9F)	初始化互斥锁。
mutex_owned(9F)	测试以确定当前线程是否持有互斥锁。仅限于在 ASSERT(9F) 中使用。
mutex_tryenter(9F)	获取互斥锁（如果可用），但不阻塞。

设置互斥锁

设备驱动程序通常会为每个驱动程序数据结构分配一个互斥锁。互斥锁通常是该结构中类型为 kmutex_t 的字段。调用 mutex_init(9F) 以初始化要使用的互斥锁。通常在执行 attach(9E) 时为每个设备互斥锁进行此调用，在执行 _init(9E) 时为全局驱动程序互斥锁进行此调用。

例如，

```
struct xxstate *xsp;
/* ... */
mutex_init(&xsp->mu, NULL, MUTEX_DRIVER, NULL);
/* ... */
```

有关互斥锁初始化的较完整示例，请参见第 6 章，驱动程序自动配置。

驱动程序在卸载之前必须使用 mutex_destroy(9F) 销毁互斥锁。通常在执行 detach(9E) 时为每个设备互斥锁进行销毁操作，在执行 _fini(9E) 时为全局驱动程序互斥锁进行销毁操作。

使用互斥锁

驱动程序如果需要读写共享数据结构，必须执行以下操作：

- 获取互斥锁
- 访问数据
- 释放互斥锁

互斥锁的作用域（即互斥锁保护的数据）完全由程序员决定。仅当访问数据结构的每个代码路径都保护数据结构并且同时持有互斥锁时，互斥锁才会保护该数据结构。

读取器/写入器锁

读取器/写入器锁可控制对数据集的访问。读取器/写入器锁之所以这样命名，是由于许多线程可同时持有读锁，但仅有一个线程可持有写锁。

大多数设备驱动程序不使用读取器/写入器锁。这些锁的速度比互斥锁要慢。这些锁仅当保护通常进行读取但不经常写入的数据时才会提高性能。在此情况下，对互斥锁的争用可能会成为一个瓶颈，因此使用读取器/写入器锁效率可能更高。下表概述了读取器/写入器函数。有关详细信息，请参见 rwlock(9F) 手册页。读取器/写入器锁函数包括：

rw_destroy(9F)	销毁读取器/写入器锁
rw_downgrade(9F)	将读取器/写入器锁的持有者从写入器降级为读取器
rw_enter(9F)	获取读取器/写入器锁
rw_exit(9F)	释放读取器/写入器锁
rw_init(9F)	初始化读取器/写入器锁
rw_read_locked(9F)	确定是否持有用于读/写操作的读取器/写入器锁
rw_tryenter(9F)	尝试在无需等待的情况下获取读取器/写入器锁
rw_tryupgrade(9F)	尝试将读取器/写入器锁的持有者从读取器升级为写入器

信号

计数信号量可用作管理设备驱动程序中线程的替代原语。有关更多信息，请参见 semaphore(9F) 手册页。信号函数包括：

sema_destroy(9F)	销毁信号。
sema_init(9F)	初始化信号。
sema_p(9F)	减小信号，可能会阻塞。
sema_p_sig(9F)	减小信号，但不会阻塞（如果信号处于待处理状态）。请参见第 72 页中的 “线程无法接收信号”。
sema_tryp(9F)	尝试减小信号，但不阻塞。
sema_v(9F)	增加信号，可能会解除阻塞等待者。

线程同步

除了保护共享数据外，驱动程序通常需要在多个线程之间同步执行。

线程同步中的条件变量

条件变量是线程同步的标准形式。这些变量专门用于互斥锁。关联互斥锁可以确保条件的检查是原子操作，并且线程可以基于关联的条件变量阻塞，同时不会忽略对条件的更改或条件已更改的信号。

condvar(9F) 函数包括：

cv_broadcast(9F)	向基于条件变量等待的所有线程发出信号。
cv_destroy(9F)	销毁条件变量。
cv_init(9F)	初始化条件变量。
cv_signal(9F)	向基于条件变量等待的一个线程发出信号。
cv_timedwait(9F)	等待条件、超时或信号。请参见第 72 页中的 “线程无法接收信号”。
cv_timedwait_sig(9F)	等待条件或超时。
cv_wait(9F)	等待条件。
cv_wait_sig(9F)	等待条件或在收到信号时返回零。请参见第 72 页中的 “线程无法接收信号”。

初始化条件变量

针对每个条件声明一个 kcondvar_t 类型的条件变量。通常，条件变量是驱动程序定义的数据结构中的一个变量。使用 cv_init(9F) 可初始化每个条件变量。与互斥锁类似，条件变量通常在执行 attach(9E) 时初始化。以下是一个初始化条件变量的典型示例：

```
cv_init(&xsp->cv, NULL, CV_DRIVER, NULL);
```

有关条件变量初始化的较完整示例，请参见第 6 章，驱动程序自动配置。

等待条件

要使用条件变量，请在等待条件的代码路径中执行以下步骤：

1. 获取用于保护条件的互斥锁。
2. 测试条件。

3. 如果测试结果表明不允许线程继续执行，请使用 cv_wait(9F) 根据条件阻塞当前线程。cv_wait(9F) 函数将在阻塞线程之前释放互斥锁，并在返回之前重新获取互斥锁。从 cv_wait(9F) 返回时，重复该测试。
4. 测试表明允许线程继续执行后，请将条件设置为其新值。例如，将设备标志设置为繁忙。
5. 释放互斥锁。

发出条件信号

请在代码路径中执行以下步骤以发出条件信号：

1. 获取用于保护条件的互斥锁。
2. 设置条件。
3. 使用 cv_broadcast(9F) 向阻塞的线程发出信号。
4. 释放互斥锁。

以下示例使用繁忙标志以及互斥锁和条件变量来强制 read(9E) 例程进行等待，直到设备不再繁忙时为止，然后开始传送。

示例 3–1 使用互斥锁和条件变量

```
static int
xxread(dev_t dev, struct uio *uiop, cred_t *credp)
{
        struct xxstate *xsp;
        /* ... */
        mutex_enter(&xsp->mu);
        while (xsp->busy)
                cv_wait(&xsp->cv, &xsp->mu);
        xsp->busy = 1;
        mutex_exit(&xsp->mu);
        /* perform the data access */
}

static uint_t
xxintr(caddr_t arg)
{
        struct xxstate *xsp = (struct xxstate *)arg;
        mutex_enter(&xsp->mu);
        xsp->busy = 0;
        cv_broadcast(&xsp->cv);
        mutex_exit(&xsp->mu);
}
```

cv_wait() 和 cv_timedwait() 函数

如果使用 cv_wait(9F) 根据某个条件将线程阻塞，但该条件不发生，则该线程将永远等待。要避免这种情况，请使用 cv_timedwait(9F)，它取决于执行唤醒的其他线

程。cv_timedwait() 采取绝对等待时间作为参数。如果时间已到但未发生事件，则 cv_timedwait() 将返回 -1。如果满足条件，则 cv_timedwait() 将返回一个正值。

cv_timedwait(9F) 要求自上次重新引导系统以来的绝对等待时间（以时钟周期表示）。通过使用 ddi_get_lbolt(9F) 检索当前值可确定该等待时间。驱动程序通常具有的是最大等待秒数或微秒数，因此需要使用 drv_usectohz(9F) 将该值转换为时钟周期，然后与 ddi_get_lbolt(9F) 的值相加。

以下示例说明如何使用 cv_timedwait(9F) 最多等待五秒钟便访问设备，然后向调用方返回 EIO。

示例 3–2 使用 cv_timedwait()

```
clock_t            cur_ticks, to;
mutex_enter(&xsp->mu);
while (xsp->busy) {
        cur_ticks = ddi_get_lbolt();
        to = cur_ticks + drv_usectohz(5000000); /* 5 seconds from now */
        if (cv_timedwait(&xsp->cv, &xsp->mu, to) == -1) {
                /*
                 * The timeout time 'to' was reached without the
                 * condition being signaled.
                 */
                /* tidy up and exit */
                mutex_exit(&xsp->mu);
                return (EIO);
        }
}
xsp->busy = 1;
mutex_exit(&xsp->mu);
```

虽然设备驱动程序写入器通常首选使用 cv_timedwait(9F) 而不是 cv_wait(9F)，但是有时选用 cv_wait(9F) 会更好。例如，如果驱动程序基于以下条件等待，则使用 cv_wait(9F) 更合适：

- 内部驱动程序状态发生变化，在此情况下状态变化可能要求执行一些命令或设置要经过的时间
- 驱动程序的某些部分必须单线程执行
- 已在管理可能超时的情况，如 "A" 取决于 "B"，同时 "B" 使用 cv_timedwait(9F)

cv_wait_sig() 函数

驱动程序可能正在等待不会产生或长时间不会发生的条件。在此类情况下，用户可发送信号中止该线程。根据驱动程序设计，信号可能无法将驱动程序唤醒。

cv_wait_sig(9F) 允许使用信号来解除阻塞线程。借助此功能，用户可以通过使用 kill(1) 向线程发送信号或键入中断字符，从而免于可能的长时间等待。如果

cv_wait_sig(9F) 由于收到信号而返回，则会返回零；如果条件发生，则返回非零值。但是，对于可能未收到信号的情况，请参见第 72 页中的 “线程无法接收信号”。

以下示例说明如何使用 cv_wait_sig(9F) 以允许使用信号解除阻塞线程。

示例 3–3 使用 cv_wait_sig()

```
mutex_enter(&xsp->mu);
while (xsp->busy) {
        if (cv_wait_sig(&xsp->cv, &xsp->mu) == 0) {
        /* Signaled while waiting for the condition */
                /* tidy up and exit */
                mutex_exit(&xsp->mu);
                return (EINTR);
        }
}
xsp->busy = 1;
mutex_exit(&xsp->mu);
```

cv_timedwait_sig() 函数

cv_timedwait_sig(9F) 与 cv_timedwait(9F) 和 cv_wait_sig(9F) 相似，不同之处在于，达到超时后，如果没有发出条件信号，则 cv_timedwait_sig() 将返回 -1，如果向线程发送了信号（例如 kill(2)），则将返回 0。

对于 cv_timedwait(9F) 和 cv_timedwait_sig(9F)，系统将使用上次重新引导系统以来的绝对时钟周期度量时间。

选择锁定方案

在设计大多数设备驱动程序时，都应该确保锁定方案简单易懂。使用额外的锁允许更多并发，但会增加开销。使用的锁越少，占用的时间越短，但允许的并发会更少。通常，对每个数据结构使用一个互斥锁，对驱动程序必须等待的每个事件或条件使用一个条件变量，对驱动程序的每个主要全局数据集使用一个互斥锁。请避免长时间持有互斥锁。选择锁定方案时，请遵循以下指导原则：

- 优先使用程序入口点的多线程语义。
- 确保所有程序入口点可重入。可以通过将静态变量更改为自动变量来减少共享数据量。
- 如果驱动程序获取了多个互斥锁，请在所有代码路径中按相同顺序获取和释放这些互斥锁。
- 在相同的功能空间中持有锁和释放锁。
- 调用可阻塞的 DDI 接口时请避免持有驱动程序互斥锁，例如使用 KM_SLEEP 调用 kmem_alloc(9F)。

要查看锁的用法，请使用 lockstat(1M)。lockstat(1M) 可监视所有内核锁定事件、收集有关事件的频率和计时数据，并显示这些数据。

有关多线程操作的更多详细信息，请参见《Multithreaded Programming Guide》。

潜在的锁定缺点

同一线程不可重复获取互斥锁。如果已持有互斥锁，则再次尝试声明此互斥锁会导致产生以下故障消息：

```
panic: recursive mutex_enter. mutex %x caller %x
```

释放当前线程未持有的互斥锁会产生以下故障消息：

```
panic: mutex_adaptive_exit: mutex not held by thread
```

以下故障消息仅在单处理器上出现：

```
panic: lock_set: lock held and only one CPU
```

lock_set 故障消息指明线程持有自旋互斥锁 (spin mutex)，并且该锁将会永久旋转，因为没有其他 CPU 可以释放此互斥锁。如果驱动程序忘记释放某个代码路径上的互斥锁，或在持有互斥锁时阻塞，则会发生此情况。

具有高级中断的设备所调用的例程（如 cv_wait(9F)）阻塞时通常会导致出现 lock_set 故障消息。另一个常见原因是高级处理程序通过调用 mutex_enter(9F) 获取自适应互斥锁。

线程无法接收信号

线程收到信号时，可以唤醒 sema_p_sig()、cv_wait_sig() 和 cv_timedwait_sig() 函数。由于某些线程无法接收信号，因此可能会出现问题。例如，如果由于应用程序调用 close(2) 而导致调用 close(9E)，则可以收到信号。但是，如果是从 exit(2) 处理（关闭所有打开的文件描述符）中调用 close(9E)，则线程无法收到信号。如果线程无法收到信号，则 sema_p_sig() 的行为与 sema_p() 相同，cv_wait_sig() 的行为与 cv_wait() 相同，cv_timedwait_sig() 的行为与 cv_timedwait() 相同。

对于可能永远不会发生的事件，请注意避免永久休眠。永远不会发生的事件会创建不可中止 (defunct) 的线程并使设备不可用，除非重新引导系统。失效进程无法接收信号。

要检测当前线程是否可接收信号，请使用 ddi_can_receive_sig(9F) 函数。如果 ddi_can_receive_sig() 函数返回 B_TRUE，则以上函数可在收到信号时唤醒。如果

ddi_can_receive_sig() 函数返回 B_FALSE，则以上函数无法在收到信号时唤醒。如果 ddi_can_receive_sig() 函数返回 B_FALSE，则驱动程序会使用替代方法（如 timeout(9F) 函数）重新唤醒。

出现此问题的一个重要情况是使用串行端口。如果远程系统声明了流量控制，并且 close(9E) 函数在尝试清空输出数据时阻塞，则端口会堵塞，直到解决流量控制情况或重新引导系统为止。此类驱动程序应检测到此情况并设置计时器，以便在流量控制情况持续过长时间时中止清空操作。

此问题还会影响 qwait_sig(9F) 函数。此函数将在《STREAMS Programming Guide》中的第 7 章 “STREAMS Framework – Kernel Level”中介绍。

◆ ◆ ◆ 第 4 章

4 属性

属性是用户定义的名称-值对结构，该结构使用 DDI/DKI 接口进行管理。本章介绍有关以下主题的信息：

设备属性

设备特性 (attribute) 信息可由称为**属性** (property) 的**名称-值**对表示法表示。

例如，设备寄存器和板载内存可由 reg 属性表示。reg 属性是描述设备硬件寄存器的软件抽象术语。reg 属性的值对设备寄存器地址位置和大小进行编码。驱动程序使用 reg 属性访问设备寄存器。

另外一个示例是 interrupt 属性。interrupt 属性表示设备中断。interrupt 属性的值对设备中断 PIN 进行编码。

可以为属性指定五种类型的值：

- **字节数组** – 任意长度的一系列字节
- **整数属性** – 整数值
- **整数数组属性** – 整数数组
- **字符串属性** – 以 null 结尾的字符串
- **字符串数组属性** – 以 null 结尾的字符串列表

没有值的属性被视为布尔属性。对于布尔属性，如果存在，则值为 True；如果不存在，则值为 False。

设备属性名称

严格地说，DDI/DKI 软件属性名称没有限制。但建议使用某些限制。IEEE 1275-1994 Standard for Boot Firmware 引导固件标准按如下方法定义属性：

> 属性是由 1 到 31 个可列显字符组成的人工可读文本字符串。属性名称不能包含大写字符或字符 "/"、"\"、":"、 "["、"]" 和 "@"。以字符 "+" 开头的属性名称保留供 IEEE 1275-1994 的将来修订使用。

根据约定，属性名称中不能使用下划线。可使用连字符 (-)。根据约定，以问号字符 (?) 结尾的属性名称包含字符串值（通常为 TRUE 或 FALSE），例如 `auto-boot?`。

有关在驱动程序配置文件中添加属性的讨论，请参见 `driver.conf(4)` 手册页。`pm(9P)` 和 `pm-components(9P)` 手册页说明了如何在电源管理中使用属性。有关应该如何在设备驱动程序中记录属性的信息，请阅读 `sd(7D)` 手册页。

创建和更新属性

要为驱动程序创建属性，或者更新现有属性，请将 DDI 驱动程序更新接口（如 `ddi_prop_update_int(9F)` 或 `ddi_prop_update_string(9F)`）中的一个接口与相应的属性类型一起使用。有关可用属性接口的列表，请参见表 4–1。这些接口通常从驱动程序的 `attach(9E)` 入口点调用。在以下示例中，`ddi_prop_update_string()` 创建一个名为 `pm-hardware-state` 且值为 `needs-suspend-resume` 的字符串属性。

```
/* The following code is to tell cpr that this device
 * needs to be suspended and resumed.
 */
(void) ddi_prop_update_string(device, dip,
    "pm-hardware-state", "needs-suspend-resume");
```

在大多数情况下，使用 `ddi_prop_update()` 例程即可满足更新属性的要求。但是，有时更新经常更改的属性值的系统开销可能会导致性能问题。有关使用属性值的本地实例以避免使用 `ddi_prop_update()` 的说明，请参见 第 79 页中的 “prop_op() 入口点”。

查找属性

驱动程序可以请求其父级的属性，而后者又可以请求其父级。驱动程序可以控制是否将请求传递到其父级以上。

例如，以下示例中的 `esp` 驱动程序为每个目标维护一个名为 `targetx-sync-speed` 的整数属性。`targetx-sync-speed` 中的 `x` 表示目标编号。`prtconf(1M)` 命令以详细模式显示驱动程序属性。以下示例列出了 `esp` 驱动程序的部分内容。

```
% prtconf -v
...
       esp, instance #0
```

```
                Driver software properties:
                    name <target2-sync-speed> length <4>
                        value <0x00000fa0>.
...
```

下表汇总了属性接口。

表 4–1 属性接口用法

系列	属性接口	说明
ddi_prop_lookup	ddi_prop_exists(9F)	查找属性，如果该属性存在，则成功返回。如果该属性不存在，则失败。
	ddi_prop_get_int(9F)	查找并返回整数属性
	ddi_prop_get_int64(9F)	查找并返回 64 位整数属性
	ddi_prop_lookup_int_array(9F)	查找并返回整数数组属性
	ddi_prop_lookup_int64_array(9F)	查找并返回 64 位整数数组属性
	ddi_prop_lookup_string(9F)	查找并返回字符串属性
	ddi_prop_lookup_string_array(9F)	查找并返回字符串数组属性
	ddi_prop_lookup_byte_array(9F)	查找并返回字节数组属性
ddi_prop_update	ddi_prop_update_int(9F)	更新或创建整数属性
	ddi_prop_update_int64(9F)	更新或创建单个 64 位整数属性
	ddi_prop_update_int_array(9F)	更新或创建整数数组属性
	ddi_prop_update_string(9F)	更新或创建字符串属性
	ddi_prop_update_string_array(9F)	更新或创建字符串数组属性
	ddi_prop_update_int64_array(9F)	更新或创建 64 位整数数组属性
	ddi_prop_update_byte_array(9F)	更新或创建字节数组属性
ddi_prop_remove	ddi_prop_remove(9F)	删除单个属性
	ddi_prop_remove_all(9F)	删除与设备关联的所有属性

尽可能使用 64 位版本的 `int` 属性接口（如 `ddi_prop_update_int64(9F)`），而不要使用 32 位版本（如 `ddi_prop_update_int(9F)`）。

对 driver.conf 文件的更改

当运行 Oracle Solaris OS 的系统升级时，可能会安装新版本的驱动程序。在升级过程中，系统中的 `driver.conf` 文件也将更新。`driver.conf` 文件由供应商和系统管理员进

行定制。在系统升级期间，系统先前的配置将继续与新驱动程序、供应商的 driver.conf 文件和管理员的 driver.conf 文件一起使用。

在 Oracle Solaris 11 发行版中，驱动程序编写者可以选择提供单独的 driver.conf 文件来包含供应商提供的驱动程序数据。新的 driver.conf 文件存储在 /etc/driver/drv 目录中。这使得系统能够保留对该文件所做的任何管理更改。如果在两个配置文件中都发现了某个驱动程序，系统将合并文件并提供包含组合后的属性的文件。供应商的 driver.conf 文件的格式与管理员的驱动程序配置文件相同。

现在可以通过新接口显式使供应商和管理配置数据可供驱动程序使用。这使得驱动程序编写者能够在驱动程序中直接编写合并逻辑代码，而不是在类操作脚本或安装前脚本以及安装后脚本中进行编写。对管理文件的定制将持久保留并且驱动程序可以决定新值与旧值的相关性。

为了让驱动程序确保以上模型正常工作，驱动程序开发者必须考虑以下事项：

- 使所设计的驱动程序有一组有规程的、可配置的选项可用。
- 在驱动程序文档和手册页中全面描述驱动程序的选项和模型。
- 如果驱动程序更改了其配置选项以致管理员设置失效或被取代，驱动程序应该确保接受先前的管理设置。要查找先前的配置，驱动程序可以在属性类型设置为 DDI_PROP_VENDOR 或 DDI_PROP_ADMIN 的情况下使用 ddi_prop_lookup(9F) 接口。

  例如，如果驱动程序支持以秒为单位的超时配置，并且新版本的驱动程序当前支持以毫秒为单位的更精细的超时粒度。对新属性进行命名时应当使其能够与先前的属性相区分。然后，驱动程序应在管理列表中查找先前的属性，如果存在，驱动器将继续接受它。以下驱动程序代码阐释了超时示例。

示例 4–1 对本地配置的超时值的驱动程序检查

```
     * Has the timeout been locally configured using the
+     * prior option of timeout in units of seconds?
+     */
+    if (ddi_prop_lookup_int(DDI_DEV_T_ANY, dip,
+        DDI_PROP_ADMIN, "timeout",&ivalues,&n) ==
+        DDI_PROP_SUCCESS) {
+        if (n != 1) {
+            ddi_prop_free(ivalues);
+            return (EINVAL);
+        }
+        /* yes - convert our working timeout accordingly */
+        dip->ms_timeout = 1000 * ivalues[0];
+        /* record the new parameter setting for confirmation */
+        (void) ddi_prop_update_int(DDI_DEV_T_NONE,
+            dip, "ms-timeout", dip->ms_timeout);
+        ddi_prop_free(ivalues);
+    }
```

prtconf(1M) 命令显示驱动程序属性，可以使用新的 -u 选项来显示原来的属性值和更改后的属性值。

prop_op() 入口点

向系统报告设备属性或驱动程序属性通常需要使用 prop_op(9E) 入口点。如果驱动程序无需创建或管理其自己的属性，则 ddi_prop_op(9F) 函数可用于此入口点。

如果在驱动程序的 cb_ops(9S) 结构中定义了 ddi_prop_op()，ddi_prop_op(9F) 可用作设备驱动程序的 prop_op(9E) 入口点。ddi_prop_op() 使叶设备可在设备的属性列表中搜索并获取属性值。

如果驱动程序需要维护其值经常更改的属性，则应在 cb_ops() 结构中定义特定于驱动程序的 prop_op 例程，而不是调用 ddi_prop_op()。此方法可避免由于重复使用 ddi_prop_update() 而造成的效率低下。然后，驱动程序应在其软状态结构或驱动程序变量中维护属性值的副本。

prop_op(9E) 入口点向系统报告特定驱动程序属性的值和设备属性的值。在许多情况下，ddi_prop_op(9F) 例程在 cb_ops(9S) 结构中可用作驱动程序的 prop_op() 入口点。ddi_prop_op() 会执行所有必需的处理过程。对于处理设备属性请求时不需要进行特殊处理的驱动程序，ddi_prop_op() 即可满足要求。

但是，有时驱动程序必须提供 prop_op() 入口点。例如，如果驱动程序维护其值经常更改的属性，则针对每次更改使用 ddi_prop_update(9F) 更新属性便不能满足要求。相反，驱动程序应在实例的软状态下维护属性的阴影副本。然后，驱动程序可在值发生变化时更新阴影副本，而无需使用任何 ddi_prop_update() 例程。prop_op() 入口点必须拦截对此属性的请求，并使用 ddi_prop_update() 例程之一更新属性的值，然后将请求传递到 ddi_prop_op() 以处理属性请求。

在以下示例中，prop_op() 拦截 temperature 属性的请求。属性发生变化时，驱动程序将更新状态结构中的变量。但是，仅当发出请求时才会更新该属性。然后，驱动程序使用 ddi_prop_op() 处理该属性请求。如果属性请求不特定于某个设备，则驱动程序不会拦截该请求。dev 参数的值等于 DDI_DEV_T_ANY（通配符设备编号）时即是这种情况。

示例 4–2 prop_op() 例程

```
static int
xx_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp)
{
        minor_t instance;
        struct xxstate *xsp;
        if (dev != DDI_DEV_T_ANY) {
                return (ddi_prop_op(dev, dip, prop_op, flags, name,
                    valuep, lengthp));
        }

        instance = getminor(dev);
        xsp = ddi_get_soft_state(statep, instance);
        if (xsp == NULL)
                return (DDI_PROP_NOTFOUND);
        if (strcmp(name, "temperature") == 0) {
```

示例 4–2 prop_op() 例程 （续）

```
                ddi_prop_update_int(dev, dip, name, temperature);
        }

        /* other cases */
}
```

◆ ◆ ◆ 第 5 章

管理事件和排队任务

驱动程序使用事件来响应状态更改。本章提供以下有关事件的信息：

驱动程序使用任务队列来管理任务之间的资源相关性。本章提供有关任务队列的以下信息：

管理事件

系统经常需要对用户操作或系统请求之类的条件更改做出响应。例如，设备可能会在某个组件开始过热时发出警告，或者可能在将 DVD 插入驱动器后启动影片播放机。设备驱动程序可以使用称为**事件**的特殊消息来通知系统发生了状态更改。

事件介绍

事件是指设备驱动程序向相关实体发送的消息，用以指示发生了状态更改。在 Oracle Solaris OS 中，事件是以用户定义的名称-值对结构的形式实现的，这些结构是使用 nvlist* 函数管理的。（请参见 nvlist_alloc(9F) 手册页。）事件由供应商、类以及子类组成。例如，可以定义一个类用于监视环境条件。环境类可以具有子类，用来指示温度、风扇状态以及电源方面的变化。

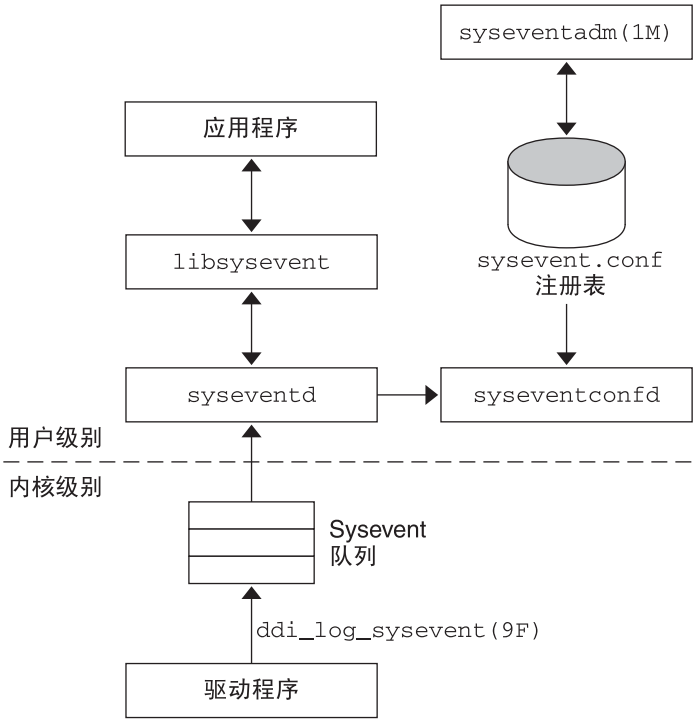
发生状态更改时，设备将通知驱动程序。驱动程序随后将使用 ddi_log_sysevent(9F) 函数在称为 sysevent 的队列中记录此事件。sysevent 队列会将事件传递到用户级，以便通过 syseventd 守护进程或 syseventconfd 守护进程进行处理。这些守护进程会将通知发送到订阅了指定事件通知的所有应用程序。

用户级应用程序的设计者可以使用以下两种方法处理事件：

- 应用程序可以使用 libsysevent(3LIB) 中的例程向 syseventd 守护进程订阅发生指定事件时的通知。
- 开发者可以编写单独的用户级应用程序来响应事件。此类型应用程序需要通过 syseventadm(1M) 进行注册。当 syseventconfd 遇到指定的事件时，应用程序会根据实际情况运行并处理此事件。

下图对此流程进行了说明。

图 5–1 事件检测



使用 ddi_log_sysevent() 记录事件

设备驱动程序使用 ddi_log_sysevent(9F) 接口生成和记录系统事件。

ddi_log_sysevent() 语法

ddi_log_sysevent() 使用以下语法：

```
int ddi_log_sysevent(dev_info_t *dip, char *vendor, char *class,
    char *subclass, nvlist_t *attr-list, sysevent_id_t *eidp, int sleep-flag);
```

其中：

dip 指向相应驱动程序处理的 *dev_info* 节点的指针。

vendor 指向定义驱动程序供应商的字符串的指针。第三方驱动程序应使用其公司的股票代号或类似的持久标识符。Oracle 提供的驱动程序使用 `DDI_VENDOR_SUNW`。

class 指向定义事件类的字符串的指针。*class* 是特定于驱动程序的值。表示影响设备的一组环境条件的字符串可能即是一个类的示例。事件使用方必须能够理解该值。

subclass 表示 *class* 参数子集的特定于驱动程序的字符串。例如，在表示环境条件的类中，事件子类可能是指设备的温度。事件使用方必须能够理解该值。

attr-list 指向列出与事件关联的名称-值特性的 `nvlist_t` 结构的指针。名称-值特性是驱动程序定义的，可以是指设备的特定特性或条件。

例如，可同时读取 CD-ROM 和 DVD 的设备。此设备可能具有一个名称为 `disc_type` 并且值等于 `cd_rom` 或 `dvd` 的特性。

与 *class* 和 *subclass* 一样，事件使用方必须能够解释名称-值对。

有关名称-值对以及 `nvlist_t` 结构的更多信息，请参见第 84 页中的 “定义事件特性”以及 `nvlist_alloc(9F)` 手册页。

如果事件没有任何特性，则此参数应设置为 `NULL`。

eidp `sysevent_id_t` 结构的地址。`sysevent_id_t` 结构用于提供事件的唯一标识。`ddi_log_sysevent` (9F) 将向此结构返回系统提供的事件序列号和时间戳。有关 `sysevent_id_t` 结构的更多信息，请参见 `ddi_log_sysevent(9F)` 手册页。

sleep-flag 指示调用方如何处理不可用资源可能性的标志。如果 *sleep-flag* 设置为 `DDI_SLEEP`，则驱动程序会阻塞，直到资源可用为止。如果设置为 `DDI_NOSLEEP`，则分配不会休眠且不能保证成功。如果返回了 `DDI_ENOMEM`，则驱动程序以后需要重试该操作。

即使设置为 `DDI_SLEEP`，此界面也可能返回错误（如系统繁忙），`sysevent`d 守护进程不响应或不尝试在中断上下文中记录事件。

记录事件的样例代码

设备驱动程序可执行以下任务来记录事件：

- 使用 `nvlist_alloc(9F)` 为属性列表分配内存
- 向特性列表添加名称-值对

- 使用 ddi_log_sysevent(9F) 函数在 sysevent 队列中记录事件
- 不再需要特性列表时调用 nvlist_free(9F)

以下示例说明如何使用 ddi_log_sysevent()。

示例 5–1 调用 ddi_log_sysevent()

```
char *vendor_name = "DDI_VENDOR_JGJG"
char *my_class = "JGJG_event";
char *my_subclass = "JGJG_alert";
nvlist_t *nvl;
/* ... */
nvlist_alloc(&nvl, nvflag, kmflag);
/* ... */
(void) nvlist_add_byte_array(nvl, propname, (uchar_t *)propval, proplen + 1);
/* ... */
if (ddi_log_sysevent(dip, vendor_name, my_class,
    my_subclass, nvl, NULL, DDI_SLEEP)!= DDI_SUCCESS)
    cmn_err(CE_WARN, "error logging system event");
nvlist_free(nvl);
```

定义事件特性

事件特性定义为名称-值对列表。Oracle Solaris DDI 提供了用于在名称-值对中存储信息的例程和结构。名称-值对保留在 nvlist_t 结构中，此结构对于驱动程序是不透明的。名称-值对的值可以是布尔值、int、字节、字符串、nvlist 或这些数据类型的数组。int 可以定义为 16 位、32 位或 64 位，可以带符号，也可不带符号。

下面是创建名称-值对列表的步骤。

1. 使用 nvlist_alloc(9F) 创建 nvlist_t 结构。

   nvlist_alloc() 接口会采用以下三个参数：

   - *nvlp*－指向 nvlist_t 结构指针的指针
   - *nvflag*－指示名称-值对的名称唯一性的标志。如果此标志设置为 NV_UNIQUE_NAME_TYPE，则会从列表中删除与新对的名称和类型相匹配的任何现有对。如果标志设置为 NV_UNIQUE_NAME，则会删除任何同名的现有对，而不考虑对的类型。只要对的类型不同，通过指定 NV_UNIQUE_NAME_TYPE，列表即可包含两个或多个同名的对，但如果指定 NV_UNIQUE_NAME，则列表中只能有一个对名称实例。如果未设置标志，则不会执行任何唯一性检查，将由列表的使用方负责处理同名的对。
   - *kmflag*－指示内核内存分配策略的标志。如果此参数设置为 KM_SLEEP，则驱动程序会阻塞，直到请求的内存可进行分配为止。KM_SLEEP 分配可能会休眠，但是保证会成功。KM_NOSLEEP 分配保证不会休眠，但是可能会在当前无可用内存时返回 NULL。

2. 使用名称-值对填充 nvlist。例如，要添加字符串，请使用 nvlist_add_string(9F)。要添加 32 位整数数组，请使用 nvlist_add_int32_array(9F)。nvlist_add_boolean(9F) 手册页包含用于添加对的可用接口的完整列表。

要取消分配列表，请使用 nvlist_free(9F)。

以下代码样例说明如何创建名称-值对列表。

示例 5–2 创建和填充名称-值对列表

```
nvlist_t*
create_nvlist()
    {
    int err;
    char *str = "child";
    int32_t ints[] = {0, 1, 2};
    nvlist_t *nvl;

    err = nvlist_alloc(&nvl, NV_UNIQUE_NAME, 0);    /* allocate list */
    if (err)
        return (NULL);
    if ((nvlist_add_string(nvl, "name", str) != 0) ||
        (nvlist_add_int32_array(nvl, "prop", ints, 3) != 0)) {
        nvlist_free(nvl);
        return (NULL);
    }
    return (nvl);
}
```

驱动程序可通过相应类型的查找函数（如 nvlist_lookup_int32_array(9F)）来检索 nvlist 中的元素，此类查找函数将要搜索的名称-值对的名称作为参数。

注 – 仅当在调用 nvlist_alloc(9F) 时指定了 NV_UNIQUE_NAME 或 NV_UNIQUE_NAME_TYPE 的情况下，这些接口才会正常工作。否则，将返回 ENOTSUP，因为此列表不能包含多个同名的对。

可以将名称-值列表中的各对放在连续内存中。此方法有助于将列表传递给已订阅了通知的实体。第一步是使用 nvlist_size(9F) 获取列表所需的内存块的大小。第二步是使用 nvlist_pack(9F) 将列表压缩到缓冲区中。收到缓冲区内容的使用方可使用 nvlist_unpack(9F) 解压缩缓冲区。

用户级开发者和内核级开发者均可使用用于处理名称-值对的函数。可以在《man pages section 3: Library Interfaces and Headers》和《man pages section 9: DDI and DKI Kernel Functions》中找到这些函数的相同手册页。有关针对名称-值对执行操作的函数的列表，请参见下表。

表 5–1 使用名称-值对的函数

手册页	用途/函数
nvlist_add_boolean(9F)	向列表中添加名称-值对。函数包括： nvlist_add_boolean()、nvlist_add_boolean_value()、nvlist_add_byte()、nvlist_add_int8()、nvlist_add_uint8()、nvlist_add_int16()、nvlist_add_uint16()、nvlist_add_int32()、nvlist_add_uint32()、nvlist_add_int64()、nvlist_add_uint64()、nvlist_add_string()、nvlist_add_nvlist()、nvlist_add_nvpair()、nvlist_add_boolean_array()、nvlist_add_int8_array、nvlist_add_uint8_array()、nvlist_add_nvlist_array()、nvlist_add_byte_array()、nvlist_add_int16_array()、nvlist_add_uint16_array()、nvlist_add_int32_array()、nvlist_add_uint32_array()、nvlist_add_int64_array()、nvlist_add_uint64_array()、nvlist_add_string_array()
nvlist_alloc(9F)	处理名称-值列表缓冲区。函数包括： nvlist_alloc()、nvlist_free()、nvlist_size()、nvlist_pack()、nvlist_unpack()、nvlist_dup()、nvlist_merge()
nvlist_lookup_boolean(9F)	搜索名称-值对。函数包括： nvlist_lookup_boolean()、nvlist_lookup_boolean_value()、nvlist_lookup_byte()、nvlist_lookup_int8()、nvlist_lookup_int16()、nvlist_lookup_int32()、nvlist_lookup_int64()、nvlist_lookup_uint8()、nvlist_lookup_uint16()、nvlist_lookup_uint32()、nvlist_lookup_uint64()、nvlist_lookup_string()、nvlist_lookup_nvlist()、nvlist_lookup_boolean_array、nvlist_lookup_byte_array()、nvlist_lookup_int8_array()、nvlist_lookup_int16_array()、nvlist_lookup_int32_array()、nvlist_lookup_int64_array()、nvlist_lookup_uint8_array()、nvlist_lookup_uint16_array()、nvlist_lookup_uint32_array()、nvlist_lookup_uint64_array()、nvlist_lookup_string_array()、nvlist_lookup_nvlist_array()、nvlist_lookup_pairs()
nvlist_next_nvpair(9F)	获取名称-值对数据。函数包括： nvlist_next_nvpair()、nvpair_name()、 nvpair_type()
nvlist_remove(9F)	删除名称-值对。函数包括： nv_remove()、nv_remove_all()

排队任务

本节讨论如何使用**任务队列**来延迟处理某些任务并将这些任务的执行委托给另一个内核线程。

任务队列简介

内核编程中的一项常见操作是对某个任务进行调度，使它以后由另一线程执行。以下示例给出了可能需要以后由另一线程执行某个任务的一些原因：

- 当前的代码路径对时间有关键要求。要执行的其他任务对时间没有关键要求。
- 其他任务可能需要获取另一个线程当前持有的锁。
- 在当前上下文中无法进行阻塞。但其他任务可能需要阻塞，例如，它需要等待内存。
- 某种情况正在阻止代码路径完成，但是当前的代码路径不能休眠或失败。需要将当前任务排入队列，以便在该情况消失后执行。
- 需要以并行方式启动多个任务。

对于上面的每种情况，任务都在不同的**上下文**中执行。不同的上下文通常是持有一组不同锁的不同内核线程，并可能具有不同的优先级。任务队列提供一个通用内核 API 来调度异步任务。

任务队列是一个任务列表，一个或多个线程为该列表提供服务。如果任务队列只有一个服务线程，则所有任务肯定会按照它们在列表中添加的先后顺序执行。如果任务队列有多个服务线程，则任务的执行顺序是未知的。

注 – 如果任务队列有多个服务线程，请确保某个任务的执行不依赖于其他任何任务的执行。任务之间的相关性会导致产生死锁。

任务队列接口

以下 DDI 接口管理任务队列。这些接口在 `sys/sunddi.h` 头文件中定义。有关这些接口的更多信息，请参见 `taskq(9F)` 手册页。

`ddi_taskq_t`	不透明句柄
`TASKQ_DEFAULTPRI`	系统缺省优先级
`DDI_SLEEP`	可以阻塞以获得内存
`DDI_NOSLEEP`	不能阻塞以获得内存

`ddi_taskq_create()`	创建任务队列
`ddi_taskq_destroy()`	销毁任务队列
`ddi_taskq_dispatch()`	在任务队列中添加任务
`ddi_taskq_wait()`	等待暂挂的任务完成
`ddi_taskq_suspend()`	暂挂任务队列
`ddi_taskq_suspended()`	检查任务队列是否已暂挂
`ddi_taskq_resume()`	恢复暂挂的任务队列

观察任务队列

在驱动程序中的典型应用是在调用 attach(9E) 时创建任务队列。大多数 `taskq_dispatch()` 调用都来自中断上下文。

本节介绍两种可用来监视任务队列所使用的系统资源的方法。任务队列会导出任务队列线程使用系统时间的相关统计信息。任务队列还会使用 DTrace SDT 探测器来确定任务队列何时开始执行某个任务，以及何时完成执行。

任务队列内核统计信息计数器

每个任务队列都有一组关联的 `kstat` 计数器。检查以下 kstat(1M) 命令的输出：

```
$ kstat -c taskq
module: unix                            instance: 0
name:   ata_nexus_enum_tq               class:    taskq
        crtime                          53.877907833
        executed                        0
        maxtasks                        0
        nactive                         1
        nalloc                          0
        priority                        60
        snaptime                        258059.249256749
        tasks                           0
        threads                         1
        totaltime                       0

module: unix                            instance: 0
name:   callout_taskq                   class:    taskq
        crtime                          0
        executed                        13956358
        maxtasks                        4
        nactive                         4
        nalloc                          0
        priority                        99
        snaptime                        258059.24981709
        tasks                           13956358
        threads                         2
        totaltime                       120247890619
```

以上所示的 kstat 输出包含以下信息：

- 任务队列的名称及其实例编号
- 已调度任务的数目 (tasks) 以及已执行任务的数目 (executed)
- 处理任务队列的内核线程数 (threads) 及其优先级 (priority)
- 处理所有任务花费的总时间（以纳秒为单位）(totaltime)

以下示例说明如何使用 kstat 命令来观察计数器（已调度任务的数目）是如何随时间而递增的：

```
$ kstat -p unix:0:callout_taskq:tasks 1 5
unix:0:callout_taskq:tasks      13994642

unix:0:callout_taskq:tasks      13994711

unix:0:callout_taskq:tasks      13994784

unix:0:callout_taskq:tasks      13994855

unix:0:callout_taskq:tasks      13994926
```

任务队列 DTrace SDT 探测器

任务队列提供了若干个有用的 SDT 探测器。本节介绍的所有探测器都具有以下两个参数：

- ddi_taskq_create() 返回的任务队列指针
- 指向 taskq_ent_t 结构的指针。在 D 脚本中使用该指针可以提取函数和参数。

可以使用这些探测器来收集有关各个任务队列以及通过这些队列执行的各个任务的精确计时信息。例如，以下脚本每隔 10 秒列显通过任务队列调度的函数：

```
# !/usr/sbin/dtrace -qs

sdt:genunix::taskq-enqueue
{
  this->tq  = (taskq_t *)arg0;
  this->tqe = (taskq_ent_t *) arg1;
  @[this->tq->tq_name,
    this->tq->tq_instance,
    this->tqe->tqent_func] = count();
}

tick-10s
{
  printa ("%s(%d): %a called %@d times\n", @);
  trunc(@);
}
```

在特定的计算机上，以上 D 脚本生成以下输出：

```
callout_taskq(1): genunix`callout_execute called 51 times
callout_taskq(0): genunix`callout_execute called 701 times
kmem_taskq(0): genunix`kmem_update_timeout called 1 times
```

```
kmem_taskq(0): genunix‘kmem_hash_rescale called 4 times
callout_taskq(1): genunix‘callout_execute called 40 times
USB_hid_81_pipehndl_tq_1(14): usba‘hcdi_cb_thread called 256 times
callout_taskq(0): genunix‘callout_execute called 702 times
kmem_taskq(0): genunix‘kmem_update_timeout called 1 times
kmem_taskq(0): genunix‘kmem_hash_rescale called 4 times
callout_taskq(1): genunix‘callout_execute called 28 times
USB_hid_81_pipehndl_tq_1(14): usba‘hcdi_cb_thread called 228 times
callout_taskq(0): genunix‘callout_execute called 706 times
callout_taskq(1): genunix‘callout_execute called 24 times
USB_hid_81_pipehndl_tq_1(14): usba‘hcdi_cb_thread called 141 times
callout_taskq(0): genunix‘callout_execute called 708 times
```

◆ ◆ ◆ 第 6 章

6 驱动程序自动配置

自动配置表示驱动程序会将代码和静态数据装入内存中。随后在系统中注册此信息。在自动配置过程中还会连接由驱动程序控制的各个设备实例。

本章介绍有关以下主题的信息：

驱动程序的装入和卸载

系统从用于自动配置的内核模块目录的 `drv` 子目录装入驱动程序二进制模块。请参见第 456 页中的 “将驱动程序复制到模块目录”。

将模块读入内存且解析了所有符号之后，系统将调用此模块的 `_init(9E)` 入口点。`_init()` 函数将调用 `mod_install(9F)`，实际上就是装入此模块。

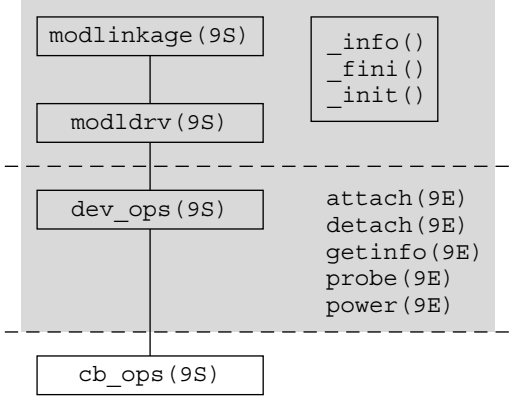
注 – 调用 `mod_install()` 期间，一旦调用了 `mod_install()`，其他线程便可以调用 `attach(9E)`。从编程角度来看，在调用 `mod_install()` 之前必须执行所有 `_init()` 初始化。如果 `mod_install()` 失败（即返回非零值），则必须取消初始化。

一旦 `_init()` 成功完成，便会在系统中正确注册驱动程序。实际上，此时驱动程序并不管理任何设备。设备管理是在设备配置过程中进行的。

为了节省系统内存或根据用户的明确请求，系统会卸载驱动程序二进制模块。从内存中删除驱动程序代码和数据之前，将调用该驱动程序的 `_fini(9E)` 入口点。当且仅当 `_fini()` 返回成功信息时，才会卸载驱动程序。

下图概述了设备驱动程序的结构。阴影区域突出显示驱动程序的数据结构和入口点。阴影区域的上半部分包括支持驱动程序装入和卸载的数据结构和入口点。下半部分与驱动程序配置相关。

图 6–1 模块装入和自动配置入口点



驱动程序必需的数据结构

为了支持自动配置，驱动程序需要静态初始化以下数据结构：

- modlinkage(9S)
- modldrv(9S)
- dev_ops(9S)
- cb_ops(9S)

驱动程序依赖于图 5-1 中的数据结构。必须提供并正确初始化这些数据结构。没有这些数据结构，可能无法正确装入驱动程序。结果导致可能无法装入必需的例程。如果驱动程序不支持某个操作，则 nodev(9F) 例程的地址可以用作占位符。在某些情况下，驱动程序支持入口点，并且仅需要返回成功信息或失败信息。在这种情况下，可以使用例程 nulldev(9F) 的地址。

注 – 应该在编译时对这些结构进行初始化。在任何其他时间，驱动程序都不应访问或更改这些结构。

---

modlinkage 结构

```
static struct modlinkage xxmodlinkage = {
    MODREV_1,       /* ml_rev */
    &xxmodldrv,     /* ml_linkage[] */
```

```
    NULL            /* NULL termination */
};
```

第一个字段是装入子系统的模块的版本号。该字段应为 MODREV_1。第二个字段指向接下来定义的驱动程序的 modldrv 结构。该结构的最后一个元素应始终为 NULL。

modldrv 结构

```
static struct modldrv xxmodldrv = {
    &mod_driverops,           /* drv_modops */
    "generic driver v1.1",    /* drv_linkinfo */
    &xx_dev_ops               /* drv_dev_ops */
};
```

该结构更加详细地描述模块。第一个字段提供有关模块安装的信息。对于驱动程序模块，该字段应设置为 &mod_driverops。第二个字段是将由 modinfo(1M) 显示的字符串。第二个字段应包含足够的信息，以便确定生成驱动程序二进制文件的源代码版本。最后一个字段指向下节所定义的驱动程序的 dev_ops 结构。

dev_ops 结构

```
static struct dev_ops xx_dev_ops = {
    DEVO_REV,                 /* devo_rev */
    0,                        /* devo_refcnt  */
    xxgetinfo,                /* devo_getinfo: getinfo(9E) */
    nulldev,                  /* devo_identify: identify(9E) */
    xxprobe,                  /* devo_probe: probe(9E) */
    xxattach,                 /* devo_attach: attach(9E) */
    xxdetach,                 /* devo_detach: detach(9E) */
    nodev,                    /* devo_reset: see devo_quiesce */
    &xx_cb_ops,               /* devo_cb_ops */
    NULL,                     /* devo_bus_ops */
    &xxpower,                 /* devo_power: power(9E) */
    ddi_quiesce_not_needed,   /* devo_quiesce: quiesce(9E) */
};
```

使用 dev_ops(9S) 结构，内核可以找到设备驱动程序的自动配置入口点。devo_rev 字段标识结构的修订号。该字段必须设置为 DEVO_REV。devo_refcnt 字段必须初始化为零。应使用相应驱动程序的入口点地址填充函数地址字段，但以下情况除外：

- 将 devo_identify 字段设置为 nulldev(9F)。identify() 入口点已过时。
- 如果不需要 probe(9E) 例程，应将 devo_probe 字段设置为 nulldev(9F)。
- 将 devo_reset 字段设置为 nodev(9F)。nodev() 函数返回 ENXIO。请参见 devo_quiesce。
- 如果不需要 power() 例程，应将 devo_power 字段设置为 NULL。提供电源管理功能的设备的驱动程序必须具有 power(9E) 入口点。请参见第 12 章，电源管理。
- 如果驱动程序不需要实现停止，请将 devo_quiesce 字段设置为 ddi_quiesce_not_needed()。对设备进行管理的驱动程序必须提供一个 quiesce(9E) 入口点。

devo_cb_ops 成员应包含 cb_ops(9S) 结构的地址。devo_bus_ops 字段必须设置为 NULL。

cb_ops 结构

```
static struct cb_ops xx_cb_ops = {
    xxopen,         /* open(9E) */
    xxclose,        /* close(9E) */
    xxstrategy,     /* strategy(9E) */
    xxprint,        /* print(9E) */
    xxdump,         /* dump(9E) */
    xxread,         /* read(9E) */
    xxwrite,        /* write(9E) */
    xxioctl,        /* ioctl(9E) */
    xxdevmap,       /* devmap(9E) */
    nodev,          /* mmap(9E) */
    xxsegmap,       /* segmap(9E) */
    xxchpoll,       /* chpoll(9E) */
    xxprop_op,      /* prop_op(9E) */
    NULL,           /* streamtab(9S) */
    D_MP | D_64BIT, /* cb_flag */
    CB_REV,         /* cb_rev */
    xxaread,        /* aread(9E) */
    xxawrite        /* awrite(9E) */
};
```

cb_ops(9S) 结构包含设备驱动程序的字符操作和块操作的入口点。驱动程序不支持的所有入口点应初始化为 nodev(9F)。例如，字符设备驱动程序应该将所有块字段（例如 cb_stategy）设置为 nodev(9F)。请注意，保留 mmap(9E) 入口点是为了兼容早期发行版。驱动程序应使用 devmap(9E) 入口点来进行设备内存映射。如果支持 devmap(9E)，应将 mmap(9E) 设置为 nodev(9F)。

streamtab 字段表明驱动程序是否基于 STREAMS。只有第 19 章，网络设备驱动程序中讨论的网络设备驱动程序基于 STREAMS。所有不基于 STREAMS 的驱动程序**必须**将 streamtab 字段设置为 NULL。

cb_flag 成员包含以下标志：

- D_MP 标志表示驱动程序对于多线程是安全的。Oracle Solaris OS 仅支持线程安全的驱动程序，因此必须设置 D_MP。
- D_64BIT 标志导致驱动程序使用 uio(9S) 结构的 uio_loffset 字段。驱动程序应在 cb_flag 字段中设置 D_64BIT 标志，以便正确处理 64 位偏移。
- D_DEVMAP 标志支持 devmap(9E) 入口点。有关 devmap(9E) 的信息，请参见第 10 章，映射设备和内核内存。

cb_rev 是 cb_ops 结构修订号。该字段必须设置为 CB_REV。

可装入驱动程序接口

设备驱动程序必须是可动态装入的。驱动程序还应是可卸载的，以帮助节省内存资源。可卸载驱动程序还应易于测试、调试和修补。

每个设备驱动程序都需要实现 _init(9E)、_fini(9E) 和 _info(9E) 入口点以支持驱动程序的装入和卸载。以下示例显示了可装入驱动程序接口的典型实现。

示例 6–1 可装入接口部分

```
static void *statep;                /* for soft state routines */
static struct cb_ops xx_cb_ops;     /* forward reference */
static struct dev_ops xx_ops = {
    DEVO_REV,
    0,
    xxgetinfo,
    nulldev,
    xxprobe,
    xxattach,
    xxdetach,
    xxreset,
    nodev,
    &xx_cb_ops,
    NULL,
    xxpower,
    ddi_quiesce_not_needed,
};

static struct modldrv modldrv = {
    &mod_driverops,
    "xx driver v1.0",
    &xx_ops
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modldrv,
    NULL
};

int
_init(void)
{
    int error;
    ddi_soft_state_init(&statep, sizeof (struct xxstate),
        estimated_number_of_instances);
    /* further per-module initialization if necessary */
    error = mod_install(&modlinkage);
    if (error != 0) {
        /* undo any per-module initialization done earlier */
        ddi_soft_state_fini(&statep);
    }
    return (error);
}

int
```

示例 6–1 可装入接口部分 （续）

```
_fini(void)
{
    int error;
    error = mod_remove(&modlinkage);
    if (error == 0) {
        /* release per-module resources if any were allocated */
        ddi_soft_state_fini(&statep);
    }
    return (error);
}

int
_info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}
```

_init() 示例

以下示例显示了典型的 _init(9E) 接口。

示例 6–2 _init() 函数

```
static void *xxstatep;
int
_init(void)
{
    int error;
    const int max_instance = 20;    /* estimated max device instances */

    ddi_soft_state_init(&xxstatep, sizeof (struct xxstate), max_instance);
    error = mod_install(&xxmodlinkage);
    if (error != 0) {
        /*
         * Cleanup after a failure
         */
        ddi_soft_state_fini(&xxstatep);
    }
    return (error);
}
```

在 _init() 中装入驱动程序期间，驱动程序应执行所有一次性资源分配或数据初始化。例如，在该例程中驱动程序应初始化所有对于该驱动程序为全局互斥锁的互斥锁。但是，驱动程序不应使用 _init(9E) 来分配或初始化与设备特定实例有关的任何内容。必须在 attach(9E) 中完成每个实例的初始化。例如，如果打印机的驱动程序可以同时处理多台打印机，则该驱动程序应在 attach() 中分配特定于每台打印机实例的资源。

注 – 一旦 _init(9E) 调用了 mod_install(9F)，驱动程序便不应更改连接至 modlinkage(9S) 结构的任何数据结构，因为系统可能会复制或更改这些数据结构。

_fini() 示例

以下示例显示了 _fini() 例程。

```
int
_fini(void)
{
    int error;
    error = mod_remove(&modlinkage);
    if (error != 0) {
        return (error);
    }
    /*
     * Cleanup resources allocated in _init()
     */
    ddi_soft_state_fini(&xxstatep);
    return (0);
}
```

同样，在 _fini() 中，驱动程序应该释放在 _init() 中分配的所有资源。驱动程序必须将其自身从系统模块列表中删除。

注 – 将驱动程序连接至硬件实例时，可能会调用 _fini()。在本示例中，mod_remove(9F) 返回失败信息。因此，在 mod_remove() 返回成功信息之前，不应释放驱动程序资源。

_info() 示例

以下示例显示了 _info(9E) 例程。

```
int
_info(struct modinfo *modinfop)
{
    return (mod_info(&xxmodlinkage, modinfop));
}
```

调用该驱动程序是为了返回模块信息。应按如上所示实现入口点。

设备配置概念

系统基于节点名称和 compatible 属性为内核设备树中的每个节点选择驱动程序（请参见第 62 页中的 “将驱动程序绑定到设备”）。相同的驱动程序可能会绑定到多个设备节点。驱动程序可以根据系统指定的实例编号来区分不同的节点。

为设备节点选择驱动程序之后，将调用该驱动程序的 probe(9E) 入口点以确定系统上是否存在该设备。如果 probe() 成功，将调用该驱动程序的 attach(9E) 入口点以设置和管理设备。当且仅当 attach() 返回成功信息时，才能打开该设备（请参见第 101 页中的 “attach() 入口点”）。

可能会取消配置设备以节省系统内存资源，或在系统仍在运行时使设备可以移除。要取消配置设备，系统首先会检查是否引用了设备实例。此检查将调用驱动程序的 getinfo(9E) 入口点以获取仅为该驱动程序所知的信息（请参见第 107 页中的 “getinfo() 入口点”）。如果未引用设备实例，将调用驱动程序的 detach(9E) 例程来取消配置设备（请参见第 106 页中的 “detach() 入口点”）。

要进行更新，每个驱动程序都必须定义内核用于设备配置的以下入口点：

- probe(9E)
- attach(9E)
- detach(9E)
- getinfo(9E)

请注意，attach()、detach() 和 getinfo() 是必需的。只有无法自我识别的设备需要 probe()。对于自标识设备，可以提供显式 probe() 例程，或者在 dev_ops 结构中为 probe() 入口点指定 nulldev(9F)。

设备实例和实例编号

系统会为每个设备指定一个实例编号。驱动程序可能无法可靠地预测指定给某个特定设备的实例编号值。驱动程序应通过调用 ddi_get_instance(9F) 来检索已指定的特定实例编号。

实例编号代表了系统中的设备。内核会为特定驱动程序的每个 dev_info（即设备树中的每个节点）指定一个实例编号。此外，实例编号可提供一种便捷的、为特定于某个物理设备的数据建立索引的机制。实例编号的最常见用法是 ddi_get_soft_state(9F)，也就是使用实例编号检索特定物理设备的软状态数据。

注意 – 对于伪设备（即伪结点的子结点），其实例编号是采用 instance 属性在 driver.conf(4) 文件中定义的。如果 driver.conf 文件不包含 instance 属性，则未定义此行为。对于硬件设备节点，当 OS 首次发现此类设备时，系统会为其指定实例编号。实例编号在系统重新引导以及 OS 升级期间保持不变。

次要节点和次要设备号

驱动程序负责管理其次要设备号名称空间。例如，sd 驱动程序需要向每个磁盘的文件系统导出八个字符次要节点和八个块次要节点。每个次要节点代表部分磁盘的块接口或字符接口。getinfo(9E) 入口点通知系统有关次要设备号到设备实例的映射（请参见第 107 页中的“getinfo() 入口点”）。

probe() 入口点

对于非自我识别设备，probe(9E) 入口点应确定系统上是否存在硬件设备。

对于 probe()，要确定是否存在设备实例，probe()需要执行通常 attach(9E) 也执行的许多任务。尤其是，probe() 可能需要映射设备寄存器。

探测设备寄存器是特定于设备的。驱动程序通常必须执行一系列硬件测试来确保硬件确实存在。测试条件必须足够严格以避免错误地识别设备。例如，在设备实际上不可用的情况下可能显示存在该设备，因为异常设备在行为上看起来与预期的设备相似。

测试返回以下标志：

- DDI_PROBE_SUCCESS（探测成功）
- DDI_PROBE_FAILURE（探测失败）
- DDI_PROBE_DONTCARE（探测不成功，但仍需调用 attach(9E)）
- DDI_PROBE_PARTIAL（现在不存在实例，但将来可能会出现）

对于给定设备实例，直到 probe(9E) 在该设备上至少成功一次时，才会调用 attach(9E)。

probe(9E) 必须释放 probe() 已分配的所有资源，因为可能会调用 probe() 多次。但是，即使 probe(9E) 已成功，也不一定要调用 attach(9E)。

可以在驱动程序的 probe(9E) 例程中使用 ddi_dev_is_sid(9F) 来确定设备是否可以自我识别。在为同一设备的自我识别版本和非自我识别版本编写驱动程序时，ddi_dev_is_sid() 非常有用。

以下示例是一个样例 probe() 例程。

示例 6–3 probe(9E) 例程

```
static int
xxprobe(dev_info_t *dip)
{
    ddi_acc_handle_t dev_hdl;
    ddi_device_acc_attr_t dev_attr;
    Pio_csr *csrp;
    uint8_t csrval;

    /*
```

示例 6–3 probe(9E) 例程 （续）

```
     * if the device is self identifying, no need to probe
     */
    if (ddi_dev_is_sid(dip) == DDI_SUCCESS)
    return (DDI_PROBE_DONTCARE);

    /*
     * Initalize the device access attributes and map in
     * the devices CSR register (register 0)
     */
    dev_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    dev_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&csrp, 0, sizeof (Pio_csr),
    &dev_attr, &dev_hdl) != DDI_SUCCESS)
    return (DDI_PROBE_FAILURE);

    /*
     * Reset the device
     * Once the reset completes the CSR should read back
     * (PIO_DEV_READY | PIO_IDLE_INTR)
     */
    ddi_put8(dev_hdl, csrp, PIO_RESET);
    csrval = ddi_get8(dev_hdl, csrp);

    /*
     * tear down the mappings and return probe success/failure
     */
    ddi_regs_map_free(&dev_hdl);
    if ((csrval & 0xff) == (PIO_DEV_READY | PIO_IDLE_INTR))
    return (DDI_PROBE_SUCCESS);
    else
    return (DDI_PROBE_FAILURE);
}
```

调用驱动程序的 probe(9E) 例程时，驱动程序并不知道正在探测的设备是否存在于总线上。因此，驱动程序可能会尝试访问不存在设备的设备寄存器。结果，在某些总线上可能会产生总线故障。

以下示例显示了使用 ddi_poke8(9F) 来检查设备是否存在的 probe(9E) 例程。ddi_poke8() 谨慎地尝试将值写入指定的虚拟地址，必要时使用父结点驱动程序协助进程。如果地址无效或无法在不出现错误的情况下写入值，则会返回错误代码。另请参见 ddi_peek(9F)。

在本示例中，使用 ddi_regs_map_setup(9F) 来映射设备寄存器。

示例 6–4 使用 ddi_poke8(9F) 的 probe(9E) 例程

```
static int
xxprobe(dev_info_t *dip)
{
    ddi_acc_handle_t dev_hdl;
```

示例 6–4 使用 ddi_poke8(9F) 的 probe(9E) 例程 （续）

```
    ddi_device_acc_attr_t dev_attr;
    Pio_csr *csrp;
    uint8_t csrval;

    /*
     * if the device is self-identifying, no need to probe
     */
    if (ddi_dev_is_sid(dip) == DDI_SUCCESS)
    return (DDI_PROBE_DONTCARE);

    /*
     * Initialize the device access attrributes and map in
     * the device's CSR register (register 0)
     */
    dev_attr.devacc_attr_version - DDI_DEVICE_ATTR_V0;
    dev_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;

    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&csrp, 0, sizeof (Pio_csr),
    &dev_attr, &dev_hdl) != DDI_SUCCESS)
    return (DDI_PROBE_FAILURE);

    /*
     * The bus can generate a fault when probing for devices that
     * do not exist.  Use ddi_poke8(9f) to handle any faults that
     * might occur.
     *
     * Reset the device.  Once the reset completes the CSR should read
     * back (PIO_DEV_READY | PIO_IDLE_INTR)
     */
    if (ddi_poke8(dip, csrp, PIO_RESET) != DDI_SUCCESS) {
    ddi_regs_map_free(&dev_hdl);
    return (DDI_FAILURE);

    csrval = ddi_get8(dev_hdl, csrp);
    /*
     * tear down the mappings and return probe success/failure
     */
    ddi_regs_map_free(&dev_hdl);
    if ((csrval & 0xff) == (PIO_DEV_READY | PIO_IDLE_INTR))
    return (DDI_PROBE_SUCCESS);
    else
    return (DDI_PROBE_FAILURE);
}
```

attach() 入口点

内核调用驱动程序的 attach(9E) 入口点来连接设备实例或针对已通过电源管理框架暂停或关闭的设备实例恢复操作。本节仅讨论连接设备实例的操作。电源管理将在第 12 章，电源管理中讨论。

调用驱动程序的 attach(9E) 入口点以连接每个绑定到驱动程序的设备实例。基于要连接的设备节点实例，并将 attach(9E) 的 cmd 参数指定为 DDI_ATTACH，来调用此入口点。attach 入口点主要包括以下类型的处理：

- 为设备实例分配软状态结构
- 初始化每实例互斥锁
- 初始化条件变量
- 注册设备的中断
- 映射设备实例的寄存器和内存
- 为设备实例创建从设备节点
- 报告设备实例已连接

驱动程序软状态管理

为了协助设备驱动程序编写人员分配状态结构，Oracle Solaris DDI/DKI 提供了一组内存管理例程，称为**软件状态管理例程**，也称为**软状态例程**。这些例程可动态分配、检索以及销毁指定大小的内存项，并可隐藏列表管理的详细信息。**实例编号**标识所需的内存项。此编号通常为系统指定的实例编号。

通常，驱动程序会为与其连接的每个设备实例分配软状态结构，方法是调用 ddi_soft_state_zalloc(9F) 并传递设备的实例编号。由于两个设备节点不能具有相同的实例编号，所以对于已经分配出去的给定实例编号，ddi_soft_state_zalloc(9F) 将失败。

驱动程序的字符入口点或块入口点（cb_ops(9S)）通过先解码来自传递到入口点函数的 dev_t 参数的设备实例编号，来引用特定的软状态结构。随后，驱动程序调用 ddi_get_soft_state(9F)，传递每个驱动程序的软状态列表和生成的实例编号。返回值 NULL 表明实际上不存在该设备并且应由驱动程序返回相应的代码。

有关实例编号和设备编号（dev_t 编号）之间关系的更多信息，请参见第 102 页中的“创建从设备节点”。

锁变量和条件变量的初始化

驱动程序在连接期间应初始化所有基于实例的锁和条件变量。添加任何中断处理程序之前，**必须**先初始化驱动程序中断处理程序所获取的所有锁。有关锁的初始化和使用的说明，请参见第 3 章，多线程。有关中断处理程序和锁问题的讨论，请参见第 8 章，中断处理程序。

创建从设备节点

连接过程的一个重要部分是为设备实例创建**次要节点**。次要节点包含由设备和 DDI 框架导出的信息。系统使用此信息为 /devices 下的次要节点创建**特殊文件**。

驱动程序调用 ddi_create_minor_node(9F) 时会创建次要节点。驱动程序提供**次要设备号**、**次要名称**、**次要节点类型**，以及次要节点是代表块设备还是字符设备。

驱动程序可以为设备创建任意数量的次要节点。Oracle Solaris DDI/DKI 期望某些类别的设备具有以特定格式创建的次要节点。例如，期望磁盘驱动程序为连接的每个物理磁盘实例创建 16 个次要节点。将创建八个代表块设备接口 `a - h` 的次要节点，另外八个次要节点代表字符设备接口 `a,raw - h,raw`。

传递给 `ddi_create_minor_node(9F)` 的**次要设备号**全部由驱动程序定义。次要设备号通常是设备实例编号和次要节点标识符的编码。在前面的示例中，驱动程序会为每个次要节点创建次要设备号，方法是将设备的实例编号左移三位，再将该结果与次要节点索引进行“或”运算。次要节点索引值的范围介于 0 和 7 之间。请注意，次要节点 `a` 和 `a,raw` 共用同一次要设备号。这些次要节点根据传递到 `ddi_create_minor_node()` 的 *spec_type* 参数来区分。

传递给 `ddi_create_minor_node(9F)` 的**次要节点类型**对设备类型进行分类，如磁盘、磁带、网络接口、帧缓存器等。

下表列出了可以创建的可能的节点类型。

表 6–1 可能节点类型

常量	说明
`DDI_NT_SERIAL`	串行端口
`DDI_NT_SERIAL_DO`	拨出端口
`DDI_NT_BLOCK`	硬盘
`DDI_NT_BLOCK_CHAN`	带有通道或目标编号的硬盘
`DDI_NT_CD`	ROM 驱动器 (CD-ROM)
`DDI_NT_CD_CHAN`	带有通道或目标编号的 ROM 驱动器
`DDI_NT_FD`	软盘
`DDI_NT_TAPE`	磁带机
`DDI_NT_NET`	网络设备
`DDI_NT_DISPLAY`	显示设备
`DDI_NT_MOUSE`	鼠标
`DDI_NT_KEYBOARD`	键盘
`DDI_NT_AUDIO`	音频设备
`DDI_PSEUDO`	通用的伪设备

节点类型 `DDI_NT_BLOCK`、`DDI_NT_BLOCK_CHAN`、`DDI_NT_CD` 和 `DDI_NT_CD_CHAN` 会使 `devfsadm(1M)` 将设备实例标识为磁盘，并在 `/dev/dsk` 或 `/dev/rdsk` 目录中创建名称。

节点类型 DDI_NT_TAPE 会使 devfsadm(1M) 将设备实例标识为磁带，并在 /dev/rmt 目录中创建名称。

节点类型 DDI_NT_SERIAL 和 DDI_NT_SERIAL_DO 会使 devfsadm(1M) 执行以下操作：

- 将设备实例标识为串行端口
- 在 /dev/term 目录中创建名称
- 向 /etc/inittab 文件中添加项

供应商提供的字符串应包括使字符串唯一的标识值，如名称或股票名称。该字符串可与 devfsadm(1M) 和 devlinks.tab 文件（请参见 devlinks(1M) 手册页）一起使用以在 /dev 中创建逻辑名称。

延迟连接

在相应实例上的 attach(9E) 成功之前，可能会对次要设备调用 open(9E)。然后 open() 必须返回 ENXIO，这将导致系统尝试连接该设备。如果 attach() 成功，则会自动重试 open()。

示例 6–5 典型 attach() 入口点

```
/*
 * Attach an instance of the driver.  We take all the knowledge we
 * have about our board and check it against what has been filled in
 * for us from our FCode or from our driver.conf(4) file.
 */
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance;
    Pio *pio_p;
    ddi_device_acc_attr_t   da_attr;
    static int pio_validate_device(dev_info_t *);

    switch (cmd) {
    case DDI_ATTACH:
    /*
     * first validate the device conforms to a configuration this driver
     * supports
     */
    if (pio_validate_device(dip) == 0)
        return (DDI_FAILURE);
    /*
     * Allocate a soft state structure for this device instance
     * Store a pointer to the device node in our soft state structure
     * and a reference to the soft state structure in the device
     * node.
     */
    instance = ddi_get_instance(dip);
    if (ddi_soft_state_zalloc(pio_softstate, instance) != 0)
        return (DDI_FAILURE);
    pio_p = ddi_get_soft_state(pio_softstate, instance);
    ddi_set_driver_private(dip, (caddr_t)pio_p);
    pio_p->dip = dip;
```

示例 6–5 典型 attach() 入口点 （续）

```
    /*
     * Before adding the interrupt, get the interrupt block
     * cookie associated with the interrupt specification to
     * initialize the mutex used by the interrupt handler.
     */
    if (ddi_get_iblock_cookie(dip, 0, &pio_p->iblock_cookie) !=
      DDI_SUCCESS) {
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }

    mutex_init(&pio_p->mutex, NULL, MUTEX_DRIVER, pio_p->iblock_cookie);
    /*
     * Now that the mutex is initialized, add the interrupt itself.
     */
    if (ddi_add_intr(dip, 0, NULL, NULL, pio_intr, (caddr_t)instance) !=
      DDI_SUCCESS) {
        mutex_destroy(&pio_p>mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }
    /*
     * Initialize the device access attributes for the register mapping
     */
    dev_acc_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
    dev_acc_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
    dev_acc_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
    /*
     * Map in the csr register (register 0)
     */
    if (ddi_regs_map_setup(dip, 0, (caddr_t *)&(pio_p->csr), 0,
        sizeof (Pio_csr), &dev_acc_attr, &pio_p->csr_handle) !=
        DDI_SUCCESS) {
        ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
        mutex_destroy(&pio_p->mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }
    /*
     * Map in the data register (register 1)
     */
    if (ddi_regs_map_setup(dip, 1, (caddr_t *)&(pio_p->data), 0,
        sizeof (uchar_t), &dev_acc_attr, &pio_p->data_handle) !=
        DDI_SUCCESS) {
        ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
        ddi_regs_map_free(&pio_p->csr_handle);
        mutex_destroy(&pio_p->mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }
    /*
     * Create an entry in /devices for user processes to open(2)
     * This driver will create a minor node entry in /devices
     * of the form:  /devices/..../pio@X,Y:pio
     */
    if (ddi_create_minor_node(dip, ddi_get_name(dip), S_IFCHR,
```

示例 6–5 典型 attach() 入口点 （续）

```
        instance, DDI_PSEUDO, 0) == DDI_FAILURE) {
        ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
        ddi_regs_map_free(&pio_p->csr_handle);
        ddi_regs_map_free(&pio_p->data_handle);
        mutex_destroy(&pio_p->mutex);
        ddi_soft_state_free(pio_softstate, instance);
        return (DDI_FAILURE);
    }
    /*
     * reset device (including disabling interrupts)
     */
    ddi_put8(pio_p->csr_handle, pio_p->csr, PIO_RESET);
    /*
     * report the name of the device instance which has attached
     */
    ddi_report_dev(dip);
    return (DDI_SUCCESS);

    case DDI_RESUME:
    return (DDI_SUCCESS);

    default:
    return (DDI_FAILURE);
    }
}
```

注 – attach() 例程不能对不同设备实例上的调用顺序做出任何假设。系统可以并行调用不同设备实例上的 attach()。系统还可以在不同设备实例上同时调用 attach() 和 detach()。

detach() 入口点

内核调用驱动程序的 detach(9E) 入口点，通过电源管理来分离设备的某个实例或暂停对设 备某个实例的操作。本节讨论分离设备实例的操作。有关电源管理问题的讨论，请参阅第 12 章，电源管理。

调用驱动程序detach() 入口点以分离绑定到该驱动程序的设备的某个实例。该入口点是使用要分离的设备节点的实例和指定为该入口点的 cmd 参数的 DDI_DETACH 来调用的。

驱动程序需要取消或等待所有超时或回调完成，然后在返回前释放分配给设备实例的所有资源。如果由于某种原因，驱动程序无法取消未完成的回调以释放资源，则驱动程序需要将设备返回至其初始状态并从入口点返回 DDI_FAILURE，使设备实例保持连接状态。

有两种类型的回调例程：可取消回调例程和不可取消回调例程。驱动程序在 detach(9E) 期间可以原子方式取消 timeout(9F) 和 bufcall(9F) 回调例程。其他类型的回

调例程，如 scsi_init_pkt(9F) 和 ddi_dma_buf_bind_handle(9F)，则不能被取消。驱动程序必须要么阻塞在 detach() 中直到回调完成，要么使分离请求失败。

示例 6–6 典型 detach() 入口点

```
/*
 * detach(9e)
 * free the resources that were allocated in attach(9e)
 */
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    Pio     *pio_p;
    int     instance;

    switch (cmd) {
    case DDI_DETACH:

    instance = ddi_get_instance(dip);
    pio_p = ddi_get_soft_state(pio_softstate, instance);

    /*
     * turn off the device
     * free any resources allocated in attach
     */
    ddi_put8(pio_p->csr_handle, pio_p->csr, PIO_RESET);
    ddi_remove_minor_node(dip, NULL);
    ddi_regs_map_free(&pio_p->csr_handle);
    ddi_regs_map_free(&pio_p->data_handle);
    ddi_remove_intr(pio_p->dip, 0, pio_p->iblock_cookie);
    mutex_destroy(&pio_p->mutex);
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_SUCCESS);

    case DDI_SUSPEND:
    default:
    return (DDI_FAILURE);
    }
}
```

getinfo() 入口点

系统调用 getinfo(9E) 以获取仅为驱动程序所知的配置信息。次要设备号到设备实例的映射完全由驱动程序控制。有时系统需要询问驱动程序特定的 dev_t 代表哪个设备。

getinfo() 函数可以采用 DDI_INFO_DEVT2INSTANCE 或 DDI_INFO_DEVT2DEVINFO 作为其 *infocmd* 参数。DDI_INFO_DEVT2INSTANCE 命令请求设备的实例编号。DDI_INFO_DEVT2DEVINFO 命令请求指向设备的 dev_info 结构的指针。

如果是 DDI_INFO_DEVT2INSTANCE，则 *arg* 为 dev_t，并且 getinfo() 必须将 dev_t 中的次要设备号转换为实例编号。在以下示例中，次要设备号**是**实例编号，因此 getinfo() 仅传回次要设备号。在这种情况下，驱动程序不能假定状态结构可用，因为可能在调用 attach() 之前调用 getinfo()。由驱动程序定义的次要设备号和实例编号之间的映射关系可以与此示例中的不同。但是，在所有情况下，映射必须是静态的。

如果是 DDI_INFO_DEVT2DEVINFO，则 *arg* 仍为 dev_t，因此，getinfo() 首先将设备的实例编号解码。然后 getinfo() 传送回保存在相应设备的驱动程序软状态结构中的 dev_info 指针，如以下示例所示。

示例 6–7 典型 getinfo() 入口点

```
/*
 * getinfo(9e)
 * Return the instance number or device node given a dev_t
 */
static int
xxgetinfo(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{
    int error;
    Pio *pio_p;
    int instance = getminor((dev_t)arg);

    switch (infocmd) {
    /*
     * return the device node if the driver has attached the
     * device instance identified by the dev_t value which was passed
     */
    case DDI_INFO_DEVT2DEVINFO:
    pio_p = ddi_get_soft_state(pio_softstate, instance);
    if (pio_p == NULL) {
        *result = NULL;
        error = DDI_FAILURE;
    } else {
        mutex_enter(&pio_p->mutex);
        *result = pio_p->dip;
        mutex_exit(&pio_p->mutex);
        error = DDI_SUCCESS;
    }
    break;
    /*
     * the driver can always return the instance number given a dev_t
     * value, even if the instance is not attached.
     */
    case DDI_INFO_DEVT2INSTANCE:
    *result = (void *)instance;
    error = DDI_SUCCESS;
    break;
    default:
    *result = NULL;
    error = DDI_FAILURE;
    }
    return (error);
}
```

注 – getinfo() 例程必须与驱动程序创建的次要节点保持同步。如果次要节点不同步，则任何热插拔操作都可能失败并导致系统混乱。

使用设备 ID

使用 Oracle Solaris DDI 接口，驱动程序可以提供**设备** *ID*，即设备的永久唯一标识符。**设备** *ID* 可用于识别或查找设备。**设备** *ID* 独立于 `/devices` 名称或设备编号 (`dev_t`)。应用程序可以使用 `libdevid(3LIB)` 中定义的函数来读取和处理由驱动程序注册的设备 ID。

在驱动程序可以导出**设备** *ID* 之前，驱动程序需要检验设备是否可以提供唯一 ID 或者将主机生成的唯一 ID 存储在正常情况下不可访问的区域中。例如，通用编号 (world-wide number, WWN) 是设备提供的唯一 ID。例如，设备 NVRAM 和保留扇区是不可访问区域，主机生成的唯一 ID 可以安全地存储在此区域中。

注册设备 ID

通常，驱动程序在其 `attach(9E)` 处理程序中初始化和注册设备 ID。如上所述，驱动程序负责注册永久**设备** *ID*。同时，驱动程序可能需要处理可直接提供唯一 ID (WWN) 的设备和向稳定存储器写入及从稳定存储器读取虚构 ID 的设备。

注册设备提供的 ID

如果设备可以为驱动程序提供唯一的标识符，则驱动程序可以直接使用此标识符初始化**设备** *ID* 并使用 Oracle Solaris DDI 注册此 ID。

```
/*
 * The device provides a guaranteed unique identifier,
 * in this case a SCSI3-WWN.  The WWN for the device has been
 * stored in the device's soft state.
 */
if (ddi_devid_init(dip, DEVID_SCSI3_WWN, un->un_wwn_len, un->un_wwn,
    &un->un_devid) != DDI_SUCCESS)
    return (DDI_FAILURE);

(void) ddi_devid_register(dip, un->un_devid);
```

注册虚构 ID

驱动程序还可能为不直接提供唯一 ID 的设备注册设备 ID。注册这些 ID 需要设备能够存储并检索保留区中的少量数据。随后，驱动程序可创建虚构设备 ID 并将其写入保留区中。

```
/*
 * the device doesn't supply a unique ID, attempt to read
 * a fabricated ID from the device's reserved data.
 */
if (xxx_read_deviceid(un, &devid_buf) == XXX_OK) {
    if (ddi_devid_valid(devid_buf) == DDI_SUCCESS) {
        devid_sz = ddi_devi_sizeof(devid_buf);
        un->un_devid = kmem_alloc(devid_sz, KM_SLEEP);
```

```
        bcopy(devid_buf, un->un_devid, devid_sz);
        ddi_devid_register(dip, un->un_devid);
        return (XXX_OK);
    }
}
/*
 * we failed to read a valid device ID from the device
 * fabricate an ID, store it on the device, and register
 * it with the DDI
 */
if (ddi_devid_init(dip, DEVID_FAB, 0, NULL, &un->un_devid)
    == DDI_FAILURE) {
    return (XXX_FAILURE);
}
if (xxx_write_deviceid(un) != XXX_OK) {
    ddi_devid_free(un->un_devid);
    un->un_devid = NULL;
    return (XXX_FAILURE);
}
ddi_devid_register(dip, un->un_devid);
return (XXX_OK);
```

注销设备 ID

通常，驱动程序会注销并释放处理 detach(9E) 时分配的所有**设备** *ID*。驱动程序首先调用 ddi_devid_unregister(9F) 来注销设备实例的**设备** *ID*。然后，驱动程序必须通过调用 ddi_devid_free(9F) 并传送已由 ddi_devid_init(9F) 返回的句柄来释放**设备** *ID* 句柄自身。驱动程序负责管理为 WWN 或序列号数据分配的任何空间。

◆ ◆ ◆ 第 7 章

7 设备访问：程控 I/O

Oracle Solaris OS 为驱动程序开发者提供了一整套用于访问设备内存的接口。这些接口旨在通过处理处理器和设备字节存储顺序之间的不匹配，并强制实施设备可能具有的任何数据顺序相关性，使驱动程序与平台无关。通过使用这些接口，可以开发一种可在 SPARC 和 x86 处理器体系结构以及每个相应处理器系列的各种平台上运行的单个源驱动程序。

本章介绍有关以下主题的信息：

设备内存

系统会为支持程控 I/O 的设备指定一个或多个总线地址空间区域，这些区域映射到设备的可寻址区域。这些映射在与设备相关的 reg 属性中描述为值对。每个值对描述一段总线地址。

驱动程序通过指定寄存器编号（即 regspec，设备的 reg 属性的索引）来标识特定的总线地址映射。reg 属性标识设备的 busaddr 和 size。驱动程序在调用 DDI 函数（如 ddi_regs_map_setup(9F)）时传递寄存器编号。驱动程序通过调用 ddi_dev_nregs(9F) 可以确定已为设备指定的可映射区域数。

管理设备和主机字节序之间的差别

主机的数据格式可以与设备的数据格式具有不同的字节序特征。在这种情况下，主机与设备间传送的数据需要进行字节交换，才能符合目标位置的数据格式要求。与主机具有相同字节序特征的设备无需对数据进行字节交换。

驱动程序通过在传递给 ddi_device_acc_attr(9S) 的 ddi_regs_map_setup(9F) 结构中设置相应的标志来指定设备的字节序特征。然后，DDI 框架在驱动程序调用 ddi_get*X* 例程（如 ddi_get8(9F)）或 ddi_put*X* 例程（如 ddi_put16(9F)）来读/写设备内存时，执行任何所需的字节交换。

管理数据排序要求

平台可以重新排列数据的负载和存储，以优化平台的性能。由于某些设备可能不允许重新排列，因此驱动程序在设置到设备的映射时需要指定设备的排序要求。

ddi_device_acc_attr 结构

此结构描述了设备的字节序和数据顺序要求。驱动程序需要对此结构进行初始化并将其作为一个参数传递给 ddi_regs_map_setup(9F)。

```
typedef struct ddi_device_acc_attr {
    ushort_t    devacc_attr_version;
    uchar_t     devacc_attr_endian_flags;
    uchar_t     devacc_attr_dataorder;
} ddi_device_acc_attr_t;
```

devacc_attr_version	指定 DDI_DEVICE_ATTR_V0
devacc_attr_endian_flags	描述设备的字节序特征。指定为一个位值，其可能值包括： ■ DDI_NEVERSWAP_ACC－从不交换数据 ■ DDI_STRUCTURE_BE_ACC－设备数据格式为大尾数法 ■ DDI_STRUCTURE_LE_ACC－设备数据格式为小尾数法
devacc_attr_dataorder	描述 CPU 根据设备的要求引用数据时必须遵循的顺序。指定为一个枚举值，其中数据访问限制的排列顺序为最严格到最不严格。 ■ DDI_STRICTORDER_ACC－主机必须按程序员指定的顺序发出引用。此标志为缺省行为。 ■ DDI_UNORDERED_OK_ACC－允许主机重新排列到设备内存的负载和存储。 ■ DDI_MERGING_OK_ACC－允许主机将单个存储合并到连续位置。此设置还表明需要重新排列。

- DDI_LOADCACHING_OK_ACC－允许主机从设备读取数据，直到发生存储。
- DDI_STORECACHING_OK_ACC－允许主机对写入设备的数据进行高速缓存。然后，主机可以延迟将数据写入设备，直到将来某一时间。

注 – 系统对数据的访问可能会比驱动程序在 devacc_attr_dataorder 中所做指定更严格。就数据访问而言，由从必须遵循严格的数据排序到可以执行高速缓存存储操作，驱动程序对主机的限制依次降低。

映射设备内存

驱动程序通常会在执行 attach(9E) 期间映射设备的所有区域。驱动程序通过调用 ddi_regs_map_setup(9F)、指定要映射的区域寄存器编号、区域的设备访问属性以及偏移和大小来映射设备内存区域。DDI 框架为设备区域设置映射并将一个不透明句柄返回给驱动程序。在从设备区域读取数据或向其中写入数据时，此数据访问句柄将作为一个参数传递给 ddi_get8(9F) 或 ddi_put8(9F) 系列例程。

驱动程序通过检查设备导出的映射数来验证设备映射的形式与驱动程序预期的形式是否匹配。驱动程序调用 ddi_dev_nregs(9F)，然后调用 ddi_dev_regsize(9F) 来验证每个映射的大小。

映射设置示例

下面的简单示例说明了 DDI 数据访问接口。此驱动程序用于虚构的小端字节序设备，该设备每次接受一个字符并在准备好接受另一个字符时生成中断。此设备实现两个寄存器集：第一个是 8 位 CSR 寄存器，第二个是 8 位数据寄存器。

示例 7–1 映射设置

```
#define CSR_REG 0
#define DATA_REG 1
/*
 * Initialize the device access attributes for the register
 * mapping
 */
dev_acc_attr.devacc_attr_version = DDI_DEVICE_ATTR_V0;
dev_acc_attr.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;
dev_acc_attr.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
/*
 * Map in the csr register (register 0)
 */
if (ddi_regs_map_setup(dip, CSR_REG, (caddr_t *)&(pio_p->csr), 0,
  sizeof (Pio_csr), &dev_acc_attr, &pio_p->csr_handle) != DDI_SUCCESS) {
    mutex_destroy(&pio_p->mutex);
```

示例 7–1 映射设置 （续）

```
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_FAILURE);
}
/*
 * Map in the data register (register 1)
 */
if (ddi_regs_map_setup(dip, DATA_REG, (caddr_t *)&(pio_p->data), 0,
  sizeof (uchar_t), &dev_acc_attr, &pio_p->data_handle) \
  != DDI_SUCCESS) {
    mutex_destroy(&pio_p->mutex);
    ddi_regs_map_free(&pio_p->csr_handle);
    ddi_soft_state_free(pio_softstate, instance);
    return (DDI_FAILURE);
}
```

设备访问函数

驱动程序结合使用 ddi_get8(9F) 和 ddi_put8(9F) 系列例程以及 ddi_regs_map_setup(9F) 返回的句柄，以与设备相互传送数据。DDI 框架自动处理为满足主机或设备的字节序格式所需的任何字节交换，并强制实施设备可能具有的任何存储排序约束。

DDI 提供了用于传送 8 位、16 位、32 位和 64 位数据的接口，以及用于重复传送多个值的接口。有关这些接口的完整列表和说明，请参见 ddi_get8(9F)、ddi_put8(9F)、ddi_rep_get8(9F) 和 ddi_rep_put8(9F) 例程系列的手册页。

以下示例建立在示例 7–1 的基础上，其中，驱动程序映射了设备的 CSR 寄存器和数据寄存器。在本示例中，调用驱动程序的 write(9E) 入口点时，会将数据缓冲区写入（每次一个字节）设备。

示例 7–2 映射设置：缓冲区

```
static  int
pio_write(dev_t dev, struct uio *uiop, cred_t *credp)
{
    int  retval;
    int  error = OK;
    Pio *pio_p = ddi_get_soft_state(pio_softstate, getminor(dev));

    if (pio_p == NULL)
        return (ENXIO);
    mutex_enter(&pio_p->mutex);
    /*
     * enable interrupts from the device by setting the Interrupt
     * Enable bit in the devices CSR register
     */
    ddi_put8(pio_p->csr_handle, pio_p->csr,
      (ddi_get8(pio_p->csr_handle, pio_p->csr) | PIO_INTR_ENABLE));
```

示例 7–2 映射设置：缓冲区 （续）

```
    while (uiop->uio_resid > 0) {
    /*
     * This device issues an IDLE interrupt when it is ready
     * to accept a character; the interrupt can be cleared
     * by setting PIO_INTR_CLEAR.  The interrupt is reasserted
     * after the next character is written or the next time
     * PIO_INTR_ENABLE is toggled on.
     *
     * wait for interrupt (see pio_intr)
     */
        cv_wait(&pio_p->cv, &pio_p->mutex);

    /*
     * get a character from the user's write request
     * fail the write request if any errors are encountered
     */
        if ((retval = uwritec(uiop)) == -1) {
            error = retval;
            break;
        }

    /*
     * pass the character to the device by writing it to
     * the device's data register
     */
        ddi_put8(pio_p->data_handle, pio_p->data, (uchar_t)retval);
    }

    /*
     * disable interrupts by clearing the Interrupt Enable bit
     * in the CSR
     */
    ddi_put8(pio_p->csr_handle, pio_p->csr,
      (ddi_get8(pio_p->csr_handle, pio_p->csr) & ~PIO_INTR_ENABLE));

    mutex_exit(&pio_p->mutex);
    return (error);
}
```

备用设备访问接口

除通过 ddi_get8(9F) 和 ddi_put8(9F) 接口系列实现所有设备访问之外，Oracle Solaris OS 还提供特定于特殊总线实现的接口。虽然在某些平台上这些函数会更加有效，但使用这些例程会限制驱动程序在设备的各总线版本间保持可移植的能力。

访问内存空间

对于内存映射访问，设备寄存器会出现在内存地址空间中。驱动程序可以将 ddi_get*X* 系列例程和 ddi_put*X* 系列用作标准设备访问接口的备用接口。

访问 I/O 空间

对于 I/O 空间访问，设备寄存器会出现在 I/O 空间中，其中每个可寻址元素都称为 I/O 端口。驱动程序可以将 ddi_io_get8(9F) 和 ddi_io_put8(9F) 例程用作标准设备访问接口的备用接口。

PCI 配置空间访问

要在不使用常规设备访问接口的情况下访问 PCI 配置空间，驱动程序需要通过调用 pci_config_setup(9F)（而非 ddi_regs_map_setup(9F)）来映射 PCI 配置空间。然后，驱动程序可以调用 pci_config_get8(9F) 和 pci_config_put8(9F) 接口系列，以访问 PCI 配置空间。

◆ ◆ ◆ 第 8 章

8 中断处理程序

本章介绍用于处理中断的机制，如分配、注册、服务以及删除中断。本章介绍有关以下主题的信息：

中断处理程序概述

中断是指设备发送给 CPU 的硬件信号。中断将通知 CPU 需要注意设备，并且 CPU 应该停止任何当前活动并对设备进行响应。如果 CPU 未在执行优先级比中断优先级高的任务，则 CPU 会暂停当前线程。然后，CPU 会调用发送中断信号的设备的中断处理程序。中断处理程序的工作是服务设备并防止此设备中断。中断处理程序返回后，CPU 便会恢复出现中断之前所执行的工作。

Oracle Solaris DDI/DKI 提供了用于执行以下任务的接口：

- 确定中断类型和注册要求
- 注册中断
- 服务中断
- 屏蔽中断
- 获取中断待处理信息
- 获取和设置优先级信息

设备中断

I/O 总线以两种常用方法来实现中断：**向量化**和**轮询**。这两种方法通常都会提供总线中断优先级别。向量化设备还会提供中断向量。轮询设备则不提供中断向量。

为了与不断发展的总线技术保持同步，Oracle Solaris OS 已经得到了增强，可适应更新类型的中断以及已经使用多年的较为传统的中断。具体来说，操作系统目前可识别三种类型的中断：

- **传统中断**－**传统**或**固定中断**是指使用早期总线技术的中断。使用这些技术，可通过一个或多个“带外”（即，独立于总线的主线）连线的外部管脚来发送中断信号。较新的总线技术（如 PCI Express）通过带内机制模拟传统中断来维持软件兼容性。主机 OS 将这些模仿中断视为传统中断。
- **消息告知中断**－消息告知中断 (message-signalled interrupt, MSI) 使用带内消息而不是使用管脚，可在主桥 (host bridge) 中确定中断的地址。（有关主桥 (host bridge) 的更多信息，请参见第 504 页中的 “PCI 局部总线”。）MSI 可以将数据与中断消息一起发送。每个 MSI 都不是共享的，这样可以保证指定给某一设备的 MSI 在系统中是唯一的。一个 PCI 函数最多可以请求 32 条 MSI 消息。
- **扩展消息告知中断**－扩展消息告知中断 (Extended message-signalled interrupt, MSI-X) 是 MSI 的增强版本。MSI-X 中断具有以下新增的优点：
  - 支持 2048 条而不是 32 条消息
  - 针对每条消息支持独立的消息地址和消息数据
  - 支持按消息屏蔽
  - 软件分配的向量少于硬件请求的向量时可具有更大灵活性。软件可以在多个 MSI-X 插槽中重用相同的 MSI-X 地址和数据。

注 – 一些较新的总线技术（如 PCI Express）要求使用 MSI，但是可以使用 INTx 仿真来处理传统中断。INTx 仿真用于实现兼容性，但是这并不被认为是好的做法。

高级别中断

总线会在**总线中断级别**设置设备中断的优先级。然后，总线中断级别将映射到处理器中断级别。映射到高于调度程序优先级别的 CPU 中断优先级的总线中断级别称为**高级别中断**。高级别中断处理程序仅限于调用以下 DDI 接口：

- 使用与高级别中断关联的中断优先级初始化的互斥锁上的 `mutex_enter(9F)` 和 `mutex_exit(9F)`
- `ddi_intr_trigger_softint(9F)`
- 以下 DDI `get` 和 `put` 例程：`ddi_get8(9F)`、`ddi_put8(9F)`、`ddi_get16(9F)`、`ddi_put16(9F)`、`ddi_get32(9F)`、`ddi_put32(9F)`、`ddi_get64(9F)` 和 `ddi_put64(9F)`。

总线中断级别本身无法确定设备是否会发生高级别中断。特定的总线中断级别可以在一个平台映射到高级别中断，而在其他平台上则映射到普通中断。

不要求驱动程序来支持具有高级别中断的设备。但是，要求驱动程序检查中断级别。如果中断优先级高于或等于系统最高优先级，中断处理程序会在高级别中断环境下运行。在这种情况下，驱动程序可能无法连接，或者驱动程序可能会使用双级别方案来处理中断。有关更多信息，请参见第 141 页中的 “处理高级别中断”。

传统中断

系统仅有的有关设备中断的信息为总线中断的优先级别和中断请求编号。例如，SPARC 计算机中 S 总线上的 IPL 即是总线中断的优先级别；x86 计算机中 ISA 总线上的 IRQ 即是中断请求编号。

注册中断处理程序之后，系统会将其添加到每个 IPL 或 IRQ 的潜在中断处理程序的列表中。出现中断时，系统必须确定与给定的 IPL 或 IRQ 关联的所有设备中实际导致此中断的设备。系统会针对指定的 IPL 或 IRQ 调用所有中断处理程序，直到一个处理程序声明中断为止。

以下总线可以支持轮询中断：

- S 总线
- ISA
- PCI

标准消息告知中断和扩展消息告知中断

标准 (MSI) 和扩展 (MSI-X) 消息告知中断均作为带内消息实现。消息告知中断可作为使用软件指定的地址和值的写操作进行发送。

MSI 中断

常规 PCI 规范包括可选的消息信号中断 (Message Signaled Interrupt, MSI) 支持。MSI 是作为发送的写操作实现的带内消息。MSI 的地址和数据由软件指定，并特定于主桥 (host bridge)。由于消息是带内消息，因此消息的接收可用于“推送”与中断关联的数据。根据定义，MSI 中断是独享的。指定给设备的每条 MSI 消息保证在系统中均为唯一消息。PCI 函数可以请求 1、2、4、8、16 或 32 条 MSI 消息。请注意，系统软件为函数分配的 MSI 消息数可以少于函数所请求的数量。可限制主桥 (host bridge) 中为设备分配的唯一 MSI 消息的数量。

MSI-X 中断

MSI-X 中断是 MSI 中断的增强版本，与 MSI 中断有相同功能，具有以下关键区别：

- 每个设备最多支持 2048 个 MSI-X 中断向量。
- 每个中断向量的地址和数据项都是唯一的。

- MSI-X 支持按函数屏蔽和按向量屏蔽。

利用 MSI-X 中断，未分配的设备中断向量可以使用先前添加或初始化的 MSI-X 中断向量共享相同的向量地址、向量数据、中断处理程序和处理程序参数。使用 ddi_intr_dup_handler(9F) 函数可相对于关联设备上未分配的中断向量为 Oracle Solaris OS 提供的资源设置别名。例如，如果为驱动程序分配了 2 个 MSI-X 中断，并且设备支持 32 个中断，则驱动程序可以使用 ddi_intr_dup_handler() 相对于设备上其他 30 个中断为其收到的 2 个中断设置别名。

ddi_intr_dup_handler() 函数可以复制使用 ddi_intr_add_handler(9F) 添加或使用 ddi_intr_enable(9F) 初始化的中断。

复制的中断最初处于禁用状态。可使用 ddi_intr_enable() 启用复制的中断。您不能删除原始 MSI-X 中断处理程序，除非删除了与此原始中断处理程序相关联的所有复制的中断处理程序。要删除复制的中断处理程序，请首先调用 ddi_intr_disable(9F)，然后调用 ddi_intr_free(9F)。当删除与该原始中断处理程序相关联的所有复制的中断处理程序后，就可以使用 ddi_intr_remove_handler(9F) 删除该原始 MSI-X 中断处理程序。有关示例，请参见 ddi_intr_dup_handler (9F) 手册页。

软件中断

Oracle Solaris DDI/DKI 支持软件中断（也称为**软中断**）。软中断通过软件而不是硬件设备启动。另外，还必须在系统中添加和删除这些中断的处理程序。软中断处理程序在中断上下文中运行，因此可用于执行许多属于中断处理程序的任务。

硬件中断处理程序必须快速执行其任务，因为它们可能必须在执行这些任务的同时暂停其他系统活动。对于高级别中断处理程序，更需要满足此要求，这些处理程序在高于系统调度程序的优先级别上运行。高级别中断处理程序将屏蔽所有较低优先级中断的操作，包括系统时钟的中断操作。因此，该中断处理程序必须避免涉及到可能导致其休眠的活动，如获取互斥锁。

如果处理程序休眠，则系统可能会挂起，因为时钟会被屏蔽，从而无法调度休眠线程。因此，高级别中断处理程序通常在高优先级别执行最少量的工作，并将其他任务委托给运行优先级别低于高级别中断处理程序的软件中断。由于软件中断处理程序运行的优先级别低于系统调度程序，因此软件中断处理程序可以执行高级别中断处理程序无法执行的操作。

DDI 中断函数

Oracle Solaris OS 提供了用于注册和取消注册中断的框架，并且提供了对消息信号中断 (Message Signaled Interrupt, MSI) 的支持。通过中断管理界面，可以处理优先级、功能和中断屏蔽，并可获取待处理信息。

中断功能函数

可使用以下函数获取中断信息：

ddi_intr_get_navail(9F)	返回可用于指定硬件设备和中断类型的中断的数量。
ddi_intr_get_nintrs(9F)	返回设备支持的指定中断类型的中断的数量。
ddi_intr_get_supported_types(9F)	返回设备和主机均支持的硬件中断类型。
ddi_intr_get_cap(9F)	针对指定的中断返回中断功能标志。

中断初始化和销毁函数

可使用以下函数创建和删除中断：

ddi_intr_alloc(9F)	为指定类型的中断分配系统资源和中断向量。
ddi_intr_free(9F)	针对指定的中断句柄释放系统资源和中断向量。
ddi_intr_set_cap(9F)	通过使用 DDI_INTR_FLAG_LEVEL 和 DDI_INTR_FLAG_EDGE 标志来设置指定中断的功能。
ddi_intr_add_handler(9F)	添加中断处理程序。
ddi_intr_dup_handler(9F)	仅适用于 MSI-X。将分配的中断向量的地址和数据对复制到同一设备上未使用的中断向量。
ddi_intr_remove_handler(9F)	删除指定的中断处理程序。
ddi_intr_enable(9F)	启用指定的中断。
ddi_intr_disable(9F)	禁用指定的中断。
ddi_intr_block_enable(9F)	仅用于 MSI。启用指定范围的中断。
ddi_intr_block_disable(9F)	仅用于 MSI。禁用指定范围的中断。
ddi_intr_set_mask(9F)	如果已启用指定的中断，则设置中断屏蔽码。
ddi_intr_clr_mask(9F)	如果已启用指定的中断，则清除中断屏蔽码。

ddi_intr_get_pending(9F)	如果主桥 (host bridge) 或设备支持这种中断待处理位，则读取此位。

优先级管理函数

可使用以下函数获取和设置优先级信息：

ddi_intr_get_pri(9F)	返回指定中断的当前软件优先级设置。
ddi_intr_set_pri(9F)	设置指定中断的中断优先级别。
ddi_intr_get_hilevel_pri(9F)	返回高级别中断的最低优先级别。

软中断函数

可使用以下函数处理软中断和软中断处理程序：

ddi_intr_add_softint(9F)	添加软中断处理程序。
ddi_intr_trigger_softint(9F)	触发指定的软中断。
ddi_intr_remove_softint(9F)	删除指定的软中断处理程序。
ddi_intr_get_softint_pri(9F)	返回指定中断的软中断优先级。
ddi_intr_set_softint_pri(9F)	更改指定软中断的相对软中断优先级。

中断函数示例

本节提供了执行以下任务的示例：

- 更改软中断优先级
- 检查待处理中断
- 设置中断屏蔽码
- 清除中断屏蔽码

示例 8–1 更改软中断优先级

使用 ddi_intr_set_softint_pri(9F) 函数将软中断优先级到更改为 9。

```
if (ddi_intr_set_softint_pri(mydev->mydev_softint_hdl, 9) != DDI_SUCCESS)
    cmn_err (CE_WARN, "ddi_intr_set_softint_pri failed");
```

示例 8–2 检查待处理中断

使用 ddi_intr_get_pending(9F) 函数检查中断是否处于待处理状态。

示例 8–2 检查待处理中断 （续）

```
if (ddi_intr_get_pending(mydevp->htable[0], &pending) != DDI_SUCCESS)
    cmn_err(CE_WARN, "ddi_intr_get_pending() failed");
else if (pending)
    cmn_err(CE_NOTE, "ddi_intr_get_pending(): Interrupt pending");
```

示例 8–3 设置中断屏蔽码

使用 ddi_intr_set_mask(9F) 函数设置中断屏蔽，以防止设备收到中断。

```
if ((ddi_intr_set_mask(mydevp->htable[0]) != DDI_SUCCESS))
    cmn_err(CE_WARN, "ddi_intr_set_mask() failed");
```

示例 8–4 清除中断屏蔽码

使用 ddi_intr_clr_mask(9F) 函数清除中断屏蔽。如果没有启用指定的中断，ddi_intr_clr_mask(9F) 函数将失败。如果 ddi_intr_clr_mask(9F) 函数成功，则设备将开始生成中断。

```
if (ddi_intr_clr_mask(mydevp->htable[0]) != DDI_SUCCESS)
    cmn_err(CE_WARN, "ddi_intr_clr_mask() failed");
```

注册中断

设备驱动程序必须首先通过调用 ddi_intr_add_handler(9F) 向系统注册中断处理程序，然后才能接收和服务中断。注册中断处理程序会为系统提供一种将中断处理程序与中断规范相关联的方法。如果设备可能负责中断，则会调用中断处理程序。此处理程序负责确定其是否应处理中断，如果是，则负责声明该中断。

提示 – 可在 mdb 或 kmdb 调试器中使用 ::interrupts 命令检索支持的 SPARC 和 x86 系统上设备的已注册中断信息。

注册传统中断

要注册驱动程序的中断处理程序，驱动程序通常会在其 attach(9E) 入口点执行以下步骤。

1. 使用 ddi_intr_get_supported_types(9F) 确定支持的中断类型。
2. 使用 ddi_intr_get_nintrs(9F) 确定支持的中断类型的数量。
3. 使用 kmem_zalloc(9F) 为 DDI 中断句柄分配内存。
4. 对于分配的每个中断类型，执行以下步骤：

a. 使用 ddi_intr_get_pri(9F) 获取中断的优先级。
b. 如果需要为中断设置新的优先级，请使用 ddi_intr_set_pri(9F)。
c. 使用 mutex_init(9F) 将锁初始化。
d. 使用 ddi_intr_add_handler(9F) 注册中断的处理程序。
e. 使用 ddi_intr_enable(9F) 启用中断。

5. 执行以下步骤以释放每个中断：
   a. 使用 ddi_intr_disable(9F) 禁用每个中断。
   b. 使用 ddi_intr_remove_handler(9F) 删除中断处理程序。
   c. 使用 mutex_destroy(9F) 删除锁。
   d. 使用 ddi_intr_free(9F) 和 kmem_free(9F) 释放中断，从而释放为 DDI 中断句柄分配的内存。

示例 8–5 注册传统中断

以下示例说明如何为名为 mydev 的设备安装中断处理程序。此示例假设 mydev 仅支持一个中断。

```
/* Determine which types of interrupts supported */
ret = ddi_intr_get_supported_types(mydevp->mydev_dip, &type);

if ((ret != DDI_SUCCESS) || (!(type & DDI_INTR_TYPE_FIXED))) {
    cmn_err(CE_WARN, "Fixed type interrupt is not supported");
    return (DDI_FAILURE);
}

/* Determine number of supported interrupts */
ret = ddi_intr_get_nintrs(mydevp->mydev_dip, DDI_INTR_TYPE_FIXED,
    &count);

/*
 * Fixed interrupts can only have one interrupt. Check to make
 * sure that number of supported interrupts and number of
 * available interrupts are both equal to 1.
 */
if ((ret != DDI_SUCCESS) || (count != 1)) {
    cmn_err(CE_WARN, "No fixed interrupts");
    return (DDI_FAILURE);
}

/* Allocate memory for DDI interrupt handles */
mydevp->mydev_htable = kmem_zalloc(sizeof (ddi_intr_handle_t),
    KM_SLEEP);
ret = ddi_intr_alloc(mydevp->mydev_dip, mydevp->mydev_htable,
    DDI_INTR_TYPE_FIXED, 0, count, &actual, 0);

if ((ret != DDI_SUCCESS) || (actual != 1)) {
    cmn_err(CE_WARN, "ddi_intr_alloc() failed 0x%x", ret);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
    return (DDI_FAILURE);
}
```

示例 8–5 注册传统中断 （续）

```
/* Sanity check that count and available are the same. */
ASSERT(count == actual);

/* Get the priority of the interrupt */
if (ddi_intr_get_pri(mydevp->mydev_htable[0], &mydevp->mydev_intr_pri)) {
    cmn_err(CE_WARN, "ddi_intr_alloc() failed 0x%x", ret);

    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}

cmn_err(CE_NOTE, "Supported Interrupt pri = 0x%x", mydevp->mydev_intr_pri);

/* Test for high level mutex */
if (mydevp->mydev_intr_pri >= ddi_intr_get_hilevel_pri()) {
    cmn_err(CE_WARN, "Hi level interrupt not supported");

    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}

/* Initialize the mutex */
mutex_init(&mydevp->mydev_int_mutex, NULL, MUTEX_DRIVER,
    DDI_INTR_PRI(mydevp->mydev_intr_pri));

/* Register the interrupt handler */
if (ddi_intr_add_handler(mydevp->mydev_htable[0], mydev_intr,
   (caddr_t)mydevp, NULL) !=DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_add_handler() failed");

    mutex_destroy(&mydevp->mydev_int_mutex);
    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}

/* Enable the interrupt */
if (ddi_intr_enable(mydevp->mydev_htable[0]) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_enable() failed");

    (void) ddi_intr_remove_handler(mydevp->mydev_htable[0]);
    mutex_destroy(&mydevp->mydev_int_mutex);
    (void) ddi_intr_free(mydevp->mydev_htable[0]);
    kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));

    return (DDI_FAILURE);
}
return (DDI_SUCCESS);
}
```

示例 8–6 删除传统中断

以下示例说明如何删除传统中断。

```
/* disable interrupt */
(void) ddi_intr_disable(mydevp->mydev_htable[0]);

/* Remove interrupt handler */
(void) ddi_intr_remove_handler(mydevp->mydev_htable[0]);

/* free interrupt handle */
(void) ddi_intr_free(mydevp->mydev_htable[0]);

/* free memory */
kmem_free(mydevp->mydev_htable, sizeof (ddi_intr_handle_t));
```

注册 MSI 中断

要注册驱动程序的中断处理程序，驱动程序通常会在其 attach(9E) 入口点执行以下步骤。

1. 使用 ddi_intr_get_supported_types(9F) 确定支持的中断类型。
2. 使用 ddi_intr_get_nintrs(9F) 确定支持的 MSI 中断类型的数量。
3. 使用 ddi_intr_alloc(9F) 为 MSI 中断分配内存。
4. 对于分配的每个中断类型，执行以下步骤：
   a. 使用 ddi_intr_get_pri(9F) 获取中断的优先级。
   b. 如果需要为中断设置新的优先级，请使用 ddi_intr_set_pri(9F)。
   c. 使用 mutex_init(9F) 将锁初始化。
   d. 使用 ddi_intr_add_handler(9F) 注册中断的处理程序。
5. 使用以下函数之一启用所有中断：
   - 使用 ddi_intr_block_enable(9F) 启用某个块中的所有中断。
   - 在循环中使用 ddi_intr_enable(9F) 单独启用每个中断。

示例 8–7 注册一组 MSI 中断

以下示例说明如何为名为 mydev 的设备注册 MSI 中断。

```
/* Get supported interrupt types */
if (ddi_intr_get_supported_types(devinfo, &intr_types) != DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_get_supported_types failed");
    goto attach_fail;
}

if (intr_types & DDI_INTR_TYPE_MSI)
    mydev_add_msi_intrs(mydevp);

/* Check count, available and actual interrupts */
```

示例 8–7 注册一组 MSI 中断 （续）

```
static int
mydev_add_msi_intrs(mydev_t *mydevp)
{
    dev_info_t    *devinfo = mydevp->devinfo;
    int           count, avail, actual;
    int           x, y, rc, inum = 0;

    /* Get number of interrupts */
    rc = ddi_intr_get_nintrs(devinfo, DDI_INTR_TYPE_MSI, &count);
    if ((rc != DDI_SUCCESS) || (count == 0)) {
        cmn_err(CE_WARN, "ddi_intr_get_nintrs() failure, rc: %d, "
            "count: %d", rc, count);
        return (DDI_FAILURE);
    }

    /* Get number of available interrupts */
    rc = ddi_intr_get_navail(devinfo, DDI_INTR_TYPE_MSI, &avail);
    if ((rc != DDI_SUCCESS) || (avail == 0)) {
        cmn_err(CE_WARN, "ddi_intr_get_navail() failure, "
            "rc: %d, avail: %d\n", rc, avail);
        return (DDI_FAILURE);
    }
    if (avail < count) {
        cmn_err(CE_NOTE, "nitrs() returned %d, navail returned %d",
            count, avail);
    }

    /* Allocate memory for MSI interrupts */
    mydevp->intr_size = count * sizeof (ddi_intr_handle_t);
    mydevp->htable = kmem_alloc(mydevp->intr_size, KM_SLEEP);

    rc = ddi_intr_alloc(devinfo, mydevp->htable, DDI_INTR_TYPE_MSI, inum,
        count, &actual, DDI_INTR_ALLOC_NORMAL);

    if ((rc != DDI_SUCCESS) || (actual == 0)) {
        cmn_err(CE_WARN, "ddi_intr_alloc() failed: %d", rc);
        kmem_free(mydevp->htable, mydevp->intr_size);
        return (DDI_FAILURE);
    }

    if (actual < count) {
        cmn_err(CE_NOTE, "Requested: %d, Received: %d", count, actual);
    }

    mydevp->intr_cnt = actual;
    /*
     * Get priority for first msi, assume remaining are all the same
     */
    if (ddi_intr_get_pri(mydevp->htable[0], &mydev->intr_pri) !=
        DDI_SUCCESS) {
        cmn_err(CE_WARN, "ddi_intr_get_pri() failed");

        /* Free already allocated intr */
        for (y = 0; y < actual; y++) {
            (void) ddi_intr_free(mydevp->htable[y]);
```

示例 8–7 注册一组 MSI 中断 （续）

```
        }

        kmem_free(mydevp->htable, mydevp->intr_size);
        return (DDI_FAILURE);
    }

    /* Call ddi_intr_add_handler() */
    for (x = 0; x < actual; x++) {
        if (ddi_intr_add_handler(mydevp->htable[x], mydev_intr,
           (caddr_t)mydevp, NULL) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "ddi_intr_add_handler() failed");

            /* Free already allocated intr */
            for (y = 0; y < actual; y++) {
                (void) ddi_intr_free(mydevp->htable[y]);
            }

            kmem_free(mydevp->htable, mydevp->intr_size);
            return (DDI_FAILURE);
        }
    }

    (void) ddi_intr_get_cap(mydevp->htable[0], &mydevp->intr_cap);
    if (mydev->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
        /* Call ddi_intr_block_enable() for MSI */
        (void) ddi_intr_block_enable(mydev->m_htable, mydev->m_intr_cnt);
    } else {
        /* Call ddi_intr_enable() for MSI non block enable */
        for (x = 0; x < mydev->m_intr_cnt; x++) {
            (void) ddi_intr_enable(mydev->m_htable[x]);
        }
    }
    return (DDI_SUCCESS);
}
```

示例 8–8 删除 MSI 中断

以下示例说明如何删除 MSI 中断。

```
static void
mydev_rem_intrs(mydev_t *mydev)
{
    int    x;

    /* Disable all interrupts */
    if (mydev->m_intr_cap & DDI_INTR_FLAG_BLOCK) {
        /* Call ddi_intr_block_disable() */
        (void) ddi_intr_block_disable(mydev->m_htable, mydev->m_intr_cnt);
    } else {
        for (x = 0; x < mydev->m_intr_cnt; x++) {
            (void) ddi_intr_disable(mydev->m_htable[x]);
        }
    }
```

示例 8–8 删除 MSI 中断 （续）

```
    /* Call ddi_intr_remove_handler() */
    for (x = 0; x < mydev->m_intr_cnt; x++) {
        (void) ddi_intr_remove_handler(mydev->m_htable[x]);
        (void) ddi_intr_free(mydev->m_htable[x]);
    }

    kmem_free(mydev->m_htable, mydev->m_intr_size);
}
```

中断资源管理

本节介绍可以生成多种不同可中断条件的设备驱动程序如何利用中断资源管理功能来优化其中断向量的分配。

中断资源管理功能

中断资源管理功能可动态管理驱动程序的中断配置，从而使设备驱动程序能够使用多个中断资源。未使用中断资源管理功能时，中断处理的配置通常仅在驱动程序的 `attach` (9E) 例程内进行。中断资源管理功能监控系统更改，根据这些更改重新计算分配给各设备的中断向量数量，并向各受影响的参与驱动程序发出关于驱动程序新中断向量分配情况的通知。参与驱动程序是注册了回调处理程序的驱动程序，如第 130 页中的 “回调接口”中所述。可能导致中断向量重新分配的更改包括添加或删除设备，或者显式请求，如第 133 页中的 “修改所请求的中断向量数量”中所述。

中断资源管理功能在各 Oracle Solaris 平台上不可用。此功能仅对利用 MSI-X 中断的 PCIe 设备可用。

注 – 利用中断资源管理功能的驱动程序必须能够在该功能不可用时正确做出调整。

中断资源管理功能可用时，可使驱动程序访问更多的中断向量，超过可通过其他方式为此驱动程序分配的数量。驱动程序在利用更多中断向量时可以更有效地处理中断条件。

中断资源管理功能根据以下约束动态调整分配给各参与驱动程序的中断向量数量：

- 可用总数。系统中存在的限定数量的中断向量。
- 请求的总数。可为驱动程序分配较少的中断向量，但始终不能超过所请求的中断向量数量。
- 对其他驱动程序公平。多个驱动程序以与各驱动程序请求的中断向量总数相关的方式共享可用中断向量的总数。

在任何给定时间为一个设备提供的可用中断向量的数量可能会有所不同：

- 在系统中动态添加或删除其他设备
- 驱动程序根据负荷动态更改所请求中断向量的数量

驱动程序必须提供以下支持，以利用中断资源管理功能：

- 回调支持。驱动程序必须注册回调处理程序，从而在系统改变其可用中断数量时获得通知。驱动程序必须能够增加或减少其使用的中断。
- 中断请求。驱动程序必须指定需要使用的中断数量。
- 中断用法。在任何时候，驱动程序都必须请求正确数量的中断，基于：
  - 其硬件能生成哪些可中断条件
  - 有多少处理器可用于并行处理那些条件
- 中断灵活性。驱动程序必须足够灵活，能够以最适合各中断向量的当前可用中断数量的方式为其分配一个或多个可中断的条件。在可用中断的数量增加或减少时，驱动程序可能需要随时重新配置这些分配。

回调接口

驱动程序必须使用以下接口来注册回调支持。

表 8–1 回调支持接口

接口	数据结构	说明
`ddi_cb_register()`	`ddi_cb_flags_t`、`ddi_cb_handle_t`	注册回调处理程序函数，以接收特定类型的操作。
`ddi_cb_unregister()`	`ddi_cb_handle_t`	取消注册回调处理程序函数。
`(*ddi_cb_func_t)()`	`ddi_cb_action_t`	接收回调操作以及与要处理的各操作相关的特定参数。

注册回调处理程序函数

使用 `ddi_cb_register(9F)` 函数为驱动程序注册回调处理程序函数。

```
int
ddi_cb_register (dev_info_t *dip, ddi_cb_flags_t cbflags,
                 ddi_cb_func_t cbfunc, void *arg1, void *arg2,
                 ddi_cb_handle_t *ret_hdlp);
```

驱动程序仅可注册一个回调函数。这是用于处理所有独立回调操作的回调函数。`cbflags` 参数确定驱动程序应在发生哪些类型的操作时接收这些操作。`cbfunc()` 例程将在驱动程序应处理相关操作时调用。在每次执行 `cbfunc()` 例程时，驱动程序都会指定应发送给其本身的两个专用参数（`arg1` 和 `arg2`）。

cbflags() 参数属于枚举类型，指定驱动程序支持哪些操作。

```
typedef enum {
        DDI_CB_FLAG_INTR
} ddi_cb_flags_t;
```

为了注册对中断资源管理操作的支持，驱动程序必须注册处理程序，并包含 DDI_CB_FLAG_INTR 标志。回调处理程序成功注册后，将通过 ret_hdlp 参数返回一个不透明的句柄。驱动程序使用完回调处理程序之后，驱动程序可使用 ret_hdlp 参数取消注册此回调处理程序。

在驱动程序的 attach(9F) 入口点中注册回调处理程序。在驱动程序的软状态中保存不透明的句柄。在驱动程序的 detach(9F) 入口点中取消注册回调处理程序。

取消注册回调处理程序函数

使用 ddi_cb_unregister(9F) 函数为驱动程序取消注册回调处理程序函数。

```
int
ddi_cb_unregister (ddi_cb_handle_t hdl);
```

在驱动程序的 detach(9F) 入口点中执行此调用。在此调用之后，驱动程序将不再接收回调操作。

驱动程序也会失去因拥有注册的回调处理函数而从系统中获得的其他所有支持。例如，此前为驱动程序提供的某些中断向量会在取消注册其回调处理函数后立即收回。成功返回之前，ddi_cb_unregister() 函数会通知驱动程序由于系统支持缺失所导致的任何最终操作。

回调处理程序函数

使用注册的回调处理函数来接收回调操作，接收特定于要处理的各操作的参数。

```
typedef int (*ddi_cb_func_t)(dev_info_t *dip, ddi_cb_action_t cbaction,
                             void *cbarg, void *arg1, void *arg2);
```

cbaction 参数指定驱动程序接收到的回调要处理哪种操作。

```
typedef enum {
        DDI_CB_INTR_ADD,
        DDI_CB_INTR_REMOVE
} ddi_cb_action_t;
```

DDI_CB_INTR_ADD 操作表示驱动程序中断可用数量增加。DDI_CB_INTR_REMOVE 操作表示驱动程序中断可用数量减少。将 cbarg 参数的类型强制转换为 int，以确定添加或删除的中断数量。cbarg 值表示可用中断数量的变化。

例如，获得可用中断数量的变化：

```
count = (int)(uintptr_t)cbarg;
```

如果 cbaction 为 DDI_CB_INT _ADD，则应添加 cbarg 数量的中断向量。如果 cbaction 为 DDI_CB_INT _REMOVE，则应释放 cbarg 数量的中断向量。

有关 arg1 和 arg2 的说明，请参见 ddi_cb_register(9F)。

回调处理函数必须能够在函数注册的整个时间段内正确执行。回调函数不能依赖任何可能会在回调函数成功取消注册之前销毁的数据结构。

回调处理函数必须返回以下值之一：

- DDI_SUCCESS（如果正确处理了操作）
- DDI_FAILURE（如果遇到内部错误）
- DDI_ENOTSUP（如果接收到无法识别的操作）

中断请求接口

驱动程序必须使用以下接口从系统请求中断向量。

表 8–2 中断向量请求接口

接口	数据结构	说明
ddi_intr_alloc()	ddi_intr_handle_t	分配中断。
ddi_intr_set_nreq()		更改所请求的中断向量数量。

分配中断

使用 ddi_intr_alloc(9F) 函数来初始分配中断。

```
int
ddi_intr_alloc (dev_info_t *dip, ddi_intr_handle_t *h_array, int type,
                int inum, int count, int *actualp, int behavior);
```

调用此函数之前，驱动程序必须分配一个可包含所请求中断数量的足够大的空句柄数组。ddi_intr_alloc() 函数会尝试分配 count 数量的中断句柄，并使用以 inum 参数指定的偏移开头的指定中断向量初始化数组。 actualp 参数返回所分配的中断向量的实际数量。

驱动程序可通过两种方式使用 ddi_intr_alloc() 函数：

- 驱动程序可以在单独的步骤中多次调用 ddi_intr_alloc() 函数，为中断句柄数组中的各成员分配中断向量。
- 驱动程序可以一次性地调用 ddi_intr_alloc() 函数，一次为设备分配所有中断向量。

如果您正在使用中断资源管理功能，则调用一次 ddi_intr_alloc() 即可分配所有中断向量。count 参数是驱动程序请求的中断向量的总数。如果 actualp 中的值小于 count

值，则系统无法完全满足请求。中断资源管理功能将保存此请求（count 转为·nreq - 请参见下文），稍后还可能会为此驱动程序分配更多中断向量。

注 – 使用中断资源管理功能时，对 ddi_intr_alloc() 的其他调用不会更改请求的中断向量总数。使用 ddi_intr_set_nreq(9F) 函数更改请求的中断向量数量。

修改所请求的中断向量数量

使用 ddi_intr_set_nreq(9F) 函数修改请求的中断向量数量。

```
int
ddi_intr_set_nreq (dev_info_t *dip, int nreq);
```

中断资源管理功能可用时，驱动程序可使用 ddi_intr_set_nreq() 函数动态调整请求的中断向量总数。附加了驱动程序之后，驱动程序可能会以此响应存在的实际负载。

驱动程序必须首先调用 ddi_intr_alloc(9F) 来请求初始数量的中断向量。完成 ddi_intr_alloc() 调用之后，驱动程序可随时调用 ddi_intr_set_nreq() 来更改请求的大小。指定的 nreq 值是驱动程序请求的中断向量的新总数。中断资源管理功能可能会根据这个新请求重新平衡系统中分配给各驱动程序的中断数量。只要中断资源管理功能重新平衡了分配给驱动程序的中断数量，各受影响的驱动程序就会接收到该驱动程序可以使用的中断向量增加或减少的回调通知。

例如，如果驱动程序将中断与其处理的特定事务并行使用，则可动态调整所请求中断向量的总数。存储驱动程序必须将 DMA 引擎与各进行中的事务相关联，因而需要中断向量。驱动程序可在 open(9F) 和 close(9F) 例程中调用 ddi_intr_set_nreq()，以便根据驱动程序的实际使用情况调整中断使用的比例。

中断用法和灵活性

支持多种不同可中断条件的设备的驱动程序必须能够将这些条件映射到任意数量的中断向量。驱动程序不能假设已经分配的中断向量仍然可用。某些当前可用中断稍后可能会被系统收回，以满足系统中其他驱动程序的需求。

驱动程序必须能够：

- 确定其硬件支持的中断数量。
- 确定适合使用的中断数量。例如，系统中的处理器总数可能会影响此项评估。
- 随时将所需中断的数量与可用中断的数量相比较。

总而言之，驱动程序必须能够选择一系列中断处理函数，并为其硬件编程，使之能够根据需求和中断可用性生成中断。在某些情况下，可能有多个针对同一个向量的中断，该中断向量的中断处理程序必须确定发生的是哪个中断。驱动程序将中断映射到中断向量的情况可能会影响设备性能。

中断资源管理实现样例

网络设备驱动程序是中断资源管理的一种理想的备选设备驱动程序类型。网络设备硬件支持多个传输和接收信道。

网络设备在一个接收信道接收到包或者在一个传输信道传输包时，设备将生成唯一的中断条件。硬件可以为可能发生的各事件发送特定的 MSI-X 中断。硬件中的表格确定为各事件生成哪个 MSI-X 中断。

为了优化性能，驱动程序会向系统请求足够多的中断，以使每个中断都有自己的中断向量。在 attach(9F) 例程中初次调用 ddi_intr_alloc(9F) 时，驱动程序会发出此请求。

随后，驱动程序评估通过 actualp 中的 ddi_intr_alloc() 接收到的实际中断数量。 可能会接收到所请求的全部中断，也可能会接收到较少的中断。

驱动程序内的独立函数使用可用中断总数计算为各事件生成哪些 MSI-X 中断。此函数会相应在硬件中对该表进行编程。

- 如果驱动程序接收了全部请求的中断向量，硬件表中的每个条目都将有自己唯一的 MSI-X 中断。中断条件和中断向量之间存在一对一的映射。硬件为每种类型的事件生成唯一的 MSI-X 中断。
- 如果驱动程序可用的中断向量更少，则硬件表中必定要多次出现某些 MSI-X 中断数。硬件为多种类型的事件生成相同的 MSI-X 中断。

驱动程序应有两个不同的中断处理程序函数。

- 一个中断处理程序执行特定任务来响应一项中断。这个简单的函数仅可处理由一种可能硬件事件所生成的中断。
- 第二个中断处理程序更为复杂。此函数用于处理有多个中断映射到同一个 MSI-X 中断向量的情况。

在这一部分的示例驱动程序中，xx_setup_interrupts() 函数使用可用中断向量的数量来为硬件编程，并为其中每一个中断向量调用相应的中断处理程序。将在两个位置调用 xx_setup_interrupts() 函数：在 xx_attach() 中调用 di_intr_alloc() 之后，在 xx_cbfunc() 回调处理程序函数中调整了中断向量分配之后。

```
int
xx_setup_interrupts(xx_state_t *statep, int navail, xx_intrs_t *xx_intrs_p);
```

xx_setup_interrupts() 函数是通过 xx_intrs_t 数据结构的数组调用的。

```
typedef struct {
        ddi_intr_handler_t      inthandler;
        void                    *arg1;
        void                    *arg2;
} xx_intrs_t;
```

无论中断资源管理功能是否可用，驱动程序中都必须存在这种 xx_setup_interrupts() 功能。驱动程序必须能够在中断向量少于附加过程中请求的数量时正常工作。如果中断资源管理功能可用，您就可以修改驱动程序，使其动态适应新的可用中断向量数量。

驱动程序必须独立于中断资源管理功能的可用性提供的其他功能，包括停止硬件和恢复硬件的能力。某些与电源管理和热插拔相关的事件需要停止和恢复。在处理中断回调操作时也必须利用停止和恢复。

在 xx_detach() 中调用了停止函数。

```
int
xx_quiesce(xx_state_t *statep);
```

在 xx_attach() 中调用了恢复函数。

```
int
xx_resume(xx_state_t *statep);
```

通过以下修改增强此设备驱动程序，使其使用中断资源管理功能：

- 注册回调处理程序。驱动程序必须为指明何时将有更少或更多的中断可用的操作注册。
- 处理回调。驱动程序必须停止其硬件、重新编程中断处理、恢复硬件以响应各个此类回调操作。

```
/*
 * attach(9F) routine.
 *
 * Creates soft state, registers callback handler, initializes
 * hardware, and sets up interrupt handling for the driver.
 */
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        xx_state_t              *statep = NULL;
        xx_intr_t               *intrs = NULL;
        ddi_intr_handle_t       *hdls;
        ddi_cb_handle_t         cb_hdl;
        int                     instance;
        int                     type;
        int                     types;
        int                     nintrs;
        int                     nactual;
        int                     inum;

        /* Get device instance */
        instance = ddi_get_instance(dip);

        switch (cmd) {
        case DDI_ATTACH:

                /* Get soft state */
                if (ddi_soft_state_zalloc(state_list, instance) != 0)
```

```
                return (DDI_FAILURE);
        statep = ddi_get_soft_state(state_list, instance);
        ddi_set_driver_private(dip, (caddr_t)statep);
        statep->dip = dip;

        /* Initialize hardware */
        xx_initialize(statep);

        /* Register callback handler */
        if (ddi_cb_register(dip, DDI_CB_FLAG_INTR, xx_cbfunc,
            statep, NULL, &cb_hdl) != 0) {
                ddi_soft_state_free(state_list, instance);
                return (DDI_FAILURE);
        }
        statep->cb_hdl = cb_hdl;

        /* Select interrupt type */
        ddi_intr_get_supported_types(dip, &types);
        if (types & DDI_INTR_TYPE_MSIX) {
                type = DDI_INTR_TYPE_MSIX;
        } else if (types & DDI_INTR_TYPE_MSI) {
                type = DDI_INTR_TYPE_MSI;
        } else {
                type = DDI_INTR_TYPE_FIXED;
        }
        statep->type = type;

        /* Get number of supported interrupts */
        ddi_intr_get_nintrs(dip, type, &nintrs);

        /* Allocate interrupt handle array */
        statep->hdls_size = nintrs * sizeof (ddi_intr_handle_t);
        statep->hdls = kmem_zalloc(statep->hdls_size, KMEM_SLEEP);

        /* Allocate interrupt setup array */
        statep->intrs_size = nintrs * sizeof (xx_intr_t);
        statep->intrs = kmem_zalloc(statep->intrs_size, KMEM_SLEEP);

        /* Allocate interrupt vectors */
        ddi_intr_alloc(dip, hdls, type, 0, nintrs, &nactual, 0);
        statep->nactual = nactual;

        /* Configure interrupt handling */
        xx_setup_interrupts(statep, statep->nactual, statep->intrs);

        /* Install and enable interrupt handlers */
        for (inum = 0; inum < nactual; inum++) {
                ddi_intr_add_handler(&hdls[inum],
                    intrs[inum].inthandler,
                    intrs[inum].arg1, intrs[inum].arg2);
                ddi_intr_enable(hdls[inum]);
        }

        break;

case DDI_RESUME:

        /* Get soft state */
        statep = ddi_get_soft_state(state_list, instance);
```

```
                if (statep == NULL)
                        return (DDI_FAILURE);

                /* Resume hardware */
                xx_resume(statep);

                break;
        }

        return (DDI_SUCESS);
}

/*
 * detach(9F) routine.
 *
 * Stops the hardware, disables interrupt handling, unregisters
 * a callback handler, and destroys the soft state for the driver.
 */
xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        xx_state_t      *statep = NULL;
        int             instance;
        int             inum;

        /* Get device instance */
        instance = ddi_get_instance(dip);

        switch (cmd) {
        case DDI_DETACH:

                /* Get soft state */
                statep = ddi_get_soft_state(state_list, instance);
                if (statep == NULL)
                        return (DDI_FAILURE);

                /* Stop device */
                xx_uninitialize(statep);

                /* Disable and free interrupts */
                for (inum = 0; inum < statep->nactual; inum++) {
                        ddi_intr_disable(statep->hdls[inum]);
                        ddi_intr_remove_handler(statep->hdls[inum]);
                        ddi_intr_free(statep->hdls[inum]);
                }

                /* Unregister callback handler */
                ddi_cb_unregister(statep->cb_hdl);

                /* Free interrupt handle array */
                kmem_free(statep->hdls, statep->hdls_size);

                /* Free interrupt setup array */
                kmem_free(statep->intrs, statep->intrs_size);

                /* Free soft state */
                ddi_soft_state_free(state_list, instance);

                break;
```

```
        case DDI_SUSPEND:

                /* Get soft state */
                statep = ddi_get_soft_state(state_list, instance);
                if (statep == NULL)
                        return (DDI_FAILURE);

                /* Suspend hardware */
                xx_quiesce(statep);

                break;
        }

        return (DDI_SUCCESS);
}

/*
 * (*ddi_cbfunc)() routine.
 *
 * Adapt interrupt usage when availability changes.
 */
int
xx_cbfunc(dev_info_t *dip, ddi_cb_action_t cbaction, void *cbarg,
    void *arg1, void *arg2)
{
        xx_state_t      *statep = (xx_state_t *)arg1;
        int             count;
        int             inum;
        int             nactual;

        switch (cbaction) {
        case DDI_CB_INTR_ADD:
        case DDI_CB_INTR_REMOVE:

                /* Get change in availability */
                count = (int)(uintptr_t)cbarg;

                /* Suspend hardware */
                xx_quiesce(statep);

                /* Tear down previous interrupt handling */
                for (inum = 0; inum < statep->nactual; inum++) {
                        ddi_intr_disable(statep->hdls[inum]);
                        ddi_intr_remove_handler(statep->hdls[inum]);
                }

                /* Adjust interrupt vector allocations */
                if (cbaction == DDI_CB_INTR_ADD) {

                        /* Allocate additional interrupt vectors */
                        ddi_intr_alloc(dip, statep->hdls, statep->type,
                            statep->nactual, count, &nactual, 0);

                        /* Update actual count of available interrupts */
                        statep->nactual += nactual;

                } else {

                        /* Free removed interrupt vectors */
```

```
                        for (inum = statep->nactual - count;
                            inum < statep->nactual; inum++) {
                                ddi_intr_free(statep->hdls[inum]);
                        }

                        /* Update actual count of available interrupts */
                        statep->nactual -= count;
                }

                /* Configure interrupt handling */
                xx_setup_interrupts(statep, statep->nactual, statep->intrs);

                /* Install and enable interrupt handlers */
                for (inum = 0; inum < statep->nactual; inum++) {
                        ddi_intr_add_handler(&statep->hdls[inum],
                            statep->intrs[inum].inthandler,
                            statep->intrs[inum].arg1,
                            statep->intrs[inum].arg2);
                        ddi_intr_enable(statep->hdls[inum]);
                }

                /* Resume hardware */
                xx_resume(statep);

                break;

        default:
                return (DDI_ENOTSUP);
        }

        return (DDI_SUCCESS);
}
```

中断处理程序功能

驱动程序框架和设备各自将要求置于中断处理程序上。所有中断处理程序均要求执行以下任务：

- **确定设备是否会中断并可能相应地拒绝此中断。**

  中断处理程序首先会检查设备，确定其是否发出了中断。如果设备未发出中断，则处理程序必须返回 DDI_INTR_UNCLAIMED。通过此步骤可实现**设备轮询**。在给定中断优先级别的任何设备都可能发出了中断。设备轮询将通知系统此设备是否已发出了中断。

- **通知设备正在对其进行服务。**

  通知设备服务是大多数设备所需的特定于设备的操作。例如，需要将 S 总线设备中断，直到驱动程序通知 S 总线设备停止。此方法可保证对在同一优先级别中断的所有 S 总线设备都进行服务。

- **执行任何与 I/O 请求有关的处理。**

  设备会由于不同原因而发生中断，如**传送完成**或**传送错误**。此步骤可涉及使用数据访问函数来读取设备的数据缓冲区，检查设备的错误寄存器，以及在数据结构中相应地设置状态字段。中断分发和处理相对比较耗时。

- **执行可以防止其他中断的任何附加处理。**

  例如，从设备中读取数据的下一项。
- **返回** `DDI_INTR_CLAIMED`。
- **必须始终声明 MSI 中断。**

  对于 MSI-X 中断，声明中断是可选的。在任一情况下都无需检查中断的拥有权，因为 MSI 和 MSI-X 中断不是与其他设备共享的。
- **支持热插拔和多个 MSI 或 MSI-X 中断的驱动程序应针对热插拔事件保留单独的中断，并针对此中断注册单独的 ISR（interrupt service routine，中断服务例程）。**

以下示例说明了名为 `mydev` 的设备的中断例程。

示例 8–9 中断示例

```
static uint_t
mydev_intr(caddr_t arg1, caddr_t arg2)
{
    struct mydevstate *xsp = (struct mydevstate *)arg1;
    uint8_t     status;
    volatile    uint8_t  temp;

    /*
     * Claim or reject the interrupt.This example assumes
     * that the device's CSR includes this information.
     */
    mutex_enter(&xsp->high_mu);

    /* use data access routines to read status */
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->high_mu);
        return (DDI_INTR_UNCLAIMED); /* dev not interrupting */
    }
    /*
     * Inform the device that it is being serviced, and re-enable
     * interrupts. The example assumes that writing to the
     * CSR accomplishes this. The driver must ensure that this data
     * access operation makes it to the device before the interrupt
     * service routine returns. For example, using the data access
     * functions to read the CSR, if it does not result in unwanted
     * effects, can ensure this.
     */
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
        CLEAR_INTERRUPT | ENABLE_INTERRUPTS);

    /* flush store buffers */
    temp = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);

    mutex_exit(&xsp->mu);
    return (DDI_INTR_CLAIMED);
}
```

中断例程执行的大多数步骤都依赖于设备本身的特定信息。查询设备的硬件手册可确定中断原因，检测错误状态并访问设备数据寄存器。

处理高级别中断

高级别中断是指中断在调度程序级别或更高级别的那类中断。此级别不允许运行调度程序。因此，调度程序无法抢占高级别中断处理程序。高级别中断不会因为调度程序而阻塞。高级别中断只能使用互斥锁进行锁定。

驱动程序必须确定设备是否在使用高级别中断。注册中断时，请在驱动程序的 attach(9E) 入口点进行此测试。请参见第 141 页中的 “高级别中断处理示例”。

- 如果从 ddi_intr_get_pri(9F) 返回的中断优先级高于或等于从 ddi_intr_get_hilevel_pri(9F) 返回的优先级，则表明驱动程序无法连接，或者驱动程序可能实现高级别的中断处理程序。高级别的中断处理程序可使用优先级较低的软件中断来处理该设备。要允许更大的并发性，请使用单独的互斥锁来防止高级中断处理程序使用数据。
- 如果从 ddi_intr_get_pri(9F) 返回的中断优先级低于从 ddi_intr_get_hilevel_pri(9F) 返回的优先级，则 attach(9E) 入口点将进行常规中断注册。在这种情况下不需要软中断。

高级互斥锁

使用表示高级别中断的中断优先级初始化的互斥锁称为**高级互斥锁**。虽然持有高级互斥锁，但是驱动程序仍会受到与高级别中断处理程序相同的限制。

高级别中断处理示例

在以下示例中，高级互斥锁 (xsp->high_mu) 仅用于保护在高级别中断处理程序和软中断处理程序之间共享的数据。受保护的数据包括高级别中断处理程序和低级处理程序使用的队列，以及用于指示低级处理程序正在运行的标志。单独的低级互斥锁 (xsp->low_mu) 可防止软中断处理程序使用驱动程序的其余部分。

示例 8–10 使用 attach() 处理高级别中断

```
static int
mydevattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct mydevstate *xsp;
    /* ... */

    ret = ddi_intr_get_supported_types(dip, &type);
    if ((ret != DDI_SUCCESS) || (!(type & DDI_INTR_TYPE_FIXED))) {
        cmn_err(CE_WARN, "ddi_intr_get_supported_types() failed");
        return (DDI_FAILURE);
    }

    ret = ddi_intr_get_nintrs(dip, DDI_INTR_TYPE_FIXED, &count);
```

示例 8–10 使用 attach() 处理高级别中断 （续）

```
/*
 * Fixed interrupts can only have one interrupt. Check to make
 * sure that number of supported interrupts and number of
 * available interrupts are both equal to 1.
 */
if ((ret != DDI_SUCCESS) || (count != 1)) {
cmn_err(CE_WARN, "No fixed interrupts found");
        return (DDI_FAILURE);
}

xsp->xs_htable = kmem_zalloc(count * sizeof (ddi_intr_handle_t),
    KM_SLEEP);

ret = ddi_intr_alloc(dip, xsp->xs_htable, DDI_INTR_TYPE_FIXED, 0,
    count, &actual, 0);

if ((ret != DDI_SUCCESS) || (actual != 1)) {
cmn_err(CE_WARN, "ddi_intr_alloc failed 0x%x", ret");
    kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
    return (DDI_FAILURE);
}

ret = ddi_intr_get_pri(xsp->xs_htable[0], &intr_pri);
if (ret != DDI_SUCCESS) {
    cmn_err(CE_WARN, "ddi_intr_get_pri failed 0x%x", ret");
    (void) ddi_intr_free(xsp->xs_htable[0]);
    kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
    return (DDI_FAILURE);
}

if (intr_pri >= ddi_intr_get_hilevel_pri()) {

    mutex_init(&xsp->high_mu, NULL, MUTEX_DRIVER,
        DDI_INTR_PRI(intr_pri));

    ret = ddi_intr_add_handler(xsp->xs_htable[0],
        mydevhigh_intr, (caddr_t)xsp, NULL);

    if (ret != DDI_SUCCESS) {
        cmn_err(CE_WARN, "ddi_intr_add_handler failed 0x%x", ret");
        mutex_destroy(&xsp>xs_int_mutex);
            (void) ddi_intr_free(xsp->xs_htable[0]);
            kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
        return (DDI_FAILURE);
    }

    /* add soft interrupt */
    if (ddi_intr_add_softint(xsp->xs_dip, &xsp->xs_softint_hdl,
        DDI_INTR_SOFTPRI_MAX, xs_soft_intr, (caddr_t)xsp) !=
        DDI_SUCCESS) {
        cmn_err(CE_WARN, "add soft interrupt failed");
        mutex_destroy(&xsp->high_mu);
        (void) ddi_intr_remove_handler(xsp->xs_htable[0]);
        (void) ddi_intr_free(xsp->xs_htable[0]);
        kmem_free(xsp->xs_htable, sizeof (ddi_intr_handle_t));
        return (DDI_FAILURE);
```

示例 8–10 使用 attach() 处理高级别中断 （续）

```
        }

        xsp->low_soft_pri = DDI_INTR_SOFTPRI_MAX;

        mutex_init(&xsp->low_mu, NULL, MUTEX_DRIVER,
            DDI_INTR_PRI(xsp->low_soft_pri));

    } else {
    /*
     * regular interrupt registration continues from here
     * do not use a soft interrupt
     */
    }

    return (DDI_SUCCESS);
}
```

高级别中断例程用于服务设备并对数据进行排队。如果低级例程未运行，则高级例程会触发软件中断，如以下示例所示。

示例 8–11 高级别中断例程

```
static uint_t
mydevhigh_intr(caddr_t arg1, caddr_t arg2)
{
    struct mydevstate    *xsp = (struct mydevstate *)arg1;
    uint8_t    status;
    volatile  uint8_t  temp;
    int    need_softint;

    mutex_enter(&xsp->high_mu);
    /* read status */
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->high_mu);
        return (DDI_INTR_UNCLAIMED); /* dev not interrupting */
    }

    ddi_put8(xsp->data_access_handle,&xsp->regp->csr,
        CLEAR_INTERRUPT | ENABLE_INTERRUPTS);
    /* flush store buffers */
    temp = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);

    /* read data from device, queue data for low-level interrupt handler */
    if (xsp->softint_running)
        need_softint = 0;
    else {
        xsp->softint_count++;
        need_softint = 1;
    }
    mutex_exit(&xsp->high_mu);

    /* read-only access to xsp->id, no mutex needed */
    if (need_softint) {
```

示例 8–11 高级别中断例程 （续）

```
        ret = ddi_intr_trigger_softint(xsp->xs_softint_hdl, NULL);
        if (ret == DDI_EPENDING) {
            cmn_err(CE_WARN, "ddi_intr_trigger_softint() soft interrupt "
                "already pending for this handler");
        } else if (ret != DDI_SUCCESS) {
            cmn_err(CE_WARN, "ddi_intr_trigger_softint() failed");
        }
    }

    return (DDI_INTR_CLAIMED);
}
```

低级中断例程由用于触发软件中断的高级别中断例程启动。低级中断例程会一直运行，直到没有其他要处理的对象为止，如以下示例所示。

示例 8–12 低级软中断例程

```
static uint_t
mydev_soft_intr(caddr_t arg1, caddr_t arg2)
{
    struct mydevstate *mydevp = (struct mydevstate *)arg1;
    /* ... */
    mutex_enter(&mydevp->low_mu);
    mutex_enter(&mydevp->high_mu);
    if (mydevp->softint_count > 1) {
        mydevp->softint_count--;
        mutex_exit(&mydevp->high_mu);
        mutex_exit(&mydevp->low_mu);
        return (DDI_INTR_CLAIMED);
    }

    if ( /* queue empty */ ) {
        mutex_exit(&mydevp->high_mu);
        mutex_exit(&mydevp->low_mu);
        return (DDI_INTR_UNCLAIMED);
    }

    mydevp->softint_running = 1;
    while (EMBEDDED COMMENT:data on queue) {
        ASSERT(mutex_owned(&mydevp->high_mu);
        /* Dequeue data from high-level queue. */
        mutex_exit(&mydevp->high_mu);
        /* normal interrupt processing */
        mutex_enter(&mydevp->high_mu);
    }

    mydevp->softint_running = 0;
    mydevp->softint_count = 0;
    mutex_exit(&mydevp->high_mu);
    mutex_exit(&mydevp->low_mu);
    return (DDI_INTR_CLAIMED);
}
```

◆ ◆ ◆ 第 9 章

9 直接内存访问 (Direct Memory Access, DMA)

许多设备都可以临时控制总线。这些设备可以执行涉及主内存和其他设备的数据传送。由于设备执行这些操作的过程中无需借助于 CPU，因此该类型的数据传送称为**直接内存访问** (direct memory access, DMA)。可以执行的 DMA 传送类型如下：

- 两个设备之间
- 设备和内存之间
- 内存和内存之间

本章仅介绍设备和内存之间的传送。本章提供有关以下主题的信息：

DMA 模型

Oracle Solaris 设备驱动程序接口/驱动程序内核接口 (Device Driver Interface/Driver-Kernel Interface, DDI/DKI) 为 DMA 提供了独立于体系结构的高级别模型。通过此模型，框架（即 DMA 例程）可以隐藏此类体系结构特定的详细信息，例如：

- 设置 DMA 映射
- 生成分散/集中列表
- 确保 I/O 和 CPU 高速缓存一致

DDI/DKI 中使用了若干个抽象术语来描述 DMA 事务的各个方面：

- **DMA 对象** – 作为 DMA 传送的源或目标的内存。

- **DMA 句柄**－成功调用 ddi_dma_alloc_handle(9F) 后返回的不透明对象。在后续的 DMA 子例程调用中可以使用 DMA 句柄来引用此类 DMA 对象。
- **DMA cookie**－ddi_dma_cookie(9S) 结构 (ddi_dma_cookie_t) 用于描述 DMA 对象中可由设备完全寻址的连续部分。该 cookie 包含对 DMA 引擎进行编程所需的 DMA 寻址信息。

设备驱动程序不会直接将对象映射到内存中，而是为内存对象分配 DMA **资源**。然后，DMA 例程将执行为 DMA 访问设置对象时所需的任何特定于平台的操作。驱动程序将收到一个 DMA **句柄**，用于标识为该对象分配的 DMA 资源。此句柄对于设备驱动程序而言是不透明的。驱动程序必须保存句柄并在后续调用中将其传递给 DMA 例程。驱动程序不应以任何方式解释句柄。

针对 DMA 句柄定义的操作可提供以下服务：

- 处理 DMA 资源
- 同步 DMA 对象
- 检索已分配资源的特性

设备 DMA 的类型

设备可执行以下三种类型的 DMA：

- 总线主控器 DMA
- 第三方 DMA
- 第一方 DMA

总线主控器 DMA

在设备用作实际**总线主控器**的情况下，驱动程序应直接对该设备的 DMA 寄存器进行编程。例如，如果 DMA 引擎驻留在设备板上，则设备便会用作总线主控器。传送地址和计数从 DMA cookie 中获取，并将传递给设备。

第三方 DMA

第三方 DMA 使用驻留在主系统板上的系统 DMA 引擎，该引擎中有若干个可供设备使用的 DMA 通道。设备依赖于系统的 DMA 引擎来执行设备与内存之间的数据传送。驱动程序使用 DMA 引擎例程（请参见 ddi_dmae(9F) 函数）对 DMA 引擎进行初始化和编程。每次进行 DMA 数据传送时，驱动程序都会对 DMA 引擎进行编程，然后会向设备发出命令，以便借助该引擎来启动传送操作。

第一方 DMA

执行第一方 DMA 时，设备使用系统 DMA 引擎中的通道来驱动该设备的 DMA 总线循环。使用 ddi_dmae_1stparty(9F) 函数可在级联模式下对此通道进行配置，以免 DMA 引擎干扰传送。

主机平台 DMA 的类型

设备运行的平台可提供直接内存访问 (direct memory access, DMA) 或直接虚拟内存访问 (direct virtual memory access, DVMA)。

在支持 DMA 的平台上，系统会为设备提供物理地址以执行传送。在此情况下，DMA 对象的传送实际上会包含许多在物理上不连续的传送。例如，当应用程序传送跨越若干连续虚拟页（但这些虚拟页映射到物理上不连续的页）的缓冲区时。要处理不连续的内存，用于这些平台的设备通常需要具有特定种类的分散/集中 DMA 功能。通常，x86 系统会为直接内存传送提供物理地址。

在支持 DVMA 的平台上，系统会为设备提供虚拟地址以执行传送。在此情况下，基础平台提供的内存管理单元 (memory management unit, MMU) 会将对这些虚拟地址的设备访问转换为正确的物理地址。设备会与可映射到不连续物理页的连续虚拟映像之间来回进行传送。在这些平台上运行的设备无需分散/集中 DMA 功能。通常，SPARC 平台会为直接内存传送提供虚拟地址。

DMA 软件组件：句柄、窗口和 Cookie

DMA **句柄**是表示对象（通常为内存缓冲区或地址）的不透明指针。设备通过 DMA 句柄可执行 DMA 传送。对 DMA 例程的若干个不同调用可使用句柄来标识为对象分配的 DMA 资源。

DMA 句柄所表示的对象全部包含在一个或多个 *DMA cookie* 中。DMA cookie 表示 DMA 引擎在数据传送中使用的一段连续内存。系统会根据以下信息将对象划分为多个 cookie：

- 驱动程序提供的 ddi_dma_attr(9S) 特性结构
- 目标对象的内存位置
- 目标对象的对齐

如果一个对象不满足 DMA 引擎的限制，则必须将该对象分为多个 *DMA* **窗口**。一次只能为一个窗口激活和分配资源。使用 ddi_dma_getwin(9F) 函数可在一个对象内的多个窗口之间切换。每个 DMA 窗口都包含一个或多个 DMA cookie。有关更多信息，请参见第 163 页中的 “DMA 窗口”。

某些 DMA 引擎可以接受多个 cookie。此类引擎不用借助系统即可执行分散/集中 I/O。如果从一个绑定中返回多个 cookie，则驱动程序应重复调用

ddi_dma_nextcookie(9F) 以检索每个 cookie。然后，必须将这些 cookie 编程到引擎中。随后可对设备进行编程，以传送这些 DMA cookie 聚集所包含的总字节数。

DMA 操作

不同类型的 DMA 之间，DMA 传送的步骤都相似。以下各节提供了执行 DMA 传送的方法。

注 – 在来自文件系统的缓冲区的块驱动程序中，不必确保 DMA 对象是否已在内存中锁定。该文件系统已在内存中锁定了数据。

执行总线主控器 DMA 传送

对于总线主控器 DMA，驱动程序应执行以下步骤：

1. 描述 DMA 特性。通过此步骤，例程可确保设备能够访问缓冲区。
2. 分配 DMA 句柄。
3. 确保 DMA 对象已在内存中锁定。请参见 physio(9F) 或 ddi_umem_lock(9F) 手册页。
4. 为该对象分配 DMA 资源。
5. 对设备的 DMA 引擎进行编程。
6. 启动引擎。
7. 传送完成后，继续执行总线主控器操作。
8. 执行所需的对象同步。
9. 释放 DMA 资源。
10. 释放 DMA 句柄。

执行第一方 DMA 传送

对于第一方 DMA，驱动程序应执行以下步骤：

1. 分配 DMA 通道。
2. 使用 ddi_dmae_1stparty(9F) 配置通道。
3. 确保 DMA 对象已在内存中锁定。请参见 physio(9F) 或 ddi_umem_lock(9F) 手册页。
4. 为该对象分配 DMA 资源。
5. 对设备的 DMA 引擎进行编程。
6. 启动引擎。
7. 传送完成后，继续执行总线主控器操作。
8. 执行所需的对象同步。
9. 释放 DMA 资源。
10. 取消分配 DMA 通道。

执行第三方 DMA 传送

对于第三方 DMA，驱动程序应执行以下步骤：

1. 分配 DMA 通道。
2. 使用 ddi_dmae_getattr(9F) 检索系统的 DMA 引擎特性。
3. 在内存中锁定 DMA 对象。请参见 physio(9F) 或 ddi_umem_lock(9F) 手册页。
4. 为该对象分配 DMA 资源。
5. 使用 ddi_dmae_prog(9F) 对系统 DMA 引擎进行编程，以执行传送。
6. 执行所需的对象同步。
7. 使用 ddi_dmae_stop(9F) 停止 DMA 引擎。
8. 释放 DMA 资源。
9. 取消分配 DMA 通道。

某些硬件平台会以特定于总线的方式限制 DMA 功能。驱动程序应使用 ddi_slaveonly(9F) 来确定设备是否位于可以执行 DMA 的插槽中。

DMA 特性

DMA 特性描述 DMA 引擎的特性和限制，其中包括：

- 设备可以访问的地址的限制
- 最大传送计数
- 地址对齐限制

设备驱动程序必须通过 ddi_dma_attr(9S) 结构向系统通知任何 DMA 引擎限制。此操作可以确保设备的 DMA 引擎可以访问系统分配的 DMA 资源。系统可能对设备特性实施附加限制，但绝不会取消驱动程序实施的任何限制。

ddi_dma_attr 结构

DMA 特性结构包含以下成员：

```
typedef struct ddi_dma_attr {
    uint_t      dma_attr_version;       /* version number */
    uint64_t    dma_attr_addr_lo;       /* low DMA address range */
    uint64_t    dma_attr_addr_hi;       /* high DMA address range */
    uint64_t    dma_attr_count_max;     /* DMA counter register */
    uint64_t    dma_attr_align;         /* DMA address alignment */
    uint_t      dma_attr_burstsizes;    /* DMA burstsizes */
    uint32_t    dma_attr_minxfer;       /* min effective DMA size */
    uint64_t    dma_attr_maxxfer;       /* max DMA xfer size */
    uint64_t    dma_attr_seg;           /* segment boundary */
    int         dma_attr_sgllen;        /* s/g length */
    uint32_t    dma_attr_granular;      /* granularity of device */
    uint_t      dma_attr_flags;         /* Bus specific DMA flags */
} ddi_dma_attr_t;
```

其中：

dma_attr_version	特性结构的版本号。dma_attr_version 应设置为 DMA_ATTR_V0。
dma_attr_addr_lo	DMA 引擎可以访问的最低总线地址。
dma_attr_addr_hi	DMA 引擎可以访问的最高总线地址。
dma_attr_count_max	指定 DMA 引擎可在一个 cookie 中处理的最大传送计数。该限制表示为最大计数减 1。此计数用作位掩码，因此计数也必须比 2 的幂小 1。
dma_attr_align	指定通过 ddi_dma_mem_alloc(9F) 分配内存时的对齐要求。例如，以页边界对齐。dma_attr_align 字段仅在分配内存时使用。在绑定操作过程中，将省略此字段。对于绑定操作，驱动程序必须确保缓冲区已正确对齐。
dma_attr_burstsizes	指定设备支持的**突发流量大小**。突发流量大小是指设备在放弃总线之前可以传输的数据量。此成员是突发流量大小的二进制编码，这些大小是 2 的幂次方。例如，如果设备能够执行 1 字节、2 字节、4 字节和 16 字节突发传输，则应将此字段设置为 0x17。系统还会使用此字段来确定对齐限制。
dma_attr_minxfer	设备可以执行的最小有效传送大小。此大小还会影响对齐和填充的限制。
dma_attr_maxxfer	描述 DMA 引擎在一个 I/O 命令中可以容纳的最大字节数。此限制仅在 dma_attr_maxxfer 小于 (dma_attr_count_max + 1) * dma_attr_sgllen 时才有意义。
dma_attr_seg	DMA 引擎的地址寄存器的上限。当地址寄存器的高 8 位为包含段号的锁存器时，通常会使用 dma_attr_seg 。低 24 位用于寻址段。在此情况下，dma_attr_seg 会设置为 0xFFFFFF，这可以防止系统在为对象分配资源时跨越 24 位段边界。
dma_attr_sgllen	指定分散/集中列表中的最大项数。dma_attr_sgllen 是 DMA 引擎在对设备的一个 I/O 请求中可以使用的 cookie 数。如果 DMA 引擎不包含任何分散/集中列表，则此字段应设置为 1。
dma_attr_granular	此字段用于提供设备 DMA 传送能力的粒度（以字节为单位）。指定海量存储设备的扇区大小即是关于如何使用该值的一个例子。如果绑定操作需要部分映射，则可使用此字段确保 DMA 窗口中的 cookie 大小之和为粒度的整数倍。但是，如果设备没有分散/集中功能，则 DDI 无法确保粒度。对于此情况，dma_attr_granular 字段的值应为 1。
dma_attr_flags	此字段可以设置为 DDI_DMA_FORCE_PHYSICAL，这表示如果系统同时支持物理 I/O 地址和虚拟 I/O 地址，则系统应返回物理 I/O 地址而非虚拟 I/O 地址。如果系统不支持物理 DMA，则 ddi_dma_alloc_handle(9F) 的返回值为 DDI_DMA_BADATTR。在

此情况下，驱动程序必须清除 DDI_DMA_FORCE_PHYSICAL 并重试该操作。

S 总线示例

在 SPARC 计算机中，S 总线上的 DMA 引擎具有以下特性：

- 仅访问 0xFF000000 到 0xFFFFFFFF 范围内的地址
- 32 位 DMA 计数器寄存器
- 可处理按字节对齐的传送
- 支持 1 字节、2 字节和 4 字节突发流量大小
- 最小有效传送大小为 1 字节
- 32 位地址寄存器
- 无分散/集中列表
- 仅对扇区执行操作，例如磁盘

在 SPARC 计算机中，S 总线上的 DMA 引擎具有以下特性结构：

```
static ddi_dma_attr_t attributes = {
    DMA_ATTR_V0,    /* Version number */
    0xFF000000,     /* low address */
    0xFFFFFFFF,     /* high address */
    0xFFFFFFFF,     /* counter register max */
    1,              /* byte alignment */
    0x7,            /* burst sizes: 0x1 | 0x2 | 0x4 */
    0x1,            /* minimum transfer size */
    0xFFFFFFFF,     /* max transfer size */
    0xFFFFFFFF,     /* address register max */
    1,              /* no scatter-gather */
    512,            /* device operates on sectors */
    0,              /* attr flag: set to 0 */
};
```

ISA 总线示例

在 x86 计算机中，ISA 总线上的 DMA 引擎具有以下特性：

- 仅访问前 16 MB 内存
- 在一次 DMA 传送中不能跨越 1 MB 的边界
- 16 位计数器寄存器
- 可处理按字节对齐的传送
- 支持 1 字节、2 字节和 4 字节突发流量大小
- 最小有效传送大小为 1 字节
- 最多可以支持 17 个分散/集中传送
- 仅对扇区执行操作，例如磁盘

在 x86 计算机中，ISA 总线上的 DMA 引擎具有以下特性结构：

```
static ddi_dma_attr_t attributes = {
    DMA_ATTR_V0,    /* Version number */
    0x00000000,     /* low address */
    0x00FFFFFF,     /* high address */
    0xFFFF,         /* counter register max */
    1,              /* byte alignment */
    0x7,            /* burst sizes */
    0x1,            /* minimum transfer size */
    0xFFFFFFFF,     /* max transfer size */
    0x000FFFFF,     /* address register max */
    17,             /* scatter-gather */
    512,            /* device operates on sectors */
    0,              /* attr flag: set to 0 */
};
```

管理 DMA 资源

本节介绍如何管理 DMA 资源。

对象锁定

为内存对象分配 DMA 资源之前，必须防止该对象移动。否则，在设备尝试向该对象进行写入时，系统会从内存中删除该对象。缺少对象会导致数据传送失败，并且可能损坏系统。防止内存对象在 DMA 传送过程中移动的过程称为**锁定对象**。

以下对象类型不要求显式锁定：

- 通过执行 `strategy(9E)` 获得的来自文件系统的缓冲区。这些缓冲区已由文件系统锁定。
- 设备驱动程序中分配的内核内存，如 `ddi_dma_mem_alloc(9F)` 分配的内核内存。

对于其他对象（如用户空间中的缓冲区），必须使用 `physio(9F)` 或 `ddi_umem_lock(9F)` 来锁定对象。使用这些函数来锁定对象通常在字符设备驱动程序的 `read(9E)` 或 `write(9E)` 例程中执行。有关示例，请参见第 252 页中的 “数据传输方法”。

分配 DMA 句柄

DMA 句柄是一个不透明的对象，用作对后续分配的 DMA 资源的引用。DMA 句柄通常在驱动程序的使用 `ddi_dma_alloc_handle(9F)` 的 `attach()` 入口点中分配。`ddi_dma_alloc_handle()` 函数采用 *dip* 引用的设备信息以及 `ddi_dma_attr(9S)` 结构描述的设备的 DMA 特性作为参数。`ddi_dma_alloc_handle()` 函数的语法如下所示：

```
int ddi_dma_alloc_handle(dev_info_t *dip,
    ddi_dma_attr_t *attr, int (*callback)(caddr_t),
    caddr_t arg, ddi_dma_handle_t *handlep);
```

其中：

dip 指向设备的 dev_info 结构的指针。

attr 指向 ddi_dma_attr(9S) 结构的指针，如第 149 页中的 “DMA 特性”中所述。

callback 用于处理资源分配故障的回调函数的地址。

arg 要传递给回调函数的参数。

handlep 指向 DMA 句柄的指针，用于存储返回的句柄。

分配 DMA 资源

以下两个接口用于分配 DMA 资源：

- ddi_dma_buf_bind_handle(9F)－与 buf(9S) 结构结合使用
- ddi_dma_addr_bind_handle(9F)－与虚拟地址结合使用

如果存在驱动程序的 xxstart() 例程，则 DMA 资源通常在 xxstart() 例程中分配。有关 xxstart() 的讨论，请参见第 280 页中的 “异步数据传输（块驱动程序）”。这两个接口的语法如下：

```
int ddi_dma_addr_bind_handle(ddi_dma_handle_t handle,
    struct as *as, caddr_t addr,
    size_t len, uint_t flags, int (*callback)(caddr_t),
    caddr_t arg, ddi_dma_cookie_t *cookiep, uint_t *ccountp);

int ddi_dma_buf_bind_handle(ddi_dma_handle_t handle,
    struct buf *bp, uint_t flags,
    int (*callback)(caddr_t), caddr_t arg,
    ddi_dma_cookie_t *cookiep, uint_t *ccountp);
```

以下参数对于 ddi_dma_addr_bind_handle(9F) 和 ddi_dma_buf_bind_handle(9F) 是通用的：

handle DMA 句柄和用于分配资源的对象。

flags 表示传送方向和其他特性的标志集。DDI_DMA_READ 表示从设备向内存传送数据。DDI_DMA_WRITE 表示从内存向设备传送数据。有关可用标志的完整讨论，请参见 ddi_dma_addr_bind_handle(9F) 或 ddi_dma_buf_bind_handle(9F) 手册页。

callback 用于处理资源分配故障的回调函数的地址。请参见 ddi_dma_alloc_handle(9F) 手册页。

arg 要传递给回调函数的参数。

cookiep 指向此对象的第一个 DMA cookie 的指针。

ccountp 指向此对象的 DMA cookie 数的指针。

对于 ddi_dma_addr_bind_handle(9F)，对象通过包含以下参数的地址范围进行描述：

as 指向地址空间结构的指针。*as* 的值必须是 NULL。

addr 对象的基本内核地址。

len 对象长度（以字节为单位）。

对于 ddi_dma_buf_bind_handle(9F)，对象通过 bp 所指向的 buf(9S) 结构进行描述。

设备寄存器结构

对于具有 DMA 功能的设备，要使用的寄存器比前面示例中所用寄存器多。

设备寄存器结构中使用以下字段来支持具有 DMA 功能但不支持分散/集中的设备：

```
uint32_t      dma_addr;      /* starting address for DMA */
uint32_t      dma_size;      /* amount of data to transfer */
```

设备寄存器结构中使用以下字段来支持具有 DMA 功能并支持分散/集中的设备：

```
struct sglentry {
    uint32_t    dma_addr;
    uint32_t    dma_size;
} sglist[SGLLEN];

caddr_t       iopb_addr;     /* When written, informs the device of the next */
                             /* command's parameter block address. */
                             /* When read after an interrupt, contains */
                             /* the address of the completed command. */
```

DMA 回调示例

在示例 9–1 中，xxstart() 用作回调函数。特定设备状态结构用作 xxstart() 的参数。xxstart() 函数将尝试启动命令。如果由于资源不可用而无法启动该命令，则会安排以后在资源可用时调用 xxstart()。

由于 xxstart() 用作 DMA 回调，因此 xxstart() 必须遵守以下规则，DMA 回调上施加了这些规则：

- 不能假定资源可用。回调必须尝试再次分配资源。
- 回调必须向系统指明分配是否成功。如果回调未能分配资源，则应返回 DDI_DMA_CALLBACK_RUNOUT，在此情况下需要以后再次调用 xxstart()。DDI_DMA_CALLBACK_DONE 表示回调成功，因此不需要再进行回调。

示例 9–1 DMA 回调示例

```
static int
xxstart(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
```

示例 9–1 DMA 回调示例 （续）

```
    struct device_reg *regp;
    int flags;
    mutex_enter(&xsp->mu);
    if (xsp->busy) {
        /* transfer in progress */
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    xsp->busy = 1;
    regp = xsp->regp;
    if ( /* transfer is a read */ ) {
        flags = DDI_DMA_READ;
    } else {
        flags = DDI_DMA_WRITE;
    }
    mutex_exit(&xsp->mu);
    if (ddi_dma_buf_bind_handle(xsp->handle,xsp->bp,flags, xxstart,
        (caddr_t)xsp, &cookie, &ccount) != DDI_DMA_MAPPED) {
        /* really should check all return values in a switch */
        mutex_enter(&xsp->mu);
        xsp->busy=0;
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    /* Program the DMA engine. */
    return (DDI_DMA_CALLBACK_DONE);
}
```

确定最大突发流量大小

驱动程序在 ddi_dma_attr(9S) 结构的 dma_attr_burstsizes 字段中指定其设备支持的 DMA 突发流量大小。此字段是所支持的突发流量大小的位图。但是，在分配 DMA 资源时，系统可能会对设备实际使用的突发流量大小施加更多限制。ddi_dma_burstsizes(9F) 例程可用来获取允许的突发流量大小。此例程将为设备返回适当的突发流量大小位图。分配 DMA 资源时，驱动程序可向系统请求用于其 DMA 引擎的适当突发流量大小。

示例 9–2 确定突发流量大小

```
#define BEST_BURST_SIZE 0x20 /* 32 bytes */

    if (ddi_dma_buf_bind_handle(xsp->handle,xsp->bp, flags, xxstart,
        (caddr_t)xsp, &cookie, &ccount) != DDI_DMA_MAPPED) {
        /* error handling */
    }
    burst = ddi_dma_burstsizes(xsp->handle);
    /* check which bit is set and choose one burstsize to */
    /* program the DMA engine */
    if (burst & BEST_BURST_SIZE) {
        /* program DMA engine to use this burst size */
```

示例 9–2 确定突发流量大小 （续）

```
    } else {
        /* other cases */
    }
```

分配专用 DMA 缓冲区

一些设备驱动程序除了执行用户线程和内核请求的传送外，可能还需要为 DMA 传送分配内存。分配专用 DMA 缓冲区的一些示例包括设置用于与设备之间进行通信的共享内存以及分配中间传送缓冲区。使用 ddi_dma_mem_alloc(9F) 可为 DMA 传送分配内存。

```
int ddi_dma_mem_alloc(ddi_dma_handle_t handle, size_t length,
    ddi_device_acc_attr_t *accattrp, uint_t flags,
    int (*waitfp)(caddr_t), caddr_t arg, caddr_t *kaddrp,
    size_t *real_length, ddi_acc_handle_t *handlep);
```

其中：

handle	DMA 句柄
length	所需分配的长度（以字节为单位）
accattrp	指向设备访问特性结构的指针
flags	数据传送模式标志。可能的值包括 DDI_DMA_CONSISTENT 和 DDI_DMA_STREAMING。
waitfp	用于处理资源分配故障的回调函数的地址。请参见 ddi_dma_alloc_handle(9F) 手册页。
arg	要传递给回调函数的参数
kaddrp	成功返回时包含已分配存储空间的地址的指针
real_length	分配的长度（以字节为单位）
handlep	指向数据访问句柄的指针

如果设备以不连续的方式进行访问，则应将 *flags* 参数设置为 DDI_DMA_CONSISTENT。由于会频繁应用于小型对象，因此使用 ddi_dma_sync(9F) 的同步步骤应尽可能为轻量级步骤。这种访问类型通常称为**一致**访问。一致访问对用于设备与驱动程序之间通信的 I/O 参数块特别有用。

在 x86 平台上，物理上连续的 DMA 内存的分配有以下要求：

- ddi_dma_attr(9S) 结构中分散/集中列表 dma_attr_sgllen 的长度必须设置为 1。
- 请勿指定 DDI_DMA_PARTIAL。DDI_DMA_PARTIAL 表示允许进行部分资源分配。

以下示例说明如何分配 IOPB 内存以及访问此内存必需的 DMA 资源。仍然必须分配 DMA 资源，并且必须将 DDI_DMA_CONSISTENT 标志传递给分配函数。

示例 9–3 使用 ddi_dma_mem_alloc(9F)

```
if (ddi_dma_mem_alloc(xsp->iopb_handle, size, &accattr,
    DDI_DMA_CONSISTENT, DDI_DMA_SLEEP, NULL, &xsp->iopb_array,
    &real_length, &xsp->acchandle) != DDI_SUCCESS) {
    /* error handling */
    goto failure;
}
if (ddi_dma_addr_bind_handle(xsp->iopb_handle, NULL,
    xsp->iopb_array, real_length,
    DDI_DMA_READ | DDI_DMA_CONSISTENT, DDI_DMA_SLEEP,
    NULL, &cookie, &count) != DDI_DMA_MAPPED) {
    /* error handling */
    ddi_dma_mem_free(&xsp->acchandle);
    goto failure;
}
```

对于顺序、单向、块大小和按块对齐的内存传送，*flags* 参数应设置为 DDI_DMA_STREAMING。这种访问类型通常称为**流**访问。

在某些情况下，使用 I/O 高速缓存可以加快 I/O 传送。I/O 高速缓存最少传送一个高速缓存行。ddi_dma_mem_alloc(9F) 例程会将 size 舍入为高速缓存行的倍数，以避免数据损坏。

ddi_dma_mem_alloc(9F) 函数将返回已分配的内存对象的实际大小。由于存在填充和对齐要求，实际大小可能会大于所请求的大小。ddi_dma_addr_bind_handle(9F) 函数要求使用实际长度。

使用 ddi_dma_mem_free(9F) 函数可以释放 ddi_dma_mem_alloc(9F) 分配的内存。

注 – 驱动程序必须确保缓冲区适当对齐。要求下限 DMA 缓冲区对齐的设备的驱动程序可能需要将数据复制到满足该要求的驱动程序中间缓冲区，然后将该中间缓冲区绑定到 DMA 的 DMA 句柄。使用 ddi_dma_mem_alloc(9F) 可分配驱动程序中间缓冲区。请务必使用 ddi_dma_mem_alloc(9F) 而非 kmem_alloc(9F) 来为要进行访问的设备分配内存。

处理资源分配故障

资源分配例程在处理分配故障时可为驱动程序提供若干选项。*waitfp* 参数用于指明分配例程是阻塞、立即返回还是安排回调，如下表所示。

表 9–1 资源分配处理

waitfp 值	表示的操作
DDI_DMA_DONTWAIT	驱动程序不想等到资源可用
DDI_DMA_SLEEP	驱动程序愿意无限期地等到资源可用
其他值	当资源可能可用时要调用的函数的地址

对 DMA 引擎进行编程

如果资源已成功分配，则必须对设备进行编程。尽管对 DMA 引擎进行编程是特定于设备的，但所有 DMA 引擎都需要一个起始地址和一个传送计数。设备驱动程序将从 ddi_dma_addr_bind_handle(9F)、ddi_dma_buf_bind_handle(9F) 或 ddi_dma_getwin(9F) 的成功调用所返回的 *DMA cookie* 中检索这两个值。这些函数都会返回第一个 DMA cookie 以及指示 DMA 对象是否包含多个 cookie 的 cookie 计数。如果 cookie 计数 *N* 大于 1，则必须对 ddi_dma_nextcookie(9F) 调用 *N*-1 次，以检索其余所有 cookie。

DMA cookie 的类型为 ddi_dma_cookie(9S)。这一类型的 cookie 包含以下字段：

```
uint64_t    _dmac_ll;       /* 64-bit DMA address */
uint32_t    _dmac_la[2];    /* 2 x 32-bit address */
size_t      dmac_size;      /* DMA cookie size */
uint_t      dmac_type;      /* bus specific type bits */
```

dmac_laddress 指定适用于对设备的 DMA 引擎进行编程的 64 位 I/O 地址。如果设备具有 64 位 DMA 地址寄存器，则驱动程序应使用此字段对 DMA 引擎进行编程。dmac_address 字段指定应该用于具有 32 位 DMA 地址寄存器的设备的 32 位 I/O 地址。dmac_size 字段包含传送计数。根据总线体系结构，驱动程序可能需要 cookie 中的 dmac_type 字段。驱动程序不应对 cookie 执行任何处理，如逻辑或算术处理。

示例 9–4 ddi_dma_cookie(9S) 示例

```
ddi_dma_cookie_t            cookie;

     if (ddi_dma_buf_bind_handle(xsp->handle,xsp->bp, flags, xxstart,
     (caddr_t)xsp, &cookie, &xsp->ccount) != DDI_DMA_MAPPED) {
         /* error handling */
      }
     sglp = regp->sglist;
     for (cnt = 1; cnt <= SGLLEN; cnt++, sglp++) {
     /* store the cookie parms into the S/G list */
     ddi_put32(xsp->access_hdl, &sglp->dma_size,
         (uint32_t)cookie.dmac_size);
     ddi_put32(xsp->access_hdl, &sglp->dma_addr,
         cookie.dmac_address);
     /* Check for end of cookie list */
     if (cnt == xsp->ccount)
         break;
     /* Get next DMA cookie */
```

示例 9–4 ddi_dma_cookie(9S) 示例 （续）

```
     (void) ddi_dma_nextcookie(xsp->handle, &cookie);
     }
     /* start DMA transfer */
     ddi_put8(xsp->access_hdl, &regp->csr,
     ENABLE_INTERRUPTS | START_TRANSFER);
```

释放 DMA 资源

DMA 传送完成后（通常在中断例程中），驱动程序可以通过调用 ddi_dma_unbind_handle(9F) 来释放 DMA 资源。

如第 161 页中的 “同步内存对象”中所述，ddi_dma_unbind_handle(9F) 可调用 ddi_dma_sync(9F)，从而无需进行任何显式同步。调用 ddi_dma_unbind_handle(9F) 之后，DMA 资源将无效，并且对资源的进一步引用会产生无法预料的结果。以下示例说明如何使用 ddi_dma_unbind_handle(9F)。

示例 9–5 释放 DMA 资源

```
static uint_t
xxintr(caddr_t arg)
{
     struct xxstate *xsp = (struct xxstate *)arg;
     uint8_t    status;
     volatile   uint8_t   temp;
     mutex_enter(&xsp->mu);
     /* read status */
     status = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
     if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
     }
     ddi_put8(xsp->access_hdl, &xsp->regp->csr, CLEAR_INTERRUPT);
     /* for store buffers */
     temp = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
     ddi_dma_unbind_handle(xsp->handle);
     /* Check for errors. */
     xsp->busy = 0;
     mutex_exit(&xsp->mu);
     if ( /* pending transfers */ ) {
        (void) xxstart((caddr_t)xsp);
     }
     return (DDI_INTR_CLAIMED);
}
```

应释放 DMA 资源。如果要在下一传送中使用不同对象，则应重新分配 DMA 资源。但是，如果始终使用同一个对象，则分配一次资源即可。只要保持对 ddi_dma_sync(9F) 的介入调用，随后便可重用资源。

释放 DMA 句柄

分离驱动程序时，必须释放 DMA 句柄。ddi_dma_free_handle(9F) 函数可销毁 DMA 句柄以及系统在该句柄上高速缓存的任何剩余资源。如果再对 DMA 句柄进行任何引用，将会产生无法预料的结果。

取消 DMA 回调

DMA 回调不能取消。取消 DMA 回调需要在驱动程序的 detach(9E) 入口点中附加一些代码。如果存在任何未完成的回调，则 detach() 例程一定不会返回 DDI_SUCCESS。请参见示例 9–6。发生 DMA 回调时，detach() 例程必须等待回调运行。回调完成时，detach() 必须防止回调自行重新安排。通过状态结构中的附加字段可以防止重新安排回调，如以下示例所示。

示例 9–6 取消 DMA 回调

```
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
     /* ... */
     mutex_enter(&xsp->callback_mutex);
     xsp->cancel_callbacks = 1;
     while (xsp->callback_count > 0) {
        cv_wait(&xsp->callback_cv, &xsp->callback_mutex);
     }
     mutex_exit(&xsp->callback_mutex);
     /* ... */
 }

static int
xxstrategy(struct buf *bp)
{
     /* ... */
     mutex_enter(&xsp->callback_mutex);
       xsp->bp = bp;
     error = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp, flags,
         xxdmacallback, (caddr_t)xsp, &cookie, &ccount);
     if (error == DDI_DMA_NORESOURCES)
       xsp->callback_count++;
     mutex_exit(&xsp->callback_mutex);
     /* ... */
}

static int
xxdmacallback(caddr_t callbackarg)
{
     struct xxstate *xsp = (struct xxstate *)callbackarg;
     /* ... */
     mutex_enter(&xsp->callback_mutex);
     if (xsp->cancel_callbacks) {
        /* do not reschedule, in process of detaching */
        xsp->callback_count--;
        if (xsp->callback_count == 0)
```

示例 9–6 取消 DMA 回调 （续）

```
            cv_signal(&xsp->callback_cv);
         mutex_exit(&xsp->callback_mutex);
         return (DDI_DMA_CALLBACK_DONE);    /* don't reschedule it */
      }
      /*
       * Presumably at this point the device is still active
       * and will not be detached until the DMA has completed.
       * A return of 0 means try again later
       */
      error = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp, flags,
          DDI_DMA_DONTWAIT, NULL, &cookie, &ccount);
      if (error == DDI_DMA_MAPPED) {
         /* Program the DMA engine. */
         xsp->callback_count--;
         mutex_exit(&xsp->callback_mutex);
         return (DDI_DMA_CALLBACK_DONE);
      }
      if (error != DDI_DMA_NORESOURCES) {
         xsp->callback_count--;
         mutex_exit(&xsp->callback_mutex);
         return (DDI_DMA_CALLBACK_DONE);
      }
      mutex_exit(&xsp->callback_mutex);
      return (DDI_DMA_CALLBACK_RUNOUT);
}
```

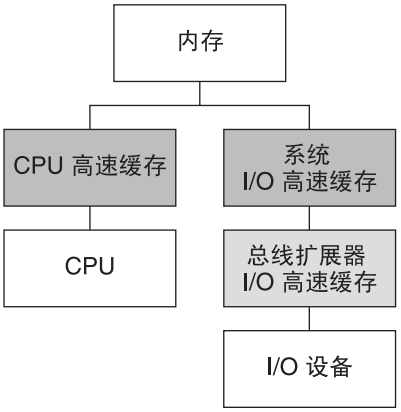
同步内存对象

在访问内存对象的过程中，驱动程序可能需要同步与各种高速缓存有关的内存对象。本节提供了有关何时以及如何同步内存对象的准则。

高速缓存

CPU 高速缓存是位于 CPU 和系统的主内存之间的极高速内存。I/O 高速缓存位于设备和系统的主内存之间，如下图所示。

图 9–1 CPU 和系统 I/O 高速缓存



尝试从主内存读取数据时，关联的高速缓存会对请求的数据进行检查。如果数据可用，高速缓存可快速提供这些数据。如果高速缓存中没有数据，则该高速缓存将从主内存中检索数据。然后，高速缓存会将数据传递给请求者并保存数据，以备在后续请求中使用。

类似地，在写循环中，数据会快速存储在高速缓存中。CPU 或设备可以继续执行，即传送数据。将数据存储在高速缓存中所需的时间比等待将数据写入内存所需的时间少得多。

采用此模型，在设备传送完成后，数据仍可位于 I/O 高速缓存中，而主内存中没有数据。如果 CPU 访问内存，CPU 可能会从 CPU 高速缓存中读取错误数据。驱动程序必须调用同步例程，以刷新 I/O 高速缓存中的数据，并使用新数据更新 CPU 高速缓存。此操作可确保内存的情况对于 CPU 而言保持一致。类似地，如果设备要对 CPU 修改的数据进行访问，则需要采用同步步骤。

可在设备和内存之间创建附加的高速缓存和缓冲区，如总线延伸架和桥。使用 `ddi_dma_sync(9F)` 可以同步**所有**适用的高速缓存。

`ddi_dma_sync()` 函数

一个内存对象可能有多个映射，如通过 DMA 句柄用于 CPU 和用于设备的映射。如果使用任何映射来修改内存对象，则具有多个映射的驱动程序需要调用 `ddi_dma_sync(9F)`。调用 `ddi_dma_sync()` 可以确保对内存对象的修改在通过不同映射访问该对象之前完成。如果对对象的任何高速缓存引用现在已过时，`ddi_dma_sync()` 函数还可以通知该对象的其他映射。此外，`ddi_dma_sync()` 还会根据需要刷新过时的高速缓存引用或使其无效。

通常，当 DMA 传送完成时，驱动程序必须调用 `ddi_dma_sync()`。此规则的例外情况是如果使用 `ddi_dma_unbind_handle(9F)` 取消分配 DMA 资源，则会代表驱动程序隐式执行 `ddi_dma_sync()`。`ddi_dma_sync()` 的语法如下：

```
int ddi_dma_sync(ddi_dma_handle_t handle, off_t off,
size_t length, uint_t type);
```

如果设备的 DMA 引擎要读取对象，则必须通过将 *type* 设置为 DDI_DMA_SYNC_FORDEV 来同步该设备看到的对象信息。如果设备的 DMA 引擎已写入内存对象并且 CPU 将读取该对象，则必须通过将 *type* 设置为 DDI_DMA_SYNC_FORCPU 来同步该 CPU 看到的对象信息。

以下示例说明如何为 CPU 同步 DMA 对象：

```
if (ddi_dma_sync(xsp->handle, 0, length, DDI_DMA_SYNC_FORCPU)
    == DDI_SUCCESS) {
    /* the CPU can now access the transferred data */
    /* ... */
} else {
    /* error handling */
}
```

如果唯一的映射是用于内核的，请使用标志 DDI_DMA_SYNC_FORKERNEL，类似于 ddi_dma_mem_alloc(9F) 所分配的内存中的情况。系统会尝试以比同步 CPU 看到的信息更快的速度来同步内核看到的信息。如果系统无法更快地同步内核看到的信息，则系统将按照如同已设置 DDI_DMA_SYNC_FORCPU 标志的情况执行相应的操作。

DMA 窗口

如果对象不满足 DMA 引擎的限制，则必须将传送分为一系列较小的传送。驱动程序本身即可对传送进行拆分。或者，驱动程序也可以允许系统仅为对象的一部分分配资源，从而创建一系列 DMA **窗口**。允许系统分配资源是首选解决方案，因为系统管理资源的效率比驱动程序高。

DMA 窗口有两个特性。*offset* 特性是从对象的开头度量的。*length* 特性是要分配的内存的字节数。在进行部分分配后，只有一系列在 *offset* 开始的 *length* 字节分配了资源。

请求 DMA 窗口的方法是将 DDI_DMA_PARTIAL 标志指定为 ddi_dma_buf_bind_handle(9F) 或 ddi_dma_addr_bind_handle(9F) 的参数。如果可以建立窗口，则这两个函数都将返回 DDI_DMA_PARTIAL_MAP。但是，系统可能会为整个对象分配资源，此时将返回 DDI_DMA_MAPPED。驱动程序应检查返回值，以确定 DMA 窗口是否正在使用。请参见以下示例。

示例 9–7 设置 DMA 窗口

```
static int
xxstart (caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct device_reg *regp = xsp->reg;
    ddi_dma_cookie_t cookie;
    int status;
    mutex_enter(&xsp->mu);
```

示例 9–7 设置 DMA 窗口 （续）

```
    if (xsp->busy) {
        /* transfer in progress */
        mutex_exit(&xsp->mu);
        return (DDI_DMA_CALLBACK_RUNOUT);
    }
    xsp->busy = 1;
    mutex_exit(&xsp->mu);
    if ( /* transfer is a read */) {
        flags = DDI_DMA_READ;
    } else {
        flags = DDI_DMA_WRITE;
    }
    flags |= DDI_DMA_PARTIAL;
    status = ddi_dma_buf_bind_handle(xsp->handle, xsp->bp,
        flags, xxstart, (caddr_t)xsp, &cookie, &ccount);
    if (status != DDI_DMA_MAPPED &&
        status != DDI_DMA_PARTIAL_MAP)
        return (DDI_DMA_CALLBACK_RUNOUT);
    if (status == DDI_DMA_PARTIAL_MAP) {
        ddi_dma_numwin(xsp->handle, &xsp->nwin);
        xsp->partial = 1;
        xsp->windex = 0;
    } else {
        xsp->partial = 0;
    }
    /* Program the DMA engine. */
    return (DDI_DMA_CALLBACK_DONE);
}
```

有两个函数可对 DMA 窗口执行操作。第一个函数 ddi_dma_numwin(9F) 可为特定的 DMA 对象返回 DMA 窗口数。另一个函数 ddi_dma_getwin(9F) 允许在对象内重新定位，即重新分配系统资源。ddi_dma_getwin () 函数用于从当前窗口切换到对象中的新窗口。由于 ddi_dma_getwin() 会将系统资源重新分配给新窗口，因此前面的窗口将变为无效。

注意 – 在向当前窗口的传送完成之前，请勿通过调用 ddi_dma_getwin() 来移动 DMA 窗口。请一直等待，直至向当前窗口的传送完成为止，即出现中断的时候。然后，调用 ddi_dma_getwin() 以避免数据损坏。

ddi_dma_getwin() 函数通常是从某个中断例程调用的，如示例 9–8 中所示。调用驱动程序会导致启动第一个 DMA 传送。后续传送将从中断例程中启动。

中断例程会检查设备的状态，以确定设备是否已成功完成传送。如果未成功完成传送，则会进行正常错误恢复。如果传送成功，例程必须确定逻辑传送是否已完成。完整的传送包括 buf(9S) 结构所指定的整个对象。在部分传送中，仅会移动一个 DMA 窗口。在部分传送中，中断例程将使用 ddi_dma_getwin(9F) 移动窗口、检索新 cookie 并启动其他 DMA 传送。

如果逻辑请求已完成，则中断例程将检查待处理的请求。如有必要，中断例程会启动传送。否则，例程将返回，而不调用其他 DMA 传送。以下示例说明了常见的流程控制。

示例 9–8 使用 DMA 窗口中断处理程序

```
static uint_t
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    uint8_t    status;
    volatile   uint8_t   temp;
    mutex_enter(&xsp->mu);
    /* read status */
    status = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
    }
    ddi_put8(xsp->access_hdl,&xsp->regp->csr, CLEAR_INTERRUPT);
    /* for store buffers */
    temp = ddi_get8(xsp->access_hdl, &xsp->regp->csr);
    if ( /* an error occurred during transfer */ ) {
        bioerror(xsp->bp, EIO);
        xsp->partial = 0;
    } else {
        xsp->bp->b_resid -= /* amount transferred */ ;
    }

    if (xsp->partial && (++xsp->windex < xsp->nwin)) {
        /* device still marked busy to protect state */
        mutex_exit(&xsp->mu);
        (void) ddi_dma_getwin(xsp->handle, xsp->windex,
            &offset, &len, &cookie, &ccount);
        /* Program the DMA engine with the new cookie(s). */
        return (DDI_INTR_CLAIMED);
    }
    ddi_dma_unbind_handle(xsp->handle);
    biodone(xsp->bp);
    xsp->busy = 0;
    xsp->partial = 0;
    mutex_exit(&xsp->mu);
    if ( /* pending transfers */ ) {
        (void) xxstart((caddr_t)xsp);
    }
    return (DDI_INTR_CLAIMED);
}
```

◆ ◆ ◆ 第 10 章

10 映射设备和内核内存

一些设备驱动程序允许应用程序通过 mmap(2) 访问设备或内核内存。例如，帧缓存器驱动程序允许将帧缓存器映射到用户线程中。另一个示例是使用共享的内核内存池与应用程序通信的伪驱动程序。本章介绍有关以下主题的信息：

内存映射概述

驱动程序必须采取如下步骤才能导出设备或内核内存：

1. 在 cb_ops(9S) 结构的 cb_flag 标志中设置 D_DEVMAP 标志。
2. 定义 devmap(9E) 驱动程序入口点并视需要定义 segmap(9E) 入口点，以导出映射。
3. 使用 devmap_devmem_setup(9F) 设置到设备的用户映射。要设置到内核内存的用户映射，请使用 devmap_umem_setup(9F)。

导出映射

本节介绍如何使用 segmap(9E) 和 devmap(9E) 入口点。

segmap(9E) 入口点

segmap(9E) 入口点负责设置 mmap(2) 系统调用所请求的内存映射。许多内存映射设备的驱动程序使用 ddi_devmap_segmap(9F) 作为入口点，而不是定义自己的 segmap(9E) 例程。通过提供 segmap() 入口点，驱动程序能够在创建映射之前或之后管理常规任务。例如，驱动程序可以检查映射权限并分配专用映射资源。驱动程序还可以调整映

射，以适应非按页对齐的设备缓冲区。segmap() 入口点在返回之前必须调用 ddi_devmap_segmap(9F) 函数。ddi_devmap_segmap() 函数调用驱动程序的 devmap(9E) 入口点以执行实际映射。

segmap() 函数的语法如下：

```
int segmap(dev_t dev, off_t off, struct as *asp, caddr_t *addrp,
     off_t len, unsigned int prot, unsigned int maxprot,
     unsigned int flags, cred_t *credp);
```

其中：

dev 要映射其内存的设备。

off 设备内存中的偏移，映射将从此位置开始。

asp 指向设备内存将要映射到的地址空间的指针。

请注意，此参数可以是 struct as *（如示例 10–1 中所示），也可以是 ddi_as_handle_t（如示例 10–2 中所示）。这是因为 ddidevmap.h 包括以下声明：

```
typedef struct as *ddi_as_handle_t
```

addrp 指向设备内存将要映射到的地址空间中的地址的指针。

len 所映射的内存的长度（以字节为单位）。

prot 指定保护的位字段。可能的设置有 PROT_READ、PROT_WRITE、PROT_EXEC、PROT_USER 和 PROT_ALL。有关详细信息，请参见手册页。

maxprot 尝试的映射可用的最大保护标志。如果用户打开只读的特殊文件，则 PROT_WRITE 位可能会被屏蔽。

flags 指示映射类型的标志。可能的值包括 MAP_SHARED 和 MAP_PRIVATE。

credp 指向用户凭证结构的指针。

在以下示例中，驱动程序控制允许只写映射的帧缓存器。如果应用程序尝试进行读取访问，并随后调用 ddi_devmap_segmap(9F) 以设置用户映射，驱动程序将返回 EINVAL。

示例 10–1 segmap(9E) 例程

```
static int
xxsegmap(dev_t dev, off_t off, struct as *asp, caddr_t *addrp,
    off_t len, unsigned int prot, unsigned int maxprot,
    unsigned int flags, cred_t *credp)
{
    if (prot & PROT_READ)
        return (EINVAL);
    return (ddi_devmap_segmap(dev, off, as, addrp,
        len, prot, maxprot, flags, cred));
}
```

以下示例说明如何处理其缓冲区未在寄存器空间中按页对齐的设备。本示例将映射从偏移 0x800 开始的缓冲区，因此 mmap(2) 会返回与该缓冲区起始位置对应的地址。devmap_devmem_setup(9F) 函数映射整页，要求映射按页对齐，并返回页起始位置的地址。如果此地址是通过 segmap(9E) 传递的，或者未定义 segmap() 入口点，则 mmap() 将返回对应于页起始位置的地址，而不是对应于缓冲区起始位置的地址。在本示例中，缓冲区偏移将与 devmap_devmem_setup 返回的按页对齐地址相加，因此，得到的返回地址就是所需的缓冲区起始地址。

示例 10–2 使用 segmap() 函数更改 mmap() 调用返回的地址

```
#define    BUFFER_OFFSET 0x800

int
xx_segmap(dev_t dev, off_t off, ddi_as_handle_t as, caddr_t *addrp, off_t len,
    uint_t prot, uint_t maxprot, uint_t flags, cred_t *credp)
{
        int rval;
        unsigned long pagemask = ptob(1L) - 1L;

        if ((rval = ddi_devmap_segmap(dev, off, as, addrp, len, prot, maxprot,
            flags, credp)) == DDI_SUCCESS) {
                /*
                 * The address returned by ddi_devmap_segmap is the start of the page
                 * that contains the buffer.  Add the offset of the buffer to get the
                 * final address.
                 */
                *addrp += BUFFER_OFFSET & pagemask);
        }
        return (rval);
}
```

devmap(9E) 入口点

devmap(9E) 入口点通过 segmap(9E) 入口点内部的 ddi_devmap_segmap(9F) 函数调用。

devmap(9E) 入口点是作为 mmap(2) 系统调用的结果调用的。调用 devmap(9E) 函数可将设备内存或内核内存导出到用户应用程序。devmap() 函数用于进行以下操作：

- 验证用户到设备内存或内核内存的映射
- 将应用程序映射中的逻辑偏移转换为设备或内核内存中的对应偏移
- 将映射信息传递给系统以设置映射

devmap() 函数的语法如下所示：

```
int devmap(dev_t dev, devmap_cookie_t handle, offset_t off,
     size_t len, size_t *maplen, uint_t model);
```

其中：

dev 要映射其内存的设备。

handle 系统创建的设备映射句柄，用来描述到设备或内核中的连续内存的映射。

off 应用程序映射中的逻辑偏移，必须通过驱动程序将其转换为设备或内核内存中的对应偏移。

len 所映射的内存的长度（以字节为单位）。

maplen 使驱动程序可将不同的内核内存区域或多个物理上不连续的内存区域与一个连续的用户应用程序映射相关联。

model 当前线程的数据模型类型。

系统在一次 mmap(2) 系统调用中将创建多个映射句柄。例如，映射可能包含多个物理上不连续的内存区域。

devmap(9E) 的首次调用使用参数 *off* 和 *len* 的初始值。应用程序将这些参数传递给 mmap(2)。devmap(9E) 将 **maplen* 设置为从 *off* 到连续内存区域末尾之间的长度。**maplen* 值必须向上进位为页面大小的倍数。 **maplen* 值可被设置为小于原始映射长度 `len`。如果这样，系统将使用调整了 *off* 和 *len* 参数的新映射句柄反复调用 devmap(9E)，直到达到初始映射长度为止。

如果一个驱动程序支持多个应用程序数据模型，则必须将 *model* 传递给 ddi_model_convert_from(9F)。`ddi_model_convert_from()` 函数可以确定当前线程与设备驱动程序之间是否存在数据模型不匹配的情况。设备驱动程序可能必须调整数据结构的形状，然后才能将结构导出到支持不同数据模型的用户线程。有关更多详细信息，请参见附录 C，使设备驱动程序支持 64 位。

如果逻辑偏移 *off* 超出了驱动程序导出的内存范围，则 devmap(9E) 入口点必将返回 `-1`。

将设备内存与用户映射相关联

通过驱动程序的 devmap_devmem_setup(9F) 入口点调用 devmap(9E) 可将设备内存导出到用户应用程序。

`devmap_devmem_setup`(9F) 函数的语法如下所示：

```
int devmap_devmem_setup(devmap_cookie_t handle, dev_info_t *dip,
    struct devmap_callback_ctl *callbackops, uint_t rnumber,
    offset_t roff, size_t len, uint_t maxprot, uint_t flags,
    ddi_device_acc_attr_t *accattrp);
```

其中：

handle 系统用来标识映射的不透明的设备映射句柄。

dip 指向设备的 `dev_info` 结构的指针。

callbackops 指向 devmap_callback_ctl(9S) 结构的指针，此指针可在映射时向驱动程序通知用户事件。

rnumber 寄存器地址空间集的索引号。

roff 在设备内存中的偏移。

len 导出的长度（以字节为单位）。

maxprot 允许驱动程序为导出的设备内存中的不同区域指定不同的保护。

flags 必须设置为 DEVMAP_DEFAULTS。

accattrp 指向 ddi_device_acc_attr(9S) 结构的指针。

roff 和 *len* 参数描述了寄存器集 *rnumber* 指定的设备内存中的一个范围。reg 属性用于描述 *rnumber* 所引用的寄存器规格。对于只有一个寄存器集的设备，将 rnumber 设置为 0 即可。范围通过 *roff* 和 *len* 定义。如果用户的应用程序映射位于通过 devmap(9E) 入口点传入的 *offset* 位置上，则可对此范围进行访问。驱动程序通常将 devmap(9E) 偏移直接传递给 devmap_devmem_setup(9F)。然后，mmap(2) 的返回地址将映射到寄存器集的起始地址。

通过 *maxprot* 参数，驱动程序可为导出的设备内存中的不同区域指定不同保护。例如，要禁止对某个区域进行写访问，可以只为该区域设置 PROT_READ 和 PROT_USER。

以下示例说明如何将设备内存导出到应用程序。驱动程序首先确定请求的映射是否位于设备内存区域之内。设备内存的大小通过使用 ddi_dev_regsize(9F) 来确定。使用 ptob(9F) 和 btopr(9F) 可将映射的长度向上舍入为页面大小的倍数。然后调用 devmap_devmem_setup(9F) 可将设备内存导出到应用程序。

示例 10–3 使用 devmap_devmem_setup() 例程

```
static int
xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off, size_t len,
    size_t *maplen, uint_t model)
{
    struct xxstate *xsp;
    int    error, rnumber;
    off_t regsize;

    /* Set up data access attribute structure */
    struct ddi_device_acc_attr xx_acc_attr = {
        DDI_DEVICE_ATTR_V0,
        DDI_NEVERSWAP_ACC,
        DDI_STRICTORDER_ACC
    };
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (-1);
    /* use register set 0 */
    rnumber = 0;
    /* get size of register set */
    if (ddi_dev_regsize(xsp->dip, rnumber, &regsize) != DDI_SUCCESS)
```

示例 10–3 使用 devmap_devmem_setup() 例程 （续）

```
        return (-1);
    /* round up len to a multiple of a page size */
       len = ptob(btopr(len));
    if (off + len > regsize)
        return (-1);
    /* Set up the device mapping */
    error = devmap_devmem_setup(handle, xsp->dip, NULL, rnumber,
    off, len, PROT_ALL, DEVMAP_DEFAULTS, &xx_acc_attr);
    /* acknowledge the entire range */
    *maplen = len;
    return (error);
}
```

将内核内存与用户映射相关联

一些设备驱动程序可能需要分配可供用户程序通过 mmap(2) 进行访问的内核内存。一个示例是为两个应用程序间的通信设置共享内存。另一个示例是在驱动程序和应用程序之间共享内存。

将内核内存导出到用户应用程序时，请执行以下步骤：

1. 使用 ddi_umem_alloc(9F) 分配内核内存。
2. 使用 devmap_umem_setup(9F) 导出内存。
3. 不再需要内存时，使用 ddi_umem_free(9F) 释放内存。

为用户访问分配内核内存

使用 ddi_umem_alloc(9F) 可以分配导出到应用程序的内核内存。 ddi_umem_alloc() 的语法如下所示：

```
void *ddi_umem_alloc(size_t size, int flag, ddi_umem_cookie_t
*cookiep);
```

其中：

size 要分配的字节数。

flag 用于确定休眠条件和内存类型。

cookiep 指向内核内存 cookie 的指针。

ddi_umem_alloc(9F) 分配按页对齐的内核内存。ddi_umem_alloc () 返回一个指向所分配内存的指针。最初，内存被零填充。分配的字节数是系统页面大小的倍数，该页面大小是通过 *size* 参数向上舍入得到的。分配的内存可在内核中使用。此内存也可导出到

应用程序。*cookiep* 是指向用来描述所分配的内核内存的内核内存 cookie 的指针。驱动程序将内核内存导出到用户应用程序时，devmap_umem_setup(9F) 中会使用 *cookiep*

flag 参数用于指示 ddi_umem_alloc(9F) 是立即阻塞还是返回，以及分配的内核内存是否可换页。flag 参数的值如下所示：

DDI_UMEM_NOSLEEP	驱动程序无需等待内存成为可用。如果内存不可用，则返回 NULL。
DDI_UMEM_SLEEP	驱动程序可以无限等待，直到内存可用为止。
DDI_UMEM_PAGEABLE	驱动程序允许内存页被换出。如果未设置，则锁定内存。

ddi_umem_lock() 函数可以执行设备锁定内存检查。此函数针对 project.max-locked-memory 中指定的限制值进行检查。如果当前项目的锁定内存使用量低于限制，则会增加项目的锁定内存字节计数。进行限制检查后，内存将会锁定。ddi_umem_unlock() 函数可以解除锁定内存，从而减少项目的锁定内存字节计数。

其中所用的记帐方法是不严密的 "full price"（足价）模式。例如，对于同一项目中 umem_lockmemory() 的具有重叠内存区域的两个调用方会被计数两次。

有关 project.max-locked-memory 和 zone.max-locked_memory 对安装了区域的 Oracle Solaris 系统的资源控制的信息，请参见《Resource Management and Oracle Solaris Zones Developer's Guide》和 resource_controls(5)。

以下示例说明如何为应用程序访问分配内核内存。驱动程序会导出一页内核内存，它将被多个应用程序用作共享存储区。应用程序第一次映射共享页时，会在 segmap(9E) 中分配内存。如果驱动程序必须支持多个应用程序数据模型，则会再分配一页。例如，64 位驱动程序可能同时将内存导出到 64 位应用程序和 32 位应用程序。64 位应用程序共享第一页，32 位应用程序共享第二页。

示例 10–4 使用 ddi_umem_alloc() 例程

```
static int
xxsegmap(dev_t dev, off_t off, struct as *asp, caddr_t *addrp, off_t len,
    unsigned int prot, unsigned int maxprot, unsigned int flags,
    cred_t *credp)
{
    int error;
    minor_t instance = getminor(dev);
    struct xxstate *xsp = ddi_get_soft_state(statep, instance);

    size_t mem_size;
        /* 64-bit driver supports 64-bit and 32-bit applications */
    switch (ddi_mmap_get_model()) {
        case DDI_MODEL_LP64:
             mem_size = ptob(2);
             break;
        case DDI_MODEL_ILP32:
             mem_size = ptob(1);
             break;
```

示例 10–4 使用 ddi_umem_alloc() 例程 （续）

```
    }

    mutex_enter(&xsp->mu);
    if (xsp->umem == NULL) {
        /* allocate the shared area as kernel pageable memory */
        xsp->umem = ddi_umem_alloc(mem_size,
            DDI_UMEM_SLEEP | DDI_UMEM_PAGEABLE, &xsp->ucookie);
    }
    mutex_exit(&xsp->mu);
    /* Set up the user mapping */
    error = devmap_setup(dev, (offset_t)off, asp, addrp, len,
        prot, maxprot, flags, credp);
    return (error);
}
```

将内核内存导出到应用程序

使用 devmap_umem_setup(9F) 可将内核内存导出到用户应用程序。devmap_umem_setup() 必须通过驱动程序的 devmap(9E) 入口点进行调用 。devmap_umem_setup() 的语法如下所示：

```
int devmap_umem_setup(devmap_cookie_t handle, dev_info_t *dip,
    struct devmap_callback_ctl *callbackops, ddi_umem_cookie_t cookie,
    offset_t koff, size_t len, uint_t maxprot, uint_t flags,
    ddi_device_acc_attr_t *accattrp);
```

其中：

handle 用于描述映射的不透明结构。

dip 指向设备的 dev_info 结构的指针。

callbackops 指向 devmap_callback_ctl(9S) 结构的指针。

cookie ddi_umem_alloc(9F) 返回的内核内存 cookie。

koff cookie 指定的内核内存中的偏移。

len 导出的长度（以字节为单位）。

maxprot 用于为导出的映射指定可能的最大保护。

flags 必须设置为 DEVMAP_DEFAULTS。

accattrp 指向 ddi_device_acc_attr(9S) 结构的指针。

handle 是系统用来标识映射的设备映射句柄。*handle* 通过 devmap(9E) 入口点传入。*dip* 是指向设备的 dev_info 结构的指针。*callbackops* 允许向驱动程序通知有关映射的用户事件。导出内核内存时，大多数驱动程序都会将 *callbackops* 设置为 NULL。

koff 和 *len* 用于在 ddi_umem_alloc(9F) 分配的内核内存中指定一个范围。如果用户的应用程序映射位于通过 devmap(9E) 入口点传入的偏移上，则可对此范围进行访问。通常，驱动程序将 devmap(9E) 偏移直接传递给 devmap_umem_setup(9F)。然后，mmap(2) 的返回地址将映射到 ddi_umem_alloc(9F) 返回的内核地址。*koff* 和 *len* 必须按页对齐。

通过 *maxprot*，驱动程序可为导出的内核内存中的不同区域指定不同的保护。例如，通过仅设置 PROT_READ 和 PROT_USER，一个区域可能不允许写访问。

以下示例说明如何将内核内存导出到应用程序。驱动程序首先检查请求的映射是否位于分配的内核内存区域之内。如果 64 位驱动程序收到来自 32 位应用程序的映射请求，则会将该请求重定向到内核存储区的第二页。此重定向可确保仅有编译到相同数据模型的应用程序才能共享相同的页。

示例 10–5 devmap_umem_setup(9F) 例程

```
static int
xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off, size_t len,
    size_t *maplen, uint_t model)
{
    struct xxstate *xsp;
    int    error;

    /* round up len to a multiple of a page size */
    len = ptob(btopr(len));
    /* check if the requested range is ok */
    if (off + len > ptob(1))
        return (ENXIO);
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);

    if (ddi_model_convert_from(model) == DDI_MODEL_ILP32)
        /* request from 32-bit application. Skip first page */
        off += ptob(1);

    /* export the memory to the application */
    error = devmap_umem_setup(handle, xsp->dip, NULL, xsp->ucookie,
        off, len, PROT_ALL, DEVMAP_DEFAULTS, NULL);
    *maplen = len;
    return (error);
}
```

释放为用户访问导出的内核内存

卸载驱动程序时，必须通过调用 ddi_umem_alloc(9F) 释放 ddi_umem_free(9F) 分配的内存。

```
void ddi_umem_free(ddi_umem_cookie_t cookie);
```

cookie 是 ddi_umem_alloc(9F) 返回的内核内存 cookie。

◆ ◆ ◆ 第 11 章

11 设备上下文管理

一些设备驱动程序（如用于图形硬件的驱动程序）可为用户进程提供对设备的直接访问。这些设备通常要求一次仅有一个进程访问设备。

本章介绍了可供设备驱动程序用于管理对此类设备的访问的接口集。本章提供有关以下主题的信息：

设备上下文简介

本节介绍设备上下文和上下文管理模型。

什么是设备上下文？

设备的**上下文**是指设备硬件的当前状态。设备驱动程序可代表进程管理该进程的设备上下文。驱动程序必须分别为访问设备的每个进程保留单独的设备上下文。设备驱动程序负责在进程访问设备时恢复正确的设备上下文。

上下文管理模型

帧缓存器可作为设备上下文管理的一个很好的示例。使用加速的帧缓存器，用户进程可以通过内存映射访问直接处理设备的控制寄存器。由于这些进程不使用传统的系统调用，因此访问设备的进程无需调用设备驱动程序。但是，如果进程要访问设备，则必须通知设备驱动程序。驱动程序需要恢复正确的设备上下文并且提供所需的任何同步。

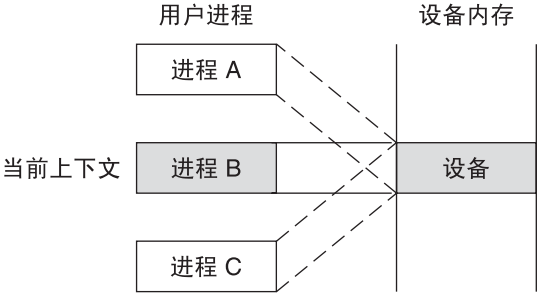
要解决这一问题，可以使用设备上下文管理接口，在用户进程访问设备的内存映射区域时通知设备驱动程序，并控制对设备硬件的访问。设备驱动程序负责同步和管理各种设备上下文。用户进程访问映射时，设备驱动程序必须为该进程恢复正确的设备上下文。

每次用户进程执行以下任一操作时，都会通知设备驱动程序：

- 访问映射
- 复制映射
- 释放映射
- 创建映射

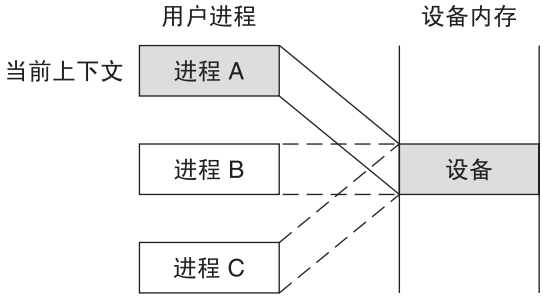
下图显示了映射到一个设备的内存中的多个用户进程。驱动程序授予了进程 B 对设备的访问权限，进程 B 不再向驱动程序通知访问情况。但是，如果进程 A 或进程 C 访问设备，仍**会**通知驱动程序。

图 11–1 设备上下文管理



在将来某一时刻，进程 A 将访问设备。设备驱动程序将收到通知，并且它将阻止进程 B 将来访问该设备。然后，驱动程序保存进程 B 的设备上下文。驱动程序恢复进程 A 的设备上下文。然后，驱动程序将向进程 A 授予访问权限，如下图所示。此时，如果进程 B 或进程 C 访问设备，则会通知设备驱动程序。

图 11–2 切换到用户进程 A 的设备上下文



在多处理器计算机中，多个进程可能会尝试同时访问设备。此情况会引起抖动。有些设备需要较长的时间才能恢复设备上下文。要防止恢复设备上下文所需的 CPU 时间超过实际使用该设备上下文所需的时间，可以使用 devmap_set_ctx_timeout(9F) 设置进程访问设备所需的最短时间。

内核可以保证一旦设备驱动程序向某一进程授予了访问权限，便不允许其他任何进程在 devmap_set_ctx_timeout(9F) 指定的时间间隔内请求访问同一设备。

上下文管理操作

执行设备上下文管理的常规步骤如下：

1. 定义 devmap_callback_ctl(9S) 结构。
2. 根据需要分配用于保存设备上下文的空间。
3. 通过 devmap_devmem_setup(9F) 设置到设备的用户映射和驱动程序通知。
4. 通过 devmap_load(9F) 和 devmap_unload(9F) 管理用户对设备的访问。
5. 根据需要释放设备上下文结构。

devmap_callback_ctl 结构

设备驱动程序必须分配并初始化 devmap_callback_ctl(9S) 结构，以便通知系统用于设备上下文管理的入口点例程。

此结构使用以下语法：

```
struct devmap_callback_ctl {
    int devmap_rev;
    int (*devmap_map)(devmap_cookie_t dhp, dev_t dev,
    uint_t flags, offset_t off, size_t len, void **pvtp);
    int (*devmap_access)(devmap_cookie_t dhp, void *pvtp,
    offset_t off, size_t len, uint_t type, uint_t rw);
    int (*devmap_dup)(devmap_cookie_t dhp, void *pvtp,
    devmap_cookie_t new_dhp, void **new_pvtp);
```

```
    void (*devmap_unmap)(devmap_cookie_t dhp, void *pvtp,
    offset_t off, size_t len, devmap_cookie_t new_dhp1,
    void **new_pvtp1, devmap_cookie_t new_dhp2,
    void **new_pvtp2);
};
```

devmap_rev	devmap_callback_ctl 结构的版本号。版本号必须设置为 DEVMAP_OPS_REV。
devmap_map	必须设置为驱动程序的 devmap_map(9E) 入口点的地址。
devmap_access	必须设置为驱动程序的 devmap_access(9E) 入口点的地址。
devmap_dup	必须设置为驱动程序的 devmap_dup(9E) 入口点的地址。
devmap_unmap	必须设置为驱动程序的 devmap_unmap(9E) 入口点的地址。

用于设备上下文管理的入口点

以下入口点用于管理设备上下文：

- devmap(9E)
- devmap_access(9E)
- devmap_contextmgt(9E)
- devmap_dup(9E)
- devmap_unmap(9E)

devmap_map() 入口点

devmap(9E) 的语法如下所示：

```
int xxdevmap_map(devmap_cookie_t handle, dev_t dev, uint_t flags,
    offset_t offset, size_t len, void **new-devprivate);
```

驱动程序从其 devmap() 入口点返回并且系统已建立到设备内存的用户映射后，将会调用 devmap_map() 入口点。通过 devmap() 入口点，驱动程序可以执行其他处理操作或分配特定于映射的专用数据。例如，为了支持上下文切换，驱动程序必须分配上下文结构。然后，驱动程序必须将上下文结构与映射关联。

系统期望驱动程序在 **new-devprivate* 中返回一个指向分配的专用数据的指针。驱动程序必须存储用于定义专用数据中的映射范围的 *offset* 和 *len*。然后当系统调用 devmap_unmap(9E) 时，驱动程序将使用此信息来确定要取消映射的映射量。

flags 指示驱动程序是否应为映射分配专用上下文。例如，如果 *flags* 设置为 MAP_PRIVATE，则驱动程序可以分配用于存储设备上下文的内存区域。如果设置了 MAP_SHARED，驱动程序将返回指向共享区域的指针。

以下示例说明了 devmap() 入口点。驱动程序分配了一个新的上下文结构。然后，驱动程序便可保存通过入口点传入的相关参数。接下来，将通过分配或通过将映射附加至已经存在的共享上下文来为映射指定新的上下文。映射访问设备的最短时间间隔设置为 1 毫秒。

示例 11–1 使用 devmap() 例程

```
static int
int xxdevmap_map(devmap_cookie_t handle, dev_t dev, uint_t flags,
    offset_t offset, size_t len, void **new_devprivate)
{
    struct xxstate *xsp = ddi_get_soft_state(statep,
                  getminor(dev));
    struct xxctx *newctx;

    /* create a new context structure */
    newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
    newctx->xsp = xsp;
    newctx->handle = handle;
    newctx->offset = offset;
    newctx->flags = flags;
    newctx->len = len;
    mutex_enter(&xsp->ctx_lock);
    if (flags & MAP_PRIVATE) {
        /* allocate a private context and initialize it */
        newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
        xxctxinit(newctx);
    } else {
        /* set a pointer to the shared context */
        newctx->context = xsp->ctx_shared;
    }
    mutex_exit(&xsp->ctx_lock);
    /* give at least 1 ms access before context switching */
    devmap_set_ctx_timeout(handle, drv_usectohz(1000));
    /* return the context structure */
    *new_devprivate = newctx;
    return(0);
}
```

devmap_access() 入口点

对转换无效的映射进行访问时，将会调用 devmap_access(9E) 入口点。映射转换在以下几种情况下无效：作为对 mmap(2) 的响应通过 devmap_devmem_setup(9F) 创建映射；通过 fork(2) 复制映射或通过调用 devmap_unload(9F) 显式使映射无效。

devmap_access() 的语法如下所示：

```
int xxdevmap_access(devmap_cookie_t handle, void *devprivate,
    offset_t offset, size_t len, uint_t type, uint_t rw);
```

其中：

handle 用户进程所访问的映射的映射句柄。

devprivate 指向与映射关联的驱动程序专用数据的指针。

offset 所访问映射内的偏移。

len 所访问内存的长度（以字节为单位）。

type 访问操作的类型。

rw 用于指定访问的方向。

系统期望 devmap_access(9E) 调用 devmap_do_ctxmgt(9F) 或 devmap_default_access(9F) 以便在 devmap_access() 返回前装入内存地址转换。对于支持上下文切换的映射，设备驱动程序应调用 devmap_do_ctxmgt()。系统将通过 devmap_access(9E) 向此例程传递所有参数和一个指向驱动程序入口点devmap_contextmgt(9E) 的指针，该指针处理上下文切换。对于不支持上下文切换的映射，驱动程序应调用 devmap_default_access(9F)。devmap_default_access() 的用途是调用 devmap_load(9F) 以装入用户转换。

以下示例说明了 devmap_access(9E) 入口点。该映射分为两个区域。在偏移 OFF_CTXMG 上开始并且长度为 CTXMGT_SIZE 字节的区域支持上下文管理。其余映射支持缺省访问。

示例 11–2 使用 devmap_access() 例程

```
#define OFF_CTXMG      0
#define CTXMGT_SIZE    0x20000
static int
xxdevmap_access(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, uint_t type, uint_t rw)
{
    offset_t diff;
    int    error;

    if ((diff = off - OFF_CTXMG) >= 0 && diff < CTXMGT_SIZE) {
        error = devmap_do_ctxmgt(handle, devprivate, off,
            len, type, rw, xxdevmap_contextmgt);
    } else {
        error = devmap_default_access(handle, devprivate,
            off, len, type, rw);
    }
    return (error);
}
```

devmap_contextmgt() 入口点

devmap_contextmgt(9E) 的语法如下所示：

```
int xxdevmap_contextmgt(devmap_cookie_t handle, void *devprivate,
    offset_t offset, size_t len, uint_t type, uint_t rw);
```

devmap_contextmgt() 应使用当前对设备具有访问权限的映射的句柄调用 devmap_unload(9F)。此方法可使对于该映射的转换无效。通过此方法，可确保下次访问当前映射时针对该映射调用 devmap_access(9E)。对于引起访问事件发生的映射，需

要验证其映射转换。相应地，驱动程序必须为进程请求访问恢复设备上下文。并且，驱动程序必须针对映射的 *handle* 调用 devmap_load(9F)，该映射生成了对此入口点的调用。

访问已通过调用 devmap_load() 对映射转换进行验证的部分映射时不会导致调用 devmap_access()。对 devmap_unload() 的后续调用将使映射转换无效。通过此调用，可再次调用 devmap_access()。

如果 devmap_load() 或 devmap_unload() 返回错误，devmap_contextmgt() 应立即返回该错误。如果设备驱动程序在恢复设备上下文时遇到硬件故障，则应返回 -1。否则，成功处理访问请求后，devmap_contextmgt() 应返回零。如果从 devmap_contextmgt() 返回非零值，则会向进程发送 SIGBUS 或 SIGSEGV。

以下示例说明如何管理单页设备上下文。

注 – xxctxsave() 和 xxctxrestore() 是与设备相关的上下文保存和恢复函数。xxctxsave() 从寄存器中读取数据并将数据保存在软状态结构中。xxctxrestore() 提取软状态结构中保存的数据并将数据写入设备寄存器。请注意，执行读取、写入和保存都需要使用 DDI/DKI 数据访问例程。

示例 11–3 使用 devmap_contextmgt() 例程

```
static int
xxdevmap_contextmgt(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, uint_t type, uint_t rw)
{
    int    error;
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctxp->xsp;
    mutex_enter(&xsp->ctx_lock);
    /* unload mapping for current context */
    if (xsp->current_ctx != NULL) {
        if ((error = devmap_unload(xsp->current_ctx->handle,
            off, len)) != 0) {
            xsp->current_ctx = NULL;
            mutex_exit(&xsp->ctx_lock);
            return (error);
        }
    }
    /* Switch device context - device dependent */
    if (xxctxsave(xsp->current_ctx, off, len) < 0) {
        xsp->current_ctx = NULL;
        mutex_exit(&xsp->ctx_lock);
        return (-1);
    }
    if (xxctxrestore(ctxp, off, len) < 0){
        xsp->current_ctx = NULL;
        mutex_exit(&xsp->ctx_lock);
        return (-1);
    }
    xsp->current_ctx = ctxp;
```

示例 11–3 使用 devmap_contextmgt() 例程 （续）

```
    /* establish mapping for new context and return */
    error = devmap_load(handle, off, len, type, rw);
    if (error)
        xsp->current_ctx = NULL;
    mutex_exit(&xsp->ctx_lock);
    return (error);
}
```

devmap_dup() 入口点

复制设备映射（例如，由调用 devmap_dup(9E) 的用户进程进行复制）时，将会调用 fork(2) 入口点。驱动程序预期会为新映射生成新的驱动程序专用数据。

devmap_dup() 的语法如下所示：

```
int xxdevmap_dup(devmap_cookie_t handle, void *devprivate,
    devmap_cookie_t new-handle, void **new-devprivate);
```

其中：

handle 正在复制的映射的映射句柄。

new-handle 已复制的映射的映射句柄。

devprivate 指向与正在复制的映射关联的驱动程序专用数据的指针。

**new-devprivate* 应设置为指向用于新映射的新驱动程序专用数据的指针。

缺省情况下使用 devmap_dup() 所创建的映射会使其映射转换无效。第一次访问映射时，无效的映射转换会强制调用 devmap_access(9E) 入口点。

以下示例说明了一个典型的 devmap_dup() 例程。

示例 11–4 使用 devmap_dup() 例程

```
static int
xxdevmap_dup(devmap_cookie_t handle, void *devprivate,
    devmap_cookie_t new_handle, void **new_devprivate)
{
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctxp->xsp;
    struct xxctx *newctx;
    /* Create a new context for the duplicated mapping */
    newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
    newctx->xsp = xsp;
    newctx->handle = new_handle;
    newctx->offset = ctxp->offset;
    newctx->flags = ctxp->flags;
    newctx->len = ctxp->len;
    mutex_enter(&xsp->ctx_lock);
    if (ctxp->flags & MAP_PRIVATE) {
```

示例 11–4 使用 devmap_dup() 例程 （续）

```
        newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
        bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
    } else {
        newctx->context = xsp->ctx_shared;
    }
    mutex_exit(&xsp->ctx_lock);
    *new_devprivate = newctx;
    return(0);
}
```

devmap_unmap() 入口点

对映射取消映射时，将会调用 devmap_unmap(9E) 入口点。用户进程退出或调用 munmap(2) 系统调用会导致取消映射。

devmap_unmap() 的语法如下所示：

```
void xxdevmap_unmap(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, devmap_cookie_t new-handle1,
    void **new-devprivate1, devmap_cookie_t new-handle2,
    void **new-devprivate2);
```

其中：

handle 正在释放的映射的映射句柄。

devprivate 指向与映射关联的驱动程序专用数据的指针。

off 逻辑设备内存中取消映射开始处的偏移。

len 所取消映射的内存的长度（以字节为单位）。

new-handle1 系统用来描述新区域的句柄，该新区域在 *off* - 1 位置结束。*new-handle1* 的值可以为 NULL。

new-devprivate1 要由驱动程序通过用于新区域的专用驱动程序映射数据进行填充的指针，该新区域在 *off* -1 位置结束。如果 *new-handle1* 为 NULL，则会忽略 *new-devprivate1*。

new-handle2 系统用来描述新区域的句柄，该新区域在 *off* + *len* 位置开始。*new-handle2* 的值可以为 NULL。

new-devprivate2 要由驱动程序通过用于新区域的驱动程序专用映射数据进行填充的指针，该新区域在 *off* + *len* 位置开始。如果 *new-handle2* 为 NULL，则会忽略 *new-devprivate2*。

devmap_unmap() 例程预期会释放通过 devmap_map(9E) 或 devmap_dup(9E) 创建此映射时分配的任何驱动程序专用资源。如果只是取消映射部分映射，则驱动程序必须在释放旧的专用数据之前为其余映射分配新的专用数据。不必针对已释放的映射的句柄调用

devmap_unload(9F)，即使此句柄指向具有有效转换的映射时也是如此。不过，为了避免将来出现 devmap_access(9E) 问题，设备驱动程序应确保当前的映射表示形式设置为“无当前映射”。

以下示例说明了一个典型的 devmap_unmap() 例程。

示例 11–5 使用 devmap_unmap() 例程

```
static void
xxdevmap_unmap(devmap_cookie_t handle, void *devprivate,
    offset_t off, size_t len, devmap_cookie_t new_handle1,
    void **new_devprivate1, devmap_cookie_t new_handle2,
    void **new_devprivate2)
{
    struct xxctx *ctxp = devprivate;
    struct xxstate *xsp = ctxp->xsp;
    mutex_enter(&xsp->ctx_lock);
    /*
     * If new_handle1 is not NULL, we are unmapping
     * at the end of the mapping.
     */
    if (new_handle1 != NULL) {
        /* Create a new context structure for the mapping */
        newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
        newctx->xsp = xsp;
        if (ctxp->flags & MAP_PRIVATE) {
            /* allocate memory for the private context and copy it */
            newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
            bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
        } else {
            /* point to the shared context */
            newctx->context = xsp->ctx_shared;
        }
        newctx->handle = new_handle1;
        newctx->offset = ctxp->offset;
        newctx->len = off - ctxp->offset;
        *new_devprivate1 = newctx;
    }
    /*
     * If new_handle2 is not NULL, we are unmapping
     * at the beginning of the mapping.
     */
    if (new_handle2 != NULL) {
        /* Create a new context for the mapping */
        newctx = kmem_alloc(sizeof (struct xxctx), KM_SLEEP);
        newctx->xsp = xsp;
        if (ctxp->flags & MAP_PRIVATE) {
            newctx->context = kmem_alloc(XXCTX_SIZE, KM_SLEEP);
            bcopy(ctxp->context, newctx->context, XXCTX_SIZE);
        } else {
            newctx->context = xsp->ctx_shared;
        }
        newctx->handle = new_handle2;
        newctx->offset = off + len;
        newctx->flags = ctxp->flags;
        newctx->len = ctxp->len - (off + len - ctxp->off);
```

示例 11–5 使用 devmap_unmap() 例程 （续）

```
        *new_devprivate2 = newctx;
    }
    if (xsp->current_ctx == ctxp)
        xsp->current_ctx = NULL;
    mutex_exit(&xsp->ctx_lock);
    if (ctxp->flags & MAP_PRIVATE)
        kmem_free(ctxp->context, XXCTX_SIZE);
    kmem_free(ctxp, sizeof (struct xxctx));
}
```

将用户映射与驱动程序通知关联

用户进程通过 mmap(2) 请求到设备的映射时，将会调用驱动程序的 segmap(9E) 入口点。如果驱动程序需要管理设备上下文，则在设置内存映射时，驱动程序必须使用 ddi_devmap_segmap(9F) 或 devmap_setup(9F)。这两个函数都会调用驱动程序的 devmap(9E) 入口点，该入口点使用 devmap_devmem_setup(9F) 将设备内存与用户映射关联。有关如何映射设备内存的详细信息，请参见第 10 章，映射设备和内核内存。

驱动程序必须向系统通知 devmap_callback_ctl(9S) 入口点才能获取对用户映射的访问通知。驱动程序通过向devmap_devmem_setup(9F) 提供一个指向 devmap_callback_ctl(9S) 结构的指针来通知系统。devmap_callback_ctl(9S) 结构描述了一组用于上下文管理的入口点。系统通过调用这些入口点来通知设备驱动程序管理有关设备映射的事件。

系统会将每个映射与一个映射句柄关联。此句柄会传递给每个用于上下文管理的入口点。该映射句柄可用来使映射转换无效和对映射转换进行验证。如果驱动程序使映射转换**无效**，则会向该驱动程序通知将来对映射的任何访问。如果驱动程序对映射转换进行**验证**，则不再向该驱动程序通知对映射的访问。映射总是在映射转换无效的情况下创建，以便第一次访问映射时将会通知驱动程序。

以下示例说明如何使用设备上下文管理接口设置映射。

示例 11–6 支持上下文管理的 devmap(9E) 入口点

```
static struct devmap_callback_ctl xx_callback_ctl = {
    DEVMAP_OPS_REV, xxdevmap_map, xxdevmap_access,
    xxdevmap_dup, xxdevmap_unmap
};

static int
xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off,
    size_t len, size_t *maplen, uint_t model)
{
    struct xxstate *xsp;
    uint_t rnumber;
    int    error;
```

示例 11–6 支持上下文管理的 devmap(9E) 入口点 （续）

```
    /* Setup data access attribute structure */
    struct ddi_device_acc_attr xx_acc_attr = {
        DDI_DEVICE_ATTR_V0,
        DDI_NEVERSWAP_ACC,
        DDI_STRICTORDER_ACC
    };
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    len = ptob(btopr(len));
    rnumber = 0;
    /* Set up the device mapping */
    error = devmap_devmem_setup(handle, xsp->dip, &xx_callback_ctl,
        rnumber, off, len, PROT_ALL, 0, &xx_acc_attr);
    *maplen = len;
    return (error);
}
```

管理映射访问

用户进程访问没有有效的映射转换的内存映射区域中的地址时，将会通知设备驱动程序。访问事件发生时，必须使当前对设备具有访问权限的进程的映射转换无效。必须恢复请求访问设备的进程的设备上下文。并且，必须对请求访问的进程的映射转换进行验证。

函数 devmap_load(9F) 和 devmap_unload(9F) 用于验证映射转换和使其无效。

devmap_load() 入口点

devmap_load(9F) 的语法如下所示：

```
int devmap_load(devmap_cookie_t handle, offset_t offset,
    size_t len, uint_t type, uint_t rw);
```

devmap_load() 可以验证对于 handle、offset 和 len 指定的映射页的映射转换。通过验证对这些页的映射转换，驱动程序将告知系统不要拦截对这些映射页的访问。并且，系统不得在未通知设备驱动程序的情况下允许继续进行访问。

必须通过映射的偏移和句柄调用 devmap_load()，该映射可生成访问事件以便完成访问。如果不针对此句柄调用 devmap_load(9F)，则不会验证映射转换，并且进程将收到 SIGBUS。

devmap_unload() 入口点

devmap_unload(9F) 的语法如下所示：

```
int devmap_unload(devmap_cookie_t handle, offset_t offset, size_t len);
```

devmap_unload() 可使对 *handle*、*offset* 和 *len* 指定的映射页的映射转换无效。通过使对这些页的映射转换无效，设备驱动程序将告知系统拦截对这些映射页的访问。并且，下次通过调用 devmap_access(9E) 入口点访问这些映射页时，系统必须通知设备驱动程序。

对于这两个函数而言，请求会影响包含 *offset* 的整页，直到包含由 *offset* + *len* 所表示的最后一个字节的整页（包含该页）。设备驱动程序必须确保对于所映射的每页设备内存而言，在任意时刻仅有一个进程具有有效转换。

如果成功，两个函数都将返回零。但是，如果在对映射转换进行验证或使其无效时出现错误，则该错误将返回给设备驱动程序。设备驱动程序必须将此错误返回给系统。

◆ ◆ ◆ 第 12 章

12 电源管理

电源管理提供控制和管理计算机系统或设备的电源使用情况的功能。使用电源管理，可使系统在空闲时消耗较少的电量，在未使用时完全关闭电源，从而节省能源。例如，桌面计算机系统耗电量很大，但经常处于空闲状态，尤其是在夜间。电源管理软件可以检测到系统未被使用的情况。因此，电源管理可以关闭系统或其中某些组件的电源。

本章介绍有关以下主题的信息：

电源管理框架

Oracle Solaris 电源管理框架依靠设备驱动程序来实现特定于设备的电源管理功能。该框架分两部分实现：

- 设备电源管理－自动关闭未使用的设备，以减少能耗
- 系统电源管理－当整个系统处于空闲状态时，自动关闭计算机

设备电源管理

通过该框架，设备在经过指定的空闲时间间隔后可降低能耗。在电源管理过程中，系统软件会检查空闲设备。电源管理框架会导出接口，通过这些接口，可以在系统软件与设备驱动程序之间进行通信。

Oracle Solaris 电源管理框架提供了下列设备电源管理功能：

- 适用于电源可管理设备且与设备无关的模型。

- dtpower(1M)，一种用于配置工作站电源管理的工具。
- 一组 DDI 接口，用于通知框架电源管理兼容性和空闲状态。

系统电源管理

系统电源管理可在关闭系统电源之前保存系统状态。因此，系统可以在重新打开时立即返回到相同状态。

要关闭整个系统并返回到关闭前的状态，请执行以下步骤：

- 停止内核线程和用户进程。以后再重新启动这些线程和进程。
- 将系统中所有设备的硬件状态保存到磁盘。以后再恢复该状态。

仅适用于 SPARC – 当前，仅在 Oracle Solaris OS 支持的某些 SPARC 系统上实现了系统电源管理。

Oracle Solaris OS 中的系统电源管理框架提供了下列系统电源管理功能：

- 与平台无关的系统空闲模型。
- 为设备驱动程序提供的一组可以覆盖用于确定哪些驱动程序具有硬件状态的方法的接口。
- 一组允许框架对驱动程序进行调用以便保存和恢复设备状态的接口。
- 用于通知进程已执行恢复操作的机制。

设备电源管理模型

以下各节详细介绍设备电源管理模型。该模型包括以下元素：

- 组件
- 空闲
- 电源级别
- 相关性
- Policy（策略）
- 设备电源管理接口
- 电源管理入口点

电源管理组件

如果在设备处于空闲状态时可以减少设备能耗，则该设备是电源可管理设备。从概念上讲，电源可管理设备由许多电源可管理硬件单元组成，这些硬件单元称为**组件**。

设备驱动程序通知系统有关设备组件及其相关电源级别的信息。因此，在驱动程序初始化期间，驱动程序会在其 attach(9E) 入口点中创建 pm-components(9P) 属性。

大多数电源可管理设备仅实现单个组件。例如，磁盘就是一个电源可管理的单组件设备，当磁盘处于空闲状态时，可以停止磁盘主轴马达以节省电能。

如果一个设备具有多个可单独控制的电源可管理单元，则该设备应实现多个组件。

例如，配有监视器的帧缓存器卡就是一个电源可管理的双组件设备。帧缓存器电子设备是第一个组件 [组件 0]。未使用帧缓存器电子设备时，其能耗将会降低。监视器是第二个组件 [组件 1]。未使用监视器时，监视器也可以进入低能耗模式。系统将帧缓存器电子设备和监视器视为一个由两个组件组成的设备。

多个电源管理组件

对于电源管理框架而言，所有组件均“一视同仁”，并且组件之间完全无关。如果组件状态不完全兼容，则设备驱动程序必须确保不会出现不需要的状态组合。例如，帧缓存器/监视器卡具有以下几种可能状态：`D0`、`D1`、`D2` 和 `D3`。与卡连接的监视器具有以下几种可能状态：`On`、`Standby`、`Suspend` 和 `Off`。这些状态并不一定相互兼容。例如，如果监视器处于 `On` 状态，则帧缓存器必须处于 `D0` 状态（即完全打开）。如果在帧缓存器处于 `D3` 状态时，其驱动程序收到一个请求，要求打开监视器电源使监视器处于 `On` 状态，则在将监视器设置为 `On` 之前，驱动程序必须调用 `pm_raise_power(9F)` 才能启动帧缓存器。如果系统在监视器处于 `On` 状态时请求降低帧缓存器的电能供给，则驱动程序必须拒绝该请求。

电源管理状态

每个设备组件都可处于以下两种状态之一：**繁忙**或**空闲**。设备驱动程序通过调用 `pm_busy_component(9F)` 和 `pm_idle_component(9F)` 通知框架设备状态的更改。最初创建组件时，组件被视为空闲状态。

电源级别

通过设备导出的 `pm-components` 属性，设备电源管理框架可了解设备支持的电源级别。电源级别值必须是正整数。对电源级别的解释由设备驱动程序编写者确定。在 `pm-components` 属性中必须按单一递增顺序列出电源级别。该框架将 0 电源级别解释为关闭。如果框架由于相关性必须打开设备电源，则它会将每个组件都设置为其最高电源级别。

以下示例显示了某驱动程序的 `.conf` 文件中的 `pm-components` 项，该驱动程序实现了一个电源管理组件（即磁盘主轴马达）。磁盘轴马达是组件 0。该轴马达支持两个电源级别。这两个级别表示“停止”和“全速旋转”。

示例 12–1 pm-component 项样例

```
pm-components="NAME=Spindle Motor", "0=Stopped", "1=Full Speed";
```

以下示例说明如何在驱动程序的 `attach()` 例程中实现示例 12–1。

示例 12–2 使用 pm-components 属性的 attach(9E) 例程

```
static char *pmcomps[] = {
    "NAME=Spindle Motor",
    "0=Stopped",
    "1=Full Speed"
};
/* ... */
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    /* ... */
    if (ddi_prop_update_string_array(DDI_DEV_T_NONE, dip,
        "pm-components", &pmcomp[0],
        sizeof (pmcomps) / sizeof (char *)) != DDI_PROP_SUCCESS)
        goto failed;
    /* ... */
```

以下示例显示了实现两个组件的帧缓存器。组件 0 是支持四个不同电源级别的帧缓存器电子设备。组件 1 表示所连接的监视器的电源管理状态。

示例 12–3 多组件 pm-components 项

```
pm-components="NAME=Frame Buffer", "0=Off", "1=Suspend", \
    "2=Standby", "3=On",
    "NAME=Monitor", "0=Off", "1=Suspend", "2=Standby", "3=On";
```

首次连接设备驱动程序时，框架并不了解设备的电源级别。在以下情况下，会进行电源转换：

- 驱动程序调用 pm_raise_power(9F) 或 pm_lower_power(9F)。
- 由于超出时间阈值，框架降低了组件的电源级别。
- 另一个设备更换了电源，而这两个设备之间存在相关性。请参见第 194 页中的 “电源管理相关性”。

进行电源转换后，框架将开始跟踪每个设备组件的电源级别。如果驱动程序已通知框架电源级别，则也会进行跟踪。驱动程序通过调用 pm_power_has_changed(9F) 通知框架电源级别的更改。

系统将计算每个可能的电源转换的缺省阈值。这些阈值基于系统空闲阈值。当组件电源级别未知时，将使用基于系统空闲阈值的其他缺省阈值。

电源管理相关性

某些设备的电源应仅在关闭其他设备的电源时关闭。例如，如果允许关闭 CD-ROM 驱动器的电源，则可能会丢失一些必需功能，如弹出 CD 的功能。

为了防止设备独立关闭电源，可以使该设备依赖于电源可能保持打开的其他设备。通常，设备依赖于帧缓存器，因为在用户使用系统时监视器通常处于打开状态。

其中，*dependent-phys-path* 是电源保持打开状态的设备，如 CD-ROM 驱动器。*phys-path* 表示要依赖于其电源状态的设备，如帧缓存器。

通过以下语法，您可以用常规方式来指定相关项：

```
device-dependency-property property phys-path
```

这种项要求任何导出属性 *property* 的设备都必须依赖于 *phys-path* 指定的设备。由于此相关性尤其适用于可移除介质设备，因此缺省情况下 `/etc/power.conf` 包含以下行：

```
device_dependent-property  removable-media  /dev/fb
```

使用此语法，除非关闭控制台帧缓存器的电源，否则无法关闭导出 `removable-media` 属性的设备的电源。

有关更多信息，请参见 `removable-media(9P)` 手册页。

设备的自动电源管理

如果启用了自动化电源管理，则具有 `pm-components(9P)` 属性的所有设备都将自动使用电源管理。当组件空闲一段缺省时间后，组件将自动降低到下一个最低电源级别。缺省时间段由电源管理框架计算，用于在系统空闲阈值内将整个设备设置为其最低能耗状态。

注 – 缺省情况下，1999 年 7 月 1 日后首次发布的所有 SPARC 桌面系统上都启用了自动电源管理。对于所有其他系统，缺省情况下禁用此功能。

设备电源管理接口

支持包含电源可管理组件的设备的设备驱动程序必须创建 `pm-components(9P)` 属性。该属性向系统指明设备包含电源可管理组件。`pm-components` 还向系统指明可用的电源级别。通常，驱动程序通过从其 `attach(9E)` 入口点调用 `ddi_prop_update_string_array(9F)` 来通知系统。另一种通知系统的方法是使用 `driver.conf(4)` 文件。有关详细信息，请参见 `pm-components(9P)` 手册页。

繁忙-空闲状态转换

驱动程序必须始终使框架了解从空闲到繁忙或从繁忙到空闲的设备状态转换。进行这些转换的位置完全特定于设备。繁忙与空闲状态之间的转换取决于设备的性质以及特定组件具备的状态转换特性。例如，SCSI 磁盘目标驱动程序通常导出单个组件，该组件表示 SCSI 目标磁盘驱动器是启动状态还是停止状态。当驱动器有未解决的请求时，该组件将标记为繁忙。完成最后一个排队请求后，该组件将标记为空闲。某些组件创建后从未标记为繁忙。例如，`pm-components(9P)` 创建的组件就一直处于空闲状态。

`pm_busy_component(9F)` 和 `pm_idle_component(9F)` 接口将通知电源管理框架繁忙/空闲状态转换。`pm_busy_component(9F)` 调用的语法如下所示：

```
int pm_busy_component(dev_info_t *dip, int component);
```

pm_busy_component(9F) 将 *component* 标记为繁忙。当组件为繁忙状态时，不应关闭该组件的电源。如果已关闭组件电源，则将该组件标记为繁忙不会更改电源级别。为此，驱动程序需要调用 pm_raise_power(9F)。对 pm_busy_component(9F) 的调用具有累积性，因此要使组件处于空闲状态，需要调用相应次数的 pm_idle_component。

pm_idle_component(9F) 例程的语法如下所示：

```
int pm_idle_component(dev_info_t *dip, int component);
```

pm_idle_component(9F) 将 *component* 标记为空闲。可以关闭空闲组件的电源。要使组件处于空闲状态，必须针对 pm_busy_component(9F) 的每次调用调用 pm_idle_component(9F) 一次。

设备能耗状态转换

设备驱动程序可以调用 pm_raise_power(9F) 来请求将组件至少设置为给定的电源级别。使用已关闭电源的组件之前，必须采用这种方式设置电源级别。例如，如果已关闭磁盘电源，则 SCSI 磁盘目标驱动程序的 read(9E) 例程可能需要启动磁盘。pm_raise_power(9F) 函数请求电源管理框架将设备能耗状态转换为较高的电源级别。通常，由框架来降低组件的电源级别。但是，设备驱动程序在分离时应调用 pm_lower_power(9F)，以便尽可能地降低未使用设备的能耗。

对于某些设备来说，关闭电源可能会产生风险。例如，某些磁带机在关闭电源时会损坏磁带。同样，某些磁盘驱动器在开关电源过程中的容错能力有限，因为每次开关电源都会导致磁头停放。应使用 no-involuntary-power-cycles(9P) 属性通知系统，设备驱动程序应控制设备的所有关开机循环。此方法可防止在分离设备驱动程序时切断设备电源，除非驱动程序已从其 detach(9E) 入口点调用 pm_lower_power(9F) 关闭设备电源。

驱动程序发现某个操作所需组件的电源级别不够高时，会调用 pm_raise_power(9F) 函数。该接口会使驱动程序将组件的当前电源级别提高到所需级别。该调用还会将依赖于该设备的所有设备恢复到全功率。

如果在不再需要访问某个设备后分离该设备，则将调用 pm_lower_power(9F)。调用 pm_lower_power(9F) 可将每个组件设置为最低电源级别，从而使设备在未使用时尽可能少地消耗电量。必须从 detach() 入口点调用 pm_lower_power() 函数。如果从驱动程序的任何其他部分调用 pm_lower_power() 函数，该函数不会起作用。

调用 pm_power_has_changed(9F) 函数可通知框架有关电源转换的情况。转换可能是由于设备更改了自己的电源级别而导致。转换也可能是由于暂停/恢复等操作而导致。pm_power_has_changed(9F) 的语法与 pm_raise_power(9F) 的语法相同。

power() 入口点

电源管理框架使用 power(9E) 入口点。

power() 使用以下语法：

```
int power(dev_info_t *dip, int component, int level);
```

需要更改组件的电源级别时，系统将调用 power(9E) 入口点。该入口点执行的操作特定于设备驱动程序。在上面提到的 SCSI 目标磁盘驱动程序示例中，如果将电源级别设置为 0，则会发送 SCSI 命令停止磁盘运转，而如果将电源级别设置为全电源级别，则会发送 SCSI 命令启动磁盘。

如果电源转换导致设备丢失状态，则驱动程序必须将任何必需的状态保存在内存中，以便将来恢复。如果电源转换要求先恢复保存的状态，然后才能再次使用设备，则驱动程序必须恢复该状态。该框架并未对哪些电源事务会导致丢失自动电源管理设备的状态作出假设，也未对哪些电源事务会要求恢复自动电源管理设备的状态作出假设。以下示例显示了一个 power() 例程样例。

示例 12–4 将 power() 例程用于单组件设备

```
int
xxpower(dev_info_t *dip, int component, int level)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);
    /*
     * Make sure the request is valid
     */
    if (!xx_valid_power_level(component, level))
        return (DDI_FAILURE);
    mutex_enter(&xsp->mu);
    /*
     * If the device is busy, don't lower its power level
     */
    if (xsp->xx_busy[component] &&
        xsp->xx_power_level[component] > level) {
        mutex_exit(&xsp->mu);
        return (DDI_FAILURE);
    }

    if (xsp->xx_power_level[component] != level) {
        /*
         * device- and component-specific setting of power level
         * goes here
         */
        xsp->xx_power_level[component] = level;
    }
    mutex_exit(&xsp->mu);
    return (DDI_SUCCESS);
}
```

以下示例是包含两个组件的设备的 power() 例程，其中，组件 1 为打开状态时，组件 0 必须也为打开状态。

示例 12–5 多组件设备的 power (9E) 例程

```
int
xxpower(dev_info_t *dip, int component, int level)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);
    /*
     * Make sure the request is valid
     */
    if (!xx_valid_power_level(component, level))
        return (DDI_FAILURE);
    mutex_enter(&xsp->mu);
    /*
     * If the device is busy, don't lower its power level
     */
    if (xsp->xx_busy[component] &&
        xsp->xx_power_level[component] > level) {
        mutex_exit(&xsp->mu);
        return (DDI_FAILURE);
    }
    /*
     * This code implements inter-component dependencies:
     * If we are bringing up component 1 and component 0
     * is off, we must bring component 0 up first, and if
     * we are asked to shut down component 0 while component
     * 1 is up we must refuse
     */
    if (component == 1 && level > 0 && xsp->xx_power_level[0] == 0) {
        xsp->xx_busy[0]++;
        if (pm_busy_component(dip, 0) != DDI_SUCCESS) {
            /*
             * This can only happen if the args to
             * pm_busy_component()
             * are wrong, or pm-components property was not
             * exported by the driver.
             */
            xsp->xx_busy[0]--;
            mutex_exit(&xsp->mu);
            cmn_err(CE_WARN, "xxpower pm_busy_component()
                failed");
            return (DDI_FAILURE);
        }
        mutex_exit(&xsp->mu);
        if (pm_raise_power(dip, 0, XX_FULL_POWER_0) != DDI_SUCCESS)
            return (DDI_FAILURE);
        mutex_enter(&xsp->mu);
    }
    if (component == 0 && level == 0 && xsp->xx_power_level[1] != 0) {
        mutex_exit(&xsp->mu);
        return (DDI_FAILURE);
    }
```

示例 12–5 多组件设备的 power (9E) 例程 （续）

```
    if (xsp->xx_power_level[component] != level) {
        /*
         * device- and component-specific setting of power level
         * goes here
         */
        xsp->xx_power_level[component] = level;
    }
    mutex_exit(&xsp->mu);
    return (DDI_SUCCESS);
}
```

系统电源管理模型

本节详细介绍系统电源管理模型。该模型包括以下组件：

- 自动关闭阈值
- 繁忙状态
- 硬件状态
- Policy（策略）
- 电源管理入口点

自动关闭阈值

经过一段可配置空闲时间后，系统可以自动关闭（即关闭电源）。该时间段称为**自动关闭阈值**。缺省情况下，将对在 1995 年 10 月 1 日到 1999 年 7 月 1 日间首次发布的 SPARC 桌面系统启用此行为。

繁忙状态

可以采用几种方法度量系统的繁忙状态。当前支持的内置度量标准项包括键盘字符、鼠标活动、tty 字符、平均负荷值、磁盘读取和 NFS 请求。其中任何一项都可使系统处于繁忙状态。除内置度量标准外，还定义了一个接口，用于运行用户指定的可以表明系统处于繁忙状态的进程。

硬件状态

导出 reg 属性的设备被视为具有硬件状态，关闭系统之前，必须保存该硬件状态。没有 reg 属性的设备被视为无状态设备。但是，设备驱动程序可以另外一种方式处理这种情况。

如果驱动程序导出值为 needs-suspend-resume 的 pm-hardware-state 属性，则必须调用具有硬件状态但没有 reg 属性的设备（如 SCSI 驱动程序），才能保存并恢复状态。否则，缺少 reg 属性即表示设备没有硬件状态。有关设备属性的信息，请参见第 4 章，属性。

具有 reg 属性但没有硬件状态的设备可以导出值为 no-suspend-resume 的 pm-hardware-state 属性。将 no-suspend-resume 与 pm-hardware-state 属性配合使用，可防止框架调用驱动程序来保存并恢复该状态。有关电源管理属性的更多信息，请参见 pm-components(9P) 手册页。

系统的自动电源管理

如果系统空闲时间达到了**自动关闭阈值**分钟数，则系统将关闭：

系统电源管理使用的入口点

系统电源管理将命令DDI_SUSPEND 传递给 detach(9E) 驱动程序入口点，以请求驱动程序保存设备硬件状态。系统电源管理将命令 DDI_RESUME 传递给 attach(9E) 驱动程序入口点，以请求驱动程序恢复设备硬件状态。

detach() 入口点

detach(9E) 的语法如下所示：

```
int detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
```

具有 reg 属性或 pm-hardware-state 属性设置为 needs-suspend-resume 的设备必须能够保存设备的硬件状态。框架调用驱动程序的 detach(9E) 入口点使驱动程序保存状态，以便在系统电源重新打开后进行恢复。要处理 DDI_SUSPEND 命令，detach(9E) 必须执行以下任务：

- 在设备恢复之前，阻止启动进一步操作，但 dump(9E) 请求除外。
- 一直等到未完成的操作完成为止。如果可以重新启动未完成的操作，则可以中止该操作。
- 取消待处理的任何超时和回调。
- 将任何易失的硬件状态保存到内存。该状态包含设备寄存器的内容，此外还可以包含下载的固件。

如果驱动程序无法暂停设备并将其状态保存到内存，则驱动程序必须返回 DDI_FAILURE。然后，框架将中止系统电源管理操作。

在某些情况下，关闭设备电源存在一定风险。例如，如果关闭内含磁带的磁带机电源，则该磁带可能会损坏。在这种情况下，attach(9E) 应执行以下操作：

- 调用 ddi_removing_power(9F) 以确定 DDI_SUSPEND 命令是否会关闭设备的电源。
- 确定关闭电源是否会产生问题。

如果上述两种操作的结果都是肯定的，则应拒绝 DDI_SUSPEND 请求。示例 12–6 给出了使用 ddi_removing_power(9F) 检查 DDI_SUSPEND 命令是否会产生问题的 attach(9E) 例程。

必须接受转储请求。框架使用 dump(9E) 入口点写出包含内存内容的状态文件。有关使用该入口点时对设备驱动程序强加的限制，请参见 dump(9E) 手册页。

使用 DDI_SUSPEND 命令调用电源可管理组件的 detach(9E) 入口点时，应保存关闭设备电源时的状态。驱动程序应取消待处理的超时。驱动程序还应禁止对 pm_raise_power(9F) 的任何调用，但 dump(9E) 请求除外。通过使用 DDI_RESUME 命令调用 attach(9E) 来恢复设备时，可以恢复超时以及对 pm_raise_power () 的调用。驱动程序必须掌握其足够的状态信息，才能够正确处理这种可能发生的情况。以下示例显示了实现 DDI_SUSPEND 命令的 detach(9E) 例程。

示例 12–6 实现 DDI_SUSPEND 的 detach(9E) 例程

```
int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    struct xxstate *xsp;
    int instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);

    switch (cmd) {
    case DDI_DETACH:
        /* ... */
    case DDI_SUSPEND:
        /*
         * We do not allow DDI_SUSPEND if power will be removed and
         * we have a device that damages tape when power is removed
         * We do support DDI_SUSPEND for Device Reconfiguration.
         */
        if (ddi_removing_power(dip) && xxdamages_tape(dip))
            return (DDI_FAILURE);
        mutex_enter(&xsp->mu);
        xsp->xx_suspended = 1;  /* stop new operations */
        /*
         * Sleep waiting for all the commands to be completed
         *
         * If a callback is outstanding which cannot be cancelled
         * then either wait for the callback to complete or fail the
         * suspend request
         *
         * This section is only needed if the driver maintains a
         * running timeout
         */
        if (xsp->xx_timeout_id) {
            timeout_id_t temp_timeout_id = xsp->xx_timeout_id;
            xsp->xx_timeout_id = 0;
```

示例 12–6 实现 DDI_SUSPEND 的 detach(9E) 例程 （续）

```
            mutex_exit(&xsp->mu);
            untimeout(temp_timeout_id);
            mutex_enter(&xsp->mu);
        }
        if (!xsp->xx_state_saved) {
            /*
             * Save device register contents into
             * xsp->xx_device_state
             */
        }
        mutex_exit(&xsp->mu);
        return (DDI_SUCCESS);
    default:
        return (DDI_FAILURE);
}
```

attach() 入口点

attach(9E) 的语法如下所示：

```
int attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
```

恢复系统电源后，每个具有 reg 属性或具有值为 needs-suspend-resume 的 pm-hardware-state 属性的设备都会使用 DDI_RESUME 命令值调用其 attach(9E) 入口点。如果系统关闭被中止，则即使尚未关闭电源，也会调用每个暂停的驱动程序以进行恢复。因此，attach(9E) 中的恢复代码不能对系统是否已实际断电作出任何假设。

电源管理框架认为组件的电源级别在执行 DDI_RESUME 时未知。根据设备性质，驱动程序编写者有两种选择：

- 如果驱动程序无需打开组件电源即可确定设备组件的实际电源级别（如通过读取寄存器），则驱动程序应通过调用 pm_power_has_changed(9F) 来通知框架每个组件的电源级别。
- 如果驱动程序无法确定组件的电源级别，则驱动程序应在首次访问各个组件之前，在内部将每个组件标记为未知并调用 pm_raise_power(9F)。

以下示例显示了使用 DDI_RESUME 命令的 attach(9E) 例程。

示例 12–7 实现 DDI_RESUME 的 attach(9E) 例程

```
int
xxattach(devinfo_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    int    instance;

    instance = ddi_get_instance(dip);
    xsp = ddi_get_soft_state(statep, instance);

    switch (cmd) {
```

示例 12–7 实现 DDI_RESUME 的 attach(9E) 例程 （续）

```
    case DDI_ATTACH:
    /* ... */
    case DDI_RESUME:
        mutex_enter(&xsp->mu);
        if (xsp->xx_pm_state_saved) {
            /*
             * Restore device register contents from
             * xsp->xx_device_state
             */
        }
        /*
         * This section is optional and only needed if the
         * driver maintains a running timeout
         */
        xsp->xx_timeout_id = timeout( /* ... */ );

        xsp->xx_suspended = 0;        /* allow new operations */
        cv_broadcast(&xsp->xx_suspend_cv);
        /* If it is possible to determine in a device-specific
         * way what the power levels of components are without
         * powering the components up,
         * then the following code is recommended
         */
        for (i = 0; i < num_components; i++) {
            xsp->xx_power_level[i] = xx_get_power_level(dip, i);
            if (xsp->xx_power_level[i] != XX_LEVEL_UNKNOWN)
                (void) pm_power_has_changed(dip, i,
                    xsp->xx_power_level[i]);
        }
        mutex_exit(&xsp->mu);
        return(DDI_SUCCESS);
    default:
        return(DDI_FAILURE);
    }
}
```

注 – detach(9E) 和 attach(9E) 接口也可用于恢复被停止的系统。

电源管理设备访问示例

如果支持电源管理，并且按示例 12–6 和 示例 12–7 中的方式使用 detach(9E) 和 attach(9E)，则可以从用户上下文（例如， read(2)、write(2) 和 ioctl(2)）访问该设备。

以下示例演示了该方法。该示例假定要执行的操作需要以电源级别 `level` 运行的组件 `component`。

示例 12–8 设备访问

```
mutex_enter(&xsp->mu);
/*
 * Block command while device is suspended by DDI_SUSPEND
 */
while (xsp->xx_suspended)
    cv_wait(&xsp->xx_suspend_cv, &xsp->mu);
/*
 * Mark component busy so xx_power() will reject attempt to lower power
 */
xsp->xx_busy[component]++;
if (pm_busy_component(dip, component) != DDI_SUCCESS) {
    xsp->xx_busy[component]--;
    /*
     * Log error and abort
     */
}
if (xsp->xx_power_level[component] < level) {
    mutex_exit(&xsp->mu);
    if (pm_raise_power(dip, component, level) != DDI_SUCCESS) {
        /*
         * Log error and abort
         */
    }
    mutex_enter(&xsp->mu);
}
```

当设备操作（例如，设备的中断处理程序执行的操作）完成时，可以使用以下示例中的代码段。

示例 12–9 设备操作完成

```
/*
 * For each command completion, decrement the busy count and unstack
 * the pm_busy_component() call by calling pm_idle_component(). This
 * will allow device power to be lowered when all commands complete
 * (all pm_busy_component() counts are unstacked)
 */
xsp->xx_busy[component]--;
if (pm_idle_component(dip, component) != DDI_SUCCESS) {
    xsp->xx_busy[component]++;
    /*
     * Log error and abort
     */
}
/*
 * If no more outstanding commands, wake up anyone (like DDI_SUSPEND)
 * waiting for all commands to  be completed
 */
```

电源管理控制流程

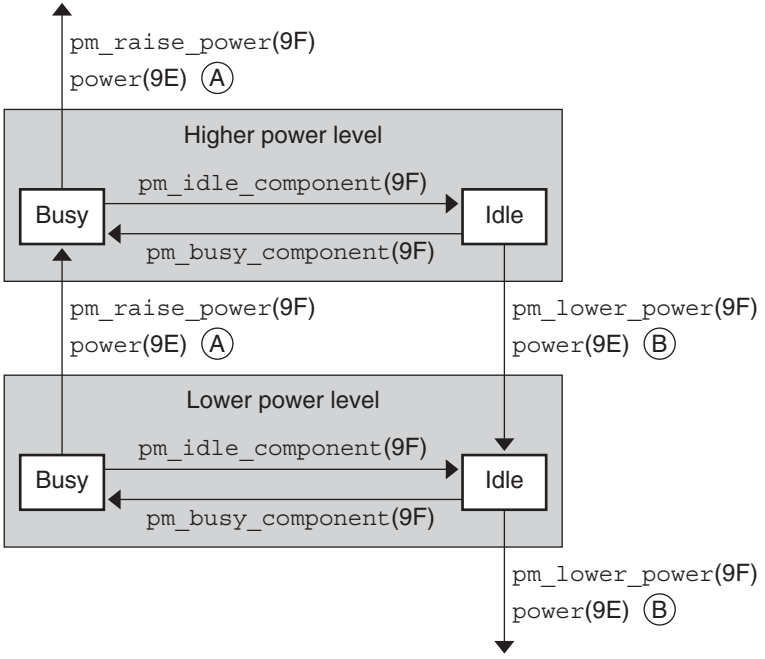
图 12–1 说明了电源管理框架中的控制流程。

完成组件活动后，驱动程序可以调用 pm_idle_component(9F) 将该组件标记为空闲。如果组件在其阈值时间内一直处于空闲状态，则框架可以将该组件的能耗降低到下一个较低级别。框架调用 power(9E) 函数将组件的能耗设置为支持的下一个较低电源级别（如果存在较低级别）。当组件处于繁忙状态时，驱动程序的 power(9E) 函数应拒绝任何降低该组件电源级别的尝试。在转换到较低级别之前 power(9E) 函数应保存可能在转换过程中丢失的任何状态。

需要较高级别的组件时，驱动程序将调用 pm_busy_component(9F)。此调用将阻止框架进一步降低能耗，然后针对组件调用 pm_raise_power(9F)。在对 pm_raise_power(9F) 的调用返回之前，框架接着调用 power(9E) 以提高组件的能耗。驱动程序的 power(9E) 代码必须恢复在较低级别中丢失、但在较高级别中需要的任何状态。

分离某个驱动程序时，该驱动程序应针对每个组件调用 pm_lower_power(9F)，以将其能耗降低到最低级别。在对 pm_lower_power(9F) 的调用返回之前，框架可以随后调用驱动程序的 power(9E) 例程以降低组件的能耗。

图 12–1 电源管理概念状态图



Ⓐ power(9E) can be called by the framework to raise the power level of a component as a result of a dependency or can be called by the framework as a result of the driver's call to pm_raise_power(9F).

Ⓑ power(9E) can be called by the framework to lower the power level of a component as a result of a device idleness, or can be called by the framework as a result of the driver's call to pm_lower_power(9F) when the driver is detaching.

Note:
9E routines are always called by the framework.
9F routines are always called by the driver.

电源管理接口的更改

在 Solaris 8 发行版之前，设备的电源管理不是自动的。开发者必须为要管理其电源的每个设备在 /etc/power.conf 中添加一项。框架假定所有设备都只支持两个电源级别：0 和标准能耗。

电源假定所有其他组件对组件 0 具有隐式相关性。当组件 0 更改为级别 0 时，使用 DDI_PM_SUSPEND 命令调用驱动程序的 detach(9E) 来保存硬件状态。当组件 0 的级别 0 发生更改时，使用命令 DDI_PM_RESUME 调用 attach(9E) 例程来恢复硬件状态。

以下接口和命令已过时，现在仍支持它们是为了适于采用二进制的场合：

- ddi_dev_is_needed(9F)
- pm_create_components(9F)
- pm_destroy_components(9F)
- pm_get_normal_power(9F)
- pm_set_normal_power(9F)
- DDI_PM_SUSPEND
- DDI_PM_RESUME

从 Solaris 8 发行版开始，如果启用了 autopm，则导出 pm-components 属性的设备将自动使用电源管理。

现在，框架可以通过 pm-components 属性了解每个设备支持的电源级别。

框架对设备不同组件之间的相关性不会作出任何假设。更改电源级别时，设备驱动程序负责根据需要保存并恢复硬件状态。

通过这些更改，电源管理框架可以处理新兴的设备技术。现在，电源管理可以节省更多的电。框架可以自动检测哪些设备能够省电。框架可以使用设备的中间能耗状态。现在，系统可以实现节能目标，而无需关闭整个系统的电源，也无需使用任何功能。

表 12–1 电源管理接口

已删除的接口	等效接口
pm_create_components(9F)	pm-components(9P)
pm_set_normal_power(9F)	pm-components(9P)
pm_destroy_components(9F)	无
pm_get_normal_power(9F)	无
ddi_dev_is_needed(9F)	pm_raise_power(9F)
无	pm_lower_power(9F)
无	pm_power_has_changed(9F)
DDI_PM_SUSPEND	无
DDI_PM_RESUME	无

◆ ◆ ◆ 第 13 章

13 强化 Oracle Solaris 驱动程序

借助故障管理体系结构 (Fault Management Architecture, FMA) I/O 故障服务，驱动程序开发者可将故障管理功能集成到 I/O 设备驱动程序中。Oracle Solaris I/O 故障服务框架定义了一组接口，使得所有驱动程序可以协调工作，并执行基本的错误处理任务和活动。总体上，Oracle Solaris FMA 除了可进行响应和恢复外，还可进行错误处理和故障诊断。FMA 是 Oracle 的预测性自我修复策略的一个组成部分。

当驱动程序除了将 I/O 故障服务框架用于错误处理和诊断外，还使用本文档中介绍的防御性编程做法时，认为该驱动程序已经过强化。驱动程序强化测试工具测试是否已正确实现 I/O 故障服务和防御性编程要求。

本文档包含以下部分：

- 第 209 页中的 “Oracle 故障管理体系结构 I/O 故障服务”为希望将故障管理功能集成到 I/O 设备驱动程序中的驱动程序开发者提供了参考。

Oracle 故障管理体系结构 I/O 故障服务

本节介绍如何为 I/O 设备驱动程序集成故障管理错误报告、错误处理和诊断。本节深入探讨了 I/O 故障服务框架以及如何在设备驱动程序内利用 I/O 故障服务 API。

本节讨论以下主题：

- 第 210 页中的 “什么是预测性自我修复？ ”提供 Oracle 故障管理体系结构的背景信息和概述。
- 第 210 页中的 “Oracle Solaris Fault Manager”介绍了更多背景信息，重点介绍 Oracle Solaris Fault Manager `fmd`(1M) 的较高层面的概述。
- 第 213 页中的 “错误处理”是面向驱动程序开发者的主要章节。本节重点介绍用于获得高可用性的最佳编码方法，以及如何在驱动程序代码中使用 I/O 故障服务来与 FMA 交互。

什么是预测性自我修复？

传统上，系统会将硬件和软件错误信息以系统日志消息的形式直接导出给管理员以及管理软件。错误检测、诊断、报告和处理通常会嵌入到每个驱动程序的代码中。

Solaris OS 预测性自我修复系统这样的系统是最早的也是最重要的自我诊断系统。自我诊断意味着系统提供根据观察到的症状自动诊断问题的技术，然后将诊断结果用于触发自动响应和恢复。硬件**故障**或软件缺陷可与一组可能观察到的、称为**错误**的症状相关联。系统检测到错误后所生成的数据称为错误报告或 *ereport*。

在具有自我修复功能的系统中，ereport 由系统捕获，并编码为由可扩展事件协议描述的一组名称-值对，从而形成 *ereport* **事件**。收集 ereport 事件和其他数据是为了便于进行自我修复，还会将其分发给名为诊断引擎的软件组件，诊断引擎设计用来诊断与系统检测到的错误症状对应的底层问题。**诊断引擎**在后台运行，并以无提示方式使用错误遥测，直到它可以生成诊断或预测故障为止。

在处理足够的遥测从而得出结论之后，诊断引擎会生成名为**故障事件**的另一个事件。然后，会向对特定故障事件感兴趣的所有代理广播该故障事件。**代理**是可对特定故障事件启动恢复和做出响应的软件组件。被称为 Oracle Solaris Fault Manager 的软件组件 `fmd(1M)` 可在 ereport 生成器、诊断引擎和代理软件之间管理事件的多路复用。

Oracle Solaris Fault Manager

Oracle Solaris Fault Manager `fmd(1M)` 负责将传入错误遥测事件分发给相应的诊断引擎。诊断引擎负责识别产生错误症状的基础硬件故障或软件缺陷。`fmd`(1M) 守护进程是故障管理器的 Oracle Solaris OS 实现。它在引导时启动，并且会装入系统中可用的所有诊断引擎和代理。Oracle Solaris Fault Manager 还为系统管理员和服务人员提供了用于观察故障管理活动的界面。

诊断、可疑列表和故障事件

进行诊断后，会以 *list.suspect* 事件的形式输出诊断。list.suspect 事件是由一个或多个可能的故障或缺陷事件构成的事件。有时候，诊断无法将错误原因的范围缩小至单个故障或缺陷。例如，底层问题可能是连接控制器与主系统总线的线路中断。问题也可能在于总线上的某个组件或总线本身。在这种特定情况下，list.suspect 事件将包含多个故障事件：一个是连接到总线的每个控制器的事件，另一个是总线本身的事件。

除了介绍诊断的故障外，故障事件还包含四个可应用诊断的有效负荷成员。

- **资源**是被诊断为有故障的组件。`fmdump(1M)` 命令将此有效负荷成员显示为 "Problem in"。
- **自动系统恢复单元** (Automated System Recovery Unit, ASRU) 是必须禁用以防止出现更多错误症状的硬件或软件组件。`fmdump`(1M) 命令将此有效负荷成员显示为 "Affects"。
- **现场可更换单元** (Field Replaceable Unit, FRU) 是必须更换或维修以便修复底层问题的组件。

- **标签**有效负荷是一个字符串，用于按照 FRU 在机箱或主板上的印刷形式给出其位置，例如，在 DIMM 插槽或 PCI 卡插槽旁边。`fmdump` 命令将此有效负荷成员显示为 "Location"。

例如，在给定时间内针对特定内存位置收到特定数量的 ECC 可纠正错误后，CPU 和内存诊断引擎会对有故障的 DIMM 发出诊断（list.suspect 事件）。

```
# fmdump -v -u 38bd6f1b-a4de-4c21-db4e-ccd26fa8573c
TIME                 UUID                                 SUNW-MSG-ID
Oct 31 13:40:18.1864 38bd6f1b-a4de-4c21-db4e-ccd26fa8573c AMD-8000-8L
100%  fault.cpu.amd.icachetag

Problem in: hc:///motherboard=0/chip=0/cpu=0
Affects: cpu:///cpuid=0
FRU: hc:///motherboard=0/chip=0
Location: SLOT 2
```

在此示例中，`fmd(1M)` 识别到资源中的一个问题，具体而言是 CPU (`hc:///motherboard=0/chip=0/cpu=0`)。为了抑制出现更多错误症状并防止发生无法纠正的错误，对一个 ASRU (`cpu:///cpuid=0`) 进行了标识，以便弃用 (retirement)。需要更换的组件是 FRU (`hc:///motherboard=0/chip=0`)。

响应代理

代理是可为了响应诊断或修复而执行操作的软件组件。例如，CPU 和内存弃用代理设计为对包含 fault.cpu.* 事件的 list.suspect 执行操作。`cpumem-retire` 代理将尝试在服务中使 CPU 脱机或弃用物理内存页。如果该代理成功，则会在故障管理器的 ASRU 缓存中为成功弃用的页或 CPU 添加一项。以下示例所示的 `fmadm(1M)` 实用程序显示了被诊断为有故障的内存等级的一项。系统无法使其脱机、将其弃用或禁用的 ASRU 也会在 ASRU 缓存中存在一项，但会将其视为被降级。降级意味着与 ASRU 关联的资源有故障，但无法从服务中将该 ASRU 删除。目前，Oracle Solaris 代理软件不能对 I/O ASRU（设备实例）执行操作。缓存中所有有故障的 I/O 资源项都处于降级状态。

```
# fmadm faulty
   STATE RESOURCE / UUID
-------- ------------------------------------------------------------------------
degraded mem:///motherboard=0/chip=1/memory-controller=0/dimm=3/rank=0
         ccae89df-2217-4f5c-add4-d920f78b4faf
-------- ------------------------------------------------------------------------
```

弃用代理的主要用途是隔离（从服务中安全删除）被诊断为有故障的硬件或软件部分。

代理还可以执行其他重要操作，例如以下操作：

- 通过 SNMP 陷阱发送警报。这样便可将诊断转换为插入现有软件机制中的 SNMP 警报。
- 发布系统日志消息。特定于消息的诊断（例如，系统日志消息代理）可以提取诊断的结果，并将其转换为管理员可用来执行特定操作的系统日志消息。
- 其他代理操作，如更新 FRUID。响应代理可以特定于平台。

消息 ID 和字典文件

系统日志消息代理会提取诊断的输出（list.suspect 事件），并将特定消息写入控制台或 /var/adm/messages。控制台消息通常可能会难以理解。FMA 提供一个每当将 list.suspect 事件发送至系统日志消息时都会生成的已定义故障消息结构，从而对此问题进行了修正。

系统日志代理会生成一个消息标识符 (message identifier, MSG ID)。事件注册表生成字典文件（.dict 文件），这些文件可将 list.suspect 事件映射到将用来标识和查看关联知识文章的结构化消息标识符。消息文件（.po 文件）则将消息 ID 映射到诊断引擎可以生成的每个可能的可疑故障列表中的本地化消息。下面是一个测试系统中发出的故障消息示例。

```
SUNW-MSG-ID: AMD-8000-7U, TYPE: Fault, VER: 1, SEVERITY: Major
EVENT-TIME: Fri Jul 28 04:26:51 PDT 2006
PLATFORM: Sun Fire V40z, CSN: XG051535088, HOSTNAME: parity
SOURCE: eft, REV: 1.16
EVENT-ID: add96f65-5473-69e6-dbe1-8b3d00d5c47b
DESC: The number of errors associated with this CPU has exceeded
acceptable levels. Refer to  http://support.oracle.com/msg/SMF-8000-05
 for more information.
AUTO-RESPONSE: An attempt will be made to remove this CPU from service.
IMPACT: Performance of this system may be affected.
REC-ACTION: Schedule a repair procedure to replace the affected CPU.
Use fmdump -v -u <EVENT_ID> to identify the module.
```

系统拓扑

为了确定可能发生故障的位置，诊断引擎需要表示出给定软件或硬件系统的拓扑。fmd(1M) 守护进程为诊断引擎提供了一个可在诊断期间使用的拓扑快照句柄。拓扑信息用来表示在每个故障事件中找到的资源、ASRU 和 FRU。拓扑也可以用来存储平台标签、FRUID 和序列号标识。

故障事件中的资源有效负荷成员始终由平台机箱周围组件的物理路径位置来表示。例如，从主系统总线桥接至 PCI 本地总线的 PCI 控制器功能由其 hc 模式路径名来表示：

```
hc:///motherboard=0/hostbridge=1/pcibus=0/pcidev=13/pcifn=0
```

故障事件中的 ASRU 有效负荷成员通常由绑定到硬件控制器、设备或功能的 Oracle Solaris 设备树实例名称来表示。对于可能由专门为 I/O 设备设计的弃用代理的将来实现执行的操作，FMA 使用 dev 模式以其本地格式表示 ASRU：

```
dev:////pci@1e,600000/ide@d
```

故障事件中的 FRU 有效负荷表示形式随距离被诊断为有故障的 I/O 资源最近的可更换组件而异。例如，中断的嵌入式 PCI 控制器的故障事件可能会将系统的主板命名为需要更换的 FRU：

```
hc:///motherboard=0
```

标签有效负荷是一个字符串，用于按照 FRU 在机箱或主板上的印刷形式给出其位置，例如，在 DIMM 插槽或 PCI 卡插槽旁边。

```
Label: SLOT 2
```

错误处理

本节介绍如何使用 I/O 故障服务 API 来处理驱动程序内的错误。本节讨论驱动程序应如何指示和初始化其故障管理功能、生成错误报告以及注册驱动程序的错误处理程序例程。

被指示提供 FMA 错误报告遥测的驱动程序将检测错误，并确定这些错误对驱动程序所提供服务的影响。检测到错误后，驱动程序应确定其服务受影响的时间以及程度。

I/O 驱动程序必须立即响应检测到的错误。相应的响应包括：

- 尝试恢复
- 重试 I/O 事务
- 尝试故障转移技术
- 向调用应用程序/堆栈报告错误
- 如果以任何其他方式均无法约束错误，则进入紧急状态

驱动程序检测到的错误以 *ereport* 的形式传递给故障管理守护进程。ereport 是由 FMA 事件协议定义的结构化事件。该事件协议是一组常用数据字段的规范，除了可疑故障列表外，这些字段还必须用于描述所有可能的错误和故障事件。Ereport 被收集为错误遥测流，并分发给诊断引擎。

声明故障管理功能

强化的设备驱动程序必须向 I/O 故障管理框架声明其故障管理功能。使用 `ddi_fm_init(9F)` 函数声明驱动程序的故障管理功能。

```
void ddi_fm_init(dev_info_t *dip, int *fmcap, ddi_iblock_cookie_t *ibcp)
```

可从驱动程序 `attach(9E)` 或 `detach(9E)` 入口点的内核上下文中调用 `ddi_fm_init()` 函数。通常会从 `attach()` 入口点调用 `ddi_fm_init()` 函数。`ddi_fm_init()` 函数根据 *fmcap* 来分配和初始化资源。*fmcap* 参数必须设置为以下故障管理功能的按位或：

- `DDI_FM_EREPORT_CAPABLE` – 在检测到错误状态时，驱动程序负责并且能够生成 FMA 协议错误事件 (ereport)。
- `DDI_FM_ACCCHK_CAPABLE` – 完成对 I/O 事务的一次或多次访问后，驱动程序负责并且能够检查错误。
- `DDI_FM_DMACHK_CAPABLE` – 完成一个或多个 DMA I/O 事务后，驱动程序负责并且能够检查错误。
- `DDI_FM_ERRCB_CAPABLE` – 驱动程序具有错误回调功能。

强化的叶驱动程序通常设置上述所有功能。但是，如果父结点不能支持任何一项请求的功能，则关联的位会被清除并按此情况返回给驱动程序。从 ddi_fm_init (9F) 返回之前，I/O 故障服务框架会创建一组故障管理功能属性：fm-ereport-capable、fm-accchk-capable、fm-dmachk-capable 和 fm-errcb-capable。可使用 prtconf(1M) 命令来观察当前支持的故障管理功能级别。

为了使驱动程序支持故障管理功能的管理选择，请导出故障管理功能级别属性并将其设置为上面 driver.conf(4) 文件中描述的值。在使用所需功能列表调用 ddi_fm_init() 之前，必须设置并读取 fm-capable 属性。

来自 bge 驱动程序的以下示例显示了 bge_fm_init() 函数，该函数调用 ddi_fm_init(9F) 函数。可在 bge_attach() 函数中调用 bge_fm_init() 函数。

```
static void
bge_fm_init(bge_t *bgep)
{
        ddi_iblock_cookie_t iblk;

        /* Only register with IO Fault Services if we have some capability */
        if (bgep->fm_capabilities) {
                bge_reg_accattr.devacc_attr_access = DDI_FLAGERR_ACC;
                dma_attr.dma_attr_flags = DDI_DMA_FLAGERR;
                /*
                 * Register capabilities with IO Fault Services
                 */
                ddi_fm_init(bgep->devinfo, &bgep->fm_capabilities, &iblk);
                /*
                 * Initialize pci ereport capabilities if ereport capable
                 */
                if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities) ||
                    DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
                        pci_ereport_setup(bgep->devinfo);
                /*
                 * Register error callback if error callback capable
                 */
                if (DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
                        ddi_fm_handler_register(bgep->devinfo,
                        bge_fm_error_cb, (void*) bgep);
        } else {
                /*
                 * These fields have to be cleared of FMA if there are no
                 * FMA capabilities at runtime.
                 */
                bge_reg_accattr.devacc_attr_access = DDI_DEFAULT_ACC;
                dma_attr.dma_attr_flags = 0;
        }
}
```

清除故障管理资源

ddi_fm_fini(9F) 函数清除为支持 *dip* 的故障管理而分配的资源。

```
void ddi_fm_fini(dev_info_t *dip)
```

可从驱动程序 attach(9E) 或 detach(9E) 入口点的内核上下文中调用 ddi_fm_fini() 函数。

来自 bge 驱动程序的以下示例显示了 bge_fm_fini() 函数，该函数调用 ddi_fm_fini(9F) 函数。可在 bge_unattach() 函数中调用 bge_fm_fini() 函数，而在 bge_attach() 和 bge_detach() 函数中调用 bge_unattach 函数。

```
static void
bge_fm_fini(bge_t *bgep)
{
        /* Only unregister FMA capabilities if we registered some */
        if (bgep->fm_capabilities) {
                /*
                 * Release any resources allocated by pci_ereport_setup()
                 */
                if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities) ||
                    DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
                        pci_ereport_teardown(bgep->devinfo);
                /*
                 * Un-register error callback if error callback capable
                 */
                if (DDI_FM_ERRCB_CAP(bgep->fm_capabilities))
                        ddi_fm_handler_unregister(bgep->devinfo);
                /*
                 * Unregister from IO Fault Services
                 */
                ddi_fm_fini(bgep->devinfo);
        }
}
```

获取故障管理功能位掩码

ddi_fm_capable(9F) 函数返回当前为 *dip* 设置的功能位掩码。

```
void ddi_fm_capable(dev_info_t *dip)
```

报告错误

本节提供有关以下主题的信息：

- 第 215 页中的 “对错误事件排队”讨论如何对错误事件排队。
- 第 216 页中的 “检测和报告与 PCI 相关的错误”介绍如何报告与 PCI 相关的错误。
- 第 217 页中的 “报告标准 I/O 控制器错误”介绍如何报告标准 I/O 控制器错误。
- 第 219 页中的 “服务影响函数”讨论如何报告错误是否对设备提供的服务产生了影响。

对错误事件排队

ddi_fm_ereport_post(9F) 函数对 ereport 事件排队，以便传送给故障管理器守护进程 fmd(1M)。

```
void ddi_fm_ereport_post(dev_info_t *dip,
                         const char *error_class,
                         uint64_t ena,
                         int sflag, ...)
```

sflag 参数指示调用方是否愿意等待系统内存和事件通道资源变为可用。

ENA 指示此错误报告的**错误编号关联** (Error Numeric Association, ENA)。ENA 可能已初始化，并且是从其他错误检测软件模块（如总线结点驱动程序）中获得的。如果 ENA 设置为 0，它将被 `ddi_fm_ereport_post()` 初始化。

名称-值对 (*nvpair*) 变量参数列表包含非数组 *data_type_t* 类型的一个或多个名称、类型、值指针 `nvpair` 元组，或者包含 `data_type_t` 数组类型的一个或多个名称、类型、元素数、值指针元组。*nvpair* 元组补足诊断所需要的 ereport 事件有效负荷。参数列表的结尾由 `NULL` 指定。

第 217 页中的 “报告标准 I/O 控制器错误”中介绍的用于 I/O 控制器的 ereport 类名和有效负荷可适用于 *error_class*。可以定义其他 ereport 类名和有效负荷，但必须在 Oracle **事件注册表**中进行注册，并伴有特定于驱动程序的诊断引擎软件或 Eversholt 故障树 (Eversholt fault tree, eft) 规则。

```
void
bge_fm_ereport(bge_t *bgep, char *detail)
{
        uint64_t ena;
        char buf[FM_MAX_CLASS];
        (void) snprintf(buf, FM_MAX_CLASS, "%s.%s", DDI_FM_DEVICE, detail);
        ena = fm_ena_generate(0, FM_ENA_FMT1);
        if (DDI_FM_EREPORT_CAP(bgep->fm_capabilities)) {
                ddi_fm_ereport_post(bgep->devinfo, buf, ena, DDI_NOSLEEP,
                    FM_VERSION, DATA_TYPE_UINT8, FM_EREPORT_VERS0, NULL);
        }
}
```

检测和报告与 PCI 相关的错误

使用 `pci_ereport_post(9F)` 时，会自动检测和报告与 PCI（包括 PCI、PCI-X 和 PCI-E）相关的错误。

```
void pci_ereport_post(dev_info_t *dip, ddi_fm_error_t *derr, uint16_t *xx_status)
```

驱动程序不需要为 PCI 本地总线配置状态寄存器中发生的错误生成特定于驱动程序的 ereport。`pci_ereport_post()` 函数可以报告数据奇偶校验错误、主机中止、目标中止、发出信号的系统错误等。

如果驱动程序要使用 `pci_ereport_post()`，则之前必须已在驱动程序的 `attach(9E)` 例程中调用了 `pci_ereport_setup(9F)`，并且之后必须在驱动程序的 `detach(9E)` 例程中调用`pci_ereport_teardown(9F)`。

下面的 `bge` 代码样例显示了从驱动程序的错误处理程序中调用 `pci_ereport_post()` 函数的 `bge` 驱动程序。

```
/*
 * The I/O fault service error handling callback function
 */
/*ARGSUSED*/
static int
bge_fm_error_cb(dev_info_t *dip, ddi_fm_error_t *err, const void *impl_data)
{
     /*
      * as the driver can always deal with an error
      * in any dma or access handle, we can just return
      * the fme_status value.
      */
     pci_ereport_post(dip, err, NULL);
     return (err->fme_status);
}
```

报告标准 I/O 控制器错误

针对 I/O 控制器的常见错误定义了一组标准的设备 ereport。只要检测到本节中所述的错误症状之一，便应生成这些 ereport。

本节中所述的 ereport 将分发给 eft 诊断引擎以进行诊断，eft 诊断引擎使用一组常用的标准规则来诊断这些 ereport。设备驱动程序检测的其他任何错误都必须在 Sun 事件注册表中定义为 ereport 事件，并必须伴有特定于设备的诊断软件或 eft 规则。

DDI_FM_DEVICE_INVAL_STATE
: 驱动程序已检测到设备处于无效状态。

  当驱动程序检测到所传送或接收的数据看起来无效时，该驱动程序应发布错误。例如，在 bge 代码中，当 `bge_chip_reset()` 和 `bge_receive_ring()` 例程检测到无效数据时，这些例程将生成 `ereport.io.device.inval_state` 错误。

```
/*
 * The SEND INDEX registers should be reset to zero by the
 * global chip reset; if they're not, there'll be trouble
 * later on.
 */
sx0 = bge_reg_get32(bgep, NIC_DIAG_SEND_INDEX_REG(0));
if (sx0 != 0) {
    BGE_REPORT((bgep, "SEND INDEX - device didn't RESET"));
    bge_fm_ereport(bgep, DDI_FM_DEVICE_INVAL_STATE);
    return (DDI_FAILURE);
}
/* ... */
/*
 * Sync (all) the receive ring descriptors
 * before accepting the packets they describe
 */
DMA_SYNC(rrp->desc, DDI_DMA_SYNC_FORKERNEL);
if (*rrp->prod_index_p >= rrp->desc.nslots) {
    bgep->bge_chip_state = BGE_CHIP_ERROR;
    bge_fm_ereport(bgep, DDI_FM_DEVICE_INVAL_STATE);
    return (NULL);
}
```

DDI_FM_DEVICE_INTERN_CORR
设备已报告自我纠正的内部错误。例如，设备的内部缓冲区中的硬件已检测到可纠正的 ECC 错误。bge 驱动程序中未使用此错误标志。

DDI_FM_DEVICE_INTERN_UNCORR
设备已报告无法纠正的内部错误。例如，设备的内部缓冲区中的硬件已检测到不可纠正的 ECC 错误。

bge 驱动程序中未使用此错误标志。

DDI_FM_DEVICE_STALL
驱动程序检测到数据传输已意外停顿。

bge_factotum_stall_check() 例程提供了停顿检测的示例。

```
dogval = bge_atomic_shl32(&bgep->watchdog, 1);
if (dogval < bge_watchdog_count)
    return (B_FALSE);

BGE_REPORT((bgep, "Tx stall detected,
watchdog code 0x%x", dogval));
bge_fm_ereport(bgep, DDI_FM_DEVICE_STALL);
return (B_TRUE);
```

DDI_FM_DEVICE_NO_RESPONSE
设备未对驱动程序命令进行响应。

```
bge_chip_poll_engine(bge_t *bgep, bge_regno_t regno,
        uint32_t mask, uint32_t val)
{
        uint32_t regval;
        uint32_t n;

        for (n = 200; n; --n) {
                regval = bge_reg_get32(bgep, regno);
                if ((regval & mask) == val)
                        return (B_TRUE);
                drv_usecwait(100);
        }
        bge_fm_ereport(bgep, DDI_FM_DEVICE_NO_RESPONSE);
        return (B_FALSE);
}
```

DDI_FM_DEVICE_BADINT_LIMIT
设备引发了过多的连续性无效中断。

bge() 驱动程序内的 bge_intr 例程提供了有问题的中断检测的示例。bge_fm_ereport() 函数是 ddi_fm_ereport_post(9F) 函数的包装。请参见第 215 页中的“对错误事件排队”中的 bge_fm_ereport() 示例。

```
if (bgep->missed_dmas >= bge_dma_miss_limit) {
    /*
     * If this happens multiple times in a row,
     * it means DMA is just not working.  Maybe
     * the chip has failed, or maybe there's a
```

```
         * problem on the PCI bus or in the host-PCI
         * bridge (Tomatillo).
         *
         * At all events, we want to stop further
         * interrupts and let the recovery code take
         * over to see whether anything can be done
         * about it ...
         */
        bge_fm_ereport(bgep,
            DDI_FM_DEVICE_BADINT_LIMIT);
        goto chip_stop;
    }
```

服务影响函数

具有故障管理功能的驱动程序必须指示错误是否影响了设备所提供的服务。检测到错误并在必要时关闭服务之后，驱动程序应调用 ddi_fm_service_impact(9F) 例程来反映设备实例的当前服务状态。诊断和恢复软件可以使用该服务状态来帮助确定问题或对问题做出反应。

当驱动程序本身检测到错误时以及框架检测到错误并将访问或 DMA 句柄标记为有故障时，均应调用 ddi_fm_service_impact() 例程。

```
void ddi_fm_service_impact(dev_info_t *dip, int svc_impact)
```

ddi_fm_service_impact() 接受以下服务影响值 (*svc_impact*)：

DDI_SERVICE_LOST	由于设备故障或软件缺陷，设备提供的服务不可用。
DDI_SERVICE_DEGRADED	驱动程序无法提供正常服务，但驱动程序可以提供部分服务或降级的服务。例如，驱动程序可能必须重复尝试执行操作才能取得成功，或者它至少要以配置的速度运行。
DDI_SERVICE_UNAFFECTED	驱动程序已检测到错误，但设备实例提供的服务不会受到影响。
DDI_SERVICE_RESTORED	设备提供的所有服务都已恢复。

调用 ddi_fm_service_impact() 时会根据服务影响例程的服务影响参数代表驱动程序生成以下 ereport：

- ereport.io.service.lost
- ereport.io.service.degraded
- ereport.io.service.unaffected
- ereport.io.service.restored

在以下 bge 代码中，驱动程序确定由于出现错误，它无法成功地重新开始传送或接收数据包。设备的服务状态转换为 DDI_SERVICE_LOST。

```
/*
 * All OK, reinitialize hardware and kick off GLD scheduling
 */
mutex_enter(bgep->genlock);
if (bge_restart(bgep, B_TRUE) != DDI_SUCCESS) {
    (void) bge_check_acc_handle(bgep, bgep->cfg_handle);
    (void) bge_check_acc_handle(bgep, bgep->io_handle);
    ddi_fm_service_impact(bgep->devinfo, DDI_SERVICE_LOST);
    mutex_exit(bgep->genlock);
    return (DDI_FAILURE);
}
```

注 – 不应从已注册的回调例程中调用 `ddi_fm_service_impact()` 函数。

◆ ◆ ◆ 第 14 章

分层驱动程序接口 (Layered Driver Interface, LDI)

LDI 是一组 DDI/DKI，内核模块可以使用它来访问系统中的其他设备。另外使用 LDI 还可以确定内核模块当前使用的设备。

本章包含以下主题：

LDI 概述

LDI 包括以下两类接口：

- **内核接口**。用户应用程序使用系统调用来打开、读取和写入由内核中的设备驱动程序管理的设备。内核模块可以使用 LDI 内核接口来打开、读取和写入由内核中的另一设备驱动程序管理的设备。例如，用户应用程序可使用 `read(2)` 读取某个设备，而内核模块可使用 `ldi_read(9F)` 来读取同一设备。 请参见第 222 页中的 “内核接口”。
- **用户接口**。LDI 用户接口可为用户进程提供有关内核中其他设备当前使用哪些设备的信息。请参见第 236 页中的 “用户接口”。

讨论 LDI 时经常用到以下术语：

- **Target Device（目标设备）**。目标设备是内核中的设备，由设备驱动程序管理并由设备使用方访问。
- **Device Consumer（设备使用方）**。设备使用方是打开并访问目标设备的用户进程或内核模块。设备使用方通常对目标设备执行 `open`、`read`、`write` 或 `ioctl` 之类的操作。
- **Kernel Device Consumer（内核设备使用方）**。内核设备使用方是一种特定类型的设备使用方，它是访问目标设备的内核模块。通常情况下，内核设备使用方不是用于管理要访问的目标设备的设备驱动程序。相反，内核设备使用方通过管理目标设备的设备驱动程序间接访问目标设备。

- **Layered Driver（分层驱动程序）**。分层驱动程序是一种特定类型的内核设备使用方。分层驱动程序是一种不直接管理任何硬件的内核驱动程序。相反，分层驱动程序通过管理目标设备的设备驱动程序间接访问这些目标设备中的一个或多个设备。例如，卷管理器和 STREAMS 多路复用器就是比较典型的分层驱动程序。

内核接口

通过某些 LDI 内核接口，LDI 可以跟踪和报告内核设备使用信息。请参见第 222 页中的“分层标识符－内核设备使用方”。

通过其他 LDI 内核接口，内核模块可以对目标设备执行 `open`、`read` 和 `write` 之类的访问操作。另外，通过这些 LDI 内核接口，内核设备使用方可以查询有关目标设备的属性和事件信息。请参见第 223 页中的“分层驱动程序句柄－目标设备”。

第 226 页中的“LDI 内核接口示例”介绍了使用其中多个 LDI 接口的驱动程序示例。

分层标识符－内核设备使用方

通过分层标识符，LDI 可以跟踪和报告内核设备使用信息。分层标识符 (`ldi_ident_t`) 用于标识内核设备使用方。内核设备使用方必须先获取分层标识符，然后才能使用 LDI 打开目标设备。

分层驱动程序是唯一受支持的内核设备使用方类型。因此，分层驱动程序必须获取与设备编号、设备信息节点或分层驱动程序流关联的分层标识符。分层标识符与分层驱动程序关联。分层标识符与目标设备没有关联。

可以通过 `libdevinfo`(3LIB) 接口、`fuser`(1M) 命令或 `prtconf`(1M) 命令，检索通过 LDI 收集的内核设备使用信息。例如，使用 `prtconf`(1M) 命令可以显示分层驱动程序正在访问哪些目标设备，或者哪些分层驱动程序正在访问特定目标设备。要了解有关如何检索设备使用情况的更多信息，请参见第 236 页中的“用户接口”。

下面介绍了 LDI 分层标识符接口：

`ldi_ident_t`	分层标识符。属于不透明类型。
`ldi_ident_from_dev`(9F)	分配和检索与 `dev_t` 设备编号关联的分层标识符。
`ldi_ident_from_dip`(9F)	分配和检索与 `dev_info_t` 设备信息节点关联的分层标识符。
`ldi_ident_from_stream`(9F)	分配和检索与流关联的分层标识符。
`ldi_ident_release`(9F)	释放使用 `ldi_ident_from_dev`(9F)、`ldi_ident_from_dip`(9F) 或 `ldi_ident_from_stream`(9F) 分配的分层标识符。

分层驱动程序句柄－目标设备

内核设备使用方必须使用分层驱动程序句柄 (ldi_handle_t) 来通过 LDI 接口访问目标设备。ldi_handle_t 类型仅对 LDI 接口有效。当 LDI 成功打开某个设备时，将分配并返回此句柄。然后，内核设备使用方可使用此句柄通过 LDI 接口访问目标设备。LDI 在关闭设备时会取消分配该句柄。有关示例，请参见第 226 页中的 “LDI 内核接口示例”。

本节讨论内核设备使用方如何访问目标设备并检索不同类型的信息。要了解内核设备使用者如何打开和关闭目标设备，请参见第 223 页中的 “打开和关闭目标设备”。要了解内核设备使用方如何对目标设备执行 read、write、strategy 和 ioctl 之类的操作，请参见第 223 页中的 “访问目标设备”。第 224 页中的 “检索目标设备信息”介绍了用于检索目标设备信息（如设备打开类型和设备次要名称）的接口。第 225 页中的 “检索目标设备属性值”介绍了用于检索目标设备属性的值和地址的接口。要了解内核设备使用方如何接收来自目标设备的事件通知，请参见第 225 页中的 “接收异步设备事件通知”。

打开和关闭目标设备

本节介绍用于打开和关闭目标设备的 LDI 内核接口。打开接口采用指向分层驱动程序句柄的指针。打开接口会尝试打开由设备编号、设备 ID 或路径名指定的目标设备。如果打开操作成功，则打开接口将分配并返回可用于访问目标设备的分层驱动程序句柄。关闭接口用于关闭与指定分层驱动程序句柄关联的目标设备，然后释放该分层驱动程序句柄。

ldi_handle_t	用于访问目标设备的分层驱动程序句柄。一种成功打开设备时返回的不透明数据结构。
ldi_open_by_dev(9F)	打开由 dev_t 设备编号参数指定的设备。
ldi_open_by_devid(9F)	打开由 ddi_devid_t 设备 ID 参数指定的设备。另外，还必须指定要打开的次要节点名称。
ldi_open_by_name(9F)	根据路径名打开设备。路径名是内核地址空间中以 NULL 结尾的字符串。路径名必须是以正斜杠字符 (/) 开头的绝对路径。
ldi_close(9F)	关闭使用 ldi_open_by_dev(9F)、ldi_open_by_devid(9F) 或 ldi_open_by_name(9F) 打开的设备。在 ldi_close(9F) 返回之后，已关闭设备的分层驱动程序句柄不再有效。

访问目标设备

本节介绍用于访问目标设备的 LDI 内核接口。通过这些接口，内核设备使用方可以对由分层驱动程序句柄指定的目标设备执行操作。内核设备使用方可以对目标设备执行 read、write、strategy 和 ioctl 之类的操作。

ldi_handle_t	用于访问目标设备的分层驱动程序句柄。属于不透明数据结构。

ldi_read(9F)	将读取请求传递到目标设备的设备入口点。块设备、字符设备和 STREAMS 设备支持此操作。
ldi_aread(9F)	将异步读取请求传递到目标设备的设备入口点。块设备和字符设备支持此操作。
ldi_write(9F)	将写入请求传递到目标设备的设备入口点。块设备、字符设备和 STREAMS 设备支持此操作。
ldi_awrite(9F)	将异步写入请求传递到目标设备的设备入口点。块设备和字符设备支持此操作。
ldi_strategy(9F)	将策略请求传递到目标设备的设备入口点。块设备和字符设备支持此操作。
ldi_dump(9F)	将转储请求传递到目标设备的设备入口点。块设备和字符设备支持此操作。
ldi_poll(9F)	将轮询请求传递到目标设备的设备入口点。块设备、字符设备和 STREAMS 设备支持此操作。
ldi_ioctl(9F)	将 ioctl 请求传递到目标设备的设备入口点。块设备、字符设备和 STREAMS 设备支持此操作。LDI 支持 STREAMS 链接和 STREAMS ioctl 命令。请参见 ldi_ioctl(9F) 手册页的 "STREAM IOCTLS" 一节。另请参见 streamio(7I) 手册页中的 ioctl 命令。
ldi_devmap(9F)	将 devmap 请求传递到目标设备的设备入口点。块设备和字符设备支持此操作。
ldi_getmsg(9F)	从流中获取消息块。
ldi_putmsg(9F)	将消息块放在流中。

检索目标设备信息

本节介绍内核设备使用方可用于检索有关指定目标设备的设备信息的 LDI 接口。目标设备由分层驱动程序句柄指定。内核设备使用方可以接收设备编号、设备打开类型、设备 ID、设备次要名称和设备大小之类的信息。

ldi_get_dev(9F)	获取由分层驱动程序句柄指定的目标设备的 dev_t 设备编号。
ldi_get_otyp(9F)	获取用于打开由分层驱动程序句柄指定的目标设备的打开标志。此标志指示目标设备是字符设备还是块设备。
ldi_get_devid(9F)	获取由分层驱动程序句柄指定的目标设备的 ddi_devid_t 设备 ID。使用完设备 ID 后，应使用 ddi_devid_free(9F) 释放 ddi_devid_t。

ldi_get_minor_name(9F)	检索包含为目标设备打开的次要节点的名称的缓冲区。使用完次要节点名称后，应使用 kmem_free(9F) 释放该缓冲区。
ldi_get_size(9F)	检索由分层驱动程序句柄指定的目标设备的分区大小。

检索目标设备属性值

本节介绍内核设备使用方可用于检索有关指定目标设备的属性信息的 LDI 接口。目标设备由分层驱动程序句柄指定。内核设备使用方可以接收属性的值和地址，以及确定某属性是否存在。

ldi_prop_exists(9F)	如果由分层驱动程序句柄指定的目标设备的属性存在，则返回 1。如果指定目标设备的属性不存在，则返回 0。
ldi_prop_get_int(9F)	搜索与由分层驱动程序句柄指定的目标设备关联的 int 整数属性。如果找到整数属性，则返回属性值。
ldi_prop_get_int64(9F)	搜索与由分层驱动程序句柄指定的目标设备关联的 int64_t 整数属性。如果找到整数属性，则返回属性值。
ldi_prop_lookup_int_array(9F)	检索由分层驱动程序句柄指定的目标设备的 int 整数数组属性值的地址。
ldi_prop_lookup_int64_array(9F)	检索由分层驱动程序句柄指定的目标设备的 int64_t 整数数组属性值的地址。
ldi_prop_lookup_string(9F)	检索由分层驱动程序句柄指定的目标设备的以 null 结尾的字符串属性值的地址。
ldi_prop_lookup_string_array(9F)	检索字符串数组的地址。字符串数组是一个指针数组，指向由分层驱动程序句柄指定的目标设备的以 null 结尾的字符串属性值。
ldi_prop_lookup_byte_array(9F)	检索字节数组的地址。字节数组是由分层驱动程序句柄指定的目标设备的属性值。

接收异步设备事件通知

通过 LDI，内核设备使用方可以注册事件通知以及接收来自目标设备的事件通知。内核设备使用方可以注册发生事件时将会调用的事件处理程序。内核设备使用方必须先打开设备并接收分层驱动程序句柄，然后才能通过 LDI 事件通知接口注册事件通知。

通过 LDI 事件通知接口，内核设备使用方可以指定事件名称以及检索关联的内核事件 cookie。然后，内核设备使用方可以将分层驱动程序句柄 (ldi_handle_t)、cookie (ddi_eventcookie_t) 及事件处理程序传递到 ldi_add_event_handler(9F) 以注册事件通

知。成功完成注册后，内核设备使用方会收到一个唯一的 LDI 事件处理程序标识符 (ldi_callback_id_t)。LDI 事件处理程序标识符属于不透明类型，只能用于 LDI 事件通知接口。

LDI 提供了一个框架，以用于注册其他设备生成的事件。LDI 本身并不定义任何事件类型，也不提供用于生成事件的接口。

下面介绍了 LDI 异步事件通知接口：

ldi_callback_id_t	事件处理程序标识符。属于不透明类型。
ldi_get_eventcookie(9F)	检索由分层驱动程序句柄指定的目标设备的事件服务 cookie。
ldi_add_event_handler(9F)	添加由 ldi_callback_id_t 注册标识符指定的回调处理程序。发生由 ddi_eventcookie_t cookie 指定的事件时，将会调用该回调处理程序。
ldi_remove_event_handler(9F)	删除由 ldi_callback_id_t 注册标识符指定的回调处理程序。

LDI 内核接口示例

本节介绍了一个使用本章前面几节中讨论的一些 LDI 调用的内核设备使用方示例。本节讨论此示例模块的下列几个方面：

- 第 226 页中的 “设备配置文件”
- 第 227 页中的 “驱动程序源文件”
- 第 235 页中的 “测试分层驱动程序”

此内核设备使用方示例名为 lyr。lyr 模块是一个分层驱动程序，它使用 LDI 调用向目标设备发送数据。在其 open(9E) 入口点中，lyr 驱动程序将打开由 lyr.conf 配置文件中的 lyr_targ 属性指定的设备。在其 write(9E) 入口点中，lyr 驱动程序将其所有传入数据写入由 lyr_targ 属性指定的设备。

设备配置文件

在下面所示的配置文件中，lyr 驱动程序向其中写入数据的目标设备为控制台。

示例 14–1 配置文件

```
#
# Use is subject to license terms.
#
#pragma ident    "%Z%%M%    %I%    %E% SMI"

name="lyr" parent="pseudo" instance=1;
lyr_targ="/dev/console";
```

驱动程序源文件

在下面所示的驱动程序源文件中，lyr_state_t 结构保存 lyr 驱动程序的软状态。该软状态包括 lyr_targ 设备的分层驱动程序句柄 (lh) 和 lyr 设备的分层标识符 (li)。有关软状态的更多信息，请参见第 483 页中的 “检索驱动程序软状态信息”。

在 lyr_open() 入口点中，ddi_prop_lookup_string(9F) 将从 lyr_targ 属性中检索要打开的 lyr 设备的目标设备的名称。ldi_ident_from_dev(9F) 函数用于获取 lyr 设备的 LDI 分层标识符。ldi_open_by_name(9F) 函数用于打开 lyr_targ 设备并获取 lyr_targ 设备的分层驱动程序句柄。

请注意，如果 lyr_open() 中发生任何故障，ldi_close(9F)、ldi_ident_release(9F) 和 ddi_prop_free(9F) 调用将会撤消所执行的所有操作。ldi_close(9F) 函数用于关闭 lyr_targ 设备。ldi_ident_release(9F) 函数用于释放 lyr 分层标识符。ddi_prop_free(9F) 函数用于释放检索 lyr_targ 设备名称时分配的资源。如果未发生故障，则会在 lyr_close() 入口点中调用 ldi_close (9F) 和 ldi_ident_release(9F) 函数。

在驱动程序模块的最后一行中，调用了 ldi_write(9F) 函数。ldi_write(9F) 函数先获取在 lyr_write() 入口点中写入 lyr 设备的数据，然后将该数据写入 lyr_targ 设备。ldi_write(9F) 函数使用 lyr_targ 设备的分层驱动程序句柄将数据写入 lyr_targ 设备。

示例 14–2 驱动程序源文件

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/cmn_err.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/stat.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/sunldi.h>

typedef struct lyr_state {
    ldi_handle_t    lh;
    ldi_ident_t     li;
    dev_info_t      *dip;
    minor_t         minor;
    int             flags;
    kmutex_t        lock;
} lyr_state_t;

#define LYR_OPENED      0x1     /* lh is valid */
#define LYR_IDENTED     0x2     /* li is valid */

static int lyr_info(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int lyr_attach(dev_info_t *, ddi_attach_cmd_t);
```

示例 14–2 驱动程序源文件 （续）

```
static int lyr_detach(dev_info_t *, ddi_detach_cmd_t);
static int lyr_open(dev_t *, int, int, cred_t *);
static int lyr_close(dev_t, int, int, cred_t *);
static int lyr_write(dev_t, struct uio *, cred_t *);

static void *lyr_statep;

static struct cb_ops lyr_cb_ops = {
    lyr_open,            /* open */
    lyr_close,           /* close */
    nodev,               /* strategy */
    nodev,               /* print */
    nodev,               /* dump */
    nodev,               /* read */
    lyr_write,           /* write */
    nodev,               /* ioctl */
    nodev,               /* devmap */
    nodev,               /* mmap */
    nodev,               /* segmap */
    nochpoll,            /* poll */
    ddi_prop_op,         /* prop_op */
    NULL,                /* streamtab  */
    D_NEW | D_MP,        /* cb_flag */
    CB_REV,              /* cb_rev */
    nodev,               /* aread */
    nodev                /* awrite */
};

static struct dev_ops lyr_dev_ops = {
    DEVO_REV,            /* devo_rev, */
    0,                   /* refcnt  */
    lyr_info,            /* getinfo */
    nulldev,             /* identify */
    nulldev,             /* probe */
    lyr_attach,          /* attach */
    lyr_detach,          /* detach */
    nodev,               /* reset */
    &lyr_cb_ops,         /* cb_ops */
    NULL,                /* bus_ops */
    NULL,                /* power */
    ddi_quiesce_not_needed,     /* quiesce */
};

static struct modldrv modldrv = {
    &mod_driverops,
    "LDI example driver",
    &lyr_dev_ops
};

static struct modlinkage modlinkage = {
    MODREV_1,
    &modldrv,
    NULL
};
```

示例 14–2 驱动程序源文件 （续）

```
int
_init(void)
{
    int rv;

    if ((rv = ddi_soft_state_init(&lyr_statep, sizeof (lyr_state_t),
        0)) != 0) {
        cmn_err(CE_WARN, "lyr _init: soft state init failed\n");
        return (rv);
    }
    if ((rv = mod_install(&modlinkage)) != 0) {
        cmn_err(CE_WARN, "lyr _init: mod_install failed\n");
        goto FAIL;
    }
    return (rv);
    /*NOTEREACHED*/
FAIL:
    ddi_soft_state_fini(&lyr_statep);
    return (rv);
}

int
_info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}

int
_fini(void)
{
    int rv;

    if ((rv = mod_remove(&modlinkage)) != 0) {
        return(rv);
    }
    ddi_soft_state_fini(&lyr_statep);
    return (rv);
}
/*
 * 1:1 mapping between minor number and instance
 */
static int
lyr_info(dev_info_t *dip, ddi_info_cmd_t infocmd, void *arg, void **result)
{
    int inst;
    minor_t minor;
    lyr_state_t *statep;
    char *myname = "lyr_info";

    minor = getminor((dev_t)arg);
    inst = minor;
    switch (infocmd) {
    case DDI_INFO_DEVT2DEVINFO:
        statep = ddi_get_soft_state(lyr_statep, inst);
        if (statep == NULL) {
```

示例 14–2 驱动程序源文件 （续）

```
            cmn_err(CE_WARN, "%s: get soft state "
                "failed on inst %d\n", myname, inst);
            return (DDI_FAILURE);
        }
        *result = (void *)statep->dip;
        break;
    case DDI_INFO_DEVT2INSTANCE:
        *result = (void *)inst;
        break;
    default:
        break;
    }

    return (DDI_SUCCESS);
}

static int
lyr_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int inst;
    lyr_state_t *statep;
    char *myname = "lyr_attach";

    switch (cmd) {
    case DDI_ATTACH:
        inst = ddi_get_instance(dip);

        if (ddi_soft_state_zalloc(lyr_statep, inst) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "%s: ddi_soft_state_zallac failed "
                "on inst %d\n", myname, inst);
            goto FAIL;
        }
        statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
        if (statep == NULL) {
            cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
                "inst %d\n", myname, inst);
            goto FAIL;
        }
        statep->dip = dip;
        statep->minor = inst;
        if (ddi_create_minor_node(dip, "node", S_IFCHR, statep->minor,
            DDI_PSEUDO, 0) != DDI_SUCCESS) {
            cmn_err(CE_WARN, "%s: ddi_create_minor_node failed on "
                "inst %d\n", myname, inst);
            goto FAIL;
        }
        mutex_init(&statep->lock, NULL, MUTEX_DRIVER, NULL);
        return (DDI_SUCCESS);
    case DDI_RESUME:
    case DDI_PM_RESUME:
    default:
        break;
    }
    return (DDI_FAILURE);
    /*NOTREACHED*/
```

示例 14–2 驱动程序源文件 （续）

```
FAIL:
    ddi_soft_state_free(lyr_statep, inst);
    ddi_remove_minor_node(dip, NULL);
    return (DDI_FAILURE);
}

static int
lyr_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    int inst;
    lyr_state_t *statep;
    char *myname = "lyr_detach";

    inst = ddi_get_instance(dip);
    statep = ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: get soft state failed on "
            "inst %d\n", myname, inst);
        return (DDI_FAILURE);
    }
    if (statep->dip != dip) {
        cmn_err(CE_WARN, "%s: soft state does not match devinfo "
            "on inst %d\n", myname, inst);
        return (DDI_FAILURE);
    }
    switch (cmd) {
    case DDI_DETACH:
        mutex_destroy(&statep->lock);
        ddi_soft_state_free(lyr_statep, inst);
        ddi_remove_minor_node(dip, NULL);
        return (DDI_SUCCESS);
    case DDI_SUSPEND:
    case DDI_PM_SUSPEND:
    default:
        break;
    }
    return (DDI_FAILURE);
}
/*
 * on this driver's open, we open the target specified by a property and store
 * the layered handle and ident in our soft state.  a good target would be
 * "/dev/console" or more interestingly, a pseudo terminal as specified by the
 * tty command
 */
/*ARGSUSED*/
static int
lyr_open(dev_t *devtp, int oflag, int otyp, cred_t *credp)
{
    int rv, inst = getminor(*devtp);
    lyr_state_t *statep;
    char *myname = "lyr_open";
    dev_info_t *dip;
    char *lyr_targ = NULL;

    statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
```

示例 14–2 驱动程序源文件 （续）

```
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
            "inst %d\n", myname, inst);
        return (EIO);
    }
    dip = statep->dip;
    /*
     * our target device to open should be specified by the "lyr_targ"
     * string property, which should be set in this driver's .conf file
     */
    if (ddi_prop_lookup_string(DDI_DEV_T_ANY, dip, DDI_PROP_NOTPROM,
        "lyr_targ", &lyr_targ) != DDI_PROP_SUCCESS) {
        cmn_err(CE_WARN, "%s: ddi_prop_lookup_string failed on "
            "inst %d\n", myname, inst);
        return (EIO);
    }
    /*
     * since we only have one pair of lh's and li's available, we don't
     * allow multiple on the same instance
     */
    mutex_enter(&statep->lock);
    if (statep->flags & (LYR_OPENED | LYR_IDENTED)) {
        cmn_err(CE_WARN, "%s: multiple layered opens or idents "
            "from inst %d not allowed\n", myname, inst);
        mutex_exit(&statep->lock);
        ddi_prop_free(lyr_targ);
        return (EIO);
    }
    rv = ldi_ident_from_dev(*devtp, &statep->li);
    if (rv != 0) {
        cmn_err(CE_WARN, "%s: ldi_ident_from_dev failed on inst %d\n",
            myname, inst);
        goto FAIL;
    }
    statep->flags |= LYR_IDENTED;
    rv = ldi_open_by_name(lyr_targ, FREAD | FWRITE, credp, &statep->lh,
        statep->li);
    if (rv != 0) {
        cmn_err(CE_WARN, "%s: ldi_open_by_name failed on inst %d\n",
            myname, inst);
        goto FAIL;
    }
    statep->flags |= LYR_OPENED;
    cmn_err(CE_CONT, "\n%s: opened target '%s' successfully on inst %d\n",
        myname, lyr_targ, inst);
    rv = 0;

FAIL:
    /* cleanup on error */
    if (rv != 0) {
        if (statep->flags & LYR_OPENED)
            (void)ldi_close(statep->lh, FREAD | FWRITE, credp);
        if (statep->flags & LYR_IDENTED)
            ldi_ident_release(statep->li);
        statep->flags &= ~(LYR_OPENED | LYR_IDENTED);
```

示例 14–2 驱动程序源文件 （续）

```
    }
    mutex_exit(&statep->lock);
    if (lyr_targ != NULL)
        ddi_prop_free(lyr_targ);
    return (rv);
}
/*
 * on this driver's close, we close the target indicated by the lh member
 * in our soft state and release the ident, li as well.  in fact, we MUST do
 * both of these at all times even if close yields an error because the
 * device framework effectively closes the device, releasing all data
 * associated with it and simply returning whatever value the target's
 * close(9E) returned.  therefore, we must as well.
 */
/*ARGSUSED*/
static int
lyr_close(dev_t devt, int oflag, int otyp, cred_t *credp)
{
    int rv, inst = getminor(devt);
    lyr_state_t *statep;
    char *myname = "lyr_close";
    statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
        cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
            "inst %d\n", myname, inst);
        return (EIO);
    }
    mutex_enter(&statep->lock);
    rv = ldi_close(statep->lh, FREAD | FWRITE, credp);
    if (rv != 0) {
        cmn_err(CE_WARN, "%s: ldi_close failed on inst %d, but will ",
            "continue to release ident\n", myname, inst);
    }
    ldi_ident_release(statep->li);
    if (rv == 0) {
        cmn_err(CE_CONT, "\n%s: closed target successfully on "
            "inst %d\n", myname, inst);
    }
    statep->flags &= ~(LYR_OPENED | LYR_IDENTED);
    mutex_exit(&statep->lock);
    return (rv);
}
/*
 * echo the data we receive to the target
 */
/*ARGSUSED*/
static int
lyr_write(dev_t devt, struct uio *uiop, cred_t *credp)
{
    int rv, inst = getminor(devt);
    lyr_state_t *statep;
    char *myname = "lyr_write";

    statep = (lyr_state_t *)ddi_get_soft_state(lyr_statep, inst);
    if (statep == NULL) {
```

示例 14–2 驱动程序源文件 （续）

```
        cmn_err(CE_WARN, "%s: ddi_get_soft_state failed on "
            "inst %d\n", myname, inst);
        return (EIO);
    }
    return (ldi_write(statep->lh, uiop, credp));
}
```

▼ 如何生成和装入分层驱动程序

1 编译驱动程序。

使用 -D_KERNEL 选项指示这是一个内核模块。

- 如果要针对 SPARC 体系结构进行编译，请使用 -xarch=v9 选项：

```
% cc -c -D_KERNEL -xarch=v9 lyr.c
```

- 如果要针对 32 位 x86 体系结构进行编译，请使用以下命令：

```
% cc -c -D_KERNEL lyr.c
```

2 链接驱动程序。

```
% ld -r -o lyr lyr.o
```

3 安装配置文件。

以 root 用户身份，将配置文件复制到计算机的内核驱动程序区域：

```
# cp lyr.conf /usr/kernel/drv
```

4 安装驱动程序二进制文件。

- 以 root 用户身份，将驱动程序二进制文件复制到 SPARC 体系结构的 sparcv9 驱动程序区域：

```
# cp lyr /usr/kernel/drv/sparcv9
```

- 以 root用户身份，将驱动程序二进制文件复制到 32 位 x86 体系结构的 drv 驱动程序区域：

```
# cp lyr /usr/kernel/drv
```

5 装入驱动程序。

以 root 用户身份，使用 add_drv(1M) 命令装入驱动程序。

```
# add_drv lyr
```

列出伪设备，确认目前是否存在 lyr 设备：

```
# ls /devices/pseudo | grep lyr
lyr@1
lyr@1:node
```

测试分层驱动程序

要测试 lyr 驱动程序，请向 lyr 设备写入一条消息，并验证该消息是否显示在 lyr_targ 设备上。

示例 14–3 向分层设备写入一条短消息

在本示例中，lyr_targ 设备是安装了 lyr 设备的系统的控制台。

如果要查看的显示屏幕也是安装了 lyr 设备的系统的控制台设备的显示屏幕，请注意，向控制台写入将会破坏显示屏幕上的信息。控制台消息将显示在窗口系统范围以外。测试 lyr 驱动程序之后，需要重画或刷新显示器。

如果要查看的显示屏幕不是安装了 lyr 设备的系统的控制台设备的显示屏幕，请登录或以其他方式查看目标控制台设备的显示屏幕上的信息。

以下命令将一条很短的消息写入 lyr 设备：

```
# echo "\n\n\t===> Hello World!! <===\n" > /devices/pseudo/lyr@1:node
```

目标控制台上将会显示以下消息：

```
console login:

    ===> Hello World!! <===

lyr:
lyr_open: opened target ’/dev/console’ successfully on inst 1
lyr:
lyr_close: closed target successfully on inst 1
```

执行 lyr_open() 和 lyr_close() 时所显示的消息来自在 lyr_open() 和 lyr_close() 入口点中执行的 cmn_err(9F) 调用。

示例 14–4 向分层设备写入一条较长的消息

以下命令将一条较长的消息写入 lyr 设备：

```
# cat lyr.conf > /devices/pseudo/lyr@1:node
```

目标控制台上将会显示以下消息：

```
lyr:
lyr_open: opened target ’/dev/console’ successfully on inst 1
#

# Use is subject to license terms.
#
#pragma ident   "%Z%%M% %I%     %E% SMI"

name="lyr" parent="pseudo" instance=1;
lyr_targ="/dev/console";
```

示例 14–4 向分层设备写入一条较长的消息 （续）

```
lyr:
lyr_close: closed target successfully on inst 1
```

示例 14–5 更改目标设备

要更改目标设备，请编辑 `/usr/kernel/drv/lyr.conf`，并将 `lyr_targ` 属性的值更改为指向其他目标设备的路径。例如，该目标设备可以是在本地终端执行 `tty` 命令后的输出结果。例如，此类设备路径可以是 `/dev/pts/4`。

在将驱动程序更新为使用新目标设备之前，应确保 `lyr` 设备未被使用。

```
# modinfo -c | grep lyr
174          3 lyr                                 UNLOADED/UNINSTALLED
```

使用 `update_drv(1M)` 命令重新装入 `lyr.conf` 配置文件：

```
# update_drv lyr
```

再次向 `lyr` 设备写入一条消息，并验证该消息是否显示在新的 `lyr_targ` 设备上。

用户接口

LDI 中包括用户级库和命令接口，用于报告设备分层和使用信息。第 236 页中的 “设备信息库接口”介绍了用于报告设备分层信息的 `libdevinfo(3LIB)` 接口。第 238 页中的 “列显系统配置命令接口”介绍了用于报告内核设备使用信息的 `prtconf(1M)` 接口。第 240 页中的 “设备用户命令接口”介绍了用于报告设备使用方信息的 `fuser(1M)` 接口。

设备信息库接口

LDI 中包括用于报告设备分层信息快照的 `libdevinfo(3LIB)` 接口。如果系统中的一个设备是同一系统中另一个设备的使用方，则会发生设备分层。仅当使用方和目标都绑定到快照中包含的设备节点时，才会报告设备分层信息。

`libdevinfo(3LIB)` 接口以有向图的形式报告设备分层信息。*lnode* 是一个抽象术语，在图中表示顶点，并被绑定到设备节点。可以使用 `libdevinfo(3LIB)` 接口来访问 lnode 的属性，如节点的名称和设备编号。

图中的边表示链接。链接既有表示设备使用方的源 lnode，也有表示目标设备的目标 lnode。

下面介绍了 `libdevinfo(3LIB)` 设备分层信息接口：

`DINFOLYR` 通过它来捕获设备分层信息的快照标志。

di_link_t	两个端点之间的有向链接。每个端点都是一个 di_lnode_t。属于不透明结构。
di_lnode_t	链接的端点。属于不透明结构。di_lnode_t 绑定到 di_node_t。
di_node_t	表示设备节点。属于不透明结构。di_node_t 不一定绑定到 di_lnode_t。
di_walk_link(3DEVINFO)	遍历快照中的所有链接。
di_walk_lnode(3DEVINFO)	遍历快照中的所有 lnode。
di_link_next_by_node(3DEVINFO)	获取以指定的 di_node_t 节点作为源节点或目标节点的下一个链接的句柄。
di_link_next_by_lnode(3DEVINFO)	获取以指定的 di_lnode_t lnode 作为源 lnode 或目标 lnode 的下一个链接的句柄。
di_link_to_lnode(3DEVINFO)	获取与 di_link_t 链接的指定端点对应的 lnode。
di_link_spectype(3DEVINFO)	获取链接的规范类型。规范类型指示如何访问目标设备。目标设备由目标 lnode 表示。
di_lnode_next(3DEVINFO)	获取与指定的 di_node_t 设备节点关联的指定 di_lnode_t lnode 的下一个实例的句柄。
di_lnode_name(3DEVINFO)	获取与指定 lnode 关联的名称。
di_lnode_devinfo(3DEVINFO)	获取与指定 lnode 关联的设备节点的句柄。
di_lnode_devt(3DEVINFO)	获取与指定 lnode 关联的设备节点的设备编号。

LDI 返回的设备分层信息可能十分复杂。因此，LDI 提供了一些接口来协助遍历设备树和设备使用情况图。通过这些接口，设备树快照的使用方可以将定制数据指针与快照中的不同结构关联。例如，应用程序遍历 lnode 时，它可以更新与每个 lnode 关联的定制指针，以标记已经识别的 lnode。

下面介绍了 libdevinfo(3LIB) 节点和链接标记接口：

di_lnode_private_set(3DEVINFO)	将指定的数据与指定的 lnode 关联。通过此关联，可以遍历快照中的 lnode。
di_lnode_private_get(3DEVINFO)	检索指向通过调用 di_lnode_private_set(3DEVINFO) 而与 lnode 关联的数据的指针。
di_link_private_set(3DEVINFO)	将指定的数据与指定的链接关联。通过此关联，可以遍历快照中的链接。

di_link_private_get(3DEVINFO)	检索指向通过调用 di_link_private_set(3DEVINFO) 而与链接关联的数据的指针。

列显系统配置命令接口

prtconf(1M) 命令已得到增强，可以显示内核设备使用信息。缺省的 prtconf(1M) 输出没有变化。如果在 prtconf(1M) 命令中指定详细选项 (-v)，则会显示设备使用信息。如果在 prtconf(1M) 命令行上指定了特定设备的路径，则会显示有关该设备的使用信息。

prtconf -v	显示设备次要节点和设备使用信息。显示内核使用方和每个内核使用方当前打开的次要节点。
prtconf *path*	显示由 *path* 指定的设备的设备使用信息。
prtconf -a *path*	显示由 *path* 指定的设备的设备使用信息，以及作为 *path* 的祖先的所有设备节点。
prtconf -c *path*	显示由 *path* 指定的设备的设备使用信息，以及作为 *path* 的子节点的所有设备节点。

示例 14–6 设备使用信息

如果需要有关特定设备的使用信息，*path* 参数的值可以是任何有效的设备路径。

```
% prtconf /dev/cfg/c0
SUNW,isptwo, instance #0
```

示例 14–7 祖先节点使用信息

要显示有关特定设备和作为其祖先的所有设备节点的使用信息，请随 prtconf(1M) 命令指定 -a 标志。祖先包括直到设备树的根的所有节点。如果随 prtconf(1M) 命令指定了 -a 标志，则还必须指定设备的 *path* 名称。

```
% prtconf -a /dev/cfg/c0
SUNW,Sun-Fire
    ssm, instance #0
        pci, instance #0
            pci, instance #0
                SUNW,isptwo, instance #0
```

示例 14–8 子节点使用信息

要显示有关特定设备和作为其子节点的所有设备节点的使用信息，请随 prtconf(1M) 命令指定 -c 标志。如果随 prtconf(1M) 命令指定了 -c 标志，则还必须指定设备的 *path* 名称。

示例 14–8 子节点使用信息 （续）

```
% prtconf -c /dev/cfg/c0
SUNW,isptwo, instance #0
    sd (driver not attached)
    st (driver not attached)
    sd, instance #1
    sd, instance #0
    sd, instance #6
    st, instance #1 (driver not attached)
    st, instance #0 (driver not attached)
    st, instance #2 (driver not attached)
    st, instance #3 (driver not attached)
    st, instance #4 (driver not attached)
    st, instance #5 (driver not attached)
    st, instance #6 (driver not attached)
    ses, instance #0 (driver not attached)
    ...
```

示例 14–9 分层和设备次要节点信息－键盘

要显示有关特定设备的设备分层和设备次要节点信息，请随 prtconf(1M) 命令指定 -v 标志。

```
% prtconf -v /dev/kbd
conskbd, instance #0
    System properties:
        ...
    Device Layered Over:
        mod=kb8042 dev=(101,0)
            dev_path=/isa/i8042@1,60/keyboard@0
    Device Minor Nodes:
        dev=(103,0)
            dev_path=/pseudo/conskbd@0:kbd
                spectype=chr type=minor
                dev_link=/dev/kbd
        dev=(103,1)
            dev_path=/pseudo/conskbd@0:conskbd
                spectype=chr type=internal
            Device Minor Layered Under:
                mod=wc accesstype=chr
                    dev_path=/pseudo/wc@0
```

本示例中，/dev/kbd 设备所在层位于硬件键盘设备 (/isa/i8042@1,60/keyboard@0) 之上。另外，本示例中，/dev/kbd 设备具有两个设备次要节点。第一个次要节点具有可用于访问该节点的 /dev 链接。第二个次要节点是一个无法通过文件系统访问的内部节点。wc 驱动程序（即工作站控制台）已经打开了第二个次要节点。请将本示例的输出与示例 14–12 的输出进行比较。

示例 14–10 分层和设备次要节点信息－网络设备

本示例说明哪些设备正在使用当前检测到的网络设备。

示例 14–10 分层和设备次要节点信息－网络设备 （续）

```
% prtconf -v /dev/iprb0
pci1028,145, instance #0
    Hardware properties:
        ...
    Interrupt Specifications:
        ...
    Device Minor Nodes:
        dev=(27,1)
            dev_path=/pci@0,0/pci8086,244e@1e/pci1028,145@c:iprb0
                spectype=chr type=minor
                alias=/dev/iprb0
        dev=(27,4098)
            dev_path=<clone>
            Device Minor Layered Under:
                mod=udp6 accesstype=chr
                    dev_path=/pseudo/udp6@0
        dev=(27,4097)
            dev_path=<clone>
            Device Minor Layered Under:
                mod=udp accesstype=chr
                    dev_path=/pseudo/udp@0
        dev=(27,4096)
            dev_path=<clone>
            Device Minor Layered Under:
                mod=udp accesstype=chr
                    dev_path=/pseudo/udp@0
```

本示例中，在采用 udp 和 udp6 的情况下链接了 iprb0 设备。请注意，此处未显示指向采用 udp 和 udp6 的次要节点的任何路径。本示例中未显示任何路径是因为次要节点是通过对 iprb 驱动程序执行 clone 打开操作创建的，因此不存在可以访问这些节点的文件系统路径。请将本示例的输出与示例 14–11 的输出进行比较。

设备用户命令接口

fuser(1M) 命令已得到增强，可以显示设备使用信息。仅当 *path* 表示设备次要节点时，fuser(1M) 命令才会显示设备使用信息。仅当指定了表示设备次要节点的 *path* 时，随 fuser(1M) 命令使用 -d 标志才有效。

`fuser` *path* 显示有关应用程序设备使用方和内核设备使用方的信息（如果 *path* 表示设备次要节点）。

`fuser -d` *path* 显示与 *path* 表示的设备次要节点关联的基础设备的所有用户。

报告内核设备使用方时采用以下四种格式之一。内核设备使用方始终用方括号 ([]) 括起来。

```
[kernel_module_name]
[kernel_module_name,dev_path=path]
[kernel_module_name,dev=(major,minor)]
```

[*kernel_module_name*,dev=(*major*,*minor*),dev_path=*path*]

如果 fuser(1M) 命令显示的是文件或设备用户，则输出由 stdout 中的进程 ID 后跟 stderr 中的字符组成。stderr 中的字符描述如何使用文件或设备。stderr 中会显示所有内核使用方信息。而 stdout 中不会显示任何内核使用方信息。

如果未使用 -d 标志，则 fuser(1M) 命令仅报告由 *path* 指定的设备次要节点的使用方。如果使用 -d 标志，则 fuser(1M) 命令会报告由 *path* 指定的次要节点的基础设备节点的使用方。以下示例说明了这两种情况下报告输出的差别。

示例 14–11 基础设备节点的使用方

大多数网络设备在打开时都会克隆其次要节点。如果请求克隆次要节点的设备使用信息，则该使用信息可能会表明没有任何进程在使用该设备。而如果请求基础设备节点的设备使用信息，则该使用信息可能会表明某个进程正在使用该设备。在本示例中，如果仅将设备 *path* 传递到 fuser(1M) 命令，则不会报告任何设备使用方。如果使用 -d 标志，则输出将表明正在采用 udp 和 udp6 来访问该设备。

```
% fuser /dev/iprb0
/dev/iprb0:
% fuser -d /dev/iprb0
/dev/iprb0:  [udp,dev_path=/pseudo/udp@0] [udp6,dev_path=/pseudo/udp6@0]
```

请将本示例的输出与示例 14–10 的输出进行比较。

示例 14–12 键盘设备的使用方

在本示例中，某个内核使用方正在访问 /dev/kbd。正在访问 /dev/kbd 设备的内核使用方是工作站控制台驱动程序。

```
% fuser -d /dev/kbd
/dev/kbd:  [genunix] [wc,dev_path=/pseudo/wc@0]
```

请将本示例的输出与示例 14–9 的输出进行比较。

第 2 部分

设计特定种类的设备驱动程序

本书的第二部分提供特定于驱动程序类型的设计信息：

- 第 15 章，字符设备驱动程序介绍了面向字符的设备的驱动程序。
- 第 16 章，块设备驱动程序介绍了面向块的设备的驱动程序。
- 第 17 章，SCSI 目标驱动程序概述了 Sun 公用 SCSI 体系结构 (Sun Common SCSI Architecture, SCSA) 和对 SCSI 目标驱动程序的要求。
- 第 18 章，SCSI 主机总线适配器驱动程序介绍了如何将 SCSA 应用到 SCSI 主机总线适配器 (Host Bus Adapter, HBA) 驱动程序。
- 第 19 章，网络设备驱动程序介绍了通用 LAN 驱动程序 (Generic LAN driver, GLD)。GLDv3 框架是 MAC 插件和 MAC 驱动程序服务例程与结构的基于函数调用的接口。
- 第 20 章，USB 驱动程序介绍了如何使用 USBA 2.0 框架编写客户机 USB 设备驱动程序。
- 第 21 章，SR-IOV 驱动程序介绍了可用于写入 SR-IOV 驱动程序的 SR-IOV 设备驱动程序和接口。

◆ ◆ ◆ 第 15 章

15 字符设备驱动程序

字符设备没有可物理寻址的存储介质（如磁带机或串行端口），在这些存储介质中 I/O 通常是以字节流的形式执行的。本章介绍字符设备驱动程序的结构，其中重点介绍字符驱动程序的入口点。此外，本章还介绍在同步和异步 I/O 传输上下文中 `physio(9F)` 和 `aphysio(9F)` 的用法。

本章介绍有关以下主题的信息：

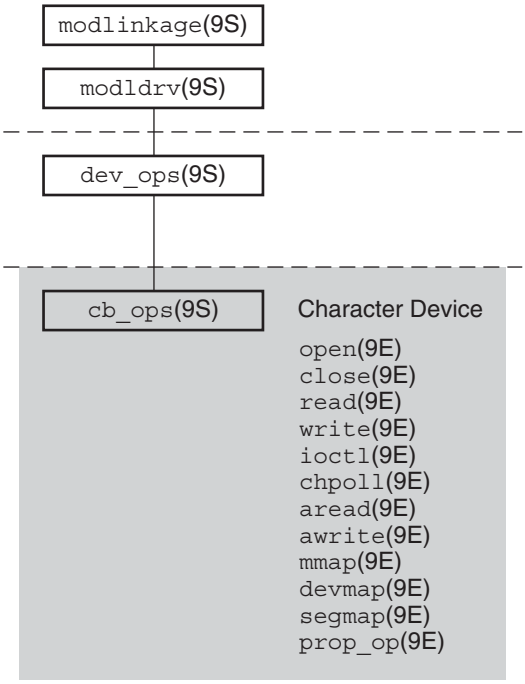
字符驱动程序结构概述

图 15–1 显示了用来定义字符设备驱动程序结构的数据结构和例程。设备驱动程序通常包括以下元素：

- 可装入设备的驱动程序段
- 设备配置部分
- 字符驱动程序入口点

下图中涂有阴影的设备访问部分列出了字符驱动程序入口点。

图 15–1 字符驱动程序结构示意图



对于每个设备驱动程序，都有一个 dev_ops(9S) 结构与之关联，该结构进而又指向 cb_ops(9S) 结构。这些结构包含指向驱动程序入口点的指针：

- open(9E)
- close(9E)
- read(9E)
- write(9E)
- ioctl(9E)
- chpoll(9E)
- aread(9E)
- awrite(9E)
- mmap(9E)
- devmap(9E)
- segmap(9E)
- prop_op(9E)

注 – 可以根据需要将其中一些入口点替换为 nodev(9F) 或 nulldev(9F)。

字符设备自动配置

attach(9E) 例程应执行所有设备需要的常见初始化任务，例如：

- 分配每个实例的状态结构
- 注册设备中断
- 映射设备寄存器
- 初始化互斥变量和条件变量
- 创建可进行电源管理的组件
- 创建次要节点

有关这些任务的代码示例，请参见第 101 页中的 “attach() 入口点”。

字符设备驱动程序将创建类型为 S_IFCHR 的次要节点。类型为 S_IFCHR 的次要节点会使代表节点的字符特殊文件最终出现在 /devices 分层结构中。

以下示例显示了字符驱动程序的典型 attach(9E) 例程。与设备有关的属性通常在 attach() 例程中声明。该示例使用了预定义的 Size 属性。在获取块设备的分区大小时，Size 与 Nblocks 属性是等效的。举例来说，如果要在磁盘设备上执行字符 I/O，就可以使用 Size 来获取分区大小。因为 Size 是 64 位属性，所以必须使用 64 位属性接口。在本例中，使用 ddi_prop_update_int64(9F)。有关属性的更多信息，请参见第 75 页中的 “设备属性”。

示例 15–1 字符驱动程序 attach() 例程

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
  int instance = ddi_get_instance(dip);
  switch (cmd) {
  case DDI_ATTACH:
      /*
       * Allocate a state structure and initialize it.
       * Map the device's registers.
       * Add the device driver's interrupt handler(s).
       * Initialize any mutexes and condition variables.
       * Create power manageable components.
       *
       * Create the device's minor node. Note that the node_type
       * argument is set to DDI_NT_TAPE.
       */
       if (ddi_create_minor_node(dip, minor_name, S_IFCHR,
           instance, DDI_NT_TAPE, 0) == DDI_FAILURE)
{
           /* Free resources allocated so far. */
           /* Remove any previously allocated minor nodes. */
           ddi_remove_minor_node(dip, NULL);
           return (DDI_FAILURE);
       }
      /*
       * Create driver properties like "Size." Use "Size"
       * instead of "size" to ensure the property works
       * for large bytecounts.
```

示例 15–1 字符驱动程序 attach() 例程 （续）

```
        */
        xsp->Size = size_of_device_in_bytes;
        maj_number = ddi_driver_major(dip);
        if (ddi_prop_update_int64(makedevice(maj_number, instance),
            dip, "Size", xsp->Size) != DDI_PROP_SUCCESS) {
            cmn_err(CE_CONT, "%s: cannot create Size property\n",
                ddi_get_name(dip));
                /* Free resources allocated so far. */
            return (DDI_FAILURE);
        }
       /* ... */
       return (DDI_SUCCESS);
case DDI_RESUME:
       /* See the "Power Management" chapter in this book. */
default:
       return (DDI_FAILURE);
  }
}
```

设备访问（字符驱动程序）

可通过 open(9E) 和 close(9E) 入口点来控制一个或多个应用程序对设备的访问。对代表字符设备的特殊文件进行 open(2) 系统调用始终会导致为驱动程序调用 open(9E) 例程。对于特定的从设备，open(9E) 可以多次被调用，而 close(9E) 例程只有在删除了对设备的最终引用时才会调用。如果通过文件描述符访问设备，则 close(2) 或 exit(2) 系统调用都可能导致对 close(9E) 的最终调用。如果通过内存映射访问设备，则 munmap(2) 系统调用可能导致对 close(9E) 的最终调用。

open() 入口点（字符驱动程序）

open() 的主要功能是检验是否允许打开请求。open(9E) 的语法如下所示：

```
int xxopen(dev_t *devp, int flag, int otyp, cred_t *credp);
```

其中：

devp 指向设备编号的指针。会向 open() 例程传递指针，以便驱动程序可以更改次要设备号。使用此指针，驱动程序能够动态创建设备的次要实例。伪终端驱动程序就是这样，只要打开该驱动程序，就会创建新的伪终端。通常，动态选择次要设备号的驱动程序使用 ddi_create_minor_node(9F) 在 attach(9E) 中仅创建一个次要设备节点，然后使用 makedevice(9F) 和 getmajor(9F) 更改 *devp 的次要设备号部分：

```
*devp = makedevice(getmajor(*devp), new_minor);
```

您不必调用 ddi_create_minor_node(9F) 来创建新的次要节点。驱动程序不得更改 *devp 的主设备号。驱动程序必须在内部跟踪可用的次要设备号。

flag 使用位指示打开设备是供读取 (FREAD)、写入 (FWRITE) 还是供可同时读写的标志。发出 open(2) 系统调用的用户线程也可以请求对设备进行独占访问 (FEXCL)，或指定不得以任何原因阻止打开操作 (FNDELAY)，但驱动程序必须强制执行两者。只写设备（例如打印机）的驱动程序可能会将 open(9E) 视为对读操作无效。

otyp 表示如何调用 open() 的整数。驱动程序必须检查 *otyp* 的值是否适用于相应设备。对于字符驱动程序，*otyp* 应为 OTYP_CHR（请参见 open(9E) 手册页）。

credp 指向包含有关调用方信息（例如用户 ID 和组 ID）的凭证结构的指针。驱动程序不会直接检查此结构，但会使用 drv_priv(9F) 来检查 root 用户权限的一般情况。在本示例中，只允许 root 或具有 PRIV_SYS_DEVICES 权限的用户打开设备进行写入。

以下示例显示了字符驱动程序的 open(9E) 例程。

示例 15–2 字符驱动程序 open(9E) 例程

```
static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *credp)
{
    minor_t        instance;

    if (getminor(*devp)         /* if device pointer is invalid */
        return (EINVAL);
    instance = getminor(*devp); /* one-to-one example mapping */
    /* Is the instance attached? */
    if (ddi_get_soft_state(statep, instance) == NULL)
        return (ENXIO);
    /* verify that otyp is appropriate */
    if (otyp != OTYP_CHR)
        return (EINVAL);
    if ((flag & FWRITE) && drv_priv(credp) == EPERM)
        return (EPERM);
    return (0);
}
```

close() 入口点（字符驱动程序）

close(9E) 的语法如下所示：

```
int xxclose(dev_t dev, int flag, int otyp, cred_t *credp);
```

close() 应执行任何必要的清除操作，以完成次要设备的使用，并准备好设备（以及驱动程序）以便再次被打开。例如，可能已使用独占访问 (FEXCL) 标志调用了打开例程。对 close(9E) 的调用允许其他打开例程继续运行。close(9E) 可以执行的其他功能包括：

- 等待输出缓冲区中的 I/O 执行完毕而后返回
- 反绕磁带（磁带设备）

- 挂断电话（调制解调器设备）

如果因为外部条件（例如流量控制）而造成 I/O 执行延迟，则等待 I/O 执行完毕的驱动程序会一直等待下去。有关如何避免此问题的信息，请参见第 72 页中的 “线程无法接收信号”。

I/O 请求处理

本节将详细讨论 I/O 请求处理。

用户地址

当用户线程发出 write(2) 系统调用时，该线程会传递用户空间中某个缓冲区的地址：

```
char buffer[] = "python";
count = write(fd, buffer, strlen(buffer) + 1);
```

系统通过分配 iovec(9S) 结构，并将 iov_base 字段设置为传递给 write(2) 的地址（在本例中为 buffer），来生成 uio(9S) 结构以描述此传输。uio(9S) 结构将被传递到驱动程序 write(9E) 例程。有关 uio(9S) 结构的详细信息，请参见第 250 页中的 “向量化的 I/O”。

iovec(9S) 中的地址位于用户空间而非内核空间。因此，既不能保证该地址当前处于内存中，也不能保证该地址是有效地址。无论哪一种情况，从设备驱动程序或从内核访问用户地址都会导致系统崩溃。因此，设备驱动程序永远不应该直接访问用户地址，而是应使用 Oracle Solaris DDI/DKI 中的数据传输例程，向内核中传送数据或从内核中读取数据。这些例程能够处理页面错误。DDI/DKI 例程可以读取适当的用户页面，以便继续进行透明复制。这些例程也可以在发生无效访问时返回错误。

使用 copyout(9F) 可将数据从内核空间复制到用户空间。copyin(9F) 可将数据从用户空间复制到内核空间。ddi_copyout(9F) 和 ddi_copyin(9F) 的运行方式与它们类似，但要在 ioctl(9E) 例程中使用。可以对每个 iovec(9S) 结构描述的缓冲区使用 copyin(9F) 和 copyout(9F)，或者 uiomove(9F) 可以对驱动程序或设备内存的连续区域执行完整的数据传入或传出操作。

向量化的 I/O

在字符驱动程序中，传输由 uio(9S) 结构进行描述。uio(9S) 结构包含有关传输方向和传输大小以及传输的其中一端缓冲区数组的信息。另一端就是设备。

uio(9S) 结构包含以下成员：

```
iovec_t       *uio_iov;       /* base address of the iovec */
                              /* buffer description array */
int           uio_iovcnt;     /* the number of iovec structures */
off_t         uio_offset;     /* 32-bit offset into file where */
                              /* data is transferred from or to */
offset_t      uio_loffset;    /* 64-bit offset into file where */
                              /* data is transferred from or to */
uio_seg_t     uio_segflg;     /* identifies the type of I/O transfer */
                              /* UIO_SYSSPACE:  kernel <-> kernel */
                              /* UIO_USERSPACE: kernel <-> user */
short         uio_fmode;      /* file mode flags (not driver setTable) */
daddr_t       uio_limit;      /* 32-bit ulimit for file (maximum */
                              /* block offset). not driver settable. */
diskaddr_t    uio_llimit;     /* 64-bit ulimit for file (maximum block */
                              /* block offset). not driver settable. */
int           uio_resid;      /* amount (in bytes) not */
                              /* transferred on completion */
```

uio(9S) 结构将被传递到驱动程序 read(9E) 和 write(9E) 入口点。之所以广泛应用此结构，是为了支持称作**集中写入**和**分散读取**的操作。向设备写入数据时，待写入的数据缓冲区在应用程序内存中不必是连续的。同样，从设备传输到内存的数据虽然不属于连续流，但也可以写入应用程序内存的非连续区域。有关分散/集中式 I/O 的更多信息，请参见 readv(2)、writev(2)、pread(2) 和 pwrite(2) 手册页。

每个缓冲区都由一个 iovec(9S) 结构描述。该结构包含指向数据区域的指针以及待传输的字节数。

```
caddr_t    iov_base;    /* address of buffer */
int        iov_len;     /* amount to transfer */
```

uio 结构包含指向 iovec(9S) 结构数组的指针。此数组的基本地址保存在 uio_iov 中，元素数目保存在 uio_iovcnt 中。

uio_offset 字段包含设备的 32 位偏移位址，应用程序需要在此处开始传输。uio_loffset 用于 64 位文件偏移。如果设备不支持偏移的概念，则可以安全地忽略这些字段。驱动程序会解释 uio_offset 或 uio_loffset，但不会同时解释两者。如果驱动程序在 cb_ops(9S) 结构中设置了 D_64BIT 标志，则该驱动程序应使用 uio_loffset。

uio_resid 字段起初是待传输的字节数，即 uio_iov 中所有 iov_len 字段的总和。此字段在返回之前**必须**由驱动程序设置为**未**传输的字节数。read(2) 和 write(2) 系统调用使用来自 read(9E) 和 write(9E) 入口点的返回值，来确定失败的传输。如果出现故障，这些例程将返回 -1。如果返回值指示成功，系统调用将返回所请求的字节数减去 uio_resid。如果驱动程序没有更改 uio_resid，则 read(2) 和 write(2) 调用将返回 0。返回值 0 表明文件结束，即使已经传输了所有数据也是如此。

支持例程 uiomove(9F)、physio(9F) 和 aphysio(9F) 直接更新 uio(9S) 结构。这些支持例程更新设备偏移以用于数据传输。如果驱动程序用于使用位置概念的可查找设备，则 uio_offset 或 uio_loffset 字段都不需要调整。以此方式对设备执行的 I/O 操作受 uio_offset 或 uio_loffset 的最大可能值约束。对磁盘的原始 I/O 操作即是此用法的一个示例。

如果设备没有位置概念，则驱动程序会采取下列步骤：

1. 保存 uio_offset 或 uio_loffset。
2. 执行 I/O 操作。
3. 将 uio_offset 或 uio_loffset 恢复为字段的初始值。

以此方式对设备执行的 I/O 操作不受 uio_offset 或 uio_loffset 的最大可能值约束。此种用法的一个示例是串行线路上的 I/O 操作。

以下示例说明了在 read(9E) 函数中保留 uio_loffset 的一种方法。

```
static int
xxread(dev_t dev, struct uio *uio_p, cred_t *cred_p)
{
    offset_t off;
    /* ... */
    off = uio_p->uio_loffset;  /* save the offset */
    /* do the transfer */
    uio_p->uio_loffset = off;  /* restore it */
}
```

同步 I/O 与异步 I/O 之间的差别

数据传输可以是**同步的**，也可以是**异步的**。决定因素取决于调度传输的入口点是立即返回还是等到 I/O 操作完成之后。

read(9E) 和 write(9E) 入口点都是同步入口点。传输在 I/O 操作完成之前不得返回。待例程返回值时，进程就会知道传输是否成功。

aread(9E) 和 awrite(9E) 入口点都是异步入口点。异步入口点调度 I/O 并立即返回。返回时，发出请求的进程即知道 I/O 被调度，并且随后必须确定 I/O 的状态。同时，该进程还可以执行其他操作。

对于发送到内核的异步 I/O 请求，不要求进程在 I/O 处理过程中等待。一个进程可以执行多个 I/O 请求，并允许内核处理数据传输细节。通过异步 I/O 请求，事务处理等应用程序可以使用并发编程方法来提高性能或缩短响应时间。但是，因使用异步 I/O 的应用程序而改善的任何性能，必须以增加编程复杂性为代价。

数据传输方法

可以使用程控 I/O 或 DMA 传输数据。同步或异步入口点都可以使用这些数据传输方法，具体视设备的功能而定。

程控 I/O 传输

程控 I/O 设备依赖 CPU 来执行数据传输。程控 I/O 数据传输与设备寄存器的其他读写操作相同。可使用各种数据访问例程，从设备内存读取值或向设备内存中存储值。

可以使用 uiomove(9F) 将数据传输到一些程控 I/O 设备。uiomove(9F) 在 uio(9S) 结构所定义的用户空间与内核之间传输数据。uiomove() 可以处理缺页，因此不必锁定要向其传输数据的内存。uiomove() 还会更新 uio(9S) 结构中的 uio_resid 字段。以下示例说明了编写 ramdisk read(9E) 例程的一种方法。它使用同步 I/O，并依赖 ramdisk 状态结构中下列字段的存在：

```
caddr_t    ram;        /* base address of ramdisk */
int        ramsize;    /* size of the ramdisk */
```

示例 15–3 使用 uiomove(9F) 的 ramdisk read(9E) 例程

```
static int
rd_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
     rd_devstate_t     *rsp;

     rsp = ddi_get_soft_state(rd_statep, getminor(dev));
     if (rsp == NULL)
       return (ENXIO);
     if (uiop->uio_offset >= rsp->ramsize)
       return (EINVAL);
     /*
      * uiomove takes the offset into the kernel buffer,
      * the data transfer count (minimum of the requested and
      * the remaining data), the UIO_READ flag, and a pointer
      * to the uio structure.
      */
     return (uiomove(rsp->ram + uiop->uio_offset,
         min(uiop->uio_resid, rsp->ramsize - uiop->uio_offset),
         UIO_READ, uiop));
}
```

另一个程控 I/O 示例是每次直接向设备内存中写入一字节数据的驱动程序。每一字节都是使用 uio(9S) 从 uwritec(9F) 结构中检索到的。随后该字节被发送到设备中。read(9E) 可以使用 ureadc(9F) 将字节从设备传输到由 uio(9S) 结构描述的区域中。

示例 15–4 使用 uwritec(9F) 的程控 I/O write(9E) 例程

```
static int
xxwrite(dev_t dev, struct uio *uiop, cred_t *credp)
{
    int    value;
    struct xxstate     *xsp;

    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL)
        return (ENXIO);
    /* if the device implements a power manageable component, do this: */
```

示例 15–4 使用 uwritec(9F) 的程控 I/O write(9E) 例程 （续）

```
    pm_busy_component(xsp->dip, 0);
    if (xsp->pm_suspended)
        pm_raise_power(xsp->dip, normal power);

    while (uiop->uio_resid > 0) {
        /*
         * do the programmed I/O access
         */
        value = uwritec(uiop);
        if (value == -1)
               return (EFAULT);
        ddi_put8(xsp->data_access_handle, &xsp->regp->data,
            (uint8_t)value);
        ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
            START_TRANSFER);
        /*
         * this device requires a ten microsecond delay
         * between writes
         */
        drv_usecwait(10);
    }
    pm_idle_component(xsp->dip, 0);
    return (0);
}
```

DMA 传输（同步）

字符驱动程序通常在 physio(9F) 和 read(9E) 中使用 write(9E) 来设置 DMA 传输，如示例 15–5 中所示。

```
int physio(int (*strat)(struct buf *), struct buf *bp,
     dev_t dev, int rw, void (*mincnt)(struct buf *),
     struct uio *uio);
```

physio(9F) 要求驱动程序提供 strategy(9E) 例程的地址。physio(9F) 可确保内存空间处于锁定状态，即在数据传输期间内存页不能被换出。由于 DMA 传输不能处理缺页，因此这种锁定对 DMA 传输来说十分必要。physio(9F) 还提供了一种将较大的传输分解为一系列较小的、更易于管理的传输的自动方法。有关更多信息，请参见第 256 页中的 “minphys() 入口点”。

示例 15–5 使用 physio(9F) 的 read(9E) 和 write(9E) 例程

```
static int
xxread(dev_t dev, struct uio *uiop, cred_t *credp)
{
     struct xxstate *xsp;
     int ret;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
        return (ENXIO);
```

示例 15–5 使用 physio(9F) 的 read(9E) 和 write(9E) 例程 （续）

```
     ret = physio(xxstrategy, NULL, dev, B_READ, xxminphys, uiop);
     return (ret);
}

static int
xxwrite(dev_t dev, struct uio *uiop, cred_t *credp)
{
     struct xxstate *xsp;
     int ret;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
        return (ENXIO);
     ret = physio(xxstrategy, NULL, dev, B_WRITE, xxminphys, uiop);
     return (ret);
}
```

在对 physio(9F) 的调用中，xxstrategy 是指向驱动程序 strategy() 例程的指针。如果将 NULL 作为 buf(9S) 结构指针传递，则指示 physio(9F) 分配 buf(9S) 结构。如果驱动程序必须向 physio(9F) 提供 buf(9S) 结构，应使用 getrbuf(9F) 来分配该结构。如果传输成功完成，physio(9F) 返回零；如果传输失败，则返回错误号。调用 strategy(9E) 后，physio(9F) 会调用 biowait(9F) 来进行阻塞，直到传输完成或失败。physio(9F) 的返回值由 buf(9S) 结构中 bioerror(9F) 设置的错误字段确定。

DMA 传输（异步）

支持 aread(9E) 和 awrite(9E) 的字符驱动程序使用 aphysio(9F) 而非 physio(9F)。

```
int aphysio(int (*strat)(struct buf *), int (*cancel)(struct buf *),
     dev_t dev, int rw, void (*mincnt)(struct buf *),
     struct aio_req *aio_reqp);
```

注 – anocancel(9F) 的地址是唯一一个当前可作为第二个参数传递到 aphysio(9F) 的值。

---

aphysio(9F) 要求驱动程序传递 strategy(9E) 例程的地址。aphysio(9F) 可确保内存空间处于锁定状态，即在数据传输期间内存页不能被换出。由于 DMA 传输不能处理缺页，因此这种锁定对 DMA 传输来说十分必要。aphysio(9F) 还提供了一种将较大的传输分解为一系列较小的、更易于管理的传输的自动方法。有关更多信息，请参见第 256 页中的 “minphys() 入口点”。

示例 15–5 和示例 15–6 说明了 aread(9E) 和 awrite(9E) 入口点与 read(9E) 和 write(9E) 入口点之间的轻微差异。这种差异主要在于，前两者使用 aphysio(9F)，而非 physio(9F)。

示例 15–6 使用 aphysio(9F) 的 aread(9E) 和 awrite(9E) 例程

```
static int
xxaread(dev_t dev, struct aio_req *aiop, cred_t *cred_p)
{
     struct xxstate *xsp;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
         return (ENXIO);
     return (aphysio(xxstrategy, anocancel, dev, B_READ,
     xxminphys, aiop));
}

static int
xxawrite(dev_t dev, struct aio_req *aiop, cred_t *cred_p)
{
     struct xxstate *xsp;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
        return (ENXIO);
     return (aphysio(xxstrategy, anocancel, dev, B_WRITE,
     xxminphys,aiop));
}
```

在对 aphysio(9F) 的调用中，xxstrategy() 是指向驱动程序策略例程的指针。*aiop* 是指向 aio_req(9S) 结构的指针。*aiop* 将被传递到 aread(9E) 和 awrite(9E)。aio_req(9S) 描述了数据在用户空间中的存储位置。如果成功调度了 I/O 请求，aphysio(9F) 返回零；如果调度失败，则返回错误号。调用 strategy(9E) 后，aphysio(9F) 会返回而不等待 I/O 完成或失败。

minphys() 入口点

minphys() 入口点是指向要由 physio(9F) 或 aphysio(9F) 调用的函数的指针。*xxminphys* 的用途在于确保所请求的传输的大小不超过驱动程序施加的限制。如果用户请求较大的传输，则会重复调用 strategy(9E)，这就要求每次都不能超过强加的限制。因为 DMA 资源有限，所以该方法非常重要。对于慢速设备（例如打印机）的驱动程序，应避免其长时间占用资源。

通常，驱动程序会传递内核函数 minphys(9F) 的地址，但驱动程序也可以定义自己的 xxminphys() 例程。xxminphys() 的职责是使 buf(9S) 结构的 b_bcount 字段保持在驱动程序限制内。驱动程序还应遵循其他系统限制。例如，驱动程序的 xxminphys() 例程应该在设置 b_bcount 字段之后且返回之前调用系统 minphys(9F) 例程。

示例 15–7 minphys(9F) 例程

```
#define XXMINVAL (512 << 10)    /* 512 KB */
static void
xxminphys(struct buf *bp)
{
```

示例 15–7 minphys(9F) 例程 （续）

```
    if (bp->b_bcount > XXMINVAL)
        bp->b_bcount = XXMINVAL
    minphys(bp);
}
```

strategy() 入口点

strategy(9E) 例程源于块驱动程序。策略函数因实现用于对块设备的 I/O 请求的有效排队的策略而得名。面向字符设备的驱动程序也可以使用 strategy(9E) 例程。在这里提供的字符 I/O 模型中，strategy(9E) 不维护请求队列，只是一次为一个请求提供服务。

在以下示例中，用于面向字符的 DMA 设备的 strategy(9E) 例程为同步数据传输分配 DMA 资源。strategy() 通过对设备寄存器进行编程来启动此命令。有关详细说明，请参见第 9 章，直接内存访问 (Direct Memory Access, DMA)。

注 – strategy(9E) 不会以参数形式接收设备编号 (dev_t)。设备编号是从传递给 strategy(9E) 的 buf(9S) 结构中的 b_edev 字段检索到的。

示例 15–8 strategy(9E) 例程

```
static int
xxstrategy(struct buf *bp)
{
     minor_t            instance;
     struct xxstate     *xsp;
     ddi_dma_cookie_t   cookie;

     instance = getminor(bp->b_edev);
     xsp = ddi_get_soft_state(statep, instance);
     /* ... */
      * If the device has power manageable components,
      * mark the device busy with pm_busy_components(9F),
      * and then ensure that the device is
      * powered up by calling pm_raise_power(9F).
      */
     /* Set up DMA resources with ddi_dma_alloc_handle(9F) and
      * ddi_dma_buf_bind_handle(9F).
      */
     xsp->bp = bp; /* remember bp */
     /* Program DMA engine and start command */
     return (0);
}
```

注 – 虽然声明了 strategy() 返回 int，但 strategy() 必须总是返回零。

在完成 DMA 传输时，设备会产生中断，从而导致对中断例程的调用。在以下示例中，xxintr() 接收指向可能产生中断的设备的状态结构的指针。

示例 15–9 中断例程

```
static u_int
xxintr(caddr_t arg)
{
     struct xxstate *xsp = (struct xxstate *)arg;
     if ( /* device did not interrupt */ ) {
        return (DDI_INTR_UNCLAIMED);
     }
     if ( /* error */ ) {
        /* error handling */
     }
     /* Release any resources used in the transfer, such as DMA resources.
      * ddi_dma_unbind_handle(9F) and ddi_dma_free_handle(9F)
      * Notify threads that the transfer is complete.
      */
     biodone(xsp->bp);
     return (DDI_INTR_CLAIMED);
}
```

驱动程序通过调用 bioerror(9F) 来指示错误。当传输完成或者使用 bioerror(9F) 指示错误后，驱动程序必须调用 biodone(9F)。

映射设备内存

通过内存映射，用户线程可以直接访问某些设备（如帧缓存器）的内存。这些设备的驱动程序通常不支持 read(9E) 和 write(9E) 接口。相反，这些驱动程序支持使用 devmap(9E) 入口点的内存映射。例如，帧缓存器驱动程序可以实现 devmap(9E) 入口点，以允许将帧缓存器映射到用户线程。

调用 devmap(9E) 入口点可以将设备内存或内核内存导出到用户应用程序。devmap() 函数是在 segmap(9E) 内从 devmap_setup(9F) 调用的，或者是代表 ddi_devmap_segmap(9F) 调用的。

segmap(9E) 入口点负责设置 mmap(2) 系统调用所请求的内存映射。许多内存映射设备的驱动程序使用 ddi_devmap_segmap(9F) 作为入口点，而不是定义自己的 segmap(9E) 例程。

有关详细信息，请参见第 10 章，映射设备和内核内存和第 11 章，设备上下文管理。

对文件描述符执行多路复用 I/O 操作

一个线程有时需要处理多个文件描述符上的 I/O。需要从温度感应设备读取温度并将此温度报告给交互显示的应用程序就是一个示例。在与用户再次交互之前等待温度时，发出读取请求但没有可用数据的程序不会进入阻塞状态。

poll(2) 系统调用为用户提供了对一组引用打开的文件的文件描述符执行多路复用 I/O 操作的机制。poll(2) 识别那些在它们上面程序可以发送或接收数据而不会阻塞的文件描述符，或在它们上面特定事件已发生的文件描述符。

要允许某个程序轮询字符驱动程序，该驱动程序必须实现 chpoll(9E) 入口点。当用户进程对与设备相关联的文件描述符发出 poll(2) 时，系统会调用 chpoll(9E)。需要支持轮询的非 STREAMS 字符设备驱动程序使用 chpoll(9E) 入口点例程。

chpoll(9E) 函数使用以下语法：

```
int xxchpoll(dev_t dev, short events, int anyyet, short *reventsp,
     struct pollhead **phpp);
```

在 chpoll(9E) 入口点中，驱动程序必须遵循下列规则：

- 在调用 chpoll(9E) 入口点时执行以下算法：

  ```
  if ( /* events are satisfied now */ ) {
      *reventsp = mask_of_satisfied_events
  } else {
      *reventsp = 0;
      if (!anyyet)
          *phpp = &local_pollhead_structure;
  }
  return (0);
  ```

  有关要检查的事件的论述，请参见 chpoll(9E) 手册页。然后，chpoll(9E) 入口点应通过在 **reventsp* 中设置返回事件，来返回满足要求的事件的掩码。

  如果没有发生任何事件，则清除事件的返回字段。如果未设置 *anyyet* 字段，则驱动程序必须返回 pollhead 结构的实例。通常在状态结构中分配 pollhead 结构。驱动程序应该将 pollhead 结构视为不透明。不能引用任何 pollhead 字段。

- 只要出现示例 15–10 中列出的 events 类型的设备条件，就会调用 pollwakeup(9F)。一次只能对一个事件调用此函数。当出现这种条件时，可以在中断例程中调用 pollwakeup(9F)。

示例 15–10 和示例 15–11 说明了如何实现轮询规程以及如何使用 pollwakeup(9F)。

以下示例说明如何处理 POLLIN 和 POLLERR 事件。首先，驱动程序读取状态寄存器以确定设备的当前状态。参数 events 指定驱动程序应该检查哪些条件。如果出现适当的条件，驱动程序会在 **reventsp* 中设置相应的位。如果未出现任何条件并且没有设置 *anyyet*，则会在 **phpp* 中返回 pollhead 结构的地址。

示例 15–10 chpoll(9E) 例程

```
static int
xxchpoll(dev_t dev, short events, int anyyet,
    short *reventsp, struct pollhead **phpp)
{
     uint8_t status;
     short revent;
     struct xxstate *xsp;

     xsp = ddi_get_soft_state(statep, getminor(dev));
     if (xsp == NULL)
         return (ENXIO);
     revent = 0;
     /*
      * Valid events are:
      * POLLIN | POLLOUT | POLLPRI | POLLHUP | POLLERR
      * This example checks only for POLLIN and POLLERR.
      */
     status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
     if ((events & POLLIN) && data available to read) {
        revent |= POLLIN;
     }
     if (status & DEVICE_ERROR) {
        revent |= POLLERR;
     }
     /* if nothing has occurred */
     if (revent == 0) {
        if (!anyyet) {
        *phpp = &xsp->pollhead;
        }
     }
       *reventsp = revent;
     return (0);
}
```

以下示例说明如何使用 pollwakeup(9F) 函数。当出现支持的条件时，通常会在中断例程中调用 pollwakeup(9F) 函数。中断例程从状态寄存器中读取状态并检查条件。然后，该例程为每个事件调用 pollwakeup(9F)，以便有可能通知轮询线程再次进行检查。请注意，不能在持有任何锁定的情况下调用 pollwakeup(9F)，这是因为如果另一个例程尝试进入 chpoll(9E) 并获取相同的锁，则会导致死锁。

示例 15–11 支持 chpoll(9E) 的中断例程

```
static u_int
xxintr(caddr_t arg)
{
     struct xxstate *xsp = (struct xxstate *)arg;
     uint8_t    status;
     /* normal interrupt processing */
     /* ... */
     status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
     if (status & DEVICE_ERROR) {
        pollwakeup(&xsp->pollhead, POLLERR);
     }
```

示例 15–11 支持 chpoll(9E) 的中断例程 （续）

```
        if ( /* just completed a read */ ) {
           pollwakeup(&xsp->pollhead, POLLIN);
        }
        /* ... */
        return (DDI_INTR_CLAIMED);
}
```

其他 I/O 控制

当用户线程对与设备相关联的文件描述符发出 ioctl(9E) 系统调用时，就会调用 ioctl(2) 例程。I/O 控制机制是获取和设置设备特定参数的统称。该机制经常用于设置设备特定模式（通过设置内部驱动程序软件标志或将命令写入设备）。也可以使用该控制机制向用户返回有关当前设备状态的信息。简而言之，控制机制可以做应用程序和驱动程序需要完成的任何事情。

ioctl() 入口点（字符驱动程序）

```
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
     cred_t *credp, int *rvalp);
```

cmd 参数表明应该执行哪个 ioctl(9E) 命令。根据约定，命令的 8-15 位指示与 I/O 控制命令有关的驱动程序。通常，字符的 ASCII 代码代表该驱动程序。驱动程序特定命令位于 0-7 位。以下示例说明了一些 I/O 命令的创建：

```
#define XXIOC            ('x' << 8)   /* 'x' is a character that represents device xx */
#define XX_GET_STATUS    (XXIOC | 1)  /* get status register */
#define XX_SET_CMD       (XXIOC | 2)  /* send command */
```

对 *arg* 的解释视命令而定。在驱动程序文档或手册页中应该介绍了这些 I/O 控制命令。在公共头文件中还会定义命令，以便应用程序能够确定命令的名称、命令执行的操作以及命令以 *arg* 的形式接受或返回的内容。进出驱动程序的任何 *arg* 数据传输都必须由驱动程序执行。

特定种类的设备（如帧缓存器或磁盘）必须支持 I/O 控制请求的标准集。这些标准 I/O 控制接口在 Solaris 8 Reference Manual Collection 中进行了介绍。例如，fbio(7I) 介绍了帧缓存器必须支持的 I/O 控制，而 dkio(7I) 则介绍了标准的磁盘 I/O 控制。有关 I/O 控制的更多信息，请参见第 261 页中的 “其他 I/O 控制”。

驱动程序必须使用 ddi_copyin(9F) 从用户级别的应用程序向内核级别的应用程序传输 *arg* 数据。驱动程序必须使用 ddi_copyout(9F) 从内核级别向用户级别传输数据。在这两种情况下，如果使用 ddi_copyin(9F) 或 ddi_copyout(9F) 失败，则会导致系统出现紧急情况。如果体系结构将内核地址空间和用户地址空间分开，或者用户地址空间被换出，系统都会出现紧急情况。

对于每个支持的 ioctl(9E) 请求，ioctl(9E) 通常是 switch 语句。

示例 15–12 ioctl(9E) 例程

```
static int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *credp, int *rvalp)
{
    uint8_t        csr;
    struct xxstate     *xsp;

    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL) {
        return (ENXIO);
    }
    switch (cmd) {
    case XX_GET_STATUS:
        csr = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
        if (ddi_copyout(&csr, (void *)arg, sizeof (uint8_t), mode) != 0) {
            return (EFAULT);
        }
        break;
    case XX_SET_CMD:
        if (ddi_copyin((void *)arg, &csr, sizeof (uint8_t), mode) != 0) {
            return (EFAULT);
        }
        ddi_put8(xsp->data_access_handle, &xsp->regp->csr, csr);
        break;
    default:
        /* generic "ioctl unknown" error */
        return (ENOTTY);
    }
    return (0);
}
```

cmd 变量识别特定的设备控制操作。如果 *arg* 包含用户虚拟地址，则会出现问题。ioctl(9E) 必须调用 ddi_copyin(9F) 或 ddi_copyout(9F)，以便在 *arg* 指向的应用程序中的数据结构与驱动程序之间传输数据。在示例 15–12 中，对于 XX_GET_STATUS 请求，*xsp->regp->csr* 的内容会被复制到 *arg* 中的地址。ioctl(9E) 可以在 **rvalp* 中存储任何作为成功发出请求的 ioctl(2) 系统调用的返回值的整数值。应当避免返回负值，例如 -1。许多应用程序假定负值表示失败。

以下示例说明了使用上一段落中所讨论的 I/O 控制的应用程序。

示例 15–13 使用 ioctl(9E)

```
#include <sys/types.h>
#include "xxio.h"     /* contains device's ioctl cmds and args */
int
main(void)
{
     uint8_t    status;
     /* ... */
     /*
```

示例 15–13 使用 ioctl(9E) （续）

```
        * read the device status
        */
       if (ioctl(fd, XX_GET_STATUS, &status) == -1) {
           /* error handling */
       }
       printf("device status %x\n", status);
       exit(0);
}
```

对有 64 位处理能力的设备驱动程序的 I/O 控制支持

Oracle Solaris 内核在适当的硬件上以 64 位模式运行，既支持 32 位应用程序，也支持 64 位应用程序。要求 64 位设备驱动程序同时支持来自这两种处理能力的程序的 I/O 控制命令。32 位程序与 64 位程序的差异在于 C 语言类型模型。32 位程序是 ILP32，而 64 位程序是 LP64。有关 C 数据类型模型的信息，请参见附录 C，使设备驱动程序支持 64 位。

如果程序和内核之间传输的数据具有不同的格式，则驱动程序必须能够处理这种模型不匹配。处理模型不匹配需要对数据进行适当的调整。

要确定是否存在模型不匹配，ioctl(9E) 模式参数会将数据模型位传递到驱动程序。如示例 15–14 中所示，该模式参数随后会被传递到 ddi_model_convert_from(9F)，以确定是否有必要进行模型转换。

模式参数的标志子字段用于将数据模型传递到 ioctl(9E) 例程中。可以将此标志设置为以下值之一：

- DATAMODEL_ILP32
- DATAMODEL_LP64

可以有条件地定义 FNATIVE，以匹配内核实现的数据模型。应使用 FMODELS 掩码来提取 *mode* 参数的标志。驱动程序随后会对数据模型进行明确检查，以确定如何复制应用程序的数据结构。

DDI 函数 ddi_model_convert_from(9F) 是一个公用例程，可帮助一些驱动程序完成它们的 ioctl() 调用。该函数将用户应用程序的数据类型模型用作参数，并返回下列值之一：

- DDI_MODEL_ILP32－从 ILP32 应用程序进行转换
- DDI_MODEL_NONE－无需转换

如果不必进行数据转换，则会返回 DDI_MODEL_NONE。当应用程序和驱动程序具有相同的数据模型时，便会发生这种情况。DDI_MODEL_ILP32 将返回到被编译为 LP64 模式而且能够与 32 位应用程序进行通信的驱动程序。

在以下示例中，驱动程序复制了包含用户地址的数据结构。数据结构的处理能力从 ILP32 更改为 LP64。相应地，此 64 位驱动程序在与 32 位应用程序进行通信时使用 32 位版本的结构。

示例 15–14 用于支持 32 位应用程序和 64 位应用程序的 ioctl(9E) 例程

```
struct args32 {
    uint32_t    addr;    /* 32-bit address in LP64 */
    int     len;
}
struct args {
    caddr_t     addr;    /* 64-bit address in LP64 */
    int     len;
}

static int
xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *credp, int *rvalp)
{
    struct  xxstate  *xsp;
    struct  args     a;
    xsp = ddi_get_soft_state(statep, getminor(dev));
    if (xsp == NULL) {
        return (ENXIO);
    }
    switch (cmd) {
    case XX_COPYIN_DATA:
        switch(ddi_model_convert_from(mode)) {
        case DDI_MODEL_ILP32:
        {
            struct args32 a32;

            /* copy 32-bit args data shape */
            if (ddi_copyin((void *)arg, &a32,
                sizeof (struct args32), mode) != 0) {
                return (EFAULT);
            }
            /* convert 32-bit to 64-bit args data shape */
            a.addr = a32.addr;
            a.len = a32.len;
            break;
        }
        case DDI_MODEL_NONE:
            /* application and driver have same data model. */
            if (ddi_copyin((void *)arg, &a, sizeof (struct args),
                mode) != 0) {
                return (EFAULT);
            }
        }
        /* continue using data shape in native driver data model. */
        break;

    case XX_COPYOUT_DATA:
        /* copyout handling */
        break;
    default:
        /* generic "ioctl unknown" error */
```

示例 15–14 用于支持 32 位应用程序和 64 位应用程序的 ioctl(9E) 例程 （续）

```
        return (ENOTTY);
    }
    return (0);
}
```

处理 copyout() 溢出

驱动程序有时需要将不再适于 32 位大小结构的本机数值复制出来。在这种情况下，驱动程序应向调用方返回 EOVERFLOW。EOVERFLOW 用于表明接口中的数据类型太小，无法保存要返回的值，如以下示例中所示。

示例 15–15 处理 copyout (9F) 溢出

```
int
    xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
     cred_t *cr, int *rval_p)
    {
        struct resdata res;
        /* body of driver */
        switch (ddi_model_convert_from(mode & FMODELS)) {
        case DDI_MODEL_ILP32: {
            struct resdata32 res32;

            if (res.size > UINT_MAX)
                    return (EOVERFLOW);
            res32.size = (size32_t)res.size;
            res32.flag = res.flag;
            if (ddi_copyout(&res32,
                (void *)arg, sizeof (res32), mode))
                    return (EFAULT);
        }
        break;

        case DDI_MODEL_NONE:
            if (ddi_copyout(&res, (void *)arg, sizeof (res), mode))
                    return (EFAULT);
            break;
        }
        return (0);
    }
```

32 位和 64 位数据结构宏

示例 15–15 中的方法适用于许多驱动程序。另一种方案是使用 <sys/model.h> 中提供的数据结构宏在应用程序和内核之间移动数据。从功能角度看，这些宏减少了代码混乱问题，并使代码的表现形式完全相同。

示例 15–16 使用数据结构宏移动数据

```
int
    xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
        cred_t *cr, int *rval_p)
    {
        STRUCT_DECL(opdata, op);

        if (cmd != OPONE)
            return (ENOTTY);

        STRUCT_INIT(op, mode);

        if (copyin((void *)arg,
            STRUCT_BUF(op), STRUCT_SIZE(op)))
                return (EFAULT);

        if (STRUCT_FGET(op, flag) != XXACTIVE ||
            STRUCT_FGET(op, size) > XXSIZE)
                return (EINVAL);
        xxdowork(device_state, STRUCT_FGET(op, size));
        return (0);
}
```

结构宏如何工作？

在 64 位设备驱动程序中，结构宏使得两种处理能力的数据结构可以使用相同的内核内存片段。内存缓冲区保存数据结构的本机形式的内容，即 LP64 和 ILP32 形式。每种结构的访问是通过条件表达式实现的。如果驱动程序以 32 位方式编译，则仅支持一种数据模型（本机形式）。不使用条件表达式。

64 位版本的宏依赖于数据结构阴影版本的定义。阴影版本描述了使用固定宽度类型的 32 位接口。将 "32" 附加到本机数据结构名称，就形成了阴影数据结构的名称。为方便起见，将阴影结构的定义放置到与本机结构相同的文件中，以降低将来的维护成本。

这些宏可以采用下列参数：

structname 数据结构的本机形式的结构名称，即在 `struct` 关键字后输入的内容。

umodel 包含用户数据模型（例如 `FILP32` 或 `FLP64`）的标志字，从 ioctl(9E) 的模式参数中提取。

handle 此名称用于引用这些宏所处理的结构的特定实例。

fieldname 结构内部的字段的名称。

何时使用结构宏

宏使您能够仅对数据项的字段进行适当地引用。宏不提供采用基于数据模型的单独代码路径的方法。如果数据结构中的字段数量很大，则应避免使用宏。如果对这些字段的引用非常频繁，也应避免使用宏。

在实现宏的过程中，宏隐藏了数据模型之间的很多差异。因此，使用此接口编写的代码通常比较容易理解。如果驱动程序以 32 位方式编译，则生成的代码较为简洁，并且无需冗长的 `#ifdefs`，但仍保留了类型检查。

声明并初始化结构句柄

可以使用 STRUCT_DECL(9F) 和 STRUCT_INIT(9F) 声明和初始化句柄及空间，以便在栈中对 `ioctl` 进行解码。STRUCT_HANDLE(9F) 和 STRUCT_SET_HANDLE(9F) 可以声明和初始化句柄，但不在栈中分配空间。如果结构非常大，或者包含在其他某个数据结构中，则后面的宏比较有用。

注 – 因为 STRUCT_DECL(9F) 和 STRUCT_HANDLE(9F) 宏扩展为数据结构声明，所以这些宏在 C 代码中应该使用这些声明进行分组。

用于声明和初始化结构的宏如下所示：

`STRUCT_DECL(`*structname*`,` *handle*`)`
为 *structname* 数据结构声明一个名为 `handle` 的**结构句柄**。`STRUCT_DECL` 按其本机形式在栈中分配空间。假定本机形式大于或等于结构的 ILP32 形式。

`STRUCT_INIT(`*handle*`,` *umodel*`)`
将 *handle* 的数据模型初始化为 `umodel`。在对使用 STRUCT_DECL(9F) 声明的结构句柄进行任何访问之前，必须调用此宏。

`STRUCT_HANDLE(`*structname*`,` *handle*`)`
声明调用了 *handle* 的**结构句柄**。它与 STRUCT_DECL(9F) 相对。

`STRUCT_SET_HANDLE(`*handle* `,` *umodel*`,` *addr* `)`
将 *handle* 的数据模型初始化为 *umodel*，然后将 *addr* 设置为用于后续处理的缓冲区。在访问使用 STRUCT_DECL(9F) 声明的结构句柄之前，请调用此宏。

结构句柄的操作

用于在结构上执行操作的宏如下所示：

`size_t STRUCT_SIZE(`*handle* `)`
返回 *handle* 所引用的结构的大小（取决于该结构的嵌入式数据模型）。

typeof *fieldname* STRUCT_FGET(*handle*, *fieldname*)
返回 *handle* 所引用的数据结构中的指定字段。此字段为非指针类型。

typeof *fieldname* STRUCT_FGETP(*handle*, *fieldname*)
返回 *handle* 所引用的数据结构中的指定字段。此字段为指针类型。

STRUCT_FSET(*handle* , *fieldname*, *val*)
将 *handle* 所引用的数据结构中的指定字段设置为值 *val*。*val* 的类型应与 *fieldname* 的类型相匹配。此字段为非指针类型。

STRUCT_FSETP(*handle* , *fieldname*, *val*)
将 *handle* 所引用的数据结构中的指定字段设置为值 *val*。此字段为指针类型。

typeof *fieldname* *STRUCT_FADDR(*handle*, *fieldname*)
返回 *handle* 所引用的数据结构中的指定字段的地址。

struct structname *STRUCT_BUF(*handle*)
返回指向 *handle* 所描述的本机结构的指针。

其他操作

其他一些结构宏如下所示：

size_t SIZEOF_STRUCT(*struct_name*, *datamodel*)
返回 *struct_name* 的大小（取决于给定的数据模型）。

size_t SIZEOF_PTR(*datamodel*)
根据给定的数据模型，返回指针的大小。

◆ ◆ ◆ 第 16 章

16 块设备驱动程序

本章介绍块设备驱动程序的结构。内核将块设备视为一组可随机访问的逻辑块。文件系统使用一系列 buf(9S) 结构来缓存块设备和用户空间之间的数据块。只有块设备可以支持文件系统。

本章介绍有关以下主题的信息：

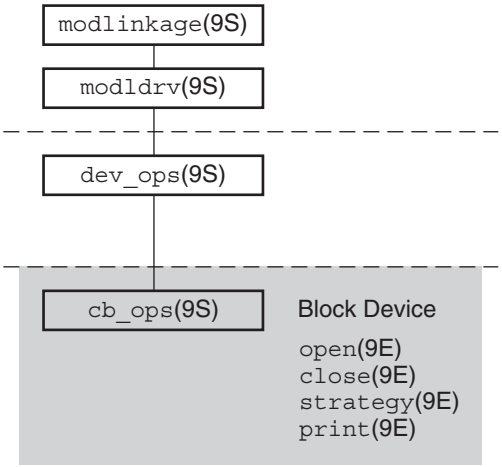
块驱动程序结构概述

图 16–1 显示了用来定义块设备驱动程序结构的数据结构和例程。设备驱动程序通常包括以下元素：

- 可装入设备的驱动程序段
- 设备配置部分
- 设备访问部分

下图中涂有阴影的设备访问部分列出了块驱动程序入口点。

图 16–1 块驱动程序结构图



对于每个设备驱动程序，都有一个 dev_ops(9S) 结构与之关联，该结构进而又引用 cb_ops(9S) 结构。有关驱动程序数据结构的详细信息，请参见第 6 章，驱动程序自动配置。

块设备驱动程序提供下列入口点：

- open(9E)
- close(9E)
- strategy(9E)
- print(9E)

注 – 可根据需要将其中一些入口点替换为 nodev(9F) 或 nulldev(9F)。

文件 I/O

文件系统是由目录和文件组成的树状分层结构。诸如 UNIX 文件系统 (UNIX File System, UFS) 之类的一些文件系统驻留在面向块的设备上。文件系统由 format(1M) 和 newfs(1M) 创建。

当应用程序向 UFS 文件系统上的普通文件发出 read(2) 或 write(2) 系统调用时，该文件系统可为所驻留在的块设备调用设备驱动程序 strategy(9E) 入口点。对于一个 read(2) 或 write(2) 系统调用，文件系统代码可多次调用 strategy(9E)。

文件系统代码为每个普通文件块确定逻辑设备地址，即**逻辑块编号**。然后采用指向块设备的 buf(9S) 结构的形式建立块 I/O 请求。驱动程序 strategy(9E) 入口点随后解释 buf(9S) 结构并完成请求。

块设备自动配置

attach(9E) 会为每个设备实例执行常见初始化任务：

- 分配每个实例的状态结构
- 映射设备寄存器
- 注册设备中断
- 初始化互斥变量和条件变量
- 创建可进行电源管理的组件
- 创建次要节点

块设备驱动程序创建类型为 S_IFBLK 的次要节点。因此，代表节点的块特殊文件会出现在 /devices 分层结构中。

块设备的逻辑设备名称位于 /dev/dsk 目录中，该名称由控制器编号、总线地址编号、磁盘编号和分片编号组成。如果节点类型设置为 DDI_NT_BLOCK 或 DDI_NT_BLOCK_CHAN，则这些名称由 devfsadm(1M) 程序创建。如果设备通过通道（即有附加寻址能力级别的总线）进行通信，则应该指定 DDI_NT_BLOCK_CHAN。SCSI 磁盘就是一个典型示例。DDI_NT_BLOCK_CHAN 可使总线地址字段 (t*N*) 出现在逻辑名称中。其他大多数设备则应该使用 DDI_NT_BLOCK。

次要设备指磁盘上的分区。对于每个次要设备，驱动程序必须创建 nblocks 或 Nblocks 属性。此整数属性给出了次要设备所支持的块数，以 DEV_BSIZE（即 512 字节）为单位。文件系统使用 nblocks 和 Nblocks 属性来确定设备限制。Nblocks 是 64 位版本的 nblocks。应该将 Nblocks 用于每个磁盘的存储容量超过 1 TB 的存储设备。有关更多信息，请参见第 75 页中的 “设备属性”。

示例 16–1 说明了一个典型的 attach(9E) 入口点，重点说明如何创建设备的次要节点和 Nblocks 属性。请注意，由于此示例使用 Nblocks 而非 nblocks，因此将调用 ddi_prop_update_int64(9F) 而非 ddi_prop_update_int(9F)。

作为附带说明，本示例还说明了如何使用 makedevice(9F) 为 ddi_prop_update_int64() 创建设备编号。makedevice 函数利用 ddi_driver_major(9F)，后者基于指向 dev_info_t 结构的指针生成主设备号。使用 ddi_driver_major() 与使用 getmajor(9F) 类似，后者用于获取 dev_t 结构指针。

示例 16–1 块驱动程序 attach() 例程

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    int instance = ddi_get_instance(dip);
    switch (cmd) {
      case DDI_ATTACH:
      /*
       * allocate a state structure and initialize it
       * map the devices registers
       * add the device driver's interrupt handler(s)
       * initialize any mutexes and condition variables
```

示例 16–1 块驱动程序 attach() 例程 （续）

```
         * read label information if the device is a disk
         * create power manageable components
         *
         * Create the device minor node. Note that the node_type
         * argument is set to DDI_NT_BLOCK.
         */
        if (ddi_create_minor_node(dip, "minor_name", S_IFBLK,
            instance, DDI_NT_BLOCK, 0) == DDI_FAILURE) {
            /* free resources allocated so far */
            /* Remove any previously allocated minor nodes */
            ddi_remove_minor_node(dip, NULL);
            return (DDI_FAILURE);
        }
        /*
         * Create driver properties like "Nblocks". If the device
         * is a disk, the Nblocks property is usually calculated from
         * information in the disk label.  Use "Nblocks" instead of
         * "nblocks" to ensure the property works for large disks.
         */
        xsp->Nblocks = size;
        /* size is the size of the device in 512 byte blocks */
        maj_number = ddi_driver_major(dip);
        if (ddi_prop_update_int64(makedevice(maj_number, instance), dip,
            "Nblocks", xsp->Nblocks) != DDI_PROP_SUCCESS) {
            cmn_err(CE_CONT, "%s: cannot create Nblocks property\n",
                ddi_get_name(dip));
            /* free resources allocated so far */
            return (DDI_FAILURE);
        }
        xsp->open = 0;
        xsp->nlayered = 0;
        /* ... */
        return (DDI_SUCCESS);

        case DDI_RESUME:
            /* For information, see Chapter 12, "Power Management," in this book. */
        default:
            return (DDI_FAILURE);
    }
}
```

控制设备访问

本节介绍块设备驱动程序中 open() 和 close() 函数的入口点。有关 open(9E) 和 close(9E) 的更多信息，请参见第 15 章，字符设备驱动程序。

open() 入口点（块驱动程序）

open(9E) 入口点用于访问给定的设备。当用户线程对与次要设备相关的块特殊文件发出 open(2) 或 mount(2) 系统调用时，或者当分层驱动程序调用 open(9E) 时，均会调用块驱动程序的 open(9E) 例程。有关更多信息，请参见第 270 页中的 “文件 I/O”。

open() 入口点会检查下列条件：

- 设备可以打开，即设备已联机并准备就绪。
- 设备可以应请求而打开。设备支持该操作。设备的当前状态与请求不冲突。
- 调用方具有打开设备的权限。

以下示例说明了块驱动程序 open(9E) 入口点。

示例 16–2 块驱动程序 open(9E) 例程

```
static int
xxopen(dev_t *devp, int flags, int otyp, cred_t *credp)
{
    minor_t         instance;
    struct xxstate        *xsp;

    instance = getminor(*devp);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (ENXIO);
    mutex_enter(&xsp->mu);
    /*
     * only honor FEXCL. If a regular open or a layered open
     * is still outstanding on the device, the exclusive open
     * must fail.
     */
    if ((flags & FEXCL) && (xsp->open || xsp->nlayered)) {
        mutex_exit(&xsp->mu);
        return (EAGAIN);
    }
    switch (otyp) {
      case OTYP_LYR:
          xsp->nlayered++;
          break;
      case OTYP_BLK:
          xsp->open = 1;
          break;
      default:
          mutex_exit(&xsp->mu);
          return (EINVAL);
    }
    mutex_exit(&xsp->mu);
    return (0);
}
```

otyp 参数用于指定设备的打开类型。OTYP_BLK 是块设备的典型打开类型。将 otyp 设置为 OTYP_BLK 后，可以多次打开设备。最后关闭设备的 OTYP_BLK 类型时，仅调用一次 close(9E)。如果将设备用作分层设备，则 otyp 设置为 OTYP_LYR。每次打开 OTYP_LYR 类型，分层驱动程序都会发出类型为 OTYP_LYR 的对应关闭。此示例跟踪每种类型的打开，因此驱动程序可以确定何时设备不用于 close(9E)。

close() 入口点（块驱动程序）

close(9E) 入口点使用与 open(9E) 相同的参数，但有一个例外。dev 是设备编号，而不是指向设备编号的指针。

close() 例程应采用与前面所述的 open(9E) 入口点所采用的相同的方式来检验 otyp。在以下示例中，close() 必须确定何时可以真正关闭设备。关闭受块打开数和分层打开数的影响。

示例 16–3 块设备 close(9E) 例程

```
static int
xxclose(dev_t dev, int flag, int otyp, cred_t *credp)
{
    minor_t instance;
    struct xxstate *xsp;

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (ENXIO);
    mutex_enter(&xsp->mu);
    switch (otyp) {
      case OTYP_LYR:
          xsp->nlayered--;
          break;
      case OTYP_BLK:
          xsp->open = 0;
          break;
      default:
          mutex_exit(&xsp->mu);
         return (EINVAL);
    }

    if (xsp->open || xsp->nlayered) {
        /* not done yet */
        mutex_exit(&xsp->mu);
        return (0);
    }
    /* cleanup (rewind tape, free memory, etc.) */
    /* wait for I/O to drain */
    mutex_exit(&xsp->mu);

    return (0);
}
```

strategy() 入口点

strategy(9E) 入口点用于从块设备读取数据缓冲区以及向块设备写入数据缓冲区。名称 *strategy* 指的是该入口点可以实现一些优化策略以对设备请求进行排序。

可以将 strategy(9E) 编写为一次处理一个请求，即同步传输。也可以将 strategy() 编写为对发送给设备的多个请求进行排队，即异步传输。选择方法时，应当考虑设备的能力和限制。

将向 strategy(9E) 例程传递一个指向 buf(9S) 结构的指针。此结构描述传输请求，并包含有关返回的状态信息。buf(9S) 和 strategy(9E) 是块设备操作的焦点。

buf 结构

以下 buf 结构成员对块驱动程序很重要：

```
int           b_flags;       /* Buffer status */
struct buf    *av_forw;      /* Driver work list link */
struct buf    *av_back;      /* Driver work list link */
size_t        b_bcount;      /* # of bytes to transfer */
union {
    caddr_t   b_addr;        /* Buffer's virtual address */
} b_un;
daddr_t       b_blkno;       /* Block number on device */
diskaddr_t    b_lblkno;      /* Expanded block number on device */
size_t        b_resid;       /* # of bytes not transferred after error */
int           b_error;       /* Expanded error field */
void          *b_private;    /* "opaque" driver private area */
dev_t         b_edev;        /* expanded dev field */
```

其中：

av_forw 和 av_back	驱动程序可用以管理其使用的一组缓冲区的指针。有关 av_forw 和 av_back 指针的讨论，请参见第 280 页中的 “异步数据传输（块驱动程序）”。
b_bcount	指定要由设备传输的字节数。
b_un.b_addr	数据缓冲区的内核虚拟地址。仅在进行 bp_mapin(9F) 调用后有效。
b_blkno	设备上用于数据传输的起始 32 位逻辑块编号，以 DEV_BSIZE（512 字节）为单位。驱动程序应使用 b_blkno 或 b_lblkno，但不能同时使用两者。
b_lblkno	设备上用于数据传输的起始 64 位逻辑块编号，以 DEV_BSIZE（512 字节）为单位。驱动程序应使用 b_blkno 或 b_lblkno，但不能同时使用两者。
b_resid	由驱动程序设置的用于表明由于发生错误而未传输的字节数。有关设置 b_resid 的示例，请参见示例 16–7。b_resid 成员会过载。此外，disksort(9F) 也会使用 b_resid。
b_error	当发生传输错误时，由驱动程序设置为错误编号。b_error 应与 b_flags B_ERROR 位一起设置。有关错误值的详细信息，请参见 Intro(9E) 手册页。驱动程序应使用 bioerror(9F)，而不是直接设置 b_error。

b_flags	表示 buf 结构的状态属性和传输属性的标志。如果设置了 B_READ，则 buf 结构指明从设备到内存的传输。否则，此结构指明从内存到设备的传输。如果在数据传输期间驱动程序遇到错误，则该驱动程序应设置 b_flags 成员中的 B_ERROR 字段。此外，该驱动程序还应在 b_error 中提供一个更明确的错误值。驱动程序应使用 bioerror(9F)，而不是设置 B_ERROR。

注意 – 驱动程序绝不能清除 b_flags。

b_private	专供驱动程序存储驱动程序专用数据。
b_edev	包含用于传输的设备的设备编号。

bp_mapin 结构

可以将 buf 结构指针传递到设备驱动程序的 strategy(9E) 例程。但是，b_un.b_addr 引用的数据缓冲区不一定映射到内核地址空间中。因此，驱动程序无法直接访问数据。大多数面向块的设备具有 DMA 功能，因此不需要直接访问数据缓冲区。这些设备改为使用 DMA 映射例程以使设备的 DMA 引擎进行数据传输。有关使用 DMA 的详细信息，请参见第 9 章，直接内存访问 (Direct Memory Access, DMA)。

如果驱动程序需要直接访问数据缓冲区，则该驱动程序必须首先使用 bp_mapin(9F) 将缓冲区映射到内核地址空间。当驱动程序不再需要直接访问数据时，应使用 bp_mapout(9F)。

注意 – 只应对已分配且由设备驱动程序拥有的缓冲区调用 bp_mapout(9F)。不得对通过 strategy(9E) 入口点传递到驱动程序的缓冲区（如文件系统）调用 bp_mapout()。bp_mapin(9F) 不保留引用计数。bp_mapout(9F) 将删除设备驱动程序之上的层所依赖的任何内核映射。

同步数据传输（块驱动程序）

本节介绍一种执行同步 I/O 传输的简单方法。此方法假设硬件是使用 DMA 一次只能传输一个数据缓冲区的简单磁盘设备。另一个假设是磁盘可以通过软件命令启动和停止。设备驱动程序的 strategy(9E) 例程等待当前请求完成，然后再接受新的请求。当传输完成时，设备中断。如果发生错误，设备也中断。

执行块驱动程序的同步数据传输的步骤如下：

1. 检查是否有无效的 buf(9S) 请求。

   检查传递到 strategy(9E) 的 buf(9S) 结构是否有效。所有驱动程序应检查以下条件：

- 请求起始于有效的块。驱动程序将 b_blkno 字段转换为正确的设备偏移，然后确定该偏移对设备而言是否有效。
- 请求不能超出设备上的最后一个块。
- 满足特定于设备的要求。

如果遇到错误，驱动程序应使用 bioerror(9F) 指示相应的错误。然后，驱动程序通过调用 biodone(9F) 来完成请求。biodone() 会通知 strategy(9E) 的调用方传输已完成。在本例中，传输因错误而停止。

2. 检查此设备是否忙。

   同步数据传输允许对设备进行单线程访问。设备驱动程序通过两种方式执行这种访问：

   - 驱动程序保持由互斥锁保护的忙标志。
   - 当设备忙时，驱动程序等待 cv_wait(9F) 的条件变量。

   如果设备忙，线程会一直等待，直到中断处理程序指示设备不再忙。cv_broadcast(9F) 或 cv_signal(9F) 函数可以指示可用的状态。有关条件变量的详细信息，请参见第 3 章，多线程。

   当设备不再忙时，strategy(9E) 例程将设备标记为可用。然后，strategy() 为传输准备缓冲区和设备。

3. 为 DMA 设置缓冲区。

   通过使用 ddi_dma_alloc_handle(9F) 为 DMA 传送准备数据缓冲区，以便分配 DMA 句柄。使用 ddi_dma_buf_bind_handle(9F) 将数据缓冲区绑定到该句柄。有关设置 DMA 资源及相关数据结构的信息，请参见第 9 章，直接内存访问 (Direct Memory Access, DMA)。

4. 开始传输。

   此时，指向 buf(9S) 结构的指针将保存在设备的状态结构中。然后中断例程通过调用 biodone(9F) 来完成传输。

   设备驱动程序随后访问设备寄存器以启动数据传输。在大多数情况下，驱动程序应通过使用互斥锁来保护设备寄存器免受其他线程干扰。在本例中，由于 strategy(9E) 是单线程的，因此没有必要保护设备寄存器。有关数据锁定的详细信息，请参见第 3 章，多线程。

   当执行线程已经启动设备的 DMA 引擎时，驱动程序可以将执行控制权返回到正在调用的例程，如下所示：

```
static int
xxstrategy(struct buf *bp)
{
    struct xxstate *xsp;
    struct device_reg *regp;
    minor_t instance;
    ddi_dma_cookie_t cookie;
    instance = getminor(bp->b_edev);
```

```
        xsp = ddi_get_soft_state(statep, instance);
        if (xsp == NULL) {
            bioerror(bp, ENXIO);
            biodone(bp);
            return (0);
        }
        /* validate the transfer request */
        if ((bp->b_blkno >= xsp->Nblocks) || (bp->b_blkno < 0)) {
            bioerror(bp, EINVAL);
            biodone(bp);
            return (0);
        }
        /*
         * Hold off all threads until the device is not busy.
         */
        mutex_enter(&xsp->mu);
        while (xsp->busy) {
            cv_wait(&xsp->cv, &xsp->mu);
        }
        xsp->busy = 1;
        mutex_exit(&xsp->mu);
        /*
         * If the device has power manageable components,
         * mark the device busy with pm_busy_components(9F),
         * and then ensure that the device
         * is powered up by calling pm_raise_power(9F).
         *
         * Set up DMA resources with ddi_dma_alloc_handle(9F) and
         * ddi_dma_buf_bind_handle(9F).
         */
        xsp->bp = bp;
        regp = xsp->regp;
        ddi_put32(xsp->data_access_handle, &regp->dma_addr,
            cookie.dmac_address);
        ddi_put32(xsp->data_access_handle, &regp->dma_size,
            (uint32_t)cookie.dmac_size);
        ddi_put8(xsp->data_access_handle, &regp->csr,
            ENABLE_INTERRUPTS | START_TRANSFER);
        return (0);
    }
```

5. 处理中断的设备。

   当设备完成数据传输时，设备会生成中断，最终导致驱动程序的中断例程被调用。注册中断时，大多数驱动程序将设备的状态结构指定为中断例程的参数。请参见 ddi_add_intr(9F) 手册页和第 123 页中的 “注册中断”。随后，中断例程可以访问正在被传输的 buf(9S) 结构，以及状态结构提供的任何其他信息。

   中断处理程序会检查设备的状态寄存器，来确定是否在未发生任何错误的情况下完成传输。如果发生错误，处理程序应该使用 bioerror(9F) 指示相应的错误。处理程序还应该清除设备的挂起中断，然后通过调用 biodone(9F) 来完成传输。

   最后一项任务是处理程序清除忙标志。处理程序随后对条件变量调用 cv_signal(9F) 或 cv_broadcast(9F)，发出设备不再忙的信号。此通知会使在 strategy(9E) 中等待设备的其他线程继续进行下一个数据传输。

   以下示例说明了一个同步中断例程。

示例 16–4 块驱动程序的同步中断例程

```
static u_int
xxintr(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;
    uint8_t status;
    mutex_enter(&xsp->mu);
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
    }
    /* Get the buf responsible for this interrupt */
    bp = xsp->bp;
    xsp->bp = NULL;
    /*
     * This example is for a simple device which either
     * succeeds or fails the data transfer, indicated in the
     * command/status register.
     */
    if (status & DEVICE_ERROR) {
        /* failure */
        bp->b_resid = bp->b_bcount;
        bioerror(bp, EIO);
    } else {
        /* success */
        bp->b_resid = 0;
    }
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
        CLEAR_INTERRUPT);
    /* The transfer has finished, successfully or not */
    biodone(bp);
    /*
     * If the device has power manageable components that were
     * marked busy in strategy(9F), mark them idle now with
     * pm_idle_component(9F)
     * Release any resources used in the transfer, such as DMA
     * resources ddi_dma_unbind_handle(9F) and
     * ddi_dma_free_handle(9F).
     *
     * Let the next I/O thread have access to the device.
     */
    xsp->busy = 0;
    cv_signal(&xsp->cv);
    mutex_exit(&xsp->mu);
    return (DDI_INTR_CLAIMED);
}
```

异步数据传输（块驱动程序）

本节介绍一种执行异步 I/O 传输的方法。驱动程序将对 I/O 请求进行排队，然后将控制权返回到调用方。还是假设硬件是一次可以传输一个缓冲区的简单磁盘设备。当数据传输完成时，设备中断。如果发生错误，也会产生中断。执行异步数据传输的基本步骤如下：

1. 检查是否有无效的 buf(9S) 请求。
2. 对请求进行排队。
3. 开始第一个传输。
4. 处理中断的设备。

检查是否有无效的 buf 请求

正如同步传输示例中所示，设备驱动程序会检查传递到 strategy(9E) 的 buf(9S) 结构是否有效。有关更多详细信息，请参见第 276 页中的 “同步数据传输（块驱动程序）”。

对请求进行排队

与同步数据传输不同，驱动程序不等待异步请求完成，而是向队列中添加请求。队列的开头可以是当前传输，也可以是保留活动请求的状态结构中的独立字段，如示例 16–5 中所示。

如果队列开始是空的，那么硬件不忙，并且 strategy(9E) 在返回之前开始传输。否则，如果在队列非空的情况下完成一个传输，则中断例程会开始一个新的传输。为方便起见，示例 16–5 仅在一个单独的例程中决定是否开始新的传输。

驱动程序可以使用 buf(9S) 结构中的 av_forw 和 av_back 成员来管理传输请求列表。可以使用单个指针管理单链接表，也可以同时使用两个指针建立双链接表。设备硬件规格指定哪种类型的列表管理（如插入策略）用于优化设备的性能。传输列表是按设备提供的列表，因此列表的头和尾都存储在状态结构中。

以下示例提供了对驱动程序共享数据（如传输列表）有访问权限的多个线程。您必须标识共享数据，并且必须用互斥锁保护这些数据。有关互斥锁的更多详细信息，请参见第 3 章，多线程。

示例 16–5 对块驱动程序的数据传输请求进行排队

```
static int
xxstrategy(struct buf *bp)
{
    struct xxstate *xsp;
    minor_t instance;
    instance = getminor(bp->b_edev);
    xsp = ddi_get_soft_state(statep, instance);
```

示例 16–5 对块驱动程序的数据传输请求进行排队 （续）

```
    /* ... */
    /* validate transfer request */
    /* ... */
    /*
     * Add the request to the end of the queue. Depending on the device, a sorting
     * algorithm, such as disksort(9F) can be used if it improves the
     * performance of the device.
     */
    mutex_enter(&xsp->mu);
    bp->av_forw = NULL;
    if (xsp->list_head) {
        /* Non-empty transfer list */
        xsp->list_tail->av_forw = bp;
        xsp->list_tail = bp;
    } else {
        /* Empty Transfer list */
        xsp->list_head = bp;
        xsp->list_tail = bp;
    }
    mutex_exit(&xsp->mu);
    /* Start the transfer if possible */
    (void) xxstart((caddr_t)xsp);
    return (0);
}
```

开始第一个传输

可实现排队的设备驱动程序通常具有 `start()` 例程。`start()` 可将下一个请求从队列中删除，并开始出入设备的数据传输。在以下示例中，不管设备状态是忙还是空闲，`start()` 将处理所有请求。

注 – 必须写入 `start()` 以便从任何上下文都可以对其进行调用。内核上下文中的策略例程与中断上下文中的中断例程都可以调用 `start()`。

每次 `strategy()` 对请求排队时，将由 strategy(9E) 调用 `start()`，以便可以启动空闲设备。如果设备忙，则 `start()` 立即返回。

在处理程序从声明的中断返回之前，也可以由中断处理程序调用 `start()`，以便可以为非空队列提供服务。如果队列是空的，则 `start()` 立即返回。

由于 `start()` 是一个专用驱动程序例程，因此 `start()` 可以采用任何参数并返回任何类型。将写入以下代码样例用作 DMA 回调，尽管没有显示那一部分。相应地，此示例必须采用 `caddr_t` 作为参数并返回 `int`。有关 DMA 回调例程的更多信息，请参见第 157 页中的 “处理资源分配故障”。

示例 16–6 开始块驱动程序的第一个数据请求

```
static int
xxstart(caddr_t arg)
{
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;

    mutex_enter(&xsp->mu);
    /*
     * If there is nothing more to do, or the device is
     * busy, return.
     */
    if (xsp->list_head == NULL || xsp->busy) {
       mutex_exit(&xsp->mu);
       return (0);
    }
    xsp->busy = 1;
    /* Get the first buffer off the transfer list */
    bp = xsp->list_head;
    /* Update the head and tail pointer */
    xsp->list_head = xsp->list_head->av_forw;
    if (xsp->list_head == NULL)
       xsp->list_tail = NULL;
    bp->av_forw = NULL;
    mutex_exit(&xsp->mu);
    /*
     * If the device has power manageable components,
     * mark the device busy with pm_busy_components(9F),
     * and then ensure that the device
     * is powered up by calling pm_raise_power(9F).
     *
     * Set up DMA resources with ddi_dma_alloc_handle(9F) and
     * ddi_dma_buf_bind_handle(9F).
     */
    xsp->bp = bp;
    ddi_put32(xsp->data_access_handle, &xsp->regp->dma_addr,
        cookie.dmac_address);
    ddi_put32(xsp->data_access_handle, &xsp->regp->dma_size,
        (uint32_t)cookie.dmac_size);
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
        ENABLE_INTERRUPTS | START_TRANSFER);
    return (0);
}
```

处理中断的设备

中断例程与异步版本类似，只是增加了对 `start()` 的调用并删除了对 `cv_signal(9F)` 的调用。

示例 16–7 异步中断的块驱动程序例程

```
static u_int
xxintr(caddr_t arg)
{
```

示例 16–7 异步中断的块驱动程序例程 （续）

```
    struct xxstate *xsp = (struct xxstate *)arg;
    struct buf *bp;
    uint8_t status;
    mutex_enter(&xsp->mu);
    status = ddi_get8(xsp->data_access_handle, &xsp->regp->csr);
    if (!(status & INTERRUPTING)) {
        mutex_exit(&xsp->mu);
        return (DDI_INTR_UNCLAIMED);
    }
    /* Get the buf responsible for this interrupt */
    bp = xsp->bp;
    xsp->bp = NULL;
    /*
     * This example is for a simple device which either
     * succeeds or fails the data transfer, indicated in the
     * command/status register.
     */
    if (status & DEVICE_ERROR) {
        /* failure */
        bp->b_resid = bp->b_bcount;
        bioerror(bp, EIO);
    } else {
        /* success */
        bp->b_resid = 0;
    }
    ddi_put8(xsp->data_access_handle, &xsp->regp->csr,
        CLEAR_INTERRUPT);
    /* The transfer has finished, successfully or not */
    biodone(bp);
    /*
     * If the device has power manageable components that were
     * marked busy in strategy(9F), mark them idle now with
     * pm_idle_component(9F)
     * Release any resources used in the transfer, such as DMA
     * resources (ddi_dma_unbind_handle(9F) and
     * ddi_dma_free_handle(9F)).
     *
     * Let the next I/O thread have access to the device.
     */
    xsp->busy = 0;
    mutex_exit(&xsp->mu);
    (void) xxstart((caddr_t)xsp);
    return (DDI_INTR_CLAIMED);
}
```

dump() 和 print() 入口点

本节讨论 dump(9E) 和 print(9E) 入口点。

dump() 入口点（块驱动程序）

dump(9E) 入口点用于在系统发生故障时将虚拟地址空间的一部分直接复制到指定的设备。在检查点操作期间，还可以使用 dump() 将内核状态复制到磁盘。有关更多信息，请参见 cpr(7) 和 dump(9E) 手册页。由于在检查点操作期间中断被禁用，因此该入口点必须能够在不使用中断的情况下执行此操作。

```
int dump(dev_t dev, caddr_t addr, daddr_t blkno, int nblk)
```

其中：

dev 接收转储的设备的设备编号。

addr 开始转储的基本内核虚拟地址。

blkno 开始转储的块。

nblk 转储的块的编号。

转储依赖于现有的驱动程序是否工作正常。

print() 入口点（块驱动程序）

```
int print(dev_t dev, char *str)
```

系统调用 print(9E) 入口点以显示有关已检测到的异常的消息。print(9E) 应调用 cmn_err(9F) 以代表系统将消息发布到控制台。以下示例说明了一个典型的 print() 入口点。

```
static int
 xxprint(dev_t dev, char *str)
 {
     cmn_err(CE_CONT, “xx: %s\n”, str);
     return (0);
 }
```

磁盘设备驱动程序

磁盘设备是一类重要的块设备驱动程序。

磁盘 ioctl

Oracle Solaris 磁盘驱动程序至少需要支持一组特定于 Oracle Solaris 磁盘驱动程序的 ioctl 命令。在 dkio(7I) 手册页中指定了这些 I/O 控制。磁盘 I/O 控制用于将磁盘信息

传入/传出设备驱动程序。磁盘实用程序命令（如 format(1M) 和 newfs(1M)）支持 Oracle Solaris 磁盘设备。强制性的 Sun 磁盘 I/O 控制如下：

DKIOCINFO	返回描述磁盘控制器的信息
DKIOCGAPART	返回磁盘的分区映射
DKIOCSAPART	设置磁盘的分区映射
DKIOCGGEOM	返回磁盘的几何参数
DKIOCSGEOM	设置磁盘的几何参数
DKIOCGVTOC	返回磁盘的卷目录
DKIOCSVTOC	设置磁盘的卷目录

磁盘性能

Oracle Solaris DDI/DKI 提供了优化 I/O 传输以提高文件系统性能的工具。它是一种管理 I/O 请求列表以便优化文件系统磁盘访问的机制。有关对 I/O 请求进行排队的说明，请参见第 280 页中的 “异步数据传输（块驱动程序）”。

diskhd 结构用于管理 I/O 请求链接表。

```
struct diskhd {
    long     b_flags;         /* not used, needed for consistency*/
    struct   buf *b_forw,     *b_back;     /* queue of unit queues */
    struct   buf *av_forw,    *av_back;    /* queue of bufs for this unit */
    long     b_bcount;        /* active flag */
};
```

diskhd 数据结构具有驱动程序可处理的两个 buf 指针。av_forw 指针指向第一个活动 I/O 请求。第二个指针 av_back 指向列表中的上一个活动请求。

一个指向此结构的指针以及一个指向要处理的当前 buf 结构的指针作为参数传递给 disksort(9F)。disksort() 例程对 buf 请求排序以优化磁盘查找。然后此例程将 buf 指针插入 diskhd 列表。disksort() 程序将 buf 结构的 b_resid 中的值用作排序关键字。驱动程序负责设置此值。大多数 Sun 磁盘驱动程序使用柱面组作为排序关键字。此方法优化了文件系统读前访问。

将数据添加到 diskhd 列表后，设备需要传输这些数据。如果设备未忙于处理请求，则 xxstart() 例程会将第一个 buf 结构拉出 diskhd 列表并开始传输。

如果设备正忙，则驱动程序会从 xxstrategy() 入口点返回。当硬件执行完数据传输时，便会产生中断。随后会调用驱动程序的中断例程为设备提供服务。在提供中断服务后，驱动程序可以调用 start() 例程来处理 diskhd 列表中的下一个 buf 结构。

◆ ◆ ◆ 第 17 章

SCSI 目标驱动程序

Oracle Solaris DDI/DKI 将 SCSI 设备的软件接口分成以下两个主要部分：**目标**驱动程序和**主机总线适配器** *(host bus adapter, HBA)* 驱动程序。**目标**驱动程序指 SCSI 总线上的设备（如磁盘或磁带机）的驱动程序。**主机总线适配器**驱动程序指主机上的 SCSI 控制器的驱动程序。SCSA 定义了这两个组件之间的接口。本章仅讨论目标驱动程序。有关主机总线适配器驱动程序的信息，请参见第 18 章，SCSI 主机总线适配器驱动程序。

注 – 术语“主机总线适配器”和 "HBA" 等效于 SCSI 规范中定义的“主机适配器”。

本章介绍有关以下主题的信息：

目标驱动程序介绍

目标驱动程序可以为字符设备驱动程序，也可以为块设备驱动程序，具体取决于设备。磁带机的驱动程序通常为字符设备驱动程序，而磁盘则由块设备驱动程序处理。本章介绍如何编写 SCSI 目标驱动程序。本章还讨论了 SCSA 对 SCSI 目标设备的块驱动程序和字符驱动程序的其他要求。

以下参考文档提供了目标驱动程序和主机总线适配器驱动程序的设计者需要的补充信息。

《Small Computer System Interface 2 (SCSI-2)》，ANSI/NCITS X3.131-1994，Global Engineering Documents，1998 年。ISBN 1199002488。

《*The Basics of SCSI*》（第四版），ANCOT Corporation 出版，1998 年，ISBN 0963743988。

另请参阅硬件供应商所提供的关于目标设备的 SCSI 命令规范。

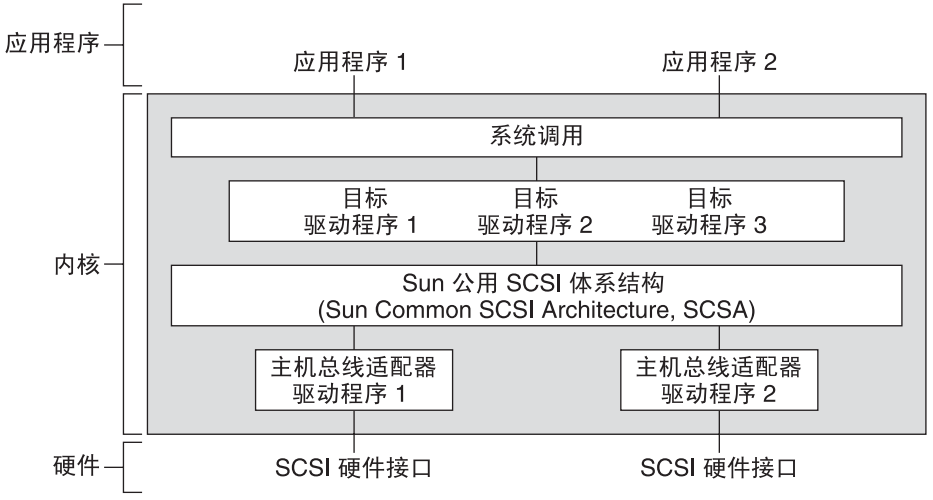
Sun 公用 SCSI 体系结构概述

Sun 公用 SCSI 体系结构 (Sun Common SCSI Architecture, SCSA) 是 Solaris DDI/DKI 编程接口，用于将 SCSI 命令从目标驱动程序传输到主机总线适配器驱动程序。该接口与主机总线适配器硬件的类型、平台、处理器体系结构以及通过该接口传输的 SCSI 命令无关。

只要符合 SCSA，目标驱动程序便可以将 SCSI 命令传送给目标设备，而无需了解主机总线适配器的硬件实现。

SCSA 在概念上将生成 SCSI 命令与通过 SCSI 总线传输命令和数据这两个过程分离开来。该体系结构定义了高级别软件组件与低级别软件组件之间的软件接口。较高级别的软件组件由一个或多个 SCSI 目标驱动程序组成，这些驱动程序可将 I/O 请求转换为适用于外围设备的 SCSI 命令。以下示例说明了 SCSI 体系结构。

图 17–1 SCSA 块图



较低级别的软件组件由 SCSA 接口层和一个或多个主机总线适配器驱动程序组成。目标驱动程序负责生成执行所需功能需要的正确 SCSI 命令，并负责处理结果。

常规控制流程

以下步骤介绍读取或写入请求的常规控制流程（假定没有发生传输错误）。

1. 调用目标驱动程序的 read(9E) 或 write(9E) 入口点。使用 physio(9F) 锁定内存，准备 buf 结构并调用策略例程。
2. 目标驱动程序的 strategy(9E) 例程将检查请求。然后，strategy() 通过使用 scsi_init_pkt(9F) 来分配 scsi_pkt(9S)。目标驱动程序初始化包，并使用 scsi_setup_cdb(9F) 函数设置 SCSI 命令描述符块 (command descriptor block, CDB)。目标驱动程序还指定超时。然后，该驱动程序提供一个指向回调函数的指针。完成命令后，主机总线适配器驱动程序将调用该回调函数。buf(9S) 指针应保存在 SCSI 包的目标专用空间中。
3. 目标驱动程序使用 scsi_transport(9F) 将包提交给主机总线适配器驱动程序。然后，目标驱动程序可以自由接受其他请求。目标驱动程序不应在包的传输过程中对其进行访问。如果主机总线适配器驱动程序或目标支持排队，则可以在传输包时提交新请求。
4. 一旦 SCSI 总线空闲而且目标不繁忙，主机总线适配器驱动程序便会选择该目标并传递 CDB。目标驱动程序将执行命令。然后，目标执行所请求的数据传输。
5. 目标发送完成状态并且命令完成后，主机总线适配器驱动程序会通知目标驱动程序。要执行通知，主机应调用 SCSI 包中指定的完成函数。此时，主机总线适配器驱动程序不再负责处理该包，同时目标驱动程序重新获得了对该包的拥有权。
6. SCSI 包的完成例程将分析返回的信息。然后，完成例程会确定 SCSI 操作是否成功。如果出现故障，目标驱动程序将再次调用 scsi_transport(9F) 以重试命令。如果主机总线适配器驱动程序不支持自动请求检测，则目标驱动程序必须在出现检查条件时，提交请求检测包才能检索检测数据。
7. 成功完成上述操作后或者如果无法重试命令，目标驱动程序将调用 scsi_destroy_pkt(9F)。scsi_destroy_pkt () 将同步数据。然后，scsi_destroy_pkt() 释放包。如果在释放包之前目标驱动程序需要访问数据，则调用 scsi_sync_pkt(9F)。
8. 最后，目标驱动程序将通知请求应用程序读取或写入事务已完成。该通知是在执行了字符设备的驱动程序中的 read(9E) 入口点并返回后进行的。否则，会通过 biodone(9F) 间接发出通知。

SCSA 允许以重叠方式和排队方式在进程的各个点执行多个此类操作。在该模型中，系统资源由主机总线适配器驱动程序负责管理。借助软件接口并使用不同复杂程度的 SCSI 总线适配器，可以在主机总线适配器驱动程序中执行目标驱动程序函数。

SCSA 函数

SCSA 定义了多种函数，用于管理资源的分配和释放、控制状态的检测和设置以及 SCSI 命令的传输。下表中列出了这些函数。

表 17–1 标准 SCSA 函数

函数名	类别
scsi_abort(9F)	错误处理
scsi_alloc_consistent_buf(9F)	
scsi_destroy_pkt(9F)	
scsi_dmafree(9F)	
scsi_free_consistent_buf(9F)	
scsi_ifgetcap(9F)	传输信息和控制
scsi_ifsetcap(9F)	
scsi_init_pkt(9F)	资源管理
scsi_poll(9F)	轮询 I/O
scsi_probe(9F)	探测器函数
scsi_reset(9F)	
scsi_setup_cdb(9F)	CDB 初始化函数
scsi_sync_pkt(9F)	
scsi_transport(9F)	命令传输
scsi_unprobe(9F)	

注 – 如果驱动程序需要使用 SCSI-1 设备，请使用 makecom(9F)。

硬件配置文件

由于 SCSI 设备不是自标识设备，因此目标驱动程序需要硬件配置文件。有关详细信息，请参见 driver.conf(4) 和 scsi_free_consistent_buf(9F) 手册页。以下是典型的配置文件：

```
name="xx" class="scsi" target=2 lun=0;
```

系统将在自动配置期间读取该文件。系统使用 *class* 属性标识驱动程序可能存在的父驱动程序。然后，系统尝试将该驱动程序连接至类为 *scsi* 的任何父驱动程序。所有主机总线适配器驱动程序都属于此类。首选使用 *class* 属性，而不是 *parent* 属性。采用此方法，任何在指定 *target* 和 *lun* ID 中查找预期设备的主机总线适配器驱动程序都可以连接至目标。目标驱动程序负责验证其 probe(9E) 例程中的类。

声明和数据结构

目标驱动程序必须包括头文件 <sys/scsi/scsi.h>。

SCSI 目标驱动程序必须使用以下命令生成二进制模块：

```
ld -r xx xx.o -N"misc/scsi"
```

scsi_device 结构

调用 probe(9E) 或 attach(9E) 例程之前，主机总线适配器驱动程序将为目标驱动程序分配并初始化 scsi_device(9S) 结构。此结构可存储有关每个 SCSI 逻辑单元的信息，包括指向信息区（包含通用信息和特定于设备的信息）的指针。对于连接到系统的每个逻辑单元，都存在一个 scsi_device(9S) 结构。目标驱动程序可以通过调用 ddi_get_driver_private(9F) 来检索指向此结构的指针。

注意 – 由于主机总线适配器驱动程序使用目标设备的 dev_info 结构中的专用字段，因此目标驱动程序不能使用 ddi_set_driver_private(9F)。

scsi_device(9S) 结构包含以下字段：

```
struct scsi_device {
    struct scsi_address           sd_address;    /* opaque address */
    dev_info_t                    *sd_dev;       /* device node */
    kmutex_t                      sd_mutex;
    void                          *sd_reserved;
    struct scsi_inquiry           *sd_inq;
    struct scsi_extended_sense    *sd_sense;
    caddr_t                       sd_private;
};
```

其中：

sd_address　为了进行 SCSI 资源分配而传递给例程的数据结构。

sd_dev　指向目标的 dev_info 结构的指针。

sd_mutex　供目标驱动程序使用的互斥锁。此互斥锁由主机总线适配器驱动程序初始化，并可被目标驱动程序用作每设备互斥锁。请勿在调用 scsi_transport(9F) 或 scsi_poll(9F) 期间持有此互斥锁。有关互斥锁的更多信息，请参见第 3 章，多线程。

sd_inq　目标设备的 SCSI 查询数据的指针。scsi_probe(9F) 例程将分配缓冲区，使用查询数据填充该缓冲区，并将该缓冲区连接到此字段。

sd_sense　指向用于包含设备中的 SCSI 请求检测数据的缓冲区的指针。目标驱动程序必须分配和管理此缓冲区。请参见第 295 页中的 “attach() 入口点（SCSI 目标驱动程序）”。

sd_private　供目标驱动程序使用的指针字段。此字段通常用于存储指向专用目标驱动程序状态结构的指针。

scsi_pkt 结构（目标驱动程序）

scsi_pkt 结构包含以下字段：

```
struct scsi_pkt {
    opaque_t  pkt_ha_private;         /* private data for host adapter */
    struct scsi_address pkt_address;  /* destination packet is for */
    opaque_t  pkt_private;            /* private data for target driver */
    void     (*pkt_comp)(struct scsi_pkt *);  /* completion routine */
    uint_t   pkt_flags;               /* flags */
    int      pkt_time;                /* time allotted to complete command */
    uchar_t  *pkt_scbp;               /* pointer to status block */
    uchar_t  *pkt_cdbp;               /* pointer to command block */
    ssize_t  pkt_resid;               /* data bytes not transferred */
    uint_t   pkt_state;               /* state of command */
    uint_t   pkt_statistics;          /* statistics */
    uchar_t  pkt_reason;              /* reason completion called */
};
```

其中：

pkt_address　scsi_init_pkt(9F) 设置的目标设备的地址。

pkt_private　用于存储目标驱动程序的专用数据的位置。*pkt_private* 通常用于保存命令的 buf(9S) 指针。

pkt_comp　完成例程的地址。主机总线适配器驱动程序将在传输命令后调用此例程。传输命令并不表示命令已成功。目标可能处于繁忙状态。另一种可能性是在经过超时时间段之前目标未响应。请参见 pkt_time 字段的说明。目标驱动程序必须在该字段中提供有效值。如果不需要通知驱动程序，则该值可以为 NULL。

注 – 有两种不同的 SCSI 回调例程。pkt_comp 字段标识**完成回调**例程，该例程将在主机总线适配器完成其处理时调用。此外，还提供了**资源回调**例程，该例程将在当前尚不可用资源可能可用时调用。请参见 scsi_init_pkt(9F) 手册页。

---

pkt_flags　提供其他控制信息，例如，在没有断开连接权限的情况下传输命令 (FLAG_NODISCON)，或禁用回调 (FLAG_NOINTR)。有关详细信息，请参见 scsi_pkt(9S) 手册页。

pkt_time　超时值（以秒为单位）。如果命令在该时间内未完成，则主机总线适配器将调用完成例程，并将 pkt_reason 设置为 CMD_TIMEOUT。目

	标驱动程序应将该字段设置为大于命令可能需要的最长时间。如果超时值为零，则不请求超时。超时从在 SCSI 总线上传输命令时开始。
pkt_scbp	指向 SCSI 状态完成块的指针。该字段由主机总线适配器驱动程序填充。
pkt_cdbp	指向 SCSI 命令描述符块（要发送到目标设备的实际命令）的指针。主机总线适配器驱动程序不会解释该字段。目标驱动程序必须使用目标设备可以处理的命令填充该字段。
pkt_resid	剩余操作。pkt_resid 字段有两种不同的用途，具体取决于如何使用 pkt_resid。使用 pkt_resid 为命令 scsi_init_pkt(9F) 分配 DMA 资源时，pkt_resid 指示不可分配的字节数。由于 DMA 硬件具有分散/集中限制或其他设备限制，因此可能**无法**分配 DMA 资源。传输命令后，pkt_resid 指示不可传输的数据字节数。该字段由主机总线适配器驱动程序在调用完成例程之前填充。
pkt_state	指示命令的状态。主机总线适配器驱动程序在命令执行过程中填充该字段。该字段的每一位分别根据以下五种命令状态设置： ■ STATE_GOT_BUS－已获取总线 ■ STATE_GOT_TARGET－已选定目标 ■ STATE_SENT_CMD－已发送命令 ■ STATE_XFERRED_DATA－已传输数据（如果适用） ■ STATE_GOT_STATUS－已从设备接收状态
pkt_statistics	包含与传输相关的统计信息（由主机总线适配器驱动程序设置）。
pkt_reason	提供调用完成例程的原因。完成例程会对该字段进行解码。然后，执行相应的操作。如果命令完成（即未发生传输错误），则该字段将设置为 CMD_CMPLT。如果该字段中存在其他值，则指示发生了错误。完成命令后，目标驱动程序应检查 pkt_scbp 字段以查看条件状态。有关更多信息，请参见 scsi_pkt(9S) 手册页。

SCSI 目标驱动程序的自动配置

SCSI 目标驱动程序必须实现标准自动配置例程 _init(9E)、_fini(9E) 和 _info(9E)。有关更多信息，请参见第 95 页中的 “可装入驱动程序接口”。

此外，还需要以下例程，但这些例程必须执行特定的 SCSI 和 SCSA 处理：

- probe(9E)
- attach(9E)
- detach(9E)
- getinfo(9E)

probe() 入口点（SCSI 目标驱动程序）

SCSI 目标设备不是自标识设备，因此目标驱动程序必须具有 probe(9E) 例程。该例程必须确定所需类型的设备是否存在以及是否正在响应。

probe(9E) 例程的常规结构和返回代码与其他设备驱动程序的结构和返回代码相同。SCSI 目标驱动程序必须在其 probe(9E) 入口点中使用 scsi_probe(9F) 例程。scsi_probe(9F) 向设备发送 SCSI 查询命令并返回指示结果的代码。如果 SCSI 查询命令成功，则 scsi_probe(9F) 将分配 scsi_inquiry(9S) 结构并使用设备的查询数据填充该结构。从 scsi_probe(9F) 返回之后，scsi_device(9S) 结构的 sd_inq 字段将指向此 scsi_inquiry(9S) 结构。

由于 probe(9E) 必须是无状态的，因此目标驱动程序必须在 probe(9E) 返回之前调用 scsi_unprobe(9F)，即使 scsi_probe(9F) 失败也是如此。

示例 17–1 显示了典型的 probe(9E) 例程。该示例中的例程从其 dev_info 结构的专用字段中检索 scsi_device(9S) 结构。该例程还检索设备的 SCSI 目标和逻辑单元号，以便列显在消息中。然后，probe(9E) 例程将调用 scsi_probe(9F) 以验证预期设备（在本例中为打印机）是否存在。

如果成功，scsi_probe(9F) 会将 scsi_inquiry(9S) 结构中设备的 SCSI 查询数据连接到 scsi_device(9S) 结构的 sd_inq 字段。然后，驱动程序便可以确定设备类型是否为打印机，相关情况将在 inq_dtype 字段中报告。如果设备是打印机，则使用 scsi_log(9F) 报告类型，并使用 scsi_dname(9F) 将设备类型转换为字符串。

示例 17–1 SCSI 目标驱动程序 probe(9E) 例程

```
static int
xxprobe(dev_info_t *dip)
{
    struct scsi_device *sdp;
    int rval, target, lun;
    /*
     * Get a pointer to the scsi_device(9S) structure
     */
    sdp = (struct scsi_device *)ddi_get_driver_private(dip);

    target = sdp->sd_address.a_target;
    lun = sdp->sd_address.a_lun;
    /*
     * Call scsi_probe(9F) to send the Inquiry command. It will
     * fill in the sd_inq field of the scsi_device structure.
     */
    switch (scsi_probe(sdp, NULL_FUNC)) {
    case SCSIPROBE_FAILURE:
    case SCSIPROBE_NORESP:
    case SCSIPROBE_NOMEM:
        /*
         * In these cases, device might be powered off,
         * in which case we might be able to successfully
         * probe it at some future time - referred to
```

示例 17–1 SCSI 目标驱动程序 probe(9E) 例程 （续）

```
         * as 'deferred attach'.
         */
        rval = DDI_PROBE_PARTIAL;
        break;
    case SCSIPROBE_NONCCS:
    default:
        /*
         * Device isn't of the type we can deal with,
         * and/or it will never be usable.
         */
        rval = DDI_PROBE_FAILURE;
        break;
    case SCSIPROBE_EXISTS:
        /*
         * There is a device at the target/lun address. Check
         * inq_dtype to make sure that it is the right device
         * type. See scsi_inquiry(9S)for possible device types.
         */
        switch (sdp->sd_inq->inq_dtype) {
        case DTYPE_PRINTER:
        scsi_log(sdp, "xx", SCSI_DEBUG,
            "found %s device at target%d, lun%d\n",
            scsi_dname((int)sdp->sd_inq->inq_dtype),
            target, lun);
        rval = DDI_PROBE_SUCCESS;
        break;
        case DTYPE_NOTPRESENT:
        default:
        rval = DDI_PROBE_FAILURE;
        break;
        }
    }
    scsi_unprobe(sdp);
    return (rval);
}
```

更全面的 probe(9E) 例程可以检查 scsi_inquiry(9S) 以确保设备是特定驱动程序期望的类型。

attach() 入口点（SCSI 目标驱动程序）

在 probe(9E) 例程验证预期设备是否存在后，将调用 attach(9E)。attach() 执行以下任务：

- 分配并初始化任何每实例数据。
- 创建从设备节点信息。
- 暂停设备或系统后恢复设备的硬件状态。有关详细信息，请参见第 101 页中的 “attach() 入口点”。

SCSI 目标驱动程序需要再次调用 scsi_probe(9F)，以检索设备的查询数据。该驱动程序还必须创建 SCSI 请求检测包。如果连接成功，则 attach() 函数不应调用 scsi_unprobe(9F)。

以下三个例程可用于创建请求检测包：scsi_alloc_consistent_buf(9F)、scsi_init_pkt(9F) 和 scsi_setup_cdb(9F)。scsi_alloc_consistent_buf(9F) 分配适用于一致 DMA 的缓冲区。然后，scsi_alloc_consistent_buf() 返回指向 buf(9S) 结构的指针。一致缓冲区的优点在于无需显式同步数据。换句话说，目标驱动程序可以在回调之后访问数据。必须使用检测缓冲区的地址初始化设备的 scsi_device(9S) 结构的 sd_sense 元素。scsi_init_pkt(9F) 创建并部分初始化 scsi_pkt(9S) 结构。scsi_setup_cdb(9F) 创建 SCSI 命令描述符块，此时是通过创建 SCSI 请求检测命令来实现。

请注意，SCSI 设备不是自标识设备，并且没有 reg 属性。因此，驱动程序必须设置 pm-hardware-state 属性。设置 pm-hardware-state 将会通知框架需要暂停该设备然后将其恢复。

以下示例显示了 SCSI 目标驱动程序的 attach() 例程。

示例 17–2 SCSI 目标驱动程序 attach(9E) 例程

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate         *xsp;
    struct scsi_pkt        *rqpkt = NULL;
    struct scsi_device     *sdp;
    struct buf         *bp = NULL;
    int            instance;
    instance = ddi_get_instance(dip);
    switch (cmd) {
      case DDI_ATTACH:
          break;
      case DDI_RESUME:
          /* For information, see the "Directory Memory Access (DMA)" */
          /* chapter in this book. */
      default:
          return (DDI_FAILURE);
    }
    /*
     * Allocate a state structure and initialize it.
     */
    xsp = ddi_get_soft_state(statep, instance);
    sdp = (struct scsi_device *)ddi_get_driver_private(dip);
    /*
     * Cross-link the state and scsi_device(9S) structures.
     */
    sdp->sd_private = (caddr_t)xsp;
    xsp->sdp = sdp;
    /*
     * Call scsi_probe(9F) again to get and validate inquiry data.
     * Allocate a request sense buffer. The buf(9S) structure
     * is set to NULL to tell the routine to allocate a new one.
```

示例 17–2 SCSI 目标驱动程序 attach(9E) 例程 （续）

```
     * The callback function is set to NULL_FUNC to tell the
     * routine to return failure immediately if no
     * resources are available.
     */
    bp = scsi_alloc_consistent_buf(&sdp->sd_address, NULL,
    SENSE_LENGTH, B_READ, NULL_FUNC, NULL);
    if (bp == NULL)
        goto failed;
    /*
     * Create a Request Sense scsi_pkt(9S) structure.
     */
    rqpkt = scsi_init_pkt(&sdp->sd_address, NULL, bp,
    CDB_GROUP0, 1, 0, PKT_CONSISTENT, NULL_FUNC, NULL);
    if (rqpkt == NULL)
        goto failed;
    /*
     * scsi_alloc_consistent_buf(9F) returned a buf(9S) structure.
     * The actual buffer address is in b_un.b_addr.
     */
    sdp->sd_sense = (struct scsi_extended_sense *)bp->b_un.b_addr;
    /*
     * Create a Group0 CDB for the Request Sense command
     */
    if (scsi_setup_cdb((union scsi_cdb *)rqpkt->pkt_cdbp,
        SCMD_REQUEST_SENSE, 0, SENSE__LENGTH, 0) == 0)
        goto failed;;
    /*
     * Fill in the rest of the scsi_pkt structure.
     * xxcallback() is the private command completion routine.
     */
    rqpkt->pkt_comp = xxcallback;
    rqpkt->pkt_time = 30; /* 30 second command timeout */
    rqpkt->pkt_flags |= FLAG_SENSING;
    xsp->rqs = rqpkt;
    xsp->rqsbuf = bp;
    /*
     * Create minor nodes, report device, and do any other initialization. */
     * Since the device does not have the 'reg' property,
     * cpr will not call its DDI_SUSPEND/DDI_RESUME entries.
     * The following code is to tell cpr that this device
     * needs to be suspended and resumed.
     */
    (void) ddi_prop_update_string(device, dip,
        "pm-hardware-state", "needs-suspend-resume");
    xsp->open = 0;
    return (DDI_SUCCESS);
failed:
    if (bp)
        scsi_free_consistent_buf(bp);
    if (rqpkt)
        scsi_destroy_pkt(rqpkt);
    sdp->sd_private = (caddr_t)NULL;
    sdp->sd_sense = NULL;
    scsi_unprobe(sdp);
    /* Free any other resources, such as the state structure. */
```

示例 17–2 SCSI 目标驱动程序 attach(9E) 例程 （续）

```
    return (DDI_FAILURE);
}
```

detach() 入口点（SCSI 目标驱动程序）

detach(9E) 入口点与 attach(9E) 是反向的入口点。detach() 必须释放在 attach() 中分配的所有资源。如果成功，则 detach 应调用 scsi_unprobe(9F)。以下示例显示了目标驱动程序的 detach() 例程。

示例 17–3 SCSI 目标驱动程序 detach(9E) 例程

```
static int
xxdetach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    struct xxstate *xsp;
    switch (cmd) {
      case DDI_DETACH:
          /*
           * Normal detach(9E) operations, such as getting a
           * pointer to the state structure
           */
          scsi_free_consistent_buf(xsp->rqsbuf);
          scsi_destroy_pkt(xsp->rqs);
          xsp->sdp->sd_private = (caddr_t)NULL;
          xsp->sdp->sd_sense = NULL;
          scsi_unprobe(xsp->sdp);
          /*
           * Remove minor nodes.
           * Free resources, such as the state structure and properties.
           */
          return (DDI_SUCCESS);
      case DDI_SUSPEND:
          /* For information, see the "Directory Memory Access (DMA)" */
          /* chapter in this book. */
      default:
          return (DDI_FAILURE);
    }
}
```

getinfo() 入口点（SCSI 目标驱动程序）

SCSI 目标驱动程序的 getinfo(9E) 例程与其他驱动程序的相应例程基本相同（有关 DDI_INFO_DEVT2INSTANCE 案例的更多信息，请参见 第 107 页中的“getinfo() 入口点”）。但是，如果是 getinfo() 例程的 DDI_INFO_DEVT2DEVINFO，则目标驱动程序必须返回指向其 dev_info 节点的指针。该指针可以保存在驱动程序状态结构中，也可以从 scsi_device(9S) 结构的 sd_dev 字段中检索。以下示例显示了替换 SCSI 目标驱动程序 getinfo() 代码段。

示例 17–4 替代 SCSI 目标驱动程序 getinfo() 代码段

```
case DDI_INFO_DEVT2DEVINFO:
    dev = (dev_t)arg;
    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);
    if (xsp == NULL)
        return (DDI_FAILURE);
    *result = (void *)xsp->sdp->sd_dev;
    return (DDI_SUCCESS);
```

资源分配

要向设备发送 SCSI 命令，目标驱动程序必须创建并初始化 scsi_pkt(9S) 结构。然后，必须将该结构传递到主机总线适配器驱动程序。

scsi_init_pkt() 函数

scsi_init_pkt(9F) 例程分配 scsi_pkt(9S) 结构并将该结构调整归零。scsi_init_pkt() 还设置指向 pkt_private、*pkt_scbp 和 *pkt_cdbp 的指针。此外，scsi_init_pkt() 还提供回调机制来处理资源不可用的情况。该函数的语法如下：

```
struct scsi_pkt *scsi_init_pkt(struct scsi_address *ap,
     struct scsi_pkt *pktp, struct buf *bp, int cmdlen,
     int statuslen, int privatelen, int flags,
     int (*callback)(caddr_t), caddr_t arg)
```

其中：

ap 指向 scsi_address 结构的指针。ap 是设备的 scsi_device(9S) 结构的 sd_address 字段。

pktp 指向要初始化的 scsi_pkt(9S) 结构的指针。如果将该指针设置为 NULL，则会分配一个新包。

bp 指向 buf(9S) 结构的指针。如果该指针为具有有效字节计数的非 null 值，则会分配 DMA 资源。

cmdlen SCSI 命令描述符块的长度（以字节为单位）。

statuslen SCSI 状态完成块的必需长度（以字节为单位）。

privatelen 要为 pkt_private 字段分配的字节数。

flags 标志集：

- PKT_CONSISTENT－如果 DMA 缓冲区是使用 scsi_alloc_consistent_buf(9F) 分配的，则必须设置该位。在这种情况下，主机总线适配器驱动程序将保证在执行目标驱动程序的命令完成回调之前正确同步数据传输。
- PKT_DMA_PARTIAL－如果驱动程序接受部分 DMA 映射，则可以设置该位。如果设置了该位，scsi_init_pkt(9F) 将分配 DMA 资源并设置 DDI_DMA_PARTIAL 标志。可以返回 scsi_pkt(9S) 结构的 pkt_resid 字段的返回值可以是非零的剩余值。非零值表示 scsi_init_pkt(9F) 无法分配的 DMA 资源字节数。

callback 指定资源不可用时要执行的操作。如果设置为 NULL_FUNC，scsi_init_pkt(9F) 将立即返回值 NULL。如果设置为 SLEEP_FUNC，则在资源可用之前，scsi_init_pkt() 不会返回。当资源可能可用时，会将任何其他有效的内核地址解释为要调用的函数的地址。

arg 要传递给回调函数的参数。

传输之前，scsi_init_pkt() 例程将同步数据。如果驱动程序需要在传输后访问数据，则驱动程序应调用 scsi_sync_pkt(9F) 以刷新任何中间高速缓存。可以使用 scsi_sync_pkt() 例程来同步所有高速缓存的数据。

scsi_sync_pkt() 函数

如果在更改数据之后目标驱动程序需要重新提交包，则必须在调用 scsi_transport(9F) 之前调用 scsi_sync_pkt(9F)。但是，如果目标驱动程序不需要访问数据，则在传输之后不需要调用 scsi_sync_pkt()。

scsi_destroy_pkt() 函数

如有必要，scsi_destroy_pkt(9F) 例程将同步与包关联的任何剩余高速缓存数据。然后，该例程会释放包以及关联的命令、状态和目标驱动程序专用的数据区。应在命令完成例程中调用该例程。

scsi_alloc_consistent_buf() 函数

对于大多数 I/O 请求，驱动程序不直接访问传递到驱动程序入口点的数据缓冲区。该缓冲区仅传递到 scsi_init_pkt(9F)。如果某个驱动程序发送的 SCSI 命令是针对该驱动程序本身检查的缓冲区，那么这些缓冲区应该支持 DMA。SCSI 请求检测命令就是一个很好的示例。scsi_alloc_consistent_buf(9F) 例程分配 buf(9S) 结构和适用于 DMA 一致操作的数据缓冲区。HBA 首先会执行任何必需的缓冲区同步，然后再执行命令完成回调。

注 – scsi_alloc_consistent_buf(9F) 将使用珍贵的系统资源。因此，应有节制地使用 scsi_alloc_consistent_buf()。

scsi_free_consistent_buf() 函数

scsi_free_consistent_buf(9F) 释放 buf(9S) 结构和使用 scsi_alloc_consistent_buf(9F) 分配的关联数据缓冲区。有关示例，请参见第 295 页中的 “attach() 入口点（SCSI 目标驱动程序）”和第 298 页中的 “detach() 入口点（SCSI 目标驱动程序）”。

生成和传输命令

主机总线适配器驱动程序负责向设备传输命令。此外，该驱动程序还负责处理低级别的 SCSI 协议。scsi_transport(9F) 例程将包提交给主机总线适配器驱动程序以进行传输。目标驱动程序负责创建有效的 scsi_pkt(9S) 结构。

生成命令

scsi_init_pkt(9F) 例程可为 SCSI CDB 分配空间，在必要时分配 DMA 资源以及设置 pkt_flags 字段，如以下示例所示：

```
pkt = scsi_init_pkt(&sdp->sd_address, NULL, bp,
    CDB_GROUP0, 1, 0, 0, SLEEP_FUNC, NULL);
```

该示例在按传递的 buf(9S) 结构指针中的指定分配 DMA 资源的同时创建了一个新包。为 Group 0（6 字节）命令分配了 SCSI CDB。pkt_flags 字段被设置为零，但没有为 pkt_private 字段分配空间。由于设置了 SLEEP_FUNC 参数，对 scsi_init_pkt(9F) 的调用将会在当前没有可用资源的情况下无限期等待资源。

下一步是使用 scsi_setup_cdb(9F) 函数初始化 SCSI CDB。

```
if (scsi_setup_cdb((union scsi_cdb *)pkt->pkt_cdbp,
    SCMD_READ, bp->b_blkno, bp->b_bcount >> DEV_BSHIFT, 0) == 0)
    goto failed;
```

该示例将生成 Group 0 命令描述符块。该示例按如下所示填充 pkt_cdbp 字段：

- 该命令本身位于第 0 个字节中。该命令通过 SCMD_READ 参数进行设置。
- 地址字段位于第 1 个字节的 0-4 位以及第 2 个字节和第 3 个字节中。地址通过 bp->b_blkno 进行设置。

- 计数字段位于第 4 个字节中。计数通过最后一个参数进行设置。在本例中，count 设置为 bp->b_bcount >> DEV_BSHIFT，其中 DEV_BSHIFT 是已转换为块数的传输字节计数。

注 – scsi_setup_cdb(9F) 不支持在 SCSI 命令块的第 1 个字节的 5-7 位中设置目标设备的逻辑单元号 (logical unit number, LUN)。此要求由 SCSI-1 定义。对于需要在命令块中设置 LUN 位的 SCSI-1 设备，请使用 makecom_g0(9F) 或某些等效的函数，而不是使用 scsi_setup_cdb(9F)。

初始化 SCSI CDB 之后，应初始化包中的三个其他字段，并在状态结构中存储为指向包的指针。

```
pkt->pkt_private = (opaque_t)bp;
pkt->pkt_comp = xxcallback;
pkt->pkt_time = 30;
xsp->pkt = pkt;
```

buf(9S) 指针保存在 pkt_private 字段中，以备将来在完成例程中使用。

设置目标功能

目标驱动程序使用 scsi_ifsetcap(9F) 设置主机适配器驱动程序的功能。**上限**是一个名称/值对，由一个以 null 结尾的字符串和一个整数值组成。可以使用 scsi_ifgetcap(9F) 检索功能的当前值。scsi_ifsetcap(9F) 允许为总线上的所有目标设置功能。

但是，通常建议不要设置非目标驱动程序拥有的目标的功能。并非所有 HBA 驱动程序都支持这种做法。缺省情况下，某些功能（如断开连接和同步）可以由 HBA 驱动程序设置。其他功能可能需要由目标驱动程序显式设置。例如，Wide-xfer 和标记排队功能必须由目标驱动器设置。

传输命令

填充 scsi_pkt(9S) 结构之后，应使用 scsi_transport(9F) 将该结构提交给总线适配器驱动程序：

```
if (scsi_transport(pkt) != TRAN_ACCEPT) {
    bp->b_resid = bp->b_bcount;
    bioerror(bp, EIO);
    biodone(bp);
}
```

从 scsi_transport(9F) 返回的其他值如下所示：

- TRAN_BUSY－表示已在运行针对指定目标的命令。
- TRAN_BADPKT－包中的 DMA 计数太大，或者主机适配器驱动程序由于其他原因拒绝了该包。
- TRAN_FATAL_ERROR－主机适配器驱动程序无法接受该包。

注 – 在调用 scsi_transport(9F) 过程中，不能持有 scsi_device(9S) 结构中的互斥锁 sd_mutex。

如果 scsi_transport(9F) 返回 TRAN_ACCEPT，则该包将由主机总线适配器驱动程序负责。调用命令完成例程之前，目标驱动程序不应该访问该包。

同步 scsi_transport() 函数

如果在包中设置了 FLAG_NOINTR，则在命令完成之前，scsi_transport(9F) 不会返回。不会执行回调。

注 – 请勿在中断上下文中使用 FLAG_NOINTR。

命令完成

当主机总线适配器驱动程序完成命令后，将调用包的完成回调例程。然后，驱动程序会将指向 scsi_pkt(9S) 结构的指针作为参数传递。对包进行解码后，完成例程将执行相应的操作。

示例 17–5 显示了一个简单的完成回调例程。该代码检查传输是否失败。如果存在失败情况，该例程将放弃运行，而不会重试命令。如果目标繁忙，则需要额外的代码以便在以后重新提交命令。

如果命令产生检查条件，则目标驱动程序需要发送请求检测命令，除非启用了自动请求检测。

否则，命令成功。在结束命令处理时，命令将销毁包并调用 biodone(9F)。

如果发生传输错误（如总线重置或奇偶校验问题），则目标驱动程序可以使用 scsi_transport(9F) 重新提交该包。重新提交之前，无需更改包中的任何值。

以下示例不会尝试重试未完成的命令。

注 – 通常，在中断上下文中会调用目标驱动程序的回调函数。因此，回调函数绝不应处于休眠状态。

示例 17–5 SCSI驱动程序的完成例程

```
static void
xxcallback(struct scsi_pkt *pkt)
{
    struct buf        *bp;
    struct xxstate    *xsp;
    minor_t           instance;
    struct scsi_status *ssp;
    /*
     * Get a pointer to the buf(9S) structure for the command
     * and to the per-instance data structure.
     */
    bp = (struct buf *)pkt->pkt_private;
    instance = getminor(bp->b_edev);
    xsp = ddi_get_soft_state(statep, instance);
    /*
     * Figure out why this callback routine was called
     */
    if (pkt->pkt_reason != CMP_CMPLT) {
        bp->b_resid = bp->b_bcount;
        bioerror(bp, EIO);
        scsi_destroy_pkt(pkt);          /* Release resources */
        biodone(bp);                    /* Notify waiting threads */ ;
    } else {
        /*
         * Command completed, check status.
         * See scsi_status(9S)
         */
        ssp = (struct scsi_status *)pkt->pkt_scbp;
        if (ssp->sts_busy) {
            /* error, target busy or reserved */
        } else if (ssp->sts_chk) {
            /* Send a request sense command. */
        } else {
            bp->b_resid = pkt->pkt_resid;  /* Packet completed OK */
            scsi_destroy_pkt(pkt);
            biodone(bp);
       }
    }
}
```

重新使用包

目标驱动程序可以采用以下方式重新使用包：

- 重新提交未更改的包。
- 使用 scsi_sync_pkt(9F) 同步数据。然后，处理驱动程序中的数据。最后，重新提交包。
- 使用 scsi_dmafree(9F) 释放 DMA 资源，然后将 pkt 指针传递给 scsi_init_pkt(9F) 以绑定到新的 bp。目标驱动程序负责重新初始化包。该 CDB 的长度必须与上一个 CDB 的长度相同。

- 如果首次调用 scsi_init_pkt(9F) 时仅分配了部分 DMA，则以后可以针对同一个包调用 scsi_init_pkt(9F)。同样，可以调用 bp 以调整下一个传输部分的 DMA 资源。

自动请求检测模式

如果使用排队（无论是标记排队还是无标记排队），则最好使用自动请求检测模式。任何后续命令都会清除应急处理状态 (contingent allegiance condition)，并因此导致检测数据丢失。大多数 HBA 驱动程序会在执行目标驱动程序回调之前开始下一条命令。其他 HBA 驱动程序可以使用单独的、较低优先级的线程执行回调。采用该方法时，如果存在检查状况，那么向目标驱动程序通知包已完成所需的时间可能会增加。在这种情况下，目标驱动程序可能无法及时提交请求检测命令以检索检测数据。

为了避免检测数据丢失，HBA 驱动程序或控制器应在检测到检查条件时发出请求检测命令。该模式称为自动请求检测模式。请注意，并非所有 HBA 驱动程序都可以使用自动请求检测模式，而某些驱动程序只能在启用了自动请求检测模式时运行。

目标驱动程序使用 scsi_ifsetcap(9F) 来启用自动请求检测模式。以下示例说明了如何启用自动请求检测。

示例 17–6 启用自动请求检测模式

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    struct xxstate *xsp;
    struct scsi_device *sdp = (struct scsi_device *)
    ddi_get_driver_private(dip);
    /*
     * Enable auto-request-sense. An auto-request-sense command might
     * fail due to a BUSY condition or transport error. Therefore,
     * it is recommended to allocate a separate request sense
     * packet as well.
     * Note that scsi_ifsetcap(9F) can return -1, 0, or 1
     */
    xsp->sdp_arq_enabled =
    ((scsi_ifsetcap(ROUTE, "auto-rqsense", 1, 1) == 1) ? 1 : 0);
    /*
     * If the HBA driver supports auto request sense then the
     * status blocks should be sizeof (struct scsi_arq_status).
     * Else, one byte is sufficient.
     */
    xsp->sdp_cmd_stat_size =  (xsp->sdp_arq_enabled ?
    sizeof (struct scsi_arq_status) : 1);
    /* ... */
}
```

如果使用 scsi_init_pkt(9F) 分配了一个包并且该包需要自动请求检测，则需要增加空间。目标驱动程序必须为状态块请求此空间，以便存储自动请求检测结构。请求检测

命令中使用的检测长度即 struct scsi_extended_sense 中的 sizeof。通过为状态块分配 struct scsi_status 中的 sizeof，可以针对各个包禁用自动请求检测。

可像往常一样使用 scsi_transport(9F) 提交包。当该包中出现检查条件时，主机适配器驱动程序将执行以下步骤：

- 发出请求检测命令（如果控制器没有自动请求检测功能）
- 获取检测数据
- 在包的状态块中填充 scsi_arq_status 信息
- 在包的 pkt_state 字段中设置 STATE_ARQ_DONE
- 调用包的回调处理程序 (pkt_comp())

目标驱动程序的回调例程应通过检查 pkt_state 中的 STATE_ARQ_DONE 位，来验证检测数据是否可用。STATE_ARQ_DONE 表明出现了检查条件，并且已执行请求检测。如果包中暂时禁用了自动请求检测，则无法保证对检测数据的后续检索。

然后，目标驱动程序应验证自动请求检测命令是否已成功完成，并对检测数据进行解码。

转储处理

在系统出现故障或执行检查点操作时，dump(9E) 入口点会将部分虚拟地址空间直接复制到指定的设备。请参见 cpr(7) 和 dump(9E) 手册页。dump(9E) 入口点必须在不使用中断的情况下能够执行该操作。

dump() 的参数如下所示：

dev 转储设备的设备编号

addr 开始转储的内核虚拟地址

blkno 设备上的第一个目标块

nblk 要转储的块数

示例 17–7 dump(9E) 例程

```
static int
xxdump(dev_t dev, caddr_t addr, daddr_t blkno, int nblk)
{
    struct xxstate     *xsp;
    struct buf         *bp;
    struct scsi_pkt    *pkt;
    int    rval;
    int    instance;

    instance = getminor(dev);
    xsp = ddi_get_soft_state(statep, instance);

    if (tgt->suspended) {
```

示例 17–7 dump(9E) 例程 （续）

```
        (void) pm_raise_power(DEVINFO(tgt), 0, 1);
    }

    bp = getrbuf(KM_NOSLEEP);
    if (bp == NULL) {
        return (EIO);
    }

/* Calculate block number relative to partition. */
    bp->b_un.b_addr = addr;
    bp->b_edev = dev;
    bp->b_bcount = nblk * DEV_BSIZE;
    bp->b_flags = B_WRITE | B_BUSY;
    bp->b_blkno = blkno;

    pkt = scsi_init_pkt(ROUTE(tgt), NULL, bp, CDB_GROUP1,
    sizeof (struct scsi_arq_status),
    sizeof (struct bst_pkt_private), 0, NULL_FUNC, NULL);
    if (pkt == NULL) {
        freerbuf(bp);
        return (EIO);
    }
    (void) scsi_setup_cdb((union scsi_cdb *)pkt->pkt_cdbp,
        SCMD_WRITE_G1, blkno, nblk, 0);
    /*
     * While dumping in polled mode, other cmds might complete
     * and these should not be resubmitted. we set the
     * dumping flag here which prevents requeueing cmds.
     */
    tgt->dumping = 1;
    rval = scsi_poll(pkt);
    tgt->dumping = 0;

    scsi_destroy_pkt(pkt);
    freerbuf(bp);

    if (rval != DDI_SUCCESS) {
        rval = EIO;
    }
    return (rval);
}
```

SCSI 选项

SCSA 定义了一个全局变量 *scsi_options*，用于进行控制和调试。在文件 <*sys/scsi/conf/autoconf.h*> 中，可以找到 `scsi_options` 中定义的位。*scsi_options* 按以下方式使用这些位：

`SCSI_OPTIONS_DR` 启用全局断开连接或重新连接。

SCSI_OPTIONS_FAST	启用全局 FAST SCSI 支持：每秒传输 10 MB。除非设置了 SCSI_OPTIONS_FAST (0x100) 位，否则 HBA 不应采用 FAST SCSI 模式运行。
SCSI_OPTIONS_FAST20	启用全局 FAST20 SCSI 支持：每秒传输 20 MB。除非设置了 SCSI_OPTIONS_FAST20 (0x400) 位，否则 HBA 不应采用 FAST20 SCSI 模式运行。
SCSI_OPTIONS_FAST40	启用全局 FAST40 SCSI 支持：每秒传输 40 MB。除非设置了 SCSI_OPTIONS_FAST40 (0x800) 位，否则 HBA 不应采用 FAST40 SCSI 模式运行。
SCSI_OPTIONS_FAST80	启用全局 FAST80 SCSI 支持：每秒传输 80 MB。除非设置了 SCSI_OPTIONS_FAST80 (0x1000) 位，否则 HBA 不应采用 FAST80 SCSI 模式运行。
SCSI_OPTIONS_FAST160	启用全局 FAST160 SCSI 支持：每秒传输 160 MB。除非设置了 SCSI_OPTIONS_FAST160 (0x2000) 位，否则 HBA 不应采用 FAST160 SCSI 模式运行。
SCSI_OPTIONS_FAST320	启用全局 FAST320 SCSI 支持：每秒传输 320 MB。除非设置了 SCSI_OPTIONS_FAST320 (0x4000) 位，否则 HBA 不应采用 FAST320 SCSI 模式运行。
SCSI_OPTIONS_LINK	启用全局链接支持。
SCSI_OPTIONS_PARITY	启用全局奇偶校验支持。
SCSI_OPTIONS_QAS	启用“快速仲裁选择”功能。QAS（快速仲裁选择）用于降低设备仲裁并访问总线时的协议开销。只有 Ultra4 (FAST160) SCSI 设备支持 QAS，但是并非所有此类设备都支持 QAS。除非设置了 SCSI_OPTIONS_QAS (0x100000) 位，否则 HBA 不应采用 QAS SCSI 模式运行。请查阅相应的 Oracle 硬件文档，以确定您的计算机是否支持 QAS。
SCSI_OPTIONS_SYNC	启用全局同步传输功能。
SCSI_OPTIONS_TAG	启用全局标记排队支持。
SCSI_OPTIONS_WIDE	启用全局 WIDE SCSI。

注 – 设置 *scsi_options* 会影响系统中存在的**所有**主机总线适配器驱动程序和所有目标驱动程序。有关控制特定主机适配器的这些选项的信息，请参阅 scsi_hba_attach(9F) 手册页。

◆ ◆ ◆ 第 18 章

18 SCSI 主机总线适配器驱动程序

本章介绍有关创建 SCSI 主机总线适配器 (host bus adapter, HBA) 驱动程序的信息。本章提供了用于说明典型 HBA 驱动程序的结构的样例代码。样例代码说明了如何使用 Sun 公用 SCSI 体系结构 (Sun Common SCSI Architecture, SCSA) 提供的 HBA 驱动程序接口。本章介绍有关以下主题的信息：

主机总线适配器驱动程序介绍

如第 17 章，SCSI 目标驱动程序中所述，DDI/DKI 可将 SCSI 设备的软件接口分成以下两个主要部分：

- 目标设备和驱动程序
- 主机总线适配器设备和驱动程序

目标设备是指连接到 SCSI 总线上的设备，如磁盘或磁带机。**目标驱动程序**是指作为设备驱动程序安装的软件组件。SCSI 总线上的每个目标设备都由一个目标驱动程序实例控制。

主机总线适配器设备是指 HBA 硬件，如 SBus 或 PCI SCSI 适配卡。**主机总线适配器驱动程序**是指作为设备驱动程序安装的软件组件。例如，SPARC 计算机中的 `esp` 驱动程序、x86 计算机中的 `ncrs` 驱动程序以及适用于这两种体系结构的 `isp` 驱动程序。一个 HBA 驱动程序实例可控制系统中配置的它的各个主机总线适配器设备。

Sun 公用 SCSI 体系结构 (Sun Common SCSI Architecture, SCSA) 定义了目标组件和 HBA 组件之间的接口。

注 – 了解 SCSI 目标驱动程序是编写有效的 SCSI HBA 驱动程序的基本先决条件。有关 SCSI 目标驱动程序的信息，请参见第 17 章，SCSI 目标驱动程序。目标驱动程序开发者通过阅读本章也会有所收益。

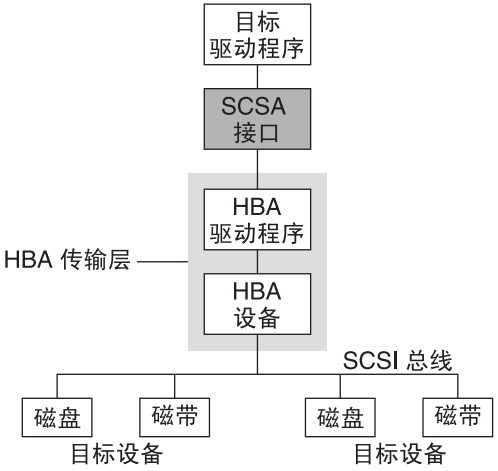
主机总线适配器驱动程序负责执行以下任务：

- 管理主机总线适配器硬件
- 接受来自 SCSI 目标驱动程序的 SCSI 命令
- 将命令传输到指定的 SCSI 目标设备
- 执行命令要求的任何数据传送
- 收集状态
- 处理自动请求检测（可选）
- 通知目标驱动程序命令执行的成败

SCSI 接口

SCSA 是 DDI/DKI 编程接口，用于将 SCSI 命令从目标驱动程序传送到主机适配器驱动程序。通过与 SCSA 保持兼容，目标驱动程序可以将 SCSI 命令和序列的任何组合轻松传递到目标设备。无需了解主机适配器的硬件实现。从概念上讲，SCSA 会将生成 SCSI 命令与将命令（和数据）传输到 SCSI 总线这两个过程分离开来。SCSA 通过 HBA **传输**层管理目标驱动程序与 HBA 驱动程序之间的连接，如下图所示：

图 18–1 SCSA 接口

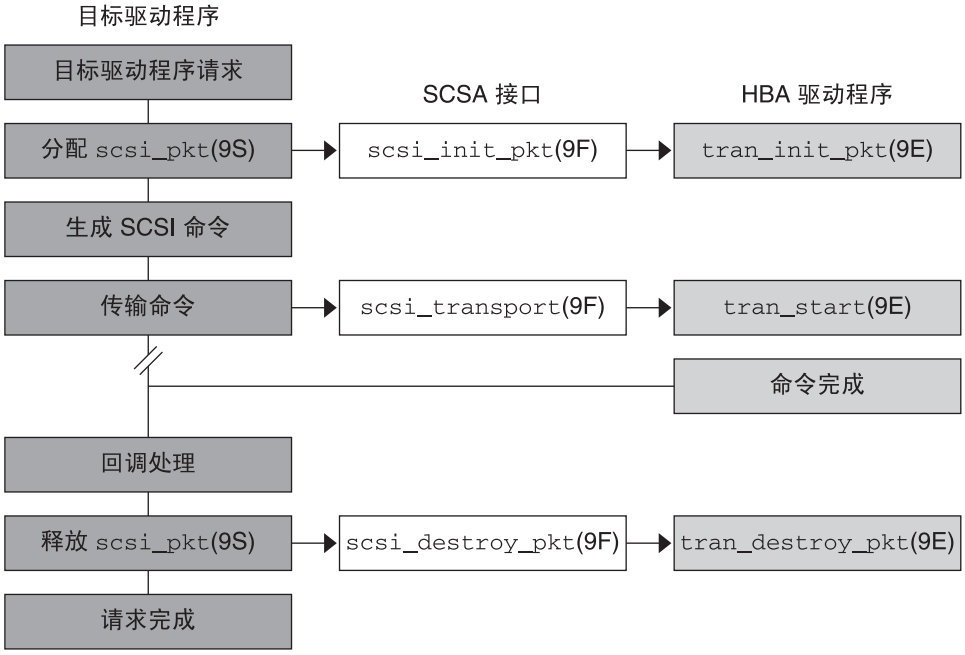


HBA **传输层**是一个硬件和软件层，负责将 SCSI 命令传输到 SCSI 目标设备。HBA 驱动程序会响应 SCSI 目标驱动程序通过 SCSA 发出的请求，提供资源分配、DMA 管理和传

输服务。另外，主机适配器驱动程序还会管理主机适配器硬件以及执行命令所需的 SCSI 协议。完成命令后，HBA 驱动程序将调用目标驱动程序的 SCSI pkt 命令完成例程。

以下示例说明了此流程，并重点介绍了如何将信息从目标驱动程序传送到 SCSA，然后再传送到 HBA 驱动程序。下图还说明了典型的传输入口点和函数调用。

图 18–2 传输层流程



SCSA HBA 接口

SCSA HBA 接口包括 HBA 入口点、HBA 数据结构和 HBA 框架。

SCSA HBA 入口点汇总

SCSA 定义了许多 HBA 驱动程序入口点。下表中列出了这些入口点。配置连接到 HBA 驱动程序的目标驱动程序实例时，系统将会调用这些入口点。另外，目标驱动程序发出 SCSA 请求时，也会调用这些入口点。有关更多信息，请参见第 326 页中的 “SCSA HBA 驱动程序入口点”。

表 18–1 SCSA HBA 入口点汇总

函数名	调用原因
tran_abort(9E)	目标驱动程序调用 scsi_abort(9F)
tran_bus_reset(9E)	系统重置总线
tran_destroy_pkt(9E)	目标驱动程序调用 scsi_destroy_pkt(9F)
tran_dmafree(9E)	目标驱动程序调用 scsi_dmafree(9F)
tran_getcap(9E)	目标驱动程序调用 scsi_ifgetcap(9F)
tran_init_pkt(9E)	目标驱动程序调用 scsi_init_pkt(9F)
tran_quiesce(9E)	系统停止总线
tran_reset(9E)	目标驱动程序调用 scsi_reset(9F)
tran_reset_notify(9E)	目标驱动程序调用 scsi_reset_notify(9F)
tran_setcap(9E)	目标驱动程序调用 scsi_ifsetcap(9F)
tran_start(9E)	目标驱动程序调用 scsi_transport(9F)
tran_sync_pkt(9E)	目标驱动程序调用 scsi_sync_pkt(9F)
tran_tgt_free(9E)	系统分离目标设备实例
tran_tgt_init(9E)	系统连接目标设备实例
tran_tgt_probe(9E)	目标驱动程序调用 scsi_probe(9F)
tran_unquiesce(9E)	系统恢复总线上的活动

SCSA HBA 数据结构

SCSA 定义了多种数据结构，以便可在目标驱动程序和 HBA 驱动程序之间交换信息。其中包括以下数据结构：

- scsi_hba_tran(9S)
- scsi_address(9S)
- scsi_device(9S)
- scsi_pkt(9S)

scsi_hba_tran() 结构

HBA 驱动程序的每个实例都必须在 scsi_hba_tran(9S) 入口点中使用 scsi_hba_tran_alloc(9F) 函数分配 attach(9E) 结构。scsi_hba_tran_alloc() 函数可初始化 scsi_hba_tran 结构。HBA 驱动程序必须初始化传输结构中的特定向量才能指向 HBA 驱动程序中的入口点。初始化 scsi_hba_tran 结构后，HBA 驱动程序通过调用 scsi_hba_attach_setup(9F) 函数将传输结构导出到 SCSA。

注意 – 由于 SCSA 将指向传输结构的指针保存在 devinfo 节点的驱动程序专用字段中，因此 HBA 驱动程序决不能使用 ddi_set_driver_private(9F)。但是，HBA 驱动程序可以使用 ddi_get_driver_private(9F) 来检索指向传输结构的指针。

SCSA 接口要求 HBA 驱动程序提供许多可通过 scsi_hba_tran 结构调用的入口点。有关更多信息，请参见第 326 页中的 “SCSA HBA 驱动程序入口点”。

scsi_hba_tran 结构包含以下字段：

```
struct scsi_hba_tran {
    dev_info_t          *tran_hba_dip;          /* HBAs dev_info pointer */
    void                *tran_hba_private;      /* HBA softstate */
    void                *tran_tgt_private;      /* HBA target private pointer */
    struct scsi_device  *tran_sd;               /* scsi_device */
    int                 (*tran_tgt_init)();     /* Transport target */
                                                /* Initialization */
    int                 (*tran_tgt_probe)();    /* Transport target probe */
    void                (*tran_tgt_free)();     /* Transport target free */
    int                 (*tran_start)();        /* Transport start */
    int                 (*tran_reset)();        /* Transport reset */
    int                 (*tran_abort)();        /* Transport abort */
    int                 (*tran_getcap)();       /* Capability retrieval */
    int                 (*tran_setcap)();       /* Capability establishment */
    struct scsi_pkt     *(*tran_init_pkt)();    /* Packet and DMA allocation */
    void                (*tran_destroy_pkt)();  /* Packet and DMA */
                                                /* Deallocation */
    void                (*tran_dmafree)();      /* DMA deallocation */
    void                (*tran_sync_pkt)();     /* Sync DMA */
    void                (*tran_reset_notify)(); /* Bus reset notification */
    int                 (*tran_bus_reset)();    /* Reset bus only */
    int                 (*tran_quiesce)();      /* Quiesce a bus */
    int                 (*tran_unquiesce)();    /* Unquiesce a bus */
    int                 tran_interconnect_type; /* transport interconnect */
};
```

下面的描述提供了有关这些 scsi_hba_tran 结构字段的更多信息：

tran_hba_dip
: 指向 HBA 设备实例 dev_info 结构的指针。函数 scsi_hba_attach_setup(9F) 可用于设置此字段。

tran_hba_private
: 指向 HBA 驱动程序维护的专用数据的指针。通常，tran_hba_private 包含指向 HBA 驱动程序状态结构的指针。

tran_tgt_private
: 指向使用克隆时 HBA 驱动程序维护的专用数据的指针。通过在调用 scsi_hba_attach_setup(9F) 时指定 SCSI_HBA_TRAN_CLONE，可对每个目标克隆一次 scsi_hba_tran(9S) 结构。借助该方法，HBA 可将此字段初始化为指向 tran_tgt_init(9E) 入口点中按目标实例的数据结构。如果未指定 SCSI_HBA_TRAN_CLONE，则

	tran_tgt_private 为 NULL，并且决不能引用 tran_tgt_private。有关更多信息，请参见第 318 页中的 “传输结构克隆”。
tran_sd	指向克隆时使用的按目标实例的 scsi_device(9S) 结构的指针。如果将 SCSI_HBA_TRAN_CLONE 传递给 scsi_hba_attach_setup(9F)，则 tran_sd 会初始化指向按目标的 scsi_device 结构。在代表目标调用任何 HBA 函数之前，将进行此初始化。如果未指定 SCSI_HBA_TRAN_CLONE，则 tran_sd 为 NULL，并且决不能引用 tran_sd。有关更多信息，请参见第 318 页中的 “传输结构克隆”。
tran_tgt_init	指向初始化目标设备实例时调用的 HBA 驱动程序入口点的指针。如果无需进行按目标的初始化，则 HBA 可保持将 tran_tgt_init 设置为 NULL。
tran_tgt_probe	指向在目标驱动程序实例调用 scsi_probe(9F) 时调用的 HBA 驱动程序入口点的指针。调用该例程可探测目标设备是否存在。如果此 HBA 无需进行目标探测定制，则 HBA 应将 tran_tgt_probe 设置为 scsi_hba_probe(9F)。
tran_tgt_free	指向在目标设备实例被销毁时调用的 HBA 驱动程序入口点的指针。如果无需进行按目标的取消分配，则 HBA 可保持将 tran_tgt_free 设置为 NULL。
tran_start	指向在目标驱动程序调用 scsi_transport(9F) 时调用的 HBA 驱动程序入口点的指针。
tran_reset	指向在目标驱动程序调用 scsi_reset(9F) 时调用的 HBA 驱动程序入口点的指针。
tran_abort	指向在目标驱动程序调用 scsi_abort(9F) 时调用的 HBA 驱动程序入口点的指针。
tran_getcap	指向在目标驱动程序调用 scsi_ifgetcap(9F) 时调用的 HBA 驱动程序入口点的指针。
tran_setcap	指向在目标驱动程序调用 scsi_ifsetcap(9F) 时调用的 HBA 驱动程序入口点的指针。
tran_init_pkt	指向在目标驱动程序调用 scsi_init_pkt(9F) 时调用的 HBA 驱动程序入口点的指针。
tran_destroy_pkt	指向在目标驱动程序调用 scsi_destroy_pkt(9F) 时调用的 HBA 驱动程序入口点的指针。
tran_dmafree	指向在目标驱动程序调用 scsi_dmafree(9F) 时调用的 HBA 驱动程序入口点的指针。

tran_sync_pkt	指向在目标驱动程序调用 scsi_sync_pkt(9F) 时调用的 HBA 驱动程序入口点的指针。
tran_reset_notify	指向在目标驱动程序调用 tran_reset_notify(9E) 时调用的 HBA 驱动程序入口点的指针。
tran_bus_reset	重置 SCSI 总线但不重置目标的函数项。
tran_quiesce	等待所有未完成的命令完成并阻塞（或排队）任何发出的 I/O 请求的函数项。
tran_unquiesce	允许 I/O 活动在 SCSI 总线上恢复的函数项。
tran_interconnect_type	表示 services.h 头文件中定义的传输互连类型的整数值。

scsi_address 结构

scsi_address(9S) 结构可为目标驱动程序实例分配和传输的各个 SCSI 命令提供传输及寻址信息。

scsi_address 结构包含以下字段：

```
struct scsi_address {
    struct scsi_hba_tran    *a_hba_tran;    /* Transport vectors */
    ushort_t                a_target;       /* Target identifier */
    uchar_t                 a_lun;          /* LUN on that target */
    uchar_t                 a_sublun;       /* Sub LUN on that LUN */
                                            /* Not used */
};
```

a_hba_tran	指向 HBA 驱动程序分配和初始化的 scsi_hba_tran(9S) 结构的指针。如果将 SCSI_HBA_TRAN_CLONE 指定为 scsi_hba_attach_setup(9F) 的标志，则 a_hba_tran 指向该结构的副本。
a_target	标识 SCSI 总线上的 SCSI 目标。
a_lun	标识 SCSI 目标的 SCSI 逻辑单元。

scsi_device 结构

HBA 框架可为目标设备的各个实例分配和初始化 scsi_device(9S) 结构。该框架调用 HBA 驱动程序的 tran_tgt_init(9E) 入口点之前，将进行分配和初始化。此结构可存储有关每个 SCSI 逻辑单元的信息，包括指向信息区（包含通用信息和特定于设备的信息）的指针。对于连接到系统的每个目标设备实例，都存在一个 scsi_device(9S) 结构。

如果按目标的初始化成功，则 HBA 框架会使用 ddi_set_driver_private(9F) 将目标驱动程序的按实例的专用数据设置为指向 scsi_device(9S) 结构。请注意，如果 tran_tgt_init () 返回成功信息或该向量为 null，则表明初始化成功。

scsi_device(9S) 结构包含以下字段：

```
struct scsi_device {
    struct scsi_address           sd_address;    /* routing information */
    dev_info_t                    *sd_dev;       /* device dev_info node */
    kmutex_t                      sd_mutex;      /* mutex used by device */
    void                          *sd_reserved;
    struct scsi_inquiry           *sd_inq;
    struct scsi_extended_sense    *sd_sense;
    caddr_t                       sd_private;    /* for driver's use */
};
```

其中：

sd_address　为了进行 SCSI 资源分配而传递给例程的数据结构。

sd_dev　指向目标的 dev_info 结构的指针。

sd_mutex　供目标驱动程序使用的互斥锁。此互斥锁通过 HBA 框架进行初始化。目标驱动程序可将此互斥锁用作按设备的互斥锁。在调用 scsi_transport(9F) 或 scsi_poll(9F) 期间，不应持有此互斥锁。有关互斥锁的更多信息，请参见第 3 章，多线程。

sd_inq　目标设备的 SCSI 查询数据的指针。scsi_probe(9F) 例程可用于分配缓冲区、填充该缓冲区并将该缓冲区附加到此字段。

sd_sense　指向用于包含设备中的请求检测数据的缓冲区的指针。目标驱动程序必须分配和管理此缓冲区本身。有关更多信息，请参见第 101 页中的 “attach() 入口点”中目标驱动程序的 attach(9E) 例程。

sd_private　供目标驱动程序使用的指针字段。此字段通常用于存储指向专用目标驱动程序状态结构的指针。

scsi_pkt 结构 (HBA)

要执行 SCSI 命令，目标驱动程序必须首先为该命令分配 scsi_pkt(9S) 结构。然后，目标驱动程序必须指定其自身的专用数据区长度、命令状态和命令长度。HBA 驱动程序负责实现 tran_init_pkt(9E) 入口点中的包分配。另外，HBA 驱动程序还负责释放其 tran_destroy_pkt(9E) 入口点中的包。有关更多信息，请参见第 292 页中的 “scsi_pkt 结构（目标驱动程序）”。

scsi_pkt(9S) 结构包含以下字段：

```
struct scsi_pkt {
    opaque_t pkt_ha_private;             /* private data for host adapter */
    struct scsi_address pkt_address;     /* destination address */
    opaque_t pkt_private;                /* private data for target driver */
    void (*pkt_comp)(struct scsi_pkt *); /* completion routine */
    uint_t  pkt_flags;                   /* flags */
    int     pkt_time;                    /* time allotted to complete command */
    uchar_t *pkt_scbp;                   /* pointer to status block */
```

```
    uchar_t *pkt_cdbp;                  /* pointer to command block */
    ssize_t pkt_resid;                  /* data bytes not transferred */
    uint_t  pkt_state;                  /* state of command */
    uint_t  pkt_statistics;             /* statistics */
    uchar_t pkt_reason;                 /* reason completion called */
};
```

其中：

`pkt_ha_private`	指向按命令的 HBA 驱动程序专用数据的指针。
`pkt_address`	指向用于为此命令提供地址信息的 `scsi_address(9S)` 结构的指针。
`pkt_private`	指向按包的目标驱动程序专用数据的指针。
`pkt_comp`	指向在传输层完成此命令时 HBA 驱动程序调用的目标驱动程序完成例程的指针。
`pkt_flags`	命令的标志。
`pkt_time`	指定命令的完成超时时间（以秒为单位）。
`pkt_scbp`	指向命令的状态完成块的指针。
`pkt_cdbp`	指向命令的命令描述符块 (command descriptor block, CDB) 的指针。
`pkt_resid`	命令完成时**未**传送的数据字节计数。此字段也可能会用于指定尚未分配资源的数据量。在传输过程中，HBA 必须修改此字段。
`pkt_state`	命令的状态。在传输过程中，HBA 必须修改此字段。
`pkt_statistics`	提供命令在传输层中发生的事件的历史记录。在传输过程中，HBA 必须修改此字段。
`pkt_reason`	命令完成的原因。在传输过程中，HBA 必须修改此字段。

按目标实例的数据

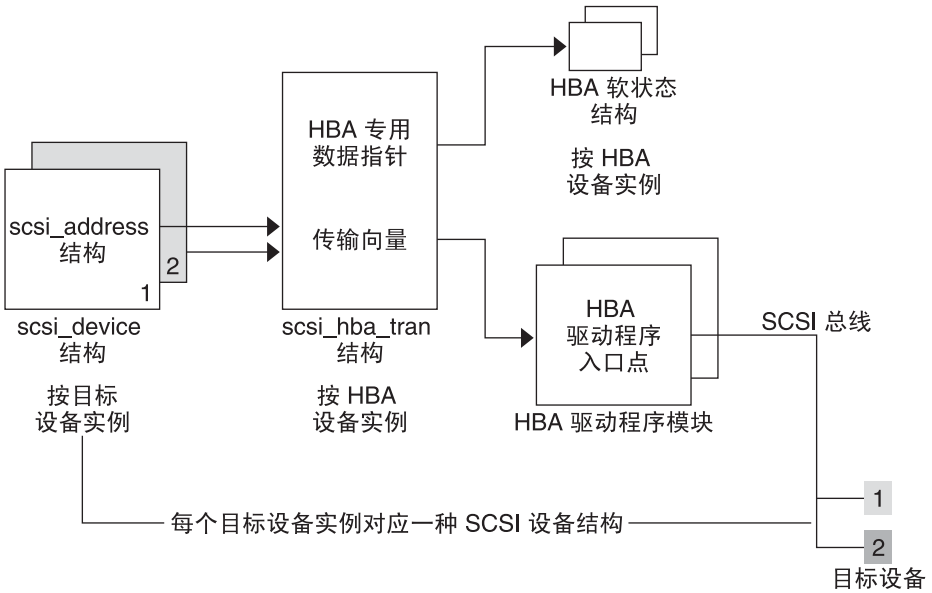
在执行 `attach(9E)` 期间，HBA 驱动程序必须分配 `scsi_hba_tran(9S)` 结构。然后，HBA 驱动程序必须将此传输结构中的向量初始化为指向 HBA 驱动程序所需的入口点。此 `scsi_hba_tran` 结构随后将传递给 `scsi_hba_attach_setup(9F)`。

`scsi_hba_tran` 结构包含 `tran_hba_private` 字段，该字段可用于引用 HBA 驱动程序的按实例状态。

每个 `scsi_address(9S)` 结构都包含一个指向 `scsi_hba_tran` 结构的指针。此外，`scsi_address` 结构还为特定的目标设备提供了目标（即 `a_target`）和逻辑单元 (`a_lun`) 地址。 通过 `scsi_device(9S)` 结构可直接或间接向 HBA 驱动程序的每个入口点传递一个指向 `scsi_address` 结构的指针。因此，HBA 驱动程序可以引用其自身的状态。HBA 驱动程序还可以标识已寻址的目标设备。

下图说明了用于传输操作的 HBA 数据结构。

图 18–3 HBA 传输结构



传输结构克隆

如果 HBA 驱动程序需要维护 scsi_hba_tran(9S) 结构中按目标的专用数据，则克隆可能会非常有用。克隆还可用于维护比 scsi_address(9S) 结构中所提供的更为复杂的地址。

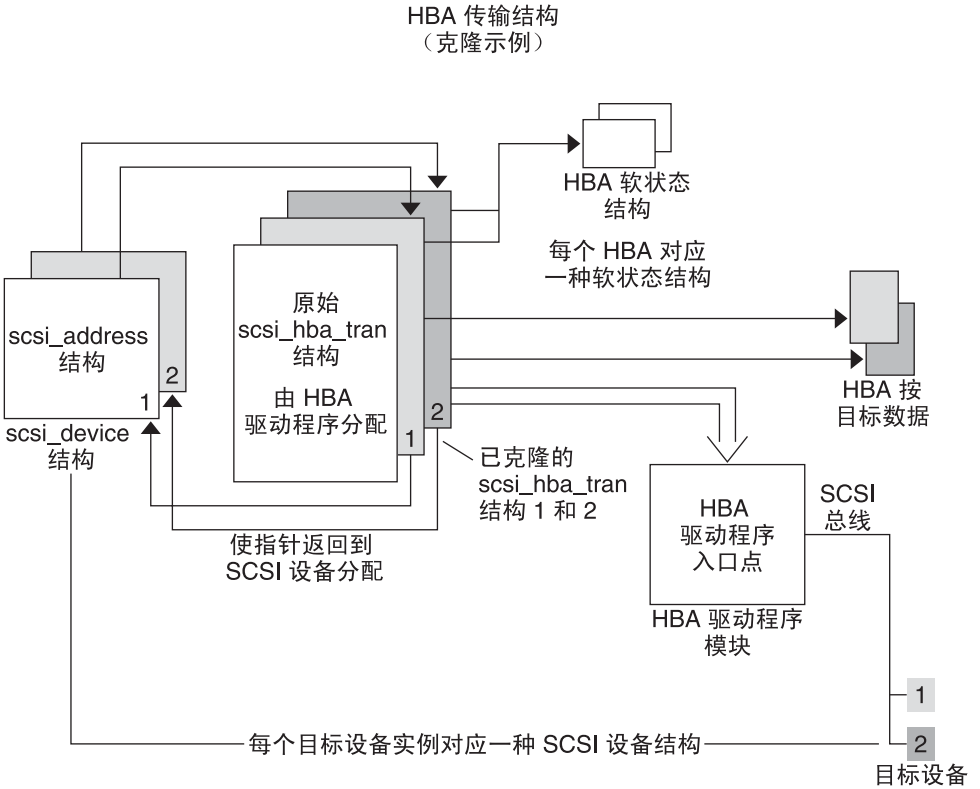
在克隆过程中，HBA 驱动程序仍必须在执行 attach(9E) 期间分配 scsi_hba_tran 结构。此外，HBA 驱动程序还必须初始化 HBA 驱动程序的 tran_hba_private 软状态指针和入口点向量。当框架开始将目标驱动程序实例连接到 HBA 驱动程序时，将会产生差异。调用 HBA 驱动程序的 tran_tgt_init(9E) 入口点之前，框架会克隆与 HBA 的该实例关联的 scsi_hba_tran 结构。相应地，为特定目标设备实例分配和初始化的每个 scsi_address 结构都会指向 scsi_hba_tran 结构的按目标实例的**副本**。scsi_address 结构不会指向 HBA 驱动程序在执行 attach() 期间分配的 scsi_hba_tran 结构。

指定克隆时，HBA 驱动程序可以使用两个重要的指针。这些指针包含在 scsi_hba_tran 结构中。第一个指针是 tran_tgt_private 字段，驱动程序可以使用该指针指向按目标的 HBA 专用数据。tran_tgt_private 指针非常有用，例如在 HBA 驱动程序需要维护比 a_target 和 a_lun 所提供的更为复杂的地址的情况下。第二个指针是 tran_sd 字段，该指针指向引用特定目标设备的 scsi_device(9S) 结构。

指定克隆时，HBA 驱动程序必须分配和初始化按目标的数据。HBA 驱动程序随后必须在执行其 tran_tgt_init(9E) 入口点过程中将 tran_tgt_private 字段初始化为指向此数据。 HBA 驱动程序必须在执行其 tran_tgt_free(9E) 入口点过程中释放按目标的数据。

克隆时，框架会在调用 HBA 驱动程序 tran_tgt_init() 入口点之前将 tran_sd 字段初始化为指向 scsi_device 结构。该驱动程序通过将 SCSI_HBA_TRAN_CLONE 标志传递给 scsi_hba_attach_setup(9F) 来请求克隆。下图说明了用于克隆传输操作的 HBA 数据结构。

图 18–4 克隆传输操作



SCSA HBA 函数

SCSA 还提供了许多函数。下表中列出了这些函数，供 HBA 驱动程序使用。

表 18–2 SCSA HBA 函数

函数名	进行调用的驱动程序入口点
scsi_hba_init(9F)	_init(9E)
scsi_hba_fini(9F)	_fini(9E)
scsi_hba_attach_setup(9F)	attach(9E)
scsi_hba_detach(9F)	detach(9E)
scsi_hba_tran_alloc(9F)	attach(9E)
scsi_hba_tran_free(9F)	detach(9E)
scsi_hba_probe(9F)	tran_tgt_probe(9E)
scsi_hba_pkt_alloc(9F)	tran_init_pkt(9E)
scsi_hba_pkt_free(9F)	tran_destroy_pkt(9E)
scsi_hba_lookup_capstr(9F)	tran_getcap(9E) 和 tran_setcap(9E)

HBA 驱动程序的相关性和配置问题

除将 SCSA HBA 入口点、结构和函数合并到驱动程序中外，开发者还必须处理驱动程序的相关性和配置问题。这些问题涉及配置属性、相关性声明、状态结构和按命令的结构、模块初始化入口点及自动配置入口点。

声明和结构

HBA 驱动程序必须包含以下头文件：

```
#include <sys/scsi/scsi.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

要向系统通知模块依赖于 SCSA 例程，必须使用以下命令生成驱动程序二进制代码。有关 SCSA 例程的更多信息，请参见第 311 页中的 “SCSA HBA 接口”。

```
% ld -r xx.o -o xx -N "misc/scsi"
```

样例代码是由 QLogic Intelligent SCSI 外围设备的简化 `isp` 驱动程序派生而来。`isp` 驱动程序支持 WIDE SCSI，每个目标最多 15 个目标设备和 8 个逻辑单元 (logical unit, LUN)。

每个命令的结构

通常，HBA驱动程序需要定义一个结构以维护目标驱动程序提交的每个命令的状态。按命令的结构的布局完全取决于设备驱动程序编写者。该布局需要反映硬件的功能和特征以及驱动程序使用的软件算法。

以下结构是每个命令的结构的示例。本章中的其余代码段将使用此结构说明 HBA 接口。

```
struct isp_cmd {
     struct isp_request     cmd_isp_request;
     struct isp_response    cmd_isp_response;
     struct scsi_pkt        *cmd_pkt;
     struct isp_cmd         *cmd_forw;
     uint32_t               cmd_dmacount;
     ddi_dma_handle_t       cmd_dmahandle;
     uint_t                 cmd_cookie;
     uint_t                 cmd_ncookies;
     uint_t                 cmd_cookiecnt;
     uint_t                 cmd_nwin;
     uint_t                 cmd_curwin;
     off_t                  cmd_dma_offset;
     uint_t                 cmd_dma_len;
     ddi_dma_cookie_t       cmd_dmacookies[ISP_NDATASEGS];
     u_int                  cmd_flags;
     u_short                cmd_slot;
     u_int                  cmd_cdblen;
     u_int                  cmd_scblen;
 };
```

模块初始化入口点

本节介绍 SCSI HBA 驱动程序执行的操作的入口点。

以下 SCSI HBA 驱动程序代码说明了典型的 dev_ops(9S) 结构。该驱动程序必须将此结构中的 devo_bus_ops 字段初始化为 NULL。SCSI HBA 驱动程序可提供特殊用途的叶驱动程序接口，在这种情况下，devo_cb_ops 字段可能会指向 cb_ops(9S) 结构。在此示例中，由于未导出任何叶驱动程序接口，因此 devo_cb_ops 字段会初始化为 NULL。

_init() 入口点（SCSI HBA 驱动程序）

_init(9E) 函数用于初始化可装入模块。_init() 在可装入模块中的其他任何例程之前调用。

在 SCSI HBA 中，_init() 函数在调用 mod_install(9F) 之前，必须先调用 scsi_hba_init(9F) 来通知框架是否存在 HBA 驱动程序。如果 scsi_hba__init () 返回非零值，则 _init() 应返回该值。否则，_init() 必须返回 mod_install(9F) 所返回的值。

该驱动程序在调用 mod_install(9F) 之前应初始化任何必需的全局状态。

如果 mod_install() 失败，则 _init() 函数必须释放分配的所有全局资源。_init() 必须在返回之前调用 scsi_hba_fini(9F)。

以下示例使用全局互斥锁说明如何分配对驱动程序的所有实例而言具有全局性的数据。该代码声明了全局互斥锁和软状态结构信息。全局互斥锁和软状态是在执行 _init() 的过程中初始化的。

_fini() 入口点（SCSI HBA 驱动程序）

如果系统准备尝试卸载 SCSI HBA 驱动程序，则会调用 _fini(9E) 函数。_fini() 函数必须调用 mod_remove(9F) 来确定是否可以卸载该驱动程序。如果 mod_remove() 返回 0，则可以卸载该模块。HBA 驱动程序必须取消分配 _init(9E) 中分配的所有全局资源。HBA 驱动程序还必须调用 scsi_hba_fini(9F)。

_fini() 必须返回 mod_remove() 所返回的值。

注 – 除非 mod_remove(9F) 返回 0，否则 HBA 驱动程序决不能释放任何资源或调用 scsi_hba_fini(9F)。

示例 18–1 说明了 SCSI HBA 的模块初始化。

示例 18–1 SCSI HBA 的模块初始化

```
static struct dev_ops isp_dev_ops = {
    DEVO_REV,       /* devo_rev */
    0,              /* refcnt  */
    isp_getinfo,    /* getinfo */
    nulldev,        /* identify */
    nulldev,        /* probe */
    isp_attach,     /* attach */
    isp_detach,     /* detach */
    nodev,          /* reset */
    NULL,           /* driver operations */
    NULL,           /* bus operations */
    isp_power,      /* power management */
    isp_quiesce,    /* quiesce */
};

/*
 * Local static data
 */
static kmutex_t      isp_global_mutex;
static void          *isp_state;

int
_init(void)
{
    int     err;

    if ((err = ddi_soft_state_init(&isp_state,
        sizeof (struct isp), 0)) != 0) {
```

示例 18–1 SCSI HBA 的模块初始化 （续）

```
        return (err);
    }
    if ((err = scsi_hba_init(&modlinkage)) == 0) {
        mutex_init(&isp_global_mutex, "isp global mutex",
        MUTEX_DRIVER, NULL);
        if ((err = mod_install(&modlinkage)) != 0) {
            mutex_destroy(&isp_global_mutex);
            scsi_hba_fini(&modlinkage);
            ddi_soft_state_fini(&isp_state);
        }
    }
    return (err);
}

int
_fini(void)
{
    int     err;

    if ((err = mod_remove(&modlinkage)) == 0) {
        mutex_destroy(&isp_global_mutex);
        scsi_hba_fini(&modlinkage);
        ddi_soft_state_fini(&isp_state);
    }
    return (err);
}
```

自动配置入口点

dev_ops(9S) 结构与每个设备驱动程序关联。通过该结构，内核可以查找驱动程序的自动配置入口点。有关这些自动配置例程的完整说明，请参见第 6 章，驱动程序自动配置。本节仅介绍与 SCSI HBA 驱动程序执行的操作关联的那些入口点。这些入口点包括 attach(9E) 和 detach(9E)。

attach() 入口点（SCSI HBA 驱动程序）

在为设备配置和附加驱动程序实例时，SCSI HBA 驱动程序的 attach(9E) 入口点将执行多个任务。对于实际设备的典型驱动程序，必须处理以下操作系统和硬件问题：

- 软状态结构
- DMA
- 传输结构
- 附加 HBA 驱动程序
- 寄存器映射
- 中断指定
- 中断处理
- 创建可管理电源的组件
- 报告附加状态

软状态结构

分配按设备实例的软状态结构时，如果发生错误，驱动程序必须仔细清理。

DMA

HBA 驱动程序必须通过正确初始化 ddi_dma_attr_t 结构来描述其 DMA 引擎的属性。

```
static ddi_dma_attr_t isp_dma_attr = {
     DMA_ATTR_V0,        /* ddi_dma_attr version */
     0,                  /* low address */
     0xffffffff,         /* high address */
     0x00ffffff,         /* counter upper bound */
     1,                  /* alignment requirements */
     0x3f,               /* burst sizes */
     1,                  /* minimum DMA access */
     0xffffffff,         /* maximum DMA access */
     (1<<24)-1,          /* segment boundary restrictions */
     1,                  /* scatter-gather list length */
     512,                /* device granularity */
     0                   /* DMA flags */
};
```

如果该驱动程序提供 DMA，则还应检查其硬件是否已安装在支持 DMA 的槽中：

```
if (ddi_slaveonly(dip) == DDI_SUCCESS) {
    return (DDI_FAILURE);
}
```

传输结构

驱动程序应进一步分配和初始化此实例的传输结构。tran_hba_private 字段会设置为指向此实例的软状态结构。如果无需特殊的探测定制，则可将 tran_tgt_probe 字段设置为 NULL 以实现缺省行为。

```
tran = scsi_hba_tran_alloc(dip, SCSI_HBA_CANSLEEP);

isp->isp_tran                   = tran;
isp->isp_dip                    = dip;

tran->tran_hba_private          = isp;
tran->tran_tgt_private          = NULL;
tran->tran_tgt_init             = isp_tran_tgt_init;
tran->tran_tgt_probe            = scsi_hba_probe;
tran->tran_tgt_free             = (void (*)())NULL;

tran->tran_start                = isp_scsi_start;
tran->tran_abort                = isp_scsi_abort;
tran->tran_reset                = isp_scsi_reset;
tran->tran_getcap               = isp_scsi_getcap;
tran->tran_setcap               = isp_scsi_setcap;
tran->tran_init_pkt             = isp_scsi_init_pkt;
tran->tran_destroy_pkt          = isp_scsi_destroy_pkt;
tran->tran_dmafree              = isp_scsi_dmafree;
```

```
tran->tran_sync_pkt             = isp_scsi_sync_pkt;
tran->tran_reset_notify         = isp_scsi_reset_notify;
tran->tran_bus_quiesce          = isp_tran_bus_quiesce
tran->tran_bus_unquiesce        = isp_tran_bus_unquiesce
tran->tran_bus_reset            = isp_tran_bus_reset
tran->tran_interconnect_type    = isp_tran_interconnect_type
```

附加 HBA 驱动程序

驱动程序应附加此设备实例并执行错误清理（如有必要）。

```
i = scsi_hba_attach_setup(dip, &isp_dma_attr, tran, 0);
if (i != DDI_SUCCESS) {
    /* do error recovery */
    return (DDI_FAILURE);
}
```

寄存器映射

驱动程序应在其设备的寄存器中进行映射。驱动程序需要指定以下项：

- 寄存器集的索引
- 设备的数据访问特征
- 要映射的寄存器的大小

```
ddi_device_acc_attr_t    dev_attributes;

     dev_attributes.devacc_attr_version = DDI_DEVICE_ATTR_V0;
     dev_attributes.devacc_attr_dataorder = DDI_STRICTORDER_ACC;
     dev_attributes.devacc_attr_endian_flags = DDI_STRUCTURE_LE_ACC;

     if (ddi_regs_map_setup(dip, 0, (caddr_t *)&isp->isp_reg,
     0, sizeof (struct ispregs), &dev_attributes,
     &isp->isp_acc_handle) != DDI_SUCCESS) {
        /* do error recovery */
        return (DDI_FAILURE);
     }
```

添加中断处理程序

驱动程序必须首先获取 *iblock cookie* 才能初始化驱动程序处理程序中使用的所有互斥锁。仅当初始化这些互斥锁后才能添加中断处理程序。

```
i = ddi_get_iblock_cookie(dip, 0, &isp->iblock_cookie};
if (i != DDI_SUCCESS) {
    /* do error recovery */
    return (DDI_FAILURE);
}

mutex_init(&isp->mutex, "isp_mutex", MUTEX_DRIVER,
(void *)isp->iblock_cookie);
i = ddi_add_intr(dip, 0, &isp->iblock_cookie,
0, isp_intr, (caddr_t)isp);
```

```
if (i != DDI_SUCCESS) {
    /* do error recovery */
    return (DDI_FAILURE);
}
```

如果需要高级处理程序，则应对驱动程序进行编码以提供此类处理程序。否则，驱动程序必须能够停止附加操作。有关高级中断处理的说明，请参见第 141 页中的 “处理高级别中断”。

创建可管理电源的组件

如果主机总线适配器只需要在所有目标适配器的电源级别为 0 时关闭电源，则使用电源管理，HBA 驱动程序只需提供 power(9E) 入口点。请参阅第 12 章，电源管理。另外，HBA 驱动程序还需要创建 pm-components(9P) 属性，用于描述设备实现的组件。

由于这些组件将缺省为空闲，并且电源管理框架的缺省相关性处理会确保主机总线适配器在目标适配器每次通电时也随之通电，因此无需再执行任何操作。如果自动启用自动电源管理，则该处理还将在所有目标适配器都断电时关闭主机总线适配器电源。

报告附加状态

最后，驱动程序应报告已附加的此驱动程序实例并返回成功信息。

```
ddi_report_dev(dip);
    return (DDI_SUCCESS);
```

detach() 入口点（SCSI HBA 驱动程序）

驱动程序会执行标准分离操作，包括调用 scsi_hba_detach(9F)。

SCSA HBA 驱动程序入口点

HBA 驱动程序可以通过 SCSA 接口与目标驱动程序协同工作。SCSA 接口要求 HBA 驱动程序提供许多可通过 scsi_hba_tran(9S) 结构调用的入口点。

这些入口点分为以下五个功能组：

- 目标驱动程序实例初始化
- 资源分配和取消资源分配
- 命令传输
- 功能管理
- 中止和重置处理
- 动态重新配置

下表按功能组列出了 SCSA HBA 入口点。

表 18–3 SCSA 入口点

功能组	组内入口点	说明
目标驱动程序实例初始化	tran_tgt_init(9E)	执行按目标的初始化（可选）
	tran_tgt_probe(9E)	探测 SCSI 总线是否存在目标（可选）
	tran_tgt_free(9E)	执行按目标的取消分配（可选）
资源分配	tran_init_pkt(9E)	分配 SCSI 包和 DMA 资源
	tran_destroy_pkt(9E)	释放 SCSI 包和 DMA 资源
	tran_sync_pkt(9E)	执行 DMA 前后同步内存
	tran_dmafree(9E)	释放 DMA 资源
命令传输	tran_start(9E)	传输 SCSI 命令
功能管理	tran_getcap(9E)	查询功能值
	tran_setcap(9E)	设置功能值
中止和重置	tran_abort(9E)	中止未完成的 SCSI 命令
	tran_reset(9E)	重置目标设备或 SCSI 总线
	tran_bus_reset(9E)	重置 SCSI 总线
	tran_reset_notify(9E)	请求向目标发出总线重置通知（可选）
动态重新配置	tran_quiesce(9E)	停止总线上的活动
	tran_unquiesce(9E)	恢复总线上的活动

目标驱动程序实例初始化

以下各节介绍了目标入口点。

tran_tgt_init() 入口点

使用 tran_tgt_init(9E) 入口点，HBA 可以分配和初始化按目标的任何资源。此外，tran_tgt_init() 还允许 HBA 将设备地址限定为该特定 HBA 的有效且可支持的地址。如果返回 DDI_FAILURE，则不会探测或附加该设备的目标驱动程序实例。

无需使用 tran_tgt_init()。如果未提供 tran_tgt_init()，则框架会尝试探测和附加相应目标驱动程序的所有可能实例。

```
static int
isp_tran_tgt_init(
    dev_info_t            *hba_dip,
```

```
    dev_info_t            *tgt_dip,
    scsi_hba_tran_t       *tran,
    struct scsi_device    *sd)
{
    return ((sd->sd_address.a_target < N_ISP_TARGETS_WIDE &&
        sd->sd_address.a_lun < 8) ? DDI_SUCCESS : DDI_FAILURE);
}
```

tran_tgt_probe() 入口点

使用 tran_tgt_probe(9E) 入口点，HBA 可以定制 scsi_probe(9F) 的操作（如有必要）。仅当目标驱动程序调用 scsi_probe() 时，才会调用此入口点。

HBA 驱动程序可以通过调用 scsi_hba_probe(9F) 并返回其返回值来保留 scsi_probe() 的正常操作。

无需使用此入口点。如果不需要此入口点，则 HBA 驱动程序应将 scsi_hba_tran(9S) 结构中的 tran_tgt_probe 向量设置为指向 scsi_hba_probe()。

scsi_probe() 可用于分配 scsi_inquiry(9S) 结构，并将 scsi_device(9S) 结构的 sd_inq 字段设置为指向 scsi_inquiry 中的数据。scsi_hba_probe() 可自动处理此任务。scsi_unprobe(9F) 随后将释放 scsi_inquiry 数据。

除分配 scsi_inquiry 数据以外，tran_tgt_probe() 必须是无状态的，因为同一 SCSI 设备可能会多次调用 tran_tgt_probe()。通常，scsi_inquiry 数据的分配通过 scsi_hba_probe() 来处理。

注 – scsi_inquiry(9S) 结构的分配通过 scsi_hba_probe() 自动处理。此信息仅在需要定制 scsi_probe() 处理时才有意义。

```
static int
isp_tran_tgt_probe(
    struct scsi_device    *sd,
    int                   (*callback)())
{
    /*
     * Perform any special probe customization needed.
     * Normal probe handling.
     */
    return (scsi_hba_probe(sd, callback));
}
```

tran_tgt_free() 入口点

使用 tran_tgt_free(9E) 入口点，HBA 可以执行目标实例的所有取消分配或清理过程。此入口点是可选的。

```
static void
isp_tran_tgt_free(
    dev_info_t            *hba_dip,
```

```
    dev_info_t            *tgt_dip,
    scsi_hba_tran_t       *hba_tran,
    struct scsi_device    *sd)
{
    /*
     * Undo any special per-target initialization done
     * earlier in tran_tgt_init(9F) and tran_tgt_probe(9F)
     */
}
```

资源分配

以下各节讨论了资源分配。

tran_init_pkt() 入口点

tran_init_pkt(9E) 入口点可为目标驱动程序请求分配和初始化 scsi_pkt(9S) 结构和 DMA 资源。

目标驱动程序调用 SCSA 函数 scsi_init_pkt(9F) 时，将会调用 tran_init_pkt(9E) 入口点。

每次调用 tran_init_pkt(9E) 入口点时，都会请求执行以下三种可能服务中的一种或多种：

- 分配和初始化 scsi_pkt(9S) 结构
- 分配用于数据传送的 DMA 资源
- 重新分配用于下一个数据传送部分的 DMA 资源

分配和初始化 scsi_pkt(9S) 结构

如果 pkt 为 NULL，则 tran_init_pkt(9E) 入口点必须通过 scsi_hba_pkt_alloc(9F) 分配 scsi_pkt(9S) 结构。

scsi_hba_pkt_alloc(9F) 可为以下各项分配空间：

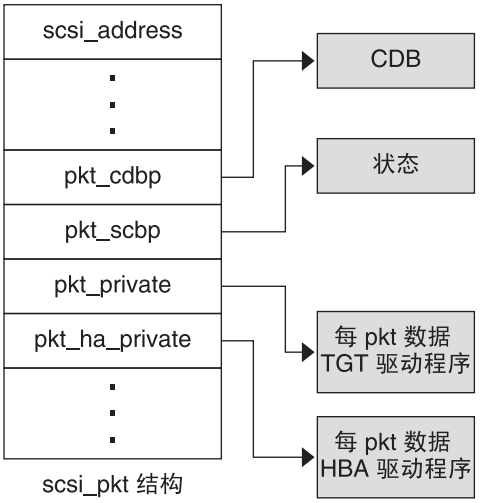
- scsi_pkt(9S)
- 长度为 cmdlen 的 SCSI CDB
- 长度为 statuslen 的 SCSI 状态的完成区
- 长度为 tgtlen 的按包的目标驱动程序专用数据区
- 长度为 hbalen 的按包的 HBA 驱动程序专用数据区

除以下成员外，必须将 scsi_pkt(9S) 结构成员（包括 pkt）初始化为零：

- pkt_scbp－状态完成
- pkt_cdbp－CDB
- pkt_ha_private－HBA 驱动程序专用数据
- pkt_private－目标驱动程序专用数据

这些成员是指向存储字段值的内存空间的指针，如下图所示。有关更多信息，请参阅第 316 页中的 “scsi_pkt 结构 (HBA)”。

图 18–5 scsi_pkt(9S) 结构指针



以下示例说明了 scsi_pkt 结构的分配和初始化。

示例 18–2 SCSI 包结构的 HBA 驱动程序初始化

```
static struct scsi_pkt                  *
isp_scsi_init_pkt(
    struct scsi_address    *ap,
    struct scsi_pkt        *pkt,
    struct buf             *bp,
    int                    cmdlen,
    int                    statuslen,
    int                    tgtlen,
    int                    flags,
    int                    (*callback)(),
    caddr_t                arg)
{
    struct isp_cmd         *sp;
    struct isp             *isp;
    struct scsi_pkt        *new_pkt;

    ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);

    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
    /*
     * First step of isp_scsi_init_pkt:  pkt allocation
     */
    if (pkt == NULL) {
        pkt = scsi_hba_pkt_alloc(isp->isp_dip, ap, cmdlen,
            statuslen, tgtlen, sizeof (struct isp_cmd),
```

示例 18–2 SCSI 包结构的 HBA 驱动程序初始化 （续）

```
            callback, arg);
        if (pkt == NULL) {
            return (NULL);
        }

        sp = (struct isp_cmd *)pkt->pkt_ha_private;
        /*
         * Initialize the new pkt
         */
        sp->cmd_pkt         = pkt;
        sp->cmd_flags       = 0;
        sp->cmd_scblen      = statuslen;
        sp->cmd_cdblen      = cmdlen;
        sp->cmd_dmahandle   = NULL;
        sp->cmd_ncookies    = 0;
        sp->cmd_cookie      = 0;
        sp->cmd_cookiecnt   = 0;
        sp->cmd_nwin        = 0;
        pkt->pkt_address    = *ap;
        pkt->pkt_comp       = (void (*)())NULL;
        pkt->pkt_flags      = 0;
        pkt->pkt_time       = 0;
        pkt->pkt_resid      = 0;
        pkt->pkt_statistics = 0;
        pkt->pkt_reason     = 0;
        new_pkt = pkt;
    } else {
        sp = (struct isp_cmd *)pkt->pkt_ha_private;
        new_pkt = NULL;
    }
    /*
     * Second step of isp_scsi_init_pkt:  dma allocation/move
     */
    if (bp && bp->b_bcount != 0) {
        if (sp->cmd_dmahandle == NULL) {
            if (isp_i_dma_alloc(isp, pkt, bp, flags, callback) == 0) {
                if (new_pkt) {
                    scsi_hba_pkt_free(ap, new_pkt);
                }
                return ((struct scsi_pkt *)NULL);
            }
        } else {
            ASSERT(new_pkt == NULL);
            if (isp_i_dma_move(isp, pkt, bp) == 0) {
                return ((struct scsi_pkt *)NULL);
            }
        }
    }
    return (pkt);
}
```

分配 DMA 资源

如果符合以下条件，则 tran_init_pkt(9E) 入口点必须分配用于数据传送的 DMA 资源：

- bp 不为 null。
- bp->b_bcount 不为零。
- 尚未为此 scsi_pkt(9S) 分配 DMA 资源。

HBA 驱动程序需要跟踪如何为特定命令分配 DMA 资源。按包的 HBA 驱动程序专用数据的标志位或 DMA 句柄可能会进行此分配。

使用 pkt 中的 PKT_DMA_PARTIAL 标志，目标驱动程序可以将数据传送按多个 SCSI 命令分类以适应整个请求。如果 HBA 硬件的分散/集中功能或系统 DMA 资源无法完成单个 SCSI 命令的请求，则此方法会非常有用。

使用 PKT_DMA_PARTIAL 标志，HBA 驱动程序可以设置 DDI_DMA_PARTIAL 标志。DDI_DMA_PARTIAL 标志有助于分配此 SCSI 命令的 DMA 资源。例如，ddi_dma_buf_bind_handle(9F) 命令可用于分配 DMA 资源。分配 DMA 资源时使用的 DMA 属性应准确说明针对 HBA 硬件执行 DMA 的能力设定的约束。如果系统只能为部分请求分配 DMA 资源，则 ddi_dma_buf_bind_handle(9F) 将返回 DDI_DMA_PARTIAL_MAP。

tran_init_pkt(9E) 入口点必须在字段 pkt_resid 中返回未为此传送分配的 DMA 资源量。

目标驱动程序可以请求 tran_init_pkt(9E) 同时为该 pkt 分配 scsi_pkt(9S) 结构和 DMA 资源。在这种情况下，如果 HBA 驱动程序无法分配 DMA 资源，则该驱动程序必须在返回前释放已分配的 scsi_pkt(9S)。scsi_pkt(9S) 必须通过调用 scsi_hba_pkt_free(9F) 进行释放。

目标驱动程序可能会首先分配 scsi_pkt(9S)，随后再为此 pkt 分配 DMA 资源。在这种情况下，如果 HBA 驱动程序无法分配 DMA 资源，则该驱动程序**决不能**释放 pkt。在这种情况下，目标驱动程序负责释放 pkt。

示例 18–3 HBA 驱动程序的 DMA 资源分配

```
static int
isp_i_dma_alloc(
    struct isp          *isp,
    struct scsi_pkt     *pkt,
    struct buf          *bp,
    int                 flags,
    int                 (*callback)())
{
    struct isp_cmd      *sp  = (struct isp_cmd *)pkt->pkt_ha_private;
    int                 dma_flags;
    ddi_dma_attr_t      tmp_dma_attr;
    int                 (*cb)(caddr_t);
    int                 i;

    ASSERT(callback == NULL_FUNC || callback == SLEEP_FUNC);

    if (bp->b_flags & B_READ) {
        sp->cmd_flags &= ~CFLAG_DMASEND;
        dma_flags = DDI_DMA_READ;
```

示例 18–3 HBA 驱动程序的 DMA 资源分配 （续）

```
} else {
    sp->cmd_flags |= CFLAG_DMASEND;
    dma_flags = DDI_DMA_WRITE;
}
if (flags & PKT_CONSISTENT) {
    sp->cmd_flags |= CFLAG_CMDIOPB;
    dma_flags |= DDI_DMA_CONSISTENT;
}
if (flags & PKT_DMA_PARTIAL) {
    dma_flags |= DDI_DMA_PARTIAL;
}

tmp_dma_attr = isp_dma_attr;
tmp_dma_attr.dma_attr_burstsizes = isp->isp_burst_size;

cb = (callback == NULL_FUNC) ? DDI_DMA_DONTWAIT : DDI_DMA_SLEEP;

if ((i = ddi_dma_alloc_handle(isp->isp_dip, &tmp_dma_attr,
  cb, 0, &sp->cmd_dmahandle)) != DDI_SUCCESS) {
    switch (i) {
      case DDI_DMA_BADATTR:
          bioerror(bp, EFAULT);
          return (0);
      case DDI_DMA_NORESOURCES:
          bioerror(bp, 0);
          return (0);
    }
}

i = ddi_dma_buf_bind_handle(sp->cmd_dmahandle, bp, dma_flags,
cb, 0, &sp->cmd_dmacookies[0], &sp->cmd_ncookies);

switch (i) {
  case DDI_DMA_PARTIAL_MAP:
      if (ddi_dma_numwin(sp->cmd_dmahandle, &sp->cmd_nwin) == DDI_FAILURE) {
          cmn_err(CE_PANIC, "ddi_dma_numwin() failed\n");
      }

      if (ddi_dma_getwin(sp->cmd_dmahandle, sp->cmd_curwin,
        &sp->cmd_dma_offset, &sp->cmd_dma_len, &sp->cmd_dmacookies[0],
        &sp->cmd_ncookies) == DDI_FAILURE) {
          cmn_err(CE_PANIC, "ddi_dma_getwin() failed\n");
      }
      goto get_dma_cookies;

  case DDI_DMA_MAPPED:
      sp->cmd_nwin = 1;
      sp->cmd_dma_len = 0;
      sp->cmd_dma_offset = 0;

  get_dma_cookies:
      i = 0;
      sp->cmd_dmacount = 0;
      for (;;) {
          sp->cmd_dmacount += sp->cmd_dmacookies[i++].dmac_size;
          if (i == ISP_NDATASEGS || i == sp->cmd_ncookies)
```

示例 18–3 HBA 驱动程序的 DMA 资源分配 （续）

```
                break;
            ddi_dma_nextcookie(sp->cmd_dmahandle,
            &sp->cmd_dmacookies[i]);
        }
        sp->cmd_cookie = i;
        sp->cmd_cookiecnt = i;
        sp->cmd_flags |= CFLAG_DMAVALID;
        pkt->pkt_resid = bp->b_bcount - sp->cmd_dmacount;
        return (1);

    case DDI_DMA_NORESOURCES:
        bioerror(bp, 0);
        break;

    case DDI_DMA_NOMAPPING:
        bioerror(bp, EFAULT);
        break;

    case DDI_DMA_TOOBIG:
        bioerror(bp, EINVAL);
        break;

    case DDI_DMA_INUSE:
        cmn_err(CE_PANIC, "ddi_dma_buf_bind_handle:"
          " DDI_DMA_INUSE impossible\n");

    default:
        cmn_err(CE_PANIC, "ddi_dma_buf_bind_handle:"
          " 0x%x impossible\n", i);
  }
  ddi_dma_free_handle(&sp->cmd_dmahandle);
  sp->cmd_dmahandle = NULL;
  sp->cmd_flags &= ~CFLAG_DMAVALID;
  return (0);
}
```

重新分配用于数据传送的 DMA 资源

对于先前分配的包含待传送数据的包，tran_init_pkt(9E) 入口点在满足以下条件时必须重新分配 DMA 资源：

- 已分配部分 DMA 资源。
- 先前调用 tran_init_pkt(9E) 时返回了非零的 pkt_resid。
- bp 不为 null。
- bp->b_bcount 不为零。

重新分配下一个传送部分的 DMA 资源时，tran_init_pkt(9E) 必须在字段 pkt_resid 中返回未为此传送分配的 DMA 资源量。

如果在尝试移动 DMA 资源时出现错误，则 tran_init_pkt(9E) 决不能释放 scsi_pkt(9S)。在这种情况下，目标驱动程序负责释放包。

如果回调参数为 NULL_FUNC，则 tran_init_pkt(9E) 入口点决不能休眠或调用可能会休眠的任何函数。如果回调参数为 SLEEP_FUNC 并且资源不会立即可用，则 tran_init_pkt(9E) 入口点会休眠。除非无法满足请求，否则 tran_init_pkt() 将休眠，直到资源可用为止。

示例 18–4 HBA 驱动程序的 DMA 资源重新分配

```
static int
isp_i_dma_move(
    struct isp         *isp,
    struct scsi_pkt    *pkt,
    struct buf         *bp)
{
    struct isp_cmd     *sp  = (struct isp_cmd *)pkt->pkt_ha_private;
    int                i;

    ASSERT(sp->cmd_flags & CFLAG_COMPLETED);
    sp->cmd_flags &= ~CFLAG_COMPLETED;
    /*
     * If there are no more cookies remaining in this window,
     * must move to the next window first.
     */
    if (sp->cmd_cookie == sp->cmd_ncookies) {
        /*
         * For small pkts, leave things where they are
         */
        if (sp->cmd_curwin == sp->cmd_nwin && sp->cmd_nwin == 1)
            return (1);
        /*
         * At last window, cannot move
         */
        if (++sp->cmd_curwin >= sp->cmd_nwin)
            return (0);
        if (ddi_dma_getwin(sp->cmd_dmahandle, sp->cmd_curwin,
          &sp->cmd_dma_offset, &sp->cmd_dma_len,
          &sp->cmd_dmacookies[0], &sp->cmd_ncookies) == DDI_FAILURE)
            return (0);
        sp->cmd_cookie = 0;
    } else {
        /*
         * Still more cookies in this window - get the next one
         */
        ddi_dma_nextcookie(sp->cmd_dmahandle, &sp->cmd_dmacookies[0]);
    }
    /*
     * Get remaining cookies in this window, up to our maximum
     */
    i = 0;
    for (;;) {
        sp->cmd_dmacount += sp->cmd_dmacookies[i++].dmac_size;
        sp->cmd_cookie++;
        if (i == ISP_NDATASEGS || sp->cmd_cookie == sp->cmd_ncookies)
            break;
        ddi_dma_nextcookie(sp->cmd_dmahandle, &sp->cmd_dmacookies[i]);
    }
    sp->cmd_cookiecnt = i;
    pkt->pkt_resid = bp->b_bcount - sp->cmd_dmacount;
```

示例 18–4 HBA 驱动程序的 DMA 资源重新分配 （续）

```
    return (1);
}
```

tran_destroy_pkt() 入口点

tran_destroy_pkt(9E) 入口点是用于取消分配 scsi_pkt(9S) 结构的 HBA 驱动程序函数。目标驱动程序调用 scsi_destroy_pkt(9F) 时，将会调用 tran_destroy_pkt() 入口点。

tran_destroy_pkt() 入口点必须释放已为包分配的所有 DMA 资源。如果释放了 DMA 资源并且所有高速缓存的数据在完成传送后仍然保留，则会进行隐式 DMA 同步。tran_destroy_pkt() 入口点通过调用 scsi_hba_pkt_free(9F) 释放 SCSI 包。

示例 18–5 HBA 驱动程序 tran_destroy_pkt (9E) 入口点

```
static void
isp_scsi_destroy_pkt(
    struct scsi_address    *ap,
    struct scsi_pkt    *pkt)
{
    struct isp_cmd *sp = (struct isp_cmd *)pkt->pkt_ha_private;
    /*
     * Free the DMA, if any
     */
    if (sp->cmd_flags & CFLAG_DMAVALID) {
        sp->cmd_flags &= ~CFLAG_DMAVALID;
        (void) ddi_dma_unbind_handle(sp->cmd_dmahandle);
        ddi_dma_free_handle(&sp->cmd_dmahandle);
        sp->cmd_dmahandle = NULL;
    }
    /*
     * Free the pkt
     */
    scsi_hba_pkt_free(ap, pkt);
}
```

tran_sync_pkt() 入口点

tran_sync_pkt(9E) 入口点可在 DMA 传送前后同步为 scsi_pkt(9S) 结构分配的 DMA 对象。目标驱动程序调用 scsi_sync_pkt(9F) 时，将会调用 tran_sync_pkt() 入口点。

如果数据传送方向是从设备到内存的 DMA 读取，则 tran_sync_pkt() 必须同步 CPU 的数据视图。如果数据传送方向是从内存到设备的 DMA 写入，则 tran_sync_pkt() 必须同步设备的数据视图。

示例 18–6 HBA 驱动程序 tran_sync_pkt (9E) 入口点

```
static void
isp_scsi_sync_pkt(
    struct scsi_address    *ap,
```

示例 18–6 HBA 驱动程序 tran_sync_pkt (9E) 入口点 （续）

```
    struct scsi_pkt        *pkt)
{
    struct isp_cmd *sp = (struct isp_cmd *)pkt->pkt_ha_private;

    if (sp->cmd_flags & CFLAG_DMAVALID) {
        (void)ddi_dma_sync(sp->cmd_dmahandle, sp->cmd_dma_offset,
        sp->cmd_dma_len,
        (sp->cmd_flags & CFLAG_DMASEND) ?
        DDI_DMA_SYNC_FORDEV : DDI_DMA_SYNC_FORCPU);
    }
}
```

tran_dmafree() 入口点

tran_dmafree(9E) 入口点可取消分配已为 scsi_pkt(9S) 结构分配的 DMA 资源。目标驱动程序调用 scsi_dmafree(9F) 时，将会调用 tran_dmafree() 入口点。

tran_dmafree() 必须仅释放为 scsi_pkt(9S) 结构分配的 DMA 资源，而不释放 scsi_pkt(9S) 本身。释放 DMA 资源时，将隐式执行 DMA 同步。

注 – scsi_pkt(9S) 在单独请求 tran_destroy_pkt(9E) 时释放。由于 tran_destroy_pkt() 还必须释放 DMA 资源，因此 HBA 驱动程序必须准确记录 scsi_pkt() 结构是否分配了 DMA 资源。

示例 18–7 HBA 驱动程序 tran_dmafree (9E) 入口点

```
static void
isp_scsi_dmafree(
    struct scsi_address    *ap,
    struct scsi_pkt        *pkt)
{
    struct isp_cmd    *sp = (struct isp_cmd *)pkt->pkt_ha_private;

    if (sp->cmd_flags & CFLAG_DMAVALID) {
        sp->cmd_flags &= ~CFLAG_DMAVALID;
        (void)ddi_dma_unbind_handle(sp->cmd_dmahandle);
        ddi_dma_free_handle(&sp->cmd_dmahandle);
        sp->cmd_dmahandle = NULL;
    }
}
```

命令传输

在命令传输过程中，HBA 驱动程序将执行以下步骤：

1. 接受来自目标驱动程序的命令。
2. 向设备硬件发出命令。
3. 传送出现的所有中断。

4. 管理超时。

tran_start() 入口点

调用 SCSI HBA 驱动程序的 tran_start(9E) 入口点可将 SCSI 命令传输到指定地址的目标。在目标驱动程序通过 HBA 驱动程序的 tran_init_pkt(9E) 入口点分配的 scsi_pkt(9S) 结构中，对 SCSI 命令进行了完整描述。如果该命令涉及数据传送，则还必须为 scsi_pkt(9S) 结构分配 DMA 资源。

目标驱动程序调用 scsi_transport(9F) 时，将会调用 tran_start() 入口点。

tran_start() 应执行基本错误检查以及命令要求的任何初始化操作。scsi_pkt(9S) 结构的 pkt_flags 字段中的 FLAG_NOINTR 标志会影响 tran_start() 的行为。如果未设置 FLAG_NOINTR，则 tran_start() 必须将命令排队以在硬件上执行并立即返回。完成命令后，HBA 驱动程序应调用 pkt 完成例程。

如果设置了 FLAG_NOINTR，则 HBA 驱动程序不会调用 pkt 完成例程。

以下示例说明如何处理 tran_start(9E) 入口点。ISP 硬件按目标设备提供了队列。对于只能管理一个活动的未完成命令的设备，驱动程序通常需要管理按目标的队列。然后，驱动程序会在完成当前命令后以循环方式启动新命令。

示例 18–8 HBA 驱动程序 tran_start (9E) 入口点

```
static int
isp_scsi_start(
    struct scsi_address    *ap,
    struct scsi_pkt        *pkt)
{
    struct isp_cmd         *sp;
    struct isp             *isp;
    struct isp_request     *req;
    u_long                 cur_lbolt;
    int                    xfercount;
    int                    rval = TRAN_ACCEPT;
    int                    i;

    sp = (struct isp_cmd *)pkt->pkt_ha_private;
    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;

    sp->cmd_flags = (sp->cmd_flags & ~CFLAG_TRANFLAG) |
                CFLAG_IN_TRANSPORT;
    pkt->pkt_reason = CMD_CMPLT;
    /*
     * set up request in cmd_isp_request area so it is ready to
     * go once we have the request mutex
     */
    req = &sp->cmd_isp_request;

    req->req_header.cq_entry_type = CQ_TYPE_REQUEST;
    req->req_header.cq_entry_count = 1;
    req->req_header.cq_flags        = 0;
    req->req_header.cq_seqno = 0;
```

示例 18–8 HBA 驱动程序 tran_start (9E) 入口点 （续）

```
    req->req_reserved = 0;
    req->req_token = (opaque_t)sp;
    req->req_target = TGT(sp);
    req->req_lun_trn = LUN(sp);
    req->req_time = pkt->pkt_time;
    ISP_SET_PKT_FLAGS(pkt->pkt_flags, req->req_flags);
    /*
     * Set up data segments for dma transfers.
     */
    if (sp->cmd_flags & CFLAG_DMAVALID) {
        if (sp->cmd_flags & CFLAG_CMDIOPB) {
            (void) ddi_dma_sync(sp->cmd_dmahandle,
            sp->cmd_dma_offset, sp->cmd_dma_len,
            DDI_DMA_SYNC_FORDEV);
        }

        ASSERT(sp->cmd_cookiecnt > 0 &&
            sp->cmd_cookiecnt <= ISP_NDATASEGS);

        xfercount = 0;
        req->req_seg_count = sp->cmd_cookiecnt;
        for (i = 0; i < sp->cmd_cookiecnt; i++) {
            req->req_dataseg[i].d_count =
            sp->cmd_dmacookies[i].dmac_size;
            req->req_dataseg[i].d_base =
            sp->cmd_dmacookies[i].dmac_address;
            xfercount +=
            sp->cmd_dmacookies[i].dmac_size;
        }

        for (; i < ISP_NDATASEGS; i++) {
            req->req_dataseg[i].d_count = 0;
            req->req_dataseg[i].d_base = 0;
        }

        pkt->pkt_resid = xfercount;

        if (sp->cmd_flags & CFLAG_DMASEND) {
            req->req_flags |= ISP_REQ_FLAG_DATA_WRITE;
        } else {
            req->req_flags |= ISP_REQ_FLAG_DATA_READ;
        }
    } else {
        req->req_seg_count = 0;
        req->req_dataseg[0].d_count = 0;
    }
    /*
     * Set up cdb in the request
     */
    req->req_cdblen = sp->cmd_cdblen;
    bcopy((caddr_t)pkt->pkt_cdbp, (caddr_t)req->req_cdb,
    sp->cmd_cdblen);
    /*
     * Start the cmd.  If NO_INTR, must poll for cmd completion.
     */
    if ((pkt->pkt_flags & FLAG_NOINTR) == 0) {
```

示例 18–8 HBA 驱动程序 tran_start (9E) 入口点 （续）

```
        mutex_enter(ISP_REQ_MUTEX(isp));
        rval = isp_i_start_cmd(isp, sp);
        mutex_exit(ISP_REQ_MUTEX(isp));
    } else {
        rval = isp_i_polled_cmd_start(isp, sp);
    }
    return (rval);
}
```

中断处理程序和命令完成

中断处理程序必须检查设备状态，以确保设备正在生成相关中断。另外，中断处理程序还必须检查出现的全部错误，并传送设备生成的所有中断。

如果传送了数据，则应检查硬件以确定实际传送的数据量。scsi_pkt(9S) 结构中的 pkt_resid 字段应该设置为剩余未传送的数据量。

通过 tran_init_pkt(9E) 分配 DMA 资源时，使用 PKT_CONSISTENT 标志标记的命令需要特殊处理。HBA 驱动程序必须确保在执行目标驱动程序的命令完成回调之前，正确同步针对该命令的数据传送。

完成命令后，需要按照以下两个要求执行操作：

- 如果已将新命令排入队列，请尽快在硬件上启动该命令。
- 调用命令完成回调。目标驱动程序已在 scsi_pkt(9S) 结构中设置了回调，用于通知目标驱动程序完成命令的时间。

如有可能，在调用 PKT_COMP 命令完成回调之前，请在硬件上启动新命令。该命令完成处理可能需要大量时间。通常，目标驱动程序会调用函数（如 biodone(9F) 和可能会调用的 scsi_transport(9F)）来启动新命令。

如果此中断是由该驱动程序请求的，则中断处理程序必须返回 DDI_INTR_CLAIMED。否则，处理程序会返回 DDI_INTR_UNCLAIMED。

以下示例说明了 SCSI HBA isp 驱动程序的中断处理程序。如果 attach(9E) 中添加了中断处理程序，则应设置 caddr_t 参数。此参数通常是一个指向按实例分配的状态结构的指针。

示例 18–9 HBA 驱动程序中断处理程序

```
static u_int
isp_intr(caddr_t arg)
{
    struct isp_cmd          *sp;
    struct isp_cmd          *head, *tail;
    u_short                 response_in;
    struct isp_response     *resp;
    struct isp              *isp = (struct isp *)arg;
    struct isp_slot         *isp_slot;
```

示例 18–9 HBA 驱动程序中断处理程序 （续）

```
    int                     n;

    if (ISP_INT_PENDING(isp) == 0) {
        return (DDI_INTR_UNCLAIMED);
    }

    do {
again:
        /*
         * head list collects completed packets for callback later
         */
        head = tail = NULL;
        /*
         * Assume no mailbox events (e.g., mailbox cmds, asynch
         * events, and isp dma errors) as common case.
         */
        if (ISP_CHECK_SEMAPHORE_LOCK(isp) == 0) {
            mutex_enter(ISP_RESP_MUTEX(isp));
            /*
             * Loop through completion response queue and post
             * completed pkts.  Check response queue again
             * afterwards in case there are more.
             */
            isp->isp_response_in =
            response_in = ISP_GET_RESPONSE_IN(isp);
            /*
             * Calculate the number of requests in the queue
             */
            n = response_in - isp->isp_response_out;
            if (n < 0) {
                n = ISP_MAX_REQUESTS -
                isp->isp_response_out + response_in;
            }
            while (n-- > 0) {
                ISP_GET_NEXT_RESPONSE_OUT(isp, resp);
                sp = (struct isp_cmd *)resp->resp_token;
                /*
                 * Copy over response packet in sp
                 */
                isp_i_get_response(isp, resp, sp);
            }
            if (head) {
                tail->cmd_forw = sp;
                tail = sp;
                tail->cmd_forw = NULL;
            } else {
                tail = head = sp;
                sp->cmd_forw = NULL;
            }
            ISP_SET_RESPONSE_OUT(isp);
            ISP_CLEAR_RISC_INT(isp);
            mutex_exit(ISP_RESP_MUTEX(isp));

            if (head) {
                isp_i_call_pkt_comp(isp, head);
            }
```

示例 18–9 HBA 驱动程序中断处理程序 （续）

```
        } else {
            if (isp_i_handle_mbox_cmd(isp) != ISP_AEN_SUCCESS) {
                return (DDI_INTR_CLAIMED);
            }
            /*
             * if there was a reset then check the response
             * queue again
             */
            goto again;
        }

    } while (ISP_INT_PENDING(isp));

    return (DDI_INTR_CLAIMED);
}

static void
isp_i_call_pkt_comp(
    struct isp              *isp,
    struct isp_cmd          *head)
{
    struct isp              *isp;
    struct isp_cmd          *sp;
    struct scsi_pkt         *pkt;
    struct isp_response     *resp;
    u_char                  status;

    while (head) {
        sp = head;
        pkt = sp->cmd_pkt;
        head = sp->cmd_forw;

        ASSERT(sp->cmd_flags & CFLAG_FINISHED);

        resp = &sp->cmd_isp_response;

        pkt->pkt_scbp[0] = (u_char)resp->resp_scb;
        pkt->pkt_state = ISP_GET_PKT_STATE(resp->resp_state);
        pkt->pkt_statistics = (u_long)
            ISP_GET_PKT_STATS(resp->resp_status_flags);
        pkt->pkt_resid = (long)resp->resp_resid;
        /*
         * If data was xferred and this is a consistent pkt,
         * do a dma sync
         */
        if ((sp->cmd_flags & CFLAG_CMDIOPB) &&
            (pkt->pkt_state & STATE_XFERRED_DATA)) {
                (void) ddi_dma_sync(sp->cmd_dmahandle,
                sp->cmd_dma_offset, sp->cmd_dma_len,
                DDI_DMA_SYNC_FORCPU);
        }

        sp->cmd_flags = (sp->cmd_flags & ~CFLAG_IN_TRANSPORT) |
            CFLAG_COMPLETED;
        /*
         * Call packet completion routine if FLAG_NOINTR is not set.
```

示例 18–9 HBA 驱动程序中断处理程序 （续）

```
         */
        if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
            pkt->pkt_comp) {
                (*pkt->pkt_comp)(pkt);
        }
    }
}
```

超时处理程序

HBA 驱动程序负责强制执行超时设置。除非在 scsi_pkt(9S) 结构中指定了零超时，否则必须在指定时间内完成命令。

如果命令超时，则 HBA 驱动程序应将 scsi_pkt(9S) 标记为 pkt_reason=CMD_TIMEOUT，而且将 pkt_statistics 的值设置为与 STAT_TIMEOUT 进行或运算所得的值。另外，HBA 驱动程序还应尝试恢复目标和总线。如果恢复能够成功执行，则驱动程序应使用 pkt_statistics 与 STAT_BUS_RESET 或 STAT_DEV_RESET 进行或运算所得的值标记 scsi_pkt(9S)。

完成恢复尝试后，HBA 驱动程序应调用命令完成回调。

注 – 如果恢复不成功或未尝试恢复，则目标驱动程序可能会通过调用 scsi_reset(9F) 尝试从超时中恢复。

ISP 硬件直接管理命令超时，并会返回超时命令的必需状态。isp 样例驱动程序的超时处理程序每 60 秒才检查一次活动命令的超时状态。

isp 样例驱动程序使用 timeout(9F) 功能来安排内核每 60 秒调用一次超时处理程序。caddr_t 参数是在执行 attach(9E) 期间初始化超时值时设置的参数。在这种情况下，caddr_t 参数是一个指向按驱动程序实例分配的状态结构的指针。

如果 ISP 硬件未将超时命令作为超时项返回，则表明出现了问题。该硬件将无法正常工作并需要重置。

功能管理

以下各节讨论了功能管理。

tran_getcap() 入口点

SCSI HBA 驱动程序的 tran_getcap(9E) 入口点是由 scsi_ifgetcap(9F) 调用的。目标驱动程序调用 scsi_ifgetcap() 可确定 SCSA 定义的一组功能中其中一个的当前值。

目标驱动程序可以通过将 whom 参数设置为非零值来请求特定目标的功能的当前设置。whom 值为零表明请求 SCSI 总线或适配器硬件的一般功能的当前设置。

对于未定义的功能或所请求的功能的当前值，tran_getcap() 入口点应返回 -1。

HBA 驱动程序可以使用函数 scsi_hba_lookup_capstr(9F) 来比较功能字符串和已定义功能的标准集。

示例 18–10 HBA 驱动程序 tran_getcap (9E) 入口点

```
static int
isp_scsi_getcap(
    struct scsi_address    *ap,
    char                   *cap,
    int                    whom)
{
    struct isp             *isp;
    int                    rval = 0;
    u_char                 tgt = ap->a_target;
    /*
      * We don't allow getting capabilities for other targets
    */
    if (cap == NULL || whom  == 0) {
        return (-1);
    }
    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
    ISP_MUTEX_ENTER(isp);

    switch (scsi_hba_lookup_capstr(cap)) {
      case SCSI_CAP_DMA_MAX:
          rval = 1 << 24; /* Limit to 16MB max transfer */
          break;
      case SCSI_CAP_MSG_OUT:
          rval = 1;
          break;
      case SCSI_CAP_DISCONNECT:
          if ((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_DR) == 0) {
              break;
          } else if (
              (isp->isp_cap[tgt] & ISP_CAP_DISCONNECT) == 0) {
                  break;
          }
          rval = 1;
          break;
      case SCSI_CAP_SYNCHRONOUS:
          if ((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_SYNC) == 0) {
              break;
          } else if (
              (isp->isp_cap[tgt] & ISP_CAP_SYNC) == 0) {
                  break;
          }
          rval = 1;
          break;
      case SCSI_CAP_WIDE_XFER:
          if ((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_WIDE) == 0) {
              break;
          } else if (
              (isp->isp_cap[tgt] & ISP_CAP_WIDE) == 0) {
```

示例 18–10 HBA 驱动程序 tran_getcap (9E) 入口点 （续）

```
                break;
            }
            rval = 1;
            break;
        case SCSI_CAP_TAGGED_QING:
            if ((isp->isp_target_scsi_options[tgt] &
              SCSI_OPTIONS_DR) == 0 ||
              (isp->isp_target_scsi_options[tgt] &
              SCSI_OPTIONS_TAG) == 0) {
                break;
            } else if (
                (isp->isp_cap[tgt] & ISP_CAP_TAG) == 0) {
                    break;
            }
            rval = 1;
            break;
        case SCSI_CAP_UNTAGGED_QING:
            rval = 1;
            break;
        case SCSI_CAP_PARITY:
            if (isp->isp_target_scsi_options[tgt] &
              SCSI_OPTIONS_PARITY) {
                rval = 1;
            }
            break;
        case SCSI_CAP_INITIATOR_ID:
            rval = isp->isp_initiator_id;
            break;
        case SCSI_CAP_ARQ:
            if (isp->isp_cap[tgt] & ISP_CAP_AUTOSENSE) {
                rval = 1;
            }
            break;
        case SCSI_CAP_LINKED_CMDS:
            break;
        case SCSI_CAP_RESET_NOTIFICATION:
           rval = 1;
            break;
        case SCSI_CAP_GEOMETRY:
            rval = (64 << 16) | 32;
            break;
        default:
            rval = -1;
            break;
    }
    ISP_MUTEX_EXIT(isp);
    return (rval);
}
```

tran_setcap() 入口点

SCSI HBA 驱动程序的 tran_setcap(9E) 入口点通过 scsi_ifsetcap(9F) 进行调用。目标驱动程序调用 scsi_ifsetcap() 可更改 SCSA 定义的一组功能中其中一个的当前值。

目标驱动程序可能会通过将 whom 参数设置为非零值来请求为特定目标设置新值。whom 值为零通常表明请求为 SCSI 总线或适配器硬件设置新值。

tran_setcap() 应相应地返回以下值：

- -1，如果功能未定义
- 0，如果 HBA 驱动程序无法将功能设置为请求值
- 1，如果 HBA 驱动程序可以将功能设置为请求值

HBA 驱动程序可以使用函数 scsi_hba_lookup_capstr(9F) 来比较功能字符串和已定义功能的标准集。

示例 18–11 HBA 驱动程序 tran_setcap (9E) 入口点

```
static int
isp_scsi_setcap(
    struct scsi_address    *ap,
    char                   *cap,
    int                    value,
    int                    whom)
{
    struct isp             *isp;
    int                    rval = 0;
    u_char                 tgt = ap->a_target;
    int                    update_isp = 0;
    /*
     * We don't allow setting capabilities for other targets
     */
    if (cap == NULL || whom == 0) {
        return (-1);
    }

    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
    ISP_MUTEX_ENTER(isp);

    switch (scsi_hba_lookup_capstr(cap)) {
      case SCSI_CAP_DMA_MAX:
      case SCSI_CAP_MSG_OUT:
      case SCSI_CAP_PARITY:
      case SCSI_CAP_UNTAGGED_QING:
      case SCSI_CAP_LINKED_CMDS:
      case SCSI_CAP_RESET_NOTIFICATION:
          /*
           * None of these are settable through
           * the capability interface.
           */
          break;
      case SCSI_CAP_DISCONNECT:
          if ((isp->isp_target_scsi_options[tgt] &
            SCSI_OPTIONS_DR) == 0) {
              break;
          } else {
              if (value) {
                  isp->isp_cap[tgt] |= ISP_CAP_DISCONNECT;
              } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_DISCONNECT;
```

示例 18–11 HBA 驱动程序 tran_setcap (9E) 入口点 （续）

```
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_SYNCHRONOUS:
        if ((isp->isp_target_scsi_options[tgt] &
          SCSI_OPTIONS_SYNC) == 0) {
              break;
        } else {
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_SYNC;
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_SYNC;
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_TAGGED_QING:
        if ((isp->isp_target_scsi_options[tgt] &
          SCSI_OPTIONS_DR) == 0 ||
          (isp->isp_target_scsi_options[tgt] &
          SCSI_OPTIONS_TAG) == 0) {
              break;
        } else {
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_TAG;
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_TAG;
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_WIDE_XFER:
        if ((isp->isp_target_scsi_options[tgt] &
          SCSI_OPTIONS_WIDE) == 0) {
              break;
        } else {
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_WIDE;
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_WIDE;
            }
        }
        rval = 1;
        break;
    case SCSI_CAP_INITIATOR_ID:
        if (value < N_ISP_TARGETS_WIDE) {
            struct isp_mbox_cmd mbox_cmd;
            isp->isp_initiator_id = (u_short) value;
            /*
             * set Initiator SCSI ID
             */
            isp_i_mbox_cmd_init(isp, &mbox_cmd, 2, 2,
              ISP_MBOX_CMD_SET_SCSI_ID,
              isp->isp_initiator_id,
              0, 0, 0, 0);
```

示例 18–11 HBA 驱动程序 tran_setcap (9E) 入口点 （续）

```
                if (isp_i_mbox_cmd_start(isp, &mbox_cmd) == 0) {
                    rval = 1;
                }
            }
            break;
        case SCSI_CAP_ARQ:
            if (value) {
                isp->isp_cap[tgt] |= ISP_CAP_AUTOSENSE;
            } else {
                isp->isp_cap[tgt] &= ~ISP_CAP_AUTOSENSE;
            }
            rval = 1;
            break;
        default:
            rval = -1;
            break;
    }
    ISP_MUTEX_EXIT(isp);
    return (rval);
}
```

中止和重置管理

以下各节讨论了 SCSI HBA 的中止入口点和重置入口点。

tran_abort() 入口点

调用 SCSI HBA 驱动程序的 tran_abort(9E) 入口点可中止当前正在传输给特定目标的所有命令。目标驱动程序调用 scsi_abort(9F) 时，将会调用此入口点。

tran_abort() 入口点会尝试中止 pkt 参数表示的命令。如果 pkt 参数为 NULL，则 tran_abort() 会尝试中止传输层中针对特定目标或逻辑单元的所有未完成命令。

每个已成功中止的命令都必须标记为 pkt_reason CMD_ABORTED 以及 pkt_statistics 与 STAT_ABORTED 进行或运算所得的值。

tran_reset() 入口点

调用 SCSI HBA 驱动程序的 tran_reset(9E) 入口点可重置 SCSI 总线或特定的 SCSI 目标设备。目标驱动程序调用 scsi_reset(9F) 时，将会调用此入口点。

如果级别为 RESET_ALL()，则 tran_reset 入口点必须重置 SCSI 总线。如果级别为 RESET_TARGET，则仅有特定目标或逻辑单元必须重置。

受重置影响的活动命令必须带有 pkt_reason CMD_RESET 标记。重置类型可确定应使用 STAT_BUS_RESET 还是使用 STAT_DEV_RESET 与 pkt_statistics 进行或运算。

在目标上尚未处于活动状态的传输层中的命令必须标记为 pkt_reason CMD_RESET 以及 pkt_statistics 与 STAT_ABORTED 进行或运算所得的值。

tran_bus_reset() 入口点

tran_bus_reset(9E) 必须重置 SCSI 总线而不重置目标。

```
#include <sys/scsi/scsi.h>

int tran_bus_reset(dev_info_t *hba-dip, int level);
```

其中：

**hba-dip* 与 SCSI HBA 关联的指针

level 必须设置为 RESET_BUS，以便仅重置 SCSI 总线而不重置目标。

执行 HBA 驱动程序的 attach(9E) 的过程中，应初始化 scsi_hba_tran(9S) 结构中的 tran_bus_reset() 向量。该向量应指向用户启动总线重置时将调用的 HBA 入口点。

实现特定于硬件。如果 HBA 驱动程序无法在不影响目标的情况下重置 SCSI 总线，则驱动程序将无法执行 RESET_BUS 或不会初始化此向量。

tran_reset_notify() 入口点

重置 SCSI 总线时，请使用 tran_reset_notify(9E) 入口点。此函数将请求 SCSI HBA 驱动程序通过回调来通知目标驱动程序。

示例 18–12 HBA 驱动程序 tran_reset_notify (9E) 入口点

```
isp_scsi_reset_notify(
    struct scsi_address    *ap,
    int                    flag,
    void                   (*callback)(caddr_t),
    caddr_t                arg)
{
    struct isp                          *isp;
    struct isp_reset_notify_entry       *p, *beforep;
    int                                 rval = DDI_FAILURE;

    isp = (struct isp *)ap->a_hba_tran->tran_hba_private;
    mutex_enter(ISP_REQ_MUTEX(isp));
    /*
     * Try to find an existing entry for this target
     */
    p = isp->isp_reset_notify_listf;
    beforep = NULL;

    while (p) {
        if (p->ap == ap)
            break;
        beforep = p;
        p = p->next;
    }

    if ((flag & SCSI_RESET_CANCEL) && (p != NULL)) {
        if (beforep == NULL) {
```

示例 18–12 HBA 驱动程序 tran_reset_notify (9E) 入口点 （续）

```
            isp->isp_reset_notify_listf = p->next;
        } else {
            beforep->next = p->next;
        }
        kmem_free((caddr_t)p, sizeof (struct isp_reset_notify_entry));
        rval = DDI_SUCCESS;
    } else if ((flag & SCSI_RESET_NOTIFY) && (p == NULL)) {
        p = kmem_zalloc(sizeof (struct isp_reset_notify_entry),
          KM_SLEEP);
        p->ap = ap;
        p->callback = callback;
        p->arg = arg;
        p->next = isp->isp_reset_notify_listf;
        isp->isp_reset_notify_listf = p;
        rval = DDI_SUCCESS;
    }
    mutex_exit(ISP_REQ_MUTEX(isp));
    return (rval);
}
```

动态重新配置

要支持最少的一组热插拔操作，驱动程序可能需要实现对总线**停止**、总线**取消停止**和总线**重置**的支持。scsi_hba_tran(9S) 结构支持这些操作。如果硬件不要求停止、取消停止或重置等操作，则无需对驱动程序进行任何更改。

scsi_hba_tran 结构包含以下字段：

```
int (*tran_quiesce)(dev_info_t *hba-dip);
int (*tran_unquiesce)(dev_info_t *hba-dip);
int (*tran_bus_reset)(dev_info_t *hba-dip, int level);
```

这些接口用于停止和取消停止 SCSI 总线。

```
#include <sys/scsi/scsi.h>

int prefixtran_quiesce(dev_info_t *hba-dip);
int prefixtran_unquiesce(dev_info_t *hba-dip);
```

tran_quiesce(9E) 和 tran_unquiesce(9E) 用于不能热插拔的 SCSI 设备。HBA 驱动程序必须实现这些函数才能支持动态重新配置 (dynamic reconfiguration, DR)。

在执行 attach(9E) 的过程中，应将 scsi_hba_tran(9S) 结构中的 tran_quiesce() 和 tran_unquiesce() 向量初始化为指向 HBA 入口点。用户启动停止和取消停止操作时，将会调用这些函数。

tran_quiesce() 入口点用于在重新配置连接到 SCSI 总线的设备之前和期间停止 SCSI 总线上的所有活动。完成重新配置操作后，SCSA 框架可调用 tran_unquiesce() 入口点来恢复 SCSI 总线上的活动。

返回成功信息之前，HBA 驱动程序需要处理 tran_quiesce()，方法是等待所有未完成的命令完成。驱动程序停止总线后，必须将任何新的 I/O 请求排入队列，直到 SCSA 框架调用对应的 tran_unquiesce() 入口点为止。

HBA 驱动程序通过启动队列中的所有目标驱动程序 I/O 请求来处理对 tran_unquiesce() 的调用。

SCSI HBA 驱动程序特定问题

本节介绍了特定于 SCSI HBA 驱动程序的问题。

安装 HBA 驱动程序

SCSI HBA 驱动程序的安装方式与叶驱动程序类似。请参见第 22 章，编译、装入、打包和测试驱动程序。差别在于 add_drv(1M) 命令必须将驱动程序类指定为 SCSI，如：

```
# add_drv -m" * 0666 root root" -i'"pci1077,1020"' -c scsi isp
```

HBA 配置属性

连接 HBA 设备实例时，scsi_hba_attach_setup(9F) 会为该 HBA 实例创建许多 SCSI 配置属性。仅当没有现有同名属性附加到 HBA 实例时，才会创建特定属性。此限制可避免覆盖 HBA 配置文件中的任何缺省属性值。

HBA 驱动程序必须使用 ddi_prop_get_int(9F) 检索每个属性。然后，HBA 驱动程序会修改或接受这些属性的缺省值来配置其特定操作。

scsi-reset-delay 属性

scsi-reset-delay 属性是一个整数，用于指定由 SCSI 总线或 SCSI 设备造成的重置延迟的恢复时间（以毫秒为单位）。

scsi-options 属性

scsi-options 属性是一个整数，用于通过单独定义的位来指定多个选项：

- SCSI_OPTIONS_DR (0x008)－如果未设置，HBA 将不会授予目标设备断开连接权限。

- SCSI_OPTIONS_LINK (0x010) – 如果未设置，HBA 将不会启用链接的命令。
- SCSI_OPTIONS_SYNC (0x020) – 如果未设置，则 HBA 驱动程序决不能协商同步数据传送。驱动程序将拒绝目标启动的协商同步数据传送的任何尝试。
- SCSI_OPTIONS_PARITY (0x040) – 如果未设置，HBA 将会运行 SCSI 总线，而不进行奇偶校验。
- SCSI_OPTIONS_TAG (0x080) – 如果未设置，HBA 将不会在命令标记排队模式下运行。
- SCSI_OPTIONS_FAST (0x100) – 如果未设置，HBA 将不会在 FAST SCSI 模式下运行总线。
- SCSI_OPTIONS_WIDE (0x200) – 如果未设置，HBA 将不会在 WIDE SCSI 模式下运行总线。

按目标的 scsi-options

HBA 驱动程序可能支持以下格式的按目标的 scsi-options 功能：

```
target<n>-scsi-options=<hex value>
```

在此示例中，<*n*> 是指目标 ID。如果定义了按目标的 scsi-options 属性，则 HBA 驱动程序将使用该值，而不使用按 HBA 驱动程序实例的 scsi-options 属性。例如，如果仅需针对某个特定目标设备禁用同步数据传送，则该方法可提供更准确的控制。按目标的 scsi-options 属性可以在 driver.conf(4) 文件中定义。

以下示例说明了按目标的 scsi-options 属性定义，用于禁用目标设备 3 的同步数据传送：

```
target3-scsi-options=0x2d8
```

x86 目标驱动程序配置属性

某些 x86 SCSI 目标驱动程序（如 cmdk 磁盘驱动程序）使用以下配置属性：

- disk（磁盘）
- queue
- flow_control

如果使用 cmdk 样例驱动程序编写适用于 x86 平台的 HBA 驱动程序，则必须在 driver.conf(4) 文件中定义所有相应属性。

注 – 这些属性定义应仅显示在 HBA 驱动程序的 driver.conf(4) 文件中。HBA 驱动程序本身不应以任何方式检查或尝试解释这些属性。这些属性仅是建议性的，并且用作 cmdk 驱动程序的附件。不应以任何方式依赖这些属性。在将来的发行版中可能不会使用这些属性定义。

disk 属性可用于定义 cmdk 支持的磁盘类型。对于 SCSI HBA，disk 属性的唯一可能值是：

- disk="scdk"－磁盘类型为 SCSI 磁盘

queue 属性用于定义磁盘驱动程序如何在执行 strategy(9E) 的过程中对传入请求的队列进行排序。以下是两个可能的值：

- queue="qsort"－disksort(9F) 提供的单向升降排队模型
- queue="qfifo"－FIFO，即先入先出排队模型

flow_control 属性用于定义如何将命令传输到 HBA 驱动程序。以下是三个可能的值：

- flow_control="dsngl"－每个 HBA 驱动程序一个命令
- flow_control="dmult"－每个 HBA 驱动程序多个命令。如果 HBA 队列已满，则驱动程序返回 TRAN_BUSY .
- flow_control="duplx"－HBA 可以支持单独的读/写队列，每个队列多个命令。FIFO 排序用于写队列。用于读队列的排队模型通过 *queue* 属性进行描述。如果 HBA 队列已满，则驱动程序返回 TRAN_BUSY

以下示例是一个供 x86 HBA PCI 设备使用的 driver.conf(4) 文件，该设备设计用于 cmdk 样例驱动程序：

```
#
# config file for ISP 1020 SCSI HBA driver
#
        flow_control="dsngl" queue="qsort" disk="scdk"
        scsi-initiator-id=7;
```

排队支持

有关**标记排队**的定义，请参阅 SCSI-2 规范。要支持标记排队，请首先检查 *scsi_options* 标志 SCSI_OPTIONS_TAG，查看是否在全局范围内启用了标记排队。接下来，检查目标是否为 SCSI-2 设备以及目标是否启用了标记排队。如果这些条件全部符合，请通过 scsi_ifsetcap(9F) 尝试启用标记排队。

如果标记排队失败，则可尝试设置**无标记排队**。在此模式下，可提交主机适配器驱动程序所需的或对其最适用的尽可能多的命令。与标记排队不同，主机适配器随后将按

照一次一个命令的方式对目标命令进行排队。在标记排队中，主机适配器将提交尽可能多的命令，直到目标指示队列已满为止。

◆ ◆ ◆ 第 19 章

19 网络设备驱动程序

这些年以来，出现了略有不同的 NIC 体系结构。MAC 层是与 NIC 硬件交互的常见 Oracle Solaris 框架。MAC 层需要能够利用尽可能多的硬件功能（如硬件分类、VLAN 标记、VLAN 剥离、硬件校验和负载转移、大段负载转移、负载分配等等），同时提供可供不同类型的硬件使用的通用模型。

要为 Oracle Solaris OS 编写网络驱动程序，请使用 Solaris 通用 LAN 驱动程序 (Generic LAN Driver, GLD) 框架。

- 对于新的以太网驱动程序，请使用 GLDv3 框架。请参见第 355 页中的 “GLDv3 网络设备驱动程序框架”。GLDv3 框架是基于函数调用的接口。
- 要维护较为陈旧的以太网、令牌环或 FDDI 驱动程序，请使用 GLDv2 框架。请参见第 376 页中的 “GLDv2 网络设备驱动程序框架”。GLDv2 是为驱动程序提供用于共享的通用代码的内核模块。
- 如果要编写 NIC 驱动程序，另请参见第 21 章，SR-IOV 驱动程序。

GLDv3 网络设备驱动程序框架

GLDv3 框架是 MAC 插件和 MAC 驱动程序服务例程与结构的基于函数调用的接口。GLDv3 框架代表符合 GLDv3 的驱动程序实现必要的 STREAMS 入口点，并处理 DLPI 兼容性。

本节讨论以下主题：

GLDv3 MAC 注册

GLDv3 为使用 MAC_PLUGIN_IDENT_ETHER 插件类型注册的驱动程序定义驱动程序 API。

GLDv3 MAC 注册过程

GLDv3 设备驱动程序必须执行以下步骤来向 MAC 层注册：

- 引入以下两个 MAC 头文件：sys/mac_ether.h 和 sys/mac_provider.h。切勿在驱动程序中包含其他任何与 MAC 相关的头文件。
- 填充 mac_callbacks 结构。
- 在其 _init() 入口点中调用 mac_fini_ops() 函数。
- 在其 attach() 入口点中调用 mac_alloc() 函数，以分配 mac_register 结构。
- 填充 mac_register 结构，并在其 attach() 入口点中调用 mac_register() 函数。
- 在其 detach() 入口点中调用 mac_unregister() 函数。
- 在其 _fini() 入口点中调用 mac_fini_ops() 函数。
- 链接 misc/mac 依赖性：

```
# ld -N"misc/mac" xx.o -o xx
```

GLDv3 MAC 注册函数

GLDv3 接口包括在使用 MAC 层注册过程中通告的驱动程序入口点和驱动程序调用的 MAC 入口点。

mac_init_ops() 和•mac_fini_ops() 函数

```
void mac_init_ops(struct dev_ops *ops, const char *name);
```

GLDv3 设备驱动程序必须在调用 mod_install(9F) 之前在其 _init(9E) 入口点中调用 mac_init_ops(9F) 函数。

```
void mac_fini_ops(struct dev_ops *ops);
```

GLDv3 设备驱动程序必须在调用 mod_remove(9F) 之后在其 _fini(9E) 入口点中调用 mac_fini_ops(9F) 函数。

示例 19–1 mac_init_ops() 和•mac_fini_ops() 函数

```
int
_init(void)
{
        int     rv;
        mac_init_ops(&xx_devops, "xx");
        if ((rv = mod_install(&xx_modlinkage)) != DDI_SUCCESS) {
                mac_fini_ops(&xx_devops);
        }
```

示例 19–1 mac_init_ops() 和•mac_fini_ops() 函数 （续）

```
        return (rv);
}

int
_fini(void)
{
        int     rv;
        if ((rv = mod_remove(&xx_modlinkage)) == DDI_SUCCESS) {
                mac_fini_ops(&xx_devops);
        }
        return (rv);
}
```

mac_alloc() 和 mac_free() 函数

```
mac_register_t *mac_alloc(uint_t version);
```

mac_alloc(9F) 函数分配一个新的 mac_register 结构，并返回此结构的指针。在将新结构传递给 mac_register() 之前初始化结构成员。在 mac_alloc() 返回之前，MAC 层会初始化 MAC 专用元素。*version* 的值必须是 MAC_VERSION_V1。

```
void mac_free(mac_register_t *mregp);
```

mac_free(9F) 函数释放此前由 mac_alloc() 分配的 mac_register 结构。

mac_register() 和 mac_unregister () 函数

```
int mac_register(mac_register_t *mregp, mac_handle_t *mhp);
```

为了使用 MAC 层注册新实例，GLDv3 驱动程序必须在 attach(9E) 入口点中调用 mac_register(9F) 函数。*mregp* 参数是指向 mac_register 注册信息结构的指针。在成功时，*mhp* 参数是指向新 MAC 实例的 MAC 句柄指针。mac_tx_update()、mac_link_update () 和 mac_rx() 等其他例程需要此句柄。

示例 19–2 mac_alloc()、mac_register () 和 mac_free() 函数及 mac_register 结构

```
int
xx_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        mac_register_t        *macp;

/* ... */

        if ((macp = mac_alloc(MAC_VERSION)) == NULL) {
                xx_error(dip, "mac_alloc failed");
                goto failed;
        }

        macp->m_type_ident = MAC_PLUGIN_IDENT_ETHER;
        macp->m_driver = xxp;
        macp->m_dip = dip;
```

示例 19–2 mac_alloc()、mac_register() 和 mac_free() 函数及 mac_register 结构 （续）

```
        macp->m_src_addr = xxp->xx_curraddr;
        macp->m_callbacks = &xx_m_callbacks;
        macp->m_min_sdu = 0;
        macp->m_max_sdu = ETHERMTU;
        macp->m_margin = VLAN_TAGSZ;

        if (mac_register(macp, &xxp->xx_mh) == DDI_SUCCESS) {
                mac_free(macp);
                return (DDI_SUCCESS);
        }

/* failed to register with MAC */
        mac_free(macp);
failed:
        /* ... */
}
```

```
int mac_unregister(mac_handle_t mh);
```

mac_unregister(9F) 函数取消注册此前通过 mac_register() 注册的 MAC 实例。 *mh* 参数是 mac_register() 分配的 MAC 句柄。 从 detach(9E) 入口点调用 mac_unregister()。

示例 19–3 mac_unregister() 函数

```
int
xx_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        xx_t        *xxp; /* driver soft state */

        /* ... */

        switch (cmd) {
        case DDI_DETACH:

                if (mac_unregister(xxp->xx_mh) != 0) {
                        return (DDI_FAILURE);
                }
        /* ... */
}
```

GLDv3 MAC 注册数据结构

本节介绍的结构是在 sys/mac_provider.h 头文件中定义的。 在 GLDv3 驱动程序中引入 sys/mac_ether.h 和 sys/mac_provider.h MAC 头文件。切勿包含其他任何于 MAC 相关的头文件。

mac_register(9S) 数据结构是 MAC 注册信息结构，由 mac_alloc() 分配，传递至 mac_register()。 •在将新结构传递给 mac_register() 之前初始化结构成员。在 mac_alloc() 返回之前，MAC 层会初始化 MAC 专用元素。 m_version 结构成员是 MAC

版本。 切勿修改 MAC 版本。 m_type_ident 结构成员是 MAC 类型标识符。 将 MAC 类型标识符设置为 MAC_PLUGIN_IDENT_ETHER。 mac_register 结构的 m_callbacks 成员是指向 mac_callbacks 结构的一个实例的指针。

mac_callbacks(9S) 数据结构是设备驱动程序用于向 MAC 层公开其入口点的结构。 MAC 层使用这些入口点来控制驱动程序。这些入口点用于完成启动和停止适配器、管理多点传送地址、设置混杂模式、查询适配器功能、获取和设置属性等任务。 有关必需和可选 GLDv3 入口点的完整列表，请参见表 19–1。在 mac_register 结构的 m_callbacks 字段中提供指向 mac_callbacks 结构的指针。

mac_callbacks 结构的 mc_callbacks 成员是一个位掩码，结合了指定驱动程序将实现哪些可选入口点的以下标志。mac_callbacks 结构的其他成员是驱动程序各入口点的指针。

MC_IOCTL　显示 mc_ioctl() 入口点。

MC_GETCAPAB　显示 mc_getcapab() 入口点。

MC_SETPROP　显示•mc_setprop() 入口点。

MC_GETPROP　显示 mc_getprop() 入口点。

MC_PROPINFO　显示 mc_propinfo() 入口点。

MC_PROPERTIES　显示所有属性入口点。设置•MC_PROPERTIES 等同于设置全部三个标志：MC_SETPROP、 MC_GETPROP 和 MC_PROPINFO。

示例 19–4 mac_callbacks 结构

```
#define XX_M_CALLBACK_FLAGS \
    (MC_IOCTL | MC_GETCAPAB | MC_PROPERTIES)

static mac_callbacks_t xx_m_callbacks = {
        XX_M_CALLBACK_FLAGS,
        xx_m_getstat,     /* mc_getstat() */
        xx_m_start,       /* mc_start() */
        xx_m_stop,        /* mc_stop() */
        xx_m_promisc,     /* mc_setpromisc() */
        xx_m_multicst,    /* mc_multicst() */
        xx_m_unicst,      /* mc_unicst() */
        xx_m_tx,          /* mc_tx() */
        NULL,             /* Reserved, do not use */
        xx_m_ioctl,       /* mc_ioctl() */
        xx_m_getcapab,    /* mc_getcapab() */
        NULL,             /* Reserved, do not use */
        NULL,             /* Reserved, do not use */
        xx_m_setprop,     /* mc_setprop() */
        xx_m_getprop,     /* mc_getprop() */
        xx_m_propinfo     /* mc_propinfo() */
};
```

GLDv3 功能

GLDv3 实现一种功能机制，允许框架查询和启用 GLDv3 驱动程序支持的功能。使用 mc_getcapab(9E) 入口点报告功能。如果驱动程序支持某种功能，请通过 mc_getcapab() 传递关于该功能的信息，例如特定于功能的入口点或标志。传递一个指向 mac_callback 结构中 mc_getcapab() 入口点的指针。有关 mac_callbacks 结构的更多信息，请参见第 358 页中的 “GLDv3 MAC 注册数据结构”。

```
boolean_t mc_getcapab(void *driver_handle, mac_capab_t cap, void *cap_data);
```

cap 参数指定所查询的功能类型。 *cap* 的值可以是 MAC_CAPAB_HCKSUM（硬件校验和负载转移）、MAC_CAPAB_LSO（大段负载转移）或 MAC_CAPAB_RINGS。使用 *cap_data* 参数将功能数据返回框架。

如果驱动程序支持 *cap* 功能，则 mc_getcapab() 入口点必须返回 B_TRUE。如果驱动程序不支持 *cap* 功能，则 mc_getcapab() 必须返回 B_FALSE。

示例 19–5 mc_getcapab() 入口点

```
static boolean_t
xx_m_getcapab(void *arg, mac_capab_t cap, void *cap_data)
{
        switch (cap) {
        case MAC_CAPAB_HCKSUM: {
                uint32_t *txflags = cap_data;
                *txflags = HCKSUM_INET_FULL_V4 | HCKSUM_IPHDRCKSUM;
                break;
        }
        case MAC_CAPAB_LSO: {
                /* ... */
                break;
        }
             case MAC_CAPAB_RINGS: {
                /* ... */
                break;
        }
        default:
                return (B_FALSE);
        }
        return (B_TRUE);
}
```

MAC 环功能

下面各节将介绍所支持的功能和需要返回的相应功能数据。

环和环组第 2 层分类

传送硬件环和接收硬件环都是 DMA 通道，可由设备驱动程序公开。环与环组关联。接收环组与一个或多个 MAC 地址关联，与某个接收组关联的任何 MAC 地址匹配的所有网络通信都必须由 NIC 通过该组的其中一个环进行传送。到接收环组的通信的控制是通过第 2 层分类在硬件中启用的。

接收环到环组的映射可以是动态的也可以是静态的。对于动态环组，环可以根据框架的要求在组之间移动，从而动态地缩小或增加组的大小。但是，对于静态环组，这些环以静态方式指定给组，并且无法更改该指定。

如果某个接收组包含多个环，则 NIC 必须使用某个散列机制（如 RSS（Receive Side Scaling，接收端缩放））在这些环中分散通信流量，从而允许为多个连接指定不同的环。

必须仅将接收组中的一个组指定为缺省组（通常是位于索引 0 处的第一个组）。此接收组具有以下特性：

- 应至少有一个环。
- 指定给 NIC 的主 MAC 客户机。主 MAC 客户机指定为 NIC 的主 MAC 地址，通常为 IP 地址。
- 必须用于接收来自网络的所有多播和广播通信流量。
- 如果将 NIC 置于混杂模式，则必须使用它来接收与指定给非缺省接收组的 MAC 地址不匹配的所有通信。

就接收环和接收环组的硬件实现而言，需注意以下几点：

- 如果实现了多个接收环，但不支持第 2 层分类，则硬件应公开单个接收环组和属于框架的该组的所有环。
- 如果实现了第 2 层硬件分类，但是不支持 RSS，则硬件应注册多个接收组，让每个组包含一个环。
- 如果同时实现了第 2 层硬件分类和 RSS，则硬件应注册多个接收组，让每个组包含一个或多个环。
- 如果第 2 层硬件分类和 RSS 都未实现，则硬件不应通告环功能，或不应通告具有单个伪环和环组的环功能（可用来动态轮询适配器的通信）。

注册环和组过程概述

向框架注册环涉及的过程包括从框架到驱动程序的各种调用。以下步骤介绍了注册过程：

1. 框架通过调用驱动程序查询驱动程序的 `MAC_CAPAB_RINGS` 功能。为传送环创建一个调用，为接收环创建一个调用。有关更多信息，请参见第 362 页中的 “MAC_CAPAB_RINGS 功能”。

2. 此框架使用从前面的步骤中获得的 mr_rget(9E) 和 mr_gget (9E) 入口点来检索有关特定环或环组的信息。有关更多信息，请参见 mr_rget(9E) 和 mr_gget(9E) 手册页。
3. 当框架要使用某个环时，它使用 mgi_start(9E) 入口点启动环组，然后使用在前面的步骤中通告的 mri_start(9E) 入口点启动环。

   现在，通信流量可以流经环，直到通过 mgi_stop(9E) 和 mri_stop(9E) 入口点将其停止。

MAC_CAPAB_RINGS 功能

为获得有关对硬件传送环和接收环的支持的信息，框架将在 *cap* 参数中发送 MAC_CAPAB_RINGS，并期待在指向 mac_capab_rings 结构的 *cap_data* 字段中接收返回的信息。

框架将分配 mac_capab_rings(9S) 结构，并为接收环将 *mr_type* 成员设置为 MAC_RING_TYPE_RX，或为传送环将其设置为 MAC_RING_TYPE_TX。然后，mac_capab_rings 结构的其余成员由驱动程序填充。

以下字段是在 mac_capab_rings 结构中定义的：

mr_version	必须设置为 MAC_RINGS_VERSION_1。
mr_rnum	环数。
mr_gnum	组数。
mr_group_type	定义以下值： ■ MAC_GROUP_TYPE_DYNAMIC－该组是动态的。 ■ MAC_GROUP_TYPE_STATIC－此组为静态组。 有关更多信息，请参见第 361 页中的 “环和环组第 2 层分类”。
mr_gget()	用于获取有关环组的更多信息的驱动程序入口点。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
mr_rget()	用于获取有关环的更多信息的驱动程序入口点。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mr_gaddring()	用于向组添加环的驱动程序入口点。请参见 mr_gaddring(9E)。
mr_gremring()	用于从组删除环的驱动程序入口点。请参见 mr_gremring(9E)。

mr_gget() 入口点

框架为与 *mr_gnum* 参数指示的组数对应的每个有效组索引调用 mr_gget(9E) 入口点。有关更多信息，请参见 mr_gget(9E)。调用 mr_gget() 之后，驱动程序在 mac_group_info 结构中返回组信息。此结构本身是由框架预分配的，并由驱动程序填充。

以下字段是在 mac_group_info 结构中定义的：

mgi_driver	一个不透明的驱动程序组句柄，框架在将来调用组入口点时使用。
mgi_count	组中的环数。
mgi_flags	组标志 MAC_GROUP_DEFAULT 将组标识为缺省组。有关更多信息，请参见第 361 页中的 “环和环组第 2 层分类”。
mgi_start	组启动入口点。
mgi_stop	组停止入口点。
mgi_addmac	添加单点传送 MAC 地址入口点。
mgi_remmac	删除单点传送 MAC 地址入口点。
mgi_addvlan	用来添加硬件 VLAN 过滤、标记和 VLAN 标记剥离的入口点。
mgi_remvlan	用来删除硬件 VLAN 过滤、标记和 VLAN 标记剥离的入口点。
mgi_setmtu	设置 RX 组 MTU 入口点
mgi_getsriov_info	用于为组检索 SR-IOV 信息的入口点。有关更多信息，请参见第 364 页中的 “环组和 SR-IOV ”。

有关详细信息，请参见mac_group_info(9S)和 mac_group_info(9E)。

注 – mgi_addmac(9E)和 mgi_remmac(9E) 入口点仅用于接收组。只要设备驱动程序支持环功能，mc_unicst(9E) 入口点就必须设置为 NULL。

注 – mgi_addvlan() 入口点执行以下操作：

- 它定义 NIC 必须允许进行传送和接收的 VLAN ID。也就是说，将丢弃未在所配置的列表中列出的任何有标记数据包。
- 如果设置了 MAC_GROUP_VLAN_TRANSPARENT_ENABLE 标志，则它还将为该特定 VLAN ID 启用硬件 VLAN 标记和剥离。

---

mr_rget() 入口点

对应于 *mr_gnum* 指示的组数和对 MAC_CAPAB_RINGS 的调用通告的 *mr_rnum* 指示的环数，框架为每个有效的组和环索引调用 mr_rget(9E) 入口点。有关详细信息，请参见 mr_rget(9E)。

完成对 mr_rget() 的调用后，驱动程序在 mac_ring_info 结构中返回环信息。此结构是由框架预分配的，并由驱动程序填充。

以下字段是在 mac_ring_info 结构中定义的：

mri_driver	一个不透明的驱动程序组句柄，框架在将来调用环入口点时使用。
mri_start	环启动入口点。
mri_stop	环停止入口点
mri_stat	环统计信息入口点。有关更多信息，请参见第 370 页中的 “GLDv3 网络统计信息”。
mri_tx	环传送入口点。有关更多信息，请参见第 367 页中的 “传输数据路径”。
mri_poll	环轮询入口点。有关更多信息，请参见第 368 页中的 “接收数据路径”。
mri_intr_ddi_handle	与此环的中断关联的 DDI 中断句柄。
mri_intr_enable(9E)	启用 RX 环上的中断。有关更多信息，请参见第 368 页中的 “接收数据路径”。
mri_intr_disable(9E)	禁用 RX 环上的中断。有关更多信息，请参见第 368 页中的 “接收数据路径”。

有关详细信息，请参见 mac_group_info(9S) 和 mac_ring_info(9S) 手册页。

注 – mri_tx() 必须仅为传送环设置，而 mri_poll() 必须仅为接收环设置。

注 – 如果某个驱动程序实现了环功能，则 mac_callbacks 结构中的 mc_tx() 入口点必须设置为 NULL。

环组和 SR-IOV

具有 SR-IOV 功能的设备驱动程序使用 MAC_CAPAB_RINGS 功能，通过实现 mgi_getsriov_info(9E) 组入口点来告诉框架它们具有 SR-IOV 功能。PF 驱动程序负责实现该入口点。

在调用 mgi_getsriov_info(9E) 后，驱动程序在 mac_sriov_info 结构中返回 SR-IOV 信息。此结构是由框架预分配的，并由驱动程序填充。

PF（Physical Function，物理功能）驱动程序实例与 VF（Virtual Functions，虚拟功能）注册相同数量的传送和接收环组。PF 驱动程序通告的这些环组很特殊，用于管理

VF。这些环组中没有任何数据流过。它们用于配置单点传送 MAC 地址、设置 MTU、添加 VLAN 过滤器、删除 VLAN 过滤器、删除 VLAN 硬件以及为 VF 执行 VLAN 标记和剥离。

注 – VF 驱动程序对驱动程序希望加入的 MAC 多播组进行编程。PF 驱动程序不会控制这些地址的编程。

由 PF 驱动程序设置的 `msi_vf_index` 结构成员捕获与环组对应的 VF 索引。这是与驱动程序调用 `pci_plist_getvf(9F)` 函数时设备驱动程序使用的索引相同的索引。

有关 SR-IOV 驱动程序的详细信息，请参见第 21 章，SR-IOV 驱动程序。

硬件校验和负载转移

为了获得关于硬件校验和负载转移支持的数据，框架将在 *cap* 参数中发送 `MAC_CAPAB_HCKSUM`。 请参见第 365 页中的 “硬件校验和负载转移功能信息”。

要在启用硬件校验和的情况下查询校验和负载转移元数据以及检索每个包的硬件校验和元数据，请使用 `mac_hcksum_get(9F)`。 请参见第 365 页中的 “`mac_hcksum_get()()` 函数标志”。

要设置校验和负载转移元数据，请使用 `mac_hcksum_set(9F)`。请参见第 366 页中的 “`mac_hcksum_set()()` 函数标志”。

有关更多信息，请参见第 368 页中的 “硬件校验和：硬件”和第 369 页中的 “硬件校验和：MAC 层”。

硬件校验和负载转移功能信息

要将关于 `MAC_CAPAB_HCKSUM` 功能的信息传递给框架，驱动程序必须在指向 `uint32_t` 的 *cap_data* 中设置以下标志的组合。 这些标志指明驱动程序能够为外发包执行的硬件校验和负载转移的级别。

`HCKSUM_INET_PARTIAL`	1 的补码的部分校验和功能
`HCKSUM_INET_FULL_V4`	针对 IPv4 包的 1 的补码的完全校验和能力
`HCKSUM_INET_FULL_V6`	针对 IPv6 包的 1 的补码的完全校验和能力
`HCKSUM_IPHDRCKSUM`	IPv4 头校验和负载转移功能

mac_hcksum_get()() 函数标志

`mac_hcksum_get()` 的 *flags* 参数是以下值的组合：

`HCK_FULLCKSUM`	计算此包的完整校验和。

HCK_FULLCKSUM_OK	完整校验和已在硬件中通过验证，证实是正确的。
HCK_PARTIALCKSUM	根据传递给 mac_hcksum_get() 的其他参数计算 1 的补码的部分校验和。HCK_PARTIALCKSUM 与 HCK_FULLCKSUM 互斥。
HCK_IPV4_HDRCKSUM	计算 IP 报头校验和。
HCK_IPV4_HDRCKSUM_OK	IP 头校验和已在硬件中通过验证，证实是正确的。

mac_hcksum_set()() 函数标志

mac_hcksum_set 的 flags() 参数是以下值的组合：

HCK_FULLCKSUM	通过 *value* 参数计算和传递完整校验和。
HCK_FULLCKSUM_OK	完整校验和已在硬件中通过验证，证实是正确的。
HCK_PARTIALCKSUM	通过 *value* 参数计算和传递部分校验和。HCK_PARTIALCKSUM 与 HCK_FULLCKSUM 互斥。
HCK_IPV4_HDRCKSUM	通过 *value* 参数计算和传递 IP 头校验和。••
HCK_IPV4_HDRCKSUM_OK	IP 头校验和已在硬件中通过验证，证实是正确的。

大段（或大量传送）负载转移

为了查询大段（或大量传送）负载转移支持，框架将在 *cap* 参数中发送 MAC_CAPA _LSO，并等待接收在指向 mac_capab_lso(9S) 结构的 *cap_data* 中返回的信息。框架将分配 mac_capab_lso 结构，并在 *cap_data* 中传递指向此结构的指针。 mac_capab_lso 结构包含一个 lso_basic_tcp_ipv4(9S) 结构和一个 lso_flags 成员。 如果驱动程序实例为 IPv4 上的 TCP 支持 LSO，请设置 lso_flags 中的 LSO_TX_BASIC_TCP_IPV4 标志，并将 lso_basic_tcp_ipv4 结构的 lso_max 成员设置为驱动程序实例支持的最大有效载荷大小。

使用 mac_lso_get(9F) 获得每个包的 LSO 元数据。 如果为此包启用了 LSO，则将在 mac_lso_get () *flags* 参数中设置 HW_LSO 标志。在大段的分段过程中使用的最大段大小 (maximum segment size, MSS) 将通过 *mss* 参数指向的位置返回。 有关更多信息，请参见第 368 页中的“大段负载转移”。

GLDv3 数据路径

数据路径入口点包括以下组件：

- 由驱动程序导出、由 GLDv3 框架调用以发送包的回调。
- 驱动程序为传输流量控制和接收包而调用的 GLDv3 框架入口点。

注 – 如果驱动程序实现了环功能，则驱动程序发送和接收的所有数据都将通过特定于环的入口点进行传送。

传输数据路径

GLDv3 框架调用的用来向驱动程序传送消息块的传送入口点的类型取决于底层驱动程序对 MAC_CAPAB_RINGS 的支持。如果驱动程序支持 MAC_CAPAB_RINGS 功能，则框架将调用 mri_tx(9E) 环入口点。否则，框架将调用 mc_tx(9E) 入口点。

相应地，设备驱动程序必须在 mc_tx() 或 mri_tx() 中提供传送入口点的指针。有关更多信息，请参见第 358 页中的 “GLDv3 MAC 注册数据结构”和第 363 页中的 “mr_rget() 入口点”。

示例 19–6 mc_tx() 和 mri_tx() 入口点

```
mblk_t *
xx_m_tx(void *arg, mblk_t *mp)
{
        xx_t    *xxp = arg;
        mblk_t   *nmp;

        mutex_enter(&xxp->xx_xmtlock);

        if (xxp->xx_flags & XX_SUSPENDED) {
                while ((nmp = mp) != NULL) {
                        xxp->xx_carrier_errors++;
                        mp = mp->b_next;
                        freemsg(nmp);
                }
                mutex_exit(&xxp->xx_xmtlock);
                return (NULL);
        }

        while (mp != NULL) {
                nmp = mp->b_next;
                mp->b_next = NULL;

                if (!xx_send(xxp, mp)) {
                        mp->b_next = nmp;
                        break;
                }
                mp = nmp;
        }
        mutex_exit(&xxp->xx_xmtlock);

        return (mp);
}
```

以下各节将讨论与将数据传输至硬件相关的主题。

流量控制

如果驱动程序因硬件资源不足而无法发送包，则驱动程序将返回无法发送的包的子链。此后，在更多描述符可用时，驱动程序必须调用 mac_tx_update(9F) 或 mac_tx_ring(9F) 以通知框架。驱动程序将根据它是否实现了环功能来调用这两个函数之一。

硬件校验和：硬件

如果驱动程序指定了硬件校验和支持（请参见第 365 页中的 “硬件校验和负载转移”），则驱动程序必须执行以下任务：

- 使用 mac_hcksum_get(9F) 检查每个包的硬件校验和元数据。
- 对硬件进行编程，以执行所需的校验和计算。

大段负载转移

如果驱动程序指定了 LSO 功能（请参见第 366 页中的 “大段（或大量传送）负载转移”），则驱动程序必须使用 mac_lso_get(9F) 来查询是否必须在包上执行 LSO。•

虚拟 LAN：硬件

管理员配置 VLAN 时，MAC 层将通过 mc_tx() 入口点，在外发包传递至驱动程序之前将所需的 VLAN 头添加到外发包中。不过，如果硬件支持 VLAN 标记，则标记将转移到硬件上。有关更多详细信息，请参见第 362 页中的 “mr_gget() 入口点”。

接收数据路径

接收数据路径可能是由中断驱动的，也可能是由轮询驱动的。

接收中断数据路径

注意：如果驱动程序不支持环功能，请在驱动程序的中断处理程序中调用mac_rx(9F) 函数，以将包含一个或多个数据包的链沿栈向上传送到 MAC 层。避免在调用 mac_rx() 或 mac_rx_ring() 的过程中持有互斥锁或其他锁。需要特别指出的是，在调用 mac_rx() 或 mac_rx_ring() 期间不得持有传输线程可使用的锁。

在中断模式中，只要数据包链被 NIC 接收并且可供驱动程序选取，就将从驱动程序向框架发送数据包链。数据包链由通过 b_next 彼此连接的一个或多个 mblk_t 组成，并且使每个数据包的处理开销降低。通过调用 mac_rx_ring() 入口点，接收的数据包将以中断模式向上传递给框架。

```
void mac_rx_ring(mac_handle_t mh, mac_ring_handle_t mrh, mblk_t *mp_chain, int64_tmr_gen_num)
```

通过 mac_register() 函数向内核进行注册后，mh_handle 将响应设备驱动程序获取的 MAC 句柄。mrh_handle 是作为 mr_rget() 调用的一部分传递给驱动程序的框架环句柄。如果接收环是通过 mri_start() 入口点启动的，则 mr_gen_num 必须设置为框架指定的生成号。驱动程序提供的环生成号将与框架内保留的环生成号进行匹配。如果它们不匹配，则接收的数据包将被视为来自早期指定的环的过时数据包，并将被丢弃。

接收轮询数据路径

除了能够通过由中断驱动的路径接收数据包之外，框架还支持基于轮询的数据路径。在轮询模式下，在栈中运行的内核线程通过一个轮询入口点从驱动程序提取数据包。这样便使栈能够有效控制何时处理数据包以及使用何种优先级，同时可基于实际负载减少进入系统的中断数量。此外，轮询还使栈能够对接收的通信流量更有效地实施带宽限制，这在虚拟化方案中尤其重要。主机根据需要在中断模式和轮询模式之间切换。当环处于轮询模式时，驱动程序不应使用 mac_rx_ring() 函数传送通过接收环接收的数据包。这能够得以保证，因为中断在轮询模式下被禁用。相反，框架将调用驱动程序作为 mac_ring_info 结构的一部分公开的 mri_poll() 入口点。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。

在中断模式和轮询模式之间切换

缺省情况下，环在启动以后应处于中断模式。只要环处于中断模式，它就应通过入口点以链的形式向上传递接收的数据包。当主机将环切换到轮询模式时，它将通过 mac_intr 结构（是之前通过 mac_ring_info 结构公开的）调用入口点来禁用其中断。

硬件校验和：MAC 层

如果驱动程序指定了硬件校验和支持（请参见第 365 页中的 “硬件校验和负载转移”），则驱动程序必须使用 mac_hcksum_set(9F) 函数将硬件校验和元数据与包关联。

虚拟 LAN：MAC 层

VLAN 包必须连同标记一起传递至 MAC 层。切勿剥离数据包中的 VLAN 头。但是，如果硬件支持 VLAN 剥离，并且框架已请求硬件剥离 VLAN 标记，则硬件可以剥离 VLAN 标记以提高性能。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。

GLDv3 状态更改通知

驱动程序可以调用以下函数来通知网络栈驱动程序的状态已更改。

```
void mac_tx_update(mac_handle_t mh);

void mac_tx_ring_update(mac_handle_t mh, mac_ring_handle_t rh)
```

mac_tx_update(9F)或 mac_tx_ring(9F) 函数通知框架有更多 TX 描述符可用。如果 mc_tx() 或 mri_tx() 返回了非空数据包链，则驱动程序必须在资源可用后立即调用 mac_tx_update() 或 mac_tx_ring_update()，通知 MAC 层重试之前作为未发送的数据包返回的数据包。有关 mc_tx() 和 mri_tx() 入口点的更多信息，请参见第 367 页中的 “传输数据路径”。

```
void mac_link_update(mac_handle_t mh, link_state_t new_state);
```

mac_link_update(9F) 函数通知 MAC 层介质链接的状态已更改。*new_state* 参数必须为以下值之一：

LINK_STATE_UP	介质链路为启动状态。
LINK_STATE_DOWN	介质链路为关闭状态。
LINK_STATE_UNKNOWN	介质链路为未知状态。

GLDv3 网络统计信息

设备驱动程序为其管理的设备实例维护一组统计信息。MAC 层通过驱动程序的 mc_getstat(9E) 入口点查询这些统计信息。

```
int mc_getstat(void *driver_handle, uint_t stat, uint64_t *stat_value);
```

GLDv3 框架使用 *stat* 指定所查询的统计信息。驱动程序使用 *stat_value* 返回 *stat* 指定的统计信息的值。如果返回了统计信息的值，则 mc_getstat() 必须返回 0。如果驱动程序不支持 *stat* 统计信息，mc_getstat() 必须返回 ENOTSUP。

所支持的 GLDv3 统计信息是通用 MAC 统计信息和特定于以太网的统计信息的联合。有关所支持的统计信息的完整列表，请参见 mc_getstat(9E) 手册页。

示例 19–7 mc_getstat() 入口点

```
int
xx_m_getstat(void *arg, uint_t stat, uint64_t *val)
{
        xx_t    *xxp = arg;

        mutex_enter(&xxp->xx_xmtlock);
        if ((xxp->xx_flags & (XX_RUNNING|XX_SUSPENDED)) == XX_RUNNING)
                xx_reclaim(xxp);
        mutex_exit(&xxp->xx_xmtlock);

        switch (stat) {
        case MAC_STAT_MULTIRCV:
                *val = xxp->xx_multircv;
                break;
        /* ... */
        case ETHER_STAT_MACRCV_ERRORS:
                *val = xxp->xx_macrcv_errors;
```

示例 19–7 mc_getstat() 入口点 （续）

```
                break;
        /* ... */
        default:
                return (ENOTSUP);
        }
        return (0);
}
```

mri_stat() 环入口点是一个强制的环入口点，支持环功能的所有设备驱动程序都必须实现该入口点。框架将使用该入口点查询为每个硬件传送环和接收环维护的统计信息。

对于硬件传送环，框架将查询以下统计信息：

- MAC_STAT_OERRORS
- MAC_STAT_OBYTES
- MAC_STAT_OPACKETS

对于硬件接收环，框架将查询以下统计信息：

- MAC_STAT_IERRORS
- MAC_STAT_RBYTES
- MAC_STAT_IPACKETS

GLDv3 属性

使用 mc_propinfo(9E) 入口点返回属性的不可变属性。此信息包括权限、缺省值和允许的值范围。使用 mc_setprop(9E) 为此特定驱动程序实例设置属性值。使用 mc_getprop(9E) 返回属性的当前值。

有关属性及其类型的完整列表，请参见 mc_propinfo(9E) 手册页。

mc_propinfo() 入口点应调用 mac_prop_info_set_perm()、mac_prop_info_set_default() 和 mac_prop_info_set_range() 函数来关联所查询属性的特定属性，例如缺省值、权限或允许的值范围等。

mac_prop_info_set_default_uint8(9F)、mac_prop_info_set_default_str(9F) 和 mac_prop_info_set_default_link_flowctrl(9F) 函数将缺省值与特定属性相关联。mac_prop_info_set_range_uint32(9F) 函数为特定属性关联值的允许范围。

mac_prop_info_set_perm(9F) 函数指定属性的权限。权限可以是以下值之一：

MAC_PROP_PERM_READ 属性为只读

MAC_PROP_PERM_WRITE	属性为只写
MAC_PROP_PERM_RW	属性可以是读取和写入

如果 mc_propinfo() 入口点未为特定属性调用 mac_prop_info_set_perm()，GLDv3 框架将假设该属性拥有对应于 MAC_PROP_PERM_RW 的读写权限。

除了 mc_propinfo(9E) 手册页中列出的属性之外，驱动程序还可公开驱动程序专用属性。使用 mac_register 结构的 m_priv_props 字段指定驱动程序支持的驱动程序专用属性。框架在 mc_setprop()、mc_getprop() 或 mc_propinfo() 中传递 MAC_PROP_PRIVATE 属性•ID。有关更多信息，请参见 mc_propinfo (9E) 手册页。

GLDv3 接口汇总

下表列出了属于 GLDv3 网络设备驱动程序框架的入口点、其他 DDI 函数和数据结构。

表 19–1 GLDv3 接口

接口名称	说明
必需的入口点	
mc_getstat(9E)	从驱动程序检索网络统计信息。请参见第 370 页中的 “GLDv3 网络统计信息”。
mc_start(9E)	启动一个驱动程序实例。GLDv3 框架在尝试任何操作之前都会调用 start 入口点。
mc_stop(9E)	停止一个驱动程序实例。MAC 层会在设备分离之前调用停止入口点。
mc_setpromisc(9E)	更改设备驱动程序实例的混杂模式。
mc_multicst(9E)	添加或删除多点传送地址。
mc_unicst(9E)	设置主单点传送地址。设备必须通过 mac_rx() 开始回传带有与新单点传送地址匹配的目标 MAC 地址的包。有关 mac_rx() 的信息，请参见第 368 页中的 “接收数据路径”。
mc_tx(9E)	发送一个或多个包。请参见第 367 页中的 “传输数据路径”。
mr_rget(9E)	获取传送和接收环信息。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mr_gget(9E)	获取传送和接收环信息。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。

表 19–1 GLDv3 接口 （续）

接口名称	说明
mr_gaddring(9E)	向接收组添加环。仅当支持动态环分组时才是必需的。请参见第 362 页中的 “MAC_CAPAB_RINGS 功能”。
mr_gremring(9E)	从接收组中删除环。仅当支持动态环分组时才是必需的。请参见第 362 页中的 “MAC_CAPAB_RINGS 功能”。
mri_tx(9E)	为 TX 环传送数据包。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mri_poll()	轮询 RX 环以检查数据包。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mri_stat()	环统计信息。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mri_intr_enable(9E)	启用 RX 环上的中断。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mri_intr_enable(9E)	禁用 RX 环上的中断。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mgi_addmac(9E)	将 MAC 地址编程到 RX 环组的驱动程序硬件中。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
mgi_remmac(9E)	从 RX 环组的驱动程序硬件中删除之前编程到其中的 MAC 地址。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
可选入口点	
mc_ioctl(9E)	可选 ioctl 驱动程序接口。此工具仅供调试之用。
mc_getcapab(9E)	检索功能。请参见第 360 页中的 “GLDv3 功能”。
mc_setprop(9E)	设置属性值。请参见第 371 页中的 “GLDv3 属性”。
mc_getprop(9E)	获得属性值。请参见第 371 页中的 “GLDv3 属性”。
mc_propinfo(9E)	获得关于属性的信息。请参见第 371 页中的 “GLDv3 属性”。
mri_start()	启动环。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mri_stop()	停止环。有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。

表 19–1 GLDv3 接口 （续）

接口名称	说明
mgi_start(9E)	环组启动。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
mgi_stop(9E)	环组停止。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
mgi_addvlan()	在硬件中启用 VLAN 过滤。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
mgi_remvlan()	删除之前编程的 VLAN 过滤器。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
mgi_setmtu()	设置 RX 组 MTU。有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
mgi_get_sriov_info()	获取 SR-IOV 信息。有关更多信息，请参见第 364 页中的 “环组和 SR-IOV ”。
数据结构	
mac_register(9S)	注册信息。请参见第 358 页中的 “GLDv3 MAC 注册数据结构”。
mac_callbacks(9S)	驱动程序回调。请参见第 358 页中的 “GLDv3 MAC 注册数据结构”。
mac_capab_lso(9S)	LSO 元数据。请参见第 366 页中的 “大段（或大量传送）负载转移”。
lso_basic_tcp_ipv4(9S)	TCP/IPv4 的 LSO 元数据。请参见第 366 页中的 “大段（或大量传送）负载转移”。
mac_capab_rings(9S)	有关更多信息，请参见第 360 页中的 “MAC 环功能”。
mac_group_info(9S)	有关更多信息，请参见第 362 页中的 “mr_gget() 入口点”。
mac_ring_info(9S)	有关更多信息，请参见第 363 页中的 “mr_rget() 入口点”。
mac_intr_t	
mac_sriov_info	
MAC 注册函数	
mac_alloc(9F)	分配新 mac_register 结构。 请参见第 356 页中的 “GLDv3 MAC 注册”。
mac_free(9F)	释放 mac_register 结构。

表 19–1 GLDv3 接口 （续）

接口名称	说明
mac_register(9F)	向 MAC 层注册。
mac_unregister(9F)	从 MAC 层取消注册。
mac_init_ops(9F)	初始化驱动程序的 dev_ops(9S) 结构。
mac_fini_ops(9F)	释放驱动程序的 dev_ops 结构。
数据传输函数	
mac_rx(9F)	向上传递接收到的包。请参见第 368 页中的 “接收数据路径”。
mac_rx_ring(9F)	向上传递接收到的包。请参见第 368 页中的 “接收数据路径”。
mac_tx_update(9F)	TX 资源可用。请参见第 369 页中的 “GLDv3 状态更改通知”。
mac_tx_ring_update(9F)	TX 资源可用。有关更多信息，请参见第 369 页中的 “GLDv3 状态更改通知”。
mac_link_update(9F)	链接状态已更改。
mac_hcksum_get(9F)	检索硬件校验和信息。请参见第 365 页中的 “硬件校验和负载转移”和第 367 页中的 “传输数据路径”。
mac_hcksum_set(9F)	附加硬件校验和信息。请参见第 365 页中的 “硬件校验和负载转移”和第 368 页中的 “接收数据路径”。
mac_lso_get(9F)	检索 LSO 信息。请参见第 366 页中的 “大段（或大量传送）负载转移”。
属性函数	
mac_prop_info_set_perm(9F)	设置属性权限。请参见第 371 页中的 “GLDv3 属性”。
mac_prop_info_set_default_uint8(9F), mac_prop_info_set_default_str(9F), mac_prop_info_set_default_link_flowctrl(9F)	设置属性值。
mac_prop_info_set_range_uint32(9F)	设置属性值范围。

GLDv2 网络设备驱动程序框架

GLDv2 是多线程、可克隆、可装入的核心模块，为局域网的设备驱动程序提供支持。Oracle Solaris OS 中的局域网 (Local area network, LAN) 设备驱动程序是基于 STREAMS 的驱动程序，使用数据链路提供者接口 (Data Link Provider Interface, DLPI) 与网络协议栈进行通信。这些协议栈使用网络驱动程序在 LAN 中发送和接收包。GLDv2 为 Oracle Solaris LAN 驱动程序实现 STREAMS 和 DLPI 的很多功能。GLDv2 提供可供许多网络驱动程序共享的通用代码。使用 GLDv2 可减少重复的代码，简化您的网络驱动程序。

有关 GLDv2 的更多信息，请参见`gld(7D)`手册页。

《STREAMS Programming Guide》中的第 II 部分, “Kernel Interface”中介绍了 STREAMS 驱动程序。具体请参见 STREAMS 指南中的第 9 章，“STREAMS 驱动程序”。STREAMS 框架是基于消息的框架。STREAMS 驱动程序特有的接口包括 STREAMS 消息队列处理入口点。

DLPI 指定 OSI 参考模型数据链路层的数据链路服务 (Data Link Service, DLS) 的接口。DLPI 使 DLS 用户能够访问和使用多种符合规定的 DLS 提供者，而无需专门确认提供者的协议。DLPI 以 M_PROTO 和 M_PCPROTO 类型的•STREAMS 消息的形式指定对 DLS 提供者的访问。DLPI 模块使用 STREAMS ioctl 调用链接至 MAC 子层。有关 DLPI 协议的更多信息，包括特定于 Oracle Solaris 的 DPLI 扩展，请参见 `dlpi(7P)` 手册页。有关 DLPI 的一般信息，请参见以下位置的 DLPI 标准：`http://www.opengroup.org/pubs/catalog/c811.htm`。

使用 GLDv2 实现的 Oracle Solaris 网络驱动程序有两个独立的部分：

- 通用组件。处理 STREAMS 和 DLPI 接口。
- 特定于设备的组件。处理特定硬件设备。
  - 通过链接 `misc/gld` 中的依赖性指明驱动程序对于 GLDv2 模块的依赖性。 在 SPARC 系统的 `/kernel/misc/sparcv9/gld`、64 位 x86 系统的 `/kernel/misc/amd64/gld` 和 32 位 x86 系统的 `/kernel/misc/gld` 中可以找到 GLDv2 模块。
  - 注册 GLDv2：驱动程序在 `attach(9E)` 入口点中为 GLDv2 提供指向其他入口点的指针。GLDv2 使用这些指针进入 `gld(9E)` 入口点。•
  - 调用•`gld(9F)` 函数来处理数据或使用某些其他 GLDv2 服务。 `gld_mac_info(9S)` 结构是 GLDv2 与特定于设备的驱动程序之间的主数据接口。

GLDv2 驱动程序必须处理格式完全的 MAC 层包，不能执行逻辑链路控制 (logical link control, LLC) 处理。

本节讨论以下主题：

- 第 377 页中的 “GLDv2 设备支持”
- 第 378 页中的 “GLDv2 DLPI 提供者”

GLDv2 设备支持

GLDv2 框架支持以下类型的设备：

- `DL_ETHER`：ISO 8802-3、IEEE 802.3 协议
- `DL_TPR`：IEEE 802.5 令牌传递环
- `DL_FDDI`：ISO 9314-2 光纤分布式数据接口

Ethernet V2 和 ISO 8802-3 (IEEE 802.3)

对于声明为 `DL_ETHER` 类型的设备，GLDv2 提供 Ethernet V2 和 ISO 8802-3 (IEEE 802.3) 包处理支持。通过 Ethernet V2，用户可以访问遵循该标准的数据链路服务提供方，而无需了解有关此提供者协议的专门知识。服务访问点 (service access point, SAP) 是用户用于与服务提供者进行通信的点。

绑定到范围为 [0-255] 的 SAP 值的流均视为相同的流，并表示用户要使用 8802-3 模式。如果 `DL_BIND_REQ` 的 SAP 值在此范围内，则 GLDv2 将计算该流上的每条后续 `DL_UNITDATA_REQ` 消息的长度。长度不包括 14 字节的介质访问控制 (media access control, MAC) 头。然后，GLDv2 将传输 MAC 帧标题 `type` 字段中具有这些长度的 8802-3 帧。这样的长度不超过 1500。

拥有范围为 [0-1500] 的 `type` 字段的帧被视为 8802-3 帧。 这些帧以 8802-3 模式沿所有打开的流进行路由。那些具有范围为 [0-255] 的 SAP 值的流均被视为处于 8802-3 模式下。如果多个流处于 8802-3 模式，则传入帧会被复制并沿这些流进行路由。

那些绑定到大于 1500 的 SAP 值的流均被假定为处于 Ethernet V2 模式下。这些流将接收以太网 MAC 头 `type` 值与流绑定到的 SAP 值完全匹配的传入包。

TPR 和 FDDI：SNAP 处理

对于 `DL_TPR` 和•`DL_FDDI` 介质类型，GLDv2 会实现最低限度的 SNAP（Sub-Net Access Protocol，子网访问协议）处理。此处理用于任何绑定到大于 255 的 SAP 值的流。范围为 [0-255] 的 SAP 值是 LLC SAP 值。此类值自然地通过介质包格式传送。大于 255 的 SAP 值需要 SNAP 头（从属于 LLC 头）来传送 16 位 Ethernet V2 样式的 SAP 值。

SNAP 头位于 LLC 头后面，目标是 SAP `0xAA`。SAP 值大于 255 的外发包要求 LLC+SNAP 头采用以下形式：

```
AA AA 03 00 00 00 XX XX
```

XX XX 表示对应于 Ethernet V2 样式 type 的 16 位 SAP。在支持非零的组织唯一标识符字段方面，此头是独一无二的。03 之外的 LLC 控制字段被视为具有 SAP 0xAA 的 LLC 包。要使用此格式之外的 SNAP 格式的客户机必须使用 LLC 并绑定到 SAP 0xAA。

将检查传入包是否符合上述格式。将符合的包与任何已绑定到包的 16 位 SNAP 类型的流进行匹配。此外，还将这些包视为与 LLC SNAP SAP 0xAA 相匹配。

针对任何 LLC SAP 接收的包将沿所有绑定到 LLC SAP 的流传递。相关内容，请参见介质类型 DL_ETHER 的介绍。

TPR：源路由

对于类型 DL_TPR 设备，GLDv2 实现最低限度的源路由支持。

源路由支持包括以下任务：

- 针对要通过桥接介质发送的包指定路由信息。路由信息存储在 MAC 头中。此信息用于确定路由。
- 了解路由。
- 索取并响应有关可能的多个路由的信息请求。
- 在可用的路由之间进行选择。

源路由会将路由信息字段添加到传出包的 MAC 头中。此外，此支持还可在传入包中识别此类字段。

GLDv2 的源路由支持并未实现《ISO 8802-2 (IEEE 802.2)》的第 9 节中指定的完整路由确定实体 (route determination entity, RDE)。但是，此支持可以与任何可能存在于同一网络或桥接网络中的 RDE 实现进行交互操作。

GLDv2 DLPI 提供者

GLDv2 实现样式 1 和样式 2 的 DLPI 提供者。物理连接点 (physical point of attachment, PPA) 是系统将自身附加到物理通信介质的点。在此物理介质上进行的所有通信都通过 PPA。样式 1 提供者根据已打开的主设备或次要设备，将流附加到特定 PPA。样式 2 提供者需要 DLS 用户使用 DL_ATTACH_REQ 显式指定所需的 PPA。在这种情况下，open(9E) 会在用户与 GLDv2 之间创建流，并且 DL_ATTACH_REQ 随后会将特定的 PPA 与该流相关联。样式 2 由次要设备号 0 表示。如果打开了次要设备号不为 0 的设备节点，则会指示样式 1，并且关联的 PPA 为次要设备号减 1。在样式 1 和样式 2 启动中，都会克隆设备。

GLDv2 DLPI 原语

GLDv2 实现某些 DLPI 原语。DL_INFO_REQ 原语请求有关 DLPI 流的信息。消息包含一个 M_PROTO 消息块。GLDv2 将在对此请求的 DL_INFO_ACK 响应中返回与设备有关的值。这些值以基于 GLDv2 的驱动程序在传递给 gld_register(9F) 函数的 gld_mac_info(9S) 结构中指定的信息为基础。

GLDv2 代表所有基于 GLDv2 驱动程序返回以下值：

- 版本为 DL_VERSION_2。
- 服务模式为 DL_CLDLS。GLDv2 实现无连接模式的服务。
- 提供者样式为 DL_STYLE1 或 DL_STYLE2，具体取决于流的启动方式。
- 不存在可选的服务质量 (Quality of Service, QOS) 支持。QOS 字段是 0。

注 – 与 DLPI 规范相反，即使在将流附加到 PPA 之前，GLDv2 也会在 DL_INFO_ACK 中返回设备的正确地址长度和广播地址。

DL_ATTACH_REQ 原语用于将 PPA 与流相关联。样式 2 DLS 提供者标识用于发送通信的物理介质时需要此请求。完成时，状态会从 DL_UNATTACHED 更改为 DL_UNBOUND。消息包含一个 M_PROTO 消息块。使用样式 1 模式时，不允许此请求。使用样式 1 打开的流已在打开完成时附加到 PPA。

DL_DETACH_REQ 原语请求将 PPA 与流相分离。仅当流使用样式 2 打开时，才允许此分离。

DL_BIND_REQ 和 DL_UNBIND_REQ 原语用于将 DLSAP（data link service access point，数据链路服务访问点）与流绑定和解除绑定。与流关联的 PPA 在完成处理该流上的 DL_BIND_REQ 之前完成初始化。可以将多个流绑定到同一 SAP。在这种情况下，每个流都接收针对此 SAP 接收的所有包的副本。

DL_ENABMULTI_REQ 和 DL_DISABMULTI_REQ 原语启用和禁用各多点传送组地址的接收。通过重复使用这些原语，应用程序或其他 DLS 用户可以创建或修改一组多点传送地址。必须将流附加到 PPA 才能接受这些原语。

DL_PROMISCON_REQ 和•DL_PROMISCOFF_REQ 原语以每个流为基础启动和关闭混杂模式。这些控制项既可在物理级别运行，也可在 SAP 级别运行。DL 提供者通过介质将所有已接收的消息路由到 DLS 用户。路由将继续进行，直到收到 DL_DETACH_REQ、收到 DL_PROMISCOFF_REQ 或关闭流为止。可以针对介质上的所有包或仅针对多点传送包指定物理级别混杂接收。

注 – 必须将流附加到 PPA 才能接受这些混杂模式原语。

DL_UNITDATA_REQ 原语用于在无连接传输中发送数据。由于此服务未得到确认，因此无法保证传送。消息包含一个 M_PROTO 消息块，以及一个或多个至少包含一个字节数据的 M_DATA 块。

在包向上游传递时，会使用 DL_UNITDATA_IND 类型。将包放入 M_PROTO 消息中，并将原语设置为 DL_UNITDATA_IND。

DL_PHYS_ADDR_REQ 原语请求当前与附加到流的 PPA 相关联的 MAC 地址。此地址将通过 DL_PHYS_ADDR_ACK 原语返回。使用样式 2 时，仅当成功执行 DL_ATTACH_REQ 之后此原语才有效。

DL_SET_PHYS_ADDR_REQ 原语更改当前与附加到流的 PPA 相关联的 MAC 地址。此原语会影响附加到此设备的所有其他当前流和将来流。更改之后，当前或随后打开并附加到此设备的所有流都将获取这一新的物理地址。在此原语再次更改物理地址或重新装入驱动程序之前，新的物理地址将一直有效。

注 – 允许超级用户在其他流绑定到 PPA 时更改同一 PPA 的物理地址。

DL_GET_STATISTICS_REQ 原语请求包含与附加到流的 PPA 相关的统计信息的 DL_GET_STATISTICS_ACK 响应。必须使用 DL_ATTACH_REQ 将样式 2 流附加到特定 PPA，此原语才能成功执行。

GLDv2 I/O 控制函数

GLDv2 可实现以下将介绍的 ioctl *ioc_cmd* 函数。如果 GLDv2 收到无法识别的 ioctl 命令，则 GLDv2 会将此命令传递给特定于设备的驱动程序的 gldm_ioctl() 例程，如 gld(9E) 中所述。

DLIOCRAW ioctl 函数可供某些 DLPI 应用程序（尤其是 snoop(1M) 命令）使用。DLIOCRAW 命令可将流置于原始模式。在原始模式下，驱动程序将在 M_DATA 消息中传递 MAC 级别的完整传入包，而不是将包转换为 DL_UNITDATA_IND 形式。DL_UNITDATA_IND 形式通常用于报告传入包。包 SAP 过滤仍将在处于原始模式下的流上执行。如果流用户要接收所有传入包，则此用户还必须选择相应的混杂模式。成功选择原始模式之后，还允许应用程序将完全格式化的包作为 M_DATA 消息发送到驱动程序以便传输。DLIOCRAW 不使用任何参数。启用之后，流将保持此模式直到关闭。

GLDv2 驱动程序需求

基于 GLDv2 的驱动程序必须包括头文件 <sys/gld.h>。

基于 GLDv2 的驱动程序必须与 -N*"misc/gld"* 选项链接：

```
%ld -r -N"misc/gld" xx.o -o xx
```

GLDv2 代表特定于设备的驱动程序实现以下函数：

- open(9E)

- close(9E)
- put(9E)（STREAMS 所必需）
- srv(9E)（STREAMS 所必需的）
- getinfo(9E)

module_info(9S) 结构的 mi_idname 元素是用于指定驱动程序名称的字符串。此字符串必须与文件系统中定义的驱动程序模块的名称完全匹配。

读端 qinit(9S) 结构应该指定以下元素：

```
qi_putp      NULL

qi_srvp      gld_rsrv

qi_qopen     gld_open

qi_qclose    gld_close
```

写端 qinit(9S) 结构应该指定以下元素：

```
qi_putp      gld_wput

qi_srvp      gld_wsrv

qi_qopen     NULL

qi_qclose    NULL
```

dev_ops(9S) 结构的 devo_getinfo 元素应该将 gld_getinfo 指定为 getinfo(9E) 例程。

驱动程序的 attach(9E) 函数会将特定于硬件的设备驱动程序与 GLDv2 功能相关联。然后，attach() 将准备设备和驱动程序以供使用。

attach(9E) 函数使用 gld_mac_alloc() 分配 gld_mac_info(9S) 结构。通常，驱动程序需要针对每台设备保存的信息比在 macinfo 结构中定义的信息多。驱动程序应该分配其他必需的数据结构，并在 gld_mac_info(9S) 结构的 gldm_private 成员中保存指向该结构的指针。

attach(9E) 例程必须初始化 macinfo 结构，如 gld_mac_info(9S) 手册页中所述。然后，attach() 例程应该调用 gld_register() 以将驱动程序与 GLDv2 模块相链接。驱动程序应该在必要时映射寄存器，并在调用 gld_register() 之前完全初始化以准备接受中断。attach(9E) 函数应该添加中断，而不应该使设备生成这些中断。驱动程序应该在调用 gld_register() 之前重置硬件，以确保硬件处于停顿状态。不得将设备置于其可能会在调用 gld_register() 之前生成中断的状态。稍后会在 GLDv2 调用驱动程序的 gldm_start() 入口点时启动设备，相关内容在 gld(9E) 手册页中介绍。gld_register() 成功执行之后，GLDv2 可能会随时调用 gld(9E) 入口点。

如果 gld_register() 成功执行，则 attach(9E) 例程应该返回 DDI_SUCCESS。如果 gld_register() 失败，则会返回 DDI_FAILURE。如果出现故障，则 attach(9E) 例程应该

取消分配在调用 gld_register() 之前分配的所有资源。然后，连接例程还应该返回 DDI_FAILURE。绝不能重新使用出现故障的 macinfo 结构。应该使用 gld_mac_free() 取消分配此类结构。

detach(9E) 函数应尝试通过调用 gld_unregister() 来尝试从 GLDv2 中取消注册驱动程序。有关 gld_unregister() 的更多信息，请参见 gld(9F) 手册页。detach(9E) 例程可以使用 ddi_get_driver_private(9F) 从设备的专用数据中获取指向所需 gld_mac_info(9S) 结构的指针。gld_unregister() 将检查可能要求不分离驱动程序的特定条件。如果检查失败，则 gld_unregister() 会返回 DDI_FAILURE，在这种情况下，驱动程序的 detach(9E) 例程必须保持设备处于运行状态并返回 DDI_FAILURE。

如果检查成功，则 gld_unregister() 会确保停止设备中断。如有必要，则会调用驱动程序的 gldm_stop() 例程。驱动程序将与 GLDv2 框架解除链接。然后，gld_unregister() 会返回 DDI_SUCCESS。在这种情况下，detach(9E) 例程应该删除中断，并使用 gld_mac_free() 取消分配在 attach(9E) 例程中分配的所有 macinfo 数据结构。然后，detach() 例程应该返回 DDI_SUCCESS。此例程必须在调用 gld_mac_free() **之前**删除中断。

GLDv2 网络统计信息

Oracle Solaris 网络驱动程序必须实现统计变量。GLDv2 可记录一些网络统计信息，但是其他统计信息必须由基于 GLDv2 的每个驱动程序进行计数。GLDv2 为基于 GLDv2 的驱动程序提供支持，以报告一组标准的网络驱动程序统计信息。GLDv2 使用 kstat(7D) 和 kstat(9S) 机制报告统计信息。DL_GET_STATISTICS_REQ DLPI 命令还可用于检索当前统计计数器。所有统计信息均以无符号数据进行维护。除非另有说明，否则统计信息为 32 位。

GLDv2 维护并报告以下统计信息。

rbytes64	已在接口上成功接收的总字节数。将存储 64 位统计信息。
rbytes	已在接口上成功接收的总字节数。
obytes64	已请求在接口上传输的总字节数。将存储 64 位统计信息。
obytes	已请求在接口上传输的总字节数。
ipackets64	已在接口上成功接收的总包数。将存储 64 位统计信息。
ipackets	已在接口上成功接收的总包数。
opackets64	已请求在接口上传输的总包数。将存储 64 位统计信息。
opackets	已请求在接口上传输的总包数。
multircv	已成功接收的多点传送包，包括组和功能地址 (long)。
multixmt	已请求传输的多点传送包，包括组和功能地址 (long)。

brdcstrcv	已成功接收的广播包 (long)。
brdcstxmt	已请求传输的广播包 (long)。
unknowns	未由任何流接受的有效已接收包 (long)。
noxmtbuf	由于传输缓冲区繁忙或无法分配传输缓冲区而在输出中放弃的包 (long)。
blocked	由于队列受控于流而使已接收的包无法沿流放置的次数 (long)。
xmtretry	在由于资源不足而延迟之后重试传输的次数 (long)。
promisc	接口的当前“混杂”状态（字符串）。

与设备有关的驱动程序将在每个实例的专用结构中跟踪以下统计信息。为了报告统计信息，GLDv2 将调用驱动程序的 gldm_get_stats() 入口点。然后，gldm_get_stats() 会在 gld_stats(9S)有关更多信息，请参见 gldm_get_stats(9E) 手册页。然后，GLDv2 将使用如下所示的命名统计变量报告已更新的统计信息。

ifspeed	接口的当前估算带宽（以 bps 为单位）。将存储 64 位统计信息。
media	设备正在使用的当前介质类型（字符串）。
intr	调用中断处理程序从而导致中断的次数 (long)。
norcvbuf	由于无法分配接收缓冲区而放弃某个有效传入包的次数 (long)。
ierrors	已接收但由于错误而无法处理的总包数 (long)。
oerrors	由于错误而无法成功传输的总包数 (long)。
missed	硬件在接收时已丢弃的包数 (long)。
uflo	传输时 FIFO 下溢的次数 (long)。
oflo	接收期间接收器下溢的次数 (long)。

以下统计信息组适用于类型为 DL_ETHER 的网络。这些统计信息由此类型的特定于设备的驱动程序维护，如上所示。

align_errors	在出现帧错误时接收的包数，即这些包不包含整数个数的八位字节 (long)。
fcs_errors	在出现 CRC 错误时接收的包数 (long)。
duplex	接口的当前双工模式（字符串）。
carrier_errors	在尝试传输时丢失载体或从未检测到载体的次数 (long)。
collisions	传输期间的以太网冲突数 (long)。
ex_collisions	传输时出现过多的冲突而导致传输失败的帧数 (long)。

tx_late_collisions	在过了一段时间后（即过了 512 位时后）发生传输冲突的次数 (long)。
defer_xmts	没有发生由于介质繁忙而延迟首次传输尝试冲突的包数 (long)。
first_collisions	在仅发生一个冲突后成功传输的包数。
multi_collisions	在发生多个冲突后成功传输的包数。
sqe_errors	已报告 SQE 测试错误的次数。
macxmt_errors	遇到传输 MAC 故障（载体和冲突故障除外）的包数。
macrcv_errors	在出现 MAC 错误（align_errors、fcs_errors 和 toolong_errors 除外）时接收的包数。
toolong_errors	接收的大于最大允许长度的包数。
runt_errors	接收的小于最小允许长度的包数 (long)。

以下统计信息组适用于类型为 DL_TPR 的网络。这些统计信息由此类型的特定于设备的驱动程序维护，如上所示。

line_errors	在出现非数据位或 FCS 错误时接收的包数。
burst_errors	已针对五个半位计时器检测到不存在转换的次数。
signal_losses	在环上检测到信号丢失情况的次数。
ace_errors	AMP 或 SMP 帧（其中 A 等于 C 等于 0）后跟其他 SMP 帧而没有中间 AMP 帧的次数。
internal_errors	站识别到内部错误的次数。
lost_frame_errors	传输期间 TRR 计时器到期的次数。
frame_copied_errors	在 FS 字段 'A' 位设置为 1 时接收发往此站的帧的次数。
token_errors	用作活动监视器的站识别到需要已传输标记的错误情况的次数。
freq_errors	传入信号的频率不同于预期频率的次数。

以下统计信息组适用于类型为 DL_FDDI 的网络。这些统计信息由此类型的特定于设备的驱动程序维护，如上所示。

mac_errors	由该 MAC 检测到出现错误但是尚未由其他 MAC 检测到出现错误的帧数。
mac_lost_errors	所接收的因出现格式错误而被剥离的帧数。
mac_tokens	已接收的标记数，即不受限制和受限制的总标记数。

mac_tvx_expired	TVX 已到期的次数。
mac_late	自重置此 MAC 或接收标记以来的 TRT 到期次数。
mac_ring_ops	环从“环未运行”状态进入“环运行”状态的次数。

GLDv2 声明和数据结构

本节介绍 gld_mac_info(9S) 和 gld_stats 结构。

gld_mac_info 结构

GLDv2 MAC 信息 (gld_mac_info) 结构是用于链接特定于设备的驱动程序与 GLDv2 的主数据接口。此结构包含 GLDv2 所需的数据，以及指向可选的其他特定于驱动程序的信息结构的指针。

可使用 gld_mac_alloc 分配 gld_mac_info() 结构，可使用 gld_mac_free() 取消分配此结构。驱动程序不能做出有关此结构长度的任何假设，因为此长度在不同发行版的 Oracle Solaris OS 和/或 GLDv2 中可能会有所不同。专用于 GLDv2 的结构成员（未在此处介绍）既不应该由特定于设备的驱动程序设置，也不应该由其读取。

gld_mac_info(9S) 结构包含以下字段。

```
caddr_t             gldm_private;              /* Driver private data */
int                 (*gldm_reset)();           /* Reset device */
int                 (*gldm_start)();           /* Start device */
int                 (*gldm_stop)();            /* Stop device */
int                 (*gldm_set_mac_addr)();    /* Set device phys addr */
int                 (*gldm_set_multicast)();   /* Set/delete multicast addr */
int                 (*gldm_set_promiscuous)(); /* Set/reset promiscuous mode */
int                 (*gldm_send)();            /* Transmit routine */
uint_t              (*gldm_intr)();            /* Interrupt handler */
int                 (*gldm_get_stats)();       /* Get device statistics */
int                 (*gldm_ioctl)();           /* Driver-specific ioctls */
char                *gldm_ident;               /* Driver identity string */
uint32_t            gldm_type;                 /* Device type */
uint32_t            gldm_minpkt;               /* Minimum packet size */
                                               /* accepted by driver */
uint32_t            gldm_maxpkt;               /* Maximum packet size */
                                               /* accepted by driver */
uint32_t            gldm_addrlen;              /* Physical address length */
int32_t             gldm_saplen;               /* SAP length for DL_INFO_ACK */
unsigned char       *gldm_broadcast_addr;      /* Physical broadcast addr */
unsigned char       *gldm_vendor_addr;         /* Factory MAC address */
t_uscalar_t         gldm_ppa;                  /* Physical Point of */
                                               /* Attachment (PPA) number */
dev_info_t          *gldm_devinfo;             /* Pointer to device's */
                                               /* dev_info node */
ddi_iblock_cookie_t gldm_cookie;               /* Device's interrupt */
                                               /* block cookie */
```

gldm_private 结构成员对设备驱动程序可见。gldm_private 还专用于特定于设备的驱动程序。GLDv2 无法使用或修改 gldm_private。通常，gldm_private 用作指向专用数据的指针，指向同时由驱动程序定义和分配的每个实例的数据结构。

以下结构成员组必须由驱动程序在调用 gld_register() 之前设置，并且此后不应该由驱动程序进行修改。由于 gld_register() 可能会使用或高速缓存结构成员的值，因此，驱动程序在调用 gld_register() 之后进行的更改可能会导致不可预测的结果。有关这些结构的更多信息，请参见 gld(9E) 手册页。

gldm_reset	指向驱动程序入口点的指针。
gldm_start	指向驱动程序入口点的指针。
gldm_stop	指向驱动程序入口点的指针。
gldm_set_mac_addr	指向驱动程序入口点的指针。
gldm_set_multicast	指向驱动程序入口点的指针。
gldm_set_promiscuous	指向驱动程序入口点的指针。
gldm_send	指向驱动程序入口点的指针。
gldm_intr	指向驱动程序入口点的指针。
gldm_get_stats	指向驱动程序入口点的指针。
gldm_ioctl	指向驱动程序入口点的指针。允许此指针为 null。
gldm_ident	指向包含设备简短说明的字符串的指针。此指针用于在系统消息中标识设备。
gldm_type	驱动程序处理的设备的类型。GLDv2 目前支持以下值： ■ DL_ETHER（ISO 8802-3 (IEEE 802.3) 和以太网总线） ■ DL_TPR（IEEE 802.5 令牌传递环） ■ DL_FDDI（ISO 9314-2 光纤分布式数据接口） 必须正确设置此结构成员，GLDv2 才能正常运行。
gldm_minpkt	最小**服务数据单元**大小：设备可以传输的最小包大小（不包括 MAC 头）。如果特定于设备的驱动程序处理任何所需的填充，则允许此大小为 0。
gldm_maxpkt	最大**服务数据单元**大小：设备可以传输的最大包大小（不包括 MAC 头）。对于以太网，此数值为 1500。
gldm_addrlen	设备所处理的物理地址的长度（以字节为单位）。对于以太网、令牌环和 FDDI，此结构成员的值应该为 6。
gldm_saplen	驱动程序所使用的 SAP 地址的长度（以字节为单位）。对于基于 GLDv2 的驱动程序，应该始终将长度设置为 -2。长度为 -2 表示支持 2 个字节的 SAP 值，并且 SAP 出现在 DLSAP

地址中的物理地址之后。有关更多详细信息，请参见 DLPI 规范中的附录 A.2“消息 DL_INFO_ACK”。

`gldm_broadcast_addr` 指向长度 `gldm_addrlen` 字节数组的指针，该数组包含要用于传输的广播地址。驱动程序必须提供保存广播地址的空间，使用相应的值填充此空间，并将 `gldm_broadcast_addr` 设置为指向此地址。对于以太网、令牌环和 FDDI，广播地址通常为 `0xFF-FF-FF-FF-FF-FF`。

`gldm_vendor_addr` 指向长度为 `gldm_addrlen` 字节的数组的指针，该数组包含供应商提供的设备网络物理地址。驱动程序必须提供保存地址的空间，使用来自设备的信息填充此空间，并将 `gldm_vendor_addr` 设置为指向此地址。

`gldm_ppa` 此设备实例的 PPA 编号。应该始终将 PPA 编号设置为从 `ddi_get_instance(9F)` 返回的实例编号。

`gldm_devinfo` 指向此设备的 `dev_info` 节点的指针。

`gldm_cookie` 以下例程之一返回的中断块 cookie：

- `ddi_get_iblock_cookie(9F)`
- `ddi_add_intr(9F)`
- `ddi_get_soft_iblock_cookie(9F)`
- `ddi_add_softintr(9F)`

此 cookie 必须对应于设备的接收中断，可从该中断中调用 `gld_recv()`。

gld_stats 结构

调用 `gldm_get_stats()` 之后，基于 GLDv2 的驱动程序会使用 (`gld_stats`) 结构将统计信息和状态信息传递给 GLDv2。请参见 `gld`(9E) 和 `gld`(7D) 手册页。当 GLDv2 报告统计信息时，将使用已由基于 GLDv2 的驱动程序填充的此结构的成员。在下面各表的注释中说明了 GLDv2 所报告的统计变量的名称。有关每条统计信息含义的更详细说明，请参见 `gld(7D)` 手册页。

驱动程序不得做出有关此结构长度的任何假设。此结构长度在不同发行版的 Oracle Solaris OS 和/或 GLDv2 中可能会有所不同。专用于 GLDv2 的结构成员（未在此处介绍）既不应该由特定于设备的驱动程序设置，也不应该由其读取。

针对所有介质类型定义了以下结构成员：

```
uint64_t    glds_speed;                 /* ifspeed */
uint32_t    glds_media;                 /* media */
uint32_t    glds_intr;                  /* intr */
uint32_t    glds_norcvbuf;              /* norcvbuf */
uint32_t    glds_errrcv;                /* ierrors */
uint32_t    glds_errxmt;                /* oerrors */
```

```
uint32_t    glds_missed;                /* missed */
uint32_t    glds_underflow;             /* uflo */
uint32_t    glds_overflow;              /* oflo */
```

针对介质类型 DL_ETHER 定义了以下结构成员：

```
uint32_t    glds_frame;                 /* align_errors */
uint32_t    glds_crc;                   /* fcs_errors */
uint32_t    glds_duplex;                /* duplex */
uint32_t    glds_nocarrier;             /* carrier_errors */
uint32_t    glds_collisions;            /* collisions */
uint32_t    glds_excoll;                /* ex_collisions */
uint32_t    glds_xmtlatecoll;           /* tx_late_collisions */
uint32_t    glds_defer;                 /* defer_xmts */
uint32_t    glds_dot3_first_coll;       /* first_collisions */
uint32_t    glds_dot3_multi_coll;       /* multi_collisions */
uint32_t    glds_dot3_sqe_error;        /* sqe_errors */
uint32_t    glds_dot3_mac_xmt_error;    /* macxmt_errors */
uint32_t    glds_dot3_mac_rcv_error;    /* macrcv_errors */
uint32_t    glds_dot3_frame_too_long;   /* toolong_errors */
uint32_t    glds_short;                 /* runt_errors */
```

针对介质类型 DL_TPR 定义了以下结构成员：

```
uint32_t    glds_dot5_line_error        /* line_errors */
uint32_t    glds_dot5_burst_error       /* burst_errors */
uint32_t    glds_dot5_signal_loss       /* signal_losses */
uint32_t    glds_dot5_ace_error         /* ace_errors */
uint32_t    glds_dot5_internal_error    /* internal_errors */
uint32_t    glds_dot5_lost_frame_error  /* lost_frame_errors */
uint32_t    glds_dot5_frame_copied_error /* frame_copied_errors */
uint32_t    glds_dot5_token_error       /* token_errors */
uint32_t    glds_dot5_freq_error        /* freq_errors */
```

针对介质类型 DL_FDDI 定义了以下结构成员：

```
uint32_t    glds_fddi_mac_error;        /* mac_errors */
uint32_t    glds_fddi_mac_lost;         /* mac_lost_errors */
uint32_t    glds_fddi_mac_token;        /* mac_tokens */
uint32_t    glds_fddi_mac_tvx_expired;  /* mac_tvx_expired */
uint32_t    glds_fddi_mac_late;         /* mac_late */
uint32_t    glds_fddi_mac_ring_op;      /* mac_ring_ops */
```

上述大多数统计变量均为表示发现特定事件次数的计数器。以下统计信息不表示次数：

`glds_speed` 接口的当前带宽估算（以 bps 为单位）。此对象应该包含那些带宽不变或无法进行准确估算的接口的标称带宽。

`glds_media` 硬件所使用的介质（连线）或连接器的类型。支持以下介质名称：

- GLDM_AUI
- GLDM_BNC
- GLDM_TP

- GLDM_10BT
- GLDM_100BT
- GLDM_100BTX
- GLDM_100BT4
- GLDM_RING4
- GLDM_RING16
- GLDM_FIBER
- GLDM_PHYMII
- GLDM_UNKNOWN

`glds_duplex` 接口的当前双工状态。支持的值包括 `GLD_DUPLEX_HALF` 和 `GLD_DUPLEX_FULL`。此外，还允许 `GLD_DUPLEX_UNKNOWN`。

GLDv2 函数参数

GLDv2 例程使用以下参数。

macinfo 指向 `gld_mac_info`(9S) 结构的指针。

macaddr 指向包含有效 MAC 地址的字符数组的开头的指针。此数组的长度由驱动程序在 `gld_mac_info`(9S) 结构的 `gldm_addrlen` 元素中指定。

multicastaddr 指向包含多点传送、组或功能地址 (functional address) 的字符数组的起始地址的指针。此数组的长度由驱动程序在 `gld_mac_info`(9S) 结构的 `gldm_addrlen` 元素中指定。

multiflag 指示是启用还是禁用多点传送地址接收的标志。可将此参数指定为 `GLD_MULTI_ENABLE` 或 `GLD_MULTI_DISABLE`。

promiscflag 指示要启用何种类型的混杂模式（如果存在）的标志。可将此参数指定为 `GLD_MAC_PROMISC_PHYS`、`GLD_MAC_PROMISC_MULTI` 或 `GLD_MAC_PROMISC_NONE`。

mp `gld_ioctl()` 使用 *mp* 作为指向包含要执行的 `ioctl` 的 STREAMS 消息块的指针。`gldm_send()` 使用 *mp* 作为指向包含要传输的包的 STREAMS 消息块的指针。`gld_recv()` 使用 *mp* 作为指向包含已接收包的消息块的指针。

stats 指向要使用统计计数器的当前值填充的 `gld_stats`(9S) 结构的指针。

q 指向要用于 `ioctl` 回复的 queue(9S) 结构的指针。

dip 指向设备的 `dev_info` 结构的指针。

name 设备接口名称。

GLDv2 入口点

入口点必须通过针对使用 GLDv2 的接口设计的特定于设备的网络驱动程序实现。

gld_mac_info(9S) 结构是用于在特定于设备的驱动程序与 GLDv2 模块之间进行通信的主结构。请参见 gld(7D) 手册页。此结构中的某些元素是指向此处所述的入口点的函数指针。特定于设备的驱动程序必须在调用 gld_register() 之前在其 attach(9E) 例程中初始化这些函数指针。

gldm_reset() 入口点

```
int prefix_reset(gld_mac_info_t *macinfo);
```

gldm_reset() 可将硬件重置为初始状态。

gldm_start() 入口点

```
int prefix_start(gld_mac_info_t *macinfo);
```

gldm_start() 可使设备生成中断。gldm_start() 还可使驱动程序调用 gld_recv()，从而将已接收的数据包传送到 GLDv2。

gldm_stop() 入口点

```
int prefix_stop(gld_mac_info_t *macinfo);
```

gldm_stop() 禁止设备生成任何中断，并阻止驱动程序为将数据包传送到 GLDv2 而调用 gld_recv()。GLDv2 依赖 gldm_stop() 例程来确保设备不再中断。gldm_stop() 必须成功执行此操作。此函数应该始终返回 GLD_SUCCESS。

gldm_set_mac_addr() 入口点

```
int prefix_set_mac_addr(gld_mac_info_t *macinfo, unsigned char *macaddr);
```

gldm_set_mac_addr() 可设置硬件用于接收数据的物理地址。利用此函数，可通过已传递的 MAC 地址 *macaddr* 对设备进行编程。如果当前没有足够的资源来执行请求，则 gldm_set_mac_add() 应该返回 GLD_NORESOURCES。如果不支持所请求的函数，则 gldm_set_mac_add() 应该返回 GLD_NOTSUPPORTED。

gldm_set_multicast() 入口点

```
int prefix_set_multicast(gld_mac_info_t *macinfo,
     unsigned char *multicastaddr, int multiflag);
```

gldm_set_multicast() 可启用和禁用设备级别的特定多点传送地址接收。如果将第三个参数 *multiflag* 设置为GLD_MULTI_ENABLE，则 gldm_set_multicast() 会将接口设置为使用

多点传送地址接收包。gldm_set_multicast() 将使用第二个参数所指向的多点传送地址。如果将 *multiflag* 设置为 GLD_MULTI_DISABLE，则允许驱动程序禁用指定的多点传送地址接收。

当 GLDv2 要启用或禁用多点传送、组或功能地址 (functional address) 接收时，便会调用此函数。GLDv2 不会做出有关设备如何支持多点传送并调用此函数以启用或禁用特定多点传送地址的假设。某些设备可能会使用散列算法和位掩码来启用多点传送地址集合。将允许此过程，并且 GLDv2 会过滤出所有多余的包。如果在设备级别禁用一个地址会导致禁用多个地址，则设备驱动程序应该保留所有必要信息。此方法可避免禁用 GLDv2 已启用但未禁用的地址。

不能调用 gldm_set_multicast() 来启用已启用的特定多点传送地址。同样，也不能调用 gldm_set_multicast() 来禁用当前未启用的地址。GLDv2 将跟踪针对同一多点传送地址发出的多个请求。GLDv2 仅在第一次请求启用特定多点传送地址或最后一次请求禁用特定多点传送地址时才调用驱动程序的入口点。如果当前没有足够的资源来执行请求，则函数应该返回 GLD_NORESOURCES。如果不支持所请求的函数，则函数应该返回 GLD_NOTSUPPORTED。

gldm_set_promiscuous() 入口点

```
int prefix_set_promiscuous(gld_mac_info_t *macinfo, int promiscflag);
```

gldm_set_promiscuous() 可启用和禁用混杂模式。当 GLDv2 要启用或禁用介质上的所有包接收时，便会调用此函数。还可以将此函数限制为针对介质上的多点传送包使用。如果将第二个参数 *promiscflag* 设置为 GLD_MAC_PROMISC_PHYS 的值，则函数会启用物理级别的混杂模式。物理级别的混杂模式会导致接收介质上的所有包。如果将 *promiscflag* 设置为 GLD_MAC_PROMISC_MULTI，则会启用所有多点传送包接收。如果将 *promiscflag* 设置为 GLD_MAC_PROMISC_NONE，则会禁用混杂模式。

在混杂多点传送模式下，无仅限多点传送混杂模式的设备的驱动程序必须将设备设置为物理混杂模式。此方法可确保接收所有多点传送包。在这种情况下，例程应该返回 GLD_SUCCESS。GLDv2 软件会过滤出所有多余的包。如果当前没有足够的资源来执行请求，则函数应该返回 GLD_NORESOURCES。如果不支持所请求的函数，gld_set_promiscuous() 应该返回 GLD_NOTSUPPORTED。

为了向前兼容，gldm_set_promiscuous() 例程应该处理所有 *promiscflag* 无法识别的值，如同这些值为 GLD_MAC_PROMISC_PHYS 一样。

gldm_send() 入口点

```
int prefix_send(gld_mac_info_t *macinfo, mblk_t *mp);
```

gldm_send() 可将要发送到设备的包进行排队以进行传输。将向此例程传递包含要发送的包的 STREAMS 消息。此消息可能包括多个消息块。send() 例程必须遍历消息中的所有消息块，以访问要发送的整个包。应对驱动程序进行适当设置，以便处理并跳过链

表中所有零长度的消息连续块。驱动程序还应该检查包是否未超过最大允许包大小。如有必要，驱动程序必须将包填充到最小允许包大小。如果发送例程成功传输包或对包排队，则应该返回 GLD_SUCCESS。

如果无法立即接受要传输的包，则发送例程应该返回 GLD_NORESOURCES。在这种情况下，GLDv2 会稍后重试。如果 gldm_send() 曾经返回 GLD_NORESOURCES，则驱动程序必须在稍后资源变得可用时调用 gld_sched()。此 gld_sched() 调用会通知 GLDv2 重试驱动程序先前无法对其进行排队以进行传输的包。（如果调用驱动程序的 gldm_stop() 例程，则会为驱动程序免除这种职责，直到驱动程序从 gldm_send() 例程返回 GLD_NORESOURCES 为止。不过，再次调用 gld_sched() 也不会导致错误操作。）

如果驱动程序的发送例程返回 GLD_SUCCESS，则驱动程序会负责释放不再需要的消息。如果硬件使用 DMA 直接读取数据，则驱动程序在硬件完全读取数据之前不得释放消息。在这种情况下，驱动程序可以释放中断例程中的消息。或者，驱动程序可以在将来发送操作的开始回收缓冲区。如果发送例程返回 GLD_SUCCESS 之外的任何内容，则驱动程序不得释放消息。如果在不存在到网络或链路伙伴的物理连接时调用 gldm_send()，则会返回 GLD_NOLINK。

gldm_intr() 入口点

```
int prefix_intr(gld_mac_info_t *macinfo);
```

当设备可能已中断时，便会调用 gldm_intr()。由于其他设备可以共享中断，因此，驱动程序必须检查设备状态以确定此设备是否实际导致了中断。如果驱动程序所控制的设备并未导致中断，则此例程必须返回 DDI_INTR_UNCLAIMED。否则，驱动程序必须修复中断并返回 DDI_INTR_CLAIMED。如果中断由成功接收包导致，则此例程应该将已接收的包放入类型为 M_DATA 的 STREAMS 消息中，并将此消息传递给 gld_recv()。

gld_recv() 将传入包向上游传递到网络协议栈的相应下一层。在调用 gld_recv 之前，此例程必须正确设置 STREAMS 消息的 b_rptr 和 b_wptr() 成员。

在调用 gld_recv() 期间，驱动程序应该避免持有互斥锁或其他锁。需要特别指出的是，在调用 gld_recv() 期间，不得持有传输线程可使用的锁。在某些情况下，调用 gld_recv() 的中断线程可发送传出包，这会导致调用驱动程序的 gldm_send() 例程。如果在调用 gld_recv() 时 gldm_send() 尝试获取 gldm_intr() 持有的互斥锁，则会由于存在递归互斥锁入口操作而出现紧急情况。在调用 gld_recv() 时，如果其他驱动程序入口点尝试获取驱动程序所持有的互斥锁，则会导致死锁。

中断代码应该针对所有错误递增统计计数器。这些错误包括分配已接收数据所需的缓冲区时出现的故障，以及所有特定于硬件的错误（如 CRC 错误或帧错误）。

gldm_get_stats() 入口点

```
int prefix_get_stats(gld_mac_info_t *macinfo, struct gld_stats *stats);
```

gldm_get_stats() 可收集硬件和/或驱动程序专用计数器的统计信息，并更新 *stats* 所指向的 gld_stats(9S) 结构。GLDv2 会针对统计信息请求调用此例程。GLDv2 会使用 gldm_get_stats() 机制从驱动程序获取与设备有关的统计信息，然后再编写统计信息请求回复。有关已定义的统计计数器的更多信息，请参见 gld_stats(9S)、gld(7D) 和 qreply(9F) 手册页。

gldm_ioctl() 入口点

```
int prefix_ioctl(gld_mac_info_t *macinfo, queue_t *q, mblk_t *mp);
```

gldm_ioctl() 可实现所有特定于设备的 ioctl 命令。如果驱动程序未实现任何 ioctl 函数，则允许此元素为 null。驱动程序负责在返回 GLD_SUCCESS 之前将消息块转换为 ioctl 回复消息，并调用 qreply(9F) 函数。此函数应该始终返回 GLD_SUCCESS。驱动程序应该根据需要报告要传递给 qreply(9F) 的消息中的所有错误。如果将 gldm_ioctl 元素指定为 NULL，则 GLDv2 会返回类型为 M_IOCNAK 的消息以及 EINVAL 错误。

GLDv2 返回值

GLDv2 中的某些入口点函数可以返回以下值（具体视上述限制而定）：

GLD_BADARG	函数检测到不适合的参数（如错误多点传送地址、错误 MAC 地址或错误包）时
GLD_FAILURE	出现硬件故障时
GLD_SUCCESS	成功时

GLDv2 服务例程

本节提供 GLDv2 服务例程的语法和说明。

gld_mac_alloc() 函数

```
gld_mac_info_t *gld_mac_alloc(dev_info_t *dip);
```

gld_mac_alloc() 可分配新的 gld_mac_info(9S) 结构并返回指向此结构的指针。可能会在 gld_mac_alloc() 返回之前初始化此结构的某些 GLDv2 专用元素。所有其他元素均初始化为 0。在将指向 gld_mac_info 结构的指针传递给 gld_register() 之前，设备驱动程序必须初始化某些结构成员，如 gld_mac_info(9S) 手册页中所述。

gld_mac_free() 函数

```
void gld_mac_free(gld_mac_info_t *macinfo);
```

gld_mac_free() 可释放先前由 gld_mac_alloc() 所分配的 gld_mac_info(9S) 结构。

gld_register() 函数

```
int gld_register(dev_info_t *dip, char *name, gld_mac_info_t *macinfo);
```

可通过设备驱动程序的 attach(9E) 例程调用 gld_register()。gld_register() 将基于 GLDv2 的设备驱动程序与 GLDv2 框架相链接。在调用 gld_register() 之前，设备驱动程序的 attach(9E) 例程使用 gld_mac_alloc() 来分配 gld_mac_info(9S) 结构，然后初始化若干结构元素。有关更多信息，请参见 gld_mac_info(9S)。成功调用 gld_register() 可执行以下操作：

- 链接特定于设备的驱动程序与 GLDv2 系统
- 使用 ddi_set_driver_private(9F) 设置特定于设备的驱动程序的专用数据指针以指向 macinfo 结构
- 创建从设备节点
- 返回 DDI_SUCCESS

传递给 gld_register() 的设备接口名称必须与存在于文件系统中的驱动程序模块的名称完全匹配。

如果 gld_register() 成功执行，则驱动程序的 attach(9E) 例程应该返回 DDI_SUCCESS。如果 gld_register() 没有返回 DDI_SUCCESS，则 attach(9E) 例程应该在调用 gld_register() 之前取消分配所有已分配的资源，然后返回 DDI_FAILURE。

gld_unregister() 函数

```
int gld_unregister(gld_mac_info_t *macinfo);
```

gld_unregister() 由设备驱动程序的 detach(9E) 函数进行调用，如果成功，会执行以下任务：

- 确保停止设备的中断，并在必要时调用驱动程序的 gldm_stop() 例程
- 删除从设备节点
- 从 GLDv2 系统中解除链接特定于设备的驱动程序
- 返回 DDI_SUCCESS

如果 gld_unregister() 返回 DDI_SUCCESS，则 detach(9E) 例程应该取消分配在 attach(9E) 例程中分配的所有数据结构，使用 gld_mac_free() 取消分配 macinfo 结构，并返回 DDI_SUCCESS。如果 gld_unregister() 没有返回 DDI_SUCCESS，则驱动程序的 detach(9E) 例程必须保持设备处于运行状态并返回 DDI_FAILURE。

gld_recv() 函数

```
void gld_recv(gld_mac_info_t *macinfo, mblk_t *mp);
```

驱动程序的中断处理程序可调用 gld_recv() 以向上游传递已接收的包。驱动程序必须构造并传递包含原始包的 STREAMS M_DATA 消息。gld_recv() 可确定哪些 STREAMS 队

列应该接收包的副本，并在必要时复制包。然后，gld_recv() 会在需要时设置 DL_UNITDATA_IND 消息的格式，并沿所有相应的流向上传递数据。

在调用 gld_recv() 期间，驱动程序应该避免持有互斥锁或其他锁。需要特别指出的是，在调用 gld_recv() 期间，不得持有传输线程可使用的锁。在某些情况下调用 gld_recv() 的中断线程可执行包括发送传出包的处理。传输包会导致调用驱动程序的 gldm_send() 例程。如果在调用 gld_recv() 时 gldm_send() 尝试获取 gldm_intr() 持有的互斥锁，则会由于存在递归互斥锁入口操作而出现紧急情况。在调用 gld_recv() 时，如果其他驱动程序入口点尝试获取驱动程序所持有的互斥锁，则会导致死锁。

gld_sched() 函数

```
void gld_sched(gld_mac_info_t *macinfo);
```

设备驱动程序可调用 gld_sched() 来重新安排已延迟的外发包。当驱动程序的 gldm_send() 例程返回 GLD_NORESOURCES 时，驱动程序必须调用 gld_sched() 以通知 GLDv2 框架重试先前无法发送的包。当资源变得可用之后，应该尽快调用 gld_sched()，以便 GLDv2 继续将外发包传递给驱动程序的 gldm_send() 例程。（如果调用了驱动程序的 gldm_stop() 例程，则在 gldm_send 返回 GLD_NORESOURCES() 之前，驱动程序不需要重试。不过，再次调用 gld_sched() 也不会导致错误操作。）

gld_intr() 函数

```
uint_t gld_intr(caddr_t);
```

gld_intr() 是 GLDv2 的主要中断处理程序。通常，将 gld_intr() 指定为设备驱动程序的 ddi_add_intr(9F) 调用中的中断例程。将中断处理程序的参数指定为 ddi_add_intr(9F) 调用中的 *int_handler_arg*。此参数必须是指向 gld_mac_info(9S) 结构的指针。gld_intr() 会在适当的情况下调用设备驱动程序的 gldm_intr() 函数，并将该指针传递给 gld_mac_info(9S) 结构。但是，要使用高级中断，驱动程序必须提供自身的高级中断处理程序，并在处理程序中触发软中断。在这种情况下，通常会将 gld_intr() 指定为 ddi_add_softintr() 调用中的软中断处理程序。gld_intr() 将返回适用于中断处理程序的值。

◆ ◆ ◆ 第 20 章

20 USB 驱动程序

本章介绍如何使用 Oracle Solaris 环境的 USBA 2.0 框架编写客户机 USB 设备驱动程序。本章讨论以下主题：

Oracle Solaris 环境中的 USB

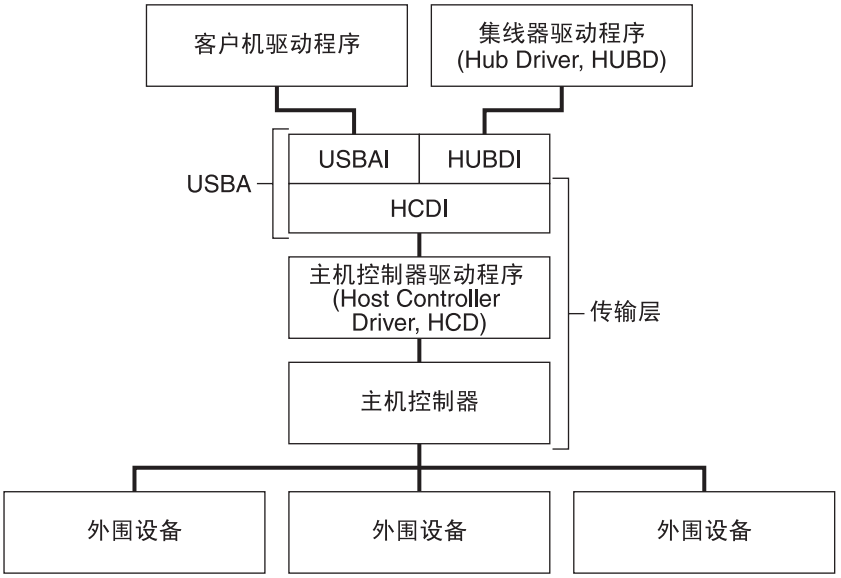
Oracle Solaris USB 体系结构包括 USBA 2.0 框架和 USB 客户机驱动程序。

USBA 2.0 框架

USBA 2.0 框架是向符合 USBA 标准的客户机驱动程序呈现 USB 设备的抽象表示方式的服务层。利用该框架，符合 USBA 标准的客户机驱动程序可以管理其 USB 设备。USBA 2.0 框架支持除高速同步管道之外的 USB 2.0 规范。有关 USB 2.0 规范的信息，请参见 `http://www.usb.org/home`。

USBA 2.0 框架与平台无关。下图显示了 Oracle Solaris USB 体系结构。在该图中，USBA 2.0 框架即是 USBA 层。此层通过与硬件无关的主机控制器驱动程序接口连接到特定于硬件的主机控制器驱动程序。主机控制器驱动程序通过其管理的主机控制器访问 USB 物理设备。

图 20–1 Oracle Solaris USB 体系结构



USBAI： Solaris USB 体系结构接口，USBA 和客户机驱动程序之间的接口

HUBDI： 集线器驱动程序接口

HCDI： 主机控制器驱动程序接口

USB 客户机驱动程序

USBA 2.0 框架本身不是设备驱动程序。本章介绍图 20–1 和图 20–2 中所示的客户机驱动程序。客户机驱动程序与各种类型的 USB 设备（如海量存储设备、打印机和人机接口设备）交互。集线器驱动程序是同时充当结点驱动程序的客户机驱动程序，它枚举其端口上的设备，并为这些设备创建 `devinfo` 节点，然后连接客户机驱动程序。本章并未介绍如何编写集线器驱动程序。

USB 驱动程序的结构与其他任何 Oracle Solaris 驱动程序相同。USB 驱动程序可以是块驱动程序、字符驱动程序或 STREAMS 驱动程序。USB 驱动程序遵循调用约定，并使用 Oracle Solaris OS 手册页第 9 节中说明的数据结构和例程。请参见 `Intro(9E)`、`Intro(9F)` 和 `Intro(9S)`。

USB 驱动程序与其他 Oracle Solaris 驱动程序的差别在于，USB 驱动程序通过调用 USBA 2.0 框架函数来访问设备，而不是直接访问设备。USBA 2.0 框架是对标准 Oracle Solaris DDI 例程的补充。请参见下图。

图 20–2 驱动程序和控制器接口

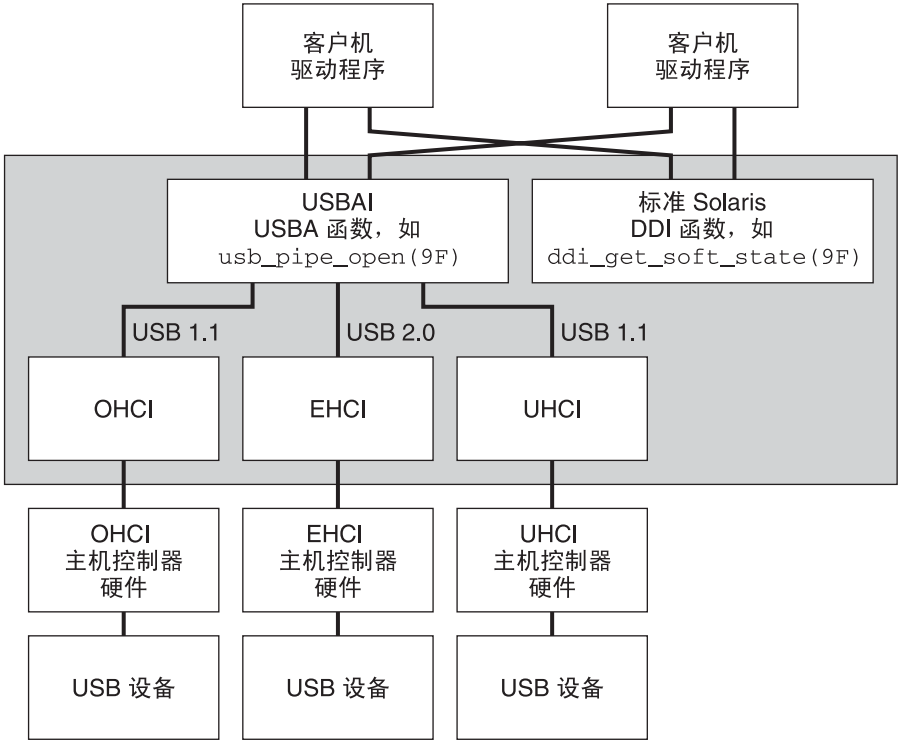


图 20–2 比图 20–1 更为详细地显示了接口。图 20–2 显示 USBA 是客户机驱动程序可以调用的内核子系统，正如客户机驱动程序可以调用 DDI 函数一样。

并非所有系统都具有图 20–2 中所示的所有主机控制器接口。OHCI（Open Host Controller Interface，开放式主机控制器接口）硬件在 SPARC 系统和第三方 USB PCI 卡上最常见。UHCI（Universal Host Controller Interface，通用主机控制器接口）硬件在 x86 系统上最常见。但是，OHCI 和 UHCI 硬件都可以在任何系统上使用。当存在 EHCI（Enhanced Host Controller Interface，增强型主机控制器接口）硬件时，EHCI 硬件与 OHCI 或 UHCI 位于相同的卡上，并共享相同的端口。

主机控制器、主机控制器驱动程序和 HCDI 构成由 USBA 控制的传输层。您不能直接调用 OHCI、EHCI 或 UHCI。可通过与平台无关的 USBA 接口间接调用它们。

绑定客户机驱动程序

本节讨论如何将驱动程序绑定到设备，还讨论具有单个接口的设备和具有多个接口的设备的兼容设备名称。

USB 设备如何显示在系统中

一个 USB 设备可以支持多种配置。在任何给定时间，只有一种配置处于活动状态。活动配置称为**当前配置**。

一种配置可以具有多个**接口**，其间可能具有针对一个函数分组两个或更多接口的介入式接口关联。一种配置的所有接口同时处于活动状态。不同的接口可由不同的设备驱动程序操作。

接口可以使用**替代设置**以不同的方式在主机系统中呈现自己。对于任何给定接口只能有一种替代设置处于活动状态。

每种替代设置通过**端点**提供设备访问。每个端点都有特定用途。主机系统通过建立到端点的信道来与设备通信。此信道称为**管道**。

USB 设备和 Oracle Solaris 设备树

如果 USB 设备具有一种配置、一个接口，没有设备类，则可将该设备表示为单个**设备节点**。如果 USB 设备具有多个接口，则可将该设备表示为分层设备结构。在分层设备结构中，每个接口的设备节点是顶层设备节点的子节点。例如，音频设备即是具有多个接口的设备，该设备向主机同时呈现为音频控制和音频流两种接口。音频控制接口和音频流接口可以分别由各自的驱动程序控制。

兼容设备名称

Oracle Solaris 软件基于每个设备中存储的标识信息为 USB 绑定生成有序的兼容设备名称列表。此信息包括设备类、子类、供应商 ID、产品 ID、修订版和协议。有关 USB 类和子类的列表，请参见 `http://www.usb.org/home`。

采用此名称分层结构，可以在没有特定于设备的驱动程序时，绑定到相对较常用的驱动程序。特定于类的驱动程序即是常规驱动程序。以 `usbif` 开头的设备名称指定单个接口的设备。有关示例，请参见示例 20–1。USBA 2.0 框架定义设备的所有兼容名称。使用 `prtconf` 命令可显示这些设备名称，如示例 20–2 中所示。

以下示例显示了 USB 鼠标设备的兼容设备名称。此鼠标设备表示完全由单个驱动程序操作的组合节点。USBA 2.0 框架为此设备节点指定了示例中所示的名称（按所示顺序）。

示例 20–1 USB 鼠标的兼容设备名称

```
1. ’usb430,100.102’      Vendor 430, product 100, revision 102
2. ’usb430,100’      Vendor 430, product 100
3. ’usbif430,class3.1.2’ Vendor 430, class 3, subclass 1, protocol 2
4. ’usbif430,class3.1’   Vendor 430, class 3, subclass 1
5. ’usbif430,class3’     Vendor 430, class 3
6. ’usbif,class3.1.2’    Class 3, subclass 1, protocol 2
7. ’usbif,class3.1’      Class 3, subclass 1
8. ’usbif,class3’    Class 3
```

请注意，上面示例中的名称按从最具体到最常规的顺序进行排列。第 1 项仅绑定到特定供应商的特定产品的特定修订版。第 3、4 和 5 项用于由供应商 430 生产的类 3 设备。第 6、7 和 8 项用于任何供应商生产的类 3 设备。绑定过程将按从上到下的顺序查找名称匹配项。要进行绑定，必须将驱动程序添加到其别名与上述其中一个名称匹配的系统。要获取在添加驱动程序时要绑定到的兼容设备名称的列表，请在 `prtconf -vp` 命令的输出中检查设备的 `compatible` 属性。

以下示例显示了键盘和鼠标的兼容属性列表。使用 `prtconf -D` 命令可显示绑定的驱动程序。

示例 20–2 列显配置命令显示的兼容设备名称

```
# prtconf -vD | grep compatible
            compatible: ’usb430,5.200’ + ’usb430,5’ + ’usbif430,class3.1.1’
+ ’usbif430,class3.1’ + ’usbif430,class3’ + ’usbif,class3.1.1’ +
’usbif,class3.1’ + ’usbif,class3’
            compatible: ’usb2222,2071.200’ + ’usb2222,2071’ +
’usbif2222,class3.1.2’ + ’usbif2222,class3.1’ + ’usbif2222,class3’ +
’usbif,class3.1.2’ + ’usbif,class3.1’ + ’usbif,class3’
```

使用最具体的名称可以更准确地确定一个设备或一组设备的驱动程序。要绑定为特定产品的特定修订版编写的驱动程序，请尽可能使用最具体的名称匹配项。例如，如果您有由供应商 430 为其产品 100 的修订版 102 编写的 USB 鼠标驱动程序，则可以使用以下命令将该驱动程序添加到系统中：

```
add_drv -n -i ’"usb430,100.102"’ specific_mouse_driver
```

要添加为供应商 430 的任何 USB 鼠标（类 3、子类 1、协议 2）编写的驱动程序，请使用以下命令：

```
add_drv -n -i ’"usbif430,class3.1.2"’ more_generic_mouse_driver
```

如果安装这两个驱动程序并连接兼容设备，则系统会将正确的驱动程序绑定到所连接的设备。例如，如果安装这两个驱动程序，并连接供应商 430、型号 100、修订版 102 的设备，则此设备将绑定到 `specific_mouse_driver`。如果连接供应商 430、型号 98 的设备，则此设备将绑定到 `more_generic_mouse_driver` 。如果连接其他供应商的鼠标，则此设备也将绑定到 `more_generic_mouse_driver`。如果有多个驱动程序可供特定设备使用，则驱动程序绑定框架将选择与兼容名称列表中第一个兼容名称匹配的驱动程序。

具有多个接口的设备

复合设备是支持多个接口的设备。复合设备的每个接口都有一个兼容名称列表。此兼容名称列表可确保将最有效的驱动程序绑定到该接口。最常规的多接口项是 `usb,device`。

对于 USB 音频复合设备，兼容名称如下：

```
1. 'usb471,101.100'     Vendor 471, product 101, revision 100
2. 'usb471,101'         Vendor 471, product 101
3. 'usb,device'         Generic USB device
```

名称 `usb,device` 是可表示任何整个 USB 设备的兼容名称。如果没有其他驱动程序请求该整个设备，则 `usb_mid(7D)` 驱动程序（USB 多接口驱动程序）将绑定到 `usb,device` 设备节点。`usb_mid` 驱动程序为物理设备的每个接口创建一个子设备节点。`usb_mid` 驱动程序还为每个接口生成一组兼容名称。生成的所有这些兼容名称都以 `usbif` 开头。系统将使用生成的这些兼容名称为每一个接口查找最佳的驱动程序。通过这种方法，可以将一个物理设备的不同接口绑定到不同的驱动程序。

例如，`usb_mid` 驱动程序通过多接口音频设备的 `usb,device` 节点名称绑定到该音频设备。然后 `usb_mid` 驱动程序创建特定于接口的设备节点。这些特定于接口的设备节点中的每个节点都有各自的兼容名称列表。对于音频控制接口节点，兼容名称列表可能类似于下例中所示的列表。

示例 20–3 USB 音频兼容设备名称

```
1. 'usbif471,101.100.config1.0' Vend 471, prod 101, rev 100, cnfg 1, iface 0
2. 'usbif471,101.config1.0'     Vend 471, product 101, config 1, interface 0
3. 'usbif471,class1.1.0'    Vend 471, class 1, subclass 1, protocol 0
4. 'usbif471,class1.1'      Vend 471, class 1, subclass 1
5. 'usbif471,class1'        Vend 471, class 1
6. 'usbif,class1.1.0'       Class 1, subclass 1, protocol 0
7. 'usbif,class1.1'         Class 1, subclass 1
8. 'usbif,class1'           Class 1
```

使用以下命令可将特定于供应商、特定于设备的客户机驱动程序（名为 `vendor_model_audio_usb`）绑定到特定于供应商、特定于设备的配置 1、接口 0 的接口兼容名称，如示例 20–3 中所示。

```
add_drv -n -i '"usbif471,101.config1.0"' vendor_model_audio_usb
```

使用以下命令可将名为 `audio_class_usb_if_driver` 的类驱动程序绑定到较常规的类 1、子类 1 的接口兼容名称，如示例 20–3 中所示：

```
add_drv -n -i '"usbif,class1.1"' audio_class_usb_if_driver
```

使用 `prtconf -D` 命令可显示设备及其驱动程序的列表。在以下示例中，`prtconf -D` 命令显示 `usb_mid` 驱动程序管理 `audio` 设备。`usb_mid` 驱动程序将 `audio` 设备拆分为多个接口。每个接口在 `audio` 设备名称下以缩进方式列出。对于缩进列表中所示的每个接口，`prtconf -D` 命令显示了哪个驱动程序管理该接口。

```
audio, instance #0 (driver name: usb_mid)
    sound-control, instance #2 (driver name: usb_ac)
    sound, instance #2 (driver name: usb_as)
    input, instance #8 (driver name: hid)
```

包含接口关联描述符的设备

如果设备包括接口关联描述符，则设备树可以在以下三个级别上进行解析：

- 如果没有特定于供应商或类的驱动程序可用，则 usb_mid(7D) USB 多接口驱动程序将绑定到复合设备的设备级别节点。
- 客户机驱动程序绑定到接口关联节点。
- 如果未找到任何客户机驱动程序，则缺省情况下会绑定 usb_ia(7D) USB 接口关联驱动程序。然后，客户机驱动程序可以绑定到此接口关联的接口级别。

usb_mid 驱动程序为每个 ia 创建一个 ia（接口关联）节点。兼容的 ia 节点名称通常以 usbia 开头。名称 usb,ia 是可将任何 ia 表示为兼容名称尾部的一个兼容名称。如果没有任何其他驱动程序申请此 ia，则 usb_ia 驱动程序将绑定到 ia 节点。usb_ia 驱动程序为每个接口创建一个子节点。作为 ia 节点子节点的接口节点与作为设备节点子节点的接口节点具有相同属性。

示例 20–4 USB 视频接口关联兼容名称

```
1. 'usbia46d,8c9.5.config1.0' vend 46d, prod 8c9, rev 5, cnfg 1, first_if_in_ia 0
2. 'usbia46d,8c9.config1.0'   vend 46d, prod 8c9, cnfg 1, first_if_in_ia 0
3. 'usbia46d,classe.3.0'      vend 46d, class e, subclass 3, protocol 0
4. 'usbia46d,classe.3'        vend 46d, class e, subclass 3
5. 'usbia46d,classe'          vend 46d, class e
6. 'usbia,classe.3.0'         class e, subclass 3, protocol 0
7. 'usbia,classe.3'           class e, subclass 3
8. 'usbia,classe'             class e
9. 'usb,ia'                   by default
```

使用以下命令可将特定于供应商和设备的、名为 vendor_model_video_usb 的客户机驱动程序绑定到特定于供应商和设备的配置 1 的 first_if_in_ia 兼容名称，如示例 20–4 中所示：

```
add_drv -n -i '"usbia46d,8c9.config1.0"' vendor_model_video_usb
```

使用以下命令可将名为 video_class_usb_ia_driver 的类驱动程序绑定到较常规的 e 类兼容名称，如示例 20–4 中所示：

```
add_drv -n -i '"usbia,classee"' video_class_usb_ia_driver
```

在以下示例中，prtconf -D 命令显示了 Web 摄像头的设备树，其中包含 video 和 audio 的 ia。usb_mid 驱动程序管理设备并分别为 video 和 audio 创建两个 ia。视频驱动程序 usbvc 绑定到视频 ia，而音频驱动程序绑定到音频 ia 的接口。

```
miscellaneous, instance #28 (driver name: usb_mid)
    video, instance #24 (driver name: usbvc)
    audio, instance #30 (driver name: usb_ia)
        sound-control, instance #38 (driver name: usb_ac)
        sound, instance #47 (driver name: usb_as)
```

检查设备驱动程序绑定

文件 /etc/driver_aliases 包含对应于系统中已存在的绑定的项。/etc/driver_aliases 文件的每一行都有一个驱动程序名称，后面依次跟随一个空格和一个设备名称。使用此文件可检查现有的设备驱动程序绑定。

注 – 请不要手动编辑 /etc/driver_aliases 文件。使用 add_drv(1M) 命令可建立绑定。使用 update_drv(1M) 命令可更改绑定。

基本设备访问

本节介绍如何访问 USB 设备以及如何注册客户机驱动程序。本节还将讨论描述符树。

连接客户机驱动程序之前

在连接客户机驱动程序之前发生下列事件：

1. PROM (OBP/BIOS) 和 USBA 框架在连接任何客户机驱动程序之前获取访问设备的权限。
2. 集线器驱动程序将在其集线器的每个端口上探测设备的标识和配置。
3. 将打开每个设备的缺省控制管道，并探测每个设备的设备描述符。
4. 使用设备和接口描述符为每个设备构建兼容名称属性。

兼容名称属性定义可单独绑定到客户机驱动程序的设备的不同部分。可以将客户机驱动程序绑定到整个设备或仅绑定到一个接口。请参见第 400 页中的 “绑定客户机驱动程序”。

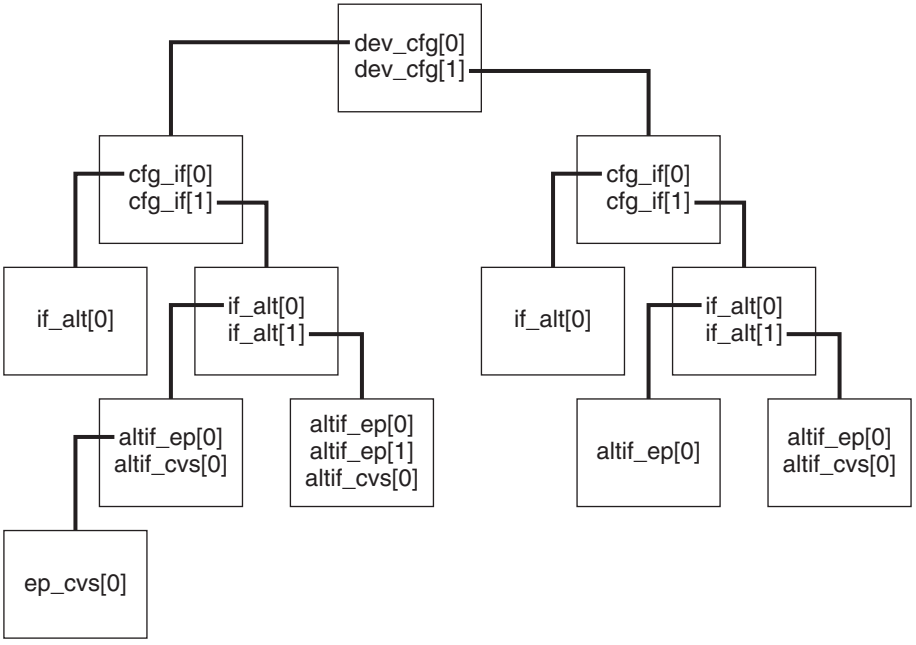
描述符树

解析描述符涉及在自然边界对齐结构成员，以及将结构成员转换为主机 CPU 的字节序。解析后的标准 USB 配置描述符、接口描述符和端点描述符可用于每种配置的分层树格式的客户机驱动程序。任何特定于原始类或特定于供应商的描述符信息也可用于同一分层树中的客户机驱动程序。

调用 usb_get_dev_data(9F) 函数可检索分层描述符树。usb_get_dev_data (9F) 手册页的“另请参见”部分列出了每个标准 USB 描述符的手册页。使用 usb_parse_data(9F) 函数可解析原始描述符信息。

具有两种配置的设备的描述符树可能与下图中所示的树类似。

图 20–3 分层 USB 描述符树



上图中所示的 `dev_cfg` 数组包含对应于相应配置的节点。每个节点包含以下信息：

- 解析的配置描述符
- 指向描述符数组的指针，这些描述符对应于该配置的接口
- 指向特定于类或特定于供应商的原始数据数组（如果存在）的指针

表示第二个索引配置的第二个接口的节点位于图中的 `dev_cfg[1].cfg_if[1]` 位置。该节点包含表示该接口的替代设置的节点数组。USB 描述符的分层结构通过该树传播。字符串描述符数据中的 ASCII 字符串连接到 USB 规范说明的存在这些字符串的位置。

配置数组是非稀疏数组，按配置索引进行索引。第一个有效配置（配置 1）是 `dev_cfg[0]`。接口和替代设置具有与其编号对齐的索引。对于每个替代设置的端点，都以连续方式进行索引。每个替代设置的第一个端点位于索引 0 位置。

此编号方案使得很容易对树进行遍历。例如，端点索引为 0、替代项为 0、接口为 1、配置索引为 1 的原始描述符数据位于以下路径定义的节点：

```
dev_cfg[1].cfg_if[1].if_alt[0].altif_ep[0].ep_descr
```

另一种直接使用描述符树的方法是使用 usb_lookup_ep_data(9F) 函数。usb_lookup_ep_data(9F) 函数采用接口、替代项、端点、端点类型和指令作为参数。您可以使用 usb_lookup_ep_data(9F) 函数遍历描述符树以获取特定端点。有关更多信息，请参见 usb_get_dev_data(9F) 手册页。

注册驱动程序以获取设备访问权限

在 USBA 2.0 框架中，客户机驱动程序执行的前两个调用是对 usb_client_attach(9F) 函数和 usb_get_dev_data(9F) 函数的调用。这两个调用来自客户机驱动程序的 attach(9E) 入口点。在调用 usb_get_dev_data(9F) 函数之前，必须先调用 usb_client_attach(9F) 函数。

usb_client_attach(9F) 函数用于向 USBA 2.0 框架注册客户机驱动程序。usb_client_attach(9F) 函数用于强制进行版本控制。所有客户机驱动程序源文件必须使用下列行开头：

```
#define USBDRV_MAJOR_VER        2
#define USBDRV_MINOR_VER        minor-version
#include <sys/usb/usba.h>
```

minor-version 的值必须小于或等于 USBA_MINOR_VER。符号 USBA_MINOR_VER 在 <sys/usb/usbai.h> 头文件中定义。<sys/usb/usbai.h> 头文件通过 <sys/usb/usba.h> 头文件包含进来。

USBDRV_VERSION 是根据 USBDRV_MAJOR_VERSION 和 USBDRV_MINOR_VERSION 生成版本号的宏。usb_client_attach() 的第二个参数必须是 USBDRV_VERSION。如果第二个参数不是 USBDRV_VERSION，或者如果 USBDRV_VERSION 反映的是无效的版本，则 usb_client_attach() 函数将失败。此限制可确保编程接口兼容性。

usb_get_dev_data() 函数可返回正确管理 USB 设备所需的信息。例如，usb_get_dev_data() 函数返回以下信息：

- 缺省控制管道
- 互斥锁初始化中使用的 *iblock_cookie*（请参见 mutex_init(9F)）
- 解析的设备描述符
- ID 字符串
- 第 404 页中的 “描述符树”中所述的树分层结构

必须调用 usb_get_dev_data() 函数。调用 usb_get_dev_data() 是检索缺省控制管道以及检索互斥锁初始化所需的 *iblock_cookie* 的唯一方法。

调用 usb_get_dev_data() 后，客户机驱动程序的 attach(9E) 例程通常会将所需的描述符和 数据从描述符树复制到驱动程序的软状态。复制到软状态的端点描述符在以后打开到这些端点的管道时会用到。attach(9E) 例程通常在复制描述符后调用 usb_free_descr_tree(9F) 以释放描述符树。或者，可以选择保留描述符树，且不复制描述符。

可以为 usb_get_dev_data(9F) 函数指定以下三个解析级别之一，以请求想要返回的描述符树的广度。如果驱动程序需要绑定到的对象不仅仅是设备，则需要更大的树广度。

- USB_PARSE_LVL_IF。如果客户机驱动程序绑定到特定接口，则驱动程序仅需要对应于该接口的描述符。在 usb_get_dev_data() 调用中指定 USB_PARSE_LVL_IF 作为解析级别将仅检索这些描述符。
- USB_PARSE_LVL_CFG。如果客户机驱动程序绑定到整个设备，请指定 USB_PARSE_LVL_CFG 以检索当前配置的所有描述符。
- USB_PARSE_LVL_ALL。指定 USB_PARSE_LVL_ALL 可检索所有配置的所有描述符。例如，要使用 usb_print_descr_tree(9F) 列显设备的所有配置的描述符转储，需要此最大的树广度。

客户机驱动程序的 detach(9E) 例程必须调用 usb_free_dev_data(9F) 函数来释放由 usb_get_dev_data() 函数分配的所有资源。usb_free_dev_data() 函数接受已使用 usb_free_descr_tree() 函数释放了描述符树的句柄。客户机驱动程序的 detach() 例程还必须调用 usb_client_detach(9F) 函数以释放由 usb_client_attach(9F) 函数分配的所有资源。

设备通信

USB 设备的工作方式是通过称为**管道**的信道来传递请求。只有管道处于打开状态时您才能提交请求。也可以刷新、查询和关闭管道。本节讨论管道、数据传输和回调以及数据请求。

USB 端点

与四种类型的 USB 端点通信的四种类型的管道包括：

- **控制**。控制管道主要用于发送命令和检索状态。控制管道适用于小型结构化数据的非定期、主机启动的请求和响应通信。控制管道是双向的。缺省管道为控制管道。请参见第 408 页中的 “缺省管道”。
- **批量传输**。批量传输管道主要用于数据传输。批量传输管道可实现大量数据的可靠传输。批量传输管道并不一定会进行数据的及时传输。批量传输管道是单向的。
- **中断**。中断管道可为少量的非结构数据提供及时而可靠的通信。通常，会对中断输入管道启动定期轮询。当设备中存在数据时，中断输入管道将数据返回到主机。一些设备具有中断输出管道。中断输出管道采用与中断输入管道相同的及时而可靠的“中断管道”特性将数据传输到设备。中断管道是单向的。
- **同步**。同步管道为传输速率恒定、与时间相关的数据（如音频设备数据）提供通道。出现错误时不会尝试重新传输数据。同步管道是单向的。

有关对应于这些端点的传输类型的更多信息，请参见 USB 2.0 规范的第 5 章或参见第 410 页中的 “请求”。

缺省管道

每个 USB 设备都有称为**缺省**端点的特殊控制端点。其信道称为缺省管道。大多数（可能并非所有）设备的设置都通过此管道进行。许多 USB 设备使用此管道作为其唯一的控制管道。

usb_get_dev_data(9F) 函数为客户机驱动程序提供缺省控制管道。此管道将会被预先打开以适应在打开其他管道之前需要的任何特殊设置。此缺省控制管道的特殊性表现在以下方面：

- 此管道是共享的。操作同一设备其他接口的驱动程序使用相同的缺省控制管道。USBA 2.0 框架仲裁此管道在不同驱动程序之间的使用。
- 此管道不能由客户机驱动程序打开、关闭或重置。之所以存在此限制，是因为管道是共享的。
- 出现异常时将会自动清除此管道。

其他管道（包括其他控制管道）必须明确打开且仅限独占打开。

管道状态

管道处于以下状态之一：

- USB_PIPE_STATE_IDLE
  - 所有控制管道、批量传输管道、中断输出管道和同步输出管道：没有正在进行的请求。
  - 中断输入管道和同步输入管道：没有正在进行的轮询。
- USB_PIPE_STATE_ACTIVE
  - 所有控制管道、批量传输管道、中断输出管道和同步输出管道：管道正在传输数据或 I/O 请求处于活动状态。
  - 中断输入管道和同步输入管道：轮询处于活动状态。
- USB_PIPE_STATE_ERROR。出现错误。如果此管道不是缺省管道，而且未启用自动清除，则客户机驱动程序必须调用 usb_pipe_reset(9F) 函数。
- USB_PIPE_STATE_CLOSING。正在关闭管道。
- USB_PIPE_STATE_CLOSED。已关闭管道。

调用 usb_pipe_get_state(9F) 函数可检索管道的状态。

打开管道

要打开管道，请将对应于要打开的管道的端点描述符传递给 usb_pipe_open(9F) 函数。使用 usb_get_dev_data(9F) 和 usb_lookup_ep_data(9F) 函数可检索描述符树中的端点描述符。usb_pipe_open(9F) 函数将句柄返回给管道。

打开管道时必须指定管道策略。管道策略包含并发异步操作的估计数量，这些并发异步操作要求使用此管道将需要的独立线程。线程的估计数量是回调期间可能发生的并行操作的数量。此估计值必须至少为 2。有关管道策略的更多信息，请参见 usb_pipe_open(9F) 手册页。

关闭管道

驱动程序必须使用 usb_pipe_close(9F) 函数关闭缺省管道之外的管道。usb_pipe_close(9F) 函数使管道中的所有剩余请求得以完成。然后该函数还留出一秒钟的时间让这些请求的所有回调得以完成。

数据传输

对于所有管道类型，编程模型如下：

1. 分配请求。
2. 使用管道传输函数之一提交该请求。请参见 usb_pipe_bulk_xfer(9F)、usb_pipe_ctrl_xfer(9F)、usb_pipe_intr_xfer(9F) 和 usb_pipe_isoc_xfer(9F) 手册页。
3. 等待完成通知。
4. 释放请求。

有关请求的更多信息，请参见第 410 页中的 “请求”。以下各节介绍各种请求类型的特性。

同步传输、异步传输和回调

传输分为同步传输和异步传输。同步传输在完成之前将会一直阻塞。异步传输完成时，将向客户机驱动程序发送回调。在 *flags* 参数中设置了 USB_FLAGS_SLEEP 标志时调用的传输函数大多数是同步的。

连续传输（如轮询）和同步传输不能是同步的。为了进行连续传输而对传输函数进行的调用（设置了 USB_FLAGS_SLEEP 标志）将会阻塞，目的只是等待资源，然后开始传输。

同步传输是要设置的最简单的传输，因为同步传输不要求任何回调函数。同步传输函数将返回传输开始状态，即使同步传输函数在完成传输前一直阻塞也是如此。完成时，可以在请求的完成原因字段和回调标志字段中查找有关传输状态的其他信息。下面将讨论完成原因字段和回调标志字段。

如果未在 *flags* 参数中指定 USB_FLAGS_SLEEP 标志，则该传输操作是异步的。此规则的例外是同步传输。异步传输操作将设置并启动传输，然后在传输完成前返回。异步传输操作将返回传输开始状态。客户机驱动程序通过回调处理程序接收传输完成状态。

回调处理程序是在异步传输完成时调用的函数。不要设置不进行回调的异步传输。两种类型的回调处理程序是正常完成处理程序和异常处理程序。您可以指定一个在这两种情况下要调用的处理程序。

- **正常完成**。可调用正常完成回调处理程序以通知传输正常完成。
- **异常**。可调用异常回调处理程序以通知传输未正常完成，然后对错误进行处理。

完成处理程序和异常处理程序将传输请求作为参数接收。异常处理程序在请求中使用完成原因和回调状态来了解所发生的情况。完成原因 (`usb_cr_t`) 指示原始事务是如何完成的。例如，完成原因 `USB_CR_TIMEOUT` 指示传输超时。又如，如果 USB 设备在使用时被移除，则客户机驱动程序可能接收 `USB_CR_DEV_NOT_RESP` 作为其未完成请求的完成原因。回调状态 (`usb_cb_flags_t`) 指示 USBA 框架为修正这种情况所执行的操作。例如，回调状态 `USB_CB_STALL_CLEARED` 指示 USBA 框架清除了运行延迟条件。有关完成原因的更多信息，请参见 `usb_completion_reason(9S)` 手册页。有关回调状态标志的更多信息，请参见 `usb_callback_flags(9S)` 手册页。

运行请求的回调上下文和管道策略会对在回调中可以执行的操作实施一些限制。

- **回调上下文**。大多数回调在内核上下文中执行，通常可以阻塞。一些回调在中断上下文中执行，它们不能阻塞。可在回调标志中设置 `USB_CB_INTR_CONTEXT` 标志以表示中断上下文。有关回调上下文的更多信息和有关阻塞的详细信息，请参见 `usb_callback_flags(9S)` 手册页。
- **管道策略**。管道策略对并发异步操作的提示限制可以并行运行的操作数，包括通过回调处理程序执行的操作。阻塞同步操作将计为一个操作。有关管道策略的更多信息，请参见 `usb_pipe_open(9F)` 手册页。

请求

本节讨论请求结构，以及分配和取消分配不同类型的请求。

分配和取消分配请求

请求以初始化的请求结构的形式实现。每种不同的端点类型接受不同类型的请求。每种类型的请求有不同的请求结构类型。下表显示了每种类型请求的结构类型。此表还列出了可用于分配和释放每种类型结构的函数。

表 20–1 请求初始化

管道或端点类型	请求结构	请求结构分配函数	请求结构释放函数
控制	`usb_ctrl_req_t`（请参见 `usb_ctrl_request(9S)` 手册页）	`usb_alloc_ctrl_req(9F)`	`usb_free_ctrl_req(9F)`
批量传输	`usb_bulk_req_t`（请参见 `usb_bulk_request(9S)` 手册页）	`usb_alloc_bulk_req(9F)`	`usb_free_bulk_req(9F)`

表 20–1 请求初始化 （续）

管道或端点类型	请求结构	请求结构分配函数	请求结构释放函数
中断	usb_intr_req_t（请参见 usb_intr_request(9S) 手册页）	usb_alloc_intr_req(9F)	usb_free_intr_req(9F)
同步	usb_isoc_req_t（请参见 usb_isoc_request(9S) 手册页）	usb_alloc_isoc_req(9F)	usb_free_isoc_req(9F)

下表列出了可用于每种类型请求的传输函数。

表 20–2 请求传输设置

管道或端点类型	传输函数
控制	usb_pipe_ctrl_xfer(9F)、usb_pipe_ctrl_xfer_wait(9F)
批量传输	usb_pipe_bulk_xfer(9F)
中断	usb_pipe_intr_xfer(9F)、usb_pipe_stop_intr_polling(9F)
同步	usb_pipe_isoc_xfer(9F)、usb_pipe_stop_isoc_polling(9F)

分配和取消分配请求的过程如下：

1. 使用相应的分配函数为所需的请求类型分配请求结构。表 20–1 中列出了请求结构分配函数的手册页。
2. 初始化结构中所需的任何字段。有关更多信息，请参见第 411 页中的 “请求特性和字段”或相应的请求结构手册页。表 20–1 中列出了请求结构的手册页。
3. 完成数据传输时，使用相应的释放函数释放请求结构。表 20–1 中列出了请求结构释放函数的手册页。

请求特性和字段

所有请求的数据都以消息块的形式传递，这样，无论驱动程序是 STREAMS 驱动程序、字符驱动程序还是块驱动程序，都将统一地对数据进行处理。消息块类型 mblk_t 在 mblk(9S) 手册页中进行了介绍。DDI 提供了多个用于处理消息块的例程。示例包括 allocb(9F) 和 freemsg(9F)。要了解用于处理消息块的其他例程，请参见 allocb(9F) 和 freemsg(9F) 手册页的“另请参见”部分。此外，还可以参见《STREAMS Programming Guide》。

所有传输类型中都包括以下请求字段。在每个字段名称中，*xxxx* 的可能值包括：*ctrl*、*bulk*、*intr* 或 *isoc*。

xxxx_client_private 此字段值是一个指针，适用于将与请求一起在客户机驱动程序中传递的内部数据。此指针不用于将数据传输到设备。

xxxx_attributes	此字段值是一组传输属性。虽然此字段对所有请求结构通用，但对于每种传输类型，此字段的初始化稍有不同。有关更多信息，请参见相应的请求结构手册页。表 20–1 中列出了这些手册页。另请参见 usb_request_attributes(9S) 手册页。
xxxx_cb	此字段值是正常传输完成的回调函数。如果异步传输在没有错误的情况下完成，则会调用此函数。
xxxx_exc_cb	此字段值是错误处理的回调函数。仅当异步传输完成且出现错误时，才会调用此函数。
xxxx_completion_reason	此字段存放传输本身的完成状态。如果出现错误，此字段将显示出现错误的内容。有关更多信息，请参见 usb_completion_reason(9S) 手册页。此字段由 USBA 2.0 框架更新。
xxxx_cb_flags	此字段列出在调用回调处理程序之前 USBA 2.0 框架 所采取的恢复操作。USB_CB_INTR_CONTEXT 标志指示回调是否在中断上下文中运行。有关更多信息，请参见 usb_callback_flags(9S) 手册页。此字段由 USBA 2.0 框架更新。

以下各节介绍针对四种不同传输类型的不同请求字段。其中介绍如何初始化这些结构字段，还介绍有关属性和参数的各种组合的限制。

控制请求

使用控制请求可沿控制管道向下启动消息传输。您可以手动设置传输，如下所示。也可以使用 usb_pipe_ctrl_xfer_wait(9F) 包装函数设置并发送同步传输。

客户机驱动程序必须初始化 *ctrl_bmRequestType*、*ctrl_bRequest*、*ctrl_wValue*、*ctrl_wIndex* 和 *ctrl_wLength* 字段，如 USB 2.0 规范中所述。

必须将请求的 *ctrl_data* 字段初始化为指向数据缓冲区。将一个正值作为缓冲区 *len* 传递时，usb_alloc_ctrl_req(9F) 函数将初始化此字段。当然，还必须初始化缓冲区以便进行任何外发传输。在所有情况下，完成传输时，客户机驱动程序必须释放请求。

可以对多个控制请求进行排队。排队的请求中可以包括同步请求和异步请求。

ctrl_timeout 字段定义等待要被处理的请求的最长时间，但不包括在队列中的等待时间。此字段适用于同步和异步请求。*ctrl_timeout* 字段中指定的值以秒为单位。

如果出现异常，*ctrl_exc_cb* 字段接受要调用的函数的地址。usb_ctrl_request(9S) 手册页中指定了此异常处理程序的参数。异常处理程序的第二个参数是 usb_ctrl_req_t 结构。通过将请求结构作为参数传递，异常处理程序可以检查该请求的 *ctrl_completion_reason* 和 *ctrl_cb_flags* 字段，以确定最佳恢复操作。

USB_ATTRS_ONE_XFER 和 USB_ATTRS_ISOC_* 标志对所有控制请求而言都是无效属性。USB_ATTRS_SHORT_XFER_OK 标志仅对主机绑定的请求有效。

批量传输请求

使用批量传输请求可发送非时间关键数据。批量传输请求可以接纳多个要完成的 USB 帧，具体取决于总体总线负载。

所有请求必须接收已初始化的消息块。有关 mblk_t 消息块类型的说明，请参见 mblk(9S) 手册页。此消息块将提供数据或存储数据，具体取决于传输方向。有关更多详细信息，请参阅 usb_bulk_request(9S) 手册页。

USB_ATTRS_ONE_XFER 和 USB_ATTRS_ISOC_* 标志对所有批量传输请求而言都是无效属性。USB_ATTRS_SHORT_XFER_OK 标志仅对主机绑定的请求有效。

usb_pipe_get_max_bulk_transfer_size(9F) 函数指定每个请求的最大字节数。检索到的值可以是客户机驱动程序的 minphys(9F) 例程中使用的最大值。

可以对多个批量传输请求进行排队。

中断请求

中断请求通常用于定期传入数据。中断请求定期在设备中轮询数据。但是，USBA 2.0 框架支持一次性的传入中断数据请求以及外发中断数据请求。所有中断请求可以利用 USB 中断传输的及时与重试特性。

USB_ATTRS_ISOC_* 标志对所有中断请求而言都是无效属性。USB_ATTRS_SHORT_XFER_OK 和 USB_ATTRS_ONE_XFER 标志仅对主机绑定的请求有效。

只有一次性轮询可以作为同步中断传输执行。如果在请求中指定 USB_ATTRS_ONE_XFER 属性，将进行一次性轮询。

定期轮询是作为异步中断传输启动。原始中断请求将被传递到 usb_pipe_intr_xfer(9F)。当轮询操作找到要返回的新数据时，将从原始请求克隆一个新的 usb_intr_req_t 结构，并使用已初始化的数据块填充该结构。分配请求时，请为 usb_alloc_intr_req(9F) 函数的 *len* 参数指定零。*len* 参数为零是因为 USBA 2.0 框架将为每个回调分配新请求，并填充该请求。分配请求结构后，请填充 *intr_len* 字段，以指定希望框架为每个轮询分配的字节数。将不会返回超出 *intr_len* 字节的数据。

客户机驱动程序必须释放它接收的每个请求。如果逆向发送消息块，请在逆向发送消息块之前从请求中分离该消息块。要从请求中分离消息块，请将该请求的数据指针设置为 NULL。将请求的数据指针设置为 NULL，可在取消分配请求时阻塞释放消息块。

调用 usb_pipe_stop_intr_polling(9F) 函数可取消定期轮询。停止轮询或关闭管道时，将通过异常回调返回原始请求结构。此返回的请求结构的完成原因将被设置为 USB_CR_STOPPED_POLLING。

请不要在轮询进行过程中启动轮询。也不要在调用 usb_pipe_stop_intr_polling(9F) 的过程中启动轮询。

同步请求

同步请求用于速率恒定、与时间相关的流数据。出现错误时不会进行重试。同步请求具有以下特定于请求的字段：

isoc_frame_no 当整个传输必须从特定帧编号开始时，请指定此字段。此字段的值必须大于当前帧编号。使用 usb_get_current_frame_number(9F) 可查找当前帧编号。请注意，当前帧编号为活动目标。对于低速和全速总线，将每 1 毫秒更新一次当前帧。对于高速总线，将每 0.125 毫秒更新一次当前帧。应设置 USB_ATTR_ISOC_START_FRAME 属性以便可以识别 *isoc_frame_no* 字段。

要忽略此帧编号字段并尽快启动，请设置 USB_ATTR_ISOC_XFER_ASAP 标志。

isoc_pkts_count 此字段是请求中的数据包数。此值受由 usb_get_max_pkts_per_isoc_request(9F) 函数返回的值和 *isoc_pkt_descr* 数组（请参见下面的内容）的大小限制。该请求中可传输的字节数等于此 *isoc_pkts_count* 值与端点的 *wMaxPacketSize* 值的乘积。

isoc_pkts_length 此字段是请求的所有包的长度之和。此值由启动器设置。应将此值设置为零，以便自动使用 *isoc_pkt_descr* 列表中 *isoc_pkts_length* 的和，且不对此元素应用任何检查。

isoc_error_count 此字段是完成时出现错误的包数。此值由 USBA 2.0 框架设置。

isoc_pkt_descr 此字段指向定义了每个数据包传输数据量的数据包描述符数组。对于传出请求，此值定义要处理的子请求的专用队列。对于传入请求，此值描述数据块的到达方式。客户机驱动程序将为传出请求分配这些描述符。框架将为传入请求分配和初始化这些描述符。此数组中的描述符包含框架初始化的字段，这些字段存储实际传输的字节数和传输的状态。有关更多详细信息，请参见 usb_isoc_request(9S) 手册页。

所有请求都必须接收已初始化的消息块。此消息块提供数据或存储数据。有关 mblk_t 消息块类型的说明，请参见 mblk(9S) 手册页。

USB_ATTR_ONE_XFER 标志是非法属性，因为系统将决定如何通过可用包数改变数据量。USB_ATTR_SHORT_XFER_OK 标志仅对主机绑定的数据有效。

无论是否设置 USB_FLAGS_SLEEP 标志， usb_pipe_isoc_xfer(9F) 函数都会使所有同步传输成为异步传输。所有同步输入请求都将启动轮询。

调用 usb_pipe_stop_isoc_polling(9F) 函数可取消定期轮询。停止轮询或关闭管道时，将通过异常回调返回原始请求结构。此返回的请求结构的完成原因将被设置为 USB_CR_STOPPED_POLLING。

轮询会一直继续，直到发生以下某个事件：

- 收到 usb_pipe_stop_isoc_polling(9F) 调用。
- 通过异常回调报告设备断开连接。
- 收到 usb_pipe_close(9F) 调用。

刷新管道

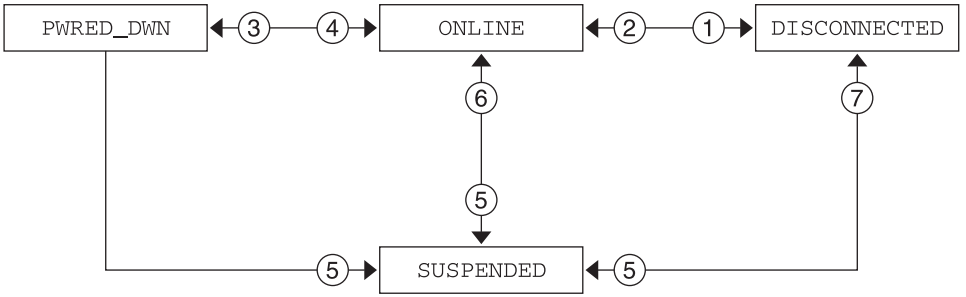
您可能需要在出现错误后清理管道，或者可能想要等待管道清除。可使用下列方法之一刷新或清除管道：

- usb_pipe_reset(9F) 函数重置管道并刷新其所有请求。如果未对处于错误状态的管道启用自动清除，则对这些管道执行上述操作。使用 usb_pipe_get_state(9F) 可确定管道的状态。
- usb_pipe_drain_reqs(9F) 函数将阻塞以等待所有挂起的请求完成，然后继续。此函数可以无限期等待，也可以在指定的时间段后超时。usb_pipe_drain_reqs(9F) 函数既不关闭管道也不刷新管道。

设备状态管理

管理 USB 设备具体涉及到热插拔、系统电源管理（检查点和恢复）以及设备电源管理这几方面。所有客户机驱动程序应实现下图中所示的基本状态机。有关更多信息，请参见 /usr/include/sys/usb/usbai.h。

图 20–4 USB 设备状态机



① 设备已拔除。
② 原始设备已重新连接。
③ 设备空闲一段时间 *T* 后转至低电状态。
④ 由设备或向该设备发送 I/O 的应用程序远程唤醒。
⑤ 通知通过 DDI_SUSPEND 保存状态。
⑥ 对于正确的设备，通知通过 DDI_RESUME 恢复状态。
⑦ 对于断开连接的设备或错误的设备，通知通过 DDI_RESUME 恢复状态。

可以使用特定于驱动程序的状态扩充此状态机及其四种状态。可以定义设备状态 `0x80` 到 `0xff`，且只有客户机驱动程序可以使用这些状态。

热插拔 USB 设备

USB 设备支持热插拔。可以随时插入或移除 USB 设备。客户机驱动程序必须处理打开的设备的移除和重新插入。使用热插拔回调可处理打开的设备。已关闭设备的插入和移除由 `attach(9E)` 和 `detach(9E)` 入口点处理。

热插拔回调

USBA 2.0 框架支持以下事件通知：

- 在热移除设备时，客户机驱动程序将收到回调。
- 在热移除之后放回设备时，客户机驱动程序将收到回调。如果未使设备的驱动程序实例脱机，则当用户将设备放回其原始端口时，可能会发生此事件回调。如果驱动程序实例保持打开状态，则不能使该驱动程序实例脱机。

客户机驱动程序必须在其 `usb_register_hotplug_cbs(9F)` 例程中调用 `attach(9E)`，以便注册事件回调。在中断之前，驱动程序必须在其 `detach(9E)` 例程中调用 `usb_unregister_hotplug_cbs(9F)`。

热插入

USB 设备的热插入的事件顺序如下：

1. 集线器驱动程序 hubd(7D) 等待端口连接状态发生变化。
2. hubd 驱动程序检测到端口连接。
3. hubd 驱动程序枚举设备，创建子设备节点，然后连接客户机驱动程序。有关兼容名称的定义，请参阅第 400 页中的 “绑定客户机驱动程序”。
4. 客户机驱动程序管理设备。驱动程序处于 ONLINE 状态。

热移除

USB 设备的热移除的事件顺序如下：

1. 集线器驱动程序 hubd(7D) 等待端口连接状态发生变化。
2. hubd 驱动程序检测到端口断开连接。
3. hubd 驱动程序将断开连接事件发送到子客户机驱动程序。如果子客户机驱动程序是 hubd 驱动程序或 usb_mid(7D) 多接口驱动程序，则子客户机驱动程序将该事件传播到其子级。
4. 客户机驱动程序在内核线程上下文中接收断开连接事件通知。内核线程上下文使驱动程序的断开连接处理程序进入阻塞状态。
5. 客户机驱动程序将转为 DISCONNECTED 状态。未完成的 I/O 传输将失败，完成原因为 device not responding。所有新 I/O 传输以及打开设备节点的尝试也将失败。要关闭管道，不需要客户机驱动程序。而要保存设备以及重新连接设备时需要恢复的驱动程序上下文，需要客户机驱动程序。
6. hubd 驱动程序试图按照从下到上的顺序使 OS 设备节点及其子节点脱机。

如果在 hubd 驱动程序试图使设备节点脱机时，未打开该设备节点，则会发生以下事件：

1. 将调用客户机驱动程序的 detach(9E) 入口点。
2. 销毁设备节点。
3. 新设备可以使用相应端口。
4. 重新开始热插拔事件序列。hubd 驱动程序等待端口连接状态发生变化。

如果在 hubd 驱动程序试图使设备节点脱机时已打开该设备节点，则会发生以下事件：

1. hubd 驱动程序将脱机请求放入定期脱机重试队列。
2. 新设备仍然不可使用相应端口。

如果在 hubd 驱动程序试图使设备节点脱机时，已打开该设备节点，但用户稍后关闭了该设备节点，则 hubd 驱动程序定期使该设备节点脱机将成功，且会发生以下事件：

1. 将调用客户机驱动程序的 detach(9E) 入口点。
2. 销毁设备节点。

3. 新设备可以使用相应端口。
4. 重新开始热插拔事件序列。hubd 驱动程序等待端口连接状态发生变化。

如果用户关闭使用该设备的所有应用程序，则端口将重新变为可用。如果应用程序未终止或未关闭该设备，则端口仍然不可用。

热重新插入

如果将先前移除的设备重新插入同一端口，同时该设备的设备节点仍处于打开状态，则会发生以下事件：

1. 集线器驱动程序 hubd(7D) 检测到端口连接。
2. hubd 驱动程序恢复总线地址和设备配置。
3. hubd 驱动程序取消脱机重试请求。
4. hubd 驱动程序将连接事件发送到客户机驱动程序。
5. 客户机驱动程序收到连接事件。
6. 客户机驱动程序确定新设备是否与先前连接的设备相同。客户机驱动程序首先通过比较设备描述符来进行此项确定。客户机驱动程序也可以比较序列号和配置描述符群。

如果客户机驱动程序确定当前设备与先前连接的设备不同，则可能会发生以下事件：

1. 客户机驱动程序可能向控制台发出警告消息。
2. 用户可能再次移除该设备。如果用户再次移除该设备，则将重新开始热移除事件序列。hubd 驱动程序检测到端口断开连接。如果用户没有再次移除该设备，则会发生以下事件：
   a. 客户机驱动程序仍然保持 DISCONNECTED 状态，所有请求和打开操作将失败。
   b. 端口仍然不可用。用户必须关闭设备并断开其连接以释放端口。
   c. 释放端口时，将重新开始热插拔事件序列。hubd 驱动程序等待端口连接状态发生变化。

如果客户机驱动程序确定当前设备与先前连接的设备相同，则可能会发生以下事件：

1. 客户机驱动程序可能恢复其状态，并继续正常操作。此策略由客户机驱动程序负责。音频扬声器就是客户机驱动程序可继续操作的典型示例。
2. 如果使用重新连接的设备继续操作是安全的，则将重新开始热插拔事件序列。hubd 驱动程序等待端口连接状态发生变化。设备再次可用。

电源管理

本节讨论设备电源管理和系统电源管理。

设备电源管理根据各个 USB 设备的 I/O 是处于活动状态还是空闲状态来管理这些设备。

系统电源管理使用检查点和恢复机制在文件中设置系统状态的检查点，然后完全关闭系统。（检查点有时称为“系统暂停”。）再次打开系统电源时，系统将恢复为其暂停前的状态。

设备电源管理

下面简要列出了要对 USB 设备进行电源管理时驱动程序需要执行的操作。后面对电源管理进行了较详细的说明。

1. 在执行 attach(9E) 期间创建电源管理组件。请参见 usb_create_pm_components(9F) 手册页。
2. 实现 power(9E) 入口点。
3. 在访问设备之前调用 pm_busy_component(9F) 和 pm_raise_power(9F)。
4. 完成设备访问后调用 pm_idle_component(9F)。

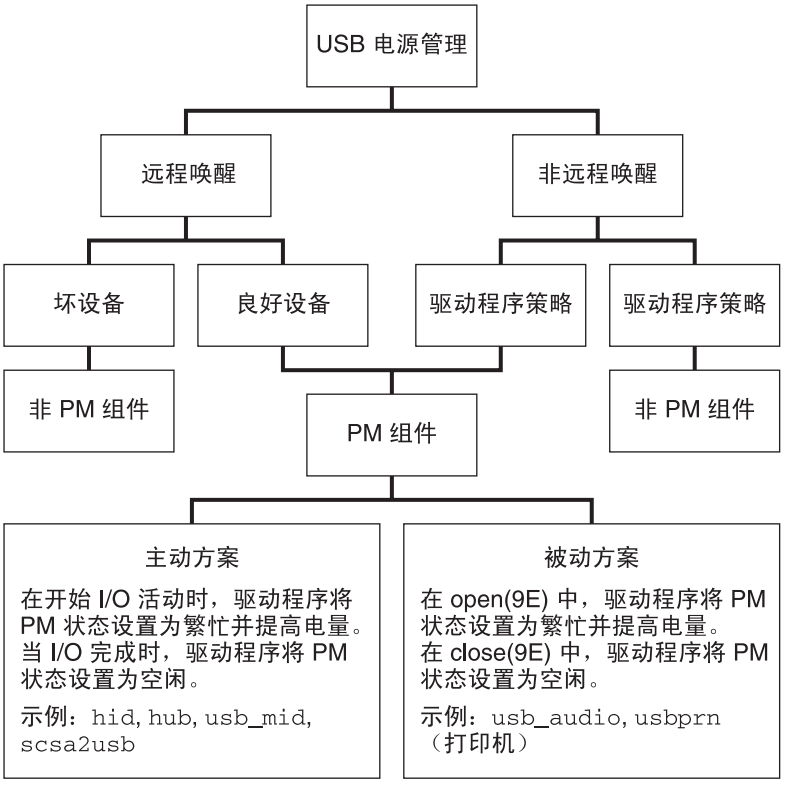
USBA 2.0 框架支持 USB 接口电源管理规范指定的四种电源级别。有关 USB 电源级别与操作系统电源级别对应关系的信息，请参见 /usr/include/sys/usb/usbai.h。

当设备进入 USB_DEV_OS_PWR_OFF 状态时，hubd 驱动程序将暂停端口。当设备进入 USB_DEV_OS_PWR_1 及以上状态时，hubd 驱动程序将恢复端口。请注意，端口暂停不同于系统暂停。端口暂停时，将仅关闭 USB 端口。第 422 页中的 “系统电源管理”中定义了系统暂停。

客户机驱动程序可以选择在设备上启用远程唤醒。请参见 usb_handle_remote_wakeup(9F) 手册页。当 hubd 驱动程序在端口上发现远程唤醒时，hubd 驱动程序将完成唤醒操作，并调用 pm_raise_power(9F) 以通知子级。

下图显示了电源管理的不同部分之间的关系。

图 20–5 USB 电源管理



驱动程序可以实现图 20–5 底部说明的两种电源管理方案之一。被动方案比主动方案简单，这是因为被动方案在设备传输期间不进行电源管理。

主动电源管理

本节介绍了实现主动电源管理方案需使用的函数。

在驱动程序的 attach(9E) 入口点执行以下工作：

1. 调用 usb_create_pm_components(9F)。
2. 可选择调用 usb_handle_remote_wakeup(9F)（使用 USB_REMOTE_WAKEUP_ENABLE 作为第二个参数），以在设备上启用远程唤醒。
3. 调用 pm_busy_component(9F)。
4. 调用 pm_raise_power(9F) 以使功耗达到 USB_DEV_OS_FULL_PWR 级别。
5. 与设备通信以初始化该设备。
6. 调用 pm_idle_component(9F)。

在驱动程序的 detach(9E) 入口点执行以下工作：

1. 调用 pm_busy_component(9F)。
2. 调用 pm_raise_power(9F) 以使功耗达到 USB_DEV_OS_FULL_PWR 级别。
3. 如果在 attach(9E) 入口点中调用了 usb_handle_remote_wakeup (9F) 函数，请在此处调用 usb_handle_remote_wakeup(9F)（使用 USB_REMOTE_WAKEUP_DISABLE 作为第二个参数）。
4. 与设备通信以干净地关闭该设备。
5. 调用 pm_lower_power(9F) 以使功耗达到 USB_DEV_OS_FULL_PWR 级别。

   这是唯一一次客户机驱动程序调用 pm_lower_power(9F)。
6. 调用 pm_idle_component(9F)。

当驱动程序线程要启动在设备上执行 I/O 操作时，该线程将执行以下任务：

1. 调用 pm_busy_component(9F)。
2. 调用 pm_raise_power(9F) 以使功耗达到 USB_DEV_OS_FULL_PWR 级别。
3. 开始 I/O 传输。

当驱动程序收到 I/O 传输已完成的通知时，驱动程序将调用 pm_idle_component(9F)。

在驱动程序的 power(9E) 入口点中，检查您要转换到的电源级别是否有效。此外，还可能需要考虑同时调用 power(9E) 的不同线程。

如果设备已空闲一段时间或者系统正在关闭，则可以调用 power(9E) 例程以使设备进入 USB_DEV_OS_PWR_OFF 状态。此状态对应于图 20–4 中所示的 PWRED_DWN 状态。如果设备将进入 USB_DEV_OS_PWR_OFF 状态，请在 power(9E) 例程中执行以下工作：

1. 使所有打开的管道进入空闲状态。例如，停止对中断管道进行的轮询。
2. 保存任何设备或需要保存的驱动程序上下文。

   在完成 power(9E) 调用后，将暂停设备所连接到的端口。

收到设备启动的远程唤醒或系统启动的唤醒时，可以调用 power(9E) 例程以打开设备电源。由于超出空闲时间或系统暂停而关闭设备电源后，将会发生唤醒通知。如果设备将进入 USB_DEV_OS_PWR_1 或以上状态，请在 power(9E) 例程中执行以下工作：

1. 恢复任何所需的设备和驱动程序上下文。
2. 在管道中重新启动适合指定电源级别的活动。例如，对中断管道启动轮询。

如果先前暂停了设备所连接到的端口，则在调用 power(9E) 之前将恢复该端口。

被动电源管理

被动电源管理方案比上面介绍的主动电源管理方案简单。在此被动方案中，在传输期间不执行任何电源管理。要实现此被动方案，请在打开设备时调用 pm_busy_component(9F) 和 pm_raise_power(9F)。然后在关闭设备时调用 pm_idle_component(9F)。

系统电源管理

系统电源管理包括：在保存整个系统的状态后关闭系统，以及在重新打开系统后恢复状态。此过程称为 *CPR*（checkpoint and resume，检查点和恢复）。在 CPR 相关方面，USB 客户机驱动程序的运行方式与其他客户机驱动程序相同。要暂停设备，请通过 DDI_SUSPEND 的 *cmd* 参数调用驱动程序的 detach(9E) 入口点。要恢复设备，请在 *cmd* 参数为 DDI_RESUME 的情况下调用驱动程序的 attach(9E) 入口点。处理 detach(9E) 例程中的 DDI_SUSPEND 命令时，请尽可能地清理设备状态和驱动程序状态，以满足后面清理恢复操作的需要。（请注意，这对应于图 20–4 中的 SUSPENDED 状态。）处理 attach(9E) 例程中的 DDI_RESUME 命令时，务必使设备达到全功率状态，以使设备与系统同步。

对于 USB 设备，暂停和恢复的处理与热插拔断开连接和重新连接类似（请参见第 416 页中的 “热插拔 USB 设备”）。CPR 与热插拔之间的重要差别是，在 CPR 的情况下，如果设备处于不可暂停的状态，驱动程序的检查点过程可能会失败。例如，如果设备正在进行错误恢复，则无法暂停设备。如果设备正忙，无法安全将其停止，也无法暂停该设备。

序列化

通常，驱动程序在持有互斥锁时不应调用 USBA 函数。因此，客户机驱动程序中的竞态条件可能很难防止。

不允许在处理异步事件（如断开连接或 CPR）的同时运行正常操作代码。这些类型的异步事件通常会清理和中断管道，可能会破坏正常操作代码。

一种管理竞态条件和保护正常操作代码的方法是，编写可以获取和释放独占访问同步对象的序列化工具。您可以按以下方法编写序列化工具：通过调用 USBA 函数安全地持有同步对象。usbskel 驱动程序样例中就采用了这种方法。

实用程序函数

本节介绍几个常规用途的函数。

设备配置工具

本节介绍与设备配置相关的函数。

获取接口编号

如果您使用的是多接口设备，usb_mid(7D) 驱动程序只会使其接口之一可用于调用驱动程序，此时您可能需要知道调用驱动程序所绑定到的接口的编号。使用 usb_get_if_number(9F) 函数执行以下任一任务：

- 返回调用驱动程序所绑定到的接口的编号。在这种情况下，usb_get_if_number(9F) 函数返回大于零的接口编号。
- 发现调用驱动程序管理整个多接口设备。驱动程序在设备级别绑定，因此 usb_mid 没有拆分设备。在这种情况下，usb_get_if_number(9F) 函数返回 USB_DEVICE_NODE。
- 发现调用驱动程序通过管理设备在其当前配置中提供的唯一接口来管理整个设备。在这种情况下，usb_get_if_number(9F) 函数返回 USB_COMBINED_NODE。

管理整个设备

如果驱动程序管理整个复合设备，则可通过使用包含供应商 ID、产品 ID 和修订版 ID 的兼容名称将该驱动程序绑定到整个设备。绑定到整个复合设备的驱动程序必须像结点驱动程序一样管理该设备的所有接口。通常，不应将驱动程序绑定到整个复合设备。应改为使用一般的多接口驱动程序 usb_mid(7D)。

使用 usb_owns_device(9F) 函数可确定驱动程序是否拥有整个设备。设备可以是复合设备。如果驱动程序拥有整个设备，则 usb_owns_device(9F) 函数将返回 TRUE。

多配置设备

在任何特定时间，主机上只能使用 USB 设备的一种配置。大多数设备仅支持一种配置。但是，少数 USB 设备支持多种配置。

对于具有多种配置的任何设备，都是采用可使用某驱动程序的第一种配置。查找匹配项时，设备配置以数字顺序处理。如果未找到任何匹配的驱动程序，则设备将被设置采用第一种配置。在这种情况下，usb_mid 驱动程序将接管该设备，并将设备拆分为多个接口节点。使用 usb_get_cfg(9F) 函数可返回设备的当前配置。

您可以使用以下两种方法中的任何一种来请求采用其他配置。使用其中任何一种方法修改设备配置，均可确保 USBA 模块保持与设备同步。

- 使用 cfgadm_usb(1M) 命令。
- 从驱动程序调用 usb_set_cfg(9F) 函数。

  由于更改设备配置会影响整个设备，因此客户机驱动程序必须满足以下所有条件，才能成功调用 usb_set_cfg(9F) 函数：

  - 客户机驱动程序必须拥有整个设备。
  - 设备不能有子节点，因为其他驱动程序可能会通过这些子节点驱动该设备。
  - 必须关闭除缺省管道之外的所有管道。
  - 设备必须具有多种配置。

注意 – 不要通过手动执行 SET_CONFIGURATION USB 请求来更改设备配置。不支持使用 SET_CONFIGURATION 请求更改配置。

修改或获取替代设置

客户机驱动程序可以调用 usb_set_alt_if(9F) 函数以更改当前选定接口的选定替代设置。请确保关闭已明确打开的所有管道。切换替代设置时，usb_set_alt_if(9F) 函数将验证是否仅打开了缺省管道。确保在调用 usb_set_alt_if(9F) 之前已正确设置了设备。

更改替代设置可能会影响对驱动程序可用的端点以及特定于类和特定于供应商的描述符。有关端点和描述符的更多信息，请参见第 404 页中的 “描述符树”。

调用 usb_get_alt_if(9F) 函数可检索当前替代设置的编号。

注 – 请求新替代设置、新配置或新接口时，必须关闭设备的除缺省管道外的所有管道。这是因为更改替代设置、配置或接口会更改设备的运行模式。此外，更改替代设置、配置或接口还会更改设备在系统中的呈现方式。

其他实用程序函数

本节介绍在 USB 设备驱动程序中有用的其他函数。

检索字符串描述符

调用 usb_get_string_descr(9F) 函数可检索给定了索引的字符串描述符。一些配置、接口或设备描述符具有关联的字符串 ID。这样的描述符包含具有非零值的字符串索引字段。将字符串索引字段值传递给 usb_get_string_descr(9F) 可检索对应的字符串。

管道专用数据工具

每个管道都有一个空间指针，专供客户机驱动程序使用。使用 usb_pipe_set_private(9F) 函数可安装一个值。使用 usb_pipe_get_private(9F) 函数可检索该值。当管道可能需要将其自己的客户机定义状态传递到回调，以进行特定处理时，此工具在回调中很有用。

清除 USB 条件

使用 usb_clr_feature(9F) 函数可执行以下任务：

- 发出 USB CLEAR_FEATURE 请求以清除端点的停止条件。
- 清除设备的远程唤醒条件。
- 在设备级别、接口级别或端点级别清除特定于设备的条件。

获取设备、接口或端点状态

使用 usb_get_status(9F) 函数可发出 USB GET_STATUS 请求，以检索设备、接口或端点的状态。

- **设备状态**。自备电源并启用远程唤醒。
- **接口状态**。根据 USB 2.0 规范返回零。
- **端点状态**。已停止端点。此状态指示运行延迟。必须清除停止状态才能重新运行设备。

  协议延迟指示发出了不支持的控制管道请求。在下一个控制传输开始时将会自动清除协议延迟。

获取设备的总线地址

使用 usb_get_addr(9F) 函数可获取设备的 USB 总线地址以用于调试目的。此地址映射到特定的 USB 端口。

USB 设备驱动程序样例

本节介绍使用 Oracle Solaris 环境的 USBA 2.0 框架的 USB 设备驱动程序模板。此驱动程序演示了本章中讨论的许多功能。此模板或框架驱动程序的名称为 usbskel。

usbskel 驱动程序是可用于启动您自己的 USB 设备驱动程序的模板。usbskel 驱动程序演示了以下功能：

- 读取设备的原始配置数据。每个 USB 设备需要能够报告设备的原始配置数据。
- 管理管道。usbskel 驱动程序打开中断管道以显示如何管理管道。
- 轮询。usbskel 驱动程序中的注释讨论如何进行轮询。
- USB 版本管理和注册。

- USB 日志。
- 支持 USB 热插拔。
- 支持 Oracle Solaris 暂停和恢复。
- 支持电源管理。
- USB 序列化。
- 使用 USB 回调。

◆ ◆ ◆ 第 21 章

21 SR-IOV 驱动程序

本章介绍了单根 IO 虚拟化 (Single Root I/O Virtualization, SR-IOV) 设备驱动程序并提供了有关以下主题的信息：

SR-IOV 简介

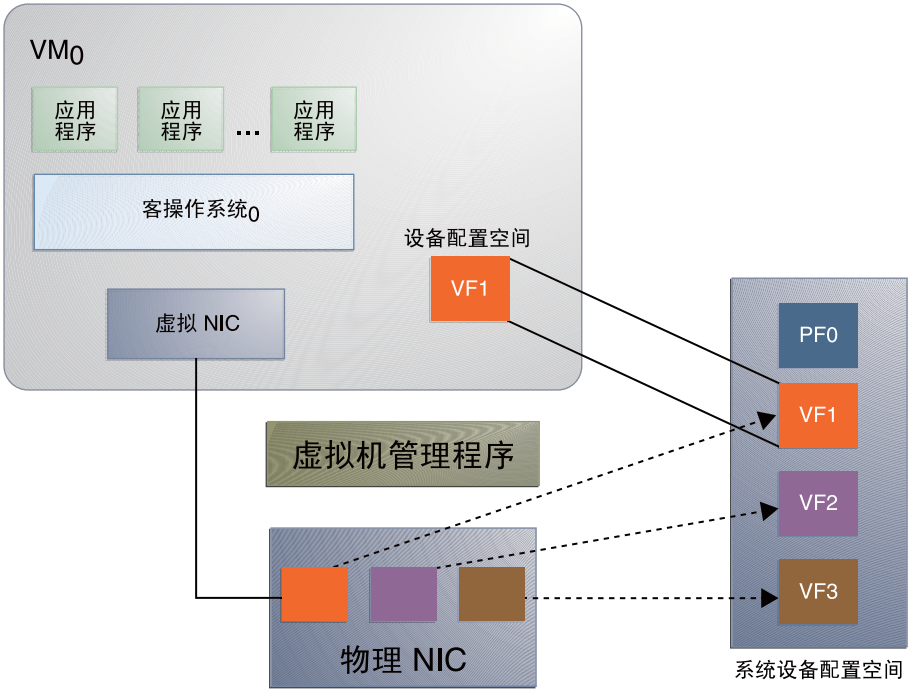
SR-IOV 技术是一种基于硬件的虚拟化解决方案，可提高性能和可伸缩性。SR-IOV 标准允许在虚拟机之间高效共享 PCIe（Peripheral Component Interconnect Express，快速外设组件互连）设备，并且它是在硬件中实现的，可以获得能够与本机性能媲美的 I/O 性能。SR-IOV 规范定义了新的标准，根据该标准，创建的新设备可允许将虚拟机直接连接到 I/O 设备。

SR-IOV 规范由 PCI-SIG 在 `http://www.pcisig.com` 上进行定义和维护。

单个 I/O 资源可由许多虚拟机共享。共享的设备将提供专用的资源，并且还使用共享的通用资源。这样，每个虚拟机都可访问唯一的资源。因此，启用了 SR-IOV 并且具有适当的硬件和 OS 支持的 PCIe 设备（例如以太网端口）可以显示为多个单独的物理设备，每个都具有自己的 PCIe 配置空间。

下图介绍了针对 PCIe 硬件的 SR-IOV 技术。

图 21–1 SR-IOV 技术



SR-IOV 中的两种新功能类型是：

物理功能 (Physical Function, PF)	用于支持 SR-IOV 功能的 PCI 功能，如 SR-IOV 规范中定义。PF 包含 SR-IOV 功能结构，用于管理 SR-IOV 功能。PF 是全功能的 PCIe 功能，可以像其他任何 PCIe 设备一样进行发现、管理和处理。PF 拥有完全配置资源，可以用于配置或控制 PCIe 设备。
虚拟功能 (Virtual Function, VF)	与物理功能关联的一种功能。VF 是一种轻量级 PCIe 功能，可以与物理功能以及与同一物理功能关联的其他 VF 共享一个或多个物理资源。VF 仅允许拥有用于其自身行为的配置资源。

每个 SR-IOV 设备都可有一个物理功能 (Physical Function, PF)，并且每个 PF 最多可有 64,000 个与其关联的虚拟功能 (Virtual Function, VF)。PF 可以通过寄存器创建 VF，这些寄存器设计有专用于此目的的属性。

一旦在 PF 中启用了 SR-IOV，就可以通过 PF 的总线、设备和功能编号（路由 ID）访问各个 VF 的 PCI 配置空间。每个 VF 都具有一个 PCI 内存空间，用于映射其寄存器集。VF 设备驱动程序对寄存器集进行操作以启用其功能，并且显示为实际存在的 PCI

设备。创建 VF 后，可以直接将其指定给 IO 来宾域或各个应用程序（如裸机平台上的 Oracle Solaris Zones）。此功能使得虚拟功能可以共享物理设备，并在没有 CPU 和虚拟机管理程序软件开销的情况下执行 I/O。

SR-IOV 的优点

SR-IOV 标准允许在 IO 来宾域之间高效共享 PCIe 设备。SR-IOV 设备可以具有数百个与某个物理功能 (Physical Function, PF) 关联的虚拟功能 (Virtual Function, VF)。VF 的创建可由 PF 通过设计用来开启 SR-IOV 功能的寄存器以动态方式进行控制。缺省情况下，SR-IOV 功能处于禁用状态，PF 充当传统 PCIe 设备。

具有 SR-IOV 功能的设备可以利用以下优点：

- 性能－从虚拟机环境直接访问硬件。
- 成本降低－节省的资本和运营开销包括：
  - 节能
  - 减少了适配器数量
  - 简化了布线
  - 减少了交换机端口

支持的平台

- Sparc：所有 T3 和 T4 系列系统都支持具有 Oracle Solaris SR-IOV 功能的设备。
- x86：基于 Intel Nehalem-EX 的系统（如 x4470 和 x4800）支持具有 Oracle Solaris SR-IOV 功能的设备。

下列接口卡支持具有 SR-IOV 功能的设备：

- 基于 Intel 的 NIC，例如 Kawela（82576，1G）
- Niantic（82599，10G）

注 – 要运行具有 SR-IOV 功能的设备，系统固件应支持 SR-IOV 功能。

词汇表

Control Domain（控制域） 用于管理虚拟化策略的域。

Root Domain（根域） 用于管理 PCIe 结构的域。

IO Domain（IO 域）	对指定给它的 IO 设备具有独占访问权限的域。
Fabric（结构）	PCIe 结构组件，例如根联合体、根端口、交换机、网桥以及端点的配置空间寄存器。
VM	虚拟机
PCI Device / PCI Component（PCI 设备/PCI 组件）	■ 单个或多个 PCI 功能 ■ 通常是一个硅晶片
PCI Function（PCI 功能）	■ PCI 结构中最小的独立可寻址单元 ■ 一组 PCI 配置空间寄存器
PCI Virtual Function（PCI 虚拟功能）	■ 硬件中的轻量级 PCI 功能 ■ 看起来基本上与软件的 PCI 功能相同

SR-IOV 设备驱动程序概述

SR-IOV 功能包括物理功能 (Physical Function, PF) 驱动程序和虚拟功能 (Virtual Function, VF) 驱动程序。以下各节介绍了 PF 和 VF 驱动程序以及必需的设备配置的详细信息。

物理功能 (Physical Function, PF) 驱动程序

SR-IOV 设备的 PF 驱动程序用于管理具有 SR-IOV 功能的设备的物理功能 (Physical Function, PF)。SR-IOV 规范中定义了支持 SR-IOV 功能的 PCI 功能。PF 包含 SR-IOV 功能结构，用于管理 SR-IOV 功能。PF 是全功能的 PCIe 功能，可以像其他任何 PCIe 设备一样进行发现、管理和处理。PF 拥有完全配置资源，可以用于配置或控制 PCIe 设备。PF 驱动程序具有以下特征：

- 仅在根域中可见
- 可能有也可能没有数据移动功能。PF 驱动程序即使在 SR-IOV 模式下也应该正常工作。
- 通过 Oracle Solaris IOV 框架提供的 API 控制 SR-IOV 功能的启用和禁用。
- 要为给定的 PF 配置的 VF 数量是由系统管理员决定的。此数值在 Sparc OVM 平台上的计算机描述符 (Machine Descriptor, MD) 中定义，或者在裸机环境中的配置文件中定义。
- PF 驱动程序在连接阶段通过借助 DDI 接口调用 Oracle Solaris IOV 框架来启用 VF。如果 PF 驱动程序在连接过程中未启用 VF，只要驱动程序回调标志指示支持具有 IOV 功能的驱动程序，Oracle Solaris IOV 框架在连接后将立即尝试配置 VF。
- PF 可以通过特定于设备的机制分别启用和禁用每个关联的 VF。

虚拟功能 (Virtual Function, VF) 驱动程序

与物理功能关联的一种功能。VF 是一种轻量级 PCIe 功能，可以与物理功能以及与同一物理功能关联的其他 VF 共享一个或多个物理资源。VF 驱动程序具有以下特征：

- 在根域和 I/O 域中都可见
- 可以通过 HW 邮箱或 OS 提供的接口启动与其 PF 的通信
- 在根域中不可见，除非满足以下条件：
  - 根域已经引导
  - PF 驱动程序连接并调用了 VF 的配置
  - 根域的 Oracle Solaris IOV 框架启用了 VF
  - 系统固件向 VF 分配了资源
- 在 I/O 域中不可见，除非满足以下条件：
  - VF 已启用且在根域中可见
  - VF 已指定给 I/O 域
  - Oracle Solaris 固件 (OBP) 在 I/O 域中检测到 VF

注 – 具有 SR-IOV 功能的 PF 和 VF 驱动程序必须注册中断资源管理 (Interrupt Resource Management, IRM) 回调并提供对此功能的支持。有关 IRM 接口的详细信息和用法，请参见第 8 章，中断处理程序。

注 – 如果 VF 是一个网络 VF，则可以在启用 `numVFs` 后配置以下参数。配置应该在启用 VF 之前完成。

- `mac-addr`
- `vlan (Virtual LAN) ID`
- `port-vlan-id`
- `alt-mac-addrs`
- `mtu`

设备配置参数

PF 驱动程序必须支持下表中列出的配置参数。这些参数可以导出到 Sparc OVM Manager 中。仅当所有参数均已配置时，配置才算完成。

表 21–1 配置参数定义

配置参数	定义	示例
与标准相关的配置参数	可以支持的 VF 数量 注 – 如果要更改 VF 的数量，需要先分离然后重新连接 PF 设备。	max-config-vfs－实际上可以配置的最大 VF 数量。当 PF 驱动程序支持的最大 VF 数量不同于 SR-IOV 功能指示的容量时，PF 驱动程序可以导出此参数。
特定于资源和设备的参数	带宽、池 和 Q 对。对这些参数所做的更改会同时影响 PF 和 VF 驱动程序。 框架可能不能识别特定于设备的参数，可能只有 PF 驱动程序能够识别这些参数。在启用 VF 之前应该识别这些参数，以便 PF 驱动程序能够正确地初始化其硬件。 要了解如何获取可导出到 IOV 框架且特定于设备的参数，请参见 igb(7D) 和 ixgbe(7D)。	■ pvid-exclusive－表示不能同时支持 port-vlan-id 和 vlan-ids。 ■ max-vf-mtu－允许 VF 使用的最大 MTU。 ■ max-vlans－网络类 PF 驱动程序支持的 vlan 插槽的最大数量。
特定于类的参数	基于设备类的通用属性。例如，以太网设备可能具有 MAC 地址、VLAN ID、端口 VLAN ID、带宽等属性。 期望使用特定于类的配置，并且这些配置可以定义每个参数的行为。	无

注 – 当设备配置参数发生更改时，应重新连接设备。

注 – 在启用 VF 之前请按以下顺序配置参数。特定于类的参数将基于特定于类的配置。

1. 与标准相关的参数
2. 特定于资源和设备的参数
3. 特定于类的参数

pci.conf 文件

通过 PCI 配置信息文件 /etc/pci.conf，系统可以保存 PCI 配置（如特定 PF 的 VF 数量）。pci.conf 文件提供了以下内容：

- －持久保留 PCI 配置以便在引导系统时可以自动创建 VF。

- —由于配置文件是 boot_archive 的一部分，所以在系统引导期间可以使用 VF。

注 – 将 `/etc/pci.conf` 添加到 `/boot/solaris/filelist.ramdisk` 文件中以便将 `/etc/pci.conf` 文件包括在 Oracle Solaris 引导进程中。

有关更多信息，请参见附录 E，pci.conf 文件。

设置设备配置参数

- Sparc：可以通过 `ldm` 命令设置这类参数。有关详细信息，请参见 `ldm(1M)` 手册页。
- x86：可通过 `pci.conf` 文件指定特定于类的参数。以下示例显示了在 `pci.conf` 文件中设置的参数。

示例 21–1 设置设备配置参数

```
[[path=/pci@0,0/pci8086,3a40@1c/pci108e,4848@0,1]]
num-vf=2

[Device_Configuration]
[[path=/pci@0,0/pci8086,3a40@1c/pci108e,4848@0,1]]
VF[0] = {
        primary-mac-addr = 0xaabbccddeeff
        alt-mac-addrs = 0x102233445556, 0x102233445557
        vlan-id = 20, 30
}

VF[1] = {
        primary-mac-addr = 0xaabbccddeef1
        alt-mac-addrs = 0x102233445568
        vlan-id = 20, 30, 40, 50
}
```

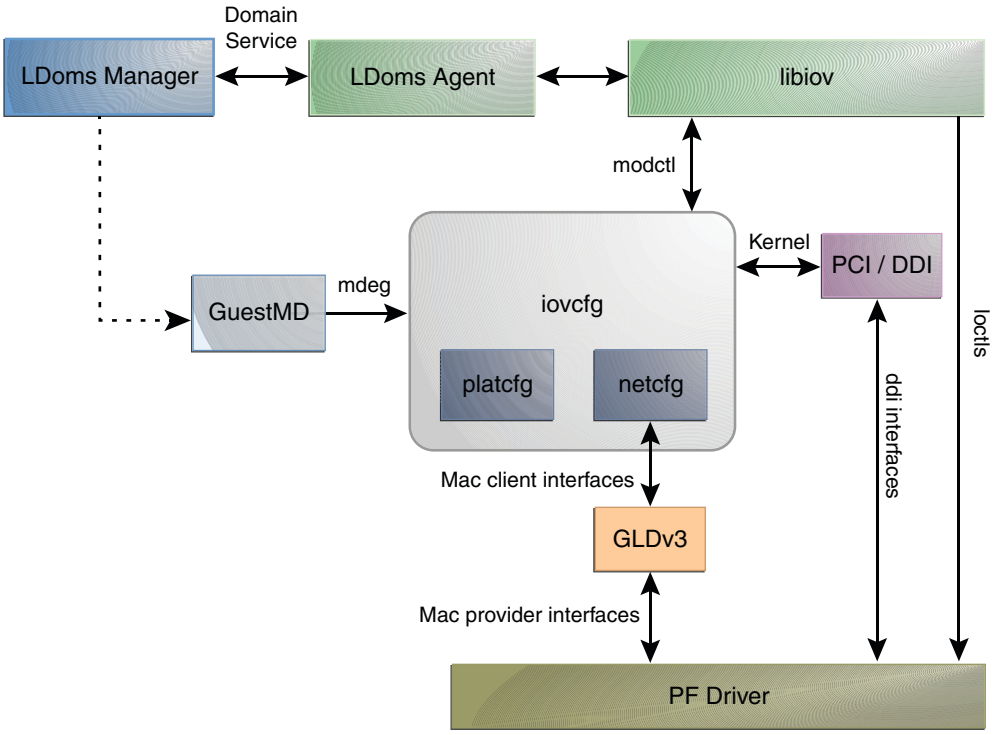
Sparc OVM 平台上的 SR-IOV 配置

Sparc OVM Manager 负责所有 Sparc OVM 平台上的 SR-IOV 配置。Sparc OVM Manager 负责以下操作：

- 获取具有 SR-IOV 功能的驱动程序的 PF 列表
- 获取驱动程序支持的特定于设备的参数
- 验证特定的设备配置
- 更新计算机描述符 (Machine Descriptor, MD) 文件的所有有效配置详细信息以及 VF 的分配和删除情况。

下图显示了 Sparc OVM 配置的概要视图。

图 21–2 Sparc OVM 配置的概要视图



裸机平台上的 SR-IOV 配置

在发行 Oracle Solaris 11 时，未提供任何可用来在裸机平台（包括 x86）上配置 SR-IOV 的工具。

引导配置序列

具有 SR-IOV 功能的 PF 驱动程序在连接期间执行以下操作：

1. 调用 `pciv_vf_config()` 函数以获取 VF 数量。
2. 为 PF 和 VF 获取特定于设备的参数并验证这些参数。
3. 相应地初始化硬件。
4. 调用 `pciv_vf_config()` 接口以启用 VF。
5. 如果 PF 驱动程序是网络驱动程序，则该驱动程序在连接期间将使用 `mac_register()` 接口向 GLDv3 框架进行注册。PF 驱动程序还执行特定于类的初始化。这将引发以下一系列操作：
   - GLDv3 接口识别 PF 设备的存在。

- PF 驱动程序导出一组新的 MAC 提供者接口。该过程使 MAC 层能够识别驱动程序是否为 PF 驱动程序。MAC 层还将获取有关 VF 驱动程序的更多信息。

有关网络驱动程序和接口的更多信息，请参见《Oracle Solaris 管理：网络接口和网络虚拟化》中的第 19 章，网络设备驱动程序。

VF 实例现在已初始化。仅当将 VF 指定给根域时才连接 VF 驱动程序。

SR-IOV 接口汇总

下表列出了可供 SR-IOV 驱动程序使用的 Oracle Solaris 接口。将调用这些接口来获取 PF 和 VF 参数、配置 VF 以及将配置参数导出到任何调用应用程序。

注 – 下表中列出的所有接口（`pciv_send()` 除外）都只适用于 PF 驱动程序。

表 21–2 SR-IOV 驱动程序的接口

接口名称	说明
`pci_param_get(9F)`	由 PCI 设备驱动程序使用，用于获取当前配置的参数的名称-值对列表。
`pci_param_free(9F)`	设备驱动程序在使用 *param* 句柄获取设备参数后必须调用此接口。此调用将释放由 `pci_param_get()` 和 `pci_param_get_ioctl()` 接口分配的资源。
`pci_plist_get(9F)`	由 SR-IOV 设备驱动程序使用，用以获取 PF 设备的名称-值对列表。
`pci_plist_getvf(9F)`	由 SR-IOV 设备驱动程序使用，用以获取 VF 设备的名称-值对列表。
`pci_param_get_ioctl(9F)`	SR-IOV 设备驱动程序的一个帮助器函数，如果驱动程序实现了 ioctl `IOV_VALIDATE_PARAM`，则该接口用于从 `arg` 参数中提取 PF 和 VF 设备的参数。
`pciv_vf_config(9F)`	由 PF 驱动程序使用，用于获取 VF 配置参数以及配置其 VF。
`pci_plist_lookup(9F)`	用于在通过 `pci_plist_get()` 和 `pci_plist_getvf()` 接口获取的列表中获取 PF 和 VF 的参数。
`pciv_send(9F)`	供 PF 和 VF 驱动程序用来传送信息。
`ddi_cb_register(9F)`	回调注册机制接口。

驱动程序 Ioctl

SR-IOV 驱动程序 ioctl 用于标识可由管理员配置的特定于设备的参数，并在应用特定配置之前验证此特定配置。定义了以下 ioctl 及其数据结构。

- 第 444 页中的 “`iov_param_ver_info` 结构”
- 第 444 页中的 “`iov_param_validate` 结构”
- 第 444 页中的 “`iov_param_desc` 结构”

SR-IOV 驱动程序的接口

以下各节介绍了 SR-IOV 驱动程序的接口。

pci_param_get() 接口

SR-IOV 驱动程序必须使用 pci_param_get(9F) 接口来获取当前配置的参数的列表。此接口在驱动程序连接期间或任何其他合适的时间被调用。返回的数据是指向参数列表的指针，包含 PF 及其相应 VF 设备的名称-值信息。

```
int pci_param_get(dev_info_t *dip, pci_param_t *php)
```

其中：

dip　指向 dev_info 结构的指针。

php　指向 param 句柄 pci_param_t 的指针。

在调用 pci_param_get 接口后，设备驱动程序应执行以下步骤来获取 PF 和 VF 的参数列表：

1. 调用 pci_plist_get(9F) 接口来获取 PF 设备的参数列表，调用 pci_plist_getvf(9F) 接口来获取所配置的 VF 的参数列表。
2. 调用 pci_plist_lookup(9F) 接口来获取设备参数。
3. 验证所有的 PF 和 VF 参数。
4. 如果参数与当前配置不匹配，则驱动程序应当使设备连接失败。
5. 调用 pci_param_free(9F) 接口来释放用于获取 PF 和所配置的 VF 设备的参数的参数句柄指针。请参见示例 21–2。

注 – 应在配置 VF 之前完成参数验证。

名称-值对是针对每台设备分别定义的。PF 有一组名称-值对，每个已配置的 VF 也有一组名称-值对。名称-值对是可选的，可以为任何或所有设备省略名称-值对。

示例 21–2 SR-IOV pci_param_get(9F) 例程

```
    pci_param_t my_params;
pci_plist_t pf_plist;
pci_plist_t  vf_plist[8];
```

示例 21–2 SR-IOV pci_param_get(9F) 例程 （续）

```
labelp = NULL;
rval = pci_param_get(dip,&my_params);
if (rval || (my_params == NULL)) {
   cmn_err(CE_NOTE, "No params available\n");
goto continue_with_attach;
}
rval = pci_plist_get(my_params, &pf_list);
if (rval || (pf_plist == NULL)) {
        cmn_err(CE_NOTE, "No params for PF \n");
goto continue_with_attach;
}
for (i = 0; i < 8; i++) {
    rval = pci_plist_getvf(my_params, i, &vf_plist[i]);
    if (rval || (vf_plist[i] == NULL)) {
     cmn_err(CE_WARN, "No params for VF %d\n", i);
     continue;
 }
}
pci_param_free(my_params);
/*
* Validate the PF and VF params lists.
* Fail the attach if the params are incompatible or exceed the
* resources available.
*/
continue_with_attach:
```

pci_param_get_ioctl() 接口

SR-IOV 设备驱动程序可以使用 pci_param_get_ioctl(9F) 接口从 *arg* 参数中提取 PF 和 VF 设备的参数（如果这些驱动程序实现了 IOV_VALIDATE_PARAM ioctl）。

int pci_param_get_ioctl(dev_info_t *dip, intptr_t *arg*, int *mode*,pci_param_t **php*)

其中：

dip　指向 dev_info 结构的指针。

arg　通过驱动程序的 ioctl 调用获取的参数

mode　通过驱动程序的 ioctl 调用获取的参数

php　指向通过调用 pci_param_get() 或 pci_param_get_ioctl() 接口获取的 *param* 句柄 pci_param_t 的指针。

在检索参数后，驱动程序应调用 pci_param_free() 接口来释放在该调用中返回的 param 句柄。

pci_plist_get() 接口

pci_plist_get(9F) 接口用于从通过 pci_param_get(9F) 调用或 pci_param_get_ioctl(9F) 调用获取的 param 句柄中获取参数列表。

```
int pci_plist_get(pci_param_t param, pci_plist_t *plist_p)
```

其中：

param　　从 pci_param_get() 或 pci_param_get_ioctl() 接口获取的句柄。

plist_p　　指向在成功返回时返回非空 plist 的 pci_plist_t 的指针。

从 pci_plist_get() 调用返回的 plist 仅适用于 PF 功能。结构 pci_plist_t 支持以下数据类型的数组：

- int8_t
- uint8
- int16_t
- uint16_t
- int32_t
- uint32_t
- int64_t
- uint64_t
- char *

pci_plist_getvf() 接口

pci_plist_getvf(9F) 接口用于获取 VF 设备的名称-值对列表。

```
int pciv_plist_getvf (pci_param_t param, uint16_t vf_index, pci_plist_t *vfplist_p
)
```

其中：

param　　从 pci_param_get() 或 pci_param_get_ioctl() 接口获取的一个句柄。

vf_index　　介于 0 到 #VFS - 1 之间的一个值。

*vfplist_p　　指向 pci_plist_t 结构的指针。

pciv_vf_config() 接口

pciv_vf_config(9F) 接口由 SR-IOV 驱动程序使用，用于获取关于 VF 的配置信息，还用于在驱动程序连接期间配置 VF。

```
#include <sys/sunddi.h>
int pciv_vf_config(dev_info_t *dip, pciv_config_vf_t *vfcfg_p)
```

其中：

`dip` 指向 `dev_info` 结构的指针。

`vfcfg_p` 指向 `pciv_config_vf` 结构的指针。

```
typedef enum {
                PCIV_VFCFG_PARAM,
                PCIV_VF_ENABLE,
                        PCIV_VF_DISABLE
                PCIV_EVT_VFENABLE_PRE,
                PCIV_EVT_VFENABLE_POST,
                PCIV_EVT_VFDISABLE_PRE,
                PCIV_EVT_VFDISABLE_POST
            } pciv_vf_config_cmd_t;
```

`pciv_config_vf` 结构包含以下字段：

```
typedef struct pciv_config_vf {
               int version;
                pciv_vf_config_cmd_t cmd;
                uint16_t num_vf;
                uint16_t first_vf_offset;
                uint16_t vf_stride;
                boolean_t ari_cap;
                uint32_t page_size;
    } pciv_config_vf_t;
```

其中：

`version` 版本号。

`cmd` 用于指示是否调用此接口来获取配置信息或连接 VF。

- `PCIV_VFCFG_PARAM`－获取配置信息
- `PCIV_VF_ENABLE`－启用 VF
  - `PCIV_EVT_VFENABLE_PRE`
  - `PCIV_EVT_VFDISABLE_PRE`
  - `PCIV_EVT_VFENABLE_POST`
  - `PCIV_EVT_VFDISABLE_POST`

`num_vf` 在后端定义的 VF 的数量。

`vf_stride` 两个 VF 之间的距离。

`first_vf_offset` 第一个 VF 与 PF 之间的偏移量。

`ari_cap` 具有 ARI 功能的分层结构。

`page_size` 指定系统页大小。

驱动程序首先应在将 cmd 字段设置为 PCIV_VFCFG_PARAM 的情况下调用 pciv_vfconfig() 接口来获取配置信息。然后，驱动程序应在将 cmd 字段设置为 PCIV_VF_ENABLE 的情况下再次调用此接口来配置 VF。

驱动程序可能会返回以下错误代码之一：

DDI_SUCCESS

DDI_FAILURE

PCIV_REQRESET

PCIV_REQREATTACH

在调用 pciv_vf_config() 接口来启用 VF 之前，驱动程序不是必须通过设置 DDI_CB_FLAG_SRIOV 位来注册回调不可。不过，要在 SR-IOV 框架未配置 VF 时接收通知，驱动程序必须在启用 VF 后通过设置 DDI_CB_FLAG_SRIOV 位来注册回调。有关更多信息，请参见第 446 页中的 “驱动程序回调”。

可支持 VF 的所有 PF 驱动程序应该通过在标志参数中设置 DDI_CB_FLAG_SRIOV 标志的情况下调用 ddi_cb_register(9F) 来告诉 PCIe 框架它们的容量。在驱动程序的连接例程中必须调用 ddi_cb_register() 函数。如果 PF 设备驱动程序在其连接例程中调用 pciv_vf_config() 函数来启用 VF，则 PF 驱动程序应在启用 VF 后调用 ddi_cb_register() 函数。

框架需要使用 DDI_CB_FLAG_SRIOV 标志才能执行以下操作：

- 向 Sparc OVM 代理指示存在可支持 VF 的设备驱动程序。Sparc OVM 代理随后将允许创建 VF 设备。如果没有此功能，用户将无法在 Sparc 平台上创建 VF。
- 框架在禁用 VF 前后将回调 PF 驱动程序。这将有助于 PF 驱动程序执行内部记帐以持续支持 VF。

pci_plist_lookup() 接口

pci_plist_lookup(9F) 接口可由驱动程序用来查找受支持的各种数据类型的名称-值对。函数将查找与接口名称指示的名称和类型匹配的 nvpair（名称-值对）。如果找到，将修改 *nelem* 和 *val* 以分别包含值中的元素数和数据的起始地址。

pci_plist_lookup() 接口支持以下数据类型：

- int pci_plist_lookup_int8(pci_plist_t *plist* , const char **name*,int8_t * *val*)
- int pci_plist_lookup_uint8(pci_plist_t *plist* , const char **name*,uint8_t * *val*)
- int pci_plist_lookup_int16(pci_plist_t *plist* , const char **name*,int16_t * *val*)

- int pci_plist_lookup_uint16(pci_plist_t *plist* , const char **name*, uint16_t * *val*)
- int pci_plist_lookup_int32(pci_plist_t *plist* , const char **name*,int32_t * *val*)
- int pci_plist_lookup_uint32(pci_plist_t *plist* , const char **name*, uint32_t * *val*)
- int pci_plist_lookup_int64(pci_plist_t *plist* , const char **name*,int64_t * *val*)
- int pci_plist_lookup_uint64(pci_plist_t *plist* , const char **name*, uint64_t * *val*)
- int pci_plist_lookup_string(pci_plist_t *plist* , const char **name*, char ** *val*)
- int pci_plist_lookup_plist(pci_plist_t *plist* , const char **name*, pci_plist_t * * *val*)
- int pci_plist_lookup_int8_array(pci_plist_t *plist*, const char **name*,int8_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_uint8_array(pci_plist_t *plist*, const char **name*, int8_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_int16_array(pci_plist_t *plist*, const char **name*,int16_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_uint16_array(pci_plist_t *plist*, const char **name*, uint16_t **val*, uint_t **nelem*)
- int pci_plist_lookup_int32_array(pci_plist_t *plist*, const char **name*,int32_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_uint32_array(pci_plist_t *plist*, const char **name*,uint32_t **val*, uint_t **nelem*)
- int pci_plist_lookup_int64_array(pci_plist_t *plist*, const char **name*, int64_t **val*, uint_t **nelem*)
- int pci_plist_lookup_uint64_array(pci_plist_t *plist*,const char **name*,uint64_t * *val*, uint_t **nelem*)
- int pci_plist_lookup_string_array(pci_plist_t *plist*, const char **name*,char ** *val*, uint_t **nelem*)

其中：

`plist` 指向要处理的 `pci_plist_t` 结构的指针。

`name` 要搜索的名称-值对的名称。

`nelem` 存储值中的元素数的地址。

`val` 数据的起始地址。

`pci_plist_lookup()` 函数在成功时返回 0，失败时返回某个错误值。支持以下错误值：

DDI_EINVAL	参数无效
ENOENT	未找到匹配的名称-值对
ENOTSUP	编码或解码方法不受支持

pci_param_free() 接口

驱动程序在使用 param 句柄获取设备参数后必须调用 pci_param_free(9F) 接口。此调用将释放由 pci_param_get() 和 pci_param_get_ioctl() 接口分配的资源。

```
int pci_param_free (pci_param_t param)
```

其中，*param* 是从 pci_param_get() 或 pci_param_get_ioctl() 接口获取的句柄。

pciv_send() 接口

pciv_send(9F) 接口由具有 SR-IOV 功能的 PF 和 VF 驱动程序用来彼此进行通信。尽管 VF 驱动程序只能与其 PF 驱动程序进行通信，但 PF 驱动程序可与其任何 VF 驱动程序进行通信。

```
int pciv_send(dev_info_t *dip, pciv_pvp_req_t *req
```

其中：

dip	指向 dev_info 结构的指针。
req	指向 pciv_pvp_req_t 结构的指针。

pciv_pvp_req_t 的结构为：

```
typedef struct pciv_pvp_req {
int pvp_dstfunc;
caddr_t pvp_buf;
size_t pvp_nbyte;
buf_cb_t pvp_cb;
caddr_t pvp_cb_arg;
uint_t pvp_flag;
} pciv_pvp_req_t;
```

其中：

pvp_dstfunc	如果由 PF 驱动程序调用，则 VF 索引的范围是 1 到 *num_vf*。如果调用方是 VF 驱动程序，则它应当始终为 PCIV_PF。
pvp_buf	要发送的调用方缓冲区的缓冲区地址。
pvp_nbyte	要传送的字节数，必须小于 8k。

`pvp_cb` 回调函数指针，如果 *pvp_flag* 设置为 `PCIV_NOWAIT`。

如果 *pvp_flag* 设置为 `PCIV_NOWAIT`，调用将立即返回，并在将 *pvp_buf* 中的数据传送到目标之前调用 *pvp_cb* 中的回调例程。随后，将允许调用方在其回调例程中释放缓冲区。

```
typedef void (*buf_cb_t)(int rc, caddr_t buf, size_t size, caddr_t cb_arg);
```

其中：

`rc` 用于传输的 DDI 返回代码。

`buf` 要发送的调用方缓冲区的缓冲区地址。

`size` 要传输的字节数。

`cb_arg` 调用方在调用例程时设置的输入参数。

`pvp_cb_arg` *pvp_cb* 的回调输入参数，如果 *pvp_flag* 设置为 `PCIV_NOWAIT`。

`pvp_flag`

- `PCIV_NOWAIT`－不等待接收方的响应。
- `PCIV_WAIT`－这是缺省状态。等待，直到接收方确认接收传输。

`pciv_send()` 接口返回以下返回值之一：

`DDI_SUCCESS` 缓冲区已成功发送。

`DDI_ENOTSUP` 设备驱动程序不支持此操作。调用方可以使用其他机制，如硬件邮箱。

`DDI_EINVAL` *pvp_nbyte* 或 *pvp_dstfunc* 无效。

`DDI_ENOMEM` 操作因缺少资源而失败。

`DDI_ETRANSPORT` 远程端未注册用于处理传入的传输的回调。

`DDI_FAILURE` 因意外原因失败。

SR-IOV 驱动程序 Ioctl

SR-IOV 驱动程序 ioctl 用于标识可由管理员配置的特定于设备的参数，并在应用特定配置之前验证此特定配置。以下各节介绍了 ioctl 数据结构和接口。

数据结构

以下各节列出了 PF 驱动程序在实现 ioctl 之前应该定义并初始化的数据结构：

iov_param_ver_info 结构

iov_param_ver_info 结构定义如下：

```
#define IOV_IOCTL (('I' << 24) | ('O' << 16) | ('V' << 8))
#define IOV_GET_VER_INFO (IOV_IOCTL | 0)
#define IOV_GET_PARAM_INFO (IOV_IOCTL | 1)
#define IOV_VALIDATE_PARAM (IOV_IOCTL | 2)
#define IOV_PARAM_DESC_VERSION  1
```

iov_param_ver_info 结构包含以下字段：

```
typedef struct iov_param_ver_info {
uint32_t version;
uint32_t num_params;
} iov_param_ver_info_t;
```

其中：

version　　版本信息

num_params　　参数个数

iov_param_validate 结构

iov_param_validate 结构定义如下：

```
  #define IOV_IOCTL (('I' << 24) | ('O' << 16) | ('V' << 8))
  #define IOV_GET_VER_INFO (IOV_IOCTL | 0)
  #define IOV_GET_PARAM_INFO (IOV_IOCTL | 1)
  #define IOV_VALIDATE_PARAM  (IOV_IOCTL | 2)
  #define IOV_PARAM_DESC_VERSION  1
```

iov_param_validate 包含以下字段：

```
typedef struct iov_param_validate {
     char pv_reason[MAX_REASON_LEN + 1];
     int32_t pv_buflen;
     /* encoded buffer containing params */
    char pv_buf[1];  /* size of buf is pv_buflen */
} iov_param_validate_t;
```

其中：

pv_reason　　ioctl 调用失败时用于解释失败原因的一个 ASCII 字符串

pv_buflen　　缓冲区 *pv_buf* 的长度

pv_buf　　包含参数的缓冲区

iov_param_desc 结构

iov_param_desc 结构定义如下：

```
    #define IOV_IOCTL  (('I' << 24) | ('O' << 16) | ('V' << 8))
  #define IOV_GET_VER_INFO (IOV_IOCTL | 0)
  #define IOV_GET_PARAM_INFO (IOV_IOCTL | 1)
  #define IOV_VALIDATE_PARAM (IOV_IOCTL | 2)
  #define IOV_PARAM_DESC_VERSION  1
```

`iov_param_desc` 结构包含以下字段：

```
typedef struct iov_param_desc {
char pd_name[MAX_PARAM_NAME_SIZE];
char pd_desc[MAX_PARAM_DESC_SIZE];
int32_t pd_flag;   /* applicable for PF or VF or both */
int32_t pd_data_type; /* integer, string, plist */
/* Following 3 are applicable for integer data types */
uint64_t pd_default_value;
uint64_t pd_min64;
uint64_t pd_max64;
char pd_default_string [MAX_PARAM_DEFAULT_STRING_SIZE];
} iov_param_desc_t;
```

其中：

`pd_name`	在 `ldm(1M)` 命令或 `pci.conf` 文件中使用，用于为参数指定值。
`pd_desc`	参数的简短说明。
`pd_flag`	指示参数是仅适用于 PF，还是仅适用于 VF，亦或是同时适用于 PF 和 VF。
`pd_default_value`	未在 `ldm()` 命令或 `pci.conf` 文件中指定参数时由驱动程序指定的值。
`pd_min64`	指定整数参数值的最小范围。
`pd_max64`	指定整数参数值的最大范围。
`pd_default_string`	指定将使用的缺省字符串（如果参数是字符串）。

IOV_GET_VER_INFO Ioctl

实现了 `IOV_GET_VER_INFO IOCTL()` ioctl 的 SR-IOV 设备驱动程序应在 `iov_param_ver_info` 结构中设置 *version* 和 *num_params* 字段，并将值返回到调用函数。调用函数随后将使用 *version* 和 *num_params* 参数确定使用 `IOV_GET_PARAM_INFO()` ioctl 调用获取参数描述所需的缓冲区大小。

IOV_GET_PARAM_INFO Ioctl

调用 `IOV_GET_PARAM_INFO()` ioctl 的驱动程序的一般控制流如下所述：

1. 保留 iov_param_desc_t 结构的数组，这些结构包含各自支持的每个可配置参数的说明。有关结构说明的信息，请参见 iov_param_desc 结构。
2. 将 iov_param_desc_t 结构的数组复制到 arg 参数。iov_param_desc_t 结构中的字段是静态字段，可在编译时定义。

   数组中元素的数量是 IOV_GET_VER_INFO() ioctl 调用返回的 num_params 值。缓冲区的大小为 sizeof (iov_param_desc_t) * num_params。

IOV_VALIDATE_PARAM Ioctl

对调用 IOV_VALIDATE_PARAM() 的驱动程序的一般控制流程如下所述：

1. 将 *arg* 参数发送到 pci_param_get_ioctl() 接口并获取指向 *pci_param_t* 结构的指针。
2. 当 *param* 验证失败时，向 *pv_reason* 数组写入一个解释性的字符串。
3. 依次调用 pci_get_plist() 接口和 pci_plist_lookup() 接口来获取设备参数。
4. 在 PF plist 中查找 *vfs* 名称-值对以获取要针对此配置的验证而配置的 VF 的数量。驱动程序应使用长度至少为 16 位的整数数据类型查找 *vfs* 名称-值对。使用 pciv_plist_getvf() 接口获取 VF 设备的 *plist* 参数。
5. 在不实际将参数应用于设备的情况下对参数进行验证。
6. 在找到有效配置时返回 0。

注意 – 上述过程中验证的参数与设备的当前配置毫不相关。它们需要单独进行验证（假定它们可以进一步配置）。如果不单独进行验证，驱动程序应返回 DDI_EINVAL 来指示配置不正确。当发现了无效配置时，驱动程序还应在 iov_param_validate 结构的 *pv_reason* 字段中提供一个解释性字符串。此字符串会将配置失败的原因告知管理员。

驱动程序回调

DDI 接口 ddi_cb_register() 和 ddi_cb_unregister() 用来注册回调。回调用于事件通知和传入数据通信。回调在传输期间充当每个事件的事件处理程序。

SR-IOV 驱动程序应实现其他回调以在配置或取消配置 VF 前后通知 PF 驱动程序。

驱动程序可以使用以下 DDI 回调注册机制接口实现回调：

- ```
  int ddi_cb_register(dev_info_t *dip, ddi_cb_flags_t flags, ddi_cb_func_t cbfunc,
  void *arg1, void *arg2,ddi_cb_handle_t *ret_hdlp)
  ```

  有关详细信息，请参见 ddi_cb_register(9F) 手册页。
  ```

-

```
typedef int(*ddi_cb_func_t)
(dev_info_t *dip, ddi_cb_action_t action,
void *cbarg, void *arg1, void *arg2)
```

其中：

`dip` 是指向 `dev_info` 结构的指针。

`cbarg` 是指向 `pciv_config_vf_t` 结构的指针。

`action` 设置为 `DDI_CB_PCIV_CONFIG_VF` 以接收有关对 VF 配置所做更改的通知。

`arg1` 在每次执行自己的 `cbfunc()` 例程期间发送给自己的专用参数。

`arg2` 在每次执行自己的 `cbfunc()` 例程期间发送给自己的专用参数。

有关更多信息，请参见第 130 页中的 “注册回调处理程序函数”。

注 – 具有 SR-IOV 功能的所有 PF 驱动程序必须使用 `ddi_cb_flags_t DDI_CB_FLAG_SRIOV` 来通知 Oracle Solaris IOV 框架 PF 驱动程序具有 SR-IOV 功能。

驱动程序 Ioctl 的样例代码

```
enum ioc_reply
igb_ioctl(igb_t *igb, struct iocblk *iocp, mblk_t *mp)
{
        int rval = 0;
        iov_param_ver_info_t *iov_param_ver;
        iov_param_validate_t pvalidate;
        pci_param_t my_params;
        char reason[81];

if (mp->b_cont == NULL)
        return (IOC_INVAL);
if ((int)iocp->ioc_count < 0)
        return (IOC_INVAL);
switch (iocp->ioc_cmd) {
        case IOV_GET_PARAM_VER_INFO:
            if (iocp->ioc_count < sizeof (iov_param_ver_info_t))
                return (IOC_INVAL);
            iov_param_ver = (iov_param_ver_info_t *)(mp->b_cont->b_rptr);
            iov_param_ver->version = IOV_PARAM_DESC_VERSION;
            iov_param_ver->num_params = NUM_OF_PARAMS;
        return (IOC_REPLY);
        case IOV_GET_PARAM_INFO:
            if (iocp->ioc_count < sizeof (pci_list))
                return (IOC_INVAL);
            memcpy((caddr_t)(mp->b_cont->b_rptr), &pci_list,sizeof (pci_list));
            return (IOC_REPLY);
        case IOV_VALIDATE_PARAM:
            if (iocp->ioc_count <= 0)
```

```
                return (IOC_INVAL);
            strcpy(reason, "Failed to read params sent\n");
            rval = pci_param_get_ioctl(igb->dip,(uintptr_t)(mp->b_cont->b_rptr),
                    iocp->ioc_flag | FKIOCTL,&my_params );
            if (rval == 0) {
                rval = validate_params(igb->dip, my_params, reason);
            pci_param_free(my_params);
    }
if (rval) {
        memcpy(mp->b_cont->b_rptr, reason, sizeof (reason));
        iocp->ioc_count = sizeof (reason);
        return (IOC_REPLY);
}
iocp->ioc_count = 0;
return (IOC_REPLY);
iov_param_ver_info_t iov_param_ver;
iov_param_validate_t pvalidate;
pci_param_t    my_params;

switch (cmd) {
case IOV_GET_PARAM_VER_INFO:
        iov_param_ver.version = IOV_PARAM_DESC_VERSION;
        iov_param_ver.num_params = NUM_OF_PARAMS;
        if (ddi_copyout(&iov_param_ver, (caddr_t)arg,
            sizeof (iov_param_ver_info_t), mode) != DDI_SUCCESS)
        return (DEFAULT);
        return (0);
case IOV_GET_PARAM_INFO:
        if (ddi_copyout(&pci_list, (caddr_t)arg,param_list_size, mode) != DDI_SUCCESS)
        return (DEFAULT);
        return (0);
case IOV_VALIDATE_PARAM:
        strcpy(reason, "Failed to read params sent\n");
        rv = pci_param_get_ioctl(state->dip, arg, mode, &my_params);
        if (rv == 0)
            rv = validate_params(state->dip, my_params, reason);
        else
            return (rv);
        pci_param_free(my_params);
    if (rv) {
        if (ddi_copyout(reason,iov_param_validate_t *)arg)->pv_reason,
             sizeof (reason), mode) != DDI_SUCCESS)
        return (DEFAULT);
 return (rv);
    }
return (0);
```

第 3 部分

生成设备驱动程序

本书的第三部分在为 Oracle Solaris OS 生成设备驱动程序方面提供了建议：

- 第 22 章，编译、装入、打包和测试驱动程序提供了有关编译、链接和安装驱动程序的信息。
- 第 23 章，调试、测试和调优设备驱动程序介绍了有关调试、测试和调优驱动程序的技术。
- 第 24 章，推荐的编码方法介绍了推荐的用于编写驱动程序的编码惯例。

◆ ◆ ◆ 第 22 章

22 编译、装入、打包和测试驱动程序

本章介绍驱动程序的开发过程，包括代码布局、编译、打包和测试。

本章介绍有关以下主题的信息：

驱动程序开发摘要

本章以及后面的两章，第 23 章，调试、测试和调优设备驱动程序和第 24 章，推荐的编码方法，提供了有关开发设备驱动程序的详细信息。

可采用以下步骤来生成设备驱动程序：

1. 编写、编译和链接新代码。

   有关文件的命名约定，请参见第 452 页中的 “驱动程序代码布局”。使用 C 编译器编译驱动程序。使用 `ld(1)` 链接驱动程序。请参见第 454 页中的 “编译和链接驱动程序”和第 455 页中的 “模块相关性”。

2. 创建必需的硬件配置文件。

   创建一个特定于名为 *xx*`.conf` 的设备的硬件配置文件，其中 *xx* 为设备的前缀。该文件用于更新 `driver.conf(4)` 文件。请参见第 455 页中的 “编写硬件配置文件”。对于伪设备驱动程序，需要创建一个 `pseudo(4)` 文件。

3. 将驱动程序复制到相应的模块目录。

   请参见第 456 页中的 “将驱动程序复制到模块目录”。

4. 使用 `add_drv(1M)` 安装设备驱动程序。

请参见第 457 页中的 “使用 add_drv 安装驱动程序”。update_drv(1M) 命令用于对驱动程序进行更改。请参见第 458 页中的 “更新驱动程序信息”。

5. 装入驱动程序。

   通过访问设备可自动装入驱动程序。请参见第 458 页中的 “装入和卸载驱动程序”。另外，也可以使用 modload(1M) 命令装入驱动程序。modload 命令不会调用模块中的任何例程，因此适用于进行测试。请参见第 468 页中的 “装入和卸载测试模块”。

6. 测试驱动程序。

   驱动程序应在以下方面进行严格的测试：

   - 第 459 页中的 “配置测试”
   - 第 459 页中的 “功能测试”
   - 第 460 页中的 “错误处理”
   - 第 460 页中的 “测试装入和卸载”
   - 第 460 页中的 “压力、性能和互操作性测试”
   - 第 461 页中的 “DDI/DKI 兼容性测试”
   - 第 461 页中的 “安装和打包测试”

   有关其他特定于驱动程序的测试，请参见第 461 页中的 “测试特定类型驱动程序”。

7. 删除驱动程序（如有必要）。

   使用 rem_drv(1M) 命令可删除设备驱动程序。请参见第 458 页中的 “删除驱动程序”。

驱动程序代码布局

设备驱动程序代码通常分为以下文件：

- 头文件（.h 文件）
- 源文件（.c 文件）
- 可选配置文件（driver.conf 文件）

头文件

头文件提供以下定义：

- 特定于设备的数据结构，如表示设备寄存器的结构
- 驱动程序定义的用于维护状态信息的数据结构
- 定义的常数，如表示设备寄存器位的常数
- 宏，如定义次要设备号与实例编号之间的静态映射的宏

某些头文件定义（如状态结构）可能只有设备驱动程序才需要。这些信息应该放在设备驱动程序本身所包含的**专用**头文件中。

应用程序可能需要的任何信息（如 I/O 控制命令）均应放在**公共**头文件中。这些文件包含在驱动程序和任何需要设备相关信息的应用程序中。

虽然专用文件和公共文件并没有命名标准，但一种约定是将专用头文件命名为 *xx*`impl.h`，将公共头文件命名为 *xx*`io.h`。

源文件

设备驱动程序的 C 源文件 （`.c` 文件）具有以下职责：

- 包含驱动程序入口点的代码和数据声明
- 包含驱动程序所需的 `#include` 语句
- 声明 `extern` 引用
- 声明局部数据
- 设置 `cb_ops` 和 `dev_ops` 结构
- 声明并初始化模块配置部分，即 `modlinkage(9S)` 和 `modldrv(9S)` 结构
- 进行任何其他必要的声明
- 定义驱动程序入口点

配置文件

一般来说，驱动程序的配置文件定义驱动程序需要的所有属性。驱动程序配置文件中的项指定了驱动程序可以探测其存在情况的可能设备实例。可以在驱动程序的配置文件中设置驱动程序的全局属性。有关更多信息，请参见 `driver.conf(4)` 手册页。

驱动程序配置文件对于非自标识设备是必需的。

驱动程序配置文件对于自标识设备 (self-identifying device, SID) 是可选的。对于自标识设备，配置文件可用于向 SID 节点中添加属性。

以下属性是**不**在驱动程序配置文件中设置的属性示例：

- 使用 S 总线外围总线的驱动程序一般从 S 总线卡获取属性信息。如果需要其他属性，驱动程序配置文件可以包含由 `sbus(4)` 定义的属性。
- PCI 总线的属性通常可以从 PCI 配置空间派生而来。如果需要专用驱动程序属性，驱动程序配置文件可以包含由 `pci(4)` 定义的属性。
- ISA 总线上的驱动程序可以使用由 `isa(4)` 定义的其他属性。

准备安装驱动程序

安装驱动程序之前，需要执行以下步骤：

1. 编译驱动程序。
2. 创建配置文件（如有必要）。
3. 通过以下任一备选方法，在系统中标识驱动程序模块：

- 将驱动程序的名称与设备节点的名称匹配。
- 使用 add_drv(1M) 或 update_drv(1M) 将模块名称通知给系统。

系统维护驱动程序模块名称与 dev_info 节点名称之间的一对一关联。例如，假设名为 mydevice 的设备包含一个 dev_info 节点。处理设备 mydevice 的驱动程序模块也命名为 mydevice。mydevice 模块驻留在名为 drv 的子目录中，该子目录位于模块路径下。如果使用 32 位内核，则该模块位于 drv/mydevice 中。如果使用 64 位 SPARC 内核，则该模块位于 drv/sparcv9/mydevice 中。如果使用 64 位 x86 内核，则该模块位于 drv/amd64/mydevice 中。

如果驱动程序是一个 STREAMS 网络驱动程序，则驱动程序名称必须满足以下约束：

- 只允许使用字母数字字符 (a-z, A-Z, 0-9) 以及下划线 ('_')。
- 名称的第一个字符和最后一个字符都不能是数字。
- 名称的长度不能超过 16 个字符。最好使用长度在 3 到 8 个字符范围内的名称。

如果驱动程序必须使用不同的名称管理 dev_info 节点，则 add_drv(1M) 实用程序可以创建别名。-i 标志指定驱动程序处理的其他 dev_info 节点的名称。update_drv 命令也可以修改已安装的设备驱动程序的别名。

编译和链接驱动程序

您需要编译每个驱动程序源文件，并将生成的对象文件链接到驱动程序模块中。Oracle Solaris OS 既与 Oracle Solaris Studio C 编译器兼容，又与 Free Software Foundation, Inc. 提供的 GNU C 编译器兼容。除非另有说明，否则本节中的示例均使用 Oracle Solaris Studio C 编译器。有关 Oracle Solaris Studio C 编译器的信息，请参见《Oracle Solaris Studio 12.3：C 用户指南》和 Oracle Solaris Studio Documentation（Oracle Solaris Studio 文档）。有关编译和链接选项的更多信息，请参见 Oracle Solaris Studio 12.3 Command-Line Reference（Oracle Solaris Studio 12.3 命令行参考信息）文档。/usr/sfw 目录中提供了 GNU C 编译器。有关 GNU C 编译器的信息，请参见 http://gcc.gnu.org/，或者查看 /usr/sfw/man 中的手册页。

以下示例显示一个名为 *xx* 的驱动程序，该驱动程序包含两个 C 源文件。生成的驱动程序模块名为 *xx*。本示例中创建的驱动程序适用于 32 位内核。您必须使用 ld -r，即使您的驱动程序只有一个对象模块也是如此。

```
% cc -D_KERNEL -c xx1.c
% cc -D_KERNEL -c xx2.c
% ld -r -o xx xx1.o xx2.o
```

必须定义 _KERNEL 符号以指示此代码定义了一个内核模块。除了驱动程序专用符号以外，不应定义任何其他符号。可以定义 DEBUG 符号，以启用任何对 ASSERT(9F) 的调用。

表 22–1 SPARC 和 x86 64 位体系结构的编译器选项

编译器	SPARC 64	x86 64
Studio 9	`cc -D_KERNEL -xarch=v9 -c xx.c`	不支持
Studio 10	`cc -D_KERNEL -xarch=v9 -c xx.c`	`cc -D_KERNEL -xarch=amd64 -xmodel=kernel -c xx.c`
Studio 11	`cc -D_KERNEL -xarch=v9 -c xx.c`	`cc -D_KERNEL -xarch=amd64 -xmodel=kernel -c xx.c`
Oracle Solaris Studio 12	`cc -D_KERNEL -m64 -c xx.c`	`cc -D_KERNEL -m64 -xmodel=kernel -c xx.c`

注 – 如果您使用 Oracle Solaris Studio 12.2 或更旧的编译器针对 32 位或 64 位 x86 体系结构进行编译，请勿添加 **`-xarch=sse2a`**，因为缺省值是 SSE2。

注意 – MMX 指令在 x86 内核中不受支持。使用 MMX 指令会引起内核故障，因此不应使用它。

驱动程序稳定后，您可能需要添加优化标志来生成符合生产质量要求的驱动程序。有关 Oracle Solaris Studio C 编译器中的优化的特定信息，请参见 Oracle Solaris Studio 12.3 Command-Line Reference（Oracle Solaris Studio 12.3 命令行参考信息）中的 cc(1) 手册页。

在设备驱动程序中，应该将全局变量视为 `volatile`。`volatile` 标记将在第 494 页中的“将变量声明为可变变量”中详细介绍。标志的使用取决于平台。请参见手册页。

模块相关性

如果驱动程序模块依赖于其他内核模块导出的符号，则可以通过装载器 ld(1) 的 `-dy` 和 `-N` 选项指定相关性。如果驱动程序依赖于 `misc/mySymbol` 导出的符号，则应使用以下示例创建驱动程序二进制文件。

```
% ld -dy -r -o xx xx1.o xx2.o -N misc/mySymbol
```

编写硬件配置文件

如果设备是非自标识设备，则内核需要该设备的硬件配置文件。如果驱动程序名为 *xx*，则该驱动程序的硬件配置文件应该命名为 *xx*`.conf`。

在 x86 平台上，设备信息现在由引导系统提供。不再需要硬件配置文件，即使非自我识别设备也不再需要它们。

有关硬件配置文件的更多信息，请参见 driver.conf(4)、pseudo(4)、sbus(4)、scsi_free_consistent_buf(9F) 和 update_drv(1M) 手册页。

在硬件配置文件中可以定义任意属性。配置文件中项的形式为 *property= value*，其中 *property* 是属性名称，*value* 是其初始值。借助配置文件方法，可以通过更改属性值来配置设备。

安装、更新和删除驱动程序

必须先将驱动程序已存在的信息通知系统，然后才能使用该驱动程序。必须使用 add_drv(1M) 实用程序来正确安装设备驱动程序。安装驱动程序之后，无需使用 add_drv 命令便可从内存中装入和卸载该驱动程序。

将驱动程序复制到模块目录

设备驱动程序模块的路径取决于以下三个条件：

- 运行驱动程序的平台
- 编译驱动程序时采用的体系结构
- 引导时是否需要该路径

设备驱动程序驻留在以下位置：

`/platform/‘uname -i‘/kernel/drv`
包含仅在特定平台上运行的 32 位驱动程序。

`/platform/‘uname -i‘/kernel/drv/sparcv9`
包含仅在基于 SPARC 的特定平台上运行的 64 位驱动程序。

`/platform/‘uname -i‘/kernel/drv/amd64`
包含仅在基于 x86 的特定平台上运行的 64 位驱动程序。

`/platform/‘uname -m‘/kernel/drv`
包含仅在特定平台系列上运行的 32 位驱动程序。

`/platform/‘uname -m‘/kernel/drv/sparcv9`
包含仅在基于 SPARC 的特定平台系列上运行的 64 位驱动程序。

`/platform/‘uname -m‘/kernel/drv/amd64`
包含仅在基于 x86 的特定平台系列上运行的 64 位驱动程序。

`/usr/kernel/drv`
包含与平台无关的 32 位驱动程序。

`/usr/kernel/drv/sparcv9`
包含基于 SPARC 的系统上与平台无关的 64 位驱动程序。

/usr/kernel/drv/amd64
包含基于 x86 的系统上与平台无关的 64 位驱动程序。

要安装 32 位驱动程序，必须将驱动程序及其配置文件复制到模块路径中的 drv 目录。例如，要将驱动程序复制到 /usr/kernel/drv，请键入：

```
$ su
# cp xx /usr/kernel/drv
# cp xx.conf /usr/kernel/drv
```

要安装 SPARC 驱动程序，请将驱动程序复制到模块路径中的 drv/sparcv9 目录。将驱动程序配置文件复制到模块路径中的 drv 目录。例如，要将驱动程序复制到 /usr/kernel/drv，应键入：

```
$ su
# cp xx /usr/kernel/drv/sparcv9
# cp xx.conf /usr/kernel/drv
```

要安装 64 位 x86 驱动程序，请将驱动程序复制到模块路径中的 drv/amd64 目录。将驱动程序配置文件复制到模块路径中的 drv 目录。例如，要将驱动程序复制到 /usr/kernel/drv，应键入：

```
$ su
# cp xx /usr/kernel/drv/amd64
# cp xx.conf /usr/kernel/drv
```

注 – 所有驱动程序配置文件 （.conf 文件）都必须放入模块路径中的 drv 目录。不能将 .conf 文件放入 drv目录的任何子目录。

使用 add_drv 安装驱动程序

使用 add_drv(1M) 命令可将驱动程序安装到系统中。如果驱动程序成功安装， add_drv 将运行 devfsadm(1M)，以便在 /dev 目录中创建逻辑名称。

```
# add_drv xx
```

在本例中，设备将自身标识为 *xx*。设备的特殊文件具有缺省的拥有权和权限 (0600 root sys)。add_drv 命令也允许为设备指定其他名称（别名）。有关显式添加别名和设置文件权限的信息，请参见 add_drv(1M) 手册页。

注 – 请勿使用 add_drv 命令安装 STREAMS 模块。有关详细信息，请参见《STREAMS Programming Guide》 。

如果驱动程序创建了不表示终端设备（如磁盘、磁带或者端口）的次要节点，则可以修改 /etc/devlink.tab，以使 devfsadm 在 /dev 中创建逻辑设备名称。另外，也可以通过安装驱动程序时运行的程序来创建逻辑名称。

更新驱动程序信息

使用 update_drv(1M) 命令可以通知系统对已安装的设备驱动程序所做的任何更改。缺省情况下，系统将会重新读取驱动程序配置文件，并重新装入驱动程序二进制模块。

删除驱动程序

要从系统中删除驱动程序，请使用 rem_drv(1M) 命令，然后从模块路径中删除驱动程序模块和配置文件。使用 add_drv(1M) 重新安装驱动程序之前，无法再使用该驱动程序。删除 SCSI HBA 驱动程序需要重新引导才能生效。

装入和卸载驱动程序

打开与设备驱动程序关联的特殊文件（访问该设备）即可装入该驱动程序。您可以使用 modload(1M) 命令将驱动程序装入内存，但 modload 不会调用模块中的任何例程。 首选方法是打开设备。

通常，当不再使用设备时，系统会自动卸载设备驱动程序。在开发过程中，要显式卸载驱动程序，可能需要使用 modunload(1M)。为了成功执行 modunload，设备驱动程序必须处于非活 动状态。对设备的任何未完成引用（如通过 open(2) 或 mmap(2) 的引用）均不应存在。

modunload 命令将与运行时相关的 module_id 用作参数。要查找 module_id，请使用 grep 在 modinfo(1M) 的输出中搜索相关的驱动程序名称。然后检查第一列。

```
# modunload -i module-id
```

要卸载当前无法装入的所有模块，请将模块 ID 指定为零：

```
# modunload -i 0
```

要成功执行 modunload(1M)，驱动程序除了需要处于非活动状态外，还必须包含正常的 detach(9E) 和 _fini(9E) 例程。

驱动程序打包

软件的标准交付方式是创建一个包含所有软件组件的软件包。软件包为软件产品所有组件的安装和删除提供了一种受控机制。在对驱动程序进行打包时，开发者可以利用映像包管理系统 (Image Packaging System, IPS) 中的驱动程序操作来执行任务。有关更多信息，请参阅以下文档：

- 《在 Oracle Solaris 11.1 中使用映像包管理系统打包和交付软件》中的第 2 章 “使用 IPS 打包软件”

- 《添加和更新 Oracle Solaris 11.1 软件包》
- pkg(5)

在 Oracle Solaris 以前的版本中，使用的是 SVR4 包管理系统，它要求在软件包中包括安装后和删除前脚本以运行 add_drv(1M) 和 rem_drv(1M) 命令。使用这些命令的现有驱动程序软件包可能仍会安装在 Oracle Solaris 11 中。

驱动程序测试条件

设备驱动程序可以正常运行后，应在分发之前全面测试该驱动程序。除了测试传统 UNIX 设备驱动程序中的功能以外，Oracle Solaris 驱动程序还需要测试电源管理功能，如驱动程序的动态装入和卸载。

配置测试

驱动程序能否处理多种设备配置是测试过程的重要部分。驱动程序可以在一种简单（或缺省）配置中正常工作后，还应该测试其他配置。根据设备不同，可以通过更改跳线或 DIP 开关来完成配置测试。如果可能的配置数较少，则应尝试测试所有配置。如果可能的配置数较多，则应为这些可能的配置定义不同的类，并抽样测试每类配置。定义这些类取决于不同配置参数之间的潜在交互。这些交互是设备类型与驱动程序编写方式之间的配合。

对于每种设备配置，必须测试基本运行情况，包括装入、打开、读取、写入、关闭和卸载驱动程序。任何取决于配置的运行情况都需要特别注意。例如，更改设备寄存器的基本内存地址不可能影响大多数驱动程序函数的行为。如果驱动程序在一个地址上正常工作，则该驱动程序在其他地址上也能正常工作。另一方面，特殊 I/O 控制调用的结果可能会因特定设备配置而异。

以不同配置装入驱动程序可确保 probe(9E) 和 attach(9E) 入口点能够在不同地址找到设备。对于基本功能测试，对于字符设备，使用常规的 UNIX 命令（如 cat(1) 或 dd(1M)）通常就足够了。对于块设备，可能需要挂载或引导。

功能测试

对驱动程序进行全面的配置测试之后，应全面测试驱动程序的所有功能。这些测试需要执行驱动程序所有入口点的操作。

许多驱动程序需要使用定制的应用程序来测试功能。但是，对于磁盘、磁带或异步板等设备的基本驱动程序，使用标准系统实用程序即可进行测试。在此过程中，应测试所有入口点，包括 devmap(9E)、chpoll(9E) 和 ioctl(9E)（如果适用）。对于每种驱动程序，ioctl () 测试可能完全不同。对于非标准设备，通常需要使用定制的测试应用程序。

错误处理

在理想环境中，驱动程序也许可以正确执行，但如果出现错误（如操作错误或数据错误），则可能会失败。因此，驱动程序测试的一个重要部分是测试驱动程序的错误处理。

应该执行驱动程序的所有可能的错误情况，包括实际硬件故障导致的错误情况。某些硬件错误情况可能难于引发，但如有可能，应尽力强制引发或模拟此类错误。在实际使用时，所有这些情况都有可能遇到。为了测试以找出这些错误的根源，应该拆除或松开电缆、拆除板以及编写错误的用户应用程序代码。另请参见第 13 章，强化 Oracle Solaris 驱动程序。

注意 – 测试时，请务必采取正确的电气预防措施。

测试装入和卸载

由于没有装入或卸载的驱动程序会导致意外的停机时间，因此必须全面测试装入和卸载。

与以下示例类似的脚本应该足够满足要求：

```
#!/bin/sh
cd <location_of_driver>
while [ 1 ]
do
    modunload -i 'modinfo | grep " <driver_name> " | cut -cl-3' &
    modload <driver_name> &
done
```

压力、性能和互操作性测试

为有助于确保驱动程序正常执行，应对该驱动程序进行强有力的压力测试。例如，通过驱动程序运行单个线程并不会测试必须等待的锁定逻辑或条件变量。设备操作应由多个进程同时执行，以使几个线程同时执行同一代码。

执行同时测试的方法取决于驱动程序。某些驱动程序需要使用特殊的测试应用程序，而在后台启动多个 UNIX 命令适用于其他驱动程序。正确的测试取决于特定驱动程序在何处使用锁定和条件变量。在多处理器计算机上测试驱动程序比在单处理器计算机上测试更有可能暴露问题。

此外，还必须测试驱动程序之间的互操作性，尤其是在不同的设备可以共享中断级别的情况下。如有可能，请配置与正在测试的设备的中断级别相同的另一个设备。压力测试可以确定驱动程序是否正确请求其自己的中断，以及是否按照预期目标运行。应

该对两个设备同时运行压力测试。即使设备不共享中断级别，该测试仍然很重要。例如，假设在测试某个网络驱动程序时，串行通信设备遇到错误。同一问题也可能会导致系统的其余部分遇到中断延迟问题。

这些压力测试下的驱动程序性能应使用 UNIX 性能度量工具进行度量。此类测试与使用 time(1) 命令以及压力测试所用命令一样简单。

DDI/DKI 兼容性测试

为确保与更高发行版的兼容性以及对当前发行版的可靠支持，每个驱动程序都应该与 DDI/DKI 兼容。检查是否仅使用了《man pages section 9: DDI and DKI Kernel Functions》和《man pages section 9: DDI and DKI Driver Entry Points》中的内核例程以及《man pages section 9: DDI and DKI Properties and Data Structures》中的数据结构。

安装和打包测试

驱动程序是以**软件包**形式提供给客户的。可以使用某个标准机制在系统中添加或删除软件包。应对用户在系统中添加或删除软件包的能力进行测试。在测试过程中，应通过 `pkg install` 命令和 `pkg uninstall` 命令直接安装或卸载软件包。有关更多信息，请参见《添加和更新 Oracle Solaris 11.1 软件包》。

测试特定类型驱动程序

本节提供了一些有关如何测试某些类型的标准设备的建议。

测试磁带机

磁带机应通过执行多次归档和恢复操作来测试。cpio(1) 和 tar(1) 命令可用于此目的。使用 dd(1M) 命令可将整个磁盘分区写入磁带。接下来，读回数据，并将数据写入另一个相同大小的分区。然后比较这两个副本。mt(1) 命令可以执行特定于磁带机的大多数 I/O 控制。请参见 mtio(7I) 手册页。尝试使用所有选项。以下三种方法可以测试磁带机的错误处理能力：

- 移除磁带并尝试各种操作
- 对磁带进行写保护并尝试写入
- 在不同操作的执行过程中关闭电源

磁带机通常实现以独占方式访问的 open(9E) 调用。可以通过打开设备，然后让另一个进程尝试打开同一设备，来测试这些 `open()` 调用。

测试磁盘驱动程序

磁盘驱动程序应在原始设备模式和块设备模式下进行测试。对于块设备测试，请在设备上创建一个新的文件系统。然后，尝试挂载该新文件系统，并尝试执行多种文件操作。

注 – 文件系统使用页缓存，因此重复读取相同文件实际上并不会执行驱动程序。通过使用 mmap(2) 对文件进行内存映射，可以强制页缓存从设备中检索数据。然后使用 msync(3C) 使内存中的副本无效。

将另一个相同大小的（未挂载）分区复制到原始设备。然后使用 fsck(1M) 之类的命令检验副本的正确性。新分区也可以挂载，然后以后与旧分区进行逐文件比较。

异步通信驱动程序

通过为串行端口设置一个 login 连接线，可对异步驱动程序进行基本级别的测试。是否为一种较好的测试方法要看用户是否可以通过此连接线登录。但是，要充分测试异步驱动程序，必须使用多个高速中断来测试所有 I/O 控制函数。涉及回送串行电缆和较高数据传输速率的测试有助于确定驱动程序的可靠性。您可以在该连接线上运行 uucp(1C)，以提供某些实践。但是，由于 uucp 执行其自己的错误处理，因此请确认驱动程序不会向 uucp 进程报告过多的错误。

这些类型的设备通常是基于 STREAMS 的。有关更多信息，请参见《STREAMS Programming Guide》。

测试网络驱动程序

可以使用标准网络实用程序对网络驱动程序进行测试。由于可在网络的每个端点上比较文件，因此 ftp(1) 和 rcp(1) 命令非常有用。该驱动程序应在网络负载较重的情况下测试，以便多个进程可以运行各种命令。

网络负载较重包括以下情况：

- 测试计算机的通信流量较大。
- 网络上所有计算机之间的通信流量较大。

执行测试时，应拔下网络电缆，以确保驱动程序可从产生的错误情况中正常恢复。另一项重要测试是让驱动程序快速连续接收多个包，即**背对背**包。在这种情况下，负载较轻的网络上相对较快的主机应向测试计算机快速连续发送多个包。请检验接收驱动程序不会丢弃第二个以及后续的包。

这些类型的设备通常是基于 STREAMS 的。有关更多信息，请参见《STREAMS Programming Guide》。

测试 SR-IOV 驱动程序

支持 SR-IOV 的驱动程序需要进行额外测试。还需要进行标准裸机测试，可以使用用于裸机测试的实用程序，如用于网络设备的 ftp 和 rcp。

有关 SR-IOV 驱动程序的信息，请参见第 21 章，SR-IOV 驱动程序。

使用以下命令来测试虚拟功能 (Virtual Function, VF) 的状态：

- 已启用 VF－`hotplug install`
- 已禁用 VF－`hotplug uninstall`
- 已指定 VF－`hotplug list`

在 SPARC 系统上测试 VF 的状态时，除 `hotplug` 命令之外，还请使用 `ldm(1M)` 命令。

在各种虚拟化配置中测试 SR-IOV 设备也很重要。在 SPARC 和 x86 平台上测试 SR-IOV 驱动程序时，请尝试以下选项：

- 请勿配置任何虚拟功能 (Virtual Function, VF)
- 仅配置一个 VF
- 以 2 的幂次方增加配置的 VF，直到达到 VF 的最大数目

在 SPARC 平台上，使用不同数目的 IO 域并在这些域中使用不同的 VF 分布来测试功能。尝试以下配置：

- 将单个 VF 指定给单个 IO 域
- 将 2 的幂次方个 VF（可至最大值）指定给单个 IO 域
- 创建 2、4 或 8 个 IO 域并将不同数目的 VF 指定给每个域
- 将一些 VF 指定给根域并将一些 VF 指定给 IO 域

如果设备或平台支持以下功能，请对其进行测试：

- 从 VF 引导 IO 域
- 以物理形式热插入或拔出 SR-IOV 卡
- 在 SR-IOV 卡上执行动态重新配置操作

如果 Oracle Solaris 的多个版本都支持您的设备，对于某些测试，可通过在根域和 IO 域间混用 OS 版本来执行最终测试。

◆ ◆ ◆ 第 23 章

23 调试、测试和调优设备驱动程序

本章简要介绍用于帮助测试、调试和调优设备驱动程序的各种工具。本章介绍有关以下主题的信息：

- 第 465 页中的 “测试驱动程序” — 测试驱动程序可能会削弱系统执行操作的能力。同时使用串行连接和替代内核能够便于从崩溃中恢复。
- 第 472 页中的 “调试工具” — 借助不可或缺的调试工具，您可以方便地使用和查看驱动程序功能，而无需运行单独的调试器。
- 第 483 页中的 “调优驱动程序” — Oracle Solaris OS 提供用于度量设备驱动程序性能的工具。为设备编写内核统计信息结构可在设备运行时导出连续的统计信息。如果确定了要改进性能的方面，DTrace 动态检测过程工具有助于更精确地确定任何问题。

测试驱动程序

为了避免数据丢失和出现其他问题，在对新设备驱动程序进行测试时应特别注意。本节将对各种测试策略进行讨论。例如，设置可通过串行连接来控制的单独系统是对新驱动程序进行测试的最安全方法。可用不同的内核变量设置来装入测试模块，以便在不同的内核条件下测试性能。如果系统崩溃，应准备好恢复备份数据、分析任何故障转储并重新生成设备目录。

启用 Deadman 功能以避免硬挂起

如果系统处于硬挂起状态，则不能中断调试器。如果启用 deadman 功能，系统将出现紧急情况，而不是无限期挂起。您随后可以使用 `kmdb(1)` 内核调试器来分析您的问题。

Deadman 功能每秒检查一次系统时钟是否正在更新。如果系统时钟未在更新，则表明您处于无限期的挂起状态中。如果系统时钟在 50 秒内未更新，则 deadman 功能将导致出现紧急情况并会将您置于调试器中。

执行以下步骤以启用 deadman 功能：

1. 确保正在使用 dumpadm(1M) 捕获崩溃映像。
2. 在 /etc/system 文件中设置 snooping 变量。有关 /etc/system 文件的信息，请参见 system(4) 手册页。

   ```
   set snooping=1
   ```
3. 重新引导系统，以便再次读取 /etc/system 文件，从而使 snooping 设置生效。

请注意，系统中的所有区域也都会继承 deadman 设置。

如果系统在启用 deadman 功能的情况下挂起，则应在控制台上看到类似如下示例的输出：

```
panic[cpu1]/thread=30018dd6cc0: deadman: timed out after 9 seconds of
clock inactivity

panic: entering debugger (continue to save dump)
```

在调试器内，使用 ::cpuinfo 命令调查时钟中断无法触发的原因并使系统时间提前。

使用串行连接进行测试

使用串行连接是测试驱动程序的一种好方法。使用 tip(1) 命令可在主机系统和测试系统之间建立串行连接。借助此方法，可将主机控制台上的 *tip* **窗口**用作测试计算机的控制台。有关其他信息，请参见 tip(1) 手册页。

tip 窗口具有如下优点：

- 可以监视与测试系统和内核调试器的交互。例如，如果驱动程序使测试系统崩溃，则该窗口可以保留会话日志，以供使用。
- 通过登录 *tip* **主机**并使用 tip(1) 连接到测试计算机可以远程访问测试计算机。

注 – 尽管调试 Oracle Solaris 设备驱动程序时不要求使用 tip 连接和另一台计算机，但仍建议使用此方法。

▼ 针对 tip 连接设置主机系统

1 使用主机系统和测试计算机上的串行端口 A 将两台计算机连接起来。

必须使用空调制解调器电缆建立此连接。

2 **在主机系统中，确保 /etc/remote 中存在对应于该连接的项。有关详细信息，请参见 remote(4) 手册页。**

终端项必须与使用的串行端口匹配。操作系统附带对应于串行端口 B 的适当项，但必须为串行端口 A 添加一个终端项：

```
debug:\
        :dv=/dev/term/a:br#9600:el=^C^S^Q^U^D:ie=%$:oe=^D:
```

注 – 波特率必须设置为 9600。

3 **在主机上的 shell 窗口中，运行 tip(1) 并指定项的名称：**

```
% tip debug
connected
```

现在，shell 窗口是一个 tip 窗口，具有到测试计算机的控制台的连接。

注意 – 请勿在主机上使用 STOP-A（针对 SPARC 计算机）或 F1-A（针对 x86 体系结构计算机）来停止测试计算机。此操作实际上会停止主机。要向测试计算机发送中断，请在 tip 窗口中键入 **~#**。仅当诸如 **~#** 命令的这些字符位于行首才能识别这些命令。如果命令无效，请按回车键或 Ctrl-U 组合键。

在 SPARC 平台上设置目标系统

一种在 SPARC 平台上快速设置测试计算机的方法是在打开计算机之前拔除键盘。然后，该计算机将自动使用串行端口 A 作为控制台。

另一种设置测试计算机的方法是使用引导 PROM 命令使串行端口 A 成为控制台。在测试计算机上，在引导 PROM ok 提示符处，将控制台 I/O 定向到串行线路。要使测试计算机始终以串行端口 A 作为控制台，请设置环境变量 input-device 和 output-device。

示例 23–1 使用引导 PROM 命令设置 input-device 和 output-device

```
ok setenv input-device ttya
ok setenv output-device ttya
```

也可以使用 eeprom 命令使串行端口 A 成为控制台。以超级用户身份执行以下命令使 input-device 和 output-device 参数指向串行端口 A。以下示例显示了 eeprom 命令。

示例 23–2 使用 eeprom 命令设置 input-device 和 output-device

```
# eeprom input-device=ttya
# eeprom output-device=ttya
```

eeprom 命令会导致在以后每次系统引导时都将控制台重定向到串行端口 A。

在 x86 平台上设置目标系统

在 x86 平台上，使用 eeprom 命令可使串行端口 A 成为控制台。此过程与 SPARC 平台过程相同。请参见第 467 页中的 “在 SPARC 平台上设置目标系统”。eeprom 命令会使控制台在重新引导期间切换到串行端口 A (COM1)。

注 – 除非 BIOS 支持控制台重定向到串行端口，否则 x86 计算机在引导过程的早期阶段之前不会将控制台控制权转交给 tip 连接。在 SPARC 计算机中，tip 连接在整个引导过程中都维护着控制台控制权。

设置测试模块

使用 /etc 目录中的 system(4) 文件可在引导时设置内核变量的值。使用内核变量，可在驱动程序中切换不同的行为，并利用内核提供的调试功能。内核变量 moddebug 和 kmem_flags 在调试中非常有用，本节稍后将对其进行讨论。另请参见第 465 页中的 “启用 Deadman 功能以避免硬挂起”。

由于仅在内核引导时读取 /etc/system 一次，因此引导后对内核变量所做的更改不可靠。修改此文件后，必须重新引导系统，更改才能生效。如果文件中的更改导致系统无法工作，请使用询问功能 (-a) 选项进行引导。然后，将 /dev/null 指定为系统文件。

注 – 后续发行版中不一定存在内核变量。

设置内核变量

set 命令可以更改模块变量或内核变量的值。要设置模块变量，请指定模块名称和变量：

```
set module_name:variable=value
```

例如，要在名为 myTest 的驱动程序中设置变量 test_debug，请按如下方式使用 set：

```
% set myTest:test_debug=1
```

要设置由内核自身导出的变量，可忽略模块名称。

还可以使用按位 OR 运算设置值，例如：

```
% set moddebug | 0x80000000
```

装入和卸载测试模块

使用命令 modload(1M)、modunload(1M) 和 modinfo(1M) 可以添加测试模块，在对驱动程序进行调试和压力测试时，这是一种非常有用的方法。正常操作中通常不需要这些命令，因为内核会自动装入需要的模块并卸载未使用的模块。moddebug 内核变量可与这些命令一起使用，以提供信息并设置控制。

使用 modload() 函数

使用 modload(1M) 可将模块强制装入内存。modload 命令可验证装入驱动程序时该驱动程序是否具有未解析的引用。装入驱动程序并不表明该驱动程序一定可以连接。驱动程序成功装入时，将调用该驱动程序的 _info(9E) 入口点，但不一定调用 attach() 入口点。

使用 modinfo() 函数

使用 modinfo(1M) 可以确认驱动程序已装入。

示例 23–3 使用 modinfo 确认已装入的驱动程序

```
$ modinfo
 Id Loadaddr   Size Info Rev Module Name
  6 101b6000    732   -   1  obpsym (OBP symbol callbacks)
  7 101b65bd  1acd0 226   1  rpcmod (RPC syscall)
  7 101b65bd  1acd0 226   1  rpcmod (32-bit RPC syscall)
  7 101b65bd  1acd0   1   1  rpcmod (rpc interface str mod)
  8 101ce8dd  74600   0   1  ip (IP STREAMS module)
  8 101ce8dd  74600   3   1  ip (IP STREAMS device)
...
$ modinfo | grep mydriver
169 781a8d78   13fb   0   1  mydriver (Test Driver 1.5)
```

info 字段中的数字是为驱动程序选择的主设备号。如果提供了模块 ID，则可使用 modunload(1M) 命令来卸载模块。模块 ID 位于 modinfo 输出的左列中。

有时，发出 modunload 后驱动程序不会按预期卸载，因为该驱动程序被确定处于忙状态。由于驱动程序确实繁忙或 detach 入口点未正确实现而导致驱动程序无法执行 detach(9E) 时，会出现上述情况。

使用 modunload()

要从内存中删除所有当前未使用的模块，请使用模块 ID 0 运行 modunload(1M) :

```
# modunload -i 0
```

设置 moddebug 内核变量

moddebug 内核变量可控制模块装入过程。moddebug 的可能值包括 :

`0x80000000` 装入或卸载模块时向控制台列显消息。

`0x40000000` 提供更详细的错误消息。

`0x20000000` 装入或卸载时列显更多详细信息，如包含地址和大小。

`0x00001000` 不自动卸载驱动程序。系统资源变少时，系统不尝试卸载设备驱动程序。

`0x00000080` 不自动卸载流。系统资源变少时，系统不尝试卸载 STREAMS 模块。

`0x00000010` 不自动卸载任何类型的内核模块。

`0x00000001` 如果与 `kmdb` 一起运行，`moddebug` 会导致执行断点，并在调用每个模块的 `_init()` 例程之前立即返回到 `kmdb`。此设置还会在执行模块的 `_info()` 和 `_fini()` 例程时生成其他调试消息。

设置 kmem_flags 调试标志

`kmem_flags` 内核变量用于启用内核的内存分配器中的调试功能。将 `kmem_flags` 设置为 `0xf` 即启用分配器的调试功能。这些功能包括运行时检查，以找出以下代码条件：

- 释放缓冲区后向缓冲区写入
- 初始化内存之前使用内存
- 写入时已过缓冲区结尾

《Oracle Solaris Modular Debugger Guide》介绍了如何使用内核内存分配器来分析此类问题。

注 – 在将 `kmem_flags` 设置为 `0xf` 的情况下进行测试和开发有助于检测潜在的内存损坏错误。由于将 `kmem_flags` 设置为 `0xf` 会更改内核内存分配器的内部行为，因此最好应不使用 `kmem_flags` 而执行全面测试。

避免测试系统中发生数据丢失

驱动程序错误有时会导致系统无法引导。通过采取预防措施（如本节中所述），可以避免在此情况下重新安装系统。

使用备用引导环境

许多与驱动程序相关的系统文件都很难重新构造（如果可以重新构造）。如果安装期间驱动程序使系统崩溃，则诸如 `/etc/name_to_major`、`/etc/driver_aliases`、`/etc/driver_classes` 和 `/etc/minor_perm` 之类的文件会损坏。请参见 `add_drv(1M)` 手册页。

为保证安全，请在正确配置测试计算机后使用 `beadm(1M)` 命令创建根文件系统的备份副本。如果打算修改 `/etc/system` 文件，请在修改之前生成该文件的副本。

通过替代内核进行引导

有关详细信息，请参见《创建和管理 Oracle Solaris 11.1 引导环境》中的第 4 章 “管理引导环境”和《Oracle Solaris 管理：常见任务》中的“从 ZFS 引导环境引导（任务列表）”。

考虑替代备份计划

如果将系统连接到网络，则可将测试计算机添加为服务器的客户机。如果出现问题，可从网络引导系统。然后，便可挂载本地磁盘，并进行任何修复。也可以直接从 Oracle Solaris 系统 CD-ROM 引导系统。

另一种从灾难中恢复的方法是获取另一个可引导的根文件系统。使用 format(1M) 创建一个大小与原始分区完全相同的分区。然后，使用 dd(1M) 复制可引导的根文件系统。创建副本后，在新文件系统上运行 fsck(1M) 以确保其完整性。

以后，如果系统无法从原始根分区进行引导，可引导备份分区。可以使用 dd(1M) 将备份分区复制到原始分区。可能会遇到这样的情况，即使根文件系统未损坏，也无法引导系统。例如，只有引导块或引导程序损坏。在这种情况下，可使用询问功能 (-a) 选项从备份分区进行引导。然后，将原始文件系统指定为根文件系统。

捕获系统故障转储

当系统出现紧急情况时，系统会将内核内存的映像写入转储设备。缺省情况下，该转储设备是最合适的交换设备。该转储是系统故障转储，它与应用程序生成的核心转储类似。在系统出现紧急情况后进行重新引导时，savecore(1M) 会检查转储设备中是否存在故障转储。如果找到转储，savecore 将生成名为 unix.*n* 的内核符号表的副本。然后，savecore 实用程序将在核心映像目录中转储名为 vmcore.*n* 的核心文件。缺省情况下，核心映像目录为 /var/crash/*machine_name*。如果 /var/crash 没有足够的空间用于核心转储，系统将显示所需的空间，但不实际保存转储。然后，可针对核心转储和已保存的内核使用 mdb(1)。

在 Oracle Solaris 操作系统中，缺省情况下会启用故障转储。dumpadm(1M) 命令用于配置系统故障转储。使用 dumpadm 命令可以验证故障转储是否已启用，并确定保存核心文件的位置。

注 – 可以阻止 savecore 实用程序填满文件系统，即在要保存转储的目录中添加一个名为 minfree 的文件。在此文件中，指定在 savecore 运行后保持可用的千字节数。如果没有足够的可用空间，则不保存核心文件。

恢复设备目录

如果驱动程序在执行 attach(9E) 期间崩溃，可能会损坏 /devices 和 /dev 目录。如果任一目录被损坏，则可引导系统并运行 fsck(1M) 以修复损坏的根文件系统，从而重新生成目录。然后，便可挂载根文件系统。通过运行 devfsadm(1M) 并在已挂载的磁盘上指定 /devices 目录可以重新创建 /devices 和 /dev 目录。

以下示例说明如何在 SPARC 系统上修复损坏的根文件系统。在此示例中，损坏的磁盘为 /dev/dsk/c0t3d0s0，替代引导磁盘为 /dev/dsk/c0t1d0s0。

示例 23–4 恢复损坏的设备目录

```
ok boot disk1
...
Rebooting with command: boot kernel.test/sparcv9/unix
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@31,0:a File and \
    args:
kernel.test/sparcv9/unix
...
# fsck /dev/dsk/c0t3d0s0** /dev/dsk/c0t3d0s0
** Last Mounted on /
** Phase 1 - Check Blocks and Sizes
** Phase 2 - Check Pathnames
** Phase 3 - Check Connectivity
** Phase 4 - Check Reference Counts
** Phase 5 - Check Cyl groups
1478 files, 9922 used, 29261 free
     (141 frags, 3640 blocks, 0.4% fragmentation)
# mount /dev/dsk/c0t3d0s0 /mnt
# devfsadm -r /mnt
```

注 – 对 /devices 和 /dev 目录进行修复后，就可以在系统的其他部分仍处于损坏状态时引导系统。此类修复只是在重新安装系统前进行的临时修复，作用是保存信息（如系统故障转储）。

调试工具

本节介绍可以应用于设备驱动程序的两个调试器。《Oracle Solaris Modular Debugger Guide》中详细介绍了这两个调试器。

- kmdb(1) **内核调试器**可提供典型的运行时调试器功能，如断点、监视点和单步执行。kmdb 调试器取代了以前发行版中的 kadb。除新功能外，在 kmdb 中还可以使用先前在 kadb 中可用的命令。kadb 只能在引导时装入，而 kmdb 可随时装入。由于 kmdb 调试器可进行执行控制，因此它是用于实时、交互调试的首选方法。
- mdb(1) **模块调试器** 作为实时调试器比 kmdb 的功能要有限一些，但 mdb 具有很多可用于事后调试的功能。

kmdb 和 mdb 调试器的用户界面大部分是一样的。因此，许多调试方法都可在这两种工具中使用相同命令来应用。这两种调试器都支持宏、dcmd 和 dmod。*dcmd*（读作为 dee-command）是调试器中的例程，它可以访问当前目标程序的任何属性。dcmd 可在运行时动态装入。*dmod*（调试器模块的缩写）是可以装入以提供非标准行为的 dcmd 包。

mdb 和 kmdb 都可向后兼容传统调试器（如 adb 和 kadb）。mdb 调试器可以执行可用于 kmdb 的所有宏以及用于 adb 的任何用户定义的传统宏。有关在何处查找标准宏集的信息，请参见《Oracle Solaris 模块调试器指南》。

事后调试

事后分析为驱动程序开发者提供了许多益处。多个开发者可以并行检查一个问题。可以针对单个故障转储使用调试器的多个实例。可以脱机执行分析，以便在可能的情况下使崩溃的系统恢复运行。事后分析允许以 dmod 形式使用用户开发的调试器功能。Dmod 可为实时调试器（如 kmdb）捆绑内存密集程度过高的功能。

如果装入 kmdb 时系统出现紧急情况，则控制权会传递给调试器，以便立即进行检查。如果不适合使用 kmdb 分析当前问题，则使用 :c 继续执行并保存故障转储不失为一种好的策略。系统重新引导时，可以使用 mdb 对已保存的故障转储执行事后分析。此过程类似于从进程核心文件调试应用程序崩溃。

注 – 在 Oracle Solaris 操作系统的早期版本中，adb(1) 是推荐用于事后分析的工具。在当前的 Oracle Solaris 操作系统中，mdb(1) 是推荐用于事后分析的工具。mdb() 功能集不仅包含传统 crash (1M) 实用程序中的命令集，还具有更多功能。Oracle Solaris 操作系统中不再提供 crash 实用程序。

使用 kmdb 内核调试器

kmdb 调试器是可提供以下功能的交互式内核调试器：

- 控制内核执行
- 检查内核状态
- 实时修改代码

本节假定您已熟悉 kmdb 调试器。本节重点介绍在设备驱动程序设计中非常有用的 kmdb 功能。要详细了解如何使用 kmdb，请参阅kmdb(1) 手册页和《Oracle Solaris Modular Debugger Guide》。如果您熟悉 kadb，请参阅 kadb(1M) 手册页以了解 kadb 与 kmdb 的主要差别。

可以任意装入和卸载 kmdb 调试器。《Oracle Solaris 模块调试器指南》中包含有关装入和卸载 kmdb 的说明。为了安全和方便起见，强烈建议使用替代内核进行引导。如本节中所述，在 SPARC 平台与 x86 平台上引导过程略有不同。

注 – 缺省情况下，当 kmdb 运行时，kmdb 使用 CPU ID 作为提示符。在本章的示例中，除非另有指定，否则使用 [0] 作为提示符。

在 SPARC 平台上使用替代内核引导 kmdb

使用以下任一命令通过 kmdb 和替代内核引导 SPARC 系统：

```
boot kmdb -D kernel.test/sparcv9/unix
boot kernel.test/sparcv9/unix -k
```

在 x86 平台上使用替代内核引导 kmdb

使用以下任一命令通过 kmdb 和替代内核引导 x86 系统：

```
b kmdb -D kernel.test/unix
b kernel.test/unix -k
```

在 kmdb 中设置断点

使用 bp 命令设置断点，如以下示例中所示。

示例 23–5 在 kmdb 中设置标准断点

```
[0]> myModule‘myBreakpointLocation::bp
```

如果尚未装入目标模块，则会显示指示这一情况的错误消息，并且不会创建断点。在这种情况下，可以使用**延迟断点**。装入指定的模块时，会自动激活延迟断点。通过在 bp 命令后面指定目标位置可以设置延迟断点。以下示例对延迟断点进行了说明。

示例 23–6 在 kmdb 中设置延迟断点

```
[0]>::bp myModule‘myBreakpointLocation
```

有关使用断点的更多信息，请参见《Oracle Solaris 模块调试器指南》。也可以通过键入以下任意一行来获取帮助：

```
> ::help bp
> ::bp dcmd
```

为驱动程序开发者提供的 kmdb 宏

kmdb(1M) 调试器支持可用于显示内核数据结构的宏。可以使用 $M 来显示 kmdb 宏。宏的使用形式为：

```
[ address ] $<macroname
```

注 – 这些宏所显示的信息以及显示信息所用的格式都不构成接口。因此，该信息和格式可以随时更改。

下表中的 kmdb 宏对于设备驱动程序的开发者特别有用。为方便起见，给出了传统的宏名称（如果适用）。

表 23–1 kmdb 宏

Dcmd	传统宏	说明
::devinfo	devinfo	列显设备节点的摘要
	devinfo_brief	
	devinfo.prop	
::walk devinfo_parents	devinfo.parent	遍历设备节点的祖先
::walk devinfo_sibling	devinfo.sibling	遍历设备节点的同级节点
::minornodes	devinfo.minor	列显与给定设备节点对应的次要节点
::major2name		列显绑定到给定设备节点的设备的名称。
::devbindings		列显绑定到给定设备节点或主设备号的设备节点。

::devinfo dcmd 显示节点状态，其值为以下所列之一：

DS_ATTACHED 驱动程序的 attach(9E) 例程成功返回。

DS_BOUND 节点已绑定到驱动程序，但尚未调用驱动程序的 probe(9E) 例程。

DS_INITIALIZED 父结点已为驱动程序指定总线地址。特定于实现的初始化已完成。此时尚未调用驱动程序的 probe(9E) 例程。

DS_LINKED 设备节点已链接至内核的设备树中，但系统尚未找到用于此节点的驱动程序。

DS_PROBED 驱动程序的 probe(9E) 例程成功返回。

DS_READY 设备已完全配置。

使用 mdb 模块调试器

mdb(1) 模块调试器可以应用于以下文件类型：

- 实时操作系统组件
- 操作系统故障转储
- 用户进程
- 用户进程核心转储
- 对象文件

mdb 调试器可为分析内核问题提供复杂的调试支持。本节概述 mdb 功能。有关 mdb 的完整讨论，请参阅《Oracle Solaris Modular Debugger Guide》。

尽管 mdb 可用来改变实时内核状态，但 mdb 缺少 kmdb 提供的内核执行控制。因此，kmdb 是进行运行时调试的首选调试器，而 mdb 调试器更多用于静态情况。

注 – mdb 的提示符为 >。

模块调试器入门

mdb 为实现调试器模块提供了大量编程 API，从而使驱动程序开发者可以实现定制调试支持。mdb 调试器还提供了许多可用功能，如命令行编辑、命令历史记录、输出页面调度程序和联机帮助。

注 – 不应再使用 adb 宏。该功能已被 mdb 中的 dcmd 替代。

---

mdb 调试器提供了一组丰富的模块和 dcmd。借助这些工具，可以调试 Oracle Solaris 内核、任何关联的模块以及设备驱动程序。通过这些功能可以执行一些任务，如：

- 阐明复杂的调试查询
- 查找特定线程分配的所有内存
- 列显内核 STREAM 的直观图
- 确定特定地址所引用的结构类型
- 在内核中查找已泄漏的内存块
- 分析内存以查找栈跟踪
- 将 dcmd 组装到用于创建定制操作且名为 *dmod* 的模块中

首先切换到崩溃目录，键入 mdb 并指定系统故障转储，如以下示例所示。

示例 23–7 针对故障转储调用 mdb

```
% cd /var/crash/testsystem
% ls
bounds     unix.0    vmcore.0
% mdb unix.0 vmcore.0
Loading modules: [ unix krtld genunix ufs_log ip usba s1394 cpc nfs ]
> ::status
debugging crash dump vmcore.0 (64-bit) from testsystem
operating system: 5.10 Generic (sun4u)
panic message: zero
dump content: kernel pages only
```

当 mdb 以 > 提示符进行响应时，便可运行命令。

要检查实时系统中正在运行的内核，请按如下所示从系统提示符处运行 mdb。

示例 23–8 针对正在运行的内核调用 mdb

```
# mdb -k
Loading modules: [ unix krtld genunix ufs_log ip usba s1394 ptm cpc ipc nfs ]
> ::status
debugging live kernel (64-bit) on testsystem
operating system: 5.10 Generic (sun4u)
```

使用 kmdb 和 mdb 执行的有用调试任务

本节提供了有用的调试任务示例。除非特别说明，否则本节中的任务均可使用 mdb 或 kmdb 来执行。本节假定您已了解 kmdb 和 mdb 的基本使用知识。请注意，此处提供的信息取决于所使用系统的类型。这些示例是使用运行 64 位内核的 Sun Blade 100 工作站生成的。

注意 – 由于修改内核结构中的数据会导致无法恢复的数据损毁，因此务必要格外谨慎。请勿修改或依赖于不属于 Oracle Solaris DDI 结构中的数据。有关属于 Oracle Solaris DDI 的结构的信息，请参见 Intro(9S) 手册页。

使用 kmdb 查找系统寄存器

kmdb 调试器可按组或单独显示计算机寄存器。要按组显示所有寄存器，请按以下示例所示使用 $r。

示例 23–9 使用 kmdb 读取 SPARC 处理器中的所有寄存器

```
[0]: $r

g0    0                                 l0      0
g1    100130a4      debug_enter         l1      edd00028
g2    10411c00      tsbmiss_area+0xe00  l2      10449c90
g3    10442000      ti_statetbl+0x1ba   l3      1b
g4    3000061a004                       l4      10474400     ecc_syndrome_tab+0x80
g5    0                                 l5      3b9aca00
g6    0                                 l6      0
g7    2a10001fd40                       l7      0
o0    0                                 i0      0
o1    c                                 i1      10449e50
o2    20                                i2      0
o3    300006b2d08                       i3      10
o4    0                                 i4      0
o5    0                                 i5      b0
sp    2a10001b451                       fp      2a10001b521
o7    1001311c      debug_enter+0x78    i7      1034bb24     zsa_xsint+0x2c4
y     0
tstate: 1604  (ccr=0x0, asi=0x0, pstate=0x16, cwp=0x4)
pstate: ag:0 ie:1 priv:1 am:0 pef:1 mm:0 tle:0 cle:0 mg:0 ig:0
winreg: cur:4 other:0 clean:7 cansave:1 canrest:5 wstate:14
tba   0x10000000
pc    edd000d8 edd000d8:        ta      %icc,%g0 + 125
```

示例 23–9 使用 kmdb 读取 SPARC 处理器中的所有寄存器 （续）

```
npc    edd000dc edd000dc:        nop
```

调试器会将每个寄存器值导出到与寄存器同名的一个变量中。如果读取该变量，则返回对应寄存器的当前值。如果写入该变量，则会更改关联的计算机寄存器值。以下示例将一台 x86 计算机上 %o0 寄存器的值由 0 更改为 1。

示例 23–10 使用 kmdb 读/写 x86 计算机中的寄存器

```
[0]> &<eax=K
        c1e6e0f0
[0]> 0>eax
[0]> &<eax=K
        0
[0]>  c1e6e0f0>eax
```

如果需要检查不同处理器的寄存器，则可使用 ::cpuregs dcmd。要检查的处理器的 ID 可以作为 dcmd 的地址或 -c 选项的值来提供，如以下示例所示。

示例 23–11 检查不同处理器的寄存器

```
[0]> 0::cpuregs
   %cs = 0x0158            %eax = 0xc1e6e0f0 kmdbmod`kaif_dvec
   %ds = 0x0160            %ebx = 0x00000000
```

以下示例从 SPARC 计算机上的处理器 0 切换到处理器 3。检查了寄存器 %g3，然后将其清除。为确认新值，再次读取 %g3。

示例 23–12 从指定的处理器中检索单个寄存器值

```
[0]> 3::switch
[3]> <g3=K
        24
[3]> 0>g3
[3]> <g3
        0
```

检测内核内存泄漏

::findleaks dcmd 可对内核故障转储中的内存泄漏提供强大、有效的检测。必须启用一整套内核内存调试功能，::findleaks 才会有效。有关更多信息，请参见第 470 页中的 “设置 kmem_flags 调试标志”。在驱动程序开发和测试期间运行 ::findleaks，以检测泄漏内存从而浪费内核资源的代码。有关 《Oracle Solaris Modular Debugger Guide》中的第 9 章 “Debugging With the Kernel Memory Allocator”Debugging With the Kernel Memory Allocator。

注 – 泄漏内核内存的代码会使系统容易受到拒绝服务攻击。

使用 mdb 编写调试器命令

mdb 调试器提供了一个功能强大的 API，用于实现为调试驱动程序而定制的调试器功能。《Oracle Solaris 模块调试器指南》详细介绍了该编程 API。

SUNWmdbdm 软件包将 mdb 源代码示例安装在目录 /usr/demo/mdb 中。可以使用 mdb 来自动完成冗长的调试日常事务，或帮助验证驱动程序是否正常工作。还可以将 mdb 调试模块与驱动程序产品一起打包。通过打包，服务人员可在客户站点处使用这些功能。

获取内核数据结构信息

Oracle Solaris 内核在可用 kmdb 或 mdb 检查的结构中提供数据类型信息。

注 – kmdb 和 mdb dcmd 只能用于包含设计用于 mdb 的压缩符号调试信息的对象。此信息当前只能用于某些 Oracle Solaris 内核模块。必须安装 SUNWzlib 软件包，才能处理符号调试信息。

以下示例说明如何显示 scsi_pkt 结构中的数据。

示例 23–13 使用调试器显示内核数据结构

```
> 7079ceb0::print -t 'struct scsi_pkt'
{
    opaque_t pkt_ha_private = 0x7079ce20
    struct scsi_address pkt_address = {
        struct scsi_hba_tran *a_hba_tran = 0x70175e68
        ushort_t a_target = 0x6
        uchar_t a_lun = 0
        uchar_t a_sublun = 0
    }
    opaque_t pkt_private = 0x708db4d0
    int (*)() *pkt_comp = sd_intr
    uint_t pkt_flags = 0
    int pkt_time = 0x78
    uchar_t *pkt_scbp = 0x7079ce74
    uchar_t *pkt_cdbp = 0x7079ce64
    ssize_t pkt_resid = 0
    uint_t pkt_state = 0x37
    uint_t pkt_statistics = 0
    uchar_t pkt_reason = 0
}
```

数据结构的大小在调试中很有用。使用 ::sizeof dcmd 可获取结构的大小，如以下示例所示。

示例 23–14 显示内核数据结构的大小

```
> ::sizeof struct scsi_pkt
sizeof (struct scsi_pkt) = 0x58
```

结构中特定成员的地址在调试中也很有用。有几种方法可用来确定成员的地址。

使用 `::offsetof` dcmd 可以获取结构中给定成员的偏移，如以下示例所示。

示例 23–15 显示内核数据结构的偏移

```
> ::offsetof struct scsi_pkt pkt_state
offsetof (struct pkt_state) = 0x48
```

使用带 `-a` 选项的 `::print` dcmd 可以显示结构中所有成员的地址，如以下示例所示。

示例 23–16 显示内核数据结构的相对地址

```
> ::print -a struct scsi_pkt
{
    0 pkt_ha_private
    8 pkt_address {
    ...
    }
    18 pkt_private
    ...
}
```

如果结合使用 `::print` 和 `-a` 选项来指定地址，则会显示每个成员的绝对地址。

示例 23–17 显示内核数据结构的绝对地址

```
> 10000000::print -a struct scsi_pkt
{
    10000000 pkt_ha_private
    10000008 pkt_address {
    ...
    }
    10000018 pkt_private
    ...
}
```

使用 `::print`、`::sizeof` 和 `::offsetof` dcmd，可在驱动程序与 Oracle Solaris 内核交互时调试问题。

注意 – 通过此功能可访问**原始**内核数据结构。您可以检查任何结构，无论该结构是否显示为 DDI 的一部分。因此，应避免依赖于未显式构成 DDI 的任何数据结构。

注 – 这些 dcmd 只能用于包含设计用于 mdb 的压缩符号调试信息的对象。符号调试信息当前只能用于某些 Oracle Solaris 内核模块。必须安装 SUNWzlib（32 位）或 SUNWzlibx（64 位）解压缩软件，才能处理符号调试信息。无论是否包含 SUNWzlib 或 SUNWzlibx 软件包，kmdb 调试器均可处理符号类型数据。

获取设备树信息

mdb 调试器提供了用于显示内核设备树的 ::prtconf dcmd。::prtconf dcmd 的输出与 prtconf(1M) 命令的输出相似。

示例 23–18 使用 ::prtconf Dcmd

```
> ::prtconf
300015d3e08      SUNW,Sun-Blade-100
    300015d3c28      packages (driver not attached)
        300015d3868      SUNW,builtin-drivers (driver not attached)
        300015d3688      deblocker (driver not attached)
        300015d34a8      disk-label (driver not attached)
        300015d32c8      terminal-emulator (driver not attached)
        300015d30e8      obp-tftp (driver not attached)
        300015d2f08      dropins (driver not attached)
        300015d2d28      kbd-translator (driver not attached)
        300015d2b48      ufs-file-system (driver not attached)
    300015d3a48      chosen (driver not attached)
    300015d2968      openprom (driver not attached)
```

可以使用宏（如 ::devinfo dcmd）来显示节点，如以下示例所示。

示例 23–19 显示单个节点的设备信息

```
> 300015d3e08::devinfo
300015d3e08      SUNW,Sun-Blade-100
        System properties at 0x300015abdc0:
            name='relative-addressing' type=int items=1
                value=00000001
            name='MMU_PAGEOFFSET' type=int items=1
                value=00001fff
            name='MMU_PAGESIZE' type=int items=1
                value=00002000
            name='PAGESIZE' type=int items=1
                value=00002000
        Driver properties at 0x300015abe00:
            name='pm-hardware-state' type=string items=1
                value='no-suspend-resume'
```

使用 ::prtconf 可以查看驱动程序在设备树中连接的位置，以及显示设备属性。还可以为 ::prtconf 指定详细 (-v) 标志，以显示每个设备节点的属性，如下所示。

示例 23–20 在详细模式下使用 ::prtconf Dcmd

```
> ::prtconf -v
DEVINFO          NAME
300015d3e08      SUNW,Sun-Blade-100
        System properties at 0x300015abdc0:
            name='relative-addressing' type=int items=1
                value=00000001
            name='MMU_PAGEOFFSET' type=int items=1
                value=00001fff
            name='MMU_PAGESIZE' type=int items=1
                value=00002000
            name='PAGESIZE' type=int items=1
                value=00002000
        Driver properties at 0x300015abe00:
            name='pm-hardware-state' type=string items=1
                value='no-suspend-resume'
        ...
        300015ce798      pci10b9,5229, instance #0
                Driver properties at 0x300015ab980:
                    name='target2-dcd-options' type=any items=4
                        value=00.00.00.a4
                    name='target1-dcd-options' type=any items=4
                        value=00.00.00.a2
                    name='target0-dcd-options' type=any items=4
                        value=00.00.00.a4
```

另一种查找驱动程序实例的方法是使用 ::devbindings dcmd。在给定驱动程序名称的情况下，该命令会显示指定驱动程序的所有实例的列表，如以下示例所示。

示例 23–21 使用 ::devbindings Dcmd 查找驱动程序实例

```
> ::devbindings dad
300015ce3d8      ide-disk (driver not attached)
300015c9a60      dad, instance #0
        System properties at 0x300015ab400:
            name='lun' type=int items=1
                value=00000000
            name='target' type=int items=1
                value=00000000
            name='class_prop' type=string items=1
                value='ata'
            name='type' type=string items=1
                value='ata'
            name='class' type=string items=1
                value='dada'
...
300015c9880      dad, instance #1
        System properties at 0x300015ab080:
            name='lun' type=int items=1
                value=00000000
            name='target' type=int items=1
                value=00000002
            name='class_prop' type=string items=1
                value='ata'
            name='type' type=string items=1
```

示例 23–21 使用 ::devbindings Dcmd 查找驱动程序实例 （续）

```
            value='ata'
        name='class' type=string items=1
            value='dada'
```

检索驱动程序软状态信息

调试驱动程序的常见问题是检索特定驱动程序实例的**软状态**。软状态使用 ddi_soft_state_zalloc(9F) 例程来分配。驱动程序可以通过 ddi_get_soft_state(9F) 获取软状态。**软状态指针**的名称是 ddi_soft_state_init (9F) 的第一个参数。根据名称，可以使用 mdb 通过 ::softstate dcmd 检索特定驱动程序实例的软状态：

```
> *bst_state::softstate 0x3
702b7578
```

在此示例中，::softstate 用来获取 bst 示例驱动程序的实例 3 的软状态。此指针引用由驱动程序使用的 bst_soft 结构，以便跟踪该实例的状态。

修改内核变量

可以使用 kmdb 和 mdb 来修改内核变量或其他内核状态。使用 mdb 修改内核状态时要格外谨慎，因为 mdb 在进行修改前不会停止内核。使用 kmdb 可以原子方式进行成组修改，因为 kmdb 会在允许用户访问之前停止内核。mdb 调试器只能进行单个原子修改。

务必要使用正确的格式指示符来进行修改。格式可以为：

- w－将每个表达式值的最低 2 个字节写入从点所指定的位置开始的目标位置
- W－将每个表达式值的最低 4 个字节写入从点所指定的位置开始的目标位置
- Z－将每个表达式值的全部 8 个字节写入从点所指定的位置开始的目标位置

使用 ::sizeof dcmd 可以确定要修改的变量的大小。

以下示例使用值 0x80000000 覆盖 moddebug 的值。

示例 23–22 使用调试器修改内核变量

```
> moddebug/W 0x80000000
    moddebug:       0 = 0x80000000
```

调优驱动程序

Oracle Solaris OS 提供了内核统计信息结构，以便针对驱动程序实现计数器。使用 DTrace 功能可以实时分析性能。本节介绍有关设备性能的以下主题：

- 第 484 页中的 “内核统计信息” – Oracle Solaris OS 提供一组数据结构和函数，用于捕获内核中的性能统计信息。内核统计信息（名为 *kstat*）可使驱动程序在系统运行时导出连续统计信息。可通过使用 kstat 函数以编程方式处理 kstat 数据。
- 第 489 页中的 “用于动态检测过程的 DTrace” – 使用 DTrace 可向驱动程序中动态添加检测过程，这样您便可以执行诸如分析系统和度量性能等任务。DTrace 利用了预定义的 kstat 结构。

内核统计信息

为了协助进行性能调优，Oracle Solaris 内核提供了 `kstat(3KSTAT)` 功能。kstat 功能提供了一套函数和数据结构，以供设备驱动程序和其他内核模块导出特定于模块的内核统计信息。

kstat 是用于记录设备使用情况的可计量方面的数据结构。kstat 存储为以 null 终止的链接列表。每个 kstat 都有一个通用的头区和一个特定于类型的数据区。头区由 `kstat_t` 结构定义。

内核统计信息结构成员

kstat 结构的成员包括：

`ks_class[KSTAT_STRLEN]`	kstat 类型分类为 `bus`、`controller`、`device_error`、`disk`、`hat`、`kmem_cache`、`kstat`、`misc`、`net`、`nfs`、`pages`、`partition`、`rps`、`ufs`、`vm` 或 `vmem`。
`ks_crtime`	kstat 的创建时间。`ks_crtime` 通常用于计算各个计数器的速率。
`ks_data`	指向 kstat 的数据区。
`ks_data_size`	数据区大小总额（以字节为单位）。
`ks_instance`	创建此 kstat 的内核模块的实例。`ks_instance` 与 `ks_module` 和 `ks_name` 结合使用，以便为 kstat 指定唯一且有意义的名称。
`ks_kid`	kstat 的唯一 ID。
`ks_module[KSTAT_STRLEN]`	标识创建此 kstat 的内核模块。`ks_module` 与 `ks_instance` 和 `ks_name` 结合使用，以便为 kstat 指定唯一且有意义的名称。`KSTAT_STRLEN` 可设置 `ks_module` 的最大长度。
`ks_name[KSTAT_STRLEN]`	为 kstat 指定的名称，与 `ks_module` 和 `ks_instance` 结合使用。`KSTAT_STRLEN` 可设置 `ks_module` 的最大长度。

ks_ndata	为以下可支持多个记录的 kstat 类型指示数据记录的个数：KSTAT_TYPE_RAW、KSTAT_TYPE_NAMED 和 KSTAT_TYPE_TIMER
ks_next	指向链表中的下一个 kstat。
ks_resv	保留的字段。
ks_snaptime	上一数据快照的时间戳，在计算速率时很有用。
ks_type	数据类型，对于二进制数据可为 KSTAT_TYPE_RAW，对于名称/值对可为 KSTAT_TYPE_NAMED，对于中断统计信息可为 KSTAT_TYPE_INTR，对于 I/O 统计信息可为 KSTAT_TYPE_IO，对于事件计时器可为 KSTAT_TYPE_TIMER。

内核统计信息结构

不同种类的 kstat 的结构如下：

kstat(9S)	由设备驱动程序导出的每条内核统计信息 (kstat) 都由头区和数据区构成。kstat(9S) 结构是统计信息的头部分。
kstat_intr(9S)	中断 kstat 的结构。中断类型包括： ■ 硬中断－源自硬件设备自身 ■ 软中断－因系统使用某些系统中断源而引起 ■ 监视程序中断－由定期计时器调用引起 ■ 虚假中断－输入了中断入口点，但没有需要提供服务的中断 ■ 多个服务－在从任何其他类型返回之前检测到中断并提供了服务 驱动程序通常只报告从其处理程序中声明的硬中断和软中断，但度量虚假类中断对自动向量化的设备很有用，以便查找特定系统配置中的任何中断延迟信息。具有多个相同类型中断的设备应使用多个结构。
kstat_io(9S)	I/O kstat 的结构。
kstat_named(9S)	命名的 kstat 的结构。命名的 kstat 是名称-值对数组。这些对位于 kstat_named 结构中。

内核统计信息函数

用于使用 kstat 的函数包括：

kstat_create(9F)
分配和初始化 kstat(9S) 结构。

kstat_delete(9F)
从系统中移除 kstat。

kstat_install(9F)
向系统中添加完全初始化的 kstat。

kstat_named_init(9F)、kstat_named_setstr(9F)
初始化已命名的 kstat。kstat_named_setstr() 将 str（一个字符串）与已命名的 kstat 指针相关联。

kstat_queue(9F)
许多 I/O 子系统都至少有两个基本的事务队列要管理。一个队列用于已接受但尚未开始处理的事务。另一个队列用于正在进行处理但尚未处理完的事务。因此，保留了两个累积时间统计量：**等待时间**和**运行时间**。等待时间是提供服务之前的时间。运行时间是提供服务期间的时间。kstat_queue() 函数系列可根据驱动程序等待队列和运行队列之间的转换来管理这些时间：

- kstat_runq_back_to_waitq(9F)
- kstat_runq_enter(9F)
- kstat_runq_exit(9F)
- kstat_waitq_enter(9F)
- kstat_waitq_exit(9F)
- kstat_waitq_to_runq(9F)

Oracle Solaris 以太网驱动程序的内核统计信息

下表中介绍的 kstat 接口是从驱动程序中获取以太网物理层统计信息的有效方法。以太网驱动程序应导出这些统计信息，以指导用户更好地诊断和修复以太网物理层问题。除 link_up 之外，所有统计信息在未提供时的缺省值均为 0。应将 link_up 统计信息的值假定为 1。

以下示例给出了所有共享的链路设置。在这种情况下，可使用 mii 来过滤统计信息。

```
kstat ce:0:mii:link_*
```

表 23–2 以太网 MII/GMII 物理层接口内核统计信息

Kstat 变量	类型	说明
xcvr_addr	KSTAT_DATA_UINT32	提供当前正在使用的收发器的 MII 地址。 ■ (0) - (31) 用于给定以太网设备的物理层设备的 MII 地址。 ■ 在没有可从外部访问的 MII 接口，因此 MII 地址不明确或不相关的情况下，使用 (-1)。
xcvr_id	KSTAT_DATA_UINT32	提供当前正在使用的收发器的特定供应商 ID 或设备 ID。

表 23–2 以太网 MII/GMII 物理层接口内核统计信息 （续）

Kstat 变量	类型	说明
xcvr_inuse	KSTAT_DATA_UINT32	指示当前正在使用的收发器的类型。IEEE aPhytType 枚举以下集合： ■ (0) 其他未定义 ■ (1) 不存在 MII 接口，但未连接任何收发器 ■ (2) 10 Mb/s Clause 7 10 Mb/s Manchester ■ (3) 100BASE-T4 Clause 23 100 Mb/s 8B/6T ■ (4) 100BASE-X Clause 24 100 Mb/s 4B/5B ■ (5) 100BASE-T2 Clause 32 100 Mb/s PAM5X5 ■ (6) 1000BASE-X Clause 36 1000 Mb/s 8B/10B ■ (7) 1000BASE-T Clause 40 1000 Mb/s 4D-PAM5 此集合比 ifMauType 指定的集合小，后者定义为包括上述所有类型及其半双工/全双工选件。由于 cap_* 统计信息可提供此信息，因此可从 xcvr_inuse 和 cap_* 的组合中派生缺少的定义，以提供 ifMayType 的所有组合。
cap_1000fdx	KSTAT_DATA_CHAR	指示设备支持 1 Gb/s 的全双工传输。
cap_1000hdx	KSTAT_DATA_CHAR	指示设备支持 1 Gb/s 的半双工传输。
cap_100fdx	KSTAT_DATA_CHAR	指示设备支持 100 Mb/s 的全双工传输。
cap_100hdx	KSTAT_DATA_CHAR	指示设备支持 100 Mb/s 的半双工传输。
cap_10fdx	KSTAT_DATA_CHAR	指示设备支持 10 Mb/s 的全双工传输。
cap_10hdx	KSTAT_DATA_CHAR	指示设备支持 10 Mb/s 的半双工传输。
cap_asmpause	KSTAT_DATA_CHAR	指示设备支持非对称暂停以太网流量控制。
cap_pause	KSTAT_DATA_CHAR	指示当 cap_pause 设置为 1，cap_asmpause 设置为 0 时，设备支持对称暂停以太网流量控制。当 cap_asmpause 设置为 1 时，cap_pause 具有以下含义： ■ cap_pause = 0 基于接收拥塞传送暂停。 ■ cap_pause = 1 接收暂停并减慢传送，以避免拥塞。
cap_rem_fault	KSTAT_DATA_CHAR	指示设备支持远程故障指示。
cap_autoneg	KSTAT_DATA_CHAR	指示设备支持自动协商。
adv_cap_1000fdx	KSTAT_DATA_CHAR	指示设备正在通告支持 1 Gb/s 的全双工传输。
adv_cap_1000hdx	KSTAT_DATA_CHAR	指示设备正在通告支持 1 Gb/s 的半双工传输。
adv_cap_100fdx	KSTAT_DATA_CHAR	指示设备正在通告支持 100 Mb/s 的全双工传输。
adv_cap_100hdx	KSTAT_DATA_CHAR	指示设备正在通告支持 100 Mb/s 的半双工传输。
adv_cap_10fdx	KSTAT_DATA_CHAR	指示设备正在通告支持 10 Mb/s 的全双工传输。

表 23–2 以太网 MII/GMII 物理层接口内核统计信息 （续）

Kstat 变量	类型	说明
adv_cap_10hdx	KSTAT_DATA_CHAR	指示设备正在通告支持 10 Mb/s 的半双工传输。
adv_cap_asmpause	KSTAT_DATA_CHAR	指示设备正在通告支持非对称暂停以太网流量控制。
adv_cap_pause	KSTAT_DATA_CHAR	指示当 adv_cap_pause 设置为 1，adv_cap_asmpause 设置为 0 时，设备正在通告支持对称暂停以太网流量控制。当 adv_cap_asmpause 设置为 1 时，adv_cap_pause 具有以下含义： ■ adv_cap_pause = 0 基于接收拥塞传送暂停。 ■ adv_cap_pause = 1 接收暂停并减慢传送，以避免拥塞。
adv_rem_fault	KSTAT_DATA_CHAR	指示设备遇到故障，设备将把该故障转发给链路合作伙伴。
adv_cap_autoneg	KSTAT_DATA_CHAR	指示设备正在通告支持自动协商。
lp_cap_1000fdx	KSTAT_DATA_CHAR	指示链路合作伙伴设备支持 1 Gb/s 的全双工传输。
lp_cap_1000hdx	KSTAT_DATA_CHAR	指示链路合作伙伴设备支持 1 Gb/s 的半双工传输。
lp_cap_100fdx	KSTAT_DATA_CHAR	指示链路合作伙伴设备支持 100 Mb/s 的全双工传输。
lp_cap_100hdx	KSTAT_DATA_CHAR	指示链路合作伙伴设备支持 100 Mb/s 的半双工传输。
lp_cap_10fdx	KSTAT_DATA_CHAR	指示链路合作伙伴设备支持 10 Mb/s 的全双工传输。
lp_cap_10hdx	KSTAT_DATA_CHAR	指示链路合作伙伴设备支持 10 Mb/s 的半双工传输。
lp_cap_asmpause	KSTAT_DATA_CHAR	指示链路合作伙伴设备支持非对称暂停以太网流量控制。
lp_cap_pause	KSTAT_DATA_CHAR	指示当 lp_cap_pause 设置为 1，lp_cap_asmpause 设置为 0 时，链路合作伙伴设备支持对称暂停以太网流量控制。当 lp_cap_asmpause 设置为 1 时，lp_cap_pause 具有以下含义： ■ lp_cap_pause = 0 链路合作伙伴将基于接收拥塞传送暂停。 ■ lp_cap_pause = 1 链路合作伙伴将接收暂停并减慢传送，以避免拥塞。
lp_rem_fault	KSTAT_DATA_CHAR	指示链路合作伙伴遇到链路故障。
lp_cap_autoneg	KSTAT_DATA_CHAR	指示链路合作伙伴设备支持自动协商。
link_asmpause	KSTAT_DATA_CHAR	指示链路正采用非对称暂停以太网流量控制来运行。
link_pause	KSTAT_DATA_CHAR	指示暂停功能的精度。指示当 link_pause 设置为 1，link_asmpause 设置为 0 时，链路正采用对称暂停以太网流量控制来运行。当 link_asmpause 设置为 1 且相对于链路的本地视图时，link_pause 具有以下含义： ■ link_pause = 0 此站将基于接收拥塞来传送暂停。 ■ link_pause = 1 此站将接收暂停并减慢传送，以避免拥塞。

表 23–2 以太网 MII/GMII 物理层接口内核统计信息 （续）

Kstat 变量	类型	说明
link_duplex	KSTAT_DATA_CHAR	指示链路双工模式。 ■ link_duplex = 0 链路关闭，且双工模式未知。 ■ link_duplex = 1 链路打开，且处于半双工模式。 ■ link_duplex = 2 链路打开，且处于全双工模式。
link_up	KSTAT_DATA_CHAR	指示链路是打开还是关闭。 ■ link_up = 0 链路关闭。 ■ link_up = 1 链路打开。

用于动态检测过程的 DTrace

DTrace 是一种全面的动态跟踪工具，用于检查用户程序和操作系统自身的行为。通过 DTrace，可以收集环境中处于关键位置（称为**探测器**）的数据。通过 DTrace 可以记录栈跟踪、时间戳、函数的参数或探测器触发频率计数等数据。由于 DTrace 允许动态插入探测器，因此无需重新编译代码。有关 DTrace 的更多信息，请参见《Oracle Solaris 11.1 Dynamic Tracing Guide》 。

◆ ◆ ◆ 第 24 章

24 推荐的编码方法

本章介绍如何编写强健的驱动程序。根据本章中所讨论的原则编写的驱动程序更易于进行调试。建议的做法还可在出现硬件和软件故障时为系统提供保护。

本章介绍有关以下主题的信息：

调试准备方法

由于以下原因，驱动程序代码比用户程序更难调试：

- 驱动程序直接与硬件进行交互
- 驱动程序在运行时不能受到操作系统的保护，而用户进程可以

请确保您的驱动程序可以支持调试。此支持便于进行维护工作和未来的开发工作。

使用唯一前缀来避免内核符号冲突

每个函数、数据元素和驱动程序预处理程序定义的名称必须对每个驱动程序都唯一。

驱动程序模块将链接到内核。对特定驱动程序唯一的每个符号名称不得与其他内核符号冲突。为避免这种冲突，特定驱动程序的每个函数和数据元素的名称必须带有该驱动程序共有的前缀。该前缀必须足以让每个驱动程序符号的名称保持唯一。通常，该前缀是驱动程序的名称，或者是驱动程序名称的缩写。例如，`xx_open()` 是驱动程序 `xx` 的 `open`(9E) 例程的名称。

在构建驱动程序时，驱动程序一定包含许多系统头文件。这些头文件中的全局可见名称无法预测。为避免与这些名称产生冲突，必须使用一个标识前缀为每个驱动程序预处理程序定义指定唯一的名称。

在进行错误诊断时，还可以借助唯一的驱动程序符号前缀来解读系统日志和故障消息。您看到的将是与 xx_attach() 有关的错误消息，而不是与二义性 attach() 函数有关的错误。

使用 cmn_err() 记录驱动程序活动

使用 cmn_err(9F) 函数可以将来自设备驱动程序的消息列显到系统日志中。用于内核模块的 cmn_err (9F) 函数与用于应用程序的 printf(3C) 函数类似。cmn_err(9F) 函数可提供其他格式字符，如用于列显设备寄存器位的 %b 格式。cmn_err(9F) 函数可以将消息写入系统日志中。使用 tail(1) 命令可以监视 /var/adm/messages 中的这些消息。

```
% tail -f /var/adm/messages
```

使用 ASSERT() 捕捉无效假设

断言是活动文档一种极有价值的形式。ASSERT(9F) 的语法如下所示：

```
void ASSERT(EXPRESSION)
```

如果预期为 true 的条件实际为 false，则 ASSERT() 宏会停止执行内核。ASSERT() 为程序员提供了对某段代码所做的假设进行验证的方法。

仅当定义了 DEBUG 编译符号时，才会定义 ASSERT() 宏。如果未定义 DEBUG，ASSERT() 宏将无效。

以下示例断言将测试特定指针值不是 NULL 的假设：

```
ASSERT(ptr != NULL);
```

如果驱动程序已使用 DEBUG 进行编译，并且执行至此时 ptr 的值为 NULL，则在控制台上会列显以下故障消息：

```
panic: assertion failed: ptr != NULL, file: driver.c, line: 56
```

注 – 由于 ASSERT(9F) 使用 DEBUG 编译符号，因此任何条件调试代码也应使用 DEBUG。

使用 mutex_owned() 验证和记录锁定要求

mutex_owned(9F) 的语法如下：

```
int mutex_owned(kmutex_t *mp);
```

驱动程序开发过程中的一项重要工作涉及正确处理多个线程。获得了互斥锁时，应始终使用注释。在**未**获得明确需要的互斥锁时，注释将更有用。要确定线程是否持有互斥锁，请在 ASSERT(9F) 中使用 mutex_owned()：

```
void helper(void)
{
    /* this routine should always be called with xsp's mutex held */
    ASSERT(mutex_owned(&xsp->mu));
    /* ... */
}
```

注 – `mutex_owned()` 只在 `ASSERT()` 宏内有效。应该使用 `mutex_owned()` 控制驱动程序的行为。

使用条件编译在开销较大的调试功能之间切换

可以使用预处理程序符号（例如 `DEBUG`）或全局变量，借助条件编译将用于调试的代码插入驱动程序中。使用条件编译，可在生产驱动程序中删除不必要的代码。使用变量可以在运行时设置调试输出量。在运行时使用 `ioctl` 或调试器设置调试级别可以指定输出。通常，可以组合使用这两种方法。

以下示例依赖编译器来删除无法访问的代码（在本例中是跟在始终为 false 的零测试之后的代码）。此示例还提供了可在 `/etc/system` 中设置或由调试器修补的局部变量。

```
#ifdef DEBUG
/* comments on values of xxdebug and what they do */
static int xxdebug;
#define dcmn_err if (xxdebug) cmn_err
#else
#define dcmn_err if (0) cmn_err
#endif
/* ... */
    dcmn_err(CE_NOTE, "Error!\n");
```

此方法可处理 `cmn_err(9F)` 具有可变数量参数的情况。另一种方法依赖于宏只有一个参数的情况，即 `cmn_err(9F)` 的用括号括起的参数列表。宏可以删除此参数。此宏还可在未定义 `DEBUG` 的情况下，通过将宏扩展为不包含任何内容来消除对优化程序的依赖性。

```
#ifdef DEBUG
/* comments on values of xxdebug and what they do */
static int xxdebug;
#define dcmn_err(X) if (xxdebug) cmn_err X
#else
#define dcmn_err(X) /* nothing */
#endif
/* ... */
/* Note:double parentheses are required when using dcmn_err. */
    dcmn_err((CE_NOTE, "Error!"));
```

可以采用多种方法扩展此技术。一种方法是根据 `xxdebug` 的值指定来自 `cmn_err(9F)` 的不同消息。但在此类情况下，必须注意不要用大量的调试信息使代码变得晦涩难懂。

另一种常见方案是编写 `xxlog()` 函数，以便使用 `vsprintf(9F)` 或 `vcmn_err(9F)` 来处理变量参数列表。

将变量声明为可变变量

volatile 是声明任何引用设备寄存器的变量时必须应用的一个关键字。如果不使用 volatile，编译时优化程序可能会意外删除重要访问。省略使用 volatile 可能会生成很难跟踪的错误。

要防止出现难懂的错误，必须正确使用 volatile。volatile 关键字指示编译器对已声明的对象使用精确语义，特别指示不能删除或重新排序对对象的访问。设备驱动程序必须使用 volatile 限定符的两个实例为：

- 当数据引用外部硬件设备寄存器（即除了存储功能之外还具有负面影响的内存）时。但请注意，如果使用 DDI 数据访问函数访问设备寄存器，则无需使用 volatile。
- 当数据引用的全局内存可由多个线程访问、不受锁定保护并且依赖于内存访问的序列时。与使用锁定相比，使用 volatile 使用的资源较少。

以下示例使用 volatile。忙标志用于防止线程在设备忙时继续执行，该标志不受锁定保护：

```
while (busy) {
    /* do something else */
}
```

测试线程将在另一个线程关闭 busy 标志时继续执行：

```
busy = 0;
```

由于 busy 会在测试线程中被频繁地访问，因此编译器可能通过将 busy 的值放在寄存器中来优化测试，并测试寄存器的内容，而无需在每次测试前都读取内存中的 busy 值。测试线程将永远无法看到 busy 的更改，其他线程将只更改内存中的 busy 值，从而导致死锁。将 busy 标志声明为 volatile 会强制在每次测试前读取其值。

注 – busy 标志的一种替代方法是使用条件变量。请参见第 68 页中的 “线程同步中的条件变量”。

使用 volatile 限定符时，请避免意外省略的风险。例如，以下代码

```
struct device_reg {
    volatile uint8_t csr;
    volatile uint8_t data;
};
struct device_reg *regp;
```

比下一个示例更可取：

```
struct device_reg {
    uint8_t csr;
```

```
    uint8_t data;
};
volatile struct device_reg *regp;
```

尽管这两个示例在功能上等效，但第二个示例要求编写人员确保类型 `struct device_reg` 的每个声明中都使用 `volatile`。第一个示例将导致所有声明中都将数据视为可变数据，因此首选该示例。如上所述，如果使用 DDI 数据访问函数访问设备寄存器，就不必将变量限定为 `volatile` 了。

如果您使用的带有 C++ 5.11 的 Oracle Solaris Studio 12.2，请使用 **`-xvector=no`** 以避免生成 MMX 指令。

可维护性

为了确保可维护性，必须使驱动程序可以执行以下操作：

- 检测有故障的设备并报告故障
- 移除 Oracle Solaris 热插拔模型支持的设备
- 添加 Oracle Solaris 热插拔模型支持的新设备
- 执行定期的运行状况检查，以启用潜在故障检测

定期运行状况检查

潜在故障是在其他某个操作发生后才显现的故障。例如，冷待机设备中出现的硬件故障在主设备出现故障之前无法检测到。此时，系统包含两个有缺陷的设备，并且可能无法继续运行。

未检测到的潜在故障往往最终会导致系统故障。如果不执行潜在故障检查，冗余系统的整体可用性将受到危害。为避免出现这种情况，设备驱动程序必须检测潜在故障，并以与报告其他故障的方法相同的方法来报告这些故障。

应为驱动程序提供对设备进行定期运行状况检查的机制。在容错情况（其中，设备可以是辅助设备或故障转移设备）下，较早地检测到发生故障的辅助设备是确保在主设备出现故障前可以修复或更换辅助设备所必需的。

定期运行状况检查可用来执行以下活动：

- 检查自上次轮询后值已更改的设备中任何寄存器或内存的位置。

  通常表现确定行为的设备功能包括心跳信号、设备计时器（例如，下载使用的本地 `lbolt`）以及事件计数器。从设备中读取更新的可预测值可提供一切事项的进行程度令人满意的置信度。
- 时间戳外发请求，如传输块或驱动程序发出的命令。

  定期运行状况检查可以查找尚未完成的任何可疑请求。
- 在设备上启动应在下一次预定检查前完成的操作。

如果此操作为中断操作，则此检查是确保硬件设备能够送出中断的理想方法。

第 4 部分

附录

附录提供了以下背景材料：

- 附录 A，硬件概述介绍了设备驱动程序的多平台硬件问题。
- 附录 B，Oracle Solaris DDI/DKI 服务汇总提供了设备驱动程序的内核函数表。同时指出了过时的函数。
- 附录 C，使设备驱动程序支持 64 位提供了更新设备驱动程序以在 64 位环境中运行的指导原则。
- 附录 D，控制台帧缓存器驱动程序介绍了如何为帧缓存器驱动程序添加必要的接口，以使驱动程序能够与 Oracle Solaris 内核终端仿真器进行交互。
- 附录 E，`pci.conf` 文件介绍了 `pci.conf(4)` 配置文件。

◆ ◆ ◆ 附 录 A

硬件概述

本附录介绍有关可以支持 Oracle Solaris OS 的硬件的一般问题。其中包括操作系统支持的处理器、总线体系结构以及内存模型。另外，还介绍了各种设备问题以及 Oracle 平台中使用的 PROM。

注 – 本附录中的材料仅用于提供信息。此信息在调试驱动程序的过程中可能会有用。但是，Oracle Solaris DDI/DKI 接口会对设备驱动程序隐藏其中的许多实现详细信息。

本附录提供有关以下主题的信息：

SPARC 处理器问题

本节介绍了许多特定于 SPARC 处理器的主题，如数据对齐、字节排序、寄存器窗口以及浮点指令的可用性。有关特定于 x86 处理器主题的信息，请参见第 501 页中的 “x86 处理器问题”。

注 – 驱动程序决不能执行浮点操作，因为内核中不支持这些操作。

SPARC 数据对齐

所有数量均必须使用标准 C 数据类型与其自然边界对齐：

- `short` 整数在 16 位边界上对齐。
- `int` 整数在 32 位边界上对齐。
- `long` 整数在 SPARC 系统的 64 位边界上对齐。有关数据模型的信息，请参见附录 C，使设备驱动程序支持 64 位。
- `long long` 整数在 64 位边界上对齐。

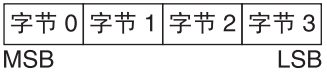
通常，编译器会处理所有对齐问题。但是，驱动程序编写者可能更关心对齐，因为只有使用正确的数据类型才能访问设备。由于设备寄存器通常是通过指针引用来访问的，因此驱动程序必须确保在访问设备时正确对齐指针。

SPARC 结构中的成员对齐

由于 SPARC 处理器强加的数据对齐限制，因此，C 结构也具有对齐要求。结构对齐要求是由最严格对齐的结构组件强加的。例如，仅包含字符的结构没有对齐限制，而对于包含 `long long` 成员的结构，则必须对其结构进行设置，保证此成员在 64 位边界上对齐。

SPARC 字节排序

SPARC 处理器使用**大尾数法**进行字节排序。整数的最高有效字节 (most significant byte, MSB) 存储在该整数的最低地址上。最低有效字节存储在此处理器中字的最高地址上。例如，字节 63 是 64 位处理器的最低有效字节。



SPARC 寄存器窗口

SPARC 处理器使用寄存器窗口。每个寄存器窗口包含八个**输入**寄存器、八个**局部**寄存器、八个**输出**寄存器以及八个**全局**寄存器。输出寄存器是下一个窗口的输入寄存器。寄存器窗口的数量范围从 2 到 32，具体取决于处理器实现。

由于驱动程序通常是使用 C 语言编写的，因此编译器通常不会指明使用了寄存器窗口这一事实。但是，当调试驱动程序时，可能必须使用寄存器窗口。

SPARC 乘法和除法指令

版本 7 SPARC 处理器没有乘法或除法指令。 乘法和除法指令是在软件中模拟实现的。由于驱动程序可能在版本 7、版本 8 或者版本 9 处理器中运行，因此请避免进行大量整数乘除。相反，请使用按位向左和向右移位来以 2 的幂进行相乘和相除。

《SPARC Architecture Manual, Version 9》介绍了有关 SPARC CPU 的更具体信息。《SPARC Compliance Definition》（版本 2.4）介绍了 SPARC V9 的应用程序二进制接口 (application binary interface, ABI) 的详细信息。本手册介绍了 32 位 SPARC V8 ABI 和 64 位 SPARC V9 ABI。可以从 SPARC International 的网站 http://www.sparc.com 上获取本文档。

x86 处理器问题

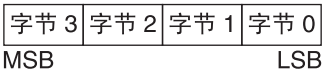
数据类型没有对齐限制。但是，x86 处理器可能需要额外的存储周期来正确处理未对齐的数据传送。

注 – 驱动程序不应执行浮点操作，因为内核中不支持这些操作。

---

x86 字节排序

x86 处理器使用**小尾数法**进行字节排序。整数的最低有效字节 (least significant byte, LSB) 存储在该整数的最低地址上。最高有效字节存储在此处理器中数据项的最高地址上。例如，字节 7 是 64 位处理器的最高有效字节。



x86 体系结构手册

Intel Corporation 和 AMD 都发布了大量有关 x86 系列处理器的书籍。请参见 http://www.intel.com 和 http://www.amd.com/。

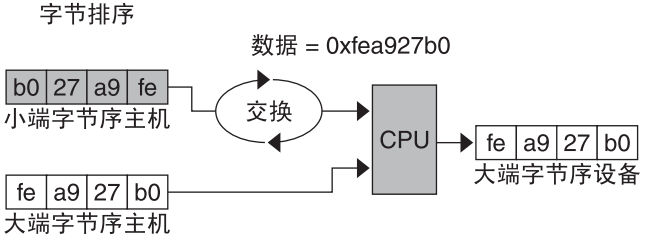
字节存储顺序

为了实现多平台、多指令集体系结构可移植性的目标，驱动程序中删除了主机总线的相关组件。要解决的第一个相关性问题是处理器的字节存储顺序，即字节排序。例如， x86 处理器系列采用小尾数法，而 SPARC 体系结构采用大尾数法。

总线体系结构显示了与处理器相同类型的字节存储顺序。例如，PCI 局部总线采用小尾数法，S 总线采用大尾数法，ISA 总线采用小尾数法等。

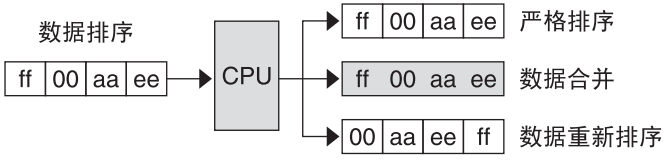
为保持处理器与总线之间的可移植性，符合 DDI 标准的驱动程序不得采用任何端字节序。虽然驱动程序可以通过运行时检查或源代码中的预处理程序指令（如 `#ifdef _LITTLE_ENDIAN`）管理其字节存储顺序，但是长期维护可能会有问题。在某些情况下，DDI 框架会使用软件方法来执行字节交换。在另外一些情况下，可以像内存管理单元 (memory management unit, MMU) 中那样通过硬件页级交换来执行字节交换，也可通过特殊计算机指令来执行字节交换。DDI 框架可以利用这些硬件功能来提高性能。

图 A–1 主机总线相关性所需的字节排序



除了不采用任何端字节排序之外，可移植驱动程序还必须独立于处理器的数据排序。在大多数情况下，必须按照驱动程序指示的顺序进行数据传送。但是，有时可以通过合并、批处理或者重新排列数据来简化数据传送，如下图中所示。例如，可以将数据合并应用于加速帧缓存器上的图形显示。驱动程序可以选择建议 DDI 框架在数据传送过程中使用其他最优传送机制。

图 A–2 数据排序主机总线相关性



存储缓冲区

为提高性能，CPU 会使用内部存储缓冲区临时存储数据。使用内部缓冲区可能会对设备 I/O 操作的同步造成影响。因此，驱动程序需要执行明确的步骤来确保在适当的时间完成对寄存器的写入。

例如，假设通过锁来同步对设备空间（如寄存器或帧缓存器）的访问。驱动程序需要检查在释放锁之前是否实际完成了向设备空间中的数据存储。释放锁时并不一定会刷新 I/O 缓冲区。

另一个示例是，确认中断时，驱动程序通常会在设备控制寄存器中设置或清除一位。驱动程序必须确保在中断程序返回之前，已开始在设备上对控制寄存器进行写入。同样，在向控制寄存器写入了某一命令之后，设备可能要求延迟，即驱动程序繁忙，需要等待。在这种情况下，驱动程序必须确保在设备延迟之前已开始在该设备上进行写入。

当读取设备寄存器不会产生不良负面影响时，只需在写入之后立即读取就可以对写入进行验证了。如果无法在不产生不良负面影响的情况下读取该特定寄存器，则可以使用同一寄存器集中的其他设备寄存器。

系统内存模型

系统内存模型用于定义内存操作（如**装入**和**存储**）的语义，并指定处理器执行这些操作的顺序与操作到达内存的顺序之间的关系。内存模型可同时适用于单处理器和共享内存多处理器。支持两种内存模型：全存储排序 (total store ordering, TSO) 和部分存储排序 (partial store ordering, PSO)。

全存储排序 (Total Store Ordering, TSO)

TSO 可保证存储、FLUSH 以及原子装入存储指令出现在给定处理器的内存中的顺序与该处理器发出这些指令的顺序相同。

x86 和 SPARC 处理器均支持 TSO。

部分存储排序 (Partial Store Ordering, PSO)

PSO 无法保证存储、FLUSH 以及原子装入存储指令出现在给定处理器的内存中的顺序与该处理器发出这些指令的顺序相同。处理器可以对存储的指令重新排序，以使内存的存储指令顺序与 CPU 发出的存储指令顺序不同。

SPARC 处理器支持 PSO；x86 处理器则不支持。

对于 SPARC 处理器，指令的**发出**顺序和**内存**顺序之间的一致性是由系统框架使用 STBAR 指令实现的。如果以上指令中的两条指令按处理器的发出顺序由 STBAR 指令分隔，或者指令引用同一位置，则这两条指令的内存存储顺序与发出顺序相同。使用 `ddi_regs_map_setup(9F)` 接口可强制执行兼容 DDI 的驱动程序中的强数据排序。兼容的驱动程序不能直接使用 STBAR 指令。

有关 SPARC 内存模型的更多详细信息，请参见《SPARC Architecture Manual, Version 9》。

总线体系结构

本节介绍了设备标识、设备寻址和中断。

设备标识

设备标识是确定系统中存在哪些设备的过程。某些设备是自标识设备，意味着设备本身向系统提供信息，以便系统可以标识需要使用的设备驱动程序。S 总线和 PCI 局部总线设备是自标识设备的示例。在 S 总线上，信息通常是从设备上 FCode PROM 中存储的小 Forth 程序派生而来。大多数 PCI 设备都会提供包含设备配置信息的配置空间。有关更多信息，请参见 `sbus(4)` 和 `pci(4)` 手册页。

所有现代总线体系结构都要求设备进行自标识。

支持的中断类型

Oracle Solaris 平台支持轮询中断和向量化中断。对于这两种中断类型，Oracle Solaris DDI/DKI 中断模型均相同。有关中断处理的更多信息，请参见第 8 章，中断处理程序。

总线特定信息

本节介绍特定于 Oracle Solaris 平台支持的总线的寻址问题和设备配置问题。

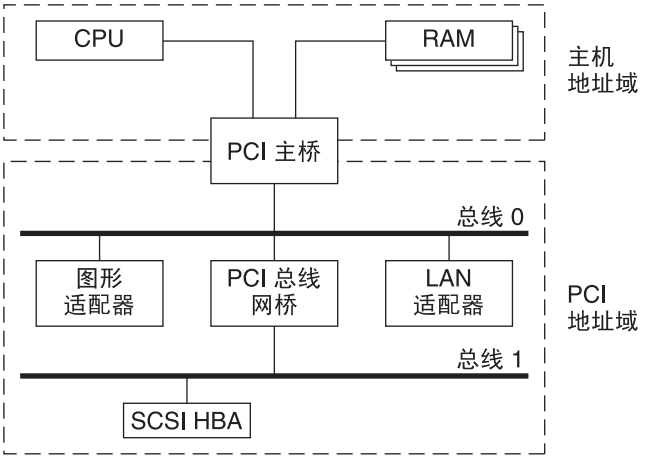
PCI 局部总线

PCI 局部总线是设计用于实现高速数据传输的高性能总线。PCI 总线驻留在系统板上。此总线通常用作高度集成的外围组件、外围附件板以及主机处理器或内存系统之间的互连机制。主机处理器、主内存和 PCI 总线本身都通过 PCI 主桥 (host bridge) 连接，如图 A–3中所示。

互连的 I/O 总线树结构通过一系列 PCI 总线网桥进行支持。可以在 PCI 主桥 (host bridge) 下扩展从属 PCI 总线网桥，以使单总线系统扩展为带有多条辅助总线的复杂系统。PCI 设备可以连接到其中的一条或多条辅助总线。此外，还可以连接其他总线网桥，如 SCSI 或 USB。

每个 PCI 设备都具有唯一的供应商 ID 和设备 ID。相同种类的多台设备会通过其驻留的总线上的唯一设备号进一步标识。

图 A–3 计算机结构图



PCI 主桥 (host bridge) 用于提供处理器和外围组件之间的互连。通过 PCI 主桥 (host bridge)，处理器可以直接访问独立于其他 PCI 总线主控器的主内存。例如，当 CPU 正在从主桥 (host bridge) 中的高速缓存控制器中提取数据时，其他 PCI 设备也可以通过该主桥 (host bridge) 访问系统内存。这种体系结构的优点在于其分隔了 I/O 总线与处理器的主机总线。

PCI 主桥 (host bridge) 还可提供 CPU 和外围 I/O 设备之间的数据访问映射。该桥会将每个外围设备映射到主机地址域，以便处理器可以通过程控 I/O 访问此设备。在局部总线端，PCI 主桥 (host bridge) 会将系统内存映射到PCI 地址域，以便 PCI 设备可以作为总线主控器访问主机内存。图 A–3 显示了两种地址域。

PCI 地址域

PCI 地址域包含三种不同的地址空间：配置、内存以及 I/O 空间。

PCI 配置地址空间

配置空间按地理位置定义。外围设备的位置通过它在互连的 PCI 总线网桥树中的物理位置确定。设备按其**总线编号**和**设备**（**插槽**）**编号**进行定位。每个外围设备在其 PCI 配置空间中都包含一组明确定义的配置寄存器。这些寄存器不仅用于标识设备，还用于为配置框架提供设备配置信息。例如，必须首先映射设备配置空间中的基址寄存器，然后设备才能响应数据访问。

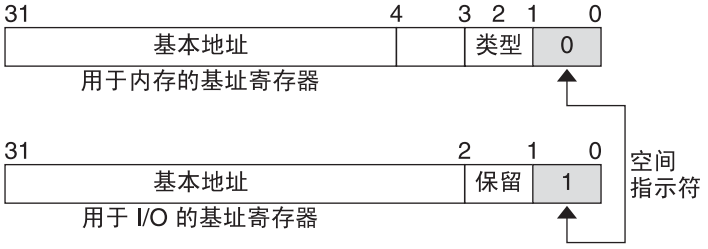
生成配置周期的方法取决于主机。x86 计算机中使用的是特殊的 I/O 端口。在其他平台上，可以将 PCI 配置空间内存映射到对应于主机地址域中 PCI 主桥 (host bridge) 的某些地址位置。处理器访问设备配置寄存器时，会将请求路由到 PCI 主桥 (host bridge)。然后，该桥会在总线上将访问转换为正确的配置周期。

PCI 配置基址寄存器

PCI 配置空间针对每台设备最多包含六个 32 位基址寄存器。这些寄存器可同时提供大小和数据类型信息。系统固件会将 PCI 地址域中的基本地址分配给这些寄存器。

每个可寻址区域既可以是内存空间，也可以是 I/O 空间。基址寄存器的位 0 包含的值用于标识类型。位 0 中的值为 0 表示内存空间，值为 1 表示 I/O 空间。下图显示了两种基址寄存器：一种表示内存类型，另一种表示 I/O 类型。

图 A–4 内存和 I/O 的基址寄存器



PCI 内存地址空间

PCI 同时支持 32 位和 64 位的内存空间地址。系统固件会将 PCI 地址域中的内存空间区域分配给 PCI 外围设备。区域的基本地址存储在设备的 PCI 配置空间的基址寄存器中。每个区域的大小必须是 2 的幂，并且所分配的基本地址必须在与区域大小相等的边界上对齐。内存空间中的设备地址会**内存映射**到主机地址域中，以便处理器的本机装入指令或存储指令可以对任何设备执行数据访问。

PCI I/O 地址空间

PCI 支持 32 位 I/O 空间。可以在不同的平台上以不同方式访问 I/O 空间。带有特殊 I/O 指令的处理器（如 Intel 处理器系列）使用 `in` 和 `out` 指令访问 I/O 空间。没有特殊 I/O 指令的计算机将映射到对应于主机地址域中 PCI 主桥 (host bridge) 的地址位置。处理器

访问内存映射的地址时，会向 PCI 主桥 (host bridge) 发送一个 I/O 请求，该主桥 (host bridge) 随后会将地址转换为 I/O 周期并将其放置在 PCI 总线上。内存映射的 I/O 通过处理器的本机装入/存储指令执行。

PCI 硬件配置文件

对于 PCI 局部总线设备，硬件配置文件应当不是必需的。但是在某些情况下，PCI 设备的驱动程序需要使用硬件配置文件来增加驱动程序的专用信息。有关完整的详细信息，请参见 `driver.conf(4)` 和 `pci(4)` 手册页。

PCI Express

标准 PCI 总线已发展为 PCI Express。PCI Express 是下一代高性能 I/O 总线，用于连接桌面、移动设备、工作站、服务器以及嵌入式计算和通信平台之类的应用程序中的外围设备。

PCI Express 可提高总线性能，减少整体系统支出，并可利用计算机设计中新的发展成果。PCI Express 使用串行的点对点类型互连在两台设备之间实现通信。通过交换机，用户可以在某个系统中将大量设备连接在一起。串行互连意味着每台设备软件包的管脚更少，这可降低成本并使性能具有高度可伸缩性。

PCI Express 总线具有内置功能，可以适应以下技术：

- QoS（Quality of Service，服务质量）
- 热插拔和热交换
- 高级电源管理
- RAS（Reliability, Available, Serviceable，可靠性、可用性、可维护性）
- 改进的错误处理
- MSI 中断

将两台设备连接在一起的 PCI Express 互连称为**链路**。链路可以是 x1、x2、x4、x8、x12、x16 或 x32 双向的信号对。这些信号称为**通道**。双工模式中每条通道的带宽 (x1) 为 500 MB/秒。虽然 PCI-X 和 PCI Express 具有不同的硬件连接，但是对驱动程序编写者来说，两种总线是相同的。PCI-X 是共享总线。例如，总线上的所有设备都共享单独的一组数据线和信号线。PCI-Express 是交换总线，通过它可以更有效地使用设备和系统总线之间的带宽。

有关 PCI Express 的更多信息，请参阅以下 Web 站点：`http://www.pcisig.com/home`

S 总线

典型的 S 总线系统由主板（包含 CPU 和 S 总线接口逻辑）、主板本身上的大量 S 总线设备以及大量 S 总线扩展槽组成。另外，还可以通过相应的总线网桥将 S 总线连接到其他类型的总线。

S 总线按地理位置进行寻址。每个 S 总线插槽位于系统中固定的物理地址上。S 总线卡具有不同的地址，具体取决于其插入的插槽。将 S 总线设备移动到新插槽会导致系统将此设备视为新设备。

S 总线使用轮询中断。S 总线设备中断时，系统仅知道若干设备中的哪些设备可能发出该中断。系统中断处理程序必须询问每台设备的驱动程序此设备是否负责中断。

S 总线物理地址空间

下表显示了 Sun UltraSPARC 2 计算机的物理地址空间布局。UltraSPARC 2 模型上的物理地址包含 41 位。该 41 位的物理地址空间会进一步分为多个通过 `PA(40:33)` 标识的 33 位地址空间。

表 A–1 Ultra 2 中的设备物理空间

PA(40:33)	33 位空间	使用情况
0x0	`0x000000000 - 0x07FFFFFF`	2 GB 主内存
0x80 – 0xDF	`Reserved on Ultra 2`	在 Ultra 2 上保留
0xE0	`Processor 0`	处理器 0
0xE1	`Processor 1`	处理器 1
0xE2 – 0xFD	`Reserved on Ultra 2`	在 Ultra 2 上保留
0xFE	`0x000000000 - 0x1FFFFFFFF`	从属 UPA (FFB)
0xFF	`0x000000000 - 0x0FFFFFFFF`	系统 I/O 空间
	`0x100000000 - 0x10FFFFFFF`	S 总线插槽 0
	`0x110000000 - 0x11FFFFFFF`	S 总线插槽 1
	`0x120000000 - 0x12FFFFFFF`	S 总线插槽 2
	`0x130000000 - 0x13FFFFFFF`	S 总线插槽 3
	`0x1D0000000 - 0x1DFFFFFFF`	S 总线插槽 D
	`0x1E0000000 - 0x1EFFFFFFF`	S 总线插槽 E
	`0x1F0000000 - 0x1FFFFFFFF`	S 总线插槽 F

物理 S 总线地址

S 总线具有 32 个地址位，如《SBus Specification》中所述。下表介绍 Ultra 2 如何使用地址位。

表 A–2 Ultra 2 S 总线地址位

位	说明
0 - 27	这些位是 S 总线卡用于寻址该卡的内容的 S 总线地址行。
28 - 31	供 CPU 用于选择其中一个 S 总线插槽。这些位会生成 SlaveSelect 行。

此寻址方案将生成表 A–1 中显示的 Ultra 2 地址。其他实现可能会使用不同数量的地址位。

Ultra 2 具有七个 S 总线插槽，其中四个是物理插槽。插槽 0 到 3 可供 S 总线卡使用。插槽 4-12 为保留插槽。插槽的使用情况如下：

- 插槽 0 到 3 是具有 DMA 主控制器功能的物理插槽。
- 插槽 D、E 以及 F 并非实际物理插槽，而是指板载直接内存访问 (direct memory access, DMA) 控制器、SCSI 控制器、以太网控制器以及音频控制器。为方便起见，会将这些设备类视为插入了插槽 D、E 以及 F。

注 – 某些 S 总线插槽是仅从属插槽。需要 DMA 功能的驱动程序应使用 `ddi_slaveonly(9F)` 来确定其设备是否位于具有 DMA 功能的插槽中。有关此函数的示例，请参见第 101 页中的 “attach() 入口点”。

S 总线硬件配置文件

通常，S 总线设备不需要硬件配置文件。但是在某些情况下，S 总线设备的驱动程序需要使用硬件配置文件来增加 S 总线卡所提供的信息。有关完整的详细信息，请参见 `driver.conf(4)` 和 `sbus(4)` 手册页。

设备问题

本节介绍特殊设备的问题。

时间关键型部分

虽然在有了锁定原语所提供的同步和保护机制的情况下可以执行大多数驱动程序操作，但是对于某些设备而言，必须在没有中断的情况下按顺序发生一系列事件。函数 `ddi_enter_critical(9F)` 与锁定原语一起将请求系统尽可能保证不会抢占或中断当前线程。在进行对 `ddi_exit_critical(9F)` 的关闭调用之前，此保证将一直有效。有关详细信息，请参见 `ddi_enter_critical(9F)` 手册页。

延迟

许多芯片指定只能在指定间隔对其进行访问。例如， Zilog Z8530 SCC 具有 1.6 微秒的“写入恢复时间”。此规范意味着通过 8530 写入字符时，必须使用 drv_usecwait(9F) 强制延迟。在某些情况下，规范不会明确指示所需的延迟，因此必须根据经验来确定延迟。

请注意不要组合可能大量存在的设备部件的延迟，例如数以千计的 SCSI 磁盘驱动器。

内部顺序逻辑

具有内部顺序逻辑的设备会将多个内部寄存器映射到同一外部地址。各种内部顺序逻辑包括以下类型：

- Intel 8251A 和 Signetics 2651 可在**两个**内部模式寄存器之间替换同一外部寄存器。通过向外部寄存器进行写入可实现对第一个内部寄存器的写入。但是，这种写入操作有不利的一面，即要在芯片中设置顺序逻辑，以使下一个读/写操作是在另一个内部寄存器上进行。
- NEC PD7201 PCC 具有多个内部数据寄存器。要将字节写入特定寄存器，必须执行两个步骤。第一步是将以下数据字节将进入的寄存器的编号写入寄存器零。然后，将数据写入指定的数据寄存器。顺序逻辑会自动设置芯片，以便发送的下一字节进入数据寄存器零。
- AMD 9513 计时器具有一个数据指针寄存器，指向数据字节将进入的数据寄存器。向数据寄存器发送字节时，该指针会递增。**无法**读取指针寄存器的当前值。

中断问题

请注意以下常见的与中断相关的问题：

- 控制器中断**不**一定会指示控制器**及**其从属设备之一**均**已就绪。对于某些控制器，中断可指示控制器或其设备之一已就绪，但不是均已就绪。
- 并非所有设备都会在禁用中断时打开电源，也不是所有设备都可随时开始中断。
- 某些设备不提供用于确定板是否已经生成中断的方法。
- 并非所有中断的板都会在被告知关闭中断时或在总线重置之后关闭中断。

SPARC 计算机上的 PROM

某些平台具有 PROM 监视器，支持在没有操作系统的情况下调试设备。本节介绍如何使用 SPARC 计算机上的 PROM 来映射设备寄存器，以便可对其进行访问。通常，可以使用 PROM 命令对设备进行充分测试，以确定设备是否正常工作。

有关 x86 引导子系统的说明，请参见 boot(1M) 手册页。

PROM 具有多种用途，包括：

- 通过打开电源或硬重置 PROM reset 命令初启计算机
- 提供交互工具来检查和设置内存、设备寄存器以及内存映射
- 引导 Oracle Solaris 系统。

  仅打开计算机电源并尝试使用其 PROM 来检查设备寄存器会失败。虽然可能正确安装了设备，但是这些映射特定于 Oracle Solaris OS，并且直到 Oracle Solaris 内核引导后才会变为活动状态。打开电源之后，PROM 仅映射基本系统设备，如键盘。
- 使用 sync 命令执行系统故障转储

Open Boot PROM 3

有关 Open Boot PROM 的完整文档，请参见《Open Boot PROM Toolkit User's Guide》和 monitor(1M) 手册页。本节中的示例引用的是 Sun4U 体系结构。其他体系结构可能要求不同的命令来执行操作。

注 – Open Boot PROM 当前在具有 S 总线或 UPA/PCI 的 Sun 计算机上使用。Open Boot PROM 使用 "ok" 提示符。在早期计算机上，可能必须键入 'n' 才能获取 "ok" 提示符。

如果 PROM 处于**安全模式**（security-mode 参数并未设置为**无**），则可能需要 PROM 口令（在 security-password 参数中设置）。

printenv 命令用于显示所有参数及其值。

使用 help 命令可以获取帮助信息。

可以使用 EMACS 样式的命令行历史记录。使用 Ctrl-N （下一步）和 Ctrl-P （上一步）可以遍历历史记录列表。

Forth 命令

Open Boot PROM 使用 Forth 编程语言。Forth 是一种基于栈的语言。必须将参数推送到栈上，然后再运行正确的命令（称为**字**），并且结果将留在栈上。

要对栈进行编号，请键入其值。

```
ok 57
ok 68
```

要在栈上添加两个顶部值，请使用 + 运算符。

```
ok +
```

结果会保留在栈上。栈显示有**单词** .s。

```
ok .s
bf
```

缺省基值为十六进制。可以使用**单词** hex 和 decimal 来切换基值。

```
ok decimal
ok .s
191
```

有关更多信息，请参见《Forth User's Guide》。

遍历 PROM 设备树

pwd、cd 和 ls 命令将遍历 PROM 设备树以查找设备。必须首先使用 cd 命令在树中建立一个位置，然后pwd 才能运行。本示例为在 S 总线上带有 cgsix 帧缓存器的 Ultra 1 工作站。

```
ok cd /
```

要查看连接到树中的当前节点的设备，请使用 ls 命令。

```
ok ls
f006a064 SUNW,UltraSPARC@0,0
f00598b0 sbus@1f,0
f00592dc counter-timer@1f,3c00
f004eec8 virtual-memory
f004e8e8 memory@0,0
f002ca28 aliases
f002c9b8 options
f002c880 openprom
f002c814 chosen
f002c7a4 packages
```

可以使用全节点名称：

```
ok cd sbus@1f,0
ok ls
f006a4e4 cgsix@2,0
f0068194 SUNW,bpp@e,c800000
f0065370 ledma@e,8400010
f006120c espdma@e,8400000
f005a448 SUNW,pll@f,1304000
f005a394 sc@f,1300000
f005a24c zs@f,1000000
```

```
f005a174 zs@f,1100000
f005a0c0 eeprom@f,1200000
f0059f8c SUNW,fdtwo@f,1400000
f0059ec4 flashprom@f,0
f0059e34 auxio@f,1900000
f0059d28 SUNW,CS4231@d,c000000
```

如果不使用前一个示例中的全节点名称，则还可以使用缩写。缩写命令行项类似于以下示例：

```
ok cd sbus
```

对于 S 总线设备，名称实际为 `device@slot,offset`。`cgsix` 设备位于插槽 2 中，并在偏移 0 处开始。如果该树中显示了 S 总线设备，则表明 PROM 已经识别了此设备。

`.properties` 命令用于显示设备的 PROM 属性。通过检查这些属性可以确定设备导出的属性。以后可以使用此信息来确保驱动程序查找的是正确的硬件属性。这些属性与可以使用 `ddi_getprop(9F)` 检索的属性相同。

```
ok cd cgsix
ok .properties
character-set            ISO8859-1
intr                     00000005 00000000
interrupts               00000005
reg                      00000002 00000000 01000000
dblbuf                   00 00 00 00
vmsize                   00 00 00 01
...
```

`reg` 属性用于定义包含以下字段的寄存器说明结构的数组：

```
uint_t        bustype;       /* cookie for related bus type*/
uint_t        addr;          /* address of reg relative to bus */
uint_t        size;          /* size of this register set */
```

对于 `cgsix` 示例，地址为 0。

映射设备

必须将设备映射到内存中才能进行测试。然后，可以使用 PROM 来验证设备是否正确操作，方法是使用数据传送命令来传送字节、字以及长字。如果可以通过 PROM 操作设备（即使使用受限的方法），则驱动程序也应该可以操作设备。

要设置设备以进行初始测试，请执行以下步骤：

1. 确定设备所在的 S 总线插槽编号。

   在本示例中，`cgsix` 设备位于插槽 2 中。

2. 确定设备使用的物理地址空间中的偏移。

   所使用的偏移特定于设备。在 `cgsix` 示例中，视频内存恰好在偏移 0x800000 处开始。

3. 使用 select-dev **字**可选择 S 总线设备以及在其中映射此设备的 map-in **字**。

   字 select-dev 采用设备路径的字符串作为其参数。map-in **单词**接受**偏移**、**插槽编号**以及**大小**作为映射的参数。与偏移一样，字节传送的大小也特定于设备。在 cgsix 示例中，大小设置为 0x100000 字节。

   在以下代码示例中，S 总线路径显示为单词 select-dev 的参数，帧缓存器的偏移、插槽编号以及大小值显示为单词 map-in 的参数。请注意 select-dev 参数中起始引号和 / 之间的空格。要使用的虚拟地址保留在栈的顶部。栈通过使用字 .s 进行显示。通过 constant 操作可为该栈分配一个名称。

```
ok " sbus@1f,0" select-dev
ok 800000 2 100000 map-in
ok .s
ffe98000
ok constant fb
```

读取和写入

PROM 提供了许多 8 位、16 位以及 32 位操作。通常，c（字符）前缀表示 8 位（一字节）操作；w（字）前缀表示 16 位（二字节）操作；L（长字）前缀表示 32 位（四字节）操作。

后缀 ! 表示写入操作。写入操作用于从栈中取出前两项。第一项是地址，第二项是值。

```
ok 55 ffe98000 c!
```

后缀 @ 表示读取操作。读取操作用于从栈中取出地址。

```
ok ffe98000 c@
ok .s
55
```

后缀 ? 用于显示值，并且不会影响栈。

```
ok ffe98000 c?
55
```

尝试查询设备时，请务必谨慎。如果未正确设置映射，则尝试读取或写入可能会导致错误。为处理这些情况，提供了特殊字。例如，cprobe、wprobe 和 lprobe 会从给定地址进行读取，但是如果此位置不响应，则会返回零；如果此位置响应，则返回非零值。

```
ok fffa4000 c@
Data Access Error

ok fffa4000 cprobe
ok .s0
```

```
ok ffe98000 cprobe
ok .s
0 ffffffffffffffff
```

使用字 dump 可以显示内存的区域。这会采用 *address* 和 *length*，并以字节为单位显示内存区域的内容。

在以下示例中，字 fill 用于使用某种模式填充视频内存。fill 会采用地址、要填充的字节数以及要使用的字节。对于字和长字，请分别使用 wfill 和 Lfill。此填充示例会导致 cgsix 基于传递的字节显示简单模式。

```
ok " /sbus" select-dev
ok 800000 2 100000 map-in
ok constant fb
ok fb 10000 ff fill
ok fb 20000 0 fill
ok fb 18000 55 fill
ok fb 15000 3 fill
ok fb 10000 5 fillok fb 5000 f9 fill
```

◆ ◆ ◆ 附 录 B

Oracle Solaris DDI/DKI 服务汇总

本附录介绍 Oracle Solaris DDI/DKI 所提供的接口。这些说明并不具有完整性和确定性，也未提供详细的使用指导，而是旨在使用一般术语介绍函数功能。有关更多详细信息，请参见 `physio(9F)`。介绍的类别如下：

模块函数

模块函数包括：

`mod_info`	查询可装入模块
`mod_install`	添加可装入模块
`mod_remove`	删除可装入模块

设备信息树节点 (dev_info_t) 函数

设备信息树节点函数包括：

`ddi_binding_name()`	返回驱动程序绑定名称
`ddi_dev_is_sid()`	指示设备是否能自我识别
`ddi_driver_major()`	返回驱动程序主设备号
`ddi_driver_name()`	返回标准化驱动程序名称
`ddi_node_name()`	返回 `devinfo` 节点名称
`ddi_get_devstate()`	检查设备状态
`ddi_get_instance()`	获取设备实例编号
`ddi_get_name()`	返回驱动程序绑定名称
`ddi_get_parent()`	查找设备信息结构的父节点
`ddi_root_node()`	获取 `dev_info` 树的根

设备 (dev_t) 函数

设备函数包括：

`ddi_create_minor_node()`	为设备创建次要节点
`ddi_getiminor()`	从外部 `dev_t` 中获取内核内部次要设备号
`ddi_remove_minor_node()`	删除设备的次要节点
`getmajor()`	获取主设备号
`getminor()`	获取次要设备号
`makedevice()`	根据主设备号和次要设备号生成设备编号

属性函数

属性函数包括 :

ddi_prop_exists()	检查属性是否存在
ddi_prop_free()	释放属性查找使用的资源
ddi_prop_get_int()	查找整数属性
ddi_prop_get_int64()	查找 64 位整数属性
ddi_prop_lookup_byte_array()	查找字节数组属性
ddi_prop_lookup_int_array()	查找整数数组属性
ddi_prop_lookup_int64_array()	查找 64 位整数数组属性
ddi_prop_lookup_string()	查找字符串属性
ddi_prop_lookup_string_array()	查找字符串数组属性
ddi_prop_remove()	删除设备的一个属性
ddi_prop_remove_all()	删除设备的所有属性
ddi_prop_undefine()	隐藏设备的一个属性
ddi_prop_update_byte_array()	创建或更新字节数组属性
ddi_prop_update_int()	创建或更新整数属性
ddi_prop_update_int64()	创建或更新 64 位整数属性
ddi_prop_update_int_array()	创建或更新整数数组属性
ddi_prop_update_int64_array()	创建或更新 64 位整数数组属性
ddi_prop_update_string()	创建或更新字符串属性
ddi_prop_update_string_array()	创建或更新字符串数组属性

表 B–1 过时的属性函数

过时的函数	替代函数
ddi_getlongprop()	请参见 ddi_prop_lookup()
ddi_getlongprop_buf()	ddi_prop_lookup()
ddi_getprop()	ddi_prop_get_int()
ddi_getproplen()	ddi_prop_lookup()
ddi_prop_create()	ddi_prop_lookup()

表 B–1 过时的属性函数 （续）

过时的函数	替代函数
ddi_prop_modify()	ddi_prop_lookup()
ddi_prop_op()	ddi_prop_lookup()

设备软件状态函数

设备软件状态函数包括：

ddi_get_driver_private()	获取设备的专用数据区的地址
ddi_get_soft_state()	获取指向实例软状态结构的指针
ddi_set_driver_private()	设置设备的专用数据区的地址
ddi_soft_state_fini()	销毁驱动程序软状态结构
ddi_soft_state_free()	释放实例软状态结构
ddi_soft_state_init()	初始化驱动程序软状态结构
ddi_soft_state_zalloc()	分配实例软状态结构

内存分配和取消分配函数

内存分配和取消分配函数包括：

kmem_alloc()	分配内核内存
kmem_free()	释放内核内存
kmem_zalloc()	分配零填充的内核内存

以下函数可以分配和释放用于 DMA 的内存。请参见第 530 页中的 “直接内存访问 (Direct Memory Access, DMA) 函数”。

ddi_dma_mem_alloc()	为 DMA 传送操作分配内存
ddi_dma_mem_free()	释放以前分配的 DMA 内存

以下函数可以分配和释放用于导出到用户空间的内存。请参见第 532 页中的 “用户空间访问函数”。

ddi_umem_alloc()	分配按页对齐的内核内存
ddi_umem_free()	释放按页对齐的内核内存

表 B–2 过时的内存分配和取消分配函数

过时的函数	替代函数
ddi_iopb_alloc()	ddi_dma_mem_alloc()
ddi_iopb_free()	ddi_dma_mem_free()
ddi_mem_alloc()	ddi_dma_mem_alloc()
ddi_mem_free()	ddi_dma_mem_free()

内核线程控制和同步函数

内核线程控制和同步函数包括：

cv_broadcast()	唤醒所有等待线程
cv_destroy()	释放已分配的条件变量
cv_init()	分配条件变量
cv_signal()	唤醒一个等待线程
cv_timedwait()	等待事件，具有超时设置
cv_timedwait_sig()	等待事件或信号，具有超时设置
cv_wait()	等待事件
cv_wait_sig()	等待事件或信号
ddi_can_receive_sig()	确定当前线程是否可以接收信号
ddi_enter_critical()	进入关键控制区
ddi_exit_critical()	退出关键控制区
mutex_destroy()	销毁互斥锁
mutex_enter()	获取互斥锁
mutex_exit()	释放互斥锁
mutex_init()	初始化互斥锁
mutex_owned()	确定当前线程是否持有互斥锁
mutex_tryenter()	尝试获取互斥锁，但不等待
rw_destroy()	销毁读取器/写入器锁
rw_downgrade()	将持有的读取器/写入器锁从写入器降级为读取器
rw_enter()	获取读取器/写入器锁

rw_exit()	释放读取器/写入器锁
rw_init()	初始化读取器/写入器锁
rw_read_locked()	确定持有的读取器/写入器锁是用于读取还是用于写入
rw_tryenter()	尝试获取读取器/写入器锁，但不等待
rw_tryupgrade()	尝试将持有的读取器/写入器锁从读取器升级为写入器
sema_destroy()	销毁信号
sema_init()	初始化信号
sema_p()	递减信号并可能阻塞
sema_p_sig()	递减信号，但信号待处理时不阻塞
sema_tryp()	尝试递减信号，但不阻塞
sema_v()	递增信号并可能解除阻塞等待程序

任务队列管理函数

下面列出了任务队列管理函数。有关这些接口的更多信息，请参见 taskq(9F) 手册页。

ddi_taskq_create()	创建任务队列
ddi_taskq_destroy()	销毁任务队列
ddi_taskq_dispatch()	在任务队列中添加任务
ddi_taskq_wait()	等待暂挂的任务完成
ddi_taskq_suspend()	暂挂任务队列
ddi_taskq_suspended()	检查任务队列是否已暂挂
ddi_taskq_resume()	恢复暂挂的任务队列

中断函数

中断函数包括：

ddi_intr_add_handler(9F)	添加中断处理程序。
ddi_intr_add_softint(9F)	添加软中断处理程序。
ddi_intr_alloc(9F)	为指定类型的中断分配系统资源和中断向量。
ddi_intr_block_disable(9F)	禁用指定范围的中断。仅适用于 MSI。

ddi_intr_block_enable(9F)	启用指定范围的中断。仅适用于 MSI。
ddi_intr_clr_mask(9F)	如果已启用指定的中断，则清除中断屏蔽码。
ddi_intr_disable(9F)	禁用指定的中断。
ddi_intr_dup_handler(9F)	仅适用于 MSI-X。将分配的中断向量的地址和数据对复制到同一设备上未使用的中断向量。
ddi_intr_enable(9F)	启用指定的中断。
ddi_intr_free(9F)	针对指定的中断句柄释放系统资源和中断向量。
ddi_intr_get_cap(9F)	针对指定的中断返回中断功能标志。
ddi_intr_get_hilevel_pri(9F)	返回高级别中断的最低优先级别。
ddi_intr_get_navail(9F)	返回可用于特定硬件设备和给定中断类型的中断的数量。
ddi_intr_get_nintrs(9F)	针对给定的中断类型获取设备支持的中断数。
ddi_intr_get_pending(9F)	读取中断待处理位（如果主桥 (host bridge) 或设备支持）。
ddi_intr_get_pri(9F)	返回指定中断的当前软件优先级设置。
ddi_intr_get_softint_pri(9F)	返回指定中断的软中断优先级。
ddi_intr_get_supported_types(9F)	返回设备和主机均支持的硬件中断类型。
ddi_intr_remove_handler(9F)	删除指定的中断处理程序。
ddi_intr_remove_softint(9F)	删除指定的软中断处理程序。
ddi_intr_set_cap(9F)	为指定的中断设置 DDI_INTR_FLAG_LEVEL 或 DDI_INTR_FLAG_EDGE 标志。
ddi_intr_set_mask(9F)	如果已启用指定的中断，则设置中断屏蔽码。
ddi_intr_set_pri(9F)	设置指定中断的中断优先级别。
ddi_intr_set_softint_pri(9F)	更改指定软中断的相对软中断优先级。
ddi_intr_trigger_softint(9F)	触发指定的软中断。

要利用新框架的功能，请使用上述接口。请勿使用下表中列出的过时接口。保留这些过时接口只是出于兼容性目的。

表 B–3 过时的中断函数

过时的中断函数	替代函数
ddi_add_intr(9F)	包含三个步骤的过程： 1. ddi_intr_alloc(9F) 2. ddi_intr_add_handler(9F) 3. ddi_intr_enable(9F)
ddi_add_softintr(9F)	ddi_intr_add_softint(9F)
ddi_dev_nintrs(9F)	ddi_intr_get_nintrs(9F)
ddi_get_iblock_cookie(9F)	包含三个步骤的过程： 1. ddi_intr_alloc(9F) 2. ddi_intr_get_pri(9F) 3. ddi_intr_free(9F)
ddi_get_soft_iblock_cookie(9F)	包含三个步骤的过程： 1. ddi_intr_add_softint(9F) 2. ddi_intr_get_softint_pri(9F) 3. ddi_intr_remove_softint(9F)
ddi_intr_hilevel(9F)	包含三个步骤的过程： 1. ddi_intr_alloc(9F) 2. ddi_intr_get_hilevel_pri(9F) 3. ddi_intr_free(9F)
ddi_remove_intr(9F)	包含三个步骤的过程： 1. ddi_intr_disable(9F) 2. ddi_intr_remove_handler(9F) 3. ddi_intr_free(9F)
ddi_remove_softintr(9F)	ddi_intr_remove_softint(9F)
ddi_trigger_softintr(9F)	ddi_intr_trigger_softint(9F)

程控 I/O 函数

程控 I/O 函数包括：

ddi_dev_nregs()	返回设备的寄存器集数
ddi_dev_regsize()	返回设备寄存器的大小
ddi_regs_map_setup()	为寄存器地址空间设置映射
ddi_regs_map_free()	释放以前映射的寄存器地址空间
ddi_device_copy()	在设备寄存器之间复制数据

ddi_device_zero()	零填充设备
ddi_check_acc_handle()	检查数据访问句柄
ddi_get8()	从映射的内存、设备寄存器或 DMA 内存中读取一个 8 位数据
ddi_get16()	从映射的内存、设备寄存器或 DMA 内存中读取一个 16 位数据
ddi_get32()	从映射的内存、设备寄存器或 DMA 内存中读取一个 32 位数据
ddi_get64()	从映射的内存、设备寄存器或 DMA 内存中读取一个 64 位数据
ddi_put8()	向映射的内存、设备寄存器或 DMA 内存中写入一个 8 位数据
ddi_put16()	向映射的内存、设备寄存器或 DMA 内存中写入一个 16 位数据
ddi_put32()	向映射的内存、设备寄存器或 DMA 内存中写入一个 32 位数据
ddi_put64()	向映射的内存、设备寄存器或 DMA 内存中写入一个 64 位数据
ddi_rep_get8()	从映射的内存、设备寄存器或 DMA 内存中读取多个 8 位数据
ddi_rep_get16()	从映射的内存、设备寄存器或 DMA 内存中读取多个 16 位数据
ddi_rep_get32()	从映射的内存、设备寄存器或 DMA 内存中读取多个 32 位数据
ddi_rep_get64()	从映射的内存、设备寄存器或 DMA 内存中读取多个 64 位数据
ddi_rep_put8()	向映射的内存、设备寄存器或 DMA 内存中写入多个 8 位数据
ddi_rep_put16()	向映射的内存、设备寄存器或 DMA 内存中写入多个 16 位数据
ddi_rep_put32()	向映射的内存、设备寄存器或 DMA 内存中写入多个 32 位数据
ddi_rep_put64()	向映射的内存、设备寄存器或 DMA 内存中写入多个 64 位数据

ddi_peek8()	从某一位置慎重读取一个 8 位的值
ddi_peek16()	从某一位置慎重读取一个 16 位的值
ddi_peek32()	从某一位置慎重读取一个 32 位的值
ddi_peek64()	从某一位置慎重读取一个 64 位的值
ddi_poke8()	向某一位置慎重写入一个 8 位的值
ddi_poke16()	向某一位置慎重写入一个 16 位的值
ddi_poke32()	向某一位置慎重写入一个 32 位的值
ddi_poke64()	向某一位置慎重写入一个 64 位的值

可以始终使用上面列出的一般程控 I/O 函数，而不必使用下面的 mem、io 和 pci_config 函数。但如果编译时已知访问类型，以下函数可作为备用函数。

ddi_io_get8()	从 I/O 空间的映射设备寄存器中读取一个 8 位数据
ddi_io_get16()	从 I/O 空间的映射设备寄存器中读取一个 16 位数据
ddi_io_get32()	从 I/O 空间的映射设备寄存器中读取一个 32 位数据
ddi_io_put8()	向 I/O 空间的映射设备寄存器中写入一个 8 位数据
ddi_io_put16()	向 I/O 空间的映射设备寄存器中写入一个 16 位数据
ddi_io_put32()	向 I/O 空间的映射设备寄存器中写入一个 32 位数据
ddi_io_rep_get8()	从 I/O 空间的映射设备寄存器中读取多个 8 位数据
ddi_io_rep_get16()	从 I/O 空间的映射设备寄存器中读取多个 16 位数据
ddi_io_rep_get32()	从 I/O 空间的映射设备寄存器中读取多个 32 位数据
ddi_io_rep_put8()	向 I/O 空间的映射设备寄存器中写入多个 8 位数据
ddi_io_rep_put16()	向 I/O 空间的映射设备寄存器中写入多个 16 位数据
ddi_io_rep_put32()	向 I/O 空间的映射设备寄存器中写入多个 32 位数据
ddi_mem_get8()	从内存空间的映射设备或 DMA 内存中读取一个 8 位数据
ddi_mem_get16()	从内存空间的映射设备或 DMA 内存中读取一个 16 位数据
ddi_mem_get32()	从内存空间的映射设备或 DMA 内存中读取一个 32 位数据
ddi_mem_get64()	从内存空间的映射设备或 DMA 内存中读取一个 64 位数据
ddi_mem_put8()	向内存空间的映射设备或 DMA 内存中写入一个 8 位数据
ddi_mem_put16()	向内存空间的映射设备或 DMA 内存中写入一个 16 位数据
ddi_mem_put32()	向内存空间的映射设备或 DMA 内存中写入一个 32 位数据

ddi_mem_put64()	向内存空间的映射设备或 DMA 内存中写入一个 64 位数据
ddi_mem_rep_get8()	从内存空间的映射设备或 DMA 内存中读取多个 8 位数据
ddi_mem_rep_get16()	从内存空间的映射设备或 DMA 内存中读取多个 16 位数据
ddi_mem_rep_get32()	从内存空间的映射设备或 DMA 内存中读取多个 32 位数据
ddi_mem_rep_get64()	从内存空间的映射设备或 DMA 内存中读取多个 64 位数据
ddi_mem_rep_put8()	向内存空间的映射设备或 DMA 内存中写入多个 8 位数据
ddi_mem_rep_put16()	向内存空间的映射设备或 DMA 内存中写入多个 16 位数据
ddi_mem_rep_put32()	向内存空间的映射设备或 DMA 内存中写入多个 32 位数据
ddi_mem_rep_put64()	向内存空间的映射设备或 DMA 内存中写入多个 64 位数据
pci_config_setup()	设置对 PCI 本地总线配置空间的访问
pci_config_teardown()	销毁对 PCI 本地总线配置空间的访问
pci_config_get8()	从 PCI 本地总线配置空间中读取一个 8 位数据
pci_config_get16()	从 PCI 本地总线配置空间中读取一个 16 位数据
pci_config_get32()	从 PCI 本地总线配置空间中读取一个 32 位数据
pci_config_get64()	从 PCI 本地总线配置空间中读取一个 64 位数据
pci_config_put8()	向 PCI 本地总线配置空间中写入一个 8 位数据
pci_config_put16()	向 PCI 本地总线配置空间中写入一个 16 位数据
pci_config_put32()	向 PCI 本地总线配置空间中写入一个 32 位数据
pci_config_put64()	向 PCI 本地总线配置空间中写入一个 64 位数据

表 B–4 过时的程控 I/O 函数

过时的函数	替代函数
ddi_getb()	ddi_get8()
ddi_getl()	ddi_get32()
ddi_getll()	ddi_get64()
ddi_getw()	ddi_get16()
ddi_io_getb()	ddi_io_get8()
ddi_io_getl()	ddi_io_get32()
ddi_io_getw()	ddi_io_get16()

表 B–4 过时的程控 I/O 函数 （续）

过时的函数	替代函数
ddi_io_putb()	ddi_io_put8()
ddi_io_putl()	ddi_io_put32()
ddi_io_putw()	ddi_io_put16()
ddi_io_rep_getb()	ddi_io_rep_get8()
ddi_io_rep_getl()	ddi_io_rep_get32()
ddi_io_rep_getw()	ddi_io_rep_get16()
ddi_io_rep_putb()	ddi_io_rep_put8()
ddi_io_rep_putl()	ddi_io_rep_put32()
ddi_io_rep_putw()	ddi_io_rep_put16()
ddi_map_regs()	ddi_regs_map_setup()
ddi_mem_getb()	ddi_mem_get8()
ddi_mem_getl()	ddi_mem_get32()
ddi_mem_getll()	ddi_mem_get64()
ddi_mem_getw()	ddi_mem_get16()
ddi_mem_putb()	ddi_mem_put8()
ddi_mem_putl()	ddi_mem_put32()
ddi_mem_putll()	ddi_mem_put64()
ddi_mem_putw()	ddi_mem_put16()
ddi_mem_rep_getb()	ddi_mem_rep_get8()
ddi_mem_rep_getl()	ddi_mem_rep_get32()
ddi_mem_rep_getll()	ddi_mem_rep_get64()
ddi_mem_rep_getw()	ddi_mem_rep_get16()
ddi_mem_rep_putb()	ddi_mem_rep_put8()
ddi_mem_rep_putl()	ddi_mem_rep_put32()
ddi_mem_rep_putll()	ddi_mem_rep_put64()
ddi_mem_rep_putw()	ddi_mem_rep_put16()
ddi_peekc()	ddi_peek8()
ddi_peekd()	ddi_peek64()

表 B–4 过时的程控 I/O 函数 （续）

过时的函数	替代函数
ddi_peekl()	ddi_peek32()
ddi_peeks()	ddi_peek16()
ddi_pokec()	ddi_poke8()
ddi_poked()	ddi_poke64()
ddi_pokel()	ddi_poke32()
ddi_pokes()	ddi_poke16()
ddi_putb()	ddi_put8()
ddi_putl()	ddi_put32()
ddi_putll()	ddi_put64()
ddi_putw()	ddi_put16()
ddi_rep_getb()	ddi_rep_get8()
ddi_rep_getl()	ddi_rep_get32()
ddi_rep_getll()	ddi_rep_get64()
ddi_rep_getw()	ddi_rep_get16()
ddi_rep_putb()	ddi_rep_put8()
ddi_rep_putl()	ddi_rep_put32()
ddi_rep_putll()	ddi_rep_put64()
ddi_rep_putw()	ddi_rep_put16()
ddi_unmap_regs()	ddi_regs_map_free()
inb()	ddi_io_get8()
inl()	ddi_io_get32()
inw()	ddi_io_get16()
outb()	ddi_io_put8()
outl()	ddi_io_put32()
outw()	ddi_io_put16()
pci_config_getb()	pci_config_get8()
pci_config_getl()	pci_config_get32()
pci_config_getll()	pci_config_get64()

表 B–4 过时的程控 I/O 函数 （续）

过时的函数	替代函数
pci_config_getw()	pci_config_get16()
pci_config_putb()	pci_config_put8()
pci_config_putl()	pci_config_put32()
pci_config_putll()	pci_config_put64()
pci_config_putw()	pci_config_put16()
repinsb()	ddi_io_rep_get8()
repinsd()	ddi_io_rep_get32()
repinsw()	ddi_io_rep_get16()
repoutsb()	ddi_io_rep_put8()
repoutsd()	ddi_io_rep_put32()
repoutsw()	ddi_io_rep_put16()

直接内存访问 (Direct Memory Access, DMA) 函数

DMA 函数包括 :

ddi_dma_alloc_handle()	分配 DMA 句柄
ddi_dma_free_handle()	释放 DMA 句柄
ddi_dma_mem_alloc()	为 DMA 传送操作分配内存
ddi_dma_mem_free()	释放以前分配的 DMA 内存
ddi_dma_addr_bind_handle()	将地址绑定到 DMA 句柄
ddi_dma_buf_bind_handle()	将系统缓冲区绑定到 DMA 句柄
ddi_dma_unbind_handle()	取消绑定 DMA 句柄中的地址
ddi_dma_nextcookie()	检索后续的 DMA cookie
ddi_dma_getwin()	激活新 DMA 窗口
ddi_dma_numwin()	检索 DMA 窗口数
ddi_dma_sync()	同步 CPU 和 I/O 内存视图
ddi_check_dma_handle()	检查 DMA 句柄
ddi_dma_set_sbus64()	允许在 S 总线上进行 64 位传送
ddi_slaveonly()	报告设备是否安装在只允许从属访问的位置

ddi_iomin()	查找 DMA 的最小对齐和传送大小
ddi_dma_burstsizes()	查找 DMA 映射的允许突发大小
ddi_dma_devalign()	查找 DMA 映射对齐和最小传送大小
ddi_dmae_alloc()	获取 DMA 通道
ddi_dmae_release()	释放 DMA 通道
ddi_dmae_getattr()	获取 DMA 引擎属性
ddi_dmae_prog()	对 DMA 通道编程
ddi_dmae_stop()	终止 DMA 引擎操作
ddi_dmae_disable()	禁用 DMA 通道
ddi_dmae_enable()	启用 DMA 通道
ddi_dmae_getcnt()	获取剩余的 DMA 引擎计数
ddi_dmae_1stparty()	配置 DMA 通道层叠模式
ddi_dma_coff()	将 DMA cookie 转换为 DMA 句柄内的偏移

表 B–5 过时的直接内存访问 (Direct Memory Access, DMA) 函数

过时的函数	替代函数
ddi_dma_addr_setup()	ddi_dma_alloc_handle()、ddi_dma_addr_bind_handle()
ddi_dma_buf_setup()	ddi_dma_alloc_handle()、ddi_dma_buf_bind_handle()
ddi_dma_curwin()	ddi_dma_getwin()
ddi_dma_free()	ddi_dma_free_handle()
ddi_dma_htoc()	ddi_dma_addr_bind_handle()、ddi_dma_buf_bind_handle()
ddi_dma_movwin()	ddi_dma_getwin()
ddi_dma_nextseg()	ddi_dma_nextcookie()
ddi_dma_segtocookie()	ddi_dma_nextcookie()
ddi_dma_setup()	ddi_dma_alloc_handle()、ddi_dma_addr_bind_handle()、ddi_dma_buf_bind_handle()
ddi_dmae_getlim()	ddi_dmae_getattr()
ddi_iopb_alloc()	ddi_dma_mem_alloc()
ddi_iopb_free()	ddi_dma_mem_free()
ddi_mem_alloc()	ddi_dma_mem_alloc()

表 B–5 过时的直接内存访问 (Direct Memory Access, DMA) 函数 （续）

过时的函数	替代函数
ddi_mem_free()	ddi_dma_mem_free()
hat_getkpfnum()	ddi_dma_addr_bind_handle()、ddi_dma_buf_bind_handle()、ddi_dma_nextcookie()

用户空间访问函数

用户空间访问函数包括：

ddi_copyin()	将数据复制到驱动程序缓冲区
ddi_copyout()	从驱动程序中复制数据
uiomove()	使用 uio 结构复制内核数据
ureadc()	向 uio 结构中添加字符
uwritec()	从 uio 结构中删除字符
getminor()	获取次要设备号
ddi_model_convert_from()	确定数据模型类型是否不匹配
IOC_CONVERT_FROM()	确定是否需要转换 M_IOCTL 内容
STRUCT_DECL()	声明并初始化指向本机形式结构实例的结构句柄
STRUCT_HANDLE()	声明并初始化指向本机形式结构实例的结构句柄
STRUCT_INIT()	声明并初始化指向本机形式结构实例的结构句柄
STRUCT_SET_HANDLE()	声明并初始化指向本机形式结构实例的结构句柄
SIZEOF_PTR()	返回指定数据模型中指针的大小
SIZEOF_STRUCT()	返回指定数据模型中结构的大小
STRUCT_SIZE()	返回应用程序数据模型中结构的大小
STRUCT_BUF()	返回指向结构的本机模式实例的指针
STRUCT_FADDR()	返回指向结构的指定字段的指针
STRUCT_FGET()	返回应用程序数据模型中结构的指定字段
STRUCT_FGETP()	返回应用程序数据模型中结构的指定指针字段
STRUCT_FSET()	设置应用程序数据模型中结构的指定字段
STRUCT_FSETP()	设置应用程序数据模型中结构的指定指针字段

表 B–6 过时的用户空间访问函数

过时的函数	替代函数
copyin()	ddi_copyin()
copyout()	ddi_copyout()
ddi_getminor()	getminor()

用户进程事件函数

用户进程事件函数包括：

pollwakeup()	通知进程事件已发生
proc_ref()	获取进程中指向信号的句柄
proc_unref()	释放进程中指向信号的句柄
proc_signal()	向进程发送信号

用户进程信息函数

用户进程信息函数包括：

ddi_get_cred()	返回指向调用方的凭证结构的指针
drv_priv()	确定进程凭证权限
ddi_get_pid()	返回进程 ID

表 B–7 过时的用户进程信息函数

过时的函数	替代函数
drv_getparm()	ddi_get_pid()、ddi_get_cred()

用户应用程序内核和设备访问函数

用户应用程序内核和设备访问函数包括：

ddi_dev_nregs()	返回设备的寄存器集数
ddi_dev_regsize()	返回设备寄存器的大小
ddi_devmap_segmap()、devmap_setup()	使用 devmap 框架设置用户与设备内存之间的映射

devmap_devmem_setup()	将设备内存导出到用户空间
devmap_load()	验证内存地址转换
devmap_unload()	使内存地址转换无效
devmap_do_ctxmgt()	对映射执行设备上下文切换
devmap_set_ctx_timeout()	为上下文管理回叫设置超时值
devmap_default_access()	缺省驱动程序内存访问函数
ddi_umem_alloc()	分配按页对齐的内核内存
ddi_umem_free()	释放按页对齐的内核内存
ddi_umem_lock()	锁定内存页
ddi_umem_unlock()	解除锁定内存页
ddi_umem_iosetup()	设置对应用程序内存的 I/O 请求
devmap_umem_setup()	将内核内存导出到用户空间
ddi_model_convert_from()	确定数据模型类型是否不匹配

表 B–8 过时的用户应用程序内核和设备访问函数

过时的函数	替代函数
ddi_mapdev()	devmap_setup()
ddi_mapdev_intercept()	devmap_load()
ddi_mapdev_nointercept()	devmap_unload()
ddi_mapdev_set_device_acc_attr()	devmap()
ddi_segmap()	devmap()
ddi_segmap_setup()	devmap_setup()
hat_getkpfnum()	devmap()
ddi_mmap_get_model()	devmap()

与时间有关的函数

与时间有关的函数包括：

ddi_get_lbolt()	返回自重新引导以来的时钟周期数
ddi_get_time()	返回当前时间（以秒为单位）
ddi_periodic_add()	以纳秒为周期发出超时请求

ddi_periodic_delete()	取消以纳秒为周期发出超时请求
delay()	使执行延迟指定的时钟周期数
drv_hztousec()	将时钟周期转换为微秒
drv_usectohz()	将微秒转换为时钟周期
drv_usecwait()	繁忙－等待指定的时间间隔
gethrtime()	获取高分辨率时间
gethrvtime()	获取高分辨率 LWP 虚拟时间
timeout()	在指定的时间长度后执行函数
untimeout()	取消以前的超时函数调用
drv_getparm()	ddi_get_lbolt()、ddi_get_time()

表 B–9 过时的与时间有关的函数

过时的函数	替代函数
drv_getparm()	ddi_get_lbolt()、ddi_get_time()

电源管理函数

电源管理函数包括：

ddi_removing_power()	使用 DDI_SUSPEND 检查设备是否断电
pci_report_pmcap()	报告 PCI 设备的电源管理功能
pm_busy_component()	将组件标记为繁忙
pm_idle_component()	将组件标记为空闲
pm_raise_power()	提高组件的电源级别
pm_lower_power()	降低组件的电源级别
pm_power_has_changed()	向电源管理框架通知有关自治电源级别的更改信息
pm_trans_check()	设备电源开关建议检查

表 B–10 过时的电源管理函数

函数名	说明
ddi_dev_is_needed()	通知系统需要某一设备组件
pm_create_components()	创建可管理电源的组件

表 B–10 过时的电源管理函数 （续）

函数名	说明
pm_destroy_components()	销毁可管理电源的组件
pm_get_normal_power()	获取设备组件的正常电源级别
pm_set_normal_power()	设置设备组件的正常电源级别

故障管理函数

故障管理函数包括：

ddi_fm_init()	根据声明的故障管理功能分配和初始化资源
ddi_fm_fini()	清除为该设备实例分配的资源，以支持声明为 ddi_fm_init() 的故障管理功能
ddi_fm_capable()	返回当前为该设备实例设置的功能位掩码
ddi_fm_handler_register()	在 IO 故障管理框架中注册错误处理程序回调例程
ddi_fm_handler_unregister()	删除使用 ddi_fm_handler_register() 注册的错误处理程序回调例程
ddi_fm_acc_err_get()	返回访问句柄的错误状态
ddi_fm_dma_err_get()	返回 DMA 句柄的错误状态
ddi_fm_acc_err_clear()	清除访问句柄的错误状态
ddi_fm_dma_err_clear()	清除 DMA 句柄的错误状态
ddi_fm_ereport_post()	将编码的故障管理错误报告名称-值对列表排入队列，以传送到 Fault Manager 守护进程 fmd(1M) 中
ddi_fm_service_impact()	报告错误的影响
pci_ereport_setup()	初始化错误报告生成支持，并设置对 PCI、PCI/X 或 PCI Express 配置空间进行后续访问所用的资源
pci_ereport_teardown()	释放 pci_ereport_setup() 为该设备实例分配和设置的所有资源
pci_ereport_post()	扫描和发布任何 PCI、PCI/X 或 PCI Express 总线错误

内核统计信息函数

内核统计信息 (kstat) 函数包括 :

`kstat_create()`	创建并初始化新的 kstat
`kstat_delete()`	从系统中删除 kstat
`kstat_install()`	向系统中添加完全初始化的 kstat
`kstat_named_init()`	初始化已命名的 kstat
`kstat_runq_back_to_waitq()`	记录从运行队列到等待队列的事务迁移
`kstat_runq_enter()`	记录向运行队列中添加的事务
`kstat_runq_exit()`	记录从运行队列中移除的事务
`kstat_waitq_enter()`	记录向等待队列中添加的事务
`kstat_waitq_exit()`	记录从等待队列中移除的事务
`kstat_waitq_to_runq()`	记录从等待队列到运行队列的事务迁移

内核日志记录和列显函数

内核日志记录和列显函数包括 :

`cmn_err()`、`vcmn_err()`	显示错误消息
`ddi_report_dev()`	通知设备
`strlog()`	将消息提交至日志驱动程序
`ddi_dev_report_fault()`	报告硬件故障
`scsi_errmsg()`	显示 SCSI 请求检测消息
`scsi_log()`	显示与 SCSI 设备有关的消息
`scsi_vu_errmsg()`	显示 SCSI 请求检测消息

缓存 I/O 函数

缓存 I/O 函数包括 :

`physio()`	执行物理 I/O
`aphysio()`	执行异步物理 I/O
`anocancel()`	禁止取消异步 I/O 请求

`minphys()`	限制 `physio()` 缓冲区大小
`biowait()`	暂停以待处理方式完成块 I/O 的进程
`biodone()`	在完成缓冲区 I/O 传送后释放缓冲区并通知阻塞的线程
`bioerror()`	指示缓冲区头中的错误
`geterror()`	返回 I/O 错误
`bp_mapin()`	分配虚拟地址空间
`bp_mapout()`	取消分配虚拟地址空间
`disksort()`	使用单向电梯查找策略对缓冲区排序
`getrbuf()`	获取原始缓冲区头
`freerbuf()`	释放原始缓冲区头
`biosize()`	返回缓冲区结构的大小
`bioinit()`	初始化缓冲区结构
`biofini()`	取消初始化缓冲区结构
`bioreset()`	在 I/O 完成后重用专用的缓冲区头
`bioclone()`	克隆另一个缓冲区
`biomodified()`	检查缓冲区是否已修改
`clrbuf()`	删除缓冲区的内容

虚拟内存函数

虚拟内存函数包括：

`ddi_btop()`	将设备字节转换为页（向下舍入）
`ddi_btopr()`	将设备字节转换为页（向上舍入）
`ddi_ptob()`	将设备页转换为字节
`btop()`	将以字节表示的大小转换为以页表示的大小（向下舍入）
`btopr()`	将以字节表示的大小转换为以页表示的大小（向上舍入）
`ptob()`	将以页表示的大小转换为以字节表示的大小

表 B–11 过时的虚拟内存函数

过时的函数	替代函数
hat_getkpfnum()	devmap()、ddi_dma_*_bind_handle()、ddi_dma_nextcookie()

设备 ID 函数

设备 ID 函数包括：

ddi_devid_init()	分配设备 ID 结构
ddi_devid_free()	释放设备 ID 结构
ddi_devid_register()	注册设备 ID
ddi_devid_unregister()	注销设备 ID
ddi_devid_compare()	比较两个设备 ID
ddi_devid_sizeof()	返回设备 ID 的大小
ddi_devid_valid()	验证设备 ID
ddi_devid_str_encode()	将设备 ID 和 minor_name 编码为以 null 结尾的 ASCII 字符串，返回指向该字符串的指针
ddi_devid_str_decode()	从以前编码的字符串中解码设备 ID 和 minor_name，分配并返回指向提取部分的指针
ddi_devid_str_free()	释放 ddi_devid_* 函数返回的所有字符串

SCSI 函数

SCSI 函数包括：

scsi_probe()	探测 SCSI 设备
scsi_unprobe()	释放在初始探测期间分配的资源
scsi_alloc_consistent_buf()	为 SCSI DMA 分配 I/O 缓冲区
scsi_free_consistent_buf()	释放以前分配的 SCSI DMA I/O 缓冲区
scsi_init_pkt()	准备完整的 SCSI 包
scsi_destroy_pkt()	释放已分配的 SCSI 包及其 DMA 资源
scsi_setup_cdb()	设置 SCSI 命令描述符块 (command descriptor block, CDB)

`scsi_transport()`	启动 SCSI 命令
`scsi_poll()`	运行轮询 SCSI 命令
`scsi_ifgetcap()`	获取 SCSI 传输功能
`scsi_ifsetcap()`	设置 SCSI 传输功能
`scsi_sync_pkt()`	同步 CPU 和 I/O 内存视图
`scsi_abort()`	中止 SCSI 命令
`scsi_reset()`	重置 SCSI 总线或目标
`scsi_reset_notify()`	向目标驱动程序通知总线重置
`scsi_cname()`	解码 SCSI 命令
`scsi_dname()`	解码 SCSI 外围设备类型
`scsi_mname()`	解码 SCSI 消息
`scsi_rname()`	解码 SCSI 包完成原因
`scsi_sname()`	解码 SCSI 感知密钥
`scsi_errmsg()`	显示 SCSI 请求检测消息
`scsi_log()`	显示与 SCSI 设备有关的消息
`scsi_vu_errmsg()`	显示 SCSI 请求检测消息
`scsi_hba_init()`	SCSI HBA 系统初始化例程
`scsi_hba_fini()`	SCSI HBA 系统完成例程
`scsi_hba_attach_setup()`	SCSI HBA 连接例程
`scsi_hba_detach()`	SCSI HBA 分离例程
`scsi_hba_probe()`	缺省 SCSI HBA 探测函数
`scsi_hba_tran_alloc()`	分配传输结构
`scsi_hba_tran_free()`	释放传输结构
`scsi_hba_pkt_alloc()`	分配 `scsi_pkt` 结构
`scsi_hba_pkt_free()`	释放 `scsi_pkt` 结构
`scsi_hba_lookup_capstr()`	返回索引匹配功能字符串

表 B–12 过时的 SCSI 函数

过时的函数	替代函数
free_pktiopb()	scsi_free_consistent_buf()
get_pktiopbs()	scsi_alloc_consistent_buf()
makecom_g0()	scsi_setup_cdb()
makecom_g0_s()	scsi_setup_cdb()
makecom_g1()	scsi_setup_cdb()
makecom_g5()	scsi_setup_cdb()
scsi_dmafree()	scsi_destroy_pkt()
scsi_dmaget()	scsi_init_pkt()
scsi_hba_attach()	scsi_hba_attach_setup()
scsi_pktalloc()	scsi_init_pkt()
scsi_pktfree()	scsi_destroy_pkt()
scsi_resalloc()	scsi_init_pkt()
scsi_resfree()	scsi_destroy_pkt()
scsi_slave()	scsi_probe()
scsi_unslave()	scsi_unprobe()

资源映射管理函数

资源映射管理函数包括：

rmallocmap()	分配资源映射
rmallocmap_wait()	分配资源映射，必要时等待
rmfreemap()	释放资源映射
rmalloc()	从资源映射中分配空间
rmalloc_wait()	从资源映射中分配空间，必要时等待
rmfree()	将空间重新释放到资源映射中

系统全局状态

`ddi_in_panic()`	确定系统是否处于紧急状态

实用程序函数

以下列表提供了可用的实用程序函数的一个子集。

`nulldev()`	零返回函数
`nodev()`	错误返回函数
`nochpoll()`	不可轮询设备的错误返回函数
`ASSERT()`	表达式验证
`bcopy()`	在内核的地址位置之间复制数据
`bzero()`	清除给定字节数的内存
`bcmp()`	比较两个字节数组
`ddi_ffs()`	查找长整数中设置的第一位
`ddi_fls()`	查找长整数中设置的最后一位
`swab()`	以 16 位半字交换字节
`strcat()`	附加两个字符串
`strncat()`	附加两个字符串，有长度限制
`strlcat()`	附加两个字符串，有长度和缓冲区限制
`strcmp()`	比较两个以 null 结尾的字符串
`strncmp()`	比较两个以 null 结尾的字符串，长度有限制
`strlen()`	确定字符串中的非空字节数
`strnlen()`	确定字符串中的非空字节数，有长度限制
`strcpy()`	将字符串从一个位置复制到另一个位置
`strncpy()`	将字符串从一个位置复制到另一个位置，长度有限制
`strlcpy()`	将字符串从一个位置复制到另一个位置（有长度和缓冲区限制）
`strcasecmp()`	`strcmp()` 的不区分大小写的版本
`strncasecmp()`	`strncmp()` 的不区分大小写的版本

`strchr()`	在字符串中查找字符
`strstr()`	定位一个字符串在另一个字符串中第一次出现的位置
`strcasestr()`	`strstr()` 的不区分大小写的版本
`strfree()`	释放与字符串关联的内存
`sprintf()`、`vsprintf()`	格式化内存中的字符
`snprintf()`	以指定的缓冲区大小格式化内存中的字符
`numtos()`	将整数转换为十进制字符串
`stoi()`	将十进制字符串转换为整数
`max()`	返回两个整数中的较大值
`min()`	返回两个整数中的较小值
`va_arg()`	查找变量参数列表中的下一个值
`va_copy()`	复制变量参数列表的状态
`va_end()`	删除指向变量参数列表的指针
`va_start()`	查找指向变量参数列表开头的指针

有关实用程序函数的详细信息，请参见《man pages section 9: DDI and DKI Kernel Functions》。

◆ ◆ ◆ 附 录 C

使设备驱动程序支持 64 位

本附录为要将设备驱动程序转换为支持 64 位内核的设备驱动程序编写人员提供信息。本附录还介绍了 32 位设备驱动程序和 64 位设备驱动程序之间的区别，并说明了将 32 位设备驱动程序转换为 64 位设备驱动程序的步骤。这些信息仅适用于常规字符设备驱动程序和块设备驱动程序。

本附录提供有关以下主题的信息：

64 位驱动程序设计简介

对于仅需支持 32 位内核的驱动程序，现有 32 位设备驱动程序将仍然有效，无需重新编译。但是，大多数设备驱动程序需要进行一些更改才能在 64 位内核中正确运行，且所有设备驱动程序都需要重新编译以创建 64 位驱动程序模块。本附录中的信息旨在指导您利用通用源代码来生成 32 位和 64 位环境的驱动程序，从而提高代码可移植性并降低维护工作量。

开始修改设备驱动程序以便使用 64 位环境之前，应了解 32 位环境与 64 位环境之间的区别。特别是必须熟悉 C 语言数据类型模型 ILP32 和 LP64。请参见下表。

表 C–1 ILP32 与 LP64 数据类型对比

C 类型	ILP32	LP64
`char`	8	8
`short`	16	16
`int`	32	32
`long`	32	64

表 C–1 ILP32 与 LP64 数据类型对比 （续）

C 类型	ILP32	LP64
long long	64	64
float	32	32
double	64	64
long double	96	128
pointer	32	64

因 ILP32 与 LP64 之间的差异而导致的特定于驱动程序的问题是本附录的主题。

除了清理常规代码以支持 LP64 的数据模型更改，驱动程序编写人员还必须提供对 32 位和 64 位应用程序的支持。

ioctl(9E)、devmap(9E) 和 mmap(9E) 入口点使应用程序和设备驱动程序之间可直接共享数据结构。如果这些数据结构在 32 位环境与 64 位环境中的大小不同，则必须修改入口点，以便驱动程序可确定应用程序的数据模型与内核的数据模型是否相同。如果数据模型不同，则可对数据结构进行调整。请参见第 263 页中的 “对有 64 位处理能力的设备驱动程序的 I/O 控制支持”、第 265 页中的 “32 位和 64 位数据结构宏”和第 172 页中的 “将内核内存与用户映射相关联”。

在许多驱动程序中，只有少量 ioctl 需要这种处理。其他 ioctl 无需更改即可应用，只要这些 ioctl 传递的数据结构大小不变。

常规转换步骤

以下各节提供了有关转换驱动程序以在 64 位环境中运行的信息。驱动程序编写人员可能需要执行以下一项或多项任务：

1. 使用硬件寄存器的固定宽度类型。
2. 使用固定宽度的公共访问函数。
3. 检查并扩展派生类型的用法。
4. 检查 DDI 数据结构中更改的字段。
5. 检查 DDI 函数中更改的参数。
6. 根据需要修改用于处理用户数据的驱动程序入口点。
7. 检查 x86 平台上使用 64 位 long 类型的结构。

下面详细说明这些步骤。

完成每一步骤后，请修复所有编译器警告，然后使用 lint 查找其他问题。对于 SC5.0（或更高）版本的 lint，要想找出 64 位问题，在使用该命令时必须指定 -Xarch=v9 和 -errchk=longptr64 选项。

注 – 请勿忽略 LP64 转换期间出现的编译警告。以前在 ILP32 环境中可安全忽略的警告现在可能表示比较严重的问题。

完成所有步骤后，同时将驱动程序作为 32 位和 64 位模块进行编译和测试。

使用硬件寄存器的固定宽度类型

许多处理硬件设备的设备驱动程序使用 C 数据结构说明硬件的布局。在 LP64 数据模型中，使用 `long` 或 unsigned long 类型定义硬件寄存器的数据结构几乎肯定不正确，因为 `long` 类型现在是 64 位。首先包括 `<sys/inttypes.h>`，然后将此类数据结构更新为使用 `int32_t` 或 `uint32_t`，而不是 32 位设备数据的 `long`。此方法可保留 32 位数据结构的二进制布局。例如，将以下代码：

```
struct device_regs {
    ulong_t        addr;
    uint_t         count;
};      /* Only works for ILP32 compilation */
```

更改为：

```
struct device_regs {
    uint32_t        addr;
    uint32_t        count;
};      /* Works for any data model */
```

使用固定宽度的公共访问函数

Oracle Solaris DDI 允许通过访问函数访问设备寄存器，以便可在多个平台间移植。DDI 公共访问函数以前以字节和字等单位指定数据大小。例如，`ddi_getl`(9F) 用于访问 32 位。此函数不存在于 64 位 DDI 环境中，且已被替换为可指定要处理的位数的函数版本。

这些例程已添加到 Solaris 2.6 操作环境的 32 位内核中，以允许驱动程序编写人员采用其早期版本。例如，要移植到 32 位和 64 位内核中，驱动程序必须使用 `ddi_get32`(9F) 而不是 `ddi_getl`(9F) 来访问 32 位数据。

所有公共访问例程都被替换为其固定宽度的对等例程。有关详细信息，请参见 `ddi_get8`(9F)、`ddi_put8`(9F)、`ddi_rep_get8`(9F) 和 `ddi_rep_put8`(9F) 手册页。

检查并扩展派生类型的用法

应尽可能使用系统派生的类型（如 `size_t`），以使产生的变量在各种函数间传递时都有效。而新的派生类型 `uintptr_t` 或 `intptr_t` 是整数类型，应该用于指针。

固定宽度的整数类型用于表示二进制数据结构或硬件寄存器的显式大小，而基础 C 语言数据类型（如 int）仍然可用于循环计数器或文件描述符。

一些系统派生类型在 32 位系统上表示 32 位，但是在 64 位系统上表示 64 位。以此方式更改大小的派生类型包括：clock_t、daddr_t、dev_t、ino_t、intptr_t、off_t、size_t、ssize_t、time_t、uintptr_t 和 timeout_id_t。

设计使用这些派生类型的驱动程序时，请特别注意这些类型的用法，尤其是驱动程序将这些值指定给其他类型（如固定宽度类型）的变量时。

检查 DDI 数据结构中更改的字段

DDI 数据结构中某些字段的数据类型（如 buf(9S)）已被更改。使用这些数据结构的驱动程序应确保正确使用这些字段。下面列出了变动很大的数据结构及字段。

buf 结构更改

以下列出的字段与传输大小（现在可超过 4 GB）有关。

```
size_t        b_bcount;          /* was type unsigned int */
size_t        b_resid;           /* was type unsigned int */
size_t        b_bufsize;         /* was type long */
```

ddi_dma_attr

ddi_dma_attr(9S) 结构定义 DMA 引擎和设备的属性。因为这些属性指定寄存器大小，所以使用了固定宽度的数据类型而不是基本类型。

ddi_dma_cookie 结构更改

```
uint32_t     dmac_address;    /* was type unsigned long */
size_t       dmac_size;       /* was type u_int */
```

ddi_dma_cookie(9S) 结构包含 32 位 DMA 地址，因此使用了固定宽度的数据类型来定义该地址。其大小已重新定义为 size_t。

csi_arq_status 结构更改

```
uint_t    sts_rqpkt_state;         /* was type u_long */
uint_t    sts_rqpkt_statistics;    /* was type u_long */
```

此结构中的这些字段无需增大，已重新定义为 32 位。

scsi_pkt 结构更改

```
uint_t      pkt_flags;           /* was type u_long */
int     pkt_time;        /* was type long */
ssize_t     pkt_resid;           /* was type long */
```

```
uint_t      pkt_state;          /* was type u_long */
uint_t      pkt_statistics;     /* was type u_long */
```

由于 scsi_pkt(9S) 结构中的 pkt_flags、pkt_state 和 pkt_statistics 字段无需增大，因此这些字段已重新定义为 32 位整数。数据传输大小 pkt_resid 字段**需要**增大，已重新定义为 ssize_t。

检查 DDI 函数中更改的参数

本节介绍已更改的 DDI 函数参数数据类型。

getrbuf() 参数更改

```
struct buf *getrbuf(int sleepflag);
```

在以前的发行版中，sleepflag 被定义为 long 类型。

drv_getparm() 参数更改

```
int drv_getparm(unsigned int parm, void *value_p);
```

在以前的发行版中，value_p 被定义为 unsigned long 类型。在 64 位内核中，drv_getparm(9F) 可提取 32 位和 64 位。此接口未定义这些量的数据类型，可能会发生简单的编程错误。

以下新例程提供更安全的替代方法：

```
clock_t       ddi_get_lbolt(void);
time_t        ddi_get_time(void);
cred_t        *ddi_get_cred(void);
pid_t         ddi_get_pid(void);
```

强烈要求驱动程序编写人员使用这些例程而不要使用 drv_getparm(9F)。

delay() 和 timeout() 参数更改

```
void delay(clock_t ticks);
timeout_id_t timeout(void (*func)(void *), void *arg, clock_t ticks);
```

delay(9F) 和 timeout(9F) 例程的 *ticks* 参数已从 long更改为 clock_t。

rmallocmap() 和 rmallocmap_wait() 参数更改

```
struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
```

rmallocmap(9F) 和 rmallocmap_wait(9F) 例程的 mapsize 参数已从 ulong_t 更改为 size_t。

scsi_alloc_consistent_buf() 参数更改

```
struct buf *scsi_alloc_consistent_buf(struct scsi_address *ap,
    struct buf *bp, size_t datalen, uint_t bflags,
    int (*callback )(caddr_t), caddr_t arg);
```

在以前的发行版中，datalen 被定义为 int 类型，bflags 被定义为 ulong 类型。

uiomove() 参数更改

```
int uiomove(caddr_t address, size_t nbytes,
    enum uio_rw rwflag, uio_t *uio_p);
```

nbytes 参数被定义为 long 类型，但是由于 nbytes 以字节为单位表示大小，因此 size_t 更适合。

cv_timedwait() 和 cv_timedwait_sig() 参数更改

```
int cv_timedwait(kcondvar_t *cvp, kmutex_t *mp, clock_t timeout);
int cv_timedwait_sig(kcondvar_t *cvp, kmutex_t *mp,    clock_t timeout);
```

在以前的发行版中，cv_timedwait(9F) 和 cv_timedwait_sig(9F) 例程的 *timeout* 参数被定义为 long 类型。由于这些例程表示时间周期，因此 clock_t 更适合。

ddi_device_copy() 参数更改

```
int ddi_device_copy(ddi_acc_handle_t src_handle,
    caddr_t src_addr, ssize_t src_advcnt,
    ddi_acc_handle_t dest_handle, caddr_t dest_addr,
    ssize_t dest_advcnt, size_t bytecount, uint_t dev_datasz);
```

src_advcnt、*dest_advcnt*、*dev_datasz* 参数的类型已更改。这些参数以前分别被定义为 long、long 和 ulong_t 类型。

ddi_device_zero() 参数更改

```
int ddi_device_zero(ddi_acc_handle_t handle,
    caddr_t dev_addr, size_t bytecount, ssize_t dev_advcnt,
    uint_t dev_datasz):
```

在以前的发行版中，*dev_advcnt* 被定义为 long 类型，*dev_datasz* 被定义为 ulong_t 类型。

ddi_dma_mem_alloc() 参数更改

```
int ddi_dma_mem_alloc(ddi_dma_handle_t handle,
    size_t length, ddi_device_acc_attr_t *accattrp,
    uint_t flags, int (*waitfp)(caddr_t), caddr_t arg,
    caddr_t *kaddrp, size_t *real_length,
    ddi_acc_handle_t *handlep);
```

在以前的发行版中，*length*、*flags* 和 *real_length* 分别被定义为 uint_t、ulong_t 和 uint_t * 类型。

修改处理数据共享的例程

如果设备驱动程序使用 ioctl(9E)、devmap(9E) 或 mmap(9E) 与 32 位应用程序共享包含 long 或指针类型的数据结构，且驱动程序已针对 64 位内核进行了重新编译，则数据结构的二进制布局将不兼容。如果当前已按 long 类型定义了字段，并且未使用 64 位数据项，请更改数据结构，以使用仍为 32 位的数据类型（int 和 unsigned int）。否则，驱动程序需要识别 ILP32 和 LP64 的不同结构形式，并确定应用程序与内核之间是否出现模型不匹配。

要处理潜在的数据模型差异，需要写入可直接与用户应用程序交互的 ioctl()、devmap() 和 mmap() 驱动程序入口点，以确定参数是否来自与内核使用相同的数据模型的应用程序。

ioctl() 中的数据共享

要确定应用程序与驱动程序之间是否存在模型不匹配，驱动程序可使用 FMODELS 掩码确定 ioctl() *mode* 参数的模型类型。在 mode 中采用以下值之一来标识应用程序的数据模型：

- FLP64－应用程序使用 LP64 数据模型
- FILP32－应用程序使用 ILP32 数据模型

第 263 页中的 “对有 64 位处理能力的设备驱动程序的 I/O 控制支持”中的代码示例说明如何使用 ddi_model_convert_from(9F) 处理此情况。

devmap() 中的数据共享

要使 64 位驱动程序和 32 位应用程序共享内存，64 位驱动程序生成的二进制布局必须与 32 位应用程序使用的布局相同。要导出到应用程序的映射内存可能需要包含与数据模型有关的数据结构。

很少内存映射设备会面临此问题，因为在内核数据模型发生变化时设备寄存器不会改变大小。但是，一些将映射导出到用户地址空间的伪设备可能要将不同数据结构导出到 ILP32 或 LP64 应用程序。要确定是否出现了数据模型不匹配，devmap(9E) 可使用 *model* 参数说明应用程序期望的数据模型。将 *model* 参数设置为以下值之一：

- DDI_MODEL_ILP32－应用程序使用 ILP32 数据模型
- DDI_MODEL_LP64－应用程序使用 LP64 数据模型

可将未经转换的模型参数传递到 ddi_model_convert_from(9F) 例程或 STRUCT_INIT()。请参见第 265 页中的 “32 位和 64 位数据结构宏”。

mmap() 中的数据共享

由于 mmap(9E) 没有可用于传递数据模型信息的参数，因此可编写驱动程序的 mmap(9E) 入口点，以使用新的 DDI 函数 ddi_model_convert_from(9F)。此函数返回以下值之一，以指示应用程序的数据类型模型：

- DDI_MODEL_ILP32－应用程序要求 ILP32 数据模型
- DDI_MODEL_ILP64－应用程序要求 LP64 数据模型
- DDI_FAILURE－未从 mmap(9E) 调用函数

与 ioctl() 和 devmap() 一样，可将模型位传递到 ddi_model_convert_from(9F) 以确定是否需要进行数据转换，或可将模型传递到 STRUCT_INIT()。

或者，迁移设备驱动程序以支持 devmap(9E) 入口点。

检查 x86 平台上 64 位 Long 数据类型的结构

您应认真检查 x86 平台上使用 64 位 long 类型（如 uint64_t）的结构。32 位模式编译与 64 位模式编译的对齐方式和大小可能不同。请参考以下示例。

```
#include &lt;studio>
#include &ltsys>

struct myTestStructure {
        uint32_t        my1stInteger;
        uint64_t        my2ndInteger;
};

main()
{
        struct myTestStructure a;

        printf("sizeof myTestStructure is: %d\n", sizeof(a));
        printf("offset to my2ndInteger is: %d\n", (uintptr_t)&a.bar - (uintptr_t)&a);
}
```

在 32 位系统中，该示例显示以下结果：

```
sizeof myTestStructure is: 12
offset to my2ndInteger is: 4
```

而在 64 位系统中，该示例显示以下结果：

```
sizeof myTestStructure is: 16
offset to my2ndInteger is: 8
```

因此，32 位应用程序与 64 位应用程序对结构的理解不同。这样，尝试在 32 位和 64 位两种环境中使用同一结构可能会导致问题。这种情况经常发生，尤其是在通过 ioctl() 调用将结构传入或传出内核的情况下。

已知的 ioctl 接口

许多 ioctl(9E) 操作对一类设备驱动程序通用。例如，大多数磁盘驱动程序实现 dkio(7I) 系列的众多 ioctls。这些接口中有许多将数据结构复制到内核中，或从内核中复制出数据结构，在 LP64 数据模型中这些数据结构的一部分已更改了大小。下节列出了对于 dkio、fbio(7I)、cdio(7I) 和 mtio(7I) 系列的 ioctls，现在需要在 64 位驱动程序 ioctl 例程中进行显式转换的 ioctls。

ioctl 命令	受影响的数据结构	参考
DKIOCGAPART	dk_map	dkio(7I)
DKIOCSAPART	dk_allmap	
DKIOGVTOC	partition	dkio(7I)
DKIOSVTOC	vtoc	
FBIOPUTCMAP	fbcmap	fbio(7I)
FBIOGETCMAP		
FBIOPUTCMAPI	fbcmap_i	fbio(7I)
FBIOGETCMAPI		
FBIOCCURSOR	fbcursor	fbio(7I)
FBIOSCURSOR		
CDROMREADMODE1	cdrom_read	cdio(7I)
CDROMREADMODE2		
CDROMCDDA	cdrom_cdda	cdio(7I)
CDROMCDXA	cdrom_cdxa	cdio(7I)
CDROMSUBCODE	cdrom_subcode	cdio(7I)
MTIOCTOP	mtop	mtio(7I)
MTIOCGET	mtget	mtio(7I)
MTIOCGETDRIVETYPE	mtdrivetype_request	mtio(7I)
USCSICMD	uscsi_cmd	scsi_free_consistent_buf(9F)

设备大小

nblocks 属性按块设备驱动程序的每一分片导出。此属性包含 512 字节块的数量，设备的每一分片都支持这些块。nblocks 属性被定义为带符号的 32 位量，这就将分片的最大大小限制为 1 TB。

每个磁盘提供 1 TB 以上存储空间的磁盘设备必须定义 Nblocks 属性，该属性仍应包含设备可支持的 512 字节块的数量。但是，Nblocks 是带符号的 64 位量，它消除了对磁盘空间的任何实际限制。

nblocks 属性现在已过时。所有磁盘设备都应提供 Nblocks 属性。

◆ ◆ ◆ 附录 D

控制台帧缓存器驱动程序

用于系统控制台的帧缓存器驱动程序必须提供相应的接口，以使系统能够在控制台上显示文本。Oracle Solaris OS 可提供增强的可视化 I/O 接口，以使内核终端仿真器能够直接在控制台帧缓存器上显示文本。本附录介绍如何为帧缓存器驱动程序添加必要的接口，以使该驱动程序能够与 Oracle Solaris 内核终端仿真器进行交互。

Oracle Solaris 控制台和内核终端仿真器

内核终端仿真器的作用是按照帧缓存器的屏幕高度、宽度和像素深度模式确定的正确位置和表示法在控制台帧缓存器中呈现文本。终端仿真器还可以驱动滚动、控制软件光标，以及解释 ANSI 终端转义序列。终端仿真器以 VGA 文本模式或像素模式访问控制台帧缓存器，具体取决于图形卡。要将您的帧缓存器驱动程序用作 Oracle Solaris 控制台帧缓存器驱动程序，它必须与 Oracle Solaris 内核终端仿真器兼容。目标平台是最重要的因素，它决定了您是否需要修改帧缓存器驱动程序，以使您的驱动程序与 Oracle Solaris 内核终端仿真器兼容。

- x86 平台－不需要修改控制台帧缓存器驱动程序，因为 x86 控制台帧缓存器驱动程序已经支持控制台帧缓存器接口。
- SPARC 平台－控制台帧缓存器驱动程序应使用本附录中所述的接口，以使驱动程序能够与 Oracle Solaris 内核终端仿真器进行交互。

x86 平台控制台通信

在 x86 平台上，Oracle Solaris 内核终端仿真器模块 (terminal emulator module, `tem`) 以独占方式使用 VGA 文本模式与 `vgatext` 模块进行交互。`vgatext` 模块使用行业标准 VGA 文本模式与 x86 兼容的帧缓存器设备进行交互。由于 `vgatext` 模块已经支持控制台帧缓存器接口，因此 x86 帧缓存器驱动程序与内核 `tem` 模块兼容。不需要向 x86 帧缓存器驱动程序添加任何特殊的接口。

本附录的其余部分仅适用于 SPARC 平台。

SPARC 平台控制台通信

SPARC 帧缓存器驱动程序通常不在 VGA 文本模式下运行。SPARC 帧缓存器驱动程序通常需要发送像素图案，以描述显示的文本和图像。内核 `tem` 要求 SPARC 驱动程序支持特定的接口，以便在屏幕上呈现数据、执行滚动和显示文本光标。驱动程序实际上如何在屏幕上呈现 `tem` 发出的数据取决于具体的设备。驱动程序通常根据硬件和视频模式在视频内存中绘制数据。

Oracle Solaris OS 提供的一些接口使内核终端仿真器能够直接驱动兼容的控制台帧缓存器。将驱动程序转换为与内核终端仿真器兼容的好处在于：

- 极大地改善性能，尤其是滚动性能
- 增强 ANSI 文本颜色功能
- 能够在控制台帧缓存器上启动登录会话，即使当系统控制台流定向到串行端口以外时也是如此

SPARC 控制台帧缓存器驱动程序不需要与内核终端仿真器兼容。如果控制台帧缓存器驱动程序不与内核终端仿真器兼容，系统将使用 OpenBoot PROM 中的 FCode 终端仿真器。

控制台帧缓存器通过 EEPROM `screen` 环境变量进行识别。系统通过检查帧缓存器驱动程序是否导出 `tem-support` DDI 属性，来确定控制台帧缓存器是否与内核终端仿真器模块兼容。如果导出了 `tem-support` 属性，则系统将在系统引导过程中配置控制台时对帧缓存器驱动程序发出 `VIS_DEVINIT` I/O 控制 (`ioctl`) 命令。如果导出了 `tem-support` DDI 属性，同时 `VIS_DEVINIT` `ioctl` 命令成功并向 `tem` 返回了兼容版本号，那么，系统会将系统控制台配置为通过内核终端仿真器利用该帧缓存器驱动程序。有关 I/O 控制驱动程序入口点的信息，请参见 `ioctl(9E)` 手册页。

支持内核终端仿真器的 SPARC 驱动程序应导出 `tem-support` DDI 属性。该属性表示驱动程序支持内核终端仿真器。如果帧缓存器驱动程序导出了 `tem-support` DDI 属性，则早在引导过程中配置控制台时就将会处理该驱动程序。如果帧缓存器驱动程序未导出 `tem-support` 属性，则在引导过程中，可能不会那么早就处理该驱动程序。

`tem-support` 设置为 1 时，此 DDI 属性表示此驱动程序与控制台内核帧缓存器接口兼容。

内核终端仿真器模块通过两种主要接口与控制台帧缓存器驱动程序进行交互：

- 通过 `ioctl` 接口（正常的系统操作期间）
- 通过轮询式 I/O 接口（独立模式期间）

下节将提供详细信息。

控制台可视化 I/O 接口

内核终端仿真器通过两个接口与控制台帧缓存器驱动程序进行交互。在正常的系统活动期间（系统成功引导后），内核终端仿真器与控制台帧缓存器驱动程序之间的通信通过 ioctl 接口进行。在独立模式期间（系统引导之前或调试期间），内核终端仿真器与控制台帧缓存器驱动程序之间的通信通过轮询式 I/O 接口进行。内核终端仿真器与控制台帧缓存器驱动程序之间的所有活动都由内核终端仿真器启动，但控制台帧缓存器驱动程序用来通知内核终端仿真器有关视频模式方面的变化的回调函数除外。

visual_io(7I) 手册页中详细说明了控制台可视化 I/O 接口。有关视频模式更改回调函数的更多信息，请参见第 558 页中的 “视频模式更改回调接口”。

I/O 控制接口

在正常的系统活动期间，内核终端仿真器通过下表中列出的 ioctl 接口与控制台帧缓存器驱动程序进行通信：

ioctl 名称	对应的数据结构	说明
VIS_DEVINIT	vis_devinit	初始化终端仿真器模块与帧缓存器之间的会话。请参见第 559 页中的 “VIS_DEVINIT”。
VIS_DEVFINI	不适用	终止终端仿真器模块与帧缓存器之间的会话。请参见第 561 页中的 “VIS_DEFINI”。
VIS_CONSDISPLAY	vis_consdisplay	以矩形显示像素。请参见第 561 页中的 “VIS_CONSDISPLAY”。
VIS_CONSCOPY	vis_conscopy	复制像素的矩形区（滚动）。请参见第 562 页中的 “VIS_CONSCOPY”。
VIS_CONSCURSOR	vis_conscursor	显示或隐藏文本光标。请参见第 562 页中的 “VIS_CONSCURSOR”。
VIS_PUTCMAP	vis_cmap	将终端仿真器模块色彩表发送到帧缓存器驱动程序。请参见第 563 页中的 “VIS_PUTCMAP”。
VIS_GETCMAP	vis_cmap	从帧缓存器读取终端仿真器模块色彩表。请参见第 563 页中的 “VIS_GETCMAP”。

轮询式 I/O 接口

轮询式 I/O 接口提供的功能与 VIS_CONSDISPLAY、VIS_CONSCOPY 和 VIS_CONSCURSOR ioctl 接口的功能相同。仅当操作系统被停止并处于**独立**模式时，才调用轮询式 I/O 接口。有关更多信息，请参见第 564 页中的 “在控制台帧缓存器驱动程序中实现轮询式 I/O”。

处于独立模式时，内核终端仿真器通过下表中列出的轮询式 I/O 接口与控制台帧缓存器驱动程序进行通信：

轮询式 I/O 函数	对应的数据结构	说明
(*display)()	vis_consdisplay	以矩形显示像素。
(*copy)()	vis_conscopy	复制像素的矩形区（滚动）。
(*cursor)()	vis_conscursor	显示或隐藏文本光标。

视频模式更改回调接口

在任何时候，控制台帧缓存器驱动程序与内核终端仿真器都必须就视频模式取得一致。**视频模式**涉及到控制台屏幕高度、宽度和深度（以像素为单位）。视频模式还涉及到内核终端仿真器与控制台帧缓存器之间的通信是在 VGA 文本模式还是在像素模式下进行。

为了让控制台帧缓存器驱动程序通知内核终端仿真器有关视频模式方面的变化，将使用下表中所述的 (*modechg_cb)() 内核终端仿真器回调函数的地址初始化控制台帧缓存器驱动程序：

回调函数	对应的数据结构	说明
(*modechg_cb)()	vis_modechg_arg vis_devinit	使终端仿真器模块与驱动程序视频模式（屏幕高度、宽度和像素深度）保持同步。

在控制台帧缓存器驱动程序中实现可视化 I/O 接口

除了视频模式更改回调外，驱动程序与内核终端仿真器之间的所有活动都由 tem（terminal emulator module，终端仿真器模块）启动。这意味着，tem 将发出本文档中介绍的所有 ioctl 命令。以下各节提供了有关每个 ioctl 命令的实现详细信息。有关更多信息，请参见 visual_io(7I) 手册页和 /usr/include/sys/visual_io.h 头文件。有关视频模式更改回调函数的详细信息，请参见第 558 页中的 “视频模式更改回调接口”。

注 – 每个 ioctl 命令都应确定是否已在 ioctl 标志参数中设置 FKIOCTL，如果未设置该位，则返回 EPERM。

VIS_DEVINIT

VIS_DEVINIT ioctl 命令将帧缓存器驱动程序初始化为系统控制台设备。该 ioctl 将传递 vis_devinit 结构的地址。

tem 首先将其视频模式更改回调函数的地址装入 vis_devinit 结构的 modechg_cb 字段，再将其软状态装入 modechg_arg 字段。然后，tem 发出 VIS_DEVINIT ioctl 命令。接下来，帧缓存器驱动程序初始化自身，并通过设置 vis_devinit 结构中的 version、width、height、linebytes、depth、mode 和 polledio 字段将自身的配置摘要返回到 tem。以下代码中显示了 vis_devinit 结构。

```
struct vis_devinit {
      /*
       * This set of fields are used as parameters passed from the
       * layered frame buffer driver to the terminal emulator.
       */
      int             version;        /* Console IO interface rev */
      screen_size_t   width;          /* Width of the device */
      screen_size_t   height;         /* Height of the device */
      screen_size_t   linebytes;      /* Bytes per scan line */
      int             depth;          /* Device depth */
      short           mode;           /* Display mode Mode */
      struct vis_polledio *polledio;  /* Polled output routines */
      /*
       * The following fields are used as parameters passed from the
       * terminal emulator to the underlying frame buffer driver.
       */
      vis_modechg_cb_t modechg_cb;   /* Video mode change callback */
      struct vis_modechg_arg *modechg_arg;  /* Mode change cb arg */
};
```

要在控制台帧缓存器驱动程序中实现 VIS_DEVINIT ioctl 命令，请按照以下通用步骤操作：

1. 定义一个 struct 以包含特定于控制台的状态。该结构由控制台帧缓存器驱动程序专用。在本附录中，该结构称为 consinfo。consinfo 结构包含诸如以下的信息：
   - 位块传输 (blit) 缓冲区的当前大小
   - 指向位块传输 (blit) 缓冲区的指针
   - 色彩表信息
   - 呈现模式信息（如行间距）的驱动程序
   - 背景色
   - 视频内存地址
   - 终端仿真器回调地址
2. 分配内存：

a. 分配足够大的位块传输 (blit) 缓冲区，以便以最高的视频深度存储像素的合理的、缺省大小的矩形区。如果传入的请求超出缓冲区的大小，可以分配额外的内存。帧缓存器驱动程序的最大字体大小为 12×22。假设 DEFAULT_HEIGHT 为 12，DEFAULT_WIDTH 为 22，最大视频深度为 32，那么，缓冲区大小应为 8448 个字节 (DEFAULT_HEIGHT × DEFAULT_WIDTH × 32)。

b. 分配 vis_polledio 结构。

c. 分配缓冲区以用于保持光标。该缓冲区的大小应相当于最大字符的大小。该缓冲区的大小将不会发生变化。

3. 从 modechg_cb 和 modechg_ctx 获取 tem 的视频更改回调地址和回调上下文，并将这些信息存储在 consinfo 结构中。
4. 使用轮询式显示、副本和光标函数的入口点地址填充 vis_polledio 结构。
5. 在 tem 传递给驱动程序的 vis_devinit 结构的字段中提供相应信息：

a. 将 version 字段设置为 VIS_CONS_REV，这是 /usr/include/sys/visual_io.h 头文件中定义的一个常量。

b. 将 mode 字段设置为 VIS_PIXEL。

c. 将 polledio 字段设置为 vis_polledio 结构的地址。

d. 将 height 字段设置为视频模式高度（以像素为单位）。

e. 将 width 字段设置为视频模式宽度（以像素为单位）。

f. 将 depth 字段设置为帧缓存器像素深度，单位为字节（例如，32 位像素深度将为 4 个字节）。

g. 将 linebytes 字段设置为 height × width × depth 的值。

将会使用 vis_devinit 结构将这些信息从驱动程序发送到 tem。通过这些信息，终端仿真器可知道如何呈现信息以及如何将信息传递给图形驱动程序。

只要控制台帧缓存器驱动程序更改了其视频模式（特别是 height、width 或 depth），驱动程序就**必须**调用 tem 的视频模式更改回调函数来更新 vis_devinit 结构并将其传递回终端仿真器。终端仿真器将其模式更改回调函数地址传入 vis_devinit 结构的 modechg_cb 字段。模式更改回调函数具有以下函数签名：

```
typedef void (*vis_modechg_cb_t)
      (struct vis_modechg_arg *, struct vis_devinit *);
```

如前面的 typedef 中所述，模式更改回调函数使用两个参数。第一个参数为 modechg_arg，第二个参数为 vis_devinit 结构。modechg_arg 会在 VIS_DEVINIT ioctl 命令初始化期间从 tem 发送到驱动程序。驱动程序必须通过每个视频模式更改回调将 modechg_arg 发送回给 tem。

6. 初始化内核控制台的上下文。具体的要求会随图形设备的功能而异。例如，该初始化的步骤可能包括：设置绘制引擎状态、初始化调色板，或者定位和映射视频内存或呈现引擎，以便数据能够以位块传输到屏幕。
7. 将 vis_devinit 结构返回给调用方。

VIS_DEFINI

VIS_DEFINI ioctl 命令可释放驱动程序的控制台资源，并完成会话。

要在控制台帧缓存器驱动程序中实现 VIS_DEVFINI ioctl 命令，请按照以下通用步骤操作：

1. 重置控制台帧缓存器驱动程序状态。
2. 清除轮询式 I/O 入口点和内核终端仿真器视频更改函数回调地址。
3. 释放内存。

VIS_CONSDISPLAY

VIS_CONSDISPLAY ioctl 命令可在指定的位置显示像素矩形区。这种显示方式又称为**以位块传输** *(blitting)* 矩形。vis_consdisplay 结构包含以驱动程序和 tem 使用的视频深度呈现矩形所必需的信息。以下代码中显示了 vis_consdisplay 结构。

```
struct vis_consdisplay {
      screen_pos_t    row;      /* Row (in pixels) to display data at */
      screen_pos_t    col;      /* Col (in pixels) to display data at */
      screen_size_t   width;    /* Width of data (in pixels) */
      screen_size_t   height;   /* Height of data (in pixels) */
      unsigned char   *data;    /* Address of pixels to display */
      unsigned char   fg_color; /* Foreground color */
      unsigned char   bg_color; /* Background color */
};
```

要在控制台帧缓存器驱动程序中实现 VIS_CONSDISPLAY ioctl 命令，请按照以下通用步骤操作：

1. 复制 vis_consdisplay 结构。
2. 验证显示参数。如果任一显示参数超出范围，则会返回错误。
3. 计算要以位块传输到视频内存的矩形的大小。根据执行 VIS_DEVINIT 期间创建的位块传输 (blit) 缓冲区大小验证此大小。如果需要，为位块传输 (blit) 缓冲区分配额外的内存。
4. 检索位块传输 (blit) 数据。内核终端仿真器已在议定的像素深度准备了此数据。该深度与执行 VIS_DEVINIT 期间 tem 传递的像素深度相同。每当设备驱动程序通过 tem 的回调更改视频模式时，都会更新像素深度。典型的像素深度为 8 位索引色彩表和 32 位真彩 (TrueColor)。
5. 使所有用户上下文无效，以使用户应用程序不能通过用户内存映射同时访问帧缓存器硬件。在轮询式 I/O 模式下，既不允许也没有必要执行此步骤，因为用户应用程序并没有运行。请务必持有锁，以便在完成 VIS_CONSDISPLAY ioctl 之前，用户无法通过缺页恢复映射。
6. 建立特定于驱动程序的控制台呈现上下文。

7. 如果帧缓存器在 8 位索引色彩模式下运行，请恢复 tem 以前通过 VIS_PUTCMAP ioctl 设置的内核控制台色彩表。建议使用延迟 (lazy) 色彩表装入方案，以优化性能。在**延迟** *(lazy)* 方案中，控制台帧缓存器只恢复自发出 VIS_DEVINIT ioctl 以来实际使用的色彩。
8. 在 tem 发送的像素坐标上显示 tem 传出的数据。您可能需要转换 RGB 像素数据字节顺序。

VIS_CONSCOPY

VIS_CONSCOPY ioctl 命令可将像素矩形区从一个位置复制到另一个位置。该 ioctl 的用途之一就是执行滚动。

要在控制台帧缓存器驱动程序中实现 VIS_CONSCOPY ioctl 命令，请按照下面的通用步骤操作：

1. 复制 vis_conscopy 结构。vis_conscopy 结构描述源和目标矩形大小与位置。
2. 验证显示参数。如果任一显示参数超出范围，则会返回错误。
3. 使所有用户上下文无效，以使用户应用程序不能通过用户内存映射同时访问帧缓存器硬件。在轮询式 I/O 模式下，既不允许也没有必要执行此步骤，因为用户应用程序并没有运行。请务必持有锁，以便在完成 VIS_CONSDISPLAY ioctl 之前，用户无法通过缺页恢复映射。
4. 调用函数以复制矩形。

注 – 为实现最佳性能，请使用图形设备的呈现引擎来实现复制功能。您需要确定如何执行驱动程序内的上下文管理以设置呈现引擎，从而实现最佳性能。

VIS_CONSCURSOR

VIS_CONSCURSOR ioctl 命令可显示或隐藏光标。以下代码中显示了 vis_conscursor 结构。

```
struct vis_conscursor {
      screen_pos_t    row;      /* Row to display cursor (in pixels) */
      screen_pos_t    col;      /* Col to display cursor (in pixels) */
      screen_size_t   width;    /* Width of cursor (in pixels) */
      screen_size_t   height;   /* Height of cursor (in pixels) */
      color_t         fg_color; /* Foreground color */
      color_t         bg_color; /* Background color */
      short           action;   /* Show or Hide cursor */
};
```

要在控制台帧缓存器驱动程序中实现 VIS_CONSCOPY ioctl 命令，请按照下面的通用步骤操作：

1. 从内核终端仿真器复制 vis_conscursor 结构。
2. 验证显示参数。如果任一显示参数超出范围，则会返回错误。
3. 使所有用户上下文无效，以使用户应用程序不能通过用户内存映射同时访问帧缓存器硬件。在轮询式 I/O 模式下，既不允许也没有必要执行此步骤，因为用户应用程序并没有运行。请务必持有锁，以便在完成 VIS_CONSDISPLAY ioctl 之前，用户无法通过缺页恢复映射。
4. 终端仿真器可通过以下两个操作之一调用 VIS_CONSCOPY ioctl：SHOW_CURSOR 和 HIDE_CURSOR。以下步骤介绍如何通过读取和写入视频内存实现此功能。您可能也可使用呈现引擎来完成此工作。是否能够使用呈现引擎取决于帧缓存器硬件。

   执行以下步骤可实现 SHOW_CURSOR 功能：

   a. 将像素保存到要在其中绘制光标的矩形内。隐藏光标时将需要使用这些保存的像素。
   b. 扫描要在其中绘制光标的矩形界定的屏幕上的所有像素。在此矩形中，将与指定光标前景色 (fg_color) 匹配的像素替换为白色像素。将与指定光标背景色 (bg_color) 匹配的像素替换为黑色像素。视觉效果为黑色光标悬停在白色文本上。此方法适用于文本的任何前景色和背景色。尝试根据色彩表位置进行反色是不切实际的。也没有必要使用更复杂的策略，例如使用 HSB（Hue, Saturation, Brightness，色调、饱和度和亮度）色彩模式进行反色。

   要实现 HIDE_CURSOR 功能，请将光标矩形下方的像素替换为通过前面的 SHOW_CURSOR 操作保存的像素。

VIS_PUTCMAP

VIS_PUTCMAP ioctl 命令可建立控制台色彩表。终端仿真器调用此函数以设置内核的色彩表。以下代码中显示了 vis_cmap 结构。该结构只适用于 8 位索引色彩模式。

```
struct vis_cmap {
      int             index;  /* Index into colormap to start updating */
      int             count;  /* Number of entries to update */
      unsigned char   *red;   /* List of red values */
      unsigned char   *green; /* List of green values */
      unsigned char   *blue;  /* List of blue values */
};
```

VIS_PUTCMAP ioctl 命令与 FBIOPUTCMAP 命令类似。VIS_PUTCMAP 命令特定于与帧缓存器终端仿真器兼容的控制台代码。

VIS_GETCMAP

终端仿真器可调用 VIS_GETCMAP ioctl 命令来检索控制台色彩表。

在控制台帧缓存器驱动程序中实现轮询式 I/O

轮询式 I/O 接口在驱动程序中作为函数实现，并由内核终端仿真器直接调用。在执行 VIS_DEVINIT ioctl 命令期间，驱动程序会将其轮询式 I/O 入口点的地址传递给终端仿真器。VIS_DEVINIT 命令由终端仿真器启动。

以下代码中显示了 vis_polledio 结构。

```
typedef void * vis_opaque_arg_t;

struct vis_polledio {
      struct vis_polledio_arg *arg;
      void    (*display)(vis_opaque_arg_t, struct vis_consdisplay *);
      void    (*copy)(vis_opaque_arg_t, struct vis_conscopy *);
      void    (*cursor)(vis_opaque_arg_t, struct vis_conscursor *);
};
```

轮询式 I/O 接口提供的功能与 VIS_CONSDISPLAY、VIS_CONSCOPY 和 VIS_CONSCURSOR ioctl 接口的功能相同。要实现轮询式 I/O 接口，应该按照上面针对相应 ioctl 命令所述的相同步骤操作。轮询式 I/O 接口必须严格遵循本节其余部分所述的其他限制。

仅当操作系统被停止并处于独立模式时，才调用轮询式 I/O 接口。每当用户进入 OpenBoot PROM 或 kmdb 调试器时，或者系统出现紧急情况时，系统就会进入**独立**模式。此时，只有一个 CPU 和一个线程处于活动状态。其他所有 CPU 和线程均会停止。分时、DDI 中断和系统服务都将关闭。

独立模式会严重限制驱动程序的功能，但会简化驱动程序同步要求。例如，用户应用程序无法通过在轮询式 I/O 例程中对控制台帧缓存器驱动程序进行内存映射这样的方式来访问该驱动程序。

在独立模式下，控制台帧缓存器驱动程序**不得**执行下列任一操作：

- 等待中断
- 等待互斥锁
- 分配内存
- 使用 DDI 或 LDI 接口
- 使用系统服务

遵守这些限制并不困难，因为操作轮询式 I/O 函数相对较为简单。例如，在使用呈现引擎时，控制台帧缓存器驱动程序可以轮询设备中的某个位，而不是等待中断。驱动程序可以使用预先分配的内存来呈现位块传输 (blit) 数据。DDI 或 LDI 接口应该是不需要的。

特定于帧缓存器的配置模块

当特定于驱动程序的 fbconfig() 模块导致分辨率或颜色深度发生变化时，该 fbconfig() 模块必须向帧缓存器驱动程序发送 ioctl。此 ioctl 将触发帧缓存器驱动程序，使其使用新的屏幕大小和深度调用终端仿真器的模式更改回调函数。在任何时候，帧缓存器驱动程序与终端仿真器必须就视频模式取得一致。如果帧缓存器驱动程序与终端仿真器未就视频模式取得一致，屏幕上的信息将难以辩认，从而没有意义。

特定于 X 窗口系统帧缓存器的 DDX 模块

当 X 窗口系统退出命令行时，帧缓存器的 DDX 模块必须向帧缓存器驱动程序发送 ioctl。此 ioctl 将触发帧缓存器驱动程序，使其调用终端仿真器的模式更改回调函数。如果 X 窗口系统在启动之后，退出之前更改了视频分辨率，这种通信将使帧缓存器驱动程序与终端仿真器就视频模式取得一致。在任何时候，帧缓存器驱动程序与终端仿真器必须就视频模式取得一致。如果帧缓存器驱动程序与终端仿真器未就视频模式取得一致，屏幕上的信息将难以辩认，从而没有意义。

开发、测试和调试控制台帧缓存器驱动程序

在活动的系统上调试控制台帧缓存器驱动程序可能会遇到问题。

- 在系统引导早期阶段遇到的错误不会生成核心转储。
- 错误或提示性消息可能无法在屏幕上正常显示。
- USB 键盘输入可能失败。

本节提供了一些建议，可帮助您开发、测试和调试控制台帧缓存器驱动程序。

测试 I/O 控制接口

要测试 ioctl 命令，请额外创建一些可通过用户应用程序调用的 ioctl 入口点。确保正确地复制参数。使用 ddi_copyin(9F) 和 ddi_copyout(9F) 例程在用户地址空间来回传输数据。然后编写一个应用程序，以验证呈现、滚动和光标行为。这样，在您开发和测试这些 ioctl 命令时，它们就不会影响您的控制台。

为确保 ioctl 命令正常工作，请引导系统，然后登录。检查在执行 prstat(1M)、ls(1)、 vi(1) 和 man(1) 等命令时，是否能够得到预期的行为。

执行以下脚本以验证 ANSI 颜色是否正常工作：

```
#!/bin/bash
printf "\n\n\n\e[37;40m             Color List       \e[m\n\n"
printf "\e[30m Color 30 black\e[m\n"
```

```
printf "\e[31m Color 31 red\e[m\n"
printf "\e[32m Color 32 green\e[m\n"
printf "\e[33m Color 33 yellow\e[m\n"
printf "\e[34m Color 34 blue\e[m\n"
printf "\e[35m Color 35 purple\e[m\n"
printf "\e[36m Color 36 cyan\e[m\n"
printf "\e[37m Color 37 white\e[m\n\n"
printf "\e[40m Backlight 40 black \e[m\n"
printf "\e[41m Backlight 41 red   \e[m\n"
printf "\e[34;42m Backlight 42 green \e[m\n"
printf "\e[43m Backlight 43 yellow\e[m\n"
printf "\e[37;44m Backlight 44 blue  \e[m\n"
printf "\e[45m Backlight 45 purple\e[m\n"
printf "\e[30;46m Backlight 46 cyan  \e[m\n"
printf "\e[30;47m Backlight 47 white \e[m\n\n"
```

测试轮询式 I/O 接口

轮询式 I/O 接口仅在以下情况下可用：

- 使用 L1+A 击键序列进入 OpenBoot PROM 时
- 使用独立调试器（如 kmdb(1)）引导系统时
- 系统出现紧急情况时

轮询式 I/O 接口仅在引导过程的特定点可用。运行系统之前从 OpenBoot PROM 发出的轮询式 I/O 请求不会呈现。同样，配置控制台之前发出的 kmdb 提示也不会呈现。

要测试轮询式 I/O 接口，请使用 L1+A 击键序列进入 OpenBoot PROM。要验证是否正在使用轮询式 I/O 接口，请在 OpenBoot PROM ok 提示符下键入以下命令：

```
ok 1b emit ." [32m This is a test" 1b emit ." [m"
```

如果以下叙述属实，则表明轮询式 I/O 接口工作正常：

- 以上命令的结果是以绿色字体显示的短语 `This is a test`。
- OpenBoot PROM 继续正常工作。
- 可以按预期方式执行滚动。
- 光标显示正常。
- 可以反复重新进入和继续运行系统。

测试视频模式更改回调函数

要确定视频模式更改回调函数是否正常工作，请登录系统，然后使用 fbconfig(1M) 多次更改帧缓存器的分辨率和深度。如果控制台能够继续正常显示文本，则表明视频模式更改回调函数工作正常。内核终端仿真器可能会调整字体大小以适合不同的屏幕大小，但这并不会对控制台帧缓存器驱动程序有重大影响。

要确定 X 窗口系统和控制台帧缓存器驱动程序是否正常交互，请在 X 窗口系统与命令行之间进行多次切换，同时，以不同的方式修改 X 窗口系统的视频分辨率和命令行分辨率。如果 X 窗口系统退出，并且控制台字符不能正常显示，则要么是 X 窗口系统未将视频模式已更改的情况通知给驱动程序控制台代码，要么是驱动程序未调用内核终端仿真器的视频模式更改回调函数。

有关测试控制台帧缓存器驱动程序的其他建议

在引导过程中，如果系统找不到或者无法成功装入与内核终端仿真器兼容的帧缓存器驱动程序，系统将向 `/var/adm/messages` 发送消息。要监视这些消息，请在单独的窗口中键入以下命令：

```
% tail -f /var/adm/messages
```

为避免调试驱动程序时 USB 发生问题，请更改 EEPROM `input-device` NVRAM 配置参数，以使用串行端口来代替键盘。有关此参数的更多信息，请参见 `eeprom(1M)` 手册页。

◆ ◆ ◆ 附录 E

pci.conf 文件

本部分介绍 pci.conf 文件及其用法和语法。

说明

引入 pci.conf 是为了保存 PCI 配置，如系统上的某个特定 PF（Physical Function，物理功能）的 VF（Virtual Function，虚拟功能）的数目。该文件具有以下几个用途：

- 持久保留 PCI 配置，以便在引导时可以自动创建 VF。
- 由于配置文件是 boot_archive 的一部分，因此在引导过程中可以使用 VF。

在裸机系统上使用 VF 时，该文件还可以供非 IOV 系统配置使用。当前，该文件只包含与 VF 相关的配置。将来，更多特定于 PCI 总线的配置甚至特定于设备的解决方法可能会包含到该文件中。VF 配置的数目保存在 "[System_Configuration]" 部分中，类似于以下示例：

```
[System Configuration]
[[path=<pf_device_path>]]
num-vf=<num_of_vf>
```

系统配置部分

文件的 [System Configuration] 部分由 Oracle Solaris PCIe 框架进行解释。无法识别的关键字将被标记为错误。整个文件中只有一个 [System Configuration] 部分，并且它必须位于文件的开头。

[System Configuration] 部分由一系列子部分组成。每个子部分必须有一个唯一的文本标签，后跟一个括在方括号中的过滤器列表（与要关注的设备相匹配）。每个子部分的内容都是一个操作列表，框架将针对每个匹配的设备执行这些操作。例如：

```
[System Configuration]
     new_e1kg_driver [[id=0x8086,0x1000,,0x108e,]] [[classcode=0x020000]]
     num-vf=4
```

双括号中的过滤器将匹配系统中具有设备 ID 0x1000、Sun 子系统供应商 ID 以及网络控制器类别代码的所有 Intel 设备。Oracle Solaris 将设备的 VF 数设置为 4。

注 – 可在过滤器中使用设备路径来将过滤器的作用域缩小到单个设备实例。

设备配置部分

[Device Configuration] 部分包含与 [System Configuration] 部分相同类型的子部分，不同之处是它的内容仅由与过滤器匹配的设备的驱动程序进行解释。

```
Device Configuration]
     # label must be file-globally unique
     igbe-sriov-test [[path=/pci@0,0/network@2]]
     # number of rx/tx ring pairs for each VF
     dma-channel-distribution=2,2,8,4
```

语法

```
/etc/pci.conf = <system section><device section>
<system section> = "[System_Configuration]" {<framework subsection>}*
<device section> = "[Device_Configuration]" {<device subsection>}*
<framework subsection> = <label> [ {filter>}* ]
[ {<framework action>}+ ]
<device subsection> = <label> [ {<filter>}+* ] <devicenvlist>
<filter> = [ "[[" "id" "=" (vendorid>) "," (<deviceid>) "," (<revisionid>) "," (<subsystem-vendorid>) "," (<subsystemid>)"
| ["[[" "classcode" "=" <classcode> (,<mask>) "]]" ] | [ "[[" "path"
"=" ,devpath> "]]"]
<framework action> = [ "num-vf" "=" <val>]
<deice nvlist> = [ {name string> "=" <value string>} +]
```

参考信息

另请参见 pci_param_get(9F)。

索引

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

高

故

关

管

光

实

使

事

视

输

属

数

总