Writing Device Drivers

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part Number 805-7378–10
February 2000

Copyright 2000 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, docs.sun.com, AnswerBook, AnswerBook2, Sun Cluster, Sun Workshop Compilers C, Power Management, Ultra, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227–14(g)(2)(6/87) and FAR 52.227–19(6/87), or DFAR 252.227–7015(b)(6/95) and DFAR 227.7202–3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2000 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, Californie 94303-4900 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées du système Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, docs.sun.com, AnswerBook, AnswerBook2, Sun Cluster, Sun Workshop Compilers C, Power Management, Ultra, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Contents

Preface

Writing Device Drivers provides information on developing device drivers for character-oriented devices, block-oriented devices, and small computer system interface (SCSI) target devices. This book discusses the development of a dynamically loadable and unloadable, multithreaded reentrant device driver applicable to all architectures that conform to the Solaris™ 8 DDI/DKI. A common driver programming approach is taken so that drivers can be written without concern for platform-specific issues, such as *endianness* and data ordering.

Who Should Use This Book

The audience for this book is UNIX® programmers familiar with UNIX device drivers. Several overview chapters at the beginning of the book provide background information for the detailed technical chapters that follow, but they are not intended as a general tutorial or text on device drivers.

How This Book Is Organized

This book is organized into the following chapters.

- Chapter 1 provides an overview of the Solaris kernel and the manner in which it represents devices as nodes in a device tree.
- Chapter 2 gives an outline of the kinds of device drivers and their basic structure. It points out the common data access routines and concludes with an illustrated roadmap of common driver entry points and structures.

- Chapter 3 describes the mechanisms of the Solaris multithreaded kernel that are of interest to driver writers.
- Chapter 4 describes the set of interfaces used to read an update device node properties.
- Chapter 5 explains the support a driver must provide for autoconfiguration.
- Chapter 6 describes the interfaces and methodologies for drivers to use to access (read or write) device memory.
- Chapter 7 describes the interrupt handling mechanisms. These include registering, servicing, and removing interrupts.
- Chapter 8 describes direct memory access (DMA) and the DMA interfaces.
- Chapter 9 explains the interfaces for Power Management™, a framework designed to regulate and reduce the power consumed by computer systems and devices.
- Chapter 10 describes the structure and functions of a driver for a character-oriented device.
- Chapter 11 describes the structure and functions of a driver for a block-oriented device.
- Chapter 12 describes the set of interfaces that allow device drivers to manage access to memory, control the context of user processes accessing a device, and take advantage of large data transfers using new MMU hardware.
- Chapter 13 describes the set of interfaces that allow device drivers to manage user access to devices.
- Chapter 14 outlines the Sun Common SCSI Architecture and describes the additional requirements of SCSI target drivers.
- Chapter 15 explains how to write a SCSI Host Bus Adapter (HBA) driver using the Sun Common SCSI Architecture (SCSA).
- Chapter 16 provides information on compiling and linking a driver, and for installing it in the system.
- Chapter 17 gives coding suggestions, debugging hints, a simple `mdb/kadb` tutorial, and some hints on testing the driver.
- Appendix A discusses multi-platform hardware issues related to device drivers.
- Appendix B summarizes, by topic, the kernel functions device driver can use.
- Appendix C provides guidelines for updating a device driver to run in a 64-bit environment.
- Appendix D provides information for creating cluster-aware drivers.
- Appendix E describes the concepts and techniques used to support hardened drivers.

Related Books and Papers

For detailed reference information about the device driver interfaces, see the man page sections 9, 9E (entry points), 9F (functions), and 9S (structures). For information on hardware issues and other driver-related issues, the following books might be helpful.

- *Application Packaging Developer's Guide*, Sun Microsystems, 2000.
- *The SPARC Architecture Manual, Version 9*. Prentice Hall, 1998. ISBN 0–13–099227–5.
- *The SPARC Architecture Manual, Version 8*. Prentice Hall, 1994. ISBN 0-13-825001-4.
- *Pentium Pro Family Developer's Manual, Volumes 1-3*. Intel Corporation, 1996. ISBN 1-55512-259-0 (Volume 1) , ISBN 1-55512-260-4 (Volume 2) , ISBN 1-55512-261-2 (Volume 3).
- *Open Boot PROM Toolkit User's Guide*, Sun Microsystems Computer Company, 1996.
- *STREAMS Programming Guide*. Sun Microsystems, 2000.
- *Multithreaded Programming Guide*. Sun Microsystems, 2000.
- *Solaris 64-bit Developer's Guide*. Sun Microsystems, 2000.
- *Solaris Modular Debugger Guide*. Sun Microsystems, 2000.

Note - In this document, the term "IA" refers to the Intel 32–bit processor architecture, which includes the Pentium, Pentium Pro, Pentium II, Pentium II Xeon, Celeron, Pentium III, and Pentium III Xeon processors, and compatible microprocessor chips made by AMD and Cyrix.

Note - The Solaris operating environment runs on two types of hardware, or platforms—SPARC™ and IA. The Solaris operating environment also runs on both 64–bit and 32–bit address spaces. The information in this document pertains to both platforms and address spaces unless called out in a special chapter, section, note, bullet, figure, table, example, or code example.

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems™, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com℠ Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

What Typographic Conventions Mean

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls –a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name%` **`su`** `Password:`

TABLE P–1 Typographic Conventions *(continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **rm** *filename*.
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide.* These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	#
MDB prompt	`>`
SPARC PROM prompt	`ok`

CHAPTER 1

Solaris Kernel and Device Tree

This chapter provides an overview of the Solaris kernel and the manner in which it represents devices as nodes in a device tree. It covers general kernel structure and function and the Solaris 8 Device Driver Interface/Driver Kernel Interface (DDI/DKI).

What Is the Kernel?

The Solaris kernel is a program that manages system resources. It insulates applications from the system hardware and provides them with essential system services such as input/output (I/O) management, virtual memory, and scheduling. The kernel consists of object modules that are dynamically loaded into memory when needed. The reader should have a working knowledge of this information.

The Solaris kernel can be separated into two parts: the first part, referred to as the kernel, manages file systems, scheduling, and virtual memory. The second part, referred to as the I/O subsystem, manages the physical components, as described in Figure 1–1.

The kernel provides a set of interfaces for applications to use called *system calls*. System calls are documented in the *Solaris 8 Reference Manual Collection* (see `Intro`(2)). The function of some system calls is to invoke a device driver to perform I/O. Device drivers are loadable kernel modules that insulate the rest of the kernel from device hardware and manage data transfers.

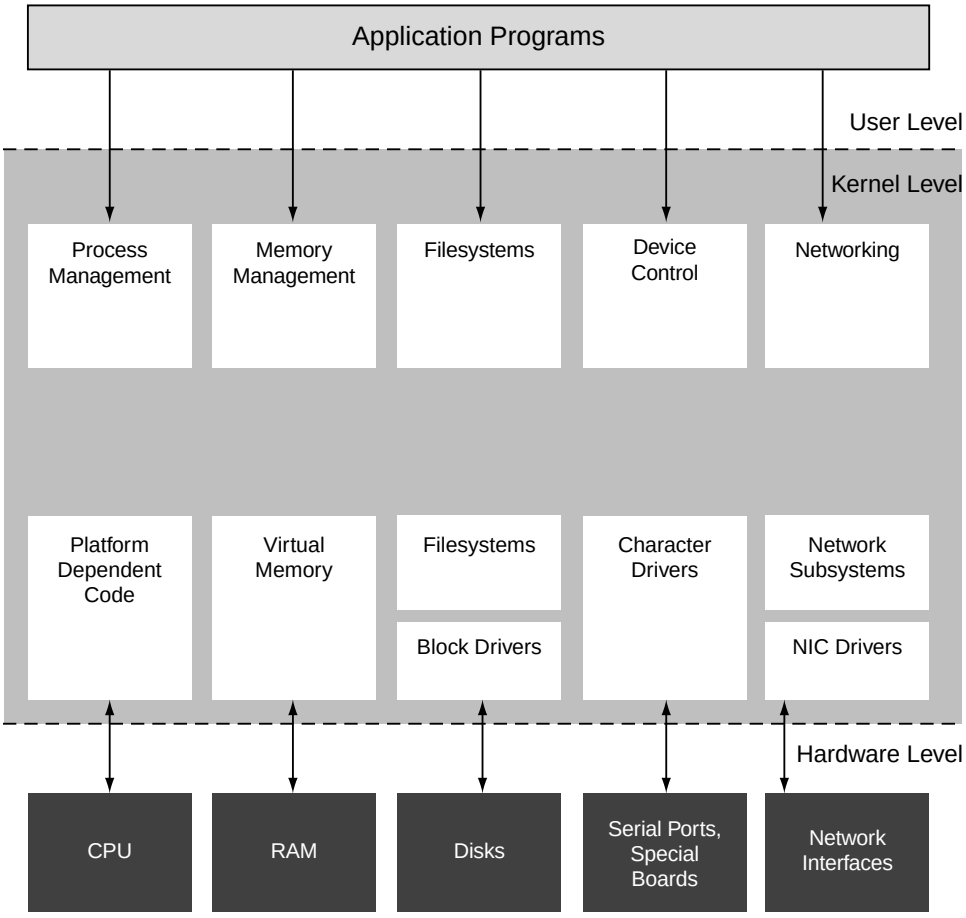


Figure 1–1 The Solaris Kernel

This book discusses the specifics of device drivers. The following sections provide additional high-level information on the Solaris operating environment.

Multithreaded Execution Environment

The Solaris kernel is multithreaded. On a multiprocessor machine, multiple kernel threads can be running kernel code, and can do so concurrently. Kernel threads can also be pre-empted by other kernel threads at any time.

The multithreading of the kernel imposes some additional restrictions on the device drivers. For more information on multithreading considerations, see Chapter 3. Device drivers must be coded to run as needed at the request of many different threads. For each thread, it must handle contention problems from overlapping I/O requests.

Virtual Memory

A complete overview of the Solaris virtual memory system is beyond the scope of this book, but two virtual memory terms of special importance are used when discussing device drivers: virtual address and address space.

- Virtual address – A *virtual address* is an address that is mapped by the memory management unit (MMU) to a physical hardware address. All addresses directly accessible by the driver are kernel virtual addresses; they refer to the *kernel address space*.
- Address space – An *address space* is a set of *virtual address segments*, each of which is a contiguous range of virtual addresses. Each user process has an address space called the *user address space*. The kernel has its own address space called the *kernel address space*.

Special Files

Devices are treated as files. They are represented in the file system by *special files*. In the Solaris operating environment these files reside in the `/devices` directory hierarchy.

Special files can be of type *block* or *character*. The type indicates which kind of device driver operates the device. Drivers can be implemented to operate on both types. For example, disk drivers export a character interface for use by the `fsck(1)` and `mkfs(1)` utilities, and a block interface for use by the filesystem.

Associated with each special file is a *device number* (`dev_t`). This consists of a *major number* and a *minor number*. The major number identifies the device driver associated with the special file. The minor number is created and used by the device driver to further identify the special file. Usually, the minor number is an encoding that identifies the device instance the driver should access and the type of access to perform. The minor number, for example, could identify a tape device used for backup and also specify whether the tape needs to be rewound when the backup operation is complete.

Solaris 8 DDI/DKI

In System V Release 4 (SVR4), the interface between device drivers and the rest of the UNIX kernel was standardized as the DDI/DKI. The Solaris 8 DDI/DKI is documented in Section 9 of the *Solaris 8 Reference Manual Collection*. The reference manual documents driver entry points, driver-callable functions, and kernel data structures used by device drivers.

The Solaris 8 DDI/DKI, like its SVR4 counterpart, is intended to standardize and document all interfaces between device drivers and the rest of the kernel. In addition, the Solaris 8 DDI/DKI is designed to allow source compatibility for drivers

on any machine running the Solaris 8 operating environment, regardless of the processor architecture (such as SPARC or IA). It is also intended to provide binary compatibility for drivers running on any Solaris 8 based processor, regardless of the specific platform architecture. Drivers using only kernel facilities that are part of the Solaris 8 DDI/DKI are known as *Solaris 8 DDI/DKI-compliant device drivers*.

The Solaris 8 DDI/DKI allows platform-independent device drivers to be written for Solaris 8 based machines. These *shrink-wrapped* (binary compatible) drivers allow third-party hardware and software to be more easily integrated into SunOS 5.8-based machines. The Solaris 8 DDI/DKI is designed to be architecture independent and enable the same driver to work across a diverse set of machine architectures.

Platform independence is accomplished by the design of DDI in Solaris 8 DDI/DKI. The following main areas are addressed:

- Dynamic loading and unloading of modules
- Power management
- Interrupt handling
- Accessing the device space from the kernel or a user process (register mapping and memory mapping)
- Accessing kernel or user process space from the device (DMA services)
- Managing device properties

Device Tree

Devices in Solaris are represented as a tree of interconnected device nodes. The tree begins at the 'root' device node, which represents the platform. Below the root node are 'branches' of the device tree, where a branch consists of one or more bus nexus devices and a terminating leaf device. The system builds a tree structure that contains information about the devices connected to the machine at boot time. The device tree can also be modified by dynamic reconfiguration operations while the system is in normal operation.

The tree structure creates a parent-child relationship between nodes. This parent-child relationship is the key to architectural independence. When a leaf or bus nexus driver requires a service that is architecturally dependent in nature, it requests its parent to provide the service. This approach enables drivers to function regardless of the architecture of the machine or the processor.

Bus nexus devices are devices that provide bus mapping and translation services to devices that are subordinate to it in the device tree. PCI - PCI bridges, PCMCIA adapters, and SCSI HBAs are all example of nexus devices. The discussion of writing drivers for nexus devices is limited to that of developing SCSI HBA drivers.

Leaf devices are typical peripheral devices such as disks, tapes, network adapters, frame buffers, and so forth. Drivers for these devices export the traditional character and block driver interfaces for use by user processes to read and write data to storage or communication devices.

Example Device Tree

Figure 1–2 illustrates a sample device tree.

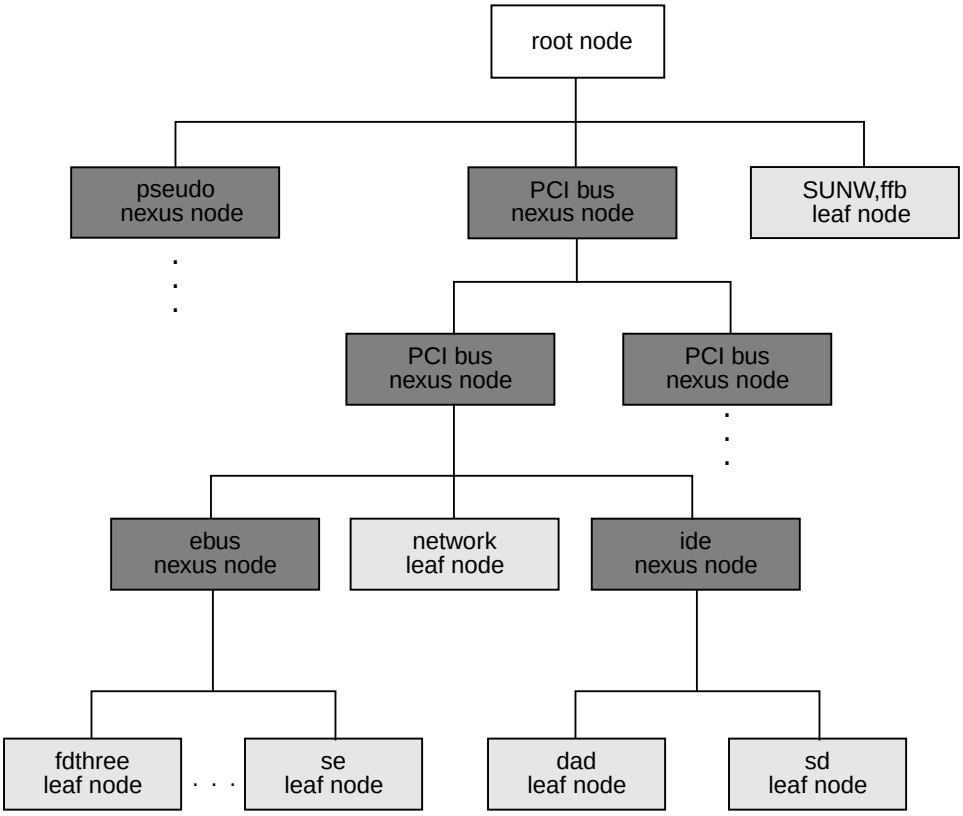


Figure 1–2 Example Device Tree

In Figure 1–2, the root node is the parent node of the child nodes `SUNW,ffb` leaf node (a frame buffer), a pseudo bus nexus node, and a PCI bus nexus node.

The `SUNW,ffb` leaf node represents a system frame buffer. The pseudo bus nexus node is the parent of any pseudo device drivers (drivers without hardware). The PCI bus nexus node further has two PCI bus nexus nodes as its children representing two PCI-to-PCI bridges.

The lower left PCI bus nexus node is the parent of the child nodes; ebus bus nexus node, network leaf node (ethernet), and ide bus nexus node.

The ebus bus nexus node is the parent of the child nodes fdthree leaf node (a floppy disk device) and se leaf node (a serial device).

The ide bus nexus node is the parent of the child nodes dad leaf node (a disk device) and sd leaf node (a CD-ROM device).

Device Drivers

Associated with each leaf or bus nexus node can be a device driver. Each driver exports a device operations structure `dev_ops`(9S) that defines the operations that the device driver can perform. The device operations structure contains function pointers for generic operations such as `attach`(9E), `detach`(9E), and `getinfo`(9E). It also contains a pointer to a set of operations specific to bus nexus drivers and a pointer to a set of operations specific to leaf drivers.

Displaying the Device Tree

The device tree can be displayed in three ways. The `libdevinfo` library provides interfaces to access the contents of the device tree programmatically. The `prtconf`(1M) command displays the complete contents of the device tree. And, the `/devices` hierarchy is a representation of the device tree; use `ls`(1) to view it.

Note - `/devices` displays only devices that have drivers configured into the system. `prtconf`(1M) shows all device nodes regardless of whether a driver for the device exists on the system or not.

`libdevinfo(3DEVINFO)`

`libdevinfo`(3DEVINFO) provides interfaces for accessing all public device configuration data. See `libdevinfo`(3LIB) for a list of interfaces. See `http://soldc.sun.com/developer/support/driver/docs/whitepapers.html` for the `libdevinfo` whitepaper.

`prtconf(1M)`

The `prtconf`(1M) command (excerpted example follows) displays all the devices in the system:

```
System Configuration:  Sun Microsystems  sun4u
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

The /devices hierarchy provides a name space representing the device tree. Following is an abbreviated listing of the /devices name space. The sample output corresponds to the example device tree and prtconf(1M) output shown previously.

```
/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
```

Binding a Driver to a Device

In addition to constructing the device tree, the kernel also determines the drivers that will be used to manage the devices.

Binding a driver to a device refers to the process by which the system selects a driver to manage a particular device. The driver binding name is the name that links a driver to a unique device node in the device information tree. For each device in the device tree, the system attempts to choose a driver from a list of installed drivers.

Each device node has a *name* property associated with it. This property can be assigned either from an external agent, such as the PROM, during system boot or from a driver.conf configuration file. In either case, the name property represents the node name assigned to a device in the device tree. The node name is the name visible in /devices and listed in the prtconf(1M) output.

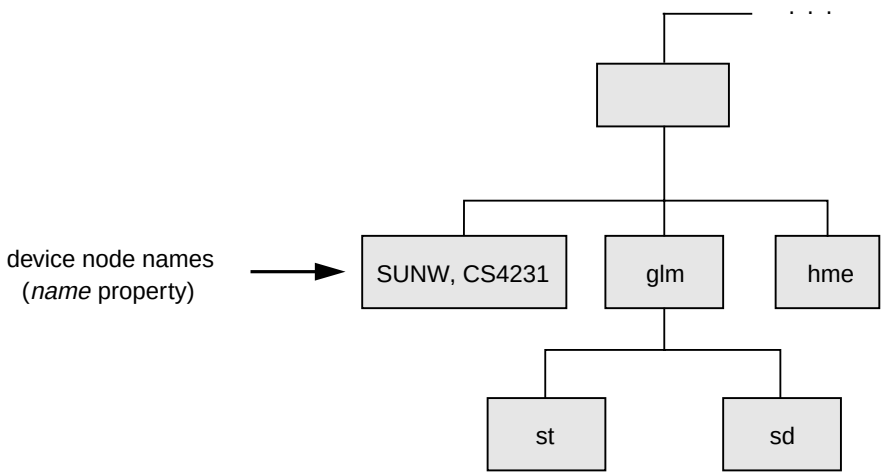


Figure 1–3 Device Node Names

A device node can also have a *compatible* property associated with it. The *compatible* property (if it exists) contains an ordered list of one or more possible driver names or driver aliases for the device.

The system uses both the *name* and the *compatible* properties to select a driver for the device. If the *compatible* property exists, the system first attempts to match the contents of the *compatible* property to a driver on the system. Beginning with the first driver name on the *compatible* property list, the system attempts to match the driver name to a known driver on the system. It processes each entry on the list until either a match is found or the end of the list is reached.

If the contents of either the *name* property or the *compatible* property match a driver on the system, then that driver is bound to the device node. If no match is found, no driver is bound to the device node.

Generic Device Names

Some devices specify a *generic* device name as the value for the *name* property. Generic device names describe the function of a device without actually identifying a specific driver for the device. For example, a SCSI host bus adapter might have a generic device name of `scsi`. An Ethernet device might have a generic device name of `ethernet`.

The *compatible* property allows the system to determine alternate driver names (like `glm` for `scsi` HBA device drivers or `hme` for `ethernet` device drivers) for devices with a generic device name.

Devices with generic device names are required to supply a *compatible* property.

Note - For a complete description of *generic device names*, see the IEEE 1275 Open Firmware Boot Standard.

Figure 1–4 and Figure 1–5 show two device nodes: one node uses a specific device name and the other uses a generic device name.

System Driver List

```
esp isp cgsix sd SUNW,ffb st pci ...
```

For the device node with a specific device name, the driver binding name `SUNW,ffb` is the same name as the device node name.

Device Node A

```
          name =  SUNW,ffb
  binding name =  SUNW,ffb

/devices/SUNW,ffb@le,0:ffb0
```

Figure 1–4 Specific Driver Node Binding

For the device node with the generic device name `display`, the driver binding name `SUNW,ffb` is the first name on the *compatible* property driver list that matches a driver on the system driver list. In this case, `display` is a generic device name for frame buffers.

Device Node B

```
          name =  display
    compatible =  fast_fb
                  SUNW,ffb
                  slow_fb
  binding name =  SUNW,ffb

/devices/display@le,0:ffb0
```

Figure 1–5 Generic Driver Node Binding

CHAPTER 2

Overview of Solaris Device Drivers

This chapter gives an overview of Solaris device drivers. It discusses what a device driver is and the types of device drivers that Solaris 8 supports. It also provides a general discussion of the routines that device drivers must implement and points out compiler-related issues.

What Is a Device Driver?

A *device driver* is a kernel module responsible for managing low-level I/O operations for a particular hardware device. Device drivers can also be software-only, emulating a device that exists only in software, such as a RAM disk or a pseudo-terminal.

A device driver contains all the device-specific code necessary to communicate with a device and provides a standard set of interfaces to the rest of the system. This interface protects the kernel from device specifics just as the system call interface protects application programs from platform specifics. Application programs and the rest of the kernel need little (if any) device-specific code to address the device. In this way, device drivers make the system more portable and easier to maintain.

Types of Device Drivers

There are several kinds of device drivers, each handling a different kind of I/O. Block device drivers manage devices with physically addressable storage media, such as disks. All other devices are considered character devices. Two types of character device drivers are standard character device drivers and STREAMS device drivers.

Block Device Drivers

Devices that support a file system are known as *block devices*. Drivers written for these devices are known as block device drivers. Block device drivers take a file system request, in the form of a `buf`(9S) structure, and issue the I/O operations to the disk to transfer the specified block. The main interface to the file system is the `strategy`(9E) routine. See Chapter 11 for more information.

Block device drivers can also provide a character driver interface that allows utility programs to bypass the file system and access the device directly. This device access is commonly referred to as the *raw* interface to a block device.

Character Device Drivers

Character device drivers normally perform I/O in a byte stream. They can also provide additional interfaces not present in block drivers, such as I/O control (ioctl) commands, memory mapping, and device polling. See Chapter 10 for more information.

Byte-Stream I/O

The main task of any device driver is to perform I/O, and many character device drivers do what is called *byte-stream* or *character* I/O. The driver transfers data to and from the device without using a specific device address. This is in contrast to block device drivers, where part of the file system request identifies a specific location on the device.

The `read`(9E) and `write`(9E) entry points handle byte-stream I/O for standard character drivers. See “I/O Request Handling” on page 145 for more information.

Memory Mapped Devices

For certain devices, such as frame buffers, it is more efficient for application programs to have direct access to device memory. Applications can map device memory into their address spaces using the `mmap`(2) system call. To support memory mapping, device drivers implement `segmap`(9E) and `devmap`(9E) entry points. For information on `devmap`(9E), see Chapter 12. For information on `segmap`(9E), see Chapter 10.

Drivers that define the `devmap`(9E) entry point usually do not define `read`(9E) and `write`(9E) entry points, as application programs perform I/O directly to the devices after calling `mmap`(2).

STREAMS Drivers

STREAMS is a separate programming model for writing a character driver. Devices that receive data asynchronously (such as terminal and network devices) are suited to a STREAMS implementation. STREAMS device drivers must provide the loading and autoconfiguration support described in Chapter 5. See the *Streams Programming Guide* for additional information on how to write STREAMS drivers.

Driver Module Entry Points

Each device driver defines a standard set of functions called *entry points*, which are listed in `Intro`(9E). These entry points are called by the Solaris kernel to load and unload the driver, autoconfigure devices, and provide the character, block, or STREAMS driver I/O services. Drivers for different types of devices have different sets of entry points according to the kinds of operations the devices perform. A driver for a memory-mapped character-oriented device, for example, supports a `devmap`(9E) entry point, while a block driver does not.

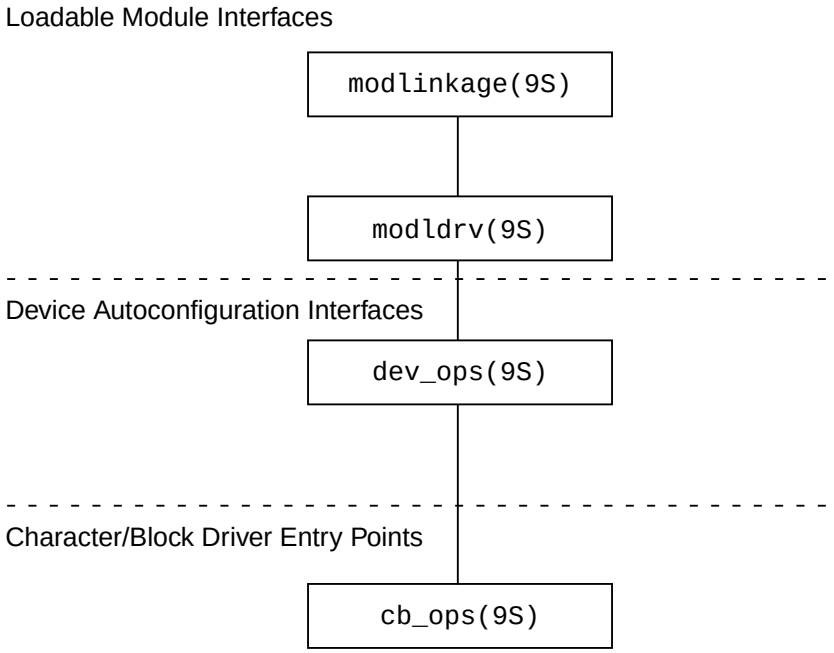


Figure 2–1 Device Driver Overview

Some operations are common to all drivers, such as the functions that are required for module loading (`_init`(9E), `_info`(9E), and `_fini`(9E)), and the required

autoconfiguration entry points attach(9E), detach(9E), and getinfo(9E). Drivers also support the optional autoconfiguration entry point for probe(9E). Most leaf drivers have open(9E) and close(9E) entry points to control access to their devices.

Traditionally, all driver function and variable names have some prefix added to them. Usually this is the name of the driver, such as xxopen( ) for the open(9E) routine of driver xx. In subsequent examples, xx is used as the driver prefix.

Note - In the Solaris 8 operating environment, only the loadable module routines must be visible outside the driver object module. Other routines can have the storage class static.

Loadable Module Entry Points

All drivers are required to implement the loadable module entry points _init(9E), _fini(9E), and _info(9E) entry points to load, unload, and report information about the driver module.

It is recommended that drivers allocate and initialize any global resources in _init(9E) and release their resources in _fini(9E).

Autoconfiguration Entry Points

Drivers are required to implement the attach(9e), detach(9e), and getinfo(9e) entry points for device autoconfiguration. Drivers might need to implement probe(9e) if the driver supports devices that are not self identifying, such as SCSI target devices.

Character and Block Driver Entry Points

Drivers for character and block devices export a cb_ops(9S) structure, which defines the driver entry points for block device access and character device access. Both types of drivers are required to support open(9E) and close(9E). Block drivers are required to support strategy(9E), while character drivers can choose to implement whatever mix of read(9E), write(9E), ioctl(9E), mmap(9E), or devmap(9E) entry points as appropriate for the type of device. Character drivers can also support a polling interface through chpoll(9E), as well as asynchronous I/O through aread(9E) and awrite(9E).

For information on character driver entry points, see Chapter 10. For information on block driver entry points, see Chapter 11.

Power Management Entry Point

Drivers for hardware devices that provide Power Management functionality can support the optional `power`(9E) entry point. See Chapter 9 for details about this entry point.

Driver Context

The driver context determines which kernel routines the driver is permitted to call. There are four contexts in which driver code executes:

- User context – A driver entry point has *user context* if it was directly invoked because of a user thread. For example, the `read`(9E) entry point of the driver, invoked by a `read`(2) system call, has user context.
- Kernel context – A driver function has *kernel context* if it was invoked by some other part of the kernel. In a block device driver, the `strategy`(9E) entry point can be called by the pageout daemon to write pages to the device. Because the page daemon has no relation to the current user thread, `strategy`(9E) has kernel context in this case.
- Interrupt context – *Interrupt context* is a more restrictive form of kernel context. Driver interrupt routines operate in interrupt context and have an interrupt level associated with them. Callback routines also operating in an interrupt context. See Chapter 7 for more information.
- High-level interrupt context – *High-level interrupt context* is a more restricted form of interrupt context. If `ddi_intr_hilevel`(9F) indicates that an interrupt is high level, the driver interrupt handler will run in high-level interrupt context. See Chapter 7 for more information.

The manual pages in section 9F document the allowable contexts for each function. For example, in kernel context the driver must not call `copyin`(9F).

Interrupt Handling

The Solaris 8 DDI/DKI addresses these aspects of device interrupt handling:

- Registering device interrupts with the system
- Removing device interrupts
- Dispatching interrupts to interrupt handlers

Device interrupt sources are contained in a property called *interrupts*, which is either provided by the PROM of a self-identifying device, in a hardware configuration file, or by the booting system on the IA platform.

Callback Functions

Certain DDI mechanisms provide a *callback* mechanism. DDI functions provide a mechanism for scheduling a callback when a condition is met. Conditions for which callback functions are used include:

- When a transfer has completed
- When a resource *might* have become available
- When a time-out period has expired

In some sense, callback functions are similar to entry points—interrupt handlers, for example. DDI functions that allow callbacks expect the callback function to perform certain tasks. In the case of DMA routines, a callback function must return a value indicating whether the callback function needs to be rescheduled in case of a failure.

Callback functions execute as a separate interrupt thread and must handle all the usual multithreading issues.

Note - A driver must cancel all scheduled callback functions before detaching a device.

Printing Messages

Device drivers do not usually print messages. Instead, the driver entry points should return error codes so that the application can determine how to handle the error. If the driver must print a message, it should use `cmn_err`(9F) to do so. This is similar to the C function `printf`(3C), which prints to the console, to the message buffer, or both.

The `format` string specifier interpreted by `cmn_err`(9F) is similar to the `printf`(3C) format string, with the addition of the format %b, which prints bit fields. Callers to `cmn_err`(9F) also specify the `level`, which indicates the label to be printed. The first character of the format string is treated specially. See `cmn_err`(9F) for more details.

`CE_PANIC` has the side effect of crashing the system. This level should be used only if the system is in such an unstable state that to continue would cause more

problems. It can also be used to get a system core dump when debugging. It should not be used in production device drivers.

Device IDs

The Solaris DDI provides interfaces that allow drivers to provide a persistent unique identifier for a device, a 'device ID', which can be used to identify or locate a device and which is independent of the devices name or number (`dev_t`). Applications can use the functions defined in `libdevid`(3LIB) to read and manipulate the device IDs registered by the drivers.

Software State Management

To assist device driver writers in allocating state structures, the Solaris 8 DDI/DKI provides a set of memory management routines called the *software state management routines* (also known as the *soft state routines*). These routines dynamically allocate, retrieve, and destroy memory items of a specified size, and hide the details of list management. An *instance number* is used to identify the desired memory item; this number can be (and usually is) the instance number assigned by the system.

Routines are provided to:

- Initialize a driver's soft state list
- Allocate space for an instance of a driver's soft state
- Retrieve a pointer to an instance of a driver's soft state
- Free the memory for an instance of a driver's soft state
- Finish using a driver's soft state list

See "Loadable Driver Interfaces" on page 65 for an example of how to use these routines.

Dynamic Memory Allocation

Device drivers must be prepared to simultaneously handle all attached devices that they claim to drive. There should be no driver limit on the number of devices that the driver handles, and all per-device information must be dynamically allocated.

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

The standard kernel memory allocation routine is `kmem_alloc`(9F). It is similar to the C library routine `malloc`(3C), with the addition of the `flag` argument. The `flag` argument can be either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether the caller is willing to block if the requested size is not available. If `KM_NOSLEEP` is set, and memory is not available, `kmem_alloc`(9F) returns `NULL`.

`kmem_zalloc`(9F) is similar to `kmem_alloc`(9F), but also clears the contents of the allocated memory.

Note - Kernel memory is a limited resource, not pageable, and competes with user applications and the rest of the kernel for physical memory. Drivers that allocate a large amount of kernel memory can cause system performance to degrade.

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

Memory allocated by `kmem_alloc`(9F) or by `kmem_zalloc`(9F) is returned to the system with `kmem_free`(9F). This is similar to the C library routine `free(3C)`, with the addition of the `size` argument. Drivers *must* keep track of the size of each object they allocate in order to call `kmem_free`(9F) later.

Programmed I/O Device Access

Programmed I/O device access is the act of reading and writing of device registers or device memory by the host CPU. The Solaris DDI provides interfaces for mapping a devices registers or memory by the kernel as well as interfaces for reading and writing to device memory from the driver. These interfaces are designed to enable drivers to be developed that are platform and bus independent, by automatically managing any difference in device and host endianness as well as enforcing any memory store ordering requirements imposed by the device.

Direct Memory Access (DMA)

Solaris defines a high level architecture independent model for supporting DMA capable devices. The Solaris DDI is designed to shield drivers from platform specific details, which enables a common driver to be developed that runs across multiple platforms and architectures.

Properties

Properties define attributes of the device or device driver. Properties can be defined by the FCode of a self-identifying device, by a hardware configuration file (see `driver.conf`(4)), or by the driver itself using the `ddi_prop_update`(9F) family of routines.

A property is a name-value pair. The name is a string that identifies the property with an associated value. The value of a property can be one of five types:

- A byte array that has an arbitrary length and whose value is a series of bytes
- An integer property whose value is an integer
- An integer array property whose value is an array of integers
- A string property whose value is a NULL-terminated string
- A string array property whose value is a list of NULL-terminated strings

A property that has no value is known as a Boolean property. It is considered to be true if it exists and false if it doesn't exist.

Driver and Device Statistics

Solaris provides a rich set of interfaces for maintaining and exporting kernel level statistics, or '`kstats`'. Drivers are free to use these interfaces to export driver and device statistics that can be used by user applications to observe the internal state of the driver. See `kstat_create`(9F) and `kstat`(3KSTAT) for additional information.

64-Bit Considerations

The Solaris system can run in 64-bit mode on appropriate hardware and provides a 64-bit kernel with a 64-bit address space for applications. To update a device driver to be 64-bit ready, driver writers need to understand the 32-bit and 64-bit C data type models, know how to use the system derived types and the fundamental C data types, and understand specific driver issues, such as how to enable a 64-bit driver and a 32-bit application to share data structures.

For details on making a device driver ready for a 64-bit environment, see Appendix C.

Kernel Programming Model

The Solaris kernel is a large collection of code that is compiled in one of two ways; it is either compiled as a 32-bit program that supports solely 32-bit applications, or as a 64-bit program that supports both 32-bit and 64-bit applications. To allow drivers and STREAMS modules to be used on both systems, you must write kernel code that is both portable between these two compilation environments and supportive of 32-bit and 64-bit applications. The resulting code must be compiled in two ways, creating two separate modules: a 32-bit module for the 32-bit kernel, and a 64-bit module for the 64-bit kernel.

Some classes of portability issues can best be solved using the standard derived types, such as `size_t`, `off_t`, `time_t`, and `caddr_t`, since these grow and shrink appropriately. To provide better support of 32-bit applications in the 64-bit kernel, fixed-width types corresponding to the sizes expected by 32-bit applications are available in `<sys/types32.h>`, for example, `size32_t`, `off32_t`, `time32_t`, and `caddr32_t`.

Other classes of portability problems, in particular those describing hardware registers or data sent over the wire, are best described using the size-invariant types in `<sys/inttypes.h>`; for example, `uint16_t`, and `int64_t`. It also includes the definition of `intptr_t` and `uintptr_t`.

See the *Solaris 64-bit Developer's Guide* for the full list of changes to derived types and more information on fixed-width types.

Data Model Concepts

The term *data model* is used here to describe the model for addresses and data that is used by the kernel and applications.

On the 32-bit kernel, the same data model is used by both kernel and applications: ILP32. There is no 64-bit application support on the 32-bit kernel.

On the 64-bit kernel, two different kinds of applications are supported concurrently: 32-bit applications using the ILP32 data model, and 64-bit applications using the LP64 data model. The 64-bit kernel itself uses the LP64 data model.

These concepts are captured in three flags that are associated with every system call, including `ioctl`(2):

- Application data model is ILP32.
- Application data model is LP64.
- Application data model is *native*.

At first sight, the most useful question to answer about the application invoking the kernel is: "Is it ILP32 or LP64?" However, a better test is: "Is the application using the same model as the kernel, or a different model?" The concept of native data model serves to answer that question; it is conditionally defined to match the data

model of the kernel implementation. This approach enables you to write substantially cleaner code.

`ioctl(9E)` Considerations

Most driver entry points are managed by the 32-bit and 64-bit system caller handlers in the kernel in such a way that a driver does not need to be concerned about whether it is performing an operation on behalf of a 32-bit or a 64-bit application.

However, `ioctl`(9E) offers a direct connection between applications and the kernel. It enables a driver to implement device-specific operations. That is, it can cause the driver to perform a device-specific command or to pass arbitrary data between the driver and the application.

The third argument to `ioctl( )` is either a simple integral value or a pointer to some other value, typically a data structure. The data structure might be different in size and alignment between a 32-bit and a 64-bit application. Because the form of the interface between the driver and application is generally a private agreement between the driver and the application, the kernel cannot intervene to automatically translate the data structures. It cannot even tell whether the argument is an integer or a pointer.

Therefore, drivers and STREAMS modules need to know how to interpret the data structures passed in from an application.

CHAPTER 3

Multithreading

This chapter describes the locking primitives and thread synchronization mechanisms of the Solaris multithreaded kernel.

Locking Primitives

In traditional UNIX systems, every section of kernel code runs until it explicitly gives up the processor by calling `sleep`(1) or is interrupted by hardware. This is not true in the Solaris operating environment. A kernel thread can be preempted at any time to run another thread. Because all kernel threads share kernel address space, and often need to read and modify the same data, the kernel provides a number of locking primitives to prevent threads from corrupting shared data. These mechanisms include mutual exclusion locks (or mutex), readers/writer locks, and semaphores.

Storage Classes of Driver Data

The storage class of data is a guide to whether the driver might need to take explicit steps to control access to the data. The three types of data storage classes are:

- Automatic (stack) data – Every thread has a private stack, drivers never need to lock automatic variables.
- Global and static data – Global and static data can be shared by any number of threads in the driver; the driver might need to lock this type of data at times.
- Kernel heap data – Any number of threads in the driver might share kernel heap data, such as data allocated by `kmem_alloc`(9F). If this data is shared, the driver needs to protect it at times.

Mutual-Exclusion Locks

A mutual-exclusion lock, or mutex, is usually associated with a set of data and regulates access to that data. Mutexes provide a way to allow only one thread at a time access to that data.

TABLE 3–1 Mutex Routines

Name	Description
mutex_init(9F)	Initializes a mutex.
mutex_destroy(9F)	Releases any associated storage.
mutex_enter(9F)	Acquires a mutex.
mutex_tryenter(9F)	Acquires a mutex if available; but does not block.
mutex_exit(9F)	Releases a mutex.
mutex_owned(9F)	Tests to determine if the mutex is held by the current thread. To be used in `ASSERT(9F)` only.

Setting Up Mutexes

Device drivers usually allocate a mutex for each driver data structure. The mutex is typically a field in the structure and is of type `kmutex_t`. `mutex_init`(9F) is called to prepare the mutex for use. This is usually done at `attach`(9E) time for per-device mutexes and `_init`(9E) time for global driver mutexes.

For example,

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

For a more complete example of mutex initialization see Chapter 5.

The driver must destroy the mutex with `mutex_destroy`(9F) before being unloaded. This is usually done at `detach`(9E) time for per-device mutexes and `_fini`(9E) time for global driver mutexes.

Using Mutexes

Every section of the driver code that needs to read or write the shared data structure must do the following:

- Acquire the mutex
- Access the data
- Release the mutex

The scope of a mutex—the data it protects—is entirely up to the programmer. A mutex protects some particular data structure *because the programmer chooses to do so* and uses it accordingly. A mutex protects a data structure only if every code path that accesses the data structure does so while holding the mutex.

Readers / Writer Locks

A *readers/writer lock* regulates access to a set of data. The readers / writer lock is so called because many threads can hold the lock simultaneously for reading, but only one thread can hold it for writing.

Most device drivers do not use readers / writer locks. These locks are slower than mutexes and provide a performance gain only when protecting data that is not frequently written but is commonly read by many concurrent threads. In this case, contention for a mutex could become a bottleneck, so using a readers / writer lock might be more efficient. See `rwlock`(9F) for more information.

TABLE 3–2 Readers / Writer Locks

Name	Description
rw_init(9F)	Initializes a readers / writer lock
rw_destroy(9F)	Destroys a readers / writer lock
rw_enter(9F)	Acquires a readers / writer lock
rw_tryenter(9F)	Attempts to acquire a reader / writer lock without waiting
rw_tryupgrade(9F)	Attempts to upgrade readers / writer lock holding from reader to writer
rw_downgrade(9F)	Downgrades a readers / writer lock holding from writer to reader

TABLE 3–2 Readers / Writer Locks *(continued)*

Name	Description
rw_exit(9F)	Releases a readers / writer lock
rw_read_locked(9F)	Determines whether readers / writer lock is held for read or write

Semaphores

Counting semaphores are available as an alternative primitive for managing threads within device drivers. See `semaphore`(9F) for more information.

TABLE 3–3 Semaphores

Name	Description
sema_init(9F)	Initialize a semaphore
sema_destroy(9F)	Destroys a semaphore
sema_p(9F)	Decrement semaphore and possibly block
sema_tryp(9F)	Attempt to decrement semaphore, but do not block
sema_p_sig(9F)	Decrement semaphore, but do not block if signal is pending
sema_v(9F)	Increment semaphore and possibly unblock waiter

Thread Synchronization

In addition to protecting shared data, drivers often need to synchronize execution among multiple threads.

Condition Variables

Condition variables are a standard form of thread synchronization. They are designed to be used with mutexes. The associated mutex is used to ensure that a condition can be checked atomically, and that the thread can block on the associated condition variable without missing either a change to the condition or a signal that the condition has changed.

Table 3–4 lists the `condvar`(9F) interfaces.

TABLE 3–4 Condition Variable Routines

Name	Description
cv_init(9F)	Initializes a condition variable
cv_destroy(9F)	Destroys a condition variable
cv_wait(9F)	Waits for condition
cv_timedwait(9F)	Waits for condition or timeout
cv_wait_sig(9F)	Waits for condition or return zero on receipt of a signal
cv_timedwait_sig(9F)	Waits for condition or timeout or signal
cv_signal(9F)	Signals one thread waiting on the condition variable
cv_broadcast(9F)	Signals all threads waiting on the condition variable

Initializing Condition Variables

Declare a condition variable (type `kcondvar_t`) for each condition. Usually, this is done in the driver's soft-state structure. Use `cv_init`(9F) to initialize each one. Similar to mutexes, condition variables are usually initialized at `attach`(9E) time. For example:

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

For a more complete example of condition variable initialization see Chapter 5.

Waiting for the Condition

To use condition variables, follow these steps in the code path waiting for the condition:

1. Acquire the mutex guarding the condition.
2. Test the condition.
3. If the test results do not allow the thread to continue, use `cv_wait`(9F) to block the current thread on the condition. `cv_wait`(9F) releases the mutex before blocking. Upon return from `cv_wait`(9F) (which will reacquire the mutex before returning), repeat the test.
4. Once the test allows the thread to continue, set the condition to its new value. For example, set a device flag to busy.
5. Release the mutex.

Signaling the Condition

Follow these steps in the code path signaling the condition:

1. Acquire the mutex guarding the condition.
2. Set the condition.
3. Signal the blocked thread with `cv_signal`(9F).
4. Release the mutex.

Code Example 3–1 uses a busy flag, and mutex and condition variables to force the `read`(9E) routine to wait until the device is no longer busy before starting a transfer.

CODE EXAMPLE 3–1 Using Mutexes and Condition Variables

```
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_timedwait(9F)

If a thread blocks on a condition with cv_wait(9F), and that condition does not occur, it can wait forever. One way to prevent this is to establish a callback with timeout(9F). This callback sets a flag indicating that the condition did not occur normally, and then unblocks the thread. The notified thread then notices that the condition did not occur and can return an error.

A better solution is to use cv_timedwait(9F). An absolute wait time is passed to cv_timedwait(9F), which returns -1 if the time is reached and the event has not occurred. It returns a positive value if the condition is met.

cv_timedwait(9F) requires an absolute wait time expressed in clock ticks since the system was last rebooted. This can be determined by retrieving the current value with ddi_get_lbolt(9F). The driver usually has a maximum number of seconds or microseconds to wait, so this value is converted to clock ticks with drv_usectohz(9F) and added to the value from ddi_get_lbolt(9F).

Code Example 3–2 shows how to use cv_timedwait(9F) to wait up to five seconds to access the device before returning EIO to the caller.

CODE EXAMPLE 3–2 Using cv_timedwait(9F)

```
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 signalled.
                 */
                /* tidy up and exit */
                mutex_exit(&xsp->mu);
                return (EIO);
        }
}
xsp->busy = 1;
mutex_exit(&xsp->mu);
```

cv_wait_sig(9F)

There is always the possibility that either the driver accidentally waits for a condition that will never occur or that the condition will not happen for a long time. In either case, the user can abort the thread by sending it a signal. Whether the signal causes the driver to wake up depends upon the driver.

cv_wait_sig(9F) allows a signal to unblock the thread. This allows the user to break out of potentially long waits by sending a signal to the thread with kill(1) or by typing the interrupt character. cv_wait_sig(9F) returns zero if it is returning because of a signal, or nonzero if the condition occurred.

Code Example 3–3 shows how to use cv_wait_sig(9F) to allow a signal to unblock the thread.

CODE EXAMPLE 3–3 Using cv_wait_sig(9F)

```
mutex_enter(&xsp->mu);
while (xsp->busy) {
        if (cv_wait_sig(&xsp->cv, &xsp->mu) == 0) {
        /* Signalled 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(9F)

cv_timedwait_sig(9F) is similar to cv_timedwait(9F) and cv_wait_sig(9F), except that it returns -1 without the condition being signaled after a timeout has been reached, or 0 if a signal (for example, kill(2)) is sent to the thread.

For both cv_timedwait(9F) and cv_timedwait_sig(9F), time is measured in absolute clock ticks since the last system reboot.

Choosing a Locking Scheme

The locking scheme for most device drivers should be kept straightforward. Using additional locks allows more concurrency but increases overhead. Using fewer locks is less time consuming but allows less concurrency. Generally, use one mutex per data structure, a condition variable for each event or condition the driver must wait for, and a mutex for each major set of data global to the driver. Avoid holding mutexes for long periods of time.

- Use the multithreading semantics of the entry point to your advantage.
- Make all entry points re-entrant and reduce the amount of shared data by changing static variable to automatic.
- If your driver acquires multiple mutexes, acquire and release the mutexes in the same order in all code paths.
- Hold and release locks within the same functional space.
- Avoid holding driver mutexes when calling DDI interfaces which can block, for example, kmem_alloc(9F) with KM_SLEEP.

To look at lock usage, use lockstat(1M). lockstat(1M) monitors all kernel lock events, gathers frequency and timing data about the events, and displays the data.

See the *Multithreaded Programming Guide* for more details on multithreaded operations.

Potential Pitfalls

Here is a set of mutex-related panics:

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

Mutexes are not re-entrant by the same thread. If you already own the mutex, you cannot own it again. Doing this leads to this panic.

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

Releasing a mutex that the current thread does not hold causes this panic.

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

This panic occurs only on a uniprocessor. It indicates that a spin mutex is held and it will spin forever, because there is no other CPU to release it. This could happen because the driver forgot to release the mutex on one code path, or blocked while holding it.

A common cause of this panic is that the device's interrupt is high-level and is calling a routine that blocks the interrupt handler while holding a spin mutex. This is obvious if the driver explicitly calls `cv_wait`(9F), but might not be so if it's blocking while grabbing an adaptive mutex with `mutex_enter`(9F).

CHAPTER 4

Properties

Device attribute (or device-related) information may be represented with a *name=value* pair notation called a *property*.

For example, a *reg* property is used to represent device registers and onboard memory. The *reg* property is a software abstraction that describes device hardware registers; its value encodes the device register address location and size. Drivers use the *reg* property to access device registers.

As another example, an *interrupt* property is a software abstraction used to represent the device interrupt; its value encodes the device-interrupt pin number.

The value of a property can be one of five types:

- A byte array that has an arbitrary length and whose value is a series of bytes
- An integer property whose value is an integer
- An integer array property whose value is an array of integers
- A string property whose value is a NULL-terminated string
- A string array property whose value is a list of NULL-terminated strings

A property that has no value is known as a Boolean property. It is considered to be true if it exists and false if it doesn't exist.

Property Names

Strictly speaking, DDI/DKI software property names are not restricted in any way; however, there are certain recommended uses. As defined in IEEE 1275-1994 (the Standard for Boot Firmware), a property "is a human readable text string consisting of one to thirty-one printable characters. Property names *shall* not contain upper case

characters or the characters "/", "\", ":", "[", "]" and "@". Property names beginning with the character "+" are reserved for use by future revisions of IEEE 1275-1994." By convention, underscores are not used in property names; use a hyphen (-) instead. Also by convention, property names ending with the question mark character (`auto-boot?`) contain values that are strings, typically true or false.

Looking up Properties

A driver can request a property from its parent, which in turn might ask its parent. The driver can control whether the request can go higher than its parent.

For example, the "esp" driver maintains an integer property for each target called `targetx-sync-speed`, where "`x`" is the target number. The `prtconf`(1M) command in its verbose mode displays driver properties. The following example shows a partial listing for the "esp" driver.

```
% prtconf -v
...
       esp, instance #0
            Driver software properties:
                name <target2-sync-speed> length <4>
                    value <0x00000fa0>.
...
```

Table 4–1 provides information on the property interfaces.

TABLE 4–1 Property Interface Uses

Family	Property Interfaces	Description
ddi_prop_lookup	ddi_prop_exists (9F)	Looks up property and returns success if one exists. Returns failure if one does not exist.
	dd_prop_get_int(9F)	Looks up and returns an integer property.
	ddi_prop_lookup_int_array(9F)	Looks up and returns an integer array property.
	ddi_prop_lookup_string(9F)	Looks up and returns a string property.
	ddi_prop_lookup_string_array(9F)	Looks up and returns a string array property.
	ddi_prop_lookup_byte_array(9F)	Looks up and returns a byte array property.

TABLE 4–1 Property Interface Uses *(continued)*

Family	Property Interfaces	Description
ddi_prop_update	ddi_prop_update_in(9F)	Updates an integer property.
	ddi_prop_update_int_array(9F)	Updates an integer array property.
	ddi_prop_update_string(9F)	Updates a string property.
	ddi_prop_update_string_array(9F)	Updates an string array property.
	ddi_prop_update_byte_array(9F)	Updates a byte array property.
ddi_prop_remove	ddi_prop_remove(9F)	Removes a property.
	ddi_prop_remove_all(9F)	Removes all properties associated with a device.

prop_op(9E)

The `prop_op`(9E) entry point reports the values of device properties to the system. In many cases, the `ddi_prop_op`(9F) routine may be used as the driver's `prop_op`(9E) entry point in the `cb_ops`(9S) structure. `ddi_prop_op`(9F) performs all of the required processing and is sufficient for drivers that do not need to perform any special processing when handling a device property request.

However, there are cases when it is necessary for the driver to provide a `prop_op`(9E) entry point. For example, if a driver maintains a property whose value changes frequently, updating the property with `ddi_prop_update`(9F) each time it changes may not be efficient. Instead, the driver can maintain a local *copy* of the property in a C variable. The driver updates the C variable when the value of the property changes and does not call one of the `ddi_prop_update`(9F) routines. In this case, the `prop_op`(9E) entry point would need to intercept requests for this property and call one of the `ddi_prop_update`(9F) routines to update the value of the property before passing the request to `ddi_prop_op`(9F) to process the property request.

In Code Example 4–1, `prop_op`(9E) intercepts requests for the `temperature` property. The driver updates a variable in the state structure whenever the property changes but only updates the property when a request is made. It then uses the system routine `ddi_prop_op`(9F) to process the property request. If the property request is not specific to a device, the driver does not intercept the request. This is indicated when the value of the dev parameter is equal to `DDI_DEV_T_ANY` (the wildcard device number).

CODE EXAMPLE 4–1 prop_op(9E) Routine

```
static int
xxprop_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) {
                ddi_prop_update_int(dev, dip, name, temperature);
        }

        /* other cases */
}
```

CHAPTER 5

Autoconfiguration

Autoconfiguration is the process of getting the driver's code and static data loaded into memory and registered with the system. Autoconfiguration also involves configuring (attaching) individual device instances that are controlled by the driver. "Loadable Driver Interfaces" on page 65 and "Device Configuration Concepts" on page 67 discuss these processes in more detail.

Driver Loading and Unloading

Figure 5–1 illustrates a structural overview of a device driver. The shaded area of this figure highlights the driver data structures required for driver loading and the interfaces exported by the driver, which is subdivided into two parts: driver loading (performed by the kernel) and driver configuration.

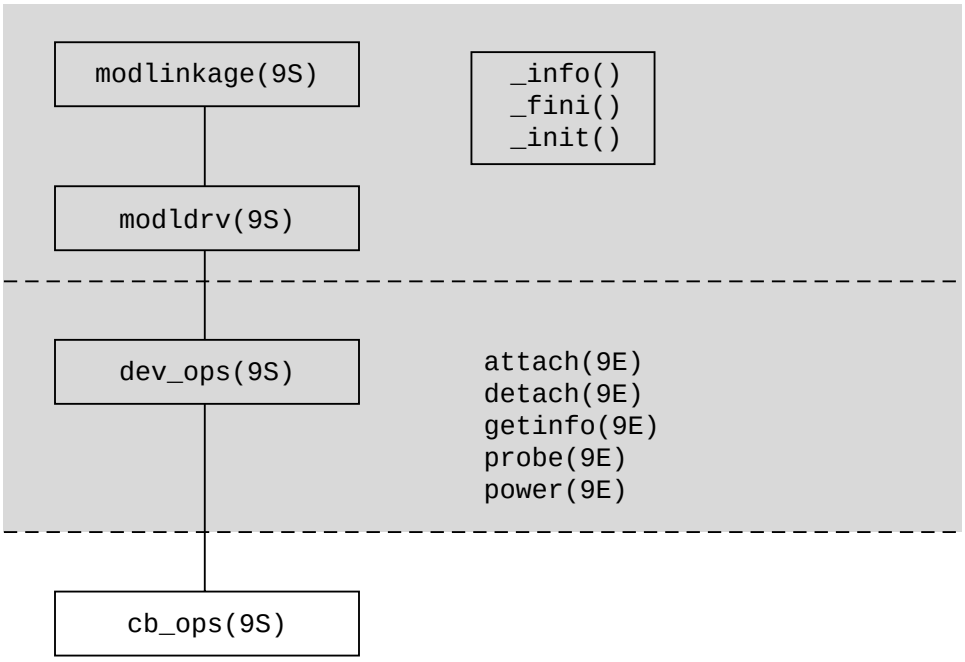


Figure 5–1 Module Loading and Autoconfiguration Entry Points

Data Structures

Drivers are required to statically intitialize a number of data structures to support autoconfiguration. These strucutres include `modlinkage`(9S), `modldrv`(9S), `dev_ops`(9S), and `cb_ops`(9S) if the driver is not a SCSI HBA.

The data structures illustrated in Figure 5-1 must be provided and initialized correctly for the driver to load and for its routines to be called. If an operation is not supported by the driver, the address of the routine `nodev`(9F) can be used to fill it in. If the driver supports the entry point, but does not need to do anything except return success, the address of the routine `nulldev`(9F) can be used.

Note - These structures should be initialized at compile-time. They should not be accessed or changed by the driver at any other time.

---

modlinkage Structure

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

The first field is the version number of the module loading subsystem and should be MODREV_1. The second field points to driver's modldrv structure defined next. The last element of the structure should always be NULL.

modldrv Structure

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

This structure describes the module in more detail. The first field provide information on how to install and un-install the module. It should be set to &mod_driverops for driver modules. The second field is a string to be displayed by modinfo(1M). It should contain sufficient information for identifying the version of source code which generated the driver binary. The last field points to the driver's dev_ops structure defined next.

dev_ops Structure

```
static struct dev_ops xx_dev_ops = {
        DEVO_REV,       /* devo_rev, */
        0,              /* devo_refcnt  */
        xxgetinfo,      /* getinfo(9E) */
        nulldev,        /* identify(9E) */
        xxprobe,        /* probe(9E) */
        xxattach,       /* attach(9E) */
        xxdetach,       /* detach(9E) */
        nodev,          /* devo_reset */
        &xx_cb_ops,     /* devo_cb_ops */
        NULL,           /* devo_bus_ops */
        &xxpower        /* power(9E) */
};
```

The dev_ops(9S) structure allows the kernel to find the autoconfiguration entry points of the device driver. The devo_rev field identifies the revision number of the structure itself, and must be set to DEVO_REV. The devo_refcnt field must be initialized to zero. The function address fields should be filled in with the address of the appropriate driver entry point. Exceptions are:

- If a probe(9E) routine is not needed, set the dev_ptr field to nulldev(9F).

- `identify`(9E) is obsolete and no longer required. Set this field to nulldev(9F).
- Set `devo_reset` to `nodev`(9F).
- Drivers for devices that provide Power Management functionality must have a `power`(9E) entry point. If a `power`(9E) routine is not needed, set this field to `NULL`.

The `devo_cb_ops` member should include the address of the `cb_ops`(9S) structure. The `devo_bus_ops` field must be set to `NULL`.

`cb_ops` Structure

```
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 */
};
```

The `cb_ops`(9S) structure contains the entry points for the character and block operations of the device driver. Any entry points the driver does not support should be initialized to `nodev`(9F). For example, character device drivers should set all the block-only fields, such as `cb_stategy`, to `nodev`(9F). Note that the `mmap`(9E) entry point is maintained for compatibility with previous releases, and drivers should use the `devmap`(9E) entry point for device memory mapping. If `devmap`(9E) is supported, set `mmap`(9E) to `nodev`(9F).

The `streamtab` field is used to determine if this is a STREAMS-based driver. The device drivers discussed in this book are not STREAMS based. For a non-STREAMS-based driver, it *must* be set to `NULL`.

The `cb_flag` member contains the following flags:

- The `D_MP` flag, indicates that the driver is safe for multi-threading. Solaris 8 operating environment only supports thread safe drivers, so, `D_MP` must be set.
- If the driver properly handles 64-bit offsets, it should set the `D_64BIT` flag in the `cb_flag` field. This specifies that the driver will use the `uio_loffset` field of the `uio`(9S) structure.
- If the driver supports the `devmap`(9E) entry point, it should set the `D_DEVMAP` flag. For information on `devmap`(9E), see Chapter 12.

`cb_rev` is the `cb_ops`(9S) structure revision number. This field must be set to CB_REV.

Loadable Driver Interfaces

Device drivers must be dynamically loadable and should be unloadable to help conserve memory resources. Drivers that can be unloaded are also easier to test and debug.

Each device driver has a section of code that defines a loadable interface. This code section defines a static pointer for the soft state routines, the structures described in “Data Structures” on page 62, and the routines involved in loading the module.

CODE EXAMPLE 5–1 Loadable Interface Section

```
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
};

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
_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(9E)

```
static void *xxstatep;
int
_init(void)
{
        int error;
        const int max_instance = 20;    /* max possible 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);
}
```

The driver should perform any one-time resource allocation or data initialization during driver loading in `_init`(9E). For example, it should initialize any mutexes global to the driver in this routine. The driver should not, however, use `_init(9E)` to allocate or initialize anything that has to do with a particular instance of the device. Per-instance initialization must be done in `attach`(9E). For example, if a driver for a printer can handle more than one printer at the same time, it should allocate resources specific to each printer instance in `attach(9E)`.

Note - Once _init(9E) has called mod_install(9F), the driver should not change any of the data structures hanging off the modlinkage(9S) structure, as the system may make copies of them or change them.

_fini(9E)

```
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);
}
```

Similarly, in _fini(9E), the driver should release any resources that were allocated in _init() and must remove itself from the system module list.

_info(9E)

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

The driver is called to return module information. The entry point should be implemented as shown above.

Device Configuration Concepts

Each driver must define the following entry points that are used by the kernel for device configuration:

- probe(9E)
- attach(9E)

- detach(9E)
- getinfo(9E)

Every device driver must have an attach(9E) and getinfo(9E) routine. probe(9E) is only required for non self-identifying devices. For self-identifying devices an explicit probe routine may be provided or nulldev(9F) may be specified in the dev_ops structure for the probe(9E) entry point.

Device Instances and Instance Numbers

The system assigns an instance number to each device. The driver may not reliably predict the value of the instance number assigned to a particular device. The driver should retrieve the particular instance number that has been assigned by calling ddi_get_instance(9F).

Instance numbers represent the system's notion of devices. Each dev_info (that is, each node in the device tree) for a particular driver is assigned an instance number by the kernel. Furthermore, instance numbers provide a convenient mechanism for indexing data specific to a particular physical device. The most common usage for this is ddi_get_soft_state(9F) which uses an instance number to retrieve soft state data for a particular physical device.

Minor Nodes and Minor Numbers

Drivers are allowed to manage their minor number name space. For example the sd driver needs to export 16 minor nodes (8 character, 8 block) to the file system for each disk. Each represents a different piece of the same disk, or a different interface to the same data (character / block). However, the driver still needs to be able to retrieve the instance number of the device in order to get soft state, and so forth.

probe(9E)

For non-self-identifying devices, this entry point should determine whether the hardware device is present on the system.

For probe to determine whether the instance of the device is present, probe(9E) may need to do many of the things also commonly done by attach(9E). In particular, it may need to map the device registers.

Probing the device registers is device specific. The driver probably has to perform a series of tests of the hardware to assure that the hardware is really there. The test criteria must be rigorous enough to avoid misidentifying devices. It may, for example, appear that the device is present when in fact it is not, because a different device appears to behave like the expected device.

DDI_PROBE_SUCCESS if the probe was successful

DDI_PROBE_FAILURE if the probe failed

DDI_PROBE_DONTCARE if the probe was unsuccessful, yet attach(9E) should still be called

DDI_PROBE_PARTIAL if the instance is not present now, but may be present in the future

For a given device instance, attach(9E) will not be called before probe(9E) has succeeded at least once on that device.

It is important that probe(9E) free all the resources it allocates, because it may be called multiple times; however, attach(9E) will not necessarily be called even if probe(9E) succeeds.

ddi_dev_is_sid(9F) may be used in a driver's probe(9E) routine to determine if the device is self-identifying. This is useful in drivers written for self-identifying and non-self-identifying versions of the same device.

Code Example 5–2 is a sample probe(9E) routine for devices on these buses.

CODE EXAMPLE 5–2 probe(9E) Routine

```
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;

    /*
     * 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 attrributes 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);
}
```

When the driver's probe(9E) routine is called, it does not know whether the device being probed exists on the bus. Therefore, it is possible that the driver may attempt to access device registers for a nonexistent device. A bus fault may be generated on some buses as a result.

Code Example 5–3 shows a probe(9E) routine that uses ddi_peek(9F) to check for the existence of the device.

CODE EXAMPLE 5–3 probe(9E) Routine Using ddi_poke8(9F)

```
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;

    /*
     * 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 attrributes 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);

    /*
     * The bus can generate a fault when probing for devices which
     * do not exist.  Use ddi_poke8(9f) to handle any faults which
     * may 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);
}
```

In this example, `ddi_regs_map_setup`(9F) is used to map the device registers. `ddi_peek8`(9F) reads a single character from the location `reg_addr`.

attach(9E)

The kernel calls a drivers `attach`(9E) entry point to attach an instance of a device or to resume operation for an instance of a device which has been suspended or shut down by the power management framework. In this section we will only discuss the operation of attaching device instances, the power management discussion is left to Chapter 9.

A drivers `attach(9E)` entry point is called to attach each instance of a device that is bound to the driver. The entry point is called with the instance of the device node to attach, with DDI_ATTACH specified as the `cmd` argument to `attach`(9E). The attach entry point will typically include the following types of processing:

- Allocating a soft state structure for the device instance
- Initializing per-instance mutexes and condition variables
- Registering the device's interrupts
- Mapping the registers and memory of the device instance
- Creating minor device nodes for the device instance
- Reporting the device instance has attached

Driver Soft State Management

To assist device driver writers in allocating state structures, the Solaris 8 DDI/DKI provides a set of memory management routines called the *software state management routines* (also known as the *soft state routines*). These routines dynamically allocate, retrieve, and destroy memory items of a specified size, and hide the details of list

management. An *instance number* is used to identify the desired memory item; this number can be (and usually is) the instance number assigned by the system.

Drivers will typically allocate a soft state structure for each device instance which attaches to the driver by calling `ddi_soft_state_zalloc`(9F), passing the instance number of the device. Since it is considered an error for there to be two device nodes with the same instance number, `ddi_soft_state_zalloc`(9F) will fail if an allocation already exists for a given instance number.

A drivers character or block entry point (`cb_ops`(9S)) will reference a particular soft state structure by first decoding the devices instance number from the `dev_t` argument that is passed to the entry point function. The driver then calls `ddi_get_soft_state`(9F) passing the per-driver soft state list and the instance number that was derived. If `ddi_get_soft_state`(9F) returns a NULL value, the driver should treat this as the device does not exist and return the appropriate code.

See "Creating Minor Device Nodes" on page 72 for additional information on how instance numbers and device numbers, or `dev_t`'s, are related.

Lock and Conditional Variable Initialization

Drivers should initialize any per-instance locks and condition variables during attach. The initialization of any locks which are acquired by the drivers interrupt handler **must** be initialized prior to adding any interrupt handlers. See Chapter 3 for a complete description of lock initialization and usage. See Chapter 7 for a discussion of the issues surrounding interrupt handler and locks.

Creating Minor Device Nodes

An important part of the attach process is the creation of minor nodes for the device instance. A `minor node` contains the information exported by the device and the DDI framework which the system uses to create a *special file* for the minor node under `/devices`.

Minor nodes are created when the driver calls `ddi_create_minor_node`(9F). The driver supplies a *minor number*, a *minor name*, a *minor node type*, and weather the minor node represents a block or character device.

Drivers can choose to create as many or as few minor nodes for a device as it wants to. Solaris expects certain classes of devices to have minor nodes created in a particular format. For example, disk drivers are expected to create 16 minor nodes for each physical disk instance attached; 8 minor nodes are created, representing the `a - h` block device interfaces, with an additional 8 minor nodes for the `a,raw - h,raw` character device interfaces.

The *minor number* passed to `ddi_create_minor_node`(9F) is defined wholly by the driver itself; the minor number is usually an encoding of the devices instance number with a minor node identifier. Taking the above example, the driver creates minor

numbers for each of the minor nodes by taking the devices instance number, shifting it left 3 bits, and OR'ing in the minor node index whose values range from 0 to 15.

The *minor node type* passed to `ddi_create_minor_node`(9F) classifies the type of device, such as disks, tapes, network interfaces, frame buffers, and so forth.

TABLE 5–1 Possible Node Types

Constant	Description
`DDI_NT_SERIAL`	Serial port
`DDI_NT_SERIAL_DO`	Dialout ports
`DDI_NT_BLOCK`	Hard disks
`DDI_NT_BLOCK_CHAN`	Hard disks with channel or target numbers
`DDI_NT_CD`	ROM drives (CD-ROM)
`DDI_NT_CD_CHAN`	ROM drives with channel or target numbers
`DDI_NT_FD`	Floppy disks
`DDI_NT_TAPE`	Tape drives
`DDI_NT_NET`	Network devices
`DDI_NT_DISPLAY`	Display devices
`DDI_NT_MOUSE`	Mouse
`DDI_NT_KEYBOARD`	Keyboard
`DDI_NT_AUDIO`	Audio Device
`DDI_PSEUDO`	General pseudo devices

The node types `DDI_NT_BLOCK`, `DDI_NT_BLOCK_CHAN`, `DDI_NT_CD`, and `DDI_NT_CD_CHAN` cause `devfsadm`(1M) to identify the device instance as a disk

and to create a symbolic link in the /dev/dsk or /dev/rdsk directory pointing to the device node in the /devices directory tree.

The node type DDI_NT_TAPE causes devfsadm(1M) to identify the device instance as a tape and to create a symbolic link from the /dev/rmt directory to the device node in the /devices directory tree.

The node types DDI_NT_SERIAL and DDI_NT_SERIAL_DO causes ports(1M) to identify the device instance as a serial port and to create symbolic links from the /dev/term and /dev/cua directories to the device node in the /devices directory tree and to entries to the port monitor database/etc/inittab.

Vendor-supplied strings should include an identifying value to make them unique, such as their name or stock symbol (if appropriate). The string can be used in conjunction with devfsadm(1M) and devlink.tab(4) to create logical names in /dev.

Deferred Attach

open(9E) might be called on a minor device before attach(9E) has succeeded on the corresponding instance. open(9E) must then return ENXIO, which will cause the system to attempt to attach the device. If the attach succeeds, the open is retried automatically.

CODE EXAMPLE 5–4 Example attach(9E) Entry Point

```
/*
 * 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;

/*
 * 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,
                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);
        }

}
```

detach(9E)

The kernel calls a driver's `detach`(9E) entry point to detach an instance of a device or to suspend operation for an instance of a device by power management. In this section we discuss the operation of detaching device instances, refer to Chapter 9 for a discussion of power management issues.

A drivers `detach`(9E) entry point is called to detach an instance of a device that is bound to the driver. The entry point is called with the instance of the device node to detach and `DDI_DETACH` specified as the *cmd* argument to the entry point.

A driver is required to cancel or wait for any time-outs or callbacks to complete, then release any resources which are allocated to the device instance before returning. If for some reason a driver cannot cancel outstanding callbacks for free resources, the driver is required to return the device to its original state and return `DDI_FAILURE` from the entry point, leaving the device instance in the attached state.

There are two types of callback routines - those which can be canceled and those which cannot. `timeout`(9F) and `bufcall`(9F) callbacks can be atomically cancelled by the driver during `detach`(9E). Other types of callbacks such as `scsi_init_pkt`(9F) and `ddi_dma_buf_bind_handle`(9F) cannot be canceled, requiring the driver to either block in `detach`(9E) until the callback completes or to fail the request to detach.

CODE EXAMPLE 5–5 `detach(9E)` Routine

```
/*
 * 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);
        /* FALLTHRU */

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

getinfo(9E)

The system calls `getinfo`(9E) to obtain configuration information that only the driver knows. The mapping of minor numbers to device instances is entirely under the control of the driver. The system sometimes needs to ask the driver which device a particular `dev_t` represents.

`getinfo`(9E) is called during module loading and at other times during the life of the driver. It can take one of two commands as its *infocmd* argument: `DDI_INFO_DEVT2INSTANCE`, which asks for a device's instance number, and

DDI_INFO_DEVT2DEVINFO, which asks for pointer to the device's dev_info structure.

In the DDI_INFO_DEVT2INSTANCE case, *arg* is a dev_t, and getinfo(9E) must translate the minor number to an instance number. In the following example, the minor number *is* the instance number, so it simply passes back the minor number. In this case, the driver must not assume that a state structure is available, since getinfo(9E) may be called before attach(9E). The mapping the driver defines between minor device number and instance number does not necessarily follow the mapping shown in the example. In all cases, however, the mapping must be static.

In the DDI_INFO_DEVT2DEVINFO case, *arg* is again a dev_t, so getinfo(9E) first decodes the instance number for the device. It then passes back the dev_info pointer saved in the driver's soft state structure for the appropriate device. This is shown in Code Example 5–6.

CODE EXAMPLE 5–6 getinfo(9E) Routine

```
/*
 * 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 instace 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);
}
```

Device IDs

The Solaris DDI provides interfaces allows drivers to provide a persistent unique identifier for a device, a *device ID*, which can be used to identify or locate a device and which is independent of the devices name or number (`dev_t`). Applications can use the functions defined in `libdevid`(3LIB) to read and manipulate the device IDs registered by the drivers.

Before a driver can export a *device ID*, it needs to verify that the device is capable of either providing a unique ID, such as WWN, or is capable of storing a host-generated unique ID in an area not accessible through normal operations, such as device NVRAM, reserved sectors, etc.

Registering Device IDs

Drivers will typically initialize and register device IDs in the drivers `attach`(9E) handler. As mentioned above, the driver is responsible for registering a *device ID* which is persistent. As such, the driver may be required to handle both devices which can provide a unique ID directly (WWN), and devices where fabricated IDs are written to and read from stable storage.

Registering a Device-Supplied ID

If the device can supply the driver with an identifier that is unique, the driver can simply initialize the *device ID* with this identifier and register the ID with the Solaris DDI.

```
/*
 * The device provides a guaranteed unique identifier,
 * in this case a SCSI3-WWN.  The WWN for the device has been
 * stored in the devices 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);
```

Registering a Fabricated ID

A driver may also register device IDs for devices which do not directly supply a unique ID. If the device is capable of storing and retrieving a small amount of data in a reserved area, the driver can create a fabricated device ID and write it to the reserved area.

```
/*
 * the device doesn't supply a unique ID, attempt to read
 * a fabricated ID from the devices 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);
```

Unregistering Device IDs

Drivers will typically unregister and free any *device IDs* they allocated as part of the `detach`(9E) handling. The driver will first call `ddi_devid_unregister`(9F) to unregister the *device ID* for the device instance. The driver must then free the *device ID* handle itself by calling `ddi_devid_free`(9F), passing the handle which had been returned by `ddi_devid_init`(9F). The driver is responsible for managing any space allocated for WWN or Serial Number data.

CHAPTER 6

Device Access — Programmed I/O

The Solaris operating environment provides driver developers with a comprehensive set of interfaces for accessing device memory. These interfaces are designed to shield the driver from platform-specific dependencies by handling mismatches between processor and device endianness as well as enforcing any data ordering dependencies the device might have. By using these interfaces, a single source driver can be developed that runs on both that SPARC and IA processor architectures as well as the various platforms from each respective processor family.

Device Memory

Devices that support programmed I/O are assigned one or more regions of bus address space that map to addressable regions of the device. These mappings are described as pairs of values in the 'reg' property associated with the device. Each value pair describes a segment of a bus address.

Drivers identify a particular bus address mapping by specifying the register number, or '`regspec`', which is an index into the devices '`reg`' property that identifies a (`busaddr`, `size`) pair. Drivers pass the register number when making calls to DDI functions such as `ddi_regs_map_setup`(9F). Drivers can determine how many mappable regions have been assigned to the device by calling `ddi_dev_nregs`(9F).

Managing Differences in Device and Host Endianness

The data format of the host can have different endian characteristics than the data format of the device. If this is the case, data transferred between the host and device

needs to be byte swapped to conform to the data format requirements of the destination location. Other devices can have the same endian characteristics of the host and require no byte swapping of the data.

Drivers specify the endian characteristics of the device by setting the appropriate flag in the `ddi_device_acc_attr`(9S) structure that is passed to `ddi_regs_map_setup`(9F). The DDI framework then performs any required byte swapping when the drivers calls `ddi_getX(9f)` or `ddi_putX(9f)` to read or write device memory.

Managing Data Ordering Requirements

Platforms can choose to reorder loads and stores of data to optimize performance of the platform. Since reordering might not be allowed by certain devices, the driver is required to specify the device's ordering requirements when setting up mappings to the device.

`ddi_device_acc_attr(9S)`

This structure describes the endian and data ordering requirements of the device. The driver is required to initialize and pass one of these structures as an argument to `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` — Specify `DDI_DEVICE_ATTR_V0`.

`devacc_attr_endian_flags` — Describes the endian characteristics of the device. Specified as a bit value whose possible values are:

- `DDI_NEVERSWAP_ACC` – Never swap data.
- `DDI_STRUCTURE_BE_ACC` – The device data format is big-endian.
- `DDI_STRUCTURE_LE_ACC` – The device data format is little-endian.

`devacc_attr_dataorder` — Describes the order in which the CPU must reference data as required by the device. Specified as an enumerated value, where data access restrictions are ordered from most to least strict.

- DDI_STRICTORDER_ACC – The host must issue the references in order, as specified by the programmer. This is the default behavior.
- DDI_UNORDERED_OK_ACC – The host is allowed to reorder loads and stores to device memory.
- DDI_MERGING_OK_ACC – The host is allowed to merge individual stores to consecutive locations. This setting also implies reordering.
- DDI_LOADCACHING_OK_ACC – The host is allowed to read data from the device until a store occurs.
- DDI_STORECACHING_OK_ACC – The host is allowed to cache data written to the device and defer writing it to the device until some future time.

Note - The system can access data more strictly than the driver specified in devacc_attr_dataorder. The restriction to the host diminishes while moving from strict data ordering to cache storing in terms of data accesses by the driver.

Mapping Device Memory

Drivers typically map all regions of a device during attach(9E). The driver maps a region of device memory by calling ddi_regs_map_setup(9F), specifying the register number of the region to map, the device access attributes for the region, an offset, and size to further define the mapping. The DDI framework sets up the mappings for the device region and returns an opaque handle to the driver. This data access handle is passed as an argument to the ddi_get8(9F) or ddi_put8(9F) family of routines when reading or writing data to that region of the device.

The driver verifies that the shape of the device mappings match what the driver is expecting by checking the number of mappings exported by the device. It calls ddi_dev_nregs(9F). It verifies the size of each mapping by calling ddi_dev_regsize(9F).

Mapping Setup

Code Example 6–1 is a simple example demonstrating the setup and use of the DDI data access interfaces. This driver is for a fictional Little Endian device that accepts

one character at a time and generates an interrupt when ready for another. This device implements two register sets—the first, an 8-bit CSR register, and the second, an 8-bit data register.

CODE EXAMPLE 6–1

```
#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);
    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);
}
```

Device Access Functions

Drivers use the `ddi_get8`(9F) and `ddi_put8`(9F) family of routines in conjuction with the handle returned by `ddi_regs_map_setup`(9F) to transfer data to and from a device. The DDI framework automatically handles any byte swapping that is required to meet host or device endian formats along with enforcing any store ordering constraints the device might have.

The DDI provides interfaces for transferring data in 8, 16, 32, and 64 bit quantities, as well as interfaces for transferring multiple values repeatedly. See `ddi_get8`(9F), `ddi_put8`(9F), `ddi_rep_get8`(9F) and `ddi_rep_put8`(9F) families of routines for a complete listing and description of these interfaces.

Code Example 6–2 builds on Code Example 6–1 where the driver mapped the device's CSR and data registers. Here, the driver's `write`(9E) entry point, when called, will write a buffer of data to the device one byte at a time.

CODE EXAMPLE 6–2

```
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));

    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 devices 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);
}
```

Alternate Device Access Interfaces

While it is both preferable and possible for a driver to implement all device accesses using the `ddi_get8`(9F) and `ddi_put8`(9F) family of interfaces, Solaris provides interfaces specific to particular bus implementations. While these functions are more efficient on some platforms, use of these routines can limit the ability of the driver to remain portable across different bus versions of the device.

Memory Space Access

With memory mapped access, device registers appear in memory address space. The `ddi_putX(9F)` and `ddi_getX(9F)` family of interfaces are available for use by drivers as an alternative to the standard device access interfaces.

I/O Space Access

With I/O space access, the device registers appear in I/O space, where each addressable element is called an I/O port. The `ddi_io_get8`(9F) and `ddi_io_put8`(9F) family of routines are available for use by drivers as an alternative to the standard device access interfaces.

PCI Configuration Space Access

To access PCI configration space without using the normal device access interfaces, a driver is required to map PCI configuration space by calling `pci_config_setup`(9F) in place of `ddi_regs_map_setup`(9F). The driver can then call the `pci_config_get8`(9F) and `pci_config_put8`(9F) family of interfaces to access PCI configuration space.

CHAPTER 7

Interrupt Handlers

Interrupt Handler Overview

An interrupt is a hardware signal from a device to a CPU. It tells the CPU that the device needs attention and that the CPU should stop performing what it is doing and respond to the device. If a CPU is available (it is not performing a task with higher priority), it suspends the current thread and eventually invokes the interrupt handler for that device. The job of the interrupt handler is to service the device and stop it from interrupting. Once the handler returns, the CPU resumes what it was doing before the interrupt occurred.

The Solaris 8 DDI/DKI provides interfaces for registering and servicing interrupts.

Interrupt Specification

The *interrupt specification* is information the system uses to bind a device interrupt source with a specific device interrupt handler. The specification describes the information provided by the hardware to the system when making an interrupt request. Because an interrupt specification is bus specific, the information it contains varies from bus to bus.

Interrupt specifications typically include a bus-interrupt level. For vectored interrupts the specifications include an interrupt vector. On IA platforms the interrupt specification defines the relative interrupt priority of the device. Because interrupt specifications are bus specific, see `isa(4)`, `eisa(4)`, `sbus(4)`, and `pci(4)` for information on interrupt specifications for these buses.

Interrupt Number

When registering interrupts the driver must provide the system with an interrupt number. This interrupt number identifies the interrupt specification for which the driver is registering a handler. Most devices have one interrupt: interrupt number 0. However, there are devices that have different interrupts for different events. A communications controller may have one interrupt for receive ready and one for transmit ready. The device driver normally knows how many interrupts the device has, but if the driver has to support several variations of a controller, it can call `ddi_dev_nintrs`(9F) to find out the number of device interrupts.

Interrupt Block Cookies

The iblock cookie is an opaque data structure that represented the information the system needs on how to block interrupts and is returned from `ddi_get_iblock_cookie`(9F). This interface uses an interrupt number to return the iblock cookie associated with a specific interrupt source. The iblock cookie must be passed to `mutex_init`(9F) when initializing driver mutexes that will be used in the interrupt routine.

Device Interrupts

There are two common ways in which buses implement interrupts: *vectored* and *polled*. Both methods commonly supply a bus-interrupt priority level. However, vectored devices also supply an interrupt vector; polled devices do not.

High-Level Interrupts

Buses prioritize device interrupts at one of several *bus-interrupt levels*. These bus interrupt levels are then mapped to different processor-interrupt levels. A bus interrupt level that maps to a CPU interrupt priority level above the scheduler priority level is called a *high-level interrupt*. High-level interrupt handlers are restricted to what DDI interfaces they can call. In particular, the only DDI routines that high-level interrupt handlers are allowed to call are:

- `mutex_enter`(9F) and `mutex_exit`(9F) on a mutex initialized with an iblock cookie associated with the high-level interrupt
- `ddi_trigger_softintr`(9F)
- `ddi_getX`/`ddi_putX` family of routines

A bus-interrupt level by itself does not determine whether a device interrupts at high level: a given bus-interrupt level may map to a high-level interrupt on one platform, but map to an ordinary interrupt on another platform.

The driver can choose whether to support devices that have high-level interrupts, but it always has to check—it cannot assume that its interrupts are not high level. The function `ddi_intr_hilevel`(9F), given an interrupt number, returns a value indicating whether the interrupt is high level.

Normal Interrupts

The only information the system has about a device interrupt is either the bus interrupt priority level (IPL, on an SBus in a SPARC machine, for example) or the interrupt request number (IRQ on an ISA bus in an IA machine, for example).

When an interrupt handler is registered, the system adds the handler to a list of potential interrupt handlers for each IPL or IRQ. Once the interrupt occurs, the system must determine which device, of all the devices associated with a given IPL or IRQ, actually interrupted. It does this by calling all the interrupt handlers for the designated IPL or IRQ, until one handler *claims* the interrupt.

The SBus, ISA, EISA, and PCI buses are capable of supporting polled interrupts.

Software Interrupts

The Solaris 8 DDI/DKI supports software interrupts, also known as *soft interrupts*. Soft interrupts are not initiated by a hardware device; they are initiated by software. Handlers for these interrupts must also be added to and removed from the system. Soft interrupt handlers run in interrupt context and therefore can be used to do many of the tasks that belong to an interrupt handler.

Hardware interrupt handlers are supposed to perform their tasks quickly, since they may suspend other system activity while running. This is particularly true for high-level interrupt handlers, which operate at priority levels greater than that of the system scheduler. High-level interrupt handlers mask the operations of all lower-priority interrupts—including those of the system clock. Consequently, the interrupt handler must avoid involving itself in an activity (such as acquiring a mutex) that might cause it to sleep.

If the handler sleeps, then the system may hang because the clock is masked and incapable of scheduling the sleeping process. For this reason, high-level interrupt handlers normally perform a minimum amount of work at high-priority levels and delegate remaining tasks to software interrupts, which run below the priority level of the high-level interrupt handler. Because software interrupt handlers run below the priority level of the system scheduler, they can do the work that the high-level interrupt handler was incapable of doing.

Software interrupt handlers must not perform as if they have work to do when they run, since (like hardware interrupt handlers) they can run because some other driver triggered a soft interrupt. For this reason, the driver must indicate to the soft interrupt handler that it should do work before triggering the soft interrupt.

Registering Interrupts

Before a device driver can receive and service interrupts, it must register an interrupt handler with the system by calling `ddi_add_intr`(9F). Registering interrupts provides the system with a way to associate an interrupt handler with an interrupt specification. The interrupt handler is called when the device might have been responsible for the interrupt. It is the handler's responsibility to determine if it should handle the interrupt and, if so, claim it.

Note - There is a potential race condition between adding the interrupt handler and initializing mutexes. The interrupt routine is eligible to be called as soon as `ddi_add_intr`(9F) returns, as another device might interrupt and cause the handler to be invoked. This may result in the interrupt routine being called before any mutexes have been initialized with the returned interrupt block cookie. If the interrupt routine acquires the mutex before it has been initialized, undefined behavior may result. To ensure that this race condition does not occur, always initialize mutexes and any other data used in the interrupt handler before adding the interrupt.

To register a driver's interrupt handler, the driver usually performs the following steps in `attach`(9E).

1. Test for high-level interrupts by calling `ddi_intr_hilevel`(9F) to find out if the interrupt specification maps to a high-level interrupt. If it does, one possibility is to post a message to that effect and return `DDI_FAILURE`. See Code Example 7–1.
2. Get the iblock cookie by calling `ddi_get_iblock_cookie`(9F).
3. Initialize any associated mutexes with the iblock cookie by calling `mutex_init`(9F).
4. Register the interrupt handler by calling `ddi_add_intr`(9F).

   Code Example 7–1 shows how to install an interrupt handler.

CODE EXAMPLE 7–1 `attach(9E)` Routine Installing an Interrupt Handler

```
static int
   xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
   {
       struct xxstate *xsp;
       switch (cmd) {
       case DDI_ATTACH:
             ...
```

```
          if (ddi_intr_hilevel(dip, inumber) != 0){
            cmn_err(CE_CONT,
                "xx: high-level interrupts are not supported\n");
            return (DDI_FAILURE);
            }
            ddi_get_iblock_cookie(dip, inumber, &xsp->iblock_cookie);
          mutex_init(&xsp->mu, NULL, MUTEX_DRIVER,
                  (void *)xsp->iblock_cookie);
            cv_init(&xsp->cv, NULL, CV_DRIVER, NULL);
            if (ddi_add_intr(dip, inumber, NULL,
                    NULL, xxintr,
                      (caddr_t)xsp) != DDI_SUCCESS){
               cmn_err(CE_WARN, "xx: cannot add interrupt handler.");
               goto failed;
            }
            return (DDI_SUCCESS);

case DDI_RESUME:
              For information, see Chapter 9
    default:
            return (DDI_FAILURE);
    }
failed:
    remove interrupt handler if necessary, destroy mutex and condition variable
    return (DDI_FAILURE);
}
```

Interrupt Handlers

The interrupt handler has a set of responsibilities to perform. Some are required by the framework, and some are required by the device. All interrupt handlers are required to do the following:

- Determine if the device is interrupting and possibly reject the interrupt.

  The interrupt handler must first examine the device and determine if it has issued the interrupt. If it has not, the handler must return `DDI_INTR_UNCLAIMED`. This step allows the implementation of *device polling*: it tells the system whether this device, among a number of devices at the given interrupt priority level, has issued the interrupt.

- Inform the device that it is being serviced.

  This is a device-specific operation, but it is required for the majority of devices. For example, SBus devices are required to interrupt until the driver tells them to stop. This guarantees that all SBus devices interrupting at the same priority level will be serviced.

- Perform any I/O request-related processing.

  Devices interrupt for different reasons, such as *transfer done* or *transfer error*. This step may involve using data access functions to read the device's data buffer,

examine the device's error register, and set the status field in a data structure accordingly. Interrupt dispatching and processing are relatively time consuming.

- Do any additional processing that could save another interrupt, for example, read the next data from the device.
- Return `DDI_INTR_CLAIMED`.

Code Example 7–2 shows an interrupt routine.

CODE EXAMPLE 7–2 Interrupt Example

```
static uint_t
   xxintr(caddr_t arg)
   {
       struct xxstate *xsp = (struct xxstate *)arg;
       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);
   }
```

Most of the steps performed by the interrupt routine depend on the specifics of the device itself. Consult the hardware manual for the device to determine the cause of the interrupt, detect error conditions, and access the device data registers.

Handling High-Level Interrupts

High-level interrupts are those that interrupt at the level of the scheduler and above. This level does not allow the scheduler to run; therefore, high-level interrupt handlers cannot be preempted by the scheduler, nor can they rely on the scheduler (cannot block)—they can only use mutual exclusion locks for locking.

Because of this, the driver must use `ddi_intr_hilevel`(9F) to determine if it uses high-level interrupts. If `ddi_intr_hilevel`(9F) returns true, the driver can fail to attach, or it can use a two-level scheme to handle interrupts.

The suggested method is to add a high-level interrupt handler, which simply triggers a lower-priority software interrupt to handle the device. The driver should allow more concurrency by using a separate mutex for protecting data from the high-level handler.

High-level Mutexes

A mutex initialized with the interrupt block cookie that represents a high-level interrupt is known as a *high-level mutex*. While holding a high-level mutex, the driver is subject to the same restrictions as a high-level interrupt handler. The only routines it can call are:

- `mutex_exit`(9F) to release the high-level mutex
- `ddi_trigger_softintr`(9F) to trigger a soft interrupt

High-Level Interrupt Handling Example

In the example presented in Code Example 7–3, the high-level mutex (`xsp->high_mu`) is used only to protect data shared between the high-level interrupt handler and the soft interrupt handler. This includes a queue that the high-level interrupt handler appends data to (and the low-level handler removes data from), and a flag that indicates the low-level handler is running. A separate low-level mutex (`xsp->low_mu`) protects the rest of the driver from the soft interrupt handler.

CODE EXAMPLE 7–3 `attach(9E)` Routine Handling High-Level Interrupts

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        struct xxstate *xsp;
        ...
        if (ddi_intr_hilevel(dip, inumber)) {
            ddi_get_iblock_cookie(dip, inumber,
```

```
                &xsp->high_iblock_cookie);
            mutex_init(&xsp->high_mu, NULL, MUTEX_DRIVER,
                (void *)xsp->high_iblock_cookie);
            if (ddi_add_intr(dip, inumber, &xsp->high_iblock_cookie,
                &xsp->high_idevice_cookie, xxhighintr, (caddr_t)xsp)
                != DDI_SUCCESS)
                goto failed;
            ddi_get_soft_iblock_cookie(dip, DDI_SOFTINT_HI,
                &xsp->low_iblock_cookie)
            mutex_init(&xsp->low_mu, NULL, MUTEX_DRIVER,
                (void *)xsp->low_iblock_cookie);
            if (ddi_add_softintr(dip, DDI_SOFTINT_HI, &xsp->id,
                &xsp->low_iblock_cookie, NULL,
                xxlowintr, (caddr_t)xsp) != DDI_SUCCESS)
                goto failed;
        } else {
            add normal interrupt handler
        }
        cv_init(&xsp->cv, NULL, CV_DRIVER, NULL);
        ...
        return (DDI_SUCCESS);
    failed:
        free allocated resources, remove interrupt handlers
        return (DDI_FAILURE);
}
```

The high-level interrupt routine services the device, and enqueues the data. The high-level routine triggers a software interrupt if the low-level routine is not running, as Code Example 7–4 demonstrates.

CODE EXAMPLE 7–4 High-level Interrupt Routine

```
static uint_t
xxhighintr(caddr_t arg)
{
        struct xxstate    *xsp = (struct xxstate *)arg;
            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 and queue the data for the 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)
             ddi_trigger_softintr(xsp->id);
     return (DDI_INTR_CLAIMED);
}
```

The low-level interrupt routine is started by the high-level interrupt routine triggering a software interrupt. Once running, it should continue to do so until there is nothing left to process, as Code Example 7–5 shows.

CODE EXAMPLE 7–5 Low-level Interrupt Routine

```
static uint_t
xxlowintr(caddr_t arg)
{
     struct xxstate *xsp = (struct xxstate *)arg;
     ....
     mutex_enter(&xsp->low_mu);
     mutex_enter(&xsp->high_mu);
        if (xsp->softint_count > 1) {
            xsp->softint_count--;
            return (DDI_INTR_CLAIMED);
    }
       if (queue empty) {
             mutex_exit(&xsp->high_mu);
             mutex_exit(&xsp->low_mu);
             return (DDI_INTR_UNCLAIMED);
     }
     xsp->softint_running = 1;
     while (data on queue) {
              ASSERT(mutex_owned(&xsp->high_mu);
          dequeue data from high-level queue;
            mutex_exit(&xsp->high_mu);
       normal interrupt processing
            mutex_enter(&xsp->high_mu);
     }
     xsp->softint_running = 0;
    xsp->softint_count = 0;
     mutex_exit(&xsp->high_mu);
     mutex_exit(&xsp->low_mu);
   return (DDI_INTR_CLAIMED);
}
```

CHAPTER 8

Direct Memory Access (DMA)

Many devices can temporarily take control of the bus and perform data transfers to (and from) main memory or other devices. Since the device is doing the work without the help of the CPU, this type of data transfer is known as *direct memory access* (DMA). DMA transfers can be performed between two devices, between a device and memory, or between memory and memory. This chapter explains transfers between a device and memory only.

DMA Model

The Solaris Device Driver Interface/Driver-Kernel Interface (DDI/DKI) provides a high-level, architecture-independent model for DMA. This allows the framework (the DMA routines) to hide such architecture-specific details as:

- Setting up DMA mappings
- Building scatter-gather lists
- Ensuring I/O and CPU caches are consistent

There are several abstractions that are used in the DDI/DKI to describe aspects of a DMA transaction. These include:

- *DMA object* – Memory that is the source or destination of a DMA transfer.
- *DMA handle* – An opaque object returned from a successful `ddi_dma_alloc_handle`(9F) call. The DMA handle can be used in subsequent DMA subroutine calls to refer to such DMA objects.
- *DMA cookie* – A `ddi_dma_cookie`(9S) structure (`ddi_dma_cookie_t`) describes a contiguous portion of a DMA object that is entirely addressable by the device. It contains DMA addressing information required to program the DMA engine.

Rather than knowing that a platform needs to map an *object* (typically a memory buffer) into a special DMA area of the kernel address space, device drivers instead allocate DMA *resources* for the object. The DMA routines then perform any platform-specific operations needed to set the object up for DMA access. The driver receives a DMA *handle* to identify the DMA resources allocated for the object. This handle is opaque to the device driver; the driver must save the handle and pass it in subsequent calls to DMA routines, but should not interpret it in any way.

Operations are defined on a DMA handle that provide the following services:

- Manipulating DMA resources
- Synchronizing DMA objects
- Retrieving attributes of the allocated resources

Types of Device DMA

Devices perform one of the following three types of DMA.

- Bus-Master DMA
- Third-party DMA
- First-party DMA

Bus-Master DMA

If the device is capable of acting as a true *bus master* (where the DMA engine resides on the device board), the driver should program the device's DMA registers directly. The transfer address and count are obtained from the DMA cookie and given to the device.

Third-party DMA

Third-party DMA utilizes a system DMA engine resident on the main system board, which has several DMA channels available for use by devices. The device relies on the system's DMA engine to perform the data transfers between the device and memory. The driver uses DMA engine routines (see `ddi_dmae`(9F)) to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

First-party DMA

Under first-party DMA, the device drives its own DMA bus cycles using a channel from the system's DMA engine. The `ddi_dmae_1stparty`(9F) function is used to configure this channel in a cascade mode so that the DMA engine will not interfere with the transfer.

Types of Host Platform DMA

The platform that the device operates on provides one of two types of memory access: direct memory access (DMA) or direct virtual memory access (DVMA).

On platforms that support DMA, the system provides the device with a physical address in order to perform transfers. In this case, the transfer of a DMA object can actually consist of a number of physically discontiguous transfers. An example of this occurs when an application transfers a buffer that spans several contiguous virtual pages that map to physically discontiguous pages. To deal with the discontiguous memory, devices for these platforms usually have some kind of scatter-gather DMA capability. Typically, IA systems provide physical addresses for direct memory transfers.

On platforms that support DVMA, the system provides the device with a virtual address to perform transfers. In this case, the underlying platform provides some form of memory management unit (MMU) that translates device accesses to these virtual addresses into the proper physical addresses. The device transfers to and from a contiguous virtual image that can be mapped to discontiguous physical pages. Devices that operate in these platforms don't need scatter-gather DMA capability. Typically, SPARC platforms provide virtual addresses for direct memory transfers.

DMA Software Components: Handles, Windows, and Cookies

A DMA *handle* is an opaque pointer representing an object (usually a memory buffer or address) where a device can perform DMA transfers. Several different calls to DMA routines use the handle to identify the DMA resources allocated for the object.

An object represented by a DMA handle is completely covered by one or more *DMA cookies*. A DMA cookie represents a contiguous piece of memory to or from which the DMA engine can transfer data. The system uses the information in a DMA attribute `ddi_dma_attr(9S)` structure provided by the driver, as well as the

memory location and alignment of the target object, to decide how to divide an object into multiple cookies.

If the object is too big to fit the request within system resource limitations, it has to be broken up into multiple *DMA windows*. Only one window is activated at one time and has resources allocated. The `ddi_dma_getwin`(9F) function is used to position between windows within an object. Each DMA window consists of one or more DMA cookies. For more information, see "DMA Windows" on page 117.

Scatter-Gather

Some DMA engines can accept more than one cookie. Such engines perform scatter-gather I/O without the help of the system. In this case, it is most efficient if the driver uses `ddi_dma_nextcookie`(9F) to get as many cookies as the DMA engine can handle and program them all into the engine. The device can then be programmed to transfer the total number of bytes covered by all these DMA cookies combined.

DMA Operations

The steps involved in a DMA transfer are similar among the types of DMA. The sections below present methods for performing DMA transfers.

Note - It is not necessary to ensure the DMA object is locked in memory in block drivers for buffers coming from the file system, as the file system has already locked the data in memory.

Bus-Master DMA

In general, the driver should perform the following steps for bus-master DMA.

1. Describe the DMA attributes. This allows the routines to ensure that the device will be able to access the buffer.
2. Allocate a DMA handle.
3. Ensure the DMA object is locked in memory (see `physio`(9F) or `ddi_umem_lock`(9F)).
4. Allocate DMA resources for the object.
5. Program the DMA engine on the device and start it (this is device specific). When the transfer is complete, continue the bus master operation.

6. Perform any required object synchronizations.
7. Release the DMA resources.
8. Free the DMA handle.

First-Party DMA

In general, the driver should perform the following steps for first-party DMA.

1. Allocate a DMA channel.
2. Configure the channel with `ddi_dmae_1stparty`(9F).
3. Ensure the DMA object is locked in memory (see `physio`(9F) or `ddi_umem_lock`(9F)).
4. Allocate DMA resources for the object.
5. Program the DMA engine on the device and start it (this is device specific). When the transfer is complete, continue the bus-master operation.
6. Perform any required object synchronizations.
7. Release the DMA resources.
8. Deallocate the DMA channel.

Third-Party DMA

In general, the driver should perform these steps for third-party DMA.

1. Allocate a DMA channel.
2. Retrieve the system's DMA engine attributes with `ddi_dmae_getattr`(9F).
3. Lock the DMA object in memory (see `physio`(9F) or `ddi_umem_lock`(9F)).
4. Allocate DMA resources for the object.
5. Program the system DMA engine to perform the transfer with `ddi_dmae_prog`(9F).
6. Perform any required object synchronizations.
7. Stop the DMA engine with `ddi_dmae_stop`(9F).
8. Release the DMA resources.
9. Deallocate the DMA channel.

Certain hardware platforms restrict DMA capabilities in a bus-specific way. Drivers should use `ddi_slaveonly`(9F) to determine if the device is in a slot in which DMA is possible.

DMA Attributes

DMA attributes describe the built-in attributes and limits of a DMA engine, including:

- Limits on addresses the device can access
- Maximum transfer count
- Address alignment restrictions

To ensure that DMA resources allocated by the system can be accessed by the device's DMA engine, device drivers must inform the system of their DMA engine limitations using a `ddi_dma_attr`(9S) structure. The system might impose additional restrictions on the device attributes, but it never removes any of the driver-supplied restrictions.

`ddi_dma_attr` Structure

The DMA attribute structure has the following members:

```
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;
```

`dma_attr_version` — Version number of the attribute structure. It should be set to DMA_ATTR_V0.

`dma_attr_addr_lo` — Lowest bus address that the DMA engine can access

`dma_attr_addr_hi` — Highest bus address that the DMA engine can access

`dma_attr_count_max` — Specifies the maximum transfer count that the DMA engine can handle in one cookie. The limit is expressed as the maximum count minus one. It is used as a bit mask, so it must also be one less than a power of two.

`dma_attr_align` — Specifies additional alignment requirements for any allocated DMA resources. This field can be used to force more restrictive alignment than implicitly specified by other DMA attributes, such as alignment on a page boundary.

`dam_attr_burstsizes`	Specifies the *burst sizes* that the device supports. A burst size is the amount of data the device can transfer before relinquishing the bus. This member is a binary encoding of burst sizes, assumed to be powers of two. For example, if the device is capable of doing 1-, 2-, 4-, and 16-byte bursts, this field should be set to 0 x 17. The system also uses this field to determine alignment restrictions.
`dma_attr_minxfer`	Minimum effective transfer size the device can perform. It also influences alignment and padding restrictions.
`dma_attr_maxxfer`	Describes the maximum number of bytes that the DMA engine can transmit or receive in one I/O command. This limitation is only significant if it is less than (`dma_attr_count_max + 1) * dma_attr_sgllen`.
`dma_attr_seg`	Upper bound of the DMA engine's address register. This is often used where the upper 8 bits of an address register are a latch containing a segment number, and the lower 24 bits are used to address a segment. In this case, `dma_attr_seg` would be set to 0xFFFFFF, and prevents the system from crossing a 24-bit segment boundary when allocating resources for the object.
`dma_attr_sgllen`	Specifies the maximum number of entries in the scatter-gather list. It is the number of cookies that the DMA engine can consume in one I/O request to the device. If the DMA engine has no scatter-gather list, this field should be set to one.
`dma_attr_granular`	Field describes the granularity of the device's DMA transfer ability, in units of bytes. This value is used to specify, for example, the sector size of a mass storage device. DMA requests will be broken into multiples of this value. If there is no scatter-gather capability, then the size of each DMA transfer will be a multiple of this value. If there is scatter-gather capability, then a single segment will not be smaller than the minimum transfer value, but can be less than the granularity; however the total transfer length of

	the scatter-gather list will be a multiple of the granularity value.
`dma_attr_flags`	This field can be set to `DDI_DMA_FORCE_PHYSICAL`, which indicates that the system should return physical rather than virtual I/O addresses if the system supports both. If the system does not support physical DMA, the return value from `ddi_dma_alloc_handle(9F)` will be `DDI_DMA_BADATTR`. In this case, the driver has to clear `DDI_DMA_FORCE_PHYSICAL` and retry the operation.

SBus Example

A DMA engine on an SBus in a SPARC machine has the following attributes:

- It can access only addresses ranging from 0xFF000000 to 0xFFFFFFFF.
- It has a 32-bit DMA counter register.
- It can handle byte-aligned transfers.
- It supports 1-, 2- and 4-byte burst sizes.
- It has a minimum effective transfer size of 1 byte.
- It has a 32-bit address register.
- It doesn't have a scatter-gather list.
- The device operates on sectors only (for example a disk).

The resulting attribute structure is:

```
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 xfer size */
    0xFFFFFFFF,        /* address register max */
    1,                 /* no scatter-gather */
    512,               /* device operates on sectors */
    0,                 /* attr flag: set to 0 */
};
```

ISA Bus Example

A DMA engine on an ISA bus in an IA machine has the following attributes:

- It accesses only the first 16 megabytes of memory.

- It cannot cross a 1 megabyte boundary in a single DMA transfer.
- It has a 16-bit counter register.
- It can handle byte-aligned transfers.
- It supports 1-, 2- and 4-byte burst sizes.
- It has a minimum effective transfer size of 1 byte.
- It can hold up to 17 scatter-gather transfers.
- The device operates on sectors only (for example a disk).

The resulting attribute structure is:

```
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 xfer size */
    0x000FFFFF,        /* address register max */
    17,                /* scatter-gather */
    512,               /* device operates on sectors */
    0,                 /* attr flag: set to 0 */
};
```

Object Locking

Before allocating the DMA resources for a memory object, the object must be prevented from moving. If it is not, the system can remove the object from memory while the device is writing to it, causing the data transfer to fail, and possibly corrupting the system. The process of preventing memory objects from moving during a DMA transfer is known as *locking down the object*.

The following object types do not require explicit locking:

- Buffers coming from the file system through `strategy`(9E). These buffers are already locked by the file system.
- Kernel memory allocated within the device driver, such as that allocated by `ddi_dma_mem_alloc`(9F).

For other objects (such as buffers from user space), `physio`(9F) or `ddi_umem_lock`(9F) must be used to lock down the objects. This is usually performed in the `read`(9E) or `write`(9E) routines of a character device driver. See “Data Transfer Methods” on page 148 for an example.

Allocating a DMA Handle

A DMA handle is an opaque object that is used as a reference to subsequently allocated DMA resources. It is usually allocated in the driver's attach entry point using `ddi_dma_alloc_handle`(9F). `ddi_dma_alloc_handle`(9F) takes the device information referred to by `dip` and the device's DMA attributes described by a `ddi_dma_attr`(9S) structure as parameters.

```
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);
```

`dip`	Pointer to the device's `dev_info` structure
`attr`	Pointer to a `ddi_dma_attr`(9S) structure as described in "DMA Attributes" on page 101
`callback`	Address of the callback function for handling resource allocation failures
`arg`	Argument to be passed to the callback function
`handlep`	Pointer to a DMA handle to store the returned handle

Allocating DMA Resources

Two interfaces allocate DMA resources:

- `ddi_dma_buf_bind_handle`(9F) – Used with `buf(9S)` structures
- `ddi_dma_addr_bind_handle`(9F) – Used with virtual addresses

DMA resources are usually allocated in the driver's `xxstart()` routine, if one exists. See "Asynchronous Data Transfers" on page 178 for discussion of *xxstart*.

```
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);
```

The following arguments are common to both `ddi_dma_addr_bind_handle`(9F) and `ddi_dma_buf_bind_handle`(9F):

`handle`	DMA handle and the object for allocating resources

`flags`	Set of flags indicating the transfer direction and other attributes. `DDI_DMA_READ` indicates a data transfer from device to memory. `DDI_DMA_WRITE` indicates a data transfer from memory to device. See `ddi_dma_addr_bind_handle`(9F) or `ddi_dma_buf_bind_handle`(9F) for a complete discussion of the allowed flags.
`callback`	Address of callback function for handling resource allocation failures. See `ddi_dma_addr_bind_handle`(9F).
`arg`	Argument to pass to the callback function
`cookiep`	Pointer to the first DMA cookie for this object
`ccountp`	Pointer to the number of DMA cookies for this object

- For `ddi_dma_addr_bind_handle`(9F), the object is described by an address range, where `as` is a pointer to an address space structure (this must be `NULL`), `addr` is the base kernel address of the object, and `len` is the length of the object in bytes.
- For `ddi_dma_buf_bind_handle`(9F), the object is described by a `buf`(9S) structure pointed to by `bp`.

Device Register Structure

DMA capable devices have more registers than have been used in previous examples. This section adds the following fields to the device register structure to support DMA-capable device examples.

For DMA engines without scatter-gather support:

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

For DMA engines with scatter-gather support:

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


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

DMA Callback Example

In Code Example 8–1, `xxstart( )` is used as the callback function and the per-device state structure is given as its argument. `xxstart( )` attempts to start the command. If the command cannot be started because resources are not available, `xxstart( )` is scheduled to be called sometime later, when resources might be available.

Because `xxstart( )` is used as a DMA callback, it must follow these rules imposed on DMA callbacks:

- It must not assume that resources are available (it must try to allocate them again).
- It must indicate to the system whether allocation succeeded by returning DDI_DMA_CALLBACK_RUNOUT if it fails to allocate resources (and needs to be called again later) or DDI_DMA_CALLBACK_DONE indicating success (so no further callback is necessary).

CODE EXAMPLE 8–1 DMA Callback Example

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

Determining Maximum Burst Sizes

Drivers specify the DMA burst sizes their device supports in the `dma_attr_burstsizes` field of the `ddi_dma_attr`(9S) structure. This is a bitmap of the supported burst sizes. However, when DMA resources are allocated, the system might impose further restrictions on the burst sizes that might be actually used by the device. The `ddi_dma_burstsizes`(9F) routine can be used to obtain the allowed burst sizes. It returns the appropriate burst size bitmap for the device. When DMA resources are allocated, a driver can ask the system for appropriate burst sizes to use for its DMA engine.

CODE EXAMPLE 8–2 Determining Burst Size

```
#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
    } else {
         other cases
    }
```

Allocating Private DMA Buffers

Some device drivers might need to allocate memory for DMA transfers to or from a device, in addition to doing transfers requested by user threads and the kernel. Examples of this are setting up shared memory for communication with the device and allocating intermediate transfer buffers. `ddi_dma_mem_alloc`(9F) is provided for allocating memory for DMA transfers.

```
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);
```

`handle`	DMA handle
`length`	Length in bytes of the desired allocation
`accattrp`	Pointer to a device access attribute structure
`flags`	Data transfer mode flags; possible values are: `DDI_DMA_CONSISTENT` and `DDI_DMA_STREAMING`

`waitfp`	Address of callback function for handling resource allocation failures. See `ddi_dma_mem_alloc`(9F).
`arg`	Argument to pass to the callback function
`kaddrp`	Pointer (on a successful return) that contains the address of the allocated storage
`real_length`	Length in bytes that was allocated
`handlep`	Pointer to a data access handle

`flags` should be set to `DDI_DMA_CONSISTENT` if the device accesses in a nonsequential fashion, or if synchronization steps using `ddi_dma_sync`(9F) should be as lightweight as possible (because of frequent use on small objects). This type of access is commonly known as *consistent* access. I/O parameter blocks that are used for communication between a device and the driver are set up this way.

On the IA platform, to allocate memory for DMA using physically contiguous pages, set the length of the scatter/gather list `dma_attr_sgllen` in the `ddi_dma_attr`(9S) structure to 1, and do not specify `DDI_DMA_PARTIAL` which would otherwise permit partial resource allocation.

Code Example 8–3 shows how to allocate IOPB memory and the necessary DMA resources to access it. DMA resources must still be allocated, and the `DDI_DMA_CONSISTENT` flag must be passed to the allocation function.

CODE EXAMPLE 8–3 Using `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;
}
```

`flags` should be set to `DDI_DMA_STREAMING` if the device is doing sequential, unidirectional, block-sized and block-aligned transfers to or from memory. This type of access is commonly known as *streaming* access.

For example, if an I/O transfer can be sped up by using an I/O cache, which at a minimum transfers (flushes) one cache line, `ddi_dma_mem_alloc`(9F) will round the size to a multiple of the cache line to avoid data corruption.

`ddi_dma_mem_alloc`(9F) returns the actual size of the allocated memory object. Because of padding and alignment requirements, the actual size might be larger than the requested size. `ddi_dma_addr_bind_handle`(9F) requires the actual length.

`ddi_dma_mem_free`(9F) is used to free the memory allocated by `ddi_dma_mem_alloc`(9F).

Note - If the memory is not properly aligned, the transfer will succeed but the system will choose a different (and possibly less efficient) transfer mode that requires fewer restrictions. For this reason, `ddi_dma_mem_alloc(9F)` is preferred over `kmem_alloc(9F)` when allocating memory for the device to access.

Handling Resource Allocation Failures

The resource-allocation routines provide the driver with several options when handling allocation failures. The `waitfp` argument indicates whether the allocation routines will block, return immediately, or schedule a callback, as shown in Table 8–1.

TABLE 8–1 Resource Allocation Handling

waitfp	Indicated Action
DDI_DMA_DONTWAIT	Driver does not want to wait for resources to become available.
DDI_DMA_SLEEP	Driver is willing to wait indefinitely for resources to become available.
Other values	The address of a function to be called when resources are likely to be available.

Programming the DMA Engine

When the resources have been successfully allocated, the device must be programmed. Although programming a DMA engine is device specific, all DMA engines require a starting address and a transfer count. Device drivers retrieve these two values from the *DMA cookie* returned by a successful call from `ddi_dma_addr_bind_handle`(9F), `ddi_dma_buf_bind_handle`(9F), or `ddi_dma_getwin`(9F). These functions all return the first DMA cookie and a cookie count indicating whether the DMA object consists of more than one cookie. If the

cookie count N is greater than 1, ddi_dma_nextcookie(9F) has to be called N-1 times to retrieve all the remaining cookies.

A cookie is of type ddi_dma_cookie(9S) and has the following fields:

```
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 */
```

The dmac_laddress specifies a 64-bit I/O address appropriate for programming the device's DMA engine. If a device has a 64-bit DMA address register, a driver should use this field to program the DMA engine. The dmac_address field specifies a 32-bit I/O address that should be used for devices that have a 32-bit DMA address register. dmac_size contains the transfer count. Depending on the bus architecture, the dmac_type field in the cookie might be required by the driver. The driver should not perform any manipulations, such as logical or arithmetic, on the cookie.

CODE EXAMPLE 8–4 ddi_dma_cookie(9S) Example

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

Note - ddi_dma_addr_bind_handle(9F) and ddi_dma_buf_bind_handle(9F) can return more DMA cookies than fit into the scatter-gather list. In this case, the driver has to continue the transfer in the interrupt routine and reprogram the scatter-gather list with the remaining DMA cookies. You must handle sgllen cookies at a time.

Freeing the DMA Resources

After a DMA transfer is completed (usually in the interrupt routine), the driver can release DMA resources by calling `ddi_dma_unbind_handle`(9F).

As described in “Synchronizing Memory Objects” on page 115, `ddi_dma_unbind_handle`(9F) calls `ddi_dma_sync`(9F), eliminating the need for any explicit synchronization. After calling `ddi_dma_unbind_handle`(9F), the DMA resources become invalid, and further references to them have undefined results. Code Example 8–5 shows how to use `ddi_dma_unbind_handle`(9F).

CODE EXAMPLE 8–5 Freeing DMA Resources

```
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);
}
```

The DMA resources should be released and reallocated if a different object will be used in the next transfer. However, if the same object is always used, the resources can be allocated once and continually reused as long as there are intervening calls to `ddi_dma_sync`(9F).

Freeing the DMA Handle

When the driver is detached, the DMA handle must be freed. `ddi_dma_free_handle`(9F) destroys the DMA handle and any residual resources the system is caching on the handle. Any further references of the DMA handle will have undefined results.

Canceling DMA Callbacks

DMA callbacks cannot be canceled. This requires some additional code in the drivers `detach`(9E) routine, as it must not return `DDI_SUCCESS` if there are any outstanding callbacks. (See Code Example 8–6.) When DMA callbacks occur, the `detach`(9E) routine must wait for the callback to run and must prevent it from rescheduling itself. This can be done using additional fields in the state structure, as shown below.

CODE EXAMPLE 8–6 Canceling DMA Callbacks

```
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)
               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);
}
```

Synchronizing Memory Objects

At various points when the memory object is accessed (including the time of removal of the DMA resources), the driver might need to synchronize the memory object with respect to various caches. This section gives guidelines on when and how to synchronize memory objects.

Cache

Cache is a very high-speed memory that sits between the CPU and the system's main memory (CPU cache), or between a device and the system's main memory (I/O cache), as shown in Figure 8–1.

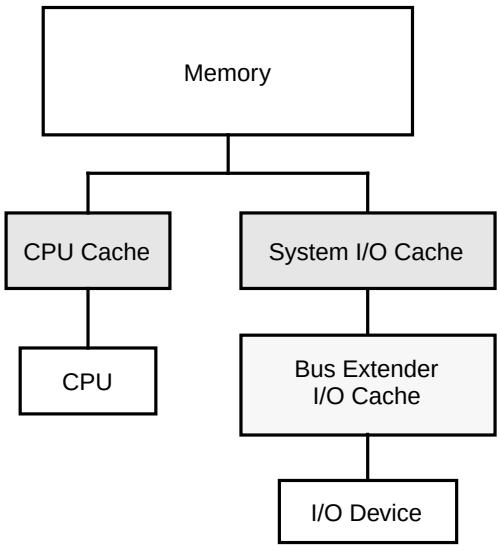


Figure 8–1 CPU and System I/O Caches

When an attempt is made to read data from main memory, the associated cache first determines whether it contains the requested data. If so, it quickly satisfies the request. If the cache does not have the data, it retrieves the data from main memory, passes the data on to the requestor, and saves the data in case that data is requested again.

Similarly, on a write cycle, the data is stored in the cache quickly and the CPU or device is allowed to continue executing (transferring). This takes much less time than it otherwise would if the CPU or device had to wait for the data to be written to memory.

An implication of this model is that after a device transfer has been completed, the data can still be in the I/O cache but not yet in main memory. If the CPU accesses the memory, it might read the wrong data from the CPU cache. To ensure a consistent view of the memory for the CPU, the driver must call a synchronization routine to flush the data from the I/O cache and update the CPU cache with the new data. Similarly, a synchronization step is required if data modified by the CPU is to be accessed by a device.

There might also be additional caches and buffers between the device and memory, such as caches associated with bus extenders or bridges. `ddi_dma_sync(9F)` is provided to synchronize *all* applicable caches.

`ddi_dma_sync(9F)`

If a memory object has multiple mappings—such as for a device (through the DMA handle), and for the CPU—and one mapping is used to modify the memory object,

the driver needs to call ddi_dma_sync(9F) to ensure that the modification of the memory object is complete before accessing the object through another mapping. ddi_dma_sync(9F) can also inform other mappings of the object that any cached references to the object are now stale. Additionally, ddi_dma_sync(9F) flushes or invalidates stale cache references as necessary.

Generally, the driver has to call ddi_dma_sync(9F) when a DMA transfer completes. The exception to this is that deallocating the DMA resources with ddi_dma_unbind_handle(9F), does an implicit ddi_dma_sync(9F) on behalf of the driver.

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

If the object is going to be read by the DMA engine of the device, the device's view of the object must be synchronized by setting *type* to DDI_DMA_SYNC_FORDEV. If the DMA engine of the device has written to the memory object, and the object is going to be read by the CPU, the CPU's view of the object must be synchronized by setting *type* to DDI_DMA_SYNC_FORCPU.

Here is an example of synchronizing a DMA object for the CPU:

```
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
}
```

If the only mapping that concerns the driver is one for the kernel (such as memory allocated by ddi_dma_mem_alloc(9F)), the flag DDI_DMA_SYNC_FORKERNEL can be used. This is a hint to the system that if it can synchronize the kernel's view faster than the CPU's view, it will do so; otherwise, it acts the same as DDI_DMA_SYNC_FORCPU.

DMA Windows

The system might be unable to allocate resources for a large object. If this occurs, the transfer must be broken into a series of smaller transfers. The driver can either do this itself, or it can let the system allocate resources for only part of the object, thereby creating a series of DMA *windows*. Allowing the system to allocate resources is the preferred solution, as the system can manage the resources more effectively than the driver.

A DMA window has attributes *offset* (from the beginning of the object) and *length*. After a partial allocation, only a range of *length* bytes starting at *offset* has resources allocated for it.

A DMA window is requested by specifying the DDI_DMA_PARTIAL flag as a parameter to ddi_dma_buf_bind_handle(9F) or ddi_dma_addr_bind_handle(9F). Both functions return DDI_DMA_PARTIAL_MAP if a window can be established. However, the system might allocate resources for the entire object (less overhead), in which case DDI_DMA_MAPPED is returned. The driver should check the return value (see Code Example 8–7) to determine whether DMA windows are in use.

CODE EXAMPLE 8–7 Setting Up DMA Windows

```
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);
        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);
}
```

Two functions operate with DMA windows. The first, ddi_dma_numwin(9F), returns the number of DMA windows for a particular DMA object. The other function, ddi_dma_getwin(9F), allows repositioning (reallocation of system resources) within the object. It shifts the current window to a new window within the object. Because ddi_dma_getwin(9F) reallocates system resources to the new window, the previous window becomes invalid.

Caution - It is a severe error to move the DMA windows with a call to `ddi_dma_getwin(9F)` before transfers into the current window are complete. Wait until the transfer to the current window is complete (when the interrupt arrives) then call `ddi_dma_getwin(9F)` or data will be corrupted.

`ddi_dma_getwin`(9F) is normally called from an interrupt routine; see Code Example 8–8. The first DMA transfer is initiated as a result of a call to the driver. Subsequent transfers are started from the interrupt routine.

The interrupt routine examines the status of the device to determine if the device completed the transfer successfully. If not, normal error recovery occurs. If the transfer was successful, the routine must determine if the logical transfer is complete (the entire transfer specified by the `buf`(9S) structure) or if this was only one DMA window. If it was only one window, it moves the window with `ddi_dma_getwin`(9F), retrieves a new cookie, and starts another DMA transfer.

If the logical request has been completed, the interrupt routine checks for pending requests and starts a transfer, if necessary. Otherwise, it returns without invoking another DMA transfer. Code Example 8–8 illustrates the usual flow control.

CODE EXAMPLE 8–8 Interrupt Handler Using DMA Windows

```
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);
}
```

CHAPTER 9

Power Management

Power management provides the ability to control and manage the electrical power usage of a computer system or device. Power management enables systems to conserve energy by using less power when idle and by shutting down completely when not in use. For example, desktop computer systems can use a significant amount of power, and often (particularly at night) are left idle. Power management software can detect that the system is not being used and power it or some of its components down.

Power Management Framework

The Solaris Power Management framework depends on device drivers to implement device-specific power management functionality. The framework is implemented in two parts:

- Device power management – Automatically turns off unused devices to reduce power consumption.
- System power management – Automatically turns off the computer when the entire system is idle.

Device Power Management

The framework allows devices to reduce their energy consumption after a specified idle time interval. To perform effective device power management, system software monitors the different devices and determines when they are not in use. Since only device drivers are able to determine when a device is idle, and only device drivers are able to reduce power consumption of a device, the Power Management

framework exports interfaces to enable communication between the system software and the device driver.

The Solaris Power Management framework provides the following:

- A device-independent model for power-manageable devices
- System software to implement a power management policy (which is controlled by a user-modifiable configuration file)
- A set of DDI interfaces for the device driver to notify the framework if the device can be power managed, and when it is idle or busy

System Power Management

System power management consists of turning off the entire computer after saving its state so that it can be returned to the same state immediately when it is turned back on.

To shut down an entire system and later return it to the state it was in prior to the shutdown, it is necessary to:

- Stop (and later restart) kernel threads and user processes.
- Save the hardware state of all devices on the system to disk (and later restore it).

SPARC platform only - System power management is currently implemented only on some SPARC systems supported by the Solaris 8 operating environment.

The Solaris operating environment System Power Management framework provides the following:

- A platform-independent model of system idleness.
- System software to implement a system power management policy (which is controlled by a user-modifiable configuration file).
- A set of interfaces for the device driver to override the method for determining which drivers have hardware state.
- A set of interfaces to allow the framework to call into the driver to save and restore the device state.
- A mechanism for notifying processes that a resume operation has occurred.

Device Power Management Model

The following sections describe the details of the device power management model. This model includes the following elements:

- Components
- Idleness
- Power levels
- Dependency
- Policy
- Device power management interfaces
- Power management entry points

Components

A device is power manageable if the power consumption of the device can be reduced when it is idle. Conceptually, a power manageable device consists of a number of power-manageable hardware units called components.

The device driver notifies the system of the existence of device components and the power levels that they support by creating a `pm-components`(9) property in its `attach`(9E) entry point as part of driver initialization.

Most devices which are power manageable implement only a single component. An example of a single-component, power-manageable device is a disk whose spindle motor can be stopped to save power when the disk is idle.

If a device has multiple power-manageable units that are separately controllable, it should implement multiple components.

An example of a two-component, power-manageable device is a frame buffer card with a monitor connected to it. Frame buffer electronics is the first component [component 0]. Its power consumption can be reduced when not in use. The monitor is the second component [component 1], which can also enter a lower power mode when not in use. The combination of frame buffer electronics and monitor is considered by the system as one device with two components.

Multiple Components

To the power management framework, all components are considered equal and completely independent of each other. If this is not true for a particular device, it is the responsibility of the device driver to ensure that undesirable state combinations do not occur. For example, with a frame buffer/monitor combination as described in the previous section, for each possible power state of the monitor (`On`, `Standby`, `Suspend`, `Off`) there are states of the frame buffer electronics (`D0`, `D1`, `D2`, `D3`) which are not allowed if the device is to work properly. If the monitor is `On`, then the frame buffer must be at `D0` (full on), so if the frame buffer driver gets a request to power up the monitor to `On` while the frame buffer is `D3`, it must ask the system to bring the frame buffer back up (by calling `pm_raise_power`(9F)) before setting the

monitor `On`. If the frame buffer driver gets a request from the system to lower the power of the frame buffer while the monitor is `On`, it must fail that request.

Idleness

Each component of a device may be in one of two states: *busy* or *idle*. The device driver notifies the framework of changes in the device state by calling `pm_busy_component`(9F) and `pm_idle_component`(9F). When components are initially created, they are considered idle.

Power Levels

From the `pm-components` property exported by the device, the Device Power Management framework knows what power levels the device supports. Power level values must be positive integers. The interpretation of power levels is determined by the device driver writer, but they must be listed in monotonically increasing order in the `pm-components` property, and a power level of 0 is interpreted by the framework to mean off. When the framework must power up a device because of a dependency, it will bring each component to its highest power level.

Code Example 9–1 is an example `pm-components` entry from the `.conf` file of a driver which implements a single power-managed component consisting of a disk spindle motor. The disk spindle motor is component 0 and it supports 2 power levels, which represent stopped and spinning full speed.

CODE EXAMPLE 9–1 Sample `pm-component` Entry

```
pm-components="NAME=Spindle Motor", "0=Stopped", "1=Full Speed";
```

Code Example 9–2 shows an example of how Code Example 9–1 could be implemented in the `attach( )` routine of the driver.

CODE EXAMPLE 9–2 `attach(9E)` Routine With `pm-components` Property

```
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;
...
```

Code Example 9–3 shows a frame buffer that implements two components. Component 0 is the frame buffer electronics that support 4 different power levels. Component 1 represents the state of power management of the attached monitor.

CODE EXAMPLE 9–3 Multiple Component `pm-components` Entry

```
pm-components="NAME=Frame Buffer", "0=Off", "1=Suspend", "2=Standby", "3=On",
        "NAME=Monitor", "0=Off", "1=Suspend", "2=Standby", "3=On";
```

When a device driver is first attached, the framework does not know the power level of the device. A power transition may occur when:

- The driver calls `pm_raise_power`(9F) or `pm_lower_power`(9F).
- The framework has lowered the power level of a component because it has exceeded its threshold time.

Once a power transition has occurred or the driver has informed the framework of the power level, the framework tracks the current power level of each component of the device. The driver can inform the framework of a power level change by calling `pm_power_has_changed`(9F).

The system calculates a default threshold for each possible transition from one power level to the next lower level, based on the system idleness threshold. These default thresholds can be overridden using `dtpower(1M)` or `power.conf(4)`. Another default threshold, based on the system idleness threshold, is used when the component power level is unknown.

Dependency

A device might depend on one or more other devices. A device depends on another device if it can be powered off only when all the components of all the devices it depends on are also powered off. For example, when the window system is not running, the frame buffer device depends on the keyboard device by default. When the window system is not running, the frame buffer components can only be powered off when the keyboard device is powered off.

The `power.conf`(4) file specifies the dependencies among devices. A parent node in the device tree implicitly depends upon its children. This dependency is handled automatically by the power management framework.

Policy

If automatic power management is enabled by `dtpower(1M)` or `power.conf`(4), then all devices with a `pm-components`(9) property will be automatically power managed. After each component has been idle for a default period, it will be automatically brought to its next lowest power level. The default period is calculated by the power management framework to get the entire device to its lowest power state within the system idleness threshold.

Note - By default automatic power management is enabled on all SPARC desktop systems first shipped after July 1, 1999. This feature is disabled by default for all other systems. To determine if automatic power management is enabled on your machine, refer to the `power.conf`(4) man page for instructions.

`dtpower(1M)` or `power.conf`(4) may be used to override the defaults calculated by the framework.

Device Power Management Interfaces

A device driver that supports a device with power-manageable components must notify the system of the existence of these components and the power levels that they support by creating a `pm-components`(9) property. This is typically done from the driver's `attach`(9E) entry point by calling `ddi_prop_update_string_array`(9F), but may be done from a `driver.conf`(4) file instead. See the `pm-components`(9) man page for details.

Busy-Idle State Transitions

The driver must keep the framework informed of device state transitions from idle to busy or busy to idle. Where these transitions happen is entirely device specific. The transitions from idle to busy and from busy to idle depend on the nature of the device and the abstraction represented by the specific component. For example, SCSI disk target drivers typically export a single component, which represents whether the SCSI target disk drive is spun up or not. It is marked busy whenever there is an outstanding request to the drive and idle when the last queued request finishes. Some components are created and never marked busy (components created by `pm-components`(9) are created in an idle state). For example, the keyboard and mouse are never marked busy but have their idle time reset each time a keystroke or mouse event is processed.

The following interfaces notify the power management framework of busy-idle state transitions.

pm_busy_component(9F)

```
int pm_busy_component(dev_info_t *dip, int component);
```

pm_busy_component(9F) marks *component* as busy. While the component is busy, it will not be powered off. If the component is already powered off, then marking it busy doesn't change its power level. The driver needs to call pm_raise_power(9F) for this purpose. Calls to pm_busy_component(9F) are stacked and require a corresponding number of calls to pm_idle_component(9F) to idle the component.

pm_idle_component(9F)

```
int pm_idle_component(dev_info_t *dip, int component);
```

pm_idle_component(9F) marks *component* as idle. An idle component is subject to being powered off. pm_idle_component(9F) must be called once for each call to pm_busy_component(9F) in order to idle the component.

Device Power State Transitions

A device driver can call pm_raise_power(9F) to request that a component be set to at least a given power level. This is necessary before using a component that has been powered off. For example, a SCSI disk target driver's read(9E) or write(9E) routine might need to spin up the disk if it had been powered off before completing the read or write. pm_raise_power(9F) requests the power management framework to initiate a device power state transition to a higher power level. Normally, reductions in component power levels are initiated by the framework. However, a device driver should call pm_lower_power(9F) when detaching, in order to reduce the power consumption of unused devices as much as possible.

pm_raise_power(9F)

```
int pm_raise_power(dev_info_t *dip, int component, int level);
```

pm_raise_power(9F) is called when the driver discovers that a component needed for some operation is at a power level less than is needed for a particular operation. This interface arranges for the driver to be called to raise the current power level of the component at least to the level specified in the request. All the devices that depend on this device are also brought back to full power by this call.

pm_lower_power(9F)

```
int pm_lower_power(dev_info_t *dip, int component, int level);
```

pm_lower_power(9F) is called when the device is detaching, once access to the device is no longer needed. It should be called for each component to set each component to its lowest power so that the device uses as little power as possible while it is not in use.

pm_power_has_changed(9F)

pm_power_has_changed(9F) is called to notify the framework when a device has made a power transition on its own, or to inform the framework of the power level of a device, for example, after a suspend-resume operation.

Entry Points Used by Device Power Management

The Power Management framework uses the power(9E) entry point.

power(9E)

```
int power(dev_info_t *dip, int component, int level);
```

The system calls the power(9E) entry point (either directly or as a result of a call to pm_raise_power( ) or pm_lower_power( )) when it determines that a component's current power level needs to be changed. The action taken by this entry point is device driver specific. In the example of the SCSI target disk driver mentioned previously, setting the power level to 0 results in sending a SCSI command to spin down the disk, while setting the power level to the full power level results in sending a SCSI command to spin up the disk.

If a power transition will cause the device to lose state, then the driver must ensure that any necessary state is saved in memory so that it can be restored when it is needed again. If a power transition will require that saved state be restored before the device can be used again, then the driver must restore that state. The framework makes no assumptions about what power transactions cause the loss of or require the restoration of state for automatically power-manage devices. Code Example 9–4 shows a sample power(9E) routine.

CODE EXAMPLE 9–4 power(9E) Routine for Single-Component Device

```
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);
}
```

Code Example 9–5 is a power(9E) routine for a device with two components, where component 0 must be on when component 1 is on.

CODE EXAMPLE 9–5 power(9E) Routine for Multiple Component Device

```
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);
        }
        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);
}
```

System Power Management Model

This section describes the details of the System Power Management model. The model includes the following components:

- Autoshutdown threshold
- Busy state
- Hardware state
- Policy
- Power management entry points

Autoshutdown Threshold

The system may be shut down (powered off) automatically after a configurable period of idleness. This period is known as the *autoshutdown threshold*. This behavior is enabled by default for SPARC desktop systems first shipped after October 1, 1995 and before July 1, 1999. It may be overridden using `dtpower(1M)` or `power.conf(4)`.

Busy State

There are several ways to measure the busy state of the system. The currently supported built-in metrics are keyboard characters, mouse activity, tty characters, load average, disk reads, and NFS requests. Any one of these metrics may make the system busy. In addition to the built-in metrics, an interface is defined for running a user-specified process that may indicate that the system is busy.

Hardware State

Devices that export a `reg` property are considered to have hardware state that must be saved prior to shutting down the system. If a device does not have a `reg` property, then it is considered to be stateless. However, this consideration can be overridden by the device driver.

A device that has hardware state but no `reg` property (such as a SCSI target driver, which has hardware at the other end of the SCSI bus), is called to save and restore its state if it exports a `pm-hardware-state` property with the value `needs-suspend-resume`. Otherwise, the lack of a `reg` property is taken to mean that the device has no hardware state. .

A device that has a `reg` property but no hardware state may export a `pm-hardware-state` property with the value `no-suspend-resume` to keep the framework from calling into the driver to save and restore that state. For more information on power management properties, see the `pm`(9) man page.

Policy

The system will be shut down if the following conditions apply:

- Autoshutdown is enabled by `dtpower(1M)` or `power.conf`(4).
- The system has been idle for *autoshutdown threshold* minutes.
- All the metrics specified in `power.conf`(4) have been satisfied.

Entry Points Used by System Power Management

System power management passes the command DDI_SUSPEND to the detach(9E) driver entry point to request the driver to save the device hardware state. It passes the command DDI_RESUME to the attach(9E) driver entry point to request the driver to restore the device hardware state.

detach(9E)

```
int detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
```

If a device has a reg property or a pm-hardware-state property with a value of needs-suspend-resume, then the framework calls into the driver's detach(9E) entry point to allow the driver to save the hardware state of the device to memory so that it can be restored after the system power returns. To process the DDI_SUSPEND command, detach(9E) must do the following:

- Block further operations from being initiated until the device is resumed (except for dump(9E) requests).
- Wait until outstanding operations have completed (or abort them if they can be restarted).
- Cancel pending timeouts and callbacks.
- Save any volatile hardware state to memory. The state includes the contents of device registers, but may also include downloaded firmware.

If, for some reason, the driver is not able to suspend the device and save its state to memory, then it must return DDI_FAILURE, and the framework aborts the system power management operation.

Dump requests must be honored. The framework uses the dump(9E) entry point to write out the state file containing the contents of memory. See dump(9E) for restrictions imposed on the device driver when using this entry point.

If the device implements power-manageable components, the device may have had its state saved and powered off when its detach(9E) entry point is called with the DDI_SUSPEND command. In this case the driver should cancel pending timeouts and suppress the call to pm_raise_power(9F) (except for dump(9E) requests) until the device is resumed by a call to attach(9E) with a command of DDI_RESUME. The driver must keep sufficient track of its state to be able to deal appropriately with this possibility. Code Example 9–6 shows an example of a detach(9E) routine with the DDI_SUSPEND command implemented.

CODE EXAMPLE 9–6 detach(9E) Routine Implementing DDI_SUSPEND

```
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:
                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;
                        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(9E)

```
    int attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
```

When power is restored to the system, each device with a `reg` property or with a `pm-hardware-state` property of value `needs-suspend-resume` has its `attach`(9E) entry point called with a command value of `DDI_RESUME`. If the system

shutdown was aborted for some reason, each driver that was suspended is called to resume, even though the power has not been shut off. Consequently, the resume code in `attach`(9E) must make no assumptions about the state of the hardware; it may or may not have lost power.

The power management framework will consider the power level of the components to be unknown at `DDI_RESUME` time. Depending on the nature of the device, the driver writer has two choices:

- If it is possible to tell the actual power level of the components of the device without powering them up (by reading a register, for example), then the driver should notify the framework of the power level of each component by calling `pm_power_has_changed`(9F).
- If it is not possible to determine the power levels of the components, then the driver should mark each of them internally as unknown and be sure to call `pm_raise_power`(9F) before the first access to each one.

Code Example 9–7 shows an example of an `attach`(9E) routine with the `DDI_RESUME` command.

CODE EXAMPLE 9–7 `attach(9E)` Routine Implementing `DDI_RESUME`

```
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) {
        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);
        }
}
```

Note - The `detach`(9E) and `attach`(9E) interfaces may also be used to resume a system that has been quiesced.

Device Access

If power management is supported, and `detach`(9E) and `attach`(9E) have code such as shown in the previous examples, the code fragment in Code Example 9–8 can be used where device access is about to be made to the device from user context (for example, in `read`(2), `write`(2), `ioctl`(2)).

In the following example, it is assumed that the operation about to be performed requires a component `component` that is operating at power level `level`.

CODE EXAMPLE 9–8 Device Access

```
...
mutex_enter(&xsp->mu);
/*
 * Block command while device is suspeded via DDI_SUSPEND
 */
while (xsp->xx_suspended)
        cv_wait(&xsp->xx_suspend_cv, &xsp->mu);

/*
 * Mark component busy so 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_SUCESS) {
                /*
                 * Log error and abort
                 */
                ...
        }
        mutex_enter(&xsp->mu);
}
...
```

The code fragment in Code Example 9–9 can be used when device operation completes (for example, in device's interrupt handler).

CODE EXAMPLE 9–9 Device Operation Completion

```
...
/*
 * 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
 */
...
```

Power Management Flow of Control

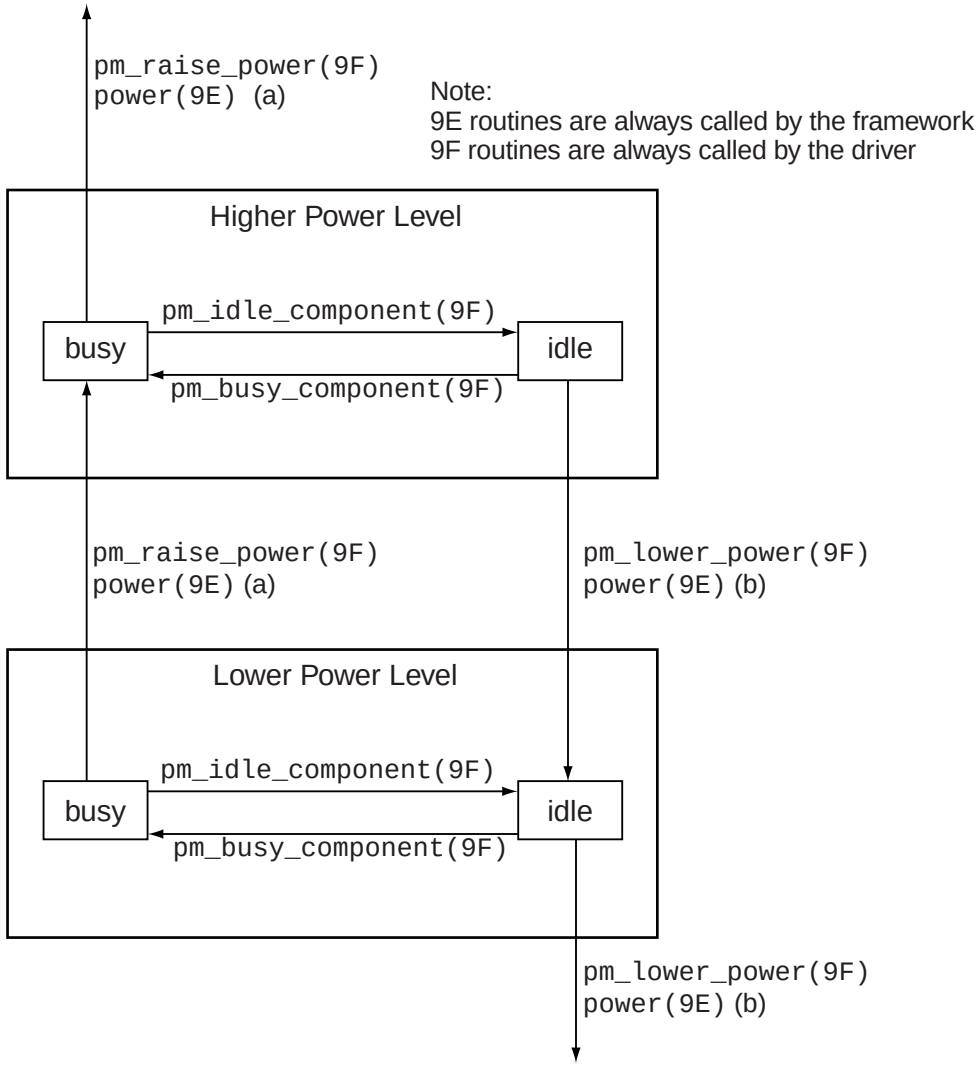
Figure 9–1 illustrates the flow of control in the power management framework.

When a component's activity is complete, a driver can call `pm_idle_component`(9F) to mark the component as idle. When the component has been idle for its threshold time, the framework may lower the power of the component to its next lower level. The framework does this by calling the `power`(9E) function to set the power level of the component to its next lower supported power level (if any). The driver's `power`(9E) function should reject any

attempt to lower the power level of a component when it is busy. The driver's power(9E) function should also save any state that will be lost as a result of the transition to the lower power level before making that transition.

When the component is needed again at a higher power level, the driver calls pm_busy_component(9F) to keep the framework from lowering the power still further, and then calls pm_raise_power(9F) on the component. The framework then calls power(9E) to raise the power of the component (before the call to pm_raise_power(9F) returns). The driver's power(9E) code must restore any state that was lost in the lower level but is needed in the higher level after making the power transition.

When a driver is detaching, it should call pm_lower_power(9F) for each component to lower its power to its lowest level. The framework may then call into the driver's power(9E) routine (before the call to pm_lower_power(9F) returns) to lower the power of the component.



(a) `power(9E)` may be called by the framework to raise the power level of a component as a result of a dependency or may be called by the framework as a result of the driver's calll to `pm_raise_power(9F)`

(b) `power(9E)` may be called by the framework to lower the power level of a component as a result of a device idleness, or may be called by the framework as a result of the driver's calll to `pm_lower_power(9F)` when the driver is detaching

Figure 9–1 Power Management Conceptual State Diagram

Changes to Power Management Interfaces

Previous to the Solaris 8 release, power management of devices was not automatic. It was necessary to add an entry to `/etc/power.conf` for each device that was to be power managed.

The framework assumed that all devices supported only two power levels: 0 and full (“normal”) power.

There was an implied dependency of all other components on component 0. Whenever component 0 changed to or from level 0, a call was made into the driver’s `detach(9E)` or `attach(9E)` routine with commands `DDI_PM_SUSPEND` and `DDI_PM_RESUME` respectively to save and restore hardware state.

The old interfaces (including `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` and `DDI_PM_RESUME`) are still supported for binary compatibility, but are obsolete.

As of the Solaris 8 release, devices which export the `pm-components` property are automatically power managed (if autopm is enabled).

The framework now knows from the `pm-components` property which power levels are supported by each device.

The framework makes no assumptions about dependencies among the different components of a device. The device driver is responsible for saving and restoring hardware state as needed when changing power levels.

These changes allow the power management framework to deal with emerging device technology, and result in greater power savings (since the framework can detect automatically which devices can save power, and can use intermediate power states of the devices), and allows the system to meet energy consumption goals without the entire system being powered down or functionality being lost.

TABLE 9–1 Power Management Interfaces

Old Interfaces	Solaris 8 Interfaces
pm_create_components(9E)	pm-conponents(9)
pm_set_normal_power(9F)	pm-components(9)
pm_destroy_components(9F)	

TABLE 9–1 Power Management Interfaces *(continued)*

Old Interfaces	Solaris 8 Interfaces
pm_get_normal_power(9E)	
ddi_dev_is_needed(9F)	pm_raise_power(9F)
	pm_lower_power(9F
	pm_power_has_changed(9F)
DDI_PM_SUSPEND	
DDI_PM_RESUME	

CHAPTER 10

Drivers for Character Devices

This chapter describes the structure of a character device driver, focusing in particular on character driver entry points. In addition, this chapter covers the use of `physio`(9F) (in `read`(9E) and `write`(9E)) and `aphysio`(9F) (in `aread`(9E) and `awrite`(9E)) in the context of synchronous and asynchronous I/O transfers.

Character Driver Structure Overview

Figure 10–1 shows data structures and routines that define the structure of a character device driver. Device drivers typically include the following:

- Device-loadable driver section
- Device configuration section
- Character driver entry points

The shaded device access section in Figure 10–1 illustrates character driver entry points.

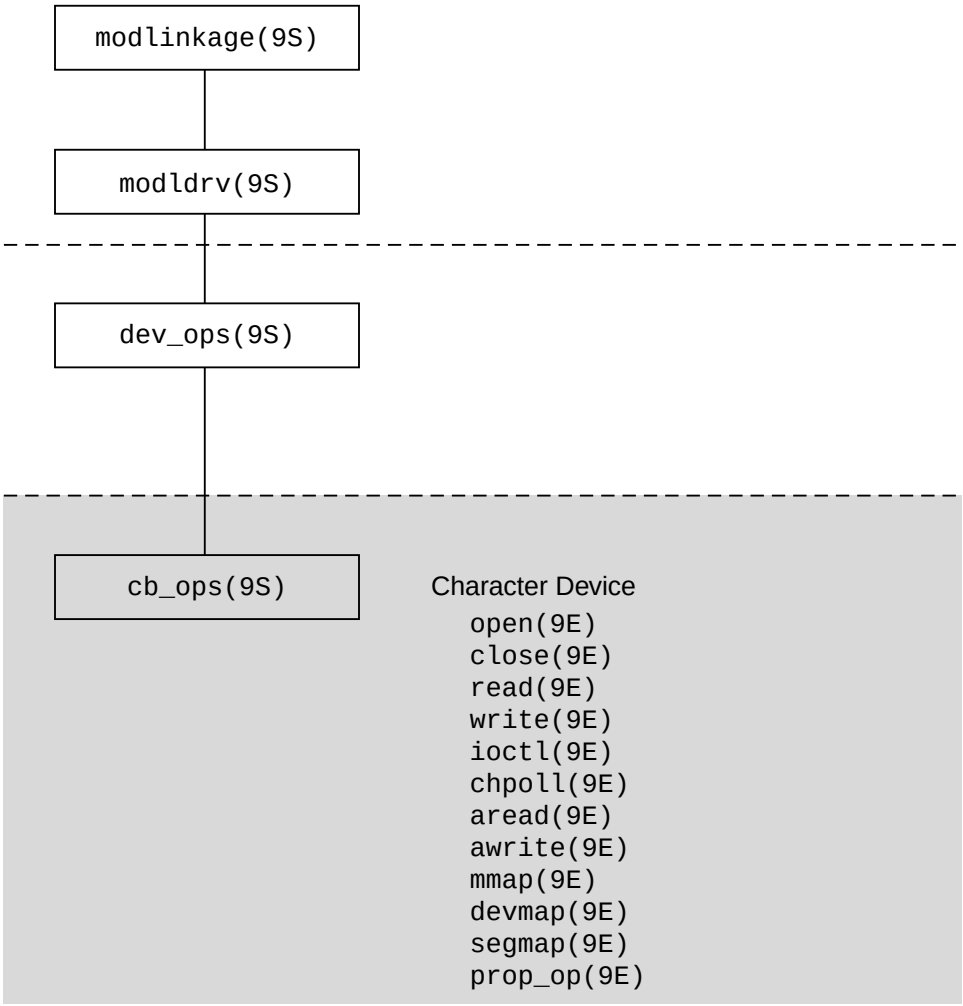


Figure 10–1 Character Driver Roadmap

Entry Points

Associated with each device driver is a `dev_ops`(9S) structure, which in turn refers to a `cb_ops`(9S) structure. These structures contain pointers to the driver entry points. Note that some of these entry points can be replaced with `nodev`(9F) or `nulldev`(9F) as appropriate.

Autoconfiguration

The `attach`(9E) routine should perform the common initialization tasks that all devices require. Typically, these tasks include:

- Allocating per-instance state structures
- Registering device interrupts
- Mapping the device's registers
- Initializing mutex and condition variables
- Creating power-manageable components
- Creating minor nodes

See "`attach(9E)`" on page 71 for code examples of these tasks.

Character device drivers create minor nodes of type `S_IFCHR`. This causes a character special file representing the node to eventually appear in the `/devices` hierarchy. Code Example 10–1 shows a character driver `attach`(9E) routine.

CODE EXAMPLE 10–1 Character Driver `attach(9E)` Routine

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
  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,
                minor_number, 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);
            }
            ...
            return (DDI_SUCCESS);
case DDI_RESUME:
        For information, see Chapter 9
default:
        return (DDI_FAILURE);
      }
}
```

Device Access

Access to a device by one or more application programs is controlled through the open(9E) and close(9E) entry points. The open(9E) routine of a character driver is always called whenever an open(2) system call is issued on a special file representing the device. For a particular minor device, open(9E) can be called many times, but the close(9E) routine is called only when the final reference to a device is removed. If the device is accessed through file descriptors, this is by a call to close(2) or exit(2). If the device is accessed through memory mapping, this could also be by a call to munmap(2).

open(9E)

```
int xxopen(dev_t *devp, int flag, int otyp, cred_t *credp);
```

The primary function of open(9E) is to verify that the open request is allowed. devp is a pointer to a device number. The open(9E) routine is passed a pointer so that the driver can change the minor number. This allows drivers to dynamically create minor instances of the device. An example of this might be a pseudo-terminal driver that creates a new pseudo-terminal whenever the driver is opened. A driver that dynamically chooses the minor number, normally creates only one minor device node in attach(9E) with ddi_create_minor_node(9F), then changes the minor number component of *devp using makedevice(9F) and getmajor(9F):

```
*devp = makedevice(getmajor(*devp), new_minor);
```

The driver must keep track of available minor numbers internally.

otyp indicates how open(9E) was called. The driver must check that the value of otyp is appropriate for the device. For character drivers, otyp should be OTYP_CHR (see the open(9E) manual page).

flag contains bits indicating whether the device is being opened for reading (FREAD), writing (FWRITE), or both. User threads issuing the open(2) system call can also request exclusive access to the device (FEXCL) or specify that the open should not block for any reason (FNDELAY), but it is the driver's responsibility to enforce both cases. A driver for a write-only device such as a printer might consider an open for reading invalid.

credp is a pointer to a credential structure containing information about the caller, such as the user ID and group IDs. Drivers should not examine the structure directly, but should instead use drv_priv(9F) to check for the common case of root privileges. In this example, only *root* is allowed to open the device for writing.

Code Example 10–2 shows a character driver open(9E) routine.

CODE EXAMPLE 10–2 Character Driver open(9E) Routine

```
static int
xxopen(dev_t *devp, int flag, int otyp, cred_t *credp)
{
    minor_t            instance;

        if (getminor(*devp) 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(9E)

```
int xxclose(dev_t dev, int flag, int otyp, cred_t *credp);
```

close(9E) should perform any cleanup necessary to finish using the minor device, and prepare the device (and driver) to be opened again. For example, the open routine might have been invoked with the exclusive access (FEXCL) flag. A call to close(9E) would allow further opens to continue. Other functions that close(9E) might perform are:

- Waiting for I/O to drain from output buffers before returning
- Rewinding a tape (tape device)
- Hanging up the phone line (modem device)

I/O Request Handling

This section gives the details of I/O request processing: from the application to the kernel, the driver, the device, the interrupt handler, and back to the user.

User Addresses

When a user thread issues a write(2) system call, it passes the address of a buffer in user space:

```
char buffer[] = "python";
count = write(fd, buffer, strlen(buffer) + 1);
```

The system builds a uio(9S) structure to describe this transfer by allocating an iovec(9S) structure and setting the iov_base field to the address passed to write(2), in this case, buffer. The uio(9S) structure is what is passed to the driver write(9E) routine (see "Vectored I/O" on page 146 for more information about the uio(9S) structure).

The problem is that this address is in user space, not kernel space, and so is not guaranteed to be currently in memory. It is not even guaranteed to be a valid address. In either case, accessing a user address directly from the device driver or from the kernel could crash the system, so device drivers should never access user addresses directly. Instead, they should always use one of the data transfer routines in the Solaris 8 DDI/DKI that transfer data into or out of the kernel. These routines are able to handle page faults, either by bringing the proper user page in and continuing the copy transparently, or by returning an error on an invalid access.

Two routines commonly used are copyout(9F) to copy data from kernel space to user space and copyin(9F) to copy data from user space to kernel space. ddi_copyout(9F) and ddi_copyin(9F) operate similarly but are to be used in the ioctl(9E) routine. copyin(9F) and copyout(9F) can be used on the buffer described by each iovec(9S) structure, or uiomove(9F) can perform the entire transfer to or from a contiguous area of driver (or device) memory.

Vectored I/O

In character drivers, transfers are described by a uio(9S) structure. The uio(9S) structure contains information about the direction and size of the transfer, plus an array of buffers for one end of the transfer (the other end is the device). The following section lists uio(9S) structure members that are important to character drivers.

uio Structure

The uio(9S) structure contains the following members:

```
iovec_t        *uio_iov;              /* base address of the iovec */
                                      /* buffer description array */
int            uio_iovcnt;            /* the number of iovec structures */
off_t          uio_offset;            /* offset into device where data */
                                      /* is transferred from or to */
offset_t       uio_loffset            /* 64-bit offset into file where */
                                      /* data is transferred from or to */
int            uio_resid;             /* amount (in bytes) not */
                                      /* transferred on completion */
```

A uio(9S) structure is passed to the driver read(9E) and write(9E) entry points. This structure is generalized to support what is called *gather-write* and *scatter-read*. When writing to a device, the data buffers to be written do not have to be contiguous in application memory. Similarly, when reading from a device into memory, the data comes off the device in a contiguous stream but can go into

noncontiguous areas of application memory. See readv(2), writev(2), pread(2), and pwrite(2) for more information on scatter-gather I/O.

Each buffer is described by an iovec(9S) structure. This structure contains a pointer to the data area and the number of bytes to be transferred.

```
caddr_t        iov_base;        /* address of buffer */
int            iov_len;         /* amount to transfer */
```

The uio structure contains a pointer to an array of iovec(9S) structures. The base address of this array is held in uio_iov, and the number of elements is stored in uio_iovcnt.

The uio_offset field contains the 32-bit offset into the device at which the application needs to begin the transfer. uio_loffset is used for 64-bit file offsets. If the device does not support the notion of an offset, these fields can be safely ignored. The driver should interpret either uio_offset or uio_loffset (but not both). If the driver has set the D_64BIT flag in the cb_ops(9S) structure, it should use uio_loffset.

The uio_resid field starts out as the number of bytes to be transferred (the sum of all the iov_len fields in uio_iov) and *must* be set by the driver to the number of bytes *not* transferred before returning. The read(2) and write(2) system calls use the return value from the read(9E) and write(9E) entry points to determine if the transfer failed (and then return -1). If the return value indicates success, the system calls return the number of bytes requested minus uio_resid. If uio_resid is not changed by the driver, the read(2) and write(2) calls will return 0 (indicating end-of-file), even though all the data was transferred.

The support routines uiomove(9F), physio(9F), and aphysio(9F) update the uio(9S) structure directly, updating the device offset to account for the data transfer. When used with a seekable device, for which the concept of position is relevant, the driver does not need to adjust either the uio_offset or uio_loffset fields. I/O performed to a device in this manner is constrained by the maximum possible value of uio_offset or uio_loffset. An example of such a usage is raw I/O on a disk.

When performing I/O on a device on which the concept of position has no relevance, the driver can save uio_offset or uio_loffset, perform the I/O operation, then restore uio_offset or uio_loffset to the field's initial value. I/O performed to a device in this manner is not constrained by the maximum possible value of uio_offset or uio_loffset. An example of such a usage is I/O on a serial line.

The following example shows one way to preserve uio_loffset in the read(9E) function.

```
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 */
}
```

Synchronous Versus Asynchronous I/O

Data transfers can be *synchronous* or *asynchronous* depending on whether the entry point scheduling the transfer returns immediately or waits until the I/O has been completed.

The `read`(9E) and `write`(9E) entry points are synchronous entry points; they must not return until the I/O is complete. Upon return from the routines, the process knows whether the transfer has succeeded.

The `aread`(9E) and `awrite`(9E) entry points are asynchronous entry points. They schedule the I/O and return immediately. Upon return, the process issuing the request knows that the I/O has been scheduled and that the status of the I/O must be determined later. In the meantime, the process can perform other operations.

When an asynchronous I/O request is made to the kernel by a user process, the process is not required to wait while the I/O is in process. A process can perform multiple I/O requests and let the kernel handle the data transfer details. This is useful in applications such as transaction processing where concurrent programming methods can take advantage of asynchronous kernel I/O operations to increase performance or response time. Any performance boost for applications using asynchronous I/O, however, comes at the expense of greater programming complexity.

Data Transfer Methods

Data can be transferred using either programmed I/O or DMA. These data transfer methods can be used by either synchronous or asynchronous entry points, depending on the capabilities of the device.

Programmed I/O Transfers

Programmed I/O devices rely on the CPU to perform the data transfer. Programmed I/O data transfers are identical to other device register read and write operations. Various data access routines are used to read or store values to device memory.

`uiomove(9F)`

`uiomove`(9F) can be used to transfer data to some programmed I/O devices. `uiomove`(9F) transfers data between the user space (defined by the `uio(9S)` structure) and the kernel. `uiomove`(9F) can handle page faults, so the memory to which data is transferred need not be locked down. It also updates the `uio_resid`

field in the uio(9S) structure. Code Example 10–3 shows one way to write a ramdisk read(9E) routine. It uses synchronous I/O and relies on the presence of the following fields in the ramdisk state structure:

```
caddr_t        ram;            /* base address of ramdisk */
int            ramsize;        /* size of the ramdisk */
```

CODE EXAMPLE 10–3 Ramdisk read(9E) Routine Using uiomove(9F)

```
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));
}
```

uwritec(9F) *and* uread(9F)

Another example of programmed I/O might be a driver writing data one byte at a time directly to the device's memory. Each byte is retrieved from the uio(9S) structure using uwritec(9F), then sent to the device. read(9E) can use ureadc(9F) to transfer a byte from the device to the area described by the uio(9S) structure.

CODE EXAMPLE 10–4 Programmed I/O write(9E) Routine Using uwritec(9F)

```
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:
        pm_busy_component(xsp->dip, 0);
        if (xsp->pm_suspended)
                ddi_dev_is_needed(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 Transfers (Synchronous)

Most character drivers use `physio`(9F) to do most of the setup work for DMA transfers in `read`(9E) and `write`(9E). This is shown in Code Example 10–5.

```
int physio(int (*strat)(struct buf *), struct buf *bp,
     dev_t dev, int rw, void (*mincnt)(struct buf *),
     struct uio *uio);
```

`physio`(9F) requires the driver to provide the address of a `strategy`(9E) routine. `physio`(9F) ensures that memory space is locked down (cannot be paged out) for the duration of the data transfer. This is necessary for DMA transfers because they cannot handle page faults. `physio`(9F) also provides an automated way of breaking a larger transfer into a series of smaller, more manageable ones. See "`minphys(9F)`" on page 152 for more information.

CODE EXAMPLE 10–5 `read(9E)` and `write(9E)` Routines Using `physio(9F)`

```
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);
     ret = physio(xxstrategy, NULL, dev, B_READ, xxminphys, uiop);
     pm_idle_component(xsp->dip, 0);
     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);
     pm_idle_component(xsp->dip, 0);
     return (ret);
}
```

In the call to `physio`(9F), `xxstrategy`() is a pointer to the driver strategy routine. Passing `NULL` as the `buf`(9S) structure pointer tells `physio`(9F) to allocate a `buf`(9S) structure. If it is necessary for the driver to provide `physio`(9F) with a `buf`(9S) structure, `getrbuf`(9F) should be used to allocate one. `physio`(9F) returns zero if the transfer completes successfully, or an error number on failure. After calling `strategy`(9E), `physio`(9F) calls `biowait`(9F) to block until the transfer is completed or fails. The return value of `physio`(9F) is determined by the error field in the `buf`(9S) structure set by `bioerror`(9F).

DMA Transfers (Asynchronous)

Character drivers supporting `aread`(9E) and `awrite`(9E) use `aphysio`(9F) instead of `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);
```

Note - The address of `anocancel`(9F) is the only value that can currently be passed as the second argument to `aphysio`(9F).

`aphysio`(9F) requires the driver to pass the address of a `strategy`(9E) routine. `aphysio`(9F) ensures that memory space is locked down (cannot be paged out) for the duration of the data transfer. This is necessary for DMA transfers because they cannot handle page faults. `aphysio`(9F) also provides an automated way of breaking a larger transfer into a series of smaller, more manageable ones. See “`minphys(9F)`” on page 152 for more information. Code Example 10–5 and Code Example 10–6 demonstrate that the `aread`(9E) and `awrite`(9E) entry points differ only slightly from the `read`(9E) and `write`(9E) entry points; the difference lies mainly in their use of `aphysio`(9F) instead of `physio`(9F).

CODE EXAMPLE 10–6 `aread(9E)` and `awrite(9E)` Routines Using `aphysio(9F)`

```
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));
}
```

In the call to aphysio(9F), xxstrategy() is a pointer to the driver strategy routine. aiop is a pointer to the aio_req(9S) structure and is also passed to aread(9E) and awrite(9E). aio_req(9S) describes where the data is to be stored in user space. aphysio(9F) returns zero if the I/O request is scheduled successfully or an error number on failure. After calling strategy(9E), aphysio(9F) returns without waiting for the I/O to complete or fail.

minphys(9F)

xxminphys() is a pointer to a function to be called by physio(9F) or aphysio(9F) to ensure that the size of the requested transfer does not exceed a driver-imposed limit. If the user requests a larger transfer, strategy(9E) will be called repeatedly, requesting no more than the imposed limit at a time. This is important because DMA resources are limited. Drivers for slow devices, such as printers, should be careful not to tie up resources for a long time.

Usually, a driver passes the address of the kernel function minphys(9F), but the driver can define its own xxminphys() routine instead. The job of xxminphys() is to keep the b_bcount field of the buf(9S) structure below a driver limit. There might be additional system limits that the driver should not circumvent, so the driver xxminphys() routine should call the system minphys(9F) routine after setting the b_bcount field and before returning.

CODE EXAMPLE 10–7 minphys(9F) Routine

```
#define XXMINVAL (512 << 10)    /* 512 KB */
static void
xxminphys(struct buf *bp)
{
       if (bp->b_bcount > XXMINVAL)
            bp->b_bcount = XXMINVAL
      minphys(bp);
}
```

strategy(9E)

The strategy(9E) routine originated in block drivers and is so called because it can implement a strategy for efficient queuing of I/O requests to a block device. A driver for a character-oriented device can also use a strategy(9E) routine. In the character I/O model presented here, strategy(9E) does not maintain a queue of requests, but rather services one request at a time.

In Code Example 10–8, the strategy(9E) routine for a character-oriented DMA device allocates DMA resources for synchronous data transfer and starts the command by programming the device register (see Chapter 8 for a detailed description).

Note - strategy(9E) does not receive a device number (dev_t) as a parameter; instead, this is retrieved from the b_edev field of the buf(9S) structure passed to strategy(9E).

CODE EXAMPLE 10–8 strategy(9E) Routine

```
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 ddi_dev_is_needed(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);
}
```

Note - Although strategy(9E) is declared to return an int, it must always return zero.

On completion of the DMA transfer, the device generates an interrupt, causing the interrupt routine to be called. In Code Example 10–9, xxintr( ) receives a pointer to the state structure for the device that might have generated the interrupt.

CODE EXAMPLE 10–9 Interrupt Routine

```
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);
}
```

The driver indicates an error by calling `bioerror`(9F). The driver must call `biodone`(9F) when the transfer is complete or after indicating an error with `bioerror`(9F).

Mapping Device Memory

Some devices, such as frame buffers, have memory that is directly accessible to user threads by way of memory mapping. Drivers for these devices typically do not support the `read`(9E) and `write`(9E) interfaces. Instead, these drivers support memory mapping with the `devmap`(9E) entry point. A typical example is a frame buffer driver that implements the `devmap`(9E) entry point to allow the frame buffer to be mapped in a user thread.

segmap(9E)

```
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);
```

`segmap`(9E) is the entry point responsible for actually setting up a memory mapping requested by the system on behalf of an `mmap`(2) system call. Drivers for many memory-mapped devices will use `ddi_devmap_segmap`(9F) as the entry point rather than define their own `segmap`(9E) routine.

If a driver wants to check mapping permissions or allocate private mapping resources before setting up the mapping, the driver can provide its own `segmap`(9E) entry point. `segmap`(9E) must call `devmap_setup`(9F) before returning.

In Code Example 10–10, the driver controls a frame buffer that allows write-only mappings. The driver returns EINVAL if the application tries to gain read access and then calls `devmap_setup`(9F) to set up the user mapping.

CODE EXAMPLE 10–10 segmap(9E) Routine

```
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 (devmap_setup(dev, (offset_t)off, as, addrp,
            (size_t)len, prot, maxprot, flags, cred));
}
```

devmap(9E)

```
int xxdevmap(dev_t dev, devmap_cookie_t handle, offset_t off,
     size_t len, size_t *maplen, uint_t model);
```

This entry point is called to export device memory or kernel memory to user applications. devmap(9E) is called from devmap_setup(9F) inside segmap(9E) or on behalf of ddi_devmap_segmap(9F). See Chapter 12 and Chapter 13 for details.

Multiplexing I/O on File Descriptors

A thread sometimes needs to handle I/O on more than one file descriptor. One example is an application program that needs to read the temperature from a temperature-sensing device and then report the temperature to an interactive display. If the program makes a read request and there is no data available, it should not block waiting for the temperature before interacting with the user again.

The poll(2) system call provides users with a mechanism for multiplexing I/O over a set of file descriptors that reference open files. poll(2) identifies those file descriptors on which a program can send or receive data without blocking, or on which certain events have occurred.

To allow a program to poll a character driver, the driver must implement the chpoll(9E) entry point.

chpoll(9E)

```
int xxchpoll(dev_t dev, short events, int anyyet, short *reventsp,
     struct pollhead **phpp);
```

The system calls chpoll(9E) when a user process issues a poll(2) system call on a file descriptor associated with the device. The chpoll(9E) entry point routine is used by non-STREAMS character device drivers that need to support polling.

In chpoll(9E), the driver must follow these rules:

- Implement the following algorithm when the chpoll(9E) entry point is called:

```
if (events are satisfied now) {
        *reventsp = mask of satisfied events;
} else {
        *reventsp = 0;
        if (!anyyet)
                *phpp = &local pollhead structure;
}
return (0);
```

xxchpoll() should check to see if certain events have occurred; see chpoll(9E). It should then return the mask of satisfied events by setting the return events in *reventsp.

If no events have occurred, the return field for the events is cleared. If the anyyet field is not set, the driver must return an instance of the pollhead structure. It is usually allocated in a state structure and should be treated as opaque by the driver. None of its fields should be referenced.

- Call pollwakeup(9F) whenever a device condition of type events, listed in Code Example 10–11, occurs. This function should be called only with one event at a time. pollwakeup(9F) might be called in the interrupt routine when the condition has occurred.

Code Example 10–11 and Code Example 10–12 show how to implement the polling discipline and how to use pollwakeup(9F).

CODE EXAMPLE 10–11 chpoll(9E) Routine

```
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 ((events & POLLERR) && (status & DEVICE_ERROR)) {
            revent |= POLLERR;
     }
     /* if nothing has occurred */
```

```
    if (revent == 0) {
           if (!anyyet) {
               *phpp = &xsp->pollhead;
           }
    }
      *reventsp = revent;
    return (0);
}
```

In Code Example 10–12, the driver can handle the `POLLIN` and `POLLERR` events. The driver first reads the status register to determine the current state of the device. The parameter `events` specifies which conditions the driver should check. If the appropriate conditions have occurred, the driver sets that bit in `*reventsp`. If none of the conditions have occurred and `anyyet` is not set, the address of the `pollhead` structure is returned in `*phpp`.

CODE EXAMPLE 10–12 Interrupt Routine Supporting `chpoll(9E)`

```
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);
    }
    if (just completed a read) {
           pollwakeup(&xsp->pollhead, POLLIN);
    }
    ...
    return (DDI_INTR_CLAIMED);
}
```

`pollwakeup`(9F) is usually called in the interrupt routine when a supported condition has occurred. The interrupt routine reads the status from the status register and checks for the conditions. It then calls `pollwakeup`(9F) for each event to possibly notify polling threads that they should check again. Note that `pollwakeup`(9F) should not be called with any locks held, as it could cause the `chpoll`(9E) routine to be entered, resulting in deadlock if that routine tries to grab the same lock.

Miscellaneous I/O Control

The `ioctl`(9E) routine is called when a user thread issues an `ioctl`(2) system call on a file descriptor associated with the device. The I/O control mechanism is a catchall for getting and setting device-specific parameters. It is frequently used to set

a device-specific mode, either by setting internal driver software flags or by writing commands to the device. It can also be used to return information to the user about the current device state. In short, it can do whatever the application and driver need it to do.

ioctl(9E)

```
int xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
     cred_t *credp, int *rvalp);
```

The `cmd` parameter indicates which command `ioctl`(9E) should perform. By convention, I/O control commands indicate the driver they belong to in bits 8-15 of the command (usually given by the ASCII code of a character representing the driver), and the driver-specific command in bits 0-7. They are usually created in the following way:

```
#define XXIOC        ('x' << 8)        /* 'x' is a character representing */
                                       /* device xx */


#define XX_GET_STATUS                  (XXIOC | 1) /* get status register */
#define XX_SET_CMD                     (XXIOC | 2) /* send command */
```

The interpretation of `arg` depends on the command. I/O control commands should be documented (in the driver documentation or a manual page) and defined in a public header file, so that applications can determine the names, what they do, and what they accept or return as `arg`. Any data transfer of `arg` (into or out of the driver) must be performed by the driver.

Certain classes of devices such as frame buffers or disks must support standard sets of I/O control requests. These standard I/O control interfaces are documented in the *Solaris 8 Reference Manual Collection*. For example, `fbio`(7I) documents the I/O controls that frame buffers must support, and `dkio`(7I) documents standard disk I/O controls. See "Miscellaneous I/O Control" on page 157 for more information on I/O control.

Drivers must use `ddi_copyin`(9F) to transfer "arg" data from the userland application to the kernel and `ddi_copyout`(9F) from kernel to userland. Failure to use `ddi_copyin`(9F) or `ddi_copyout`(9F) will result in panics on architectures that separate kernel and user address spaces, or if the user address has been swapped out.

`ioctl`(9E) is usually a switch statement with a case for each supported `ioctl`(9E) request.

CODE EXAMPLE 10–13 `ioctl(9E)` Routine

```
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);
}
```

The `cmd` variable identifies a specific device control operation. If `arg` contains a user virtual address, `ioctl`(9E) must call `ddi_copyin`(9F) or `ddi_copyout`(9F) to transfer data between the data structure in the application program pointed to by `arg` and the driver. In Code Example 10–13, for the case of an `XX_GET_STATUS` request the contents of `xsp->regp->csr` are copied to the address in `arg`. When a request succeeds, `ioctl`(9E) can store in `*rvalp` any integer value to be the return value of the `ioctl`(2) system call that made the request. Negative return values, such as -1, should be avoided, as they usually indicate the system call failed, and many application programs assume that negative values indicate failure.

An application that uses the I/O controls discussed above could look like Code Example 10–14.

CODE EXAMPLE 10–14 Using `ioctl(9E)`

```
#include <sys/types.h>
#include "xxio.h"               /* contains device's ioctl cmds and arguments */
int
main(void)
{
        uint8_t        status;
        ...

        /*
         * read the device status
         */
        if (ioctl(fd, XX_GET_STATUS, &status) == -1) {
```

```
                error handling
        }
        printf("device status %x\n", status);
        exit(0);
}
```

I/O Control Support for 64-Bit Capable Device Drivers

The Solaris kernel runs in 64-bit mode on suitable hardware and supports both 32-bit and 64-bit applications. A 64-bit device driver is required to support I/O control commands from 32-bit and 64-bit user mode programs. The difference between a 32-bit program and a 64-bit program is its C language type model: a 32-bit program is ILP32 and a 64-bit program is LP64. See Appendix C for information on C data type models.

Any data that flows between programs and the kernel and vice versa (for example using `ddi_copyin`(9F) or `ddi_copyout`(9F)) will either need to be identical in format regardless of the type model of the kernel and application, or the device driver should be able to handle a model mismatch between it and the application and adjust the data format accordingly.

To determine if there is a model mismatch, the `ioctl`(9E) mode parameter passes the data model bits to the driver. As Code Example 10–15 shows, the mode parameter is then passed to `ddi_model_convert_from`(9F) to determine if any model conversion is necessary.

The data model is passed to the `ioctl`(9E) routine using the mode field or flags argument. The flag will be set to one of:

- `FILP32`
- `FLP64`

with `FNATIVE` conditionally defined to match the data model of the kernel implementation. The flag should be extracted using the `FMODELS` mask. The driver can then determine the data model explicitly to work out how to copy the application data structure.

The DDI function `ddi_model_convert_from`(9F) is a convenience routine that can assist some drivers with their `ioctl( )` calls. The function takes the data type model of the user application as an argument and returns one of the following values:

- `DDI_MODEL_ILP32` — Convert from ILP32 application
- `DDI_MODEL_NONE` — No conversion needed

`DDI_MODEL_NONE` is returned if no data conversion is necessary. This is the case when application and driver have the same data model. `DDI_MODEL_ILP32` is returned if the driver is compiled to the LP64 data model and is communicating with a 32-bit application.

In the following example, the driver copies a data structure that contains a user address. Because the data structure changes size from ILP32 to LP64, the 64-bit driver uses a 32-bit version of the structure when communicating with a 32-bit application.

CODE EXAMPLE 10–15 `ioctl(9E)` Routine to Support 32-bit and 64-bit Applications

```
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 & FMODELS)) {
                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 */
                return (ENOTTY);
        }
```

```
        return (0);
}
```

Handling copyout(9F) Overflow

So far, this discussion has considered only data coming into the driver. What happens when data must be copied out of the driver?

However, sometimes a driver needs to copy out a native quantity that no longer fits in the 32-bit sized structure. In this case, the driver should return EOVERFLOW to the caller as an indication that the data type in the interface is too small to hold the value to be returned, as shown in Code Example 10–16.

CODE EXAMPLE 10–16

```
int
    xxioctl(dev_t dev, int cmd, intptr_t arg, int mode,
             cred_t *cr, int *rval_p)
    {
                struct resdata res;

                ... body of driver code ...

                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 (copyout(&res32,
                                        (void *)arg, sizeof (res32)))
                                        return (EFAULT);
                }
                break;

                case DDI_MODEL_NONE:
                                if (copyout(&res, (void *)arg, sizeof (res)))
                                        return (EFAULT);
                                break;
                }
                return (0);
    }
```

32–bit and 64–bit Data Structure Macros

While the method shown in the previous example works well for many drivers, an alternate scheme is to use the data structure macros provided in <sys/model.h> to

move data between the application and the kernel. These make the code less cluttered and behave identically, from a functional perspective.

CODE EXAMPLE 10–17

```
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);
}
```

How Do the Structure Macros Work?

In a 64-bit device driver, these macros do all that is necessary to use the same piece of kernel memory as a buffer for the contents of the native form of the data structure (that is, the LP64 form), and for the ILP32 form of the same structure. This usually means that each structure access is implemented by a conditional expression. When compiled as a 32-bit driver, only one data model is supported, only native form exists, so no conditional expression is used.

The 64-bit versions of the macros depend on the definition of a shadow version of the data structure that describes the 32-bit interface using fixed-width types. The name of the shadow data structure is formed by appending “32” to the name of the native data structure. For convenience, place the definition of the shadow structure in the same file as the native structure to ease future maintenance costs.

The macros take arguments such as:

structname — The structure name (as would appear after the `struct` keyword) of the native form of the data structure

umodel — A flag word containing the user data model, such as `FILP32` or `FLP64`, extracted from the mode parameter of `ioctl`(9E)

handle	The name used to refer to a particular instance of a structure that is manipulated by these macros
fieldname	The name of the field within the structure

When to Use Structure Macros

Macros allow you to make in-place references only to the fields of a data item. They do not provide a way to take separate code paths based on the data model. They should be avoided if the number of fields in the data structure is large or the frequency of references to these fields is high.

Because the macros hide many of the differences between data models in the implementation of the macros, code written with this interface is generally easier to read. When compiled as a 32-bit driver, the resulting code is compact without needing clumsy `#ifdefs`, while preserving type checking.

Macros are best suited for making in-place references to the fields of a data structure, particularly if the number of fields in the data structure is small and the frequency of references to these fields is low.

Declaring and Initializing Structure Handles

`STRUCT_DECL`(9F) and `STRUCT_INIT`(9F) can be used to declare and initialize a handle and space for decoding an ioctl on the stack. `STRUCT_HANDLE`(9F) and `STRUCT_SET_HANDLE`(9F) declare and initialize a handle without allocating space on the stack. The latter macros can be useful if the structure is very large, or is contained in some other data structure.

Note - Because the `STRUCT_DECL( )` and `STRUCT_HANDLE( )` macros expand to data structure declarations, they should be grouped with such declarations in C code.

```
STRUCT_DECL(structname, handle)
```

Declares a *structure handle* called `handle` for a struct `structname` data structure, and allocates space for its native form on the stack. The native form is assumed to be larger than or equal to the ILP32 form of the structure.

```
STRUCT_INIT(handle, umodel)
```

Initializes the data model for `handle` to `umodel`. This macro must be invoked before any access is made to a structure handle declared with `STRUCT_DECL( )`.

```
STRUCT_HANDLE(structname, handle)
```

Declares a *structure handle* called `handle`. Contrast with `STRUCT_DECL( )`.

```
STRUCT_SET_HANDLE(handle, umodel, addr)
```

Initializes the data model for `handle` to `umodel`, and sets `addr` as the buffer used for subsequent manipulation. Invoke this macro before accessing a structure handle declared with `STRUCT_HANDLE( )`.

Operations on Structure Handles

```
size_t STRUCT_SIZE(handle)
```

Returns the size of the structure referred to by `handle`, according to its embedded data model.

typeof fieldname `STRUCT_FGET(handle, fieldname)`

Returns the indicated field (non-pointer type) in the data structure referred to by `handle`.

typeof fieldname `STRUCT_FGETP(handle, fieldname)`

Returns the indicated field (pointer type) in the data structure referred to by `handle`.

```
STRUCT_FSET(handle, fieldname, val)
```

Sets the indicated field (non-pointer type) in the data structure referred to by `handle` to value `val`. The type of `val` should match the type of `fieldname`.

```
STRUCT_FSETP(handle, fieldname, val)
```

Sets the indicated field (pointer type) in the data structure referred to by `handle` to value `val`.

typeof fieldname `*STRUCT_FADDR(handle, fieldname)`

Returns the address of the indicated field in the data structure referred to by `handle`.

```
struct structname *STRUCT_BUF(handle)
```

Returns a pointer to the native structure described by `handle`.

Other Operations

`size_t SIZEOF_STRUCT(struct_name, datamodel)`

Returns the size of `struct_name` based on the given data model.

`size_t SIZEOF_PTR(datamodel)`

Returns the size of a pointer based on the given data model.

CHAPTER 11

Drivers for Block Devices

This chapter describes the structure of block device drivers. The kernel views a block device as a set of randomly accessible logical blocks. The file system buffers the data blocks between a block device and the user space using a list of `buf`(9S) structures. Only block devices can support a file system.

Block Driver Structure Overview

Figure 11–1 shows data structures and routines that define the structure of a block device driver. Device drivers typically include the following:

- Device-loadable driver section
- Device configuration section
- Device access section

Block Driver Device Access

The shaded device access section in Figure 11–1 illustrates block driver entry points.

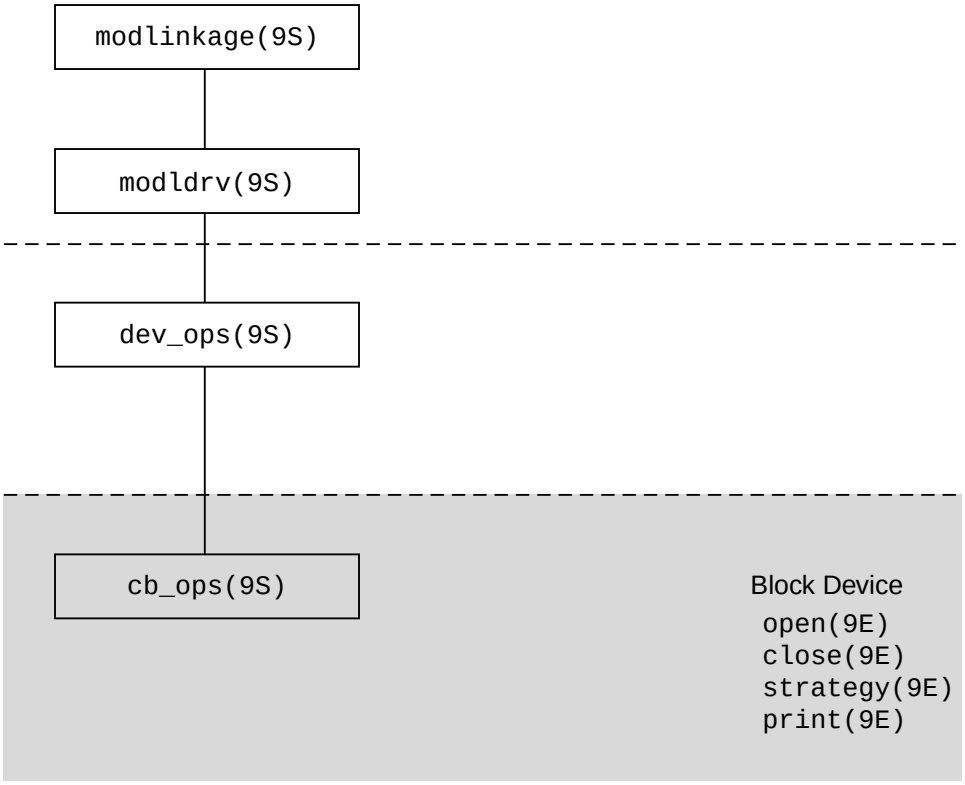


Figure 11–1 Block Driver Roadmap

File I/O

A file system is a tree-structured hierarchy of directories and files. Some file systems, such as the UNIX File System (UFS), reside on block-oriented devices. File systems are created by `format`(1M) and `newfs`(1M).

When an application issues a `read`(2) or `write`(2) system call to an ordinary file on the UFS file system, the file system can call the device driver `strategy`(9E) entry point for the block device on which the file resides. The file system code can call `strategy`(9E) several times for a single `read`(2) or `write`(2) system call.

It is the file system code that determines the logical device address, or *logical block number*, for each block and builds a block I/O request in the form of a `buf`(9S) structure. The driver `strategy`(9E) entry point then interprets the `buf`(9S) structure and completes the request.

Entry Points

Associated with each device driver is a dev_ops(9S) structure, which in turn refers to a cb_ops(9S) structure. See Chapter 5, for details regarding driver data structures.

Note - Some of the entry points can be replaced by nodev(9F) or nulldev(9F) as appropriate.

Autoconfiguration

attach(9E) should perform the common initialization tasks for each instance of a device. Typically, these tasks include:

- Allocating per-instance state structures
- Mapping the device's registers
- Registering device interrupts
- Initializing mutex and condition variables
- Creating power mangeable components
- Creating minor nodes

Block device drivers create minor nodes of type S_IFBLK. This causes a block special file representing the node to eventually appear in the /devices hierarchy.

Logical device names for block devices appear in the /dev/dsk directory, and consist of a controller number, bus-address number, disk number, and slice number. These names are created by the devfsadm(1M) program if the node type is set to DDI_NT_BLOCK or DDI_NT_BLOCK_CHAN. DDI_NT_BLOCK_CHAN should be specified if the device communicates on a channel (a bus with an additional level of addressability), such as SCSI disks, and causes a bus-address field (tN) to appear in the logical name. DDI_NT_BLOCK should be used for most other devices.

For each minor device (which corresponds to each partition on the disk), the driver must also create an nblocks property. This is an integer property giving the number of blocks supported by the minor device expressed in units of DEV_BSIZE (512 bytes). The file system uses the nblocks property to determine device limits. See "Properties" on page 43 for details.

Code Example 11–1 shows a typical attach(9E) entry point with emphasis on creating the device's minor node and the nblocks property.

CODE EXAMPLE 11–1 Block Driver `attach(9E)` Routine

```
static int
xxattach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
     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 mutexs and condition variables
       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,
                   minor_number,  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.
             */
             xsp->nblocks = size of device in 512 byte blocks;
            if (ddi_prop_update_int(makedevice(DDI_MAJOR_T_UNKNOWN,
                   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 9
       default:
              return (DDI_FAILURE);
     }
}
```

Properties are associated with device numbers. In Code Example 11–1, `attach`(9E) builds a device number using `makedevice`(9F). At this point, however, only the minor number component of the device number is known, so it must use the special major number `DDI_MAJOR_T_UNKNOWN` to build the device number.

Controlling Device Access

This section describes aspects of the open(9E) and close(9E) entry points that are specific to block device drivers. See Chapter 10 for more information on open(9E) and close(9E).

open(9E)

The open(9E) entry point is used to gain access to a given device. The open(9E) routine of a block driver is called when a user thread issues an open(2) or mount(2) system call on a block special file associated with the minor device, or when a layered driver calls open(9E). See "File I/O" on page 168 for more information.

The open(9E) entry point should check for the following:

- The device can be opened; for example, it is online and ready.
- The device can be opened as requested; the device supports the operation, and the device's current state does not conflict with the request.
- The caller has permission to open the device.

Code Example 11–2 demonstrates a block driver open(9E) entry point.

CODE EXAMPLE 11–2 Block Driver open(9E) Routine

```
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);
}
```

The `otyp` argument is used to specify the type of open on the device. `OTYP_BLK` is the typical open type for a block device. A device can be opened several times with `otyp` set to `OTYP_BLK`, although `close`(9E) will be called only once when the final close of type `OTYP_BLK` has occurred for the device. `otyp` is set to `OTYP_LYR` if the device is being used as a layered device. For every open of type `OTYP_LYR`, the layering driver issues a corresponding close of type `OTYP_LYR`. The example keeps track of each type of open so the driver can determine when the device is not being used in `close`(9E).

close(9E)

The arguments of the `close`(9E) entry point are identical to arguments of `open`(9E), except that `dev` is the device number, as opposed to a pointer to the device number.

The `close`(9E) routine should verify `otyp` in the same way as was described for the `open`(9E) entry point. In Code Example 11–3, `close`(9E) must determine when the device can really be closed based on the number of block opens and layered opens.

CODE EXAMPLE 11–3 Block Device `close(9E)` Routine

```
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(9E)

The strategy(9E) entry point is used to read and write data buffers to and from a block device. The name *strategy* refers to the fact that this entry point might implement some optimal strategy for ordering requests to the device.

strategy(9E) can be written to process one request at a time (synchronous transfer), or to queue multiple requests to the device (asynchronous transfer). When choosing a method, the abilities and limitations of the device should be taken into account.

The strategy(9E) routine is passed a pointer to a buf(9S) structure. This structure describes the transfer request, and contains status information on return. buf(9S) and strategy(9E) are the focus of block device operations.

buf Structure

The following buf structure members are important to block drivers:

```
      int                   b_flags;          /* Buffer Status */
    struct buf             *av_forw;       /* Driver work list link */
    struct buf             *av_back;       /* Driver work lists 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 */
```

b_flags contains status and transfer attributes of the buf structure. If B_READ is set, the buf structure indicates a transfer from the device to memory; otherwise, it indicates a transfer from memory to the device. If the driver encounters an error during data transfer, it should set the B_ERROR field in the b_flags member and provide a more specific error value in b_error. Drivers should use bioerror(9F) rather than setting B_ERROR.

Caution - Drivers should never clear `b_flags`.

`av_forw` and `av_back` are pointers that the driver can use to manage a list of buffers by the driver. See “Asynchronous Data Transfers” on page 178 for a discussion of the `av_forw` and `av_back` pointers.

`b_bcount` specifies the number of bytes to be transferred by the device.

`b_un.b_addr` is the kernel virtual address of the data buffer.

`b_blkno` is the starting 32-bit logical block number on the device for the data transfer, expressed in `DEV_BSIZE` (512 bytes) units. The driver should use either `b_blkno` or `b_lblkno`, but not both.

`b_lblkno` is the starting 64-bit logical block number on the device for the data transfer, expressed in `DEV_BSIZE` (512 bytes) units. The driver should use either `b_blkno` or `b_lblkno`, but not both.

`b_resid` is set by the driver to indicate the number of bytes that were not transferred because of an error. See Code Example 11–8 for an example of setting `b_resid`. The `b_resid` member is overloaded: it is also used by `disksort`(9F).

`b_error` is set to an error number by the driver when a transfer error occurs. It is set in conjunction with the `b_flags` `B_ERROR` bit. See `Intro`(9E) for details regarding error values. Drivers should use `bioerror`(9F) rather than setting `b_error` directly.

`b_private` is for exclusive use by the driver to store driver-private data.

`b_edev` contains the device number of the device involved in the transfer.

bp_mapin(9F)

When a `buf` structure pointer is passed into the device driver’s `strategy`(9E) routine, the data buffer referred to by `b_un.b_addr` is not necessarily mapped in the kernel’s address space. This means that the driver cannot directly access the data. Most block-oriented devices have DMA capability, and therefore do not need to access the data buffer directly. Instead, they use the DMA mapping routines to allow the device’s DMA engine to do the data transfer. For details about using DMA, see Chapter 8.

If a driver needs to directly access the data buffer (as opposed to having the device access the data), it must first map the buffer into the kernel’s address space using `bp_mapin`(9F). `bp_mapout`(9F) should be used when the driver no longer needs to access the data directly.

Caution - `bp_mapout(9F)` should only be called on buffers that have been allocated and are owned by the device driver. It must not be called on buffers passed to the driver through the `strategy(9E)` entry point (for example a file system). Because `bp_mapin(9F)` does not keep a reference count, `bp_mapout(9F)` will remove any kernel mapping that a layer above the device driver might rely on.

Synchronous Data Transfers

This section presents a simple method for performing synchronous I/O transfers. It assumes that the hardware is a simple disk device that can transfer only one data buffer at a time using DMA, and that the disk can be spun up and spun down by software command. The device driver's `strategy`(9E) routine waits for the current request to be completed before accepting a new one. The device interrupts when the transfer is complete or when an error occurs.

1. **Check for invalid `buf(9S)` requests.**

   Check the `buf`(9S) structure passed to `strategy`(9E) for validity. All drivers should check that:

   1. The request begins at a valid block. The driver converts the `b_blkno` field to the correct device offset and then determines if the offset is valid for the device.
   2. The request does not go beyond the last block on the device.
   3. Device-specific requirements are met.

   If an error is encountered, the driver should indicate the appropriate error with `bioerror`(9F) and complete the request by calling `biodone`(9F). `biodone`(9F) notifies the caller of `strategy`(9E) that the transfer is complete (in this case, because of an error).

2. **Check whether the device is busy.**

   Synchronous data transfers allow single-threaded access to the device. The device driver enforces this by maintaining a busy flag (guarded by a mutex), and by waiting on a condition variable with `cv_wait`(9F) when the device is busy.

   If the device is busy, the thread waits until a `cv_broadcast`(9F) or `cv_signal`(9F) from the interrupt handler indicates that the device is no longer busy. See Chapter 3 for details on condition variables.

   When the device is no longer busy, the `strategy`(9E) routine marks it as busy and prepares the buffer and the device for the transfer.

3. **Set up the buffer for DMA.**

Prepare the data buffer for a DMA transfer by allocating a DMA handle using `ddi_dma_alloc_handle`(9F) and binding the data buffer to the handle using `ddi_dma_buf_bind_handle`(9F). See Chapter 8 for information on setting up DMA resources and related data structures.

4. Begin the transfer.

At this point, a pointer to the `buf`(9S) structure is saved in the state structure of the device. This is so that the interrupt routine can complete the transfer by calling `biodone`(9F).

The device driver then accesses device registers to initiate a data transfer. In most cases, the driver should protect the device registers from other threads by using mutexes. In this case, because `strategy`(9E) is single-threaded, guarding the device registers is not necessary. See Chapter 3 for details about data locks.

Once the executing thread has started the device's DMA engine, the driver can return execution control to the calling routine. (See Code Example 11–4.)

CODE EXAMPLE 11–4 Synchronous Block Driver `strategy(9E)` Routine

```
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 (see Chapter 9),
     mark the device busy with pm_busy_components(9F)
   , and then ensure that the device
     is powered up by calling ddi_dev_is_needed(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. Handle the interrupting device.

When the device finishes the data transfer it generates an interrupt, which eventually results in the driver's interrupt routine being called. Most drivers specify the state structure of the device as the argument to the interrupt routine when registering interrupts (see `ddi_add_intr`(9F) and "Registering Interrupts" on page 90). The interrupt routine can then access the `buf`(9S) structure being transferred, plus any other information available from the state structure.

The interrupt handler should check the device's status register to determine if the transfer completed without error. If an error occurred, the handler should indicate the appropriate error with `bioerror`(9F). The handler should also clear the pending interrupt for the device and then complete the transfer by calling `biodone`(9F).

As the final task, the handler clears the busy flag and calls `cv_signal`(9F) or `cv_broadcast`(9F) on the condition variable, signaling that the device is no longer busy. This allows other threads waiting for the device (in `strategy`(9E)) to proceed with the next data transfer.

CODE EXAMPLE 11–5 Synchronous Block Driver Interrupt Routine

```
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);
    }
```

Asynchronous Data Transfers

This section presents a method for performing asynchronous I/O transfers. The driver queues the I/O requests and then returns control to the caller. Again, the assumption is that the hardware is a simple disk device that allows one transfer at a time. The device interrupts when a data transfer has completed or when an error occurs.

1. **Check for invalid `buf(9S)` requests.**

   As in the synchronous case, the device driver should check the `buf`(9S) structure passed to `strategy`(9E) for validity. See “Synchronous Data Transfers” on page 175 for more details.

2. **Enqueue the request.**

   Unlike synchronous data transfers, a driver does not wait for an asynchronous request to complete. Instead, it adds the request to a queue. The head of the queue can be the current transfer, or a separate field in the state structure can be used to hold the active request (as in this example). If the queue was initially empty, then the hardware is not busy, and `strategy`(9E) starts the transfer before returning. Otherwise, whenever a transfer completes and the queue is non-empty, the interrupt routine begins a new transfer. This example actually places the decision of whether to start a new transfer into a separate routine for convenience.

The driver can use the `av_forw` and the `av_back` members of the `buf`(9S) structure to manage a list of transfer requests. A single pointer can be used to manage a singly linked list, or both pointers can be used together to build a doubly linked list. The device hardware specification will specify which type of list management (such as insertion policies) will optimize the performance of the device. The transfer list is a per-device list, so the head and tail of the list are stored in the state structure.

Code Example 11–6 is designed to allow multiple threads access to the driver shared data, so it is extremely important to identify any such data (such as the transfer list) and protect it with a mutex. See Chapter 3 for more details about mutex locks.

CODE EXAMPLE 11–6 Asynchronous Block Driver `strategy(9E)` Routine

```
static int
   xxstrategy(struct buf *bp)
   {
        struct xxstate *xsp;
        minor_t instance;
        instance = getminor(bp->b_edev);
        xsp = ddi_get_soft_state(statep, instance);
        ...
        validate transfer request
        ...
        Add the request to the end of the queue. Depending on the device, a sorting algorithm, such as disksort(9F)
   may 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);
   }
```

3. Start the first transfer.

Device drivers that implement queuing usually have a `start( )` routine. `start( )` is so called because it is this routine that dequeues the next request and starts the data transfer to or from the device. In this example, `start( )` processes all requests, regardless of the state of the device (busy or free).

Note - `start( )` must be written so that it can be called from any context, since it can be called by both the strategy routine (in kernel context) and the interrupt routine (in interrupt context).

`start( )` is called by `strategy(9F)` every time it queues a request so that an idle device can be started. If the device is busy, `start( )` returns immediately.

`start( )` is also called by the interrupt handler before it returns from a claimed interrupt so that a nonempty queue can be serviced. If the queue is empty, `start( )` returns immediately.

Since `start( )` is a private driver routine, it can take any arguments and return any type. Code Example 11–7 is written as if it will also be used as a DMA callback (although that portion is not shown), so it must take a `caddr_t` as an argument and return an `int`. See "Handling Resource Allocation Failures" on page 111 for more information about DMA callback routines.

CODE EXAMPLE 11–7 Block Driver `start( )` Routine

```
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 (see Chapter 9),
   mark the device busy with pm_busy_components, and then ensure that the device
         is powered up by calling ddi_dev_is_needed.
         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);
    }
```

4. Handle the interrupting device.

The interrupt routine is similar to the asynchronous version, with the addition of the call to `start()` and the removal of the call to `cv_signal`(9F).

CODE EXAMPLE 11–8 Asynchronous Block Driver Interrupt Routine

```
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)
   (9E), 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);
   }
```

Miscellaneous Entry Points

This section discusses the dump(9E) and print(9E) entry points.

dump(9E)

The dump(9E) entry point is used to copy a portion of virtual address space directly to the specified device in the case of a system failure. It is also used to copy the state of the kernel out to disk during a checkpoint operation (see cpr(7), dump(9E)). It must be capable of performing this operation without the use of interrupts, since they are disabled during the checkpoint operation.

```
int dump(dev_t dev, caddr_t addr, daddr_t blkno, int nblk)
```

dev is the device number of the device to dump to, addr is the base kernel virtual address at which to start the dump, blkno is the first block to dump to, and nblk is the number of blocks to dump. The dump depends upon the existing driver working properly.

print(9E)

```
int print(dev_t dev, char *str)
```

The print(9E) entry point is called by the system to display a message about an exception it has detected. print(9E) should call cmn_err(9F) to post the message to the console on behalf of the system. Here is an example:

```
static int
 xxprint(dev_t dev, char *str)
 {
     cmn_err(CE_CONT, ''xx: %s\n'', str);
     return (0);
 }
```

Disk Device Drivers

Disk devices represent an important class of block device drivers.

Disk ioctls

Solaris disk drivers need to support a minimum set of ioctl commands specific to Solaris disk drivers. These I/O controls are specified in the `dkio(7)` manual page. Disk I/O controls transfer disk information to or from the device driver. A Solaris disk device is one that is supported by disk utility commands such as `format(1M)` and `newfs(1M)`. Table 11–1 lists the mandatory Sun disk I/O controls.

TABLE 11–1 Mandatory Solaris Disk ioctls

ioctl	Description
`DKIOCINFO`	Returns information describing the disk controller
`DKIOCGAPART`	Returns a disk's partition map
`DKIOCSAPART`	Sets a disk's partition map
`DKIOCGGEOM`	Returns a disk's geometry
`DKIOCSGEOM`	Sets a disk's geometry
`DKIOCGVTOC`	Returns a disk's Volume Table of Contents
`DKIOCSVTOC`	Sets a disk's Volume Table of Contents

Disk Performance

The Solaris DDI/DKI provides facilities to optimize I/O transfers for improved file system performance. It supports a mechanism to manage the list of I/O requests so as to optimize disk access for a file system. See "Asynchronous Data Transfers" on page 178 for a description of enqueuing an I/O request.

The `diskhd` structure is used to manage a linked list of I/O requests.

```
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 */
};
```

The diskhd data structure has two buf pointers that the driver can manipulate. The av_forw pointer points to the first active I/O request. The second pointer, av_back, points to the last active request on the list.

A pointer to this structure is passed as an argument to disksort(9F), along with a pointer to the current buf structure being processed. The disksort(9F) routine is used to sort the buf requests in a fashion that optimizes disk seek and then inserts the buf pointer into the diskhd list. The disksort program uses the value that is in b_resid of the buf structure as a sort key. The driver is responsible for setting this value. Most Sun disk drivers use the cylinder group as the sort key. This tends to optimize the file system read-ahead accesses.

Once data has been added to the diskhd list, the device needs to transfer the data. If the device is not busy processing a request, the xxstart( ) routine pulls the first buf structure off the diskhd list and starts a transfer.

If the device is busy, the driver should return from the xxstrategy( ) entry point. Once the hardware is done with the data transfer, it generates an interrupt. The driver's interrupt routine is then called to service the device. After servicing the interrupt, the driver can then call the start( ) routine to process the next buf structure in the diskhd list.

CHAPTER 12

Mapping Device or Kernel Memory

Some device drivers allow applications to access device or kernel memory using mmap(2). Examples are frame buffer drivers, which allow the frame buffer to be mapped into a user thread, or a pseudo driver that communicates with an application using a shared kernel memory pool. This chapter describes how to associate device or kernel memory with user mappings.

Memory Mapping Operations

In general, the steps a driver must take to export device or kernel memory are:

1. Set the D_DEVMAP flag in the cb_flag flag of the cb_ops(9S) structure.
2. Define a devmap(9E) driver entry point to export the mapping.
3. Use devmap_devmem_setup(9F) to set up user mappings to a device. To set up user mappings to kernel memory, use devmap_umem_setup(9F).

Exporting the Mapping

The devmap(9E) entry point is called as a result of the mmap(2) system call. devmap(9E) is used to:

- Validate the user mapping to the device or kernel memory.
- Translate the logical offset within the application mapping to the corresponding offset within the device or kernel memory.
- Pass the mapping information to the system for setting up the mapping.

devmap(9E)

```
int devmap(dev_t dev, devmap_cookie_t handle, offset_t off,
     size_t len, size_t *maplen, uint_t model);
```

dev	Device whose memory is to be mapped
handle	Device-mapping handle that the system creates and uses to describe a mapping to contiguous device or kernel memory
off	Logical offset within the application mapping which has to be translated by the driver to the corresponding offset within the device or kernel memory
len	Length (in bytes) of the memory being mapped.
maplen	Allows driver to associate different kernel memory regions or multiple physically discontiguous memory regions with one contiguous user application mapping
model	Data model type of the current thread

The system creates multiple mapping handles in one `mmap`(2) system call (for example, if the mapping contains multiple physically discontiguous memory regions).

Initially `devmap`(9E) is called with parameters `off` and `len`, which were passed by the application to `mmap`(2). `devmap`(9E) sets `*maplen` to the length from `off` to the end of a contiguous memory region. `*maplen` must be rounded up to a multiple of a page size. If `*maplen` is set to less than the original mapping length `len`, the system will repeatedly call `devmap`(9E) with a new mapping handle and adjusted `off` and `len` parameters until the initial mapping length is satisfied.

If a driver supports multiple application data models, `model` has to be passed to `ddi_model_convert_from`(9F) to determine whether there is a data model mismatch between the current thread and the device driver. The device driver might have to adjust the shape of data structures before exporting them to a user thread that supports a different data model. See Appendix C for more details.

`devmap`(9E) must return `-1` if the logical offset, `off`, is out of the range of memory exported by the driver.

Associating Device Memory With User Mappings

devmap_devmem_setup(9F) is provided to export device memory to user applications.

Note - devmap_devmem_setup(9F) has to be called from the driver's devmap(9E) entry point.

```
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);
```

where

handle Opaque device-mapping handle that the system uses to identify the mapping

dip Pointer to the device's dev_info structure

callbackops Pointer to a devmap_callback_ctl(9S) structure that allows the driver to be notified of user events on the mapping

rnumber Index number to the register address space set

roff Offset into the device memory

len Length in bytes that is exported

maxprot Allows the driver to specify different protections for different regions within the exported device memory

flags Must be set to DEVMAP_DEFAULTS

accattrp Pointer to a ddi_device_acc_attr(9S) structure

roff and len describe a range within the device memory specified by the register set rnumber. The register specifications referred to by rnumber are described by the reg property. For devices with only one register set, pass zero for rnumber. The range described by roff and len are made accessible to the user's application mapping at the offset passed in by the devmap(9E) entry point. Usually the driver passes the devmap(9E) offset directly to devmap_devmem_setup(9F). The return address of mmap(2) then maps to the beginning address of the register set.

maxprot allows the driver to specify different protections for different regions within the exported device memory. For example, one region might not allow write access by setting only PROT_READ and PROT_USER.

Code Example 12–1 shows how to export device memory to an application. The driver first determines whether the requested mapping falls within the device memory region. The size of the device memory is determined using ddi_dev_regsize(9F). The length of the mapping is rounded up to a multiple of a page size using ptob(9F) and btopr(9F), and devmap_devmem_setup(9F) is called to export the device memory to the application.

CODE EXAMPLE 12–1 devmap_devmem_setup(9F) Routine

```
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)
              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);
}
```

Associating Kernel Memory With User Mappings

Some device drivers might need to allocate kernel memory that is made accessible to user programs by using `mmap`(2). Examples of this are setting up shared memory for communication between two applications or between driver and application.

In general, the steps for exporting kernel memory to user applications are:

1. Allocate kernel memory using `ddi_umem_alloc`(9F).
2. Export the memory using `devmap_umem_setup`(9F).
3. Free the memory using `ddi_umem_free`(9F) when no longer needed.

Allocating Kernel Memory for User Access

`ddi_umem_alloc`(9F) is provided to allocate kernel memory that is exported to applications:

```
void *ddi_umem_alloc(size_t size, int flag, ddi_umem_cookie_t *cookiep);
```

size	Number of bytes to allocate
flag	Used to determine the sleep conditions and the memory type
cookiep	Pointer to a kernel memory cookie

`ddi_umem_alloc`(9F) allocates page-aligned kernel memory and returns a pointer to the allocated memory. The initial contents of the memory is zero-filled. The number of bytes allocated is a multiple of the system page size (roundup of `size`). The allocated memory can be used in the kernel and can be exported to applications. `cookiep` is a pointer to the kernel memory cookie that describes the kernel memory being allocated. It is used in `devmap_umem_setup`(9F) when the driver exports the kernel memory to a user application.

The `flag` argument indicates whether `ddi_umem_alloc`(9F) will block or return immediately, and whether the allocated kernel memory is pageable. Table 12–1 lists the values for `flag`.

TABLE 12–1 `ddi_umem_alloc(9F)` flag Values

Values	Indicated Action
`DDI_UMEM_NOSLEEP`	Driver does not need to wait for memory to become available. Return NULL if memory unavailable.
`DDI_UMEM_SLEEP`	Driver can wait indefinitely for memory to become available.
`DDI_UMEM_PAGEABLE`	Driver allows memory to be paged out. If not set, the memory is locked down.

Code Example 12–2 shows how to allocate kernel memory for application access. The driver exports one page of kernel memory, which is used by multiple applications as a shared memory area. The memory is allocated in `segmap`(9E) when an application maps the shared page the first time. An additional page is allocated if the driver has to support multiple application data models (for example a 64-bit driver exporting memory to 64-bit and 32-bit applications). 64-bit applications share the first page, and 32-bit applications share the second page.

CODE EXAMPLE 12–2 `ddi_umem_alloc(9F)` Routine

```
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;
        #ifdef  _MULTI_DATAMODEL
            /* 64-bit driver supports 64-bit and 32-bit applications */
                mem_size = ptob(2);
        #else
                mem_size = ptob(1);
        #endif /* _MULTI_DATAMODEL */

        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);
}
```

Exporting Kernel Memory to Applications

devmap_umem_setup(9F) is provided to export kernel memory to user applications. devmap_umem_setup(9F) must be called from the driver's devmap(9E) entry point:

```
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);
```

handle	Opaque structure used to describe the mapping
dip	Pointer to the device's dev_info structure.
callbackops	Pointer to a devmap_callback_ctl(9S) structure
cookie	Kernel memory cookie returned by ddi_umem_alloc(9F)
koff	Offset into the kernel memory specified by cookie
len	Length in bytes that is exported
maxprot	Specifies the maximum protection possible for the exported mapping
flags	Must be set to DEVMAP_DEFAULTS
accattrp	Pointer to a ddi_device_acc_attr(9S) structure

handle is a device-mapping handle that the system uses to identify the mapping. It is passed in by the devmap(9E) entry point. dip is a pointer to the device's dev_info structure. callbackops allows the driver to be notified of user events on the mapping. Most drivers set callbackops to NULL when kernel memory is exported.

koff and len specify a range within the kernel memory allocated by ddi_umem_alloc(9F). This range will be made accessible to the user's application mapping at the offset passed in by the devmap(9E) entry point. Usually the driver will pass the devmap(9E) offset directly to devmap_umem_setup(9F). The return address of mmap(2) will then map to the kernel address returned by ddi_umem_alloc(9F). koff and len must be page aligned.

maxprot enables the driver to specify different protections for different regions within the exported kernel memory. For example, one region might not allow write access by only setting PROT_READ and PROT_USER.

Code Example 12–3 shows how to export kernel memory to an application. The driver first checks if the requested mapping falls within the allocated kernel memory region. If a 64-bit driver receives a mapping request from a 32-bit application, the request is redirected to the second page of the kernel memory area. This ensures that only applications compiled to the same data model will share the same page.

CODE EXAMPLE 12–3 devmap_umem_setup(9F) Routine

```
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);

        #ifdef  _MULTI_DATAMODEL
        if (ddi_model_convert_from(model) == DDI_MODEL_ILP32) {
                /* request from 32-bit application. Skip first page */
                off += ptob(1);
        }
        #endif  /* _MULTI_DATAMODEL */
        /* 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);
}
```

Freeing Kernel Memory Exported for User Access

When the driver is unloaded, the memory that was allocated by ddi_umem_alloc(9F) must be freed by calling ddi_umem_free(9F).

```
void ddi_umem_free(ddi_umem_cookie_t cookie);
```

cookie	Kernel memory cookie returned by ddi_umem_alloc(9F)

cookie is the kernel memory cookie returned by ddi_umem_alloc(9F).

CHAPTER 13

Device Context Management

Some device drivers, such as those for graphics hardware, provide user processes with direct access to the device. These devices often require that only one process at a time accesses the device.

This chapter describes the set of interfaces that allow device drivers to manage access to such devices.

What Is a Device Context?

The *context* of a device is the current state of the device hardware. The device driver manages the device context for a process on behalf of the process. It must maintain a separate device context for each process that accesses the device. It is the device driver's responsibility to restore the correct device context when a process accesses the device.

Context Management Model

An accelerated frame buffer is an example of a device that allows user processes (such as graphics applications) to directly manipulate the control registers of the device through memory-mapped access. Because these processes are not using the traditional I/O system calls (`read`(2), `write`(2), and `ioctl`(2)), the device driver is no longer called when a process accesses the device. However, it is important that the device driver be notified when a process is about to access a device so that it can restore the correct device context and provide any needed synchronization.

To resolve this problem, the device context management interfaces enable a device driver to be notified when a user process accesses memory-mapped regions of the device, and to control accesses to the device's hardware. Synchronization and management of the various device contexts are responsibilities of the device driver. When a user process accesses a mapping, the device driver must restore the correct device context for that process.

A device driver will be notified whenever one of the following events occurs:

- Access to a mapping by a user process
- Duplication of a mapping by a user process
- Freeing of a mapping by a user process
- Creation of a mapping by a user process

Figure 13–1 shows multiple user processes that have memory-mapped a device. The driver has granted process B access to the device, and process B no longer notifies the driver of accesses. However, the driver *is* still notified if either process A or process C accesses the device.

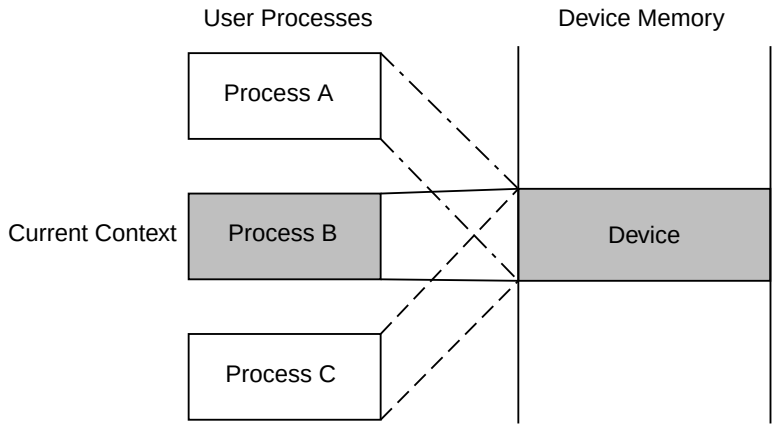


Figure 13–1 Device Context Management

At some point in the future, process A accesses the device. The device driver is notified of this and blocks future access to the device by process B. It then saves the device context for process B, restores the device context of process A, and grants access to process A, as illustrated in Figure 13–2. At this point, the device driver will be notified if either process B or process C accesses the device.

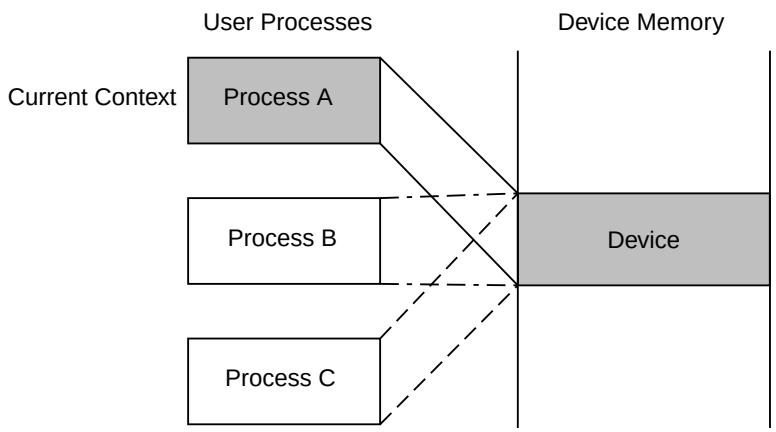


Figure 13–2 Device Context Switched to User Process A

Multiprocessor Considerations

On a multiprocessor machine, multiple processes could be attempting to access the device at the same time. This can cause thrashing. Some devices require a longer time to restore a device context. To prevent more CPU time from being used to restore a device context than to actually use that device context, the minimum time that a process needs to have access to the device can be set using `devmap_set_ctx_timeout`(9F).

The kernel guarantees that once a device driver has granted access to a process, no other process will be allowed to request access to the same device for the time interval specified by `devmap_set_ctx_timeout`(9F).

Context Management Operation

In general, the steps for performing device context management are:

1. Define a `devmap_callback_ctl`(9S) structure.
2. Allocate space to save device context if necessary.
3. Set up user mappings to the device and driver notifications with `devmap_devmem_setup`(9F).
4. Manage user access to the device with `devmap_load`(9F) and `devmap_unload`(9F).
5. Free the device context structure, if needed.

devmap_callback_ctl Structure

The device driver must allocate and initialize a devmap_callback_ctl(9S) structure to inform the system of its device context management entry point routines. This structure contains the following fields:

```
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	The version number of the devmap_callback_ctl(9S) structure. It must be set to DEVMAP_OPS_REV.
devmap_map	Must be set to the address of the driver's devmap_map(9E) entry point.
devmap_access	Must be set to the address of the driver's devmap_access(9E) entry point.
devmap_dup	Must be set to the address of the driver's devmap_dup(9E) entry point.
devmap_unmap	Must be set to the address of the driver's devmap_unmap(9E) entry point.

Device Context Management Entry Points

The following device driver entry points are used to manage device context.

devmap_map(9E)

```
int xxdevmap_map(devmap_cookie_t handle, dev_t dev, uint_t flags,
    offset_t offset, size_t len, void **new_devprivate);
```

This entry point is called after the driver returns from its devmap(9E) entry point and the system has established the user mapping to the device memory. The devmap_map(9E) entry point enables a driver to perform additional processing or to allocate mapping specific private data. For example, in order to support context

switching, the driver has to allocate a context structure and associate it with the mapping.

The system expects the driver to return a pointer to the allocated private data in `*new_devprivate`. The driver must store `offset` and `len`, which define the range of the mapping, in its private data. Later, when the system calls `devmap_unmap`(9E), the driver will use `offset` and `len` stored in `new_devprivate` to check if the entire mapping, or just a part of it, is being unmapped.

`flags` indicates whether the driver should allocate a private context for the mapping. For example, a driver can allocate a memory region to store the device context if `flags` is set to `MAP_PRIVATE`, or it might return a pointer to a shared region if `MAP_SHARED` is set.

Code Example 13–1 shows an example of a `devmap_map`(9E) entry point. The driver allocates a new context structure and saves relevant parameters passed in by the entry point. Then the mapping is assigned a new context by either allocating a new one or attaching it to an already existing shared context. The minimum time interval that the mapping should have access to the device is set to one millisecond.

CODE EXAMPLE 13–1 `devmap_map(9E)` Routine

```
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 strcuture */
        *new_devprivate = newctx;
        return(0);
}
```

devmap_access(9E)

```
int xxdevmap_access(devmap_cookie_t handle, void *devprivate,
    offset_t offset, size_t len, uint_t type, uint_t rw);
```

This entry point is called when an access is made to a mapping whose translations are invalid. Mapping translations are invalidated when the mapping is created with devmap_devmem_setup(9F) in response to mmap(2), duplicated by fork(2), or explicitly invalidated by a call to devmap_unload(9F).

handle	Mapping handle of the mapping that was accessed by a user process.
devprivate	Pointer to the driver private data associated with the mapping.
offset	Offset within the mapping that was accessed.
len	Length in bytes of the memory being accessed.
type	Type of access operation.
rw	Specifies the direction of access.

The system expects devmap_access(9E) to call either devmap_do_ctxmgt(9F) or devmap_default_access(9F) to load the memory address translations before it returns. For mappings that support context switching, the device driver should call devmap_do_ctxmgt(9F). This routine is passed all parameters from devmap_access(9E), as well as a pointer to the driver entry point devmap_contextmgt(9E), which handles the context switching. For mappings that do not support context switching, the driver should call devmap_default_access(9F), whose only purpose is to call devmap_load(9F) to load the user translation.

Code Example 13–2 shows an example of a devmap_access(9E) entry point. The mapping is devided into two regions. The region starting at offset OFF_CTXMG with a length of CTXMGT_SIZE bytes supports context management. The rest of the mapping supports default access.

CODE EXAMPLE 13–2 devmap_access(9E) Routine

```
#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(9E)

```
int xxdevmap_contextmgt(devmap_cookie_t handle, void *devprivate,
    offset_t offset, size_t len, uint_t type, uint_t rw);
```

In general, devmap_contextmgt(9E) should call devmap_unload(9F), with the handle of the mapping that currently has access to the device, to invalidate the translations for that mapping. This ensures that a call to devmap_access(9E) occurs for the current mapping the next time it is accessed. To validate the mapping translations for the mapping that caused the access event to occur, the driver must restore the device context for the process requesting access and call devmap_load(9F) on the handle of the mapping that generated the call to this entry point.

Accesses to portions of mappings that have had their mapping translations validated by a call to devmap_load(9F) do not generate a call to devmap_access(9E). A subsequent call to devmap_unload(9F) invalidates the mapping translations and allows devmap_access(9E) to be called again.

If either devmap_load(9F) or devmap_unload(9F) returns an error, devmap_contextmgt(9E) should immediately return that error. If the device driver encounters a hardware failure while restoring a device context, a -1 should be returned. Otherwise, after successfully handling the access request, devmap_contextmgt(9E) should return zero. A return of other than zero from devmap_contextmgt(9E) will cause a SIGBUS or SIGSEGV to be sent to the process.

Code Example 13–3 shows how to manage a one-page device context.

Note - xxctxsave( ) and xxctxrestore( ) are device-dependent context save and restore functions. xxctxsave( ) reads data from the registers using the Solaris 8 DDI/DKI data access routines and saves it in the soft state structure. xxctxrestore( ) takes data saved in the soft state structure and writes it to device registers using the Solaris 8 DDI/DKI data access routines.

CODE EXAMPLE 13–3 devmap_contextmgt(9E) Routine

```
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;
        /* 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(9E)

```
int xxdevmap_dup(devmap_cookie_t handle, void *devprivate,
    devmap_cookie_t new_handle, void **new_devprivate);
```

This entry point is called when a device mapping is duplicated, for example, by a user process calling `fork`(2). The driver is expected to generate new driver private data for the new mapping.

`handle`	Mapping handle of the mapping being duplicated.
`new_handle`	Mapping handle of the mapping that was duplicated.
`devprivate`	Pointer to the driver private data associated with the mapping being duplicated.
`*new_devprivate`	Should be set to point to the new driver private data for the new mapping.

Mappings created with `devmap_dup`(9E) will, by default, have their mapping translations invalidated. This will force a call to the `devmap_access`(9E) entry point the first time the mapping is accessed.

Code Example 13–4 shows a devmap_dup(9E) routine.

CODE EXAMPLE 13–4 devmap_dup(9E) Routine

```
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) {
                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(9E)

```
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);
```

This entry point is called when a mapping is unmapped. This can be caused by a user process exiting or calling the munmap(2) system call.

handle	Mapping handle of the mapping being freed.
devprivate	Pointer to the driver private data associated with the mapping.
off	Offset within the logical device memory at which the unmapping begins.
len	Length in bytes of the memory being unmapped.
new_handle1	Handle that the system uses to describe the new region that ends at off - 1. new_handle1 may be NULL.

new_devprivate1	Pointer to be filled in by the driver with the driver -private mapping data for the new region that ends at off - 1. It is ignored if new_handle1 is NULL.
new_handle2	Handle that the system uses to describe the new region that begins at off + len. new_handle2 may be NULL.
new_devprivate2	Pointer to be filled in by the driver with the driver private mapping data for the new region that begins at off + len. It is ignored if new_handle2 is NULL.

The devmap_unmap(9E) routine is expected to free any driver private resources that were allocated when this mapping was created, either by devmap_map(9E) or by devmap_dup(9E). If only a part of the mapping is being unmapped, the driver must allocate a new private data for the remaining mapping before freeing the old private data. There is no need to call devmap_unload(9F) on the handle of the mapping being freed, even if it is the mapping with the valid translations. However, to prevent future problems in devmap_access(9E), the device driver should make sure that its representation of the current mapping is set to "no current mapping".

Code Example 13–5 shows an example of a devmap_unmap(9E) routine.

CODE EXAMPLE 13–5 devmap_unmap(9E) Routine

```
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);
                *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));
}
```

Associating User Mappings With Driver Notifications

When a user process requests a mapping to a device with `mmap`(2), the driver's `segmap`(9E) entry point is called. The driver must use `ddi_devmap_segmap`(9F) or `devmap_setup`(9F) when setting up the memory mapping if it needs to manage device contexts. Both functions will call the driver's `devmap`(9E) entry point, which uses `devmap_devmem_setup`(9F) to associate the device memory with the user mapping. See Chapter 12 for details on how to map device memory.

For the driver to get notifications on accesses to the user mapping, it has to inform the system of the `devmap_callback_ctl`(9S) entry points. It does this by providing a pointer to a `devmap_callback_ctl(9S)` structure to `devmap_devmem_setup(9F)`. A `devmap_callback_ctl(9S)` structure describes a set of context management entry points that are called by the system to notify a device driver to manage events on the device mappings.

The system associates each mapping with a mapping handle. This handle is passed to each of the context management entry points. The mapping handle can be used to invalidate and validate the mapping translations. If the driver *invalidates* the mapping translations, it will be notified of any future access to the mapping. If the driver *validates* the mapping translations, it will no longer be notified of accesses to the mapping. Mappings are always created with the mapping translations invalidated so that the driver will be notified on first access to the mapping.

Code Example 13–6 shows how to set up a mapping using the device context management interfaces.

CODE EXAMPLE 13–6 devmap(9E) Entry Point With Context Management Support

```
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;

        /* 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);
}
```

Managing Mapping Accesses

The device driver is notified when a user process accesses an address in the memory-mapped region that does not have valid mapping translations. When the access event occurs, the mapping translations of the process that currently has access to the device must be invalidated. The device context of the process requesting access to the device must be restored, and the translations of the mapping of the process requesting access must be validated.

The functions devmap_load(9F) and devmap_unload(9F) are used to validate and invalidate mapping translations.

devmap_load(9F)

```
int devmap_load(devmap_cookie_t handle, offset_t offset,
    size_t len, uint_t type, uint_t rw);
```

devmap_load(9F) validates the mapping translations for the pages of the mapping specified by handle,offset, and len. By validating the mapping translations for these pages, the driver is telling the system not to intercept accesses to these pages of the mapping and to allow accesses to proceed without notifying the device driver.

devmap_load(9F) must be called with the offset and the handle of the mapping that generated the access event for the access to complete. If devmap_load(9F) is not called on this handle, the mapping translations will not be validated, and the process will receive a SIGBUS.

devmap_unload(9F)

```
int devmap_unload(devmap_cookie_t handle, offset_t offset,
     size_t len);
```

devmap_unload(9F) invalidates the mapping translations for the pages of the mapping specified by handle, offset, and len. By invalidating the mapping translations for these pages, the device driver is telling the system to intercept accesses to these pages of the mapping and notify the device driver the next time these pages of the mapping are accessed by calling the devmap_access(9E) entry point.

For both functions, requests affect the entire page containing the offset and all pages up to and including the entire page containing the last byte, as indicated by offset + len. The device driver must ensure that for each page of device memory being mapped only one process has valid translations at any one time.

Both functions return zero if they are successful. If, however, there was an error in validating or invalidating the mapping translations, that error is returned to the device driver. It is the device driver's responsibility to return this error to the system.

CHAPTER 14

SCSI Target Drivers

SCSI Target Driver Overview

The Solaris DDI/DKI divides the software interface to SCSI devices into two major parts: *target* drivers and *Host bus adapter (HBA)* drivers. *Target* refers to a driver for a device on a SCSI bus, such as a disk or a tape drive. *Host bus adapter* refers to the driver for the SCSI controller on the host machine. SCSA defines the interface between these two components. This chapter discusses target drivers only. See Chapter 15 for information on host bus adapter drivers.

Note - The terms “host bus adapter” or “HBA” used in this manual are equivalent to the phrase “host adapter” defined in SCSI specifications.

Target drivers can be either character or block device drivers, depending on the device. Drivers for tape drives are usually character device drivers, while disks are handled by block device drivers. This chapter describes how to write a SCSI target driver and discusses the additional requirements that SCSA places on block and character drivers for SCSI target devices.

Reference Documents

The following reference documents provide supplemental information needed by the designers of target drivers and host bus adapter drivers.

Small Computer System Interface 2 (SCSI-2), ANSI/NCITS X3.131-1994, Global Engineering Documents, 1998. ISBN 1199002488.

The Basics of SCSI, Fourth Edition, ANCOT Corporation, 1998. ISBN 0963743988.

Also refer to the SCSI command specification for the target device, provided by the hardware vendor.

Sun Common SCSI Architecture Overview

The Sun Common SCSI Architecture (SCSA) is the Solaris DDI/DKI programming interface for the transmission of SCSI commands from a target driver to a host bus adapter driver. This interface is independent of the type of host bus adapter hardware, the platform, the processor architecture, and the SCSI command being transported across the interface.

By conforming to the SCSA, the target driver can pass any SCSI command to a target device without knowledge of the hardware implementation of the host bus adapter.

The SCSA conceptually separates building the SCSI command (by the target driver) from transporting the SCSI command and data across the SCSI bus. The architecture defines the software interface between high-level and low-level software components. The higher-level software component consists of one or more SCSI target drivers, which translate I/O requests into SCSI commands appropriate for the peripheral device. Figure 14–1 illustrates the SCSI architecture.

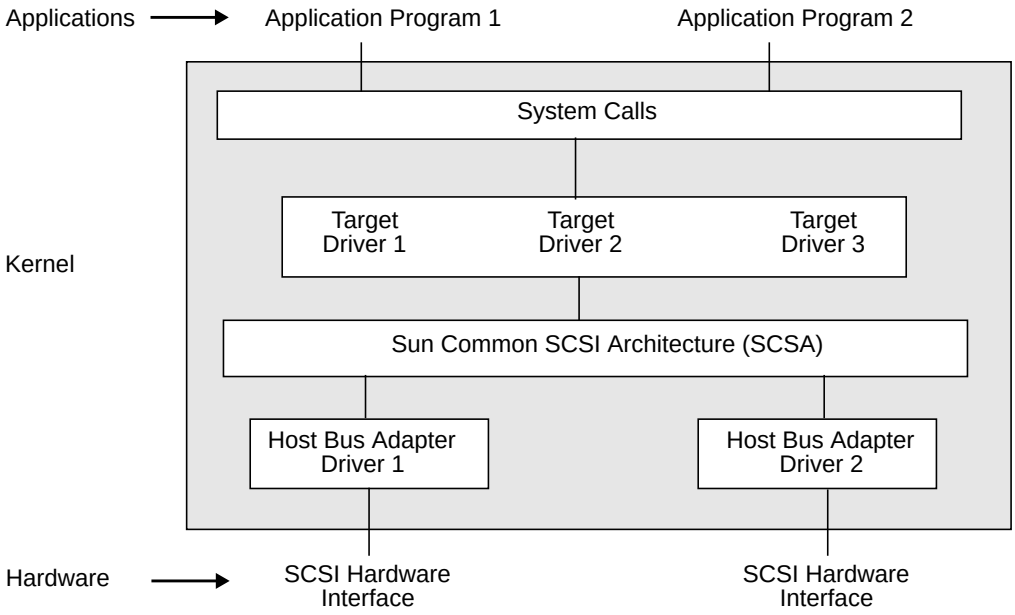


Figure 14–1 SCSA Block Diagram

The lower-level software component consists of a SCSA interface layer and one or more host bus adapter drivers. The target driver is responsible for the generation of the proper SCSI commands required to execute the desired function and for processing the results.

General Flow of Control

Assuming no transport errors occur, the following steps describe the general flow of control for a read or write request.

1. The target driver's `read(9E)` or `write(9E)` entry point is invoked. `physio(9F)` is used to lock down memory, prepare a `buf` structure, and call the strategy routine.
2. The target driver's `strategy`(9E) routine checks the request and allocates a `scsi_pkt`(9S) using `scsi_init_pkt`(9F). The target driver initializes the packet and sets the SCSI command descriptor block (CDB) using the `scsi_setup_cdb`(9F) function. The target driver also specifies a timeout and provides a pointer to a callback function, which is called by the host bus adapter driver on completion of the command. The `buf`(9S) pointer should be saved in the SCSI packet's target-private space.
3. The target driver submits the packet to the host bus adapter driver using `scsi_transport`(9F). The target driver is then free to accept other requests. The target driver should not access the packet while it is in transport. If either the host

bus adapter driver or the target supports queueing, new requests can be submitted while the packet is in transport.

4. As soon as the SCSI bus is free and the target not busy, the host bus adapter driver selects the target and passes the CDB. The target executes the command and performs the requested data transfers.
5. After the target sends completion status and the command completes, the host bus adapter driver notifies the target driver by calling the completion function that was specified in the SCSI packet. At this time the host bus adapter driver is no longer responsible for the packet, and the target driver has regained ownership of the packet.
6. The SCSI packet's completion routine analyzes the returned information and determines whether the SCSI operation was successful. If a failure has occurred, the target driver retries the command by calling `scsi_transport`(9F) again. If the host bus adapter driver does not support auto request sense, the target driver must submit a request sense packet to retrieve the sense data in the event of a check condition.
7. If either the command was completed successfully or cannot be retried, the target driver calls `scsi_destroy_pkt`(9F), which synchronizes the data and frees the packet. If the target driver needs to access the data before freeing the packet, it calls `scsi_sync_pkt`(9F).
8. Finally, the target driver notifies the application program that originally requested the read or write that the transaction is complete, either by returning from the `read`(9E) entry point in the driver (for a character device) or indirectly through `biodone`(9F).

SCSA allows the execution of many of such operations, both overlapped and queued, at various points in the process. The model places the management of system resources on the host bus adapter driver. The software interface enables the execution of target driver functions on host bus adapter drivers using SCSI bus adapters of varying degrees of sophistication.

SCSA Functions

SCSA defines functions to manage the allocation and freeing of resources, the sensing and setting of control states, and the transport of SCSI commands. These functions are listed in Table 14–1.

TABLE 14–1 Standard SCSA Functions

Function Name	Category
`scsi_init_pkt(9F)`	Resource management
`scsi_sync_pkt(9F)`	
`scsi_dmafree(9F)`	
`scsi_destroy_pkt(9F)`	
`scsi_alloc_consistent_buf(9F)`	
`scsi_free_consistent_buf(9F)`	
`scsi_transport(9F)`	Command transport
`scsi_ifgetcap(9F)`	Transport information and control
`scsi_ifsetcap(9F)`	
`scsi_abort(9F)`	Error handling
`scsi_reset(9F)`	
`scsi_poll(9F)`	Polled I/O
`scsi_probe(9F)`	Probe functions
`scsi_unprobe(9F)`	
`scsi_setup_cdb(9F)`	CDB initialization function

Note - If a driver needs to work with a SCSI-1 device, it should use the `makecom`(9F).

SCSI Target Drivers

Hardware Configuration File

Because SCSI devices are not self-identifying, a hardware configuration file is required for a target driver (see `driver.conf`(4) and `scsi`(4) for details). A typical configuration file looks like this:

```
name="xx" class="scsi" target=2 lun=0;
```

The system reads the file during autoconfiguration and uses the *class* property to identify the driver's possible parent. The system then attempts to attach the driver to any parent driver that is of class *scsi*. All host bus adapter drivers are of this class. Using the *class* property rather than the *parent* property enables the target driver to be attached to any host bus adapter driver that finds the expected device at the specified *target* and *lun* ids. The target driver is responsible for verifying this in its `probe`(9E) routine.

Declarations and Data Structures

Target drivers must include the header file `<sys/scsi/scsi.h>`.

SCSI target drivers must use the following command to generate a binary module:

```
ld -r xx xx.o -N"misc/scsi"
```

`scsi_device` Structure

The host bus adapter driver allocates and initializes a `scsi_device`(9S) structure for the target driver before either the `probe`(9E) or `attach`(9E) routine is called. This structure stores information about each SCSI logical unit, including pointers to information areas that contain both generic and device-specific information. There is one `scsi_device`(9S) structure for each logical unit attached to the system. The target driver can retrieve a pointer to this structure by calling `ddi_get_driver_private`(9F).

Caution - Because the host bus adapter driver uses the private field in the target device's `dev_info` structure, target drivers must not use `ddi_set_driver_private`(9F).

The `scsi_device`(9S) structure contains the following fields:

```
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;
};
```

`sd_address` Data structure that is passed to the SCSI resource allocation routines.

`sd_dev` Pointer to the target's `dev_info` structure.

`sd_mutex` Mutex for use by the target driver. This is initialized by the host bus adapter driver and can be used by the target driver as a per-device mutex. Do not hold this mutex across a call to `scsi_transport`(9F) or `scsi_poll`(9F). See Chapter 3 for more information on mutexes.

`sd_inq` Pointer for the target device's SCSI inquiry data. The `scsi_probe`(9F) routine allocates a buffer, fills it in with inquiry data, and attaches it to this field.

`sd_sense` Pointer to a buffer to contain SCSI request sense data from the device. The target driver must allocate and manage this buffer itself; see "`attach(9E)`" on page 217.

`sd_private` Pointer field for use by the target driver. It is commonly used to store a pointer to a private target driver state structure.

scsi_pkt Structure

This structure contains the following fields:

```
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 */
};
```

`pkt_address` Target device's address set by `scsi_init_pkt`(9F)

`pkt_private` — Place to store private data for the target driver. It is commonly used to save the `buf`(9S) pointer for the command.

`pkt_comp` — Address of the completion routine. The host bus adapter driver calls this routine when it has transported the command. This does not mean that the command succeeded; the target might have been busy or might not have responded before the time-out time elapsed (see the description for `pkt_time` field). The target driver must supply a valid value in this field, though it can be `NULL` if the driver does not want to be notified.

Note - There are two different SCSI callback routines. The `pkt_comp` field identifies a *completion callback* routine, which is called when the host bus adapter completes its processing. There is also a *resource callback* routine, called when currently unavailable resources are likely to be available (as in `scsi_init_pkt`(9F)).

`pkt_flags` — Provides additional control information, for example, to transport the command without disconnect privileges (`FLAG_NODISCON`) or to disable callbacks (`FLAG_NOINTR`). See `scsi_pkt`(9S) for details.

`pkt_time` — Time-out value (in seconds). If the command is not completed within this time, the host bus adapter calls the completion routine with `pkt_reason` set to `CMD_TIMEOUT`. The target driver should set this field to longer than the maximum time the command might take. If the timeout is zero, no timeout is requested. Timeout starts when the command is transmitted on the SCSI bus.

`pkt_scbp` — Pointer to the SCSI status completion block; this is filled in by the host bus adapter driver.

`pkt_cdbp` — Pointer to the SCSI command descriptor block, the actual command to be sent to the target device. The host bus adapter driver does not interpret this field. The target driver must fill it in with a command that the target device can process.

`pkt_resid` — Residual of the operation. When allocating DMA resources for a command `scsi_init_pkt`(9F), `pkt_resid` indicates the number of bytes for which DMA resources could *not* be allocated because of DMA hardware scatter-gather or other device limitations. After command transport, `pkt_resid` indicates the number of

data bytes *not* transferred; this is filled in by the host bus adapter driver before the completion routine is called.

`pkt_state` — Indicates the state of the command. The host bus adapter driver fills in this field as the command progresses. One bit is set in this field for each of the five following command states:

- `STATE_GOT_BUS` – Acquired the bus
- `STATE_GOT_TARGET` – Selected the target
- `STATE_SENT_CMD` – Sent the command
- `STATE_XFERRED_DATA` – Transferred data (if appropriate)
- `STATE_GOT_STATUS` – Received status from the device

`pkt_statistics` — Contains transport-related statistics set by the host bus adapter driver.

`pkt_reason` — Gives the reason the completion routine was called. The main function of the completion routine is to decode this field and take the appropriate action. If the command completed—in other words, if there were no transport errors—this field is set to `CMD_CMPLT`; other values in this field indicate an error. After a command is completed, the target driver should examine the `pkt_scbp` field for a check condition status. See `scsi_pkt`(9S) for more information.

Autoconfiguration

SCSI target drivers must implement the standard autoconfiguration routines `_init`(9E), `_fini`(9E), and `_info`(9E). See “Loadable Driver Interfaces” on page 65 for more information.

`probe`(9E), `attach`(9E), `detach`(9E), and `getinfo`(9E) are also required, but they must perform SCSI (and SCSA) specific processing.

`probe(9E)`

SCSI target devices are not self-identifying, so target drivers must have a `probe`(9E) routine. This routine must determine whether the expected type of device is present and responding.

The general structure and return codes of the probe(9E) routine are the same as those of other device drivers. SCSI target drivers must use the scsi_probe(9F) routine in their probe(9E) entry point. scsi_probe(9F) sends a SCSI inquiry command to the device and returns a code indicating the result. If the SCSI inquiry command is successful, scsi_probe(9F) allocates a scsi_inquiry(9S) structure and fills it in with the device's inquiry data. Upon return from scsi_probe(9F), the sd_inq field of the scsi_device(9S) structure points to this scsi_inquiry(9S) structure.

Because probe(9E) must be stateless, the target driver must call scsi_unprobe(9F) before probe(9E) returns, even if scsi_probe(9F) fails.

Code Example 14–1 shows a typical probe(9E) routine. It retrieves its scsi_device(9S) structure from the private field of its dev_info structure. It also retrieves the device's SCSI target and logical unit numbers so that it can print them in messages. The probe(9E) routine then calls scsi_probe(9F) to verify that the expected device (a printer in this case) is present.

If scsi_probe(9F) succeeds, it has attached the device's SCSI inquiry data in a scsi_inquiry(9S) structure to the sd_inq field of the scsi_device(9S) structure. The driver can then determine if the device type is a printer (reported in the inq_dtype field). If it is, the type is reported with scsi_log(9F), using scsi_dname(9F) to convert the device type into a string.

CODE EXAMPLE 14–1 SCSI Target Driver probe(9E) Routine

```
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 may be powered off,
            * in which case we may be able to successfully
            * probe it at some future time - referred to
            * 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);
}
```

A more thorough `probe`(9E) routine could also check other fields of the `scsi_inquiry`(9S) structure as necessary to make sure that the device is of the type expected by a particular driver.

`attach(9E)`

After the `probe`(9E) routine has verified that the expected device is present, `attach`(9E) is called. This routine allocates and initializes any per-instance data, creates minor device node information, and restores the hardware state of a device when the device or the system has been suspended. See "`attach(9E)`" on page 71 for details of this. In addition to these steps, a SCSI target driver again calls `scsi_probe`(9F) to retrieve the device's inquiry data and also creates a SCSI request sense packet. If the attach is successful, the attach function should not call `scsi_unprobe`(9F).

Three routines are used to create the request sense packet: `scsi_alloc_consistent_buf`(9F), `scsi_init_pkt`(9F), and `scsi_setup_cdb`(9F). `scsi_alloc_consistent_buf`(9F) allocates a buffer suitable for consistent DMA and returns a pointer to a `buf`(9S) structure. The advantage of a consistent buffer is that no explicit synchronization of the data is required. In other words, the target driver can access the data after the callback. The `sd_sense` element of the device's `scsi_device`(9S) structure must be initialized

with the address of the sense buffer. `scsi_init_pkt`(9F) creates and partially initializes a `scsi_pkt`(9S) structure. `scsi_setup_cdb`(9F) creates a SCSI command descriptor block, in this case creating a SCSI request sense command.

Note that since a SCSI device is not self-identifying and does not have a *reg* property, the driver must set the *pm-hardware-state* property to inform the framework that this device needs to be suspended and resumed.

Code Example 14–2 shows the SCSI target driver's `attach`(9E) routine.

CODE EXAMPLE 14–2 SCSI Target Driver `attach(9E)` Routine

```
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 Chapter 9
            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. 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
    return (DDI_FAILURE);
}
```

detach(9E)

The detach(9E) entry point is the inverse of attach(9E); it must free all resources that were allocated in attach(9E). If successful, the detach should call scsi_unprobe(9F). Code Example 14–3 shows a target driver detach(9E) routine.

CODE EXAMPLE 14–3 SCSI Target Driver detach(9E) Routine

```
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 Chapter 9
    default:
            return (DDI_FAILURE);
    }
}
```

getinfo(9E)

The getinfo(9E) routine for SCSI target drivers is much the same as for other drivers; see "getinfo(9E)" on page 77 for more information on DDI_INFO_DEVT2INSTANCE case. However, in the DDI_INFO_DEVT2DEVINFO case of the getinfo(9E) routine, the target driver must return a pointer to its dev_info node. This pointer can be saved in the driver state structure or can be retrieved from the sd_dev field of the scsi_device(9S) structure. Code Example 14–4 shows an alternative SCSI target driver getinfo(9E) code fragment.

CODE EXAMPLE 14–4 Alternative SCSI Target Driver getinfo(9E) Code Fragment

```
...
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);
...
```

Resource Allocation

To send a SCSI command to the device, the target driver must create and initialize a scsi_pkt(9S) structure and pass it to the host bus adapter driver.

scsi_init_pkt(9F)

The scsi_init_pkt(9F) routine allocates and zeros a scsi_pkt(9S) structure; it also sets pointers to pkt_private, *pkt_scbp, and *pkt_cdbp. Additionally, it provides a callback mechanism to handle the case where resources are not available. This function takes the following arguments:

`ap` Pointer to a `scsi_address` structure. This is the `sd_address` field of the device's `scsi_device`(9S) structure.

`pktp` Pointer to the `scsi_pkt`(9S) structure to be initialized. If this is set to `NULL`, a new packet is allocated.

`bp` Pointer to a `buf`(9S) structure. If this is non-NULL and contains a valid byte count, DMA resources are allocated.

`cmdlen` Length of the SCSI command descriptor block in bytes.

`statuslen` Required length of the SCSI status completion block in bytes.

`privatelen` Number of bytes to allocate for the `pkt_private` field.

`flags` Set of flags. Possible bits include:

- `PKT_CONSISTENT` – This bit must be set if the DMA buffer was allocated using `scsi_alloc_consistent_buf`(9F). In this case, the host bus adapter driver guarantees that the data transfer is properly synchronized before performing the target driver's command completion callback.

- `PKT_DMA_PARTIAL` – This bit can be set if the driver accepts a partial DMA mapping. If set, `scsi_init_pkt`(9F) allocates DMA resources with the `DDI_DMA_PARTIAL` flag set. The `pkt_resid` field of the `scsi_pkt`(9S) structure can be returned with a nonzero residual, indicating the number of bytes for which `scsi_init_pkt`(9F) was unable to allocate DMA resources.

`callback` Specifies the action to take if resources are not available. If set to `NULL_FUNC`, `scsi_init_pkt`(9F) returns immediately (returning NULL). If set to `SLEEP_FUNC`, it does not return until resources are available. Any other valid kernel address is interpreted as the address of a function to be called when resources are likely to be available.

`arg` Parameter to pass to the callback function.

The `scsi_init_pkt`(9F) routine synchronizes the data prior to transport. If the driver needs to access the data after transport, it should call `scsi_sync_pkt`(9F) to flush any intermediate caches. The `scsi_sync_pkt`(9F) routine can be used to synchronize any cached data.

scsi_sync_pkt(9F)

If the target driver needs to resubmit the packet after changing the data, `scsi_sync_pkt`(9F) must be called before calling `scsi_transport`(9F). However, if the target driver does not need to access the data, there is no need to call `scsi_sync_pkt`(9F) after the transport.

scsi_destroy_pkt(9F)

The `scsi_destroy_pkt`(9F) routine synchronizes any remaining cached data associated with the packet, if necessary, and then frees the packet and associated command, status, and target driver-private data areas. This routine should be called in the command completion routine.

scsi_alloc_consistent_buf(9F)

For most I/O requests, the data buffer passed to the driver entry points is not accessed directly by the driver; it is just passed on to `scsi_init_pkt`(9F). If a driver sends SCSI commands that operate on buffers the driver examines itself (such as the SCSI request sense command), the buffers should be DMA consistent. The `scsi_alloc_consistent_buf`(9F) routine allocates a `buf`(9S) structure and a data buffer suitable for DMA-consistent operations. The HBA will perform any necessary synchronization of the buffer before performing the command completion callback.

scsi_free_consistent_buf(9F)

`scsi_free_consistent_buf`(9F) releases a `buf`(9S) structure and the associated data buffer allocated with `scsi_alloc_consistent_buf`(9F). See "`attach(9E)`" on page 217 and "`detach(9E)`" on page 219 for examples.

Caution - `scsi_alloc_consistent_buf(9F)` uses scarce system resources; use it sparingly.

Building and Transporting a Command

The host bus adapter driver is responsible for transmitting the command to the device and handling the low-level SCSI protocol. The `scsi_transport`(9F) routine hands a packet to the host bus adapter driver for transmission. It is the target driver's responsibility to create a valid `scsi_pkt`(9S) structure.

Building a Command

The routine scsi_init_pkt(9F) allocates space for a SCSI CDB, allocates DMA resources if necessary, and sets the pkt_flags field:

```
pkt = scsi_init_pkt(&sdp->sd_address, NULL, bp,
CDB_GROUP0, 1, 0, 0, SLEEP_FUNC, NULL);
```

This example creates a new packet and allocates DMA resources as specified in the passed buf(9S) structure pointer. A SCSI CDB is allocated for a Group 0 (6-byte) command, the pkt_flags field is set to zero, but no space is allocated for the pkt_private field. This call to scsi_init_pkt(9F), because of the SLEEP_FUNC parameter, waits indefinitely for resources if none are currently available.

The next step is to initialize the SCSI CDB, using the scsi_setup_cdb(9F) function:

```
    if (scsi_setup_cdb((union scsi_cdb *)pkt->pkt_cdbp,
         SCMD_READ, bp->b_blkno, bp->b_bcount >> DEV_BSHIFT, 0) == 0)
         goto failed;
```

This example builds a Group 0 command descriptor block and fills in the pkt_cdbp field as follows:

- The command itself (byte 0) is set from the parameter (SCMD_READ).
- The address field (bits 0-4 of byte 1 and bytes 2 and 3) is set from bp->b_blkno.
- The count field (byte 4) is set from the last parameter. In this case it is set to bp->b_bcount >> DEV_BSHIFT, where DEV_BSHIFT is the byte count of the transfer converted to the number of blocks.

Note - scsi_setup_cdb(9F) does not support setting a target device's logical unit number (LUN) in bits 5-7 of byte 1 of the SCSI command block, as defined by SCSI-1. For SCSI-1 devices requiring the LUN bits set in the command block, use makecom_g0(9F) (or equivalent) rather than scsi_setup_cdb(9F).

After initializing the SCSI CDB, initialize three other fields in the packet and store as a pointer to the packet in the state structure.

```
pkt->pkt_private = (opaque_t)bp;
pkt->pkt_comp = xxcallback;
pkt->pkt_time = 30;
xsp->pkt = pkt;
```

The buf(9S) pointer is saved in the pkt_private field for later use in the completion routine.

Setting Target Capabilities

The target drivers use `scsi_ifsetcap`(9F) to set the capabilities of the host adapter driver. A *cap* is a name-value pair whose name is a null terminated character string and whose value is an integer. The current value of a capability can be retrieved using `scsi_ifgetcap`(9F). `scsi_ifsetcap`(9F) allows capabilities to be set for all targets on the bus.

In general, however, setting capabilities of targets that are not owned by the target driver is not recommended and is not universally supported by HBA drivers. Some capabilities (such as disconnect and synchronous) can be set by default by the HBA driver but others might need to be explicitly set by the target driver (wide-xfer or tagged-queueing, for example).

Transporting a Command

After creating and filling in the `scsi_pkt`(9S) structure, the final step is to hand it to the host bus adapter driver using `scsi_transport`(9F):

```
if (scsi_transport(pkt) != TRAN_ACCEPT) {
     bp->b_resid = bp->b_bcount;
     bioerror(bp, EIO);
     biodone(bp);
 }
```

The other return values from `scsi_transport`(9F) are:

- `TRAN_BUSY` – There is already a command in progress for the specified target.
- `TRAN_BADPKT` – The DMA count in the packet was too large, or the host adapter driver rejected this packet for other reasons.
- `TRAN_FATAL_ERROR` – The host adapter driver is unable to accept this packet.

Warning - The mutex `sd_mutex` in the `scsi_device`(9S) structure must not be held across a call to `scsi_transport(9F)`.

If `scsi_transport`(9F) returns `TRAN_ACCEPT`, the packet is the responsibility of the host bus adapter driver and should not be accessed by the target driver until the command completion routine is called.

Synchronous `scsi_transport(9F)`

If `FLAG_NOINTR` is set in the packet, then `scsi_transport`(9F) will not return until the command is complete, and no callback will be performed.

Note - `FLAG_NOINTR` should never be used in interrupt context.

Command Completion

Once the host bus adapter driver has done all it can with the command, it invokes the packet's completion callback routine, passing a pointer to the `scsi_pkt`(9S) structure as a parameter. The completion routine decodes the packet and takes the appropriate action.

Code Example 14–5 presents a simple completion callback routine. This code checks for transport failures and gives up rather than retry the command. If the target is busy, extra code is required to resubmit the command at a later time.

If the command results in a check condition, the target driver needs to send a request sense command, unless auto request sense has been enabled.

Otherwise, the command succeeded. If this is the end of processing for the command, it destroys the packet and calls `biodone`(9F).

In the event of a transport error (such as a bus reset or parity problem), the target driver can resubmit the packet using `scsi_transport`(9E). There is no need to change any values in the packet prior to resubmitting.

This example does not attempt to retry incomplete commands.

Note - Normally, the target driver's callback function is called in interrupt context. Consequently, the callback function should never sleep.

CODE EXAMPLE 14–5 SCSI Driver Completion Routine

```
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);
            }
        }
}
```

Reuse of Packets

A target driver can reuse packets in the following ways:

- Resubmit the packet unchanged.
- Use `scsi_sync_pkt`(9F) to synchronize the data, then process the data in the driver and resubmit.
- Free DMA resources, using `scsi_dmafree`(9F), and pass the `pkt` pointer to `scsi_init_pkt`(9F) for binding to a new `bp`. The target driver is responsible for reinitializing the packet. The CDB has to have the same length as the previous CDB.
- If partial DMA was allocated during the first call to `scsi_init_pkt`(9F), subsequent calls to `scsi_init_pkt`(9F) can be made for the same packet and `bp` to adjust the DMA resources to the next portion of the transfer.

Auto-Request Sense Mode

Auto-request sense mode is most desirable if tagged or untagged queuing is used. A contingent allegiance condition is cleared by any subsequent command and, consequently, the sense data is lost. Most HBA drivers will start the next command before performing the target driver callback. Other HBA drivers can use a separate and lower-priority thread to perform the callbacks, which might increase the time it takes to notify the target driver that the packet completed with a check condition. In this case, the target driver might not be able to submit a request sense command in time to retrieve the sense data.

To avoid this loss of sense data, the HBA driver, or controller, should issue a request sense command as soon as a check condition has been detected; this mode is known as auto-request sense mode. Note that not all HBA drivers are capable of auto-request sense mode, and some can only operate with auto-request-sense mode enabled.

A target driver enables auto-request-sense mode by using `scsi_ifsetcap`(9F). Code Example 14–6 shows enabling auto request sense.

CODE EXAMPLE 14–6 Enabling Auto Request Sense

```
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 cmd may
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) may 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);
    ...
}
```

When a packet is allocated using `scsi_init_pkt`(9F) and auto request sense is desired on this packet, then the target driver must request additional space for the status block to hold the auto request sense structure. The sense length used in the request sense command is `sizeof` (`struct scsi_extended_sense`). Auto request sense can be disabled per individual packet by just allocating `sizeof` (`struct scsi_status`) for the status block.

The packet is submitted using `scsi_transport`(9F) as usual. When a check condition occurs on this packet, the host adapter driver:

- Issues a request sense command if the controller doesn't have auto-request-sense capability.
- Obtains the sense data.
- Fills in the `scsi_arq_status` information in the packet's status block.
- Sets `STATE_ARQ_DONE` in the packet's `pkt_state` field.
- Calls the packet's callback handler (`pkt_comp`).

The target driver's callback routine should verify that sense data is available by checking the `STATE_ARQ_DONE` bit in `pkt_state`, which implies that a check condition has occurred and a request sense has been performed. If auto-request-sense has been temporarily disabled in a packet, there is no guarantee that the sense data can be retrieved at a later time.

The target driver should then verify whether the auto request sense command completed successfully and decode the sense data.

Dump Handling

dump(9E)

The dump(9E) entry point is used to copy a portion of virtual address space directly to the specified device in the case of system failure or checkpoint operation. See cpr(7) and dump(9E). The dump(9E) entry point must be capable of performing this operation without the use of interrupts.

dev is the device number of the dump device, addr is the kernel virtual address at which to start the dump, blkno is the first destination block on the device, and nblk is the number of blocks to dump.

CODE EXAMPLE 14–7 dump(9E) Routine

```
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) {
        (void) ddi_dev_is_needed(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 Options

SCSA defines a global variable, `scsi_options`, which can be used for debug and control. The defined bits in *scsi_options* can be found in the file `<sys/scsi/conf/autoconf.h>`. Table 14–2 shows their meanings when set.

TABLE 14–2 SCSA Options

Option	Description
`SCSI_OPTIONS_DR`	Enables global disconnect / reconnect.
`SCSI_OPTIONS_SYNC`	Enables global synchronous transfer capability.
`SCSI_OPTIONS_LINK`	Enables global link support.
`SCSI_OPTIONS_PARITY`	Enables global parity support.
`SCSI_OPTIONS_TAG`	Enables global tagged queuing support.

TABLE 14–2 SCSA Options *(continued)*

Option	Description
SCSI_OPTIONS_FAST	Enables global FAST SCSI support: 10MB/sec transfers, as opposed to 5 MB/sec.
SCSI_OPTIONS_FAST20	Enables global FAST20 SCSI support: 20MB/sec transfers.
SCSI_OPTIONS_FAST40	Enables global FAST40 SCSI support: 40MB/sec transfers.
SCSI_OPTIONS_FAST80	Enables global FAST80 SCSI support: 80MB/sec transfers.
SCSI_OPTIONS_WIDE	Enables global WIDE SCSI.

Note - The setting of *scsi_options* affects *all* host adapter and target drivers present on the system (as opposed to `scsi_ifsetcap(9F)`). Refer to `scsi_hba_attach`(9F) for information on controlling these options for a particular host adapter.

CHAPTER 15

SCSI Host Bus Adapter Drivers

This chapter contains information on creating SCSI host bus adapter (HBA) drivers and provides sample code illustrating the structure of a typical HBA driver and showing the use of the HBA driver interfaces provided by the Sun Common SCSI Architecture (SCSA).

As described in Chapter 14, the Solaris 8 DDI/DKI divides the software interface to SCSI devices into two major parts:

- Target devices and drivers
- Host bus adapter devices and drivers

Target device refers to a device on a SCSI bus, such as a disk or a tape drive. *Target driver* refers to a software component installed as a device driver. Each target device on a SCSI bus is controlled by one instance of the target driver.

Host bus adapter device refers to HBA hardware, such as an SBus or PCI SCSI adapter card. *Host bus adapter driver* refers to a software component installed as a device driver, such as the `esp` driver on a SPARC machine or the `ncrs` driver on an IA machine, and the `isp` driver which works on both. An instance of the HBA driver controls each of its host bus adapter devices configured in the system.

The Sun Common SCSI Architecture (SCSA) defines the interface between these target and HBA components.

Note - Understanding SCSI target drivers is an essential prerequisite to writing effective SCSI HBA drivers. For information on SCSI target drivers, see Chapter 14. Target driver developers can also benefit from reading this chapter.

The host bus adapter driver is responsible for:

- Managing host bus adapter hardware
- Accepting SCSI commands from the SCSI target driver
- Transporting the commands to the specified SCSI target device

- Performing any data transfers that the command requires
- Collecting status
- Handling auto-request sense (optional)
- Informing the target driver of command completion (or failure)

SCSI Interface

SCSA is the Solaris 8 DDI/DKI programming interface for the transmission of SCSI commands from a target driver to a host adapter driver. By conforming to the SCSA, the target driver can pass any combination of SCSI commands and sequences to a target device without knowledge of the hardware implementation of the host adapter. SCSA conceptually separates the building of a SCSI command (by the target driver) from the transporting of the command to and data to and from the SCSI bus (by the HBA driver) for the appropriate target device. SCSA manages the connections between the target and HBA drivers through an HBA *transport* layer, as shown in Figure 15–1.

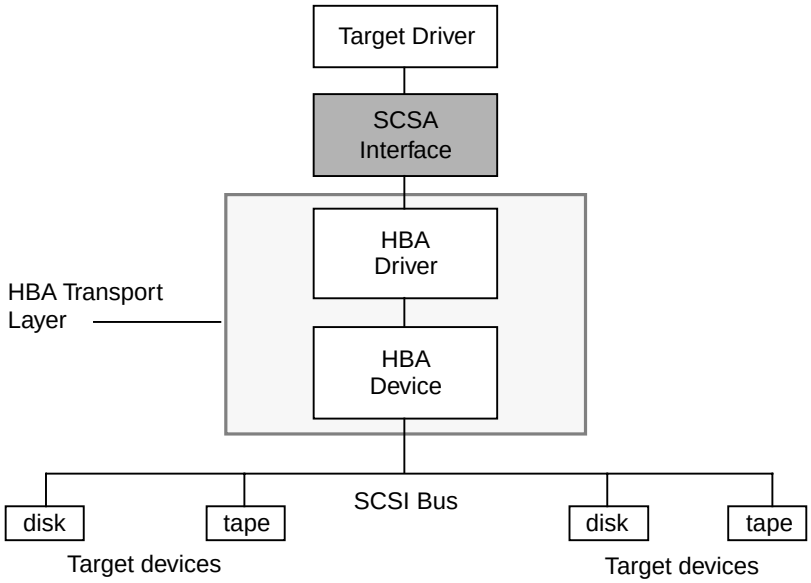


Figure 15–1 SCSA Interface

HBA Transport Layer

The *HBA transport layer* is a software and hardware layer responsible for transporting a SCSI command to a SCSI target device. The HBA driver provides resource allocation, DMA management, and transport services in response to requests made by SCSI target drivers through SCSA. The host adapter driver also manages the host adapter hardware and the SCSI protocols necessary to perform the commands. When a command has been completed, the HBA driver calls the target driver's SCSI `pkt` command completion routine.

Figure 15–2 illustrates this flow, with emphasis placed on the transfer of information from target drivers to SCSA to HBA drivers. Typical transport entry points and function calls are included.

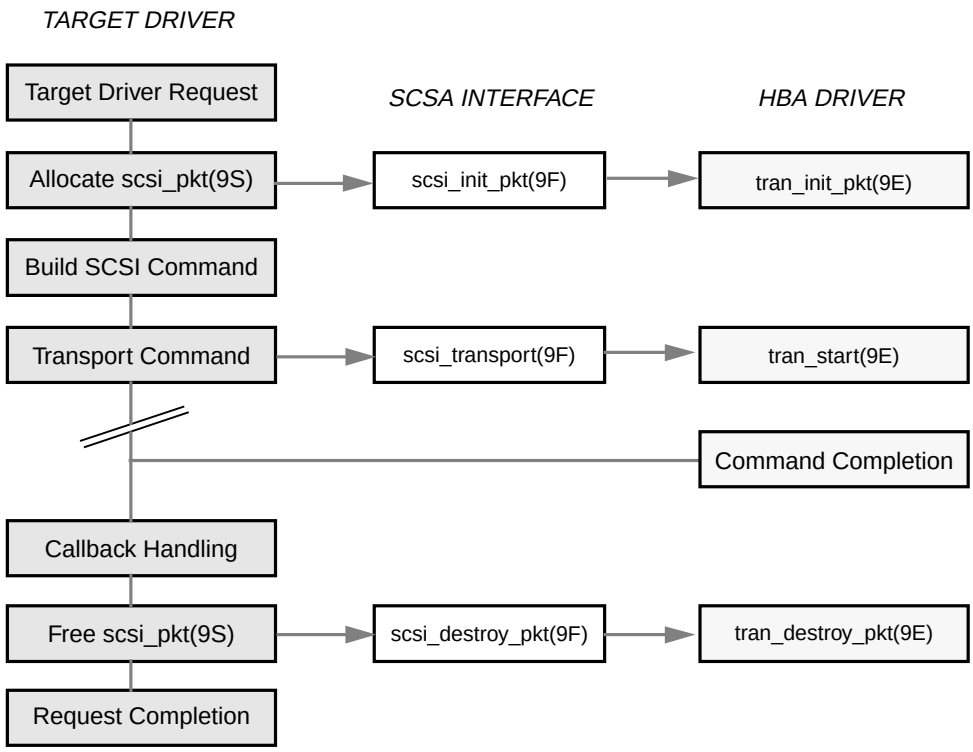


Figure 15–2 Transport Layer Flow

SCSA HBA Interfaces

SCSA HBA interfaces include HBA entry points, HBA data structures, and an HBA framework.

SCSA HBA Entry Point Summary

SCSA defines a number of HBA driver entry points, listed in Table 15–1. These entry points are called by the system when configuring a target driver instance connected to the HBA driver, or when the target driver makes a SCSA request. See “SCSA HBA Entry Points” on page 252 for more information.

TABLE 15–1 SCSA HBA Entry Point Summary

Function Name	Called as a Result of
`tran_tgt_init(9E)`	System attaching target device instance
`tran_tgt_probe(9E)`	Target driver calling `scsi_probe`(9F)
`tran_tgt_free(9E)`	System detaching target device instance
`tran_start(9E)`	Target driver calling `scsi_transport`(9F)
`tran_reset(9E)`	Target driver calling `scsi_reset`(9F)
`tran_abort(9E)`	Target driver calling `scsi_abort`(9F)
`tran_getcap(9E)`	Target driver calling `scsi_ifgetcap`(9F)
`tran_setcap(9E)`	Target driver calling `scsi_ifsetcap`(9F)
`tran_init_pkt(9E)`	Target driver calling `scsi_init_pkt`(9F)
`tran_destroy_pkt(9E)`	Target driver calling `scsi_destroy_pkt`(9F)

TABLE 15–1 SCSA HBA Entry Point Summary *(continued)*

Function Name	Called as a Result of
`tran_dmafree(9E)`	Target driver calling `scsi_dmafree`(9F)
`tran_sync_pkt(9E)`	Target driver calling `scsi_sync_pkt`(9F)
`tran_reset_notify(9E)`	Target driver calling `scsi_reset_notify`(9F)
`tran_quiesce(9E)`	System quiescing bus
`tran_unquiesce(9E)`	System resuming activity on bus
`tran_bus_reset(9E)`	System resetting bus

SCSA HBA Data Structures

SCSA defines data structures to enable the exchange of information between the target and HBA drivers. These data structures include:

- `scsi_hba_tran`(9S)
- `scsi_address`(9S)
- `scsi_device`(9S)
- `scsi_pkt`(9S)

`scsi_hba_tran` Structure

Each instance of an HBA driver must allocate a `scsi_hba_tran`(9S) structure using `scsi_hba_tran_alloc`(9F) in the `attach`(9E) entry point. `scsi_hba_tran_alloc`(9F) initializes the `scsi_hba_tran`(9S) structure before it returns. The HBA driver must initialize specific vectors in the transport structure to point to entry points within the HBA driver. Once initialized, the HBA driver exports the transport structure to SCSA by calling `scsi_hba_attach_setup`(9F).

Caution - Because SCSA keeps a pointer to the transport structure in the driver-private field on the devinfo node, HBA drivers must not use `ddi_set_driver_private`(9F). They can, however, use `ddi_get_driver_private`(9F) to retrieve the pointer to the transport structure.

The `scsi_hba_tran`(9S) structure contains the following fields:

```
struct scsi_hba_tran {
        dev_info_t      *tran_hba_dip;
        void            *tran_hba_private;      /* HBA softstate */
        void            *tran_tgt_private;      /* target-specific info */
        struct scsi_device      *tran_sd;
        int             (*tran_tgt_init)();
        int             (*tran_tgt_probe)();
        void            (*tran_tgt_free)();
        int             (*tran_start)();
        int             (*tran_reset)();
        int             (*tran_abort)();
        int             (*tran_getcap)();
        int             (*tran_setcap)();
        struct scsi_pkt *(*tran_init_pkt)();
        void            (*tran_destroy_pkt)();
        void            (*tran_dmafree)();
        void            (*tran_sync_pkt)();
        int             (*tran_reset_notify)();
        int             (*tran_get_bus_addr)();
        int             (*tran_get_name)();
        int             (*tran_clear_aca)();
        int             (*tran_clear_task_set)();
        int             (*tran_terminate_task)();
        int             (*tran_get_eventcookie)();
        int             (*tran_add_eventcall)();
        int             (*tran_remove_eventcall)();
        int             (*tran_post_event)();
        int             (*tran_quiesce)();
        int             (*tran_unquiesce)();
        int             (*tran_bus_reset)();
};
```

Note - Code fragments presented subsequently in this chapter use these fields to describe practical HBA driver operations. See “SCSA HBA Entry Points” on page 252 for more information.

`tran_hba_dip` — Pointer to the HBA device instance `dev_info` structure. The function `scsi_hba_attach_setup`(9F) sets this field.

`tran_hba_private` — Pointer to private data maintained by the HBA driver. Usually, `tran_hba_private` contains a pointer to the state structure of the HBA driver.

`tran_tgt_private`	Pointer to private data maintained by the HBA driver when using cloning. By specifying `SCSI_HBA_TRAN_CLONE` when calling `scsi_hba_attach_setup`(9F), the `scsi_hba_tran`(9S) structure is cloned once per target, permitting the HBA to initialize this field to point to a per-target instance data structure in the `tran_tgt_init`(9E) entry point. If `SCSI_HBA_TRAN_CLONE` is not specified, `tran_tgt_private` is NULL and must not be referenced. See "Transport Structure Cloning (Optional)" on page 242 for more information.
`tran_sd`	Pointer to a per-target instance `scsi_device`(9S) structure used when cloning. If `SCSI_HBA_TRAN_CLONE` is passed to `scsi_hba_attach_setup`(9F), `tran_sd` is initialized to point to the per-target `scsi_device` structure before any HBA functions are called on behalf of that target. If `SCSI_HBA_TRAN_CLONE` is not specified, `tran_sd` is NULL and must not be referenced. See "Transport Structure Cloning (Optional)" on page 242 for more information.
`tran_tgt_init`	Pointer to the HBA driver entry point called when initializing a target device instance. If no per-target initialization is required, the HBA can leave `tran_tgt_init` set to NULL.
`tran_tgt_probe`	Pointer to the HBA driver entry point called when a target driver instance calls `scsi_probe`(9F) to probe for the existence of a target device. If no target probing customization is required for this HBA, the HBA should set `tran_tgt_probe` to `scsi_hba_probe`(9F).
`tran_tgt_free`	Pointer to the HBA driver entry point called when a target device instance is destroyed. If no per-target deallocation is necessary, the HBA can leave `tran_tgt_free` set to NULL.
`tran_start`	Pointer to the HBA driver entry point called when a target driver calls `scsi_transport`(9F).
`tran_reset`	Pointer to the HBA driver entry point called when a target driver calls `scsi_reset`(9F).
`tran_abort`	Pointer to the HBA driver entry point called when a target driver calls `scsi_abort`(9F).

`tran_getcap`	Pointer to the HBA driver entry point called when a target driver calls `scsi_ifgetcap(9F)` .
`tran_setcap`	Pointer to the HBA driver entry point called when a target driver calls `scsi_ifsetcap(9F)`.
`tran_init_pkt`	Pointer to the HBA driver entry point called when a target driver calls `scsi_init_pkt`(9F).
`tran_destroy_pkt`	Pointer to the HBA driver entry point called when a target driver calls `scsi_destroy_pkt`(9F).
`tran_dmafree`	Pointer to the HBA driver entry point called when a target driver calls `scsi_dmafree`(9F).
`tran_sync_pkt`	Pointer to the HBA driver entry point called when a target driver calls `scsi_sync_pkt`(9F).
`tran_reset_notify`	Pointer to the HBA driver entry point called when a target driver calls `tran_reset_notify`(9E).

`scsi_address` Structure

The `scsi_address`(9S) structure provides transport and addressing information for each SCSI command allocated and transported by a target driver instance.

The `scsi_address`(9S) structure contains the following fields:

```
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;       /* Sublun on that Lun */
                                                /* Not used */
};
```

`a_hba_tran`	Pointer to the `scsi_hba_tran`(9S) structure, as allocated and initialized by the HBA driver. If `SCSI_HBA_TRAN_CLONE` was specified as the flag to `scsi_hba_attach_setup`(9F), `a_hba_tran` points to a copy of that structure.
`a_target`	Identifies the SCSI target on the SCSI bus.
`a_lun`	Identifies the SCSI logical unit on the SCSI target.

scsi_device Structure

The HBA framework allocates and initializes a scsi_device(9S) structure for each instance of a target device before calling an HBA driver's tran_tgt_init(9E) entry point. This structure stores information about each SCSI logical unit, including pointers to information areas that contain both generic and device-specific information. There is one scsi_device(9S) structure for each target device instance attached to the system.

If the per-target initialization is successful (in other words, if either tran_tgt_init(9E) returns success or the vector is NULL), the HBA framework will set the target driver's per-instance private data to point to the scsi_device(9S) structure, using ddi_set_driver_private(9F).

The scsi_device(9S) structure contains the following fields:

```
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 */
};
```

sd_address	Data structure that is passed to the SCSI resource allocation routines.
sd_dev	Pointer to the target's dev_info structure.
sd_mutex	Mutex for use by the target driver. This is initialized by the HBA framework and can be used by the target driver as a per-device mutex. This mutex should not be held across a call to scsi_transport(9F) or scsi_poll(9F). See Chapter 3 for more information on mutexes.
sd_inq	Pointer for the target device's SCSI inquiry data. The scsi_probe(9F) routine allocates a buffer, fills it in, and attaches it to this field.
sd_sense	Pointer to a buffer to contain Request Sense data from the device. The target driver must allocate and manage this buffer itself; see the target driver's attach(9E) routine in "attach(9E)" on page 71 for more information.
sd_private	Pointer field for use by the target driver. It is commonly used to store a pointer to a private target driver state structure.

scsi_pkt Structure

To execute SCSI commands, a target driver must first allocate a scsi_pkt(9S) structure for the command, specifying its own private data area length, the command status, and the command length. The HBA driver is responsible for implementing the packet allocation in the tran_init_pkt(9E) entry point. The HBA driver is also responsible for freeing the packet in its tran_destroy_pkt(9E) entry point. See scsi_pkt(9S) in Chapter 14, for more information.

The scsi_pkt(9S) structure contains these fields:

```
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 */
};
```

pkt_ha_private	Pointer to per-command HBA-driver private data.
pkt_address	Pointer to the scsi_address(9S) structure providing address information for this command.
pkt_private	Pointer to per-packet target-driver private data.
pkt_comp	Pointer to the target driver completion routine called by the HBA driver when the transport layer has completed this command.
pkt_flags	Flags for the command.
pkt_time	Specifies the completion timeout in seconds for the command.
pkt_scbp	Pointer to the status completion block for the command.
pkt_cdbp	Pointer to the command descriptor block (CDB) for the command.
pkt_resid	Count of the data bytes *not* transferred when the command has been completed or the amount of data for which resources have not been allocated. The HBA must modify this field during transport.

`pkt_state`	State of the command. The HBA must modify this field during transport.
`pkt_statistics`	Provides a history of the events the command experienced while in the transport layer. The HBA must modify this field during transport.
`pkt_reason`	Reason for command completion. The HBA must modify this field during transport.

Per-Target Instance Data

An HBA driver must allocate a `scsi_hba_tran`(9S) structure during `attach`(9E) and initialize the vectors in this transport structure to point to the required HBA driver entry points. This `scsi_hba_tran`(9S) structure is then passed into `scsi_hba_attach_setup`(9F).

The `scsi_hba_tran`(9S) structure contains a `tran_hba_private` field, which can be used to refer to the HBA driver's per-instance state.

Each `scsi_address`(9S) structure contains a pointer to the `scsi_hba_tran`(9S) structure and also provides the target (`a_target`) and logical unit (`a_lun`) addresses for the particular target device. Because every HBA driver entry point is passed a pointer to the `scsi_address`(9S) structure, either directly or indirectly through the `scsi_device`(9S) structure, the HBA driver can reference its own state and can identify the target device being addressed.

Figure 15–3 illustrates the HBA data structures for transport operations.

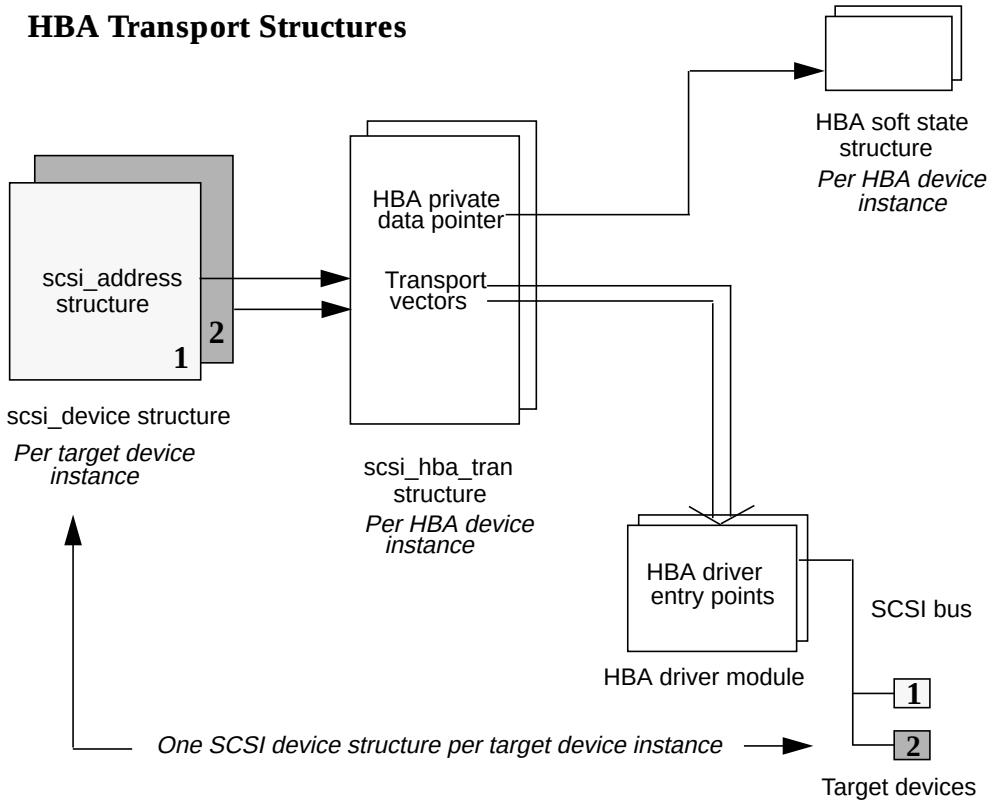


Figure 15–3 HBA Transport Sructures

Transport Structure Cloning (Optional)

Cloning can be useful if an HBA driver needs to maintain per-target private data in the `scsi_hba_tran`(9S) structure, or if it needs to maintain a more complex address than is provided in the `scsi_address`(9S) structure.

When cloning, the HBA driver must still allocate a `scsi_hba_tran`(9S) structure at `attach`(9E) time and initialize the `tran_hba_private` soft state pointer and HBA entry point vectors as before. The difference occurs when the framework begins to connect an instance of a target driver to the HBA driver. Before calling the HBA driver's `tran_tgt_init`(9E) entry point, the framework duplicates (clones) the `scsi_hba_tran`(9S) structure associated with that instance of the HBA. This means that each `scsi_address`(9S) structure, allocated and initialized for a particular target device instance, points to a per-target instance *copy* of the `scsi_hba_tran`(9S) structure, not to the `scsi_hba_tran`(9S) structure allocated by the HBA driver at `attach`(9E) time.

Two important pointers that an HBA driver can use when it has specified cloning are contained in the `scsi_hba_tran`(9S) structure. The first pointer is the `tran_tgt_private` field, which the driver can use to point to per-target HBA private data. This is useful, for example, if an HBA driver needs to maintain a more complex address than the `a_target` and `a_lun` fields in the `scsi_address`(9S) structure allow. The second pointer is the `tran_sd` field, which is a pointer to the `scsi_device`(9S) structure referring to the particular target device.

When specifying cloning, the HBA driver must allocate and initialize the per-target data and initialize the `tran_tgt_private` field to point to this data during its `tran_tgt_init`(9E) entry point. The HBA driver must free this per-target data during its `tran_tgt_free`(9E) entry point.

When cloning, the framework initializes the `tran_sd` field to point to the `scsi_device`(9S) structure before the HBA driver `tran_tgt_init`(9E) entry point is called. The driver requests cloning by passing the `SCSI_HBA_TRAN_CLONE` flag to `scsi_hba_attach_setup`(9F). Figure 15–4 illustrates the HBA data structures for cloning transport operations.

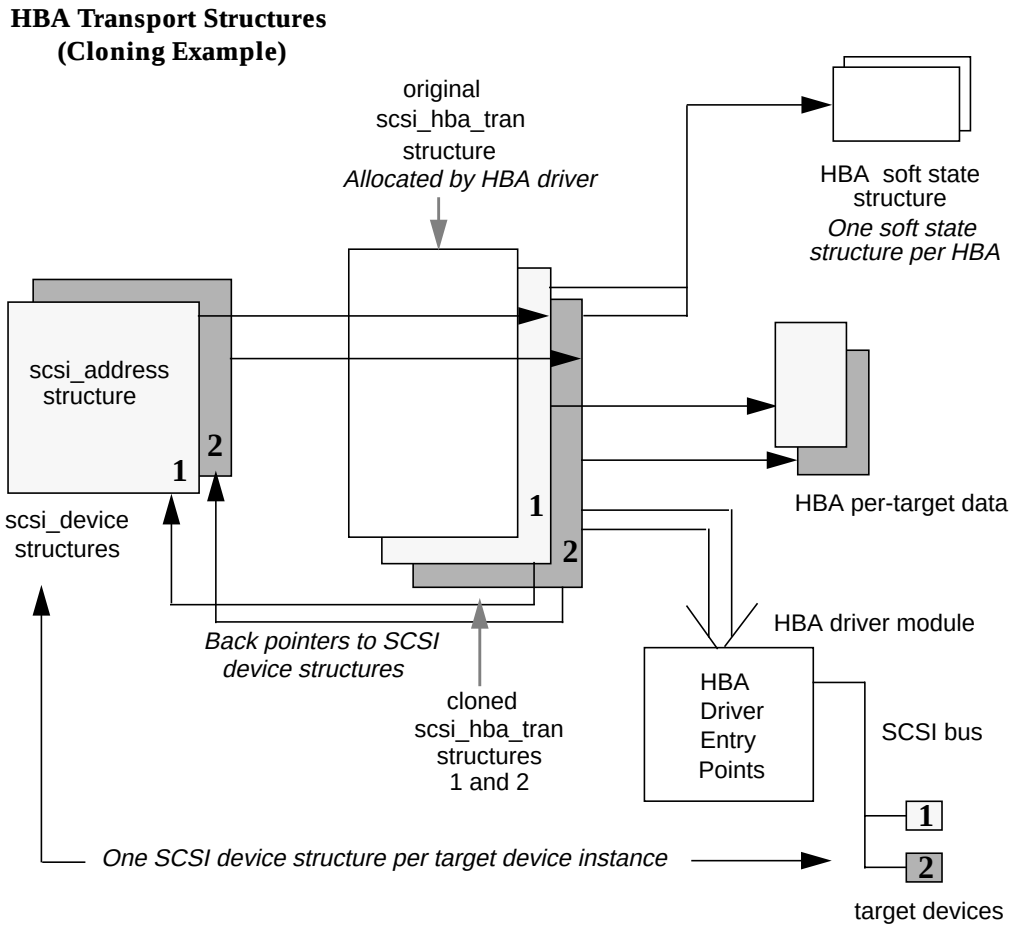


Figure 15–4 Cloning Transport Operation

SCSA HBA Functions

SCSA also provides a number of functions, listed in Table 15–2, intended for use by HBA drivers.

TABLE 15–2 SCSA HBA Functions

Function Name	Called by Driver Entry Point
`scsi_hba_init(9F)`	`_init(9E)`
`scsi_hba_fini(9F)`	`_fini(9E)`

TABLE 15–2 SCSA HBA Functions *(continued)*

Function Name	Called by Driver Entry Point
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) and tran_setcap(9E)

HBA Driver Dependency and Configuration Issues

In addition to incorporating SCSA HBA entry points, structures, and functions into a driver, HBA driver developers must also concern themselves with issues surrounding driver dependency and configuration. These issues are summarized in the following list:

- Configuration properties
- Dependency declarations
- State structure and per-command structure
- Module initialization entry points
- Autoconfiguration entry points

Declarations and Structures

HBA drivers must include the following header files:

```
#include <sys/scsi/scsi.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
```

To inform the system that the module depends on SCSA routines (see “SCSA HBA Interfaces” on page 234 for more information), the driver binary must be generated with the following command:

```
% ld -r xx.o -o xx -N"misc/scsi"
```

The code samples are derived from a simplified `isp` driver for the QLogic Intelligent SCSI Peripheral device. The `isp` driver supports WIDE SCSI, with up to 15 target devices and 8 logical units (LUNs) per target.

Per-Command Structure

An HBA driver will usually need to define a structure to maintain state for each command submitted by a target driver. The layout of this per-command structure is entirely up to the device driver writer and needs to reflect the capabilities and features of the hardware and the software algorithms used in the driver.

The following structure is an example of a per-command structure. The remaining code fragments of this chapter use this structure to illustrate the HBA interfaces.

```
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;
 };
```

Module Initialization Entry Points

Drivers for different types of devices have different sets of entry points, depending on the operations they perform. Some operations, however, are common to all

drivers, such as the as _init(9E), _info(9E), and _fini(9E) entry points for module initialization. Chapter 2 gives a complete description of these loadable module routines. This section describes only those entry points associated with operations performed by SCSI HBA drivers.

The following code for a SCSI HBA driver illustrates a representative dev_ops(9S) structure. The driver must initialize the devo_bus_ops field in this structure to NULL. A SCSI HBA driver can provide leaf driver interfaces for special purposes, in which case the devo_cb_ops field might point to a cb_ops(9S) structure. In this example, no leaf driver interfaces are exported, so the devo_cb_ops field is initialized to NULL.

_init(9E)

The _init(9E) function initializes a loadable module and is called before any other routine in the loadable module.

In a SCSI HBA, the _init(9E) function must call scsi_hba_init(9F) to inform the framework of the existence of the HBA driver before calling mod_install(9F). If scsi_hba_init(9F) returns a nonzero value, _init(9E) should return this value. Otherwise, _init(9E) must return the value returned by mod_install(9F).

The driver should initialize any required global state before calling mod_install(9F).

If mod_install(9F) fails, the _init(9E) function must free any global resources allocated and must call scsi_hba_fini(9F) before returning.

The following code sample uses a global mutex to show how to allocate data that is global to all instances of a driver. The code declares global mutex and soft-state structure information. The global mutex and soft state are initialized during _init(9E).

_fini(9E)

The _fini(9E) function is called when the system is about to try to unload the SCSI HBA driver. The _fini(9E) function must call mod_remove(9F) to determine if the driver can be unloaded. If mod_remove(9F) returns 0, the module can be unloaded, and the HBA driver must deallocate any global resources allocated in _init(9E) and must call scsi_hba_fini(9F).

_fini(9E) must return the value returned by mod_remove(9F).

Note - The HBA driver must not free any resources or call scsi_hba_fini(9F) unless mod_remove(9F) returns 0.

Code Example 15–1 shows SCSI HBA module intialization.

CODE EXAMPLE 15–1 SCSI HBA Module Initialization

```
static struct dev_ops isp_dev_ops = {
        DEVO_REV,           /* devo_rev */
        0,                  /* refcnt  */
        isp_getinfo,        /* getinfo */
        nulldev,            /* probe */
        isp_attach,         /* attach */
        isp_detach,         /* detach */
        nodev,              /* reset */
        NULL,               /* driver operations */
        NULL,               /* bus operations */
        isp_power,          /* power management */
};

/*
 * 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) {
                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);
}
```

Autoconfiguration Entry Points

Associated with each device driver is a `dev_ops`(9S) structure, which allows the kernel to locate the autoconfiguration entry points of the driver. A complete description of these autoconfiguration routines is given in Chapter 5. This section describes only those entry points associated with operations performed by SCSI HBA drivers. These include `attach`(9E) and `detach`(9E).

`attach(9E)`

The `attach`(9E) entry point for a SCSI HBA driver must perform a number of tasks to configure and attach an instance of the driver for the device. For a typical driver of real devices, the following operating system and hardware concerns must be addressed:

- Soft-state structure
- DMA
- Transport structure
- Attaching an HBA driver
- Register mapping
- Interrupt specification
- Interrupt handling
- Create power manageable components
- Report attachment status

Soft State Structure

The driver should allocate the per-device-instance soft state structure, being careful to clean up properly if an error occurs.

DMA

The HBA driver must describe the attributes of its DMA engine by properly initializing the `ddi_dma_attr_t` structure.

```
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 */
};
```

The driver, if providing DMA, should also check that its hardware is installed in a DMA-capable slot:

```
       if (ddi_slaveonly(dip) == DDI_SUCCESS) {
            return (DDI_FAILURE);
        }
```

Transport Structure

The driver should further allocate and initialize a transport structure for this instance. The `tran_hba_private` field is set to point to this instance's soft-state structure. `tran_tgt_probe` can be set to NULL to achieve the default behavior, if no special probe customization is needed.

```
    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
```

Attaching an HBA Driver

The driver should attach this instance of the device, and perform error cleanup if necessary.

```
    i = scsi_hba_attach_setup(dip, &isp_dma_attr, tran, 0);
    if (i != DDI_SUCCESS) {
         do error recovery
            return (DDI_FAILURE);
     }
```

Register Mapping

The driver should map in its device's registers, specifying the index of the register set, the data access characteristics of the device and the size of the register set to be mapped.

```
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);
     }
```

Adding an Interrupt Handler

The driver must first obtain the *iblock cookie* to initialize mutexes used in the driver handler. Only after those mutexes have been initialized can the interrupt handler be added.

```
    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);
     }
```

The driver should determine if a high-level interrupt handler is required. If a high-level handler is required and the driver is not coded to provide one, the driver must be rewritten to include either a high-level interrupt or fail the attach. See "Handling High-Level Interrupts" on page 93 for a description of high-level interrupt handling.

Create Power Manageable Components

If the host bus adapter hardware supports power management, and it is sufficient to have the host bus adapter powered down only when all of the target adapters are power manageable and are at power level 0, then the host bus adapter driver only needs to provide a `power(9E)` entry point as described in Chapter 9 and create a

pm-components(9) property that describes the components that the device implements.

Nothing more is necessary, since the components will default to idle, and the power management framework's default dependency processing will ensure that the host bus adapter will be powered up whenever an target adapter is powered up and will automatically power down the host bus adapter whenever all of the target adapters are powered down (provided that automatic power management is enabled).

Report Attachment Status

Finally, the driver should report that this instance of the device is attached and return success.

```
    ddi_report_dev(dip);
     return (DDI_SUCCESS);
```

detach(9E)

The driver should perform standard detach operations. See "detach(9E)" on page 76.

Code Example 15–2 provides an example of the isp_detach( ) function.

CODE EXAMPLE 15–2 isp_detach

```
static int
isp_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    switch (cmd) {
    case DDI_DETACH:
           /*
            * At present, detaching HBA drivers is not supported
            */
            return (DDI_FAILURE);

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

SCSA HBA Entry Points

For an HBA driver to work with target drivers using the SCSA interface, each HBA driver must supply a number of entry points, callable through the scsi_hba_tran(9S) structure.These entry points fall into five functional groups:

- Target driver instance initialization

- Resource allocation amd deallocation
- Command transport
- Capability management
- Abort and reset handling
- Dynamic reconfiguration

Table 15–3 lists the SCSA HBA entry points arranged by function groups.

TABLE 15–3 SCSA Entry Points

Function Groups	Entry Points Within Group	Description
Target Driver Instance Initialization	`tran_tgt_init(9E)`	Performs per-target initialization (optional)
	`tran_tgt_probe(9E)`	Probes SCSI bus for existence of a target (optional)
	`tran_tgt_free(9E)`	Performs per-target deallocation (optional)
Resource Allocation	`tran_init_pkt(9E)`	Allocates SCSI packet and DMA resources
	`tran_destroy_pkt(9E)`	Frees SCSI packet and DMA resources
	`tran_sync_pkt(9E)`	Synchronizes memory before and after DMA
	`tran_dmafree(9E)`	Frees DMA resources
Command Transport	`tran_start(9E)`	Transports a SCSI command
Capability Management	`tran_getcap(9E)`	Inquires about a capability's value
	`tran_setcap(9E)`	Sets a capability's value
Abort and Reset	`tran_abort(9E)`	Aborts one or all outstanding SCSI commands
	`tran_reset(9E)`	Resets a target device or the SCSI bus
	`tran_bus_reset(9E)`	Resets the SCSI bus

TABLE 15–3 SCSA Entry Points *(continued)*

Function Groups	Entry Points Within Group	Description
	`tran_reset_notify(9E)`	Request to notify target of bus reset (optional)
Dynamic Reconfiguration	`tran_quiesce(9E)`	Stops activity on the bus
	`tran_unquiesce(9E)`	Resume activity on the bus

Target Driver Instance Initialization

The following sections explain target entry points.

`tran_tgt_init(9E)`

The `tran_tgt_init`(9E) entry point allows the HBA to allocate and/or initialize any per-target resources. It also allows the HBA to qualify the device's address as valid and supportable for that particular HBA. By returning `DDI_FAILURE`, the instance of the target driver for that device will not be probed or attached.

This entry point is not required, and if none is supplied, the framework will attempt to probe and attach all possible instances of the appropriate target drivers.

```
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(9E)`

The `tran_tgt_probe`(9E) entry point enables the HBA to customize the operation of `scsi_probe`(9F), if necessary. This entry point is called only when the target driver calls `scsi_probe`(9F).

The HBA driver can retain the normal operation of `scsi_probe`(9F) by calling `scsi_hba_probe`(9F) and returning its return value.

This entry point is not required, and if not needed, the HBA driver should set the tran_tgt_probe vector in the scsi_hba_tran(9S) structure to point to scsi_hba_probe(9F).

scsi_probe(9F) allocates a scsi_inquiry(9S) structure and sets the sd_inq field of the scsi_device(9S) structure to point to the data in scsi_inquiry(9S). scsi_hba_probe(9F) handles this automatically. scsi_unprobe(9F) then frees the scsi_inquiry(9S) data.

Other than during the allocation of scsi_inquiry(9S) data, normally handled by scsi_hba_probe(9F), tran_tgt_probe(9E) must be stateless, as the same SCSI device might call it multiple times.

Note - The allocation of the scsi_inquiry(9S) structure is handled automatically by scsi_hba_probe(9F). This is only of concern if custom scsi_probe(9F) handling is what you want.

```
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(9E)

The tran_tgt_free(9E) entry point enables the HBA to perform any deallocation or clean-up procedures for an instance of a target. This entry point is optional.

```
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)
 }
```

Resource Allocation

The following sections discuss resource allocation.

tran_init_pkt(9E)

The tran_init_pkt(9E) entry point is the HBA driver function that allocates and initializes, on behalf of the target driver, a scsi_pkt(9S) structure and DMA resources for a target driver request.

The tran_init_pkt(9E) entry point is called when the target driver calls the SCSA function scsi_init_pkt(9F).

Each call of the tran_init_pkt(9E) entry point is a request to perform one or more of three possible services:

- Allocation and initialization of a scsi_pkt(9S) structure
- Allocation of DMA resources for data transfer
- Reallocation of DMA resources for the next portion of the data transfer

Allocation and Initialization of a scsi_pkt(9S) Structure

The tran_init_pkt(9E) entry point must allocate a scsi_pkt(9S) structure if pkt is NULL through scsi_hba_pkt_alloc(9F).

scsi_hba_pkt_alloc(9F) allocates the following:

- scsi_pkt(9S)
- SCSI CDB of length cmdlen
- SCSI status completion area of length statuslen
- Per-packet target driver private data area of length tgtlen
- Per-packet HBA driver private data area of length hbalen

The scsi_pkt(9S) structure members, as well as pkt itself, must be initialized to zero except for the following members: pkt_scbp (status completion), pkt_cdbp (CDB), pkt_ha_private (HBA driver private data), pkt_private (target driver private data). These members are pointers to memory space where the values of the fields are stored, as illustrated in Figure 15–5. For more information, refer to “scsi_pkt Structure” on page 240.

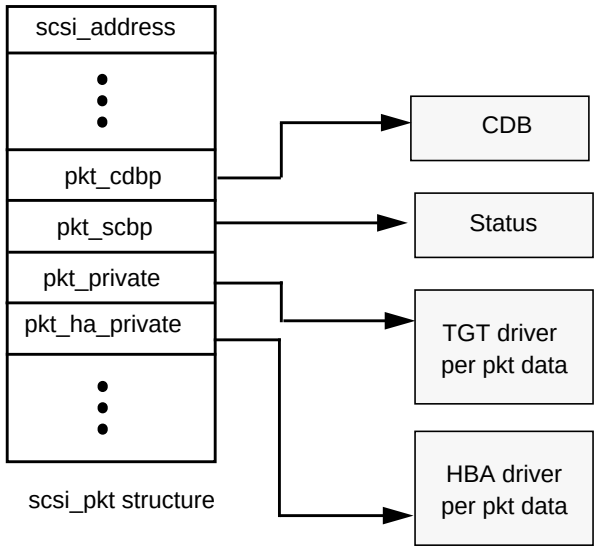


Figure 15–5 `scsi_pkt(9S)` Structure Pointers

Code Example 15–3 provides an example of allocation and initialization of a `scsi_pkt`(9S) structure.

CODE EXAMPLE 15–3 HBA Driver Initialization of a SCSI Packet Structure

```
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),
                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);
    }
```

Allocation of DMA Resources

If `bp` is not NULL and `bp->b_bcount` is not zero and DMA resources have not yet been allocated for this `scsi_pkt`(9S), the `tran_init_pkt`(9E) entry point must allocate DMA resources for a data transfer. The HBA driver needs to keep track of

whether DMA resources have been allocated for a particular command with a flag bit or a DMA handle in the per-packet HBA driver private data.

By setting the PKT_DMA_PARTIAL flag in the pkt, the target driver indicates it can accept breaking up the data transfer into multiple SCSI commands to accommodate the complete request. This might be necessary if the HBA hardware scatter-gather capabilities or system DMA resources are insufficient to accommodate the complete request in a single SCSI command.

If the PKT_DMA_PARTIAL flag is set, the HBA driver can set the DDI_DMA_PARTIAL flag when allocating DMA resources (using, for example, ddi_dma_buf_bind_handle(9F)) for this SCSI command. The DMA attributes used when allocating the DMA resources should accurately describe any constraints placed on the ability of the HBA hardware to perform DMA. If the system can only allocate DMA resources for part of the request, ddi_dma_buf_bind_handle(9F) will return DDI_DMA_PARTIAL_MAP.

The tran_init_pkt(9E) entry point must return the amount of DMA resources not allocated for this transfer in the field pkt_resid.

A target driver can make one request to tran_init_pkt(9E) to simultaneously allocate both a scsi_pkt(9S) structure and DMA resources for that pkt. In this case, if the HBA driver is unable to allocate DMA resources, it must free the allocated scsi_pkt(9S) before returning. The scsi_pkt(9S) must be freed by calling scsi_hba_pkt_free(9F).

The target driver might first allocate the scsi_pkt(9S) and allocate DMA resources for this pkt at a later time. In this case, if the HBA driver is unable to allocate DMA resources, it must *not* free pkt. The target driver in this case is responsible for freeing the pkt.

CODE EXAMPLE 15–4 HBA Driver Allocation of DMA Resources

```
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;
    } 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)
                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);
}
```

Reallocation of DMA Resources for Next Portion of Data Transfer

For a previously allocated packet with data remaining to be transferred, the `tran_init_pkt`(9E) entry point must reallocate DMA resources when the following conditions apply:

- Partial DMA resources have already been allocated.
- A non-zero `pkt_resid` was returned in the previous call to `tran_init_pkt`(9E).
- `bp` is not NULL.
- `bp->b_bcount` is not zero.

When reallocating DMA resources to the next portion of the transfer, `tran_init_pkt`(9E) must return the amount of DMA resources not allocated for this transfer in the field `pkt_resid`.

If an error occurs while attempting to move DMA resources, `tran_init_pkt`(9E) must not free the `scsi_pkt`(9S). The target driver in this case is responsible for freeing the pkt.

If the callback parameter is `NULL_FUNC`, the `tran_init_pkt`(9E) entry point must not sleep or call any function that might sleep. If the callback parameter is `SLEEP_FUNC` and resources are not immediately available, the `tran_init_pkt`(9E) entry point should sleep until resources are available, unless the request is impossible to satisfy.

CODE EXAMPLE 15–5 HBA Driver DMA Resource Reallocation

```
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;
    return (1);
}
```

tran_destroy_pkt(9E)

The `tran_destroy_pkt`(9E) entry point is the HBA driver function that deallocates `scsi_pkt`(9S) structures. The `tran_destroy_pkt`(9E) entry point is called when the target driver calls `scsi_destroy_pkt`(9F).

The `tran_destroy_pkt`(9E) entry point must free any DMA resources allocated for the packet. Freeing the DMA resources causes an implicit DMA synchronization if any cached data remained after the completion of the transfer. The `tran_destroy_pkt`(9E) entry point frees the SCSI packet itself by calling `scsi_hba_pkt_free`(9F).

CODE EXAMPLE 15–6 HBA Driver `tran_destroy_pkt(9E)` Entry Point

```
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(9E)

The tran_sync_pkt(9E) entry point is the HBA driver function that synchronizes the DMA object allocated for the scsi_pkt(9S) structure before or after a DMA transfer. The tran_sync_pkt(9E) entry point is called when the target driver calls scsi_sync_pkt(9F).

If the data transfer direction is a DMA read from device to memory, tran_sync_pkt(9E) must synchronize the CPU's view of the data. If the data transfer direction is a DMA write from memory to device, tran_sync_pkt(9E) must synchronize the device's view of the data.

CODE EXAMPLE 15–7 HBA Driver tran_sync_pkt(9E) Entry Point

```
static void
isp_scsi_sync_pkt(
    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) {
           (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(9E)

The tran_dmafree(9E) entry point is the HBA driver function that deallocates DMA resources allocated for a scsi_pkt(9S) structure. The tran_dmafree(9E) entry point is called when the target driver calls scsi_dmafree(9F).

tran_dmafree(9E) must free only DMA resources allocated for a scsi_pkt(9S) structure, not the scsi_pkt(9S) itself. Freeing the DMA resources implicitly performs a DMA synchronization.

Note - The scsi_pkt(9S) will be freed in a separate request to tran_destroy_pkt(9E). Because tran_destroy_pkt(9E) must also free DMA resources, it is important that the HBA driver keep accurate note of whether scsi_pkt(9S) structures have DMA resources allocated.

CODE EXAMPLE 15–8 HBA Driver tran_dmafree(9E) Entry Point

```
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;
    }
}
```

Command Transport

As part of command transport, the HBA driver accepts a command from the target driver, issues the command to the device hardware, services any interrupts that occur, and manages timeouts.

tran_start(9E)

The tran_start(9E) entry point for a SCSI HBA driver is called to transport a SCSI command to the addressed target. The SCSI command is described entirely within the scsi_pkt(9S) structure, which the target driver allocated through the HBA driver's tran_init_pkt(9E) entry point. If the command involves a data transfer, DMA resources must also have been allocated for the scsi_pkt(9S) structure.

The tran_start(9E) entry point is called when a target driver calls scsi_transport(9F).

tran_start(9E) should perform basic error checking along with whatever initialization the command requires. If the flag FLAG_NOINTR is not set in the pkt_flags field of the scsi_packet(9S) structure, tran_start(9E) must queue the command for execution on the hardware and return immediately. Upon completion of the command, the HBA driver should call the pkt completion routine.

For commands with the FLAG_NOINTR bit set in the pkt_flags field of the scsi_pkt(9S) structure, tran_start(9E) should not return until the command has been completed, and the HBA driver should not call the *pkt* completion routine.

Code Example 15–9 demonstrates how to handle the tran_start(9E) entry point. The ISP hardware provides a queue per-target device. For devices that can manage only one active outstanding command, the driver itself is typically required to manage a per-target queue and starts up a new command upon completion of the current command in a round-robin fashion.

CODE EXAMPLE 15–9 HBA Driver tran_start(9E) Entry Point

```
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;
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 dma transfers data segments.
 */
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) {
        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);
}
```

Interrupt Handler and Command Completion

The interrupt handler must check the status of the device to be sure the device is generating the interrupt in question. It must also check for any errors that have occurred and service any interrupts generated by the device.

If data was transferred, the hardware should be checked to determine how much data was actually transferred, and the `pkt_resid` field in the `scsi_pkt`(9S) structure should be set to the residual of the transfer.

For commands marked with the `PKT_CONSISTENT` flag when DMA resources were allocated through `tran_init_pkt`(9E), the HBA driver must ensure that the data transfer for the command is correctly synchronized before the target driver's command completion callback is performed.

Once a command has completed, there are two requirements:

- Start a new command (if one is queued up) on the hardware as quickly as possible.
- Call the command completion callback as set up in the `scsi_pkt`(9S) structure by the target driver to notify the target driver that the command is now complete.

It is important to start a new command on the hardware, if possible, before calling the PKT_COMP command completion callback. The command completion handling can take considerable time, as the target driver will typically call functions such as biodone(9F) and possibly scsi_transport(9F) to begin a new command.

The interrupt handler must return DDI_INTR_CLAIMED if this interrupt is claimed by this driver; otherwise, the handler returns DDI_INTR_UNCLAIMED.

Code Example 15–10 shows an interrupt handler for the SCSI HBA isp driver. The caddr_t argument is the parameter set up when the interrupt handler was added in attach(9E) and is typically a pointer to the state structure allocated per instance.

CODE EXAMPLE 15–10 HBA Driver Interrupt Handler

```
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;
    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);
            }
        } 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,
         * we need to 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.
         */
        if (((pkt->pkt_flags & FLAG_NOINTR) == 0) &&
            pkt->pkt_comp) {
            (*pkt->pkt_comp)(pkt);
        }
    }
}
```

Timeout Handler

The HBA driver should be prepared to time out the command if it is not complete within a specified time unless a zero timeout was specified in the `scsi_pkt`(9S) structure.

When a command times out, the HBA driver should mark the `scsi_pkt`(9S) with `pkt_reason` set to `CMD_TIMEOUT` and `pkt_statistics` OR'd with `STAT_TIMEOUT`. The HBA driver should also attempt to recover the target and/or bus and, if this recovery can be performed successfully, mark the `scsi_pkt`(9S) with `pkt_statistics` OR'd with either `STAT_BUS_RESET` or `STAT_DEV_RESET`.

Once the command has timed out and the target and bus recovery attempt has completed, the HBA driver should call the command completion callback.

Note - If recovery was unsuccessful or not attempted, the target driver might attempt to recover from the timeout by calling `scsi_reset`(9F).

The ISP hardware manages command timeout directly and returns timed-out commands with the necessary status, so the `isp` sample driver timeout handler checks active commands for timeout state only once every 60 seconds.

The `isp` sample driver uses the `timeout`(9F) facility to arrange for the kernel to call the timeout handler every 60 seconds. The `caddr_t` argument is the parameter set up when the timeout is initialized at `attach`(9E) time. In this case, the `caddr_t` argument is a pointer to the state structure allocated per driver instance.

If the driver discovers timed-out commands that have not been returned as timed-out by the ISP hardware, the hardware is not functioning correctly and needs to be reset.

Capability Management

The following sections discuss capability management.

`tran_getcap(9E)`

The `tran_getcap`(9E) entry point for a SCSI HBA driver is called when a target driver calls `scsi_ifgetcap`(9F) to determine the current value of one of a set of SCSA-defined capabilities.

The target driver can request the current setting of the capability for a particular target by setting the `whom` parameter to nonzero. A `whom` value of 0 means the request is for the current setting of the capability for the SCSI bus or for adapter hardware in general.

`tran_getcap`(9E) should return `-1` for undefined capabilities or the current value of the requested capability.

The HBA driver can use the function `scsi_hba_lookup_capstr`(9F) to compare the capability string against the canonical set of defined capabilities.

CODE EXAMPLE 15–11 HBA Driver `tran_getcap(9E)` Entry Point

```
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) {
        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(9E)

The `tran_setcap`(9E) entry point for a SCSI HBA driver is called when a target driver calls `scsi_ifsetcap`(9F) to change the current one of a set of SCSA-defined capabilities.

The target driver might request that the new value be set for a particular target by setting the `whom` parameter to nonzero. A `whom` value of 0 means the request is to set the new value for the SCSI bus or for adapter hardware in general.

`tran_setcap`(9E) should return `-1` for undefined capabilities, `0` if the HBA driver cannot set the capability to the requested value, or `1` if the HBA driver is able to set the capability to the requested value.

The HBA driver can use the function `scsi_hba_lookup_capstr`(9F) to compare the capability string against the canonical set of defined capabilities.

CODE EXAMPLE 15–12 HBA Driver `tran_setcap(9E)` Entry Point

```
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 via
     * 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;
        }
    }
    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);
                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);
    }
```

Abort and Reset Management

The following sections discuss the abort and reset entry points for SCSI HBA.

tran_abort(9E)

The tran_abort(9E) entry point for a SCSI HBA driver is called to abort one or all the commands currently in transport for a particular target. This entry point is called when a target driver calls scsi_abort(9F).

The tran_abort(9E) entry point should attempt to abort the command denoted by the *pkt* parameter. If the *pkt* parameter is NULL, tran_abort(9E) should attempt to abort all outstanding commands in the transport layer for the particular target or logical unit.

Each command successfully aborted must be marked with pkt_reason CMD_ABORTED and pkt_statistics OR'd with STAT_ABORTED.

tran_reset(9E)

The tran_reset(9E) entry point for a SCSI HBA driver is called to reset either the SCSI bus or a particular SCSI target device. This entry point is called when a target driver calls scsi_reset(9F).

The tran_reset(9E) entry point must reset the SCSI bus if level is RESET_ALL. If level is RESET_TARGET, just the particular target or logical unit must be reset.

Active commands affected by the reset must be marked with pkt_reason CMD_RESET, and with pkt_statistics OR'd with either STAT_BUS_RESET or STAT_DEV_RESET, depending on the type of reset.

Commands in the transport layer, but not yet active on the target, must be marked with pkt_reason CMD_RESET, and pkt_statistics OR'd with STAT_ABORTED.

tran_bus_reset(9E)

tran_bus_reset(9E) must reset the SCSI bus without resetting targets.

```
#include <sys/scsi/scsi.h>

int tran_bus_reset(dev_info_t *hba_dip, int level);
```

Where level must be the following:

RESET_BUS	Reset the SCSI bus only, not the targets

The tran_bus_reset( ) vector in the scsi_hba_tran(9S) structure should be initialized during the HBA driver's attach(9E) to point to an HBA entry point to be called when a user initiates a bus reset.

Implementation is hardware specific. If it is not possible to reset the SCSI bus without affecting the targets, the HBA driver should fail RESET_BUS or not initialize this vector.

tran_reset_notify(9E)

The tran_reset_notify(9E) entry point for a SCSI HBA driver is called to request that the HBA driver notify the target driver by callback when a SCSI bus reset occurs.

CODE EXAMPLE 15–13 HBA Driver tran_reset_notify(9E) Entry Point

```
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) {
            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);
}
```

Dynamic Reconfiguration

To support the minimal set of hot-plugging operations, drivers might need to implement support for bus quiesce, bus unquiesce, and reset. The `scsi_hba_tran`(9S) structure has been extended to support these new operations. If quiesce / unquiesce / reset is not required by hardware, no driver changes are needed.

The following new fields have been added to the `scsi_hba_tran` structure:

```
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);
```

The new driver entry points are introduced in the following sections.

`tran_quiesce(9E)` and `tran_unquiesce(9E)`

Quiesce and unquiesce a SCSI bus.

```
#include <sys/scsi/scsi.h>

int prefixtran_quiesce(dev_info_t *hba_dip);

int prefixtran_unquiesce(dev_info_t *hba_dip);
```

`tran_quiesce`(9E) and `tran_unquiesce`(9E) are required to be implemented by an HBA driver to support dynamic reconfiguration (DR) of SCSI devices on buses that were not designed to support hot-plugging.

The `tran_quiesce( )` and `tran_unquiesce( )` vectors in the `scsi_hba_tran`(9S) structure should be initialized during the HBA driver's `attach`(9E) to point to HBA entry points so they are called when a user initiates quiesce and unquiesce operations.

`tran_quiesce`(9E) is called by the SCSA framework to stop all activity on a SCSI bus prior to and during the reconfiguration of devices attached to the SCSI bus. `tran_unquiesce`(9E) is called by the SCSA framework to resume activity on the SCSI bus after the reconfiguration operation has been completed.

HBA drivers are required to handle `tran_quiesce`(9E) by waiting for all outstanding commands to complete before returning success. After the HBA has quiesced the bus, it must queue any new I/O requests from target drivers until the SCSA framework calls the corresponding `tran_unquiesce`(9E) entrypoint.

HBA drivers handle calls to `tran_unquiesce`(9E) by starting any target driver I/O requests that were queued by the HBA during the time the bus was quiesced.

SCSI HBA Driver Specific Issues

The section covers issues specific to SCSI HBA drivers.

Installing HBA Drivers

A SCSI HBA driver is installed like a leaf driver (see Chapter 16), except that the `add_drv(1M)` command must specify the driver class as SCSI, such as:

```
# add_drv -m" * 0666 root root" -i'"pci1077,1020"' -c scsi isp
```

HBA Configuration Properties

When attaching an instance of an HBA device, `scsi_hba_attach_setup`(9F) creates a number of SCSI configuration parameter properties for that HBA instance. A particular property is created only if there is no existing property of the same name already attached to the HBA instance, permitting a default property value to be overridden in an HBA configuration file.

An HBA driver must use `ddi_prop_get_int`(9F) to retrieve each property. The HBA driver then modifies (or accepts the default value of) the properties to configure its specific operation.

`scsi-reset-delay` Property

The `scsi-reset-delay` property is an integer specifying the SCSI bus or device reset delay recovery time in milliseconds.

`scsi-options` Property

The `scsi-options` property is an integer specifying a number of options through individually defined bits. The bits in `scsi_options` are:

- SCSI_OPTIONS_DR (0x008) – If not set, the HBA should not grant disconnect privileges to a target device.
- SCSI_OPTIONS_LINK (0x010) – If not set, the HBA should not enable linked commands.
- SCSI_OPTIONS_SYNC (0x020) – If not set, the HBA should not negotiate synchronous data transfer, and should reject any attempt to negotiate synchronous data transfer initiated by a target.
- SCSI_OPTIONS_PARITY (0x040) – If not set, the HBA should run the SCSI bus without parity.
- SCSI_OPTIONS_TAG (0x080) – If not set, the HBA should not operate in Command Tagged Queuing mode.
- SCSI_OPTIONS_FAST (0x100) – If not set, the HBA should not operate the bus in FAST SCSI mode.
- SCSI_OPTIONS_WIDE (0x200) – If not set, the HBA should not operate the bus in WIDE SCSI mode.

Per-target scsi-options

An HBA driver might support a per-target scsi-options feature in the following format:

```
target<n>-scsi-options=<hex value>
```

In this example, < *n*> is the target ID. If the per-target scsi-options property is defined for a particular target, the HBA driver uses the value of the per-target scsi-options property for that target rather than the per-HBA driver instance scsi-options property. This can provide more precise control if, for example, synchronous data transfer needs to be disabled for just one particular target device. The per-target scsi-options property can be defined in the driver.conf(4) file.

Here is an example of a per-target scsi-options property definition to disable synchronous data transfer for target device 3:

```
target3-scsi-options=0x2d8
```

IA Target Driver Configuration Properties

Some IA SCSI target drivers (such as the *cmdk* disk target driver) use the following configuration properties:

- disk
- queue
- flow_control

When using the *cmdk* sample driver to write an HBA driver for an IA platform, one or more of these properties (as appropriate to the HBA driver and hardware) need to be defined in the `driver.conf`(4) file.

Note - These property definitions should appear only in an HBA driver's `driver.conf`(4) file. The HBA driver itself should not inspect or attempt to interpret these properties in any way. These properties are advisory only and serve as an adjunct to the cmdk driver. They should not be relied upon in any way. The property definitions might or might not be used in future releases.

The `disk` property can be used to define the type of disk supported by cmdk. For a SCSI HBA, the only possible value for the `disk` property is:

- `disk="scdk"` – Disk type is a SCSI disk.

The `queue` property defines how the disk driver sorts the queue of incoming requests during `strategy`(9E). There are two possible values:

- `queue="qsort"` – One-way elevator queuing model, provided by `disksort`(9F).
- `queue="qfifo"` – FIFO (first in, first out) queuing model

The `flow_control` property defines how commands are transported to the HBA driver. There are three possible values:

- `flow_control="dsngl"` – Single command per HBA driver
- `flow_control="dmult"` – Multiple commands per HBA driver—when the HBA queue is full, the driver returns `TRAN_BUSY`.
- `flow_control="duplx"` – The HBA can support separate read and write queues, with multiple commands per queue. FIFO ordering is used for the write queue; the queuing model used for the read queue is described by the *queue* property. When an HBA queue is full, the driver returns `TRAN_BUSY`.

Here is an example of a `driver.conf`(4) file for use with an IA HBA PCI device designed for use with the *cmdk* sample driver:

```
#
# config file for ISP 1020 SCSI HBA driver
#
       flow_control="dsngl" queue="qsort" disk="scdk"
       scsi-initiator-id=7;
```

Support for Queuing

The following sections discuss queuing.

Tagged Queuing

For a definition of tagged queuing, refer to the SCSI-2 specification. To support tagged queuing, first check the *scsi_options* flag `SCSI_OPTIONS_TAG` to see if tagged queuing is enabled globally. Next, check to see if the target is a SCSI-2 device and whether it has tagged queuing enabled. If this is all true, attempt to enable tagged queuing by using `scsi_ifsetcap`(9F).

Untagged Queuing

If tagged queuing fails, you can attempt to set untagged queuing. In this mode, you submit as many commands as you think necessary or optimal to the host adapter driver. Then the host adapter queues the commands to the target one at a time (as opposed to tagged queuing, where the host adapter submits as many commands as it can until the target indicates that the queue is full).

CHAPTER 16

Compiling, Loading, Packaging, and Testing Drivers

This chapter describes the procedure for driver development, including code layout, compilation, packaging, and testing.

Driver Code Layout Structure

The code for a device driver is usually divided into the following files:

- Header files (`.h` files)
- Source files (`.c` files)
- Optional configuration file (`driver.conf` file)

Header Files

Header files define:

- Data structures specific to the device, such as a structure representing the device registers
- Data structures defined by the driver for maintaining state information
- Defined constants, such as those representing the bits of the device registers
- Macros, such as those defining the static mapping between the minor device number and the instance number

Some of this information, such as the state structure, might be needed only by the device driver. This information should go in *private* header files that are only included by the device driver itself.

Any information that an application might require, such as the I/O control commands, should be in *public* header files. These are included by the driver and any applications that need information about the device.

While there is no standard for naming private and public files, one convention is to name the private header file `xximpl.h` and the public header file `xxio.h`.

`.c` Files

A `.c` file for a device driver contains the data declarations and the code for the entry points of the driver. It contains the `#include` statements the driver needs, declares extern references, declares local data, sets up the `cb_ops` and `dev_ops` structures, declares and initializes the module configuration section, makes any other necessary declarations, and defines the driver entry points.

`driver.conf` Files

`driver.conf` files are required for devices that are not self-identifying. Entries in the `driver.conf` file specify possible device instances the driver will probe for existence. See `scsi`(4) and `driver.conf`(4). Driver global properties can also be set by entries in the `driver.conf`(4) file. `driver.conf` files are optional for SID devices, where the entries can be used to add properties into self-identifying device nodes. See `sbus(4)` and `pci`(4).

Preparing for Installation

Before the driver is actually installed, it must be compiled into a binary, and a configuation file created, if necessary. The driver's module name must either match the name of the device nodes, or the system must be informed that this driver should manage other names.

Module Naming

The system maintains a one-to-one association between the name of the driver module and the name of the `dev_info` node. For example, a `dev_info` node for a

device named *wombat* is handled by a driver module called `wombat` in a subdirectory called `drv` (resulting in `drv/wombat`) found in the module path.

If the driver should manage `dev_info` nodes with different names, the `add_drv`(1M) utility can create aliases. The `-i` flag specifies the names of other `dev_info` nodes that the driver handles.

Compiling and Linking the Driver

Compile each driver source file and link the resulting object files into a driver module. The example below shows a driver called *xx* that has two C-language source files and generates a driver module *xx*. This driver created in this example is intended for the 32–bit kernel:

```
% cc -D_KERNEL -c xx1.c
% cc -D_KERNEL -c xx2.c
% ld -r -o xx xx1.o xx2.o
```

The `_KERNEL` symbol must be defined while compiling kernel (driver) code. No other symbols (such as `sun4m`) should be defined, aside from driver private symbols. `DEBUG` can also be defined to enable any calls to `assert`(9F). There is no need to use the `-I` flag for the standard headers.

Drivers intended for the 64-bit SPARC kernel should specify the –`xarch=v9` option. Use the following compile line:

```
% cc -D_KERNEL -xarch=v9 -c xx1.c
```

After the driver is stable, optimization flags can be used to build a production quality driver. For the Sun WorkShop Compilers C, the normal `-O` flag, or its equivalent `-xO3`, can be used. Note that `-xO3` is the highest level of optimization device drivers should use (see `cc(1)`).

The following compile line was used to create 64–bit SPARC drivers provided with the Solaris 8 operating environment:

```
% cc -D_KERNEL -xcg92 -xarch=v9 -xcode=abs32 -xO3 -c xx1.c
```

Where –`xcg92` refers to the code generator, and the use of –`xcode=abs32` leads to more compact code.

Note - Running `ld` –`r` is necessary even if there is only one object module.

Module Dependencies

If the driver module depends on symbols exported by another kernel module, the dependency can be specified by the –`N` option of `ld`. If the driver depends on a

symbol exported by `misc/foo`, the example below should be used to create the driver binary. See also `ld`(1).

```
% ld -r -o xx xx1.o xx2.o -N misc/foo
```

Writing a Hardware Configuration File

If the device is non-self-identifying, the kernel requires a hardware configuration file for it. If the driver is called *xx*, the hardware configuration file for it should be called *xx*`.conf`. See `driver.conf`(4), `pseudo`(4), `sbus`(4), and `scsi`(4) for more information on hardware configuration files. On the Intel platform, device information is now supplied by the booting system. Hardware configuration files should no longer be needed, even for non-self-identifying devices.

Arbitrary properties can be defined in hardware configuration files by adding entries of the form *property=value*, where *property* is the property name, and *value* is its initial value. This allows devices to be configured by changing the property values.

Installing and Removing Drivers

Before a driver can be used, the system must be informed that it exists. The `add_drv`(1M) utility *must* be used to correctly install the device driver. After the driver is installed, it can be loaded and unloaded from memory without using `add_drv`(1M) again.

Copying the Driver to a Module Directory

Device drivers reside in different directories depending on the platform they run on and whether they are needed at boot time. Platform-dependent device drivers reside in the following locations:

- `/platform/‘uname -i‘/kernel/drv` – Contains 32–bit drivers that run only on a specific platform, such as the Ultra™ 2
- `/platform/‘uname -m‘/kernel/drv` – Contains 32–bit drivers that run on a family of platforms. This directory might not be present on some platforms.

Platform-independent drivers reside in either of these directories:

- `/usr/kernel/drv` – Contains 32–bit drivers not required for system booting
- `/kernel/drv` – Contains 32–bit drivers required for booting

- 64–bit SPARC drivers reside in the `drv/sparcv9` directory in the module path

To install a 32–bit driver, the driver and its configuration file must be copied to a `drv` directory in the module path. For example, to copy a driver to `/usr/kernel/drv`, type:

```
$ su
# cp xx /usr/kernel/drv
# cp xx.conf /usr/kernel/drv
```

To install a 64-bit SPARC driver, copy the driver to a `drv/sparcv9` directory in the module path. Copy the driver configuration file to the `drv` directory in the module path. For example, to copy a driver to `/usr/kernel/drv`, type:

```
$ su
# cp xx /usr/kernel/drv/sparcv9
# cp xx.conf /usr/kernel/drv
```

Optionally Editing `/etc/devlink.tab`

If the driver creates minor nodes that do not represent disks, tapes, or ports (terminal devices), `/etc/devlink.tab` can be modified to cause `devfsadm`(1M) to create logical device names in `/dev`.

Alternatively, logical names can be created by a program run at driver installation time.

Running `add_drv`(1M)

Run `add_drv`(1M) to install the driver in the system. If the driver installs successfully, `add_drv(1M)` will run `devfsadm`(1M) to create the logical names in `/dev`.

```
# add_drv xx
```

This is a simple case in which the device identifies itself as *xx* and the device special files will have default ownership and permissions (`0600 root sys`). `add_drv`(1M) also allows additional names for the device (aliases) to be specified. See `add_drv`(1M) to determine how to add aliases and set file permissions explicitly.

Note - `add_drv`(1M) should not be run when installing a STREAMS module. See the *STREAMS Programming Guide* for details.

Removing the Driver

To remove a driver from the system, use `rem_drv`(1M), then delete the driver module and configuration file from the module path. The driver cannot be used again until it is reinstalled with `add_drv`(1M). Removing a SCSI HBA driver will require a reboot to take effect.

Loading Drivers

Opening a special file associated with the device driver causes the driver to be loaded. `modload`(1M) can also be used to load the driver into memory, but it does not call any routines in the module. Opening the device is the preferred method.

Unloading Drivers

Normally, the system automatically unloads device drivers when they are no longer in use. During development, it might be necessary to use `modunload`(1M) to unload the driver explicitly. In order for `modunload`(1M) to be successful, the device driver must not be active; there must be no outstanding references to the device, such as through `open`(2) or `mmap`(2).

Use `modunload`(1M) to unload a driver from the system. `modunload` takes a `module_id` that is run time dependent as an argument. It can be found by grepping the output of `modinfo` for the driver name in question and looking at the first column.

```
# modunload -i module_id
```

To unload all currently unloadable modules, specify module ID zero:

```
# modunload -i 0
```

In addition to being inactive, the driver must have working `detach`(9E) and `_fini`(9E) routines for `modunload`(1M) to succeed.

Driver Packaging

The normal delivery vehicle for unbundled software is to create a package with all the components of the software packaged together. A package provides a controlled mechanism for installation and removal of all the components of a software product, including applications, configuration tools, drivers, man pages, and other documentation.

Package Postinstall

Packages that include a driver must run `add_drv`(1M) to complete the installation of the driver, after installing the driver binary and other components, onto a system. Here is an example package `postinstall` script to run `add_drv`(1M) to complete the driver installation.

```
#!/bin/sh
#
#       @(#)postinstall 1.1

PATH="/usr/bin:/usr/sbin:${PATH}"
export PATH

#
# Driver info
#
DRV=<driver-name>
DRVALIAS="<company-name>,<driver-name>"
DRVPERM='* 0666 root sys'

ADD_DRV=/usr/sbin/add_drv

#
# Select the correct add_drv options to execute.
# add_drv touches /reconfigure to cause the
# next boot to be a reconfigure boot.
#
if [ "${BASEDIR}" = "/" ]; then
        #
        # On a running system, modify the
        # system files and attach the driver
        #
        ADD_DRV_FLAGS=""
else
        #
        # On a client, modify the system files
        # relative to BASEDIR
        #
        ADD_DRV_FLAGS="-b ${BASEDIR}"
fi

#
```

```
# Make sure add_drv has not been previously executed
# before attempting to add the driver.
#
grep "^${DRV} " $BASEDIR/etc/name_to_major > /dev/null 2>&1
if [ $? -ne 0 ]; then
        ${ADD_DRV} ${ADD_DRV_FLAGS} -m "${DRVPERM}" -i "${DRVALIAS}" ${DRV}
        if [ $? -ne 0 ]; then
                echo "postinstall: add_drv $DRV failed\n" >&2
                exit 1
        fi
fi
exit 0
```

Package Preremove

When removing a package, a package that includes a driver must run `rem_drv`(1M) to complete the removal of the driver, before removing the driver binary and other components from a system. Here is an example package `preremove` script to run `rem_drv`(1M) to complete the driver removal.

```
#!/bin/sh
#
#       @(#)preremove  1.1

PATH="/usr/bin:/usr/sbin:${PATH}"
export PATH

#
# Driver info
#
DRV=<driver-name>
REM_DRV=/usr/sbin/rem_drv

#
# Select the correct rem_drv options to execute.
# rem_drv touches /reconfigure to cause the
# next boot to be a reconfigure boot.
#
if [ "${BASEDIR}" = "/" ]; then
        #
        # On a running system, modify the
        # system files and remove the driver
        #
        REM_DRV_FLAGS=""
else
        #
        # On a client, modify the system files
        # relative to BASEDIR
        #
        REM_DRV_FLAGS="-b ${BASEDIR}"
fi

${REM_DRV} ${REM_DRV_FLAGS} ${DRV}

exit 0
```

Testing

Once a device driver is functional, it should be thoroughly tested before it is distributed. In addition to testing traditional UNIX device driver features, Solaris 8 drivers require testing of power management features, such as dynamic loading and unloading of drivers.

Configuration Testing

A driver's ability to handle multiple device configurations is an important part of the test process. Once the driver is working on a simple, or default, configuration, additional configurations should be tested. Depending upon the device, this can be accomplished by changing jumpers or DIP switches. If the number of possible configurations is small, all of them should be tried. If the number is large, various classes of possible configurations should be defined, and a sampling of configurations from each class should be tested. The designation of such classes depends on how the different configuration parameters might interact, which in turn depends on the device and on how the driver was written.

For each device configuration, the basic functions must be tested, which include loading, opening, reading, writing, closing, and unloading the driver. Any function that depends upon the configuration deserves special attention. For example, changing the base memory address of device registers is not likely to affect the behavior of most driver functions; if the driver works well with one address, it is likely to work as well with a different address, provided the configuration code enables it to work at all. On the other hand, a special I/O control call might have different effects depending upon the particular device configuration.

Loading the driver with varying configurations ensures that the `probe`(9E) and `attach`(9E) entry points can find the device at different addresses. For basic functional testing, using regular UNIX commands such as `cat`(1) or `dd`(1M) is usually sufficient for character devices. Mounting or booting might be required for block devices.

Functionality Testing

After a driver has been completely tested for configuration, all of its functionality should be thoroughly tested. This requires exercising the operation of all the driver's entry points.

Many drivers will require custom applications to test functionality, but basic drivers for devices such as disks, tapes, or asynchronous boards can be tested using standard system utilities. All entry points should be tested in this process, including

devmap(9E), chpoll(9E), and ioctl(9E), if applicable. The ioctl(9E) tests might be quite different for each driver, and for nonstandard devices, a custom testing application will be required.

Error Handling

A driver might perform correctly in an ideal environment, but fail to handle cases where a device encounters an error or an application specifies erroneous operations or sends bad data to the driver. Therefore, an important part of driver testing is the testing of its error handling.

All possible error conditions of a driver should be exercised, including error conditions for actual hardware malfunctions. Some hardware error conditions might be difficult to induce, but an effort should be made to cause them or to simulate them if possible. All of these conditions could be encountered in the field. Cables should be removed or loosened, boards should be removed, and erroneous user application code should be written to test those error paths.

Caution - Be sure to take proper electrical precautions when testing.

Testing Loading and Unloading

Because a driver that will not load or unload can force unscheduled downtime, it is important that loading and unloading be thoroughly tested.

A script like the following should suffice:

```
#!/bin/sh
cd <location_of_driver>
while [ 1 ]
do
        modunload -i 'modinfo | grep " <driver_name> " | cut -cl-3' &
        modload <driver_name> &
done
```

Stress, Performance, and Interoperability Testing

To help ensure that the driver performs well, it should be subjected to vigorous stress testing. Running single threads through a driver will not test any of the locking logic and might not test condition variable waits. Device operations should be performed by multiple processes at once to cause several threads to execute the same code simultaneously. The way to do this depends upon the driver; some drivers will

require special testing applications, but starting several UNIX commands in the background will be suitable for others. It depends upon where the particular driver uses locks and condition variables. Testing a driver on a multiprocessor machine is more likely to expose problems than testing on a single-processor machine.

Interoperability between drivers must also be tested, particularly because different devices can share interrupt levels. If possible, configure another device at the same interrupt level as the one being tested. Then stress-test the driver to determine if it correctly claims its own interrupts and otherwise operates according to expectations. Stress tests should be run on both devices at once. Even if the devices do not share an interrupt level, this test can still be valuable; for example, if serial communication devices start to experience errors while a network driver is being tested, this could indicate that the network driver is causing the rest of the system to encounter interrupt latency problems.

Driver performance under these stress tests should be measured using UNIX performance-measuring tools. This can be as simple as using the `time`(1) command along with commands used for stress tests.

DDI/DKI Compliance Testing

To ensure compatibility with later releases and reliable support for the current release, every driver should be Solaris 8 DDI/DKI compliant. One way to determine if the driver is compliant is by inspection. The driver can be visually inspected to ensure that only kernel routines and data structures specified in *man pages section 9F: DDI and DKI Kernel Functions* and *man pages section 9S: DDI and DKI Data Structures* of the *Solaris 8 Reference Manual Collection* are used.

The Solaris 8 Driver Developer Kit (DDK) includes a DDI compliance tool (DDICT) that checks device driver C source code for non-DDI/DKI compliance and issues either error or warning messages when it finds non-compliant code. For best results, all drivers should be written to pass DDICT. See `www.sun.com/solaris/ddk`.

Installation and Packaging Testing

Drivers are delivered to customers in *packages*. A package can be added and removed from the system using a standard mechanism (see the *Application Packaging Guide).*

Test to be sure the driver has been correctly packaged, to ensure that the end user can add it to and remove it from a system. In testing, the package should be installed and removed from every type of media on which it will be released and on several system configurations. Packages must not make unwarranted assumptions about the directory environment of the target system. Certain valid assumptions, however, can be made about where standard kernel files are kept. It is a good idea to test adding and removing of packages on newly installed machines that have not been modified for a development environment. It is a common packaging error for a package to use

a tool or file that exists only in a development environment, or only on the driver writer's own development system. For example, no tools from Source Compatibility package, SUNWscpu, should be used in driver installation programs.

The driver installation must be tested on a minimal Solaris system without any of the optional packages installed.

Testing Specific Types of Drivers

Because each type of device is different, it is difficult to describe how to test them all specifically. This section provides some information about how to test certain types of standard devices.

Tape Drivers

Tape drivers should be tested by performing several archive and restore operations. The `cpio`(1) and `tar`(1) commands can be used for this purpose. The `dd`(1M) command can be used to write an entire disk partition to tape, which can then be read back and written to another partition of the same size, and the two copies compared. The `mt`(1) command will exercise most of the I/O controls that are specific to tape drivers (see `mtio`(7I)); all the options should be attempted. The error handling of tape drivers can be tested by attempting various operations with the tape removed, attempting writes with the write protect on, and removing power during operations. Tape drivers typically implement exclusive-access `open`(9E) calls, which should be tested by having a second process try to open the device while a first process already has it open.

Disk Drivers

Disk drivers should be tested in both the raw and block device modes. For block device tests, a new file system should be created on the device and mounted. Multiple file operations can be performed on the device at this time.

Note - The file system uses a page cache, so reading the same file over and over again will not really exercise the driver. The page cache can be forced to retrieve data from the device by memory-mapping the file (with `mmap`(2)), and using `msync`(3C) to invalidate the in-memory copies.

Another (unmounted) partition of the same size can be copied to the raw device and then commands such as `fsck`(1M) can be used to verify the correctness of the copy. The new partition can also be mounted and compared to the old one on a file-by-file basis.

Asynchronous Communication Drivers

Asynchronous drivers can be tested at the basic level by setting up a `login` line to the serial ports. A good test is if a user can log in on this line. To sufficiently test an asynchronous driver, however, all the I/O control functions must be tested, with many interrupts at high speed. A test involving a loopback serial cable and high data transfer rates will help determine the reliability of the driver. Running `uucp`(1C) over the line also provides some exercise; however, since `uucp`(1C) performs its own error handling, it is important to verify that the driver is not reporting excessive numbers of errors to the `uucp`(1C) process.

These types of devices are usually STREAMS based.

Network Drivers

Network drivers can be tested using standard network utilities. `ftp`(1) and `rcp`(1) are useful because the files can be compared on each end of the network. The driver should be tested under heavy network loading, so that various commands can be run by multiple processes. Heavy network loading means:

- Traffic to the test machine is heavy.
- Traffic among all machines on the network is heavy.

Network cables should be unplugged while the tests are executing to ensure that the driver recovers gracefully from the resulting error conditions. Another important test is for the driver to receive multiple packets in rapid succession (*back-to-back* packets). In this case, a relatively fast host on a lightly loaded network should send multiple packets in quick succession to the test machine. It should be verified that the receiving driver does not drop the second and subsequent packets.

These types of devices are usually STREAMS based.

CHAPTER 17

Debugging

Debugging is the process of finding and eliminating faults from software. Almost every device driver writer will be faced with a difficult bug at some point in the development process. This chapter presents an overview of the tools available to make this process easier.

Note - In this document the term ”IA” refers to the Intel 32–bit processor architecture, which includes the Pentium, Pentium Pro, Pentium II, Pentium II Xeon, Celeron, Pentium III, and Pentium III Xeon processors and compatible microprocessor chips made by AMD and Cyrix.

Machine Configuration

Before you begin developing a Solaris driver, it is helpful to set up your test platform for this purpose. It is safest to test on a separate test system. This section explains how to set up a pair of machines for development, and how to prepare a test system for disaster recovery.

Setting Up a `tip(1)` Connection

A serial connection can be made between a test system (the machine executing the code to be debugged) and a host system using `tip`(1). This connection enables a window on the host system, called a *tip window*, to be used as the console of the test machine. See `tip`(1) for additional information.

Using a tip window confers the following advantages:

- Interactions with the test system or `kadb` can be monitored. For example, the window can keep a log of the session for use if the driver crashes the test system.
- The test machine can be accessed remotely by logging into a host machine (often called a *tip host*) and using `tip`(1) to connect to the test machine.

Note - A tip connection (and a second machine) is not required to debug a Solaris 8 device driver, but is recommended.

Setting Up the Host System

To set up the host system, do the following:

1. **Connect the host system to the test machine using serial port A on both machines. This connection must be made with a *null modem* cable.**

2. **On the host system, make an entry in `/etc/remote` for the connection if it is not already there (see `remote`(4)).**

   The terminal entry must match the serial port being used. The Solaris 8 operating environment comes with the correct entry for serial port B, but a terminal entry must be added for serial port A:

```
debug:\
        :dv=/dev/term/a:br#9600:el=^C^S^Q^U^D:ie=%$:oe=^D:
```

   Note - The baud rate must be set to 9600.

   ---

3. **In a shell window on the host, run `tip`(1) and specify the name of the entry**:

```
% tip debug
connected
```

   The shell window is now a *tip window* connected to the console of the test machine.

Caution - Do not use `STOP-A` (for SPARC machines) or `F1-A` (for IA machines) on the host machine to send a break to stop the test machine. This action actually stops the host machine. To send a break to the test machine, type **~#** in the tip window. Commands such as this are recognized only if they are the first characters on a line, so press the `Return` key or `Control-U` first if there is no effect.

Setting Up the Test System for SPARC Platforms

A quick way to set up the test machine is to unplug the keyboard before turning on the machine. The machine then automatically uses serial port A as the console.

Another way to set up the test machine is to use boot PROM commands to make serial port A the console. On the test machine, at the boot PROM `ok` prompt, direct console I/O to the serial line. To make the test machine always come up with serial port A as the console, set the environment variables *input-device* and *output-device.*

```
ok setenv input-device ttya
ok setenv output-device ttya
```

The `eeprom` command can also be used to make serial port A the console. As root user, execute the following commands to make the *input-device* and *output-device* parameters point to serial port A.

```
# eeprom input-device=ttya
# eeprom output-device=ttya
```

Executing the `eeprom` commands causes the console to switch to serial port A during reboot.

Setting Up the Test System for IA Platforms

On IA platforms, use the `eeprom` command to make serial port A the console. The procedure for this is the same as for SPARC platform and is discussed above. Executing the `eeprom` commands causes the console to switch to serial port A (COM1) during reboot.

Note - Unlike SPARC machines, where the *tip* connection maintains console control throughout the boot process, IA machines don't transfer console control to the *tip* connection until an early stage in the boot process.

Preparing for Disasters

It is possible for a driver to render the system incapable of booting. To avoid system reinstallation in this event, some advance work must be done.

Back Up Critical System Files

A number of driver-related system files are difficult, if not impossible, to reconstruct. Files such as `/etc/name_to_major`,`/etc/driver_aliases`, `/etc/driver_classes`, and `/etc/minor_perm` can be corrupted if the driver crashes the system during installation (see `add_drv`(1M)).

To be safe, after the test machine is in the proper configuration, make a backup copy of the root file system. If you plan on modifying the `/etc/system` file, make a backup copy of the file before modifying it.

Prepare and Boot an Alternate Kernel

A good strategy to avoid rendering a system inoperable is to make a copy of the kernel and associated binaries, and to boot that instead of the default kernel. To do so, make a copy of the drivers in `/platform/*` as follows:

```
# cp -r /platform/'uname -i'/kernel /platform/'uname -i'/kernel.test
```

When developing your driver, place it in `/platform/'uname -i'/kernel.test/drv` and boot that kernel instead of the default kernel:

```
# reboot -- kernel.test/unix
```

or from the PROM:

```
ok boot kernel.test/unix
```

This results in the test kernel and drivers being booted:

```
Rebooting with command: boot kernel.test/unix
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a  File and args:
kernel.test/unix
SunOS Release 5.8 Version Generic 32-bit
Copyright 1983-2000 Sun Microsystems, Inc.  All rights reserved.
...
```

Alternately, the module path can be changed by booting with the ask (`-a`) option:

```
ok boot -a
```

This results in a series of prompts which you can use to configure the way the kernel boots:

```
Rebooting with command: boot -a
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a  File and args: -a
Enter filename [kernel/sparcv9/unix]: kernel.test/sparcv9/unix
Enter default directory for modules
[/platform/sun4u/kernel.test /kernel /usr/kernel]: <CR>
Name of system file [etc/system]: <CR>
SunOS Release 5.8 Version Generic 64-bit
Copyright 1983-2000 Sun Microsystems, Inc.  All rights reserved.
root filesystem type [ufs]: <CR>
Enter physical name of root device
[/sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a]: <CR>
```

Prepare Other Back Up Plans

If the system is attached to a network, the test machine can be added as a client of a server. If a problem occurs, the system can be booted off the network. The local disks can then be mounted and fixed. Alternatively, the system can be booted directly from the Solaris 8 CD-ROM.

Another way to recover from disaster is to have another bootable root file system. Use `format`(1M) to make a partition the exact size of the original, then use `dd`(1M) to copy it. After making a copy, run `fsck`(1M) on the new file system to ensure its integrity.

Later, if the system cannot boot from the original root partition, boot the backup partition and use `dd`(1M) to copy the backup partition onto the original one. If the system will not boot but the root file system is undamaged (just the boot block or boot program was destroyed), boot off the backup partition with the ask (`-a`) option, then specify the original file system as the root file system.

Saving System Crash Dumps

When the system panics, it writes the memory image to the dump device. The dump device by default is the most suitable swap device. The dump is a system crash dump, similar to core dumps generated by applications. On rebooting after a panic, `savecore`(1M) checks the dump device for a crash dump. If one is found, it makes a copy of the kernel's symbol table (called `unix.n`) and dumps a core file (called `vmcore.n`) in the core image directory which by default is `/var/crash/`*machine_name*. There must be enough space in `/var/crash` to contain the core dump or it will be truncated. `mdb`(1) can then be used on the core dump and the saved kernel.

In the Solaris 8 operating system, crash dump is enabled by default. The `dumpadm`(1M) command is used to configure system crash dumps. Use the `dumpadm`(1M) command to verify that crash dumps are enabled and to determine the location of the directory where core files are saved. See `dumpadm`(1M) for more information.

Note - `savecore(1M)` can be prevented from filling the file system if there is a file called `minfree` in the directory in which the dump will be saved. This file contains a number of kilobytes to remain free after `savecore(1M)` has run. However, if not enough space is available, the core file is not saved.

Disaster Recovery

If the `/devices` or `/dev` directories are damaged—most likely to occur if the driver crashes during `attach`(9E)—they can be recreated by booting the system and running `fsck`(1M) to repair the damaged root file system. The root file system can

then be mounted. Re-create /dev and /devices by running devfsadm(1M) and specifying the /devices directory on the mounted disk.

On SPARC, for example, if the damaged disk is /dev/dsk/c0t3d0s0, and an alternate boot disk is /dev/dsk/c0t1d0s0, do the following:

```
ok boot disk1
...
Rebooting with command: boot kernel.test/unix
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@31,0:a  File and args:
kernel/unix
SunOS Release 5.8 Version Generic 32-bit
Copyright 1983-2000 Sun Microsystems, Inc.  All rights reserved.
...
# 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
```

Caution - Fixing /devices and /dev may allow the system to boot, but other parts of the system can still be corrupted. This is only a temporary fix to allow saving information (such as system crash dumps) before reinstalling the system.

Recommended Coding Practices

This section provides information to make drivers easier to debug. Because the driver is operating much closer to the hardware and without the protection of the operating system, debugging kernel code is more difficult than debugging user-level code. For example, a stray pointer access can crash the entire system. It is important to build debugging support into your driver, both for development and maintenance.

Use cmn_err(9F) to Log Driver Activity

cmn_err(9F) is used to print messages to the console from within the device driver. It provides additional format characters (such as %b) to print device register bits. See cmn_err(9F) and “Printing Messages” on page 40 for more information.

Note - Though `printf()` and `uprintf()` currently exist, they should not be used if the driver is to be Solaris DDI-compliant.

Use ASSERT(9F) to Catch Invalid Assumptions

```
void ASSERT(EXPRESSION)
```

ASSERT(9F) is a macro used to halt the execution of the kernel if a condition *expected* to be true is *actually* false. ASSERT provides a way for the programmer to validate the assumptions made by a piece of code.

ASSERT is defined only when the compilation symbol DEBUG is defined; ASSERT is compiled out of the code if DEBUG is not defined and therefore has no effect.

For example, if a driver pointer should be non-NULL and is not, the following assertion could be used to check the code:

```
ASSERT(ptr != NULL);
```

If the driver is compiled with DEBUG defined and the assertion fails, a message is printed to the console and the system panics:

```
panic: assertion failed: ptr != NULL, file: driver.c, line: 56
```

Note - Because ASSERT(9F) uses the DEBUG compilation symbol, any conditional debugging code should also be based on DEBUG.

Assertions are an extremely valuable form of active documentation.

Use mutex_owned(9F) to Validate and Document Locking Requirements

```
int mutex_owned(kmutex_t *mp);
```

A significant portion of driver development involves properly handling multiple threads. Comments should always be used when a mutex is acquired; they are even more useful when an apparently necessary mutex is *not* acquired. To determine if a mutex is held by a thread, use mutex_owned(9F) within ASSERT(9F):

```
void helper(void)
{
        /* this routine should always be called with xsp's mutex held */
        ASSERT(mutex_owned(&xsp->mu));
        ...
```

```
}
```

Warning - `mutex_owned`(9F) is only valid within ASSERT(9F) macros. Under no circumstances should you use it to control the behavior of a driver.

Use Conditional Compilation to Toggle Costly Debugging Features

Debugging code can be placed in a driver by conditionally compiling code based on a preprocessor symbol such as `DEBUG` or by using a global variable. Conditional compilation has the advantage that unnecessary code can be removed in the production driver. Using a variable allows the amount of debugging output to be chosen at runtime. This can be accomplished by setting a debugging level at runtime with an ioctl or through a debugger. Commonly, these two methods are combined.

The following example relies on the compiler to remove unreachable code (the code following the always-false test of zero), and also provides a local variable that can be set in `/etc/system` or patched by a debugger.

```
#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");
```

This method handles the fact that `cmn_err`(9F) has a variable number of arguments. Another method relies on the macro having one argument, a parenthesized argument list for `cmn_err`(9F), which the macro removes. It also removes the reliance on the optimizer by expanding the macro to nothing if `DEBUG` is not defined.

```
#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!"));
```

You can extend this in many ways, such as by having different messages from `cmn_err`(9F), depending on the value of `xxdebug`, but be careful not to obscure the code with too much debugging information.

Another common scheme is to write an `xxlog()` function, which uses `vsprintf`(9F) or `vcmn_err`(9F) to handle variable argument lists.

Runtime Debugging Tools

This section describes some of the mechanisms that can be used to debug drivers at runtime. Runtime debugging is typically performed during driver development; this process is substantially simplified if you have followed the coding practices described in the previous section.

/etc/system

The `/etc/system` file serves several purposes, but for driver development, the most important is that it allows you to set the value of kernel variables at boot time. This can be used to toggle different behaviors in a driver, or to enable certain debugging features made available by the kernel.

`/etc/system` is read only once, while the kernel is booting. After this file is modified, the system must be rebooted for the changes to take effect. If a change in the file causes the system not to work, boot with the ask (`-a`) option and specify `/dev/null` as the system file.

The `set` command is used to change the value of module or kernel variables:

- To set module variables, specify the module name and the variable:

  ```
  set module_name:variable=value
  ```

  For example, to set the variable `test_debug` in the driver *test*, use the following set command:

  ```
  set test:test_debug=1
  ```

- To set a variable exported by the kernel itself, omit the module name. Other assignments are also supported, such as bitwise OR'ing a value into an existing value:

  ```
  set moddebug | 0x80000000
  ```

See `system`(4) for more information.

Note - Most kernel variables are not guaranteed to be present in subsequent releases.

---

moddebug

`moddebug` is a kernel variable that controls the module loading process. The possible values are:

`0x80000000`	Prints messages to the console when loading or unloading modules
`0x40000000`	Gives more detailed error messages
`0x20000000`	Prints more detail when loading or unloading (such as including the address and size)
`0x00001000`	No auto-unloading drivers: the system will not attempt to unload the device driver when the system resources become low
`0x00000080`	No auto-unloading streams: the system will not attempt to unload the streams module when the system resources become low
`0x00000010`	No auto-unloading of kernel modules of any type
`0x00000001`	If running with `kadb`, `moddebug` causes a breakpoint to be executed and a return to `kadb` immediately before each module's `_init`(9E) routine is called. Also generates additional debug messages when the module's `_info` and `_fini` routines are executed.

kmem_flags

`kmem_flags` is a kernel variable used to enable debugging features in the kernel's memory allocator. Setting `kmem_flags` to `0xf` enables the allocator's debugging features. These include runtime checks to find:

- Code that writes to a buffer after it is freed
- Code using memory before it is initialized
- Code that writes past the end of a buffer

The "Debugging With the Kernel Memory Allocator" in the *Solaris Modular Debugger Guide* describes how the kernel memory allocator can be used to determine the root cause of these problems.

Note - Testing and developing with `kmem_flags` set to `0xf` is extremely valuable since it can detect latent memory corruption bugs. Because setting `kmem_flags` to `0xf` changes the internal behavior of the kernel memory allocator, it remains important to thoroughly test without `kmem_flags` as well.

---

modload, modunload, and modinfo

The kernel automatically loads needed modules and unloads unused ones, so modload(1M), modunload(1M), and modinfo(1M) are not very useful for system administration. However, they can be useful when debugging and stress testing driver load/unload scenarios.

modload(1M) can be used to force a module into memory. The kernel might subsequently unload the module, but modload(1M) can be used to verify that the driver has no unresolved references when loaded. Keep in mind that loading a driver does *not* mean that the driver will attach. A driver that loads successfully will have its _info(9E) entrypoint called, but will not neccessarily attach.

You can use modinfo(1M) to confirm that your driver is loaded. Here is an example:

```
$ 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)
```

The number in the info field is the major number chosen for the driver. modunload(1M) can be used to unload a module, given a module ID (which can be found in the leftmost column of modinfo(1M) output). A common bug is that a driver refuses to unload, even after a modunload(1M) is issued. Note that a driver will not unload if the system thinks the driver is busy. This occurs when the driver fails detach(9E), either because the driver really is busy, or because the detach entry point is implemented incorrectly.

To remove all currently unused modules from memory, run modunload with a module ID of 0:

```
# modunload -i 0
```

The kadb Kernel Debugger

kadb(1M) is a kernel debugger with facilities for disassembly, breakpoints, watch points, data display, and stack tracing. This section provides a tutorial on some of the features of kadb. For further information, consult the kadb(1M) man page.

Starting kadb

In order to start up kadb, the system must be booted with kadb(1M) enabled:

```
ok boot kadb
...
Rebooting with command: boot kadb
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a
File and args: kadb
kadb: kernel/sparcv9/unix
Size: 499808+109993+132503 Bytes
/platform/sun4u/kernel/sparcv9/unix loaded - 0x11e000 bytes used
SunOS Release 5.8 Version Generic 64-bit
Copyright 1983-2000 Sun Microsystems, Inc.  All rights reserved
....
```

By default, `kadb`(1M) boots (and debugs) `kernel/unix`, or `kernel/sparcv9/unix` on a system capable of running a 64-bit kernel. To boot `kadb` with an alternate kernel, pass the `–D` flag to boot, as follows:

```
ok boot kadb -D kernel.test/unix
...
Rebooting with command: boot kadb -D kernel.test/unix
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a  File
and args: kadb -D kernel.test/unix
kadb: kernel.test/unix
Size: 482384+67201+88883 Bytes
/platform/sun4u/kernel.test/unix loaded - 0xfe000 bytes used
SunOS Release 5.8 Version dacf-fixes:11/13/99 32-bit
Copyright 1983-2000 Sun Microsystems, Inc.  All rights reserved.
...
```

In this example, the 32-bit version of the alternate kernel *kernel.test* was booted. Another option is to pass `kadb` the `–d` flag, which causes `kadb` to prompt for the kernel name. The `–d` flag also causes `kadb`(1M) to provide a prompt after it has loaded the kernel, so breakpoints can be set.

```
ok boot kadb -d
...
Rebooting with command: boot kadb -d
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a  File
and args: kadb -d
kadb: kernel.test/unix
Size: 482384+67201+88883 Bytes
/platform/sun4u/kernel.test/unix loaded - 0xfc000 bytes used
stopped at      _start:         sethi   %hi(0x10006c00), %g1
kadb[0]:
```

At this point you can set breakpoints or continue execution with the `:c` command.

Note - Kernel modules are dynamically loaded. Consequently, driver symbols are not available until the driver is loaded. To set breakpoints in modules that have not been loaded, use deferred breakpoints. For information on deferred breakpoints, refer to “Breakpoints” on page 313.

`kadb`(1M) passes any kernel flags to the booted kernel. For example, to boot an alternate kernel and pass the `–r` flag:

```
ok boot kadb -r -D kernel.test/unix
...
Rebooting with command: boot kadb -r -D kernel.test/unix
Boot device: /sbus@1f,0/espdma@e,8400000/esp@e,8800000/sd@0,0:a
File and args: kadb -r -D kernel.test/unix
kadb: kernel.test/unix
Size: 482384+67201+88883 Bytes
/platform/sun4u/kernel.test/unix loaded - 0xfe000 bytes used
SunOS Release 5.8 Version Generic 32-bit
Copyright 1983-2000 Sun Microsystems, Inc.  All rights reserved.
obpsym: symbolic debugging is available.
Read 208377 bytes from misc/forthdebug
configuring IPv4 interfaces: le0.
Hostname: test
Configuring /dev and /devices
```

After the system is booted, sending a *break* passes control to kadb(1M). A break is generated with STOP-A (on the console of SPARC machines), or wtih F1-A (on the console of IA machines), or by using ~# (if the console is connected through a *tip* window).

```
...

The system is ready.

test console login: ~
stopped at      edd000d8:       ta      %icc,%g0 + 125
kadb[0]:
```

The number in brackets is the CPU that kadb(1M) is currently executing on; the remaining CPUs are halted. The CPU number is zero on a uniprocessor system.

Warning - Before rebooting or turning off the power, always halt the system cleanly (with init 0 or shutdown). Buffers might not be flushed otherwise. If the shutdown must occur from the boot PROM prompt, make sure to flush buffers using the sync command at the ok prompt.

To return control to the operating system, use :c.

```
kadb[0]: :c
test console login:
```

Exiting

To exit kadb(1M), use $q. On SPARC machines, this will exit to the ok prompt. On IA machines, you will be prompted to reboot the system.

```
kadb[0]: $q
Type ‘go’ to resume
ok
```

kadb(1M) can be resumed by typing go at the ok prompt.

Warning - No other commands should be performed from the PROM if the system is to be resumed. PROM commands other than `go` can change system state that the Solaris 8 operating environment depends upon.

Staying at the `kadb`(1M) prompt for too long can cause the system to lose track of the time of day, and can cause network connections to time out.

Commands

The general form of a `kadb` command is:

```
[ address ] [ ,count ] command [;]
```

If *address* is omitted, the current location is used ('.' could also be used to represent the current location). The address can be a kernel symbol. If *count* is omitted, it defaults to 1.

Commands to `kadb` consist of a *verb* followed by a *modifier* or list of modifiers. Verbs can be:

/	Prints locations starting at *address* in the kernel address space
=	Prints the value of *address* itself
>	Assigns a value to a debugger variable or machine register
<	Reads a value from a debugger variable or machine register
RETURN	Repeats the previous command with a count of 1. Increments '.' (the current location)

With `/` and `=`, output format specifiers can be used. Lowercase letters normally print 2 bytes, uppercase letters print 4 bytes:

o, O	2-, 4-byte octal
g	8-byte octal
G	8-byte unsigned octal
d, D	2-, 4-byte decimal
e	8-byte decimal
E	8-byte unsigned decimal

x, X	2-, 4-byte hexadecimal
J	8-byte hexadecimal
K	4–byte hexadecimal for 32–bit programs, 8–byte hexadecimal for 64–bit programs. Use this format specifier to examine pointers.
u, U	2-, 4-byte unsigned decimal
c	Prints the addressed character
C	Prints the addressed character using ^ escape notation
s	Prints the addressed string
S	Prints the addressed string using ^ escape notation
i	Prints as machine instructions (disassemble)
a	Prints the value of '.' in symbolic form
w, W	2-, 4-byte write
Z	8-byte write

Warning - When using w, W or Z to modify a kernel variable, make sure that the size of the variable matches the size of the write you are performing. If you specify an incorrect size you could corrupt neighboring data.

For example, to set a bit in the `moddebug` variable when debugging a driver, first examine the value of `moddebug`, then set it to the desired bit.

```
kadb[0]: moddebug/X
moddebug:
moddebug:       1000
kadb[0]: moddebug/W 0x80001000
moddebug:       0x1000 = 0x80001000
```

Routines can be disassembled with the '`i`' command. This is useful when tracing crashes, since the only information might be the program counter at the time of the crash. For example, to print the first four instructions of the `kmem_alloc` function:

```
kadb[0]: kmem_alloc,4/i
kmem_alloc:
kmem_alloc: save    %sp, -0x60, %sp
sub     %i0, 0x1, %l6
sra     %l6, 0x3, %i5
tst     %i5
```

Specify symbolic notation with the '`a`' command, to show the addresses:

```
kadb[0]: kmem_alloc,4/ai
kmem_alloc:
kmem_alloc:     save    %sp, -0x60, %sp
kmem_alloc+4:   sub     %i0, 0x1, %l6
kmem_alloc+8:   sra     %l6, 0x3, %i5
kmem_alloc+0xc: tst     %i5
```

Register Identifiers

You can discover what machine registers are available on your processor architecture using the `$r` command. This example shows the output of `$r` on a SPARC system with the sun4u architecture:

```
kadb[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
npc   edd000dc edd000dc:        nop
```

`kadb` exports each of these registers as a debugger variable with the same name. Reading from the variable fetches the current value of the register. Writing to the variable changes the value of the associated machine register. For example, you can change the value of the '`%o0`' register:

```
kadb[0]: <o0=K
                0
kadb[0]: 0x1>o0
kadb[0]: <o0=K
                1
```

Display and Control Commands

The following commands display and control the status of kadb(1M):

`$b`	Display all breakpoints
`$c`	Display stack trace
`$d`	Change default radix to value of dot
`$q`	Quit
`$r`	Display registers
`$M`	Display built-in macros

'`$c`' is useful when a breakpoint is reached, but is usually not useful if kadb(1M) is entered at a random time. The number of arguments to print can be passed following the '`$c`' ('`$c 2`' for two arguments).

Breakpoints

In kadb(1M), breakpoints can be set. When reached, the kernel will automatically drop back into kadb. The standard form of a breakpoint command is:

```
[module_name#] addr [, count]:b [command]
```

`addr` is the address at which the program will be stopped and the debugger will receive control, `count` is the number of times that the breakpoint address occurs before stopping, and `command` is almost any `adb(1)` command.

The optional `module_name` specifies deferred breakpoints that are set when the module is loaded. `module_name` identifies a particular module that contains `addr`. If the module has been loaded, `kadb` will try to set a regular breakpoint; if the module is not loaded, `kadb` will set a deferred breakpoint. When the module is loaded, `kadb` will try to resolve the location of the breakpoint and convert the breakpoint to a regular breakpoint.

Other breakpoint commands are:

`:c`	Continue execution
`:d`	Delete breakpoint
`:s`	Single step
`:e`	Single step, but step over function calls
`:u`	Stop after return to caller of current function
`:z`	Delete all breakpoints

The following example sets a breakpoint in `scsi_transport`(9F), a commonly used routine. Upon reaching the breakpoint, '`$c`' is used to print a stack trace. The top of the stack is displayed first. Note that `kadb` does not know how many arguments were passed to each function.

```
stopped at      edd000d8:       ta      %icc,%g0 + 125
kadb[0]: scsi_transport:b
kadb[0]: :c
test console login: root
breakpoint at:
scsi_transport: save    %sp, -0x60, %sp
kadb[0]: $c
scsi_transport(702bb578,1000,1,10000,0,702bb7fe)
sdstrategy(1019c8c0,702bb61c,0,0,702bb578,70cad7b8) + 704
bdev_strategy(1042a808,70cad7b8,705f3efc,40,10597900,2000) + 98
ufs_getpage_miss(70cad7b8,0,10597900,0,0,4023ba8c) + 2b0
ufs_getpage(0,0,0,0,2000,4023ba8c) + 7c0
segvn_fault(4023ba8c,2000,ff3b0000,0,0,0) + 7c8
as_fault(1,ff3b0000,70d98030,2000,0,ff3b0000) + 49c
pagefault(0,0,70df8048,705c7450,0,ff3b0000) + 4c
trap(10080,10000,ff3c4ea4,70df8048,ff3b0000,1) + db4
kadb[0]: $b
breakpoints
count   bkpt            type      len   command
1       scsi_transport  :b instr  4
kadb[0]: scsi_transport:d
kadb[0]: :c
```

Conditional Breakpoints

Breakpoints can also be set to occur only if a certain condition is met. By providing a command, the breakpoint will be taken only if the count is reached or the command returns zero. For example, a breakpoint that occurs only on certain I/O controls could be set in the driver's `ioctl`(9E) routine. This is the general syntax of conditional breakpoints:

```
address,count:b command
```

In this example, address is the address at which to set the breakpoint. count is the number of times the breakpoint should be ignored (note that 0 means break only when the command returns zero). command is the kadb(1M) command to execute.

Here is an example of breaking only in the sdioctl() routine if the DKIOGVTOC (get volume table of contents) I/O control occurs.

```
kadb[0]: sdioctl+4,0:b <i1-0x40B
kadb[0]: $b
breakpoints
count   bkpt            type      len   command
0       sdioctl+4       :b instr  4     <i1-0x40B
kadb[0]: :c
```

Adding four to sdioctl skips to the second instruction in the routine, bypassing the save instruction that establishes the stack. The '<i1' refers to the first input register, which is the second parameter to the routine (the cmd argument of ioctl(9E)). The count of zero is impossible to reach, so it stops only when the command returns zero, which is when 'i1 - 0x40B' is true. This means i1 contains 0x40B (the value of the ioctl(9E) command, determined by examining the ioctl definition).

To force the breakpoint to be reached, the prtvtoc(1M) command is used. It is known to issue this I/O control:

```
# prtvtoc /dev/rdsk/c0t0d0s0
breakpoint at:
sdioctl+4:      mov     %i5, %l0
kadb[0]: $c
sdioctl(800000,40b,ffbefb54,100005,704a3ce8,4026bc7c) + 4
ioctl(3,40b,70ca27b8,40b,ffbefb54,0) + 1e0
```

Macros

kadb(1M) supports macros that are used for displaying kernel data structures. kadb(1M) macros can be displayed with $M. Macros are used in the form:

```
[ address ] $<macroname
```

threadlist is a useful macro that displays the stacks of all the threads in the system.

```
kadb[0]: $<threadlist

                ============== thread_id        10404000
p0+0x300:
                process args    sched

t0+0xa8:        lwp             procp           wchan
                1041b810        10424688        0
t0+0x24:
                pc              sp
                sched+0x4f4     10403be8
?(10404000,1040c000,2,10424604,0,6e)
_start(10006ef4,1041adb0,1041adb0,1041adb0,10462910,50) + 15c
```

```
...
                ============== thread_id        40043e60
p0+0x300:
                process args    sched

40043f08:       lwp             procp           wchan
                0               10424688        10473c56
40043e84:
                pc              sp
                cv_wait+0x60    40043c08
?(10473c56,10473c5c,0,40043cd0,40043e60,10093084)
ufs_thread_idle(10471e80,0,10473c5c,10424688,81010100,0) + bc
thread_start(0,0,0,0,0,0) + 4
...
```

Another useful macro is `thread`. Given a thread ID, this macro prints the corresponding `thread` structure. This can be used to look at a certain thread found with the `threadlist` macro, to look at the owner of a mutex, or to look at the current thread, as shown here:

```
kadb[0]: <g7$<thread
70e87ac0:       link            stk             startpc
                0               4026bc80        0
70e87acc:       bound_cpu       affinitycnt     bind_cpu
                0               0               -1
70e87ad4:       flag    proc_flag       schedflag
                0       4               3
70e87ada:       preempt preempt_lk      state
                0       0               4
70e87ae0:       pri     epri
                40      0
70e87ae4:
                pc              sp
                10098350        4026b618
70e87aec:       wchan0          wchan           sobj_ops
                0               0               0
70e87af8:       cid             clfuncs         cldata
                1               10470ffc        702c0488
70e87b04:       ctx             lofault         onfault
                0               0               0
...
```

Note - No type information is kept in the kernel, so using a macro on an inappropriate address results in garbage output.

Macros do not necessarily output all the fields of the structures, nor is the output necessarily in the order given in the structure definition. Occasionally, memory needs to be dumped for certain structures and then matched with the structure definition in the kernel header files.

Warning - Drivers should never reference header files and structures *not* listed in *man pages section 9S: DDI and DKI Data Structures*. However, examining non-DDI-compliant structures (such as `thread` structures) can be useful in debugging drivers.

Output Pager

Some `kadb` commands (like `$<threadlist`) output lots of data, which can scroll off of the screen very rapidly. `kadb` provides a simple output pager to remedy this problem. The pager command is *lines*`::more`, where *lines* represents the number of lines to print before pausing the console output. Keep in mind that this does not take into account lines that wrap because they are wider than the terminal width. Here is an example usage:

```
kadb[0]: 0t10::more
kadb[0]: $<threadlist

                ============== thread_id        10408000
p0+0x4c0:
                process args    sched

t0+0x128:       lwp             procp           wchan
            10429ed0         104393e8         0
t0+0x38:
                pc              sp
                sched+0x4e4     104071f1
?(10408000,10414c00,2,104393e8,10439308,0)
_start(10007588,104292e0,104292e0,104292e0,1043b8b0,10429360) + 200

                ============== thread_id        2a10001fd40
p0+0x4c0:
                process args    sched
--More-- <SPACE>

...
```

Pressing the space bar at the "`--More--`" prompt pages the output by the number of lines specified to `::more` (in this case, 10). Pressing "Return" prints only the next line of output. You can abort the output and return to the `kadb` prompt by typing `Ctrl-C`. To disable the pager, issue '`0::more`' at the `kadb` prompt.

Example: `kadb` on a Deadlocked Thread

This example shows how `kadb` can be used to debug a driver bug. This example was taken from the development of the ramdisk sample driver. This driver exports physical memory as a virtual disk. In this case, the `dd(1M)` command hangs while

trying to copy some data onto the device and cannot be aborted. Though a crash dump could be forced, for illustrative purposes, kadb(1M) will be used. After logging into the system remotely, ps was used to determine that the system was still running; and only the dd(1M) command is hung.

At this point, the system is rebooted with kadb, which can now be entered by typing STOP-A on the system console. After the rest of the kernel has loaded, moddebug is patched to see if loading is the problem:

```
stopped at:
edd000d8:       ta      %icc,%g0 + 125
kadb[0]: moddebug/X
moddebug:
moddebug:       0
kadb[0]: moddebug/W 0x80000000
moddebug:       0x0             =       0x80000000
kadb[0]: :c
```

modload(1M) is used to load the driver, to separate module loading from the real access:

```
# modload /home/driver/drv/ramdisk
```

It loads without errors, so loading is not the problem. The condition is recreated with dd(1M):

```
# dd if=/dev/zero of=/devices/pseudo/ramdisk@0:c,raw
```

dd(1M) hangs. At this point, kadb(1M) is entered and the stack examined:

```
stopped at:
edd000d8:       ta      %icc,%g0 + 125
kadb[0]: $c
intr_vector() + 7dcfc0d8
debug_enter(0,0,10431e50,10,1,b0) + 78
zsa_xsint(80,7044a06c,44,7044a000,ff0113,0) + 278
zs_high_intr(7044a000,1,1,1042f78c,10424680,100949d0) + 20c
sbus_intr_wrapper(704dfad4,0,702bd048,7029cec0,630,10260250) + 30
current_thread(4001fe60,1041a550,10424698,10424698,10150f08,0) + 180
idle(1040b6c0,0,0,1041a550,704d6a98,0) + 54
thread_start(0,0,0,0,0,0) + 4
```

The presence of idle on the current thread stack indicates that this thread is not the cause of the deadlock. To determine the deadlocked thread, the entire thread list is checked:

```
kadb[0]: $<threadlist
...
                ============== thread_id        70cef120
70c8b1c0:
                process args    dd if=/dev/zero of=/devices/pseudo/ramdisk@0:c,raw

70cef1c8:       lwp             procp           wchan
                70fa9080        70c8aec0        70691fc8
70cef144:
                pc              sp
                sema_p+0x290    40313a78
```

```
?(70691fc8,10424680,1,1042b99c,10460f8c,70691fc8)
biowait(70691f60,1041a6c4,70691f60,70c385d0,40313bcc,705c73a0) + 8c
default_physio(1042e8fc,200,129,100,70eb5b54,705c73a0) + 3bc
write(2002,70aac1d0,70f9f9ac,200,4,200) + 23c
...
```

Of all the threads, only one has a stack trace which references the ramdisk driver. It seems that the process running `dd(1M)` is blocked in `biowait`(9F). `biowait`(9F)'s first parameter is a `buf`(9S) structure. The next step is to examine this structure:

```
kadb[0]:  70691f60$70691f60$
70691f60:       flags           forw            back
                204129          0               0
70691f6c:       av_forw         av_back         bcount
                0               0               512
70691fa0:       bufsize         error           edev
                0               0               1180000
70691f7c:       un.b_addr       _b_blkno        resid
                710e8000        0                0
70691f94:       proc            iodone          vp
                70c8aec0        0               0
70691f98:       pages
                0
```

The `resid` field is 0, which indicates that the transfer is complete. `physio`(9F) is still blocked, however. The reference for `physio`(9F) in the *Solaris 8 Reference Manual Collection* points out that `biodone`(9F) should be called to unblock `biowait`(9F). This is the problem; `rd_strategy()` did not call `biodone`(9F). Adding a call to `biodone`(9F) before returning fixes this problem.

Post-Mortem Debugging

When `kadb` is running and the system panics, control is passed to the debugger so that you can investigate the source of the problem. However, `kadb` is not always the best tool for problem analysis; frequently it is easier to use '`:c`' to continue execution and allow the system to save a crash dump. When the system reboots, you can perform *post-mortem* analysis on the saved crash dump. This process is analogous to debugging an application crash from a process core file.

Post-mortem analysis offers several advantages to driver developers: it allows more than one developer to examine a problem in parallel; it allows developers to retrieve information on a problem that occurred in production at a customer site, where it is not acceptable to debug interactively; it is necessary to perform certain types of advanced kernel analysis, such as checking for kernel memory leaks.

Getting Started With MDB

MDB provides sophisticated debugging support for analyzing kernel problems. This section provides an overview of MDB's features. For a more complete discussion of MDB's capabilities, refer to the *Solaris Modular Debugger Guide.*

MDB's command syntax is compatible with the `kadb` syntax and MDB can execute all of the `kadb` (and legacy `adb`) macros. These are stored in `/usr/lib/adb` and in `/usr/platform/‘uname -i‘/lib/adb` for 32-bit kernels; and in `/usr/lib/adb/sparcv9` and `/usr/platform/‘uname -i‘/lib/adb/sparcv9` for 64-bit kernels.

In addition to macro files, MDB supports 'debugger commands' or *dcmds*. These dcmds can be dynamically loaded at runtime from a set of debugger modules. MDB provides a first-class programming API for implementing debugger modules so that driver developers can implement their own custom debugging support. MDB also provides a host of usability features, such as command line editing, command history, an output pager, and online help.

MDB provides a rich set of modules and dcmds for debugging the Solaris kernel and associated modules and device drivers. These facilities allow you to formulate complex debugging queries: locate all the memory allocated by a particular thread; print a visual picture of a kernel STREAM; determine what type of structure a particular address refers to; locate leaked memory blocks in the kernel; analyze memory to locate stack traces; and more.

Note - In earlier versions of the Solaris operating environment, `adb`(1) was the recommended tool for post-mortem analysis. In the Solaris 8 operating environment, `mdb`(1) is the new recommended tool for post-mortem analysis; it provides an upward-compatible syntax and feature set. In addition, mdb includes features that surpass the set of commands available from the legacy `crash(1M)` utility.

To get started, run `mdb` and supply it with a system crash dump:

```
% cd /var/crash/testsystem
% ls
bounds     unix.0     vmcore.0
% mdb unix.0 vmcore.0
Loading modules: [ unix krtld genunix ip logindmux ptm pts nfs lofs ]
> ::status
debugging crash dump vmcore.1 (32-bit) from testsystem
operating system: 5.8 generic (sun4u)
```

When mdb responds with the '>' prompt, it is ready for commands. To examine the running kernel on a live system, type:

```
# mdb -k
Loading modules: [ unix krtld genunix ip logindmux ptm nfs ipc ]
> ::status
debugging live kernel (32-bit) on testsystem
operating system: 5.8 Generic (sun4u)
```

Important MDB Commands

This section provides a tutorial for some of the MDB debugger commands most applicable to driver authors. Note that the information presented here is dependent on the type of system used. A Sun Ultra 1 workstation running the 32-bit kernel was used to produce these examples.

The *Solaris Modular Debugger Guide* provides details about each debugger command discussed here, as well as more information about all aspects of MDB. Online help is available from within MDB using the `::help` built-in command.

Navigating the Device Tree

MDB provides the `::prtconf` dcmd to display the kernel device tree. The output of this dcmd is similar to the output of the `prtconf(1M)` command:

```
> ::prtconf
DEVINFO  NAME
704c9f00 SUNW,Ultra-1
    704c9e00 packages (driver not attached)
        704c9c00 terminal-emulator (driver not attached)
        704c9b00 deblocker (driver not attached)
        704c9a00 obp-tftp (driver not attached)
        704c9900 disk-label (driver not attached)
        704c9800 sun-keyboard (driver not attached)
        704c9700 ufs-file-system (driver not attached)
    704c9d00 chosen (driver not attached)
    704c9600 openprom (driver not attached)
        704c9400 client-services (driver not attached)
    704c9500 options, instance #0
    704c9300 aliases (driver not attached)
    704c9200 memory (driver not attached)
    704c9100 virtual-memory (driver not attached)
    704c9000 counter-timer (driver not attached)
    704c8f00 sbus, instance #0
        704c8d00 SUNW,CS4231 (driver not attached)
        704c8c00 auxio (driver not attached)
        704c8b00 flashprom (driver not attached)
        704c8a00 SUNW,fdtwo (driver not attached)
...
```

Each line of output represents a node in the kernel's device tree. The address to the left of each node name is the address of the `devinfo` node. The node can then be displayed using the `$<devinfo` macro or the `::devinfo` dcmd:

```
> 704c9f00::devinfo
704c9f00 SUNW,Ultra-1
         Driver properties at 0x704bb208:
           pm-hardware-state: "no-suspend-resume"
         System properties at 0x704bb190:
           relative-addressing: 0x1
           MMU_PAGEOFFSET: 0x1fff
           MMU_PAGESIZE: 0x2000
           PAGESIZE: 0x2000
```

```
> 704c9f00$<devinfo
0x704cbd20:
                name:
SUNW,Ultra-1
0x704c9f00:     parent          child           sibling
                0               704c9e00        0
0x704c9f10:     addr            nodeid          instance
                704be758        f00297e4        ffffffff
0x704c9f1c:     ops             parent_data     driver_data
                rootnex_ops     702baad0        0
0x704c9f28:     drv_prop_ptr    sys_prop_ptr    minor
                704bb208        704bb190        0
0x704c9f34:     next
                0
```

Use `::prtconf` to see where your driver has attached in the device tree, and to display device properties. You can also specify the verbose (–v) flag to `::prtconf` to display the properties for each device node:

```
> ::prtconf -v
DEVINFO  NAME
704c9f00 SUNW,Ultra-1
         Driver properties at 0x704bb208:
           pm-hardware-state: "no-suspend-resume"
         System properties at 0x704bb190:
           relative-addressing: 0x1
           MMU_PAGEOFFSET: 0x1fff
           MMU_PAGESIZE: 0x2000
           PAGESIZE: 0x2000
         ...
         704c8400 espdma, instance #0
            704c8200 esp, instance #0
                     Driver properties at 0x702ba7d8:
                       target0-sync-speed: 0x2710
                       target0-TQ
                       scsi-selection-timeout: <000000fa.> (device: <0x3d/0x
                       00000000>)
                       scsi-options: <00001ff8.> (device: <0x3d/0x00000000>)
                       scsi-watchdog-tick: <0000000a.> (device: <0x3d/0x00000000>)
                       scsi-tag-age-limit: <00000002.> (device: <0x3d/0x00000000>)
                       scsi-reset-delay: <00000bb8.> (device: <0x3d/0x00000000>)
         ...
                704c7c00 sd, instance #1 (driver not attached)
                         System properties at 0x704ba4e8:
                           lun: 0x0
                           target: 0x1
                           class_prop: "atapi"
                           class: "scsi"
         ...
```

Another way to locate instances of your driver is the `::devbindings` dcmd. Given a driver name, it displays a list of all instances of the named driver:

```
> ::devbindings sd
704c8100 sd (driver not attached)
704c7d00 sd, instance #0
         Driver properties at 0x702ba5a8:
           pm-hardware-state: "needs-suspend-resume"
```

```
          ddi-kernel-ioctl
        System properties at 0x704ba588:
          lun: 0x0
          target: 0x0
          class_prop: "atapi"
          class: "scsi"
704c7c00 sd, instance #1 (driver not attached)
        System properties at 0x704ba4e8:
          lun: 0x0
          target: 0x1
          class_prop: "atapi"
          class: "scsi"
704c7b00 sd, instance #2 (driver not attached)
        System properties at 0x704ba448:
          lun: 0x0
          target: 0x2
          class_prop: "atapi"
          class: "scsi"
...
```

Retrieving Driver Soft State Information

A common problem when debugging a driver is retrieving the "soft state" for a particular driver instance. The soft state is allocated with the `ddi_soft_state_zalloc(9F)` routine and obtained by drivers using `ddi_get_soft_state(9F)`. If you know the name of the *soft state pointer* (the first argument to `ddi_soft_state_init(9F)`), MDB lets you retrieve the soft state for a particular driver instance using the `::softstate` dcmd:

```
> bst_state::softstate 0x3
702b7578
```

In this case, `::softstate` is used to fetch the soft state for instance 3 of the `bst` sample driver. This pointer points to a `bst_soft` structure used by the driver to track state for this instance.

Detecting Kernel Memory Leaks

The `::findleaks` dcmd provides powerful and efficient detection of memory leaks in kernel crash dumps. The full set of kernel memory debugging features should be enabled for `::findleaks` to be effective. For more information see “`kmem_flags`” on page 306. Running `::findleaks` during driver development and testing can detect code which is leaking memory and wasting kernel resources. See “Debugging With the Kernel Memory Allocator” in the *Solaris Modular Debugger Guide* for a complete discussion of `::findleaks`.

Note - Use `::findleaks` to detect and eliminate kernel memory leaks caused by your code. Code that leaks kernel memory can render the system vulnerable to denial-of-service attacks.

Writing Debugger Commands

MDB provides a powerful API for implementing new debugger facilities that you can use to debug your driver. The *Solaris Modular Debugger Guide* explains the programming API in more detail, and the `SUNWmdbdm` package installs sample MDB source code in the directory `/usr/demo/mdb`. You can use MDB to automate lengthy debugging chores or help to validate that your driver is behaving properly. You can also package your MDB debugging modules with your driver product so that these facilities will be available to service personnel at a customer site.

APPENDIX **A**

Hardware Overview

This chapter discusses general issues about hardware capable of supporting the Solaris 8 operating environment. This includes issues related to the processor, bus architectures, and memory models supported by the Solaris 8 operating environment, various device issues, and the PROM used in Sun platforms.

Note - The information presented here is for informational purposes only and might be of help during driver debugging. However, the Solaris 8 DDI/DKI hides many of these implementation details from device drivers.

SPARC Processor Issues

This section describes a number of SPARC processor-specific topics including data alignment, byte ordering, register windows, and availability of floating-point instructions. For information on IA processor-specific topics, see “IA Processor Issues” on page 327.

SPARC Data Alignment

All quantities must be aligned on their natural boundaries. Using standard C data types:

- `short` integers are aligned on 16-bit boundaries.
- `int` integers are aligned on 32-bit boundaries.

- `long` integers are aligned on either 32-bit boundaries or 64-bit boundaries, depending on whether the data model of the kernel is 64-bit or 32-bit. For information on data models, see Appendix C.
- `long long` integers are aligned on 64-bit boundaries.

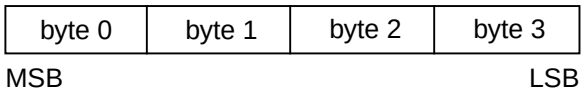
  Usually, the compiler handles alignment issues. However, driver writers are more likely to be concerned about alignment as they must use the proper data types to access their device. Since device registers are commonly accessed through a pointer reference, drivers must ensure that pointers are properly aligned when accessing the device.

SPARC Structure Member Alignment

Because of the data alignment restrictions imposed by the SPARC processor, C structures also have alignment requirements. Structure alignment requirements are imposed by the most strictly-aligned structure component. For example, a structure containing only characters has no alignment restrictions, while a structure containing a `long long` member must be constructed to guarantee that this member falls on a 64-bit boundary.

SPARC Byte Ordering

The SPARC processor uses *big-endian* byte ordering; in other words, the most significant byte (MSB) of an integer is stored at the lowest address of the integer.



SPARC Register Windows

SPARC processors use register windows. Each register window consists of 8 *in* registers, 8 *local* registers, and 8 *out* registers (which are the *in* registers of the next window). There are also 8 *global* registers. The number of register windows ranges from 2 to 32, depending on the processor implementation.

Because drivers are normally written in C, the compiler usually hides the fact that register windows are used. However, it might be necessary to use them when debugging the driver.

SPARC Floating-Point Operations

Drivers should not perform floating-point operations, as they are not supported in the kernel.

SPARC Multiply and Divide Instructions

The Version 7 SPARC processors do not have multiply or divide instructions. These instructions are emulated in software and should be avoided. Because a driver cannot determine whether it is running on a Version 7, Version 8, or Version 9 processor, intensive integer multiplication and division should be avoided if possible. Instead, use bitwise left and right shifts to multiply and divide by powers of two.

The *SPARC Architecture Manual, Version 9*, contains more specific information on the SPARC CPU. The *SPARC Compliance Definition*, Version 2.4, contains details of the SPARC V9 application binary interface (ABI). It describes the 32–bit SPARC V8 ABI and the 64–bit SPARC V9 ABI. You can obtain this document from SPARC International at `http://www.sparc.com`.

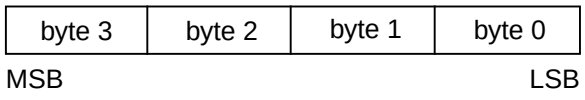
IA Processor Issues

Data types have no alignment restrictions on data types. However, extra memory cycles might be required for the IA processor to properly handle misaligned data transfers.

Drivers should not perform floating-point operations, as they are not supported in the kernel.

IA Byte Ordering

The IA processor uses *little-endian* byte ordering. The least significant byte (LSB) of an integer is stored at the lowest address of the integer.



IA Architecture Manuals

Intel Corporation publishes a number of books on the IA family of processors:

- Intel Corporation, *Pentium Pro Family Developer's Manual*, Volume 1, 1996. ISBN 1-55512-250–0.
- Intel Corporation, *Pentium Pro Family Developer's Manual*, Volume 2, 1996. ISBN 1-55512-260–4.
- Intel Corporation, *Pentium Pro Family Developer's Manual*, Volume 3, 1996. ISBN 1-55512-261–2.

Endianness

To achieve the goal of multiple platform, multiple instruction set architecture portability, host bus dependencies were removed from the drivers. The first dependency issue to be addressed was the endianness (or byte ordering) of the processor. For example, the IA processor family is little-endian while the SPARC architecture is big-endian.

Bus architectures display the same endianness types as processors. The PCI local bus, for example, is little-endian, the SBus is big-endian, the ISA bus is little-endian, and so on.

To maintain portability between processors and buses, DDI-compliant drivers must be endian neutral. Although drivers could conceivably manage their endianness by runtime checks or by preprocessor directives like `#ifdef _LITTLE_ENDIAN` or `_BIG_ENDIAN` statements in the source code, long-term maintenance would be troublesome. In some cases, the DDI framework performs the byte swapping using a software approach. In other cases, where byte swapping can be done by hardware (as in memory management unit (MMU) page-level swapping or by special machine instructions), the DDI framework will take advantage of the hardware features to improve performance.

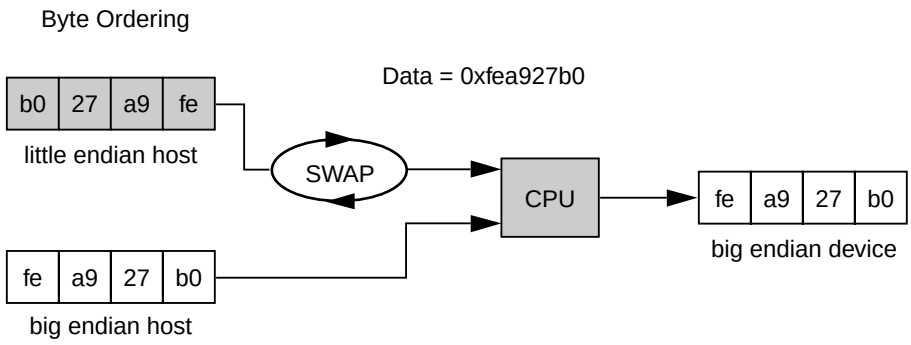


Figure A–1 Byte Ordering Host Bus Dependency

Along with being endian-neutral, portable drivers must also be independent from data ordering of the processor. Under most circumstances, data must be transferred in the sequence instructed by the driver. However, sometimes data can be merged, batched, or reordered to streamline the data transfer, as illustrated in Appendix A. For example, data merging can be applied to accelerate graphics display on frame buffers. Drivers have the option to advise the DDI framework to use other optimal data transfer mechanisms during the transfer.

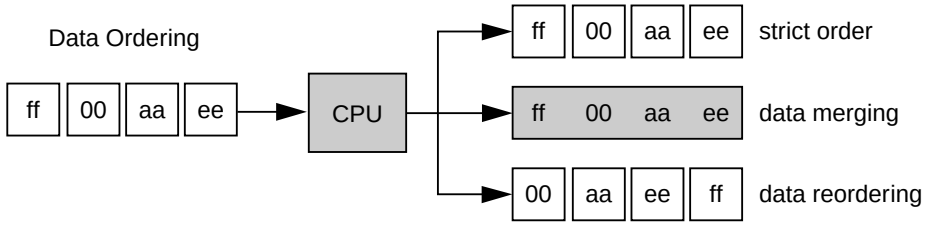


Figure A–2 Data Ordering Host Bus Dependency

Store Buffers

To improve performance, the CPU uses internal store buffers to temporarily store data. This can affect the synchronization of device I/O operations. Therefore, the driver needs to take explicit steps to make sure that writes to registers are completed at the proper time.

For example, when access to device space (such as registers or a frame buffer) is synchronized by a lock, the driver needs to check that the store to the device space

has actually completed before releasing the lock. Releasing the lock does not guarantee the flushing of I/O buffers.

To give another example, when acknowledging an interrupt, the driver usually sets or clears a bit in a device control register. The driver must ensure that the write to the control register has reached the device before the interrupt handler returns. Similarly, if the device requires a delay (the driver busy-waits) after writing a command to the control register, the driver must ensure that the write has reached the device before delaying.

If the device registers can be read without undesirable side effects, verification of a write can be as simple as reading the register immediately after writing to it. If that particular register cannot be read without undesirable side effects, another device register in the same register set can be used.

System Memory Model

The system memory model defines the semantics of memory operations such as *load* and *store* and specifies how the order in which these operations are issued by a processor is related to the order in which they reach memory. The memory model applies to both uniprocessors and shared-memory multiprocessors. Two memory models are supported: total store ordering (TSO) and partial store ordering (PSO).

Total Store Ordering (TSO)

TSO guarantees that the sequence in which store, FLUSH, and atomic load-store instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor.

Both IA and SPARC processors support TSO.

Partial Store Ordering (PSO)

PSO does not guarantee that the sequence in which store, FLUSH, and atomic load-store instructions appear in memory for a given processor is identical to the sequence in which they were issued by the processor. The processor can reorder the stores so that the sequence of stores to memory is not the same as the sequence of stores issued by the CPU.

SPARC processors support PSO; IA processors do not.

For SPARC processors, conformance between *issuing* order and *memory* order is provided by the system framework using the STBAR instruction: if two of the above

instructions are separated by an STBAR in the issuing order of a processor, or if they reference the same location, the memory order of the two instructions is the same as the issuing order. Note that enforcement of strong data ordering in DDI-compliant drivers is provided by the `ddi_regs_map_setup`(9F) interface. Compliant drivers cannot use the STBAR instruction directly.

See the *SPARC Architecture Manual, Version 9*, for more details on the SPARC memory model.

Bus Architectures

This section describes a number of bus-specific topics including device identification, device addressing, and interrupts.

Device Identification

Device identification is the process of determining which devices are present in the system.

Self-Identifying Devices

Some devices are self-identifying—the device itself provides information to the system so that it can identify the device driver that needs to be used. SBus and PCI local bus devices are examples of self-identifying devices. On SBus, the information is usually derived from a small Forth program stored in the FCode PROM on the device. PCI devices provide a configuration space containing device configuration information. See `sbus`(4) and `pci`(4) for more information.

Non-Self-Identifying Devices

Devices that do not provide information to the system to identify themselves are called non-self-identifying devices. Drivers for these devices must have a `probe`(9E) routine that determines whether the device is really present. In addition, information about the device must be provided in a hardware configuration file (see `driver.conf`(4)) so that the system can provide `probe`(9E) with the information it needs to contact the device.

Interrupts

Solaris supports both polling and vectored interrupts. The Solaris 8 DDI/DKI interrupt model is the same for both. See Chapter 7 for more information about interrupt handling.

Bus Specifics

This section covers addressing and device configuration issues specific to the buses that Solaris supports.

PCI Local Bus

The PCI local bus is a high-performance bus designed for high-speed data transfer. The PCI bus resides on the system board and is normally used as an interconnect mechanism between highly integrated peripheral components, peripheral add-on boards, and host processor or memory systems. The host processor, main memory, and the PCI bus itself are connected through a PCI host bridge, as shown in Figure A–3.

A tree structure of interconnected I/O buses is supported through a series of PCI bus bridges. Subordinate PCI bus bridges can be extended underneath the PCI host bridge to allow a single bus system to be expanded into a complex system with multiple secondary buses. PCI devices can be connected to one or more of these secondary buses. In addition, other bus bridges, such as SCSI or USB, can be connected.

Every PCI device has a unique vendor ID and device ID. Multiple devices of the same kind are further identified by their unique device numbers on the bus where they reside.

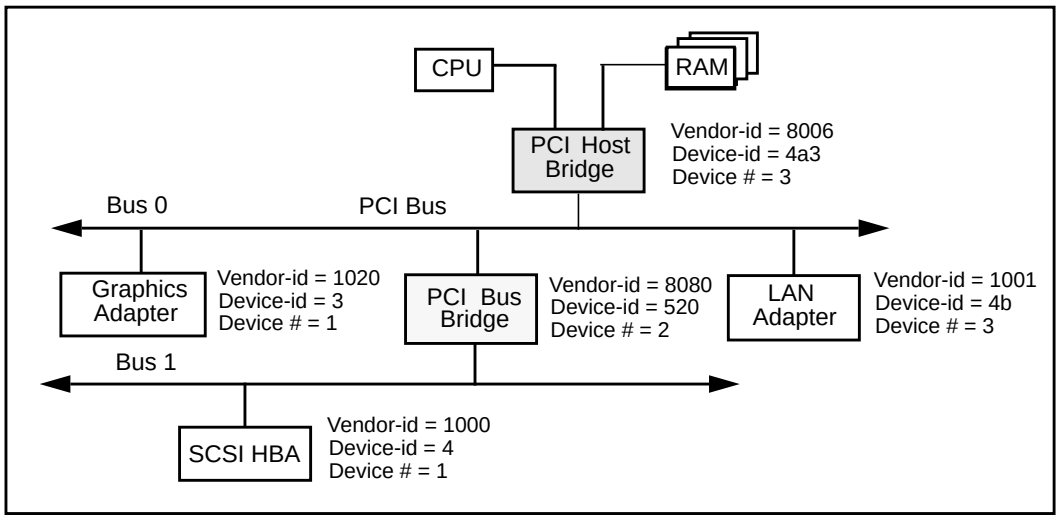


Figure A–3 Machine Block Diagram

The PCI host bridge provides an interconnect between the processor and peripheral components. Through the PCI host bridge, the processor can directly access main memory independent of other PCI bus masters. For example, while the CPU is fetching data from the cache controller in the host bridge, other PCI devices can also access the system memory through the host bridge. The advantage of this architecture lies in its separation of the I/O bus from the processor's host bus.

The PCI host bridge also provides data access mappings between the CPU and peripheral I/O devices. It maps every peripheral device to the host address domain so that the processor can access the device through programmed I/O. On the local bus side, the PCI host bridge maps the system memory to the PCI address domain so that the PCI device can access the host memory as a bus master. Figure A–3 shows the two address domains.

PCI Address Domain

The PCI address domain consists of three distinct address spaces: configuration, memory, and I/O space.

PCI Configuration Address Space

Configuration space is defined geographically; in other words, the location of a peripheral device is determined by its physical location within an interconnected tree of PCI bus bridges. A device is located by its *bus number* and *device* (*slot*) *number*. Each peripheral device contains a set of well-defined configuration registers in its PCI configuration space. The registers are used not only to identify devices but also to supply device configuration information to the configuration framework. For

example, base address registers in the device configuration space must be mapped before a device can respond to data access.

The method for generating configuration cycles is host dependent. In IA machines, special I/O ports are used. On other platforms, the PCI configuration space can be memory-mapped to certain address locations corresponding to the PCI host bridge in the host address domain. When a device configuration register is accessed by the processor, the request is routed to the PCI host bridge. The bridge then translates the access into proper configuration cycles on the bus.

PCI Configuration Base Address Registers

The PCI configuration space consists of up to six 32-bit base address registers for each device. These registers provide both size and data type information. System firmware assigns base addresses in the PCI address domain to these registers.

Each addressable region can be either memory or I/O space. The value contained in bit 0 of the base address register identifies the type. A value of 0 in bit 0 indicates a memory space and a value of 1 indicates an I/O space. Figure A–4 shows two base address registers: one for memory; the other for I/O types.

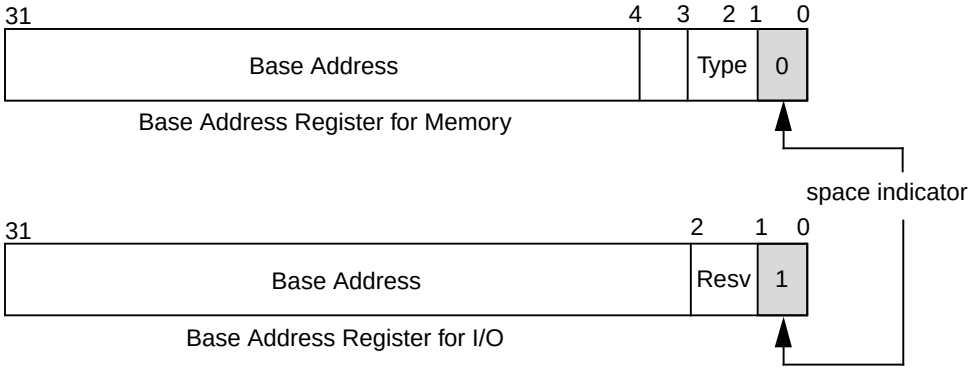


Figure A–4 Base Address Registers for Memory and I/O

PCI Memory Address Space

PCI supports both 32-bit and 64-bit addresses for memory space. System firmware assigns regions of memory space in the PCI address domain to PCI peripherals. The base address of a region is stored in the base address register of the device's PCI configuration space. The size of each region must be a power of two, and the assigned base address must be aligned on a boundary equal to the size of the region. Device addresses in memory space are *memory-mapped* into the host address domain

so that data access to any device can be performed by the processor's native load or store instructions.

PCI I/O Address Space

PCI supports 32-bit I/O space. I/O space can be accessed differently on different platforms. Processors with special I/O instructions, like the Intel processor family, access the I/O space with `in` and `out` instructions. Machines without special I/O instructions will map to the address locations corresponding to the PCI host bridge in the host address domain. When the processor accesses the memory-mapped addresses, an I/O request will be sent to the PCI host bridge. It then translates the addresses into I/O cycles and puts them on the PCI bus. Memory-mapped I/O is performed by the native load/store instructions of the processor.

PCI Hardware Configuration Files

Hardware configuration files should be unnecessary for PCI local bus devices. However, on some occasions drivers for PCI devices need to use hardware configuration files to augment the driver private information. See `driver.conf`(4) and `pci`(4) for further details.

SBus

Typical SBus systems consist of a motherboard (containing the CPU and SBus interface logic), a number of SBus devices on the motherboard itself, and a number of SBus expansion slots. An SBus can also be connected to other types of buses through an appropriate bus bridge.

The SBus is geographically addressed; each SBus slot exists at a fixed physical address in the system. An SBus card has a different address, depending on which slot it is plugged into. Moving an SBus device to a new slot causes the system to treat it as a new device.

The SBus uses polling interrupts. When an SBus device interrupts, the system only knows which of several devices might have issued the interrupt. The system interrupt handler must ask the driver for each device whether it is responsible for the interrupt.

SBus Physical Address Space

Table A–1 shows the physical address space layout of the Sun Ultra 2 computer. A physical address on the Ultra 2 consists of 41 bits. The 41-bit physical address space is further broken down into multiple 33-bit address spaces identified by `PA(40:33)`.

TABLE A–1 Device Physical Space in the Ultra 2

PA(40:33)	33-bit Space	Usage
0x0	0x000000000 - 0x07FFFFFFF	2GB Main memory
0x80 – 0xDF	Reserved on Ultra 2	Reserved on Ultra 2
0xE0	Processor 0	Processor 0
0xE1	Processor 1	Processor 1
0xE2 – 0xFD	Reserved on Ultra 2	Reserved on Ultra 2
0xFE	0x000000000 - 0x1FFFFFFFF	UPA Slave (FFB)
0xFF	0x000000000 - 0x0FFFFFFFF	System I/O space
	0x100000000 - 0x10FFFFFFF	SBus Slot 0
	0x110000000 - 0x11FFFFFFF	SBus Slot 1
	0x120000000 - 0x12FFFFFFF	SBus Slot 2
	0x130000000 - 0x13FFFFFFF	SBus Slot 3
	0x1D0000000 - 0x1DFFFFFFF	SBus Slot D
	0x1E0000000 - 0x1EFFFFFFF	SBus Slot E
	0x1F0000000 - 0x1FFFFFFFF	SBus Slot F

Physical SBus Addresses

The SBus has 32 address bits, as described in the *SBus Specification*. Table A–2 describes how the Ultra 2 uses the address bits.

TABLE A–2 Ultra 2 SBus Address Bits

Bits	Description
0 - 27	These bits are the SBus address lines used by an SBus card to address the contents of the card.
28 - 31	Used by the CPU to select one of the SBus slots. These bits generate the SlaveSelect lines.

This addressing scheme yields the Ultra 2 addresses shown in Table A–1. Other implementations might use a different number of address bits.

The Ultra 2 has seven SBus slots, four of which are physical. Slots 0 through 3 are available for SBus cards. Slots 4-12 are reserved. The slots are used in the following way:

- Slots 0–3 are physical slots that have DMA-master capability.
- Slots D, E, and F are not actual physical slots, but refer to the onboard direct memory access (DMA), SCSI, Ethernet, and audio controllers. For convenience, these are viewed as being plugged into slots D, E, and F.

Note - Some SBus slots are slave-only slots. Drivers that require DMA capability should use `ddi_slaveonly`(9F) to determine if their device is in a DMA-capable slot. For an example of this function, see "`attach(9E)`" on page 71.

SBus Hardware Configuration Files

Hardware configuration files are normally unnecessary for SBus devices. However, on some occasions, drivers for SBus devices need to use hardware configuration files to augment the information provided by the SBus card. See `driver.conf`(4) and `sbus`(4) for further details.

ISA Bus

The following sections describe the ISA bus.

ISA Bus Memory and I/O Space

Two address spaces are provided: memory address space and I/O address space. Depending on the device, registers can appear in one or both of these address spaces,

and are self-identifying. Table A–3 shows the registers for memory and I/O address spaces in the ISA bus.

TABLE A–3 ISA Bus Address Space

ISA Space Name	Address Size	Data Transfer Size	Physical Address Range
Main memory	24	16	0x0-0xffffff
I/O	—	8/16	0x0-0xfff

Hardware Configuration Files

In the Solaris 8 operating environment, the use of hardware configuration files to provide arguments to `probe`(9E) on IA platforms is highly discouraged, since probes can lead to system hangs and resets. Exact device configuration information is maintained by the booting system and is passed to the `probe(9E)` function.

Bootable (Realmode) Drivers

A separate realmode driver might need to be developed for the booting system. See the *Realmode Drivers* white paper in the Driver Development Site at `http://soldc.sun.com/developer/support/driver` for information on realmode drivers. Hardware configuration files may be needed on some occasions to augment the information provided by the booting system. See `driver.conf`(4) and `isa`(4) for further details.

Device Issues

This section describes issues with special devices.

Timing-Critical Sections

While most driver operations can be performed without mechanisms for synchronization and protection beyond those provided by the locking primitives, some devices require that a sequence of events occur in order without interruption.

In conjunction with the locking primitives, the function `ddi_enter_critical`(9F) asks the system to guarantee, to the best of its ability, that the current thread will neither be pre-empted nor interrupted. This stays in effect until a closing call to `ddi_exit_critical`(9F) is made. See `ddi_enter_critical`(9F) for details.

Delays

Many chips specify that they can be accessed only at specified intervals. For example, the Zilog Z8530 SCC has a "write recovery time" of 1.6 microseconds. This means that a delay must be enforced with `drv_usecwait`(9F) when writing characters with an 8530. In some instances, it is unclear from the specifications what delays are needed; in such cases, they must be determined empirically.

Internal Sequencing Logic

Devices with internal sequencing logic map multiple internal registers to the same external address. There are various kinds of internal sequencing logic:

- The Intel 8251A and the Signetics 2651 alternate the same external register between *two* internal mode registers. Writing to the first internal register is accomplished by writing to the external register. This write, however, has the side effect of setting up the sequencing logic in the chip so that the next read / write operation refers to the second internal register.
- The NEC PD7201 PCC has multiple internal data registers. To write a byte into a particular register, two steps must be performed. The first step is to write into register zero the number of the register into which the following byte of data will go. The data is then written to the specified data register. The sequencing logic automatically sets up the chip so that the next byte sent will go into data register zero.
- The AMD 9513 timer has a data pointer register that points at the data register into which a data byte will go. When sending a byte to the data register, the pointer is incremented. *The current value of the pointer register cannot be read.*

Interrupt Issues

The following are some common interrupt-related issues:

- A controller interrupt does *not* necessarily indicate that *both* the controller *and* one of its slave devices are ready. For some controllers, an interrupt can indicate that either the controller is ready or one of its devices is ready, but not both.
- Not all devices power up with interrupts disabled and can begin interrupting at any time.

- Some devices do not provide a way to determine that the board has generated an interrupt.
- Not all interrupting boards shut off interrupts when told to do so or after a bus reset.

PROM on SPARC Machines

Some platforms have a PROM monitor that provides support for debugging a device without an operating system. This section describes how to use the PROM on SPARC machines to map device registers so that they can be accessed. Usually, the device can be exercised enough with PROM commands to determine if the device is working correctly.

See `boot`(1M) for a description of the IA boot subsystem.

The PROM has several purposes; it serves to:

- Bring the machine up from power on, or from a hard reset PROM `reset` command
- Provide an interactive tool for examining and setting memory, device registers, and memory mappings
- Boot the Solaris system or the kernel debugger `kadb`(1M)

  Simply powering up the computer and attempting to use its PROM to examine device registers can fail. While the device might be correctly installed, those mappings are Solaris operating environment specific and do not become active until the Solaris kernel is booted. Upon power up, the PROM maps only essential system devices, such as the keyboard.
- Take a system crash dump using the `sync` command

Open Boot PROM 3

For complete documentation on the Open Boot PROM, see the *Open Boot PROM Toolkit User's Guide* and `monitor`(1M). The examples in this section refer to a Sun-4u architecture; other architectures might require different commands to perform actions.

Note - The Open Boot PROM is currently used on Sun machines with an SBus or UPA/PCI. The Open Boot PROM uses an "`ok`" prompt. On older machines, it might be necessary to type '`n`' to get the "`ok`" prompt.

If the PROM is in *secure mode* (the `security-mode` parameter is not set to *none*), the PROM password might be required (set in the `security-password` parameter).

The `printenv` command displays all parameters and their values.

Help

Help is available with the `help` command.

History

EMACS-style command-line history is available. Use Control-N (next) and Control-P (previous) to traverse the history list.

Forth Commands

The Open Boot PROM uses the Forth programming language. This is a stack-based language; arguments must be pushed on the stack before running the correct command (called a *word*), and the result is left on the stack.

To place a number on the stack, type its value.

```
ok 57
ok 68
```

To add the two top values on the stack, use the + operator.

```
ok +
```

The result remains on the stack. The stack is shown with the `.s` *word*.

```
ok .s
bf
```

The default base is hexadecimal. The `hex` and `decimal` *words* can be used to switch bases.

```
ok decimal
ok .s
191
```

See the *Forth User's Guide* for more information.

Walking the PROMs Device Tree

The commands `pwd`, `cd`, and `ls` walk the PROM device tree to get to the device. The `cd` command must be used to establish a position in the tree before `pwd` will work. This example is from an Ultra 1 workstation with a `cgsix` frame buffer on an SBus.

```
ok cd /
```

To see the devices attached to the current node in the tree, use `ls`.

```
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
```

The full node name can be used:

```
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
```

Rather than using the full node name in the previous example, you could have used an abbreviation. The abbreviated command line entry looks like this:

```
ok cd sbus
```

The name is actually `device@slot,offset` (for SBus devices). The `cgsix` device is in slot 2 and starts at offset 0. If an SBus device is displayed in this tree, the device has been recognized by the PROM.

The `.properties` command displays the PROM properties of a device. These can be examined to determine which properties the device exports (this is useful later to ensure that the driver is looking for the correct hardware properties). These are the same properties that can be retrieved with `ddi_getprop`(9F).

```
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
...
```

The *reg* property defines an array of register description structures containing the following fields:

```
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 */
```

For the `cgsix` example, the address is 0.

Mapping the Device

To test the device, it must be mapped into memory. The PROM can then be used to verify proper operation of the device by using data-transfer commands to transfer bytes, words, and long words. If the device can be operated from the PROM, even in a limited way, the driver should also be able to operate the device.

To set up the device for initial testing, perform the following steps:

1. Determine the SBus slot number the device is in. In this example, the *cgsix* device is located in slot 2.
2. Determine the offset within the physical address space used by the device.

   The offset used is specific to the device. In the `cgsix` example, the video memory happens to start at an offset of 0x800000.
3. Use the `select-dev` *word* to select the sbus device and the `map-in` *word* to map the device in.

   The select-dev word takes a string of the device path as its argument. The `map-in` *word* takes an *offset*, a *slot number*, and a *size* as arguments to map. Like the offset, the size of the byte transfer is specific to the device. In the `cgsix` example, the size is set to 0x100000 bytes.

   In the following code example, the sbus path is displayed as an argument to the `select-dev` word, and the offset, slot number, and size values for the frame buffer are displayed as arguments to the `map-in` word. Notice that there should be a space between the opening quote and / in the `select-dev` argument. The virtual address to use remains on top of the stack. The stack is shown using the `.s` word. It can be assigned a name with the `constant` operation.

```
ok "/sbus@1f,0" select-dev
   ok 800000 2 100000 map-in
   ok .s
   ffe98000
   ok constant fb
```

Reading and Writing

The PROM provides a variety of 8-bit, 16-bit, and 32-bit operations. In general, a `c` (character) prefix indicates an 8-bit (one byte) operation; a `w` (word) prefix indicates a 16-bit (two byte) operation; and an `L` (longword) prefix indicates a 32-bit (four byte) operation.

A suffix of ! is used to indicate a write operation. The write operation takes the first two items off the stack; the first item is the address, and the second item is the value.

```
ok 55 ffe98000 c!
```

A suffix of @ is used to indicate a read operation. The read operation takes one argument (the address) off the stack.

```
ok ffe98000 c@
ok .s
55
```

A suffix of ? is used to display the value, without affecting the stack.

```
ok ffe98000 c?
55
```

Be careful when trying to query the device. If the mappings are not set up correctly, trying to read or write could cause errors. Special words are provided to handle these cases. `cprobe`, `wprobe`, and `lprobe`, for example, read from the given address but return zero if the location does not respond, or nonzero if it does.

```
ok fffa4000 c@
Data Access Error

ok fffa4000 cprobe
ok .s
0

ok ffe98000 cprobe
ok .s
0 ffffffffffffffff
```

A region of memory can be shown with the `dump` word. This takes an *address* and a *length*, and displays the contents of the memory region in bytes.

In the following example, the `fill` word is used to fill video memory with a pattern. `fill` takes the address, the number of bytes to fill, and the byte to use (there is also a `wfill` and an `Lfill` for words and longwords). This causes the *cgsix* to display simple patterns based on the byte passed.

```
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 fill
ok fb 5000 f9 fill
```

APPENDIX **B**

Summary of Solaris 8 DDI/DKI Services

Introduction

This chapter discusses the interfaces provided by the Solaris 8 DDI/DKI. These descriptions should not be considered complete or definitive, nor do they provide a thorough guide to usage. The descriptions are intended to describe what the functions do in general terms. See *man pages section 9F: DDI and DKI Kernel Functions* for more detailed information. The categories are:

This appendix does not discuss STREAMS interfaces; to learn more about network drivers, see the *STREAMS Programming Guide*.

Module Functions

TABLE B–1 Module Functions

Function Name	Description
mod_install	add a loadable module
mod_remove	remove a loadable module
mod_info	query a loadable module

Device Information Tree Node (`dev_info_t`) Functions

TABLE B–2 Device Information Tree Node (`dev_info_t`) Functions

Function Name	Description
ddi_driver_name	return normalized driver name
ddi_node_name	return the devinfo node name

TABLE B–2 Device Information Tree Node (`dev_info_t`) Functions *(continued)*

Function Name	Description
ddi_get_name	return driver binding name
ddi_binding_name	return driver binding name
ddi_dev_is_sid	tell whether a device is self-identifying
ddi_get_instance	get device instance number
ddi_get_devstate	check device state
ddi_get_parent	find the parent of a device information structure
ddi_root_node	get the root of the dev_info tree

Device (`dev_t`) Functions

TABLE B–3 Device (`dev_t`) Functions

Function Name	Description
ddi_create_minor_node	create a minor node for a device
ddi_remove_minor_node	remove a minor mode for a device
makedevice	make device number from major and minor numbers
getmajor	get major device number
getminor	get minor device number
ddi_getiminor	get kernel internal minor number from an external dev_t

Property Functions

TABLE B–4 Property Functions

Function Name	Description
ddi_prop_exists	check for the existence of a property
ddi_prop_get_int	lookup integer property
ddi_prop_lookup_byte_array	look up byte array property
ddi_prop_lookup_int_array	look up integer array property
ddi_prop_lookup_string	look up string property
ddi_prop_lookup_string_array	look up string array property
ddi_prop_free	free resources consumed by property lookup
ddi_prop_undefine	hide a property of a device
ddi_prop_update_byte_array	create or update byte array property
ddi_prop_update_int	create or update integer property
ddi_prop_update_int_array	create or update integer array property
ddi_prop_update_string	create or update string property
ddi_prop_update_string_array	create or update string array property
ddi_prop_remove	remove a property of a device
ddi_prop_remove_all	remove all properties of a device

TABLE B–5 Deprecated Property Functions

Deprecated Functions	Replacements
ddi_getprop	ddi_prop_get_int
ddi_getproplen	see ddi_prop_lookup
ddi_getlongprop	see ddi_prop_lookup

TABLE B–5 Deprecated Property Functions *(continued)*

Deprecated Functions	Replacements
ddi_getlongprop_buf	see ddi_prop_lookup
ddi_prop_create	see ddi_prop_lookup
ddi_prop_modify	see ddi_prop_lookup
ddi_prop_op	see ddi_prop_lookup

Device Software State Functions

TABLE B–6 Device Software State Functions

Function Name	Description
ddi_get_driver_private	get the address of the device's private data area
ddi_set_driver_private	set the address of the device's private data area
ddi_soft_state_init	initialize driver soft state structure
ddi_soft_state_fini	destroy driver soft state structure
ddi_soft_state_zalloc	allocate instance soft state structure
ddi_soft_state_free	free instance soft state structure
ddi_get_soft_state	get pointer to instance soft state structure

Memory Allocation and Deallocation Functions

TABLE B–7 Memory Allocation and Deallocation Functions

Function Name	Description
kmem_alloc	allocate kernel memory
kmem_free	free kernel memory
kmem_zalloc	allocate zero-filled kernel memory

These functions allocate and free memory intended to be used for DMA. See “Direct Memory Access (DMA) Functions” on page 362.

TABLE B–8

Function Name	Description
ddi_dma_mem_alloc	allocate memory for DMA transfer
ddi_dma_mem_free	free previously allocated DMA memory

These functions allocate and free memory intended to be exported to user space. See “User Space Access Functions” on page 364.

TABLE B–9

Function Name	Description
ddi_umem_alloc	allocate page-aligned kernel memory
ddi_umem_free	free page-aligned kernel memory

TABLE B–10 Deprecated Memory Allocation and Deallocation Functions

Deprecated Functions	Replacement
ddi_iopb_alloc	ddi_dma_mem_alloc
ddi_iopb_free	ddi_dma_mem_free

TABLE B–10 Deprecated Memory Allocation and Deallocation Functions *(continued)*

Deprecated Functions	Replacement
ddi_mem_alloc	ddi_dma_mem_alloc
ddi_mem_free	ddi_dma_mem_free

Kernel Thread Control and Synchronization Functions

TABLE B–11 Kernel Thread Control and Synchronization Functions

Function Name	Description
mutex_init	initialize mutual exclusion lock
mutex_destroy	destroy mutual exclusion lock
mutex_enter	acquire mutual exclusion lock
mutex_tryenter	attempt to acquire mutual exclusion lock without waiting
mutex_exit	release mutual exclusion lock
mutex_owned	determine if current thread is holding mutual exclusion lock
cv_init	allocate a condition variable
cv_destroy	free an allocated condition variable
cv_wait	await an event
cv_wait_sig	await an event or signal
cv_timedwait	await an event with timeout
cv_timedwait_sig	await an event or signal with timeout

TABLE B–11 Kernel Thread Control and Synchronization Functions *(continued)*

Function Name	Description
cv_signal	wakeup one waiting thread
cv_broadcast	wakeup all waiting threads
rw_init	initialize a readers / writer lock
rw_destroy	destroy a readers / writer lock
rw_enter	acquire a readers / writer lock
rw_tryenter	attempt to acquire a readers / writer lock without waiting
rw_tryupgrade	attempt to upgrade readers / writer lock holding from reader to writer
rw_downgrade	downgrade a readers / writer lock holding from writer to reader
rw_exit	release a readers / writer lock
rw_read_locked	determine whether readers / writer lock is held for read or write
sema_init	initialize a semaphore
sema_destroy	destroy a semaphore
sema_p	decrement semaphore and possibly block
sema_tryp	attempt to decrement semaphore, but do not block
sema_p_sig	decrement semaphore, but do not block if signal is pending
sema_v	increment semaphore and possibly unblock waiter
ddi_enter_critical	enter a critical region of control
ddi_exit_critical	exit a critical region of control

TABLE B–11 Kernel Thread Control and Synchronization Functions *(continued)*

Interrupt Functions

TABLE B–12 Interrupt Functions

Function Name	Description
ddi_dev_nintrs	return the number of interrupt specifications a device has
ddi_intr_hilevel	indicate interrupt type
ddi_get_iblock_cookie	get a hardware interrupt block cookie
ddi_add_intr	register a hardware interrupt handler
ddi_remove_intr	unregister a hardware interrupt handler
ddi_get_soft_iblock_cookie	get a software interrupt block cookie
ddi_add_softintr	register a software interrupt handler
ddi_trigger_softintr	trigger a software interrupt
ddi_remove_softintr	unregister a software interrupt handler

Programmed I/O Functions

TABLE B–13 Programmed I/O Functions

Function Name	Description
ddi_dev_nregs	return the number of register sets a device has
ddi_dev_regsize	return the size of a device's register

TABLE B–13 Programmed I/O Functions *(continued)*

Function Name	Description
ddi_regs_map_setup	set up a mapping for a register address space
ddi_regs_map_free	free a previously mapped register address space
ddi_device_copy	copy data from one device register to another device register
ddi_device_zero	zero fill the device
ddi_check_acc_handle	check data access handle
ddi_get8	read 8–bit data from mapped memory, device register or DMA memory
ddi_get16	read 16–bit data from mapped memory, device register or DMA memory
ddi_get32	read 32–bit data from mapped memory, device register or DMA memory
ddi_get64	read 64–bit data from mapped memory, device register or DMA memory
ddi_put8	write 8–bit data to mapped memory, device register or DMA memory
ddi_put16	write 16–bit data to mapped memory, device register or DMA memory
ddi_put32	write 32–bit data to mapped memory, device register or DMA memory
ddi_put64	write 64–bit data to mapped memory, device register or DMA memory
ddi_rep_get8	read multiple 8–bit data from mapped memory, device register or DMA memory
ddi_rep_get16	read multiple 16–bit data from mapped memory, device register or DMA memory
ddi_rep_get32	read multiple 32–bit data from mapped memory, device register or DMA memory

TABLE B–13 Programmed I/O Functions *(continued)*

Function Name	Description
ddi_rep_get64	read multiple 64–bit data from mapped memory, device register or DMA memory
ddi_rep_put8	write multiple 8–bit data to mapped memory, device register or DMA memory
ddi_rep_put16	write multiple 16–bit data to mapped memory, device register or DMA memory
ddi_rep_put32	write multiple 32–bit data to mapped memory, device register or DMA memory
ddi_rep_put64	write multiple 64–bit data to mapped memory, device register or DMA memory
ddi_peek8	cautiously read an 8–bit value from a location
ddi_peek16	cautiously read a 16–bit value from a location
ddi_peek32	cautiously read a 32–bit value from a location
ddi_peek64	cautiously read a 64–bit value from a location
ddi_poke8	cautiously write an 8–bit value to a location
ddi_poke16	cautiously write a 16–bit value to a location
ddi_poke32	cautiously write a 32–bit value to a location
ddi_poke64	cautiously write a 64–bit value to a location

The general programmed I/O functions above can always be used rather than the `mem`, `io`, and `pci_config` functions below. However, the below functions may be used as alternatives in cases where the type of access is known at compile time.

TABLE B–14 Alternate Access Mechanisms

Function Name	Description
ddi_io_get8	read 8-bit data from mapped device register in I/O space
ddi_io_get16	read 16-bit data from mapped device register in I/O space
ddi_io_get32	read 32-bit data from mapped device register in I/O space
ddi_io_put8	write 8-bit data to mapped device register in I/O space
ddi_io_put16	write 16-bit data to mapped device register in I/O space
ddi_io_put32	write 32-bit data to mapped device register in I/O space
ddi_io_rep_get8	read multiple 8-bit data from mapped device register in I/O space
ddi_io_rep_get16	read multiple 16-bit data from mapped device register in I/O space
ddi_io_rep_get32	read multiple 32-bit data from mapped device register in I/O space
ddi_io_rep_put8	write multiple 8-bit data to mapped device register in I/O space
ddi_io_rep_put16	write multiple 16-bit data to mapped device register in I/O space
ddi_io_rep_put32	write multiple 32-bit data to mapped device register in I/O space
ddi_mem_get8	read 8-bit data from mapped device in memory space or DMA memory
ddi_mem_get16	read 16-bit data from mapped device in memory space or DMA memory

TABLE B–14 Alternate Access Mechanisms *(continued)*

Function Name	Description
ddi_mem_get32	read 32-bit data from mapped device in memory space or DMA memory
ddi_mem_get64	read 64-bit data from mapped device in memory space or DMA memory
ddi_mem_put8	write 8-bit data to mapped device in memory space or DMA memory
ddi_mem_put16	write 16-bit data to mapped device in memory space or DMA memory
ddi_mem_put32	write 32-bit data to mapped device in memory space or DMA memory
ddi_mem_put64	write 64-bit data to mapped device in memory space or DMA memory
ddi_mem_rep_get8	read multiple 8-bit data from mapped device in memory space or DMA memory
ddi_mem_rep_get16	read multiple 16-bit data from mapped device in memory space or DMA memory
ddi_mem_rep_get32	read multiple 32-bit data from mapped device in memory space or DMA memory
ddi_mem_rep_get64	read multiple 64-bit data from mapped device in memory space or DMA memory
ddi_mem_rep_put8	write multiple 8-bit data to mapped device in memory space or DMA memory
ddi_mem_rep_put16	write multiple 16-bit data to mapped device in memory space or DMA memory
ddi_mem_rep_put32	write multiple 32-bit data to mapped device in memory space or DMA memory
ddi_mem_rep_put64	write multiple 64-bit data to mapped device in memory space or DMA memory
pci_config_setup	setup access to PCI Local Bus Configuration space

TABLE B–14 Alternate Access Mechanisms *(continued)*

Function Name	Description
pci_config_teardown	tear down access to PCI Local Bus Configuration space
pci_config_get8	read 8-bit data from the PCI Local Bus Configuration space
pci_config_get16	read 16-bit data from the PCI Local Bus Configuration space
pci_config_get32	read 32-bit data from the PCI Local Bus Configuration space
pci_config_get64	read 64-bit data from the PCI Local Bus Configuration space
pci_config_put8	write 8-bit data to the PCI Local Bus Configuration space
pci_config_put16	write 16-bit data to the PCI Local Bus Configuration space
pci_config_put32	write 32-bit data to the PCI Local Bus Configuration space
pci_config_put64	write 64-bit data to the PCI Local Bus Configuration space

TABLE B–15 Deprecated Programmed I/O Functions

Deprecated Function	Replacement
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

TABLE B–15 Deprecated Programmed I/O Functions *(continued)*

Deprecated Function	Replacement
ddi_io_getw	ddi_io_get16
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

TABLE B–15 Deprecated Programmed I/O Functions *(continued)*

Deprecated Function	Replacement
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
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

TABLE B–15 Deprecated Programmed I/O Functions *(continued)*

Deprecated Function	Replacement
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
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

Direct Memory Access (DMA) Functions

TABLE B–16 Direct Memory Access (DMA) Functions

Function Name	Description
ddi_dma_alloc_handle	allocate a DMA handle
ddi_dma_free_handle	free DMA handle
ddi_dma_mem_alloc	allocate memory for DMA transfer
ddi_dma_mem_free	free previously allocated DMA memory
ddi_dma_addr_bind_handle	bind an address to a DMA handle
ddi_dma_buf_bind_handle	bind a system buffer to a DMA handle
ddi_dma_unbind_handle	unbind the address in a DMA handle
ddi_dma_nextcookie	retrieve subsequent DMA cookie
ddi_dma_getwin	activate a new DMA window
ddi_dma_numwin	retrieve number of DMA windows
ddi_dma_sync	synchronize CPU and I/O views of memory
ddi_check_dma_handle	check DMA handle
ddi_dma_set_sbus64	allow 64-bit transfers on SBus
ddi_slaveonly	tell if a device is installed in a slave access only location
ddi_iomin	find minimum alignment and transfer size for DMA
ddi_dma_burstsizes	find out the allowed burst sizes for a DMA mapping
ddi_dma_devalign	find DMA mapping alignment and minimum transfer size
ddi_dmae_alloc	acquire a DMA channel

TABLE B–16 Direct Memory Access (DMA) Functions *(continued)*

Function Name	Description
ddi_dmae_release	release a DMA channel
ddi_dmae_getattr	get DMA engine attributes
ddi_dmae_prog	program a DMA channel
ddi_dmae_stop	terminate DMA engine operation
ddi_dmae_disable	disable a DMA channel
ddi_dmae_enable	enable a DMA channel
ddi_dmae_getcnt	get DMA engine count remaining
ddi_dmae_1stparty	configure DMA channel cascade mode
ddi_dma_coff	convert a DMA cookie to an offset within a DMA handle

TABLE B–17 Deprecated Direct Memory Access (DMA) Functions

Deprecated Function	Replacement
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

TABLE B–17 Deprecated Direct Memory Access (DMA) Functions *(continued)*

Deprecated Function	Replacement
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
ddi_mem_free	ddi_dma_mem_free
hat_getkpfnum	ddi_dma_addr_bind_handle, ddi_dma_buf_bind_handle, ddi_dma_nextcookie

User Space Access Functions

TABLE B–18 User Space Access Functions

Function Name	Description
ddi_copyin	copy data to a driver buffer
ddi_copyout	copy data from a driver
uiomove	copy kernel data using uio structure
ureadc	add character to a uio structure
uwritec	remove a character from a uio structure
ddi_getiminor	get kernel internal minor number from an external dev_t
ddi_model_convert_from	determine data model type mismatch

TABLE B–18 User Space Access Functions *(continued)*

Function Name	Description
IOC_CONVERT_FROM	determine if there is a need to translate M_IOCTL contents
STRUCT_DECL	establish handle to application data in possibly differing data model
STRUCT_HANDLE	establish handle to application data in possibly differing data model
STRUCT_INIT	establish handle to application data in possibly differing data model
STRUCT_SET_HANDLE	establish handle to application data in possibly differing data model
SIZEOF_PTR	return size of pointer in specified data model
SIZEOF_STRUCT	return size of structure in specified data model
STRUCT_SIZE	return size of structure in application data model
STRUCT_BUF	return a pointer to the native mode instance of the structure
STRUCT_FADDR	return a pointer to the specified field of a structure
STRUCT_FGET	return specified field of a structure in application data model
STRUCT_FGETP	return specified pointer field of a structure in application data model
STRUCT_FSET	set specified field of a structure in application data model
STRUCT_FSETP	set specified pointer field of a structure in application data model

TABLE B–19 Deprecated User Space Access Functions

Deprecated Functions	Replacement
copyin	ddi_copyin
copyout	ddi_copyout

User Process Event Functions

TABLE B–20 User Process Event Functions

Function Name	Description
pollwakeup	inform a process that an event has occurred
proc_ref	get a handle on a process to signal
proc_unref	release a handle on a process to signal
proc_signal	send a signal to a process

User Process Information Functions

TABLE B–21 User Process Information Functions

Function Name	Description
ddi_get_cred	returns a pointer to the credential structure of the caller
drv_priv	determine process credentials privilege
ddi_get_pid	return the process ID

TABLE B–22 Deprecated User Process Information Functions

Deprecated Functions	Replacement
drv_getparm	ddi_get_pid, ddi_get_cred

User Application Kernel and Device Access Functions

TABLE B–23 User Application Kernel and Device Access Functions

Function Name	Description
ddi_dev_nregs	return the number of register sets a device has
ddi_dev_regsize	return the size of a device's register
ddi_devmap_segmap	set up a user mapping to device memory
devmap_setup	set up a user mapping to device memory
devmap_devmem_setup	export device memory to user space
devmap_load	control validation of memory address translations
devmap_unload	control validation of memory address translations
devmap_do_ctxmgt	perform device context switching on a mapping
devmap_set_ctx_timeout	set the timeout value for the context management callback
devmap_default_access	default driver memory access function
ddi_umem_alloc	allocate page-aligned kernel memory
ddi_umem_free	free page-aligned kernel memory
ddi_umem_lock	lock memory pages
ddi_umem_unlock	unlock memory pages

TABLE B–23 User Application Kernel and Device Access Functions *(continued)*

Function Name	Description
ddi_umem_iosetup	setup I/O requests to application memory
devmap_umem_setup	export kernel memory to user space
ddi_model_convert_from	determine data model type mismatch

TABLE B–24 Deprecated User Application Kernel and Device Access Functions

Deprecated Functions	Replacement
ddi_mapdev	devmap_setup
ddi_mapdev_intercept	devmap_load
ddi_mapdev_nointercept	devmap_unload
ddi_mapdev_set_device_acc_attr	see devmap(9e)
ddi_segmap	see devmap(9e)
ddi_segmap_setup	devmap_setup
hat_getkpfnum	see devmap(9e)
ddi_mmap_get_model	see devmap(9e)

Time-Related Functions

TABLE B–25 Time-Related Functions

Function Name	Description
ddi_get_lbolt	returns clock ticks since reboot
ddi_get_time	return the current time in seconds

TABLE B–25 Time-Related Functions *(continued)*

Function Name	Description
delay	delay execution for a specified number of clock ticks
drv_hztousec	convert clock ticks to microseconds
drv_usectohz	convert microseconds to clock ticks
drv_usecwait	busy-wait for specified interval
timeout	execute a function after a specified length of time
untimeout	cancel previous timeout function call
drv_getparm	ddi_get_lbolt, ddi_get_time

TABLE B–26 Deprecated Time-Related Functions

Deprecated Function	Replacement
drv_getparm	ddi_get_lbolt, ddi_get_time

Power Management Functions

TABLE B–27 Power Management Functions

Function Name	Description
pm_busy_component	mark component as busy
pm_idle_component	mark component as idle
pm_raise_power	raise the power level of a component
pm_lower_power	lower the power level of a component
pm_power_has_changed	notify power management framework of autonomous power level change

TABLE B–27 Power Management Functions *(continued)*

Function Name	Description
pm_trans_check	device power cycle advisory check
pci_report_pmcap	report power management capability of a PCI device
ddi_dev_is_needed	inform the system that a device's component is required

TABLE B–28 Deprecated Power Management Functions

Function Name	Description
pm_create_components	see pm-components(9)
pm_destroy_components	see pm-components(9)
pm_get_normal_power	see pm-components(9)
pm_set_normal_power	

Kernel Statistics Functions

TABLE B–29 Kernel Statistics Functions

Function Name	Description
kstat_create	create and initialize a new kstat
kstat_delete	remove a kstat from the system
kstat_install	add a fully initialized kstat to the system
kstat_named_init	initialize a named kstat
kstat_runq_back_to_waitq	record transaction migration from run queue to wait queue

TABLE B–29 Kernel Statistics Functions *(continued)*

Function Name	Description
kstat_runq_enter	record transaction add to run queue
kstat_runq_exit	record transaction removal from run queue
kstat_waitq_enter	record transaction add to wait queue
kstat_waitq_exit	record transaction removal from wait queue
kstat_waitq_to_runq	record transaction migration from wait queue to run queue

Kernel Logging and Printing Functions

TABLE B–30 Kernel Logging and Printing Functions

Function Name	Description
cmn_err	display an error message
vcmn_err	display an error message
ddi_report_dev	announce a device
strlog	submit messages to the log driver
ddi_dev_report_fault	report a hardware failure
scsi_errmsg	display a SCSI request sense message
scsi_log	display a SCSI-device-related message
scsi_vu_errmsg	display a SCSI request sense message

Buffered I/O Functions

TABLE B–31 Buffered I/O Functions

Function Name	Description
physio	perform physical I/O
aphysio	perform asynchronous physical I/O
anocancel	prevent cancellation of asynchronous I/O request
minphys	limit physio buffer size
biowait	suspend processes pending completion of block I/O
biodone	release buffer after buffer I/O transfer and notify blocked threads
bioerror	indicate error in buffer header
geterror	return I/O error
bp_mapin	allocate virtual address space
bp_mapout	deallocate virtual address space
disksort	single direction elevator seek sort for buffers
getrbuf	get a raw buffer header
freerbuf	free a raw buffer header
biosize	returns size of a buffer structure
bioinit	initialize a buffer structure
biofini	uninitialize a buffer structure
bioreset	reuse a private buffer header after I/O is complete
bioclone	clone another buffer
biomodified	check if a buffer is modified
clrbuf	erase the contents of a buffer

Virtual Memory Functions

TABLE B–32 Virtual Memory Functions

Function Name	Description
ddi_btop	convert device bytes to pages (round down)
ddi_btopr	convert device bytes to pages (round up)
ddi_ptob	convert device pages to bytes
btop	convert size in bytes to size in pages (round down)
btopr	convert size in bytes to size in pages (round up)
ptob	convert size in pages to size in bytes

TABLE B–33 Deprecated Virtual Memory Functions

Deprecated Functions	Replacement
hat_getkpfnum	see devmap(9e), ddi_dma_*_bind_handle, ddi_dma_nextcookie

Device ID Functions

TABLE B–34 Device ID Functions

Function Name	Description
ddi_devid_init	allocate a device id structure
ddi_devid_free	free a device id structure
ddi_devid_register	register a device id
ddi_devid_unregister	unregister a device id
ddi_devid_compare	compare two device ids

TABLE B–34 Device ID Functions *(continued)*

Function Name	Description
ddi_devid_sizeof	return the size of a device id
ddi_devid_valid	validate a device id

SCSI Functions

TABLE B–35 SCSI Functions

Function Name	Description
scsi_probe	probe a SCSI device
scsi_unprobe	free resources allocated during initial probing
scsi_alloc_consistent_buf	allocate an I/O buffer for SCSI DMA
scsi_free_consistent_buf	free a previously allocated SCSI DMA I/O buffer
scsi_init_pkt	prepare a complete SCSI packet
scsi_destroy_pkt	free an allocated SCSI packet and its DMA resource
scsi_setup_cdb	setup SCSI command descriptor block (CDB)
scsi_transport	start a SCSI command
scsi_poll	run a polled SCSI command
scsi_ifgetcap	get SCSI transport capability
scsi_ifsetcap	set SCSI transport capability
scsi_sync_pkt	synchronize CPU and I/O views of memory
scsi_abort	abort a SCSI command
scsi_reset	reset a SCSI bus or target
scsi_reset_notify	notify target driver of bus resets

TABLE B–35 SCSI Functions *(continued)*

Function Name	Description
scsi_cname	decode a SCSI command
scsi_dname	decode a SCSI peripheral device type
scsi_mname	decode a SCSI message
scsi_rname	decode a SCSI packet completion reason
scsi_sname	decode a SCSI sense key
scsi_errmsg	display a SCSI request sense message
scsi_log	display a SCSI-device-related message
scsi_vu_errmsg	display a SCSI request sense message
scsi_hba_init	SCSI HBA system initialization routine
scsi_hba_fini	SCSI HBA system completion routine
scsi_hba_attach_setup	SCSI HBA attach routine
scsi_hba_detach	SCSI HBA detach routine
scsi_hba_probe	default SCSI HBA probe function
scsi_hba_tran_alloc	allocate a transport structure
scsi_hba_tran_free	free a transport structure
scsi_hba_pkt_alloc	allocate a scsi_pkt structure
scsi_hba_pkt_free	free a scsi_pkt structure
scsi_hba_lookup_capstr	return index matching capability string

TABLE B–36 Deprecated SCSI Functions

Deprecated Functions	Replacement
free_pktiopb	scsi_free_consisten_buf
get_pktiopb	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

Resource Map Management Functions

TABLE B–37 Resource Map Management Functions

Function Name	Description
rmallocmap	allocate resource map
rmallocmap_wait	allocate resource map, wait if necessary

TABLE B–37 Resource Map Management Functions *(continued)*

Function Name	Description
rmfreemap	free resource map
rmalloc	allocate space from a resource map
rmalloc_wait	allocate space from a resource map, wait if necessary
rmfree	free space back into a resource map

System Global State

TABLE B–38 System Global State

Function Name	Description
ddi_in_panic	determine if system is in panic state

Utility Functions

TABLE B–39 Utility Functions

Function Name	Description
nulldev	zero return function
nodev	error return function
nochpoll	error return function for non-pollable devices
ASSERT	expression verification

TABLE B–39 Utility Functions *(continued)*

Function Name	Description
bcopy	copy data between address locations in the kernel
bzero	clear memory for a given number of bytes
bcmp	compare two byte arrays
ddi_ffs	find first bit set in a long integer
ddi_fls	find last bit set in a long integer
swab	swap bytes in 16-bit halfwords
strcmp	compare two null-terminated strings
strncmp	compare two null-terminated strings, with length limit
strlen	determine the number of non-null bytes in a string
strcpy	copy a string from one location to another
strncpy	copy a string from one location to another, with length limit
strchr	find a character in a string
sprintf	format characters in memory
vsprintf	format characters in memory
numtos	convert integer to decimal string
stoi	convert decimal string to an integer
max	return the larger of two integers
min	return the lesser of two integers
va_arg	handle variable argument list

TABLE B–39 Utility Functions *(continued)*

Function Name	Description
va_copy	handle variable argument list
va_end	handle variable argument list
va_start	handle variable argument list

APPENDIX C

Making a Device Driver 64-Bit Ready

This appendix provides information for device driver writers who are converting their device drivers to support the 64-bit kernel. It presents the differences between 32-bit and 64-bit device drivers and describes the steps to convert 32-bit device drivers to 64-bit. This information is specific to regular character and block device drivers only.

Introduction

For drivers that need only support for the 32-bit kernel, existing 32-bit device drivers will continue to work without recompilation. However, most device drivers require some changes to run correctly in the 64-bit kernel, and all device drivers require recompilation to create a 64-bit driver module. The information in this appendix will enable drivers for 32-bit and 64-bit environments to be generated from common source code, thus increasing code portability and reducing the maintenance effort.

General Issues

Before starting to clean up a device driver for the 64-bit environment, you should understand how the 32-bit environment differs from the 64-bit environment. In particular, it is important to be familiar with the C language data type models ILP32 and LP64, and to be aware of driver-specific issues. Driver-specific issues are the subject of this appendix, while more general issues are covered extensively in *Solaris 64-bit Developer's Guide*.

Driver-Specific Issues

In addition to general code cleanup to support the data model changes for LP64, driver writers have to provide support for both 32-bit and 64-bit applications.

The `ioctl(9E)`, `devmap(9E)`, and `mmap(9E)` entry points allow data structures to be shared directly between applications and device drivers. If those data structures change size between the 32-bit and 64-bit environments, then the entry points must be modified so that the driver can determine whether the data model of the application is the same as that of the kernel. When the data models differ, data structures can be adjusted, using the techniques discussed in the previous chapter.

Practically speaking, in many drivers, only a few ioctls need this kind of handling; the others will work without change as long as they pass around data structures that do not change in size.

General Conversion Steps

The sections below provide information on converting drivers to run in a 64-bit environment. Driver writers might need to do one or more of the following:

1. Use fixed-width types for hardware registers.
2. Use fixed-width common access functions.
3. Check and extend use of derived types.
4. Check changed fields within DDI data structures.
5. Check changed arguments of DDI functions.
6. Modify the driver entry points that handle user data, where needed.

These steps are explained in detail below.

After each step is complete, fix all compiler warnings, and use `lint` to look for other problems. The SC5.0 (or newer) version of `lint` should be used with `–Xarch=v9` and `–errchk=longptr64` specified to find 64–bit problems. See the notes on using and interpreting the output of `lint` in the *Solaris 64-bit Developer's Guide*.

Caution - Do not ignore compilation warnings during conversion for LP64. Even those that were safe to ignore previously in the ILP32 environment might now indicate a more serious problem.

After all the steps are complete, compile and test the driver as both a 32-bit and 64-bit modules.

Use Fixed-width Types for Hardware Registers

Many device drivers that manipulate hardware devices use C data structures to describe the layout of the hardware. In the LP64 data model, data structures that use long or unsigned long to define hardware registers are almost certainly incorrect, since long is now a 64-bit quantity. Start by including <sys/inttypes.h>, and update this class of data structure to use int32_t or uint32_t instead of long for 32-bit device data. This preserves the binary layout of 32-bit data structures. For example, change:

```
struct device_regs {
        ulong_t            addr;
        uint_t             count;
};      /* Only works for ILP32 compilation */
```

to:

```
struct device_regs {
        uint32_t            addr;
        uint32_t            count;
};      /* Works for any data model */
```

Use Fixed-width Common Access Functions

The Solaris DDI permits device registers to be accessed by access functions for portability to multiple platforms. Previously, the DDI common access functions specified the size of data in terms of bytes, words, and so on. For example, ddi_getl(9F) is used to access 32-bit quantities. This function is not available in the 64-bit DDI environment, and has been replaced by versions that are specified using the number of bits that they manipulate.

These routines were added to the 32-bit kernel in the Solaris 2.6 operating environment, to permit their early adoption by driver writers. For example, to be portable to both 32-bit and 64-bit kernels, the driver must use ddi_get32(9F) to access 32-bit data rather than ddi_getl(9F).

The entire set of common access routines is replaced by their fixed-width equivalents. See ddi_get8(9F), ddi_put8(9F), ddi_rep_get8(9F), and ddi_rep_put8(9F).

Check and Extend Use of Derived Types

System-derived types, such as size_t, should be used where possible so that the resulting variables make sense when passed between functions. The new derived types uintptr_t or intptr_t should be used as the integral type for pointers.

Fixed-width integer types are useful for representing explicit sizes of binary data structures or hardware registers, while fundamental C language data types, such as int, can still be used for loop counters or file descriptors.

Some system-derived types represent 32-bit quantities on a 32-bit system but represent 64-bit quantities on a 64-bit system. Derived types that change size in this way include: `clock_t`, `daddr_t`, `dev_t`, `ino_t`, `intptr_t`, `off_t`, `size_t`, `ssize_t`, `time_t`, `uintptr_t`, and `timeout_id_t`.

Drivers that use these derived types should pay particular attention to their use, particularly if they are assigning these values to variables of another derived type, such as a fixed-width type.

Check Changed Fields in DDI Data Structures

The data types of some of the fields within DDI data structures, such as `buf(9S)`, have been changed. Drivers that use these data structures should make sure that these fields are being used appropriately. The data structures and the fields that were changed in a significant way are listed below.

buf(9S)

```
size_t          b_bcount;           /* was type unsigned int */
size_t          b_resid;            /* was type unsigned int */
size_t          b_bufsize;          /* was type long */
```

The fields changed here pertain to transfer size, which can now exceed more than 4GB in future systems.

ddi_dma_attr(9S)

This structure defines attributes of the DMA engine and the device. Because these attributes specify register sizes, fixed-width data types have been used instead of fundamental types.

ddi_dma_cookie(9S)

```
uint32_t    dmac_address;       /* was type unsigned long */
size_t      dmac_size;          /* was type u_int */
```

This structure contains a 32-bit DMA address, so a fixed-width data type has been used to define it. The size has been redefined as `size_t`.

scsi_arq_status(9S)

```
uint_t       sts_rqpkt_state;           /* was type u_long */
uint_t       sts_rqpkt_statistics;      /* was type u_long */
```

These fields do not need to grow and have been redefined as 32-bit quantities.

scsi_pkt(9S)

```
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 */
```

Because the `pkt_flags`, `pkt_state`, and `pkt_statistics` fields do not need to grow, they have been redefined as 32-bit integers. The data transfer size `pkt_resid` field *does* grow and has been redefined as `ssize_t`.

Check Changed Arguments of DDI Functions

Some DDI function argument data types have been changed. These routines are listed below.

getrbuf(9F)

```
struct buf *getrbuf(int sleepflag);
```

In previous releases, `sleepflag` was defined as a type `long`.

drv_getparm(9F)

```
int drv_getparm(unsigned int parm, void *value_p);
```

In previous releases, `value_p` was defined as type `unsigned long *`.

In the 64-bit kernel, `drv_getparm(9F)` can be used to fetch both 32-bit and 64-bit quantities, yet the interface does not define the data types of these quantities, which encourages simple programming errors.

The following new routines offer a safer alternative:

```
clock_t           ddi_get_lbolt(void);
time_t            ddi_get_time(void);
cred_t            *ddi_get_cred(void);
pid_t             ddi_get_pid(void);
```

Driver writers are strongly urged to use these routines instead of `drv_getparm(9F)`.

delay(9F) and timeout(9F)

```
void delay(clock_t ticks);
timeout_id_t timeout(void (*func)(void *), void *arg, clock_t ticks);
```

The *ticks* argument to both of these routines has been changed from `long` to `clock_t`.

rmallocmap(9F) and rmallocmap_wait(9F)

```
struct map *rmallocmap(size_t mapsize);
struct map *rmallocmap_wait(size_t mapsize);
```

The `mapsize` argument to both of these routines has been changed from `ulong_t` to `size_t`.

scsi_alloc_consistent_buf(9F)

```
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);
```

In previous releases, `datalen` was defined as an `int` and `bflags` was defined as a `ulong`.

uiomove(9F)

```
int uiomove(caddr_t address, size_t nbytes,
        enum uio_rw rwflag, uio_t *uio_p);
```

The `nbytes` argument was defined as a type `long`, but because it represents a size in bytes, `size_t` is more appropriate.

cv_timedwait(9F) and cv_timedwait_sig(9F)

```
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);
```

In previous releases, the timeout argument to both of these routines was defined to be of type `long`. Because they represent time in ticks, `clock_t` is more appropriate.

ddi_device_copy(9F)

```
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);
```

The src_advcnt, dest_advcnt, dev_datasz arguments have changed type. These were previously defined as long, long, and ulong_t respectively.

ddi_device_zero(9F)

```
int ddi_device_zero(ddi_acc_handle_t handle,
        caddr_t dev_addr, size_t bytecount, ssize_t dev_advcnt,
        uint_t dev_datasz):
```

In previous releases, dev_advcnt was defined as a type long and dev_datasz as a ulong_t.

ddi_dma_mem_alloc(9F)

```
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);
```

In previous releases, length, flags, and real_length were defined with types uint_t, ulong_t, and uint_t *.

Modify Routines That Handle Data Sharing

If a device driver shares data structures that contain longs or pointers with a 32-bit application using ioctl(9E), devmap(9E), or mmap(9E), and the driver is recompiled for a 64-bit kernel, the binary layout of data structures will be incompatible. If a field is currently defined in terms of type long, and there is no actual need for 64-bit data items, change the data structure to use data types that remain as 32–bit quantities (int and unsigned int). Otherwise, the driver needs to be aware of the different structure shapes for ILP32 and LP64 and determine whether there is a model mismatch between the application and the kernel.

To handle potential data model differences, the ioctl(9E), devmap(9E), and mmap(9E) driver entry points, which interact directly with user applications, need to be written to determine whether the argument came from an application using the same data model as the kernel.

ioctl(9E)

To determine whether there is a model mismatch between the application and the driver, the driver uses the FMODELS mask to determine the model type from the

ioctl(9E) mode argument. The following values are OR-ed into mode to identify the application data model:

- FLP64 – Application uses the LP64 data model.
- FILP32 – Application uses the ILP32 data model.

Look at the code examples in the previous chapter to see how this can be handled using either ddi_model_convert_from( ) or the data structure macros.

devmap(9E)

To enable a 64-bit driver and a 32-bit application to share memory, the binary layout generated by the 64-bit driver must be the same as consumed by the 32-bit application. The mapped memory being exported to the application might need to contain data-model-dependent data structures.

Few memory mapped devices face this problem because the device registers do not change size when the kernel data model changes. However, some pseudo-devices that export mappings to the user address space might want to export different data structures to ILP32 or LP64 applications. To determine whether there is a data model mismatch, devmap(9E) uses the model parameter to describe the data model expected by the application. The model parameter is set to one of the following:

- DDI_MODEL_ILP32 – The application uses the ILP32 data model.
- DDI_MODEL_LP64 – The application uses the LP64 data model.

The model parameter can be passed untranslated to the ddi_model_convert_from(9F) routine or to STRUCT_INIT( ).

mmap(9E)

Because mmap(9E) does not have a parameter that can be used to pass data model information, the driver's mmap(9E) entry point can be written to use the new DDI function ddi_mmap_get_model(9F). This function returns one of the following values to indicate the application's data type model:

- DDI_MODEL_ILP32 – Application expects the ILP32 data model.
- DDI_MODEL_ILP64 – Application expects the LP64 data model.
- DDI_FAILURE – Function was not called from mmap(9E).

As with ioctl(9E) and devmap(9E), the model bits can be passed to ddi_model_convert_from(9F) to determine whether data conversion is necessary, or the model can be handed to STRUCT_INIT( ).

Alternatively, migrate the device driver to support the devmap(9E) entry point.

Well-known ioctl Interfaces

Many ioctl operations are common to a class of device drivers. For example, most disk drivers implement many of the `dkio(7I)` family of ioctls. Many of these interfaces copy in or copy out data structures from the kernel, and some of these data structures have changed size in the LP64 data model. The following section lists the ioctls that now require explicit conversion in 64-bit driver ioctl routines for the `dkio(7I)`, `fdio(7I)`, `fbio(7I)`, `cdio(7I)`, and `mtio(7I)` families of ioctls.

ioctl command	Affected data structure	Reference
DKIOCGAPART DKIOCSAPART	struct dk_map struct dk_allmap	dkio(4)
DKIOGVTOC DKIOSVTOC	struct partition struct vtoc	dkio(4)
FBIOPUTCMAP FBIOGETCMAP	struct fbcmap	fbio(4)
FBIOPUTCMAPI FBIOGETCMAPI	struct fbcmap_i	fbio(4)
FBIOSCURSOR FBIOSCURSOR	struct fbcursor	fbio(4)
CDROMREADMODE1 CDROMREADMODE2	struct cdrom_read	cdio(4)
CDROMCDDA	struct cdrom_cdda	cdio(4)
CDROMCDXA	struct cdrom_cdxa	cdio(4)
CDROMSUBCODE	struct cdrom_subcode	cdio(4)
FDIOCMD	struct fd_cmd	fdio(4)
FDRAW	struct fd_raw	fdio(4)
MTIOCTOP	struct mtop	mtio(4)
MTIOCGET	struct mtget	mtio(4)

ioctl command	Affected data structure	Reference
MTIOCGETDRIVETYPE	struct mtdrivetype_request	mtio(4)
USCSICMD	struct uscsi_cmd	scsi(4)

Device Sizes

The `nblocks` property is exported by each slice of a block device driver. It contains the number of 512 byte blocks that each slice of the device can support. The `nblocks` property is defined as a signed 32-bit quantity, which limits the maximum size of a slice to 1 Tbyte.

Disk devices that provide more than 1 Tbyte of storage per disk, must define the `Nblocks` property, which should still contain the number of 512 byte blocks that the device can support. However, `Nblocks` is a signed 64-bit quantity, which removes any practical limit on disk space.

The `nblocks` property is now deprecated; all disk devices should provide the `Nblocks` property.

APPENDIX **D**

DDI Interfaces for Cluster-Aware Drivers

The device node types supported by the Solaris operating environment can be divided roughly into two categories: physical and pseudo devices. This categorization is important when the device nodes are created and used by Sun™ Cluster.

The concept of device classes and the necessary interface modifications and additions are introduced in this release of the Solaris operating environment so that device driver writers can adopt the new interfaces for use with future versions of Sun Cluster. The device classes will not have an impact on base Solaris operation because they are ignored by the base kernel without Sun Cluster software installed.

Device Classification

Sun Cluster introduces a new device classification scheme. These new classifications are based on the extended behavior of the devices in a Sun Cluster environment.

Enumerated Devices	`ENUMERATED_DEV`
Node Specific Devices	`NODESPECIFIC_DEV`
Global Devices	`GLOBAL_DEV`
Node Bound Devices	`NODEBOUND_DEV`

The `ddi_create_minor_node`(9F) routine has been enhanced to add the capability of reporting the additional device classification of the device minor nodes created by the device driver. The device node class is specified using the *flag*

parameter. If the device class is not indicated, the default class for pseudo devices will be NODESPECIFIC_DEV and for physical devices will be ENUMERATED_DEV. These device classes do not effect the creation of the device node in a non-clustered environment; but they are required for device drivers intended for use in a clustered environment. The device categories are described in the following sections.

Enumerated Devices

Enumerated devices are physical devices with a one-to-one correspondence between a particular device node and a host where that device node is present. Examples of this category include various disk and tape devices, such as /dev/dsk/c0t0d0s0 and /dev/rmt/0l. Nearly all physical devices belong to this category. This is the default category for all non-pseudo devices.

Node Specific Devices

Node specific devices include devices that report particular information about the host where the device node is opened. An example of such a device is the /dev/kmem device. Opening this device gives access to host-specific information on the local host. Administrative pseudo device nodes used in configuring or gathering information about a particular device driver also fit this category. The Sun Cluster software ensures the creation of two user device nodes for each instance of a kernel device node in the cluster, so that the intended device node can be accessed both locally and remotely.

Global Devices

Global devices are node invariant pseudo devices such as /dev/ip. In principle, the open instance of a device, such as ip or tcp, does not depend on which host, in the cluster, the open occurs. A single copy of each device is in the kernel. All device I/O requests for this device class are performed locally and the device node can be accessed by a remote host within the cluster. This is the default behavior for all pseudo devices in the system.

Node Bound Devices

A node bound device is a pseudo device that maintains a cluster-wide state. This device should, in principle, be opened on one node only. Devices such as /dev/ticotsord belong to this class (see the ticotsord(7D) man page). Highly available devices with automatic fail-over also belong to this class. Only one pseudo

node is present but all opens are directed to the same node, with the exception of HA devices, where the hosting node might change, transparent to the device user.

Minor Number Space Management

dev_t consists of a major and a minor number space. The major number space is managed by base Solaris and the minor number space is managed by the device driver space. With Sun Cluster, the minor number behaves differently within the user space and the kernel space.

Sun Cluster preserves the assumption that two equal dev_ts point to the same device regardless of the host where the process is executed. This model satisfies the expectations of programs that depend on this feature to establish the equivalence of two devices. Sun Cluster introduces a dual view of minor numbers and the necessary interfaces to implement this dual view. In-kernel dev_ts correspond to the major number of the driver in addition to the minor number that the driver has created using ddi_create_minor_node(9F). External minor numbers (viewed from the user space) are managed and assigned unique cluster-wide numbers by the device configuration manager in Sun Cluster.

This dual numbering scheme has one unfortunate side effect, namely that a particular minor number created in the kernel can result in creation of a different minor number in the user space. This discrepancy might be unexpected by user space programs that expect to be able to ascertain some device characteristics from the minor number pattern.

An example of the discrepancy is the use of minor number bit patterns in specifying the particular slice of a disk or the density of a tape device. This class of problems is primarily alleviated by the use of globally unique instance numbers. By encoding the instance number of a device in the minor, the driver can guarantee the creation of cluster-wide unique dev_t values; this avoids minor numbers that do not have the same value between the kernel and the user space.

All dev_t values that are passed in through the standard Solaris entry points, such as open(9E), close(9E) and ioctl(9E), encode the kernel minor number. The getminor(9F) interface can be used to extract this minor number. However, if the dev_t value is passed as a part of the ioctl data from the user space, the dev_t value has the minor number from the user space encoded. A new DDI interface, ddi_getiminor(9F), has been introduced to ensure that the driver can map between internal and external minor numbers.

Device Interfaces

The following interface sets up a driver and prepares it for use:

```
int ddi_create_minor_node(dev_info_t *dip, char *name,
            int spec_type, int minor_num, char *node_type, int flag);
```

ddi_create_minor_node(9F) advertises a minor device node, which will eventually appear in the /devices directory and refer to the device specified by dip. If the device is a clone device, then flag is set to CLONE_DEV. If it is not a clone device, then flag is set to 0. For device drivers intended for use in a clustered environment, flag must specify the device node class of GLOBAL_DEV, NODEBOUND_DEV, NODESPECIFIC_DEV, or ENUMERATED_DEV.

The following new interface is used to translate between user-visible device numbers and in kernel device numbers:

```
minor_t ddi_getiminor(dev_t dev);
```

ddi_getiminor(9F) extracts the minor number as a device number.

APPENDIX **E**

Hardened Drivers

Overview of the Process

Hardening is the process of ensuring that a driver works correctly in spite of faults in the I/O device that it controls or other faults originating outside the system core. A hardened driver must not panic, hang the system, or allow the uncontrolled spread of corrupted data as the result of any such faults.

A hardened driver obeys all the rules of a standard Solaris device driver plus some additional rules:

- Each piece of hardware should be controlled by a separate instance of the device driver.
- Programmed I/O (PIO) must be performed *only* through the DDI access functions, using the appropriate data access handle.
- Device driver must assume that data it receives from the device could be corrupted. The driver must check the integrity of the data before using it.
- Driver must control the effects of any faults that it detects. Device-supplied data must be integrity checked before it is released to the rest of the system.
- Driver must ensure that all device writes using DMA buffers must be contained within pages of memory controlled entirely by the driver. This prevents a DMA fault from corrupting an arbitrary part of the system's main memory.
- Driver must not be an unlimited drain on system resources if the device locks up. It should time-out if a device claims to be continuously busy. The driver should also detect a pathological (stuck) interrupt request and take appropriate action.
- Driver must free up resources after a fault. For example, the system must be able to close all minor devices and detach driver instances, even after the hardware fails.

Responsibilities of the Driver Writer

The developer must take responsibility for:

- Correct use of the DDI functions
- Handling devices with deviant interrupt logic
- Detecting any corruption of device I/O

Device Driver Instances

The Solaris kernel allows multiple instances of a driver. Each instance has its own data space but shares the text and some global data with other instances. The device is managed on a per-instance basis and hardened drivers should use a separate instance for each piece of hardware, unless the driver is designed to handle fail-over internally. There can be multiple instances of a driver per slot—for example, multifunction cards—which is standard behavior for Solaris device drivers.

Exclusive Use of DDI Access Handles

All programmed I/O (PIO) access by a hardened driver must use DDI access functions from the `ddi_get`, `ddi_put`, `ddi_rep_get`, and `ddi_rep_put` families of routines. It should not directly access the mapped registers by the address returned from `ddi_regs_map_setup`(9F). Using an access handle ensures that an I/O fault is controlled and its effects confined to the returned value, rather than possibly corrupting other parts of the machine state. (Avoid the `ddi_peek`(9F) and `ddi_poke`(9F) routines because they do not use access handles.)

The DDI access mechanism is important because it provides an opportunity to control how data is read into the kernel. DDI access routines provide protection by constraining the effect of bus timeout traps.

Detecting Corrupted Data

The following subsections consider where data corruption can occur and the steps you can take to detect it.

Corruption of Device Management and Control Data

The driver should assume that any data obtained from the device, whether by PIO or DMA, could have been corrupted. In particular, extreme care should be taken with pointers, memory offsets, or array indexes read or calculated from data supplied by the device. Such values can be *malignant*, meaning they can cause a kernel panic if dereferenced. All such values should be checked for range and alignment (if required) before use.

Even if a pointer is not malignant, it can still be misleading. For example, it can point at a valid instance of an object, but not the correct one. Where possible, the driver should cross-check the pointer with the pointed-to object, or otherwise validate the data obtained through it.

Other types of data can also be misleading, such as packet lengths, status words, or channel IDs. Each should be checked to the extent possible: a packet length can be range-checked to ensure that it is not negative or larger than the containing buffer; a status word can be checked for "impossible" bits; and a channel ID can be matched against a list of valid IDs.

Where a value is used to identify a STREAM, the driver must ensure that the STREAM still exists. The asynchronous nature of STREAMS processing means that a STREAM can be dismantled while device interrupts are still outstanding.

The driver should not reread data from the device; the data should be read once, validated, and stored in the driver's local state. This avoids the hazard presented by data that, although correct when initally read and validated, is incorrect when reread later.

The driver should also ensure that all loops are bounded, so that a device returning a continuous BUSY status, or claiming that another buffer needs to be processed, does not lock up the entire system.

Corruption of Received Data

Device errors can result in corrupted data being placed in receive buffers. Such corruption is indistinguishable from corruption that occurs beyond the domain of the device—for example, within a network. Typically, existing software is already in place to handle such corruption, for example, through integrity checks at the transport layer of a protocol stack or within the application using the device.

If the received data will not be checked for integrity at a higher layer—as in the case of a disk driver, for example—it can be integrity checked within the driver itself. Methods of detecting corruption in received data are typically device-specific (checksums, CRC, and so forth).

Detecting Faults

Any ancestor of a device driver can disable the data path to the device if it detects a fault. When PIO access is disabled, reads from the device return `undefined` values, while writes are ignored. If DMA access is disabled, the device might be prevented from accessing memory, or it might receive undefined data on reads and have writes discarded.

For a device driver to detect that a data path has been disabled, the following DDI routines are provided:

- `ddi_check_acc_handle`(9F)
- `ddi_check_dma_handle`(9F)

Each function checks whether any faults affecting the data path, represented by the supplied *handle*, have been detected. If one of these functions returns `DDI_FAILURE`, indicating that the data path has failed, the driver should report the fault using `ddi_dev_report_fault`(9F), perform any necessary cleanup, and, where possible, return an appropriate error to its caller.

Containment of Faults

Preservation of system integrity requires that faults are detected before they alter the system state. Consequently, the driver must be sure to test for faults whenever data returned from the device is going to be used by the system.

- The `ddi_check_acc_handle`(9F) and `ddi_check_dma_handle`(9F) calls should be made at significant junctures, such as just before passing a data block to the upper layers.
- Data must not be forwarded out of the driver if the device has failed.
- The driver must consider other possible impacts of the failure on the integrity of the system. The driver must ensure that kernel resources, such as memory, are not permanently lost when data cannot be forwarded. Threads should not remain blocked waiting for signals that will never be generated.
- The driver should limit its processing while in the failed state (for example, to free messages in `wput` routines, attempts to permanently disable interrupts from a failed board, and so forth).

Handling Stuck Interrupts

The driver must identify stuck interrupts because a persistently asserted interrupt severely affects system performance, almost certainly stalling a single-processor machine.

Sometimes the driver has a hard time identifying a particular interrupt as a hoax. For network drivers, if a receive interrupt is indicated but no new buffers have been

made available, no work was needed. When this is an isolated occurrence, it is not a problem, as the actual work might already have been completed by another routine (read service, for example).

On the other hand, continuous interrupts with no work for the driver to process can indicate a stuck interrupt line. For this reason, all platforms allow a number of apparently hoax interrupts to occur before taking defensive action.

A hung device, while appearing to have work to do, might be failing to update its buffer descriptors. The driver should ascertain that it is not fooled by such repetitive requests.

On certain platforms, platform–specific bus drivers might be capable of identifying a persistently unclaimed interrupt and can disable the offending device. However, this relies on the driver's ability to identify the good interrupts and return the appropriate value. The driver should therefore return a `DDI_INTR_UNCLAIMED` result, unless it detects that the device legitimately asserted an interrupt (that is, the device actually requires the driver to do some useful work).

The legitimacy of other, more incidental, interrupts is much harder to certify. To this end, an interrupt expected flag is a useful tool for evaluating whether an interrupt is valid. Consider an interrupt such as *descriptor free*, which can be generated if, previously, all the device's descriptors had been allocated. If the driver detects that it has taken the last descriptor from the card, it can set an interrupt expected flag. If this flag is not set when the associated interrupt is delivered, it is suspicious.

Some informative interrupts might not be predictable, such as one indicating that a medium has become disconnected or frame sync has been lost. The easiest method of detecting whether such an interrupt is stuck is to mask this particular source on first occurrence until the next polling cycle.

If the interrupt occurs again while disabled, this should be considered a false interrupt. Some devices have interrupt status bits that can be read even if the mask register has disabled the associated source and might not be causing the interrupt. Driver designers can devise more appropriate algorithms specific to their devices.

Avoid looping on interrupt status bits indefinitely. Break such loops if none of the status bits set at the start of a pass requires any real work.

DMA Isolation

A defective PCI device might initiate an improper DMA transfer over the bus. This data transfer could corrupt good data previously delivered. A device that fails might generate a corrupt address that can contaminate memory not even belonging to this driver.

In systems with an IOMMU, a device can write only to pages mapped as writable for DMA. It is therefore important that DMA writes data only into pages that are owned by that driver instance and not shared with any other kernel structure. While the

page in question is mapped as writable for DMA, the driver should be suspicious of data in that page, and the page must be unmapped from the IOMMU before it is passed beyond the driver, or before any validation of the data.

To guarantee that a whole aligned page is allocated, `ddi_umem_alloc`(9F) can be used, or multiple pages allocated and the memory below the first page boundary ignored. The size of an IOMMU page can be found using `ddi_ptob`(9F).

Alternatively, the driver can choose to copy the data into a safe part of memory before processing it. If this is done, the data must first be synchronized using `ddi_dma_sync`(9F).

Calls to `ddi_dma_sync`(9F) should specify `SYNC_FOR_DEV` before using DMA to transfer data to a device, and `SYNC_FOR_CPU` after using DMA to transfer data from the device to memory.

Some PCI devices are able to use dual address cycles (64-bit addresses) to avoid the IOMMU. This gives the device the potential to corrupt any region of main memory. Hardened device drivers must not attempt to use such a mode and should disable it.

Thread Interaction

Kernel panics in a device driver are often caused by unexpected interaction of kernel threads after a device failure. When a device fails, threads can interact in ways that the designer had not anticipated.

If processing routines terminate early, the condition variable waiters are blocked because an expected signal is never given. Attempting to inform other modules of the failure or handling unanticipated callbacks can result in undesirable thread interactions. Consider the sequence of mutex acquisition and relinquishing that can occur during device failures.

Threads that originate in an upstream STREAMS module can run into unfortunate paradoxes if used to return to that module unexpectedly. You might use alternative threads to handle exception messages. For instance, a `wput` procedure might use a read-side service routine to communicate an `M_ERROR`, rather than doing it directly with a read-side `putnext`.

A failing STREAMS device that cannot be quiesced during close (because of the fault) can generate an interrupt after the stream has been dismantled. The interrupt handler must not attempt to use a stale stream pointer to try to process the message.

Threats From Top-Down Requests

While protecting the system from defective hardware, the driver designer also needs to protect against driver misuse. Although the driver can assume that the kernel

infrastructure is always correct (a trusted core), user requests passed to it can be potentially destructive.

For example, a user can request an action to be performed upon a user-supplied data block (`M_IOCTL`) that is smaller than that indicated in the control part of the message. The driver should never trust a user application.

The design should consider the construction of each type of `ioctl` that it can receive with a view to the potential harm that it could cause. The driver should make checks to be sure that it does not process malformed `ioctls`.

Adaptive Strategies

A driver can continue to provide a service with faulty hardware, attempting to work around the identified problem by using an alternative strategy for accessing the device. Given the unpredictability of broken hardware and the risk associated with additional design complexity, adaptive strategies are not always wise. At most, they should be limited to periodic interrupt polling and retry attempts. Periodically retrying the device lets the driver know when a device has recovered. Periodic polling can control the interrupt mechanism, after a driver has been forced to disable interrupts.

Ideally, a system always has an alternative device to provide a vital system service. Service multiplexors in kernel or user space offer the best method of maintaining system services when a device fails. Such practices are beyond the scope of this appendix.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W