PRENTICE HALL

PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Embedded Linux Primer

Second Edition

A Practical Real-World Approach

Christopher Hallinan

Embedded Linux Primer, Second Edition

The Prentice Hall Open Source Software Development Series

Arnold Robbins, Series Editor

Visit **informit.com/opensourcedev** for a complete list of available publications.

Open Source technology has revolutionized the computing world. From MySQL to the Python programming language, these technologies are in use on many different systems, ranging from proprietary systems, to Linux systems, to traditional UNIX systems, to mainframes. **The Prentice Hall Open Source Software Development Series** is designed to bring you the best of these Open Source technologies. Not only will you learn how to use them for your projects, but you will learn from them. By seeing real code from real applications, you will learn the best practices of Open Source developers the world over.

Embedded Linux Primer, Second Edition

A Practical, Real-World Approach

Christopher Hallinan

Prentice Hall Professional Technical Reference
Upper Saddle River, NJ · Boston · Indianapolis · San Francisco
New York · Toronto · Montreal · London · Munich · Paris · Madrid
Capetown · Sydney · Tokyo · Singapore · Mexico City

Library of Congress Cataloging-in-Publication Data:

Hallinan, Christopher.
Embedded Linux primer : a practical real-world approach / Christopher Hallinan.
p. cm.
ISBN 978-0-13-701783-6 (hardback : alk. paper) 1. Linux. 2. Operating systems (Computers) 3. Embedded computer systems--Programming. I. Title.

QA76.76.O63H34462 2011
005.4'32--dc22

2010032891

ISBN-13: 978-0-137-01783-6
ISBN-10: 0-137-01783-9

Text printed in the United States on recycled paper at Courier in Westford, Massachusetts.
First printing September 2010

Editor-in-Chief: Mark L. Taub
Executive Acquisitions Editor: Debra Williams Cauley
Development Editor: Michael Thurston
Managing Editor: Kristy Hart
Project Editors: Alexandra Maurer and Jovana San Nicolas-Shirley
Copy Editor: Gayle Johnson
Indexer: Heather McNeill
Proofreader: Sarah Kearns
Technical Reviewers: Robert P.J. Day, Kurt Lloyd, Jon Masters, Sandra Terrace, and Mark A. Yoder
Publishing Coordinator: Kim Boedigheimer
Cover Designer: Alan Clements
Compositor: Tricia Bronkella

To my grandmother Edythe Diorio Ricciuti, who, at one hundred and five and counting, continues to provide inspiration to her loving family through her deep faith, unshakable moral compass, and selfless dedication to others.

This page intentionally left blank

Contents

Foreword for the First Edition

Computers are everywhere.

This fact, of course, is no surprise to anyone who hasn't been living in a cave during the past 25 years or so. And you probably know that computers aren't just on our desktops, in our kitchens, and, increasingly, in our living rooms, holding our music collections. They're also in our microwave ovens, our regular ovens, our cell phones, and our portable digital music players.

And if you're holding this book, you probably know a lot, or are interested in learning more about, these embedded computer systems.

Until not too long ago, embedded systems were not very powerful, and they ran special-purpose, proprietary operating systems that were very different from industry-standard ones. (Plus, they were much harder to develop for.) Today, embedded computers are as powerful as, if not more powerful than, a modern home computer. (Consider the high-end gaming consoles, for example.)

Along with this power comes the capability to run a full-fledged operating system such as Linux. Using a system such as Linux for an embedded product makes a lot of sense. A large community of developers are making this possible. The development environment and the deployment environment can be surprisingly similar, which makes your life as a developer much easier. And you have both the security of a protected address space that a virtual memory-based system gives you and the power and flexibility of a multiuser, multiprocess system. That's a good deal all around.

For this reason, companies all over the world are using Linux on many devices such as PDAs, home entertainment systems, and even, believe it or not, cell phones!

I'm excited about this book. It provides an excellent "guide up the learning curve" for the developer who wants to use Linux for his or her embedded system. It's clear, well-written, and well-organized; Chris's knowledge and understanding show through at every turn. It's not only informative and helpful; it's also enjoyable to read.

I hope you learn something and have fun at the same time. I know I did.

Arnold Robbins
Series Editor

Foreword for the Second Edition

Smart phones. PDAs. Home routers. Smart televisions. Smart Blu-ray players. Smart yo-yos. OK, maybe not. More and more of the everyday items in our homes and offices, used for work and play, have computers embedded in them. And those computers are running GNU/Linux.

You may be a GNU/Linux developer used to working on desktop (or notebook) Intel Architecture systems. Or you may be an embedded systems developer used to more traditional embedded and/or real-time operating systems. Whatever your background, if you're entering the world of embedded Linux development, Dorothy's "Toto, I've a feeling we're not in Kansas anymore" applies to you. Welcome to the adventure!

Dorothy had a goal, and some good friends, but no *guide*. You, however, are better off, since you're holding an amazing field guide to the world of embedded Linux development. Christopher Hallinan lays it all out for you—the how, the where, the why, and also the "what not to do." This book will keep you out of the school of hard knocks and get you going easily and quickly on the road to building your product.

It is no surprise that this book has been a leader in its market. This new edition is even better. It is up to date and brings all the author's additional experience to bear on the subject.

I am very proud to have this book in my series. But what's more important is that you will be proud of yourself for having built a better product because you read it! Enjoy!

Arnold Robbins
Series Editor

Preface

Although many good books cover Linux, this one brings together many dimensions of information and advice specifically targeted to the embedded Linux developer. Indeed, some very good books have been written about the Linux kernel, Linux system administration, and so on. This book refers to many of the ones I consider to be at the top of their categories.

Much of the material presented in this book is motivated by questions I've received over the years from development engineers in my capacity as an embedded Linux consultant and from my direct involvement in the commercial embedded Linux market.

Embedded Linux presents the experienced software engineer with several unique challenges. First, those with many years of experience with legacy real-time operating systems (RTOSs) find it difficult to transition their thinking from those environments to Linux. Second, experienced application developers often have difficulty understanding the relative complexities of a cross-development environment.

Although this is a primer, intended for developers new to embedded Linux, I am confident that even developers who are experienced in embedded Linux will benefit from the useful tips and techniques I have learned over the years.

Practical Advice for the Practicing Embedded Developer

This book describes my view of what an embedded engineer needs to know to get up to speed fast in an embedded Linux environment. Instead of focusing on Linux kernel internals, the kernel chapters in this book focus on the project nature of the kernel and leave the internals to the other excellent texts on the subject. You will learn the organization and layout of the kernel source tree. You will discover the

binary components that make up a kernel image, how they are loaded, and what purpose they serve on an embedded system.

In this book, you will learn how the Linux kernel build system works and how to incorporate your own custom changes that are required for your projects. You will learn the details of Linux system initialization, from the kernel to user space initialization. You will learn many useful tips and tricks for your embedded project, from bootloaders, system initialization, file systems, and Flash memory to advanced kernel- and application-debugging techniques. This second edition features much new and updated content, as well as new chapters on open source build systems, USB and udev, highlighting how to configure and use these complex systems on your embedded Linux project.

Intended Audience

This book is intended for programmers who have working knowledge of programming in C. I assume that you have a rudimentary understanding of local area networks and the Internet. You should understand and recognize an IP address and how it is used on a simple local area network. I also assume that you understand hexadecimal and octal numbering systems and their common usage in a book such as this.

Several advanced concepts related to C compiling and linking are explored, so you will benefit from having at least a cursory understanding of the role of the linker in ordinary C programming. Knowledge of the GNU make operation and semantics also will prove beneficial.

What This Book Is Not

This book is not a detailed hardware tutorial. One of the difficulties the embedded developer faces is the huge variety of hardware devices in use today. The user manual for a modern 32-bit processor with some integrated peripherals can easily exceed 3,000 pages. There are no shortcuts. If you need to understand a hardware device from a programmer's point of view, you need to spend plenty of hours in your favorite reading chair with hardware data sheets and reference guides, and many more hours writing and testing code for these hardware devices!

This is also not a book about the Linux kernel or kernel internals. In this book, you won't learn about the intricacies of the Memory Management Unit (MMU)

used to implement Linux's virtual memory-management policies and procedures; there are already several good books on this subject. You are encouraged to take advantage of the "Suggestions for Additional Reading" sections found at the end of every chapter.

Conventions Used

Filenames, directories, utilities, tools, commands, and code statements are presented in a `monospace` font. Commands that the user enters appear in bold monospace. New terms or important concepts are presented in italics.

When you see a pathname preceded by three dots, this refers to a well-known but unspecified top-level directory. The top-level directory is context-dependent but almost universally refers to a top-level Linux source directory. For example, `.../arch/powerpc/kernel/setup_32.c` refers to the `setup_32.c` file located in the architecture branch of a Linux source tree. The actual path might be something like `~/sandbox/linux.2.6.33/arch/power/kernel/setup_32.c`.

How This Book Is Organized

Chapter 1, "Introduction," provides a brief look at the factors driving the rapid adoption of Linux in the embedded environment. Several important standards and organizations relevant to embedded Linux are introduced.

Chapter 2, "The Big Picture," introduces many concepts related to embedded Linux upon which later chapters are built.

Chapter 3, "Processor Basics," presents a high-level look at the more popular processors and platforms that are being used to build embedded Linux systems. We examine selected products from many of the major processor manufacturers. All the major architecture families are represented.

Chapter 4, "The Linux Kernel: A Different Perspective," examines the Linux kernel from a slightly different perspective. Instead of kernel theory or internals, we look at its structure, layout, and build construction so that you can begin learning your way around this large software project and, more important, learn where your own customization efforts must be focused. This includes detailed coverage of the kernel build system.

Chapter 5, "Kernel Initialization," details the Linux kernel's initialization process. You will learn how the architecture- and bootloader-specific image components are concatenated to the image of the kernel proper for downloading to Flash and booting by an embedded bootloader. The knowledge you gain here will help you customize the Linux kernel to your own embedded application requirements.

Chapter 6, "User Space Initialization," continues the detailed examination of the initialization process. When the Linux kernel has completed its own initialization, application programs continue the initialization process in a predetermined manner. Upon completing Chapter 6, you will have the necessary knowledge to customize your own userland application startup sequence.

Chapter 7, "Bootloaders," is dedicated to the bootloader and its role in an embedded Linux system. We examine the popular open-source bootloader U-Boot and present a porting example. We briefly introduce additional bootloaders in use today so that you can make an informed choice about your particular requirements.

Chapter 8, "Device Driver Basics," introduces the Linux device driver model and provides enough background to launch into one of the great texts on device drivers, listed in "Suggestions for Additional Reading" at the end of the chapter.

Chapter 9, "File Systems," describes the more popular file systems being used in embedded systems today. We include coverage of the JFFS2, an important embedded file system used on Flash memory devices. This chapter includes a brief introduction to building your own file system image, one of the more difficult tasks the embedded Linux developer faces.

Chapter 10, "MTD Subsystem," explores the Memory Technology Devices (MTD) subsystem. MTD is an extremely useful abstraction layer between the Linux file system and hardware memory devices, primarily Flash memory.

Chapter 11, "BusyBox," introduces BusyBox, one of the most useful utilities for building small embedded systems. We describe how to configure and build BusyBox for your particular requirements, along with detailed coverage of system initialization unique to a BusyBox environment. Appendix C, "BusyBox Commands," lists the available BusyBox commands from a recent BusyBox release.

Chapter 12, "Embedded Development Environment," takes a detailed look at the unique requirements of a typical cross-development environment. Several techniques are presented to enhance your productivity as an embedded developer, including the powerful NFS root mount development configuration.

Chapter 13, "Development Tools," examines many useful development tools. Debugging with gdb is introduced, including coverage of core dump analysis. Many more tools are presented and explained, with examples including `strace`, `ltrace`, `top`, and `ps`, and the memory profilers `mtrace` and `dmalloc`. The chapter concludes with an introduction to the more important binary utilities, including the powerful `readelf` utility.

Chapter 14, "Kernel Debugging Techniques," provides a detailed examination of many debugging techniques useful for debugging inside the Linux kernel. We introduce the use of the kernel debugger KGDB and present many useful debugging techniques using the combination of gdb and KGDB as debugging tools. Included is an introduction to using hardware JTAG debuggers and some tips for analyzing failures when the kernel won't boot.

Chapter 15, "Debugging Embedded Linux Applications," moves the debugging context from the kernel to your application programs. We continue to build on the gdb examples from the previous two chapters, and we present techniques for multithreaded and multiprocess debugging.

Chapter 16, "Open Source Build Systems," replaces the kernel porting chapter from the first edition. That chapter had become hopelessly outdated, and proper treatment of that topic in modern kernels would take a book of its own. I think you will be pleased with the new Chapter 16, which covers the popular build systems available for building complete embedded Linux distributions. Among other systems, we introduce OpenEmbedded, a build system that has gained significant traction in commercial and other open source projects.

Chapter 17, "Linux and Real Time," introduces one of the more interesting challenges in embedded Linux: configuring for real time via the `PREEMPT_RT` option. We cover the features available with RT and how they can be used in a design. We also present techniques for measuring latency in your application configuration.

Chapter 18, "Universal Serial Bus," describes the USB subsystem in easy-to-understand language. We introduce concepts and USB topology and then present several examples of USB configuration. We take a detailed look at the role of sysfs and USB to help you understand this powerful facility. We also present several tools that are useful for understanding and troubleshooting USB.

Chapter 19, "udev," takes the mystery out of this powerful system configuration utility. We examine udev's default behavior as a foundation for understanding how

to customize it. Several real-world examples are presented. For BusyBox users, we examine BusyBox's mdev utility.

The appendixes cover the GNU Public License, U-Boot configurable commands, BusyBox commands, SDRAM interface considerations, resources for the open source developer, and a sample configuration file for one of the more popular hardware JTAG debuggers, the BDI-2000.

Follow Along

You will benefit most from this book if you can divide your time between this book and your favorite Linux workstation. Grab an old x86 computer to experiment on an embedded system. Even better, if you have access to a single-board computer based on another architecture, use that. The BeagleBoard makes an excellent low-cost platform for experimentation. Several examples in this book are based on that platform. You will benefit from learning the layout and organization of a very large code base (the Linux kernel), and you will gain significant knowledge and experience as you poke around the kernel and learn by doing.

Look at the code and try to understand the examples produced in this book. Experiment with different settings, configuration options, and hardware devices. You can gain much in terms of knowledge, and besides, it's loads of fun. If you are so inclined, please log on and contribute to the website dedicated to this book, www.embeddedlinuxprimer.com. Feel free to create an account, add content and comments to other contributions, and share your own successes and solutions as you gain experience in this growing segment of the Linux community. Your input will help others as they learn. It is a work in progress, and your contributions will help it become a valuable community resource.

GPL Copyright Notice

Portions of open-source code reproduced in this book are copyrighted by a large number of individual and corporate contributors. The code reproduced here has been licensed under the terms of the GNU Public License (GPL).

Appendix A contains the text of the GNU Public License.

Acknowledgments for the First Edition

I am constantly amazed by the graciousness of open source developers. I am humbled by the talent in our community that often far exceeds my own. During the course of this project, I reached out to many people in the Linux and open source community with questions. Most often my questions were answered quickly and with encouragement. In no particular order, I'd like to express my gratitude to the following members of the Linux and open source community who contributed answers to my questions:

Dan Malek provided inspiration for some of the contents of Chapter 2.

Dan Kegel and Daniel Jacobowitz patiently answered my toolchain questions.

Scott Anderson provided the original ideas for the gdb macros presented in Chapter 14.

Brad Dixon continues to challenge and expand my technical vision through his own.

George Davis answered my ARM questions.

Jim Lewis provided comments and suggestions on the MTD coverage.

Cal Erickson answered my gdb use questions.

John Twomey advised me on Chapter 3.

Lee Revell, Sven-Thorsten Dietrich, and Daniel Walker advised me on real-time Linux content. Klaas van Gend provided excellent feedback and ideas for my development tools and debugging content.

Many thanks to AMCC, Embedded Planet, Ultimate Solutions, and United Electronic Industries for providing hardware for the examples. Many thanks to my employer, Monta Vista Software, for tolerating the occasional distraction and for providing software for some of the examples. Many others contributed ideas, encouragement, and support over the course of the project. To them I am also grateful.

I offer my sincere appreciation to my primary review team, who promptly read each chapter and provided excellent feedback, comments, and ideas. Thanks to Arnold Robbins, Sandy Terrace, Kurt Lloyd, and Rob Farber. Thanks also to David Brief, who reviewed the proposal and provided valuable input on the book's organization. Many thanks to Arnold for helping this newbie learn the ropes of writing a technical book. Although I have made every attempt to eliminate mistakes, those that remain are solely my own.

I want to thank Mark L. Taub for bringing this project to fruition and for his encouragement and infinite patience. I want to thank the production team, including Kristy Hart, Jennifer Cramer, Krista Hansing, and Cheryl Lenser.

Finally, a very special and heartfelt thank-you to Cary Dillman, who read each chapter as it was written, and for her constant encouragement and occasional sacrifice throughout the project.

Acknowledgments for the Second Edition

First I must acknowledge the guidance, experience, and endless patience of Debra Williams Cauley, Executive Acquisitions Editor, without whom this project would never have happened.

Many thanks to my dedicated primary review team: Robert P.J. Day, Sandy Terrace, Kurt Lloyd, Jon Masters, and series editor Arnold Robbins. I cannot say enough about the value of their individual contributions to the quality of this book.

Thanks also to Professor Mark A. Yoder and his embedded Linux class for giving the manuscript a thorough classroom test.

A special thanks to Freescale Semiconductor for providing hardware that served as the basis for many of the examples in this book. I would not have enjoyed this support without the efforts of Kalpesh Gala, who facilitated these arrangements.

Thanks also to Embedded Planet and Tim Van de Walle, who provided hardware for some of the examples.

Several individuals were especially helpful with advice and answers to questions during the project. In no particular order, my appreciation and thanks are extended to Cedric Hombourger, Klaas van Gend, George Davis, Sven-Thorsten Dietrich, Jason Wessels, and Dave Anders.

I also want to thank the production team who endured my sometimes-hectic schedule. They include Alexandra Maurer, Michael Thurston, Jovana San Nicolas-Shirley, Gayle Johnson, Heather McNeill, Tricia Bronkella, and Sarah Kearns.

With every project of this magnitude, countless people provide input in the form of an answer to a quick question, or perhaps an idea from a conversation. They are too numerous to mention but nonetheless deserve credit for their willing and sometimes unknowing support.

In the first edition, I specifically thanked Cary Dillman for her tireless efforts to review my chapters as they were written. She is now my lovely wife, Cary Hallinan. Cary continued her support by providing much-needed inspiration, patience, and occasional sacrifice throughout the second-edition project.

About the Author

Christopher Hallinan is a technical marketing engineer for the Embedded Systems Division of Mentor Graphics, living and working in Florida. He has spent more than 25 years in the networking and communications industry, mostly in various product development, management, and marketing roles, where he developed a strong background in the space where hardware meets software. Prior to joining Mentor Graphics, he spent nearly seven years as a field applications engineer for Monta Vista Software. Before that, Hallinan spent four years as an independent Linux consultant, providing custom Linux board ports, device drivers, and bootloaders. His introduction to the open source community was through contributions to the popular U-Boot bootloader. When not messing about with Linux, he is often found singing and playing a Taylor or Martin.

Chapter 1

Introduction

In This Chapter

The move away from proprietary embedded operating systems is causing quite a stir in the corporate boardrooms of many traditional embedded operating system (OS) companies. For many well-founded reasons, Linux is being adopted as the operating system in many products beyond its traditional stronghold in server applications. Examples of these embedded systems include cellular phones, DVD players, video games, digital cameras, network switches, and wireless networking gear. It is quite likely that Linux is already in your home or automobile. Linux has been commonly selected as the embedded operating system in devices including set-top boxes, high-definition televisions, Blu-ray DVD players, automobile infotainment centers, and many other devices encountered in everyday life.

1.1 Why Linux?

Because of the numerous economic and technical benefits, we are seeing strong growth in the adoption of Linux for embedded devices. This trend has crossed virtually all markets and technologies. Linux has been adopted for embedded products in the worldwide public switched telephone network, global data networks, and wireless cellular handsets, as well as radio node controllers and backhaul infrastructure that operates these networks. Linux has enjoyed success in automobile applications, consumer products such as games and PDAs, printers, enterprise switches and routers, and many other products. Tens of millions of cell phones are now shipping worldwide with Linux as the operating system of choice. The adoption rate of embedded Linux continues to grow, with no end in sight.

Here are some of the reasons for the growth of embedded Linux:

- Linux supports a vast variety of hardware devices, probably more than any other OS.
- Linux supports a huge variety of applications and networking protocols.
- Linux is scalable, from small consumer-oriented devices to large, heavy-iron, carrier-class switches and routers.

- Linux can be deployed without the royalties required by traditional proprietary embedded operating systems.
- Linux has attracted a huge number of active developers, enabling rapid support of new hardware architectures, platforms, and devices.
- An increasing number of hardware and software vendors, including virtually all the top-tier chip manufacturers and independent software vendors (ISVs), now support Linux.

For these and other reasons, we are seeing an accelerated adoption rate of Linux in many common household items, ranging from high-definition televisions to cellular handsets.

1.2 Embedded Linux Today

It may come as no surprise that Linux has experienced significant growth in the embedded space. Indeed, the fact that you are reading this book indicates that Linux has touched your life. It is difficult to estimate the market size, because many companies continue to build their own embedded Linux distributions.

LinuxDevices.com, the popular news and information portal founded by Rick Lehrbaum, now owned by Ziff Davis, conducts an annual survey of the embedded Linux market. In its latest survey, it reports that Linux has emerged as the dominant operating system used in thousands of new designs each year. In fact, nearly half the respondents reported using Linux in an embedded design. The next most popular operating system reportedly was used by only about one in eight respondents. Commercial operating systems that once dominated the embedded market were reportedly used by fewer than one in ten respondents. Even if you find reason to dispute these results, no one can ignore the momentum in the embedded Linux marketplace today.

1.3 Open Source and the GPL

One of the fundamental factors driving the adoption of Linux is the fact that it is open source. For a fascinating and insightful look at the history and culture of the open source movement, read Eric S. Raymond's book, referenced at the end of this chapter.

The Linux kernel is licensed under the terms of the GNU GPL[1] (General Public License), which leads to the popular myth that Linux is free. In fact, the second

[1] See http://www.gnu.org/licenses/gpl.html for complete text of the license.

paragraph of the GNU GPL Version 3 declares: "When we speak of free software, we are referring to freedom, not price." Most professional development managers agree: You can download Linux without charge, but development and deployment with any OS on an embedded platform carries an (often substantial) cost. Linux is no different in this regard.

The GPL is remarkably short and easy to read. Here are some of its most important characteristics:

- The license is self-perpetuating.
- The license grants the user freedom to run the program.
- The license grants the user the right to study and modify the source code.
- The license grants the user permission to distribute the original code and his modifications.
- The license is viral. In other words, it grants these same rights to anyone to whom you distribute GPL software.

When software is released under the terms of the GPL, it must forever carry that license.[2] Even if the code is highly modified, which is allowed and even encouraged by the license, the GPL mandates that it must be released under the same license. The intent of this feature is to guarantee freedom of access to the software, including modified versions of the software (or derived works, as they are commonly called).

No matter how the software was obtained, the GPL grants the licensee unlimited distribution rights, without the obligation to pay royalties or per-unit fees. This does not mean that vendors can't charge for their GPL software—this is a reasonable and common business practice. It means that once in possession of GPL software, it is permissible to modify and redistribute it, whether or not it is a derived (modified) work. However, as dictated by the GPL, the authors of the modified work are obligated to release the work under the terms of the GPL if they decide to do so. Any distribution of a derived work, such as shipment to a customer, triggers this obligation.

1.3.1 Free Versus Freedom

Two popular phrases are often repeated in the discussion about the free nature of open source: "free as in freedom" and "free as in beer." (The author is particularly fond of the latter.) The GPL exists to guarantee "free as in freedom" of a particular body of

[2] If all the copyright holders agreed, the software could in theory be released under a new license. This would be a very unlikely scenario indeed, especially for a large software base with thousands of contributors.

software. It guarantees your freedom to use it, study it, and change it. It also guarantees these freedoms for anyone to whom you distribute your modified code. This concept has become fairly widely understood.

One of the misconceptions frequently heard is that Linux is "free as in beer." You can obtain Linux free of cost. You can download a Linux kernel in a few minutes. However, as any professional development manager understands, certain costs are associated with any software to be incorporated into a design. These include the costs of acquisition, integration, modification, maintenance, and support. Add to that the cost of obtaining and maintaining a properly configured toolchain, libraries, application programs, and specialized cross-development tools compatible with your chosen architecture, and you can quickly see that it is a nontrivial exercise to develop the needed software components and development environment necessary to develop and deploy your embedded Linux-based system.

1.4 Standards and Relevant Bodies

As Linux continues to gain market share in the desktop, enterprise, and embedded market segments, new standards and organizations have emerged to help influence the use and acceptance of Linux. This section introduces the standards you might want to familiarize yourself with.

1.4.1 Linux Standard Base

Probably the single most relevant standard for a Linux distribution maintainer is the Linux Standard Base (LSB). The goal of the LSB is to establish a set of standards designed to enhance the interoperability of applications among different Linux distributions. Currently, the LSB spans several architectures, including IA32/64, Power Architecture 32- and 64-bit, AMD64, and others. The standard is divided into a core component and the individual architectural components.

The LSB specifies common attributes of a Linux distribution, including object format, standard library interfaces, a minimum set of commands and utilities and their behavior, file system layout, system initialization, and so on.

You can learn more about the LSB at the link given at the end of this chapter.

1.4.2 Linux Foundation

According to its website, the Linux Foundation "is a non-profit consortium dedicated to fostering the growth of Linux." The Linux Foundation sponsors the work of Linus Torvalds, the creator of Linux. The Linux Foundation sponsors several working groups to define standards and participate in the development of features targeting many important Linux platform attributes. The next two sections introduce some of these initiatives.

1.4.3 Carrier-Grade Linux

A significant number of the world's largest networking and telecommunications equipment manufacturers are either developing or shipping carrier-class equipment running Linux as the operating system. Significant features of carrier-class equipment include high reliability, high availability, and rapid serviceability. These vendors design products using redundant hot-swap architectures, fault-tolerant features, clustering, and often real-time performance.

The Linux Foundation Carrier Grade Linux workgroup has produced a specification defining a set of requirements for carrier-class equipment. The current version of the specification covers seven functional areas:

- **Availability**—Requirements that provide enhanced availability, including online maintenance operations, redundancy, and status monitoring
- **Clusters**—Requirements that facilitate redundant services, such as cluster membership management and data checkpointing
- **Serviceability**—Requirements for remote servicing and maintenance, such as SNMP and diagnostic monitoring of fans and power supplies
- **Performance**—Requirements to define performance and scalability, symmetric multiprocessing, latencies, and more
- **Standards**—Requirements that define standards to which CGL-compliant equipment shall conform
- **Hardware**—Requirements related to high-availability hardware, such as blade servers and hardware-management interfaces
- **Security**—Requirements to improve overall system security and protect the system from various external threats

1.4.4 Mobile Linux Initiative: Moblin

Several mobile handsets (cellular phones) available on the worldwide market have been built around embedded Linux. It has been widely reported that tens of millions of handsets have been shipped with Linux as the operating system platform. The only certainty is that more are coming. This promises to be one of the most explosive market segments for what was formerly the role of a proprietary real-time operating system. This speaks volumes about the readiness of Linux for commercial embedded applications.

The Linux Foundation sponsors a workgroup originally called the Mobile Linux Initiative, now referred to as Moblin. Its purpose is to accelerate the adoption of Linux on next-generation mobile handsets and other converged voice/data portable devices, according to the Linux Foundation website. The areas of focus for this working group include development tools, I/O and networking, memory management, multimedia, performance, power management, security, and storage. The Moblin website can be found at http://moblin.org. You can try out a Moblin release, such as Fedora/Moblin, found at http://fedoraproject.org/wiki/Features/FedoraMoblin, or the Ubuntu Moblin remix found on the author's Dell Mini 10 Netbook.

The embedded Linux landscape is continuously evolving. As this second edition was being prepared, the Moblin and Maemo project merged to become MeeGo. You can learn more about MeeGo, and even download a MeeGo image to try out, at http://meego.com/.

1.4.5 Service Availability Forum

If you are engaged in building products for environments in which high reliability, availability, and serviceability (RAS) are important, you should be aware of the Service Availability Forum (SA Forum). This organization is playing a leading role in defining a common set of interfaces for use in carrier-grade and other commercial equipment for system management. The SA Forum website is at www.saforum.org.

1.5 Summary

Embedded Linux has won the race. Indeed, you probably have embedded Linux in your car or home. This chapter examined the reasons why and developed a perspective for the material to come:

- Adoption of Linux among developers and manufacturers of embedded products continues to accelerate.
- Use of Linux in embedded devices continues to grow at an exciting pace.
- Many factors are driving the growth of Linux in the embedded market.
- Several standards and relevant organizations are influencing embedded Linux.

1.5.1 Suggestions for Additional Reading

The Cathedral and the Bazaar
Eric S. Raymond
O'Reilly Media, Inc., 2001

Linux Standard Base Project
http://www.linuxfoundation.org/collaborate/workgroups/lsb

Linux Foundation
http://www.linuxfoundation.org/

Chapter 2

The Big Picture

In This Chapter

Often the best path to understanding a given task is to have a good grasp of the big picture. Many fundamental concepts can present challenges to the newcomer to embedded systems development. This chapter takes you on a tour of a typical embedded system and the development environment with specific emphasis on the concepts and components that make developing these systems unique and often challenging.

2.1 Embedded or Not?

Several key attributes are associated with embedded systems. You wouldn't necessarily call your desktop PC an embedded system. But consider a desktop PC hardware platform in a remote data center that performs a critical monitoring and alarm task. Assume that this data center normally is not staffed. This imposes a different set of requirements on this hardware platform. For example, if power is lost and then restored, you would expect this platform to resume its duties without operator intervention.

Embedded systems come in a variety of shapes and sizes, from the largest multiple-rack data storage or networking powerhouses to tiny modules such as your personal MP3 player or cellular handset. Following are some of the usual characteristics of an embedded system:

- Contains a processing engine, such as a general-purpose microprocessor.
- Typically designed for a specific application or purpose.
- Includes a simple (or no) user interface, such as an automotive engine ignition controller.
- Often is resource-limited. For example, it might have a small memory footprint and no hard drive.
- Might have power limitations, such as a requirement to operate from batteries.
- Not typically used as a general-purpose computing platform.
- Generally has application software built in, not user-selected.

- Ships with all intended application hardware and software preintegrated.
- Often is intended for applications without human intervention.

Most commonly, embedded systems are resource-constrained compared to the typical desktop PC. Embedded systems often have limited memory, small or no hard drives, and sometimes no external network connectivity. Frequently, the only user interface is a serial port and some LEDs. These and other issues can present challenges to the embedded system developer.

2.1.1 BIOS Versus Bootloader

When power is first applied to the desktop computer, a software program called the BIOS immediately takes control of the processor. (Historically, BIOS was an acronym meaning Basic Input/Output Software, but the term has taken on a meaning of its own as the functions it performs have become much more complex than the original implementations.) The BIOS might actually be stored in Flash memory (described shortly) to facilitate field upgrade of the BIOS program itself.

The BIOS is a complex set of system-configuration software routines that have knowledge of the low-level details of the hardware architecture. Most of us are unaware of the extent of the BIOS and its functionality, but it is a critical piece of the desktop computer. The BIOS first gains control of the processor when power is applied. Its primary responsibility is to initialize the hardware, especially the memory subsystem, and load an operating system from the PC's hard drive.

In a typical embedded system (assuming that it is not based on an industry-standard x86 PC hardware platform), a bootloader is the software program that performs the equivalent functions. In your own custom embedded system, part of your development plan must include the development of a bootloader specific to your board. Luckily, several good open source bootloaders are available that you can customize for your project. These are introduced in Chapter 7, "Bootloaders."

Here are some of the more important tasks your bootloader performs on power-up:

- Initializes critical hardware components, such as the SDRAM controller, I/O controllers, and graphics controllers.
- Initializes system memory in preparation for passing control to the operating system.
- Allocates system resources such as memory and interrupt circuits to peripheral controllers, as necessary.

- Provides a mechanism for locating and loading your operating system image.
- Loads and passes control to the operating system, passing any required startup information. This can include total memory size, clock rates, serial port speeds, and other low-level hardware-specific configuration data.

This is a simplified summary of the tasks that a typical embedded-system bootloader performs. The important point to remember is this: If your embedded system will be based on a custom-designed platform, these bootloader functions must be supplied by you, the system designer. If your embedded system is based on a commercial off-the-shelf (COTS) platform such as an ATCA chassis,[1] the bootloader (and often the Linux kernel) typically is included on the board. Chapter 7 discusses bootloaders in more detail.

2.2 Anatomy of an Embedded System

Figure 2-1 is a block diagram of a typical embedded system. This is a simple example of a high-level hardware architecture that might be found in a wireless access point. The system is architected around a 32-bit RISC processor. Flash memory is used for nonvolatile program and data storage. Main memory is synchronous dynamic random-access memory (SDRAM) and might contain anywhere from a few megabytes to hundreds of megabytes, depending on the application. A real-time clock module, often backed up by battery, keeps the time of day (calendar/wall clock, including date). This example includes an Ethernet and USB interface, as well as a serial port for console access via RS-232. The 802.11 chipset or module implements the wireless modem function.

Often the processor in an embedded system performs many functions beyond the traditional core instruction stream processing. The hypothetical processor shown in Figure 2-1 contains an integrated UART for a serial interface and integrated USB and Ethernet controllers. Many processors contain integrated peripherals. Sometimes they are referred to as system on chip (SOC). We look at several examples of integrated processors in Chapter 3, “Processor Basics.”

[1] ATCA platforms are introduced in Chapter 3.

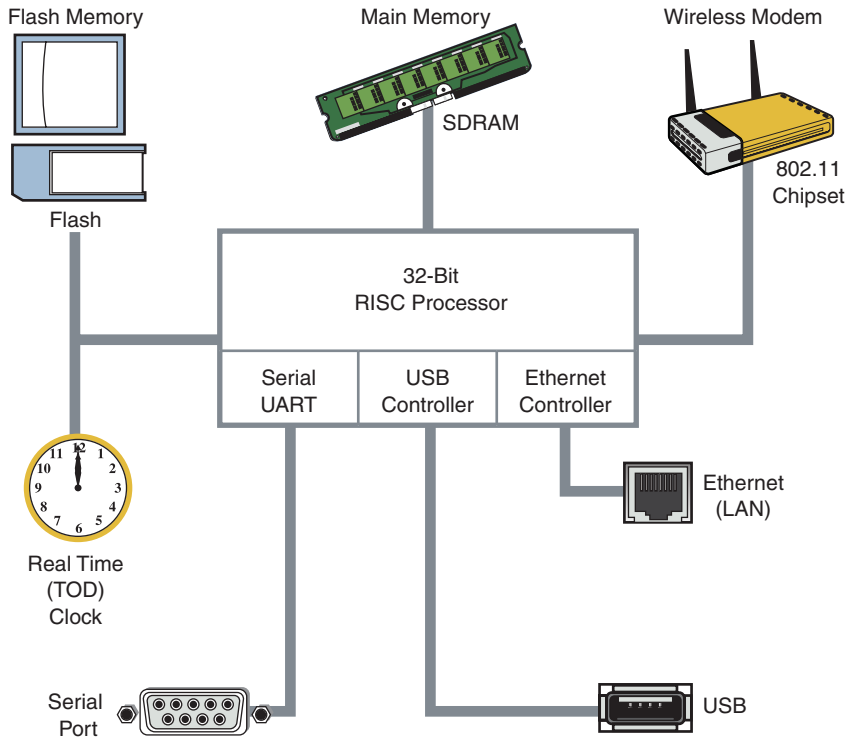

FIGURE 2-1 Embedded system

2.2.1 Typical Embedded Linux Setup

Often the first question posed by the newcomer to embedded Linux is, just what do you need to begin development? To answer that question, Figure 2-2 shows a typical embedded Linux development setup.

Figure 2-2 is a common arrangement. It shows a host development system, running your favorite desktop Linux distribution, such as Red Hat, SUSE, or Ubuntu Linux. The embedded Linux target board is connected to the development host via an RS-232 serial cable. You plug the target board's Ethernet interface into a local Ethernet hub or switch, to which your development host is also attached via Ethernet. The development host contains your development tools and utilities along with target files, which normally are obtained from an embedded Linux distribution.

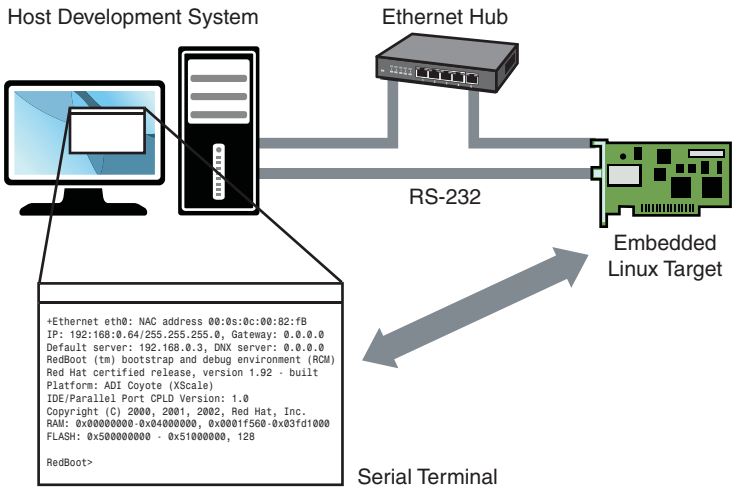

FIGURE 2-2 Embedded Linux development setup

For this example, our primary connection to the embedded Linux target is via the RS-232 connection. A serial terminal program is used to communicate with the target board. Minicom is one of the most commonly used serial terminal applications and is available on virtually all desktop Linux distributions.[2] The author has switched to using `screen` as his terminal of choice, replacing the functionality of minicom. It offers much more flexibility, especially for capturing traces, and it's more forgiving of serial line garbage often encountered during system bringup or troubleshooting. To use `screen` in this manner on a USB-attached serial dongle, simply invoke it on your serial terminal and specify the speed:

```
$ screen /dev/ttyUSB0 115200
```

2.2.2 Starting the Target Board

When power is first applied, a bootloader supplied with your target board takes immediate control of the processor. It performs some very low-level hardware initialization, including processor and memory setup, initialization of the UART controlling the serial port, and initialization of the Ethernet controller. Listing 2-1 displays the characters received from the serial port, resulting from power being applied to the target.

[2] You may have to install minicom from your distribution's repository. On Ubuntu, for example, you would execute `sudo apt-get install minicom` to install minicom on your desktop.

For this example, we have chosen a target board from Freescale Semiconductor, the PowerQUICC III MPC8548 Configurable Development System (CDS). It contains the MPC8548 PowerQUICC III processor. It ships from Freescale with the U-Boot bootloader preinstalled.

LISTING 2-1 Initial Bootloader Serial Output

```
U-Boot 2009.01 (May 20 2009 - 09:45:35)

CPU:   8548E, Version: 2.1, (0x80390021)
Core:  E500, Version: 2.2, (0x80210022)
Clock Configuration:
       CPU:990  MHz, CCB:396  MHz,
       DDR:198  MHz (396 MT/s data rate), LBC:49.500 MHz
L1:    D-cache 32 kB enabled
       I-cache 32 kB enabled
Board: CDS Version 0x13, PCI Slot 1
CPU Board Revision 0.0 (0x0000)
I2C:   ready
DRAM:  Initializing
    SDRAM: 64 MB
    DDR: 256 MB
FLASH: 16 MB
L2:    512 KB enabled
Invalid ID (ff ff ff ff)
    PCI: 64 bit, unknown MHz, async, host, external-arbiter
                Scanning PCI bus 00
PCI on bus 00 - 02

    PCIE connected to slot as Root Complex (base address e000a000)
PCIE on bus 3 - 3
In:    serial
Out:   serial
Err:   serial
Net:   eTSEC0, eTSEC1, eTSEC2, eTSEC3
=>
```

When power is applied to the MPC8548CDS board, U-Boot performs some low-level hardware initialization, which includes configuring a serial port. It then prints a banner line, as shown in the first line of Listing 2-1. Next the CPU and core are displayed, followed by some configuration data describing clocks and cache configuration. This is followed by a text string describing the board.

When the initial hardware configuration is complete, U-Boot configures any hardware subsystems as directed by its static configuration. Here we see I2C, DRAM, FLASH, L2 cache, PCI, and network subsystems being configured by U-Boot. Finally, U-Boot waits for input from the console over the serial port, as indicated by the => prompt.

2.2.3 Booting the Kernel

Now that U-Boot has initialized the hardware, serial port, and Ethernet network interfaces, it has only one job left in its short but useful life span: to load and boot the Linux kernel. All bootloaders have a command to load and execute an operating system image. Listing 2-2 shows one of the more common ways U-Boot is used to manually load and boot a Linux kernel.

LISTING 2-2 Loading the Linux Kernel

```
=> tftp 600000 uImage
Speed: 1000, full duplex
Using eTSEC0 device
TFTP from server 192.168.0.103; our IP address is 192.168.0.18
Filename 'uImage'.
Load address: 0x600000
Loading: ###################################################################
         ###################################################################
done
Bytes transferred = 1838553 (1c0dd9 hex)
=> tftp c00000 dtb
Speed: 1000, full duplex
Using eTSEC0 device
TFTP from server 192.168.0.103; our IP address is 192.168.0.18
Filename 'dtb'.
Load address: 0xc00000
Loading: ##
done
Bytes transferred = 16384 (4000 hex)
=> bootm 600000 - c00000
## Booting kernel from Legacy Image at 00600000 ...
   Image Name:   MontaVista Linux 6/2.6.27/freesc
   Image Type:   PowerPC Linux Kernel Image (gzip compressed)
   Data Size:    1838489 Bytes =  1.8 MB
   Load Address: 00000000
   Entry Point:  00000000
```

LISTING 2-2 Continued

```
   Verifying Checksum ... OK
## Flattened Device Tree blob at 00c00000
   Booting using the fdt blob at 0xc00000
   Uncompressing Kernel Image ... OK
   Loading Device Tree to 007f9000, end 007fffff ... OK
Using MPC85xx CDS machine description
Memory CAM mapping: CAM0=256Mb, CAM1=0Mb, CAM2=0Mb residual: 0Mb

...
< Lots of Linux kernel boot messages, removed for clarity >
...

freescale-8548cds login:  <<--- Linux login prompt
```

The `tftp` command at the start of Listing 2-2 instructs U-Boot to load the kernel image `uImage` into memory over the network using the TFTP[3] protocol. The kernel image, in this case, is located on the development workstation (usually the same machine that has the serial port connected to the target board). The `tftp` command is passed an address that is the physical address in the target board's memory where the kernel image will be loaded. Don't worry about the details now; Chapter 7 covers U-Boot in much greater detail.

The second invocation of the `tftp` command loads a board configuration file called a *device tree*. It is referred to by other names, including *flat device tree* and *device tree binary* or dtb. You will learn more about this file in Chapter 7. For now, it is enough for you to know that this file contains board-specific information that the kernel requires in order to boot the board. This includes things such as memory size, clock speeds, onboard devices, buses, and Flash layout.

Next, the `bootm` (boot from memory image) command is issued, to instruct U-Boot to boot the kernel we just loaded from the address specified by the `tftp` command. In this example of using the `bootm` command, we instruct U-Boot to load the kernel that we put at `0x600000` and pass the device tree binary (dtb) we loaded at `0xc00000` to the kernel. This command transfers control to the Linux kernel. Assuming that your kernel is properly configured, this results in booting the Linux kernel to a console command prompt on your target board, as shown by the login prompt.

Note that the `bootm` command is the death knell for U-Boot. This is an important concept. Unlike the BIOS in a desktop PC, most embedded systems are architected

[3] This and other servers you will be using are covered in detail in Chapter 12, "Embedded Development Environment."

in such a way that when the Linux kernel takes control, the bootloader ceases to exist. The kernel claims any memory and system resources that the bootloader previously used. The only way to pass control back to the bootloader is to reboot the board.

One final observation is worth noting. All the serial output in Listing 2-2 up to and including this line is produced by the U-Boot bootloader:

```
Loading Device Tree to 007f9000, end 007fffff ... OK
```

The rest of the boot messages are produced by the Linux kernel. We'll have much more to say about this later, but it is worth noting where U-Boot leaves off and where the Linux kernel image takes over.

2.2.4 Kernel Initialization: Overview

When the Linux kernel begins execution, it spews out numerous status messages during its rather comprehensive boot process. In the example being discussed here, the Linux kernel displayed approximately 200 `printk`[4] lines before it issues the login prompt. (We omitted them from the listing to clarify the point being discussed.) Listing 2-3 reproduces the last several lines of output before the login prompt. The goal of this exercise is not to delve into the details of the kernel initialization (this is covered in Chapter 5, "Kernel Initialization"). The goal is to gain a high-level understanding of what is happening and what components are required to boot a Linux kernel on an embedded system.

LISTING 2-3 Linux Final Boot Messages

```
...
Looking up port of RPC 100005/1 on 192.168.0.9
VFS: Mounted root (nfs filesystem).
Freeing unused kernel memory: 152k init
INIT: version 2.86 booting
...

freescale-8548cds login:
```

Shortly before issuing a login prompt on the serial terminal, Linux *mounts* a *root file system*. In Listing 2-3, Linux goes through the steps required to mount its root file system remotely (via Ethernet) from an NFS[5] server on a machine with the IP

[4] `printk()` is the function in the kernel responsible for displaying messages to the system console.

[5] NFS and other required servers are covered in Chapter 12.

address 192.168.0.9. Usually, this is your development workstation. The root file system contains the application programs, system libraries, and utilities that make up a Linux system.

The important point in this discussion should not be understated: *Linux requires a file system*. Many legacy embedded operating systems did not require a file system. This fact is a frequent surprise to engineers making the transition from legacy embedded OSs to embedded Linux. A file system consists of a predefined set of system directories and files in a specific layout on a hard drive or other medium that the Linux kernel *mounts* as its root file system.

Note that Linux can mount a root file system from other devices. The most common, of course, is to mount a partition from a hard drive as the root file system, as is done on your Linux laptop or workstation. Indeed, NFS is pretty useless when you ship your embedded Linux widget out the door and away from your development environment. However, as you progress through this book, you will come to appreciate the power and flexibility of NFS root mounting as a development environment.

2.2.5 First User Space Process: `init`

Another important point should be made before we move on. Notice in Listing 2-3 this line:

```
INIT: version 2.86 booting
```

Until this point, the kernel itself was executing code, performing the numerous initialization steps in a context known as *kernel context*. In this operational state, the kernel owns all system memory and operates with full authority over all system resources. The kernel has access to all physical memory and to all I/O subsystems. It executes code in kernel virtual address space, using a stack created and owned by the kernel itself.

When the Linux kernel has completed its internal initialization and mounted its root file system, the default behavior is to spawn an application program called `init`. When the kernel starts `init`, it is said to be running in *user space* or user space context. In this operational mode, the user space process has restricted access to the system and must use kernel system calls to request kernel services such as device and file I/O. These user space processes, or programs, operate in a virtual memory space picked at random[6] and managed by the kernel. The kernel, in cooperation with specialized memory-management hardware in the processor, performs virtual-to-physical address translation for the user space process. The single biggest benefit of this architecture is that an

[6] It's not actually random, but for purposes of this discussion, it might as well be. This topic will be covered in more detail later.

error in one process can't trash the memory space of another. This is a common pitfall in legacy embedded OSs that can lead to bugs that are some of the most difficult to track down.

Don't be alarmed if these concepts seem foreign. The objective of this section is to paint a broad picture from which you will develop more detailed knowledge as you progress through the book. These and other concepts are covered in great detail in later chapters.

2.3 Storage Considerations

One of the most challenging aspects of embedded Linux development is that most embedded systems have limited physical resources. Although the Core™ 2 Duo machine on your desktop might have 500GB of hard drive space, it is not uncommon to find embedded systems with a fraction of that amount. In many cases, the hard drive typically is replaced by smaller and less expensive nonvolatile storage devices. Hard drives are bulky, have rotating parts, are sensitive to physical shock, and require multiple power supply voltages, which makes them unsuitable for many embedded systems.

2.3.1 Flash Memory

Nearly everyone is familiar with Compact Flash and SD cards used in a wide variety of consumer devices, such as digital cameras and PDAs (both great examples of embedded systems). These modules, based on *Flash* memory technology, can be thought of as solid-state hard drives, capable of storing many megabytes—and even gigabytes—of data in a tiny footprint. They contain no moving parts, are relatively rugged, and operate on a single common power supply voltage.

Several manufacturers of Flash memory exist. Flash memory comes in a variety of electrical formats, physical packages, and capacities. It is not uncommon to see embedded systems with as little as 4MB or 8MB of nonvolatile storage. More typical storage requirements for embedded Linux systems range from 16MB to 256MB or more. An increasing number of embedded Linux systems have nonvolatile storage into the gigabyte range.

Flash memory can be written to and erased under software control. Rotational hard drive technology remains the fastest writable medium. Flash writing and erasing speeds have improved considerably over time, although Flash write and erase time is still considerably slower. You must understand some fundamental differences between hard drive and Flash memory technology to properly use the technology.

Flash memory is divided into relatively large erasable units, referred to as erase blocks. One of the defining characteristics of Flash memory is how data in Flash is written and erased. In a typical NOR[7] Flash memory chip, data can be changed from a binary 1 to a binary 0 under software control using simple data writes directly to the cell's address, one bit or word at a time. However, to change a bit from a 0 back to a 1, an entire *erase block* must be erased using a special sequence of control instructions to the Flash chip.

A typical NOR Flash memory device contains many erase blocks. For example, a 4MB Flash chip might contain 64 erase blocks of 64KB each. Flash memory is also available with nonuniform erase block sizes, to facilitate flexible data-storage layouts. These are commonly called boot block or boot sector Flash chips. Often the bootloader is stored in the smaller blocks, and the kernel and other required data are stored in the larger blocks. Figure 2-3 illustrates the block size layout for a typical top boot Flash.

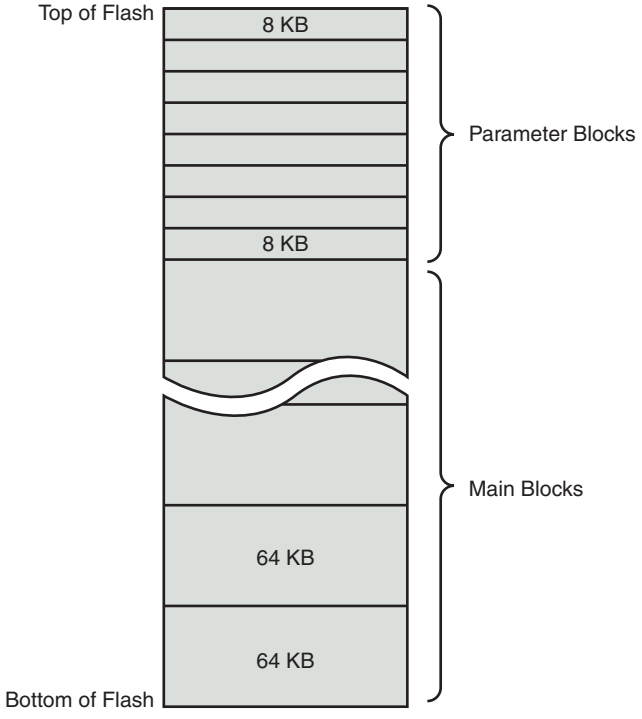

FIGURE 2-3 Boot block Flash architecture

[7] There are several types of Flash technologies. NOR Flash is one of the most commonly used in small embedded systems.

To modify data stored in a Flash memory array, the block in which the modified data resides must be completely erased. Even if only 1 byte in a block needs to be changed, the entire block must be erased and rewritten.[8] Flash block sizes are relatively large compared to traditional hard-drive sector sizes. In comparison, a typical high-performance hard drive has writable sectors of 512 or 1024 bytes. The ramifications of this might be obvious: Write times for updating data in Flash memory can be many times that of a hard drive, due in part to the relatively large quantity of data that must be erased and written back to the Flash for each update. In the worst case, these write cycles can take several seconds.

Another limitation of Flash memory that must be considered is Flash memory cell write lifetime. A NOR Flash memory cell has a limited number of write cycles before failure. Although the number of cycles is fairly large (100,000 cycles per block is typical), it is easy to imagine a poorly designed Flash storage algorithm (or even a bug) that can quickly destroy Flash devices. It goes without saying that you should avoid configuring your system loggers to output to a Flash-based device.

2.3.2 NAND Flash

NAND Flash is a relatively new Flash technology. When NAND Flash hit the market, traditional Flash memory such as that described in the preceding section was called NOR Flash. These distinctions relate to the internal Flash memory cell architecture. NAND Flash devices improve on some of the limitations of traditional (NOR) Flash by offering smaller block sizes, resulting in faster and more efficient writes and generally more efficient use of the Flash array.

NOR Flash devices interface to the microprocessor in a fashion similar to many microprocessor peripherals. That is, they have a parallel data and address bus that are connected directly[9] to the microprocessor data/address bus. Each byte or word in the Flash array can be individually addressed in a random fashion. In contrast, NAND devices are accessed serially through a complex interface that varies among vendors. NAND devices present an operational model more similar to that of a traditional hard drive and associated controller. Data is accessed in serial bursts, which are far smaller than NOR Flash block size. Write cycle lifetime for NAND Flash is an order of magnitude greater than for NOR Flash, although erase times are significantly smaller.

[8] Remember, you can change a 1 to a 0 a byte at a time, but you must erase the entire block to change any bit from a 0 to a 1.

[9] Directly in the logical sense. The actual circuitry may contain bus buffers or bridge devices and so on.

In summary, NOR Flash can be directly accessed by the microprocessor, and code can even be executed directly out of NOR Flash. (However, for performance reasons, this is rarely done, and then only on systems in which resources are extremely scarce.) In fact, many processors cannot cache instruction accesses to Flash, as they can with DRAM. This further degrades execution speed. In contrast, NAND Flash is more suitable for bulk storage in file system format than raw binary executable code and data storage.

2.3.3 Flash Usage

An embedded system designer has many options in the layout and use of Flash memory. In the simplest of systems, in which resources are not overly constrained, raw binary data (perhaps compressed) can be stored on the Flash device. When booted, a file system image stored in Flash is read into a Linux ramdisk block device, mounted as a file system, and accessed only from RAM. This is often a good design choice when the data in Flash rarely needs to be updated. Any data that does need to be updated is relatively small compared to the size of the ramdisk. It is important to realize that any changes to files in the ramdisk are lost upon reboot or power cycle.

Figure 2-4 illustrates a common Flash memory organization that is typical of a simple embedded system in which nonvolatile storage requirements of dynamic data are small and infrequent.

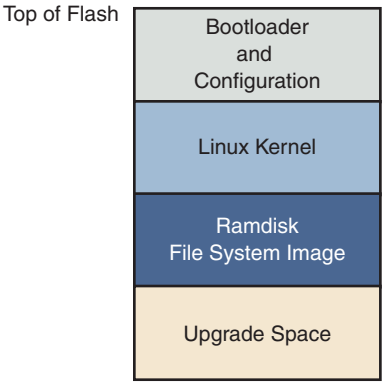

FIGURE 2-4 Typical Flash memory layout

The bootloader is often placed in the top or bottom of the Flash memory array. Following the bootloader, space is allocated for the Linux kernel image and the ramdisk

file system image,[10] which holds the root file system. Typically, the Linux kernel and ramdisk file system images are compressed, and the bootloader handles the decompression task during the boot cycle.

For dynamic data that needs to be saved between reboots and power cycles, another small area of Flash can be dedicated, or another type of nonvolatile storage[11] can be used. This is a typical configuration for embedded systems that have requirements to store configuration data, as might be found in a wireless access point aimed at the consumer market, for example.

2.3.4 Flash File Systems

The limitations of the simple Flash layout scheme just described can be overcome by using a Flash file system to manage data on the Flash device in a manner similar to how data is organized on a hard drive. Early implementations of file systems for Flash devices consisted of a simple block device layer that emulated the 512-byte sector layout of a common hard drive. These simple emulation layers allowed access to data in file format rather than unformatted bulk storage, but they had some performance limitations.

One of the first enhancements to Flash file systems was the incorporation of wear leveling. As discussed earlier, Flash blocks are subject to a finite write lifetime. Wear-leveling algorithms are used to distribute writes evenly over the physical erase blocks of the Flash memory in order to extend the life of the Flash memory chip.

Another limitation that arises from the Flash architecture is the risk of data loss during a power failure or premature shutdown. Consider that the Flash block sizes are relatively large and that average file sizes being written are often much smaller relative to the block size. You learned previously that Flash blocks must be written one block at a time. Therefore, to write a small 8KB file, you must erase and rewrite an entire Flash block, perhaps 64KB or 128KB in size; in the worst case, this can take several seconds to complete. This opens a significant window to risk of data loss due to power failure.

One of the more popular Flash file systems in use today is JFFS2, or Journaling Flash File System 2. It has several important features aimed at improving overall performance, increasing Flash lifetime, and reducing the risk of data loss in the case of power failure. The more significant improvements in the latest JFFS2 file system include improved wear leveling, compression and decompression to squeeze more data

[10] We discuss ramdisk file systems in more detail in Chapter 9, "File Systems."

[11] Real-time clock modules and serial EEPROMs are often choices for nonvolatile storage of small amounts of data.

into a given Flash size, and support for Linux hard links. This topic is covered in detail in Chapter 9 and in Chapter 10, "MTD Subsystem," when we discuss the Memory Technology Device (MTD) subsystem.

2.3.5 Memory Space

Virtually all legacy embedded operating systems view and manage system memory as a single large, flat address space. That is, a microprocessor's address space exists from 0 to the top of its physical address range. For example, if a microprocessor had 24 physical address lines, its top of memory would be 16MB. Therefore, its hexadecimal address would range from `0x00000000` to `0x00ffffff`. Hardware designs commonly place DRAM starting at the bottom of the range, and Flash memory from the top down. Unused address ranges between the top of DRAM and bottom of Flash would be allocated for addressing of various peripheral chips on the board. This design approach is often dictated by the choice of microprocessor. Figure 2-5 shows a typical memory layout for a simple embedded system.

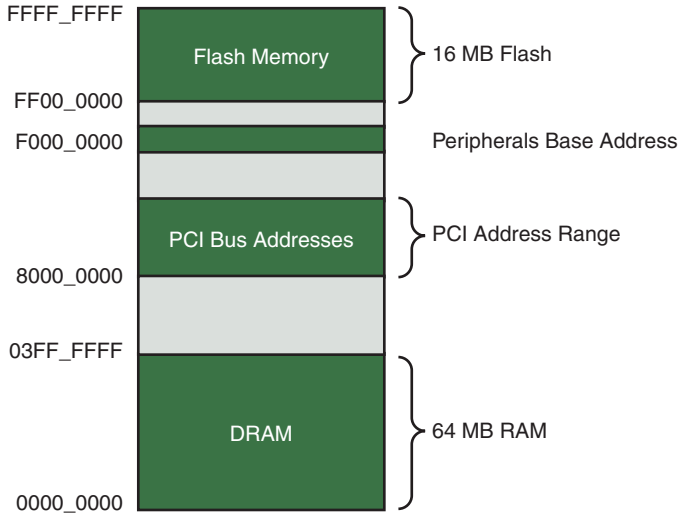

FIGURE 2-5 Typical embedded system memory map

In traditional embedded systems based on legacy operating systems, the OS and all the tasks[12] had equal access rights to all resources in the system. A bug in one process could wipe out memory contents anywhere in the system, whether it belonged to itself, the OS, another task, or even a hardware register somewhere in the address space. Although this approach had simplicity as its most valuable characteristic, it led to bugs that could be challenging to diagnose.

High-performance microprocessors contain complex hardware engines called Memory Management Units (MMUs). Their purpose is to enable an operating system to exercise a high degree of management and control over its address space and the address space it allocates to processes. This control comes in two primary forms: *access rights* and *memory translation*. Access rights allow an operating system to assign specific memory-access privileges to specific tasks. Memory translation allows an operating system to virtualize its address space, which has many benefits.

The Linux kernel takes advantage of these hardware MMUs to create a *virtual memory* operating system. One of the biggest benefits of virtual memory is that it can make more efficient use of physical memory by presenting the appearance that the system has more memory than is physically present. The other benefit is that the kernel can enforce access rights to each range of system memory that it allocates to a task or process, to prevent one process from errantly accessing memory or other resources that belong to another process or to the kernel itself.

The next section examines in more detail how this works. A tutorial on the complexities of virtual memory systems is beyond the scope of this book.[13] Instead, we examine the ramifications of a virtual memory system as it appears to an embedded systems developer.

2.3.6 Execution Contexts

One of the very first chores that Linux performs is to configure the hardware MMU on the processor and the data structures used to support it, and to enable address translation. When this step is complete, the kernel runs in its own virtual memory space. The virtual kernel address selected by the kernel developers in recent Linux kernel versions defaults to `0xC0000000`. In most architectures, this is a configurable parameter.[14] If we

[12] In this discussion, the word task is used to denote any thread of execution, regardless of the mechanism used to spawn, manage, or schedule it.

[13] Many good books cover the details of virtual memory systems. See the last section of this chapter for recommendations.

[14] However, there is seldom a good reason to change it.

looked at the kernel's symbol table, we would find kernel symbols linked at an address starting with `0xC0xxxxxx`. As a result, any time the kernel executes code in kernel space, the processor's instruction pointer (program counter) contains values in this range.

In Linux, we refer to two distinctly separate operational contexts, based on the environment in which a given thread[15] is executing. Threads executing entirely within the kernel are said to be operating in *kernel context*. Application programs are said to operate in *user space context*. A user space process can access only memory it owns, and it is required to use kernel system calls to access privileged resources such as file and device I/O. An example might make this more clear.

Consider an application that opens a file and issues a read request, as shown in Figure 2-6. The read function call begins in user space, in the C library `read()` function. The C library then issues a read request to the kernel. The read request results in a context switch from the user's program to the kernel, to service the request for the file's data. Inside the kernel, the read request results in a hard-drive access requesting the sectors containing the file's data.

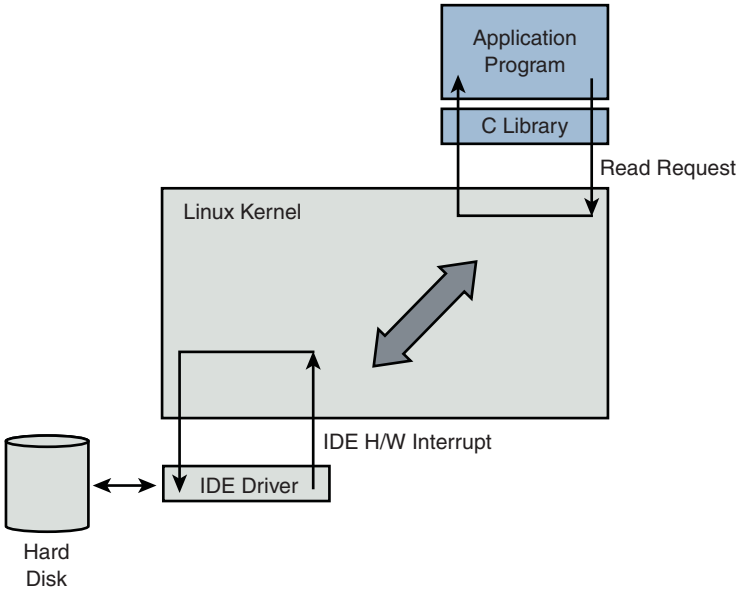

FIGURE 2-6 Simple file read request

[15] The term *thread* is used here in the generic sense to indicate any sequential flow of instructions.

Usually the hard-drive read request is issued asynchronously to the hardware itself. That is, the request is posted to the hardware, and when the data is ready, the hardware interrupts the processor. The application program waiting for the data is *blocked* on a wait queue until the data is available. Later, when the hard disk has the data ready, it posts a hardware interrupt. (This description is intentionally simplified for the purposes of this illustration.) When the kernel receives the hardware interrupt, it suspends whatever process was executing and proceeds to read the waiting data from the drive.

To summarize this discussion, we have identified two general execution contexts—user space and kernel space. When an application program executes a system call that results in a context switch and enters the kernel, it is executing kernel code on behalf of a process. You will often hear this referred to as *process context* within the kernel. In contrast, the interrupt service routine (ISR) handling the IDE drive (or any other ISR, for that matter) is kernel code that is not executing on behalf of any particular process. This is typically called *interrupt context*.

Several limitations exist in this operational context, including the limitation that the ISR cannot block (sleep) or call any kernel functions that might result in blocking. For further reading on these concepts, consult the references at the end of this chapter.

2.3.7 Process Virtual Memory

When a process is spawned—for example, when the user types `ls` at the Linux command prompt—the kernel allocates memory for the process and assigns a range of virtual-memory addresses to the process. The resulting address values bear no fixed relationship to those in the kernel, nor to any other running process. Furthermore, there is no direct correlation between the physical memory addresses on the board and the virtual memory as seen by the process. In fact, it is not uncommon for a process to occupy multiple different physical addresses in main memory during its lifetime as a result of paging and swapping.

Listing 2-4 is the venerable "Hello World," modified to illustrate the concepts just discussed. The goal of this example is to illustrate the address space that the kernel assigns to the process. This code was compiled and run on an embedded system containing 256MB of DRAM memory.

LISTING 2-4 Hello World, Embedded Style

```
#include <stdio.h>

int bss_var;        /* Uninitialized global variable */

int data_var = 1;   /* Initialized global variable */

int main(int argc, char **argv)
{
  void *stack_var;            /* Local variable on the stack */

  stack_var = (void *)main;   /* Don't let the compiler */
                              /* optimize it out */

  printf("Hello, World! Main is executing at %p\n", stack_var);
  printf("This address (%p) is in our stack frame\n", &stack_var);

  /* bss section contains uninitialized data */
  printf("This address (%p) is in our bss section\n", &bss_var);

  /* data section contains initializated data */
  printf("This address (%p) is in our data section\n", &data_var);

  return 0;
}
```

Listing 2-5 shows the console output that this program produces. Notice that the process called `hello` thinks it is executing somewhere in high RAM just above the 256MB boundary (`0x10000418`). Notice also that the stack address is roughly halfway into a 32-bit address space, well beyond our 256MB of RAM (`0x7ff8ebb0`). How can this be? DRAM is usually contiguous in systems like these. To the casual observer, it appears that we have nearly 2GB of DRAM available for our use. These *virtual addresses* were assigned by the kernel and are backed by physical RAM somewhere within the 256MB range of available memory on our embedded board.

LISTING 2-5 Hello Output

```
root@192.168.4.9:~# ./hello
Hello, World! Main is executing at 0x10000418
This address (0x7ff8ebb0) is in our stack frame
This address (0x10010a1c) is in our bss section
This address (0x10010a18) is in our data section
root@192.168.4.9:~#
```