Device Driver Tutorial

Part No: E36866
July 2014

ORACLE

Contents

Using This Documentation 11

1 Introduction to Device Drivers 13
- Oracle Solaris Operating System Definition 13
- Kernel Overview 13
  - Differences Between Kernel Modules and User Programs 13
  - User and Kernel Address Spaces on x86 and SPARC Machines 16
  - Device Drivers 16
  - Driver Directory Organization 18
- Devices as Files 19
  - Devices Directories 20
  - Device Tree 20
  - Character and Block Devices 21
  - Device Names 21
  - Device Numbers 22
- Development Environment and Tools 23
  - Writing a Driver 24
  - Building a Driver 25
  - Installing a Driver 27
  - Adding, Updating, and Removing a Driver 28
  - Loading and Unloading a Driver 29
  - Testing a Driver 29

2 Template Driver Example 31
- Overview of the Template Driver Example 31
- Writing the Template Driver 32
  - Writing the Loadable Module Configuration Entry Points 32
  - Writing the Autoconfiguration Entry Points 37
  - Writing the User Context Entry Points 44
  - Writing the Driver Data Structures 48

Writing the Device Configuration File 54
Building and Installing the Template Driver 54
Testing the Template Driver 55
Adding the Template Driver 55
Reading and Writing the Device 56
Removing the Template Driver 57
Complete Template Driver Source 58

3 Reading and Writing Data in Kernel Memory 63
Displaying Data Stored in Kernel Memory 63
Writing Quote Of The Day Version 1 63
Building, Installing, and Using Quote Of The Day Version 1 65
Displaying Data on Demand 66
Writing Quote Of The Day Version 2 66
Building, Installing, and Using Quote Of The Day Version 2 75
Modifying Data Stored in Kernel Memory 77
Writing Quote Of The Day Version 3 77
Building and Installing Quote Of The Day Version 3 96
Using Quote Of The Day Version 3 97

4 Tips for Developing Device Drivers 103
Device Driver Coding Tips 103
Device Driver Testing Tips 106
Device Driver Debugging and Tuning Tips 108

Index 111

Figures

FIGURE 1-1 Typical Device Driver Entry Points 17

FIGURE 1-2 Entry Points for Different Types of Drivers 17

FIGURE 1-3 Typical Device Driver Interactions 18

FIGURE 2-1 Entry Points for the `dummy` Example 32

Tables

TABLE 2-1 Get Driver Information Entry Point Arguments 42

Examples

EXAMPLE 3-1 Quote Of The Day Version 1 Source File 64
EXAMPLE 3-2 Quote Of The Day Version 1 Configuration File 65
EXAMPLE 3-3 Quote Of The Day Version 2 Source File 70
EXAMPLE 3-4 Quote Of The Day Version 2 Configuration File 75
EXAMPLE 3-5 Quote Of The Day Version 3 Source File 87
EXAMPLE 3-6 Quote Of The Day Version 3 Header File 96
EXAMPLE 3-7 Quote Of The Day Version 3 Configuration File 96
EXAMPLE 3-8 Quote Of The Day I/O Control Command Source File 99

Using This Documentation

- **Overview** – This *Device Driver Tutorial* is a hands-on guide that shows you how to develop a simple device driver for the Oracle Solaris™ Operating System (Oracle Solaris OS). *Device Driver Tutorial* also explains how device drivers work in the Oracle Solaris OS. This book is a companion to "Writing Device Drivers for Oracle Solaris 11.2 ". *Writing Device Drivers* is a thorough reference document that discusses many types of devices and drivers. *Device Driver Tutorial* examines complete drivers but does not provide a comprehensive treatment of all driver types. *Device Driver Tutorial* often points to *Writing Device Drivers* and other books for further information.
- **Audience** – You should read this tutorial if you need to develop, install, and configure device drivers for the Oracle Solaris OS. You also should read this book if you need to maintain existing drivers or add new functionality to existing Oracle Solaris OS drivers. Information about the kernel provided in this book also will help you troubleshoot any problems you might encounter installing or configuring Oracle Solaris systems.
- **Required knowledge** –

  To write device drivers for the Oracle Solaris OS, you should have the following background:

  - Be an experienced C programmer
  - Have experience with data structures, especially with linked lists
  - Understand bit operations
  - Understand indirect function calls
  - Understand caching
  - Understand multithreading (see the "Multithreaded Programming Guide ")
  - Be familiar with a UNIX® shell
  - Understand the basics of UNIX system and I/O architecture

  The most important information you need to have to write a device driver are the characteristics of the device. Get a detailed specification for the device you want to drive.

  Experience with Oracle Solaris OS compilers, debuggers, and other tools will be very helpful to you. You also need to understand where the file system fits with the kernel and the application layer. These topics are discussed in this tutorial.

Product Documentation Library

Late-breaking information and known issues for this product are included in the documentation library at http://www.oracle.com/pls/topic/lookup?ctx=E36784.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

♦ ♦ ♦ CHAPTER 1

1 Introduction to Device Drivers

This chapter gives an overview of the Oracle Solaris Operating System and kernel. This chapter also gives an overview of the driver development environment and the development tools available to you.

Oracle Solaris Operating System Definition

The Oracle Solaris Operating System (Oracle Solaris OS) is implemented as an executable file that runs at boot time. The Oracle Solaris OS is referred to as the *kernel*. The kernel contains all of the routines that are necessary for the system to run. Because the kernel is essential for the running of the machine, the kernel runs in a special, protected mode that is called *kernel mode*. In contrast, user-level applications operate in a restricted mode called *user mode* that has no access to kernel instructions or to the kernel address space. Device drivers run in kernel mode and are prevented from directly accessing processes in user mode.

Kernel Overview

The kernel manages the system resources, including file systems, processes, and physical devices. The kernel provides applications with system services such as I/O management, virtual memory, and scheduling. The kernel coordinates interactions of all user processes and system resources. The kernel assigns priorities, services resource requests, and services hardware interrupts and exceptions. The kernel schedules and switches threads, pages memory, and swaps processes.

Differences Between Kernel Modules and User Programs

This section discusses several important differences between kernel modules and user programs.

Execution Differences Between Kernel Modules and User Programs

The following characteristics of kernel modules highlight important differences between the execution of kernel modules and the execution of user programs:

- **Kernel modules have separate address space.** A module runs in *kernel space*. An application runs in *user space*. System software is protected from user programs. Kernel space and user space have their own memory address spaces. For important information about address spaces, see "User and Kernel Address Spaces on x86 and SPARC Machines" on page 16.
- **Kernel modules have higher execution privilege.** Code that runs in kernel space has greater privilege than code that runs in user space. Driver modules potentially have a much greater impact on the system than user programs. Test and debug your driver modules carefully and thoroughly to avoid adverse impact on the system. For more information, see "Device Driver Testing Tips" on page 106.
- **Kernel modules do not execute sequentially.** A user program typically executes sequentially and performs a single task from beginning to end. A kernel module does not execute sequentially. A kernel module registers itself in order to serve future requests.
- **Kernel modules can be interrupted.** More than one process can request your driver at the same time. An interrupt handler can request your driver at the same time that your driver is serving a system call. In a symmetric multiprocessor (SMP) system, your driver could be executing concurrently on more than one CPU.
- **Kernel modules must be preemptable.** You cannot assume that your driver code is safe just because your driver code does not block. Design your driver assuming your driver might be preempted.
- **Kernel modules can share data.** Different threads of an application program usually do not share data. By contrast, the data structures and routines that constitute a driver are shared by all threads that use the driver. Your driver must be able to handle contention issues that result from multiple requests. Design your driver data structures carefully to keep multiple threads of execution separate. Driver code must access shared data without corrupting the data. For more information, see Chapter 3, "Multithreading," in "Writing Device Drivers for Oracle Solaris 11.2 " and "Multithreaded Programming Guide ".

Structural Differences Between Kernel Modules and User Programs

The following characteristics of kernel modules highlight important differences between the structure of kernel modules and the structure of user programs:

- **Kernel modules do not define a main program.** Kernel modules, including device drivers, have no `main` routine. Instead, a kernel module is a collection of subroutines and data. A device driver is a kernel module that forms a software interface to an input/output

(I/O) device. The subroutines in a device driver provide *entry points* to the device. The kernel uses a device number attribute to locate the `open` routine and other routines of the correct device driver. See “Device Drivers” on page 16 for more information on entry points. See “Device Numbers” on page 22 for a description of device numbers.

- **Kernel modules are linked only to the kernel.** Kernel modules do not link in the same libraries that user programs link in. The only functions a kernel module can call are functions that are exported by the kernel. If your driver references symbols that are not defined in the kernel, your driver will compile but will fail to load. Oracle Solaris OS driver modules should use prescribed DDI/DKI (Device Driver Interface, Driver-Kernel Interface) interfaces. When you use these standard interfaces you can upgrade to a new Oracle Solaris release or migrate to a new platform without recompiling your driver. For more information on the DDI, see“DDI/DKI Interfaces” in “Writing Device Drivers for Oracle Solaris 11.2 ”. Kernel modules can depend on other kernel modules by using the `-N` option during link editing. See the `ld`(1) man page for more information.
- **Kernel modules use different header files.** Kernel modules require a different set of header files than user programs require. The required header files are listed in the man page for each function. See “man pages section 9: DDI and DKI Kernel Functions ” for DDI/DKI functions, “man pages section 9: DDI and DKI Driver Entry Points ” for entry points, and “man pages section 9: DDI and DKI Properties and Data Structures ” for structures. Kernel modules can include header files that are shared by user programs if the user and kernel interfaces within such shared header files are defined conditionally using the `_KERNEL` macro.
- **Kernel modules should avoid global variables.** Avoiding global variables in kernel modules is even more important than avoiding global variables in user programs. As much as possible, declare symbols as `static`. When you must use global symbols, give them a prefix that is unique within the kernel. Using this prefix for private symbols within the module also is a good practice.
- **Kernel modules can be customized for hardware.** Kernel modules can dedicate process registers to specific roles. Kernel code can be optimized for a specific processor.
- **Kernel modules can be dynamically loaded.** The collection of subroutines and data that constitute a device driver can be compiled into a single loadable module of object code. This loadable module can then be statically or dynamically linked into the kernel and unlinked from the kernel. You can add functionality to the kernel while the system is up and running. You can test new versions of your driver without rebooting your system.

Data Transfer Differences Between Kernel Modules and User Programs

Data transfer between a device and the system typically is slower than data transfer within the CPU. Therefore, a driver typically suspends execution of the calling thread until the data transfer is complete. While the thread that called the driver is suspended, the CPU is free to execute other threads. When the data transfer is complete, the device sends an interrupt. The driver handles the interrupt that the driver receives from the device. The driver then tells the

CPU to resume execution of the calling thread. See Chapter 8, “Interrupt Handlers,” in “Writing Device Drivers for Oracle Solaris 11.2 ”.

Drivers must work with user process (virtual) addresses, system (kernel) addresses, and I/O bus addresses. Drivers sometimes copy data from one address space to another address space and sometimes just manipulate address-mapping tables. See “Bus Architectures” in “Writing Device Drivers for Oracle Solaris 11.2 ”.

User and Kernel Address Spaces on x86 and SPARC Machines

On SPARC machines, the system panics when a kernel module attempts to directly access user address space. You must make sure your driver does not attempt to directly access user address space on a SPARC machine.

On x86 machines, the system does not enter an error state when a kernel module attempts to directly access user address space. You still should make sure your driver does not attempt to directly access user address space on an x86 machine. Drivers should be written to be as portable as possible. Any driver that directly accesses user address space is a poorly written driver.

Caution - A driver that works on an x86 machine might not work on a SPARC machine because the driver might access an invalid address.

Do not access user data directly. A driver that directly accesses user address space is using poor programming practice. Such a driver is not portable and is not supportable. Use the `ddi_copyin`(9F) and `ddi_copyout`(9F) routines to transfer data to and from user address space. These two routines are the only supported interfaces for accessing user memory. “Modifying Data Stored in Kernel Memory” on page 77 shows an example driver that uses `ddi_copyin`(9F) and `ddi_copyout`(9F).

The `mmap`(2) system call maps pages of memory between a process's address space and a file or shared memory object. In response to an `mmap`(2) system call, the system calls the `devmap`(9E) entry point to map device memory into user space. This information is then available for direct access by user applications.

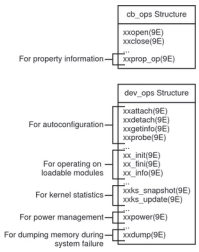
Device Drivers

A device driver is a loadable kernel module that manages data transfers between a device and the OS. Loadable modules are loaded at boot time or by request and are unloaded by request.

A device driver is a collection of C routines and data structures that can be accessed by other kernel modules. These routines must use standard interfaces called *entry points*. Through the use of entry points, the calling modules are shielded from the internal details of the driver. See "Device Driver Entry Points" in "Writing Device Drivers for Oracle Solaris 11.2 " for more information on entry points.

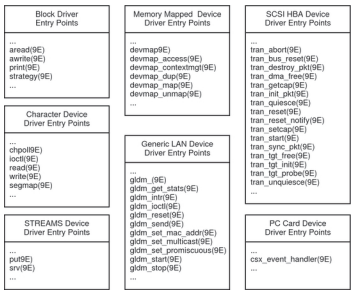
A device driver declares its general entry points in its `dev_ops`(9S) structure. A driver declares entry points for routines that are related to character or block data in its `cb_ops`(9S) structure. Some entry points and structures that are common to most drivers are shown in the following diagram.

FIGURE 1-1 Typical Device Driver Entry Points



The Oracle Solaris OS provides many driver entry points. Different types of devices require different entry points in the driver. The following diagram shows some of the available entry points, grouped by driver type. No single device driver would use all the entry points shown in the diagram.

FIGURE 1-2 Entry Points for Different Types of Drivers



In the Oracle Solaris OS, drivers can manage physical devices, such as disk drives, or software (pseudo) devices, such as bus nexus devices or ramdisk devices. In the case of hardware

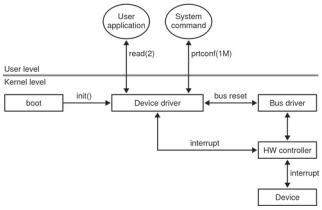
devices, the device driver communicates with the hardware controller that manages the device. The device driver shields the user application layer from the details of a specific device so that application level or system calls can be generic or device independent.

Drivers are accessed in the following situations:

- **System initialization.** The kernel calls device drivers during system initialization to determine which devices are available and to initialize those devices.
- **System calls from user processes.** The kernel calls a device driver to perform I/O operations on the device such as `open`(2), `read`(2), and `ioctl`(2).
- **User-level requests.** The kernel calls device drivers to service requests from commands such as `prtconf`(1M).
- **Device interrupts.** The kernel calls a device driver to handle interrupts generated by a device.
- **Bus reset.** The kernel calls a device driver to re-initialize the driver, the device, or both when the bus is reset. The bus is the path from the CPU to the device.

The following diagram illustrates how a device driver interacts with the rest of the system.

FIGURE 1-3 Typical Device Driver Interactions



Driver Directory Organization

Device drivers and other kernel modules are organized into the following directories in the Oracle Solaris OS. See the `kernel`(1M) and `system`(4) man pages for more information about kernel organization and how to add directories to your kernel module search path.

`/kernel`	These modules are common across most platforms. Modules that are required for booting or for system initialization belong in this directory.
`/platform/`\`uname -i\``/kernel`	These modules are specific to the platform identified by the command `uname -i`.

`/platform/`uname -m`/kernel`	These modules are specific to the platform identified by the command `uname -m`. These modules are specific to a hardware class but more generic than modules in the `uname -i` kernel directory.
`/usr/kernel`	These are user modules. Modules that are not essential to booting belong in this directory. This tutorial instructs you to put all your drivers in the `/usr/kernel` directory.

One benefit of organizing drivers into different directories is that you can selectively load different groups of drivers on startup when you boot interactively at the boot prompt as shown in the following example. See the `boot`(1M) man page for more information.

```
Type    b [file-name] [boot-flags] <ENTER>      to boot with options
or      i <ENTER>                               to enter boot interpreter
or      <ENTER>                                 to boot with defaults

                  <<< timeout in 5 seconds >>>

Select (b)oot or (i)nterpreter: b -a
bootpath: /pci@0,0/pci8086,2545@3/pci8086,
Enter default directory for modules [/platform/i86pc/kernel /kernel
/usr/kernel]: /platform/i86pc/kernel /kernel
```

In this example, the `/usr/kernel` location is omitted from the list of directories to search for modules to load. You might want to do this if you have a driver in `/usr/kernel` that causes the kernel to panic during startup or on attach. Instead of omitting all `/usr/kernel` modules, a better method for testing drivers is to put them in their own directory. Use the `moddir` kernel variable to add this test directory to your kernel modules search path. The `moddir` kernel variable is described in `kernel`(1M) and `system`(4). Another method for working with drivers that might have startup problems is described in "Device Driver Testing Tips" on page 106.

Devices as Files

In UNIX, almost everything can be treated as a file. UNIX user applications access devices as if the devices were files. Files that represent devices are called *special files* or *device nodes*. Device special files are divided into two classes: *block* devices and *character* devices. See "Character and Block Devices" on page 21 for more information.

Every I/O service request initially refers to a named file. Most I/O operations that read or write data perform equally well on ordinary or special files. For example, the same `read`(2) system call reads bytes from a file created with a text editor and reads bytes from a terminal device.

Control signals also are handled as files. Use the `ioctl`(9E) function to manipulate control signals.

Devices Directories

The Oracle Solaris OS includes both `/dev` and `/devices` directories for device drivers. Almost all the drivers in the `/dev` directory are links to the `/devices` directory. The `/dev` directory is UNIX standard. The `/devices` directory is specific to the Oracle Solaris OS.

By convention, file names in the `/dev` directory are more readable. For example, the `/dev` directory might contain files with names such as `kdb` and `mouse` that are links to files such as `/devices/pseudo/conskbd@0:kbd` and `/devices/pseudo/consms@0:mouse`. The `prtconf`(1M) command shows device names that are very similar to the file names in the `/devices` directory. In the following example, only selected output of the command is shown.

```
% prtconf -P
        conskbd, instance #0
        consms, instance #0
```

Entries in the `/dev` directory that are not links to the `/devices` directory are device nodes or special files created by `mknod`(1M) or `mknod`(2). These are zero-length files that just have a major number and minor number attached to them. Linking to the physical name of the device in the `/devices` directory is preferred to using `mknod`(1M).

Prior to the Oracle Solaris 10 OS, `/devices` was an on-disk filesystem composed of subdirectories and files. Beginning with the Oracle Solaris 10 OS, `/devices` is a virtual filesystem that creates these subdirectories and special files on demand.

For more information about the devices file system, see the `devfs`(7FS) man page.

Device Tree

The device files in the `/devices` directory are also called the *device tree*.

The device tree shows relationships among devices. In the device tree, a directory represents a *nexus* device. A nexus is a device that can be a parent of other devices. In the following example, `pci@1f,0` is a nexus device. Only selected output from the command is shown.

```
# ls -l /devices
drwxr-xr-x   4 root     sys          512 date time pci@1f,0/
crw-------   1 root     sys      111,255 date time pci@1f,0:devctl
```

You can use `prtconf`(1M) or `prtpicl`(1M) to see a graphic representation of the device tree. See "Overview of the Device Tree" in "Writing Device Drivers for Oracle Solaris 11.2 " for more information about the device tree.

Character and Block Devices

A file in the device tree that is not a directory represents either a *character* device or a *block* device.

A block device can contain addressable, reusable data. An example of a block device is a file system. Any device can be a character device. Most block devices also have character interfaces. Disks have both block and character interfaces. In your `/devices/pseudo` directory, you might find devices such as the following:

```
brw-r-----   1 root     sys       85,  0 Nov  3 09:43 md@0:0,0,blk
crw-r-----   1 root     sys       85,  0 Nov  3 09:43 md@0:0,0,raw
brw-r-----   1 root     sys       85,  1 Nov  3 09:43 md@0:0,1,blk
crw-r-----   1 root     sys       85,  1 Nov  3 09:43 md@0:0,1,raw
brw-r-----   1 root     sys       85,  2 Nov  3 09:43 md@0:0,2,blk
crw-r-----   1 root     sys       85,  2 Nov  3 09:43 md@0:0,2,raw
```

Block devices have a `b` as the first character of their file mode. Character devices have a `c` as the first character of their file mode. In this example, the block devices have `blk` in their names and the character devices have `raw` in their names.

The `md`(7D) device is a metadevice that provides disk services. The block devices access the disk using the system's normal buffering mechanism. The character devices provide for direct transmission between the disk and the user's read or write buffer.

Device Names

This section shows a complex device name and explains the meaning of each part of the name in `/dev` and also in `/devices`. The following example is the name of a disk slice:

```
/dev/dsk/c0t0d0s7 -> ../../devices/pci@1c,600000/scsi@2/sd@0,0:h
```

First, examine the name of the file in the `/dev` directory. These names are managed by the `devfsadmd`(1M) daemon.

`c0`	Controller 0
`t0`	Target 0. On SCSI controllers, this value is the disk number.
`d0`	SCSI LUN. This value indicates a virtual partitioning of a target or single physical device.
`s7`	Slice 7 on the target 0 disk.

For the same device, compare the name of the file in the `/devices` directory. These names show the physical structure and real device names. Note that some of the components of the device name in the `/devices` directory are subdirectories.

`pci@1c,600000`	PCI bus at address `1c,600000`. These addresses are meaningful only to the parent device.
`scsi@2`	SCSI controller at address `2` on the PCI bus at address `1c,600000`. This name corresponds to the `c0` in `/dev/dsk/c0t0d0s7`.
`sd@0,0`	SCSI disk at address `0,0` on the SCSI controller at address `2`. This name represents target 0, LUN 0 and corresponds to the `t0d0` in `/dev/dsk/c0t0d0s7`. The `sd` name and driver can also apply to IDE CD-ROM devices.
`sd@0,0:h`	Minor node `h` on the SCSI disk at address `0,0`. This name corresponds to the `s7` in `/dev/dsk/c0t0d0s7`.

Device Numbers

A *device number* identifies a particular device and minor node in the device tree. The `dev_t` parameter that is required in many DDI/DKI routines is this device number.

Each device has a major number and a minor number. A device number is a *major,minor* pair. A long file listing shows the device number in the column where file sizes are usually listed. In the following example, the device number is 86,255. The device major number is 86, and the device minor number is 255.

```
% ls -l /devices/pci@0,0:devctl
crw-------   1 root     sys       86,255 date time /devices/pci@0,0:devctl
```

In the Oracle Solaris OS, the major number is chosen for you when you install the driver so that it will not conflict with any other major number. The kernel uses the major number to associate the I/O request with the correct driver code. The kernel uses this association to decide which driver to execute when the user reads or writes the device file. All devices and their major numbers are listed in the file `/etc/name_to_major`.

```
% grep 86 /etc/name_to_major
pci 86
```

The minor number is assigned in the driver. The minor number must map each driver to a specific device instance. Minor numbers usually refer to sub-devices. For example, a disk driver might communicate with a hardware controller device that has several disk drives attached. Minor nodes do not necessarily have a physical representation.

The following example shows instances 0, 1, and 2 of the `md` device. The numbers 0, 1, and 2 are the minor numbers.

```
brw-r-----   1 root     sys       85,  0 Nov  3 09:43 md@0:0,0,blk
crw-r-----   1 root     sys       85,  0 Nov  3 09:43 md@0:0,0,raw
brw-r-----   1 root     sys       85,  1 Nov  3 09:43 md@0:0,1,blk
crw-r-----   1 root     sys       85,  1 Nov  3 09:43 md@0:0,1,raw
brw-r-----   1 root     sys       85,  2 Nov  3 09:43 md@0:0,2,blk
crw-r-----   1 root     sys       85,  2 Nov  3 09:43 md@0:0,2,raw
```

In the name `sd@0,0:h,`, `h` represents a minor node. When the driver receives a request for minor node `h`, the driver actually receives a corresponding minor number. The driver for the `sd` node interprets that minor number to be a particular section of disk, such as slice 7 mounted on `/export`.

Chapter 2, "Template Driver Example" shows how to use the `ddi_get_instance`(9F) routine in your driver to get an instance number for the device you are driving.

Development Environment and Tools

This section summarizes the driver development process and provides some pointers to resources. For more information on the development process, see "Driver Development Summary" in "Writing Device Drivers for Oracle Solaris 11.2 ".

Oracle offers training courses in Oracle Solaris OS internals, crash dump analysis, writing device drivers, DTrace, Oracle Solaris Studio, and other topics useful to Oracle Solaris developers. See http://education.oracle.com/ for more information.

The general steps in writing a device driver are as follows:

1. Write a `.c` source file using the interfaces and structures defined in man page sections `9E`, `9F`, and `9S`. Most of the include files you need are in `/usr/include/sys`. The function and structure man pages show which include files you need.
2. Write a `.conf` hardware configuration file to define property values for your driver.
3. Compile and link your driver. Always use the `-D_KERNEL` option when you compile a driver for the Oracle Solaris OS. The default compile result is 32-bit. To get a 64-bit result on a 64-bit platform, specify the appropriate 64-bit option as described in "Building a Driver" on page 25.
4. Copy your driver binary file and your driver configuration file to the appropriate `[`*platform*`]/kernel` directories. See "Driver Directory Organization" on page 18 for descriptions of driver directories.
5. Use the `add_drv`(1M) command to load your driver. When your driver is loaded, you can see your driver in `/dev` and `/devices`. You can also see an entry for your driver in the `/etc/name_to_major` file.

Writing a Driver

A driver consists of a C source file and a hardware configuration file.

Writing a Driver Module

The C code for a driver is a collection of data and functions that define a kernel module. As noted in "Structural Differences Between Kernel Modules and User Programs" on page 14, a driver has no `main` routine. Many of the subroutines of a driver are special functions called entry points. See "Device Drivers" on page 16 for information about entry points.

The function man pages provide both the function declaration that you need in your driver and the list of header files you need to include. Make sure you consult the correct man page. For example, the following command displays the `ioctl`(2) man page. The `ioctl`(2) system call cannot be used in a device driver.

```
% man ioctl
```

Use one of the following commands to display the `ioctl`(9E) man page. The `ioctl`(9E) subroutine is a device driver entry point.

```
% man ioctl.9e
% man -s 9e ioctl
```

By convention, the names of functions and data that are unique to this driver begin with a common prefix. The prefix is the name of this driver or an abbreviation of the name of this driver. Use the same prefix for all names that are specific to this driver. This practice makes debugging much easier. Instead of seeing an error related to an ambiguous `attach` function, you see an error message about `mydriver_attach` or `newdriver_attach`.

A 64-bit system can run both 32-bit user programs and 64-bit user programs. A 64-bit system runs 32-bit programs by converting all data needed between the two data models. A 64-bit kernel supports both 64-bit and 32-bit user data. Whenever a 64-bit driver copies data between kernel space and user space, the driver must use the `ddi_model_convert_from`(9F) function to determine whether the data must be converted between 32-bit and 64-bit models. For an example, see "Reporting and Setting Device Size and Re-initializing the Device" on page 85.

The Oracle Solaris Studio IDE includes the following three source editors: GVIM, XEmacs, and the built-in Source Editor provided by NetBeans. The IDE provides online help for these tools. You can also run GVIM and XEmacs from the command line. See `vim`(1) and `xemacs`(1).

For more information, see the following resources:

- For more information about writing device drivers, see "Device Driver Coding Tips" on page 103 and "Writing Device Drivers for Oracle Solaris 11.2 ".

- For simple example source files, see Chapter 2, “Template Driver Example” and Chapter 3, “Reading and Writing Data in Kernel Memory”.

Writing a Configuration File

A driver that is not self-identifying might need a configuration file named *node_name*`.conf`, where *node_name* is the prefix for the device. A self-identifying driver is a driver that can obtain all the property information it needs from the DDI property interfaces such as `ddi_prop_get_int`(9F) and `ddi_prop_lookup`(9F). The minimum information that a configuration file must contain is the name of the device node and the name or type of the device's parent.

On the x86 platform, device information is supplied by the booting system. Hardware configuration files should no longer be needed, even for non-self-identifying devices.

For more information about device driver configuration files, see the `driver.conf`(4) man page. For an example configuration file, see “Writing the Device Configuration File” on page 54.

Building a Driver

This section tells you how to compile and link a driver for different architectures.

Make sure you have installed the Oracle Solaris OS at the Developer level or above.

A 64-bit kernel cannot use a 32-bit driver. A 64-bit kernel can use only 64-bit drivers. All parts of any particular program must use the same data model. A device driver is not a complete program. The kernel is a complete program. A driver is a part of the kernel program. If you want your device to work with the Oracle Solaris OS in 32-bit mode and in 64-bit mode, then you must provide both a 32-bit driver and a 64-bit driver.

By default, compilation on the operating system yields a 32-bit result on every architecture. To obtain a 64-bit result, use the compilation options specified in this section for 64-bit architectures.

Use the `prtconf`(1M) command with the `-x` option to determine whether the firmware on this system is 64-bit ready.

Compiling with Oracle Solaris Studio

Use the `-D_KERNEL` option to indicate that this code defines a kernel module.

- If you are compiling for a 64-bit SPARC architecture using Sun Studio 9, Sun Studio 10, or Sun Studio 11, use the `-xarch=v9` option:

```
% cc -D_KERNEL -xarch=v9 -c mydriver.c
% ld -r -o mydriver mydriver.o
```

  If you are compiling for a 64-bit SPARC architecture using Oracle Solaris Studio 12, use the `-m64` option:

```
% cc -D_KERNEL -m64 -c mydriver.c
% ld -r -o mydriver mydriver.o
```

- If you are compiling for a 64-bit x86 architecture using Sun Studio 10 or Sun Studio 11, use both the `-xarch=amd64` option and the `-xmodel=kernel` option:

```
% cc -D_KERNEL -xarch=amd64 -xmodel=kernel -c mydriver.c
% ld -r -o mydriver mydriver.o
```

  If you are compiling for a 64-bit x86 architecture using Oracle Solaris Studio 12, use the `-m64` option, the `-xarch=sse2a` option, and the `-xmodel=kernel` option:

```
% cc -D_KERNEL -m64 -xarch=sse2a -xmodel=kernel -c mydriver.c
% ld -r -o mydriver mydriver.o
```

- If you are compiling for a 32-bit architecture, use the following build commands:

```
% cc -D_KERNEL -c mydriver.c
% ld -r -o mydriver mydriver.o
```

Note - Sun Studio 9 does not support 64-bit x86 architectures. Use Sun Studio 10, Sun Studio 11, or Oracle Solaris Studio 12 to compile and debug drivers for 64-bit x86 architectures.

For more information on compile and link options, see the Oracle Solaris Studio 12.3 Command-line Reference and the Oracle Solaris Studio 12.2: C User's Guide . To learn more about Oracle Solaris Studio, go to http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html.

Compiling with the GNU C Compiler

Use the `-D_KERNEL` option to indicate that this code defines a kernel module. These examples show options that are required for correct functionality of the result.

- If you are compiling for a 64-bit SPARC architecture, use the following build commands:

```
% gcc -D_KERNEL -m64 -mcpu=v9 -mcmodel=medlow -fno-pic -mno-fpu
-ffreestanding -nodefaultlibs -c mydriver.c
```

```
% ld -r -o mydriver mydriver.o
```

You might also want to use the `-mtune=ultrasparc` option and the `-O2` option.

- If you are compiling for a 64-bit x86 architecture, use the following build commands:

```
% gcc -D_KERNEL -m64 -mcmodel=kernel -mno-red-zone -ffreestanding
-nodefaultlibs -c mydriver.c
% ld -r -o mydriver mydriver.o
```

You might also want to use the `-mtune=opteron` option and the `-O2` option.

- If you are compiling for a 32-bit architecture, use the following build commands:

```
% gcc -D_KERNEL -ffreestanding -nodefaultlibs -c mydriver.c
% ld -r -o mydriver mydriver.o
```

For more information on these and other options, see the `gcc`(1) man page. See also the GCC web site at http://gcc.gnu.org/.

Installing a Driver

After you write and build your driver, you must install the driver binary. To install a driver, copy the driver binary and the configuration file to the appropriate `/kernel/drv` directory.

Make sure you are user `root` when you install a driver.

Copy the configuration file to the kernel driver area of the system.

```
# cp mydriver.conf /usr/kernel/drv
```

Install drivers in the `/tmp` directory until you are finished modifying and testing the `_info`, `_init`, and `attach` routines. See “Device Driver Testing Tips” on page 106 for more information.

Copy the driver binary to the `/tmp` directory.

```
# cp mydriver /tmp
```

Link to the driver from the kernel driver directory.

- On a 64-bit SPARC architecture, link to the `sparcv9` directory:

```
# ln -s /tmp/mydriver /usr/kernel/drv/sparcv9/mydriver
```

- On a 64-bit x86 architecture, link to the `amd64` directory:

```
# ln -s /tmp/mydriver /usr/kernel/drv/amd64/mydriver
```

- On a 32-bit architecture, create the link as follows:

```
# ln -s /tmp/mydriver /usr/kernel/drv/mydriver
```

When the driver is well tested, copy the driver directly to the appropriate kernel driver area of the system.

- On a 64-bit SPARC architecture, copy the driver to the `sparcv9` directory:

  ```
  # cp mydriver /usr/kernel/drv/sparcv9/mydriver
  ```

- On a 64-bit x86 architecture, copy the driver to the `amd64` directory:

  ```
  # cp mydriver /usr/kernel/drv/amd64/mydriver
  ```

- On a 32-bit architecture, copy the driver to the kernel driver area of the system:

  ```
  # cp mydriver /usr/kernel/drv/mydriver
  ```

Adding, Updating, and Removing a Driver

Use the `add_drv`(1M) command to make the installed driver usable. Be sure you are user `root` when you use the `add_drv`(1M) command.

```
# add_drv mydriver
```

The following events take place when you add a driver:

- The `_info`(9E), `_init`(9E), and `attach`(9E) entry points are called in that order.
- The driver is added to the `/devices` directory.
- The driver is the most recent module listed by `modinfo`(1M).
- The driver is the most recent module listed in the file `/etc/name_to_major`.

The file `/etc/driver_aliases` might be updated. The `/etc/driver_aliases` file shows which devices are bound to which drivers. If a driver is not listed in the `/etc/driver_aliases` file, then the OS does not load that driver or attach to that driver. Each line of the `/etc/driver_aliases` file shows a driver name followed by a device name. You can search this file to determine which driver is managing your device.

Note - Do not edit the `/etc/driver_aliases` file manually. Use the `add_drv`(1M) command to establish a device binding. Use the `update_drv`(1M) command to change a device binding.

The example drivers shown in this book manage pseudo devices. If your driver manages real hardware, then you need to use the `-c` and `-i` options on the `add_drv`(1M) command or

the -i option on the update_drv(1M) command. To specify a device class or device ID, you might find the following sites useful. This information also is useful to search the /etc/driver_aliases file to find out whether a device already is supported.

- List of devices currently supported by the OS: http://www.oracle.com/webfolder/technetwork/hcl/index.html
- Searchable PCI vendor and device lists: http://www.pcidatabase.com/
- Repository of vendor IDs, device IDs, subsystems, and device classes used in PCI devices: http://pciids.sourceforge.net/

Use the update_drv(1M) command to notify the system about attribute changes to an installed device driver. By default, the update_drv(1M) command reloads the hardware configuration file for the specified driver. Use the prtconf(1M) command to review the current configuration information for a device and driver. For example, the -D option shows which driver manages a particular device. The -P option shows information about pseudo devices.

Use the rem_drv(1M) command to update the system driver configuration files so that the driver is no longer usable. The rem_drv(1M) command does not physically delete driver files. If possible, the rem_drv(1M) command unloads the driver from memory.

Loading and Unloading a Driver

A driver is loaded into memory when a device that the driver manages is accessed. A driver might be unloaded from memory when the driver is not being used. Normally, you do not need to load a driver into memory manually or unload a driver from memory manually.

To manually load a loadable module into memory, use the modload(1M) command.

While you are developing your driver, you might want to manually unload the driver and then update the driver. To manually unload a loadable module from memory, use the modunload(1M) command.

Testing a Driver

Drivers should be thoroughly tested in the following areas:

- Configuration
- Functionality
- Error handling

- Loading, unloading, and removing

  All drivers will need to be removed eventually. Make sure that your driver can be successfully removed.
- Stress, performance, and interoperability
- DDI/DKI compliance
- Installation and packaging

For detailed information on how to test your driver and how to avoid problems during testing, see the following references:

- “Device Driver Testing Tips” on page 106
- “Criteria for Testing Drivers” in “Writing Device Drivers for Oracle Solaris 11.2 ”
- Chapter 23, “Debugging, Testing, and Tuning Device Drivers,” in “Writing Device Drivers for Oracle Solaris 11.2 ”

Additional testing is specific to the type of driver.

♦ ♦ ♦ CHAPTER 2

Template Driver Example

This chapter shows you how to develop a very simple, working driver. This chapter explains how to write the driver and configuration file, compile the driver, load the driver, and test the driver.

The driver that is shown in this chapter is a pseudo device driver that merely writes a message to a system log every time an entry point is entered. This driver demonstrates the minimum functionality that any character driver must implement. You can use this driver as a template for building a complex driver.

This chapter discusses the following driver development steps:

- "Overview of the Template Driver Example" on page 31
- "Writing the Template Driver" on page 32
- "Writing the Device Configuration File" on page 54
- "Building and Installing the Template Driver" on page 54
- "Testing the Template Driver" on page 55
- "Complete Template Driver Source" on page 58

Overview of the Template Driver Example

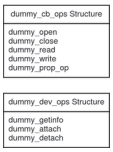
This example guides you through the following steps:

1. Create a directory where you can develop your driver and open a new text file named `dummy.c`.
2. Write the entry points for loadable module configuration: `_init`(9E), `_info`(9E), and `_fini`(9E).
3. Write the entry points for autoconfiguration: `attach`(9E), `detach`(9E), `getinfo`(9E), and `prop_op`(9E).
4. Write the entry points for user context: `open`(9E), `close`(9E), `read`(9E), and `write`(9E).
5. Define the data structures: the character and block operations structure `cb_ops`(9S), the device operations structure `dev_ops`(9S), and the module linkage structures `modldrv`(9S) and `modlinkage`(9S).

6. Create the driver configuration file `dummy.conf`.
7. Build and install the driver.
8. Test the driver by loading the driver, reading from and writing to the device node, and unloading the driver.

The entry points that are to be created in this example are shown in the following diagram.

FIGURE 2-1 Entry Points for the `dummy` Example



Writing the Template Driver

This section describes the entry points and data structures that are included in this driver and shows you how to define them. All of these data structures and almost all of these entry points are required for any character device driver.

This section describes the following entry points and data structures:

- Loadable module configuration entry points
- Autoconfiguration entry points
- User context entry points
- Character and block operations structure
- Device operations structure
- Module linkage structures

First, create a directory where you can develop your driver. This driver is named `dummy` because this driver does not do any real work. Next, open a new text file named `dummy.c`.

Writing the Loadable Module Configuration Entry Points

Every kernel module of any type must define at least the following three loadable module configuration entry points:

- The _init(9E) routine initializes a loadable module. The _init(9E) routine must at least call the mod_install(9F) function and return the success or failure value that is returned by mod_install(9F).
- The _info(9E) routine returns information about a loadable module. The _info(9E) routine must at least call the mod_info(9F) function and return the value that is returned by mod_info(9F).
- The _fini(9E) routine prepares a loadable module for unloading. The _fini(9E) routine must at least call the mod_remove(9F) function and return the success or failure value that is returned by mod_remove(9F). When mod_remove(9F) is successful, the _fini(9E) routine must undo everything that the _init(9E) routine did.

The mod_install(9F), mod_info(9F), and mod_remove(9F) functions are used in exactly the same way in every driver, regardless of the functionality of the driver. You do not need to investigate what the values of the arguments of these functions should be. You can copy these function calls from this example and paste them into every driver you write.

In this section, the following code is added to the dummy.c source file:

```
/* Loadable module configuration entry points */
int
_init(void)
{
    cmn_err(CE_NOTE, "Inside _init");
    return(mod_install(&ml));
}

int
_info(struct modinfo *modinfop)
{
    cmn_err(CE_NOTE, "Inside _info");
    return(mod_info(&ml, modinfop));
}

int
_fini(void)
{
    cmn_err(CE_NOTE, "Inside _fini");
    return(mod_remove(&ml));
}
```

Declaring the Loadable Module Configuration Entry Points

The _init(9E), _info(9E), and _fini(9E) routine names are not unique to any particular kernel module. You customize the behavior of these routines when you define them in your module, but the names of these routines are not unique. These three routines are declared in the modctl.h header file. You need to include the modctl.h header file in your dummy.c file. Do not declare these three routines in dummy.c.

Defining the Module Initialization Entry Point

The _init(9E) routine returns type int and takes no arguments. The _init(9E) routine must call the mod_install(9F) function and return the success or failure value that is returned by mod_install(9F).

The mod_install(9F) function takes an argument that is a modlinkage(9S) structure. See "Defining the Module Linkage Structures" on page 52 for information about the modlinkage(9S) structure.

This driver is supposed to write a message each time an entry point is entered. Use the cmn_err(9F) function to write a message to a system log. The cmn_err(9F) function usually is used to report an error condition. The cmn_err(9F) function also is useful for debugging in the same way that you might use print statements in a user program. Be sure to remove cmn_err calls that are used for development or debugging before you compile your production version driver. You might want to use cmn_err calls in a production driver to write error messages that would be useful to a system administrator.

The cmn_err(9F) function requires you to include the cmn_err.h header file, the ddi.h header file, and the sunddi.h header file. The cmn_err(9F) function takes two arguments. The first argument is a constant that indicates the severity of the error message. The message written by this driver is not an error message but is simply a test message. Use CE_NOTE for the value of this severity constant. The second argument the cmn_err(9F) function takes is a string message.

The following code is the _init(9E) routine that you should enter into your dummy.c file. The ml structure is the modlinkage(9S) structure that is discussed in "Defining the Module Linkage Structures" on page 52.

```
int
_init(void)
{
    cmn_err(CE_NOTE, "Inside _init");
    return(mod_install(&ml));
}
```

Defining the Module Information Entry Point

The _info(9E) routine returns type int and takes an argument that is a pointer to an opaque modinfo structure. The _info(9E) routine must return the value that is returned by the mod_info(9F) function.

The mod_info(9F) function takes two arguments. The first argument to mod_info(9F) is a modlinkage(9S) structure. See “Defining the Module Linkage Structures” on page 52 for information about the modlinkage(9S) structure. The second argument to mod_info(9F) is the same modinfo structure pointer that is the argument to the _info(9E) routine. The mod_info(9F) function returns the module information or returns zero if an error occurs.

Use the cmn_err(9F) function to write a message to the system log in the same way that you used the cmn_err(9F) function in your _init(9E) entry point.

The following code is the _info(9E) routine that you should enter into your dummy.c file. The ml structure is discussed in “Defining the Module Linkage Structures” on page 52. The modinfop argument is a pointer to an opaque structure that the system uses to pass module information.

```
int
_info(struct modinfo *modinfop)
{
    cmn_err(CE_NOTE, "Inside _info");
    return(mod_info(&ml, modinfop));
}
```

Defining the Module Unload Entry Point

The _fini(9E) routine returns type int and takes no arguments. The _fini(9E) routine must call the mod_remove(9F) function and return the success or failure value that is returned by mod_remove(9F).

When mod_remove(9F) is successful, the _fini(9E) routine must undo everything that the _init(9E) routine did. The _fini(9E) routine must call mod_remove(9F) because the _init(9E) routine called mod_install(9F). The _fini(9E) routine must deallocate anything that was allocated, close anything that was opened, and destroy anything that was created in the _init(9E) routine.

The _fini(9E) routine can be called at any time when a module is loaded. In normal operation, the _fini(9E) routine often fails. This behavior is normal because the kernel allows the module to determine whether the module can be unloaded. If mod_remove(9F) is successful, the module determines that devices were detached, and the module can be unloaded. If mod_remove(9F) fails, the module determines that devices were not detached, and the module cannot be unloaded.

The following actions take place when mod_remove(9F) is called:

- The kernel checks whether this driver is busy. This driver is busy if one of the following conditions is true:
  - A device node that is managed by this driver is open.
  - Another module that depends on this driver is open. A module depends on this driver if the module was linked using the `-N` option with this driver named as the argument to that `-N` option. See the `ld`(1) man page for more information.
- If the driver is busy, then `mod_remove`(9F) fails and `_fini`(9E) fails.
- If the driver is not busy, then the kernel calls the `detach`(9E) entry point of the driver.
  - If `detach`(9E) fails, then `mod_remove`(9F) fails and `_fini`(9E) fails.
  - If `detach`(9E) succeeds, then `mod_remove`(9F) succeeds, and `_fini`(9E) continues its cleanup work.

The `mod_remove`(9F) function takes an argument that is a `modlinkage`(9S) structure. See "Defining the Module Linkage Structures" on page 52 for information about the `modlinkage`(9S) structure.

Use the `cmn_err`(9F) function to write a message to the system log in the same way that you used the `cmn_err`(9F) function in your `_init`(9E) entry point.

The following code is the `_fini`(9E) routine that you should enter into your `dummy.c` file. The `ml` structure is discussed in "Defining the Module Linkage Structures" on page 52.

```
int
_fini(void)
{
    cmn_err(CE_NOTE, "Inside _fini");
    return(mod_remove(&ml));
}
```

Including Loadable Module Configuration Header Files

The `_init`(9E), `_info`(9E), `_fini`(9E), and `mod_install`(9F) functions require you to include the `modctl.h` header file. The `cmn_err`(9F) function requires you to include the `cmn_err.h` header file, the `ddi.h` header file, and the `sunddi.h` header file.

The following header files are required by the three loadable module configuration routines that you have written in this section. Include this code near the top of your `dummy.c` file.

```
#include <sys/modctl.h>  /* used by _init, _info, _fini */
#include <sys/cmn_err.h> /* used by all entry points for this driver */
#include <sys/ddi.h>     /* used by all entry points for this driver */
#include <sys/sunddi.h>  /* used by all entry points for this driver */
```

Writing the Autoconfiguration Entry Points

Every character driver must define at least the following autoconfiguration entry points. The kernel calls these routines when the device driver is loaded.

- The attach(9E) routine must call ddi_create_minor_node(9F). The ddi_create_minor_node(9F) function provides the information the system needs to create the device files.
- The detach(9E) routine must call ddi_remove_minor_node(9F) to deallocate everything that was allocated by ddi_create_minor_node(9F). The detach(9E) routine must undo everything that the attach(9E) routine did.
- The getinfo(9E) routine returns requested device driver information through one of its arguments.
- The prop_op(9E) routine returns requested device driver property information through a pointer. You can call the ddi_prop_op(9F) function instead of writing your own prop_op(9E) entry point. Use the prop_op(9E) entry point to customize the behavior of the ddi_prop_op(9F) function.

In this section, the following code is added:

```
/* Device autoconfiguration entry points */
static int
dummy_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    cmn_err(CE_NOTE, "Inside dummy_attach");
    switch(cmd) {
    case DDI_ATTACH:
        dummy_dip = dip;
        if (ddi_create_minor_node(dip, "0", S_IFCHR,
            ddi_get_instance(dip), DDI_PSEUDO,0)
            != DDI_SUCCESS) {
            cmn_err(CE_NOTE,
                "%s%d: attach: could not add character node.",
                "dummy", 0);
            return(DDI_FAILURE);
        } else
            return DDI_SUCCESS;
    default:
        return DDI_FAILURE;
    }
}

static int
dummy_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    cmn_err(CE_NOTE, "Inside dummy_detach");
    switch(cmd) {
    case DDI_DETACH:
```

```
        dummy_dip = 0;
        ddi_remove_minor_node(dip, NULL);
        return DDI_SUCCESS;
    default:
        return DDI_FAILURE;
    }
}

static int
dummy_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
    void **resultp)
{
    cmn_err(CE_NOTE, "Inside dummy_getinfo");
    switch(cmd) {
    case DDI_INFO_DEVT2DEVINFO:
        *resultp = dummy_dip;
        return DDI_SUCCESS;
    case DDI_INFO_DEVT2INSTANCE:
        *resultp = 0;
        return DDI_SUCCESS;
    default:
        return DDI_FAILURE;
    }
}

static int
dummy_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp)
{
    cmn_err(CE_NOTE, "Inside dummy_prop_op");
    return(ddi_prop_op(dev,dip,prop_op,flags,name,valuep,lengthp));
}
```

Declaring the Autoconfiguration Entry Points

The `attach`(9E), `detach`(9E), `getinfo`(9E), and `prop_op`(9E) entry point routines need to be uniquely named for this driver. Choose a prefix to use with each entry point routine.

Note - By convention, the prefix used for function and data names that are unique to this driver is either the name of this driver or an abbreviation of the name of this driver. Use the same prefix throughout the driver. This practice makes debugging much easier.

In the example shown in this chapter, `dummy_` is used for the prefix to each function and data name that is unique to this example.

The following declarations are the autoconfiguration entry point declarations you should have in your `dummy.c` file. Note that each of these functions is declared `static`.

```
static int dummy_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int dummy_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
```

```
static int dummy_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
    void **resultp);
static int dummy_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp);
```

Defining the Device Attach Entry Point

The attach(9E) routine returns type `int`. The `attach`(9E) routine must return either `DDI_SUCCESS` or `DDI_FAILURE`. These two constants are defined in `sunddi.h`. All of the autoconfiguration entry point routines except for `prop_op`(9E) return either `DDI_SUCCESS` or `DDI_FAILURE`.

The `attach`(9E) routine takes two arguments. The first argument is a pointer to the `dev_info` structure for this driver. All of the autoconfiguration entry point routines take a `dev_info` argument. The second argument is a constant that specifies the attach type. The value that is passed through this second argument is either `DDI_ATTACH` or `DDI_RESUME`. Every `attach`(9E) routine must define behavior for at least `DDI_ATTACH`.

The `DDI_ATTACH` code must initialize a device instance. In a realistic driver, you define and manage multiple instances of the driver by using a state structure and the `ddi_soft_state`(9F) functions. Each instance of the driver has its own copy of the state structure that holds data specific to that instance. One of the pieces of data that is specific to each instance is the device instance pointer. Each instance of the device driver is represented by a separate device file in `/devices`. Each device instance file is pointed to by a separate device instance pointer. See “Managing Device State” on page 67 for information about state structures and `ddi_soft_state`(9F) functions. See “Devices as Files” on page 19 for information about device files and instances.

This `dummy` driver allows only one instance. Because this driver allows only one instance, this driver does not use a state structure. This driver still must declare a device instance pointer and initialize the pointer value in the `attach`(9E) routine. Enter the following code near the beginning of `dummy.c` to declare a device instance pointer for this driver:

```
dev_info_t *dummy_dip;  /* keep track of one instance */
```

The following code is the `dummy_attach` routine that you should enter into your `dummy.c` file. You can copy the name portion of this function definition directly from the declaration you entered in “Declaring the Autoconfiguration Entry Points” on page 38.

```
static int
dummy_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    cmn_err(CE_NOTE, "Inside dummy_attach");
    switch(cmd) {
    case DDI_ATTACH:
```

```
        dummy_dip = dip;
        if (ddi_create_minor_node(dip, "0", S_IFCHR,
            ddi_get_instance(dip), DDI_PSEUDO,0)
            != DDI_SUCCESS) {
            cmn_err(CE_NOTE,
                "%s%d: attach: could not add character node.",
                "dummy", 0);
            return(DDI_FAILURE);
        } else
            return DDI_SUCCESS;
    default:
        return DDI_FAILURE;
    }
}
```

First, use `cmn_err`(9F) to write a message to the system log, as you did in your `_init`(9E) entry point. Then provide `DDI_ATTACH` behavior. Within the `DDI_ATTACH` code, first assign the device instance pointer from the `dummy_attach` argument to the `dummy_dip` variable that you declared above. You need to save this pointer value in the global variable so that you can use this pointer to get information about this instance from `dummy_getinfo` and detach this instance in `dummy_detach`. In this `dummy_attach` routine, the device instance pointer is used by the `ddi_get_instance`(9F) function to return the instance number. The device instance pointer and the instance number both are used by `ddi_create_minor_node`(9F) to create a new device node.

A realistic driver probably would use the `ddi_soft_state`(9F) functions to create and manage a device node. This `dummy` driver uses the `ddi_create_minor_node`(9F) function to create a device node. The `ddi_create_minor_node`(9F) function takes six arguments. The first argument to the `ddi_create_minor_node`(9F) function is the device instance pointer that points to the `dev_info` structure of this device. The second argument is the name of this minor node. The third argument is `S_IFCHR` if this device is a character minor device or is `S_IFBLK` if this device is a block minor device. This `dummy` driver is a character driver.

The fourth argument to the `ddi_create_minor_node`(9F) function is the minor number of this minor device. This number is also called the instance number. The `ddi_get_instance`(9F) function returns this instance number. The fifth argument to the `ddi_create_minor_node`(9F) function is the node type. The `ddi_create_minor_node`(9F) man page lists the possible node types. The `DDI_PSEUDO` node type is for pseudo devices. The sixth argument to the `ddi_create_minor_node`(9F) function specifies whether this is a clone device. This is not a clone device, so set this argument value to 0.

If the `ddi_create_minor_node`(9F) call is not successful, write a message to the system log and return `DDI_FAILURE`. If the `ddi_create_minor_node`(9F) call is successful, return `DDI_SUCCESS`. If this `dummy_attach` routine receives any `cmd` other than `DDI_ATTACH`, return `DDI_FAILURE`.

Defining the Device Detach Entry Point

The detach(9E) routine takes two arguments. The first argument is a pointer to the dev_info structure for this driver. The second argument is a constant that specifies the detach type. The value that is passed through this second argument is either DDI_DETACH or DDI_SUSPEND. Every detach(9E) routine must define behavior for at least DDI_DETACH.

The DDI_DETACH code must undo everything that the DDI_ATTACH code did. In the DDI_ATTACH code in your attach(9E) routine, you saved the address of a new dev_info structure and you called the ddi_create_minor_node(9F) function to create a new node. In the DDI_DETACH code in this detach(9E) routine, you need to reset the variable that pointed to the dev_info structure for this node. You also need to call the ddi_remove_minor_node(9F) function to remove this node. The detach(9E) routine must deallocate anything that was allocated, close anything that was opened, and destroy anything that was created in the attach(9E) routine.

The following code is the dummy_detach routine that you should enter into your dummy.c file. You can copy the name portion of this function definition directly from the declaration you entered in "Declaring the Autoconfiguration Entry Points" on page 38.

```
static int
dummy_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    cmn_err(CE_NOTE, "Inside dummy_detach");
    switch(cmd) {
    case DDI_DETACH:
        dummy_dip = 0;
        ddi_remove_minor_node(dip, NULL);
        return DDI_SUCCESS;
    default:
        return DDI_FAILURE;
    }
}
```

First, use cmn_err(9F) to write a message to the system log, as you did in your _init(9E) entry point. Then provide DDI_DETACH behavior. Within the DDI_DETACH code, first reset the dummy_dip variable that you set in dummy_attach above. You cannot reset this device instance pointer unless you remove all instances of the device. This dummy driver supports only one instance.

Next, call the ddi_remove_minor_node(9F) function to remove this device node. The ddi_remove_minor_node(9F) function takes two arguments. The first argument is the device instance pointer that points to the dev_info structure of this device. The second argument is the name of the minor node you want to remove. If the value of the minor node argument is NULL, then ddi_remove_minor_node(9F) removes all instances of this device. Because the DDI_DETACH code of this driver always removes all instances, this dummy driver supports only one instance.

If the value of the cmd argument to this dummy_detach routine is DDI_DETACH, remove all instances of this device and return DDI_SUCCESS. If this dummy_detach routine receives any cmd other than DDI_DETACH, return DDI_FAILURE.

Defining the Get Driver Information Entry Point

The getinfo(9E) routine takes a pointer to a device number and returns a pointer to a device information structure or returns a device instance number. The return value of the getinfo(9E) routine is DDI_SUCCESS or DDI_FAILURE. The pointer or instance number requested from the getinfo(9E) routine is returned through a pointer argument.

The getinfo(9E) routine takes four arguments. The first argument is a pointer to the dev_info structure for this driver. This dev_info structure argument is obsolete and is no longer used by the getinfo(9E) routine.

The second argument to the getinfo(9E) routine is a constant that specifies what information the getinfo(9E) routine must return. The value of this second argument is either DDI_INFO_DEVT2DEVINFO or DDI_INFO_DEVT2INSTANCE. The third argument to the getinfo(9E) routine is a pointer to a device number. The fourth argument is a pointer to the place where the getinfo(9E) routine must store the requested information. The information stored at this location depends on the value you passed in the second argument to the getinfo(9E) routine.

The following table describes the relationship between the second and fourth arguments to the getinfo(9E) routine.

TABLE 2-1 Get Driver Information Entry Point Arguments

cmd	*arg*	*resultp*
DDI_INFO_DEVT2DEVINFO	Device number	Device information structure pointer
DDI_INFO_DEVT2INSTANCE	Device number	Device instance number

The following code is the dummy_getinfo routine that you should enter into your dummy.c file. You can copy the name portion of this function definition directly from the declaration you entered in “Declaring the Autoconfiguration Entry Points” on page 38.

```
static int
dummy_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
    void **resultp)
{
    cmn_err(CE_NOTE, "Inside dummy_getinfo");
```

```
    switch(cmd) {
    case DDI_INFO_DEVT2DEVINFO:
        *resultp = dummy_dip;
        return DDI_SUCCESS;
    case DDI_INFO_DEVT2INSTANCE:
        *resultp = 0;
        return DDI_SUCCESS;
    default:
        return DDI_FAILURE;
    }
}
```

First, use `cmn_err`(9F) to write a message to the system log, as you did in your `_init`(9E) entry point. Then provide `DDI_INFO_DEVT2DEVINFO` behavior. A realistic driver would use *arg* to get the instance number of this device node. A realistic driver would then call the `ddi_get_soft_state`(9F) function and return the device information structure pointer from that state structure. This `dummy` driver supports only one instance and does not use a state structure. In the `DDI_INFO_DEVT2DEVINFO` code of this `dummy_getinfo` routine, simply return the one device information structure pointer that the `dummy_attach` routine saved.

Next, provide `DDI_INFO_DEVT2INSTANCE` behavior. Within the `DDI_INFO_DEVT2INSTANCE` code, simply return 0. This `dummy` driver supports only one instance. The instance number of that one instance is 0.

Defining the Report Driver Property Information Entry Point

The `prop_op`(9E) entry point is required for every driver. If your driver does not need to customize the behavior of the `prop_op`(9E) entry point, then your driver can use the `ddi_prop_op`(9F) function for the `prop_op`(9E) entry point. Drivers that create and manage their own properties need a custom `prop_op`(9E) routine. This `dummy` driver uses a `prop_op`(9E) routine to call `cmn_err`(9F) before calling the `ddi_prop_op`(9F) function.

The `prop_op`(9E) entry point and the `ddi_prop_op`(9F) function both require that you include the `types.h` header file. The `prop_op`(9E) entry point and the `ddi_prop_op`(9F) function both take the same seven arguments. These arguments are not discussed here because this `dummy` driver does not create and manage its own properties. See the `prop_op`(9E) man page to learn about the `prop_op`(9E) arguments.

The following code is the `dummy_prop_op` routine that you should enter into your `dummy.c` file. You can copy the name portion of this function definition directly from the declaration you entered in "Declaring the Autoconfiguration Entry Points" on page 38.

```
static int
dummy_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp)
```

```
{
    cmn_err(CE_NOTE, "Inside dummy_prop_op");
    return(ddi_prop_op(dev,dip,prop_op,flags,name,valuep,lengthp));
}
```

First, use `cmn_err`(9F) to write a message to the system log, as you did in your `_init`(9E) entry point. Then call the `ddi_prop_op`(9F) function with exactly the same arguments as the `dummy_prop_op` function.

Including Autoconfiguration Header Files

All of the autoconfiguration entry point routines and all of the user context entry point routines require that you include the `ddi.h` and `sunddi.h` header files. You already included these two header files for the `cmn_err`(9F) function.

The `ddi_create_minor_node`(9F) function requires the `stat.h` header file. The `dummy_attach` routine calls the `ddi_create_minor_node`(9F) function. The `prop_op`(9E) and the `ddi_prop_op`(9F) functions require the `types.h` header file.

The following code is the list of header files that you now should have included in your `dummy.c` file for the four autoconfiguration routines you have written in this section and the three loadable module configuration routines you wrote in the previous section.

```
#include <sys/modctl.h>  /* used by _init, _info, _fini */
#include <sys/types.h>   /* used by prop_op, ddi_prop_op */
#include <sys/stat.h>    /* defines S_IFCHR used by ddi_create_minor_node */
#include <sys/cmn_err.h> /* used by all entry points for this driver */
#include <sys/ddi.h>     /* used by all entry points for this driver */
                         /* also used by ddi_get_instance, ddi_prop_op */
#include <sys/sunddi.h>  /* used by all entry points for this driver */
                         /* also used by ddi_create_minor_node, */
                         /* ddi_get_instance, and ddi_prop_op */
```

Writing the User Context Entry Points

User context entry points correspond closely to system calls. When a system call opens a device file, then the `open`(9E) routine in the driver for that device is called.

All character and block drivers must define the `open`(9E) user context entry point. However, the `open`(9E) routine can be `nulldev`(9F). The `close`(9E), `read`(9E), and `write`(9E) user context routines are optional.

- The `open`(9E) routine gains access to the device.

- The close(9E) routine relinquishes access to the device. The close(9E) routine must undo everything that the open(9E) routine did.
- The read(9E) routine reads data from the device node.
- The write(9E) routine writes data to the device node.

In this section, the following code is added:

```
/* Use context entry points */
static int
dummy_open(dev_t *devp, int flag, int otyp, cred_t *cred)
{
    cmn_err(CE_NOTE, "Inside dummy_open");
    return DDI_SUCCESS;
}

static int
dummy_close(dev_t dev, int flag, int otyp, cred_t *cred)
{
    cmn_err(CE_NOTE, "Inside dummy_close");
    return DDI_SUCCESS;
}

static int
dummy_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
    cmn_err(CE_NOTE, "Inside dummy_read");
    return DDI_SUCCESS;
}

static int
dummy_write(dev_t dev, struct uio *uiop, cred_t *credp)
{
    cmn_err(CE_NOTE, "Inside dummy_write");
    return DDI_SUCCESS;
}
```

Declaring the User Context Entry Points

The user context entry point routines need to be uniquely named for this driver. Use the same prefix for each of the user context entry points that you used for each of the autoconfiguration entry point routines. The following declarations are the entry point declarations you should have in your dummy.c file:

```
static int dummy_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int dummy_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
static int dummy_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
    void **resultp);
static int dummy_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp);
static int dummy_open(dev_t *devp, int flag, int otyp, cred_t *cred);
static int dummy_close(dev_t dev, int flag, int otyp, cred_t *cred);
static int dummy_read(dev_t dev, struct uio *uiop, cred_t *credp);
```

```
static int dummy_write(dev_t dev, struct uio *uiop, cred_t *credp);
```

Defining the Open Device Entry Point

The open(9E) routine returns type `int`. The `open`(9E) routine should return either `DDI_SUCCESS` or the appropriate error number.

The `open`(9E) routine takes four arguments. This `dummy` driver is so simple that this `dummy_open` routine does not use any of the `open`(9E) arguments. The examples in Chapter 3, "Reading and Writing Data in Kernel Memory" show the `open`(9E) routine in more detail.

The following code is the `dummy_open` routine that you should enter into your `dummy.c` file. You can copy the name portion of this function definition directly from the declaration you entered in "Declaring the User Context Entry Points" on page 45. Write a message to the system log and return success.

```
static int
dummy_open(dev_t *devp, int flag, int otyp, cred_t *cred)
{
    cmn_err(CE_NOTE, "Inside dummy_open");
    return DDI_SUCCESS;
}
```

Defining the Close Device Entry Point

The close(9E) routine returns type `int`. The `close`(9E) routine should return either `DDI_SUCCESS` or the appropriate error number.

The `close`(9E) routine takes four arguments. This `dummy` driver is so simple that this `dummy_close` routine does not use any of the `close`(9E) arguments. The examples in Chapter 3, "Reading and Writing Data in Kernel Memory" show the `close`(9E) routine in more detail.

The `close`(9E) routine must undo everything that the `open`(9E) routine did. The `close`(9E) routine must deallocate anything that was allocated, close anything that was opened, and destroy anything that was created in the `open`(9E) routine. In this `dummy` driver, the `open`(9E) routine is so simple that nothing needs to be reclaimed or undone in the `close`(9E) routine.

The following code is the `dummy_close` routine that you should enter into your `dummy.c` file. You can copy the name portion of this function definition directly from the declaration you entered in "Declaring the User Context Entry Points" on page 45. Write a message to the system log and return success.

```
static int
dummy_close(dev_t dev, int flag, int otyp, cred_t *cred)
{
    cmn_err(CE_NOTE, "Inside dummy_close");
    return DDI_SUCCESS;
}
```

Defining the Read Device Entry Point

The `read`(9E) routine returns type `int`. The `read`(9E) routine should return either `DDI_SUCCESS` or the appropriate error number.

The `read`(9E) routine takes three arguments. This `dummy` driver is so simple that this `dummy_read` routine does not use any of the `read`(9E) arguments. The examples in Chapter 3, "Reading and Writing Data in Kernel Memory" show the `read`(9E) routine in more detail.

The following code is the `dummy_read` routine that you should enter into your `dummy.c` file. You can copy the name portion of this function definition directly from the declaration you entered in "Declaring the User Context Entry Points" on page 45. Write a message to the system log and return success.

```
static int
dummy_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
    cmn_err(CE_NOTE, "Inside dummy_read");
    return DDI_SUCCESS;
}
```

Defining the Write Device Entry Point

The `write`(9E) routine returns type `int`. The `write`(9E) routine should return either `DDI_SUCCESS` or the appropriate error number.

The `write`(9E) routine takes three arguments. This `dummy` driver is so simple that this `dummy_write` routine does not use any of the `write`(9E) arguments. The examples in Chapter 3, "Reading and Writing Data in Kernel Memory" show the `write`(9E) routine in more detail.

The following code is the `dummy_write` routine that you should enter into your `dummy.c` file. You can copy the name portion of this function definition directly from the declaration you entered in "Declaring the User Context Entry Points" on page 45. Write a message to the system log and return success.

```
static int
dummy_write(dev_t dev, struct uio *uiop, cred_t *credp)
```

```
{
    cmn_err(CE_NOTE, "Inside dummy_write");
    return DDI_SUCCESS;
}
```

Including User Context Header Files

The four user context entry point routines require your module to include several header files. You already have included the `types.h` header file, the `ddi.h` header file, and the `sunddi.h` header file. You need to include the `file.h`, `errno.h`, `open.h`, `cred.h`, and `uio.h` header files.

The following code is the list of header files that you now should have included in your `dummy.c` file for all the entry points you have written in this section and the previous two sections:

```
#include <sys/modctl.h>  /* used by modlinkage, modldrv, _init, _info, */
                         /* and _fini */
#include <sys/types.h>   /* used by open, close, read, write, prop_op, */
                         /* and ddi_prop_op */
#include <sys/file.h>    /* used by open, close */
#include <sys/errno.h>   /* used by open, close, read, write */
#include <sys/open.h>    /* used by open, close, read, write */
#include <sys/cred.h>    /* used by open, close, read */
#include <sys/uio.h>     /* used by read */
#include <sys/stat.h>    /* defines S_IFCHR used by ddi_create_minor_node */
#include <sys/cmn_err.h> /* used by all entry points for this driver */
#include <sys/ddi.h>     /* used by all entry points for this driver */
                         /* also used by ddi_get_instance and */
                         /* ddi_prop_op */
#include <sys/sunddi.h>  /* used by all entry points for this driver */
                         /* also used by ddi_create_minor_node, */
                         /* ddi_get_instance, and ddi_prop_op */
```

Writing the Driver Data Structures

All of the data structures described in this section are required for every device driver. All drivers must define a `dev_ops`(9S) device operations structure. Because the `dev_ops`(9S) structure includes a pointer to the `cb_ops`(9S) character and block operations structure, you must define the `cb_ops`(9S) structure first. The `modldrv`(9S) linkage structure for loadable drivers includes a pointer to the `dev_ops`(9S) structure. The `modlinkage`(9S) module linkage structure includes a pointer to the `modldrv`(9S) structure.

Except for the loadable module configuration entry points, all of the required entry points for a driver are initialized in the character and block operations structure or in the device operations structure. Some optional entry points and other related data also are initialized in these data structures. Initializing the entry points in these data structures enables the driver to be dynamically loaded.

The loadable module configuration entry points are not initialized in driver data structures. The `_init`(9E), `_info`(9E), and `_fini`(9E) entry points are required for all kernel modules and are not specific to device driver modules.

In this section, the following code is added:

```
/* cb_ops structure */
static struct cb_ops dummy_cb_ops = {
    dummy_open,
    dummy_close,
    nodev,              /* no strategy - nodev returns ENXIO */
    nodev,              /* no print */
    nodev,              /* no dump */
    dummy_read,
    dummy_write,
    nodev,              /* no ioctl */
    nodev,              /* no devmap */
    nodev,              /* no mmap */
    nodev,              /* no segmap */
    nochpoll,           /* returns ENXIO for non-pollable devices */
    dummy_prop_op,
    NULL,               /* streamtab struct; if not NULL, all above */
                        /* fields are ignored */
    D_NEW | D_MP,       /* compatibility flags: see conf.h */
    CB_REV,             /* cb_ops revision number */
    nodev,              /* no aread */
    nodev               /* no awrite */
};

/* dev_ops structure */
static struct dev_ops dummy_dev_ops = {
    DEVO_REV,
    0,                          /* reference count */
    dummy_getinfo,              /* no getinfo(9E) */
    nulldev,                    /* no identify(9E) - nulldev returns 0 */
    nulldev,                    /* no probe(9E) */
    dummy_attach,
    dummy_detach,
    nodev,                      /* no reset - nodev returns ENXIO */
    &dummy_cb_ops,
    (struct bus_ops *)NULL,
    nodev,                      /* no power(9E) */
    ddi_quiesce_not_needed,     /* no quiesce(9E) */
};

/* modldrv structure */
static struct modldrv md = {
    &mod_driverops,     /* Type of module. This is a driver. */
    "dummy driver",     /* Name of the module. */
    &dummy_dev_ops
};

/* modlinkage structure */
static struct modlinkage ml = {
    MODREV_1,
    &md,
    NULL
```

```
};

/* dev_info structure */
dev_info_t *dummy_dip;  /* keep track of one instance */
```

Defining the Character and Block Operations Structure

The cb_ops(9S) structure initializes standard character and block interfaces. See the cb_ops(9S) man page to learn what each element is and what the value of each element should be. This dummy driver does not use all of the elements in the cb_ops(9S) structure. See the description that follows the code sample.

When you name this structure, use the same dummy_ prefix that you used for the names of the autoconfiguration routines and the names of the user context routines. Prepend the static type modifier to the declaration.

The following code is the cb_ops(9S) structure that you should enter into your dummy.c file:

```
static struct cb_ops dummy_cb_ops = {
    dummy_open,
    dummy_close,
    nodev,              /* no strategy - nodev returns ENXIO */
    nodev,              /* no print */
    nodev,              /* no dump */
    dummy_read,
    dummy_write,
    nodev,              /* no ioctl */
    nodev,              /* no devmap */
    nodev,              /* no mmap */
    nodev,              /* no segmap */
    nochpoll,           /* returns ENXIO for non-pollable devices */
    dummy_prop_op,
    NULL,               /* streamtab struct; if not NULL, all above */
                        /* fields are ignored */
    D_NEW | D_MP,       /* compatibility flags: see conf.h */
    CB_REV,             /* cb_ops revision number */
    nodev,              /* no aread */
    nodev               /* no awrite */
};
```

Enter the names of the open(9E) and close(9E) entry points for this driver as the values of the first two elements of this structure. Enter the names of the read(9E) and write(9E) entry points for this driver as the values of the sixth and seventh elements of this structure. Enter the name of the prop_op(9E) entry point for this driver as the value of the thirteenth element in this structure.

The strategy(9E), print(9E), and dump(9E) routines are for block drivers only. This dummy driver does not define these three routines because this driver is a character driver. This driver does not define an ioctl(9E) entry point because this driver does not use I/O control

commands. This driver does not define devmap(9E), mmap(9E), or segmap(9E) entry points because this driver does not support memory mapping. This driver does not does not define aread(9E) or awrite(9E) entry points because this driver does not perform any asynchronous reads or writes. Initialize all of these unused function elements to nodev(9F). The nodev(9F) function returns the ENXIO error code.

Specify the nochpoll(9F) function for the chpoll(9E) element of the cb_ops(9S) structure because this driver is not for a pollable device. Specify NULL for the streamtab(9S) STREAMS entity declaration structure because this driver is not a STREAMS driver.

The compatibility flags are defined in the conf.h header file. The D_NEW flag means this driver is a new-style driver. The D_MP flag means this driver safely allows multiple threads of execution. All drivers must be multithreaded-safe, and must specify this D_MP flag. The D_64BIT flag means this driver supports 64-bit offsets and block numbers. See the conf.h header file for more compatibility flags.

The CB_REV element of the cb_ops(9S) structure is the cb_ops(9S) revision number. CB_REV is defined in the devops.h header file.

Defining the Device Operations Structure

The dev_ops(9S) structure initializes interfaces that are used for operations such as attaching and detaching the driver. See the dev_ops(9S) man page to learn what each element is and what the value of each element should be. This dummy driver does not use all of the elements in the dev_ops(9S) structure. See the description that follows the code sample.

When you name this structure, use the same dummy_ prefix that you used for the names of the autoconfiguration routines and the names of the user context routines. Prepend the static type modifier to the declaration.

The following code is the dev_ops(9S) structure that you should enter into your dummy.c file:

```
static struct dev_ops dummy_dev_ops = {
    DEVO_REV,
    0,                          /* reference count */
    dummy_getinfo,              /* no getinfo(9E) */
    nulldev,                    /* no identify(9E) - nulldev returns 0 */
    nulldev,                    /* no probe(9E) */
    dummy_attach,
    dummy_detach,
    nodev,                      /* no reset - nodev returns ENXIO */
    &dummy_cb_ops,
    (struct bus_ops *)NULL,
    nodev,                      /* no power(9E) */
    ddi_quiesce_not_needed,     /* no quiesce(9E) */
};
```

The DEVO_REV element of the dev_ops(9S) structure is the driver build version. DEVO_REV is defined in the devops.h header file. The second element in this structure is the driver reference count. Initialize this value to zero. The driver reference count is the number of instances of this driver that are currently open. The driver cannot be unloaded if any instances of the driver are still open.

The next six elements of the dev_ops(9S) structure are the names of the getinfo(9E), identify(9E), probe(9E), attach(9E), detach(9E), and reset functions for this particular driver. The identify(9E) function is obsolete. Initialize this structure element to nulldev(9F). The probe(9E) function determines whether the corresponding device exists and is valid. This dummy driver does not define a probe(9E) function. Initialize this structure element to nulldev. The nulldev(9F) function returns success. The reset function is obsolete. Initialize the reset function to nodev(9F).

The next element of the dev_ops(9S) structure is a pointer to the cb_ops(9S) structure for this driver. You initialized the cb_ops(9S) structure for this driver in "Defining the Character and Block Operations Structure" on page 50. Enter &dummy_cb_ops for the value of the pointer to the cb_ops(9S) structure.

The next element of the dev_ops(9S) structure is a pointer to the bus operations structure. Only nexus drivers have bus operations structures. This dummy driver is not a nexus driver. Set this value to NULL because this driver is a leaf driver.

The next element of the dev_ops(9S) structure is the name of the power(9E) routine for this driver. The power(9E) routine operates on a hardware device. This driver does not drive a hardware device. Set the value of this structure element to nodev.

The last element of the dev_ops(9S) structure is the name of the quiesce(9E) routine for this driver. The quiesce(9E) routine operates on a hardware device. This driver does not drive a hardware device. Set the value of this structure element to ddi_quiesce_not_needed(9F).

Defining the Module Linkage Structures

Two other module loading structures are required for every driver. The modlinkage(9S) module linkage structure is used by the _init(9E), _info(9E), and _fini(9E) routines to install, remove, and retrieve information from a module. The modldrv(9S) linkage structure for loadable drivers exports driver-specific information to the kernel. See the man pages for each structure to learn what each element is and what the value of each element should be.

The following code defines the modldrv(9S) and modlinkage(9S) structures for the driver shown in this chapter:

```
static struct modldrv md = {
    &mod_driverops,     /* Type of module. This is a driver. */
    "dummy driver",     /* Name of the module. */
    &dummy_dev_ops
};

static struct modlinkage ml = {
    MODREV_1,
    &md,
    NULL
};
```

The first element in the `modldrv`(9S) structure is a pointer to a structure that tells the kernel what kind of module this is. Set this value to the address of the `mod_driverops` structure. The `mod_driverops` structure tells the kernel that the `dummy.c` module is a loadable driver module. The `mod_driverops` structure is declared in the `modctl.h` header file. You already included the `modctl.h` header file in your `dummy.c` file, so do not declare the `mod_driverops` structure in `dummy.c`. The `mod_driverops` structure is defined in the `modctl.c` source file.

The second element in the `modldrv`(9S) structure is a string that describes this module. Usually this string contains the name of this module and the version number of this module. The last element of the `modldrv`(9S) structure is a pointer to the `dev_ops`(9S) structure for this driver. You initialized the `dev_ops`(9S) structure for this driver in “Defining the Device Operations Structure” on page 51.

The first element in the `modlinkage`(9S) structure is the revision number of the loadable modules system. Set this value to `MODREV_1`. The next element of the `modlinkage`(9S) structure is the address of a null-terminated array of pointers to linkage structures. Driver modules have only one linkage structure. Enter the address of the `md` structure for the value of this element of the `modlinkage`(9S) structure. Enter the value `NULL` to terminate this list of linkage structures.

Including Data Structures Header Files

The `cb_ops`(9S) and `dev_ops`(9S) structures require you to include the `conf.h` and `devops.h` header files. The `modlinkage`(9S) and `modldrv`(9S) structures require you to include the `modctl.h` header file. You already included the `modctl.h` header file for the loadable module configuration entry points.

The following code is the complete list of header files that you now should have included in your `dummy.c` file:

```
#include <sys/devops.h>  /* used by dev_ops */
#include <sys/conf.h>    /* used by dev_ops and cb_ops */
#include <sys/modctl.h>  /* used by modlinkage, modldrv, _init, _info, */
                         /* and _fini */
#include <sys/types.h>   /* used by open, close, read, write, prop_op, */
```

```
                        /* and ddi_prop_op */
#include <sys/file.h>    /* used by open, close */
#include <sys/errno.h>   /* used by open, close, read, write */
#include <sys/open.h>    /* used by open, close, read, write */
#include <sys/cred.h>    /* used by open, close, read */
#include <sys/uio.h>     /* used by read */
#include <sys/stat.h>    /* defines S_IFCHR used by ddi_create_minor_node */
#include <sys/cmn_err.h> /* used by all entry points for this driver */
#include <sys/ddi.h>     /* used by all entry points for this driver */
                         /* also used by cb_ops, ddi_get_instance, and */
                         /* ddi_prop_op */
#include <sys/sunddi.h>  /* used by all entry points for this driver */
                         /* also used by cb_ops, ddi_create_minor_node, */
                         /* ddi_get_instance, and ddi_prop_op */
```

Writing the Device Configuration File

This driver requires a configuration file. The minimum information that a configuration file must contain is the name of the device node and the name or type of the device's parent. In this simple example, the node name of the device is the same as the file name of the driver. Create a file named `dummy.conf` in your working directory. Put the following single line of information into `dummy.conf`:

```
name="dummy" parent="pseudo";
```

Building and Installing the Template Driver

This section shows you how to build and install the driver for a 32-bit platform. See "Building a Driver" on page 25 and "Installing a Driver" on page 27 for build and install instructions for SPARC architectures and for 64-bit x86 architectures.

Compile and link the driver. Use the `-D_KERNEL` option to indicate that this code defines a kernel module. The following example shows compiling and linking for a 32-bit architecture using the Oracle Solaris Studio C compiler:

```
% cc -D_KERNEL -c dummy.c
% ld -r -o dummy dummy.o
```

Make sure you are user `root` when you install the driver.

Install drivers in the `/tmp` directory until you are finished modifying and testing the `_info`, `_init`, and `attach` routines. Copy the driver binary to the `/tmp` directory. Link to the driver from the kernel driver directory. See "Device Driver Testing Tips" on page 106 for more information.

```
# cp dummy /tmp
```

Link to the following directory for a 32-bit architecture:

```
# ln -s /tmp/dummy /usr/kernel/drv/dummy
```

Copy the configuration file to the kernel driver area of the system.

```
# cp dummy.conf /usr/kernel/drv
```

Testing the Template Driver

This `dummy` driver merely writes a message to a system log each time an entry point routine is entered. To test this driver, watch for these messages to confirm that each entry point routine is successfully entered.

The `cmn_err`(9F) function writes low priority messages such as the messages defined in this `dummy` driver to `/dev/log`. The `syslogd`(1M) daemon reads messages from `/dev/log` and writes low priority messages to `/var/adm/messages`.

In a separate window, enter the following command and monitor the output as you perform the tests described in the remainder of this section:

```
% tail -f /var/adm/messages
```

Adding the Template Driver

Make sure you are user `root` when you add the driver. Use the `add_drv`(1M) command to add the driver:

```
# add_drv dummy
```

You should see the following messages in the window where you are viewing `/var/adm/messages`:

```
date time machine dummy: [ID 513080 kern.notice] NOTICE: Inside _info
date time machine dummy: [ID 874762 kern.notice] NOTICE: Inside _init
date time machine dummy: [ID 678704 kern.notice] NOTICE: Inside dummy_attach
```

The `_info`(9E), `_init`(9E), and `attach`(9E) entry points are called in that order when you add a driver.

The `dummy` driver has been added to the `/devices` directory:

```
% ls -l /devices/pseudo | grep dummy
drwxr-xr-x   2 root     sys          512 date time dummy@0
crw-------   1 root     sys       92,  0 date time dummy@0:0
```

The dummy driver also is the most recent module listed by modinfo(1M):

```
% modinfo
 Id Loadaddr   Size Info Rev Module Name
180 ed192b70    544  92   1  dummy (dummy driver)
```

The module name, dummy driver, is the value you entered for the second member of the modldrv(9S) structure. The value 92 is the major number of this module.

```
% grep dummy /etc/name_to_major
dummy 92
```

The Loadaddr address of ed192b70 is the address of the first instruction in the dummy driver. This address might be useful, for example, in debugging.

```
% mdb -k
> dummy`_init $m
    BASE    LIMIT     SIZE NAME
ed192b70 ed192ff0      480 dummy
> $q
```

The dummy driver also is the most recent module listed by prtconf(1M) in the pseudo device section:

```
% prtconf -P
    pseudo, instance #0
        dummy, instance #0 (driver not attached)
```

A driver is automatically loaded when a device that the driver manages is accessed. A driver might be automatically unloaded when the driver is not in use.

If your driver is in the /devices directory but modinfo(1M) does not list your driver, you can use either of the following methods to load your driver:

- Use the modload(1M) command.
- Access the device. The driver is loaded automatically when a device that the driver manages is accessed. The following section describes how to access the dummy device.

Reading and Writing the Device

Make sure you are user root when you perform the tests described in this section. If you are not user root, you will receive “Permission denied” error messages when you try to access the /devices/pseudo/dummy@0:0 special file. Notice the permissions that are shown for /devices/pseudo/dummy@0:0 in “Adding the Template Driver” on page 55.

Test reading from the device. Your dummy device probably is named /devices/pseudo/dummy@0:0. The following command reads from your dummy device even if it has a slightly different name:

```
# cat /devices/pseudo/dummy*
```

You should see the following messages in the window where you are viewing `/var/adm/messages`:

```
date time machine dummy: [ID 136952 kern.notice] NOTICE: Inside dummy_open
date time machine dummy: [ID 623947 kern.notice] NOTICE: Inside dummy_getinfo
date time machine dummy: [ID 891851 kern.notice] NOTICE: Inside dummy_prop_op
date time machine dummy: [ID 623947 kern.notice] NOTICE: Inside dummy_getinfo
date time machine dummy: [ID 891851 kern.notice] NOTICE: Inside dummy_prop_op
date time machine dummy: [ID 623947 kern.notice] NOTICE: Inside dummy_getinfo
date time machine dummy: [ID 709590 kern.notice] NOTICE: Inside dummy_read
date time machine dummy: [ID 550206 kern.notice] NOTICE: Inside dummy_close
```

Test writing to the device:

```
# echo hello > `ls /devices/pseudo/dummy*`
```

You should see the following messages in the window where you are viewing `/var/adm/messages`:

```
date time machine dummy: [ID 136952 kern.notice] NOTICE: Inside dummy_open
date time machine dummy: [ID 623947 kern.notice] NOTICE: Inside dummy_getinfo
date time machine dummy: [ID 891851 kern.notice] NOTICE: Inside dummy_prop_op
date time machine dummy: [ID 623947 kern.notice] NOTICE: Inside dummy_getinfo
date time machine dummy: [ID 891851 kern.notice] NOTICE: Inside dummy_prop_op
date time machine dummy: [ID 623947 kern.notice] NOTICE: Inside dummy_getinfo
date time machine dummy: [ID 672780 kern.notice] NOTICE: Inside dummy_write
date time machine dummy: [ID 550206 kern.notice] NOTICE: Inside dummy_close
```

As you can see, this output from the write test is almost identical to the output you saw from the read test. The only difference is in the seventh line of the output. Using the `cat`(1) command causes the kernel to access the `read`(9E) entry point of the driver. Using the `echo`(1) command causes the kernel to access the `write`(9E) entry point of the driver. The text argument that you give to `echo`(1) is ignored because this driver does not do anything with that data.

Removing the Template Driver

Make sure you are user `root` when you unload the driver. Use the `rem_drv`(1M) command to unload the driver and remove the device from the `/devices` directory:

```
# rem_drv dummy
```

You should see the following messages in the window where you are viewing `/var/adm/messages`:

```
date time machine dummy: [ID 513080 kern.notice] NOTICE: Inside _info
date time machine dummy: [ID 617648 kern.notice] NOTICE: Inside dummy_detach
date time machine dummy: [ID 812373 kern.notice] NOTICE: Inside _fini
```

The dummy device is no longer in the /devices directory:

```
# ls /devices/pseudo/dummy*
/devices/pseudo/dummy*: No such file or directory
```

The next time you want to read from or write to the dummy device, you must load the driver again using add_drv(1M).

You can use the modunload(1M) command to unload the driver but not remove the device from /devices. Then the next time you read from or write to the dummy device, the driver is automatically loaded.

Press Control-C to stop tailing the /var/adm/messages messages.

Complete Template Driver Source

The following code is the complete source for the dummy driver described in this chapter:

```
/*
 * Minimalist pseudo-device.
 * Writes a message whenever a routine is entered.
 *
 * Build the driver:
 *      cc -D_KERNEL -c dummy.c
 *      ld -r -o dummy dummy.o
 * Copy the driver and the configuration file to /usr/kernel/drv:
 *      cp dummy.conf /usr/kernel/drv
 *      cp dummy /tmp
 *      ln -s /tmp/dummy /usr/kernel/drv/dummy
 * Add the driver:
 *      add_drv dummy
 * Test (1) read from driver (2) write to driver:
 *      cat /devices/pseudo/dummy@*
 *      echo hello > `ls /devices/pseudo/dummy@*`
 * Verify the tests in another window:
 *      tail -f /var/adm/messages
 * Remove the driver:
 *      rem_drv dummy
 */

#include <sys/devops.h>  /* used by dev_ops */
#include <sys/conf.h>    /* used by dev_ops and cb_ops */
#include <sys/modctl.h>  /* used by modlinkage, modldrv, _init, _info, */
                         /* and _fini */
#include <sys/types.h>   /* used by open, close, read, write, prop_op, */
                         /* and ddi_prop_op */
#include <sys/file.h>    /* used by open, close */
#include <sys/errno.h>   /* used by open, close, read, write */
#include <sys/open.h>    /* used by open, close, read, write */
#include <sys/cred.h>    /* used by open, close, read */
#include <sys/uio.h>     /* used by read */
#include <sys/stat.h>    /* defines S_IFCHR used by ddi_create_minor_node */
```

```
#include <sys/cmn_err.h> /* used by all entry points for this driver */
#include <sys/ddi.h>     /* used by all entry points for this driver */
                         /* also used by cb_ops, ddi_get_instance, and */
                         /* ddi_prop_op */
#include <sys/sunddi.h>  /* used by all entry points for this driver */
                         /* also used by cb_ops, ddi_create_minor_node, */
                         /* ddi_get_instance, and ddi_prop_op */

static int dummy_attach(dev_info_t *dip, ddi_attach_cmd_t cmd);
static int dummy_detach(dev_info_t *dip, ddi_detach_cmd_t cmd);
static int dummy_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
    void **resultp);
static int dummy_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp);
static int dummy_open(dev_t *devp, int flag, int otyp, cred_t *cred);
static int dummy_close(dev_t dev, int flag, int otyp, cred_t *cred);
static int dummy_read(dev_t dev, struct uio *uiop, cred_t *credp);
static int dummy_write(dev_t dev, struct uio *uiop, cred_t *credp);

/* cb_ops structure */
static struct cb_ops dummy_cb_ops = {
    dummy_open,
    dummy_close,
    nodev,              /* no strategy - nodev returns ENXIO */
    nodev,              /* no print */
    nodev,              /* no dump */
    dummy_read,
    dummy_write,
    nodev,              /* no ioctl */
    nodev,              /* no devmap */
    nodev,              /* no mmap */
    nodev,              /* no segmap */
    nochpoll,           /* returns ENXIO for non-pollable devices */
    dummy_prop_op,
    NULL,               /* streamtab struct; if not NULL, all above */
                        /* fields are ignored */
    D_NEW | D_MP,       /* compatibility flags: see conf.h */
    CB_REV,             /* cb_ops revision number */
    nodev,              /* no aread */
    nodev               /* no awrite */
};

/* dev_ops structure */
static struct dev_ops dummy_dev_ops = {
    DEVO_REV,
    0,                          /* reference count */
    dummy_getinfo,              /* no getinfo(9E) */
    nulldev,                    /* no identify(9E) - nulldev returns 0 */
    nulldev,                    /* no probe(9E) */
    dummy_attach,
    dummy_detach,
    nodev,                      /* no reset - nodev returns ENXIO */
    &dummy_cb_ops,
    (struct bus_ops *)NULL,
    nodev,                      /* no power(9E) */
    ddi_quiesce_not_needed,     /* no quiesce(9E) */
};
```

```
/* modldrv structure */
static struct modldrv md = {
    &mod_driverops,     /* Type of module. This is a driver. */
    "dummy driver",     /* Name of the module. */
    &dummy_dev_ops
};

/* modlinkage structure */
static struct modlinkage ml = {
    MODREV_1,
    &md,
    NULL
};

/* dev_info structure */
dev_info_t *dummy_dip;  /* keep track of one instance */


/* Loadable module configuration entry points */
int
_init(void)
{
    cmn_err(CE_NOTE, "Inside _init");
    return(mod_install(&ml));
}

int
_info(struct modinfo *modinfop)
{
    cmn_err(CE_NOTE, "Inside _info");
    return(mod_info(&ml, modinfop));
}

int
_fini(void)
{
    cmn_err(CE_NOTE, "Inside _fini");
    return(mod_remove(&ml));
}

/* Device configuration entry points */
static int
dummy_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
    cmn_err(CE_NOTE, "Inside dummy_attach");
    switch(cmd) {
    case DDI_ATTACH:
        dummy_dip = dip;
        if (ddi_create_minor_node(dip, "0", S_IFCHR,
            ddi_get_instance(dip), DDI_PSEUDO,0)
            != DDI_SUCCESS) {
            cmn_err(CE_NOTE,
                "%s%d: attach: could not add character node.",
                "dummy", 0);
            return(DDI_FAILURE);
        } else
            return DDI_SUCCESS;
    default:
```

```
        return DDI_FAILURE;
    }
}

static int
dummy_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
    cmn_err(CE_NOTE, "Inside dummy_detach");
    switch(cmd) {
    case DDI_DETACH:
        dummy_dip = 0;
        ddi_remove_minor_node(dip, NULL);
        return DDI_SUCCESS;
    default:
        return DDI_FAILURE;
    }
}

static int
dummy_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg,
    void **resultp)
{
    cmn_err(CE_NOTE, "Inside dummy_getinfo");
    switch(cmd) {
    case DDI_INFO_DEVT2DEVINFO:
        *resultp = dummy_dip;
        return DDI_SUCCESS;
    case DDI_INFO_DEVT2INSTANCE:
        *resultp = 0;
        return DDI_SUCCESS;
    default:
        return DDI_FAILURE;
    }
}

/* Main entry points */
static int
dummy_prop_op(dev_t dev, dev_info_t *dip, ddi_prop_op_t prop_op,
    int flags, char *name, caddr_t valuep, int *lengthp)
{
    cmn_err(CE_NOTE, "Inside dummy_prop_op");
    return(ddi_prop_op(dev,dip,prop_op,flags,name,valuep,lengthp));
}

static int
dummy_open(dev_t *devp, int flag, int otyp, cred_t *cred)
{
    cmn_err(CE_NOTE, "Inside dummy_open");
    return DDI_SUCCESS;
}

static int
dummy_close(dev_t dev, int flag, int otyp, cred_t *cred)
{
    cmn_err(CE_NOTE, "Inside dummy_close");
    return DDI_SUCCESS;
}
```

```
static int
dummy_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
    cmn_err(CE_NOTE, "Inside dummy_read");
    return DDI_SUCCESS;
}

static int
dummy_write(dev_t dev, struct uio *uiop, cred_t *credp)
{
    cmn_err(CE_NOTE, "Inside dummy_write");
    return DDI_SUCCESS;
}
```

♦ ♦ ♦ CHAPTER 3

Reading and Writing Data in Kernel Memory

In this chapter, you will extend the very simple prototype driver you developed in the previous chapter. The driver you will develop in this chapter displays data read from kernel memory. The first version of this driver writes data to a system log every time the driver is loaded. The second version of this driver displays data at user request. In the third version of this driver, the user can write new data to the device.

Displaying Data Stored in Kernel Memory

The pseudo device driver presented in this section writes a constant string to a system log when the driver is loaded.

This first version of the Quote Of The Day driver (`qotd_1`) is even more simple than the `dummy` driver from the previous chapter. The `dummy` driver includes all functions that are required to drive hardware. This `qotd_1` driver includes only the bare minimum functions it needs to make a string available to a user command. For example, this `qotd_1` driver has no `cb_ops`(9S) structure. Therefore, this driver defines no `open`(9E), `close`(9E), `read`(9E), or `write`(9E) function. If you examine the `dev_ops`(9S) structure for this `qotd_1` driver, you see that no `getinfo`(9E), `attach`(9E), or `detach`(9E) function is defined. This driver contains no function declarations because all the functions that are defined in this driver are declared in the `modctl.h` header file. You must include the `modctl.h` header file in your `qotd_1.c` file.

This `qotd_1` driver defines a global variable to hold its text data. The `_init`(9E) entry point for this driver uses the `cmn_err`(9F) function to write the string to a system log. The `dummy` driver also uses the `cmn_err`(9F) function to display messages. The `qotd_1` driver is different from the `dummy` driver because the `qotd_1` driver stores its string in kernel memory.

Writing Quote Of The Day Version 1

Enter the source code shown in the following example into a text file named `qotd_1.c`.

EXAMPLE 3-1 Quote Of The Day Version 1 Source File

```
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/devops.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

#define QOTD_MAXLEN     128

static const char qotd[QOTD_MAXLEN]
        = "Be careful about reading health books. \
You may die of a misprint. - Mark Twain\n";

static struct dev_ops qotd_dev_ops = {
        DEVO_REV,               /* devo_rev */
        0,                      /* devo_refcnt */
        ddi_no_info,            /* devo_getinfo */
        nulldev,                /* devo_identify */
        nulldev,                /* devo_probe */
        nulldev,                /* devo_attach */
        nulldev,                /* devo_detach */
        nodev,                  /* devo_reset */
        (struct cb_ops *)NULL,  /* devo_cb_ops */
        (struct bus_ops *)NULL, /* devo_bus_ops */
        nulldev,                /* devo_power */
        ddi_quiesce_not_needed, /* devo_quiesce */
};

static struct modldrv modldrv = {
        &mod_driverops,
        "Quote of the Day 1.0",
        &qotd_dev_ops};

static struct modlinkage modlinkage = {
        MODREV_1,
        (void *)&modldrv,
        NULL
};

int
_init(void)
{
        cmn_err(CE_CONT, "QOTD: %s\n", qotd);
        return (mod_install(&modlinkage));
}

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

```
}
```

Enter the configuration information shown in the following example into a text file named `qotd_1.conf`.

EXAMPLE 3-2 Quote Of The Day Version 1 Configuration File

```
name="qotd_1" parent="pseudo" instance=0;
```

Building, Installing, and Using Quote Of The Day Version 1

Compile and link the driver. Use the `-D_KERNEL` option to indicate that this code defines a kernel module. The following example shows compiling and linking for a 32-bit architecture using the Oracle Solaris Studio C compiler:

```
% cc -D_KERNEL -c qotd_1.c
% ld -r -o qotd_1 qotd_1.o
```

Note that the name of the driver, `qotd_1`, must match the `name` property in the configuration file.

Make sure you are user `root` when you install the driver.

Copy the driver binary to the `/tmp` directory as discussed in "Device Driver Testing Tips" on page 106.

```
# cp qotd_1 /tmp
# ln -s /tmp/qotd_1 /usr/kernel/drv/qotd_1
```

Copy the configuration file to the kernel driver area of the system.

```
# cp qotd_1.conf /usr/kernel/drv
```

This `qotd_1` driver writes a message to a system log each time the driver is loaded. The `cmn_err`(9F) function writes low priority messages such as the message defined in this `qotd_1` driver to `/dev/log`. The `syslogd`(1M) daemon reads messages from `/dev/log` and writes low priority messages to `/var/adm/messages`.

To test this driver, watch for the message in `/var/adm/messages`. In a separate window, enter the following command:

```
% tail -f /var/adm/messages
```

Make sure you are user `root` when you load the driver. Use the `add_drv`(1M) command to load the driver:

```
# add_drv qotd_1
```

You should see the following messages in the window where you are viewing `/var/adm/messages`:

```
date time machine pseudo: [ID 129642 kern.info] pseudo-device: devinfo0
date time machine genunix: [ID 936769 kern.info] devinfo0 is /pseudo/devinfo@0
date time machine qotd: [ID 197678 kern.notice] QOTD_1: Be careful about
reading health books. You may die of a misprint. - Mark Twain
```

This last line is the content of the variable output by the `cmn_err`(9F) function in the `_init`(9E) entry point. The `_init`(9E) entry point is called when the driver is loaded.

Displaying Data on Demand

The sample code in this section creates a pseudo device that is controlled by the driver. The driver stores data in the device and makes the data available when the user accesses the device for reading.

This section first discusses the important code differences between these two versions of the Quote Of The Day driver. This section then shows you how you can access the device to cause the quotation to display.

Writing Quote Of The Day Version 2

The driver that controls the pseudo device is more complex than the driver shown in the previous section. This section first explains some important features of this version of the driver. This section then shows all the source for this driver.

The following list summarizes the differences between the two versions of the Quote Of The Day driver:

- Version 2 of the driver defines a state structure that holds information about each instance of the device.
- Version 2 defines a `cb_ops`(9S) structure and a more complete `dev_ops`(9S) structure.
- Version 2 defines `open`(9E), `close`(9E), `read`(9E), `getinfo`(9E), `attach`(9E), and `detach`(9E) entry points.
- Version 1 uses the `cmn_err`(9F) function to write a constant string to a system log in the `_init`(9E) entry point of the driver. The `_init`(9E) entry point is called when the driver is loaded. Version 2 uses the `uiomove`(9F) function to copy the quotation from kernel memory. The copied data is returned by the `read`(9E) entry point. The `read`(9E) entry point is called when the driver is accessed for reading.
- Version 2 of the driver uses `ASSERT`(9F) statements to check the validity of data.

The following sections provide more detail about the additions and changes in Version 2 of the Quote Of The Day driver.

Managing Device State

The `_init`(9E) and `_fini`(9E) entry points and all six new entry points defined in this driver maintain a soft state for the device. Most device drivers maintain state information with each instance of the device they control. An instance usually is a sub-device. For example, a disk driver might communicate with a hardware controller device that has several disk drives attached. See "Retrieving Driver Soft State Information" in "Writing Device Drivers for Oracle Solaris 11.2 " for more information about soft states.

This sample driver allows only one instance. The instance number is assigned in the configuration file. See Example 3-4. Most device drivers allow any number of instances of a device to be created. The system manages the device instance numbers, and the DDI soft state functions manage the instances.

The following flow gives an overview of how DDI soft state functions manage a state pointer and the state of a device instance:

1. The `ddi_soft_state_init`(9F) function initializes the state pointer. The state pointer is an opaque handle that enables allocation, deallocation, and tracking of a state structure for each instance of a device. The state structure is a user-defined type that maintains data specific to this instance of the device. In this example, the state pointer and state structure are declared after the entry point declarations. See `qotd_state_head` and `qotd_state` in Example 3-3.
2. The `ddi_soft_state_zalloc`(9F) function uses the state pointer and the device instance to create the state structure for this instance.
3. The `ddi_get_soft_state`(9F) function uses the state pointer and the device instance to retrieve the state structure for this instance of the device.
4. The `ddi_soft_state_free`(9F) function uses the state pointer and the device instance to free the state structure for this instance.
5. The `ddi_soft_state_fini`(9F) function uses the state pointer to destroy the state pointer and the state structures for all instances of this device.

The `ddi_soft_state_zalloc`(9F), `ddi_get_soft_state`(9F), and `ddi_soft_state_free`(9F) functions coordinate access to the underlying data structures in a way that is safe for multithreading. No additional locks should be necessary.

Initializing and Unloading

The _init(9E) entry point first calls the ddi_soft_state_init(9F) function to initialize the soft state. If the soft state initialization fails, that error code is returned. If the soft state initialization succeeds, the _init(9E) entry point calls the mod_install(9F) function to load a new module. If the module install fails, the _init(9E) entry point calls the ddi_soft_state_fini(9F) function and returns the error code from the failed module install.

Your code must undo everything that it does. You must call ddi_soft_state_fini(9F) if the module install fails because the _init(9E) call succeeded and created a state pointer.

The _fini(9E) entry point must undo everything the _init(9E) entry point did. The _fini(9E) entry point first calls the mod_remove(9F) function to remove the module that the _init(9E) entry point installed. If the module remove fails, that error code is returned. If the module remove succeeds, the _fini(9E) entry point calls the ddi_soft_state_fini(9F) function to destroy the state pointer and the state structures for all instances of this device.

Attaching and Detaching

The attach(9E) entry point first calls the ddi_get_instance(9F) function to retrieve the instance number of the device information node. The attach(9E) entry point uses this instance number to call the ddi_soft_state_zalloc(9F), ddi_get_soft_state(9F), and ddi_create_minor_node(9F) functions.

The attach(9E) entry point calls the ddi_soft_state_zalloc(9F) function to create a state structure for this device instance. If creation of the soft state structure fails, attach(9E) writes an error message to a system log and returns failure. This device instance is not attached. If creation of the soft state structure succeeds, attach(9E) calls the ddi_get_soft_state(9F) function to retrieve the state structure for this device instance.

If retrieval of the state structure fails, attach(9E) writes an error message to a system log, calls the ddi_soft_state_free(9F) function to destroy the state structure that was created by ddi_soft_state_zalloc(9F), and returns failure. This device instance is not attached. If retrieval of the state structure succeeds, attach(9E) calls the ddi_create_minor_node(9F) function to create the device node.

At the top of this driver source file, a constant named QOTD_NAME is defined that holds the string name of the device. This constant is one of the arguments that is passed to

ddi_create_minor_node(9F). If creation of the device node fails, attach(9E) writes an error message to a system log, calls the ddi_soft_state_free(9F) function to destroy the state structure that was created by ddi_soft_state_zalloc(9F), calls the ddi_remove_minor_node(9F) function, and returns failure. This device instance is not attached.

If creation of the device node succeeds, this device instance is attached. The attach(9E) entry point assigns the instance number that was retrieved with ddi_get_instance(9F) to the instance member of the state structure for this instance. Then attach(9E) assigns the dev_info structure pointer that was passed in the attach(9E) call to the dev_info structure pointer member of the state structure for this instance. The ddi_report_dev(9F) function writes a message in the system log file when the device is added or when the system is booted. The message announces this device as shown in the following example:

```
% dmesg
date time machine pseudo: [ID 129642 kern.info] pseudo-device: qotd_20
date time machine genunix: [ID 936769 kern.info] qotd_20 is /pseudo/qotd_2@0
```

The detach(9E) entry point first calls the ddi_get_instance(9F) function to retrieve the instance number of the device information node. The detach(9E) entry point uses this instance number to call the ddi_soft_state_free(9F) function to destroy the state structure that was created by ddi_soft_state_zalloc(9F) in the attach(9E) entry point. The detach(9E) entry point then calls the ddi_remove_minor_node(9F) function to remove the device that was created by ddi_create_minor_node(9F) in the attach(9E) entry point.

Opening the Device, Closing the Device, and Getting Module Information

The open(9E) and close(9E) entry points are identical in this sample driver. In each case, the entry point first calls the getminor(9F) function to retrieve the minor number of the device. Then each entry point uses this instance number to call the ddi_get_soft_state(9F) function to retrieve the state structure for this device instance. If no state structure is retrieved, an error code is returned. If a state structure is retrieved, the open(9E) and close(9E) entry points both verify the type of this device. If this device is not a character device, the EINVAL (invalid) error code is returned.

If the user wants device information for this device instance, the getinfo(9E) entry point returns the device information from the state structure. If the user wants the instance number of this device instance, the getinfo(9E) entry point uses the getminor(9F) function to return the minor number.

Reading the Data

The read(9E) entry point first calls the getminor(9F) function to retrieve the minor number of the device. The read(9E) entry point uses this instance number to call the ddi_get_soft_state(9F) function to retrieve the state structure for this device instance. If no state structure is retrieved, read(9E) returns an error code. If a state structure is retrieved, read(9E) calls the uiomove(9F) function to copy the quotation from the driver to the uio(9S) I/O request structure.

Checking Data Validity

Version 2 of the driver uses ASSERT(9F) statements to check the validity of data. If the asserted expression is true, the ASSERT(9F) statement does nothing. If the asserted expression is false, the ASSERT(9F) statement writes an error message to the console and causes the system to panic.

To use ASSERT(9F) statements, include the sys/debug.h header file in your source and define the DEBUG preprocessor symbol. If you do not define the DEBUG preprocessor symbol, then the ASSERT(9F) statements do nothing. Simply recompile to activate or inactivate ASSERT(9F) statements.

Quote Of The Day Version 2 Source

Enter the source code shown in the following example into a text file named qotd_2.c.

EXAMPLE 3-3 Quote Of The Day Version 2 Source File

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/uio.h>
#include <sys/stat.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/devops.h>
#include <sys/debug.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

#define QOTD_NAME       "qotd"
```

```
#define QOTD_MAXLEN     128

static const char qotd[QOTD_MAXLEN]
        = "You can't have everything. \
Where would you put it? - Steven Wright\n";

static void *qotd_state_head;

struct qotd_state {
        int             instance;
        dev_info_t      *devi;
};

static int qotd_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int qotd_attach(dev_info_t *, ddi_attach_cmd_t);
static int qotd_detach(dev_info_t *, ddi_detach_cmd_t);
static int qotd_open(dev_t *, int, int, cred_t *);
static int qotd_close(dev_t, int, int, cred_t *);
static int qotd_read(dev_t, struct uio *, cred_t *);

static struct cb_ops qotd_cb_ops = {
        qotd_open,              /* cb_open */
        qotd_close,             /* cb_close */
        nodev,                  /* cb_strategy */
        nodev,                  /* cb_print */
        nodev,                  /* cb_dump */
        qotd_read,              /* cb_read */
        nodev,                  /* cb_write */
        nodev,                  /* cb_ioctl */
        nodev,                  /* cb_devmap */
        nodev,                  /* cb_mmap */
        nodev,                  /* cb_segmap */
        nochpoll,               /* cb_chpoll */
        ddi_prop_op,            /* cb_prop_op */
        (struct streamtab *)NULL,       /* cb_str */
        D_MP | D_64BIT,         /* cb_flag */
        CB_REV,                 /* cb_rev */
        nodev,                  /* cb_aread */
        nodev                   /* cb_awrite */
};

static struct dev_ops qotd_dev_ops = {
        DEVO_REV,               /* devo_rev */
        0,                      /* devo_refcnt */
        qotd_getinfo,           /* devo_getinfo */
        nulldev,                /* devo_identify */
        nulldev,                /* devo_probe */
        qotd_attach,            /* devo_attach */
        qotd_detach,            /* devo_detach */
        nodev,                  /* devo_reset */
        &qotd_cb_ops,           /* devo_cb_ops */
        (struct bus_ops *)NULL, /* devo_bus_ops */
        nulldev,                /* devo_power */
        ddi_quiesce_not_needed, /* devo_quiesce */
};

static struct modldrv modldrv = {
        &mod_driverops,
```

```
        "Quote of the Day 2.0",
        &qotd_dev_ops};

static struct modlinkage modlinkage = {
        MODREV_1,
        (void *)&modldrv,
        NULL
};

int
_init(void)
{
        int retval;

        if ((retval = ddi_soft_state_init(&qotd_state_head,
            sizeof (struct qotd_state), 1)) != 0)
                return retval;
        if ((retval = mod_install(&modlinkage)) != 0) {
                ddi_soft_state_fini(&qotd_state_head);
                return (retval);
        }

        return (retval);
}

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

int
_fini(void)
{
        int retval;

        if ((retval = mod_remove(&modlinkage)) != 0)
                return (retval);
        ddi_soft_state_fini(&qotd_state_head);

        return (retval);
}

/*ARGSUSED*/
static int
qotd_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg, void **resultp)
{
        struct qotd_state *qsp;
        int retval = DDI_FAILURE;

        ASSERT(resultp != NULL);

        switch (cmd) {
        case DDI_INFO_DEVT2DEVINFO:
                if ((qsp = ddi_get_soft_state(qotd_state_head,
                    getminor((dev_t)arg))) != NULL) {
                        *resultp = qsp->devi;
                        retval = DDI_SUCCESS;
```

```
                } else
                        *resultp = NULL;
                break;
        case DDI_INFO_DEVT2INSTANCE:
                *resultp = (void *)getminor((dev_t)arg);
                retval = DDI_SUCCESS;
                break;
        }

        return (retval);
}

static int
qotd_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int instance = ddi_get_instance(dip);
        struct qotd_state *qsp;

        switch (cmd) {
        case DDI_ATTACH:
                if (ddi_soft_state_zalloc(qotd_state_head, instance)
                    != DDI_SUCCESS) {
                        cmn_err(CE_WARN, "Unable to allocate state for %d",
                            instance);
                        return (DDI_FAILURE);
                }
                if ((qsp = ddi_get_soft_state(qotd_state_head, instance))
                    == NULL) {
                        cmn_err(CE_WARN, "Unable to obtain state for %d",
                            instance);
                        ddi_soft_state_free(dip, instance);
                        return (DDI_FAILURE);
                }
                if (ddi_create_minor_node(dip, QOTD_NAME, S_IFCHR, instance,
                    DDI_PSEUDO, 0) != DDI_SUCCESS) {
                        cmn_err(CE_WARN, "Cannot create minor node for %d",
                            instance);
                        ddi_soft_state_free(dip, instance);
                        ddi_remove_minor_node(dip, NULL);
                        return (DDI_FAILURE);
                }
                qsp->instance = instance;
                qsp->devi = dip;

                ddi_report_dev(dip);
                return (DDI_SUCCESS);
        case DDI_RESUME:
                return (DDI_SUCCESS);
        default:
                return (DDI_FAILURE);
        }
}

static int
qotd_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        int instance = ddi_get_instance(dip);
```

```
        switch (cmd) {
        case DDI_DETACH:
                ddi_soft_state_free(qotd_state_head, instance);
                ddi_remove_minor_node(dip, NULL);
                return (DDI_SUCCESS);
        case DDI_SUSPEND:
                return (DDI_SUCCESS);
        default:
                return (DDI_FAILURE);
        }
}

/*ARGSUSED*/
static int
qotd_open(dev_t *devp, int flag, int otyp, cred_t *credp)
{
        int instance = getminor(*devp);
        struct qotd_state *qsp;

        if ((qsp = ddi_get_soft_state(qotd_state_head, instance)) == NULL)
                return (ENXIO);

        ASSERT(qsp->instance == instance);

        if (otyp != OTYP_CHR)
                return (EINVAL);

        return (0);
}

/*ARGSUSED*/
static int
qotd_close(dev_t dev, int flag, int otyp, cred_t *credp)
{
        struct qotd_state *qsp;
        int instance = getminor(dev);

        if ((qsp = ddi_get_soft_state(qotd_state_head, instance)) == NULL)
                return (ENXIO);

        ASSERT(qsp->instance == instance);

        if (otyp != OTYP_CHR)
                return (EINVAL);

        return (0);
}

/*ARGSUSED*/
static int
qotd_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
        struct qotd_state *qsp;
        int instance = getminor(dev);

        if ((qsp = ddi_get_soft_state(qotd_state_head, instance)) == NULL)
                return (ENXIO);
```

```
        ASSERT(qsp->instance == instance);

        return (uiomove((void *)qotd, min(uiop->uio_resid, strlen(qotd)),
            UIO_READ, uiop));
}
```

Enter the configuration information shown in the following example into a text file named `qotd_2.conf`.

EXAMPLE 3-4 Quote Of The Day Version 2 Configuration File

```
name="qotd_2" parent="pseudo" instance=0;
```

Building, Installing, and Using Quote Of The Day Version 2

Version 2 of the driver uses ASSERT(9F) statements to check the validity of data. To use ASSERT(9F) statements, include the `sys/debug.h` header file in your source and define the `DEBUG` preprocessor symbol.

Compile and link the driver. If you use ASSERT(9F) statements to check the validity of data, you must define the `DEBUG` preprocessor symbol:

```
% cc -D_KERNEL -DDEBUG -c qotd_2.c
% ld -r -o qotd_2 qotd_2.o
```

The following example shows compiling and linking for a 32-bit architecture if you are not using ASSERT(9F) statements:

```
% cc -D_KERNEL -c qotd_2.c
% ld -r -o qotd_2 qotd_2.o
```

Make sure you are user `root` when you install the driver.

Copy the driver binary to the `/tmp` directory as discussed in "Building and Installing the Template Driver" on page 54.

```
# cp qotd_2 /tmp
# ln -s /tmp/qotd_2 /usr/kernel/drv/qotd_2
```

Copy the configuration file to the kernel driver area of the system.

```
# cp qotd_2.conf /usr/kernel/drv
```

In a separate window, enter the following command:

```
% tail -f /var/adm/messages
```

Make sure you are user `root` when you load the driver. Use the `add_drv`(1M) command to load the driver:

```
# add_drv qotd_2
```

You should see the following messages in the window where you are viewing `/var/adm/messages`:

```
date time machine pseudo: [ID 129642 kern.info] pseudo-device: devinfo0
date time machine genunix: [ID 936769 kern.info] devinfo0 is /pseudo/devinfo@0
date time machine pseudo: [ID 129642 kern.info] pseudo-device: qotd_20
date time machine genunix: [ID 936769 kern.info] qotd_20 is /pseudo/qotd_2@0
```

When this version of the Quote Of The Day driver loads, it does not display its quotation. The `qotd_1` driver wrote a message to a system log through its `_init`(9E) entry point. This `qotd_2` driver stores its data and makes the data available through its `read`(9E) entry point.

You can use the `modinfo`(1M) command to display the module information for this version of the Quote Of The Day driver. The module name is the value you entered for the second member of the `modldrv` structure. The value `96` is the major number of this module.

```
% modinfo | grep qotd
182 ed115948    754  96   1  qotd_2 (Quote of the Day 2.0)
% grep qotd /etc/name_to_major
qotd_1 94
qotd_2 96
```

This driver also is the most recent module listed by `prtconf`(1M) in the pseudo device section:

```
% prtconf -P | grep qotd
        qotd_1, instance #0 (driver not attached)
        qotd_2, instance #0
```

When you access this `qotd_2` device for reading, the command you use to access the device retrieves the data from the device node. The command then displays the data in the same way that the command displays any other input. To get the name of the device special file, look in the `/devices` directory:

```
% ls -l /devices/pseudo/qotd*
crw-------   1 root   sys   96, 0 date time /devices/pseudo/qotd_2@0:qotd
```

This output shows that `qotd_2@0:qotd` is a character device. This listing also shows that only the `root` user has permission to read or write this device. Make sure you are user `root` when you test this driver. To test the `qotd_2` driver, you can use the `more`(1) command to access the device file for reading:

```
# more /devices/pseudo/qotd_2@0:qotd
```

```
You can't have everything. Where would you put it? - Steven Wright
You can't have everything. Where would you put it? - Steven Wright
```

Modifying Data Stored in Kernel Memory

In this third version of the Quote Of The Day driver, the user can write to the data that is stored in kernel memory. The pseudo device that is created in this section is a pseudo-disk device or ramdisk device. A ramdisk device simulates a disk device by allocating kernel memory that is subsequently used as data storage. See `ramdisk`(7D) for more information about ramdisk devices.

As in Version 2 of the Quote Of The Day driver, this Version 3 driver stores its data and makes the data available through its `read`(9E) entry point. This Version 3 driver overwrites characters from the beginning of the data when the user writes to the device.

This section first discusses the important code differences between this version and the previous version of the Quote Of The Day driver. This section then shows you how you can modify and display the quotation.

In addition to changes in the driver, Quote Of The Day Version 3 introduces a header file and an auxiliary program. The header file is discussed in the following section. The utility program is discussed in "Using Quote Of The Day Version 3" on page 97.

Writing Quote Of The Day Version 3

This third version of the Quote Of The Day driver is more complex than the second version because this third version enables a user to change the text that is stored in the device.

This section first explains some important features of this version of the driver. This section then shows all the source for this driver, including the header file and the configuration file.

The following list summarizes the new features in Version 3 of the Quote Of The Day driver:

- Version 3 of the driver allocates and frees kernel memory.
- Version 3 uses condition variables and mutexes to manage thread synchronization.
- Version 3 copies data from user space to kernel space to enable the user to change the quotation.
- Version 3 adds two new entry points: `write`(9E) and `ioctl`(9E).
- Version 3 adds a third new routine. The `qotd_rw` routine is called by both the `read`(9E) entry point and the `write`(9E) entry point.

- As in Version 2, Version 3 of the driver uses the `uiomove`(9F) function to make the quotation available to the user. Version 3 uses the `ddi_copyin`(9F) function to copy the new quotation and the new device size from user space to kernel space. Version 3 uses the `ddi_copyout`(9F) function to report the current device size back to the user.
- Because the driver copies data between kernel space and user space, Version 3 of the driver uses the `ddi_model_convert_from`(9F) function to determine whether the data must be converted between 32-bit and 64-bit models. The 64-bit kernel supports both 64-bit and 32-bit user data.
- Version 3 defines one new constant to tell the driver whether the device is busy. Another new constant tells the driver whether the quotation has been modified. Version 3 defines four new constants to help the driver undo everything it has done.
- Version 3 includes a separate utility program to test the driver's I/O controls.

The following sections provide more detail about the additions and changes in Version 3 of the Quote Of The Day driver. The `dev_ops`(9S) structure and the `modlinkage`(9S) structure are the same as they were in Version 2 of the driver. The `modldrv`(9S) structure has not changed except for the version number of the driver. The `_init`(9E), `_info`(9E), `_fini`(9E), `getinfo`(9E), `open`(9E), and `close`(9E) functions are the same as in Version 2 of the driver.

Attaching, Allocating Memory, and Initializing a Mutex and a Condition Variable

The `qotd_attach` entry point first allocates and gets the device soft state. The `qotd_attach` routine then creates a minor node. All of this code is the same as in Version 2 of the Quote Of The Day driver. If the call to `ddi_create_minor_node`(9F) is successful, the `qotd_attach` routine sets the `QOTD_DIDMINOR` flag in the new `flags` member of the `qotd_state` state structure.

Version 3 of the Quote Of The Day driver defines four new constants that keep track of four different events. A routine can test these flags to determine whether to deallocate, close, or remove resources. All four of these flags are set in the `qotd_attach` entry point. All four of these conditions are checked in the `qotd_detach` entry point, and the appropriate action is taken for each condition.

Note that operations are undone in the `qotd_detach` entry point in the opposite order in which they were done in the `qotd_attach` entry point. The `qotd_attach` routine creates a minor node, allocates memory for the quotation, initializes a mutex, and initializes a condition variable. The `qotd_detach` routine destroys the condition variable, destroys the mutex, frees the memory, and removes the minor node.

After the minor node is created, the `qotd_attach` routine allocates memory for the quotation. For information on allocating and freeing memory in this driver, see “Allocating and Freeing

Kernel Memory" on page 80. If memory is allocated, the qotd_attach routine sets the QOTD_DIDALLOC flag in the flags member of the state structure.

The qotd_attach routine then calls the mutex_init(9F) function to initialize a mutex. If this operation is successful, the qotd_attach routine sets the QOTD_DIDMUTEX flag. The qotd_attach routine then calls the cv_init(9F) function to initialize a condition variable. If this operation is successful, the qotd_attach routine sets the QOTD_DIDCV flag.

The qotd_attach routine then calls the strlcpy(9F) function to copy the initial quotation string to the new quotation member of the device state structure. Note that the strlcpy(9F) function is used instead of the strncpy(9F) function. The strncpy(9F) function can be wasteful because it always copies *n* characters, even if the destination is smaller than *n* characters. Try using strncpy(9F) instead of strlcpy(9F) and note the difference in the behavior of the driver.

Finally, the initial quotation length is copied to the new quotation length member of the state structure. The remainder of the qotd_attach routine is the same as in Version 2.

Checking for Changes, Cleaning Up, and Detaching

The qotd_detach routine is almost all new. The qotd_detach routine must first get the soft state because the qotd_detach routine needs to check the flags member of the state structure.

The first flag the qotd_detach routine checks is the QOTD_CHANGED flag. The QOTD_CHANGED flag indicates whether the device is in the initial state. The QOTD_CHANGED flag is set in the qotd_rw routine and in the qotd_ioctl entry point. The QOTD_CHANGED flag is set any time the user does anything to the device other than simply inspect the device. If the QOTD_CHANGED flag is set, the size or content of the storage buffer has been modified. See "Writing New Data" on page 84 for more information on the QOTD_CHANGED flag. When the QOTD_CHANGED flag is set, the detach operation fails because the device might contain data that is valuable to the user and the device should not be removed. If the QOTD_CHANGED flag is set, the qotd_detach routine returns an error that the device is busy.

Once the quotation has been modified, the device cannot be detached until the user runs the qotdctl command with the -r option. The -r option reinitializes the quotation and indicates that the user is no longer interested in the contents of the device. See "Exercising the Driver's I/O Controls" on page 98 for more information about the qotdctl command.

The qotd_detach routine then checks the four flags that were set in the qotd_attach routine. If the QOTD_DIDCV flag is set, the qotd_detach routine calls the cv_destroy(9F) function. If the QOTD_DIDMUTEX flag is set, the qotd_detach routine calls the mutex_destroy(9F) function. If the QOTD_DIDALLOC flag is set, the qotd_detach routine calls the ddi_umem_free(9F)

function. Finally, if the QOTD_DIDMINOR flag is set, the qotd_detach routine calls the ddi_remove_minor_node(9F) function.

Allocating and Freeing Kernel Memory

One of the new members of the device state structure supports memory allocation and deallocation. The qotd_cookie member receives a value from the ddi_umem_alloc(9F) function. The qotd_cookie value is then used by the ddi_umem_free(9F) function to free the memory.

Version 3 of the Quote Of The Day driver allocates kernel memory in three places:

- After the minor node is created
- In the QOTDIOCSSZ case of the qotd_ioctl entry point
- In the QOTDIOCDISCARD case of the qotd_ioctl entry point

The qotd_attach routine allocates memory after the minor node is created. Memory must be allocated to enable the user to modify the quotation. The qotd_attach routine calls the ddi_umem_alloc(9F) function with the DDI_UMEM_NOSLEEP flag so that the ddi_umem_alloc(9F) function will return immediately. If the requested amount of memory is not available, ddi_umem_alloc(9F) returns NULL immediately and does not wait for memory to become available. If no memory is allocated, qotd_attach calls qotd_detach and returns an error. If memory is allocated, qotd_attach sets the QOTD_DIDALLOC flag so that this memory will be freed by qotd_detach later.

The second place the driver allocates memory is in the QOTDIOCSSZ case of the qotd_ioctl entry point. The QOTDIOCSSZ case sets a new size for the device. A new size is set when the user runs the qotdctl command with the -s option. See “Exercising the Driver's I/O Controls” on page 98 for more information about the qotdctl command. This time, the ddi_umem_alloc(9F) function is called with the DDI_UMEM_SLEEP flag so that ddi_umem_alloc(9F) will wait for the requested amount of memory to be available. When the ddi_umem_alloc(9F) function returns, the requested memory has been allocated.

Note that you cannot always use the DDI_UMEM_SLEEP flag. See the CONTEXT sections of the ddi_umem_alloc(9F), kmem_alloc(9F), and kmem_zalloc(9F) man pages. Also note the behavioral differences among these three functions. The kmem_zalloc(9F) function is more efficient for small amounts of memory. The ddi_umem_alloc(9F) function is faster and better for large allocations. The ddi_umem_alloc(9F) function is used in this qotd_3 driver because ddi_umem_alloc(9F) allocates whole pages of memory. The kmem_zalloc(9F) function might save memory because it might allocate smaller chunks of memory. This

qotd_3 driver demonstrates a ramdisk device. In a production ramdisk device, you would use ddi_umem_alloc(9F) to allocate page-aligned memory.

After the current quotation is copied to the new space, the qotd_ioctl routine calls the ddi_umem_free(9F) function to free the memory that was previously allocated.

The third place the driver allocates memory is in the QOTDIOCDISCARD case of the qotd_ioctl entry point. The QOTDIOCDISCARD case is called from the qotdctl command. The qotdctl command with the -r option sets the quotation back to its initial value. If the number of bytes allocated for the current quotation is different from the initial number of bytes, then new memory is allocated to reinitialize the quotation. Again, the DDI_UMEM_SLEEP flag is used so that when the ddi_umem_alloc(9F) function returns, the requested memory has been allocated. The qotd_ioctl routine then calls the ddi_umem_free(9F) function to free the memory that was previously allocated.

Managing Thread Synchronization

The Quote Of The Day Version 3 driver uses condition variables and mutual exclusion locks (mutexes) together to manage thread synchronization. See the "Multithreaded Programming Guide " for more information about mutexes, condition variables, and thread synchronization.

In this driver, the mutex and condition variable both are initialized in the qotd_attach entry point and destroyed in the qotd_detach entry point. The condition variable is tested in the qotd_rw routine and in the qotd_ioctl entry point.

The condition variable waits on the QOTD_BUSY condition. This condition is needed because the driver must do some operations that rely on exclusive access to internal structures without holding a lock. Accessing the storage buffer or its metadata requires mutual exclusion, but the driver cannot hold a lock if the operation might sleep. Instead of holding a lock in this case, the driver waits on the QOTD_BUSY condition.

The driver acquires a mutex when the driver tests the condition variable and when the driver accesses the storage buffer. The mutex protects the storage buffer. Failure to use a mutual exclusion when accessing the storage buffer could allow one user process to resize the buffer while another user process tries to read the buffer, for example. The result of unprotected buffer access could be data corruption or a panic.

The condition variable is used when functions are called that might need to sleep. The ddi_copyin(9F), ddi_copyout(9F), and uiomove(9F) functions can sleep. Memory allocation can sleep if you use the SLEEP flag. Functions must not hold a mutex while they are sleeping. Sleeping while holding a mutex can cause deadlock. When a function might sleep, set the QOTD_BUSY flag and take the condition variable, which drops the mutex. To avoid race conditions, the QOTD_BUSY flag can be set or cleared only when holding the mutex. For more

information on deadlock, see “Using Mutual Exclusion Locks” in “Multithreaded Programming Guide ” and “Avoiding Deadlock” in “Multithreaded Programming Guide ”.

Locking Rules for Quote Of The Day Version 3

The locking rules for this `qotd_3` driver are as follows:

1. You must have exclusive access to do any of the following operations. To have exclusive access, you must own the mutex or you must set `QOTD_BUSY`. Threads must wait on `QOTD_BUSY`.
   - Test the contents of the storage buffer.
   - Modify the contents of the storage buffer.
   - Modify the size of the storage buffer.
   - Modify variables that refer to the address of the storage buffer.
2. If your operation does not need to sleep, do the following actions:
   a. Acquire the mutex.
   b. Wait until `QOTD_BUSY` is cleared. When the thread that set `QOTD_BUSY` clears `QOTD_BUSY`, that thread also should signal threads waiting on the condition variable and then drop the mutex.
   c. Perform your operation. You do not need to set `QOTD_BUSY` before you perform your operation.
   d. Drop the mutex.

   The following code sample illustrates this rule:

```
mutex_enter(&qsp->lock);
while (qsp->flags & QOTD_BUSY) {
        if (cv_wait_sig(&qsp->cv, &qsp->lock) == 0) {
                mutex_exit(&qsp->lock);
                ddi_umem_free(new_cookie);
                return (EINTR);
        }
}
memcpy(new_qotd, qsp->qotd, min(qsp->qotd_len, new_len));
ddi_umem_free(qsp->qotd_cookie);
qsp->qotd = new_qotd;
qsp->qotd_cookie = new_cookie;
qsp->qotd_len = new_len;
qsp->flags |= QOTD_CHANGED;
mutex_exit(&qsp->lock);
```

3. If your operation must sleep, do the following actions:
   a. Acquire the mutex.
   b. Set `QOTD_BUSY`.

c. Drop the mutex.
d. Perform your operation.
e. Reacquire the mutex.
f. Signal any threads waiting on the condition variable.
g. Drop the mutex.

These locking rules are very simple. These three rules ensure consistent access to the buffer and its metadata. Realistic drivers probably have more complex locking requirements. For example, drivers that use ring buffers or drivers that manage multiple register sets or multiple devices have more complex locking requirements.

Lock and Condition Variable Members of the State Structure

The device state structure for Version 3 of the Quote Of The Day driver contains two new members to help manage thread synchronization:

- The `lock` member is used to acquire and exit mutexes for the current instance of the device. The `lock` member is an argument to each `mutex`(9F) function call. The `lock` member also is an argument to the `cv_wait_sig`(9F) function call. In the `cv_wait_sig`(9F) function call, the `lock` value ensures that the condition will not be changed before the `cv_wait_sig`(9F) function returns.
- The `cv` member is a condition variable. The `cv` member is an argument to each `condvar`(9F) (`cv_`) function call.

Creating and Destroying Locks and Condition Variables

Version 3 of the Quote Of The Day driver defines two constants to make sure the mutex and condition variable are destroyed when the driver is finished with them. The driver uses these constants to set and reset the new `flags` member of the device state structure.

- The `QOTD_DIDMUTEX` flag is set in the `qotd_attach` entry point immediately after a successful call to `mutex_init`(9F). If the `QOTD_DIDMUTEX` flag is set when the `qotd_detach` entry point is called, the `qotd_detach` entry point calls the `mutex_destroy`(9F) function.
- The `QOTD_DIDCV` flag is set in the `qotd_attach` entry point immediately after a successful call to `cv_init`(9F). If the `QOTD_DIDCV` flag is set when the `qotd_detach` entry point is called, the `qotd_detach` entry point calls the `cv_destroy`(9F) function.

Waiting on Signals

In the `qotd_rw` and `qotd_ioctl` routines, the `cv_wait_sig`(9F) calls wait until the condition variable is signaled to proceed or until a `signal`(3C) is received. Either the `cv_signal`(9F) function or the `cv_broadcast`(9F) function signals the `cv` condition variable to proceed.

A thread can wait on a condition variable until either the condition variable is signaled or a `signal`(3C) is received by the process. The `cv_wait`(9F) function waits until the condition variable is signaled but ignores `signal`(3C) signals. This driver uses the `cv_wait_sig`(9F) function instead of the `cv_wait`(9F) function because this driver responds if a signal is received by the process performing the operation. If a `signal`(3C) is taken by the process, this driver returns an interrupt error and does not complete the operation. The `cv_wait_sig`(9F) function usually is preferred to the `cv_wait`(9F) function because this implementation offers the user program more precise response. The `signal`(3C) causes an effect closer to the point at which the process was executing when the `signal`(3C) was received.

In some cases, you cannot use the `cv_wait_sig`(9F) function because your driver cannot be interrupted by a `signal`(3C). For example, you cannot use the `cv_wait_sig`(9F) function during a DMA transfer that will result in an interrupt later. In this case, if you abandon the `cv_wait_sig`(9F) call, you have nowhere to put the data when the DMA transfer is finished, and your driver will panic.

Writing New Data

The `cb_ops`(9S) structure for Version 3 of the Quote Of The Day driver declares two new entry points that support modifying the quotation. The two new entry points are `write`(9E) and `ioctl`(9E). The `qotd_rw` routine is a third new routine in Version 3 of the driver. The `qotd_rw` routine is called by both the `read`(9E) entry point and the `write`(9E) entry point.

The device state structure for Version 3 of the Quote Of The Day driver contains two new members that are used to modify the quotation. The `qotd` string holds the quotation for the current instance of the device. The `qotd_len` member holds the length in bytes of the current quotation.

Version 3 of the driver also defines two new constants that support modifying the quotation. In place of `QOTD_MAXLEN`, Version 3 of the driver defines `QOTD_MAX_LEN`. `QOTD_MAX_LEN` is used in the `qotd_ioctl` entry point to test whether the user has entered a string that is too long. Version 3 of the driver also defines `QOTD_CHANGED`. The `QOTD_CHANGED` flag is set in the `qotd_rw` routine and in the `qotd_ioctl` entry point when a new quotation is copied from the user.

When the qotd_3 device is opened for writing, the kernel calls the qotd_write entry point. The qotd_write entry point then calls the qotd_rw routine and passes a UIO_WRITE flag. The new qotd_read entry point is exactly the same as the qotd_write entry point, except that the qotd_read entry point passes a UIO_READ flag. The qotd_rw routine supports both reading and writing the device and thereby eliminates much duplicate code.

The qotd_rw routine first gets the device soft state. Then the qotd_rw routine checks the length of the I/O request in the uio(9S) I/O request structure. If this length is zero, the qotd_rw routine returns zero. If this length is not zero, the qotd_rw routine enters a mutex.

While the device is busy, the qotd_rw routine checks whether the condition variable has been signaled or a signal(3C) is pending. If either of these conditions is true, the qotd_rw routine exits the mutex and returns an error.

When the device is not busy, the qotd_rw routine checks whether the data offset in the uio(9S) I/O request structure is valid. If the offset is not valid, the qotd_rw routine exits the mutex and returns an error. If the offset is valid, the local length variable is set to the difference between the offset in the I/O request structure and the length in the device state structure. If this difference is zero, the qotd_rw routine exits the mutex and returns. If the device was opened for writing, the qotd_rw routine returns a space error. Otherwise, the qotd_rw routine returns zero.

The qotd_rw routine then sets the QOTD_BUSY flag in the flags member of the device state structure and exits the mutex. The qotd_rw routine then calls the uiomove(9F) function to copy the quotation. If the rw argument is UIO_READ, then the quotation is transferred from the state structure to the I/O request structure. If the rw argument is UIO_WRITE, then the quotation is transferred from the I/O request structure to the state structure.

The qotd_rw routine then enters a mutex again. If the device was opened for writing, the qotd_rw routine sets the QOTD_CHANGED flag. The qotd_rw routine then sets the device to not busy, calls cv_broadcast(9F) to unblock any threads that were blocked on this condition variable, and exits the mutex.

Finally, the qotd_rw routine returns the quotation. The quotation is written to the device node.

Reporting and Setting Device Size and Re-initializing the Device

The behavior of the ioctl(9E) entry point depends on the command value passed in to the entry point. These constants are defined in the new qotd.h header file. The qotd_ioctl routine reports the size of the space allocated for the quotation, sets a new amount of space to allocate for the quotation, or resets the quotation back to its initial value.

If the request is to report the size of the space allocated for the quotation, then the `qotd_ioctl` routine first sets a local size variable to the value of the quotation length in the state structure. If the device was not opened for reading, the `qotd_ioctl` routine returns an error.

Because the `qotd_ioctl` routine transfers data between kernel space and user space, the `qotd_ioctl` routine must check whether both spaces are using the same data model. If the return value of the `ddi_model_convert_from`(9F) function is `DDI_MODEL_ILP32`, then the driver must convert to 32-bit data before calling `ddi_copyout`(9F) to transfer the current size of the quotation space. If the return value of the `ddi_model_convert_from`(9F) function is `DDI_MODEL_NONE`, then no data type conversion is necessary.

If the request is to set a new size for the space allocated for the quotation, then the `qotd_ioctl` routine first sets local variables for the new size, the new quotation, and a new memory allocation cookie. If the device was not opened for writing, the `qotd_ioctl` routine returns an error.

The `qotd_ioctl` routine then checks again for data model mismatch. If the return value of the `ddi_model_convert_from`(9F) function is `DDI_MODEL_ILP32`, then the driver declares a 32-bit size variable to receive the new size from `ddi_copyin`(9F). When the new size is received, the size is converted to the data type of the kernel space.

If the new size is zero or is greater than `QOTD_MAX_LEN`, the `qotd_ioctl` routine returns an error. If the new size is valid, then the `qotd_ioctl` routine allocates new memory for the quotation and enters a mutex.

While the device is busy, the `qotd_ioctl` routine checks whether the condition variable has been signaled or a `signal`(3C) is pending. If either of these conditions is true, the `qotd_ioctl` routine exits the mutex, frees the new memory it allocated, and returns an error.

When the device is not busy, the `qotd_ioctl` routine uses `memcpy`(9F) to copy the quotation from the driver's state structure to the new space. The `qotd_ioctl` routine then frees the memory currently pointed to by the state structure, and updates the state structure members to the new values. The `qotd_ioctl` routine then sets the `QOTD_CHANGED` flag, exits the mutex, and returns.

If the request is to discard the current quotation and reset to the initial quotation, then the `qotd_ioctl` routine first sets local variables for the new quotation and a new memory allocation cookie. If the device was not opened for writing, the `qotd_ioctl` routine returns an error. If the space allocated for the current quotation is different from the space allocated for the initial quotation, then the `qotd_ioctl` routine allocates new memory that is the size of the initial space and enters a mutex.

While the device is busy, the `qotd_ioctl` routine checks whether the condition variable has been signaled or a `signal`(3C) is pending. If either of these conditions is true, the `qotd_ioctl` routine exits the mutex, frees the new memory it allocated, and returns an error.

When the device is not busy, the `qotd_ioctl` routine frees the memory currently pointed to by the state structure, updates the memory state structure members to the new values, and resets the length to its initial value. If the size of the current quotation space was the same as the initial size and no new memory was allocated, then `qotd_ioctl` calls `bzero`(9F) to clear the current quotation. The `qotd_ioctl` routine then calls the `strlcpy`(9F) function to copy the initial quotation string to the quotation member of the state structure. The `qotd_ioctl` routine then unsets the `QOTD_CHANGED` flag, exits the mutex, and returns.

Once the `QOTD_CHANGED` flag has been set, the only way to unset it is to run the `qotdctl` command with the `-r` option. See "Exercising the Driver's I/O Controls" on page 98 for more information about the `qotdctl` command.

Quote Of The Day Version 3 Source

Enter the source code shown in the following example into a text file named `qotd_3.c`.

EXAMPLE 3-5 Quote Of The Day Version 3 Source File

```
#include <sys/types.h>
#include <sys/file.h>
#include <sys/errno.h>
#include <sys/open.h>
#include <sys/cred.h>
#include <sys/uio.h>
#include <sys/stat.h>
#include <sys/ksynch.h>
#include <sys/modctl.h>
#include <sys/conf.h>
#include <sys/devops.h>
#include <sys/debug.h>
#include <sys/cmn_err.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>

#include "qotd.h"

#define QOTD_NAME       "qotd_3"

static const char init_qotd[]
        = "On the whole, I'd rather be in Philadelphia. - W. C. Fields\n";
static const size_t init_qotd_len = 128;

#define QOTD_MAX_LEN    65536           /* Maximum quote in bytes */
#define QOTD_CHANGED    0x1             /* User has made modifications */
#define QOTD_DIDMINOR   0x2             /* Created minors */
#define QOTD_DIDALLOC   0x4             /* Allocated storage space */
#define QOTD_DIDMUTEX   0x8             /* Created mutex */
#define QOTD_DIDCV      0x10            /* Created cv */
#define QOTD_BUSY       0x20            /* Device is busy */
```

```
static void *qotd_state_head;

struct qotd_state {
        int             instance;
        dev_info_t      *devi;
        kmutex_t        lock;
        kcondvar_t      cv;
        char            *qotd;
        size_t          qotd_len;
        ddi_umem_cookie_t qotd_cookie;
        int             flags;
};

static int qotd_getinfo(dev_info_t *, ddi_info_cmd_t, void *, void **);
static int qotd_attach(dev_info_t *, ddi_attach_cmd_t);
static int qotd_detach(dev_info_t *, ddi_detach_cmd_t);
static int qotd_open(dev_t *, int, int, cred_t *);
static int qotd_close(dev_t, int, int, cred_t *);
static int qotd_read(dev_t, struct uio *, cred_t *);
static int qotd_write(dev_t, struct uio *, cred_t *);
static int qotd_rw(dev_t, struct uio *, enum uio_rw);
static int qotd_ioctl(dev_t, int, intptr_t, int, cred_t *, int *);

static struct cb_ops qotd_cb_ops = {
        qotd_open,              /* cb_open */
        qotd_close,             /* cb_close */
        nodev,                  /* cb_strategy */
        nodev,                  /* cb_print */
        nodev,                  /* cb_dump */
        qotd_read,              /* cb_read */
        qotd_write,             /* cb_write */
        qotd_ioctl,             /* cb_ioctl */
        nodev,                  /* cb_devmap */
        nodev,                  /* cb_mmap */
        nodev,                  /* cb_segmap */
        nochpoll,               /* cb_chpoll */
        ddi_prop_op,            /* cb_prop_op */
        (struct streamtab *)NULL,       /* cb_str */
        D_MP | D_64BIT,         /* cb_flag */
        CB_REV,                 /* cb_rev */
        nodev,                  /* cb_aread */
        nodev                   /* cb_awrite */
};

static struct dev_ops qotd_dev_ops = {
        DEVO_REV,               /* devo_rev */
        0,                      /* devo_refcnt */
        qotd_getinfo,           /* devo_getinfo */
        nulldev,                /* devo_identify */
        nulldev,                /* devo_probe */
        qotd_attach,            /* devo_attach */
        qotd_detach,            /* devo_detach */
        nodev,                  /* devo_reset */
        &qotd_cb_ops,           /* devo_cb_ops */
        (struct bus_ops *)NULL, /* devo_bus_ops */
        nulldev,                /* devo_power */
        ddi_quiesce_not_needed, /* devo_quiesce */
};
```

```
static struct modldrv modldrv = {
        &mod_driverops,
        "Quote of the day 3.0",
        &qotd_dev_ops};

static struct modlinkage modlinkage = {
        MODREV_1,
        (void *)&modldrv,
        NULL
};

int
_init(void)
{
        int retval;

        if ((retval = ddi_soft_state_init(&qotd_state_head,
            sizeof (struct qotd_state), 1)) != 0)
                return retval;
        if ((retval = mod_install(&modlinkage)) != 0) {
                ddi_soft_state_fini(&qotd_state_head);
                return (retval);
        }

        return (retval);
}

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

int
_fini(void)
{
        int retval;

        if ((retval = mod_remove(&modlinkage)) != 0)
                return (retval);
        ddi_soft_state_fini(&qotd_state_head);

        return (retval);
}

/*ARGSUSED*/
static int
qotd_getinfo(dev_info_t *dip, ddi_info_cmd_t cmd, void *arg, void **resultp)
{
        struct qotd_state *qsp;
        int retval = DDI_FAILURE;

        ASSERT(resultp != NULL);

        switch (cmd) {
        case DDI_INFO_DEVT2DEVINFO:
                if ((qsp = ddi_get_soft_state(qotd_state_head,
```

```
                    getminor((dev_t)arg))) != NULL) {
                        *resultp = qsp->devi;
                        retval = DDI_SUCCESS;
                } else
                        *resultp = NULL;
                break;
        case DDI_INFO_DEVT2INSTANCE:
                *resultp = (void *)getminor((dev_t)arg);
                retval = DDI_SUCCESS;
                break;
        }

        return (retval);
}

static int
qotd_attach(dev_info_t *dip, ddi_attach_cmd_t cmd)
{
        int instance = ddi_get_instance(dip);
        struct qotd_state *qsp;

        switch (cmd) {
        case DDI_ATTACH:
                if (ddi_soft_state_zalloc(qotd_state_head, instance)
                    != DDI_SUCCESS) {
                        cmn_err(CE_WARN, "Unable to allocate state for %d",
                            instance);
                        return (DDI_FAILURE);
                }
                if ((qsp = ddi_get_soft_state(qotd_state_head, instance))
                    == NULL) {
                        cmn_err(CE_WARN, "Unable to obtain state for %d",
                            instance);
                        ddi_soft_state_free(dip, instance);
                        return (DDI_FAILURE);
                }
                if (ddi_create_minor_node(dip, QOTD_NAME, S_IFCHR, instance,
                    DDI_PSEUDO, 0) != DDI_SUCCESS) {
                        cmn_err(CE_WARN, "Unable to create minor node for %d",
                            instance);
                        (void)qotd_detach(dip, DDI_DETACH);
                        return (DDI_FAILURE);
                }
                qsp->flags |= QOTD_DIDMINOR;
                qsp->qotd = ddi_umem_alloc(init_qotd_len, DDI_UMEM_NOSLEEP,
                    &qsp->qotd_cookie);
                if (qsp->qotd == NULL) {
                        cmn_err(CE_WARN, "Unable to allocate storage for %d",
                            instance);
                        (void)qotd_detach(dip, DDI_DETACH);
                        return (DDI_FAILURE);
                }
                qsp->flags |= QOTD_DIDALLOC;
                mutex_init(&qsp->lock, NULL, MUTEX_DRIVER, NULL);
                qsp->flags |= QOTD_DIDMUTEX;
                cv_init(&qsp->cv, NULL, CV_DRIVER, NULL);
                qsp->flags |= QOTD_DIDCV;
```

```
                (void)strlcpy(qsp->qotd, init_qotd, init_qotd_len);
                qsp->qotd_len = init_qotd_len;
                qsp->instance = instance;
                qsp->devi = dip;

                ddi_report_dev(dip);
                return (DDI_SUCCESS);
        case DDI_RESUME:
                return (DDI_SUCCESS);
        default:
                return (DDI_FAILURE);
        }
}


static int
qotd_detach(dev_info_t *dip, ddi_detach_cmd_t cmd)
{
        int instance = ddi_get_instance(dip);
        struct qotd_state *qsp;

        switch (cmd) {
        case DDI_DETACH:
                qsp = ddi_get_soft_state(qotd_state_head, instance);
                if (qsp != NULL) {
                        ASSERT(!(qsp->flags & QOTD_BUSY));
                        if (qsp->flags & QOTD_CHANGED)
                                return (EBUSY);
                        if (qsp->flags & QOTD_DIDCV)
                                cv_destroy(&qsp->cv);
                        if (qsp->flags & QOTD_DIDMUTEX)
                                mutex_destroy(&qsp->lock);
                        if (qsp->flags & QOTD_DIDALLOC) {
                                ASSERT(qsp->qotd != NULL);
                                ddi_umem_free(qsp->qotd_cookie);
                        }
                        if (qsp->flags & QOTD_DIDMINOR)
                                ddi_remove_minor_node(dip, NULL);
                }
                ddi_soft_state_free(qotd_state_head, instance);
                return (DDI_SUCCESS);
        case DDI_SUSPEND:
                return (DDI_SUCCESS);
        default:
                return (DDI_FAILURE);
        }
}

/*ARGSUSED*/
static int
qotd_open(dev_t *devp, int flag, int otyp, cred_t *credp)
{
        int instance = getminor(*devp);
        struct qotd_state *qsp;

        if ((qsp = ddi_get_soft_state(qotd_state_head, instance)) == NULL)
                return (ENXIO);
```

```
        ASSERT(qsp->instance == instance);

        if (otyp != OTYP_CHR)
                return (EINVAL);

        return (0);
}

/*ARGSUSED*/
static int
qotd_close(dev_t dev, int flag, int otyp, cred_t *credp)
{
        struct qotd_state *qsp;
        int instance = getminor(dev);

        if ((qsp = ddi_get_soft_state(qotd_state_head, instance)) == NULL)
                return (ENXIO);

        ASSERT(qsp->instance == instance);

        if (otyp != OTYP_CHR)
                return (EINVAL);

        return (0);
}

/*ARGSUSED*/
static int
qotd_read(dev_t dev, struct uio *uiop, cred_t *credp)
{
        return qotd_rw(dev, uiop, UIO_READ);
}

/*ARGSUSED*/
static int
qotd_write(dev_t dev, struct uio *uiop, cred_t *credp)
{
        return qotd_rw(dev, uiop, UIO_WRITE);
}

static int
qotd_rw(dev_t dev, struct uio *uiop, enum uio_rw rw)
{
        struct qotd_state *qsp;
        int instance = getminor(dev);
        size_t len = uiop->uio_resid;
        int retval;

        if ((qsp = ddi_get_soft_state(qotd_state_head, instance)) == NULL)
                return (ENXIO);

        ASSERT(qsp->instance == instance);

        if (len == 0)
                return (0);

        mutex_enter(&qsp->lock);
```

```
        while (qsp->flags & QOTD_BUSY) {
                if (cv_wait_sig(&qsp->cv, &qsp->lock) == 0) {
                        mutex_exit(&qsp->lock);
                        return (EINTR);
                }
        }

        if (uiop->uio_offset < 0 || uiop->uio_offset > qsp->qotd_len) {
                mutex_exit(&qsp->lock);
                return (EINVAL);
        }

        if (len > qsp->qotd_len - uiop->uio_offset)
                len = qsp->qotd_len - uiop->uio_offset;

        if (len == 0) {
                mutex_exit(&qsp->lock);
                return (rw == UIO_WRITE ? ENOSPC : 0);
        }

        qsp->flags |= QOTD_BUSY;
        mutex_exit(&qsp->lock);

        retval = uiomove((void *)(qsp->qotd + uiop->uio_offset), len, rw, uiop);

        mutex_enter(&qsp->lock);
        if (rw == UIO_WRITE)
                qsp->flags |= QOTD_CHANGED;
        qsp->flags &= ~QOTD_BUSY;
        cv_broadcast(&qsp->cv);
        mutex_exit(&qsp->lock);

        return (retval);
}

/*ARGSUSED*/
static int
qotd_ioctl(dev_t dev, int cmd, intptr_t arg, int mode, cred_t *credp,
    int *rvalp)
{
        struct qotd_state *qsp;
        int instance = getminor(dev);

        if ((qsp = ddi_get_soft_state(qotd_state_head, instance)) == NULL)
                return (ENXIO);

        ASSERT(qsp->instance == instance);

        switch (cmd) {
        case QOTDIOCGSZ: {
                /* We are not guaranteed that ddi_copyout(9F) will read
                 * automatically anything larger than a byte.  Therefore we
                 * must duplicate the size before copying it out to the user.
                 */
                size_t sz = qsp->qotd_len;

                if (!(mode & FREAD))
                        return (EACCES);
```

```
#ifdef _MULTI_DATAMODEL
                switch (ddi_model_convert_from(mode & FMODELS)) {
                case DDI_MODEL_ILP32: {
                        size32_t sz32 = (size32_t)sz;
                        if (ddi_copyout(&sz32, (void *)arg, sizeof (size32_t),
                            mode) != 0)
                                return (EFAULT);
                        return (0);
                }
                case DDI_MODEL_NONE:
                        if (ddi_copyout(&sz, (void *)arg, sizeof (size_t),
                            mode) != 0)
                                return (EFAULT);
                        return (0);
                default:
                        cmn_err(CE_WARN, "Invalid data model %d in ioctl\n",
                            ddi_model_convert_from(mode & FMODELS));
                        return (ENOTSUP);
                }
#else /* ! _MULTI_DATAMODEL */
                if (ddi_copyout(&sz, (void *)arg, sizeof (size_t), mode) != 0)
                        return (EFAULT);
                return (0);
#endif /* _MULTI_DATAMODEL */
        }
        case QOTDIOCSSZ: {
                size_t new_len;
                char *new_qotd;
                ddi_umem_cookie_t new_cookie;
                uint_t model;

                if (!(mode & FWRITE))
                        return (EACCES);

#ifdef _MULTI_DATAMODEL
                model = ddi_model_convert_from(mode & FMODELS);

                switch (model) {
                case DDI_MODEL_ILP32: {
                        size32_t sz32;
                        if (ddi_copyin((void *)arg, &sz32, sizeof (size32_t),
                            mode) != 0)
                                return (EFAULT);
                        new_len = (size_t)sz32;
                        break;
                }
                case DDI_MODEL_NONE:
                        if (ddi_copyin((void *)arg, &new_len, sizeof (size_t),
                            mode) != 0)
                                return (EFAULT);
                        break;
                default:
                        cmn_err(CE_WARN, "Invalid data model %d in ioctl\n",
                            model);
                        return (ENOTSUP);
                }
#else /* ! _MULTI_DATAMODEL */
```

```
                if (ddi_copyin((void *)arg, &new_len, sizeof (size_t),
                    mode) != 0)
                        return (EFAULT);
#endif /* _MULTI_DATAMODEL */

                if (new_len == 0 || new_len > QOTD_MAX_LEN)
                        return (EINVAL);

                new_qotd = ddi_umem_alloc(new_len, DDI_UMEM_SLEEP, &new_cookie);

                mutex_enter(&qsp->lock);
                while (qsp->flags & QOTD_BUSY) {
                        if (cv_wait_sig(&qsp->cv, &qsp->lock) == 0) {
                                mutex_exit(&qsp->lock);
                                ddi_umem_free(new_cookie);
                                return (EINTR);
                        }
                }
                memcpy(new_qotd, qsp->qotd, min(qsp->qotd_len, new_len));
                ddi_umem_free(qsp->qotd_cookie);
                qsp->qotd = new_qotd;
                qsp->qotd_cookie = new_cookie;
                qsp->qotd_len = new_len;
                qsp->flags |= QOTD_CHANGED;
                mutex_exit(&qsp->lock);

                return (0);
        }
        case QOTDIOCDISCARD: {
                char *new_qotd = NULL;
                ddi_umem_cookie_t new_cookie;

                if (!(mode & FWRITE))
                        return (EACCES);

                if (qsp->qotd_len != init_qotd_len) {
                        new_qotd = ddi_umem_alloc(init_qotd_len,
                            DDI_UMEM_SLEEP, &new_cookie);
                }

                mutex_enter(&qsp->lock);
                while (qsp->flags & QOTD_BUSY) {
                        if (cv_wait_sig(&qsp->cv, &qsp->lock) == 0) {
                                mutex_exit(&qsp->lock);
                                if (new_qotd != NULL)
                                        ddi_umem_free(new_cookie);
                                return (EINTR);
                        }
                }
                if (new_qotd != NULL) {
                        ddi_umem_free(qsp->qotd_cookie);
                        qsp->qotd = new_qotd;
                        qsp->qotd_cookie = new_cookie;
                        qsp->qotd_len = init_qotd_len;
                } else {
                        bzero(qsp->qotd, qsp->qotd_len);
                }
                (void)strlcpy(qsp->qotd, init_qotd, init_qotd_len);
```

```
                qsp->flags &= ~QOTD_CHANGED;
                mutex_exit(&qsp->lock);

                return (0);
        }
        default:
                return (ENOTTY);
        }
}
```

Enter the definitions shown in the following example into a text file named `qotd.h`.

EXAMPLE 3-6 Quote Of The Day Version 3 Header File

```
#ifndef _SYS_QOTD_H
#define _SYS_QOTD_H

#define QOTDIOC         ('q' << 24 | 't' << 16 | 'd' << 8)

#define QOTDIOCGSZ      (QOTDIOC | 1)   /* Get quote buffer size */
#define QOTDIOCSSZ      (QOTDIOC | 2)   /* Set new quote buffer size */
#define QOTDIOCDISCARD  (QOTDIOC | 3)   /* Discard quotes and reset */

#endif /* _SYS_QOTD_H */
```

Enter the configuration information shown in the following example into a text file named `qotd_3.conf`.

EXAMPLE 3-7 Quote Of The Day Version 3 Configuration File

```
name="qotd_3" parent="pseudo" instance=0;
```

Building and Installing Quote Of The Day Version 3

Compile and link the driver. The following example shows compiling and linking for a 32-bit architecture:

```
% cc -D_KERNEL -c qotd_3.c
% ld -r -o qotd_3 qotd_3.o
```

Make sure you are user `root` when you install the driver.

Copy the driver binary to the `/tmp` directory as discussed in “Building and Installing the Template Driver” on page 54.

```
# cp qotd_3 /tmp
# ln -s /tmp/qotd_3 /usr/kernel/drv/qotd_3
```

Copy the configuration file to the kernel driver area of the system.

```
# cp qotd_3.conf /usr/kernel/drv
```

In a separate window, enter the following command:

```
% tail -f /var/adm/messages
```

Make sure you are user `root` when you load the driver. Use the `add_drv`(1M) command to load the driver:

```
# add_drv qotd_3
```

You should see the following messages in the window where you are viewing `/var/adm/messages`:

```
date time machine pseudo: [ID 129642 kern.info] pseudo-device: qotd_30
date time machine genunix: [ID 936769 kern.info] qotd_30 is /pseudo/qotd_3@0
```

Using Quote Of The Day Version 3

This section describes how to read and write the `qotd_3` device and how to test the driver's I/O controls. The I/O controls include retrieving the size of the storage buffer, setting a new size for the storage buffer, and reinitializing the storage buffer size and contents.

Reading the Device

When you access this `qotd_3` device for reading, the command you use to access the device retrieves the data from the device node. The command then displays the data in the same way that the command displays any other input. To get the name of the device special file, look in the `/devices` directory:

```
% ls -l /devices/pseudo/qotd*
crw-------   1 root   sys   122,  0 date time /devices/pseudo/qotd_3@0:qotd_3
```

To read the `qotd_3` device, you can use the `cat`(1) command:

```
# cat /devices/pseudo/qotd_3@0:qotd_3
On the whole, I'd rather be in Philadelphia. - W. C. Fields
```

Writing the Device

To write to the `qotd_3` device, you can redirect command-line input:

```
# echo "A life is not important except in the impact it has on others.
- Jackie Robinson" >> /devices/pseudo/qotd_3@0:qotd_3
# cat /devices/pseudo/qotd_3@0:qotd_3
A life is not important except in the impact it has on others. - Jackie
```

```
Robinson
```

Exercising the Driver's I/O Controls

In addition to changes in the driver, Quote Of The Day Version 3 introduces a new utility program. The `qotdctl` command enables you to test the driver's I/O controls.

The source for this command is shown in Example 3-8. Compile the `qotdctl` utility as follows:

```
% cc -o qotdctl qotdctl.c
```

The `qotdctl` command has the following options:

`-g`	Get the size that is currently allocated. Call the `ioctl`(9E) entry point of the driver with the `QOTDIOCGSZ` request. The `QOTDIOCGSZ` request reports the current size of the space allocated for the quotation.
`-s` *size*	Set the new size to be allocated. Call the `ioctl`(9E) entry point of the driver with the `QOTDIOCSSZ` request. The `QOTDIOCSSZ` request sets a new size for the quotation space.
`-r`	Discard the contents and reset the device. Call the `ioctl`(9E) entry point of the driver with the `QOTDIOCDISCARD` request. Invoking `qotdctl` with the `-r` option is the only way to unset the `QOTD_CHANGED` flag in the device. The device cannot be detached while the `QOTD_CHANGED` flag is set. This protects the contents of the ramdisk device from being unintentionally or automatically removed. For example, a device might be automatically removed by the automatic device unconfiguration thread. When you are no longer interested in the contents of the device, run the `qotdctl` command with the `-r` option. Then you can remove the device.
`-h`	Display help text.
`-V`	Display the version number of the `qotdctl` command.
`-d` *device*	Specify the device node to use. The default value is `/dev/qotd0`.

Use the `qotdctl` command to test the driver's I/O controls:

```
# ./qotdctl -V
qotdctl 1.0
# ./qotdctl -h
Usage: ./qotdctl [-d device] {-g | -h | -r | -s size | -V}
# ./qotdctl -g
open: No such file or directory
```

By default, the `qotdctl` command accesses the `/dev/qotd0` device. The `qotd_3` device in this example is `/devices/pseudo/qotd_3@0:qotd_3`. Either define a link from `/dev/qotd0` to `/devices/pseudo/qotd_3@0:qotd_3` or use the `-d` option to specify the correct device:

```
# ./qotdctl -d /devices/pseudo/qotd_3@0:qotd_3 -g
128
# ./qotdctl -d /devices/pseudo/qotd_3@0:qotd_3 -s 512
# ./qotdctl -d /devices/pseudo/qotd_3@0:qotd_3 -g
512
# ./qotdctl -d /devices/pseudo/qotd_3@0:qotd_3 -r
# cat /devices/pseudo/qotd_3@0:qotd_3
On the whole, I'd rather be in Philadelphia. - W. C. Fields
```

If you try to remove the device now, you will receive an error message:

```
# rem_drv qotd_3
Device busy
Cannot unload module: qotd_3
Will be unloaded upon reboot.
```

The device is still marked busy because you have not told the driver that you are no longer interested in this device. Run the `qotdctl` command with the `-r` option to unset the `QOTD_CHANGED` flag in the driver and mark the device not busy:

```
# ./qotdctl -r
```

Enter the source code shown in the following example into a text file named `qotdctl.c`.

EXAMPLE 3-8 Quote Of The Day I/O Control Command Source File

```
#include <sys/ioctl.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <stdlib.h>

#include "qotd.h"

static const char *DEFAULT_DEVICE = "/dev/qotd0";
static const char *VERSION = "1.0";

static void show_usage(const char *);
static void get_size(const char *);
static void set_size(const char *, size_t);
static void reset_dev(const char *);

int
main(int argc, char *argv[])
{
        int op = -1;
        int opt;
        int invalid_usage = 0;
        size_t sz_arg;
```

```
const char *device = DEFAULT_DEVICE;

while ((opt = getopt(argc, argv,
    "d:(device)g(get-size)h(help)r(reset)s:(set-size)V(version)"))
    != -1) {
        switch (opt) {
        case 'd':
                device = optarg;
                break;
        case 'g':
                if (op >= 0)
                        invalid_usage++;
                op = QOTDIOCGSZ;
                break;
        case 'h':
                show_usage(argv[0]);
                exit(0);
                /*NOTREACHED*/
        case 'r':
                if (op >= 0)
                        invalid_usage++;
                op = QOTDIOCDISCARD;
                break;
        case 's':
                if (op >= 0)
                        invalid_usage++;
                op = QOTDIOCSSZ;
                sz_arg = (size_t)atol(optarg);
                break;
        case 'V':
                (void) printf("qotdctl %s\n", VERSION);
                exit(0);
                /*NOTREACHED*/
        default:
                invalid_usage++;
                break;
        }
}

if (invalid_usage > 0 || op < 0) {
        show_usage(argv[0]);
        exit(1);
}

switch (op) {
case QOTDIOCGSZ:
        get_size(device);
        break;
case QOTDIOCSSZ:
        set_size(device, sz_arg);
        break;
case QOTDIOCDISCARD:
        reset_dev(device);
        break;
default:
        (void) fprintf(stderr,
            "internal error - invalid operation %d\n", op);
        exit(2);
```

```
        }

        return (0);
}

static void
show_usage(const char *execname)
{
        (void) fprintf(stderr,
         "Usage: %s [-d device] {-g | -h | -r | -s size | -V}\n", execname);
}

static void
get_size(const char *dev)
{
        size_t sz;
        int fd;

        if ((fd = open(dev, O_RDONLY)) < 0) {
                perror("open");
                exit(3);
        }

        if (ioctl(fd, QOTDIOCGSZ, &sz) < 0) {
                perror("QOTDIOCGSZ");
                exit(4);
        }

        (void) close(fd);

        (void) printf("%zu\n", sz);
}

static void
set_size(const char *dev, size_t sz)
{
        int fd;

        if ((fd = open(dev, O_RDWR)) < 0) {
                perror("open");
                exit(3);
        }

        if (ioctl(fd, QOTDIOCSSZ, &sz) < 0) {
                perror("QOTDIOCSSZ");
                exit(4);
        }

        (void) close(fd);
}

static void
reset_dev(const char *dev)
{
        int fd;

        if ((fd = open(dev, O_RDWR)) < 0) {
                perror("open");
```

```
                exit(3);
        }

        if (ioctl(fd, QOTDIOCDISCARD) < 0) {
                perror("QOTDIOCDISCARD");
                exit(4);
        }

        (void) close(fd);
}
```

♦ ♦ ♦ CHAPTER 4

Tips for Developing Device Drivers

This chapter provides some general guidelines for writing device drivers.

The guidelines are organized into the following categories:

- "Device Driver Coding Tips" on page 103
- "Device Driver Testing Tips" on page 106
- "Device Driver Debugging and Tuning Tips" on page 108

Device Driver Coding Tips

Use these guidelines when you write the code for your driver:

- Use a prefix based on the name of your driver to give global variables and functions unique names.

  The name of each function, data element, and driver preprocessor definition must be unique for each driver.

  A driver module is linked into the kernel. The name of each symbol unique to a particular driver must not collide with other kernel symbols. To avoid such collisions, each function and data element for a particular driver must be named with a prefix common to that driver. The prefix must be sufficient to uniquely name each driver symbol. Typically, this prefix is the name of the driver or an abbreviation for the name of the driver. For example, `xx_open` would be the name of the `open`(9E) routine of driver `xx`.

  When building a driver, a driver must necessarily include a number of system header files. The globally-visible names within these header files cannot be predicted. To avoid collisions with these names, each driver preprocessor definition must be given a unique name by using an identifying prefix.

  A distinguishing driver symbol prefix also is an aid to deciphering system logs and panics when troubleshooting. Instead of seeing an error related to an ambiguous `attach` function, you see an error message about `xx_attach`.
- If you are basing your design on an existing driver, modify the configuration file before adding the driver.

The -n option in the add_drv(1M) command enables you to update the system configuration files for a driver without loading or attaching the driver.

- Use the cmn_err function to log driver activity.

  You can use the cmn_err(9F) function to display information from your driver similar to the way you might use print statements to display information from a user program. The cmn_err(9F) function writes low priority messages to /dev/log. The syslogd(1M) daemon reads messages from /dev/log and writes low priority messages to /var/adm/messages. Use the following command to monitor the output from your cmn_err(9F) messages:

  ```
  % tail -f /var/adm/messages
  ```

  Be sure to remove cmn_err calls that are used for development or debugging before you compile your production version driver. You might want to use cmn_err calls in a production driver to write error messages that would be useful to a system administrator.
- Clean up allocations and other initialization activities when the driver exits.

  When the driver exits, whether intentionally or prematurely, you need to perform such tasks as closing opened files, freeing allocated memory, releasing mutex locks, and destroying any mutexes that have been created. In addition, the system must be able to close all minor devices and detach driver instances even after the hardware fails. An orderly approach is to reverse _init actions in the _fini routine, reverse open operations in the close routine, and reverse attach operations in the detach routine.
- Use ASSERT(9F) to catch unexpected error returns.

  ASSERT is a macro that halts the kernel execution if a condition that was expected to be true turns out to be false. To activate ASSERT, you need to include the sys/debug.h header file and specify the DEBUG preprocessor symbol during compilation.
- Use mutex_owned to validate and document locking requirements.

  The mutex_owned(9F) function helps determine whether the current thread owns a specified mutex. To determine whether a mutex is held by a thread, use mutex_owned within ASSERT.
- Use conditional compilation to toggle “costly” debugging features.

  The OS provides various debugging functions, such as ASSERT and mutex-owned, that can be turned on by specifying the DEBUG preprocessor symbol when the driver is compiled. With conditional compilation, unnecessary code can be removed from the production driver. This approach can also be accomplished by using a global variable.
- Use a separate instance of the driver for each device to be controlled.
- Use DDI functions as much as possible in your device drivers.

  These interfaces shield the driver from platform-specific dependencies such as mismatches between processor and device endianness and any other data order dependencies. With

these interfaces, a single-source driver can run on the SPARC platform, x86 platform, and related processor architectures.

- Anticipate corrupted data.

  Always check that the integrity of data before that data is used. The driver must avoid releasing bad data to the rest of the system.

- A device should only write to DMA buffers that are controlled solely by the driver.

  This technique prevents a DMA fault from corrupting an arbitrary part of the system's main memory.

- Use the `ddi_umem_alloc`(9F) function when you need to make DMA transfers.

  This function guarantees that only whole, aligned pages are transferred.

- Set a fixed number of attempts before taking alternate action to deal with a stuck interrupt.

  The device driver must not be an unlimited drain on system resources if the device locks up. The driver 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.

- Use care when setting the sequence for mutex acquisitions and releases so as to avoid unwanted thread interactions if a device fails.

- Check for malformed `ioctl` requests from user applications.

  User requests can be destructive. The design of the driver should take into consideration the construction of each type of potential `ioctl` request.

- Try to avoid situations where a driver continues to function without detecting a device failure.

  A driver should switch to an alternative device rather than try to work around a device failure.

- All device drivers in the OS must support hotplugging.

  All devices need to be able to be installed or removed without requiring a reboot of the system.

- All device drivers should support 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.

- Apply the `volatile` keyword to any variable that references a device register.

  Without the `volatile` keyword, the compile-time optimizer can delete important accesses to a register.

- Perform periodic health checks to detect and report faulty devices.

  A periodic health check should include the following activities:

  - Check any register or memory location on the device whose value might have been altered since the last poll.
  - Timestamp outgoing requests such as transmit blocks or commands that are issued by the driver.

- Initiate a test action on the device that should be completed before the next scheduled check.

Device Driver Testing Tips

Testing a device driver can cause the system to panic and can harm the kernel.

The following tips can help you avoid problems when testing your driver:

- Install the driver in a temporary location.

  Install drivers in the `/tmp` directory until you are finished modifying and testing the `_info`, `_init`, and `attach` routines. Copy the driver binary to the `/tmp` directory. Link to the driver from the kernel driver directory.

  If a driver has an error in its `_info`, `_init`, or `attach` function, your machine could get into a state of infinite panic. The OS automatically reboots itself after a panic and loads any drivers it can during boot. If you have an error in your `attach` function that panics the system when you load the driver, then the system will panic again when it tries to reboot after the panic. The system will continue the cycle of panic, reboot, panic as it attempts to reload the faulty driver every time it reboots after panic.

  To avoid an infinite panic, keep the driver in the `/tmp` area until it is well tested. Link to the driver in the `/tmp` area from the kernel driver area. The OS removes all files from the `/tmp` area every time the system reboots. If your driver causes a panic, the OS reboots successfully because the driver has been removed automatically from the `/tmp` area. The link in the kernel driver area points to nothing. The faulty driver did not get loaded, so the system does not go back into a panic. You can modify the driver, copy it again to the `/tmp` area, and continue testing and developing. When the driver is well tested, copy it to the `/usr/kernel/drv` area so that it will remain available after a reboot.

  The following example shows you where to link the driver for a 32-bit platform. For other architectures, see the instructions in "Installing a Driver" on page 27.

  ```
  # cp mydriver /tmp
  # ln -s /tmp/mydriver /usr/kernel/drv/mydriver
  ```

- Enable the deadman feature to avoid a hard hang.

  If your system is in a hard hang, then you cannot break into the debugger. If you enable the deadman feature, the system panics instead of hanging indefinitely. You can then use the `kmdb(1)` kernel debugger to analyze your problem.

  The deadman feature checks every second whether the system clock is updating. If the system clock is not updating, then you are in an indefinite hang. If the system clock has not been updated for 50 seconds, the deadman feature induces a panic and puts you in the debugger.

  Take the following steps to enable the deadman feature:

1. Make sure you are capturing crash images with dumpadm(1M).
2. Set the snooping variable in the /etc/system file.

   ```
   set snooping=1
   ```

3. Reboot the system so that the /etc/system file is read again and the snooping setting takes effect.

Note that any zones on your system inherit the deadman setting as well.

If your system hangs while the deadman feature is enabled, you should see output similar to the following example on your console:

```
panic[cpu1]/thread=30018dd6cc0: deadman: timed out after 9 seconds of
clock inactivity

panic: entering debugger (continue to save dump)
```

Inside the debugger, use the ::cpuinfo command to investigate why the clock interrupt was not able to fire and advance the system time.

- Use a serial connection to control your test machine from a separate host system.

  This technique is explained in "Testing With a Serial Connection" in "Writing Device Drivers for Oracle Solaris 11.2 ".
- Use an alternate kernel.

  Booting from a copy of the kernel and the associated binaries rather than from the default kernel avoids inadvertently rendering the system inoperable.
- Use an additional kernel module to experiment with different kernel variable settings.

  This approach isolates experiments with the kernel variable settings. See "Setting Up Test Modules" in "Writing Device Drivers for Oracle Solaris 11.2 ".
- Make contingency plans for potential data loss on a test system.

  If your test system is set up as a client of a server, then you can boot from the network if problems occur. You could also create a special partition to hold a copy of a bootable root file system. See "Avoiding Data Loss on a Test System" in "Writing Device Drivers for Oracle Solaris 11.2 ".
- Capture system crash dumps if your test system panics.
- Use fsck(1M) to repair the damaged root file system temporarily if your system crashes during the attach(9E) process so that any crash dumps can be salvaged. See "Recovering the Device Directory" in "Writing Device Drivers for Oracle Solaris 11.2 ".
- Install drivers in the /tmp directory until you are finished modifying and testing the _info, _init, and attach routines.

  Keep a driver in the /tmp directory until the driver has been well tested. If a panic occurs, the driver will be removed from /tmp directory and the system will reboot successfully.

Device Driver Debugging and Tuning Tips

The Oracle Solaris OS provides various tools for debugging and tuning your device driver:

- You might receive the following warning message from the `add_drv`(1M) command:

  ```
  Warning: Driver (driver_name) successfully added to system but failed to attach
  ```

  This message might have one of the following causes:
  - The hardware has not been detected properly. The system cannot find the device.
  - The configuration file is missing. See "Writing a Configuration File" on page 25 for information on when you need a configuration file and what information goes into a configuration file. Be sure to put the configuration file in `/kernel/drv` or `/usr/kernel/drv` and *not* in the driver directory.
- Use the `kmdb`(1) kernel debugger for runtime debugging.

  The `kmdb` debugger provides typical runtime debugger facilities, such as breakpoints, watch points, and single-stepping. For more information, see "Oracle Solaris Modular Debugger Guide ".
- Use the `mdb`(1) modular debugger for postmortem debugging.

  Postmortem debugging is performed on a system crash dump rather than on a live system. With postmortem debugging, the same crash dump can be analyzed by different people or processes simultaneously. In addition, `mdb` enables you to create special macros called *dmods* to perform rigorous analysis on the dump. For more information, see "Oracle Solaris Modular Debugger Guide ".
- Use the `kstat`(3KSTAT) facility to export module-specific kernel statistics for your device driver.
- Use the DTrace facility to add instrumentation to your driver dynamically so that you can perform tasks such as analyzing the system and measuring performance. For information on DTrace, see the "Oracle Solaris 11.2 Dynamic Tracing Guide " .
- If your driver does not behave as expected on a 64-bit platform, make sure you are using a 64-bit driver. By default, compilation on the Oracle Solaris OS yields a 32-bit result on every architecture. To obtain a 64-bit result, follow the instructions in "Building a Driver" on page 25.

  Use the `file`(1) command to determine whether you have a 64-bit driver.

  ```
  % file qotd_3
  qotd_3:         ELF 32-bit LSB relocatable 80386 Version 1
  ```
- If you are using a 64-bit system and you are not certain whether you are currently running the 64-bit kernel or the 32-bit kernel, use the `-k` option of the `isainfo`(1) command. The `-v` option reports all instruction set architectures of the system. The `-k` option reports the instruction set architecture that is currently in use.

```
% isainfo -v
64-bit sparcv9 applications
        vis2 vis
32-bit sparc applications
        vis2 vis v8plus div32 mul32
% isainfo -kv
64-bit sparcv9 kernel modules
```

- If your driver seems to have an error in a function that you did not write, make sure you have called that function with the correct arguments and specified the correct include files. Many kernel functions have the same names as system calls and user functions. For example, `read` and `write` can be system calls, user library functions, or kernel functions. Similarly, `ioctl` and `mmap` can be system calls or kernel functions. The `man mmap` command displays the `mmap`(2) man page. To see the arguments, description, and include files for the kernel function, use the `man mmap.9e` command. If you do not know whether the function you want is in section 9E or section 9F, use the `man -l mmap` command, for example.

Index

Numbers and Symbols

/dev directory, 20, 21
/devices directory, 20, 20, 22, 28
/devices/pseudo directory, 21, 55, 76
/etc/driver_aliases file, 28
/etc/name_to_major file, 28, 56, 76
/usr/kernel directory, 19, 19
/var/adm/messages file, 55, 65
_fini entry point, 32, 35, 68
_info entry point, 27, 32, 34, 106
_init entry point, 27, 32, 34, 68, 106

A

add_drv command, 28, 55
  use in modifying existing drivers, 103
alternate kernels
  use in testing, 107
ASSERT kernel function, 66, 70, 75, 104
attach entry point, 27, 37, 39, 68, 106

B

blk device, 21
block device, 21
boot command, 19
bzero kernel function, 87

C

cat command, 56
cb_ops driver structure, 16, 48, 50, 84
cc command, 25
character device, 21
close entry point, 44, 46, 69
cmn_err kernel function, 34, 55, 63, 104
commands
  add_drv, 28, 55, 103
  boot, 19
  cat, 56
  cc, 25
  dmesg, 69
  echo, 57
  fsck, 107
  gcc, 26
  kernel, 18
  ld, 15, 25, 36
  mknod, 20
  modinfo, 28, 56, 76
  modload, 56
  modunload, 58
  more, 76
  prtconf, 20, 20, 25, 29, 56, 76
  prtpicl, 20
  rem_drv, 29, 29, 57
  syslogd, 55, 65
  update_drv, 29
compiling, 25
condition variables, 81
conditional compilation, 104
condvar kernel functions, 83
configuration files, 25, 54
crash dumps
  use in testing, 107
cv_broadcast kernel function, 84, 85
cv_destroy kernel function, 79, 83
cv_init kernel function, 79, 83
cv_signal kernel function, 84

cv_wait kernel function, 84
cv_wait_sig kernel function, 83, 84

D

data loss
  avoiding while testing, 107
data model
  converting, 24, 78
ddi_copyin kernel function, 16, 78, 81, 86
ddi_copyout kernel function, 16, 78, 86
ddi_create_minor_node kernel function, 37, 40, 68, 78
ddi_get_instance kernel function, 23, 40, 68
ddi_get_soft_state kernel function, 67, 68, 69, 70
ddi_model_convert_from kernel function, 24, 78, 86
ddi_prop_get_int kernel function, 25
ddi_prop_lookup kernel function, 25
ddi_prop_op kernel function, 37, 43
ddi_remove_minor_node kernel function, 37, 41, 68, 79
ddi_report_dev kernel function, 69
ddi_soft_state kernel function, 39
ddi_soft_state_fini kernel function, 67, 68
ddi_soft_state_free kernel function, 67, 68
ddi_soft_state_init kernel function, 67, 68
ddi_soft_state_zalloc kernel function, 67, 68
ddi_umem_alloc kernel function, 80
ddi_umem_free kernel function, 79, 80
deadman kernel feature, 106
debugging device drivers
  tips, 108
detach entry point, 36, 37, 41, 69
dev_info device structure, 40, 41, 42
dev_ops driver structure, 16, 48, 51
devfs devices file system, 20
devfsadmd devices file system administration command, 21
device drivers, 16
  adding, 28
  coding tips, 103
  compiling, 25
  condition variables, 81
  conditional compilation, 104
  debugging tips, 108
  development guidelines, 103
  directories, 18
    adding, 19
  entry points, 14, 16, 17, 32
    See also entry points
  how used, 18
  I/O controls, 85, 98
  installing, 27, 106
  linking, 25
  loading, 19, 29, 55
  mutexes, 81
  naming conventions, 103
  recommended housekeeping, 104
  removing, 29, 57
  structures *See* driver structures
  test areas, 29
  testing, 106
  thread synchronization, 81
  tuning, 108
  unloading, 29, 58
  updating, 29
device instance pointer (dip), 40, 41, 42
device number, 22
device structures
  dev_info, 40, 41, 42
device tree, 20
devices
  blk, 21
  block, 19, 21
  character, 19, 21, 32
  configuration files, 25, 54
  device tree, 20
  directories, 20, 21
  exclusive access, 82
  file system
    devfs, 20
    devfsadmd, 21
  instances, 22, 40, 41, 42
  md metadevice, 21
  names, 21
  nexus, 17, 20
  numbers, 20, 22, 40
  prefixes, 24, 38
  properties, 25, 43
  pseudo, 17, 31

ramdisk, 17, 77
raw, 21
reading, 56, 76, 97
special files, 19
state, 67
writing, 57, 77, 84, 97
devmap entry point, 16
dmesg command, 69
driver structures
cb_ops, 16, 48, 50, 84
character and block operations structure, 50
dev_ops, 16, 48, 51
device operations structure, 51
modinfo, 34
modldrv, 48, 52
modlinkage, 34, 48, 52
module linkage structures, 52
driver.conf file, 25
drivers *See* device drivers
DTrace analyzer, 108

E

echo command, 57
entry points
_fini, 32, 35, 68
_info, 27, 32, 34, 106
_init, 27, 32, 34, 68, 106
attach, 27, 37, 39, 68, 106
autoconfiguration, 37
close, 44, 46, 69
detach, 36, 37, 41, 69
devmap, 16
getinfo, 37, 42, 69
ioctl, 19, 84, 85, 98
loadable module configuration, 32
open, 44, 46, 69
prop_op, 37, 43
read, 44, 47, 70
user context, 44
write, 44, 47, 84

F

files
/etc/name_to_major, 28, 56, 76
/var/adm/messages, 55, 65
driver.conf, 25
system, 18
fsck command, 107
functions
kstat, 108
printf, 104
signal, 84, 85, 86

G

gcc command, 26
getinfo entry point, 37, 42, 69
getminor kernel function, 69, 70
GNU C, 26

H

hotplugging, 105

I

I/O controls, 85, 98
instance number, 22, 40, 41, 42
interrupts
avoiding problems, 105
ioctl entry point, 19, 84, 85, 98
ioctl requests
avoiding problems, 105

K

kernel, 13
address space, 14, 16
privilege, 14
See also kernel mode
kernel command, 18
kernel functions
ASSERT, 66, 70, 75, 104
bzero, 87
cmn_err, 34, 55, 63, 104

condvar, 83
cv_broadcast, 84, 85
cv_destroy, 79, 83
cv_init, 79, 83
cv_signal, 84
cv_wait, 84
cv_wait_sig, 83, 84
ddi_copyin, 16, 78, 81, 86
ddi_copyout, 16, 78, 86
ddi_create_minor_node, 37, 40, 68, 78
ddi_get_instance, 23, 40, 68
ddi_get_soft_state, 67, 68, 69, 70
ddi_model_convert_from, 24, 78, 86
ddi_prop_get_int, 25
ddi_prop_lookup, 25
ddi_prop_op, 37, 43
ddi_remove_minor_node, 37, 41, 68, 79
ddi_report_dev, 69
ddi_soft_state, 39
ddi_soft_state_fini, 67, 68
ddi_soft_state_free, 67, 68
ddi_soft_state_init, 67, 68
ddi_soft_state_zalloc, 67, 68
ddi_umem_alloc, 80
ddi_umem_free, 79, 80
getminor, 69, 70
kmem_alloc, 80
kmem_zalloc, 80
memcpy, 86
mod_info, 32, 34
mod_install, 32, 34, 68
mod_remove, 32, 35, 68
mutex, 83
mutex_destroy, 79, 83
mutex_init, 79, 83
mutex_owned, 104
nochpoll, 50
nodev, 50, 52
nulldev, 44, 52
strlcpy, 79, 87
strncpy, 79
uiomove, 70, 78, 81, 85
kernel mode, 13
kernel modules
  use in testing, 107
kernel statistics, 108
kernel structures
  uio, 70, 85
kmdb kernel debugger, 106, 108
kmem_alloc kernel function, 80
kmem_zalloc kernel function, 80
kstat function, 108

L

ld command, 15, 25, 36
linking, 15, 25, 36

M

major number, 20, 22
mdb modular debugger, 108
memcpy kernel function, 86
metadevice, 21
minor number, 20, 22, 40
mknod command, 20
mknod system call, 20
mmap system call, 16
mod_info kernel function, 32, 34
mod_install kernel function, 32, 34, 68
mod_remove kernel function, 32, 35, 68
moddir kernel variable, 19
modinfo command, 28, 56, 76
modinfo driver structure, 34
modldrv driver structure, 48, 52
modlinkage driver structure, 34, 48, 52
modload command, 56
modunload command, 58
more command, 76
mutex kernel function, 83
mutex_destroy kernel function, 79, 83
mutex_init kernel function, 79, 83
mutex_owned kernel function, 104
mutexes, 81
  avoiding problems, 105

N

naming
  unique prefix for driver symbols, 103
naming conventions, 103
nexus device, 20
`nochpoll` kernel function, 50
`nodev` kernel function, 50, 52
`nulldev` kernel function, 44, 52

O

`open` entry point, 44, 46, 69
Oracle Solaris Studio, 25

P

PCI ID numbers, 28
power management, 105
prefix
  unique prefix for driver symbols, 103
prefixes, 24, 38
`printf` function, 104
`prop_op` entry point, 37, 43
protected mode, 13
`prtconf` command, 20, 20, 25, 29, 56, 76
`prtpicl` command, 20

Q

`QOTD_BUSY` condition, 81, 82

R

`raw` device, 21
`read` entry point, 44, 47, 70
`read` system call, 19
`rem_drv` command, 29, 29, 57
restricted mode, 13

S

serial connections
  use in testing, 107
`signal` function, 84, 85, 86
snooping kernel variable, 106
soft state, 67
SPARC
  address space, 16
  compiling, 25
special files, 19
state structures, 66, 67
`strlcpy` kernel function, 79, 87
`strncpy` kernel function, 79
`syslogd` command, 55, 65
system calls
  `mknod`, 20
  `mmap`, 16
  `read`, 19
`system` configuration information file, 18
system crash dumps
  use in testing, 107

T

testing device drivers, 106
thread synchronization, 81
tuning device drivers
  tips, 108

U

`uio` kernel structure, 70, 85
`uiomove` kernel function, 70, 78, 81, 85
`update_drv` command, 29
user mode, 13

V

`volatile` keyword, 105

W

`write` entry point, 44, 47, 84

X

x86

address space, 16
compiling, 25