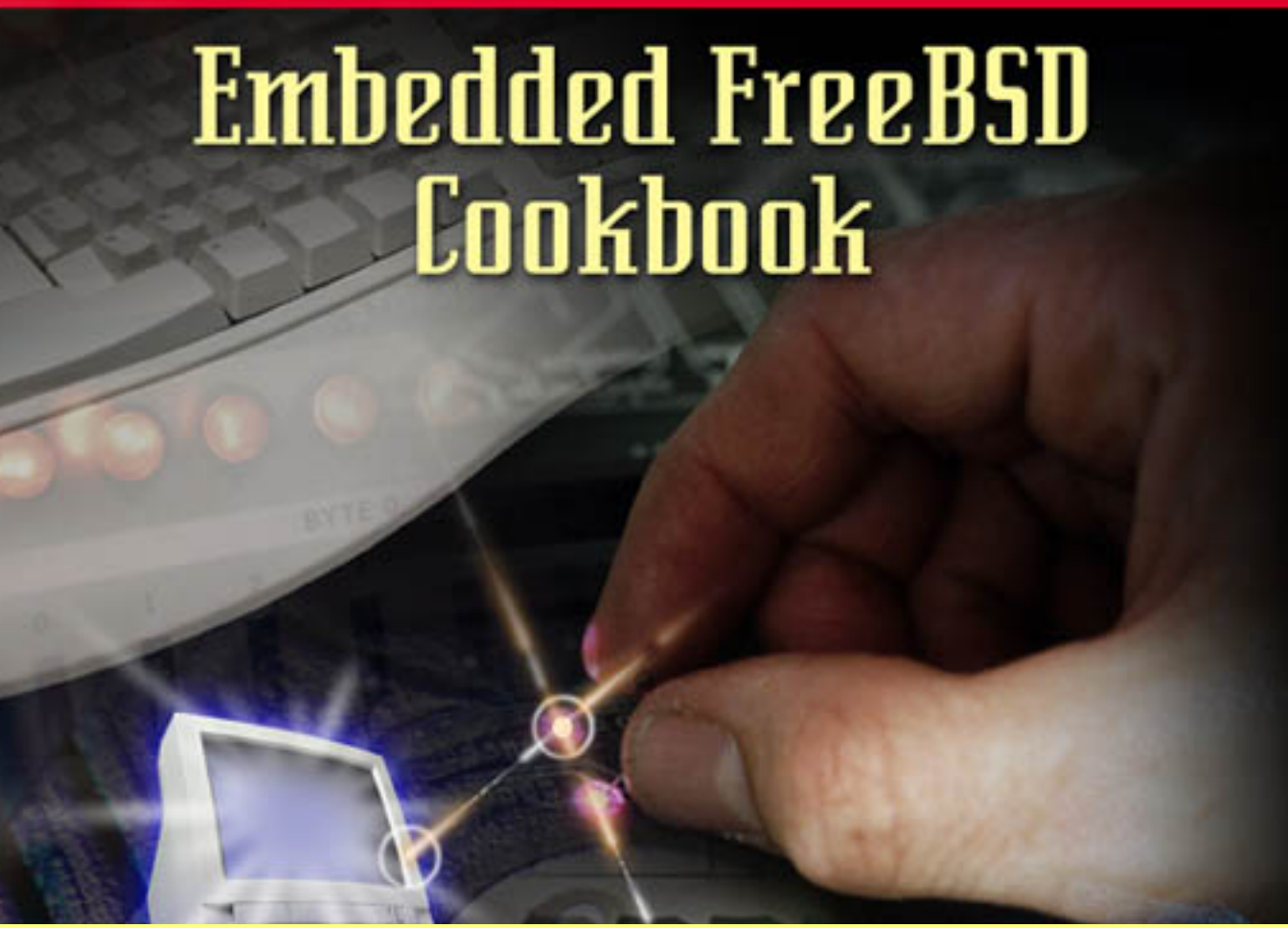
EMBEDDED TECHNOLOGY™
SERIES

Embedded FreeBSD Cookbook



Paul Cevoli

CD-ROM Included
Contains FreeBSD design tools!

Embedded FreeBSD Cookbook

A Volume in the
Embedded Technology™ Series

Embedded FreeBSD Cookbook

by Paul Cevoli

Amsterdam Boston London New York Oxford Paris
San Diego San Francisco Singapore Sydney Tokyo

Newnes is an imprint of Elsevier Science.

Recognizing the importance of preserving what has been written, Elsevier Science prints its books on acid-free paper whenever possible.

Library of Congress Cataloging-in-Publication Data

ISBN: 1-5899-5004-6

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

The publisher offers special discounts on bulk orders for this book.
For information, please contact:

Manager of Special Sales
Elsevier Science
200 Wheeler Road
Burlington, MA 01803

For information on all Newnes publications available, contact our World Wide Web home page at http://www.newnespress.com

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Table of Contents

	Preface	vii
Chapter One	**Getting Started**	1
	Overview	1
	Embedded Systems	1
	Internet Appliances	2
	The DIO Server Appliance	4
	Summary	8
Chapter Two	**Systems Programming**	9
	Overview	9
	Process	9
	Daemons	21
	Summary	25
Chapter Three	**System Calls**	27
	Overview	27
	Library Functions and System Calls	27
	Creating a System Call	32
	Summary	48
Chapter Four	**Device Drivers**	49
	Overview	49
	Driver Environment	49
	Driver Structure	51
	The DIO2 Device Driver	59
	Summary	76
Chapter Five	**Midlevel Interface Library**	77
	Overview	77
	Shared Libraries	77
	Accessing the Device Driver	79
	PCI-DIO24 Hardware Registers	82
	The DIO24 Application Interface Library	87
	Summary	101
Chapter Six	**Daemons**	103
	Overview	103
	Introduction to TCP/IP	103
	Socket System Calls	107
	The DIO Daemon	115
	Summary	122

Chapter Seven	**Remote Management**	123
	Overview	123
	Using Secure Shell (SSH)	123
	The DIOShell	130
	Summary	142
Chapter Eight	**JNI Layer**	143
	Overview	143
	The JDK	143
	Creating the JNI Layer	145
	Summary	156
Chapter Nine	**Web Access Using Tomcat**	157
	Overview	157
	Tomcat	157
	The JSP Overview	160
	The DIO JSP Page	162
	Summary	165
Chapter Ten	**Building the Kernel**	167
	Overview	167
	Building the DIO Kernel	177
	Building the FreeBSD Kernel	180
	Summary	182
Chapter Eleven	**System Startup**	183
	Overview	183
	Disk Geometry	183
	Master Boot Record	185
	PC BIOS	189
	FreeBSD Boot Loader	190
	Starting DIO Components	194
	Summary	196
Chapter Twelve	**The CompactFlash Boot Device**	197
	Overview	197
	Solid-State Devices	197
	Installing the TARC CompactFlash Adapter	198
	Configuring the CompactFlash Device	199
	Copying the Files to the Boot Device	201
	Startup Configuration	201
	Summary	205
Appendix A	**The FreeBSD License**	207
Appendix B	**PCI Configuration**	209
Appendix C	**Kernel Loadable Modules**	215
	Index	229

Preface

Discussing embedded systems in general is difficult, because each embedded system is unique. Rather than presenting a list of general principles for handling embedded development issues, this book presents examples of problems encountered and solutions to those problems using real hardware and software. In that sense, it is a "cookbook" for developers that offers design "recipes" that can be elaborated on or modified as needed to solve other design problems.

In addition to the source code provided to develop the DIO appliance, this book contains real "how-to" information for obtaining releases of Open Source software and describes the steps to install, configure and program. Whether you are developing an actual appliance or experimenting with the operating system, this book will help you familiarize yourself with all the development issues, not just development concepts.

This book presents a set of common issues that are encountered during the development of an embedded web appliance. Each chapter covers a specific topic, discusses background information on the topic, and presents a design solution for that topic. The chapters are not meant to present all possible solutions, but rather provide information that can be used to help develop your own solutions.

Developing an embedded system involves many skills beyond writing source code. Identifying technologies that can be used to solve a problem, installing, configuring, and packing are all skills that are equally important but are often overlooked. All of these issues are covered, to give you a complete picture of the embedded development process.

Prerequisites and Other Resources

This is a book about developing an internet appliance using FreeBSD and the software tools contained in the FreeBSD distribution. As such, it assumes the reader has some background in programming in C and C++ in a Unix environment. In addition to basic programming skills it would be helpful if you had a basic understanding of data structures and system programming. This book is for readers that already know how to write and compile code

and want to learn more about the FreeBSD environment and explore topics that are different from a typical programming book. A few of the topics do not require that you have a detailed background in computer hardware but assume you have some knowledge and are willing to learn the skills necessary.

I wrote this book for people who wanted to delve a little deeper into the FreeBSD operating system. Many of the issues presented in this book go beyond programming in C or C++ and present issues that an embedded systems engineer would be faced with during product development. This is not a book about FreeBSD internals and how to hack the kernel. FreeBSD has distinguished roots and there are already excellent resources available to learn about FreeBSD operating system internals and system administration. I recommend *The Design and Implementation of the 4.4 BSD Operating System*, *The Complete FreeBSD*, and *The FreeBSD Handbook*, located at http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/index.html. In addition to these resources, the FreeBSD web site contains numerous mailing lists, user groups, newsgroups and web resources. There are also excellent references for programming Unix, such as *Advanced Programming in the Unix Environment* and *The C Programming Language*, both available on amazon.com.

Organization

This book contains 12 chapters and 3 appendices.

Chapter 1, *Getting Started*, introduces you to embedded systems and describes Internet appliances and FreeBSD. Additionally, the major pieces of hardware and software used to build the Internet appliance described in this book are covered.

Chapter 2, *Systems Programming*, introduces you to the Unix process and daemons. A process is fundamental to Unix programming. A Unix daemon is a special type of process. The details of creating a daemon from a Unix process are discussed.

Chapter 3, *System Calls*, describes exactly what a system call is and how it is implemented in FreeBSD.

Chapter 4, *Device Drivers*, provides a description of FreeBSD device drivers and their data structures. Tools available to driver writers are discussed and used to develop an actual device driver for a PCI data acquisition controller.

Chapter 5, *Midlevel Interface Library*, presents a discussion of shared libraries and how user code accesses a device driver.

Chapter 6, *Daemons*, builds on the topics presented in Chapter 2 by implementing a Unix daemon that uses sockets to provide a simple protocol to read and write to the data acquisition board.

Chapter 7, *Remote Management*, discusses how to provide a secure method for remote management. A configuration shell is developed and may be accessed remotely via SSH.

Chapter 8, *JNI Layer*, introduces the user to JNI, an interface to allow Java programmers to call C code.

Chapter 9, *Web Access using Tomcat*, provides the steps necessary to display dynamic web content. The procedure for configuring Tomcat, a JSP server, and writing JSP pages is discussed. You will develop a JSP page that displays the status of the data acquisition controller.

Chapter 10, *Building the Kernel*, discusses the steps for building a custom kernel based on the hardware and specific features of the DIO Internet appliance.

Chapter 11, *System Startup*, provides a discussion of the FreeBSD booting process and modifications necessary to the startup scripts to run the services required by the DIO appliance such as Tomcat, ssh and loading custom KLDs.

Chapter 12, *The CompactFlash Boot Device*, provides a description of partitioning a flash device and loading the code developed in the previous chapters of this book onto the flash to make a living, breathing appliance server.

Appendix A, *The FreeBSD License*, is a copy of the FreeBSD license.

Appendix B, *PCI Configuration*, discusses the PCI configuration space. Knowledge of PCI configuration registers is needed for developing device drivers for PCI controllers. The appendix describes the registers in PCI configure space and their uses.

Appendix C, *Kernel Loadable Modules*, covers KLDs, which are used for system calls and device drivers in Chapters 2 and 4. The appendix provides

a discussion of the individual components that make KLDs and their uses.

In order to get the most out of this book, I recommend not getting bogged down in the details but trying to gain a general understanding of the topic through studying one way to implement a solution to that topic. By becoming familiar with each step involved in developing an embedded system, you should be able to apply the concepts to any development task you may encounter.

What's on the CD-ROM?

The accompanying CD-ROM contains the source code for the programs used in the book and a fully searchable pdf version of the entire text.

Conventions

`Macro Text`

Macro text is used for source code.

`Bold Macro Text`

Bold Macro Text is used for program output and shell input.

CHAPTER ONE 1

Getting Started

Overview

Embedded computer systems permeate all aspects of our daily lives. Alarm clocks, coffee makers, digital watches, cell phones, and automobiles are just a few of the devices that make use of embedded systems. The design and development of such systems is unique, because the design constraints are different for each system. Essential to the development of an embedded system is an understanding of the hardware and software used for development.

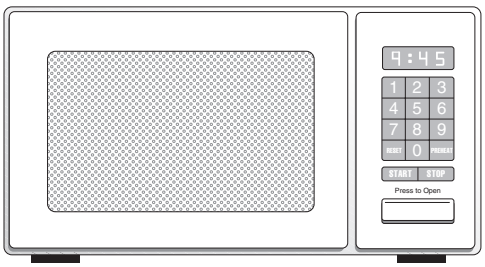
Embedded Systems

An embedded system consists of hardware and software designed to solve a specific application. Many applications consist of more than simple computer hardware and software. For example, an industrial vision system designed to control a robotic arm consists of an embedded computer, camera, display, and the robotic arm. Each of these components are embedded systems on their own.

Embedded systems have evolved over the years from simple self-contained, single-purpose systems to fully integrated, web-aware systems. The rapid changes in technology and added requirements have caused developers to take a new approach to the design and development of those systems. Let's take a quick look at how the product markets and requirements have evolved.

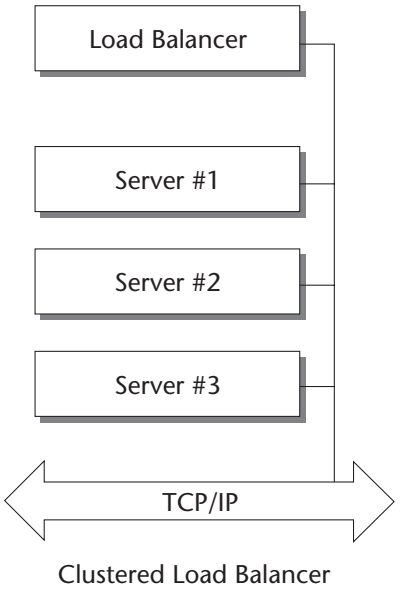
Classic embedded systems have been considered dedicated solutions to a single application. In these classic systems, the hardware was custom designed to solve a specific application and the operating system was

developed internally. All the software was self-contained in nonvolatile RAM and there was a limited user interface. Examples of these types of embedded systems are microwave ovens, MP3 players and cell phones.



Microwave Oven

Figure 1-1. Classic embedded system



Clustered Load Balancer

Figure 1-2. Load balancer

As the Internet grew, the requirements of embedded systems also began to grow in complexity. In addition to solving classic embedded systems problems, system designers were required to add connectivity for sending and receiving data or providing an automated method for software upgrades. Rather than increase the development effort, system designers have moved toward using third-party hardware and evaluating open source software. Examples of these next-generation embedded systems are load balancers, Virtual Private Networks (VPN) and Ethernet switches.

Internet Appliances

Many of the embedded systems developed today are what would be called servers just a few years ago. These systems are developed using PC hardware, run an embedded application, have Internet connectivity, and run network services for remote configuration. These latest incarnations of embedded systems are called Internet appliances.

Network connectivity requires the addition of network protocols, services, and networking hardware. Web services require the addition of more complex application software. These additional requirements significantly increase the hardware cost, development, and complexity of embedded systems development.

The added performance requirements add scalability to the list of system requirements, further limiting the choice of solutions of computer design. Additionally the investment of developing an embedded solution must be protected; as the hardware evolves, the final choice of hardware and software must be compatible from release to release. With the rapid development schedules of the high-tech sector, hardware and software solutions must be flexible enough to meet the needs of evolving markets and customers.

To address the increased complexity of embedded systems, a new category has become popular—the appliance server. An appliance server is a network-enabled embedded computer designed to perform a single task and provide superior performance and higher reliability than a general-purpose server. Applications ideal for appliance servers include VPN, network attached storage (NAS), and load balancing.

Development Issues

While the cost of computer components and time to market continue to decrease, functionality and features for a typical embedded system continue to increase. In order to respond to this, many project developers choose to use third-party hardware and open-source software and develop only those components that provide value-added features. The classic model of embedded systems development using custom hardware and homegrown operating systems has been replaced in many of the systems being developed today. Let's take a look at a few of the development issues related to embedded systems relying on third parties for some of the core components.

Compatibility of upgrades

By choosing third-party hardware and open-source software, a system designer ensures access to compatible technology from multiple vendors and allows the completed system to be developed quicker and for a lower cost. Systems designers can focus resources toward developing their application, rather than keeping resources focused on maintaining the operating system or developing the next hardware platform.

Time to market

An open-source operating system reduces the time to qualify and develop the software solution. The server appliances address increased time pressure to market and hardware flexibility by using an increasingly rich set of third-party hardware and software. By relying on readily available hardware and software, the server appliance developer can focus on solving the application.

Labor pool

The increased complexity of Internet appliances requires the transition from simple embedded tools to an off-the-shelf integrated development environment with tools that facilitate the development of secure, reliable, and long-running systems. Additionally, an off-the-shelf integrated development environment reduces the time to market, as well as training time.

Licensing

Traditional real-time operating system software licenses restrict the source to the licensee and may require a royalty fee for distribution. In contrast, open-source software licenses allow the licensee to make the software freely available to anyone who wishes to use it.

The DIO Server Appliance

The remainder of this chapter discusses an Internet appliance that will be developed in this book, the digital input-output (DIO) server appliance. The DIO server appliance is an embedded system that provides the capability to read and write digital IO lines. The digital lines are monitored via the Internet using sockets, a secure command shell, or a browser.

The software uses an open-source operating system and tools, FreeBSD 4.4 and the GNU development suite. The system software and application

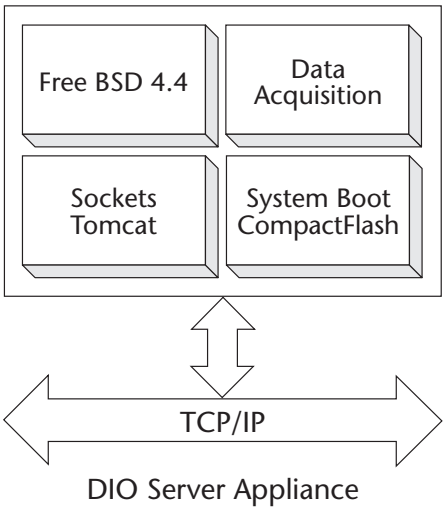


Figure 1-3. DIO server appliance

are developed in the remaining chapters of this book, presenting the design, development, and implementation of an actual server appliance.

The hardware to develop the DIO server appliance is a standard PC and a third-party digital IO data acquisition card. The boot device is a CompactFlash adapter and CompactFlash card, which can be purchased over the Internet.

Software

FreeBSD

FreeBSD is an optimal solution for Internet appliances and other embedded systems that require Internet connectivity, networking performance, and reliability. Additionally, FreeBSD comes with an industry standard set of software development and configuration management tools and application software. It is a Unix-compatible, open-source operating system that offers unprecedented reliability and security. It runs some of the Internet's busiest web sites, such as Yahoo and Hotmail, and supplies the basis of embedded products like the AMI Stortrends NAS and the IBM Whistle/InterJet II.

The core of FreeBSD is largely based on BSD/OS, which was developed in the mid-1970s and is known for excellent support, stability, small footprint, and simple installation. Much of the early BSD Unix development was funded by DARPA to support the development of Internet protocol, TCP/IP.

FreeBSD, as implied by the name, is available free of charge. It can be downloaded directly from the FreeBSD website at ftp://ftp.FreeBSD.org/pub/FreeBSD/. Or, if you're like me and want to have an actual CD, as well as support the FreeBSD effort, there are a number of retail outlets where FreeBSD can be purchased, such as the BSD Mall, http://www.bsdmall.com/freebsd1.html, or a number of local retail outlets such as Staples, CompUSA and Frys.

Besides providing financial support by purchasing an official release, there is yet another way to support FreeBSD. The FreeBSD distribution is an open source project. If you're interested, you can contribute time and source code development. Information about contributing to FreeBSD development can be found on the FreeBSD website, http://www.freebsd.org/doc/en_US.ISO8859-1/articles/contributing/index.html.

In addition to its mature networking technology, the FreeBSD kernel contains support for many disk and storage management facilities and secure

networking protocols. The features and benefits of FreeBSD are summed up in the following paragraphs.

- **Security** FreeBSD offers security features that make it suitable for e-commerce applications, secure Internet transmission, and virtual private networks. Many fixes to security-related bugs have been incorporated into FreeBSD over the years, to ensure that it is suitable for use in security-critical environments.
- **Robustness** FreeBSD is based on software that has been in development for more than 20 years. Its continued development is focused on quality rather than quantity, and changes to the core software are carefully controlled. The core FreeBSD kernel and its features represent the highest quality embedded operating system on the market.
- **Small Footprint** FreeBSD is fully customizable and may be configured to run with an absolute bare minimum of software, lending itself to some of the most limited embedded applications. PicoBSD, a targeted version of FreeBSD, contains fully bootable systems that fit on a floppy disk. In addition to the flexibility, many embedded systems boot from a DiskOn-Chip or CompactFlash device. Many of these devices are readily supported by FreeBSD.
- **License** FreeBSD is distributed using the BSD License, which permits, but does not require, the sharing of the source code. Because of the BSD license, and the fact that many embedded systems require the inclusion of proprietary technology for application-specific hardware included with these systems, or intrinsic to the design itself, such as on-board custom components, BSD systems tend to be a superior choice relative to other Open Source systems, where such intellectual property cannot be kept private due to licensing issues.

GNU Development Tools

The standard FreeBSD distribution contains the GNU suite of development tools, consisting of compilers, linkers, librarians, debuggers, performance management, and configuration management tools. In addition to their depth, the GNU tools provide updates and numerous support options, in the form of mailing lists and news groups.

Java

Using Java, system designers can develop applications that can be run on a browser anywhere. In addition to the ease of developing the network portion of the appliance, using the Java Native Interface (JNI), system developers can bridge legacy systems to the added requirements of network-centric embedded systems.

Hardware

The DIO server appliance developed in this book uses only third-party hardware. The hardware requirements are a network-capable PC-based hardware platform, a PCI-based digital IO controller, and a solid-state boot device and ATAPI interface card.

Server

The Network Engines Roadster is a high-performance 1U Internet appliance that provides an easily customized solution for any application. The Network Engines Roadster includes an Intel Celeron processor, 32 MB of RAM, dual Ethernet ports, one PCI slot and two serial ports.

Data Acquisition

The DIO Internet appliance hardware we will be using, the Measurement Computing PCI-DIO24 Digital IO Controller, can read and write digital IO signals. It is a PCI controller that provides 24 bits of digital IO. In addition to the PCI-DIO24 controller, the DIO server appliance uses the C37FF-2 Cable and CIO-MINI 37 Terminal for connecting the digital signals from the controller.

Boot Device

Embedded systems tend to be deployed in more rigorous environments than a typical desktop computer. A hard drive may become damaged or wear out in this type of environment. A trend in embedded systems development is to use CompactFlash. A CompactFlash device appears similar to a standard IDE drive. In order to use a CompactFlash device as a boot device, a CompactFlash adapter is required. There are numerous CompactFlash-to-IDE adapters available.

TAPR CompactFlash Adapter II

The Tucson Amateur Packet Radio (TAPR) Club sells an IDE CompactFlash Adapter that can be used with generally available CompactFlash cards. The CompactFlash Adapter plugs into an IDE slot and a power connecter. Once the CompactFlash adapter is connected and a CompactFlash device inserted, the device can be used just like an IDE hard drive.

Sandisk 32 MB CompactFlash Disk

The Sandisk CompactFlash is a small flash memory device that serves as our embedded boot device. In conjunction with the TAPR CompactFlash Adapter, we are able to use the Sandisk device as an IDE boot device. Sandisk CompactFlash devices are available in any office supply store; these are the same devices used in digital cameras and portable MP3 players.

Summary

In this chapter we've discussed the use of third-party hardware and open-source software for developing an Internet appliance. The remainder of the book is focused on the development of the DIO Internet appliance using FreeBSD. Each chapter discusses a topic related to embedded system development and provides a solution. By the end of the book, we will have developed a working appliance.

CHAPTER TWO 9

Systems Programming

Overview

Fundamental to any programming task in FreeBSD is the *process*, which is an executing program. It could be a network file system (NFS) daemon serving files, a gcc compile or a shell displaying the date—all these tasks are performed in the context of a process. An understanding of processes is critical for grasping concepts presented in later chapters. This chapter introduces systems programming using processes.

Many system services are provided by a special type of process known as a daemon process. As part of our discussion of processes, we will look at the characteristics of a daemon process and develop skeleton source code for a daemon. In this chapter we will cover topics including

- A FreeBSD process
- Process creation and termination
- Process attributes
- Daemon processes

Process

A running program is an instance of a process. One of the numerous features provided by a FreeBSD kernel is a protected environment called User Mode. In User Mode, a process cannot access hardware or protected system variables. If a process attempts to access protected memory, the process is terminated. A process running in memory consists of five segments: text, initialized data, uninitialized data, stack and heap.

Text is typically read only and contains the machine instructions for the program. Initialized data contains variables that are preinitialized by the program text. Uninitialized data (traditionally called bss) contains data that is not initialized. The stack is used to pass parameters between functions and to contain local variables and function return addresses. The heap is an operating-system-provided area used for dynamic memory allocation.

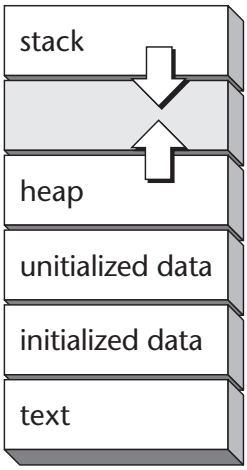


Figure 2-1. Process Running in Memory

We can look at the size of a program's sections, invoking the size command. Output of the size command for the `hostname` program is in Listing 2-1.

```
# size hostname
  text      data       bss       dec       hex     filename
 39452      4020      1968     45440      b180     hostname
```

Listing 2-1

The output shows the text size as 39452 bytes, the data section as 4020 bytes and bss size of 1968 bytes. Stack and heap are assigned by the operating system for each process. The text and data sections contain data. Since the bss section is uninitialized, the image only contains its size; bss memory is set to zeros by the operating system when the program is loaded.

Process Creation

When a process is created a complete copy of the original process, known as the parent, is created. The newly created process is called the child. Once process creation is completed, the child process is scheduled to execute. This yields two running instances of the same program. The processes can only be differentiated by the process identifier.

The `fork` System Call

A new process is created when a running process invokes the `fork` system call. The `fork` system call is the only way to create a new process.

```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
```

Fork is unique in that the single call to fork returns twice. Fork returns the process identifier (PID) of the child to the parent process, and it returns 0 to the newly created child process.

When a process is created, the following list of attributes is inherited from the parent file descriptors: process group ID, access groups, working directory, root directory, control terminal, resources, interval timers, resource limits, file mode mask, and signal mask.

The execve System Call

Many times a process is created to run another program. The execve system call is used this way and is invoked immediately following a fork system. An execve system call will replace the currently executing program with a new program.

```
#include <unistd.h>
int execve(const char *path, char *const argv[], char *const envp[]);
```

The execve system call takes three parameters; path is the path and filename of the new program to execute; argv contains a pointer to the argument string to pass to the program; envp contains a list of environment variables.

The execve system call returns –1 on error. The execve system call has numerous wrappers in the Standard C Library, known as the exec family of functions.

Process Termination

A process can terminate intentionally by calling exit or unintentionally by receiving a signal from another process. Regardless of the reason, when a process terminates, a notification is returned to the parent. If the parent process does not receive the notification, the child process becomes a zombie. The child will remain a zombie until the parent retrieves the child's exit status.

The `_exit` System Call

A process will terminate when the program invokes the `_exit` system call. The `_exit` system call will cause the `SIGCHLD` signal to be thrown to the parent process.

```
#include <unistd.h>

void _exit(int status);
```

The `_exit` system call will never return. The `_exit` system call is more commonly called by the Standard C library function `exit`.

The `wait` System Call

The `wait` system call allows a parent process to check to see if termination information is available for a child process. `Wait` will suspend execution until the child process returns.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

On success, the child process identifier (PID) is returned to the parent process, or –1 is returned if there is an error. The child's exit status is returned by the status parameter.

An Example

Listing 2-1 illustrates the usage of the `fork`, `wait`, `execve`, and `exit` calls. The parent process makes a `fork` system call to create a child process. There are two control paths in the main program, one for the parent and another for the child.

```
int main(int argc, char **argv)
{
    pid_t   pid;
    int   status;

    if ((pid = fork()) > 0)
    {
      printf("%d: waiting for the child\n", pid);

      wait(&status);
```

```
      printf("%d: child status = %d\n", pid, status);

   }
   else if (pid == 0)
   {
      execve("/bin/date", NULL, NULL);
   }

   exit(0);
}
```

Listing 2-1

The `fork` system calls returns to both the parent and the child processes. The parent process calls the `wait` system call, causing the parent process to sleep until the child process exits. The child process calls `execve` to run the date program and display the date to the console.

The output of the program in Listing 2-1 from my system looks like this:

```
# ./process
542: waiting for the child
Mon Jan  7 19:54:46 EST 2002
542: child status = 0
```

Process IDs

Every process is created with a unique ID called its process identifier, or PID. In addition to its own PID, every process contains its parent process ID, or PPID. A listing of processes, their PIDs, and PPIDs may be obtained by using the `ps -aj` command. Here is a partial listing from my system.

```
USER  PID  PPID  PGID    SESS  JOBC  STAT  TT     TIME COMMAND
root  453   451   453  fb0700   0   Is+   p0   0:00.01 /bin/cat
root  456   454   456  fb6d00   0   Is+   p1   0:00.04 /bin/csh
root  458   456   458  fb6d00   1   S     p1   0:08.21 emacs
root  459   458   459  fc7dc0   0   Ss    p2   0:00.06 /bin/csh -i
```

The **getpid** and **getppid** System Calls

A process can retrieve its PID and PPID by calling the `getpid` and `getppid` functions, respectively.

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

Every process has a parent process. When a process is created, the parent process ID is assigned so it can return the termination status.

Security

A process is assigned two users on creation, the real user and the effective user.

User ID and Group ID

Every process has a user identifier, UID, and a group identifier, GID. The UID and GID of a process are the username and group of the user that invoked the program. UIDs are mapped to user names in `/etc/passwd`. GIDs are mapped to group names in the `/etc/group` file. A process can retrieve its user ID and group ID by calling the `getuid` and `getgid` system calls.

The **getuid** and **getgid** System Calls

The `getuid` and `getpid` system calls return the UID and GID for the running process. Both system calls always succeed.

```
#include <unistd.h>
#include <sys/types.h>

uid_t getuid(void);
uid_t getgid(void);
```

Effective User ID and Effective Group ID

When a process is created, it is assigned an effective user ID and an effective group ID. Under most circumstances, the effective ID and the real ID are the same. It is possible for a program to be configured so that it executes as a different user or group. For example, the `passwd` utility needs root user access so it has permission to edit the `/etc/passwd` file. The `passwd` utility is configured so it executes with its effective user ID as root and gives `passwd` the correct permission to edit the `/etc/passwd` file.

The **geteuid** and **getegid** System Calls

The `geteuid` and `getegid` system calls return the effective group and effective user, respectively.

```
#include <unistd.h>
#include <sys/types.h>
```

```
uid_t geteuid(void);
uid_t getegid(void);
```

The `geteuid` and `getegid` system calls always succeed.

The `seteuid` and `setegid` System Calls

A process may be able to change the effective user ID or effective group ID by invoking the `seteuid` and `setegid` system calls. The `seteuid` and `setegid` system calls can set the effective user ID and effective group ID.

```
#include <unistd.h>
#include <sys/types.h>

uid_t seteuid(void);
uid_t setguid(void);
```

The `seteuid` and `setegid` system call return 0 on success and –1 on error.

An Example

Using the functions described for user and group IDs, Listing 2-2 prints the process real and effective user and group IDs.

```
int main(int argc, char **argv)
{
    printf("UID  = %d\n", getuid());
    printf("GID  = %d\n", getgid());
    printf("EUID = %d\n", geteuid());
    printf("EGID = %d\n", getegid());

    exit(0);
}
```

Listing 2-2

The output of the program on my system displays as follows:

```
# ./ids
UID  = 1001
GID  = 1001
EUID = 1001
EGID = 1001
```

A quick look at the `/etc/passwd` and `/etc/group` files shows that this program is executing as user paul and group paul, which is my logon name and group.

Here is my logon name in the `/etc/password` file; column three contains my UID.

paul:*:1001:1001:Paul Cevoli:/home/paul:/bin/sh

Here is my login group entry from the `/etc/group` file; column three contains my GID.

paul:*:1001:

Process Groups

A process group consists of a group of related processes. In addition to the PID and PPID, a process contains the process group ID, the PGID. Each process group contains a unique process group identifier. Every process group is able to have a single process group leader; a process group leader is denoted by the process group ID being the same as the PID.

The `setpgid` and `getpgid` System Calls

A process is able to set and get its PGID through the use of the `setpgid` and `getpgid` system calls.

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgrp);
pid_t getpgid(pid_t pid);
```

A process is only able to set the process group of itself and its children processes. *One use of process groups is to send signals to a group of processes.*

Files

A child process inherits the environment of the parent process that created it. This includes files descriptors, file permissions and the current working directory.

File Descriptors

A file descriptor is a low-level interface used to access an IO interface. File descriptors are implemented as an int. Every process contains a list of open file descriptors. A child process inherits any open file descriptors from the parent.

Permissions

Every process contains a default set of file permissions called file creation mode mask. The file creation mode mask is 9 bits that represent read, write and execute permissions for the file owner, file group, and everybody else. A processes file creation mode mask is modified by the `umask` system call.

The **`umask`** System Call

The file permission mask is typically set by the `umask` command in the user's login shell. The file permission `amsk` can be modified by the `umask` system call. The `umask` system call sets the process file creation mask to the value contained in `umask`.

```
#include <sys/stat.h>
mode_t umask(mode_t numask);
```

The previous value of the file creation mask is returned to the caller.

Current Working Directory

Every process contains a current working directory, which is the directory where the process was started.

The **`chdir`** System Call

A process may change its current working directory by invoking the `chdir` system call. The path argument contains the path to be used as the current working directory.

```
#include <unistd.h>
int chdir(const char *path);
```

The `chdir` system call returns 0 on success and –1 on error.

Resources

A process inherits the resource limits from its parent process. Resources may be read or modified by the `setrlimit` and `getrlimit` system calls.

The `setrlimit` and `getrlimit` System Calls

The `setrlimit` and `getrlimit` system calls read and modify process resource limits. `Setrlimit` and `getrlimit` take a resource parameter that specifies the resource to be read or written; a list of resources is contained in `/usr/include/sus/resource.h`. The second argument is a `struct rlimit` pointer:

```
struct rlimit {
    rlim_t rlim_cur; /* current (soft) limit */
    rlim_t rlim_max; /* maximum value for rlim_cur */
};
```

which is used for the resource value.

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlp);
int setrlimit(int resource, const struct rlimit *rlp);
```

The `setrlimit` and `getrlimit` system calls return 0 on success and –1 on error.

Sessions

A set of process groups can be collected into a session, a set of processes that are associated with a controlling terminal. Sessions are used to group a user login shell and the process it creates or to create an isolated environment for a daemon process.

The `setsid` System Call

```
#include <unistd.h>

pid_t setsid(void);
```

The **`setsid`**`()` function creates a new session. The calling process is the session leader of the new session, is the process group leader of a new process group, and has no controlling terminal. The calling process is the only process in either the session or the process group.

Controlling Terminal

A session may have a controlling terminal; this is the device used to logon. The session leader that initiated the connection of the controlling terminal is considered the controlling process. The controlling terminal is established for us when we login. There are times when a program wants to talk to the controlling terminal.

Scheduling

Priority

Execution time is made available to a process according to its process priority. Priorities range from 0 through 127. Process priorities are defined in `/usr/include/sys/param.h`. The lower the process priority, the more favorable scheduling priority it receives.

The `getpriority` and `setpriority` System Calls

A process priority may be read and modified using the `setpriority` and `getpriority` system calls.

```
#include <sys/time.h>
#include <sys/resource.h>

int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
```

The `getpriority` and `setpriority` system calls work differently based on the `which` parameter.

which	Definition
PRIO_PROCESS	process identifier
PRIO_PGRP	process group identifier
PRIO_USER	user ID

The `setpriority` and `getpriority` system calls return 0 on success and –1 on error.

State

A process is in one of five states at any time. The process state is used internally by the FreeBSD kernel to organize processes. Process states and their definitions are described below.

	Description
SIDL	Initial state of a process on creation while waiting for resources to be allocated.
SRUN	The process is ready to run.
SSLEEP	The process is suspended waiting for an event.
SSTOP	The process is being debugged or suspended.
SZOMB	The process has exited and is waiting to notify its parent.

Signals

Signals are mechanisms to notify a process that a system event has occurred. Every process contains a table that defines an associated action to handle a signal that is delivered to a process; at process creation, all signals contain a default action. A signal is handled in one of three ways: it is ignored, caught, or handled by the default action set by the kernel.

A user can define a function that is invoked when a process receives a signal; this is called a signal handler. The signal handler is said to catch the signal. Signals are defined in `/usr/include/sys/signal.h`. It is important to note that two signals, SIGSTOP and SIGKILL, cannot be caught and will terminate a process.

The `signal` Function

The `signal` function is used to specify a user-defined signal handler for a specific signal. The `signal` function is a wrapper for the `sigaction` system call.

```
#include <signal.h>

void (*sig_t) (int)
sig_t signal(int sig, sig_t func);
```

The `sig` parameter specifies which signal the signal handler will catch. The `func` parameter allows a user to specify a user-defined signal handler. The default action of the signal may be reset, by specifying SIG_DFL as the signal handler. A signal may be ignored, by specifying SIG_IGN as the signal handler. Ignoring a signal causes subsequent instances of the signal to be ignored and pending instances to be discarded.

If successful, `signal` returns the previous action; otherwise, SIG_ERR is returned and the global variable `errno` is set to indicate the error.

The `kill` System Call

The **`kill`** system call sends a signal to a process or group process group.

```
#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
```

The `pid` parameter is the process ID to send the signal. A process ID greater than zero is sent to that specific process. A process ID of zero sends the signal to all the members of the process group of the sending process. A process ID of –1 sends the signal to all processes if the super user; otherwise, it is sent to all processes with the same user ID as the sending process. `Sig` is the signal sent.

The `kill()` function returns the value 0 if successful; otherwise the value –1 is returned and the global variable `errno` is set to indicate the error.

Daemons

A daemon is a special process that runs in the background, usually providing a service. Daemon processes are usually started at system boot time and remain running until the system is halted. All daemons have some common process attributes. In the following section, we will cover issues common to daemon processes.

- **Fork**

  The first thing a daemon process does is to create a child process via the fork system. After the child process is created, both the parent and the

child run the same program simultaneously. A daemon process will disassociate itself from the parent process and put itself in the background.

- **Create a new session**

  The next step is to create a new session. Creating a new session provides us with the following benefits: the process becomes a session leader of a new session, the process becomes the group leader of a new process group, and the process has no controlling terminal.

- **Close file descriptors**

  Recall that a child process inherits its environment from the parent process. If the parent process has any open file descriptors, the child will have a copy of the open file descriptors. It's a good idea to assume any file descriptor may be open and close them all.

- **Change the current working directory**

  In addition to open file descriptors, the child inherits the current working directory. If the current working directory is a mounted file system, that file system would not be able to be unmounted.

- **Set the file mode creation mask**

  The file mode creation mask is inherited from the parent and may be set to deny certain permissions.

- **Handle Child Exit Status**

  If a daemon forks child processes to handle requests it must handle the child SIGCLD signal or the child process will become a zombie.

A Skeleton Daemon Function

The `handle_sigcld` Function

Listing 2-2 demonstrates the code necessary to handle the exit process of a child process. Recall if a parent doesn't retrieve the exit code of a child process, the child process becomes a zombie process. A parent process can avoid creating zombie processes by installing the `handle_sigcld` signal handler to catch the SIGCLD signal when a child exits.

```
void    handle_sigcld()
{
    int pid;
    int status;

    while ((pid = wait3(&status, WNOHANG, (struct rusage *)NULL))
> 0)
        ;
}
```

Listing 2-2

Note that the `handle_sigcld` function invokes the wait3 system call as NOHANG. This is so the parent continues execution and does not block waiting for a child to exit. This allows the parent to continue responding to requests or to perform other tasks.

The `init_daemon` Function

The `init_daemon` function in Listing 2-3 handles the details of making the child process a daemon process as described in this section.

```
void
init_daemon()
{
    int     childpid  = 0;
    int     fd         = 0;
    struct rlimit    max_files;

    /*
    **    create a child process
    */
    if ((childpid = fork()) < 0)
    {
        /* handle the error condition */
        exit(-1);
    }
    else if (childpid > 0)
    {
        /* this is the parent, we may successfully exit */
        exit(0);
    }
```

```
    /* now executing as the child process */

    /* become the session leader */
    setsid();

    /*
    **     close all open file descriptors, need to get the maximum
    **     number of open files from getrlimit.
    */
    bzero(&max_files, sizeof(struct rlimit));
    getrlimit(RLIMIT_NOFILE, &max_files);
    for (fd = 0; fd < max_files.rlim_max; fd++)
    {
       close(fd);
    }

    /*
    **     the current working directory is held open by the
    **     kernel for the life of a process so the file system
    **     cannot be unmounted, reset the current working
    **     directory to root.
    */
    chdir("/");

    /*
    **     a process inherits file access permissions from the
    **     process which created it, reset the user access mask
    */
    umask(0);

    /*
    **     we don't care about the exit status but we need to
    **     cleanup after each process that handles a request so
    **     the system isn't flooded with zombie processes
    */
    signal(SIGCHLD, handle_sigcld);
}
```

Listing 2-3

Summary

In this chapter we've covered process attributes in FreeBSD, as well as the common systems calls used to create, terminate, and manage a process. Our discussion of processes was completed by introducing a special type of process known as the daemon process. The details of creating a daemon process have been presented and encapsulated into a general purpose function, `init_daemon`.

We'll revisit the discussion of daemon processes again in Chapter 6, when we implement a daemon to handle DIO queries using sockets. Next, let's jump into the implementation of FreeBSD system calls.

System Calls

Overview

As mentioned in the previous chapter, the FreeBSD kernel provides an execution environment for a process called User Mode. A process executing in User Mode cannot directly access kernel memory, kernel data structures or hardware. Access to kernel memory and hardware resources is restricted to the FreeBSD kernel, which executes in a privileged mode of the underlying processor known as Kernel Mode. FreeBSD provides a detailed interface for a user process to obtain basic services from the kernel known as system calls.

The benefit of restricting certain tasks to the kernel is twofold. First, it relieves a system engineer from the details of programming device hardware. Second, by providing and enforcing a detailed interface to the system call interface, the kernel can ensure that the parameters passed by a user program are correct and will not cause a system crash. The added protection in the kernel provides a stable run-time environment for applications.

In this chapter we will discuss the details of FreeBSD system calls. After our discussion of system calls is complete, we will implement a system call that reads and writes kernel memory along with an application that provides simple debugging capabilities.

Library Functions and System Calls

The Standard C Library provides an API consisting of header files and libraries that contain implementations of many commonly needed functions and IO routines. The Standard C library is implemented as both library functions and system calls.

To a system designer, library functions and system calls appear to be the same. Both have defined function prototypes and return values. To an application programmer, the functional implementation is not important. The interface, function name, parameters, and return value are important. The difference resides in the implementation.

When a program calls a library function program, execution is passed to the library function, the requested operation is performed, and control returned to the calling function. All this occurs in the context and address space of the running process.

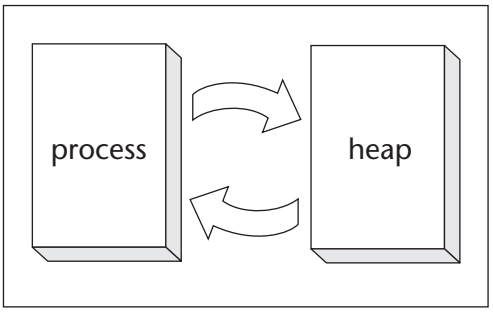


Figure 3-1. Library Function Call

A system call is a request to the kernel to execute a privileged operation on behalf of a user process. When a program makes a system call, program execution transitions from User Mode to Kernel Mode, the kernel performs the requested operation, then program execution transitions back from Kernel Mode to User Mode. In order for the operation to be performed, the process must make a request to the kernel.

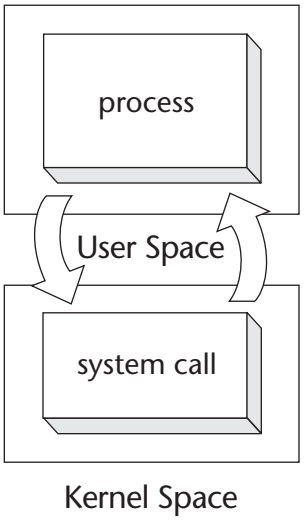


Figure 3-2. System Call

System Call Implementation

System calls are implemented in two steps. The first step is to transition from User Mode into Kernel Mode, which is accomplished by using a software interrupt. In addition, data must be passed from User Mode to Kernel. In this section we will look at these details.

Software Interrupt

On an Intel x86 processor, a program executing in User Mode performs a switch into Kernel Mode by executing a software interrupt. A software interrupt is an event that is processed asynchronously by the processor, using an interrupt handler. An interrupt handler is called in response to the invocation of a hardware or software interrupt. A software interrupt is different from a hardware interrupt, because it occurs synchronously within the running process.

Software interrupts are invoked on an Intel x86 processor by executing the INT instruction. The INT instruction takes a number between 0 and 255 that designates the interrupt vector. System calls are assigned to interrupt vector 128 (0x80). A process will invoke a system call by executing the INT $080 instruction.

Every interrupt vector contains an entry in the Interrupt Descriptor Table (IDT). The IDT is a table of 256 longwords. Each longword represents the address for the interrupt handler for the specific interrupt vector. For a system call, the system call interrupt handler is located at 0x200. 0x200 is derived by interrupt vector 0x80 * 4 = 0x200.

Once the processor encounters either a software or hardware interrupt, it fetches the address from the Interrupt Descriptor Table (IDT) for the specific interrupt vector and executes that interrupt vector. After the interrupt handler is completed, execution returns to the user process.

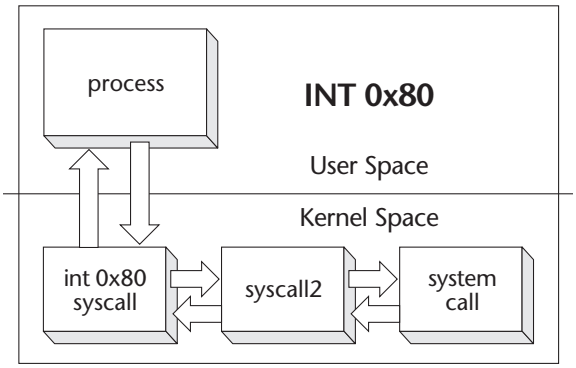


Figure 3-3. Interrupt Handling

Passing Data

The kernel implements many different system calls. To distinguish between different system calls, a number is passed that designates which system call is being made; this number is called the system call number. A list of defined system calls can be found in `/sys/kern/syscalls.master`. Before a system call is made, the system call number is pushed onto the program stack.

In addition to the system call number, many system calls take parameters. Parameters are passed to the kernel using the C calling convention. The caller pushes the system call parameters on the stack, one after another, in reverse order, right to left, so that the first argument specified to the function is pushed last. On return from the system call, the caller restores the stack to its original value before the call; this is called popping the stack.

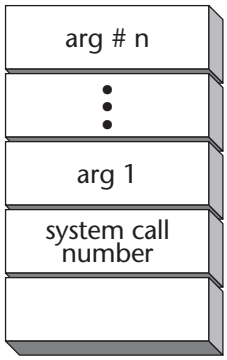


Figure 3-4. System Call Number

An Example

To demonstrate how the compiler generates system calls, let's look at the generated output of a program that makes a system call. Listing 3-1 contains a program that makes an open and close system call.

```
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char **argv)
{
 int fd;

 fd = open("file.dat", O_RDWR);
 close(fd);

}
```

Listing 3-1

> NOTE: In order to simplify the listed output, optimizations are turned off so the assembly code is easier to read, and the program is linked statically. The compile line used is:
>
> ```
> # gcc -00 -static -o open open.c
> ```

After the program in Listing 3-1 has been compiled and linked, we can use the `objdump` utility to disassemble the output to look at the compiler generated system calls.

```
# objdump –disassemble open
```

Listing 3-2 contains a partial listing of the open program disassembly. Of particular interest is the pushing of parameters and system call numbers onto the stack and the execution of the INT instruction.

```
080481c4 <main>:
 80481cd:   6a 02                  push   $0x2
 80481cf:   68 01 84 04 08         push   $0x8048401
 80481d4:   e8 7f 00 00 00         call   8048258 <_open>
 80481d9:   83 c4 10               add    $0x10,%esp

08048258 <_open>:
 8048258:   8d 05 05 00 00 00      lea    0x5,%eax
 804825e:   cd 80                  int    $0x80

 8048262:   c3                     ret
```

Listing 3-2

Listing 3-2 shows the steps performed by the compiler to generate the assembly code to make an open system call. The instructions located at `80481cd` and `80481cf` push the open mode and filename onto the stack. The main program then calls the `open` system call. `Open` then pushes its system call number, 5, onto the stack in line `8048258`, then invokes the software interrupt. After the return from `open`, the stack is restored to its original value in line `80481d9`.

truss

In the previous sections, we've taken a detailed look at how system calls are implemented. In this section, we'll show how a standard utility uses system calls to accomplish its task. `truss` is a FreeBSD command containing a utility used to trace system calls and see signals received by a named process. Using the `truss` command we'll look at how the `pwd` command makes system calls. The `pwd` command displays the current working directory of a process. Listing 3-3 contains the output of the `truss` command when used to analyze the `pwd` command.

```
readlink("/etc/malloc.conf",0xbfbff578,63)       ERR#2 'No such
                        file or directory'
mmap(0x0,4096,0x3,0x1002,-1,0x0)          = 671432704
(0x28054000)
break(0x8058000)                          = 0 (0x0)
break(0x8059000)                          = 0 (0x0)
sigaction(SIGSYS,0xbfbff660,0xbfbff648)   = 0 (0x0)
__getcwd(0x8058000,0x3fc)                 = 0 (0x0)
sigaction(SIGSYS,0xbfbff648,0x0)          = 0 (0x0)
fstat(1,0xbfbff388)                       = 0 (0x0)
```

```
ioctl(1,TIOCGETA,0xbfbff3bc)                    = 0 (0x0)
write(1,0x8058400,6)                            = 6 (0x6)
exit(0x0)           process exit, rval          = 0
```

Listing 3-3 truss output

The output generated by `truss` shows the system calls made by the `pwd` command. Of particular interest is are the system calls to `sigaction`. Recall from the previous chapter, `sigaction` is used to install a signal catcher. This code is protecting against a nonexistent system call.

Once the SIGSYS signal catcher is installed, the program calls `getcwd`, the internal system call to return the current working directory of the current process. After the return from `getcwd`, the signal catcher for SIGSYS is restored.

Before the program exits, the write system call is invoked to write the current working directory to the control terminal.

Creating a System Call

A system call can be developed as a kernel-loadable module (KLD). Appendix C presents an overview of kernel-loadable modules. In this section, we will discuss the pieces of KLD, along with our implementation of the `copymem` system call.

Adding a System Call

In the previous sections, we've covered the implementation and practical uses of system calls. In the following sections of this chapter, we are going to develop a system call, `copymem`, which reads or writes kernel memory. The `copymem` system call tasks four parameter; kernel address, user address, number of bytes, and direction for a copy. The prototype is:

```
int copymem(int kern_addr, int user_addr, int nbytes, int direction);
```

Once the implementation of the `copymem` system call is complete, we will finish our discussion of system calls by implementing a simple utility that accepts commands to read and write kernel memory using the `copymem` system call.

Load Handler

The first piece of code necessary for a KLD system call is the load handler. The load handler is responsible for handling any initialization or cleanup

when the module is being loaded, unloaded, or the system is shutting down. Every KLD is required to have a load handler. Listing 3-4 implements the load handler for the `copymem` system call.

```
static int32_t
load (struct module *module, int cmd, void *arg)
{
  int32_t err = 0;

  switch (cmd)
  {
  case MOD_LOAD:
    uprintf("copymem system call loaded succesfully\n");
    break;

  case MOD_UNLOAD:
    uprintf("copymem system call unloaded succesfully\n");
    break;

  default:
    err = EINVAL;
    break;
  }

  return(err);
}
```

Listing 3-4

The load handler accepts three arguments. The first is a linked list of modules currently loaded in the system. The second argument is the `cmd` parameter that represents the condition for load handler being called. The defined commands are:

Command	Action
MOD_LOAD	Passed when the module is loaded
MOD_UNLOAD	Passed when the module is unloaded
MOD_SHUTDOWN	Passed when the system is shutting down

Table 3-1 Load Module Commands

The final argument is a user-defined argument passed to the load handler. This is discussed further in the SYSCALL_MODULE section.

System Call Arguments

In the system call discussion, we showed how a user program passes the parameters to a system call on the stack. In the kernel implementation, the parameters are passed as a structure argument. Each element of the structure is one parameter passed from the user program.

Listing 3-5 defines the `copymem_args` structure. The `copymem` structure has four elements, one for each of the four parameters passed to the `copymem` system call.

```
struct copymem_args
{
   int32_t kernel_addr;  /* kernel address       */
   int32_t user_addr;    /* user provided buffer */
   int32_t len;          /* length of transfer   */
   int32_t direction;    /* to kernel 1, from 0  */
};
```

Listing 3-5

The `copymem` system call accepts four parameters: the kernel mode address, a user mode address, the length of the copy, and a number representing the direction of the copy. The two directions implemented by `copymem` are from kernel mode to user mode or from user mode to kernel mode.

The `copymem` System Call

Now that we have the load handler and arguments defined, we can implement the actual `copymem` system call. System calls take two arguments. The first is the `proc` structure, which contains the current state and settings for the calling process; the `proc` structure is defined in `/usr/include/sys/proc.h`.

The second argument is the argument structure; in our case, it will be the `copymem_args` structure defined in the previous section. Listing 3-6 contains the source code listing of `copymem`.

```
static int32_t
copymem(struct proc *p, struct copymem_args *uap)
{
   int stat;
```

```
  if (uap->direction == 0)
  {
    stat = copyin((void *)uap->user_addr, (void *)uap-
                        >kernel_addr, uap->len);
  }
  else if (uap->direction == 1)
  {
    stat = copyout((void *)uap->kernel_addr, (void *)uap-
                        >user_addr, uap->len);
  }

#if DEBUG
  if (stat != 0)
  {
    uprintf("copy failed, stat = %d\n", stat);
  }
#endif

  return(0);
}
```

Listing 3-6

The `copymem` function implements two paths of control based on the direction parameter. Memory is either copied from user mode to kernel mode or kernel mode to user mode. Because of the processor hardware details, addressing in user mode and kernel mode may be slightly different, so copying memory between user mode and kernel mode may be more involved than using the Standard C library `memcpy` function. The FreeBSD kernel provides function utilities to handle the details of copy memory between user mode and kernel mode.

The **`copyin`** and **`copyout`** Functions

The `copyin` and `copyout` functions are used to copy memory between user mode and kernel mode.

```
#include <sys/types.h>
#include <sys/systm.h>

int copyin(const void *uaddr, void *kaddr, size_t len);
int copyout(const void *kaddr, void *uaddr, size_t len);
```

The `copyin` function copies *len* bytes of data from the user mode address *uaddr* to the kernel mode address *kaddr*. The `copyout` function copies *len* bytes of data from the kernel mode address *kaddr* to the user mode address *uaddr*.

The `sysent` Structure

Every system call in the FreeBSD kernel has a `sysent` structure, defined in `/usr/include/sys/sysent.h`. The `sysent` structure takes two elements, the number of parameters, which is two as defined by the `dumpmem_args` structure, and the name of the system call function. Listing 3-7 defines the `copymem sysent`.

```
static struct sysent copymem_sysent =
{
   4,    /* number of parameters */
   copymem   /* system call      */
};
```

Listing 3-7

System calls are contained in the `sysent` structure, defined in `/sys/kern/init_sysent.c`. When a KLD system call is made, a new entry is added to the kernel global `sysent` structure. Listing 3-8 contains a partial listing of the `sysent` structure.

```
/* The casts are bogus but will do for now. */
struct sysent sysent[] = {
  { 0, (sy_call_t *)nosys },                    /* 0 = syscall */
  { AS(rexit_args), (sy_call_t *)exit },        /* 1 = exit */
  { 0, (sy_call_t *)fork },                     /* 2 = fork */
  { AS(read_args), (sy_call_t *)read },         /* 3 = read */
  { AS(write_args), (sy_call_t *)write },       /* 4 = write */
  { AS(open_args), (sy_call_t *)open },         /* 5 = open */
  { AS(close_args), (sy_call_t *)close },       /* 6 = close */
```

Listing 3-8

The System Call Number

A system call number must be declared; since there is no system call number defined, this value should be set to NO_SYSCALL. The kernel defines the system call number dynamically.

```
static int32_t syscall_num = NO_SYSCALL;
```

When the KLD system call is loaded into the kernel, `systent entry copymem_sysent` is assigned to the first open index in the kernel global sysent structure. The index into the sysent array is the system call number.

The specifics of installing a new system call are found in the `syscall_register` function listed in `/sys/kern/kern_syscalls.c`. Listing 3-9 contains the `syscall_register` function.

```
int
syscall_register(int *offset, struct sysent *new_sysent,
         struct sysent *old_sysent)
{
        if (*offset == NO_SYSCALL) {
                int i;

                for (i = 1; i < SYS_MAXSYSCALL; ++i)
                        if (sysent[i].sy_call == (sy_call_t
                                        *)lkmnosys)
                                break;
                if (i == SYS_MAXSYSCALL)
                        return ENFILE;
                *offset = i;
        } else if (*offset < 0 || *offset >= SYS_MAXSYSCALL)
                return EINVAL;
        else if (sysent[*offset].sy_call != (sy_call_t *)lkmnosys)
                return EEXIST;

        *old_sysent = sysent[*offset];
        sysent[*offset] = *new_sysent;
        return 0;
}
```

Listing 3-9

When a new system call is added, the kernel function `syscall_register` is called. The offset and `sysent` structure for the new call are passed. If the offset is NO_SYSCALL, `syscall_register` scans the `sysent` structure looking for an empty system call location. If one is found, the system call is inserted and the offset is set to the index of the `sysent` structure, where the call has been inserted.

The SYSCALL_MODULE Macro

The final task for creating a system call module is to declare the module. The macro used to define a system call is the SYSCALL_MODULE defined in `/usr/include/sys/sysent.h`. The SYSCALL_MODULE macro gets passed the following parameters:

Argument	Description
name	Name is a generic name used for the system call.
offset	Offset is the system call number. This is the index into the kernel global `sysent` structure.
sysent	The `sysent` structure defined for this system call.
evh	The load handler function name.
arg	This is reserved and usually set to NULL.

The `copymem` SYSCALL_MODULE declaration is contained in Listing 3-10.

```
/* declare the system call */
SYSCALL_MODULE(copymem, &syscall_num, &copymem_sysent, load, NULL);
```

Listing 3-10

The SYSCALL_MODULE macro takes five arguments:

`Copymem` is a unique name for the KLD. The second parameter `syscall_num` represents the system call number, which also represents the offset in the kernel global sysent structure containing system calls. The third parameter contains the `sysent` structure for the new system call. The fourth argument is a pointer to the load handler for this KLD. The fifth and final argument represents a pointer for the user-defined KLD data to the load handler.

A Simple Debugger

In the previous section we created a new system call that provides a mechanism for a user program to read and write kernel memory. The remainder of this chapter defines a program that implements a simple command parser, giving us a utility for reading and modifying kernel memory.

Command Definitions

The `copymem` utility is command driven. To simplify command parsing, a structure data type is defined, `command_t`, which contains an ASCII command string, a function pointer, and a help string.

```
/*
**  command definition
*/
typedef struct
{
  char  *command;       /* string representing command     */
  fptr  functionptr;    /* pointer to command implementation*/
  char  *helpstring;    /* text help string                */
} command_t;
```

Listing 3-11

The command element contains an ASCII string that is used to compare the command with user input. The function pointer contains a pointer to a routine that implements the command.

```
/*
**  command function pointer definition
*/
typedef void (*fptr) (int, char *);
```

Listing 3-12

The function type, `fptr`, is defined as the prototype for all command handlers. Every command is passed by two parameters. The first parameter is the system call number. Because `copymem` is a KLD system, there is no system call wrapper, so the system call is made by using the `syscall` system call. It takes the system call number and system call parameters and performs the system call as defined in the previous section.

```
#include <sys/syscall.h>
#include <unistd.h>
int syscall(int number, ...);
```

The second is the command string entered by the user. Every command handler is self-contained so each command handler parses its own arguments from the command string.

Command Table

The command table represents all the commands implemented by the `copymem` utility. Our implementation of the `copymem` utility has four commands: read kernel memory, write kernel memory, quit, and help.

A command table is defined so the parser can iterate through all the commands after receiving user input. All the command handlers are forward declared so we can declare our command table.

```
/*
**       command table definition
*/
void     read_handler(int num, char* args);
void     write_handler(int num, char* args);
void     quit_handler(int num, char* args);
void     help_handler(int num, char* args);

command_t     commands[ ] =
{
   "read",    read_handler,  "read address length - reads memory",
  "write",    write_handler, "write address length - writes memory",
   "quit",    quit_handler,  "quit - exits program",
   "help",    help_handler,  "help - displays command help",
   NULL,      0,              NULL,         /* terminating record */
};
```

Listing 3-13

The command table contains all the implemented commands. Each entry in the command table links the ASCII command with its command handler and help string.

Adding a new command is straightforward. Declare the command handler and add the ASCCI string, command handler, and help string to the command table. After adding the entry to the command table, implement the command handler.

The `dumpmem` Function

The `dumpmem` function, contained in Listing 3-14, is a utility function that dumps memory in hexadecimal and ASCII character format. `Dumpmem` is called by the `read_handler` function to display the request kernel memory.

```
/*
**    name:    dumpmem
**    effect: dumps memory in hexadecimal and ASCII formats
*/
static int32_t
dumpmem(uint32_t kernp, uint8_t* userp, uint32_t len)
{
  int32_t i, j;
  int32_t rows = len / CHARS_PER_ROW;
  uint8_t *ptr = (uint8_t *)userp;

  printf("\n\n");

  for (i = 0; i < rows; i++)
  {
    uint32_t    kernaddr;

    kernaddr = kernp + (i * CHARS_PER_ROW);

    printf("%08x:  ", kernaddr);

    for (j = 0; j < CHARS_PER_ROW; j++)
      printf("%02x ", ptr[ i * CHARS_PER_ROW + j] );

    for (j = 0; j < CHARS_PER_ROW; j++)
    {
    if (isprint((int) ptr[ I * CHARS_PER_ROW + j] ))
        printf("%c", ptr[ i * CHARS_PER_ROW + j] );
      else
        printf(".");
    }

    printf("\n");
  }

  printf("\n");

  return(0);
}
```

Listing 3-14

`Dumpmem` is used to display the memory in three columns. The first column contains the kernel address; the second column is 8 bytes of data displayed in hexadecimal notation. The last column is the same 8 bytes of data contained

in the second column in ASCII format. If character is nonprintable, the dot (.) character is printed.

Command Function Handlers

Each command implements its own command handler. The command handler is responsible for parsing its parameters and performing the command action.

Read Command

The `read` command is used to read and display kernel memory. The `read` command takes two parameters, the kernel address and a length. The maximum size of a `read` is 4096 bytes defined by the BUFFER_MAX macro. This maximum is an arbitrary value.

```
void read_handler(int num, char* iobuf)
{
  int32_t        stat = 0;
  int32_t        i;
  uint32_t       kerneladdr;
  uint32_t       length;

  /* verify the command arguments */
  i = sscanf(iobuf, "%*s %x %x", &kerneladdr, &length);
  if (i != 2)
    return;

  /* limit the command to our buffer size */
  if (length > BUFFER_MAX)
    length = BUFFER_MAX;

  /* perform the command */
  stat = syscall(num, kerneladdr, iobuf, length, KERNEL_READ);
  if (stat != 0)
  {
    printf("syscall failed\n");
    return;
  }

  dumpmem(kerneladdr, iobuf, length);
}
```

Listing 3-14

After parsing the parameters, the call to `copymem` is made. Because `copymem` is a user-defined system call, the call is made using the `syscall` function.

Write Handler

The write command is used to write kernel memory. The write command takes two parameters, the kernel address and a length. The maximum size of a write is 4096 bytes defined by the BUFFER_MAX macro. This maximum is an arbitrary value.

```
void write_handler(int num, char* iobuf)
{
  int32_t        stat = 0;
  int32_t        i;
  uint32_t       kerneladdr;
  uint32_t       length;

  /* verify command arguments */
  i = sscanf(iobuf, "%*s %x %x", &kerneladdr, &length);
  if (i != 2)
    return;

  /* limit the command to our buffer size */
  if (length > BUFFER_MAX)
    length = BUFFER_MAX;

  /* read the input buffer */
  printf(">> ");
  for (i = 0; i < length; i++)
    iobuf[ i] = getchar();

  printf("\n");

  /* perform the command */
  stat = syscall(num, kerneladdr, iobuf, length, KERNEL_WRITE);
  if (stat != 0)
  {
    printf("syscall failed\n");
  }
}
```

Listing 3-15

After parsing the write parameters, the write command accepts additional input from the user, the data to write to the location. Once the data is input, the write handler calls the `copymem` system call to write the kernel memory.

Quit Handler

The quit handler is used to clean up and exit the program. The quit command handler does not use the input parameters; they are ignored.

```
void quit_handler(int num, char* args)
{
   /*
   **     since this is the program exit we must free the user
   **     buffer malloced in the main program
   */
   free(args);
   exit(0);
}
```

Listing 3-16

The quit handler frees the user buffer used for commands, then exits the program normally.

Help Handler

The help command handler is used to display help text strings. The help command handler does not use the input parameters; they are ignored.

```
/*
**       display help commands
*/
void help_handler(int num, char* args)
{
   command_t*   cmdptr;

   /* parse the users command */
   cmdptr = &commands[ 0] ;
   while (cmdptr->command != NULL)
   {
      printf("\t%s\n", cmdptr->helpstring);
      cmdptr++;
   }
}
```

Listing 3-17

Every command entry contains a help text string. The help command handler iterates through all the commands and displays each command help string.

The `main` Program

The `copymem` program handles initialization and command parsing. Since the `copymem` system call is a KLD and the system call number is dynamically assigned, the system call number must be determined by querying the kernel module subsystem. This can be accomplished by calling `modfind` and `modstat`. The `modfind` call takes the name of a kernel module; the `copymem` system call is named `copymem`; `modfind` returns the module ID.

```
#include <sys/param.h>
#include <sys/module.h>
int modfind(const char *modname);
```

Once we have the module ID, `modstat` is called to obtain the `module_stat` structure. The `module_stat` structure contains the system call number in the `module_stat.data.intval` element.

```
#include <sys/param.h>
#include <sys/module.h>
int modstat(int modid, struct module_stat *stat);
```

Now that we have the dynamically assigned system call number for `copymem`, the system call number is passed to the handler functions, so they can make the appropriate system call.

Once we have determined the system call number, a user buffer is allocated to pass the command arguments to the command handler functions. The size of the user buffer BUFFER_MAX is arbitrary. The `main` function listing is contained below.

```
int main(int argc, char** argv)
{
  int   err;
  int   syscall_num;
  char* userp = NULL;
  struct module_stat stat;

  /* verify the module is loaded */
  memset(&stat, 0, sizeof(struct module_stat));
  stat.version = sizeof(struct module_stat);
```

```
  err = modstat(modfind("copymem"), &stat);
  if (err != 0)
  {
    printf("%s unable to obtain dumpmem system call
information\n", argv[0]);
    exit(0);
  }

  /* retrieve the system call number */
  syscall_num = stat.data.intval;

  userp = (uint8_t *)malloc(BUFFER_MAX);
  if (userp == NULL)
  {
    printf("%s: unable to allocate user buffer\n");
    exit(-1);
  }

  while (1)
  {
    command_t*  cmdptr;

    printf("\n> ");
    gets(userp);

    /* parse the users command */
    cmdptr = &commands[0];
    while (cmdptr->command != NULL)
    {
      if (strncmp(userp, cmdptr->command, strlen(cmdptr->com-
mand)) == 0)
      {
        cmdptr->functionptr(syscall_num, userp);
      }

      cmdptr++;
    }
  }

  /* since quit handles cleanup and exit we'll never get here */
  return(err);
}
```

Listing 3-18

After initialization, the `main` program enters an endless loop, processing commands and calling the command handler functions. The program is terminated when the user enters the quit command. Then the quit command handler cleans up and exits.

An example

Now that we've created basic kernel debugger functionality, let's try an example. Contained in the kernel is the OS version. For a simple, benign test in modifying kernel memory, we can use our newly developed `copymem` utility to modify the OS version in kernel memory.

Before we begin our test, let's display the OS version using the standard `uname` command.

```
# uname -r
4.4-RELEASE
```

Here, `uname` displays the OS version 4.4-RELEASE. The OS version is contained in the kernel global variable named `osversion`. We can determine the address in memory OS `osversion` by scanning the kernel program using the `nm` utility and looking for the `osversion` variable.

```
# nm /kernel | grep osrelease
c0205a94 r __set_sysctl_set_sym_sysctl___kern_osrelease
c023f4c2 D osrelease
c022aca0 d sysctl___kern_osrelease
```

From the output of the selected output of the `nm` command, we see that the kernel address of `osrelease` is c023f4c2. We can now run our `copymem` utility to modify this location in memory. Before we modify the memory, we will display it to verify the address is correct. After verifying the address, we overwrite the version with new data, then display the same location again to see that the address has been modified.

```
# ./copymem

> read c023f4c2 10


c023f4c2:  34 2e 34 2d 52 45 4c 45 4.4-RELE
c023f4ca:  41 53 45 00 00 00 c0 b6 ASE.....
```

```
> write c023f4c2 8
>> cookbook


>
> read c023f4c2 10


c023f4c2:  63 6f 6f 6b 62 6f 6f 6b cookbook
c023f4ca:  41 53 45 00 00 00 c0 b6 ASE.....


> quit
```

Using `copymem`, we have successfully modified the `osversion` string. As one final check, we will rerun the `uname` command and see that, in fact, we have modified the FreeBSD OS version string.

```
# uname -r
cookbookASE
```

The output of the `uname` utility now displays the expected results.

Summary

In this chapter we have discussed the details of FreeBSD system calls. Understanding how these work is an excellent introduction to kernel hacking. As part of our discussion, we've implemented a system call and utility that will allow us to read and write a kernel memory from an application program.

CHAPTER FOUR 49

Device Driver

Overview

A device driver is an extension of the FreeBSD kernel that implements a standard software interface to hardware. Device drivers consist of data structures and a fixed set of functions provided by the device driver writer. The kernel calls driver functions in response to conditions, such as a driver load, power management event, device interrupt, or an application requesting a service.

In order to develop a device driver, an understanding of the related kernel data structures and driver method functions is needed. This chapter covers topics including:

- Driver environment
- Device driver kernel data structures
- Driver method functions
- Steps for developing a FreeBSD device driver
- The PCIO-DIO24 device driver

Driver Environment

A FreeBSD device driver contains two major components, autoconfiguration, device_method_t, and the device switch table, cdevsw. Before discussing the implementation of the data structures, let's take a look at the environment of a device driver and the role each of the data structures plays in a running FreeBSD system.

Autoconfiguration

The autoconfiguration code detects the hardware at load time and is responsible for allocating hardware resources during load, deallocating hardware resources on unload, and putting hardware in a consistent state in response to power management events. The auto configuration code typically is only used at load and unload time.

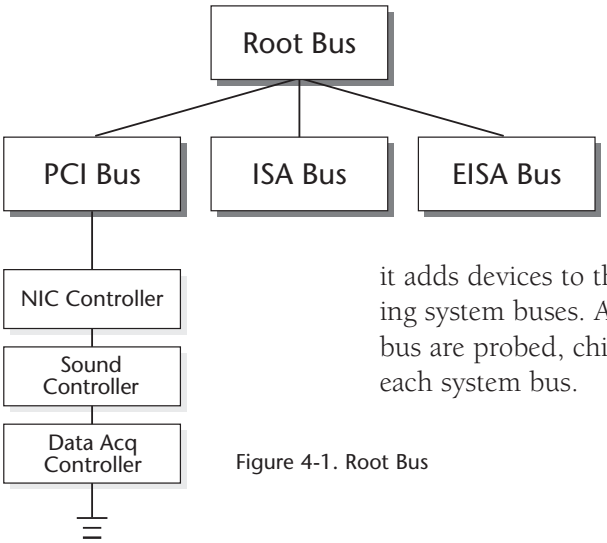


Figure 4-1. Root Bus

The FreeBSD kernel maintains a tree of device objects. At system startup a root device is created called the `root_bus`. When the kernel code boots, it adds devices to the root device, representing system buses. As devices on each system bus are probed, child devices are added to each system bus.

Autoconfiguration is the procedure carried out by the FreeBSD kernel that dynamically finds and enables hardware. The kernel probes for system buses and, for each bus that is found, devices are attached, initialized, and configured. During autoconfiguration, a device driver probe routine is called. Probe is responsible for detecting the hardware to determine if any other devices are attached. Once a device is successfully probed the FreeBSD kernel must attach to it. The attached function initializes the device hardware and any software state.

The Device Switch Table

The device switch table consists of a set of routines that comprise the upper and lower halves of the device driver. The upper half provides the system call implementation for the device driver such as read, write, open, ioctl and close. Upper half functions execute synchronously with a user process and are preemptable, permitted to block. The lower half routines interface with the device registers and implement the hardware interrupt service routine. Lower half routines are not preemptable and cannot block.

A typical device driver accepts requests from the upper half, and then enqueues the request to be handled by the lower half. Each request is enqueued in a common data structure shared by the upper and lower halves. Because the upper and lower halves of a device driver run independently, it is critical that access to any data shared by the upper and lower halves of a device driver is properly synchronized.

KLDs Revisited

As with the system call described in Chapter 3, FreeBSD provides support for dynamically loadable device drivers. In addition to the typical device driver data structures required by a FreeBSD device driver, KLD framework data structures for dynamic load and unload features for the DIO device driver are described and implemented.

Driver Structure

A FreeBSD device driver consists of various entry points into driver functions or methods that the FreeBSD I/O subsystem calls when it wants the driver to perform a specific function. Two structures are provided for FreeBSD device driver writers to implement the driver functions. `device_method_t` is the structure used for auto configuration functions and `cdevsw` is used for system calls and interrupts. The following sections discuss the more common function callbacks and the conditions that cause the functions to be called.

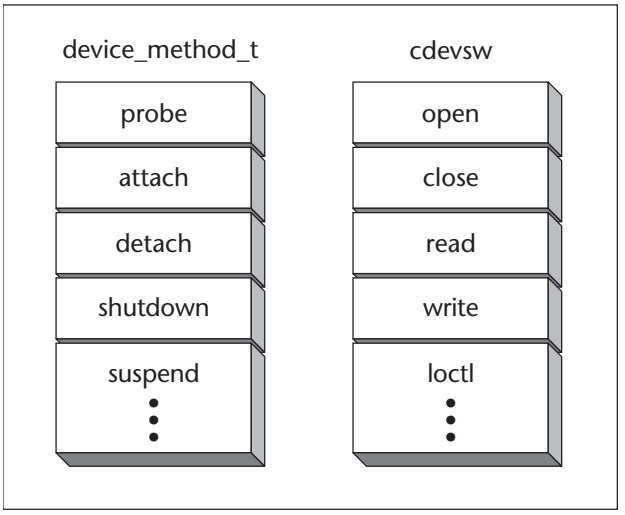


Figure 4-2. Device Driver

Driver Data Structures

The `device_method_t` Structure

The driver's `device_method_t` is the list that contains the driver's autoconfiguration method functions. The `device_method_t` contains a list of functions that are implemented for this driver and is terminated by a null entry. The `device_method_t` structure is defined in `/usr/include/sys/bus.h`.

```
typedef struct device_op_desc   *device_op_desc_t;

typedef struct device_method    device_method_t;

typedef int (*devop_t)(void);

struct device_method {
     device_op_desc_t    desc;
     devop_t     func;
};
```

The following is a list of the more common device method functions with brief descriptions of each.

`int` **`probe`**`(device_t dev)`

The `probe` method is called when the driver is loaded and is used to determine if the device is present. If `probe` is successful in finding the device then it should return 0; otherwise an appropriate error code should be returned.

`int` **`attach`**`(device_t dev)`

If the probe is successful, then the `attach` driver method is called, which is responsible for initializing the hardware, allocating system resources and adding the device switch table entry into the kernel global device switch table. `attach` should return 0 on success.

`int` **`detach`**`(device_t dev)`

`Detach` is called when a driver is about to be removed from the system. `detach` is responsible for putting the hardware in a consistent state, deallocating system resources and removing the device switch table entry from the kernel global device switch table. The `detach` method should return 0 on success.

`int` **`shutdown`**`(device_t dev)`

The `shutdown` method is called during system shutdown to allow a driver to place its hardware in a quiescent state.

`int` **`suspend`**`(device_t dev)`

The `suspend` method is used by the power management subsystem to allow the driver save configuration before power is removed.

`int` **`resume`**`(device_t dev)`

The `resume` method is used by the power management subsystem to allow the driver to initialize before power is applied.

The `cdevsw` Structure

In addition to the autoconfiguration component of the device driver, each driver contains a device table structure, represented by the `cdevsw` structure. The `cdevsw` table contains functions that implement standard system calls such as `open`, `close`, `read`, `write`, and `ioctl`.

The `cdevsw` structure defines an entry in FreeBSD's kernel device switch table. The device switch table, `cdevsw`, is a kernel data structure that contains an entry for every device driver in the kernel. The `cdevsw` structure is defined in `/usr/include/sys/conf.h`.

```
/*
 * Character device switch table
 */
struct cdevsw {
    d_open_t    *d_open;
    d_close_t   *d_close;
    d_read_t    *d_read;
    d_write_t   *d_write;
    d_ioctl_t   *d_ioctl;
    d_poll_t    *d_poll;
    d_mmap_t    *d_mmap;
    d_strategy_t    *d_strategy;
    const char  *d_name;    /* base device name, e.g. 'vn' */
    int     d_maj;
    d_dump_t    *d_dump;
    d_psize_t   *d_psize;
```

```
    u_int       d_flags;
    int     d_bmaj;
    /* additions below are not binary compatible with 4.2 and
            below */
    d_kqfilter_t    *d_kqfilter;
};
```

We'll take a look at each individual element of the `cdevsw`.

```
static int d_open(dev_t dev, int flags, int devtype, struct proc *p)
```

`open` is called when a process calls the open system call with the device special filename. It is responsible for enforcing any rules or restrictions that the device may have.

```
static int d_close(dev_t dev, int flags, int devtype, struct proc *p)
```

`close` is called when a process calls the close system call with the file descriptor obtained by calling open with the device special name. It is responsible for any device cleanup task, such as putting the device in a consistent state.

```
static int d_read(dev_t dev, struct uio* uio,  int flags)
```

`read` is called when a process calls the read system call with the file descriptor obtained by calling open with the device special name.

```
static int d_write(dev_t dev, struct uio* uio,  int flags)
```

`write` is called when a process calls the write system call with the file descriptor obtained by calling open with the device special name.

```
static int d_ioctl(dev_t dev, ulong cmd, caddr_t data, int flag,
struct proc *p)
```

`ioctl` is called when a program calls the `ioctl` system call with the file descriptor obtained by calling open with the device special name. It is used to provide any device-dependent operations.

```
static int d_poll(dev_t dev, int which, struct proc *p)
```

`poll` is used by the driver to see if a specific event has occurred.

`static int` **`d_mmap`**`(dev_t dev, struct vm_offset_t offset, int nprot)`

`mmap` is used to map memory into a process space.

`static void` **`d_strategy`**`(struct buf *bp);`

Used to start a read or write operation on the lower half of the driver.

`char` **`d_name`**

`d_name` represents the device driver name.

`int` **`d_maj`**

`d_maj` is the device major number. Every device has a major number. Typically, when you begin developing a device driver you would consult `/usr/src/sys/conf/majors` to find an unused major number and use that.

`static int` **`d_dump_t`**`(dev_t dev);`

`dump` is used to save the contents of physical onto secondary storage when the system is about to crash.

`static int` **`d_psize_t`**`(dev_t dev);`

`psize` returns the size of a disk drive partition.

There is one more function worth noting that is not contained in the `cdevsw` structure but is an integral part of many device drivers—that is the interrupt handler. The interrupt handler is assigned at driver load time and is considered the lower half of the device driver.

`static void` **`d_intr`**`(void* arg)`

`d_intr` represents the device interrupt handler. It is responsible for any hardware function to clear a pending interrupt and to initiate any operations in response to a hardware interrupt.

The `device_t` Structure

A device object is an abstract representation of a hardware device. Every piece of hardware attached to the FreeBSD kernel is represented by a device

object. A device object that has been successfully probed and attached contains device class and device driver objects. The device object is accessed via a set of `set` and `get` routines. There are functions to add, remove and traverse parent and child nodes. We are developing a driver for a simple controller so our focus will be on the routines that relate to that. The routines used to access the driver specific fields are:

```
typedef struct device       *device_t;
```

The device structure is listed in `/usr/include/sys/bus_private.h` and listed below.

```
struct device {
    /*
     * Device hierarchy.
     */
    TAILQ_ENTRY(device)  link;   /* list of devices in parent */
    device_t        parent;
    device_list_t   children; /* list of subordinate devices */

    /*
     * Details of this device.
     */
    device_ops_tops;
    driver_t        *driver;
    devclass_t      devclass; /* device class which we are in */
    int         unit;
    char*       nameunit; /* name+unit e.g. foodev0 */
    char*       desc;    /* driver specific description */
    int         busy;    /* count of calls to device_busy() */
    device_state_t  state;
    u_int32_t       devflags; /* api level flagdevice_get_flags()*/
    u_short     flags;
    u_char      order;  /* order from device_add_child_ordered() */
    u_char      pad;
    void        *ivars;
    void        *ivars;
    void        *softc;
};
```

The `device_t` structure contains a set of accessor functions to set and get specific entries of the `device_t` function. A `device_t` accessor function is prefixed by `device_`. As we discuss the DIO device driver, you will notice

calls to the defined accessor functions. Each function is named appropriately to represent the action it performs on the `device_t` structure. This naming convention assists in the description and understanding of the DIO driver code.

The `driver_t` Structure

Every driver in the FreeBSD kernel is represented by its driver object. The driver object contains the name of the device, a list of driver method functions, type of device and size of the private data structure. The driver object is declared in the driver source code file.

```
typedef struct driver         driver_t;

struct driver {
    const char      *name;    /* driver name */
    device_method_t *methods; /* method table */
    size_t          softc;    /* size of device softc struct */
    void            *priv;    /* driver private data */
    device_ops_t    ops;      /* compiled method table */
    int             refs;     /* # devclasses containing driver */
};
```

The `softc` Structure

In addition to the kernel data structures, every driver contains a data structure used for local state information, configuration information, or any other data that must be saved. This structure is called the driver `softc` structure. Each driver declares its own `softc` structure. A standard use of the `softc` structure is for passing data between the upper and lower halves of the device driver.

`struct devclass_t`

The `devclass` object represents a class of devices. It has two objectives. The first is to provide a mapping from name to device. The second is to maintain a list of drivers. For example, a `devclass` with name `pci` would keep a list of all the drivers for PCI hardware.

```
typedef struct devclass     *devclass_t;
```

The `DRIVER_MODULE` Macro

```
DRIVER_MODULE(name, busname, driver_t driver, devclass_t devclass,
modeventhand_t evh, void *arg);
```

FreeBSD provides support for dynamic loading and unloading of drivers. The autoconfiguration code is the method to communicate with the FreeBSD dynamic kernel linker (KLD) subsystem.

The `dev_t` Structure

Every device in the system is represented by a `dev_t` structure, which contains the device switch table, maximum IO size and device flags. The `dev_t` structure is responsible for mapping the upper level calls to the actual device driver implementation. The `dev_t` and `specinfo` declarations are found in `/usr/include/sys/conf.h`.

```
typedef struct specinfo *dev_t;

struct specinfo {
    u_int       si_flags;
#define SI_STASHED  0x0001  /* created in stashed storage */
    udev_t      si_udev;
    LIST_ENTRY(specinfo) si_hash;
    SLIST_HEAD(, vnode) si_hlist;
    char    si_name[ SPECNAMELEN + 1];
    void    *si_drv1, *si_drv2;
    struct cdevsw   *si_devsw;
    int     si_iosize_max;  /* maximum I/O size (for physio &al) */
    union {
        struct {
            struct tty *__sit_tty;
        } __si_tty;
        struct {
            struct disk *__sid_disk;
            struct mount *__sid_mountpoint;
            int __sid_bsize_phys; /* min physical block size */
            int __sid_bsize_best; /* optimal block size */
        } __si_disk;
    } __si_u;
};
```

The `make_dev` Function

The `make_dev` function creates a `dev_t` structure and places it in the list of devices for the systems. All loaded devices contain an entry in the `dev_t` list.

```
dev_t make_dev(struct cdevsw *cdevsw, int minor, uid_t uid, gid_t
gid, int perms, char *name, ...)
```

cdevsw represents the device switch table for the device driver.

minor designates which minor device number is being created.

uid is the user id that owns the dev_t device that is created.

gid is the group ID that owns the dev_t device that is created

perms represents the permissions assigned to the dev_t device that is created.

name contains the device name being created.

make_dev creates a new dev_t structure. If successful, the device name represented by name is created in the /dev directory. The device is owned by the user and group contained in the call to make_dev and will consist of the permissions contained in the call to make_dev.

The **destroy_dev** Function

The destroy_dev function destroys a dev_t entry in the list of available devices in the system.

void destroy_dev(*dev_t dev*);

destroy_dev takes one parameter.

dev, the returned values from make_dev.

destroy_dev has no return value.

The DIO24 Device Driver

In this section we will develop a FreeBSD character device driver that accesses the features of the PCI-DIO24 controller. Before writing any driver code, it is a good idea to create a list of tasks the device driver is going to perform. The list of tasks to be implemented for the DIO device driver are:

- Probe the PCI-DIO24 hardware
- Allocate hardware resources during load
- Deallocate hardware resources during unload

- Read and write to the PCI DIO24 hardware registers
- Handle the PCI-DIO24 interrupt

Skeleton Driver Source

In developing a device driver, it's common to start from a skeleton device driver code base, a driver that is similar to the driver being written or a driver shell that implements the empty prototypes of common driver functions, and add features as needed. FreeBSD provides a shell script that generates a shell device driver, `make_device_driver.sh` in `/usr/share/example/drivers`. The output of the shell script contains all the necessary driver data structures, stub driver callback functions and a KLD development environment.

NOTE

The **`make_device_driver.sh`** script contained in FreeBSD 4.4 only generates an ISA device driver. I obtained the current version of **`make_device_driver.sh`** using CVSUP, which generates both ISA and PCI drivers.

CVSUP is a software package for distributing and updating collections of files across a network. All the necessary files and options are provided in the **`/usr/share/examples/drivers`** directory to update the **`make_device_driver.sh`** script.

To generate the shell device driver, call `make_device_driver.sh` with the device name to be created. For this device, I've called it `dio`, for digital IO.

```
# make_device_driver.sh dio
```

The output of the `make_device_driver.sh` script is contained in three directories. The directories and their contents are summarized in Table 4-2.

Directory	Contents
/sys/dev/dio	Contains the source code for the driver, dio.c
/usr/src/sys/sys	Contains the file dioio.h used to define driver ioctl codes
/sys/modules/dio	Contains a complete KLD build environment

make_device_driver.sh output

Table 4-2

The sample DIO device driver source contains a shell implementation of both ISA and PCI device drivers. Because the PCI-DIO24 is a PCI card, the first modification I made to the DIO source code was to remove all references to the ISA callback functions. Once this was complete, I performed a complete build to make sure the remaining code was still correct.

DIO Driver Functions

The `dio_pci_probe` Function

The first task of the device driver is to detect if its hardware is present. This is called probing the device. Device probing is performed by the probe device method. `Dio.c` contains a function for probing a PCI device implemented in the function `dio_pci_probe`.

`dio_pci_probe` calls the kernel function `pci_get_devid` to return the Vendor ID and Device ID for this device. `pci_get_dev` returns a 32-bit long-word of which the upper 16 bits contain the PCI Device ID and the lower 16 bits contains the PCI vendor ID for this card. The following code shows the modifications to the `pci_ids` structure for the PCI-DIO24.

```
static struct _pcsid
{
    u_int32_t   type;
    const char  *desc;
} pci_ids[] = {
        { 0x00281307,   "Measurement Computing PCI-DIO24 Digital
          I/O Card"  },
    { 0x00000000,   NULL                        }
};
```

The DIO device driver can now be built and loaded, and it successfully recognizes the PCI-DIO24 card. The complete `dio_pci_probe` function is listed below.

```
static int
dio_pci_probe (device_t device)
{
    u_int32_t   type = pci_get_devid(device);
    struct _pcsid   *ep =pci_ids;

    while (ep->type && ep->type != type)
```

```
            ++ep;
    if (ep->desc) {
        device_set_desc(device, ep->desc);
        return 0; /* If there might be a better driver, return -2 */
    } else {
        return ENXIO;
    }
}
```

`dio_pci_probe` scans the `pci_ids` structure in an attempt to match the vendor and device IDs. When a match is found, the verbal description of the device is updated using the kernel function `device_set_desc`.

TIP

FreeBSD has a utility that dumps information about PCI devices, called `pciconf`. Utilities such as `pciconf` are used for debugging and verifying your hardware is working correctly before developing a device driver.

The `pciconf` utility is used to detect that the hardware is successfully loaded and recognized by FreeBSD. `pciconf -l` will give you a list of PCI devices present in your system. Once you have installed a hardware peripheral, it should always be listed in the output of the `pciconf` command.

The `dio_pci_attach` Function

Once a device driver successfully probes the hardware, the kernel calls the driver attach method. The attach method is responsible for allocating hardware resources and for the PCI-DIO24 making the character device. The PCI_DIO24 board has three hardware resources, an interrupt and two IO ports. dio.c contains the attach method `dio_pci_attach`.

```
static int
dio_pci_attach(device_t device)
{
        interror;
    struct dio_softc *scp = DEVICE2SOFTC(device);

        error =  dio_attach(device, scp);
        if (error) {
                dio_pci_detach(device);
        }
```

```
    /* handle softc initialization */
         dio_softc_init(device);

         return (error);
}
```

`dio_pci_attach` uses two utility routines to accomplish its tasks, `dio_attach` and `dio_allocate_resources`.

`dio_attach` is responsible for adding the interrupt to the list of system interrupt handlers after the interrupt has been allocated.

The `dio_allocate_resource` function handles the allocation of the interrupt and register hardware resources.

The `dio_attach` Function

The attach device handler function is called if the controller hardware is found. The attach device handler is responsible for creating the `dev_t` structure, allocating hardware resources and setting up the device interrupt.

```
dio_attach(device_t device, struct dio_softc * scp)
{
    device_t parent = device_get_parent(device);
    int unit    = device_get_unit(device);

    scp->dev = make_dev(&dio_cdevsw, 0,
            UID_ROOT, GID_OPERATOR, 0600, "dio%d", unit);
    scp->dev->si_drv1 = scp;

    if (dio_allocate_resources(device)) {
        goto errexit;
    }

    /* register the interrupt handler */
    /*
     * The type should be one of:
     *  INTR_TYPE_TTY
     *  INTR_TYPE_BIO
     *  INTR_TYPE_CAM
     *  INTR_TYPE_NET
     *  INTR_TYPE_MISC
     * This will probably change with SMPng. INTR_TYPE_FAST may be
```

```
     * OR'd into this type to mark the interrupt fast. However,
     * fast interrupts cannot be shared at all so special precautions
     * are necessary when coding fast interrupt routines.
     */


    if (scp->res_irq) {
        /* default to the tty mask for registration */  /* XXX */
        if (BUS_SETUP_INTR(parent, device, scp->res_irq,
INTR_TYPE_TTY,
                diointr, scp, &scp->intr_cookie) == 0) {
            /* do something if successful */
        } else {
            goto errexit;
        }
    }

    return 0;

errexit:
    /*
     * Undo anything we may have done
     */
    dio_detach(device, scp);
    return (ENXIO);
}
```

`dio_attach` calls the kernel function `make_dev` to create the `dev_t` entry in the `dev-t list`, then calls `dio_allocate_resource` to reserve the PCI-DIO24 controller's hardware resources.

After successfully allocating the hardware resources, including IO registers and interrupt, `dio_attach` calls the kernel MACRO BUS_SETUP_INTR, which is responsible for installing the interrupt handler, `diointr`, at the requested IRQ level. The IRQ level is retrieved during the resource allocation.

Additionally, if there is an error during the attach setup, it is usually good form to detach all resources and leave the system in the same state as when called.

The `dio_alloc_resources` Function

`dio_alloc_resources` allocates all three PCI_DIO24 hardware resources.

When allocating PCI resources, the ID value passed into `bus_alloc_resources` should be the offset into the PCI Configuration registers.

```
static int
dio_allocate_resources(device_t device)
{
    int error;
    struct dio_softc *scp = DEVICE2SOFTC(device);

       /* allocate the interrupt status/control port, bus address
1 */
       scp->rid_badr1 = 0x14;
    scp->res_badr1 = bus_alloc_resource(device, SYS_RES_IOPORT,
            &scp->rid_badr1, 0ul, ~0ul, 1, RF_ACTIVE);
    if (scp->res_badr1 == NULL) {
                scp->res_badr1 = 0;
        goto errexit;
    }

       /* allocate the data/configuration port, bus address 2 */
       scp->rid_badr2 = 0x18;
    scp->res_badr2 = bus_alloc_resource(device, SYS_RES_IOPORT,
            &scp->rid_badr2, 0ul, ~0ul, 1, RF_ACTIVE);
    if (scp->res_badr2 == NULL) {
                scp->res_badr2 = 0;
        goto errexit;
    }

       /* allocate the interrupt */
    scp->res_irq = bus_alloc_resource(device, SYS_RES_IRQ,
            &scp->rid_irq, 0ul, ~0ul, 1, RF_SHAREABLE|RF_ACTIVE);
    if (scp->res_irq == NULL) {
        goto errexit;
    }

    return (0);

errexit:
    error = ENXIO;
    /* cleanup anything we may have assigned. */
    dio_deallocate_resources(device);
    return (ENXIO); /* for want of a better idea */
}
```

`dio_allocate resources` performs its function through the use of the kernel function `bus_allocate_resource`.

The `dio_pci_detach` Function

When a device driver shuts down, whether during unload or reboot, the driver's detach routine is called. The detach routine is responsible for cleaning up and deallocating hardware resources.

```
static int
dio_pci_detach (device_t device)
{
         interror;
       struct dio_softc *scp = DEVICE2SOFTC(device);

         error =  dio_detach(device, scp);
         return (error);
}
```

The detach flow is structured in a similar design as the attach flow. There are two utility routines. `dio_detach` is responsible for removing the interrupt from the system list of interrupt handlers. Also, `dio_deallocate_resources` is used to return the register and interrupt resources to the system.

The `dio_detach` Function

The detach handler function is called in response to an event causing your hardware to shut down—for example, when the system administrator has requested your driver to unload or your PCcard hardware has been removed.

```
static int
dio_detach(device_t device, struct dio_softc *scp)
{
    device_t parent = device_get_parent(device);

    /*
     * At this point stick a strong piece of wood into the
     * device to make sure it is stopped safely. The alternative
     * is to simply REFUSE to detach if it's busy. What you do
     * depends on your specific situation.
     *
     * Sometimes the parent bus will detach you anyway, even if
     * you are busy. You must cope with that possibility. Your
     * hardware might even already be gone in the case of card
```

```
     * bus or pccard devices.
     */


    /*
     * Take our interrupt handler out of the list of handlers
     * that can handle this irq.
     */
    if (scp->intr_cookie != NULL) {
        if (BUS_TEARDOWN_INTR(parent, device,
            scp->res_irq, scp->intr_cookie) != 0) {
                printf("intr teardown failed.. continuing\n");
        }
        scp->intr_cookie = NULL;
    }

    /*
     * deallocate any system resources we may have
     * allocated on behalf of this driver.
     */
    return dio_deallocate_resources(device);
}
```

`dio_detach` handles removing the interrupt from the list of interrupt handlers so the interrupt resource can be returned to the system. Deallocating an active interrupt is a sure-fire recipe for crashing the system.

The detach routine should return 0 if it is successful.

The `dio_deallocate_resources` Function

The `dio_deallocate_resources` function handles shutting down and releasing hardware resources back to FreeBSD.

```
static int
dio_deallocate_resources(device_t device)
{
    struct dio_softc *scp = DEVICE2SOFTC(device);

    if (scp->res_irq != 0) {
        bus_deactivate_resource(device, SYS_RES_IRQ,
            scp->rid_irq, scp->res_irq);
        bus_release_resource(device, SYS_RES_IRQ,
            scp->rid_irq, scp->res_irq);
        scp->res_irq = 0;
```

```
    }

    if (scp->res_badr1 != 0) {
        bus_deactivate_resource(device, SYS_RES_IOPORT,
            scp->rid_badr1, scp->res_badr1);
        bus_release_resource(device, SYS_RES_IOPORT,
            scp->rid_badr1, scp->res_badr1);
        scp->res_badr1 = 0;
    }

    if (scp->res_badr2 != 0) {
        bus_deactivate_resource(device, SYS_RES_IOPORT,
            scp->rid_badr2, scp->res_badr2);
        bus_release_resource(device, SYS_RES_IOPORT,
            scp->rid_badr2, scp->res_badr2);
        scp->res_badr2 = 0;
    }

    if (scp->dev) {
        destroy_dev(scp->dev);
    }
    return (0);
}
```

The `dio_deallocate_resources` function uses two kernel functions to perform its task, `bus_deactivate_resource` and `bus_release_resource`.

The `dioopen` Function

The `dioopen` function is called when a process opens the device special file `/dev/dio0`. When open is called, the open character driver open function is called.

```
static int
dioopen(dev_t dev, int oflags, int devtype, struct proc *p)
{
    struct dio_softc *scp = DEV2SOFTC(dev);

    /*
     * Do processing
     */
        if (scp->open_count == 0)
```

```
        {
          /* any inits that require no open file descriptors */
        }


        scp->open_count++;

    return (0);
}
```

The `open` function does nothing more than increment a counter to track the number of opens. The `open` function should return 0 if there is no error.

The `dioclose` Function

The `dioclose` function is called when the last file descriptor opened to the DIO driver is closed. Our implementation simply clears the open count. The close function should return 0 if there is no error and the appropriate error code, if an error occurs.

```
static int
dioclose(dev_t dev, int fflag, int devtype, struct proc *p)
{
    struct dio_softc *scp = DEV2SOFTC(dev);

    /*
     * Do processing
     */
        scp->open_count = 0;

    return (0);
}
```

It is a common misconception that the close call is paired with open calls—i.e., that every open has a corresponding close. This is not the case.

The `dioioctl` Function

An `ioctl` is a device-specific interface to a device driver. Each `ioctl` can be used to either send or receive data from a user application or control device behavior. `Ioctls` are defined so that each `ioctl` is a unique number within your driver.

The `dioioctl` device method function handles four `ioctl` codes, each

used to read or write a PCI-DIO24 controller register.

```
static int
dioioctl (dev_t dev, u_long cmd, caddr_t data, int flag, struct
proc *p)
{
    struct dio_softc *scp = DEV2SOFTC(dev);
         struct dioreg_t *arg = (struct dioreg_t*) data;
         volatile int port = 0;

         /* check the args */
         if (arg->size != sizeof(struct dioreg_t))
         {
           return (ENXIO);
         }

    switch (cmd) {

         case DHIOPORTREAD:
       if (arg->reg != CONFIG)
       {
         port = scp->res_badr2->r_start + arg->reg;
         arg->val = DIO_INB(port);
       }
       else
       {
         /* copy the shadow value of the CONFIG register */
         arg->val = scp->config;
       }
            break;

         case DHIOPORTWRITE:
       port = scp->res_badr2->r_start + arg->reg;
       DIO_OUTB(port, arg->val);

       /* shadow the CONFIG register, reg is write only */
       if (arg->reg == CONFIG)
       {
         scp->config = arg->val;
       }
            break;

         case DHIOCTRLREAD:
           port = scp->res_badr1->r_start + arg->reg;
           arg->val = DIO_INL(port);
           break;
```

```
        case DHIOCTRLWRITE:
          port = scp->res_badr1->r_start + arg->reg;
          DIO_OUTL(port, arg->val);
          break;

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

Register Shadowing

One notable exception to this code is the `config` register. The `config` register is a write-only register. Any attempt to read the `config` register returns undefined results. To provide read capability of the `config` register, a copy of each write to the `config` register is saved, or cached. By caching a copy of the write to the `config` register, the driver has the value to provide to an application when a `config` register read is performed. This technique is called register shadowing.

Defining `ioctl` Codes

The DIO device driver has four `ioctl`s defined, an `ioctl` to read and write to both of the defined PCI-DIO24 register base addresses. It is important to note that there are multiple registers offset from PCI Base Address 2. Each `ioctl` is provided with a mechanism to set the offset, allowing access to any of the PCI-DIO24 registers.

Before defining our `ioctl`s, it is necessary to understand the details for defining them. These can be found in the file `/usr/include/sys/ioccom.h`, which contains macros used to define `ioctl`. These macros are summed up in the following table:

_IO(g, n)	Used to define a void ioctl
_IOR(g, n, t)	Used to define an ioctl that reads data
_IOW(g, n, t)	Used to define an ioctl that writes data
_IOWR(g, n, t)	Used to define an ioctl that reads and writes data

ioctl macros

Table 4-3

Each of these macros is used as a basis for defining `ioctls`. For example, the sample `ioctl` provided in `dioio.h`, DHIOCRESET, uses _IO, it doesn't read or write any data from the driver, and it issues the command to reset the card.

The parameters for the `ioctl` macros are:

g	A unique code, typically an ASCII definition such as 'N'
n	A unique number for this ioctl
t	The size of the data used for this ioctl

ioctl macro parameters

Table 4-4

Four `ioctls` have been defined, one each for a read or write PCI configuration registers listed in base address registers one and two.

```
#define DHIOPORTREAD    _IOWR('D', 1, struct dioreg_t)  /* read
bar2  */
#define DHIOPORTWRITE   _IOW('D',  2, struct dioreg_t)  /* write
bar2 */
#define DHIOCTRLREAD    _IOWR('D', 3, struct dioreg_t)  /* read
bar1  */
#define DHIOCTRLWRITE   _IOW('D',  4, struct dioreg_t)  /* write
bar1 */
```

Each `ioctl` represents either a read or write operation to one of the two defined register spaces, bar1 and bar2.

In addition to the `ioctl` codes defined above, the format of the data passed between an application and the device driver must be defined. Since the `ioctls` define read and write registers, minimally the data passed to the device driver should contain the offset into the register space and the value to read or write. For the purpose of data integrity, the size of the structure being passed has been added. This structure used to pass data between an application and the device is defined in `dioio.h` and is called `dioreg_t`.

```
struct dioreg_t
{
  int32_t   size;
```

```
  int32_t   reg;
  int32_t   val;
};
```

`dioreg_t` contains three elements: `size` represents the size of the structure; `reg` is the register that is being read or written; and, finally, `val`, which contains the value to be written to the register if the `ioctl` is a write operation or it is the value of the register if it is a read operation.

The `diointr` Function

The `diointr` function is called when the PCI-DIO24 controller generates an interrupt. The requirements for the interrupt handler in our application are minimal—if there is an interrupt, clear it and proceed.

```
static void
diointr(void *arg)
{
    struct dio_softc *scp = (struct dio_softc *) arg;

      uint32_t interrupt_stat;
      volatile uint32_t port = scp->res_badr1->r_start + INTSR;

      /* read the interrupt status register */
      interrupt_stat = DIO_INL(port);

      /*
      ** if there is an interrupt pending clear it otherwise
      ** do nothing. The controller is the PLX Technologies
      ** 9052, see the spec sheet at http://www.plxtech.com
      ** for details.
      */

      if (interrupt_stat & 0x04)
      {
        /* interrupt is active clear it */
        DIO_OUTL(port, interrupt_stat | 0x0400);
      }

    return;
}
```

We learn from the PLX 9052 specification that to check to see if an interrupt is pending, the Interrupt Status register is read. The DIO interrupt handler

reads the interrupt status register, then tests the interrupt pending bit, 0x04. If it is determined that an interrupt is pending, then the interrupt is cleared by writing the interrupt clear bit 0x0400 back to the interrupt status register.

A typical interrupt handler would perform some action, reading a value from a sample register and storing it in an array or set an event in response to some external event. Interrupt handlers are one of the most application- and device-specific pieces of source code to write.

The Device File

The device file is a special file used by an application to gain access to the device driver. Device files are files opened by an application, like an ordinary file, but FreeBSD accesses a device driver instead of a file. Device files reside in the /dev directory and are created with the `mknod` command.

`mknod` creates a device special file of the specified major and minor number provided to the `mknod` command. The syntax of the `mknod` command is as follows:

mknod *name* **c** *major minor*

Major numbers are defined in the file, `/usr/src/sys/conf/majors`. For the DIO device file, the minor number is zero. Minor numbers are provided so that drivers, such as the terminal driver, can have multiple instances of a single device driver. Each driver instance is represented by a different minor number.

The syntax for creating the DIO device file is as follows:

```
# mknod DIO c 200 0
```

With the creation of the device file, driver development is completed. However, there is one additional step. FreeBSD provides a utility, MAKEDEV, which provides a convenient method for administrators to create device files. It is a generally accepted practice to add your device file creation to the MAKEDEV script when creating a new device driver. The modification to MAKEDEV for the DIO device driver is listed below.

```
dio*)
    unit=`expr $i : 'dio\(.*\)'`
```

```
    mknod dio$unit c 200 `unit2minor $unit`
    ;;
```

MAKEDEV is invoked with the name of the device with the minor number appended to the name. For example,

```
# MAKEDEV dio0
```

The MAKEDEV script strips off the minor number from the device name for use by the `mknod` command. This is represented by the `unit2minor $unit` syntax.

Building, Loading and Unloading

The `make_device_driver.sh` script provides a complete kernel-loadable driver build environment. The `make` file contains commands to build, clean, install, and uninstall the device driver. Let's take a look at the build environment and each command in a little more detail.

`make_device_driver.sh` created a KLD build directory for the DIO device driver in the `/sys/modules/dio` directory. Before attempting any of the commands described below, cd to the `sys/modules/dio` directory.

```
# cd /sys/modules/dio
```

A common source of bugs and inconsistencies during driver development stems from building a driver with stale object files. The *make clean* command deletes all the object and the driver files forcing all the components to be built and linked.

```
# make clean
```

A device driver consists of include files and other disjoint pieces of source code that must be compiled in a specific order. The KLD driver build environment contains a command that computes and resolves build dependencies. The *make depend* command handles the details of computing the dependencies for the device driver build.

```
# make depend
```

Once the build dependencies have been resolved and are up to date, the next revision of the device driver can be built. The *make* command is used to build all the components of the device driver and link them together to create the device driver module.

```
# make
```

The output of the build process is the file `dio.ko`, the driver KLD.

During the development process a driver is built, tested, modified and features added. KLDs are installed and uninstalled dynamically. In order to iterate revisions of the device driver, it is necessary to uninstall the current revision. The *make uninstall* command unloads the current running version. `Make uninstall` calls `kldunload` to remove the driver from the kernel.

```
# make uninstall
```

To install a version of your driver *make install* dynamically loads the driver into the running system. `Make install` calls `kldload` to load the driver into the running kernel.

```
# make install
```

Additional commands may be added by each individual driver developer. However, the tools provided by FreeBSD give the driver developer a robust set of utilities to hit the ground running for developing a device driver.

Summary

This chapter covered the details needed by a device driver developer to implement a working FreeBSD device driver. Major data structures, available tools and a description of the driver build environment have been covered, giving the driver developer the necessary information to implement a functioning FreeBSD device driver.

CHAPTER FIVE 5

Midlevel Interface Library

Overview

In the previous chapter, we developed a FreeBSD device driver to read and write the PCI-DIO24 hardware registers and service the controller interrupt. In this chapter, we will develop a shared library that uses the DIO device driver and serves as the software interface for programming the PCI-DIO24 data acquisition board.

Providing an application interface library isolates the device programming details and operating system specifics to the library and device driver. This design enables system engineer tasks to interface with the PCI-DIO24 controller using a simple *C* interface and minimizes the impact of changing the underlying hardware or operating system.

Topics discussed in this chapter are:

- Shared libraries
- Creating and using shared libraries
- Accessing FreeBSD device drivers
- Low-level system calls
- PCI-DIO24 register definitions
- Design and implementation of the digital IO interface shared library

Shared Libraries

A shared library is code that is grouped together to provide specific functionality that can be reused across multiple programs. For example, without

the Standard C library, `libc`, every software engineer would need to write low-level IO routines, string routines, math routines, and so forth.

There are typically two types of libraries developed by system engineers, static and shared. If a program is linked to a static library, the components of the library become part of the final executable, making the output one monolithic program. All linker symbols are resolved at link time, and the program is self-contained. If the library is modified for bug fixes or new features, the application must be rebuilt to incorporate the changes made to the static library.

When linking with a shared library, the final executable contains references to the shared library. Shared libraries are loaded when a program is run, and the symbols are resolved at run-time. If a shared library is modified, only the library is replaced. A program can use the updated library the next time it runs.

Creating and Using Shared Libraries

Shared libraries are loaded at run-time and are not linked to a specific run-time address. Shared libraries use the same naming conventions as static libraries; the name of the shared library is prefixed with `lib`, but the extension is `.so`, instead of `.a`, which is used for static libraries.

When compiling the source code for a shared library, the code must be linked as position independent code. This is accomplished by specifying the `–fpic` switch to `gcc`.

```
# gcc –c –fpic -o dioif.o dioif.c
```

When building a shared library, `gcc` needs to know that the output is a shared library. This is done by specifying the –shared switch. For example, the code presented in this chapter is contained in the file `dioif.c`. In order to build a shared library, the following line would be used with `gcc`.

```
# gcc –shared -o libdioif.so dioif.o
```

Linking with a shared library is the same as linking with a static library; i.e., the –l switch specifies the library, and the -L switch specifies a path to find the library.

```
# gcc –o diotest diotest.c –L. -ldioif
```

The `ldd` command can be used to list the shared libraries that link into an executable. All the listed libraries must be available when the program is run or a warning message is printed and the program will not execute.

Loading Shared Libraries

A program linked with shared libraries contains a list of required shared libraries but not the location where the libraries exist. When a program is executed, the dynamic linker, `ld-elf.so.1`, searches the default library path, consisting of `/lib` and `/usr/lib`. If a shared library is installed in another directory, the dynamic linker needs to be told where to search for those shared libraries.

There are two methods for adding additional shared library search paths to the dynamic linker.

The first solution is to link the program and specify the location of the shared libraries, using the –Wl,–rpath option to gcc. This is a linker switch that adds the named directory to the run-time search path for shared libraries.

```
# gcc –o diotest diotest.c –L. –ldioif –Wl,-rpath /usr/proj/lib
```

When `diotest` is executed `libdioif.so` will be loaded from `/usr/proj/lib`.

The other solution is to set the LD_LIBRARY_PATH environment variable. This is a colon-separated list of directories. If LD_LIBRARY_PATH is set, the directories listed are searched, in addition to the default libraries searched by the linker.

Accessing the Device Driver

A device driver is a kernel extension that is inaccessible to a user program. Access to device drivers is provided through special files, known as device files, and low-level system calls. In the previous chapter, we developed a device driver for the PCI-DIO24 controller. In this section, we will look at the necessary details to access the device driver and program the PCI-DIO24 controller.

Device Files

In FreeBSD, all input and output is performed by file operations. Devices are represented by special files called device files. A device file appears as a regular filename to an application, but it is bound to a specific device driver through a device number. A device number is derived from the device major and minor numbers. The major number identifies the device driver bound to a specific device. A minor number identifies a sub device for the major devices. The minor number is used by the device driver to determine which device to send the IO request to.

Let's take a look at the device files for the standard ATAPI disk driver, `ad0`. If you issue the `ls -l ad0*` commands in the `/dev` directory, you'll see the following files with two numbers separated by commas. These are the major and minor numbers.

```
crw-r---  2 root  operator  116,   0 Oct 24 17:07 ad0a
crw-r---  2 root  operator  116,   1 Oct 24 17:07 ad0b
crw-r---  2 root  operator  116,   2 Oct 24 17:07 ad0c
crw-r---  2 root  operator  116,   3 Oct 24 17:07 ad0d
crw-r---  2 root  operator  116,   4 Oct 24 17:07 ad0e
crw-r---  2 root  operator  116,   5 Oct 24 17:07 ad0f
crw-r---  2 root  operator  116,   6 Oct 24 17:07 ad0g
crw-r---  2 root  operator  116,   7 Oct 24 17:07 ad0h
```

The device `ad0a` is an ATAPI hard disk with major number 116. The minor numbers 0 through 7 represent different partitions on the hard disk.

Device Driver System Calls

A FreeBSD device driver is accessed through a standard set of low-level IO system calls. Once a program has opened the device file for a specific device, the program is able to access the device driver and a standard set of low-level IO system calls in FreeBSD. These calls are listed in the following sections.

The open System Call

Before a program can access a device driver and physical hardware, it must obtain a file descriptor. A file descriptor is obtained by opening the device file with the open system call.

```
#include <fcntl.h>
int open(const char *path, int flags, …);
```

The path argument is the name of the device file for a specific device driver. The `flags` parameter is a mask that specifies how the file is to be opened. The flags are defined in the header file `fcntl.h`.

The open system call returns a file descriptor if the call is successful or –1 otherwise. A file may not be opened if the file does not exist or a user does not have permission to access the file. If an error occurs, the system variable `errno` is set to the appropriate value.

The `close` System Call

After a program has finished accessing a device, the file descriptor should be closed. This allows the file descriptor to be reused. If the device driver provides exclusive access, closing the device will allow another program to access the device.

```
#include <unistd.h>
int close(int d);
```

The close system call takes a single parameter, a file descriptor obtained by the open system call. Close returns 0 on success and –1 otherwise. If an error occurs, the system variable `errno` is set to the appropriate value.

The `ioctl` System Call

`Ioctl` is an all-purpose interface for controller hardware. A device driver can provide a specific function through the use of the `ioctl` function. Using an `ioctl` function generally requires detailed knowledge of the device being used.

```
#include <sys/ioctl.h>
int ioctl(int d, unsigned long request, char* argp);
```

Here, `ioctl` takes a file descriptor returned by the open system call. The second argument is a request code defined by the device being used. The third argument is a pointer to a buffer provided for arguments required for the request.

`ioctl` returns –1 if an error occurs. If an error occurs, the system variable `errno` is set to the appropriate value.

The `read` System Call

Input is performed by calling the read system call. The read system call takes a buffer and length and receives data from the device driver.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
ssize_t read(int d, void *buf, size_t nbytes);
```

The first parameter to read is a file descriptor provided by the open system call. The second argument is a buffer to return the data being read. The last argument is the number of bytes being read.

`Read` returns the number of bytes read or –1 if an error occurs. If an error occurs, the system variable `errno` is set to the appropriate value. Read will return 0 if an end of file is encountered.

The `write` System Call

Output is performed by the write system call. The write system call takes a buffer pointer and length and passes it to the driver.

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>

ssize_t write(int d, const void *buf, size_t nbytes);
```

The first parameter to read is a file descriptor provided by the open system call. The second argument is a buffer to return the data being written. The last argument is the number of bytes being written. `Write` returns the number of bytes written or –1 if an error occurs. If an error occurs, the system variable `errno` is set to the appropriate value.

PCI-DIO24 Hardware Registers

Before developing the interface library for the PCI-DIO24 board, we must have a thorough understanding of the hardware registers. Hardware registers are special memory locations provided by a controller or used to control the hardware. The PCI-DIO24 contains five hardware registers. Each register is located in PCI configuration space.

The details for programming the registers are found in the PCI-DIO24 User's Manual provided by Measurement Computing along with the controller. The PCI-DIO24 controller uses a PLX 9052 PCI Slave interface controller. Additional details are found in the PCI 9052 Data Book provided by PLX Technology.

A PCI controller uses hardware resources such as IO port registers, memory-mapped registers, and interrupts. The specifics of the resources being used are contained in the device PCI Configuration registers. PCI Configuration contains 6 base address registers, 0–5. Additional details concerning PCI configuration registers are contained in Appendix C.

The name, location, register space, and description of each of the five PCI-DIO24 registers are contained in Table 5-1.

Register	Register Space	Byte Offset	Function
Configuration Register	Base Address 2	3	Controls IO data line direction
Port A Data	Base Address 2	0	Port A data lines 0 – 7
Port B Data	Base Address 2	1	Port B data lines 8 – 15
Port C Data	Base Address 2	2	Port C data lines 16 – 23
Interrupt Status/Control	Base Address 1	4C	Interrupt Control

Table 5-1. PCI-DIO24 Registers

The details of each register are provided in the following sections.

Configuration Register

The Configuration Register is responsible for programming the direction of the digital data ports. A data port is a subset of digital lines. The Configuration Register is an 8-bit register located at byte offset 3 from the PCI Base Address Register 2.

The PCI-DIO24 has four data ports, each containing a subset of the PCI-DIO24 digital lines. All of the digital lines in a specific port must be programmed in the same direction, input or output. However, each port may be programmed independently of other ports. Each bit of the Configuration Register is defined in Table 5-2.

Bit	Description	Value
0	Port C Low Direction	0 output, 1 input
1	Port B Direction	0 output, 1 input
2	Reserved	NA
3	Port C High Direction	0 output, 1 input
4	Port A Direction	0 output, 1 input
5	Reserved	NA
6	Reserved	NA
7	Reserved	Always set to 1

Table 5-2. Configuration Register

Bits 0, 1, 3, and 4 are used to program the data port directions. Setting these bits to a value of 1 will program the corresponding data port digital lines as inputs; alternately, setting the bits to a value of 0 will program the corresponding data port digital lines as outputs.

It is important to note that bit 7 must always be set to 1; this is a function of the configuration hardware register. Writing to this register with bit 7 set to 0 will produce undefined results.

Port A Data Register

The Port A Data Register is an 8-bit register located at byte offset 0 from PCI Base Address Register 2. The Port A Data register corresponds to digital IO lines 0 through 7. Each bit of the Port A Data register is defined in Table 5-3.

The Port A Data register bits correspond to digital lines 0 though 7. The direction of Port A digital IO lines is programmed by Configuration Register bit 4.

Bit	Description
0	Digital line 0
1	Digital line 1
2	Digital line 2
3	Digital line 3
4	Digital line 4
5	Digital line 5
6	Digital line 6
7	Digital line 7

Table 5-3. Port A Data Register

Port B Data Register

The Port B Data Register is an 8-bit register located at byte offset 1 from PCI Base Address Register 2. The Port B Data register is responsible for digital I/O lines 8 through 15. Each bit of the Port B Data register is defined in Table 5-4.

The Port B Data register corresponds to digital lines 8 through 15. The direction of Port B digital IO lines is programmed by Configuration Register bit 1.

Bit	Description
0	Digital line 8
1	Digital line 9
2	Digital line 10
3	Digital line 11
4	Digital line 12
5	Digital line 13
6	Digital line 14
7	Digital line 15

Table 5-4. Port B Data Register

Port C Data Register

The Port C Data Register is an 8-bit register located at byte offset 2 from PCI Base Address Register 2. The Port C Data register is responsible for digital I/O lines 16 through 23. Each bit of the Port C Data register is defined in Table 5-5.

Bit	Description
0	Digital line 16
1	Digital line 17
2	Digital line 18
3	Digital line 19
4	Digital line 20
5	Digital line 21
6	Digital line 22
7	Digital line 23

Table 5-5. Port C Data Register

The Port C Data register bits correspond to digital lines 16 though 23. The Port C Data register is different from Port A and Port B data registers in that its lines are split into two groups, Port C Low and Port C High. Each of these groups, Port C Low and Port C High, can be programmed independently. Port C Low bits correspond to bits 0 through 3 in the Port C data register. The direction of these lines is programmed by Configuration Register bit 0. Port C High corresponds to bits 4 through 7 in the Port C data register. The direction of these lines is programmed by Configuration Register bit 3.

Interrupt Status Register

The Interrupt Status Register is a 32-bit register located at PCI Base Address Register 1 at byte offset hexadecimal 4C. This register is responsible for programming interrupt enable/disable, interrupt polarity, interrupt status,

and PCI interrupt disable. Each bit of the Interrupt Status register is defined in Table 5-6.

Bit	Description	Value
0	Interrupt Enable	0 disabled, 1 enabled
1	Interrupt Polarity	0 active low, 1 active high
2	Interrupt Status	0 interrupt not active, 1 interrupt active
3	Reserved	NA
4	Reserved	NA
5	Reserved	NA
6	PCI Interrupt Enable	0 disabled, 1 enabled
7	Reserved	NA
8	Reserved	NA
9	Reserved	NA
10	Interrupt Clear	Writing a 1 clears this interrupt
31:11	Reserved	NA

Table 5-6. Interrupt Status Register

The Interrupt Status Register contains bits that program the controller interrupt, enable and disable the controller interrupt, and provide the interrupt status.

The interrupt enable bit, bit 0, enables or disables the controller interrupt; when set to 1, the interrupt is enabled; otherwise, it is disabled.

Interrupt polarity bit, bit 1, controls the type of interrupt. This bit controls on which edge the interrupt is active, low to high transition, or high to low transition. The default is active high.

The interrupt status bit, bit 2, provides us with a status bit to determine if an interrupt is active.

The PCI interrupt bit, bit 6, controls the PCI interrupt enable. A value of 1 enables PCI interrupts; 0 disables the PCI interrupts.

Writing to the Interrupt clear, bit 10, clears a pending interrupt.

The DIO24 Application Interface Library

The implementation of the DIO24 Interface library consists of four functional categories. Each functional category is related to programming the different features of the controller. The categories are register utilities, reading and writing the digital IO lines, programming the configuration register, and programming the interrupt controller. Each of the following sections addresses the functions for the functional category.

The `dio_get` and `dio_set` Functions

All accesses to the PCI-DIO24 hardware registers are provided by two internal library functions, `dio_get` and `dio_set`. These two functions are responsible for obtaining a file descriptor to the device, if necessary; packing the data into the `ioctl` structure; and making the appropriate `ioctl` call to the driver.

```
static int32_t dio_get(int32_t reg, int32_t *pval)
static int32_t dio_set(int32_t reg, int32_t val)
```

The implementation of each of these functions is similar. Listing 5-1 demonstrates how the file descriptor to the device driver is obtained. If the internal file descriptor variable `diofd` is not set, the function attempts to open the device and obtain a file descriptor using the following code.

```
/* if the device is not open, open it */
if (diofd == -1)
{
      diofd = open("/dev/dio0", O_RDWR);
        if (diofd == -1)
        {
        return(-1);
        }
}
```

Listing 5-1

Once a valid file descriptor is obtained, the code packages the request into the appropriate ioctl structure. Recall from the previous chapter that all `ioctl` to the DIO device driver accept a `dioreg_t` structure, containing the `dioreg_t` size, register, and value for the `ioctl`. The register structure used by the DIO device driver is shown in Listing 5-2.

```
struct dioreg_t
{
  int32_t   size;
  int32_t   reg;
  int32_t   val;
};
```

Listing 5-2

The `dio_set` and `dio_get` functions set a local `dioreg_t` structure and make the `ioctl` call. The code to fill in the `ioctl` and call the driver is contained in Listing 5-3.

```
/* fill in the device structures */
regt.size = sizeof(struct dioreg_t);
regt.reg  = reg;
regt.val  = val;
/* send the request to the driver */
if (reg == INTSR)
{
      ioctl(diofd, DHIOCTRLWRITE, &regt);
}
else
{
      ioctl(diofd, DHIOPORTWRITE, &regt);
}
```

Listing 5-3

Here, `dio_set` and `dio_get` must distinguish which `ioctl` to send the register access request to. Recall there are two register spaces, base address 1 and base address 2. Since the interrupt status register, INTSR, is the only register that accesses PCI base address 1, if the specified register is INTSR, the `ioctl` is sent to the control register in PCI base address 1; otherwise, the `ioctl` is sent to the port write `ioctl`, which accesses PCI base address 2.

Accessing the Digital IO Lines

Access to the digital IO lines is provided by two functions, `dio_set_line` and `dio_get_line`. The PCI-DIO24 contains 24 digital IO lines. To identify each line, a C `enum` is provided to represent each.

The `dioline_t enum`

The `enum` data type is used repeatedly throughout the digital IO library. By defining `enums`, we force the compiler to enforce type safety to our functions, which comes in handy during debugging. Listing 5-4 shows the `dioline_t enum`.

```
typedef enum dioline_t
{
   line0  = 0,
   line1  = 1,
   line2  = 2,
   line3  = 3,
   line4  = 4,
   line5  = 5,
   line6  = 6,
   line7  = 7,
   line8  = 8,
   line9  = 9,
   line10 = 10,
   line11 = 11,
   line12 = 12,
   line13 = 13,
   line14 = 14,
   line15 = 15,
   line16 = 16,
   line17 = 17,
   line18 = 18,
   line19 = 19,
   line20 = 20,
   line21 = 21,
   line22 = 22,
   line23 = 23,
   totallines
} dioline_t;
```

Listing 5-4

The `dioline_t enum` provides us with a convenient type, and since the first argument to `dio_set_line` and `dio_get_line` is type `dioline_t`, this will force the compiler to do parameter type checking. Additionally, if the underlying hardware is changed and the line values needs to be modified, the source code changes are limited to the header file and a recompile of the remaining code base.

The diolinestate_t enum

As with the line number type, there is a line state enum, diolinestate_t, which defines all the possible states for the digital IO line. Since the PCI-DIO24 provides only digital IO lines, the states are either set or clear, set signifying the active high state and clear signifying an active low state.

```
typedef enum diolinestate_t
{
  clear = 0,
  set = 1
} diolinestate_t;
```

Listing 5-5

The dio_set_line and dio_get_line Functions

Reading and writing the digital IO lines are performed by the dio_get_line and dio_set_line functions.

```
#include <dioif.h>
int32_t dio_set_line(dioline_t line, diolinestate_t val)
int32_t dio_set_line(dioline_t line, dioline_state_t val)
```

Here, dio_set_line and dio_get_line are responsible for mapping the line passed in by the user to the port where that line resides. The mapping is accomplished by the following switch statement in Listing 5-6.

```
/* convert the line to the register */
  switch (line)
  {
  case line0:
  case line1:
  case line2:
  case line3:
  case line4:
  case line5:
  case line6:
  case line7:
    reg = PORTADATA;
    break;

  case line8:
  case line9:
```

```
  case line10:
  case line11:
  case line12:
  case line13:
  case line14:
  case line15:
    reg = PORTBDATA;
    break;

  case line16:
  case line17:
  case line18:
  case line19:
  case line20:
  case line21:
  case line22:
  case line23:
    reg = PORTCDATA;
    break;

  default:
return(-1);
    break;
}
```

Listing 5-6

The switch statement takes the line input. Once the correct register is determined, the register and value are passed to the `dio_set` or `dio_get` routine. In the following case, `dio_get`:

```
/* read the data register */
dio_get(reg, &val);
```

This function calls the utility register read routine to handle the details of reading the value of the digital IO line.

Programming the Interrupt Controller

The interrupt controller functions are used to enable and disable the local interrupt, enable and disable the PCI interrupt, and program the interrupt polarity. The DIO interface library contains functions to read and write each of these bits in the interrupt controller status register.

As with the line setting functions, `enums` are used to define the values for specific states.

The `diointstate_t enum`

At any time an interrupt is enabled or disabled, to help in defining the details of the interface library API the `diointstate_t` is defined, which represents the state of the interrupt and the PCI interrupt. Both sets of functions that enable the interrupt and PCI interrupt take the `dioinstate_t` parameter. Listing 5-7 shows the `diointstate_t` data type.

```
typedef enum diointstate_t
{
  disable = 0,
  enable = 1
} diointstate_t;
```

Listing 5-7

Here, `dioinstate_t` has two values, defined as `disable` and `enable`. These values correspond to the bit settings for these bits in the interrupt controller register.

The `dio_set_int` and `dio_get _int` Functions

The PCI-DIO24 contains a bit to program the local interrupt. The local interrupt is either enabled or disabled.

```
#include <dioif.h>
int32_t dio_set_int(diointstate_t state)
int32_t dio_get_int(diointstate_t* state)
```

The local interrupt is controlled by programming bit 0 in the interrupt status control register. The local interrupt bit is defined by the INT_BIT literal.

```
#define INT_BIT 0x01
```

Setting the local interrupt bit involves reading the interrupt control status register, modifying the local interrupt bit, then writing the new value back to the interrupt control register. Listing 5-8 demonstrates the code to set the interrupt bit.

```
/* read the current value of the register */
if (dio_get(INTSR, &val) < 0)
{
  return(-1);
}

/* set only the interrupt bit */
if (state == enable)
  val |= INT_BIT;
else
  val &= ~INT_BIT;

/* write the updated value back */
if (dio_set(INTSR, val) < 0)
{
  return(-1);
}
```

Listing 5-8

To read the state of the local interrupt bit, the interrupt status control register is read, and then the state of bit 0 is read. The following code demonstrates the method for reading the local interrupt state. After reading the interrupt status control register, bit 0 is checked to determine the state of the local interrupt. Listing 5-9 demonstrates the code to read the interrupt bit.

```
/* read the current value of the status register */
if (dio_get(INTSR, &val) < 0)
{
  return(-1);
}
/* get the status of the interrupt bit */
if (val & INT_BIT)
   *state = enable;
else
   *state = disable;
```

Listing 5-9

INT_BIT is defined as

```
#define INT_BIT 0x01
```

The `if` tests the state and sets the user's input parameter appropriately.

The `dio_set_pciint` and `dio_get_pciint` Functions

As with the local interrupt, the PCI-DIO24 contains a bit to program the PCI interrupt. Access to the PCI interrupt bit is provided by the `dio_set_pciint` and `dio_get_pciint` functions.

```
#include <dioif.h>
int32_t dio_set_pciint(diointstate_t state)
int32_t dio_get_pciint(diointstate_t* state)
```

The PCI interrupt is programmed in bit 6 of the interrupt status control register. The PCI `int` bit is defined locally as:

```
#define PCIINT_BIT  0x40
```

Setting the PCI interrupt bit consists of reading the current value of the interrupt status control register, modifying the value of the PCI interrupt bit, and then writing the new value back to the interrupt control status register. Listing 5-10 demonstrates the code to write the PCI interrupt bit.

```
if (dio_get(INTSR, &val) < 0)
{
  return (-1);
}

if (state == enable)
  val |= PCIINT_BIT;
else
  val &= ~PCIINT_BIT;

if (dio_set(INTSR, val) < 0)
{
  return(-1);
}
```

Listing 5-10

As with the local interrupt, the interrupt status control register is read, then the state of the PCI interrupt bit is tested, and the user parameter is set appropriately. Listing 5-11 lists the code that reads the status register to obtain the PCI interrupt bit.

```
/* read the current value of the status register */
if (dio_get(INTSR, &val) < 0)
{
  return(-1);
}

if (val & PCIINT_BIT)
    *state = enable;
else
    *state = disable;
```

Listing 5-11

The diopolarity `enum`

Interrupt polarity determines whether the interrupt is generated on the rising or falling edge of the interrupt signal. The choices are active high, where the interrupt is generated on the rising edge, or active low, where the interrupt is generated on the falling edge. The states are defined by the `dio_polarity_t enum`. Listing 5-12 defines the diopolarity `enum`.

```
typedef enum diopolarity_t
{
  activelo = 0,
  activehi = 1
} diopolarity_t;
```

Listing 5-12

The `dio_set_polarity` and `dio_get_polarity` functions

The `dio_set_polarity` and `dio_get_polarity` functions program on which edge the interrupt is generated.

```
#include <dioif.h>
int32_t dio_set_polarity(diopolarity_t pol)
int32_t dio_get_polarity(diopolarity_t* pol)
```

The interrupt polarity bit is bit 2 in the interrupt control status register, and is defined as follows:

```
#define POL_BIT     0x02
```

Setting the polarity bit consists of reading the current value of the interrupt status control register, modifying the polarity bit, then writing the new value back to the interrupt status register. Listing 5-13 demonstrates the code to read the polarity bit.

```
/* read the register */
if (dio_get(INTSR, &val) < 0)
{
  return(-1);
}

/* update the polaity bit only */
if (pol == activehi)
  val |= POL_BIT;
else
  val &= ~POL_BIT;

/* write the register */
if (dio_set(INTSR, val) < 0)
{
  return(-1);
}
```

Listing 5-13

To get the status of the polarity bit, the interrupt status control register is read and bit 2 is tested to obtain the current state of the interrupt polarity. Listing 5-14 demonstrates code to read the polarity register.

```
/* read the current value of the status register */
if (dio_get(INTSR, &val) < 0)
{
  return(-1);
}

if ((val & POL_BIT) == 0)
  *pol == activelo;
else
  *pol = activehi;
```

Listing 5-14

Programming the Configuration Register

The configuration register programs the direction of the data IO ports. The DIO interface library contains functions to read and write the configuration register, as well as a utility function that provides the direction of an individual digital IO line.

The `dioconfig_t enum`

Each of the bits that controls a port has a bit defined in the `dioconfig_t enum`. These defines are used to program the data port directions. Listing 5-15 shows the `dioconfig_t` data type. These defines directly coincide with the bits in the config register to program the port directions.

```
typedef enum dioconfig_t
{
  porta_in  = 0x10, /* port a configured as input lines */
  portb_in  = 0x02, /* port b configured as input lines */
  portcl_in = 0x01, /* port c low configured as input lines */
  portch_in = 0x08, /* port c high configured as input lines */
} dioconfig_t;
```

Listing 5-15

The `dio_set_config` and `dio_get_config` Functions

The `dio_set_config` and `dio_get_config` functions are used to read and write the config register. The config parameter represents a bitwise ORed value of the `dioconfig_t` data type.

```
#include <dioif.h>

int32_t dio_set_config(dio_config_t config)
int32_t dio_get_config(dio_config_t* config)
```

Here, `dio_set_config` sets the direction of the digital ports, and it returns 0 on success or –1 on failure.

Listing 5-16 represents the code to write the config bits, the parameter passed directly through to the DIO device driver.

```
newcfg = (uint32_t) cfg;
newcfg |= 0x80;  /* PCI-DIO24 needs high bit set */
```

```
if (dio_set(CONFIG, cfg) < 0)
{
   return(-1);
}
```

Listing 5-16

In the code below, `dio_get_config` returns the direction of the digital ports, and it returns 0 on success or –1 on failure. Listing 5-16 demonstrates the code to read the `config` register.

```
if (dio_get(CONFIG, (int32_t*)cfg) < 0)
{
   return(-1);
}
```

Listing 5-16

As with the config write, the config register is read and the result passed directly back to the calling function.

The `diodirection_t enum`

Each digital IO line is programmed to be either input or output; the `diodirection_t enum` defines a digital line state as either lineout or linein. Listing 5-17 demonstrates the `diodirection_t enum`.

```
typedef enum diodirection_t
{
   lineout = 0,
   linein = 1,
} diodirection_t;
```

Listing 5-17

This `enum` is used as output for the `dio_get_direction` function.

The `dio_get_direction` Function

The function `dio_get_direction` returns the direction of the specified line.

```
#include <dioif.h>
int32_t dio_get_direction(dioline_t line, diodirection_t* dir)
```

The implementation of `dio_get_direction` is straightforward. The value of the configuration register is necessary so we can check the port direction bits. Listing 5-18 shows the code to read the config register.

```
/* read the configuration register */
if (dio_get_config(&direction) < 0)
{
  return(-1);
}
```

Listing 5-18

Once we have a valid copy of the configuration register, it is necessary to determine which port direction needs to be checked. The port direction bit is determined by the input digital line number. The digital line corresponds to a specific port bit which is determined in a case statement.

```
/*
** check the direction of the request lines port configuration
** bit
*/
switch (line)
{
case line0:
case line1:
case line2:
case line3:
case line4:
case line5:
case line6:
case line7:
  value = (((direction & porta_in) != 0) ? linein : lineout);
  break;

case line8:
case line9:
case line10:
case line11:
```

```
case line12:
case line13:
case line14:
case line15:
  value = (((direction & portb_in) != 0) ? linein : lineout);
  break;

case line16:
case line17:
case line18:
case line19:
  value = (((direction & portcl_in) != 0) ? linein : lineout);
  break;

case line20:
case line21:
case line22:
case line23:
  value = (((direction & portch_in) != 0) ? linein : lineout);
  break;
}
```

Listing 5-19

Lines 0 through 7 correspond to port A, lines 8 through 15 correspond to port B, lines 16 through 19 to port C Low and lines 20 through 23 to Port C High. Once we have the correct port register value, it just needs to be checked to see if it is set or not. If the value is 0, the port is programmed as an output, and otherwise the port is programmed as an input. Listing 5-20 shows the code to set the correct direction of the digital line.

```
if (value == 0)
  *val = lineout;
else
  *val = linein;
```

Listing 5-20

The `dio_get_direction` function returns 0 on success or –1 on failure.

Summary

In this chapter we've connected the pieces between the DIO device driver, which resides in the kernel, with a shared library residing in user space. As previously described, a system programmer is now provided with an intuitive C interface that can be programmed to without worrying about the bits, bytes and nibbles of driver programming and operating system internals.

In the upcoming chapters, we will continue to build on this to provide an even higher level interface, so we can monitor the PCI-DIO24 digital lines over the Internet and via the World Wide Web.

Daemons

Overview

A feature of Internet appliances is that they can be monitored and configured via the Internet. This chapter begins a series of chapters that focuses on implementing Internet access to the DIO Internet appliance through Internet interfaces and protocols, such as sockets, SSL, and Java Server Pages. This chapter shows how to implement a socket-based daemon and a custom protocol used by remote clients to communicate with the DIO appliance. Topics covered include:

- introduction to TCP/IP
- sockets
- digital IO protocol
- DIOD server daemon

Introduction to TCP/IP

TCP/IP is a protocol used by computers to exchange information over a network. TCP/IP development was funded by the Department of Defense (DoD) in the 1970s, in order to provide a method to share research information for the DoD. Since that time, TCP/IP has grown to become the major protocol of the Internet. One of the major benefits of TCP/IP is that it provides a defined method for a heterogeneous system to exchange information.

Before jumping into the details of TCP/IP, let's look at a few definitions and background information. Each computer, or endpoint, connected to a network is referred to as a host. The TCP/IP protocol defines the set of rules for

the process of transferring data between hosts. Because of the complexity of the protocol, it is defined in layers. TCP/IP consists of four layers, each responsible for a different aspect of the communication and its own protocol. These combined layers are commonly referred to as the TCP/IP stack.

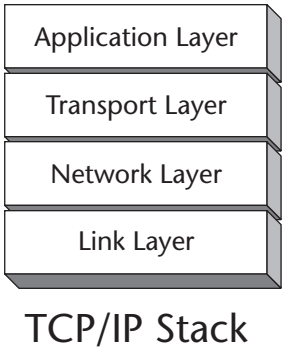
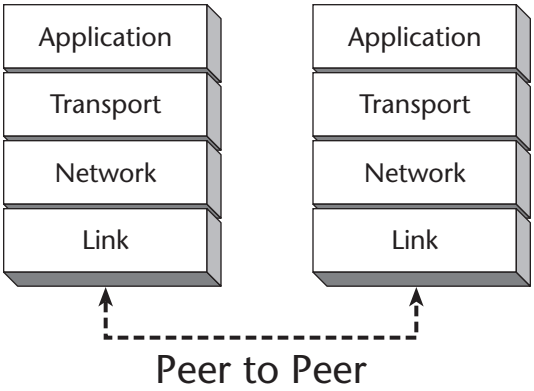


Figure 6-1. TCP/IP Stack



Each layer on the transmitting host has a peer-to-peer relationship with the receiving host. Every layer is responsible for a specific aspect of communication, such as physical transfer of data, addressing, routing, flow control, and data organization/reorganization.

Figure 6-2. Peer-to-Peer Relationship

The following sections describe each of the four TCP/IP layers in more detail.

Link Layer

At the bottom of the TCP/IP stack is the Link Layer. It handles the physical aspects of data transfer and provides a reliable, error-free transmissions medium. The link layer is comprised of hardware and software represented by the network interface controller (NIC) and the device driver in the operating system. Each NIC is independently addressable, using a unique hardware address.

There are numerous hardware implementations of the Link Layer, which include, but are not limited to, Ethernet, token ring, FDDI, and serial lines. Each different hardware implementation uses an independent addressing scheme. The Link Layer provides an independent mechanism to send higher layer protocol data and to resolve higher layer protocol addresses.

Network Layer

The network layer of the TCP/IP stack is implemented using the Internet Protocol (IP). IP is a connectionless, unreliable protocol. It is connectionless because there is no state information maintained for successive packets and unreliable, because there is no guarantee that a packet arrives at its destination. The purpose of the IP layer is to provide a constant addressing scheme for sending and receiving data packets and a mechanism to map between IP addressing and the underlying Link Layer addressing.

In the previous section, we mentioned that the Link Layer consists of a variety of different hardware addressing schemes. The IP protocol defines a consistent addressing method, the IP address, and protocols to map IP addresses to the underlying hardware addresses. These protocols are known as Address Resolution Protocol (ARP), Reverse Address Resolution Protocol (RARP), and Internet Control Message Protocol (ICMP).

IP Addressing

An IP address is a unique 32-bit number used for addressing IP packets. An IP address is written as four octets—i.e., 192.10.0.1.

There are five different classes of IP addresses: Class A, Class B, Class C, Class D, and Class E. Each class is distinguished by the upper bits of the IP address.

The different classes are used to distribute different numbers of bits between the network ID and the host ID.

Three classes of addresses are used by the IP layer: unicast, multicast, and broadcast. Unicast addresses are destined for a single host. Multicast addresses are destined for a specific group of hosts. Broadcast addresses are destined for all hosts on a given network.

ARP

ARP is the mechanism that maps the Link Layer hardware address to the host IP address. When a packet is being sent to another IP address, the

mapping between the IP address and the hardware address must be resolved. The ARP protocol sends a broadcast request to the network with the destination IP address. Each host receives the packet and checks to see if the request IP address matches its own IP address. If the IP address does not match, the packet is ignored. If the IP address does match, the host responds to it with its IP address and hardware address.

RARP

RARP is the protocol used by a diskless host to determine its IP address. When a host boots, it typically reads its IP address from a configuration file contained on the host's local disk. If the host booting is diskless, it will send an RARP request to the network, asking for someone to reply with its IP address.

ICMP

ICMP is used to communicate error messages and special conditions. ICMP messages may be acted upon by either the IP layer or upper layers of the TCP/IP protocol.

Transport Layer

The transport layer provides the method for addressing network data to a specific application. The TCP/IP stack contains two protocols for sending packets, Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). Regardless of which protocol being used, TCP or UDP, both the client and server machines must agree on the address, or port, used to transfer data.

Ports

The main mechanism to address network data to a specific application is a port. A port is a 16-bit integer. Reserved ports are listed in the `/etc/services` file.

UDP

UDP provides a connectionless, unreliable protocol. A UDP transfer consists of the sending of one packet without the overhead of establishing a connection or verification that the data has successfully reached its destination.

TCP

TCP provides a reliable, connection-oriented protocol. Connection-oriented means that two applications must establish a TCP connection before they begin exchanging data. TCP provides reliability by breaking each transfer up into manageable segments and verifying that each segment arrives correctly at its destination. In addition to the verification of data, TCP consists of flow control, so a destination host does not lose data due to lack of buffer space.

Application Layer

The application layer consists of network-aware applications that send and receive packets, using UDP and TCP ports. The application layer is not specifically a layer of the TCP/IP stack but a customer of the TCP/IP stack. As part of the application layer, a user will find services that provide network file and print sharing services, name resolution services, and user interfaces.

Socket System Calls

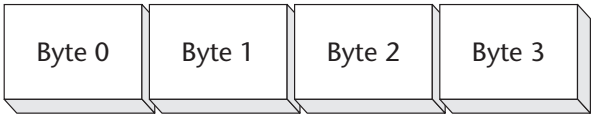
The programming interface for developing socket-based solutions using FreeBSD is sockets. Socket-based applications typically are written to solve networked applications. Networked applications are typically written as two interacting components, client and server. A server handles requests from a number of clients. A client, or application, makes requests to a server. An example of the client-server model is the World Wide Web. Your computer browser is a client application; web sites, or the computers that host the web sites, are the servers.

Implementation of a server consists of establishing a listening socket, waiting for a client connection, accepting the client request, performing the client request, and closing the connection. The DIO daemon developed later in this chapter is an example of a socket-based server.

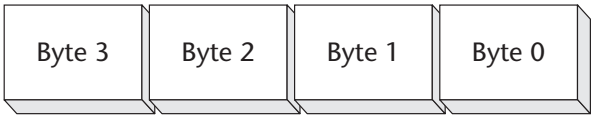
A client creates a connection socket, sends its request to the server, processes any received data, and closes the connection. Each server may process requests from many clients.

Network Byte Order

Any application that transfers data from one computer to another computer must be concerned with how the data is read. Different processors represent data in different formats. The two most widely accepted ways of representing data are known as big endian and little endian. Big endian systems store the most significant byte first, and little endian systems store the least significant byte first. For socket applications, the standard method of data transfer is big endian, also known as the network byte order.



Big Endian



Little Endian

The socket system call interface provides functions to convert shorts and longs between host and network byte order. Each of these calls is described below.

The `htonl` System Call

The `htonl` system call converts a host longword (32 bits) to network byte order.

```
#include <sys/param.h>
u_long htonl(u_long hostlong);
```

The return value is the converted long word.

The `htons` System Call

The `htons` system call converts a host short word (16 bits) to network byte order.

```
#include <sys/param.h>
u_short htons(u_short hostshort);
```

The return value is the converted short word.

The `ntohl` System Call

The `ntohl` system call converts a network longword (32 bits) to host byte order.

```
#include <sys/param.h>
u_long ntohl(u_long netlong);
```

The return value is the converted long word.

The `ntohs` System Call

The `ntohs` system call converts a host short word (16 bits) to host byte order.

```
#include <sys/param.h>
u_short ntohs(u_short netshort);
```

The return value is the converted short word.

Sockets

FreeBSD provides a set of system calls used for socket communications. In this section, the socket system calls are described and categorized based on their use. Subsequent sections demonstrate usage of the socket system calls.

Connection Initiation

Before data is transferred between a client and server, a connection must be established. Socket connections are created by calling socket system calls. Socket system calls that create a connection are the same for both clients and servers. In the following sections, the sockets system calls used for creating sockets and establishing a socket connection are presented.

The `socket` System Call

The basic means of communication over a network is a socket, which is an end point of communication. Sockets are created by calling the socket system call.

```
#include <sys/types.h>
#include <socket.h>

int socket(int domain, int type, int protocol);
```

Here, `socket` returns a descriptor to be used with subsequent socket calls. Socket takes three parameters. The first is the domain, which selects the domain in which communication will take place; there are many domains defined for sockets. Our implementation of the diod daemon is Internet-based, so PF_INET is the domain that represents the Internet protocol family.

The second parameter, type, defines the semantics of the connection. The defined types are listed in the following table.

Type	Notes
`SOCK_STREAM`	Sequenced reliable full duplex byte stream
`SOCK_DGRAM`	Connectionless unreliable communication
`SOCK_RAW`	Interface to internal Internet protocols
`SOCK_RDM`	Reliable delivery packet
`SOCK_SEQPACKET`	Sequenced packet stream

The third parameter, protocol, is used to select the protocol for the connection. Most sockets only support one protocol for each specified connection; the protocol parameter is typically 0.

Socket returns a valid socket if successful, or –1 if an error occurs.

The **`bind`** System Call

Before data can be transmitted using a socket, the socket must be associated with a service port. The `bind` system call is used to bind a socket to a service port.

```
#include <sys/types.h>
#include <socket.h>

int bind(int s, struct sockaddr* addr, socketlen_t len);
```

The `bind` system call takes three parameters. The first parameter is the socket descriptor returned by the socket system call. The second parameter is a protocol specific address. The diod is Internet-based in the PF_INET protocol; the parameter is a `sockaddr_in` structure. The `sockaddr_in` structure is defined in `/usr/include/netinet/in.h`.

```
struct sockaddr_in {
    u_char  sin_len;
    u_char  sin_family;
    u_short sin_port;
    struct  in_addr sin_addr;
    char    sin_zero[ 8] ;
} ;
```

The `sin_len` element represents the length of the structure. The `sin_family` describes the address family, AF_INET. The `sin_port` is the 16-bit port number to bind. The `sin_addr` element is a 32-bit IP address that represents the host to connect. The third parameter is the length of the structure passed as the sockaddr parameter. Bind returns 0 if successful and –1 if there is an error.

The `close` System Call

When a network service is shut down, proper cleanup is recommended. The socket descriptor should be closed, using the `close` system call, the same as a file descriptor.

```
#include <unistd.h>

int close(int fd);
```

The descriptor passed is the socket descriptor returned by the socket system call. The close system call terminates the socket. Close returns 0 on success, –1 on error.

Server

The DIO server we are developing is a connection-based server. A connection-based application is similar to file IO. A connection is opened with another process and that connection remains with the same process and host for the remainder of the transfer operation. The DIO server operations consist of requests to perform data acquisition operations. The types of requests will be discussed in more detail in the protocol section. The remainder of this section focuses on the socket calls for creating socket connections.

The `listen` System Call

Once the server has successfully created a socket, it must designate itself willing to listen for incoming requests. Listening on a socket is performed by calling the `listen` system call.

```
#include <sys/types.h>
#include <socket.h>

int listen(int s, int backlog);
```

The first parameter is the socket descriptor returned by the socket system call. The second parameter, backlog, specifies the maximum number of outstanding connections that may be queued by the server. Listen returns 0 on success or –1 on error.

The `accept` System Call

After the port is configured to listen, connections may be accepted. The `accept` system call will block until a connection is requested by a client or a signal is received.

```
#include <sys/types.h>
#include <socket.h>

int accept(int s, struct sockaddr* addr, socketlen_t len);
```

The `accept` system call takes the first connection on the connection queue and creates another socket.

Accept returns a nonnegative value on success or –1 on error.

Client

Because the DIO server is a connection-based server, a client requiring services from the DIO server must create and connect to the socket provided. This section describes the socket system calls used to connect to the socket provided by the DIO server.

The `connect` System Call

```
#include <sys/types.h>
#include <sys/socket.h>

int connect(int s, struct sockaddr* serv, socketlen_t len);
```

The first parameter is the socket descriptor returned by the socket system call. The second parameter is a protocol-specific parameter. For our use, it contains the IP address of the server and the socket number of the service that the client is requesting. The last parameter is the length of the structure passed in as the second parameter. Connect returns 0 on success or –1 on error.

Connection Data Transfer

After the connection is established between the client and the server, data is exchanged using the `send` and `recv` system calls.

The `send` System Call

Data is sent through a socket using the `send` system call.

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int s, void* msg, size_t len, int flags);
```

The first parameter is the socket descriptor returned by the `socket` system call. The second parameter is a pointer to the data being sent. The third parameter is the length in bytes of the data being sent. The last parameter contains flags used for data transfer.

Flag	Description
`MSG_OOB`	Out of band data
`MSG_PEEK`	Look at incoming message
`MSG_DONTROUTE`	Send message direct
`MSG_EOR`	Packet completes record
`MSG_EOF`	Packet completes transmission

`Send` returns the number of characters sent on success or –1 on error.

The `recv` System Call

Data is retrieved on a socket by calling the `recv` system call.

```
#include <sys/types.h>
#include <sys/socket.h>

int recv(int s, void* msg, size_t len, int flags);
```

The first parameter is the socket descriptor returned by the socket system call. The second parameter is the buffer to store the data. The third parameter contains the number of bytes to read from the socket. The last parameter contains one or more flags.

Flag	Description
MSG_OOB	Process out of band data
MSG_PEEK	The data is copied into the buffer but is not removed from the input queue.
MSG_WAITALL	Wait for a complete request or error

Recv returns the number of bytes received or –1 on error.

Connectionless Data Transfer

An alternative to a connection-oriented server is a connectionless server. A connectionless server is different, because a connection is not established prior to transferring data. In a connectionless protocol, the server blocks on a port until data is received from a client.

The recvfrom System Call

The recvfrom() system call receives a message from a socket and captures the address from which the data was sent. Unlike the recv() call, which can only be used on a connected stream socket or bound datagram socket, recvfrom() can be used to receive data on a socket, whether or not it is connected. If no messages are available at the socket, the recvfrom() call blocks until a message arrives, unless the socket is nonblocking .

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct
sockaddr *from, socklen_t *fromlen);
```

If a socket is nonblocking, –1 is returned and the external variable errno is set to EWOULDBLOCK.

The `sendto` System Call

`Sendto` is called to transmit data to a socket. The `sendto()` call tranmits a message through a connection-oriented or connectionless socket. If it's a connectionless socket, the message is sent to the address specified by *to*. If *s* is a connection-oriented socket, the *to* and *tolen* parameters are ignored.

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t sendto(int s, const void *msg, size_t len, int flags,
const struct sockaddr *to, socklen_t tolen);
```

The `sendto` system call returns the number of characters transmitted or –1, if there is an error.

The DIO Daemon

Now that we've introduced the socket system calls, we will use them to create the DIO daemon. The DIO daemon is a connection-oriented, socket-based daemon used to handle requests related to the PCI-DIO24 data acquisition board. The DIO daemon, in connection with a simple protocol developed in this section, is used to provide a client with a simple mechanism to query digital IO lines on our server.

Protocol

Before delving into the details of the digital IO daemon, we will define the protocol used between the client and server. The protocol consists of the operations that are performed by the digital IO daemon and the format of the requests. The complete protocol consists of an enum that defines the set of operations that are provided by the DIO daemon and a structure used to transfer data between the DIO server and client.

The `operation_t enum`

The DIO daemon performs three operations: read the direction of a line, read a line, and write a line. These operations are defined by the `enum operation_t`.

```
/*
**  define the operations for the digital io protocol
*/
typedef enum operation_t
{
   read_line = 0,    /* read a value to a digital IO line   */
   write_line,   /* write a value to a digital IO line*/
   read_direction,   /* request the direction of an IO line */
}  operation_t;
```

The DIO interface library has more capabilities than these three operations. For the sake of simplicity, we will only provide the capabilities to query the state of the lines and read or write the lines, based on their current state.

Providing the capability to configure the PCI-DIO24 controller by the server could create race conditions. For example, one client could be reading or writing a digital line, while another client is reconfiguring the digital IO lines. Additionally, a client could reconfigure lines with equipment attached, causing damage to the controller or equipment connected to the controller.

The `diod_request` Structure

In the previous section, we defined the operation performed by the DIO daemon; now we can define the format of the requests and responses. The `diod_request_t` type contains the diod server request.

```
typedef struct diod_request_t
{
   int32_t   size;       /* size of this structure          */
   uint32_t  magic;  /* predefined magic number             */
   int32_t   sequence;   /* sequence sent by server         */
   operation_t operation;/* requested service               */
   int32_t   line;       /* digital line for service request*/
   int32_t   value;  /* read or write value                 */
}  diod_request_t;
```

The first two elements, size and magic, are used to verify that the data received contains a valid request. The size element contains the size of `diod_request_t`. The magic element contains a fixed value unique to a `diod_request_t` structure.

The sequence element contains a nonzero value, passed from the server to client. This field is used by the client to verify that the request was received and handled by the server.

The operation element contains the type of request. Valid operations were covered in the previous section.

The last two elements contain the data specific to the DIO request. The line element contains the digital line where the operation is to be performed.

The last field is the value; if this is a write operation, this is the value to be written; if it is a read operation, this is the value read from the DIO line.

Server

This section describes the details of the diod server. The server is broken down into a few basic functions that initialize the daemon, read, process, and return client requests. The diod daemon runs as a daemon; in addition to the functions described here, the daemon `init_daemon` and `handle_sigcld` function developed in Chapter 2 are used by the diod daemon.

The `process_request` Function

Process request is a utility function used to handle each client request. Once a connect is made by a client, the `process_request` function reads the request, validates that the data read contains a valid diod request, calls the appropriate DIO interface library function, packages the results, and returns them to the client.

```
int32_t process_request(int32_t sockfd)
{
    int32_t      n;
    diod_request_t req;
    diod_request_t* preq = &req;

    /*
    **     a client connection has been made, read
    **     and verify we have a valid request
    */
    n = recv(sockfd, preq, sizeof(diod_request_t), 0);
    if (n != sizeof(diod_request_t))
```

```
    {
        return(-1);
    }

    /* verify the data and contents of the request */
    if (preq->magic != DLOG_MAGIC)
    {
        printf("incorrect magic number, n = %#x\n", preq->magic);
        return(-1);
    }

    if (preq->size != sizeof(diod_request_t))
    {
        printf("invalid request\n");
        return(-1);
    }

    /*
    **     this is a valid request, parse and handle it
    */
    switch(preq->operation)
    {
    case read_line:
        dio_get_line((dioline_t)preq->line, (diolinestate_t
                              *)&preq->value);
        break;

    case write_line:
        dio_set_line((dioline_t)preq->line, (diolinestate_t)preq-
                              >value);
        break;

    case read_direction:
        dio_get_direction((dioline_t)preq->line,
(diodirection_t*)&preq->value);
        break;
    }

    /*
    **     set a non-zero value so we can verify this request has
    **     been serviced
    */
    preq->sequence = sockfd;
```

```
    /*
    **     the request is completed send the results back to the
    **     client process
    */
    send(sockfd, preq, sizeof(diod_request_t), 0);

    return(0);
}
```

The process request function is called after the socket connection is established. The purpose of `process_request` is threefold: to read the request from the client via the `recv` system call; if the request is valid, perform the requested operation via the DIO interface library; return the results to the calling client via the send system call.

The `init_dio` Function

The `init_dio` function handles the details of programming the PCI-DIO24 controller for our specific application. Our application consists of polling the digital IO lines; the `init_dio` function programs the interrupt, then configures the digital IO ports to meet our design requirements.

```
void
init_dio()
{
    /*
    ** before setting up the socket, program the board to suit
the
    ** needs of our application.
    **
    ** for this application we will be polling only, disable the
    ** interrupts
    */
    dio_set_int(disable);
    dio_set_pciint(disable);

    /* set the digital lines, ports a and cl are input, b and ch
are output */
    dio_set_config((dioconfig_t)(porta_in | portcl_in));
}
```

The calls to `dio_set_int` and `dio_set_pciint` disable the PCI-DIO24 controller interrupt. After the interrupt is disabled, the Digital IO ports are

programmed. Digital IO ports A and CL are programmed as inputs; B and CH are left as outputs. This configuration gives us 12 digital input lines and 12 digital output lines.

The `main` Function

The main function is responsible for initialization, creating the socket connection, and handling client requests. The main program calls the `init_daemon` function created in Chapter 2 to create a daemon process. After returning from the `init_daemon` function, the `init_dio` function is called; `init_dio` is responsible for programming the PCI-DIO24 controller for our application.

With the initialization complete, the main program focuses on setting up the connection. The socket is created and bound for DIO client requests. Once the socket is successfully set up, the diod daemon listens for a client request.

We are now online and waiting for client requests; each request that arrives is accepted, a new process is created, and the request is handed off to the child process to be completed. The parent process goes back and waits for the next request.

The child process really only needs to hand off the request to the `process_request` function and exit. `Process_request` takes the socket file descriptor for the socket, so it reads the client request, then sends the response back.

```
int main(int argc, char** argv)
{
    int fd                  = 0;
    int32_t sockfd   = -1;
    int32_t stat     = 0;
    struct  sockaddr_in  server_addr;

    /* turn ourselves into a daemon */
    init_daemon();

    /* handle the controller initialization */
    init_dio();

    /*
    **     create a TCP/IP socket
    */
```

```
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
    {
        printf("unable to create socket\n");
        exit(-1);
    }

    /*
    **     bind the socket to our Internet address and service port
    */
    bzero(&server_addr, sizeof(struct sockaddr_in));
    server_addr.sin_family       = AF_INET;
    server_addr.sin_addr.s_addr      = htonl(INADDR_ANY);
    server_addr.sin_port         = htons(DIOD_TCP_PORT);

    stat = bind(sockfd, (struct sockaddr *) &server_addr, size-
of(struct sockaddr_in));
    if (stat < 0)
    {
        exit(-1);
    }

    /*
    **     wait for client requests
    */
    listen(sockfd, 5);

    while (1)
    {
        int32_t child_pid= 0;
        int32_t sockfd_new   = 0;
        uint32_t client_len  = sizeof(struct sockaddr_in);
        struct  sockaddr_in  client_addr;

        bzero(&client_addr, sizeof(struct sockaddr_in));
        client_len   = sizeof(struct sockaddr_in);

        sockfd_new = accept(sockfd, (struct sockaddr
*)&client_addr, &client_len);
        if (sockfd_new < 0)
        {
            continue;
        }
```

```
        /*
        **  fork a child process to handle the request, then go
        **  back and wait for more
        */
        child_pid = fork();
        if (child_pid < 0)
        {
            exit(-1);
        }
        else if (child_pid == 0)
        {
            close(sockfd);

            /*
            **         handle the client request
            */
            process_request(sockfd_new);
            exit(0);
        }
    }

    dio_shutdown();


    exit(0);
}
```

The main function will run forever, creating child processes to handle client requests.

Summary

In this chapter we have covered the basics of TCP/IP and details of socket system calls, and developed a connection-oriented socket daemon to handle network requests for DIO lines. Using the code provided in this chapter, along with the code in the previous three chapters, we have developed a data acquisition appliance that can be monitored over the Internet.

In the upcoming chapters, we will change our focus from the system's programming details to developing application software that allows the DIO appliance to be monitored and configured over the Internet.

Remote Management

Overview

Many Internet appliances are installed in a remote location that is not easily accessible. One feature often required by an Internet appliance is remote management. Remote management is often implemented using telnet, rlogin and rsh. Although these tools are reliable, they are insecure. The Secure Shell (SSH) was developed to provide remote access to systems that use encryption to pass data between systems for security. The standard FreeBSD installation contains an open source implementation of SSH, called OpenSSH.

While SSH can protect an Internet appliance against attacks such as IP spoofing and packet snooping, it cannot protect the system once someone has logon access. In order to provide additional security, we will develop a custom shell that only allows a user to access features related to the DIO appliance. This limits the damage a cracker can do, if access is gained.

In this chapter, we will cover topics including:

- SSH
- Configuring SSH
- Creating a management logon account
- Developing a custom DIO management shell

Using Secure Shell (SSH)

Remote configuration is not a new concept to Internet appliances. For years, systems engineers have provided remote access using rsh, rlogin and telnet. These tools provide methods for remote configuration but are prone to being

cracked because the data transmitted, including usernames and passwords, is not encrypted. Rsh and rlogin can provide minimal security by being configured to provide access only to trusted hosts; however, IP addresses can be spoofed, making one host appear as another.

SSH provides a method for allowing remote access to another machine using encryption. It also provides login capabilities, while encrypting the data exchanged between multiple systems. Transmitting encrypted data minimizes exposure to crackers snooping Internet connections.

SSH is implemented using the client-server module—i.e., the SSH server listens for SSH connections from an SSH client. The SSH client and server exchange two keys to establish a connection. One key is the public key to which everyone has access, and the other is the private key. Both keys are required for successful data exchange. Data that is encrypted using the public key can only be decrypted using the private key.

Getting SSH

OpenSSH is part of the standard distribution of FreeBSD. However, if you need a patch or want to run the latest version of OpenSSH, it is contained in the ports package distributed with FreeBSD. To update the installed version of OpenSSH, you must `cd` to the OpenSSH port directory, and then execute the standard commands for updating a port. The latest copy of OpenSSH can be retrieved by executing the following commands.

```
# cd /usr/ports/security/openssh
# make
# make install
# make clean
```

These commands make, install, and clean up the latest version of the OpenSSH FreeBSD software.

Running SSH

The SSH daemon, `sshd`, comes as part of the standard FreeBSD distribution. To run `sshd` at system start, the following lines must be added to the `/etc/rc.conf` file. The `rc.conf` file is used to provide custom startup configuration.

```
sshd_enable="YES"
```

The `rc.conf` file is a script file that contains variables used to customize system startup. This file is not executing as a script; it is included by other files to conditionalize system startup.

The `rc` script is a command script used to control reboots; `rc` reads the contents of the `rc.conf` script to determine the custom components for starting the system. A closer look at the `rc` script shows how the `re.conf` file is read.

```
if [  -r /etc/defaults/rc.conf ]; then
    . /etc/defaults/rc.conf
    source_rc_confs
elif [  -r /etc/rc.conf ]; then
    . /etc/rc.conf
fi
```

Looking through the `rc` script further, you will find code that shows the ssh daemon being started, conditional on the `sshd_enable` variable. If the `sshd_enable` variable is set to yes, the `rc` starts the ssh. The code in my `rc` script that does this is shown below.

```
case ${sshd_enable} in
[Yy][Ee][Ss])
    if [  -x ${sshd_program:-/usr/sbin/sshd} ]; then
        echo -n ' sshd';
        ${sshd_program:-/usr/sbin/sshd} ${sshd_flags}
    fi
    ;;
esac
```

Once the system boots, it is ready to accept SSH connections.

Configuring SSH

The standard file used to configure SSH is `/etc/ssh/sshd_config`. The `sshd_config` file is a simple configuration file, containing the file format with "*ValueName Value*" pairs using a simple space as a separator. A copy of a default `sshd_config` is contained in Listing 7-1.

```
# This is ssh server systemwide configuration file.
#
# $FreeBSD: src/crypto/openssh/sshd_config,v 1.4.2.5 2001/01/18
22:36:53 green Exp $

Port 22
#Protocol 2,1
#ListenAddress 0.0.0.0
#ListenAddress ::
HostKey /etc/ssh/ssh_host_key
HostDsaKey /etc/ssh/ssh_host_dsa_key
ServerKeyBits 768
LoginGraceTime 120
KeyRegenerationInterval 3600
PermitRootLogin no
# ConnectionsPerPeriod has been deprecated completely

# After 10 unauthenticated connections, refuse 30% of the new
# ones, and refuse any more than 60 total.
MaxStartups 10:30:60
# Don't read ~/.rhosts and ~/.shosts files
IgnoreRhosts yes
# Uncomment if you don't trust ~/.ssh/known_hosts for
RhostsRSAAuthentication
#IgnoreUserKnownHosts yes
StrictModes yes
X11Forwarding yes
X11DisplayOffset 10
PrintMotd yes
KeepAlive yes

# Logging
SyslogFacility AUTH
LogLevel INFO
#obsoletes QuietMode and FascistLogging

RhostsAuthentication no
#
# For this to work you will also need host keys in
/etc/ssh_known_hosts
RhostsRSAAuthentication no
#
RSAAuthentication yes
```

```
# To disable tunneled clear text passwords, change to no here!
PasswordAuthentication yes
PermitEmptyPasswords no
# Uncomment to disable s/key passwords
#SkeyAuthentication no
#KbdInteractiveAuthentication yes

# To change Kerberos options
#KerberosAuthentication no
#KerberosOrLocalPasswd yes
#AFSTokenPassing no
#KerberosTicketCleanup no

# Kerberos TGT Passing does only work with the AFS kaserver
#KerberosTgtPassing yes

CheckMail yes
#UseLogin no

# Uncomment if you want to enable sftp
#Subsystem  sftp    /usr/libexec/sftp-server
```

Listing 7-1

The `sshd_config` file options are described in the following sections.

Port 22

The Port option specifies the port number ssh daemon listens to for incoming connections.

ListenAddress 192.168.1.1

The ListenAddress options specifies the IP address of the interface network on which the ssh daemon server socket is bind. The default is 0.0.0.0.

HostKey /etc/ssh/ssh_host_key

The HostKey option specifies the location containing the private host key.

ServerKeyBits 1024

The ServerKeyBits options specifies how many bits to use in the server key. These bits are used to generate its RSA key when the daemon starts.

LoginGraceTime 600

The LoginGraceTime options specifies how long, in seconds, after a connection request the server will wait before disconnecting if the user has not successfully logged in.

KeyRegenerationInterval 3600

The KeyRegenerationInterval options specifies how long, in seconds, the server should wait before automatically regenerated its key.

PermitRootLogin no

The PermitRootLogin options specifies whether root can log in using ssh.

IgnoreRhosts yes

The IgnoreRhosts options specifies whether rhosts or shosts files should not be used in authentication. For security reasons it is recommended to no use rhosts or shosts files for authentication.

IgnoreUserKnownHosts yes

The IgnoreUserKnownHosts options specifies whether the ssh daemon should ignore the user's $HOME/.ssh/known_hosts during RhostsRSAAuthentication.

StrictModes yes

The StrictModes option specifies whether ssh should verify user's permissions in their home directory and rhosts files before accepting login.

X11Forwarding no

The X11Forwarding options specifies whether X11 forwarding should be enabled or not on this server. Since the DIO appliance runs without GUI installed, we can safely turn this option off.

PrintMotd yes

The PrintMotd option specifies whether the ssh daemon should print the contents of the /etc/motd file when a user logs in interactively.

SyslogFacility AUTH

The SyslogFacility option specifies the facility code used when logging messages from sshd. The facility specifies the subsystem that produced the message—in our case, AUTH.

LogLevel INFO

The LogLevel option specifies the level that is used when logging messages from sshd.

RhostsAuthentication no

The RhostsAuthentication option specifies whether sshd can try to use rhosts based authentication. Because rhosts authentication is insecure you shouldn't use this option.

RhostsRSAAuthentication no

The RhostsRSAAuthentication option specifies whether to try rhosts authentication in concert with RSA host authentication.

RSAAuthentication yes

The RSAAuthentication option specifies whether to try RSA authentication. This option must be set to yes for better security in your sessions.

PasswordAuthentication yes

The PasswordAuthentication option specifies whether we should use password-based authentication.

PermitEmptyPasswords no

The PermitEmptyPasswords option specifies whether the server allows logging in to accounts with a null password.

AllowUsers admin

The AllowUsers option specifies and controls which users can access ssh services.

Remote Administration Account

In order to restrict the operation allowed by a remote configuration, the only connection allowed remotely is to login via the dioadmin account. The dioadmin is a special account that runs a custom shell developed in this chapter. When the shell is exiting, the user is logged out.

```
dioadmin:*:1002:1002:DIO Administrator:/home/dioadmin:/bin/sh
```

The dioadmin account has its own UID and GID 1002. It is configured to run the Bourne Shell. When the Bourne Shell executes, it reads a customized file in the user's home directory called `.profile`. The dioadmin `.profile` file contains two lines:

```
/usr/local/bin/dioshell
exit
```

This effectively allows the user to run the dioshell. When dioadmin exits, the dioshellcontrol returns to the `.profile` file, and the dioadmin account is logged off.

The DIOShell

In the previous section we discussed SSH and created a new account to be used for the DIO administrator. The DIO administrator account is created with a custom shell, the DIOShell, which is limited to commands to DIO operations. The remainder of this chapter deals with the implementation of the DIOShell.

The `command_t` Structure

The DIOShell is implemented using a simple command parser. Every command is implemented as a command string, a command handler function, and a help string. The command handler function contains two arguments, the complete string entered by the user and the command specific string.

```
typedef void (*fptr) (char *, uint8_t *);

typedef struct
{
  char  *command;        /* string representing command      */
  fptr  functionptr;/* pointer to command implementation    */
  char  *helpstring;/* text help string                     */
} command_t;
```

Listing 7-2

Every command is implemented by a `command_t` structure.

The command Table

The command table represents all the available commands in the DIOShell. Each function is forward declared, so the command table may be created. The DIOShell is a table-driven utility. User input is parsed, and the first argument is matched to the commands contained in the command table.

If a command is matched, the command handler function is called to complete command processing.

```
void    readline_handler(char* args, uint8_t* p);
void    writeline_handler(char* args, uint8_t* p);
void    config_handler(char* args, uint8_t* p);
void    int_handler(char* args, uint8_t* p);
void    pciint_handler(char* args, uint8_t* p);
void    setpolarity_handler(char* args, uint8_t* p);
void    getpolarity_handler(char* args, uint8_t* p);
void    getdirection_handler(char* args, uint8_t* p);
void    quit_handler(char* args, uint8_t* p);
void    help_handler(char* args, uint8_t* p);

command_t       commands[ ] =
{
  "read",       readline_handler,       "[ 0-23] reads the spec-
ified digital IO line",
  "write",      writeline_handler,      "[ 0-23] [ 0 | 1] writes
the specified digital IO lines",
  "configin",   config_handler,         "[ A,B,CL,CH] [ in | out]
config the digital IO ports",
  "int",        int_handler,            "[ enable | disable]
enable or disable interrupts",
  "pciint",     pciint_handler,         "[ enable | disable]
enable or disable pci interrupts",
  "setpol",     setpolarity_handler,    "[ hi | lo] set the
interrupt polarity",
  "getpol",     getpolarity_handler,    "get the interrupt
polarity",
  "getdir",     getdirection_handler,   "read the direction of
the specified line",
  "quit",       quit_handler,           "exits program",
  "help",       help_handler,           "displays command
help",
  NULL,         0,                      NULL,           /*
terminating record */
};
```

Listing 7-3

The command table, `command`, in Listing 7-3 contains all the commands implemented by the DIOShell. Commands included are read line, write line, configure the ports, configure the interrupt, configure the PCI interrupt, set

and get the interrupt polarity. Each command is described in more detail in subsequent sections.

Since the DIOShell command is table driven, the last entry contains a terminating record. This is to denote the end of the table. Without the terminating record, we would need some other mechanism to determine when we've reached the end of the table and keep the command parsing loop from running rampant over process memory.

The `readline_handler` Function

The read handler function is used to read a digital IO line. Read line commands are entered using syntax

```
> read line
```

where `line` is a value between 0 and 23. `Read` will print the value of the requested line 0 or 1.

```
void
readline_handler(char* args, uint8_t* p)
{
  DIOline_t     line;
  DIOlinestate_t    state;

  sscanf(args, "%*s %d %d", &line, &state);

  /* check the parameters */
  if ((line < line0) || (line > line23))
  {
    printf("illegal line number\n");
    return;
  }

  if (dio_get_line(line, &state) == 0)
  {
    printf("line %d = %d\n", line, state);
  }
  else
  {
    printf("read line failed\n");
  }
}
```

Listing 7-4

Here, `readline_handler` parses the input provided by the user. If the input is valid, the `dio_get_line` function is called to read the state of the digital IO line.

The `writeline_handler` Function

The `writeline_handler` function is used to write a digital IO line. The `write line` command is entered using the following syntax,

```
> write line value
```

where `line` is a value between 0 and 23, and `value` is either 0 or 1. `Write` will set the requested line output to the value input by the user.

```
void
writeline_handler(char* args, uint8_t* p)
{
  DIOline_t     line;
  DIOlinestate_t    state;

  sscanf(args, "%*s %d %d", &line, &state);

  /* check the parameters */
  if ((line < line0) || (line > line23))
  {
    printf("illegal line number\n");
    return;
  }

  /* check the parameters */
  if ((state != set) && (state != clear))
  {
    printf("illegal state\n");
    return;
  }

  if (dio_set_line(line, state) != 0)
  {
    printf("write line failed\n");
  }
}
```

Listing 7-5

The write handler function uses `sscanf` to parse the user's input. If the command arguments are correct, `dio_set_line` is called to set the requested lines output.

The `config_handler` Function

The `config_handler` function is used to change the port configurations. The `config` command is entered using the following syntax:

```
> configin port
```

where ports is any combination of A, B, CL and CH. The specified ports are configured as digital inputs.

```
void
config_handler(char* args, uint8_t* p)
{
  char  port[ 32] ;
  DIOconfig_t cfg = 0;

  sscanf(args, "%*s %s", port);

  /* scan the string looking for ports */
  if ((strstr(port, "A") != NULL) || (strstr(port, "a") != NULL))
  {
    cfg |= porta_in;
  }

  if ((strstr(port, "B") != NULL) || (strstr(port, "b") != NULL))
  {
    cfg |= portb_in;
  }

  if ((strstr(port, "CL") != NULL) || (strstr(port, "cl") !=
        NULL))
  {
    cfg |= portcl_in;
  }

  if ((strstr(port, "CH") != NULL) || (strstr(port, "ch") !=
        NULL))
  {
    cfg |= portch_in;
```

```
  }

  /* we've built the cfg mask write it to the register */
  if (dio_set_config(cfg) != 0)
  {
    printf("config failed\n");
  }
}
```

Listing 7-6

The `config` function uses `sscanf` to parse the user's input. The argument to the `config` in command can take multiple ports represented by their letters, A, B, CL, and CH. The port letters can be contained in any order in the input string. The only requirement is that the ports are listed together, with no spaces. Once the ports are determined, the `dio_set_config` function is called to configure the input ports.

The `int_handler` Function

The `int_handler` function is used to enable or disable interrupts. The `int` command is entered using the following syntax:

```
> int value
```

where value is either string enable or disable.

```
void
int_handler(char* args, uint8_t* p)
{
  char state[32];
  DIOintstate_t s;

  sscanf(args, "%*s %s", state);

  if ((strcmp(state, "enable") == 0) || (strcmp(state, "ENABLE")
                                    == 0))
  {
    s = enable;
  }
  else if ((strcmp(state, "disable") == 0) || (strcmp(state,
                                    "DISABLE") == 0))
  {
    s = disable;
```

```
  }
  else
  {
    printf("illegal state\n");
    return;
  }

  if (dio_set_int(s) != 0)
  {
    printf("set int failed\n");
                                    }
}
```

Listing 7-7

The `int_handler` function parses the user input looking for either the enable or disable string to determine the interrupt setting. If the user input is correct, `dio_set_int` is called to set the interrupt state.

The `pciint_handler` Function

The `pciint_handler` function is used to set `pci` interrupts. The `pciint` command is entered using the following syntax,

```
> pciint value
```

where `value` is either the string enable or disable.

```
void
pciint_handler(char* args, uint8_t* p)
{
  char state[ 32];
  DIOintstate_t s;

  sscanf(args, "%*s %s", state);

  if ((strcmp(state, "enable") == 0) || (strcmp(state, "ENABLE")
                                    == 0))
  {
    s = enable;
  }
  else if ((strcmp(state, "disable") == 0) || (strcmp(state,
                                    "DISABLE") == 0))
  {
    s = disable;
```

```
  }
  else
  {
    printf("illegal state\n");
    return;
  }

  if (dio_set_pciint(s) != 0)
  {
    printf("set pciint failed\n");
  }
}
```

Listing 7-8

The `pciint_handler` function parses the user input looking for either the enable or disable string to determine the interrupt setting. If the user input is correct, `dio_set_pciint` is called to set the interrupt state.

The `setpolarity_handler` Function

The `setpolarity_handler` function is used to set the interrupt polarity. The set polarity command is entered using the following syntax:

```
> setpol value
```

where `value` is either high or low.

```
void
setpolarity_handler(char* args, uint8_t* p)
{
  char state[ 32];
  DIOpolarity_t polarity;

  sscanf(args, "%*s %s", state);

  if ((strcmp(state, "hi") == 0) || (strcmp(state, "HI")) == 0)
  {
    polarity = activehi;
  }
  else if ((strcmp(state, "lo") == 0) || (strcmp(state, "LO"))
                              == 0)
  {
    polarity = activelo;
  }
```

```
  else
  {
    printf("invalid polarity\n");
    return;
  }

  if (dio_set_polarity(polarity) != 0)
  {
    printf("error writing polarity\n");
  }
}
```

Listing 7-9

The `int_handler` function parses the user input looking for either the high or low string to determine the interrupt polarity setting. If the user input is correct, `dio_set_polarity` is called to set the interrupt polarity.

The `getpolarity_handler` Function

The `getpolarity_handler` function is used to read the interrupt polarity. The get polarity command is entered using the following syntax:

```
> getpol
```

`Getpol` takes "not" arguments.

```
void
getpolarity_handler(char* args, uint8_t* p)
{
  DIOpolarity_t polarity;

  if (dio_get_polarity(&polarity) == 0)
  {
    printf("pol = %d\n", polarity);
  }
  else
  {
    printf("error reading polarity\n");
  }
}
```

Listing 7-10

The `getpolarity_handler` calls the `dio_get_polarity` function; if the call is successful, the interrupt polarity is displayed by the DIO shell.

The `getdirection_handler` Function

The `getdirection_handler` function is used to read the direction of a digital IO line. The get direction command is entered using the following syntax:

```
> getdir line

where line is a value between 0 and 23.
void
getdirection_handler(char* args, uint8_t* p)
{
  DIOline_t  line;
  DIOdirection_t direction;

  sscanf(args, "%*s %d", &line);

  /* check the parameters */
  if ((line < line0) || (line > line23))
  {
    printf("illegal line number\n");
    return;
  }

  if (dio_get_direction(line, &direction) == 0)
  {
    printf("line %d %s\n", direction, (direction == linein ?
                                  "in" : "out"));
  }
  else
  {
    printf("error reading direction\n");
  }
}
```

Listing 7-11

The `getdirection_handler` function parses the user input looking for the line number. After determining the line, `dio_get_direction` is called, and the digital IO line direction is displayed, either in or out, depending on the port's configured direction.

The `quit_handler` Function

The `quit_handler` function is used to exit the shell. The `quit` command is entered using the following syntax.

```
> quit
```

`Quit` takes no arguments.

```
void
quit_handler(char* args, uint8_t* p)
{
  /*
  **     since this is the program exit we must free the user
  **     buffer malloced in the main program
  */
  free(args);
  exit(0);
}
```

Listing 7-12

The `quit` command handler releases resources and exits the shell.

The `help_handler` Function

The `help_handler` function is used to display command help. The `help` command is entered using the following syntax.

```
> help
```

`Help` takes no arguments.

```
void
help_handler(char* args, uint8_t* p)
{
  command_t*   cmdptr;

  cmdptr = &commands[0];
  while (cmdptr->command != NULL)
  {
    printf("\t%s %s\n", cmdptr->command, cmdptr->helpstring);
    cmdptr++;
  }
}
```

Listing 7-13

The `help` handler iterates through the command table, displaying the `help` string for each command.

The DIOShell Utility

The DIOShell main handles all initialization and allocation of resources, then enters the command parsing loop until the quit command is called. The utility initialization amounts to the allocation of a command buffer of BUFFER_MAX bytes. Once the buffer is successfully allocated, the parser loop is entered.

```
int main(int argc, char *argv)
{
  uint8_t* userp = NULL;

  userp = (uint8_t *)malloc(BUFFER_MAX);
  if (userp == NULL)
  {
    printf("%s: unable to allocate user buffer\n");
    exit(-1);
  }

  while (1)
  {
    command_t*  cmdptr;

    printf("\n> ");
    gets(userp);

    /* parse the users command */
    cmdptr = &commands[0];
    while (cmdptr->command != NULL)
    {
      if (strncmp(userp, cmdptr->command, strlen(cmdptr->com-
                                mand)) == 0)
      {
        cmdptr->functionptr(userp, (char *)NULL);
    break;
      }

      cmdptr++;
    }

    if (cmdptr->command == NULL)
    {
      printf(" invalid command\n");
    }
  }
```

```
  /* since quit handles cleanup and exit we'll never get here */
  exit(0);
}
```

Listing 7-14

The command parsing loop consists of displaying the > prompt, reading the user input, then parsing the command. The command parsing is performed by matching the first string entered by the user to the command element of each entry in the command table. If there is a match, the command handler function is called. All additional parsing and actions are handled by each individual command handler.

The Makefile

```
BASE=DIOShell
DEST=/usr/local/bin
INCLUDES=-I../../inc
LIBRARIES=-L /usr/local/lib -ldioif

DIOShell: DIOShell.c Makefile
    gcc $(INCLUDES) -o $(BASE) $(LIBRARIES) $(BASE).c
    cp $(BASE) $(DEST)

clean:
    rm -f $(BASE)
    rm -f $(DEST)/$(BASE)
    rm -f *.o
```

Summary

This chapter covered the basics of installing, building, and configuring OpenSSH, the secure shell, using FreeBSD. The secure shell provides the basic login allowing secure access to the DIO appliance.

After the discussion of OpenSSH we developed a framework used to implement a basic command line parser and engine that uses callbacks to implement actions based on user input. This framework is used to implement the DIOShell, a digital IO specific shell for the DIO Appliance.

JNI Layer

Overview

The next phase of development in the DIO appliance project is to make the DIO interface Web-aware. The first step toward this is to develop a Java Native Interface (JNI) for the DIO interface library. The JNI is a programming interface, included in the Java Development Kit (JDK) for FreeBSD, for writing Java native methods for calling *C/C++* routines.

- Installing the JDK for FreeBSD
- Defining JNI DIO features
- Implementing native C data types in Java
- Implementing the JNI native interface layer for the DIO interface library

The JDK

Before we begin developing the JNI layer for the DIO interface library, our development environment must be configured for Java development. Configuring the development environment consists of downloading a copy of the Java Development Kit and setting a few environment variables.

Getting the JDK

The first step in developing the JNI layer is obtaining a copy of the JDK. The JDK is not a standard item in the FreeBSD distribution but is available in the FreeBSD ports. To obtain the latest copy of the JDK, change the working directory to the the JDK ports directory, `/usr/ports/java/jdk` and execute the following commands:

```
# cd /usr/ports/java/jdk
# make
# make install
# make clean
```

These instructions can be used to update, build and install any of the FreeBSD port packages. The `make` command retrieves and builds the latest version of the specific port. The next step, `make install`, installs the binaries on your system. After the port is updated, built, and installed, it's a good idea to clean the binaries and temporary build files so disk space isn't exhausted. The `make clean` instruction handles the details of cleaning up the build files.

The JDK_HOME Environment Variable

Many of the JDK tools, build environment, and run-time environment depend on the `JDK_HOME` environment variable to run correctly. After building and installing the JDK, you'll want to set your `JDK_HOME` environment variable to the directory where the JDK was installed using the `make install` command. On my development system, the JDK is installed in `/usr/local/jdk1.1.8`. A quick look at my environment using the `setenv` command shows my `JDK_HOME` environment variable set.

```
JDK_HOME=/usr/local/jdk1.1.8
```

A good place to set the `JDK_HOME` environment variable is in your startup shell scripts. My account uses the `csh`; the following line is added to my `.cshrc` script so it is set at logon.

```
setenv JDK_HOME /usr/local/jdk1.1.8
```

The CLASSPATH Environment Variable

The CLASSPATH environment variable tells the Java Virtual Machine and other Java applications (for example, the Java tools located in the `jdk1.1.8/bin` directory) in which directory to find the class libraries, including user-defined class libraries. The DIO class libraries are kept in a separate directory in `/usr/local/dio`, the directory used to keep the DIO software.

```
# setenv CLASSPATH=/usr/local/dio/class
```

The CLASSPATH variable is set in the `login` script.

The LD_LIBRARY_PATH Environment Variable

LD_LIBRARY_PATH is an environment variable where the system's shared-library loader looks for shared libraries before looking in the system's default shared-library directories. The default shared-library search path is `/lib, /usr/lib, /usr/local/lib.`

```
# setenv LD_LIBRARY_PATH=/usr/local/dio/lib
```

As with the JDK_HOME and CLASSPATH environment variables, the LD_LIBRARY_PATH variable is set in `login` startup scripts.

Creating the JNI Layer

In this section we are going to develop a JNI layer to read and write digital IO lines and read the configured direction of a line. Since we've already developed a C interface to perform these tasks, an intermediate layer is developed, the JNI, which provides the mechanism to call C code from Java code.

Before we turn the crank and begin emitting mass quantities of Java source code, let's take a look at the procedure for developing our JNI interface. First, we need to identify the features that are going to be called from Java. As previously mentioned, they are read line, write line, and read direction. These three capabilities have been implemented by the DIO interface library as `dio_set_line`, `dio_get_line`, and `dio_read_direction`. In addition to the functions, we are going to implement the enums used by these C functions. Java doesn't support enums as a native type, so we'll need to be a little creative.

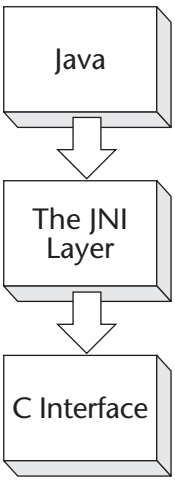


Figure 8-1.
The JNI Layer

With the features defined, we'll take the first step in developing the JNI layer—defining a Java interface for the DIO interface library functions. The Java interface is a Java class that uses native functions as the bridge to call our C DIO interface library. The Java functions call the native functions, which, in turn, call the C functions.

After the Java class is defined, the Java compiler, javac, is run, which produces the java class files. Java class files are byte codes typically executed by the Java environment.

The Java class files are used as input to another tool in the JDK, javah. Javah produces a native header file for functions that were declared native in the Java class. The generated header file is used for inclusion in the C file that implements the native functions.

Finally, once the Java object is completed, the native implementation is completed, and the code is compiled into class files and shared libraries and located in the appropriate directories pointed to by the LD_LIBRARY_PATH and CLASSPATH environment variables, so it can be executed.

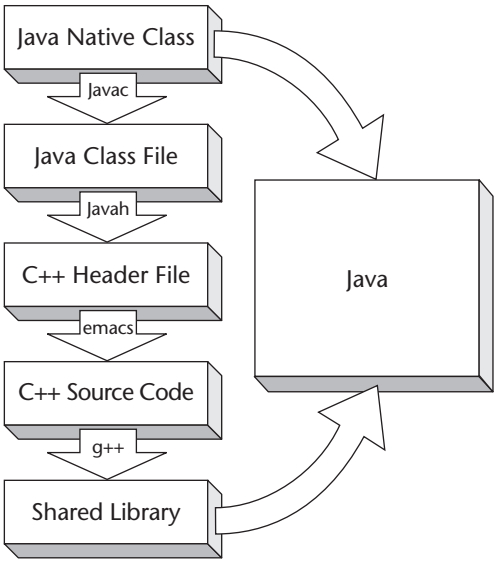


Figure 8-2. Structure of JNI Layer

Implementing C enums in Java

Even though Java has borrowed heavily from C++ syntax, some features of the C++ language are not implemented in Java. One such feature is enums. Since the `dio_set_line`, `dio_get_line` and `dio_read_direction` functions all use enums, we are going to define corresponding types in Java.

There have been a few slightly different definitions of enums in the C language. Initially they were just integers with no type checking. In later definitions, they became proper types but with minimal type algebra. For clarity, I'll define my intended use of enums, and we'll proceed with the Java implementation. A C language enum is a list of objects grouped together into a fixed set. Each of the objects within the list have a defined value.

In Java, we can create the grouping using a class with a private constructor. By making the constructor private, the creation of class objects is restricted. Because we've made a class with a private constructor, creation of new objects is limited to that class. We can now create the members of the enum within that class. Let's take a look at the DIOLineState class.

The DIOLineState Class

The C implementation of the line state is an enum with two states, clear and set. Let's take another look at the `diolinestate_t`, defined in `dioif.h` in Listing 8-1.

```
typedef enum diolinestate_t
{
   clear = 0,
   set = 1
} diolinestate_t;
```

Listing 8-1

The `diolinestate_t` enum contains two members, `clear` with a value of 0 and `set` with a value of 1.

The DIOLineState class in Figure 8-2 provides an implementation of two states scoped by DIOLineState. The DIOLineState constructor is private, limiting the instantiation of DIOLineState object to class members. In essence we've created a name scope, DIOLineState, similar to the C name scope `diolinestate_t`. Now that we're resolved the naming scope, we have the remaining task of defining the line states. Both set and clear are defined as public static final members, making them publicly accessible fixed values of the DIOLineState class.

```
public final class DIOLineState
{
     public static final DIOLineState clear = new
DIOLineState(0);
     public static final DIOLineState set   = new
DIOLineState(1);

     public int State()
     {
     return(state_);
     };

     private DIOLineState(int state)
     {
     state_ = state;
     };

     private int state_;
}
```

Listing 8-2

Using the DIOLineState class defined in Figure 8-2, we've created two states scoped by the DIOLineState name that can be used by Java programs, DIOLineState::set and DIOLineState::clear.

The DIOLineNumber Class

As with the DIOLineState class, the DIOLineNumber class uses a private constructor to limit the scope of the member functions for the DIOLineNumber class. The DIOLineNumber class contains a static final method for each of the enum values in the C enum, effectively emulating the C enum values. Listing 8-3 contains a complete listing of the DIOLineNumber class.

```
public final class DIOLineNumber
{
  public static final DIOLineNumber line0  = new DIOLineNumber(0);
  public static final DIOLineNumber line1  = new DIOLineNumber(1);
  public static final DIOLineNumber line2  = new DIOLineNumber(2);
  public static final DIOLineNumber line3  = new DIOLineNumber(3);
  public static final DIOLineNumber line4  = new DIOLineNumber(4);
  public static final DIOLineNumber line5  = new DIOLineNumber(5);
  public static final DIOLineNumber line6  = new DIOLineNumber(6);
  public static final DIOLineNumber line7  = new DIOLineNumber(7);
  public static final DIOLineNumber line8  = new DIOLineNumber(8);
  public static final DIOLineNumber line9  = new DIOLineNumber(9);
  public static final DIOLineNumber line10 = new DIOLineNumber(10);
  public static final DIOLineNumber line11 = new DIOLineNumber(11);
  public static final DIOLineNumber line12 = new DIOLineNumber(12);
  public static final DIOLineNumber line13 = new DIOLineNumber(13);
  public static final DIOLineNumber line14 = new DIOLineNumber(14);
  public static final DIOLineNumber line15 = new DIOLineNumber(15);
  public static final DIOLineNumber line16 = new DIOLineNumber(16);
  public static final DIOLineNumber line17 = new DIOLineNumber(17);
  public static final DIOLineNumber line18 = new DIOLineNumber(18);
  public static final DIOLineNumber line19 = new DIOLineNumber(19);
  public static final DIOLineNumber line20 = new DIOLineNumber(20);
  public static final DIOLineNumber line21 = new DIOLineNumber(21);
  public static final DIOLineNumber line22 = new DIOLineNumber(22);
  public static final DIOLineNumber line23 = new DIOLineNumber(23);

    public int Number()
    {
    return(number_);
    };
```

```
    private DIOLineNumber(int number)
    {
    number_ = number;
    };

    private int number_;
}
```

Listing 8-3

DIOLineNumber contains additional member functions used to set and get the actual line number; these are necessary for the native functions to pass the correct integer value to the C DIO interface library.

The DIOLineDirection Class

The DIOLineDirection class implementation follows the same pattern as the DIOLineNumber and DIOLineState classes defined in the previous sections. The class contains two public declarations that represent the values of the line state. These values, `lineout` and `linein`, directly correspond to the C enum `dio_direction`, defined in the DIO Interface library developed in Chapter 5. Listing 8-4 shows our implementation of the DIOLineDirections class.

```
public final class DIOLineDirection
{
  public static final DIOLineDirection lineout = new
DIOLineDirection(0);
  public static final DIOLineDirection linein  = new
DIOLineDirection(1);

    public int Direction()
    {
    return(direction_);
    };

    private DIOLineDirection(int direction)
    {
    direction_ = direction;
    };

    private int direction_;
}
```

Listing 8-4

The implementation of the DIOLineDirection class contains a private constructor limiting the values of the class, and a private variable, `direction_`, that contains the value of `direction`, which will be used to pass into the C implementation of the DIO interface.

The Java Code

Now that some of our basic data types have been defined, it's time to define the Java-to-C interface class. Perhaps the most important task in defining the JNI class is to first define which features are included. For simplicity's sake, we will define a Java class that provides the same capabilities provided by the `diod` daemon, a JNI layer that provides member functions to read, write, and determine the direction of the PCI-DIO24 digital IO lines. The interfaces for each of these functions are contained in the DIO interface library; the JNI provides a way to call these functions from Java.

Developing the JNI layer requires nothing more than the use of the native attribute keyword. The native keyword allows the implementation of the member function to be deferred. The actual implementation of the native member functions will occur in a C++ class, defined in a later step.

The DIOIfJNI Class

Let's take a look at the DIOIfJNI class. The DIOIfJNI class is used to create a bridge between the Java world and the DIO interface library. The first task is to import the classes we used to implement the DIO interface enums in Java, `DIOLineNumber`, `DIOLineState`, and `DIOLineDirection`.

The Java implementation of each of our functions—read line, write line, and read direction—are similar; they call a native function that calls the DIO interface library and returns any data obtained from the DIO interface library back to the calling function. Listing 8-5 contains a subset interface to the DIO interface library and the contents of the file `DIOIfJNI.java`.

```
import DIOLineNumber;
import DIOLineState;
import DIOLineDirection;

public class DIOIfJNI
{
```

```
  public native int         SetLineNative(int line, int value);
  public void               SetLine(DIOLineNumber line,
DIOLineState value)
  {
       int stat = SetLineNative(line.Number(), value.State());
  }

  public native int         GetLineNative(int line);
  public DIOLineState       GetLine(DIOLineNumber line)
  {
       int value;
       DIOLineState state;

       value = GetLineNative(line.Number());
       if (value == 0)
       return DIOLineState.clear;
       else
       return DIOLineState.set;
  }

  public native int         GetDirectionNative(int line);
  public DIOLineDirection   GetDirection(DIOLineNumber line)
  {
       int dir;
       DIOLineDirection direction;

       dir = GetDirectionNative(line.Number());
       if (dir == 0)
       return DIOLineDirection.lineout;
       else
       return DIOLineDirection.linein;
  }
}
```

Listing 8-5

The DIOIfJNI library defines three Java member functions that call into the DIO interface library, using the JNI, GetLine, SetLine, and GetDirection. For each of the Java functions there is a corresponding native function that will be used to the corresponding C function, GetLineNative, SetLineNative, and GetDirectionNative.

Generate the Class Files

With the initial Java class defined, the class needs to be compiled using the Java compiler, javac. The output of the Java compiler, the Java class file, is necessary for the next step, so we'll be unable to proceed without compiling. The Java class is compiled using the following command:

```
# javac DIOIfJNI.java
```

The output of this command will be the file `DIOIfJNI.class`. The byte codes output from the javac compiler represent the implementation of the DIOIfJNI class. When working with native methods, the bytecodes are also used to generate a header file for the native code to be developed. We'll learn more about this in the next step of developing the JNI layer.

Generate the Header Files

The Java class file is used as input for the javah tool. The javah tool uses the `java.class` file and generates ANSI function prototypes for the defined native member functions. It is important that the class files reside in a directory contained in the CLASSPATH environment variable.

```
# javah –jni DIOIfJNI
```

The –jni option in the example tells javah to generate a JNI function prototype header. The output of the javah tool is the file `DIOIfJNI.H`; Listing 8-6 contains the complete listing of this file. `DIOIfJNI.h` is available for inclusion by the file containing the native source code implementation.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class DIOIfJNI */

#ifndef _Included_DIOIfJNI
#define _Included_DIOIfJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     DIOIfJNI
 * Method:    SetLineNative
 * Signature: (II)I
 */
```

```
JNIEXPORT jint JNICALL Java_DIOIfJNI_SetLineNative
  (JNIEnv *, jobject, jint, jint);

/*
 * Class:     DIOIfJNI
 * Method:    GetLineNative
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_DIOIfJNI_GetLineNative
  (JNIEnv *, jobject, jint);

/*
 * Class:     DIOIfJNI
 * Method:    GetDirectionNative
 * Signature: (I)I
 */
JNIEXPORT jint JNICALL Java_DIOIfJNI_GetDirectionNative
  (JNIEnv *, jobject, jint);

#ifdef __cplusplus
}
#endif
#endif
```

Listing 8-6

Each of the three function prototypes takes two parameters, in addition to the formal parameters contained in the native method function. The first argument is a pointer to a JNIEnv object, containing the JVM execution environment. The second argument, `jobject`, is similar to a *this* reference in C++.

One other important note is that the generated header files include the file `jni.h`. This file contains all the necessary JNI definitions. It is important to point the `include` directory path for the C compiler to the JDK header files. We'll look at this in more detail in the Makefile section of this chapter.

The Native Code

Now it's time to write the implementations of the native functions. The native functions represent the bridge from Java to C. Native functions are written using C++ and are called by the Java member functions in the DIOIfJNI object. These functions, in turn, call directly through to the C implementation of the DIO interface library. The primary object of the native functions is to provide the glue to massage Java data types onto C data

types, pass the appropriate data between the layers, and provide access to the C library from Java.

The `Java_DIOIfJNI_SetLineNative` Function

The `Java_DIOIfJNI_SetLineNative` function is used to call the DIO interface library `dio_set_line` which, in turn, is used to write a digital IO line.

```
JNIEXPORT jint JNICALL Java_DIOIfJNI_SetLineNative
  (JNIEnv *env, jobject thisObj, jint line, jint value)
{
     dioline_t       dioline = static_cast<dioline_t>(line);
     diolinestate_t  diostate;

     dio_set_line(dioline, diostate);
}
```

Listing 8-7

The `Java_DIOIfJNI_SetLineNative` accepts the parameters passed by the Java caller and passes them through to the `dio_set_line` function, implemented by the DIO interface library.

The `Java_DIOIfJNI_GetLineNative` Function

The `Java_DIOIfJNI_GetLineNative` function is used to call the DIO interface library `dio_get_line` which, in turn, is used to read a digital IO line.

```
JNIEXPORT jint JNICALL Java_DIOIfJNI_GetLineNative
  (JNIEnv *, jobject, jint line)
{
     dioline_t       dioline = static_cast<dioline_t>(line);
     diolinestate_t  diostate;

     dio_get_line(dioline, &diostate);

     return (diostate);
}
```

Listing 8-8

The `Java_DIOIfJNI_GetLineNative` provides the means to call the `dio_get_line` function, which reads the state of a digital IO line. The line is passed into the `dio_get_line` and a local variable is used to contain the state of the digital line provided by the `dio_get_line` function. After `dio_get_line` is called the return value is returned to the Java calling function.

The `Java_DIOIfJNI_GetDirectionNative` Function

The `Java_DIOIfJNI_GetDirectionNative` function is used to call the DIO interface library `dio_get_direction` which, in turn, is used to read the direction of a digital IO line.

```
JNIEXPORT jint JNICALL Java_DIOIfJNI_GetDirectionNative
  (JNIEnv *, jobject, jint line)
{
    dioline_t       dioline = static_cast<dioline_t>(line);
    diodirection_t  diodirection;
    int32_t         stat;

    dio_get_direction(dioline, &diodirection);

    return (diodirection);
}
```

Listing 8-9

As with the previous function, `Java_DIOIfJNI_GetDirectionNative` passes the requested line number directly through to the `dio_get_direction` function; the line direction is contained in a local variable and passed back to the Java calling function.

The Makefile

When developing the JNI interface, it is necessary to include the JDK include file directories. The two directories necessary are `include` and `include/freebsd` . In the JNI layer Makefile, these directories were added to the CFLAGS variable. Listing 8-10 shows the complete Makefile for developing the JNI interface for the DIO appliance.

```
CFLAGS= -fpic -shared -I../inc -I$(JDK_HOME)/include -
I$(JDK_HOME)/include/freebsd

all: libdioifjni.so

DIOIfJNI.h: DIOIfJNI.java DIOLineNumber.java DIOLineState.java
DIOLineDirection.java
    javac DIOIfJNI.java
    javah -jni DIOIfJNI

libdioifjni.so: DIOIfJNI.h DIOIfJNI.cpp
```

```
    g++ $(CFLAGS) -o libdioifjni.so DIOIfJNI.cpp -L/usr/local/lib
                              -ldioif
    cp libdioifjni.so /usr/local/lib

clean:
    rm -f libdioifjni.so
    rm -f DIOIfJNI.h
```

Listing 8-10

In addition to the typical C compiler rules, additional rules are created for executing the Java compiler and automatically generating the JNI header file using javah.

Summary

In this chapter, we've taken a close look at installing the JDK for Java and JNI development, along with setting up an environment for development and debugging. After setting up the JDK development environment, we've developed a JNI interface that supports an interface to a few of the basic functions implemented in the DIO interface library, through the implementation of the `DIOIfJNI` object. With the implementation of the `set line`, `get line`, and `get direction` functions as a foundation, you could implement the remainder to DIO interface class in the JNI layer.

Using the implementation of the `DIOIfJNI` object, we are now able to development a JSP page and monitor the state of the digital IO lines from a web page, which is the topic of discussion in our next chapter.

Web Access Using Tomcat

Overview

In the previous chapter, we developed a JNI layer for the DIO appliance library. This represents the first of two steps in implementing a web interface to monitor the DIO's digital IO lines. The second step is to develop a Java Server Page (JSP) that uses the JNI object developed in Chapter 8 to read and write the digital IO lines via a web page.

JSP is a technology that allows us to develop web pages that display dynamic content. Using the JNI object developed in Chapter 8, along with the JSP page developed in this chapter, we will be able to develop a web page that displays the DIO lines dynamically. In order to display JSP pages we'll install and configure a JSP engine. The FreeBSD ports collection contains Tomcat, a Java-based web server that provides support for JSP.

In this chapter, we will discuss issues related to JSP developing, including:

- Installing the Tomcat JSP server on FreeBSD.
- Setting up the Tomcat environment.
- The elements of a JSP page.
- Developing the DIO JSP interface.

Tomcat

As with the JDK in the previous chapter, our first step is to download and install the Tomcat server that provides JSP support. Tomcat is provided in the ports collection and is downloaded, installed, and configured using the standard ports installation procedure. Let's review those steps now.

Installing Tomcat

The version of Tomcat installed on my system is Tomcat 3.2.3.

```
# cd /usr/ports/www/jakarta-tomcat
# make
# make install
# make clean
```

These instructions may be used to update, build, and install any of the FreeBSD port packages. The `make` command retrieves and builds the latest version of the specific port. The next step, `make install`, installs the binaries on your system. After the port is updated, built, and installed, it's a good idea to clean the binaries and temporary build files so disk space isn't exhausted; `make clean` handles the details of cleaning up the build files.

The TOMCAT_HOME Environment Variable

Once the Tomcat port has been compiled and installed, the environment should be set up. This consists of setting the TOMCAT_HOME environment. The correct value of TOMCAT_HOME is the directory in which Tomcat was installed. On my system this is `/usr/local/tomcat`.

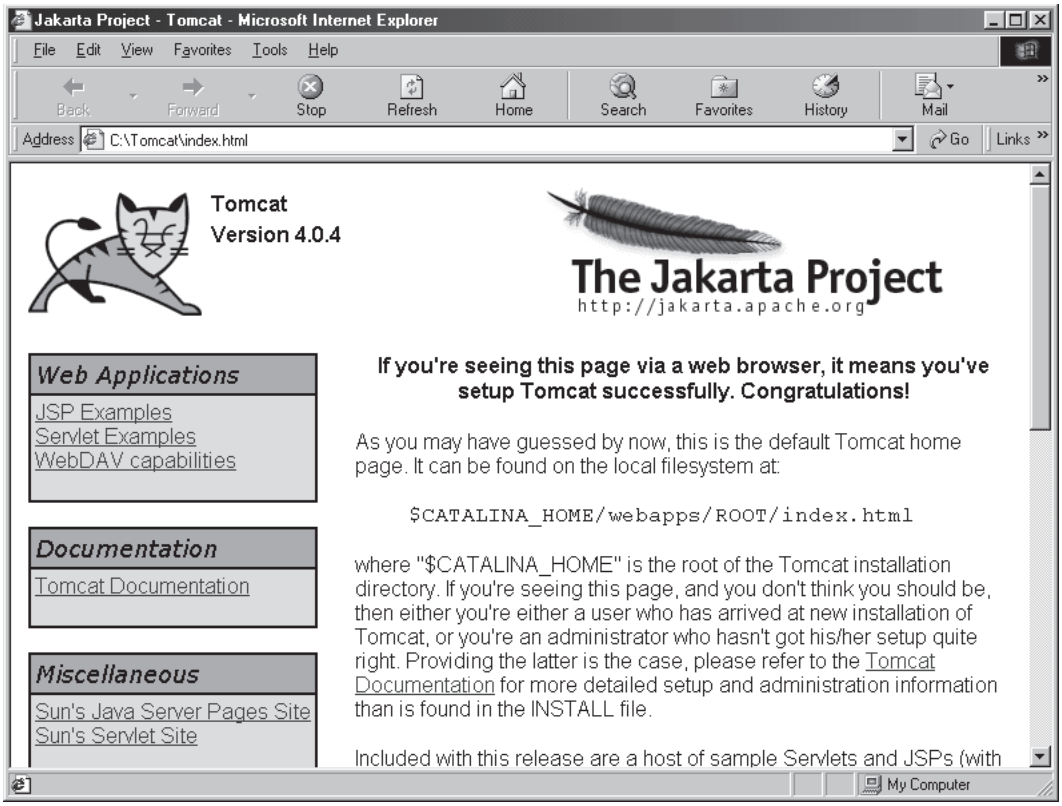
Starting and Stopping Tomcat

bin

As part of the Tomcat port installation, a script, `tomcat.sh`, is installed in the `/usr/local/etc/rc.d` directory. This script is responsible for starting Tomcat at system boot time. Once Tomcat is installed, you can restart your system and verify that it is working properly by starting your browser and pointing it to the url,

http://localhost:8080

You should see the Tomcat main page in your browser.



The Tomcat Directory Structure

Once Tomcat has been installed, it creates a series of directories in the TOMCAT_HOME directory. Let's take a quick look and familiarize ourselves with each of the directories and their contents.

bin

During installation, Tomcat installs a series of scripts in the `TOMCAT_HOME/bin` directory for Tomcat administration. Two of the most used scripts are `startup.sh` and `shutdown.sh`. These scripts are used to start and stop Tomcat.

Starting Tomcat is performed by the following command:

```
# cd $TOMCAT_HOME/bin
# startup.sh
```

Stopping Tomcat is performed by the following command:

```
# cd $TOMCAT_HOME/bin
# shutdown.sh
```

doc

The doc directory contains Tomcat documentation in html format, so you can point your browser there and peruse the documentation.

lib

The lib directory contains jar files used by Tomcat. This lib directory is automatically appended to the CLASSPATH environment variable.

logs

The logs directory is used to store Tomcat's logs.

webapps

Before venturing into JSP syntax, there is one more important detail. Now that Tomcat is running successfully, where are JSP pages placed? The default location for content is the `$TOMCAT_HOME/webapps/root` directory.

conf

The conf directory contains Tomcat's configuration files. Two of the most used configuration files are `server.xml` and `web.xml`. These files are used to configure Tomcat and control settings for web applications using Tomcat.

JSP Overview

JSP is used to display dynamic content. A typical JSP page consists of an HTML file with special directives that display dynamic content. For the DIO display page, we'll be developing a standard html page, using JSP directives to call the JNI code developed in the previous chapter to read the DIO digital input lines. Before jumping into the details of JSP, let's look at a simple JSP page.

```
<%@ page language="java" contentType="text/html" %>
<html>
  <head>
    <title>
      Hello DIO JSP Page
    </title>
  </head>
```

```
  <body>
    <% out.println("Hello DIO!\n"); %>
  </body>
</html>
```

Listing 9-1

This JSP page, `hellodio.jsp`, consists of HTML with JSP directives for using Java. We'll explore JSP syntax in the following sections. To view the `hellodio` page, enter the following into your browser,

http://localhost:8080/hellodio.jsp. The `hellodio.jsp` page displays:

```
Hello DIO!
```

in your browser.

JSP Syntax

As mentioned in the previous section, a JSP page looks like html, with JSP codes that tell the JSP engine where to look for supporting Java classes or for inserting Java code to generate dynamic content into the web page.

Comments

Like all programming languages, JSP provides an element for adding comments to your JSP code. A comment is denoted by the <%— —%> tags.

```
<%— JSP is really cool %>
```

Directives

JSP directives are used to specify how the JSP page is handled. A JSP directive does not produce any visible output. A directive is specified by the <%@ %> tag. For example, in Listing 9-1, the Java language was selected using the page directive.

```
<%@ page language="java" contentType="text/html" %>
```

There are three types of directives: page, language, and contentType. The page directive in Listing 9-1 is used for specifying the language and content type. In this case, Java is the language and html is the content. In addition, the page directive has many methods for controlling page attributes.

The language attribute is used to specify the scripting language. Although JSP is able to use multiple scripting languages, Tomcat only supports Java. Subsequently, Java is the default language, but it is set in the page directive for the sake of clarity.

Finally, the contentType attribute is used to specify the type of content produced by the JSP page. Once again looking at Listing 9-1, the output is html, the most common type of page. Other types of pages are text/plain, text/xml for applications, and so forth.

Declarations

The next type of JSP statement is a declaration statement. The declaration statement is denoted by the <%! %> tags. A declaration allows the programmer to define page level variables for storing information that a page may need.

```
<%! int jspvar = 27; %>
```

Expressions

JSP expressions are used to evaluate expressions and convert the result into text for display in the output page. JSP expressions use the <%= %> tags.

Code Scriptlets

Code scriptlets, denoted by the <% %> tags, are used to run Java code when the page is serviced by the Web server. In Listing 9-1, we used a JSP scriptlet to write Hello DIO! to the output page.

```
<% out.println("Hello DIO!\n"); %>
```

The DIO JSP Page

Now that we've taken a look at the JSP syntax, it's time to develop our DIO JSP page for displaying the states of digital IO lines via your favorite web browser. Let's take a look at Listing 9-2. The JSP begins by declaring the language and contentType. Once again, the settings are the default values but the JSP code is better understood by declaring these here. After the page directive, the JSP page contains a few lines of html to set the title page for your browser.

The body of the JSP page contains the code necessary to generate output for displaying the digital IO lines. First we load the DIO interface library. Note we're using the system loader because we're using a native library. After loading the DIO interface library, the code consists of a few output statements and using the DIOIfJNI interface to determine the direction of the lines. If they are input, it reads them and displays the values.

```
<%— Copyright (c) 2002 Paul Cevoli and Butterworth Heinemann —%>

<%@ page language="java" contentType ="text/html" %>

<html>
  <head>
    <title> DIO Appliance JSP Page </title>
  </head>

  <body>
    <%
    System.loadLibrary("dioifjni");
    %>

    <%— declare an instance of the DIO JNI object —%>
    <%!
    DIOIfJNI DIO = new DIOIfJNI();
    %>

    <%
      out.println("\n");
      out.println("DIO Appliance JSP Monitor\n");

      <%— display the line status, using lines 0 thru 7 —%>

      if (dio.GetDirection(DIOLineNumber.line0) ==
                           DIOLineDirection.linein)
      {
        if (dio.GetLine(DIOLineNumber.line0) ==
                           DIOLineState.clear)
       out.println("0");
    else
       out.println("1");
      }

      if (dio.GetDirection(DIOLineNumber.line1) ==
```

```
                              DIOLineDirection.linein)
  {
    if (dio.GetLine(DIOLineNumber.line1) ==
                              DIOLineState.clear)
   out.println("0");
else
   out.println("1");
  }

  if (dio.GetDirection(DIOLineNumber.line2) ==
                              DIOLineDirection.linein)
  {
    if (dio.GetLine(DIOLineNumber.line2) ==
                              DIOLineState.clear)
   out.println("0");
else
   out.println("1");
  }

  if (dio.GetDirection(DIOLineNumber.line3) ==
                              DIOLineDirection.linein)
  {
    if (dio.GetLine(DIOLineNumber.line3) ==
                              DIOLineState.clear)
   out.println("0");
else
   out.println("1");
  }

  if (dio.GetDirection(DIOLineNumber.line4) ==
                              DIOLineDirection.linein)
  {
    if (dio.GetLine(DIOLineNumber.line4) ==
                              DIOLineState.clear)
   out.println("0");
else
   out.println("1");
  }

  if (dio.GetDirection(DIOLineNumber.line5) ==
                              DIOLineDirection.linein)
  {
    if (dio.GetLine(DIOLineNumber.line5) ==
```

```
                                DIOLineState.clear)
        out.println("0");
    else
        out.println("1");
       }

       if (dio.GetDirection(DIOLineNumber.line6) ==
                                DIOLineDirection.linein)
       {
          if (dio.GetLine(DIOLineNumber.line6) ==
                                DIOLineState.clear)
        out.println("0");
    else
        out.println("1");
       }

       if (dio.GetDirection(DIOLineNumber.line7) ==
                                DIOLineDirection.linein)
       {
          if (dio.GetLine(DIOLineNumber.line7) ==
                                DIOLineState.clear)
        out.println("0");
    else
        out.println("1");
       }
      %>
  </body>
</html>
```

Listing 9-2

The JSP page displays lines 0–7 for sake of brevity. Displaying the other lines could consist of adding the code testing the relevant lines or extending the DIOIfJNI class to use iterators, and then iterate through all the lines.

Summary

In this chapter we reviewed the syntax of JSP pages and developed a JSP page to display digital IO lines for the DIO appliance. The programming tasks are now completed for the DIO appliance. In the remaining chapters, we'll focus on building the kernel and making sure our components start properly.

Building the Kernel

Overview

The kernel is the core of the FreeBSD operating system. It is responsible for managing resources, enforcing security, networking, disk access, and more. The default kernel is located in the file `/kernel`.

In the design and development of an embedded system, disk space and memory are critical resources. Unlike a typical desktop computer packed with RAM and disk space, an embedded computer only needs minimal support to perform its task, with respect to software as well as hardware.

During FreeBSD installation, a kernel is installed on your computer called the GENERIC kernel. The GENERIC kernel is created to support as many computers and configurations as possible to simplify the installation process. This chapter discusses how to configure and build a custom FreeBSD kernel to suit our hardware and DIO application.

The Kernel Configuration File

The first step toward building a kernel is to create a custom kernel configuration file, which allows us to tailor the kernel for our specific hardware. One recommended method for creating a custom kernel config file is to start with the generic config file, located in `/usr/i386/conf/GENERIC` and remove the extra peripherals. We will create our config file using this method, but first we'll look at the GENERIC file.

The GENERIC kernel provides support for many popular peripherals contained in PCs. Most of these devices can be removed to give us a much smaller kernel. Kernel configuration files are located in the `/sys/i386/conf` directory.

GENERIC Configuration File

In the following sections you'll find definitions of the main parts of a kernel configuration file.

The `machine` Keyword

The machine keyword defines the CPU architecture the kernel will execute on. The GENERIC Configuration file for the x86 FreeBSD distribution is i386. Only one machine keyword can be specified in the config file.

```
machine     i386
```

The `cpu` Keyword

The `cpu` keyword defines the CPU types the kernel will execute on. GENERIC is configured to run on Intel 386 and greater CPUs. A kernel can be configured to run on more than one CPU. However, the CPUs defined in the configuration file must be binary-compatible. If the DIO runs on a Celeron processor, our configuration will consist of I686_CPU architecture.

```
cpu     I386_CPU
cpu     I486_CPU
cpu     I586_CPU
cpu     I686_CPU
```

The `ident` Keyword

Here, `ident` is used to specify the name of the kernel. Each configuration file should have a unique name. The value specified by the `ident` keyword is the value displayed on the system console during boot.

```
ident       GENERIC
```

The `maxusers` Keyword

The `maxusers` keyword tunes the sizes of kernel internal data structures. The larger this value, the more system memory is used. Because the DIO appliance is a dedicated system, we will lower this value to consume less system memory.

```
maxusers    32
```

Kernel Options

The following sections provide a description of each of the global kernel options for the GENERIC configuration file.

The `makeoptions` Keyword

The `makeoptions` keyword specifies compiler options that are processed by the config command and passed to the C compiler. For example, if we wanted the debugging option to analyze system crash dump, the following line would be added.

```
makeoptions DEBUG=-g    #Build kernel with gdb(1) debug symbols
```

The `options` Keyword

The `options` keyword is used to customize the kernel by setting various options.

MATH_EMULATE If your computer contains an i386 or i486 processor without a math coprocessor, the kernel may provide math coprocessor support. Since our data logger uses a Celeron processor, which contains floating-point support, this will be removed.

```
options     MATH_EMULATE    #Support for x87 emulation
```

INET These options are used for networking access, INET and INET6. The DIO appliance will not be connected to an IPV6 network, so these will be removed.

```
options     INET            #InterNETworking
options     INET6           #IPv6 communications protocols
```

File Systems FreeBSD provides support for numerous file systems. FFS, originally known as UFS, is the default file system that the data logger uses. Additionally, we will be booting from a flash device and running using a memory file system and would like to keep support for MSDOS floppy disks and CDROMs for a future upgrade path.

```
options     FFS             #Berkeley Fast Filesystem
options     FFS_ROOT        #FFS usable as root device [ keep
this!]
```

```
options     SOFTUPDATES      #Enable FFS soft updates support
options     MFS              #Memory Filesystem
options     MD_ROOT          #MD is a potential root device
options     NFS              #Network Filesystem
options     NFS_ROOT         #NFS usable as root device, NFS
required
options     MSDOSFS          #MSDOS Filesystem
options     CD9660           #ISO 9660 Filesystem
options     CD9660_ROOT      #CD-ROM usable as root, CD9660
required
options     PROCFS           #Process filesystem
options     COMPAT_43        #Compatible with BSD 4.3 [KEEP THIS!]
```

SCSI Delay The kernel has a tunable value for probing the SCSI bus. The data logger does not have any SCSI peripherals, so this will be removed.

```
options     SCSI_DELAY=15000 #Delay (in ms) before probing SCSI
```

Console This allows users to grab the console; this is particularly useful for X Windows. Since the DIO appliance does not run X Windows, this will be removed.

```
options     UCONSOLE     #Allow users to grab the console
```

Ktrace Support for the `ktrace` system call. This feature is useful for debugging and reverse engineering system utilities. This will be removed.

```
options     KTRACE       #ktrace(1) support
```

System V Interprocess Communication These options provide support for System V interprocess communication, shared memory, messages, and semaphores. These are most useful for X Windows, but other system utilities may use these as well.

```
options     SYSVSHM      #SYSV-style shared memory
options     SYSVMSG      #SYSV-style message queues
options     SYSVSEM      #SYSV-style semaphores
```

Multiprocessor Support These options provide multiprocessor support; since the data logger runs on a single CPU Celeron, these will be removed.

```
# To make an SMP kernel, the next two are needed
#options    SMP         # Symmetric MultiProcessor Kernel
#options    APIC_IO     # Symmetric (APIC) I/O
```

Controllers and Device Drivers

Buses The next section provides support for system buses. The data logger uses ISA and PCI buses; EISA will be removed.

```
device      isa
device      eisa
device      pci
```

Floppy Drive Controllers The data logger will at most support a single floppy drive; the second device may be removed.

```
# Floppy drives
device      fdc0    at isa? port IO_FD1 irq 6 drq 2
device      fd0 at fdc0 drive 0
device      fd1 at fdc0 drive 1
```

ATAPI Controllers The next section provides support for ATA and ATAPI devices. The DIO appliance uses ATA disk and ATAPI CDROM support to boot and access peripherals for debugging and upgrades.

```
device      ata0    at isa? port IO_WD1 irq 14
device      ata1    at isa? port IO_WD2 irq 15
device      ata
device      atadisk         # ATA disk drives
device      atapicd         # ATAPI CDROM drives
device      atapifd         # ATAPI floppy drives
device      atapist         # ATAPI tape drives
options     ATA_STATIC_ID   #Static device numbering
```

SCSI Controllers The GENERIC kernel provides support for many SCSI controllers and devices. The DIO appliance does not use any SCSI peripherals, so all of these may be removed.

```
device      ahb         # EISA AHA1742 family
device      ahc         # AHA2940 and onboard AIC7xxx devices
device      amd         # AMD 53C974 (Tekram DC-390(T))
device      isp         # Qlogic family
device      ncr         # NCR/Symbios Logic
```

```
device      sym         # NCR/Symbios Logic (newer chipsets)
options     SYM_SETUP_LP_PROBE_MAP=0x40
                        # Allow ncr to attach legacy NCR devices
when
                        # both sym and ncr are configured

device      adv0    at isa?
device      adw
device      bt0 at isa?
device      aha0    at isa?
device      aic0    at isa?

device      ncv         # NCR 53C500
device      nsp         # Workbit Ninja SCSI-3
device      stg         # TMC 18C30/18C50

# SCSI peripherals
device      scbus       # SCSI bus (required)
device      da          # Direct Access (disks)
device      sa          # Sequential Access (tape etc)
device      cd          # CD
device      pass        # Passthrough device (direct SCSI access)
```

RAID Controllers As with SCSI, the GENERIC kernel provides support for numerous RAID peripherals. The data logger does not use any RAID devices, so all of these may be removed.

```
# RAID controllers interfaced to the SCSI subsystem
device      asr     # DPT SmartRAID V, VI and Adaptec SCSI RAID
device      dpt     # DPT Smartcache - See LINT for options!
device      mly     # Mylex AcceleRAID/eXtremeRAID

# RAID controllers
device      aac     # Adaptec FSA RAID, Dell PERC2/PERC3
device      ida     # Compaq Smart RAID
device      amr     # AMI MegaRAID
device      mlx     # Mylex DAC960 family
device      twe     # 3ware Escalade
```

Keyboard and Mouse The next section builds support for the keyboard, consoles, mouse, and FreeBSD splash screen. We'll keep support for all of these for debugging and administration of the DIO, with the exception of the splash screen, which will be removed.

```
# atkbdc0 controls both the keyboard and the PS/2 mouse
device      atkbdc0 at isa? port IO_KBD
device      atkbd0  at atkbdc? irq 1 flags 0x1
device      psm0    at atkbdc? irq 12

device      vga0    at isa?

# splash screen/screen saver
pseudo-device   splash

# syscons is the default console driver, resembling an SCO con
          sole
device      sc0 at isa? flags 0x100

# Enable this and PCVT_FREEBSD for pcvt vt220 compatible console
          driver
#device     vt0 at isa?
#options    XSERVER       # support for X server on a vt console
#options    FAT_CURSOR    # start with block cursor
# If you have a ThinkPAD, uncomment this along with the rest of
# the PCVT lines
#options    PCVT_SCANSET=2      # IBM keyboards are non-std
```

Floating Point The floating point device is required by the kernel; do not remove this line.

```
# Floating point support - do not disable.
device      npx0    at nexus? port IO_NPX irq 13
```

Power Management FreeBSD has the option of providing power management. The data logger does not use power management, so this will be removed.

```
# Power management support (see LINT for more options)
device      apm0     at nexus? disable flags 0x20 # Advanced
Power Management
```

PCCARD Support The GENERIC kernel provides support for PCCARD peripherals. These can be removed.

```
# PCCARD (PCMCIA) support
device      card
device      pcic0   at isa? irq 0 port 0x3e0 iomem 0xd0000
```

```
device      pcic1   at isa? irq 0 port 0x3e2 iomem 0xd4000 disable
```

Serial Port Four serial ports are supported by default. The Roadster hardware has two; the first two serial ports are kept for support, administration, and debugging.

```
# Serial (COM) ports
device      sio0    at isa? port IO_COM1 flags 0x10 irq 4
device      sio1    at isa? port IO_COM2 irq 3
device      sio2    at isa? disable port IO_COM3 irq 5
device      sio3    at isa? disable port IO_COM4 irq 9
```

Parallel Port The parallel port contains printer support, TCP/IP support, and SCSI support. We'll keep printer support for debugging and administration.

```
# Parallel port
device      ppc0    at isa? irq 7
device      ppbus       # Parallel port bus (required)
device      lpt     # Printer
device      plip        # TCP/IP over parallel
device      ppi     # Parallel port interface device
#device     vpo     # Requires scbus and da
```

Ethernet Controllers There are numerous network interface cards built into the FreeBSD kernel. The data logger uses an Intel EtherExpress Pro 100; other controllers may be removed. One important note about the EtherExpress Pro 100: support is added on top of the MII bus support, so this driver must be kept in the kernel configuration file, in order to properly build the EtherExpress Pro 100 support.

```
# PCI Ethernet NICs.
device      de      # DEC/Intel DC21x4x (``Tulip'')
device      txp     # 3Com 3cR990 (``Typhoon'')
device      vx      # 3Com 3c590, 3c595 (``Vortex'')

# PCI Ethernet NICs that use the common MII bus controller code.
# NOTE: Be sure to keep the 'device miibus' line in order to use
# these NICs!
device      miibus      # MII bus support
device      dc      # DEC/Intel 21143 and various workalikes
device      fxp     # Intel EtherExpress PRO/100B (82557, 82558)
```

```
device      pcn     # AMD Am79C97x PCI 10/100 NICs
device      rl      # RealTek 8129/8139
device      sf      # Adaptec AIC-6915 (``Starfire'')
device      sis     # Silicon Integrated Systems SiS 900/SiS 7016
device      ste     # Sundance ST201 (D-Link DFE-550TX)
device      tl      # Texas Instruments ThunderLAN
device      tx      # SMC EtherPower II (83c170 ``EPIC'')
device      vr      # VIA Rhine, Rhine II
device      wb      # Winbond W89C840F
device      wx      # Intel Gigabit Ethernet Card (``Wiseman'')
device      xl      # 3Com 3c90x (``Boomerang'', ``Cyclone'')

# ISA Ethernet NICs.
# 'device ed' requires 'device miibus'
device      ed0 at isa? port 0x280 irq 10 iomem 0xd8000
device      ex
device      ep
device      fe0 at isa? port 0x300
# Xircom Ethernet
device      xe
# PRISM I IEEE 802.11b wireless NIC.
device      awi
# WaveLAN/IEEE 802.11 wireless NICs. Note: the WaveLAN/IEEE really
# exists only as a PCMCIA device, so there is no ISA attachment
# needed and resources will always be dynamically assigned by the
# pccard code.
device      wi
# Aironet 4500/4800 802.11 wireless NICs. Note: the declaration
# below will work for PCMCIA and PCI cards, as well as ISA cards
# set to ISA PnP mode (the factory default). If you set the
# switches on your ISA card for a manually chosen I/O address and
# IRQ, you must specify those parameters here.
device      an
# The probe order of these is presently determined by
i386/isa/isa_compat.c.
device      ie0 at isa? port 0x300 irq 10 iomem 0xd0000
#device     le0 at isa? port 0x300 irq 5 iomem 0xd0000
device      lnc0    at isa? port 0x280 irq 10 drq 0
device      cs0 at isa? port 0x300
device      sn0 at isa? port 0x300 irq 10
```

Pseudo Devices

Pseudo devices are devices that are built into the kernel but do not contain

any hardware. Common uses for pseudo devices are logging, pseudo terminals, vnodes and snoop devices. The DIO appliance uses the network loopback and ethernet support pseudo devices.

```
# Pseudo devices - the number indicates how many units to allocate.
pseudo-device    loop    # Network loopback
pseudo-device    ether       # Ethernet support
pseudo-device    sl  1   # Kernel SLIP
pseudo-device    ppp 1   # Kernel PPP
pseudo-device    tun     # Packet tunnel.
pseudo-device    pty     # Pseudo-ttys (telnet etc)
pseudo-device    md      # Memory "disks"
pseudo-device    gif     # IPv6 and IPv4 tunneling
pseudo-device    faith   1   # IPv6-to-IPv4 relaying (translation)

# The `bpf' pseudo-device enables the Berkeley Packet Filter.
# Be aware of the administrative consequences of enabling this!
pseudo-device    bpf     #Berkeley packet filter
```

USB Support

USB support is provided to connect USB devices to FreeBSD. Although the Roadster hardware does supply USB ports, the DIO appliance does not use the USB bus. Support for USB will be removed.

```
# USB support
device       uhci    # UHCI PCI->USB interface
device       ohci    # OHCI PCI->USB interface
device       usb     # USB Bus (required)
device       ugen    # Generic
device       uhid    # "Human Interface Devices"
device       ukbd    # Keyboard
device       ulpt    # Printer
device       umass   # Disks/Mass storage - Requires scbus and da
device       ums     # Mouse
device       uscanner# Scanners
# USB Ethernet, requires mii
device       aue     # ADMtek USB ethernet
device       cue     # CATC USB ethernet
device       kue     # Kawasaki LSI USB ethernet
```

Building the DIO Kernel

Hardware Inventory

First, take an inventory of your hardware. The Network Engines Roadster used for this project is a basic Intel PC clone. Our inventory results in the hardware list shown in Table 10-1.

Table 10-1. Hardware Inventory

Peripheral	Options
CPU	366 MHz Celeron
Memory	32 MB SDRAM
Floppy Drive	AT Floppy
CDROM	IDE CD-ROM
Parallel	Port 1
Buses	ISA, PCI
Serial Ports	2
Boot Device	32 MB ATAPI Flash, MFS Support
Network	Intel EtherExpress Pro 100

Next, we need to create a simple kernel config file. I suggest making a copy of the GENERIC config file and removing the keywords that represent hardware and options that are not representative of the DIO appliance, as discussed in the previous sections. I named my config file DIO.

The DIO Kernel Configuration File

Here is a copy of the updated configuration file, DIO. You'll notice that many of the irrelevant devices have been removed and many of the other system settings have been tuned to represent the listed hardware.

```
#
# DIO - Digital Appliance Kernel Configuration File
#
# For more information on this file, please read the handbook
# section on Kernel Configuration Files:
#
#    http://www.FreeBSD.org/handbook/kernelconfig-config.html
#
# The handbook is also available locally in /usr/share/doc/hand
# book if you've installed the doc distribution, otherwise always
# see FreeBSD World Wide Web server (http://www.FreeBSD.org/) for
# the latest information.
```

```
#
# An exhaustive list of options and more detailed explanations
# of the device lines is also present in the ./LINT configuration
# file. If you are in doubt as to the purpose or necessity of a
# line, check first in LINT.
#
# $FreeBSD: src/sys/i386/conf/GENERIC,v 1.246.2.34 2001/08/12
# 13:13:46 joerg Exp $

machine     i386
cpu     I686_CPU
ident       DIO
maxusers    8

#makeoptions    DEBUG=-g        #Build kernel with gdb(1) debug
                                #symbols

options     MATH_EMULATE     #Support for x87 emulation
options     INET             #InterNETworking
options     FFS          #Berkeley Fast Filesystem
options     FFS_ROOT     #FFS usable as root device [ keep this!]
options     MFS          #Memory Filesystem
options     MD_ROOT          #MD is a potential root device
options     NFS          #Network Filesystem
options     NFS_ROOT         #NFS usable as root device, NFS
required
options     MSDOSFS          #MSDOS Filesystem
options     CD9660           #ISO 9660 Filesystem
options     CD9660_ROOT      #CD-ROM usable as root, CD9660
required
options     PROCFS           #Process filesystem
options     COMPAT_43        #Compatible with BSD 4.3 [ KEEP THIS!]
options     SCSI_DELAY=15000 #Delay (in ms) before probing SCSI
options     UCONSOLE         #Allow users to grab the console
options     USERCONFIG       #boot -c editor
options     SYSVSHM          #SYSV-style shared memory
options     SYSVMSG          #SYSV-style message queues
options     SYSVSEM          #SYSV-style semaphores
options     P1003_1B         #Posix P1003_1B real-time extensions
options     _KPOSIX_PRIORITY_SCHEDULING
options     ICMP_BANDLIM         #Rate limit bad replies
options     KBD_INSTALL_CDEV     # install a CDEV entry in /dev
```

```
# To make an SMP kernel, the next two are needed
#options    SMP         # Symmetric MultiProcessor Kernel
#options    APIC_IO         # Symmetric (APIC) I/O

device      isa
device      eisa
device      pci

# Floppy drives
device      fdc0    at isa? port IO_FD1 irq 6 drq 2
device      fd0 at fdc0 drive 0
#
# If you have a Toshiba Libretto with its Y-E Data PCMCIA floppy,
# don't use the above line for fdc0 but the following one:
#device     fdc0

# ATA and ATAPI devices
device      ata0    at isa? port IO_WD1 irq 14
device      ata1    at isa? port IO_WD2 irq 15
device      ata
device      atadisk         # ATA disk drives
device      atapicd         # ATAPI CDROM drives
device      atapifd         # ATAPI floppy drives
device      atapist         # ATAPI tape drives
options     ATA_STATIC_ID   #Static device numbering

# atkbdc0 controls both the keyboard and the PS/2 mouse
device      atkbdc0 at isa? port IO_KBD
device      atkbd0  at atkbdc? irq 1 flags 0x1
device      psm0    at atkbdc? irq 12

device      vga0    at isa?

# syscons is the default console driver, resembling an SCO con-
sole
device      sc0 at isa? flags 0x100

# Floating point support - do not disable.
device      npx0    at nexus? port IO_NPX irq 13

# Serial (COM) ports
device      sio0    at isa? port IO_COM1 flags 0x10 irq 4
device      sio1    at isa? port IO_COM2 irq 3
```

```
# Parallel port
device      ppc0    at isa? irq 7
device      ppbus   # Parallel port bus (required)
device      lpt     # Printer
device      ppi     # Parallel port interface device

# PCI Ethernet NICs.

# PCI Ethernet NICs that use the common MII bus controller code.
# NOTE: Be sure to keep the 'device miibus' line in order to use
#these NICs!
device      miibus  # MII bus support
device      fxp     # Intel EtherExpress PRO/100B (82557, 82558)

# Pseudo devices - the number indicates how many units to
#allocate.
pseudo-device   loop            # Network loopback
pseudo-device   ether           # Ethernet support
pseudo-device   sl  1       # Kernel SLIP
pseudo-device   ppp 1       # Kernel PPP
pseudo-device   tun         # Packet tunnel.
pseudo-device   pty         # Pseudo-ttys (telnet etc)
pseudo-device   md          # Memory "disks"
pseudo-device   gif         # IPv6 and IPv4 tunneling
pseudo-device   faith   1   # IPv6-to-IPv4 relaying (translation)
```

Building the FreeBSD Kernel

Now that you've tuned your kernel config file to fit your hardware, it's time to build the FreeBSD kernel. This is a multiple-step, but straightforward, process. It consists of the following steps.

Config

First, you must change your directory to the kernel config directory. Then run the config on your configuration file.

```
# cd /sys/i386/conf
# config DIO
```

If you see error messages, you must fix your kernel config file before going

on to the next step. Generally, if you see the following message, you may proceed to the next step.

```
# Don't forget to do a ``make depend''
# Kernel build directory is ../../compile/DIO
```

To proceed to build your kernel, change the directory to the DIO kernel build directory.

```
# cd ../../compile/DIO
```

Starting Clean

It's always a good idea to start with a clean build and remove any stale object files. We'll clean any object files created in a prior build. Cleaning the build directory will cause the kernel build time to increase, but it's better to start with a fresh set of object files, in case there is a problem with the new kernel. You can eliminate the possibility of building with stale files.

Cleaning object files consists of running the `make clean` command in the kernel build directory.

```
# make clean
```

Running `make clean` will display lots of output. It's ok, as it's cleaning up object files and dependency files for the kernel.

Generating Dependencies

The kernel is a complex program `make` needs to generate file dependencies. Config has done most of the work for you already and stored the method for generating dependencies in the `make` file. Running `make depend` in the kernel build directory generates the dependencies.

```
# make depend
```

Running `make depend` will also generate lots of output.

Building the Kernel

You're almost there! Now, it's time to build the kernel. `make` is ready to build all the sources and link the necessary files and libraries to make your custom kernel. To do this, run the `make` command in the kernel build directory.

```
# make
```

As with previous kernel building commands, `make` will generate lots of output. It is important to notice error messages during this process; although if there are any serious error messages, the build process stops.

Installing the Kernel

After the kernel is successfully built, it must be installed. The `makefile` contains an install option for installing the kernel and any kernel modules that may have been built as part of the kernel build process. Typing `make install` in the kernel build directory will install the kernel and any associated kernel modules.

```
# make install
```

Here, `make install` performs two important steps. First, it installs the new kernel into the root partition. Second, and more importantly, it creates a copy of your current kernel, named `kernel.old`, in the root partition. This is done in case your new kernel doesn't boot or contains a fatal error. During the boot process, you can select to boot from `kernel.old` at the `boot:` prompt at system startup. This could keep you from having to reinstall your entire system, if there is an error.

Summary

In this chapter, we discussed and reviewed the components of the GENERIC kernel configuration file. In addition, we looked at the DIO appliance hardware and created a custom kernel configuration file for our appliance. In the next chapter, we will look at booting and FreeBSD system startup.

System Startup

Overview

One of the most important aspects of an embedded system development is the system startup and boot process. There are numerous reasons for understanding the booting and system startup process, such as adding the initialization of a custom component to the boot sequence, understanding what types of devices are bootable for system specification, or starting components or ensuring that components of the embedded system are started, as in the case of the DIO appliance. This chapter focuses on the FreeBSD booting process.

First, a quick look at the PC booting process will provide the background for the three FreeBSD boot stages. After the discussion of the FreeBSD boot process, the chapter continues with a discussion of the FreeBSD booting stage components and system startup. Once the discussion of the boot process is complete, we'll add code to start the DIO daemon and load the `copymem` system call and DIO device driver developed in previous chapters.

Disk Geometry

The boot process for each different device is slightly different. The DIO appliance boot device consists of a CompactFlash card, which appears as a hard disk. Since our discussion assumes booting from a hard disk, it is tailored to that technique. Before we jump into a discussion of the boot process, it will help to understand the layout of the disk and where the pieces of loader reside on the disk.

Platters

A hard disk consists of flat round disks called platters. Each platter is coated with a special material designed to store information in the form of magnetic patterns. They are composed of two main substances: a substrate material that forms the bulk of the platter and gives it structure and rigidity, and a magnetic media coating that actually holds the magnetic impulses that represent the data.

Heads

The read/write heads of the hard disk are the interface between the magnetic physical media on which the data is stored and the electronic components that make up the rest of the hard disk. The heads handle converting bits to magnetic pulses and storing them on the platters and reversing the process to read the data back.

Cylinders

Platters are organized into specific structures to enable the organized storage and retrieval of data. Each platter contains information recorded in concentric circles called cylinders. A cylinder is similar in structure to the annual rings of a tree.

Sectors

Each cylinder is further broken down into smaller pieces called sectors. Each sector holds 512 bytes of information.

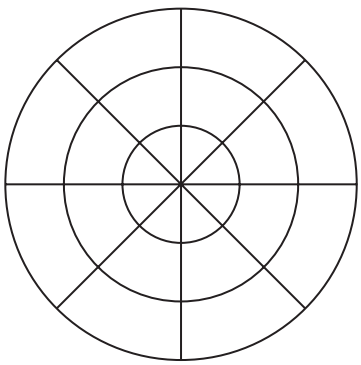


Figure 11-1. Disk Sectors

Addressing

Each sector on a disk is addressed using a format of head, cylinder and sector. For example, the first sector on a disk resides at head 0, cylinder 0, sector 1.

Master Boot Record

In the first sector of a hardware disk is the Master Boot Record (MBR). The MBR is a special record that contains the information for loading the operating system. The MBR resides on the first sector of the hard disk, head 0, cylinder 0, sector 1 and is one sector long, 512 bytes. The MBR contains three components: the boot loader, partition table and a magic number. The boot loader contained in the MBR is the first piece of code executed by the BIOS.

Boot Loader

The boot loader begins at offset 0 and is 446 bytes long. The boot code reads the partition table, also contained in the MBR, and attempts to load the first sector of the active partition in the partition table. Only one partition can be active.

Partition Table

Beginning at offset 1BEH is the partition table. The partition table contains four entries of 16 bytes each that describe how the disk is partitioned. The partition table entries begin at offset 1BEH, 1CEH, 1DEH and 1EEH in the MBR.

Each entry contains information about the partition such as beginning and ending position on the disk, what type of partition, and length in sectors. A description of each partition table entry is contained in Table 11-1.

Offset	Content
00H	Partition state: 00H if not active, 80H if active
01H	Head where partition begins
02H	Sector where partition begins
03H	Cylinder where partition begins
04H	Partition type
05H	Head where partition ends
06H	Sector where partition ends
07H	Cylinder where partition ends
08H	Distance from the MBR to the first sector of the partition, in sectors
0CH	Length of the partition, in sectors

Table 11-1

The partition table contains up to four distinct sections of the disk; partitions may not overlap. A common use of partitions is to configure a computer to boot multiple operating systems. Each partition may have a different operating system loaded.

State

The first entry in the partition is the state, which is either value 0 or 80H. A value of 80H denotes that this partition is active and can be booted.

Start of partition

The next three bytes contain the start of the partition in head, sector, cylinder format. Disks traditionally access storage by head, cylinder and sector offset. With the constant increase in capacity of hard drives, the space reserved in the MBR became too small to address a complete hard drive. To handle this issue, a new addressing scheme was developed that used the bits in the head, cylinder and sector fields. This new scheme is called Logical Block Addressing (LBA).

Let's look at an example using the start of the partition offset 2, the sector, and 3, the cylinder. The address of the starting sector is computed using the offsets 2 and 3 from the partition table and a few bitwise operations.

```
Cylinder  = (Offset(3) | ((Offset(2) & C0H) << 2);
Sector = Offset(2) & 3FH;
Start = Cylinder | Sector;
```

The address of the sector is computed by combining all 8 bits of the cylinder contained in offset 3 with the upper 2 bits of the sector value. These 10 bits contain the upper 10 bits of the address of the sector. The lower 6 bits of the sector in offset 2 contain the sector within the cylinder. The address is the combination of the computed cylinder and the computed sector.

Type

The type of the partition represents the file system type. A few of the common types of partitions are contained in Table 11-2.

Type	Description
00H	Empty
01H	DOS 12 bit FAT
04H	DOS 16 bits
05H	Extended partition
82H	Linux Swap
83H	Linux Native
A5H	BSD
B7H	BSDI
B8H	BSDI swap

Table 11-2

End of partition

The next three bytes contain the end of the partition in head, sector, cylinder format. The logical block address (LBA) of the partition end is computed using the same method as the start of the partition.

Distance from MBR

The next four bytes represent the LBA of the partition. The LBA is the sector offset from the beginning of the disk to the beginning of the partition.

Length

The last four bytes contain the size of the partition in sectors.

Magic Number

The magic number is AA55H and is located at offset 1FEH. Whenever the MBR is read, the magic number is read and tested to make sure the sector read contains the value AA55H.

An Example

Let's take a look at the first sector on my development machine. We'll use two utilities, dd and hexdump, to read and display the contents of sector 1 track 0, the MBR.

```
# dd if=/dev/ad0s1a of=boot.bin count=1
1+0 records in
1+0 records out
512 bytes transferred in 0.026990 secs (18970 bytes/sec)
# hexdump -C -v boot.bin
00000000  eb 3c 00 00 00 00 00 00  00 00 00 00 02 00 00 00  |.<..............|
00000010  00 00 00 00 00 00 00 00  12 00 02 00 00 00 00 00  |................|
00000020  00 00 00 00 00 16 1f 66  6a 00 51 50 06 53 31 c0  |.......fj.QP.S1.|
00000030  88 f0 50 6a 10 89 e5 e8  c7 00 8d 66 10 cb fc 31  |..Pj.......f...1|
00000040  c9 8e c1 8e d9 8e d1 bc  00 7c 89 e6 bf 00 07 fe  |.........|......|
00000050  c5 f3 a5 be ee 7d 80 fa  80 72 2c b6 01 e8 67 00  |.....}...r,...g.|
00000060  b9 01 00 be be 8d b6 01  80 7c 04 a5 75 07 e3 19  |.........|..u...|
00000070  f6 04 80 75 14 83 c6 10  fe c6 80 fe 05 72 e9 49  |...u.........r.I|
00000080  e3 e1 be ac 7d eb 52 31  d2 89 16 00 09 b6 10 e8  |....}.R1........|
00000090  35 00 bb 00 90 8b 77 0a  01 de bf 00 b0 b9 00 ac  |5.....w.........|
000000a0  29 f1 f3 a4 29 f9 30 c0  f3 aa e8 03 00 e9 60 13  |)...).0.......`.|
000000b0  fa e4 64 a8 02 75 fa b0  d1 e6 64 e4 64 a8 02 75  |..d..u....d.d..u|
000000c0  fa b0 df e6 60 fb c3 bb  00 8c 8b 44 08 8b 4c 0a  |....`......D..L.|
000000d0  0e e8 53 ff 73 2a be a7  7d e8 1c 00 be b1 7d e8  |..S.s*..}.....}.|
000000e0  16 00 30 e4 cd 16 c7 06  72 04 34 12 ea 00 00 ff  |..0.....r.4.....|
000000f0  ff bb 07 00 b4 0e cd 10  ac 84 c0 75 f4 b4 01 f9  |...........u....|
00000100  c3 52 b4 08 cd 13 88 f5  5a 72 f5 80 e1 3f 74 ed  |.R......Zr...?t.|
00000110  fa 66 8b 46 08 52 66 0f  b6 d9 66 31 d2 66 f7 f3  |.f.F.Rf...f1.f..|
00000120  88 eb 88 d5 43 30 d2 66  f7 f3 88 d7 5a 66 3d ff  |....C0.f....Zf=.|
00000130  03 00 00 fb 77 44 86 c4  c0 c8 02 08 e8 40 91 88  |....wD.......@..|
00000140  fe 28 e0 8a 66 02 38 e0  72 02 88 e0 bf 05 00 c4  |.(..f.8.r.......|
00000150  5e 04 50 b4 02 cd 13 5b  73 0a 4f 74 1c 30 e4 cd  |^.P....[s.Ot.0..|
00000160  13 93 eb eb 0f b6 c3 01  46 08 73 03 ff 46 0a d0  |........F.s..F..|
00000170  e3 00 5e 05 28 46 02 77  88 c3 2e f6 06 ba 08 80  |..^.(F.w........|
00000180  0f 84 79 ff bb aa 55 52  b4 41 cd 13 5a 0f 82 6f  |..y...UR.A..Z..o|
```

```
00000190  ff 81 fb 55 aa 0f 85 64  ff f6 c1 01 0f 84 5d ff  |...U...d......].|
000001a0  89 ee b4 42 cd 13 c3 52  65 61 64 00 42 6f 6f 74  |...B...Read.Boot|
000001b0  00 20 65 72 72 6f 72 0d  0a 00 80 90 90 90 00 00  |. error.........|
000001c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 80 00  |................|
000001f0  01 00 a5 ff ff ff 00 00  00 00 50 c3 00 00 55 aa  |..........P...U.|
```

We can see the last two bytes contain a valid magic number, AA55H. They are reversed in the display because the x86 is little endian architecture and hexdump is displaying the output in bytes, which reverses the order.

Let's use a more focused version of the hexdump command to dump just the bytes of the partition table.

```
# hexdump -C -s 0x1be -v boot.bin
000001be  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001ce  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001de  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
000001ee  80 00 01 00 a5 ff ff ff  00 00 00 00 50 c3 00 00  |............P...|
```

We can see from the hexdump of the MBR on my machine, the last entry in the partition table contains a FreeBSD partition which is active and is the length of the complete disk. Let's take a more detailed look at our boot partition located at offset 1EEH.

Offset 0 shows this is the active partition, 80H, and offset 4 shows this is a FreeBSD partition, A5H. Offsets 1, 2 and 3 give us the starting sector of this partition in head, sector, cylinder format. This tells us that the partition starts with the MBR. Offsets 5, 6 and 7 give us the ending sector of the partition in head, sector and cylinder format. The values of FFH in all these fields denote that the partition uses the whole disk. Offsets 8-B give us the distance from the MBR to the beginning of the partition; this value is 0. Once again, denoting the partition begins with the MBR. The final four bytes of the partition table contain the number of sectors in the partition.

Boot Slice

Now that we've identified the partition table, we'll take a look at how FreeBSD uses it. Within a disk partition, FreeBSD creates a slice, which is used by FreeBSD to implement the traditional Unix filesystem of 8 partitions, `a` through `f`. As you may have noticed, the term "partition" is used to represent numerous things, which is why the term "slice" was brought into play.

Let's define a few terms to avoid confusion.

A slice is a section of a disk; each disk contains at most four slices. The slices are defined by a table contained in the MBR.

A partition is a section of a slice. Each partition may contain a file system or swap space in FreeBSD.

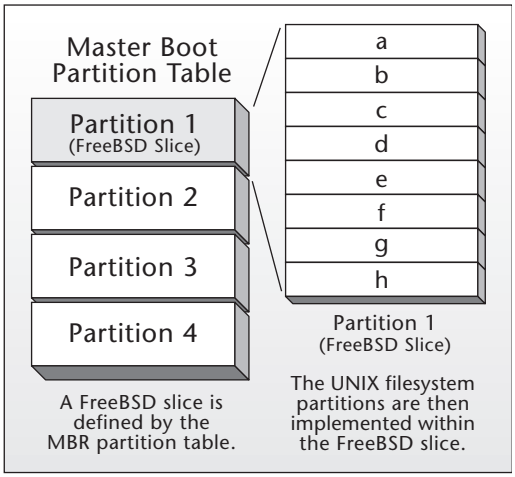


Figure 11-2. Partition Table and Slice

Unix Partitions

A Unix disk is divided into as many as 8 partitions. A partition represents a separate entity on the disk. FreeBSD disks typically use 4 of the 8 available partitions: the `a` partition is used for the root file system, the `b` partition is used for the swap partition. The `f` partition contains the user partition, and the `e` partition is used for the var filesystem. An important note is the `c` partition, which historically is used to represent the whole disk.

We can look at the /etc/fstab file to see how the disk is partitioned.

```
/dev/ad0s1b     none        swap    sw          0     0
/dev/ad0s1a     /           ufs     rw          1     1
/dev/ad0s1f     /usr        ufs     rw          2     2
/dev/ad0s1e     /var        ufs     rw          2     2
```

This disk contains four partitions. The partition is denoted by the last letter of the device name in column 1. Partition a contains the root file system, `b` the swap partition, `f` the user partition and `e` the var partition.

PC BIOS

After turning on or resetting your computer program, execution begins with generic code contained in the PC called the BIOS. The BIOS is the lowest level software in a computer and provides an interface between the software and the hardware. The BIOS (basic input/output system) has the task of initializing the hardware, loading and running the boot loader contained in the MBR.

The boot loader is a program that resides in the MBR and is loaded and run by the BIOS. The BIOS loads it into memory and begins program execution. The information contained in the MBR includes information to find the boot loader on disk, a program to read the boot loader into memory and begin execution.

Once a PC is powered on, the BIOS has a list of tasks to perform:

1. A series of tests are performed on existing hardware to ensure the hardware is working properly.
2. Hardware resources are initialized and assigned.
3. Configured boot devices are searched for a valid boot sector.
4. The boot sector is loaded and control is transferred to the boot loader.

After the system boots, the BIOS reads the MBR into location 7C00H; the last two bytes of that sector should contain the MBR magic number AA55H. If the last two bytes are AA55H, control is passed to the boot loader routine; otherwise the system stops.

FreeBSD Boot Loader

Once the BIOS loads the MBR into memory, the FreeBSD booting process begins. It consists of three stages, each stage providing more features and increasing in size. The first two stages are actually part of the same program but are split into two due to space constraints. The third stage is an options boot loader. Individual components of the boot process are located in the /boot directory.

```
# ls -l /boot
total 533
-r—r—r—      1 root  wheel      512  Sep 18 13:28 boot0
-r—r—r—      1 root  wheel      512  Sep 18 13:28 boot1
-r—r—r—      1 root  wheel     7680  Sep 18 13:28 boot2
-r-xr-xr-x   1 root  wheel   149504  Sep 18 13:28 cdboot
drwxr-xr-x   2 root  wheel      512  Oct 24 16:52 defaults
-r-xr-xr-x   1 root  wheel   147456  Sep 18 13:28 loader
-r—r—r—      1 root  wheel     9237  Sep 18 13:28 loader.4th
-rw-r—r—     1 root  wheel       67  Dec 16 16:51 loader.conf
-r—r—r—      1 root  wheel    12064  Sep 18 13:28 loader.help
-r—r—r—      1 root  wheel      338  Sep 18 13:28 loader.rc
```

```
-r—r—r—      1 root  wheel      512  Sep 18 13:28 mbr
-r-xr-xr-x   1 root  wheel   149504  Sep 18 13:28 pxeboot
-r—r—r—      1 root  wheel    25121  Sep 18 13:28 support.4th
```

The source code that represents the FreeBSD boot stages resides in /sys/i386/biosboot directory. For a dedicated FreeBSD system, the boot code contained in the MBR resides in /boot/boot0. The source code for boot0 resides in /sys/boot/i386. Let's take a closer look at the boot stages.

boot1 and boot2

The first stage is loaded by the MBR to location 7C00H and is limited to 512 bytes. This first boot copies itself to 10000H and loads the second-stage boot into memory. The second-stage boot resides in the first 15 sectors of the boot slice, bringing the total of the first and second boot to 16 sectors or 8K bytes. The first and second stages of the boot process are actually built together so boot1 knows exactly where boot2 starts execution and calls that entrypoint.

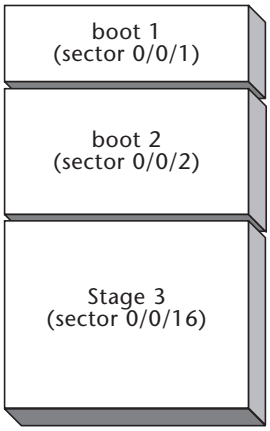


Figure 11-3. Boot Stages

Stage two consists of the boot2 program, which understands how to read the FreeBSD file system so it can find the files necessary to boot and provides a simple interface to the user to choose the kernel or loader to run. The second stage loads the third stage loader into memory before passing control to the third stage, /boot/loader. The source code for boot1 and boot2 is found in /sys/boot/i386.

Stage 3

Loader, /boot/loader, is started by the second-stage bootstrap loader. The third-stage loader copies the kernel into memory and starts executing it. The kernel is loaded from the FreeBSD file system so the third-stage loader has the information to read the filesystem. Loader uses configuration files contained in the /boot directory for load options and parameters. The files, /boot/loader.conf, /boot/loader.rc are parsed for load options. The loader copies the kernel image into memory and passes parameters to the kernel via the stack.

System Startup

After the kernel is loaded and the system starts up, the kernel creates a user daemon to complete initialization call init, which is PID 1.

init

Once the kernel is loaded, control is passed to the init daemon. The init daemon is responsible for transitioning through the different user-mode levels and starting resources, file systems, networking daemons and configuration. Init is responsible for making sure the file systems are consistent and starting system daemons and initializing terminals for user login based on the resource configuration files.

The resource configuration files are executed by a main Bourne Shell script, /etc/rc. The rc script reads a series of files that contain system configuration code. The rc file doesn't need modifications; its behavior can be changed by setting and clearing variables in the /etc/rc.conf and /etc/defaults/rc.conf files. Let's take a closer look at the compoents of the rc file that are relevant to the DIO appliance.

Configuration

One of the first tasks of the init daemon is to read two files that contain global configuration files for the system that enable and disable systems daemons started by init. The first file /etc/defaults/rc.conf is a global confiuration file and should not be changed. The second file /etc/rc.conf is a system-tuneable file. Variables may be set in /etc/rc.conf to override the values in /etc/defaults/rc.conf.

```
# If there is a global system configuration file, suck it in.
#
if [  -r /etc/defaults/rc.conf ]; then
    . /etc/defaults/rc.conf
    source_rc_confs
elif [  -r /etc/rc.conf ]; then
    . /etc/rc.conf
fi
```

We see the rc file checks for the existence of the rc.conf files. If they exist, they are read.

In Chapter 7 we modified the rc.conf to ensure the SSH daemon sshd was started by rc.

System Settings

One of the configuration files is for system tuning. The sysctl utility is used to tune FreeBSD kernel parameters in a running system. The rc script reads the settings in /etc/rc.sysctl and executes these statements using sysctl.

```
# Set sysctl variables as early as we can
#
if [  -r /etc/rc.sysctl ]; then
    . /etc/rc.sysctl
fi
```

The sysctl utility is used for parameter tuning. In our case, since we're booting from a flash device, the number of writes must be limited. One of the ways to do this is to disable swapping. Swapping is the method of moving unused pages of memory to disk, to free memory for executing programs. Because the DIO appliance is a dedicated system, swapping is not necessary.

Our rc.sysctl file contains the following line:

```
swap_enabled=0
```

Customization

One of the last tasks rc performs is to look for user-defined startup scripts. The rc script searches a defined directory looking for files that end in the suffix .sh. The local_startup variable is set by /etc/defaults/rc.conf, which defines the local path to search. The default value is /usr/local/etc/rc.d.

```
# For each valid dir in $local_startup, search for init scripts
# matching *.sh
#
case ${local_startup} in
[Nn][Oo] | '')
    ;;
*)
    echo -n 'Local package initialization:'
    slist=""
```

```
    for dir in ${local_startup}; do
        if [ -d "${dir}" ]; then
            for script in ${dir}/*.sh; do
                slist="${slist} ${script_name_sep} ${script}"
            done
        fi
    done
    script_save_sep="$IFS"
    IFS="${script_name_sep}"
    for script in ${slist}; do
        if [ -x "${script}" ]; then
            (set -T
            trap 'exit 1' 2
            ${script} start)
        fi
    done
    IFS="${script_save_sep}"
    echo '.'
    ;;
esac
```

Starting DIO Components

Up to this point, we've discussed the FreeBSD boot process. In order for the DIO appliances to run correctly, the components developed in the previous chapters must be loaded and started. From the previous section we've discovered that the rc script looks in /usr/local/etc/rc.d for scripts that have the suffix .sh and runs those at system start. Let's take a look:

```
-r-xr-xr-x  1 root  wheel  504 Dec 10 07:39 tomcat.sh
```

We'll create a file, dio.sh, and put it in /etc/local/etc. All local scripts contain the same format. Each script is a Bourne Shell script.

The dio.sh Script

In addition to the standard system daemons, the DIO appliance will load the copymem system call, the DIO device driver, and start the diod daemon. In order to accomplish this, we've added a script, diosh, to the /usr/local/etc/rc.d directory. Let's take a look at the code.

```
#!/bin/sh

case "$1" in
    start)
        if [  -f /modules/copymem.ko ]; then
            kldload copymem.ko
        fi
        if [  -f /modules/dio.ko ]; then
            kldload dio.ko
        fi
        if [  -f /usr/local/dio/bin/diod ]; then
            /usr/local/dio/bin/diod > /dev/null && echo ' diod'
            ps -agx | grep "/usr/local/dio/bin/diod" | awk '{
print $1 }' > /var/run/diod.pid
        fi
        ;;
    stop)
        kill -9 `cat /var/run/diod.pid`
        ;;
    *)
        echo ""
        echo "Usage: `basename $0` {  start | stop }"
        echo ""
        exit 64
        ;;
esac
```

The first line starts the Bourne Shell. The dio.sh script is called with a parameter `start`, during system startup, or `stop`, during system shutdown.

During system startup, the dio.sh script performs three tasks. First, if the copymem module exists, it loads it using the KLD loader. Next, if the DIO device driver exists, then it also loads it using the KLD loader. Finally, if the diod daemon exists, it is started. Once the diod daemon is started, the PID of the diod daemon is saved in /var/run/diod.pid. It is common practice to save the PID of a daemon in the /var/run directory so the daemon can be killed on system shutdown.

The next case is for system shutdown. During system shutdown the diod daemon is killed. The PID is retrieved from the /var/run/diod.pid file created during system initialization.

Summary

In this chapter we've taken a look at the PC booting process and the stages of booting FreeBSD, and we've added the DIO components developed in previous chapters to the FreeBSD system startup. Our system will now be started automatically and be ready for use after each reboot. Now, on to the next chapter where all the pieces will be tied together and built into a CompactFlash device.

The CompactFlash Boot Device

Overview

This chapter focuses on creating a boot device from a solid-state device, specifically, the Sandisk 32MB CompactFlash device. Solid-state devices provide increased stability due to the lack of moving parts. However, due to the limited disk space and write capacity, some basic system-level issues need to be addressed, such as limiting writes to the CompactFlash boot device, not using swap space, and running with memory file systems.

Specific topics that will be covered in this chapter include

- Solid-state devices
- Installing and verifying the TARC CompactFlash Adapter
- Configuring the CompactFlash device
- CompactFlash system startup issues

Solid-state Devices

A CompactFlash device is a nonvolatile solid-state device used for storage. To the FreeBSD kernel, a CompactFlash device appears as an IDE disk drive. An important consideration for using solid-state devices is that each sector has a limited write capacity. For this reason an embedded system typically uses the CompactFlash device to boot the system, and then the system executes out of a memory file system. Also, there is no swap partition configured on a CompactFlash device.

Installing the TARC CompactFlash Adapter

In order to use a CompactFlash device as a FreeBSD boot device, the DIO appliance must have a CompactFlash adapter. The Tucson Amateur Radio Club (TARC) distributes such an adapter. More information on this adapter can be found at http://www.tapr.org. The TARC CompactFlash adapter allows any Type I or Type II Compact Flash device to be used as a standard IDE drive.

The TARC CompactFlash adapter uses an IDE connection, a standard 3.5" floppy driver power connector and a CompactFlash device. During development, I chose to connect the CompactFlash adapter as the primary slave device, as this configuration allows a simple configuration and test setup. Once the development life cycle is complete, the Compact Flash boot adapter will be configured as the primary boot device.

Before making any connections, be sure to shut down and power off your system. After the IDE connection is complete, connect the 3.5-inch power connector to the TARC CompactFlash adapter. Once the power and IDE connections are complete, install the CompactFlash memory into the TARC adapter.

Once the physical installation is complete, we'll verify that the CompactFlash has been installed and is working properly. Since the CompactFlash device appears as a standard IDE device, FreeBSD should recognize the device using a standard kernel. We can verify this by using the `dmesg` command and looking at the output of the probed devices during system startup.

```
# dmesg

[selected output]
ata0-master: DMA limited to UDMA33, non-ATA66 compliant cable
ad0: 12419MB <ST313021A> [ 25232/16/63] at ata0-master UDMA33
ad2: 30MB <SunDisk SDCFB-32> [ 490/4/32] at ata1-master PIO1
acd0: CDROM <CD-912E/ATK> at ata0-slave using PIO3
```

Looking at the selected output, we can see the CompactFlash is detected and present at device ad2. Now that we have successfully installed the CompactFlash adapter, we'll look at configuring the device and loading our DIO appliance software.

Configuring the CompactFlash Device

Configuring the CompactFlash device is similar to configuring any other boot device for FreeBSD. The development hardware we're using has the development disk installed as the primary master device (ad0) and the CompactFlash installed as the secondary master device (ad2).

To create the initial configuration for the CompactFlash device, we'll use the standard tools for creating a FreeBSD installation. Because the CompactFlash device appears as an IDE disk, all the tools run normally. For our first task, we will partition and format the CompactFlash device so we can load the DIO appliance software. To get started, change directory to the /stand directory and run `sysinstall`.

```
# cd /stand
# ./sysinstall
```

Partitioning the CompactFlash Device

After starting `sysinstall`, choose custom from the installation menu, then choose partition. `Sysinstall` will ask you to select the drive to partition. Select ad2, the CompactFlash device.

In the partition menu, delete any existing partitions using the d option. After all the existing partitions are deleted, create a new partition using the c option. The size of the partition should be the default size, 62720 sectors, and the type should be 165, a FreeBSD partition. After creating a partition, choose the w option to write this to the flash device. When you are prompted, if you are absolutely sure, choose `[ Yes ]`.

After choosing to write the partition information to the CompactFlash, you then will be prompted to install the boot manager. You should choose the FreeBSD Boot Manager. The CompactFlash device is now partitioned. You can exit from the partition menu by choosing the q option.

Creating the Disklabel

The next step is to create a disklabel in the existing partition. Select the label options from the sysinstall menu. In the label menu, create a partition that will be mounted as the root partition, /, that consists of the entire space available.

```
/dev/ad2s1a /   30MB    UFS
```

Once this is complete, write it out to the CompactFlash device. You may exit the `sysinstall` utility. Typically a FreeBSD installation uses multiple partitions, such as root and swap and var. However, since the DIO appliance does not use swap or the var filesystem, those partitions are not necessary.

Formatting the File System

The CompactFlash is now almost ready for prime time. The next step is to format the file system, accomplished using the `newfs` command.

```
# newfs /dev/ad2s1a
Warning: 2848 sector(s) in last cylinder unallocated
/dev/ad2s1a:    62688 sectors in 16 cylinders of 1 tracks, 4096
sectors
    30.6MB in 1 cyl groups (16 c/g, 32.00MB/g, 7616 i/g)
super-block backups (for fsck -b #) at:
 32
```

Upon completion of the `newfs` command, the CompactFlash device can be mounted. Once the mount is completed, files can be copied and the system boot testing can begin.

Mounting the File System

With the CompactFlash partitioned and file system formatted, the Compact-Flash can now be mounted. Mounting the device is accomplished using the `mount` command.

```
# mount /dev/ad2s1a /flash
```

After mounting the device, we can verify that it is properly mounted by displaying the file system using the `df` command.

```
# df /flash
Filesystem   1K-blocks      Used    Avail Capacity  Mounted on
/dev/ad2s1a       30359         1    27930     0%    /flash
```

Using the output of `df` we can see that the CompactFlash device, /dev/ad2s1a, is mounted on /flash and contains just under 30 MB of disk capacity.

Copying the Files to the Boot Device

We've now come to the final step of creating the boot device. The CompactFlash is partitioned, formatted and mounted, and it's time to copy the files from our development disk to the CompactFlash device. Creating a system image for an embedded device is somewhat of a "black art." There are a variety of ways to determine the components of your embedded system. I'll briefly describe two methods here to be used as a guideline for creating your final image. Whether you use these methods or create your own method, no amount of experience can circumvent the often unheralded and underappreciated—but extremely critical, particularly for an embedded system project—step of creating the final image. No amount of preparation can replace the time-consuming process of iteration and testing.

The Iterative Approach

Our system is configured so that we can iterate and test the CompactFlash device. Copy the required files to the flash disk. Once you're ready to test the CompactFlash device, type the space bar at the boot prompt and enter ad(2,a). This causes a boot from the CompactFlash device. If there are files missing, reboot the system normally, make the necessary changes and try again.

The Installation Approach

Another way to develop your release image is to install FreeBSD in a conventional manner to a hard drive, add your application software and then verify your system is working as expected. Then pare your system to the size required by your boot device. After paring down and testing your system as required, dd the entire filesystem to be transferred to your boot device.

Startup Configuration

Much of the diskless boot is handled by the `rc.diskless2` script. Control of a diskless FreeBSD system is handled by /etc/rc.diskless2. In order for rc.diskless2 to be invoked, the following line must be added to /etc/rc.conf:

```
diskless_mount=/etc/rc.diskless2
```

Let's take a look at the rc.diskless2 script in Listing 12-1.

```
# Copyright (c) 1999  Matt Dillon
# All rights reserved.
#
# Redistribution and use in source and binary forms, with or
# without modification, are permitted provided that the following
# conditions are met:
# 1. Redistributions of source code must retain the above
#    copyright notice, this list of conditions and the
#    following disclaimer.
# 2. Redistributions in binary form must reproduce the above
#    copyright notice, this list of conditions and the following
#    disclaimer in the documentation and/or other materials
#    provided with the distribution.
#
# THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS
# IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT
# NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
# FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT
# SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT,
# INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
# DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
# SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
# OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
# LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
# (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF
# THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY
# OF SUCH DAMAGE.
#
# $FreeBSD: src/etc/rc.diskless2,v 1.5.2.8 2001/07/24 09:49:37 dd
# Exp $
#

#
# rc.diskless2
#

# Provide a function for normalizing the mounting of memory
# filesystems. This should allow the rest of the code here to
# remain as close as possible between 5-current and 4-stable.
#    $1 = size
#    $2 = mount point
#    $3 = md unit number (ignored in pre 5.0 systems)
#    $4 = (optional) bytes-per-inode
mount_md() {
    if [  -n "$4" ]; then
```

```
        bpi="-i $4"
    fi
    /sbin/mount_mfs -s $1 -T qp120at $bpi dummy $2
}

# If there is a global system configuration file, suck it in.
#
if [  -r /etc/defaults/rc.conf ]; then
    . /etc/defaults/rc.conf
    source_rc_confs
elif [  -r /etc/rc.conf ]; then
    . /etc/rc.conf
fi

echo "+++ mfs_mount of /var"
mount_md ${varsize:=65536}  /var 1

echo "+++ populate /var using /etc/mtree/BSD.var.dist"
/usr/sbin/mtree -deU -f /etc/mtree/BSD.var.dist -p /var

echo "+++ create log files based on the contents of /etc/newsys
           log.conf"
LOGFILES=`/usr/bin/awk '$1 != "#" { printf "%s ", $1 } '
/etc/newsyslog.conf`
if [  -n "$LOGFILES" ]; then
    /usr/bin/touch $LOGFILES
fi

mount -a        # chown and chgrp are in /usr

#
# XXX make sure to create one dir for each printer as requested
#by lpd
#

# If /tmp is a symlink, assume it points to somewhere writable,
# like /var/tmp, otherwise, use a small memory filesystem for
# /tmp.
if [  ! -h /tmp ]; then
    mount_md ${tmpsize:=20480}  /tmp 2
fi

# extract a list of device entries, then copy them to a writable
# fs
```

```
(cd /; find -x dev | cpio -o -H newc) > /tmp/dev.tmp
mount_md 4096 /dev 3 512
(cd /; cpio -i -H newc -d < /tmp/dev.tmp)
```

Listing 12-1

Listing 12-1 shows the code for the rc.diskless2 scripts provided with the FreeBSD 4.4 release. This script handles booting a diskless system and handles the special requirements for that type of system.

First, rc.diskless2 reads in the global configuration rc.conf files. Next the /var directory is mounted with a default size of 65536 sectors. Following the creating of var in memory, the directory structure for var is created by the mtree command and necessary log files are created based on the contents of /etc/newsyslog.conf. With the var file system created and the necessary log files created, the file systems can be mounted via the mount command. Next, the /tmp directory is created in memory with a default size of 20480 sectors. Finally the /dev is created in memory and then populated.

As you can see, many of the details of booting a diskless system are handled by the rc.diskless2 script. With a few custom modifications, our system will be ready for prime time. Let's take a closer look at some of the settings.

Configuring Read-only File Systems

The /var directory is used for many temporary files and log files. Once you have configured your system to run rc.diskless2 system, the /var directory will be mounted as a memory file system. One of the parameters contained in the rc.diskless2 script is varsize. The varsize variable represents the size, in sectors, for the /var directory. The default is 65536, larger than available memory. We'll set varsize to a value that better represents the requirements of the DIO appliance.

```
varsize=8192
```

As with the /var directory, the /tmp directory is created by the rc.diskless2 script. The default size is 10480 sectors. We'll set this to 8192 sectors, as this value better represents the DIO appliance's requirement. The size of the tmp directory is controlled by the tmpsize variable.

```
tmpsize=8192
```

With the /var and /tmp directories created in memory, we can now change the mounting of the root directory to read-only. Changing the mounting options for the root directory requires a modification to the /etc/fstab file.

```
/dev/ad2s1a          /           ufs ro          1     1
```

Mounting the root partition, /, as read only ensures that the DIO appliance application does not write to the CompactFlash device.

Summary

This chapter presents the details of creating a FreeBSD image that boots from CompactFlash. Using CompactFlash as a boot device makes the process of creating a boot image easier because the kernel does not need any special device drivers or configuration options. The CompactFlash device in conjunction with the TAPR CompactFlash adapter appears as an IDE disk. Depending on the application and product, there are other devices available as boot devices, such as PCCard memory and M Systems's DiskOnChip memory. These devices can be used with FreeBSD, but require more configuration steps.

The development of our DIO is now complete, and you should be comfortable using FreeBSD's many powerful features. In summary, let's take a look at a few other embedded appliances that use FreeBSD as the core embedded operating system.

The AMI StorTrends NAS is a networked attached-storage device that uses FreeBSD as its embedded operating system. The StorTrends NAS boots from a Flash device and provides access to storage via SMB/CIFS or NFS. Among other features are TCP/IP, DHCP, DNS, NTP, SMTP and SNMP connectivity and configuration. The StorTrends NAS is managed and configured remotely via a web browser.

The IBM InterJet II is another network appliance that uses FreeBSD as its embedded operating system. The InterJet II is a small network appliance whose features include: e-mail server, Apache, Firewall, FTP, DNS and DHCP services. Like the StorTrends NAS, the InterJet II is configured and managed using a web connection and web browser.

Juniper Networks develops cable IP services and systems. FreeBSD provides the foundation for their development of a next-generation routing architecture, as FreeBSD has the ability to scale and support the tremendous growth projections for the Internet.

These commercial products demonstrate the rich features and flexibility of FreeBSD for use with embedded applications.

The FreeBSD License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE FREEBSD PROJECT ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE FREEBSD PROJECT OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The views and conclusions contained in the software and documentation are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the FreeBSD Project or FreeBSD, Inc.

PCI Configuration

This chapter provides a description of the PCI bus and the PCI configuration registers. The PCI-DIO24 Digital IO card is a PCI bus data acquisition controller. In order for the FreeBSD DIO device driver to correctly detect the presence of the PCI-DIO24 controller, the device driver must read the PCI configuration registers. Access to the PCI configuration registers is conveniently hidden from the FreeBSD device driver writer though the use of the PCI kernel subsystem.

The PCI Bus

PCI is an acronym for Peripheral Component Interconnect. The PCI bus specification defines a system interconnect for computer components that require fast access to each other and/or system memory. A typical PCI device consists of a PCI interface controller on a PCI expansion card. Examples of PCI expansion cards are network controllers, display adapters, SCSI controllers and Fibre Channel host bus adapters. One benefit of the PCI specification is its vendor and platform independence. PCI buses are found in Intel, Apple and Sun computers.

The PCI Bus specification is managed by the PCI Special Interest Group (PCI-SIG). For information regarding the PCI Specification contact:

PCI Special Interest Group (PCI-SIG)
5440 SW Westgate Dr., #217
Portland, OR 97221
Phone: 503-291-2569
FAX: 503-297-1090
administration@pcisig.com

PCI Configuration Registers

This section provides a description of the PCI device configuration header and PCI configuration registers. Any functional PCI device contains a block of 64 double words called the PCI configuration header. The first 16 double words are defined by the PCI specification and are known as the configuration header region.

The PCI configuration header region can be in two formats known as type 0 or type 1. Header type 1 is used for PCI-to-PCI bridges. Header type 0 is for all other devices. Figure B-1 illustrates the format of type 0 PCI device configuration registers.

Offset	Byte3	Byte2	Byte1	Byte0
0	Device ID		Vendor ID	
4	Status Register		Command Register	
8	Class Code			Revision ID
12	BIST	Header Type	Latency Timer	Cache Line Size
16	Base Address 0			
20	Base Address 1			
24	Base Address 2			
28	Base Address 3			
32	Base Address 4			
36	Base Address 5			
40	Card CIS Pointer			
44	Subsystem ID		Subsystem Vendor ID	
48	Expansion ROM Base Address			
54	Reserved			
56	Reserved			
60	Max Latency	Min Grant	Interrupt Pin	Interrupt Line

Figure B-1. PCI Configuration Header

The subsequent sections describe each mandatory PCI configuration register that is implemented by all PCI devices. The most notable for device driver writers are the vendor ID and device ID registers. These registers are used to determine if the physical device is installed in the system.

Vendor ID Register

The Vendor ID Register is a 16-bit register that contains a unique value identifying the device manufacturer. Every PCI manufacturer registers with the PCI SIG to obtain a unique Vendor ID. The Measurement Computing Vendor ID is hexadecimal 1307.

Device ID Register

The Device ID register is a 16-bit register that contains a unique value, defined by the manufacturer, for this PCI device. The PCI-DIO24 Device ID is hexadecimal 28.

Command Register

The command register is a 16-bit register that provides control over the device's ability to respond to PCI accesses. Currently only bits 0:9 are defined. Each of the defined bits represents a capability. The PCI designer sets the bits that are implemented by the device. The bits are defined as follows:

Bit	Function
0	IO Access Enable
1	Memory Access Enable
2	Master Enable
3	Special Cycle Recognition
4	Memory Write and Invalidate Enable
5	VGA Palette Snoop Enable
6	Parity Error Response
7	Wait Cycle Enable
8	System Error Enable
9	Fast Back-to-Back Enable
10:15	Reserved

Status Register

The Status Register is a 16-bit register that provides the device status. The bits are defined in the status register as follows:

Bit	Function
0:4	Reserved
5	66 MHz Capable
6	UDF Supported
7	Fast Back-to-Back Capable
8	Data Parity Reported
9:10	Device Select
11	Signaled Target Abort
12	Received Target Abort
13	Received Master Abort
14	Signaled System Error
15	Detected Parity Error

Revision ID

The revision ID register contains an 8-bit value assigned by the manufacturer that contains the device revision number.

Class Code

The class code register is a 24-bit register that defines the base class, sub-class and programming interface.

0:7	8:15	16:23
Programming interface	Sub-Class Code	Class Code

The class code, sub-class code and programming interface are defined by the PCI Interface specification.

Cache Line Size

The cache line size register is an 8-bit register that defines the system cache line size in double word increments.

Latency Timer

The latency time is an 8-bit register that defines the minimum amount of time, in PCI clock cycles, that the bus master can retain ownership of the bus.

Header Type

The header type register is an 8-bit register that defines the type of device. Bit 7 is set to 0 or 1, defining single function or multi function.

BIST

The BIST register is an 8-bit register that can be used for PCI devices that implement Built In Self Test (BIST).

Base Address Registers

The base address registers are 32-bit registers used to determine the PCI memory mapped and IO spaces used by the device. A PCI device may have up to 6 base address registers that are used to utilize memory mapped or IO address space.

CardBus CIS Pointer

The CardBus CIS Pointer Register is a 32-bit register implemented by devices that share silicon between the cardbus and PCI bus.

Subsystem Vendor ID

The subsystem vendor ID is a 16-bit register used to uniquely identify an add-in card. The subsystem vendor ID is obtained from the PCI-SIG.

Subsystem ID

The subsystem ID is a 16-bit register used to define additional features for a PCI device. The subsystem ID is defined by the vendor.

Expansion ROM Address

For devices that that contain power-on self test (POST) code, BIOS and interrupt service routines, the Expansion ROM address register is a 32-bit register that contains the starting address and size of the ROM code.

Maximum Latency

The maximum latency register is an 8-bit register that specifies how often the device needs to access the PCI bus.

Minimum Grant

The minimum grant register is an 8-bit register that defines how long the master would like to retain PCI bus ownership whenever it initiates a transaction.

Interrupt PIN

The interrupt pin register is an 8-bit register that defines which of the 4 PCI interrupt request pins the PCI device is connected to.

Interrupt Line

The interrupt line register is an 8-bit register used to identify which of the system interrupt lines on the system interrupt controller the PCI device interrupt is routed on.

Kernel Loadable Modules

Overview

One of the features of FreeBSD is a dynamic kernel linker that provides system engineers and system administrators with the capability to load and unload drivers and system calls in a running system. This appendix provides the necessary background information for writing kernel loadable modules (KLDs). In addition to covering the details of the different types of FreeBSD KLDs, at the completion of this appendix you will have skeleton code that can be used for your own KLDs.

In this chapter we will cover

- The core components of a KLD
- System calls as KLD
- Device drivers as KLD
- KLD commands

Kernel Loadable Modules

Every KLD contains three core components: the Makefile, which provides a simple environment to build a KLD and provides the developer with a rapid prototype and build environment; the load handler function, which provides the entry points for the load, unload and system shutdown behavior for the KLD and the module data structure; and the module data structure, which contains the name and entry point of the load function.

When a new KLD is loaded into a running system, dynamic load execution of the KLD starts based on the entry point in the module data structure.

Makefile

Building a KLD is a straightforward process and the details of KLD building are provided in a system-includable makefile located in /usr/share/mk/bsd.kmod.mk. Your makefile declares the source files that consist of the KLD in the makefile variable SRCS and declares the name of the KLD in the variable KMOD by including the system makefile in the /usr/share/mk directory; the rest of the details are handled for you. Listed below is a sample makefile for a skeleton KLD.

```
SRCS=kld_generic.c
KMOD=kld_generic

.include <bsd.kmod.mk>
```

After the declaration of the SRCS and KMOD variables, a standard KLD makefile includes the KMOD makefile template, bsd.kmod.mk.

The Load Handler Function

The load handler function is called by the kernel dynamic linker when a module is loaded or unloaded or the system is shut down. The function prototype for the load handler function is defined in /usr/include/sys/module.h

```
typedef int (*modeventhand_t)(module_t mod, int
/*modeventtype_t*/ what, void *arg);
```

The load handler function takes three arguments: a module, the event and a user-defined argument.

```
static int load_handler(struct module *m, int cmd, void* arg)
{
    int stat = 0;

    switch(cmd)
    {
    case MOD_LOAD:
         break;

    case MOD_UNLOAD:
         break;
```

```
    case MOD_SHUTDOWN:
        break;

    default:
      stat = EINVAL;
    }

    return(stat);
}
```

The module pointer is a linked list of currently loaded modules and contains information relevant to loaded modules. Commands are defined by the modeventtype defined in /usr/include/sys/module.h. The cmd argument represents the reason why the load handler is being called. There are three conditions under which a KLD load handler is called.

```
typedef enum modeventtype {
     MOD_LOAD,
     MOD_UNLOAD,
     MOD_SHUTDOWN
}  modeventtype_t;
```

Command	Description
MOD_LOAD	The KLD is being loaded
MOD_UNLOAD	The KLD is being unloaded
MOD_SHUTDOWN	The system is shutting down

The last argument is a user-defined parameter. The arg parameter represents a void pointer that can be used by the KLD developer to pass information into the KLD.

The moduledata_t Structure

The moduledata_t structure contains the data to interface to the dynamic kernel loader. There are three elements in the moduledata_t stucture.

```
typedef struct moduledata {
    char        *name;  /* module name */
    modeventhand_t  evhand; /* event handler */
    void        *priv;  /* extra data */
} moduledata_t;
```

The name is a string used by the kernel linker for this module. The load handler function was discussed in the previous section and contains the KLD load function. The last element is a pointer used to pass user-defined data to the load handler.

```
static moduledata_t kld_generic_module =
{
  "kld_generic",              /* KLD name
*/
  load_handler,               /* event handler
*/
  NULL,                        /* private data passed to event
handler */
};
```

The DECLARE_MODULE Macro

The core pieces of a KLD are all united by the DECLARE_MODULE macro. DECLARE_MODULE takes four parameters. The first parameter represents a unique name for the kernel module. The second parameter is data for the type of load modules; the third argument is a subsystem type and the final parameter signifies load order.

```
#define DECLARE_MODULE(name, data, sub, order) \
    SYSINIT(name##module, sub, order, module_register_init,
&data) \
    struct __hack
```

The listing below contains a sample declaration for our generic module.

```
DECLARE_MODULE(kld_generic, kld_generic_module, SI_SUB_KLD,
SI_ORDER_ANY);
```

The first parameter is a generic name for the KLD, kld_generic. The second argument contains the KLD defined modeuledata_t structure. The third argument is a system type. System types are defined in /usr/include/sys/kernel.h. The final argument contains the load order of the KLD, SI_ORDER_ANY.

The DECLARE_MODULE has individual invocations based on the type of KLD being developed—system call or device driver. The next sections will look at each of the different invocations of the DECLARE_MODULE macro.

System Calls

System calls are a specific type of KLD module. This section describes additional components necessary for a KLD system call.

The System Call Function

A system call KLD contains the system call function, which contains two parameters, the proc structure and the system call arguments. A sample prototype is listed below.

```
typedef int sy_call_t __P((struct proc *, void *));
```

The first argument to a system call is always the proc structure. The second argument is a user-defined structure that represents the system call parameters.

An example system call, kld_syscall, is listed below.

```
static int
kld_syscall (struct proc *p, void *arg)
{
  uprintf("KLD System Call\n");
  return(0);
}
```

The first argument to kld_syscall is the proc structure, which represents the current state of the calling process and. is defined in /usr/include/sys/proc.h. The second argument contains the parameters to the system call. For the kld_syscall example there are none.

The sysent Structure

Each system call contains a sysent entry in the kernel global sysent table. The sysent struct takes two elements. The first element is the number of parameters passed to the system call, and the second element is a function pointer to the system call.

```
struct sysent {     /* system call table */
    int sy_narg;    /* number of arguments */
    sy_call_t *sy_call; /* implementing function */
};


static struct sysent kld_syscall_sysent =
{
  0,                /* number of arguments
*/
  kld_syscall       /* system call function    pointer
*/
};
```

The KLD system call creates a sysent structure to be insterted in the kernel global sysent table.

The offset Variable

The FreeBSD kernel contains a table of all system calls. Each KLD system call defines a static global variable offset that represents the system call number. The system call number value represents the index into the kernel global sysent table for that system call. A KLD system call assigns the offset value to NOS_SYCALL. When the system call is loaded, the kernel linker finds the first available slot in the sysent table and assigns the system call to that slot.

```
#define NO_SYSCALL (-1)

static int offset = NO_SYSCALL;
```

The SYSCALL_MODULE Macro

The SYSCALL_MODULE macro is the KLD system call version of the previously defined DECLARE_MODULE macro. The SYSCALL_MODULE macro is defined in /usr/include/sus/sysent.h and contains five arguments.

```
#define SYSCALL_MODULE(name, offset, new_sysent, evh, arg)      \
static struct syscall_module_data name##_syscall_mod = {        \
        evh, arg, offset, new_sysent
\
};
\
```

The first argument is the generic name for this system call. The second parameter is the offset representing the system call number. The third parameter contains the sysent entry to be added to the kernel global sysent structure. The fourth parameter is the system call load handler function. The final parameter is a user-defined parameter to be passed to the system call loaded handler in parameter four.

You may have noticed that the system call example we've developed doesn't contain a moduledata_t structure. Further analysis of the SYSCALL_MODULE macro shows that the moduledata_t structure is generated by the SYSCALL_MODULE declaration.

For our sample KLD system call the declaration takes the following form:.

```
SYSCALL_MODULE(kld_syscall, &offset, &kld_syscall_sysent,
load_handler, NULL);
```

Device Drivers

The dynamic kernel linker provides support for the dynamic load and unload device drivers. This section covers the required components for a KLD device driver.

The cdevsw Structure

Every device driver contains a device switch entry. The device switch table is defined in /usr/include/conf.h and it contains the device handler functions, device major number and flags.

```
struct cdevsw {
    d_open_t    *d_open;
    d_close_t   *d_close;
    d_read_t    *d_read;
    d_write_t   *d_write;
    d_ioctl_t   *d_ioctl;
    d_poll_t    *d_poll;
    d_mmap_t    *d_mmap;
    d_strategy_t    *d_strategy;
    const char  *d_name;    /* base device name, e.g. 'vn' */
    int     d_maj;
```

```
    d_dump_t    *d_dump;
    d_psize_t   *d_psize;
    u_int       d_flags;
    int     d_bmaj;
    /* additions below are not binary compatible with 4.2
    /*and below */
    d_kqfilter_t    *d_kqfilter;
};
```

Before declaring a cdevsw entry for the KLD device driver, a little background work is required. Let's take at look at these steps before the cdevsw entry is declared.

The first component of a KLD device driver is the device switch table entry. A KLD driver contains driver handler functions for open, close, read and write. Each function is forward declared so it can be listed in the device switch table.

```
d_open_t    kld_open;
d_close_t   kld_close;
d_read_t    kld_read;
d_write_t   kld_write;
```

Before creating the device switch table, a device major number is required. The file /sys/conf/majors contains the defined major device numbers. Numbers 32 through 38 are predefined for KLD device drivers; 35 is a reasonable value for development.

```
#define KLD_DRIVER_MAJOR    35
```

Now with the required components for the device switch table defined, we can declare the KLD driver cdevsw switch entry. An example cdevsw entry is listed below.

```
static struct cdevsw kld_cdevsw =
{
  kld_open,             /* open function             */
  kld_close,            /* close function            */
  kld_read,             /* read function             */
  kld_write,            /* write function            */
  noioctl,              /* ioctl function            */
```

```
  nopoll,               /* poll function              */
  nommap,               /* mmap function              */
  nostrategy,           /* strategy function          */
  "kld_driver",         /* driver name                */
  KLD_DRIVER_MAJOR,     /* major number               */
  nodump,               /* dump function              */
  nopsize,              /* psize function             */
  D_TTY,                /* driver flags               */
  -1                    /* block major number         */
};
```

The dev_t Variable

In addition to the cdevsw entry, each KLD driver must also declare a dev_t variable, the device type. The dev_t variable holds the dev_t for the driver load entry. The dev_t value is required to destroy the driver on module unload. Each dev_t variable is statically defined in the driver source file.

```
static dev_t kld_dev;
```

The Driver load_handler Function

The KLD driver load handler requires additional steps compared to a generic load handler. During KLD driver load a device must be created. The device is a handle used by the call to make_dev during driver load. Consequently, when the module is unloaded the driver must be destroyed.

```
static int load_handler(struct module *m, int cmd, void* arg)
{
    int stat = 0;

    switch(cmd)
    {
    case MOD_LOAD:
      /*
      ** make_dev creates a dev_t structure for a new device
      */
      kld_dev = make_dev(&kld_cdevsw,/* device switch table
*/
                         0,    /* minor number for this device
*/
            UID_ROOT,   /* user device owner              */
```

```
            GID_WHEEL,  /* group device owner              */
            0600,   /* device permissions            */
            "klddriver");   /* device name                 */
      break;

    case MOD_UNLOAD:
      /*
      **     destroy device registration on module unload
      */
      destroy_dev(kld_dev);
      break;

    case MOD_SHUTDOWN:
      break;

    default:
      stat = EINVAL;
    }

    return(stat);
}
```

The DEV_MODULE Macro

A KLD device driver declares a DEV_MODULE macro which is defined in the /usr/include/sys/conf.h and contains three arguments.

The first argument is the device name. The second argument is the load handler, and the last argument is a user-defined parameter passed to the load handler.

```
#define DEV_MODULE(name, evh, arg)                          \
static moduledata_t name##_mod = {                          \
     #name,                                          \
     evh,                                            \
     arg                                        \
};                                              \
DECLARE_MODULE(name, name##_mod, SI_SUB_DRIVERS, SI_ORDER_MIDDLE)
```

As with the system call KLD the DEV_MODULE macro declares the module-data_t structure used by the KLD.

Our example declaration for the KLD sample device driver is as follows:

```
DEV_MODULE(kld_driver_sample, load_handler, NULL);
```

The Driver Functions

Now that the required KLD components are defined, the remainder of driver development consists of implementing the driver handler functions. This generic KLD device driver just contains print statements in each function. The functions are listed below.

The open Function

The open function is called when an application calls the open system call for the driver device note.

```
int kld_open(dev_t dev, int oflags, int devtype, struct proc *p)
{
  uprintf("kld driver: open\n");
  return (0);
}
```

The close Function

The close function is called when the last open handle to the device driver is closed.

```
int kld_close(dev_t dev, int fflag, int devtype, struct proc *p)
{
  uprintf("kld driver: close\n");
  return (0);
}
```

The read Function

The read driver function is called when an application calls the read system call with an open file handle to the device driver.

```
int kld_read(dev_t dev, struct uio *uio, int ioflag)
{
  uprintf("kld driver: read\n");
  return (0);
}
```

The write Function

The write driver function is called when an application calls the read system call with an open file handle to the device driver.

```
int kld_write(dev_t dev, struct uio *uio, int ioflag)
{
  uprintf("kld driver: write\n");
  return (0);
}
```

The Device File

The final step for the driver function is to create the device file. Device files are created using the `mknod` command.

```
# mknod kldd c 35 0
```

The device node is the user space name used to access a device driver.

Commands

KLDs use three system commands to load, unload and display the status of KLDs. A brief description of each commands follows.

The `kldstat` Command

The `kldstat` command lists the current load modules.

```
# kldstat
Id Refs Address    Size     Name
 1    2 0xc0100000 19fe48   kernel
 2    1 0xc146c000 19000    usb.ko
```

The `kldload` Command

The `kldload` command is used to load a KLD into the kernel. `Kldload` takes one command, the name of the KLD file, typically with the extension .ko

```
# kldload -v ./kld_driver.ko
Loaded ./kld_driver.ko, id=5
# kldstat
Id Refs Address    Size     Name
 1    3 0xc0100000 19fe48   kernel
 2    1 0xc146c000 19000    usb.ko
 5    1 0xc148e000 2000     kld_driver.ko
```

The `kldunload` Command

The `kldunload` command is used to unload the module from the kernel. `Kldunload` can unload using a module id or module name.

```
# kldunload -n kld_driver.ko
# kldstat
Id Refs Address    Size     Name
 1    2 0xc0100000 19fe48   kernel
 2    1 0xc146c000 19000    usb.ko
```

INDEX

Numbers and Symbols

_exit system call, 12

A

accept system call, 112
adapter, CompactFlash, 198
appliance server, 3
appliances, Internet, 2–3, 123
application interface library, 77–101
ARP, 105
ATAPI disk driver, 80, 171
autoconfiguration code, 50

B

big endian, 108
bind system call, 110
BIOS, PC, 189–190
boot device, 7
boot loader, 185, 190–191
boot slice, 188–190
boot1, 191
boot2, 191
Bourne Shell, 129, 192
BSD license, 6
bss, 10
byte order, 108

C

C37FF–2 cable, 7
cdevsw structure, 53
chdir system call, 17
child process, 10
CIO–MINI 37 terminal, 7
CLASSPATH environment variable, 144
close system call, 81, 111
command handlers, 39
command_t structure, 130
CompactFlash, 7
    adapters, 7–8
    boot device, 197–205
compatibility issues, 3
config register, 71, 97
config_handler function, 134
connect system call, 112
connectionless data transfer, 114–115
controlling terminal, 19
copyin and copyout functions, 36
copymem system call, 32–36, 38
    command table, 39
cpu keyword, 168
current working directory, 17
CVSUP, 60
cylinders, 184

D

daemon, 21–24, 103–122, 115–117
dependencies, 181
destroy_dev function, 58
dev_t structure, 58
devclass, 57
device driver, 49–76
    environment, 49–51
    structure, 51
    accessing, 79
    system calls, 80
device file, 74–75, 80
device switch table, 50
device_method_t structure, 52
device_t structure, 55–57
digital input-output server appliance, see
    DIO server appliance
DIO daemon, 115

DIO JSP page, 162–165
DIO kernel, 177–180
DIO server appliance, 4–5
    hardware requirements, 6–7
dio.sh script, 194
dio_alloc_resources
    function, 64–66
dio_attach function, 63–64
dio_deallocate_resources
    function, 67
dio_detach function, 66
dio_get and dio_set functions, 87
dio_pci_attach function, 62–63
dio_pci_detach function, 66
dio_pci_probe function, 61
dio_set_line and
    dio_get_line, 88, 90
dio_set_polarity and
    dio_get_polarity, 95
dioclose function, 69
DIOIfJNI class, 150
diointr function, 73
dioioctl function, 69–71
dioopen function, 68–69
DIOShell, 130–142
    command table, 130–132
disk geometry, 183–184
driver_t structure, 57
dumpmem function, 40

E

embedded system
    definition, 1
    next generation, 2
enum data type, 88–90, 92, 95
Ethernet controllers, 174
Ethernet switches, 2
execve system call, 11

F

file
    descriptor, 17, 22
    permissions, 17
file system, formatting, 200
fork system call, 10–11, 21
FreeBSD, definition and benefits, 5

G

GENERIC kernel, 167
get_direction_handler
    function, 139
getgid system call, 14–16
getpgid system call, 16
getpid system call, 13
getpolarity_handler function, 138
getppid system call, 13
getpriority system call, 19
getrlimit system call, 17–18
getuid system call, 14–16
GID, 4
GNU development suite, 4, 6
group identifier, see GID

H

handle_sigcld function, 22
hardware inventory, 177
heads, hard disk, 184
help handler, 44
help_handler function, 140

I

Internet appliances, 2–3
init_daemon function, 23
implementing system calls, 28–29

interrupt, software, 29
INT, 29
Interrupt Descriptor Table, see IDT
IDT, 29
`ioctl`, 69, 71, 81
interrupt controller, 91–96
Internet Protocol, 105
    addressing, 105
IP, see Internet Protocol
ICMP, 106
`init_dio` function, 119
`int_handler` function, 135
`ident` keyword, 168
init daemon, 192

J

Java Development Kit, see JDK
Java Native Interface, see JNI layer
Java Server Page, see JSP
Java, 6
javah tool, 152
Java-to-C interface class, 150
JDK, 143–145
JDK_HOME environment variable, 148
JNI layer, 143–156
JSP, 157, 160–165
    syntax, 161–162

K

`kill` system call, 21
kernel, 27
    building, 167–182
Kernel Mode, 27
kernel-loadable module, see KLD
KLD, 32, 51, 215–227
    Makefile, 216
    device drivers, 221–223
`kldstat` command, 226
`kldload` command, 227
`kldunload` command, 227

L

LD_LIBRARY_PATH environment variable, 145
library functions, 27–28
license, FreeBSD, 207
licensing, 4
`listen` system call, 111
little endian, 108
load handler, 33, 216

M

`main` function, 44–45, 120–122
`make_dev` function, 58–59
`mknod` command, 74
MAKEDEV utility, 74
`make clean` command, 74
`make depend` command, 75
`make` command, 76
`machine` keyword, 168
`maxusers` keyword, 168
`make options` keyword, 169
master boot record, see MBR
MBR, 185, 190
magic number, 187
`moduledata_t` structure, 218

N

NAS, see Network attached storage
native functions, 153–156
network attached storage (NAS), 3
Network Engines Roadster, 7

O

open system call, 80
options keyword, 169

P

parent process, 10
partition table, disk, 185
passwd utility, 14
PCI bus, 209–214
PCI configuration registers, 210
pciconf utility, 62
PCI-DIO24 Digital IO Controller, 7, 59–60
    hardware registers, 82–86
    application interface library, 87–88
pciint_handler function, 136
PGID, 16
PID, 12, 13
platters, 184
process
    creation, 10
    definition, 10
    group, 16
    identifier, see PID
    security, 14
    state, 20
    termination, 11–13
pseudo devices, 176
pwd command, 31

Q

quit handler, 40, 44
quit_handler function, 139–140

R

RAID controllers, 172
RARP, 106
read command, 42, 82
readline_handler function, 132
recv system call, 113
recvfrom system call, 114
register shadowing, 71
remote administration account, 129–130
resources, 17–18
root_bus, 50

S

Sandisk CompactFlash disk, 7
scheduling, 19–20
    priority, 19
scriptlets, 162
SCSI controllers, 171
sectors, hard drive, 184-186
    addressing, 184
Secure Shell, see SSH
send system call, 113
sendto system call, 115
server, DIO, 117–122
session, 18, 22
setpgid system call, 16
setpolarity_handler function, 125
setpriority system call, 19
setrlimit system call, 17–18
setsid system call, 18
shared libraries, 77–79
shell device driver, 60
sigaction system call, 27
signal function, 20
signals, 20
size command, 10
socket system calls, 107, 109
sockets, 109–111
softc structure, 57
software interrupts, 29
solid-state devices, 197
SSH, 123–129

`sshd`, 124
SYSCALL_MODULE macro, 37, 220
`sysent` structure, 36, 220
system call number, 30
system calls, 27–47, 219
    number, 36
system startup, 192

T

TAPR CompactFlash Adapter II, 7
TCP/IP, 103–107
    application layer, 107
    link layer, 104
    network layer, 105
    transport layer, 106
Tomcat server, 157–165
    directory structure, 159–160
`truss` command, 31

U

User Mode, 10, 27
user identifier, see UID
UID, 14
umask system call, 17
UDP, 106
Unix partitions, 189

V

Virtual Private Network (VPN), 2
VPN, see Virtual Private Network

W

`wait` system call, 12
`write` command, 42, 82
`writeline_handler` function, 133

LIMITED WARRANTY AND DISCLAIMER OF LIABILITY

[[NEWNES.]] AND ANYONE ELSE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE ACCOMPANYING CODE ("THE PRODUCT") CANNOT AND DO NOT WARRANT THE PERFORMANCE OR RESULTS THAT MAY BE OBTAINED BY USING THE PRODUCT. THE PRODUCT IS SOLD "AS IS" WITHOUT WARRANTY OF ANY KIND (EXCEPT AS HEREAFTER DESCRIBED), EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY WARRANTY OF PERFORMANCE OR ANY IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. **[[NEWNES.]]** WARRANTS ONLY THAT THE MAGNETIC CD-ROM(S) ON WHICH THE CODE IS RECORDED IS FREE FROM DEFECTS IN MATERIAL AND FAULTY WORKMANSHIP UNDER THE NORMAL USE AND SERVICE FOR A PERIOD OF NINETY (90) DAYS FROM THE DATE THE PRODUCT IS DELIVERED. THE PURCHASER'S SOLE AND EXCLUSIVE REMEDY IN THE EVENT OF A DEFECT IS EXPRESSLY LIMITED TO EITHER REPLACEMENT OF THE CD-ROM(S) OR REFUND OF THE PURCHASE PRICE, AT **[[NEWNES.]]**'S SOLE DISCRETION.

IN NO EVENT, WHETHER AS A RESULT OF BREACH OF CONTRACT, WARRANTY OR TORT (INCLUDING NEGLIGENCE), WILL **[[NEWNES.]]** OR ANYONE WHO HAS BEEN INVOLVED IN THE CREATION OR PRODUCTION OF THE PRODUCT BE LIABLE TO PURCHASER FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT OR ANY MODIFICATIONS THEREOF, OR DUE TO THE CONTENTS OF THE CODE, EVEN IF **[[NEWNES.]]** HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

ANY REQUEST FOR REPLACEMENT OF A DEFECTIVE CD-ROM MUST BE POSTAGE PREPAID AND MUST BE ACCOMPANIED BY THE ORIGINAL DEFECTIVE CD-ROM, YOUR MAILING ADDRESS AND TELEPHONE NUMBER, AND PROOF OF DATE OF PURCHASE AND PURCHASE PRICE. SEND SUCH REQUESTS, STATING THE NATURE OF THE PROBLEM, TO ELSEVIER SCIENCE CUSTOMER SERVICE, 6277 SEA HARBOR DRIVE, ORLANDO, FL 32887, 1-800-321-5068. **[[NEWNES.]]** SHALL HAVE NO OBLIGATION TO REFUND THE PURCHASE PRICE OR TO REPLACE A CD-ROM BASED ON CLAIMS OF DEFECTS IN THE NATURE OR OPERATION OF THE PRODUCT.

SOME STATES DO NOT ALLOW LIMITATION ON HOW LONG AN IMPLIED WARRANTY LASTS, NOR EXCLUSIONS OR LIMITATIONS OF INCIDENTAL OR CONSEQUENTIAL DAMAGE, SO THE ABOVE LIMITATIONS AND EXCLUSIONS MAY NOT **[[NEWNES.]]** APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS THAT VARY FROM JURISDICTION TO JURISDICTION.

THE RE-EXPORT OF UNITED STATES ORIGIN SOFTWARE IS SUBJECT TO THE UNITED STATES LAWS UNDER THE EXPORT ADMINISTRATION ACT OF 1969 AS AMENDED. ANY FURTHER SALE OF THE PRODUCT SHALL BE IN COMPLIANCE WITH THE UNITED STATES DEPARTMENT OF COMMERCE ADMINISTRATION REGULATIONS. COMPLIANCE WITH SUCH REGULATIONS IS YOUR RESPONSIBILITY AND NOT THE RESPONSIBILITY OF **[[NEWNES.]]**.