

PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Embedded Linux Primer

Second Edition

A Practical Real-World Approach



Christopher Hallinan

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informati.com/aw

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

Copyright © 2011 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, write to:

Pearson Education, Inc.

Rights and Contracts Department
501 Boylston Street, Suite 900
Boston, MA 02116
Fax (617) 671-3447

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

Contents

Foreword for the First Edition	xxv
Foreword for the Second Edition.....	xxvi
Preface.....	xxvii
Acknowledgments for the First Edition	xxxiii
Acknowledgments for the Second Edition	xxxv
About the Author	xxxvi
Chapter 1 Introduction.....	1
1.1 Why Linux?.....	2
1.2 Embedded Linux Today	3
1.3 Open Source and the GPL	3
1.3.1 Free Versus Freedom	4
1.4 Standards and Relevant Bodies	5
1.4.1 Linux Standard Base	5
1.4.2 Linux Foundation	6
1.4.3 Carrier-Grade Linux	6
1.4.4 Mobile Linux Initiative: Moblin.....	7
1.4.5 Service Availability Forum.....	7
1.5 Summary.....	8
1.5.1 Suggestions for Additional Reading.....	8

Chapter 2 The Big Picture	9
2.1 Embedded or Not?	10
2.1.1 BIOS Versus Bootloader	11
2.2 Anatomy of an Embedded System.....	12
2.2.1 Typical Embedded Linux Setup	13
2.2.2 Starting the Target Board	14
2.2.3 Booting the Kernel.....	16
2.2.4 Kernel Initialization: Overview	18
2.2.5 First User Space Process: <code>init</code>	19
2.3 Storage Considerations.....	20
2.3.1 Flash Memory.....	20
2.3.2 NAND Flash	22
2.3.3 Flash Usage.....	23
2.3.4 Flash File Systems	24
2.3.5 Memory Space	25
2.3.6 Execution Contexts.....	26
2.3.7 Process Virtual Memory.....	28
2.3.8 Cross-Development Environment.....	30
2.4 Embedded Linux Distributions	32
2.4.1 Commercial Linux Distributions	33
2.4.2 Do-It-Yourself Linux Distributions	33
2.5 Summary.....	34
2.5.1 Suggestions for Additional Reading.....	35
Chapter 3 Processor Basics	37
3.1 Stand-Alone Processors.....	38
3.1.1 IBM 970FX.....	39
3.1.2 Intel Pentium M	39

3.1.3	Intel Atom™	40
3.1.4	Freescale MPC7448	40
3.1.5	Companion Chipsets	41
3.2	Integrated Processors: Systems on Chip	43
3.2.1	Power Architecture.....	44
3.2.2	Freescale Power Architecture	44
3.2.3	Freescale PowerQUICC I.....	45
3.2.4	Freescale PowerQUICC II	46
3.2.5	PowerQUICC II Pro.....	47
3.2.6	Freescale PowerQUICC III	48
3.2.7	Freescale QorIQ™	48
3.2.8	AMCC Power Architecture	50
3.2.9	MIPS	53
3.2.10	Broadcom MIPS	54
3.2.11	Other MIPS.....	55
3.2.12	ARM	55
3.2.13	TI ARM.....	56
3.2.14	Freescale ARM	58
3.2.15	Other ARM Processors	59
3.3	Other Architectures.....	59
3.4	Hardware Platforms	60
3.4.1	CompactPCI	60
3.4.2	ATCA	60
3.5	Summary.....	61
3.5.1	Suggestions for Additional Reading.....	62

Chapter 4 The Linux Kernel: A Different Perspective.....	63
4.1 Background.....	64
4.1.1 Kernel Versions	65
4.1.2 Kernel Source Repositories.....	67
4.1.3 Using git to Download a Kernel	68
4.2 Linux Kernel Construction.....	68
4.2.1 Top-Level Source Directory.....	69
4.2.2 Compiling the Kernel	69
4.2.3 The Kernel Proper: vmlinux	72
4.2.4 Kernel Image Components	73
4.2.5 Subdirectory Layout.....	77
4.3 Kernel Build System	78
4.3.1 The Dot-Config.....	78
4.3.2 Configuration Editor(s)	80
4.3.3 Makefile Targets.....	83
4.4 Kernel Configuration	89
4.4.1 Custom Configuration Options	91
4.4.2 Kernel Makefiles	95
4.5 Kernel Documentation.....	96
4.6 Obtaining a Custom Linux Kernel	96
4.6.1 What Else Do I Need?	97
4.7 Summary.....	97
4.7.1 Suggestions for Additional Reading.....	98
Chapter 5 Kernel Initialization.....	99
5.1 Composite Kernel Image: Piggy and Friends	100
5.1.1 The Image Object.....	103
5.1.2 Architecture Objects	104

5.1.3	Bootstrap Loader	105
5.1.4	Boot Messages.....	106
5.2	Initialization Flow of Control.....	109
5.2.1	Kernel Entry Point: <code>head.o</code>	111
5.2.2	Kernel Startup: <code>main.c</code>	113
5.2.3	Architecture Setup	114
5.3	Kernel Command-Line Processing	115
5.3.1	The <code>__setup</code> Macro	116
5.4	Subsystem Initialization.....	122
5.4.1	The <code>* __initcall</code> Macros	122
5.5	The <code>init</code> Thread.....	125
5.5.1	Initialization Via <code>initcalls</code>	126
5.5.2	<code>initcall_debug</code>	127
5.5.3	Final Boot Steps.....	127
5.6	Summary.....	129
5.6.1	Suggestions for Additional Reading.....	130
Chapter 6	User Space Initialization	131
6.1	Root File System	132
6.1.1	FHS: File System Hierarchy Standard	133
6.1.2	File System Layout.....	133
6.1.3	Minimal File System	134
6.1.4	The Embedded Root FS Challenge	136
6.1.5	Trial-and-Error Method	137
6.1.6	Automated File System Build Tools.....	137

6.2	Kernel's Last Boot Steps.....	137
6.2.1	First User Space Program	139
6.2.2	Resolving Dependencies.....	139
6.2.3	Customized Initial Process	140
6.3	The init Process.....	140
6.3.1	inittab.....	143
6.3.2	Sample Web Server Startup Script.....	145
6.4	Initial RAM Disk	146
6.4.1	Booting with initrd.....	147
6.4.2	Bootloader Support for initrd	148
6.4.3	initrd Magic: linuxrc	150
6.4.4	The initrd Plumbing.....	151
6.4.5	Building an initrd Image.....	152
6.5	Using initramfs.....	153
6.5.1	Customizing initramfs.....	154
6.6	Shutdown.....	156
6.7	Summary.....	156
6.7.1	Suggestions for Additional Reading.....	157
Chapter 7	Bootloaders	159
7.1	Role of a Bootloader.....	160
7.2	Bootloader Challenges.....	161
7.2.1	DRAM Controller	161
7.2.2	Flash Versus RAM.....	162
7.2.3	Image Complexity.....	162
7.2.4	Execution Context	165

7.3	A Universal Bootloader: Das U-Boot.....	166
7.3.1	Obtaining U-Boot	166
7.3.2	Configuring U-Boot	167
7.3.3	U-Boot Monitor Commands	169
7.3.4	Network Operations	170
7.3.5	Storage Subsystems	173
7.3.6	Booting from Disk.....	174
7.4	Porting U-Boot	174
7.4.1	EP405 U-Boot Port	175
7.4.2	U-Boot Makefile Configuration Target	176
7.4.3	EP405 First Build	177
7.4.4	EP405 Processor Initialization.....	178
7.4.5	Board-Specific Initialization	181
7.4.6	Porting Summary.....	184
7.4.7	U-Boot Image Format.....	185
7.5	Device Tree Blob (Flat Device Tree).....	187
7.5.1	Device Tree Source.....	189
7.5.2	Device Tree Compiler	192
7.5.3	Alternative Kernel Images Using DTB	193
7.6	Other Bootloaders.....	194
7.6.1	Lilo	194
7.6.2	GRUB	195
7.6.3	Still More Bootloaders	197
7.7	Summary.....	197
7.7.1	Suggestions for Additional Reading.....	198

Chapter 8 Device Driver Basics	201
8.1 Device Driver Concepts	202
8.1.1 Loadable Modules.....	203
8.1.2 Device Driver Architecture	204
8.1.3 Minimal Device Driver Example.....	204
8.1.4 Module Build Infrastructure	205
8.1.5 Installing a Device Driver	209
8.1.6 Loading a Module.....	210
8.1.7 Module Parameters	211
8.2 Module Utilities	212
8.2.1 insmod	212
8.2.2 lsmod	213
8.2.3 modprobe.....	213
8.2.4 depmod	214
8.2.5 rmmod	215
8.2.6 modinfo	216
8.3 Driver Methods.....	217
8.3.1 Driver File System Operations	217
8.3.2 Allocation of Device Numbers	220
8.3.3 Device Nodes and mknod	220
8.4 Bringing It All Together.....	222
8.5 Building Out-of-Tree Drivers	223
8.6 Device Drivers and the GPL.....	224
8.7 Summary.....	225
8.7.1 Suggestions for Additional Reading.....	226

Chapter 9 File Systems.....	227
9.1 Linux File System Concepts	228
9.1.1 Partitions	229
9.2 ext2	230
9.2.1 Mounting a File System	232
9.2.2 Checking File System Integrity	233
9.3 ext3	235
9.4 ext4	237
9.5 ReiserFS	238
9.6 JFFS2.....	239
9.6.1 Building a JFFS2 Image	240
9.7 <i>cramfs</i>	242
9.8 Network File System	244
9.8.1 Root File System on NFS.....	246
9.9 Pseudo File Systems.....	248
9.9.1 /proc File System.....	249
9.9.2 sysfs	252
9.10 Other File Systems	255
9.11 Building a Simple File System	256
9.12 Summary.....	258
9.12.1 Suggestions for Additional Reading.....	259
Chapter 10 MTD Subsystem	261
10.1 MTD Overview	262
10.1.1 Enabling MTD Services.....	263
10.1.2 MTD Basics.....	265
10.1.3 Configuring MTD on Your Target.....	267

10.2 MTD Partitions	267
10.2.1 Redboot Partition Table Partitioning.....	269
10.2.2 Kernel Command-Line Partitioning	273
10.2.3 Mapping Driver.....	274
10.2.4 Flash Chip Drivers.....	276
10.2.5 Board-Specific Initialization.....	276
10.3 MTD Utilities.....	279
10.3.1 JFFS2 Root File System	281
10.4 UBI File System	284
10.4.1 Configuring for UBIFS.....	284
10.4.2 Building a UBIFS Image.....	284
10.4.3 Using UBIFS as the Root File System	287
10.5 Summary.....	287
10.5.1 Suggestions for Additional Reading.....	288
Chapter 11 BusyBox	289
11.1 Introduction to BusyBox.....	290
11.1.1 BusyBox Is Easy	291
11.2 BusyBox Configuration.....	291
11.2.1 Cross-Compiling BusyBox.....	293
11.3 BusyBox Operation.....	293
11.3.1 BusyBox init	297
11.3.2 Sample rcs Initialization Script	299
11.3.3 BusyBox Target Installation.....	300
11.3.4 BusyBox Applets	302
11.4 Summary.....	303
11.4.1 Suggestions for Additional Reading.....	304

Chapter 12 Embedded Development Environment	305
12.1 Cross-Development Environment	306
12.1.1 “Hello World” Embedded.....	307
12.2 Host System Requirements.....	311
12.2.1 Hardware Debug Probe	311
12.3 Hosting Target Boards.....	312
12.3.1 TFTP Server	312
12.3.2 BOOTP/DHCP Server.....	313
12.3.3 NFS Server	316
12.3.4 Target NFS Root Mount.....	318
12.3.5 U-Boot NFS Root Mount Example	320
12.4 Summary.....	322
12.4.1 Suggestions for Additional Reading.....	323
Chapter 13 Development Tools	325
13.1 GNU Debugger (GDB)	326
13.1.1 Debugging a Core Dump	327
13.1.2 Invoking GDB.....	329
13.1.3 Debug Session in GDB.....	331
13.2 Data Display Debugger	333
13.3 cbrowser/cscope.....	335
13.4 Tracing and Profiling Tools.....	337
13.4.1 strace	337
13.4.2 strace Variations.....	341
13.4.3 ltrace	343
13.4.4 ps	344
13.4.5 top.....	346

13.4.6	mtrace	348
13.4.7	dmalloc	350
13.4.8	Kernel Oops	353
13.5	Binary Utilities	355
13.5.1	readelf	355
13.5.2	Examining Debug Information Using readelf	357
13.5.3	objdump	359
13.5.4	objcopy	360
13.6	Miscellaneous Binary Utilities	361
13.6.1	strip	361
13.6.2	addr2line	361
13.6.3	strings	362
13.6.4	ldd	362
13.6.5	nm	363
13.6.6	prelink	364
13.7	Summary	364
13.7.1	Suggestions for Additional Reading	365
Chapter 14	Kernel Debugging Techniques	367
14.1	Challenges to Kernel Debugging	368
14.2	Using KGDB for Kernel Debugging	369
14.2.1	KGDB Kernel Configuration	371
14.2.2	Target Boot with KGDB Support	372
14.2.3	Useful Kernel Breakpoints	376
14.2.4	Sharing a Console Serial Port with KGDB	377
14.2.5	Debugging Very Early Kernel Code	379
14.2.6	KGDB Support in the Mainline Kernel	380

14.3 Kernel Debugging Techniques.....	381
14.3.1 gdb Remote Serial Protocol.....	382
14.3.2 Debugging Optimized Kernel Code.....	385
14.3.3 GDB User-Defined Commands.....	392
14.3.4 Useful Kernel GDB Macros	393
14.3.5 Debugging Loadable Modules.....	402
14.3.6 <code>printf</code> Debugging.....	407
14.3.7 Magic SysReq Key	409
14.4 Hardware-Assisted Debugging.....	410
14.4.1 Programming Flash Using a JTAG Probe	411
14.4.2 Debugging with a JTAG Probe	413
14.5 When It Doesn't Boot	417
14.5.1 Early Serial Debug Output	417
14.5.2 Dumping the <code>printf</code> Log Buffer	417
14.5.3 KGDB on Panic.....	420
14.6 Summary.....	421
14.6.1 Suggestions for Additional Reading.....	422
Chapter 15 Debugging Embedded Linux Applications	423
15.1 Target Debugging.....	424
15.2 Remote (Cross) Debugging	424
15.2.1 <code>gdbserver</code>	427
15.3 Debugging with Shared Libraries	429
15.3.1 Shared Library Events in GDB.....	431
15.4 Debugging Multiple Tasks.....	435
15.4.1 Debugging Multiple Processes.....	435
15.4.2 Debugging Multithreaded Applications	438
15.4.3 Debugging Bootloader/Flash Code	441

15.5	Additional Remote Debug Options.....	442
15.5.1	Debugging Using a Serial Port	442
15.5.2	Attaching to a Running Process	442
15.6	Summary.....	443
15.6.1	Suggestions for Additional Reading.....	444
Chapter 16	Open Source Build Systems	445
16.1	Why Use a Build System?	446
16.2	Scratchbox.....	447
16.2.1	Installing Scratchbox.....	447
16.2.2	Creating a Cross-Compilation Target.....	448
16.3	Buildroot.....	451
16.3.1	Buildroot Installation.....	451
16.3.2	Buildroot Configuration	451
16.3.3	Buildroot Build.....	452
16.4	OpenEmbedded	454
16.4.1	OpenEmbedded Composition	455
16.4.2	BitBake Metadata.....	456
16.4.3	Recipe Basics.....	456
16.4.4	Metadata Tasks.....	460
16.4.5	Metadata Classes.....	461
16.4.6	Configuring OpenEmbedded	462
16.4.7	Building Images	463
16.5	Summary.....	464
16.5.1	Suggestions for Additional Reading.....	464

Chapter 17 Linux and Real Time.....	465
17.1 What Is Real Time?	466
17.1.1 Soft Real Time	466
17.1.2 Hard Real Time	467
17.1.3 Linux Scheduling.....	467
17.1.4 Latency	467
17.2 Kernel Preemption	469
17.2.1 Impediments to Preemption.....	469
17.2.2 Preemption Models.....	471
17.2.3 SMP Kernel	472
17.2.4 Sources of Preemption Latency	473
17.3 Real-Time Kernel Patch.....	473
17.3.1 Real-Time Features	475
17.3.2 O(1) Scheduler	476
17.3.3 Creating a Real-Time Process.....	477
17.4 Real-Time Kernel Performance Analysis	478
17.4.1 Using Ftrace for Tracing.....	478
17.4.2 Preemption Off Latency Measurement.....	479
17.4.3 Wakeup Latency Measurement	481
17.4.4 Interrupt Off Timing	483
17.4.5 Soft Lockup Detection.....	484
17.5 Summary.....	485
17.5.1 Suggestion for Additional Reading.....	485
Chapter 18 Universal Serial Bus	487
18.1 USB Overview	488
18.1.1 USB Physical Topology.....	488
18.1.2 USB Logical Topology	490

18.1.3	USB Revisions	491
18.1.4	USB Connectors.....	492
18.1.5	USB Cable Assemblies.....	494
18.1.6	USB Modes	494
18.2	Configuring USB	495
18.2.1	USB Initialization.....	497
18.3	sysfs and USB Device Naming	500
18.4	Useful USB Tools	502
18.4.1	USB File System	502
18.4.2	Using <code>usbview</code>	504
18.4.3	USB Utils (<code>lsusb</code>).....	507
18.5	Common USB Subsystems.....	508
18.5.1	USB Mass Storage Class.....	508
18.5.2	USB HID Class	511
18.5.3	USB CDC Class Drivers.....	512
18.5.4	USB Network Support.....	515
18.6	USB Debug.....	516
18.6.1	<code>usbmon</code>	517
18.6.2	Useful USB Miscellanea.....	518
18.7	Summary.....	519
18.7.1	Suggestions for Additional Reading.....	519
Chapter 19	udev	521
19.1	What Is udev?.....	522
19.2	Device Discovery.....	523
19.3	Default udev Behavior.....	525

19.4 Understanding udev Rules.....	527
19.4.1 Modalias	530
19.4.2 Typical udev Rules Configuration	533
19.4.3 Initial System Setup for udev	535
19.5 Loading Platform Device Drivers	538
19.6 Customizing udev Behavior.....	540
19.6.1 udev Customization Example: USB Automounting	540
19.7 Persistent Device Naming.....	541
19.7.1 udev Helper Utilities.....	542
19.8 Using udev with busybox	545
19.8.1 busybox mdev.....	545
19.8.2 Configuring mdev.....	547
19.9 Summary.....	548
19.9.1 Suggestions for Additional Reading.....	548
Appendix A GNU Public License.....	549
Preamble	550
Terms and Conditions for Copying, Distribution, and Modification.....	551
No Warranty	555
Appendix B U-Boot Configurable Commands	557
Appendix C BusyBox Commands.....	561
Appendix D SDRAM Interface Considerations	571
D.1 SDRAM Basics	572
D.1.1 SDRAM Refresh.....	573
D.2 Clocking	574

D.3 SDRAM Setup.....	575
D.4 Summary	580
D.4.1 Suggestions for Additional Reading.....	580
Appendix E Open Source Resources	581
Source Repositories and Developer Information.....	582
Mailing Lists	582
Linux News and Developments.....	583
Open Source Legal Insight and Discussion	583
Appendix F Sample BDI-2000 Configuration File	585
Index.....	593

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, multi-process 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 `malloc`. 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 multi-threaded 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.

Chapter 7

Bootloaders

In This Chapter

■ 7.1	Role of a Bootloader	160
■ 7.2	Bootloader Challenges	161
■ 7.3	A Universal Bootloader: Das U-Boot	166
■ 7.4	Porting U-Boot	174
■ 7.5	Device Tree Blob (Flat Device Tree)	187
■ 7.6	Other Bootloaders	194
■ 7.7	Summary	197

Previous chapters have referred to and even provided examples of bootloader operations. A critical component of an embedded system, the bootloader provides the foundation from which the primary system software is spawned. This chapter starts by examining the bootloader's role in a system. We follow this with an introduction to some common features of bootloaders. Armed with this background, we take a detailed look at a popular bootloader used for embedded systems. We conclude this chapter by introducing a few of the more popular bootloaders.

Numerous bootloaders are in use today. It would be impractical to go into much detail on even the most popular ones. Therefore, we have chosen to explain concepts and use examples based on one of the more popular bootloaders in the open source community for Power Architecture, MIPS, ARM, and other architectures: the U-Boot bootloader.

7.1 Role of a Bootloader

When power is first applied to a processor board, many elements of hardware must be initialized before even the simplest program can run. Each architecture and processor has a set of predefined actions and configurations upon release of reset, which includes fetching initialization code from an onboard storage device (usually Flash memory). This early initialization code is part of the bootloader and is responsible for breathing life into the processor and related hardware components.

Most processors have a default address from which the first bytes of code are fetched upon application of power and release of reset. Hardware designers use this information to arrange the layout of Flash memory on the board and to select which address range(s) the Flash memory responds to. This way, when power is first applied, code is fetched from a well-known and predictable address, and software control can be established.

The bootloader provides this early initialization code and is responsible for initializing the board so that other programs can run. This early initialization code is almost always written in the processor's native assembly language. This fact alone presents many challenges, some of which we examine here.

Of course, after the bootloader has performed this basic processor and platform initialization, its primary role is fetching and booting a full-blown operating system. It is responsible for locating, loading, and passing control to the primary operating system. In addition, the bootloader might have advanced features, such as the capability to validate an OS image, upgrade itself or an OS image, or choose from among several OS images based on a developer-defined policy. Unlike the traditional PC-BIOS model, when the OS takes control, the bootloader is overwritten and ceases to exist.¹

7.2 Bootloader Challenges

Even a simple “Hello World” program written in C requires significant hardware and software resources. The application developer does not need to know or care much about these details. This is because the C runtime environment transparently provides this infrastructure. A bootloader developer enjoys no such luxury. Every resource that a bootloader requires must be carefully initialized and allocated before it is used. One of the most visible examples of this is Dynamic Random Access Memory (DRAM).

7.2.1 DRAM Controller

DRAM chips cannot be directly read from or written to like other microprocessor bus resources. They require specialized hardware controllers to enable read and write cycles. To further complicate matters, DRAM must be constantly refreshed, or the data contained within will be lost. Refresh is accomplished by sequentially reading each location in DRAM in a systematic manner within the timing specifications set forth by the DRAM manufacturer. Modern DRAM chips support many modes of operation, such as burst mode and dual data rate for high-performance applications. It is the DRAM controller’s responsibility to configure DRAM, keep it refreshed within the manufacturer’s timing specifications, and respond to the various read and write commands from the processor.

Setting up a DRAM controller is the source of much frustration for the newcomer to embedded development. It requires detailed knowledge of DRAM architecture, the controller itself, the specific DRAM chips being used, and the overall hardware design. This topic is beyond the scope of this book, but you can learn more about this important concept by consulting the references at the end of this chapter. Appendix D,

¹ Some embedded designs protect the bootloader and provide callbacks to bootloader routines, but this is almost never a good design approach. Linux is far more capable than bootloaders, so there is often little point in doing so.

“SDRAM Interface Considerations,” provides more background on this important topic.

Very little can happen in an embedded system until the DRAM controller and DRAM itself have been properly initialized. One of the first things a bootloader must do is enable the memory subsystem. After it is initialized, memory can be used as a resource. In fact, one of the first actions many bootloaders perform after memory initialization is to copy themselves into DRAM for faster execution.

7.2.2 Flash Versus RAM

Another complexity inherent in bootloaders is that they are required to be stored in nonvolatile storage but usually are loaded into RAM for execution. Again, the complexity arises from the level of resources available for the bootloader to rely on. In a fully operational computer system running an operating system such as Linux, it is relatively easy to compile a program and invoke it from nonvolatile storage. The runtime libraries, operating system, and compiler work together to create the infrastructure necessary to load a program from nonvolatile storage into memory and pass control to it. The aforementioned “Hello World” program is a perfect example. When compiled, it can be loaded into memory and executed simply by typing the name of the executable (`hello`) on the command line (assuming, of course, that the executable exists somewhere on your PATH).

This infrastructure does not exist when a bootloader gains control upon power-on. Instead, the bootloader must create its own operational context and move itself, if required, to a suitable location in RAM. Furthermore, additional complexity is introduced by the requirement to execute from a read-only medium.

7.2.3 Image Complexity

As application developers, we do not need to concern ourselves with the layout of a binary executable file when we develop applications for our favorite platform. The compiler and binary utilities are preconfigured to build a binary executable image containing the proper components needed for a given architecture. The linker places startup (prologue) and shutdown (epilogue) code into the image. These objects set up the proper execution context for your application, which typically starts at `main()`.

This is absolutely not the case with a typical bootloader. When the bootloader gets control, there is no context or prior execution environment. A typical system might

not have any DRAM until the bootloader initializes the processor and related hardware. Consider what this means. In a typical C function, any local variables are stored on the stack, so a simple function like the one shown in Listing 7-1 is unusable.

LISTING 7-1 Simple C Function with a Local Variable

```
int setup_memory_controller(board_info_t *p)
{
    unsigned int *dram_controller_register = p->dc_reg;
...
```

When a bootloader gains control on power-on, there is no stack and no stack pointer. Therefore, a simple C function similar to Listing 7-1 will likely crash the processor, because the compiler will generate code to create and initialize the pointer `dram_controller_register` on the stack, which does not yet exist. The bootloader must create this execution context before any C functions are called.

When the bootloader is compiled and linked, the developer must exercise complete control over how the image is constructed and linked. This is especially true if the bootloader is to relocate itself from Flash to RAM. The compiler and linker must be passed a handful of parameters defining the characteristics and layout of the final executable image. Two primary characteristics conspire to add complexity to the final binary executable image: code organization compatible with the processor's boot requirements, and the execution context, described shortly.

The first characteristic that presents complexity is the need to organize the startup code in a format compatible with the processor's boot sequence. The first executable instructions must be at a predefined location in Flash, depending on the processor and hardware architecture. For example, the AMCC Power Architecture 405GP processor seeks its first machine instructions from a hard-coded address of `0xFFFF_FFFC`. Other processors use similar methods with different details. Some processors can be configured at power-on to seek code from one of several predefined locations, depending on hardware configuration signals.

How does a developer specify the layout of a binary image? The linker is passed a linker description file, also called a linker command script. This special file can be thought of as a recipe for constructing a binary executable image. Listing 7-2 is a snippet from an existing linker description file in use in the U-Boot bootloader, which we'll discuss shortly.

LISTING 7-2 Linker Command Script: Reset Vector Placement

```
SECTIONS
{
    .resetvec 0xFFFFFFF : 
    {
        *(.resetvec)
    } = 0xffff
    ...
}
```

A complete description of linker command scripts syntax is beyond the scope of this book. Consult the GNU LD manual referenced at the end of this chapter. Looking at Listing 7-2, we see the beginning of the definition for the output section of the binary ELF image. It directs the linker to place the section of code called `.resetvec` at a fixed address in the output image, starting at location `0xFFFF_FFFC`. Furthermore, it specifies that the rest of this section shall be filled with all 1s (`0xffff`.) This is because an erased Flash memory array contains all 1s. This technique not only saves wear and tear on the Flash memory, but it also significantly speeds up programming of that sector.

Listing 7-3 is the complete assembly language file from a recent U-Boot distribution that defines the `.resetvec` code section. It is contained in an assembly language file called `.../cpu/ppc4xx/resetvec.s`. Notice that this code section cannot exceed 4 bytes in length in a machine with only 32 address bits. This is because only a single instruction is defined in this section, no matter what configuration options are present.

LISTING 7-3 Source Definition of `.resetvec`

```
/* Copyright MontaVista Software Incorporated, 2000 */
#include <config.h>
.section .resetvec, "ax"
#if defined(CONFIG_440)
    b _start_440
#else
#if defined(CONFIG_BOOT_PCI) && defined(CONFIG_MIP405)
    b _start_pci
#else
    b _start
#endif
#endif
```

This assembly language file is easy to understand, even if you have no assembly language programming experience. Depending on the particular configuration (as specified

by the `CONFIG_*` macros), an unconditional branch instruction (`b` in Power Architecture assembler syntax) is generated to the appropriate start location in the main body of code. This branch location is a 4-byte Power Architecture instruction. As we saw in the snippet from the linker command script shown in Listing 7-2, this simple branch instruction is placed in the absolute Flash address of `0xFFFF_FFFC` in the output image. As mentioned earlier, the 405GP processor fetches its first instruction from this hard-coded address. This is how the first sequence of code is defined and provided by the developer for this particular architecture and processor combination.

7.2.4 Execution Context

The other primary reason for bootloader image complexity is the lack of execution context. When the sequence of instructions from Listing 7-3 starts executing (recall that these are the first machine instructions after power-on), the resources available to the running program are nearly zero. Default values designed into the hardware ensure that fetches from Flash memory work properly. This also ensures that the system clock has some default values, but little else can be assumed.² The reset state of each processor is usually well defined by the manufacturer, but the reset state of a board is defined by the hardware designers.

Indeed, most processors have no DRAM available at startup for temporary storage of variables or, worse, for a stack that is required to use C program calling conventions. If you were forced to write a “Hello World” program with no DRAM and, therefore, no stack, it would be quite different from the traditional “Hello World” example.

This limitation places significant challenges on the initial body of code designed to initialize the hardware. As a result, one of the first tasks the bootloader performs on startup is to configure enough of the hardware to enable at least some minimal amount of RAM. Some processors designed for embedded use have small amounts of on-chip static RAM available. This is the case with the 405GP we’ve been discussing. When RAM is available, a stack can be allocated using part of that RAM, and a proper context can be constructed to run higher-level languages such as C. This allows the rest of the processor and platform initialization to be written in something other than assembly language.

² The details differ, depending on architecture, processor, and details of the hardware design.

7.3 A Universal Bootloader: Das U-Boot

Many open source and commercial bootloaders are available, and many more one-of-a-kind homegrown designs are in widespread use today. Most of these have some level of commonality of features. For example, all of them have some capability to load and execute other programs, particularly an operating system. Most interact with the user through a serial port. Support for various networking subsystems (such as Ethernet) is a very powerful but less common feature.

Many bootloaders are specific to a particular architecture. The capability of a bootloader to support a wide variety of architectures and processors can be an important feature to larger development organizations. It is not uncommon for a single development organization to have multiple processors spanning more than one architecture. Investing in a single bootloader across multiple platforms ultimately results in lower development costs.

This section studies an existing bootloader that has become very popular in the embedded Linux community. The official name of this bootloader is Das U-Boot. It is maintained by Wolfgang Denx and hosted at www.denx.de/wiki/U-Boot. U-Boot supports multiple architectures and has a large following of embedded developers and hardware manufacturers who have adopted it for use in their projects and who have contributed to its development.

7.3.1 Obtaining U-Boot

The simplest way to get the U-Boot source code is via `git`. If you have `git` installed on your desktop or laptop, simply issue this command:

```
$ git clone git://git.denx.de/u-boot.git
```

This creates a directory called `u-boot` in the directory in which you executed this command.

If you don't have `git`, or you prefer to download a snapshot instead, you can do so through the `git` server at `denx.de`. Point your browser to `http://git.denx.de/` and click the "summary" link on the first project, `u-boot.git`. This takes you to a summary screen and provides a "snapshot" link, which generates and downloads a tarball that you can install on your system. Select the most recent snapshot, which is at the top of the "shortlog" list.

7.3.2 Configuring U-Boot

For a bootloader to be useful across many processors and architectures, some method of configuring the bootloader is necessary. As with the Linux kernel itself, a bootloader is configured at compile time. This method significantly reduces the complexity of the binary bootloader image, which in itself is an important characteristic.

In the case of U-Boot, board-specific configuration is driven by a single header file specific to the target platform, together with a few soft links in the source tree that select the correct subdirectories based on target board, architecture, and CPU. When configuring U-Boot for one of its supported platforms, issue this command:

```
$ make <platform>_config
```

Here, *platform* is one of the many platforms supported by U-Boot. These platform configuration targets are listed in the top-level U-Boot makefile. For example, to configure for the Spectrum Digital OSK, which contains a TI OMAP 5912 processor, issue this command:

```
$ make omap5912osk_config
```

This configures the U-Boot source tree with the appropriate soft links to select ARM as the target architecture, the ARM926 core, and the 5912 OSK as the target platform.

The next step in configuring U-Boot for this platform is to edit the configuration file specific to this board. This file is found in the U-Boot `../include/configs` subdirectory and is called `omap5912osk.h`. The README file that comes with the U-Boot source code describes the details of configuration and is the best source of this information. (For existing boards that are already supported by U-Boot, it may not be necessary to edit this board-specific configuration file. The defaults may be sufficient for your needs. Sometimes minor edits are needed to update memory size or flash size, because many reference boards can be purchased with varying configurations.)

U-Boot is configured using configuration variables defined in a board-specific header file. Configuration variables have two forms. Configuration options are selected using macros in the form of `CONFIG_XXXX`. Configuration settings are selected using macros in the form of `CONFIG_SYS_XXXX`. In general, configuration *options* (`CONFIG_XXX`) are user-configurable and enable specific U-Boot operational features. Configuration *settings* (`CONFIG_SYS_XXX`) usually are hardware-specific and require detailed knowledge of the underlying processor and/or hardware platform. Board-specific U-Boot configuration is driven by a header file dedicated to that specific platform that contains

configuration options and settings appropriate for the underlying platform. The U-Boot source tree includes a directory where these board-specific configuration header files reside. They can be found in `.../include/configs` from the top-level U-Boot source directory.

You can select numerous features and modes of operation by adding definitions to the board-configuration file. Listing 7-4 is a partial configuration header file for the Yosemite board based on the AMCC 440EP processor.

LISTING 7-4 Portions of the U-Boot Board-Configuration Header File

```
/*-----
 * High Level Configuration Options
-----*/
/* This config file is used for Yosemite (440EP) and Yellowstone (440GR)*/
#ifndef CONFIG_YELLOWSTONE
#define CONFIG_440EP      1 /* Specific PPC440EP support      */
#define CONFIG_HOSTNAME  yosemite
#else
#define CONFIG_440GR      1 /* Specific PPC440GR support      */
#define CONFIG_HOSTNAME  yellowstone
#endif
#define CONFIG_440      1 /* ... PPC440 family          */
#define CONFIG_4xx      1 /* ... PPC4xx family          */
#define CONFIG_SYS_CLK_FREQ 66666666 /* external freq to pll */
<...>
/*-----
 * Base addresses -- Note these are effective addresses where the
 * actual resources get mapped (not physical addresses)
-----*/
#define CONFIG_SYS_FLASH_BASE    0xfc000000 /* start of FLASH      */
#define CONFIG_SYS_PCI_MEMBASE   0xa0000000 /* mapped pci memory*/
#define CONFIG_SYS_PCI_MEMBASE1  CONFIG_SYS_PCI_MEMBASE + 0x10000000
#define CONFIG_SYS_PCI_MEMBASE2  CONFIG_SYS_PCI_MEMBASE1 + 0x10000000
#define CONFIG_SYS_PCI_MEMBASE3  CONFIG_SYS_PCI_MEMBASE2 + 0x10000000
<...>
#endif CONFIG_440EP
#define CONFIG_CMD_USB
#define CONFIG_CMD_FAT
#define CONFIG_CMD_EXT2
#endif
<...>
/*-----
```

LISTING 7-4 Continued

```
* External Bus Controller (EBC) Setup
*-----*/
#define CONFIG_SYS_FLASH      CONFIG_SYS_FLASH_BASE
#define CONFIG_SYS_CPLD      0x80000000

/* Memory Bank 0 (NOR-FLASH) initialization          */
#define CONFIG_SYS_EBC_PB0AP    0x03017300
#define CONFIG_SYS_EBC_PB0CR    (CONFIG_SYS_FLASH | 0xda000)

/* Memory Bank 2 (CPLD) initialization          */
#define CONFIG_SYS_EBC_PB2AP    0x04814500
#define CONFIG_SYS_EBC_PB2CR    (CONFIG_SYS_CPLD | 0x18000)
<...>
```

Listing 7-4 gives you an idea of how U-Boot itself is configured for a given board. An actual board-configuration file can contain hundreds of lines similar to those found here. In this example, you can see the definitions for the CPU (`CONFIG_440EP`), board name (`CONFIG_HOSTNAME`), clock frequency, and Flash and PCI base memory addresses. We have included examples of configuration variables (`CONFIG_XXX`) and configuration settings (`CONFIG_SYS_XXX`). The last few lines are actual processor register values required to initialize the external bus controller for memory banks 0 and 1. You can see that these values can come only from detailed knowledge of the board and processor.

Many aspects of U-Boot can be configured using these mechanisms, including what functionality will be compiled into U-Boot (support for DHCP, memory tests, debugging support, and so on). This mechanism can be used to tell U-Boot how much and what kind of memory is on a given board, and where that memory is mapped. You can learn much more by looking at the U-Boot code directly, especially the excellent README file.

7.3.3 U-Boot Monitor Commands

U-Boot supports more than 70 standard command sets that enable more than 150 unique commands using `CONFIG_CMD_*` macros. A command set is enabled in U-Boot through the use of configuration setting (`CONFIG_*`) macros. For a complete list from a recent U-Boot snapshot, consult Appendix B, “U-Boot Configurable Commands.” Table 7-1 shows just a few, to give you an idea of the capabilities available.

TABLE 7-1 Some U-Boot Configurable Commands

Command Set	Description Commands
CONFIG_CMD_FLASH	Flash memory commands
CONFIG_CMD_MEMORY	Memory dump, fill, copy, compare, and so on
CONFIG_CMD_DHCP	DHCP support
CONFIG_CMD_PING	Ping support
CONFIG_CMD_EXT2	EXT2 file system support

To enable a specific command, define the macro corresponding to the command you want. These macros are defined in your board-specific configuration file. Listing 7-4 shows several commands being enabled in the board-specific configuration file. There you see CONFIG_CMD_USB, CONFIG_CMD_FAT, and CONFIG_CMD_EXT2 being defined conditionally if the board is a 440EP.

Instead of specifying each individual CONFIG_CMD_* macro in your own board-specific configuration header, you can start from the full set of commands defined in .../include/config_cmd_all.h. This header file defines every command available. A second header file, .../include/config_cmd_default.h, defines a list of useful default U-Boot command sets such as tftpboot (boot an image from a tftpsserver), bootm (boot an image from memory), memory utilities such as md (display memory), and so on. To enable your specific combination of commands, you can start with the default and add and subtract as necessary. Listing 7-4 adds the USB, FAT, and EXT2 command sets to the default. You can subtract in a similar fashion, starting from config_cmd_all.h:

```
#include "config_cmd_all.h"  
#undef CONFIG_CMD_DHCP  
#undef CONFIG_CMD_FAT  
#undef CONFIG_CMD_FDOS  
<...>
```

Take a look at any board-configuration header file in .../include/configs/ for examples.

7.3.4 Network Operations

Many bootloaders include support for Ethernet interfaces. In a development environment, this is a huge time saver. Loading even a modest kernel image over a serial port

can take minutes versus a few seconds over an Ethernet link, especially if your board supports Fast or Gigabit Ethernet. Furthermore, serial links are more prone to errors from poorly behaved serial terminals, line noise, and so on.

Some of the more important features to look for in a bootloader include support for the BOOTP, DHCP, and TFTP protocols. If you're unfamiliar with these, BOOTP (Bootstrap Protocol) and DHCP (Dynamic Host Configuration Protocol) enable a target device with an Ethernet port to obtain an IP address and other network-related configuration information from a central server. TFTP (Trivial File Transfer Protocol) allows the target device to download files (such as a Linux kernel image) from a TFTP server. References to these protocol specifications are listed at the end of this chapter. Servers for these services are described in Chapter 12, "Embedded Development Environment."

Figure 7-1 illustrates the flow of information between the target device and a BOOTP server. The client (U-Boot, in this case) initiates the exchange by sending a broadcast packet searching for a BOOTP server. The server responds with a reply packet that includes the client's IP address and other information. The most useful data includes a filename used to download a kernel image.

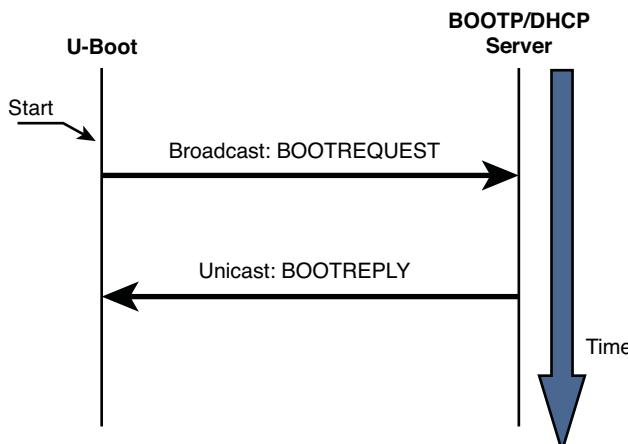


FIGURE 7-1 BOOTP client/server handshake

In practice, dedicated BOOTP servers no longer exist as stand-alone servers. DHCP servers included with your favorite Linux distribution also support BOOTP protocol packets and are almost universally used for BOOTP operations.

The DHCP protocol builds on BOOTP. It can supply the target with a wide variety of configuration information. In practice, the information exchange is often limited by the target/bootloader DHCP client implementation. Listing 7-5 shows a DHCP server configuration block identifying a single target device. This is a snippet from a DHCP configuration file from the Fedora Core 2 DHCP implementation.

LISTING 7-5 DHCP Target Specification

```
host coyote {  
    hardware ethernet 00:0e:0c:00:82:f8;  
    netmask 255.255.255.0;  
    fixed-address 192.168.1.21;  
    server-name 192.168.1.9;  
    filename "coyote-zImage";  
    option root-path "/home/sandbox/targets/coyote-target";  
}  
...
```

When this DHCP server receives a packet from a device matching the hardware Ethernet address contained in Listing 7-5, it responds by sending that device the parameters in this target specification. Table 7-2 describes the fields in the target specification.

TABLE 7-2 DHCP Target Parameters

DHCP Target Parameter	Purpose	Description
host	Hostname	Symbolic label from the DHCP configuration file
hardware ethernet	Ethernet hardware address	Low-level Ethernet hardware address of the target's Ethernet interface
fixed-address	Target IP address	The IP address that the target will assume
netmask	Target netmask	The IP netmask that the target will assume
server-name	TFTP server IP address	The IP address to which the target will direct requests for file transfers, the root file system, and so on
filename	TFTP filename	The filename that the bootloader can use to boot a secondary image (usually a Linux kernel)
root-path	NFS root path	Defines the network path for the remote NFS root mount

When the bootloader on the target board has completed the BOOTP or DHCP exchange, these parameters are used for further configuration. For example, the bootloader uses the target IP address (`fixed-address`) to bind its Ethernet port to this IP address. The bootloader then uses the `server-name` field as a destination IP address to request the file contained in the `filename` field, which, in most cases, represents a Linux kernel image. Although this is the most common use, this same scenario could be used to download and execute manufacturing test and diagnostics firmware.

It should be noted that the DHCP protocol supports many more parameters than those detailed in Table 7-2. These are simply the more common parameters you might encounter for embedded systems. See the DHCP specification referenced at the end of this chapter for complete details.

7.3.5 Storage Subsystems

Many bootloaders support the capability of booting images from a variety of nonvolatile storage devices in addition to the usual Flash memory. The difficulty in supporting these types of devices is the relative complexity in both hardware and software. To access data on a hard drive, for example, the bootloader must have device driver code for the IDE controller interface, as well as knowledge of the underlying partition scheme and file system. This is not trivial and is one of the tasks more suited to full-blown operating systems.

Even with the underlying complexity, methods exist for loading images from this class of device. The simplest method is to support the hardware only. In this scheme, no knowledge of the file system is assumed. The bootloader simply raw-loads from absolute sectors on the device. This scheme can be used by dedicating an unformatted partition from sector 0 on an IDE-compatible device (such as CompactFlash) and loading the data found there without any structure imposed on the data. This is a simple configuration for loading a kernel image or other binary image from a block storage device. Additional partitions on the device can be formatted for a given file system and can contain complete file systems. After the kernel boots, Linux device drivers can be used to access the additional partitions.

U-Boot can load an image from a specified raw partition or from a partition with a file system structure. Of course, the board must have a supported hardware device (an IDE subsystem), and U-Boot must be so configured. Adding `CONFIG_CMD_IDE` to the board-specific configuration file enables support for an IDE interface, and adding `CONFIG_CMD_BOOTD` enables support for booting from a raw partition. If you are porting

U-Boot to a custom board, you will likely have to modify U-Boot to understand your particular hardware.

7.3.6 Booting from Disk

As just described, U-Boot supports several methods for booting a kernel image from a disk subsystem. This simple command illustrates one of the supported methods:

```
=> diskboot 0x400000 0:0
```

To understand this syntax, you must first understand how U-Boot numbers disk devices. The 0:0 in this example specifies the device and partition. In this simple example, U-Boot performs a raw binary load of the image found on the first IDE device (IDE device 0) from the first partition (partition 0) found on this device. The image is loaded into system memory at physical address 0x400000.

After the kernel image has been loaded into memory, the U-Boot `bootm` command (boot from memory) is used to boot the kernel:

```
=> bootm 0x400000
```

7.4 Porting U-Boot

One of the reasons U-Boot has become so popular is the ease with which new platforms can be supported. Each board port must supply a subordinate makefile that supplies board-specific definitions to the build process. These files are all given the name `config.mk`. They exist in the `.../board/vendor/boardname` subdirectory under the U-Boot top-level source directory, where `boardname` specifies a particular board.

As of a recent U-Boot snapshot, more than 460 different board configuration files are named `config.mk` under the `.../boards` subdirectory. In this same U-Boot version, 49 different CPU configurations are supported (counted in the same manner). Note that, in some cases, the CPU configuration covers a family of chips, such as `ppc4xx`, that supports several processors in the Power Architecture `4xx` family. U-Boot supports a large variety of popular CPUs and CPU families in use today, and a much larger collection of reference boards based on these processors.

If your board contains one of the supported CPUs, porting U-Boot is straightforward. If you must add a new CPU, plan on substantially more effort. The good news is that someone before you has probably done the bulk of the work. Whether you are

porting to a new CPU or a new board based on an existing CPU, study the existing source code for specific guidance. Determine what CPU is closest to yours, and clone the functionality found in that CPU-specific directory. Finally, modify the resulting sources to add the specific support for your new CPU's requirements.

7.4.1 EP405 U-Boot Port

The same logic used in porting to a different CPU applies to porting U-Boot to a new board. Let's look at an example. We will use the Embedded Planet EP405 board, which contains the AMCC Power Architecture 405GP processor. The particular board used for this example was provided courtesy of Embedded Planet and came with 64MB of SDRAM and 16MB of on-board Flash. Numerous other devices complete the design.

The first step is to see how close we can come to an existing board. Many boards in the U-Boot source tree support the 405GP processor. A quick grep of the board-configuration header files narrows the choices to those that support the 405GP processor:

```
$ cd .../u-boot/include/configs  
$ grep -l CONFIG_405GP *
```

In a recent U-Boot snapshot, 28 board configuration files are configured for the 405GP. After examining a few, we choose the AR405.h configuration as a baseline. It supports the LXT971 Ethernet transceiver, which is also on the EP405. The goal is to minimize any development work by borrowing from similar architectures in the spirit of open source.

We'll tackle the easy steps first. We need a custom board configuration header file for our EP405 board. Copy the board configuration file to a new file with a name appropriate for your board. We'll call ours `EP405.h`. These commands are issued from the top-level U-Boot source tree:

```
$ cp .../include/configs/AR405.h .../include/configs/EP405.h
```

After you have copied the configuration header file, you must create the board-specific directory and make a copy of the AR405 board files. We don't know yet if we need all of them. That step will come later. After copying the files to your new board directory, edit the filenames appropriately for your board name:

```
$ cd board    <<< from top-level U-Boot source directory  
$ mkdir ep405  
$ cp esd/ar405/* ep405
```

Now comes the hard part. Jerry Van Baren, a developer and U-Boot contributor, detailed a humorous but realistic process for porting U-Boot in an e-mail posting to the U-Boot mailing list. His complete process, documented in pseudo-C, can be found in the U-Boot README file. The following summarizes the hard part of the porting process in Jerry's style and spirit:

```
while (!running) {
    do {
        Add / modify source code
    } until (compiles);
    Debug;
...
}
```

Jerry's process, as summarized here, is the simple truth. When you have selected a baseline from which to port, you must add, delete, and modify source code until it compiles, and then debug it until it is running without error! There is no magic formula. Porting any bootloader to a new board requires knowledge of many areas of hardware and software. Some of these disciplines, such as setting up SDRAM controllers, are rather specialized and complex. Virtually all of this work involves detailed knowledge of the underlying hardware. Therefore, be prepared to spend many entertaining hours poring over your processor's hardware reference manual, along with the data sheets of numerous other components that reside on your board.

7.4.2 U-Boot Makefile Configuration Target

Now that we have a code base to start from, we must make some modifications to the top-level U-Boot makefile to add the configuration steps for our new board. Upon examining this makefile, we find a section for configuring the U-Boot source tree for the various supported boards. This section can be found starting with the `unconfig` target in the top-level makefile. We now add support for our new board to allow us to configure it. Because we derived our board from the ESD AR405, we will use that rule as the template for building our own. If you follow along in the U-Boot source code, you will see that these rules are placed in the makefile in alphabetical order according to their configuration names. We will be good open-source citizens and follow that lead. We call our configuration target `EP405_config`, again in concert with the U-Boot conventions. Listing 7-6 details the edits you will need to make in your top-level makefile.

LISTING 7-6 Makefile Edits

```
ebony_config:      unconfig
    @$(MKCONFIG) $(@:_config=) ppc ppc4xx ebony amcc

+EP405_config:      unconfig
+    @$(MKCONFIG) $(@:_config=) ppc ppc4xx ep405 ep
+
ERIC_config:      unconfig
    ./mkconfig $(@:_config=) ppc ppc4xx eric
```

Our new configuration rule has been inserted as shown in the three lines preceded by the + character (unified diff format). Edit the top-level makefile using your favorite editor.

Upon completing the steps just described, we have a U-Boot source tree that represents a starting point. It probably will not compile cleanly, so that should be our first step. At least the compiler can give us some guidance on where to start.

7.4.3 EP405 First Build

We now have a U-Boot source tree with our candidate files. Our first step is to configure the build tree for our newly installed EP405 board. Using the configuration target we just added to the top-level makefile, we configure the tree. Listing 7-7 gives you a starting point for where you need to focus your efforts.

LISTING 7-7 Configure and Build for EP405

```
$ make ARCH=ppc CROSS_COMPILE=ppc_405- EP405_config
Configuring for EP405 board...
$ # Now do the build
$ make ARCH=ppc CROSS_COMPILE=ppc_405-
<...lots of build steps...>
make[1]: Entering directory '/home/chris/sandbox/u-boot/board/ep/ep405'
ppc_440ep-gcc -g -Os -mrelocatable -fPIC -ffixed-r14 -meabi -D__KERNEL__
-DTEXT_BASE=0xFFFFC0000 -I/home/chris/sandbox/u-boot/include -fno-builtin -ffree-standing
-nostdinc -isystem /opt/pro5/montavista/pro/devkit/ppc/440ep/bin/../lib/
gcc/powerpc-montavista-linux-gnu/4.2.0/include -pipe -DCONFIG_PPC -D__powerpc__
-DCONFIG_4xx -ffixed-r2 -mstring -msoft-float -Wa,-m405 -mcpu=405 -Wall -Wstrict-prototypes
-fno-stack-protector -o ep405.o ep405.c -c
ep405.c:25:19: error: ar405.h: No such file or directory
ep405.c:44:22: error: fpgadata.c: No such file or directory
ep405.c:48:27: error: fpgadata_x130.c: No such file or directory
ep405.c:54:28: error: ../common/fpga.c: No such file or directory
ep405.c: In function 'board_early_init_f':
```

LISTING 7-7 Continued

```
ep405.c:75: warning: implicit declaration of function 'fpga_boot'  
ep405.c:91: error: 'ERROR_FPGA_PRG_INIT_LOW' undeclared (first use in this func-  
tion)  
ep405.c:91: error: (Each undeclared identifier is reported only once  
ep405.c:91: error: for each function it appears in.)  
ep405.c:94: error: 'ERROR_FPGA_PRG_INIT_HIGH' undeclared (first use in this func-  
tion)  
ep405.c:97: error: 'ERROR_FPGA_PRG_DONE' undeclared (first use in this function)  
make[1]: *** [ep405.o] Error 1  
make[1]: Leaving directory '/home/chris/sandbox/u-boot/board/ep/ep405'  
make: *** [board/ep/ep405/libep405.a] Error 2
```

At first glance, we notice we need to edit our cloned `ep405.c` file and fix up a few references. These include the board header file and references to the FPGA. We can eliminate these, because the EP405 board doesn't contain an FPGA like the AR405 we derived from. These edits should be straightforward, so we'll leave them as an exercise for the reader. Again, there is no formula better than Jerry's: edit-compile-repeat until the file compiles cleanly. Then comes the hard part—actually making it work. It was not difficult. Less than an hour of editing had the file compiling without errors.

7.4.4 EP405 Processor Initialization

The first task that your new U-Boot port must do correctly is initialize the processor and the memory (DRAM) subsystems. After reset, the 405GP processor core is designed to fetch instructions starting from `0xFFFF_FFFC`. The core attempts to execute the instructions found here. Because this is the top of the memory range, the instruction found here must be an unconditional branch instruction.

This processor core is also hard-coded to configure the upper 2MB memory region so that it is accessible without programming the external bus controller, to which Flash memory is usually attached. This forces the requirement to branch to a location within this address space, because the processor is incapable of addressing memory anywhere else until our bootloader code initializes additional memory regions. We must branch to somewhere at or above `0xFFE0_0000`. How do we know all this? Because we read the 405GP user manual!

The behavior of the 405GP processor core, as just described, places requirements on the hardware designer to ensure that, on power-up, nonvolatile memory (Flash) is mapped to the required upper 2MB memory region. Certain attributes of this initial

memory region assume default values on reset. For example, this upper 2MB region will be configured for 256 wait states, three cycles of address to chip select delay, three cycles of chip select to output enable delay, and seven cycles of hold time.³ This allows maximum freedom for the hardware designer to select appropriate devices or methods of getting instruction code to the processor directly after reset.

We've already seen how the reset vector is installed to the top of Flash in Listing 7-2. When configured for the 405GP, our first lines of code will be found in the file `.../cpu/ppc4xx/start.s`. The U-Boot developers intended this code to be processor-generic. In theory, there should be no need for board-specific code in this file. You will see how this is accomplished.

You don't need to understand Power Architecture assembly language in any depth to understand the logical flow in `start.s`. Many frequently asked questions (FAQs) have been posted to the U-Boot mailing list about modifying low-level assembly code. In nearly all cases, it is not necessary to modify this code if you are porting to one of the many supported processors. It is mature code, with many successful ports running on it. You need to modify the board-specific code (at a bare minimum) for your port. If you find yourself troubleshooting or modifying the early startup assembler code for a processor that has been around for a while, you are most likely heading down the wrong road.

Listing 7-8 reproduces a portion of `start.s` for the 4xx architecture.

LISTING 7-8 U-Boot 4xx Startup Code

```
...
#ifndef __ASSEMBLY__
#define __ASSEMBLY__

#if defined(CONFIG_405GP) || defined(CONFIG_405CR) ||
defined(CONFIG_405) || defined(CONFIG_405EP)
/*-----
 * Clear and set up some registers.
 */
addi    r4,r0,0x0000
mtspr  sgr,r4
mtspr  dcwr,r4
mtesr  r4          /* clear Exception Syndrome Reg */
mttcr  r4          /* clear Timer Control Reg */
mtxer  r4          /* clear Fixed-Point Exception Reg */
mtevpr r4          /* clear Exception Vector Prefix Reg */
addi    r4,r0,0x1000 /* set ME bit (Machine Exceptions) */
oris    r4,r4,0x0002 /* set CE bit (Critical Exceptions) */
mtmsr  r4          /* change MSR */
#endif
#endif
```

³This data was taken directly from the 405GP user's manual, referenced at the end of this chapter.

LISTING 7-8 Continued

```
addi    r4,r0,(0xFFFF-0x10000) /* set r4 to 0xFFFFFFFF (status in the */
                                /* dbsr is cleared by setting bits to 1) */
mtdbsr r4                      /* clear/reset the dbsr */

/*-----
 * Invalidate I and D caches. Enable I cache for defined memory regions *
 * to speed things up. Leave the D cache disabled for now. It will be *
 * enabled/left disabled later based on user-selected menu options. *
 * Be aware that the I cache may be disabled later based on the menu *
 * options as well. See miscLib/main.c. */
/*----- */

bl      invalidate_icache
bl      invalidate_dcache

/*-----
 * Enable two 128MB cachable regions. */
/*----- */

addis   r4,r0,0x8000
addi    r4,r4,0x0001
mticcr r4                      /* instruction cache */
isync

addis   r4,r0,0x0000
addi    r4,r4,0x0000
mtdccr r4                      /* data cache */
```

The first code to execute in `start.S` for the 405GP processor starts about a third of the way into the source file, where a handful of processor registers are cleared or set to sane initial values. The instruction and data caches are then invalidated, and the instruction cache is enabled to speed up the initial load. Two 128MB cacheable regions are set up—one at the high end of memory (the Flash region), and the other at the bottom (normally the start of system DRAM). U-Boot eventually is copied to RAM in this region and executed from there. The reason for this is performance: raw reads from RAM are an order of magnitude (or more) faster than reads from Flash. However, for the 4xx CPU, there is another subtle reason for enabling the instruction cache, as you shall soon discover.

7.4.5 Board-Specific Initialization

The first opportunity for any board-specific initialization comes in `.../cpu/ppc4xx/start.s` just after the cacheable regions have been initialized. Here we find a call to an external assembler language routine called `ext_bus_cntlr_init`:

```
bl ext_bus_cntlr_init /* Board-specific bus cntrl init */
```

This routine is defined in `.../board/ep405/init.s`, in the new board-specific directory for our board. It provides a hook for very early hardware-based initialization. This is one of the files that has been customized for our EP405 platform. This file contains the board-specific code to initialize the 405GP's external bus controller for our application. Listing 7-9 contains the meat of the functionality from this file. This is the code that initializes the 405GP's external bus controller.

LISTING 7-9 External Bus Controller Initialization

```
.globl ext_bus_cntlr_init
ext_bus_cntlr_init:
    mflr r4          /* save link register      */
    bl ..getAddr
..getAddr:
    mflr r3          /* get _this_ address      */
    mtlr r4          /* restore link register   */
    addi r4,0,14     /* prefetch 14 cache lines... */
    mtctr r4         /* ...to fit this function */
                    /* cache (8x14=112 instr) */
..ebcloop:
    icbt r0,r3       /* prefetch cache line for [r3] */
    addi r3,r3,32    /* move to next cache line */
    bdnz ..ebcloop   /* continue for 14 cache lines */

/*----- */
/* Delay to ensure all accesses to ROM are complete */
/* before changing bank 0 timings */
/* 200usec should be enough. */
/* 200,000,000 (cycles/sec) X .000200 (sec) = */
/* 0x9C40 cycles */
/*----- */

    addis r3,0,0x0
    ori r3,r3,0xA000 /* ensure 200usec have passed t */
```

LISTING 7-9 Continued

```
mtctr    r3

..spinlp:
    bdnz    ..spinlp      /* spin loop                  */

    /*-----*/
    /* Now do the real work of this function          */
    /* Memory Bank 0 (Flash and SRAM) initialization */
    /*-----*/

    addi    r4,0,pb0ap      /* *ebccfga = pb0ap;        */
    mtdcr  ebccfga,r4
    addis    r4,0,EBC0_B0AP@h   /* *ebccfgd = EBC0_B0AP;   */
    ori     r4,r4,EBC0_B0AP@1
    mtdcr  ebccfgd,r4

    addi    r4,0,pb0cr      /* *ebccfga = pb0cr;        */
    mtdcr  ebccfga,r4
    addis    r4,0,EBC0_B0CR@h   /* *ebccfgd = EBC0_B0CR;   */
    ori     r4,r4,EBC0_B0CR@1
    mtdcr  ebccfgd,r4

    /*-----*/
    /* Memory Bank 4 (NVRAM & BCSR) initialization */
    /*-----*/

    addi    r4,0,pb4ap      /* *ebccfga = pb4ap;        */
    mtdcr  ebccfga,r4
    addis    r4,0,EBC0_B4AP@h   /* *ebccfgd = EBC0_B4AP;   */
    ori     r4,r4,EBC0_B4AP@1
    mtdcr  ebccfgd,r4

    addi    r4,0,pb4cr      /* *ebccfga = pb4cr;        */
    mtdcr  ebccfga,r4
    addis    r4,0,EBC0_B4CR@h   /* *ebccfgd = EBC0_B4CR;   */
    ori     r4,r4,EBC0_B4CR@1
    mtdcr  ebccfgd,r4

    blr                 /* return                  */
```

Listing 7-9 was chosen because it is typical of the subtle complexities involved in low-level processor initialization. It is important to realize the context in which this

code is running. It is executing from Flash, before any DRAM is available. There is no stack. This code is preparing to make fundamental changes to the controller that governs access to the very Flash it is executing from. It is well documented for this particular processor that executing code from Flash while modifying the external bus controller to which the Flash is attached can lead to errant reads and a resulting processor crash.

The solution is shown in this assembly language routine. Starting at the label `..getAddr`, and for the next seven assembly language instructions, the code essentially prefetches itself into the instruction cache, using the `icbt` instruction. When the entire subroutine has been successfully read into the instruction cache, it can proceed to make the required changes to the external bus controller without fear of a crash, because it is executing directly from the internal instruction cache. Subtle, but clever! This is followed by a short delay to make sure that all the requested i-cache reads have completed.

When the prefetch and delay have completed, the code proceeds to configure Memory Bank 0 and Memory Bank 4 appropriately for our board. The values come from detailed knowledge of the underlying components and their interconnection on the board. Consult the last section in this chapter for all the details of the Power Architecture assembler and the 405GP processor from which this example was derived.

Consider making a change to this code without a complete understanding of what is happening here. Perhaps you added a few lines and increased its size beyond the range that was prefetched into the cache. It would likely crash (worse, it might crash only sometimes), but stepping through this code with a debugger would not yield a single clue as to why.

The next opportunity for board-specific initialization comes after a temporary stack has been allocated from the processor's data cache. This is the branch to initialize the SDRAM controller around line 727 of `.../cpu/ppc4xx/start.s`:

```
bl sdram_init
```

The execution context now includes a stack pointer and some temporary memory for local data storage—that is, a partial C context, allowing the developer to use C for the relatively complex task of setting up the system SDRAM controller and other initialization tasks. In our EP405 port, the `sdram_init()` code resides in `.../board/ep405/ep405.c` and is customized for this particular board and DRAM configuration. Because this board does not use a commercially available memory SIMM, it is not possible to determine the configuration of the DRAM dynamically, as with so many other boards supported by U-Boot. It is hard-coded in `sdram_init`.

Many off-the-shelf memory DDR modules have an SPD (Serial Presence Detect) PROM containing parameters that identify the memory module and its architecture and organization. These parameters can be read under program control via I2C and can be used as input to determine proper parameters for the memory controller. U-Boot has support for this technique but may need modifications to work with your specific board. Many examples of its use can be found in the U-Boot source code. The configuration option `CONFIG_SPD_EEPROM` enables this feature. You can grep for this option to find examples of its use.

7.4.6 Porting Summary

By now, you can appreciate some of the difficulties of porting a bootloader to a hardware platform. There is simply no substitute for detailed knowledge of the underlying hardware. Of course, we'd like to minimize our investment in time required for this task. After all, we usually are not paid based on how well we understand every hardware detail of a given processor, but rather on our ability to deliver a working solution in a timely manner. Indeed, this is one of the primary reasons open source has flourished. You just saw how easy it is to port U-Boot to a new hardware platform—not because you're an expert on the processor, but because many before us have done the bulk of the hard work already.

Listing 7-10 is the complete list of new or modified files that complete the basic EP405 port for U-Boot. Of course, if there had been new hardware devices for which no support exists in U-Boot, or if we were porting to a new CPU that is not yet supported in U-Boot, this would have been a much more significant effort. The point to be made here, at the risk of sounding redundant, is that there is simply no substitute for detailed knowledge of both the hardware (CPU and subsystems) and the underlying software (U-Boot) to complete a port successfully in a reasonable time frame. If you start the project from that frame of mind, you will have a successful outcome.

LISTING 7-10 New or Changed Files for U-Boot EP405 Port

```
$ git diff HEAD --stat
Makefile |      3 +
board/ep/ep405/Makefile |   53 +++++
board/ep/ep405/config.mk |  30 ++
board/ep/ep405/ep405.c | 329 ++++++++++++++
board/ep/ep405/ep405.h |  44 ===
board/ep/ep405/flash.c | 749 ++++++++++++++++++++++
include/configs/EP405.h | 272 ++++++
7 files changed, 1480 insertions(+), 0 deletions(-)
```

Recall that we derived all the files in the `.../board/ep405` directory from another directory. Indeed, we didn't create any files from scratch for this port. We borrowed from the work of others and customized where necessary to achieve our goals.

7.4.7 U-Boot Image Format

Now that we have a working bootloader for our EP405 board, we can load and run programs on it. Ideally, we want to run an operating system such as Linux. To do this, we need to understand the image format that U-Boot requires. U-Boot expects a small header on the image file that identifies several attributes of the image. U-Boot provides the `mkimage` tool (part of the U-Boot source code) to build this image header.

Recent Linux kernel distributions have built-in support for building images directly bootable by U-Boot. Both the `arm` and `powerpc` branches of the kernel source tree support a target called `uImage`. Let's look at the Power Architecture case.

Browsing through the makefile `.../arch/powerpc/boot/Makefile`, we see the `uImage` target defining a call to an external wrapper script called, you guessed it, `wrapper`. Without delving into the syntactical tedium, the wrapper script sets up some default variable values and eventually calls `mkimage`. Listing 7-11 reproduces this processing from the wrapper script.

LISTING 7-11 `mkimage` from Wrapper Script

```
case "$platform" in
uboot)
    rm -f "$ofile"
    mkimage -A ppc -O linux -T kernel -C gzip -a $membase -e $membase \
    $uboot_version -d "$vmz" "$ofile"
    if [ -z "$cacheit" ]; then
        rm -f "$vmz"
    fi
    exit 0
;;
esac
```

The `mkimage` utility creates the U-Boot header and prepends it to the supplied kernel image. It writes the resulting image to the final parameter passed to `mkimage`—in this case, the value of the `$ofile` variable, which in this example will be called `uImage`. The parameters are as follows:

- `-A` specifies the target image architecture.
- `-O` species the target image OS—in this case, Linux.

- `-T` specifies the target image type—in this case, a kernel.
- `-c` specifies the target image compression type—in this case, `gzip`.
- `-a` sets the U-Boot `loadaddress` to the value specified.
- `-e` sets the U-Boot image entry point to the supplied value.
- `-n` is a text field used to identify the image to the human user (supplied in the `uboot_version` variable).
- `-d` is the executable image file to which the header is prepended.

Several U-Boot commands use this header data both to verify the integrity of the image (U-Boot also puts a CRC signature in the header) and to identify the image type. U-Boot has a command called `iminfo` that reads the image header and displays the image attributes from the target image. Listing 7-12 contains the results of loading a `uImage` (bootable Linux kernel image formatted for U-Boot) to the EP405 board via U-Boot’s `tftp` command and executing the `iminfo` command on the image.⁴

LISTING 7-12 U-Boot iminfo Command

```
=> tftp 400000 uImage-ep405
ENET Speed is 100 Mbps - FULL duplex connection
TFTP from server 192.168.1.9; our IP address is 192.168.1.33
Filename 'uImage-ep405'.
Load address: 0x400000
Loading: ##### done
Bytes transferred = 891228 (d995c hex)
=> iminfo

## Checking Image at 00400000 ...
Image Name: Linux-2.6.11.6
Image Type: PowerPC Linux Kernel Image (gzip compressed)
Data Size: 891164 Bytes = 870.3 kB
Load Address: 00000000
Entry Point: 00000000
Verifying Checksum ... OK
=>
```

⁴ We changed the name of the `uImage` to reflect the target it corresponds to. In this example, we appended `-ep405` to indicate it is a kernel for that target.

7.5 Device Tree Blob (Flat Device Tree)

One of the more challenging aspects of porting Linux (and U-Boot) to your new board is the recent requirement for a device tree blob (DTB). It is also referred to as a flat device tree, device tree binary, or simply device tree. Throughout this discussion, these terms are used interchangeably. The DTB is a database that represents the hardware components on a given board. It is derived from the IBM OpenFirmware specifications and has been chosen as the default mechanism to pass low-level hardware information from the bootloader to the kernel.

Prior to the requirement for a DTB, U-Boot would pass a board information structure to the kernel, which was derived from a header file in U-Boot that had to exactly match the contents of a similar header file in the kernel. It was very difficult to keep them in sync, and it didn't scale well. This was, in part, the motivation for incorporating the flat device tree as a method to communicate low-level hardware details from the bootloader to the kernel.

Similar to U-Boot or other low-level firmware, mastering the DTB requires complete knowledge of the underlying hardware. You can do an Internet search to find some introductory documents that describe the device tree. A great starting point is the Denx Software Engineering wiki page. References are provided at the end of this chapter.

To begin, let's see how the DTB is used during a typical boot sequence. Listing 7-13 shows a boot sequence on a Power Architecture target using U-Boot. The Freescale MPC8548CDS system was used for this example.

LISTING 7-13 Booting Linux with the Device Tree Blob from U-Boot

```
=> tftp $loadaddr 8548/uImage
Speed: 1000, full duplex
Using eTSEC0 device
TFTP from server 192.168.11.103; our IP address is 192.168.11.18
Filename '8548/uImage'.
Load address: 0x600000
Loading: #####
done
Bytes transferred = 1838553 (1c0dd9 hex)
=> tftp $fdtaddr 8548/dtb
Speed: 1000, full duplex
Using eTSEC0 device
TFTP from server 192.168.11.103; our IP address is 192.168.11.18
```

LISTING 7-13 Continued

```
Filename '8548/dtb'.
Load address: 0xc00000
Loading: ##
done
Bytes transferred = 16384 (4000 hex)
=> bootm $loadaddr - $fdtaddr
## 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
    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
<.... Linux begins booting here...>
...and away we go!!
```

The primary difference here is that we loaded two images. The large image (1.8MB) is the kernel image. The smaller image (16KB) is the flat device tree. Notice that we placed the kernel and DTB at addresses 0x600000 and 0xc00000, respectively. All the messages from Listing 7-13 are produced by U-Boot. When we use the `bootm` command to boot the kernel, we add a third parameter, which tells U-Boot where we loaded the DTB.

By now, you are probably wondering where the DTB came from. The easy answer is that it was provided as a courtesy by the board/architecture developers as part of the Linux kernel source tree. If you look at the `powerpc` branch of any recent Linux kernel tree, you will see a directory called `.../arch/powerpc/boot/dts`. This is where the “source code” for the DTB resides.

The hard answer is that you must provide a DTB for your custom board. Start with something close to your platform, and modify from there. At the risk of sounding redundant, there is no easy path. You must dive in and learn the details of your hardware platform and become proficient at writing device nodes and their respective properties. Hopefully, this section will start you on your way toward that proficiency.

7.5.1 Device Tree Source

The device tree blob is “compiled” by a special compiler that produces the binary in the proper form for U-Boot and Linux to understand. The `dtc` compiler usually is provided with your embedded Linux distribution, or it can be found at <http://jdl.com/> software. Listing 7-14 shows a snippet of the device tree source (DTS) from a recent kernel source tree.

LISTING 7-14 Partial Device Tree Source Listing

```
/*
 * MPC8548 CDS Device Tree Source
 *
 * Copyright 2006, 2008 Freescale Semiconductor Inc.
 *
 * This program is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License as published by the
 * Free Software Foundation; either version 2 of the License, or (at your
 * option) any later version.
 */

/dts-v1/;

/ {
    model = "MPC8548CDS";
    compatible = "MPC8548CDS", "MPC85xxCDS";
    #address-cells = <1>;
    #size-cells = <1>;

    aliases {
        ethernet0 = &enet0;
        ethernet1 = &enet1;
        ethernet2 = &enet2;
        ethernet3 = &enet3;
        serial0 = &serial0;
        serial1 = &serial1;
        pci0 = &pci0;
        pci1 = &pci1;
        pci2 = &pci2;
        rapidio0 = &rio0;
    };

    cpus {
```

LISTING 7-14 Continued

```
#address-cells = <1>;
#size-cells = <0>

PowerPC,8548@0 {
    device_type = "cpu";
    reg = <0x0>;
    d-cache-line-size = <32>;      // 32 bytes
    i-cache-line-size = <32>;      // 32 bytes
    d-cache-size = <0x8000>;        // L1, 32K
    i-cache-size = <0x8000>;        // L1, 32K
    timebase-frequency = <0>;      // 33 MHz, from uboot
    bus-frequency = <0>;          // 166 MHz
    clock-frequency = <0>;          // 825 MHz, from uboot
    next-level-cache = <&L2>;
};

memory {
    device_type = "memory";
    reg = <0x0 0x8000000>; // 128M at 0x0
};

localbus@e0000000 {
    #address-cells = <2>;
    #size-cells = <1>;
    compatible = "simple-bus";
    reg = <0xe0000000 0x5000>;
    interrupt-parent = <&mpic>;

    ranges = <0x0 0x0 0xff000000 0x01000000>; /*16MB Flash*/

    flash@0,0 {
        #address-cells = <1>;
        #size-cells = <1>;
        compatible = "cfi-flash";
        reg = <0x0 0x0 0x1000000>;
        bank-width = <2>;
        device-width = <2>;
        partition@0x0 {
            label = "free space";
            reg = <0x00000000 0x00f80000>;
        };
    };
}
```

LISTING 7-14 Continued

```

partition@0x100000 {
    label = "bootloader";
    reg = <0x00f80000 0x00080000>;
    read-only;
};

};

};

<...truncated here...>

```

This is a long listing, but it is well worth the time spent studying it. Although it may seem obvious, it is worth noting that this device tree source is specific to the Freescale MPC8548CDS Configurable Development System. Part of your job as a custom embedded Linux developer is to adopt this DTS to your own MPC8548-based system.

Some of the data shown in Listing 7-14 is self-explanatory. The flat device tree is made up of device nodes. A device node is an entry in the device tree, usually describing a single device or bus. Each node contains a set of properties that describe it. It is, in fact, a tree structure. It can easily be represented by a familiar tree view, as shown in Listing 7-15.

LISTING 7-15 Tree View of DTS

```

/- Model: model = "MPC8548CDS", etc.

|
|---- cpus: #address-cells = <1>, etc.
|    |
|    |---- PowerPC,8548@0, etc.

|
|--- Memory: device_type = "memory", etc.

|
|---- localbus@e0000000: #address-cells = <2>, etc.
|    |
|    |---- flash@0,0: #address-cells = <1>, etc.

|
<...>

```

In the first few lines of Listing 7-14, we see the processor model and a property indicating compatibility with other processors in the same family. The first child node describes the CPU. Many of the CPU device node properties are self-explanatory. For example, we can see that the 8548 CPU has data and instruction cache line sizes of

32 bytes and that these caches are both 32KB in size (0x8000 bytes.) We see a couple properties that show clock frequencies, such as `timebase-frequency` and `clock-frequency`, both of which indicate that they are set by U-Boot. That would be natural, because U-Boot configures the hardware clocks.

The properties called `address-cells` and `size-cells` are worth explaining. A “cell” in this context is simply a 32-bit quantity. `address-cells` and `size-cells` simply indicate the number of cells (32-bit fields) required to specify an address (or size) in the child node.

The `memory` device node offers no mysteries. From this node, it is obvious that this platform contains a single bank of memory starting at address 0, which is 128MB in size.

For complete details of flat device tree syntax, consult the references at the end of this chapter. One of the most useful is the document produced by Power.org, found at www.power.org/resources/downloads/Power_ePAPR_APPROVED_v1.0.pdf.

7.5.2 Device Tree Compiler

Introduced earlier, the device tree compiler (`dtc`) converts the human-readable device tree source into the machine-readable binary that both U-Boot and the Linux kernel understand. Although a `git` tree is hosted on [kernel.org](https://git.kernel.org/) for `dtc`, the device tree source has been merged into the kernel source tree and is built along with any Power Architecture kernel from the `.../arch/powerpc` branch.

It is quite straightforward to use the device tree compiler. A typical command to convert source to binary looks like this:

```
$ dtc -O dtb -o myboard.dtb -b 0 myboard.dts
```

In this command, `myboard.dts` is the device tree human-readable source, and `myboard.dtb` is the binary created by this command invocation. The `-O` flag specifies the output format—in this case, the device tree blob binary. The `-o` flag names the output file, and the `-b 0` parameter specifies the physical boot CPU in the multicore case.

Note that the `dtc` compiler allows you to go in both directions. The command example just shown performs a compile from source to device tree binary, whereas a command like this produces source from the binary:

```
$ dtc -I dtb -O dts mpc8548.dtb >mpc8548.dts
```

You can also build the DTB for many well-known reference boards directly from the kernel source. The command looks similar to the following:

```
$ make ARCH=powerpc mpc8548cds.dtb
```

This produces a binary device tree blob from a source file with the same base name (`mpc8548cds`) and the `.dts` extension. These are found in `.../arch/powerpc/boot/dts`. A recent kernel source tree had 120 such device tree source files for a range of Power Architecture boards.

7.5.3 Alternative Kernel Images Using DTB

Entering `make ARCH=powerpc help` at the top-level Linux kernel source tree outputs many lines of useful help, describing the many build targets available. Several architecture-specific targets combine the device tree blob with the kernel image. One good reason to do this is if you are trying to boot a newer kernel on a target that has an older version of U-Boot that does not support the device tree blob. On a recent Linux kernel, Listing 7-16 reproduces the `powerpc` targets defined for the `powerpc` architecture.

LISTING 7-16 Architecture-Specific Targets for Powerpc

* zImage	- Build default images selected by kernel config
zImage.*	- Compressed kernel image (<code>arch/powerpc/boot/zImage.*</code>)
uImage	- U-Boot native image format
cuImage.<dt>	- Backwards compatible U-Boot image for older versions which do not support device trees
dtbImage.<dt>	- zImage with an embedded device tree blob
simpleImage.<dt>	- Firmware independent image.
treeImage.<dt>	- Support for older IBM 4xx firmware (not U-Boot)
install	- Install kernel using (your) <code>~/bin/installkernel</code> or (distribution) <code>/sbin/installkernel</code> or install to <code>\$(INSTALL_PATH)</code> and run <code>lilo</code>
*_defconfig	- Select default config from <code>arch/powerpc/configs</code>

The `zImage` is the default, but many targets use `uImage`. Notice that some of these targets have the device tree binary included in the composite kernel image. You need to decide which is most appropriate for your particular platform and application.

7.6 Other Bootloaders

Here we introduce the more popular bootloaders, describe where they might be used, and summarize their features. This is not intended to be a thorough tutorial; doing so would require a book of its own. Consult the last section of this chapter for further study.

7.6.1 Lilo

The Linux Loader, or Lilo, was widely used in commercial Linux distributions for desktop PC platforms; as such, it has its roots in the Intel x86/IA32 architecture. Lilo has several components. It has a primary bootstrap program that lives on the first sector of a bootable disk drive.⁵ The primary loader is limited to a disk sector size, usually 512 bytes. Therefore, its primary purpose is simply to load and pass control to a secondary loader. The secondary loader can span multiple sectors and does most of the bootloader's work.

Lilo is driven by a configuration file and utility that is part of the Lilo executable. This configuration file can be read or written to only under control of the host operating system. That is, the configuration file is not referenced by the early boot code in either the primary or secondary loaders. Entries in the configuration file are read and processed by the Lilo configuration utility during system installation or administration. Listing 7-17 shows a simple `lilo.conf` configuration file describing a typical dual-boot Linux and Windows installation.

LISTING 7-17 Sample Lilo Configuration: `lilo.conf`

```
# This is the global lilo configuration section
# These settings apply to all the "image" sections

boot = /dev/hda
timeout=50
default=linux

# This describes the primary kernel boot image
# Lilo will display it with the label 'linux'
image=/boot/myLinux-2.6.11.1
    label=linux
    initrd=/boot/myInitrd-2.6.11.1.img
```

⁵ This is mostly for historical reasons. From the early days of PCs, BIOS programs loaded only the first sector of a disk drive and passed control to it.

LISTING 7-17 Continued

```
read-only
append="root=LABEL=/"

# This is the second OS in a dual-boot configuration
# This entry will boot a secondary image from /dev/hda1
other=/dev/hda1
    optional
    label=that_other_os
```

This configuration file instructs the Lilo configuration utility to use the master boot record of the first hard drive (`/dev/hda`). It contains a delay instruction to wait for the user to press a key before the timeout (5 seconds, in this case). This allows the system operator to select from a list of OS images to boot. If the system operator presses the Tab key before the timeout, Lilo presents a list to choose from. Lilo uses the label tag as the text to display for each image.

The images are defined with the image tag in the configuration file. In Listing 7-17, the primary (default) image is a Linux kernel image with a filename of `myLinux-2.6.11.1`. Lilo loads this image from the hard drive. It then loads a second file to be used as an initial ramdisk. This is the file `myInitrd-2.6.11.1.img`. Lilo constructs a kernel command line containing the string `"root=LABEL=/"` and passes this to the Linux kernel upon execution. This instructs Linux where to get its root file system after boot.

7.6.2 GRUB

Many current commercial Linux distributions now ship with the GRUB bootloader. GRUB, or GRand Unified Bootloader, is a GNU project. It has many enhanced features not found in Lilo. The biggest difference between GRUB and Lilo is GRUB's capability to understand file systems and kernel image formats. Furthermore, GRUB can read and modify its configuration at boot time. GRUB also supports booting across a network, which can be a tremendous asset in an embedded environment. GRUB offers a command-line interface at boot time to modify the boot configuration.

Like Lilo, GRUB is driven by a configuration file. Unlike Lilo's static configuration, however, the GRUB bootloader reads this configuration at boot time. This means that the configured behavior can be modified at boot time for different system configurations.

Listing 7-18 is a sample GRUB configuration file. This is the configuration file from the PC on which this book was written. The GRUB configuration file is called `grub.conf`⁶ and usually is placed in a small partition dedicated to storing boot images. On the machine from which this example was taken, that directory is called `/boot`.

LISTING 7-18 Sample GRUB Configuration File: grub.conf

```
default=0
timeout=3
splashimage=(hd0,1)/grub/splash.xpm.gz

title Fedora Core 2 (2.6.9)
    root (hd0,1)
    kernel /bzImage-2.6.9 ro root=LABEL=/ rhgb proto=imps quiet
    initrd /initrd-2.6.9.img

title Fedora Core (2.6.5-1.358)
    root (hd0,1)
    kernel /vmlinuz-2.6.5-1.358 ro root=LABEL=/ rhgb quiet

title That Other OS
    rootnoverify (hd0,0)
    chainloader +1
```

GRUB first presents the user with a list of images that are available to boot. The title entries from Listing 7-18 are the image names presented to the user. The default tag specifies which image to boot if no keys have been pressed in the timeout period, which is 3 seconds in this example. Images are counted starting from 0.

Unlike Lilo, GRUB can actually read a file system on a given partition to load an image from. The `root` tag specifies the root partition from which all filenames in the `grub.conf` configuration file are rooted. In this sample configuration, the root is partition number 1 on the first hard disk drive, specified as `root (hd0,1)`. Partitions are numbered from 0; this is the second partition on the first hard disk.

The images are specified as filenames relative to the specified root. In Listing 7-18, the default boot image is a Linux 2.6.9 kernel with a matching initial ramdisk image called `initrd-2.6.9.img`. Notice that the GRUB syntax has the kernel command-line parameters on the same line as the kernel file specification.

⁶ Some newer distributions call this file `menu.lst`.

7.6.3 Still More Bootloaders

Numerous other bootloaders have found their way into specific niches. For example, Redboot is another open source bootloader that Intel and the XScale community have adopted for use on various evaluation boards based on the Intel IXP and Marvel PXA processor families. Micromonitor is in use by board vendors such as Cogent and others. YAMON⁷ has found popularity in MIPs circles. LinuxBIOS is used primarily in X86 environments. In general, when you consider a boot loader, you should consider some important factors up front:

- Does it support my chosen processor?
- Has it been ported to a board similar to my own?
- Does it support the features I need?
- Does it support the hardware devices I intend to use?
- Is there a large community of users where I might get support?
- Are there any commercial vendors from which I can purchase support?

These are some of the questions you must answer when considering what bootloader to use in your embedded project. Unless you are doing something on the “bleeding edge” of technology using a brand-new processor, you are likely to find that someone has already done the bulk of the hard work in porting a bootloader to your chosen platform. Use the resources listed at the end of this chapter to help make your final decisions.

7.7 Summary

This chapter examined the role of the bootloader and discovered the limited execution context in which a bootloader must exist. We covered one of the most popular bootloaders, U-Boot, in some detail. We walked through the steps of a typical port to a board with similar support in U-Boot. We briefly introduced additional bootloaders in use today so that you can make an informed choice for your particular requirements.

- The bootloader’s role in an embedded system cannot be overstated. It is the first piece of software that takes control upon applying power.

⁷ In an acknowledgment of the number of bootloaders in existence, the YAMON user’s guide bills itself as Yet Another MONitor.

- Das U-Boot has become a popular universal bootloader for many processor architectures. It supports a large number of processors, reference hardware platforms, and custom boards.
- U-Boot is configured using a series of configuration variables in a board-specific header file. Appendix B contains a list of all the standard U-Boot command sets supported in a recent U-Boot release.
- Porting U-Boot to a new board based on a supported processor is relatively straightforward.
- There is no substitute for detailed knowledge of your processor and hardware platform when bootloader modification or porting must be accomplished.
- You may need a device tree binary for your board, especially if it is Power Architecture and soon perhaps ARM.

7.7.1 Suggestions for Additional Reading

Application Note: Introduction to Synchronous DRAM

Maxwell Technologies

www.maxwell.com/pdf/me/app_notes/Intro_to_SDRAM.pdf

Using LD, the GNU linker

Free Software Foundation

<http://sourceware.org/binutils/docs/ld/index.html>

The DENX U-Boot and Linux Guide (DLUG) for TQM8xxL

Wolfgang Denx, et al., Denx Software Engineering

www.denx.de/twiki/bin/view/DULG/Manual

RFC 793, “Trivial File Transfer Protocol”

The Internet Engineering Task Force

www.ietf.org/rfc/rfc783.txt

RFC 951, “Bootstrap Protocol”

The Internet Engineering Task Force

www.ietf.org/rfc/rfc951.txt

RFC 1531, “Dynamic Host Control Protocol”

The Internet Engineering Task Force

www.ietf.org/rfc/rfc1531.txt

PowerPC 405GP Embedded Processor user manual
International Business Machines, Inc.

Programming Environments Manual for 32-bit Implementations of the PowerPC Architecture
Freescale Semiconductor, Inc.

Lilo Bootloader
www.tldp.org/HOWTO/LILO.html

GRUB Bootloader
www.gnu.org/software/grub/

Device tree documentation
Linux Kernel Source Tree
.../Documentation/powerpc/booting-without-of.txt

Device trees everywhere
David Gibson, Benjamin Herrenschmidt
<http://ozlabs.org/people/dgibson/papers/dtc-paper.pdf>

Excellent list of flat device tree references
www.denx.de/wiki/U-Boot/UBootFdtInfo#Background_Information_on_Flatte

Index

Symbol

\ (UNIX line-continuation character), 119

A

“A Non-Technical Look Inside the EXT2 File System” website, 259

Abatron website, 410

access rights, 26

add-symbol-file command, 403

addr2line utility, 361

adduser BusyBox command, 562

adjtimex BusyBox command, 562

Almesberger, Werner, 157

AltiVec, 41

AMCC

Power Architecture processors, 50-53

Yosemite board kernel debugging example, 381-382

announcement of Linux, 64

applications, multithreaded, 438-441

ar BusyBox command, 562

architecture

device drivers, 204

embedded systems, 12

init user space process, 19

kernel, booting, 16-18

kernel initialization, 18-19

setup, 13-14

target boards, starting, 15-16

setup routine, 114

specific targets, 193

ARM

Corporate Backgrounder website, 56

processors, 55

additional companies, 59

Freescale, 58-59

TI, 56-57

website, 59

arp BusyBox command, 562

arping BusyBox command, 562

ash BusyBox command, 562

ATCA hardware platforms, 60-61

autoconf.h file, 82-83

automating root file system builds, 137

autotools.bbclass class, 461

B

backtrace command, 330

basename BusyBox command, 562

bbconfig BusyBox command, 562

bbsh BusyBox command, 562

BDI-2000 configuration file sample, 586-592

BeagleBoard, 57, 62, 513

big kernel locks (BKLs), 473

binary tools

addr2line, 361

ldd, 362-363

nm, 363-364

objcopy, 360-361

objdump, 359

prelink, 364

readelf, 355-359

resources, 365

strings, 362

strip, 361

BIOS, 11

BitBake Hello World recipe processing, 458-459

BitBake (OpenEmbedded), 137, 456

BKLs (big kernel locks), 473

blkid BusyBox command, 562

board-specific initialization, 181-184

boot blocks, 21-22

booting

from disks, 174

kernel, 16-18

KGDB enabled with U-Boot, 373-374

- messages, 106-109
troubleshooting, 417-420
- Booting Linux: The History and the Future,** 157
- bootloaders,** 11
challenges
 DRAM controllers, 161-162
 execution context, 165
 image complexity, 162-165
 storage, 162
debugging, 441
GRUB, 195-196
initial serial output, 15
initrd support, 148-150
Lilo, 194-195
Micromonitor, 197
Redboot, 197
roles, 160-161
selecting, 197
startup tasks, 11
- U-Boot**
 booting from disks, 174
 BOOTP client/server handshakes, 171
 commands, 169-170
 configuring, 167-169
 DHCP target identification, 172-173
 DTBs on boot sequence, 187-188
 Ethernet interface support, 170
 finding, 166
 image formats, 185-186
 porting, 174-185
 reference website, 198
 storage subsystems, 173
 website, 166
- bootm command,** 17
- BOOTP (Bootstrap Protocol),** 171
servers, configuring, 313-316
U-Boot bootloader support, 171
website, 198, 323
- bootstrap loaders,** 105-106
- bottom-half processing,** 468
- brctl BusyBox command,** 562
- breakpoints**
 KGDB, 376
 target memory, 383
- Broadcom SiByte processors,** 54-55
- building**
 file systems, 256-257
 JFFS2 images, 240-242
 UBIFS images, 284-287
- build numbers,** 109
- Buildroot,** 137, 451
 configuring, 451-452
 installing, 451
- output, 452-454
website, 464
- build systems**
benefits, 446-447
Buildroot, 451-454
kernel
 autoconf.h file, 82-83
 configuration editors, 80-82
 custom configuration options, 91-94
 dot-config file, 78-80
 final sequence example, 101
 Kconfig files, 89-91
 makefiles, 95
 makefile targets, 83-89
OpenEmbedded, 454-463
Scratchbox, 447-450
- bunzip2 BusyBox command,** 562
- BusyBox**
applets, 302-303
commands, listing of, 563-570
configuring, 291-293
cross-compiling, 293
default startup, 298
description, 295
launching, 293
mdev, 545-547
output example, 294-295
overview, 290-291
rcs initialization script, 299-300
symlinks, 300-302
system initialization, 297-299
target directory structure, 295
toolkit, 135
website, 304
- busybox command,** 562
- bzcat BusyBox command,** 562
- bzImage targets,** 83
- bzip2 BusyBox command,** 562
- C**
- C function with local variables listing,** 163
- cable assemblies (USB), 494
- cal BusyBox command,** 562
- carrier-grade, 6
- cat BusyBox command,** 562
- cav BusyBox command,** 562
- cbrowser utility,** 335-336, 365
- CDC (Communications Device Class) drivers,** 512-515
- cell write lifetimes (Flash memory),** 22
- CFI (Common Flash Interface),** 270

chat BusyBox command, 562
 chattr BusyBox command, 562
 chcon BusyBox command, 562
 checking file system integrity, 233-235
 chgrp BusyBox command, 562
 chipsets, 41-43
 chmod BusyBox command, 562
 chown BusyBox command, 562
 chpasswd BusyBox command, 562
 chpst BusyBox command, 562
 chroot BusyBox command, 562
 chrt BusyBox command, 563
 chvt BusyBox command, 563
 cksum BusyBox command, 563
 classes (OpenEmbedded metadata), 461-462
 clear BusyBox command, 563
 clocks, configuring, 574-575
 cmp BusyBox command, 563
 coldplug processing (udev), 537-538
command-line
 options, 341-342
 partitions, 273-274
 processing, 115-116
 code listing, 119-121
 parameters, 115-116
 setup macro, 116-118
commands. *See also* **utilities**
 add-symbol-file, 403
 backtrace, 330
 bootm, 17
 BusyBox, listing of, 563-570
 connect, 393
 continue, 382
 dd, 257
 detach, 443
 e2fsck, 233-235
 GDB user-defined, 392-393
 git, 166
 i shared, 432
 iminfo, 186
 kgdboc, 380
 kgdbwait, 380
 ldd, 139, 362-363, 432-433
 make distclean, 78
 make gconfig, 81
 make menuconfig, 291
 mkcramfs, 242
 mkfs.ext2, 257
 mkfs.jffs2, 241
 modinfo, 539
 modprobe, 532-533
 mount, 232
 shutdown, 156
 stop-on-solib-events, 432
 tftp, 17
 ubiformat, 286
 U-Boot bootloader supported, 169-170
 U-Boot configurable, 558-560
 udevadm, 523-524, 543-544
commBusyBox command, 563
commercial distributions, 33
Common Flash Interface (CFI), 270
Common Flash Memory Interface Specification, 288
Communications Device Class (CDC), 512-515
CompactPCI hardware platform, 60
companion chipsets, 41-43
compiling
 DTBs, 192-193
 dtc, 192-193
 kernel, 70-72
 native compilation, 30
components required, 97
composite kernel image
 architecture objects, 104
 boot messages, 106-109
 bootstrap loaders, 105-106
 constructing, 100-102
 final build sequence example, 101
 Image object, 103
 piggy assembly file, 104
configuration descriptors, 491
configuration editors, 80-82
configuring
 board-specific MTD partitions, 276-278
 BOOTP servers, 313-316
 Buildroot, 451-452
 BusyBox, 291-293
 busybox mdev, 547
 clocks, 574-575
 device drivers, 205-208
 ARM system example, 208
 directory, creating, 206
 makefile, editing, 208
 menu items, adding, 206-207
 output, 208
 DHCP servers, 313-316
 DRAM controllers, 161-162
 inittab file, 143-144
 KGDB
 kernel, 371
 runtime, 380-381
 MTD, 263, 267
 NFS kernel support, 247
 NFS servers, 316-318

OpenEmbedded, 462-463
Scratchbox environment, 449
SDRAM controllers, 575-579
 memory bank control register, 578
 timing requirements, 578-579
 U-Boot sdram_init() function, 576-577
TFTP servers, 312-313
UBIFS, 284
U-Boot
 bootloader, 167-169
 build tree, 177-178
 makefile targets, 176-177
udev rules, 533-535
USB, 495-497
 core makefile, 496-497
 Freescale Semiconductor iMX31 Applications Processor example, 496
 volume of options, 495
connect command, 393
connections
 Ethernet, 512-515
 KGDB, 374-375
connectors (USB), 492-493
contexts (execution), 26
continue command, 382
controllers (SDRAM), configuring, 575-579
core dumps, debugging, 327-329
cp BusyBox command, 563
cpio BusyBox command, 563
cpp search directories, 309
cramfs file system, 242-244
cramfs project README file website, 259
crond BusyBox command, 563
crontab BusyBox command, 563
cross-compiling
 BusyBox, 293
 targets, 448-450
cross debugging, 424
cross-development environments, 30-31
 default cross-search directories, 310
 flexibility, 307
 Hello World program, 307-309
 hosts, 306
 layout, 307
 overview, 306
 targets, 306
cross-strip utility, 426-427
cross tools, distributions, 33
cryptpw BusyBox command, 563
cttyhack BusyBox command, 563
customizing
 initramfs, 154-155
 udev, 540

D

Das U-Boot. *See* U-Boot bootloader
dateBusyBox command, 563
dc BusyBox command, 563
dd BusyBox command, 563
dd command, 257
DDD (Data Display Debugger), 333-335
 debug session, 335
 invoking, 334
 resources, 365
deallocvt BusyBox command, 563
debugging
 booting, 417
 early serial debug output, 417
 KGDB trapping crashes on panic, 420
 printk log buffer, dumping, 417-419
 bootloaders, 441
 cbrowser, 335-336, 365
 core dumps, 327-329
 cross, 424
 DDD, 333-335, 365
 dmalloc, 365
 Flash code, 441
 GDB, 326
 backtrace command, 330
 core dumps, 327-329
 debug sessions, 331-333
 invoking, 329-331
 resources, 365
 sessions, 331-333
 stack frames, 330
 website, 422
 hardware-assisted. *See* JTAG probes
 with JTAG probes, 413-417
kernel. *See* kernel debugging
multiple processes, 435-438
multithreaded applications, 438-441
real time kernel patch, 473-475
 features, 475-476
 O(1) scheduler, 476
 preemption modes, 474-475
 real-time processes, creating, 477
remote. *See* remote debugging
shared libraries, 429
 events, 431-434
 finding, 433
 initial target memory segment mapping, 430-431
 invoking ldd command, 432-433
 locations, 433
 </proc/pid>/maps memory segments, 434
 requirements, 430
 viewing, 432

- target, 424
- USB, 516-518
 - device driver support*, 518
 - Ethernet dongle insertion debug output example*, 516
 - platform-specific options*, 517
 - usbmon utility*, 517-518
- delgroup BusyBox command, 563
- deluser BusyBox command, 563
- Denx, Wolfgang, 166
- depmod BusyBox command, 563
- depmod utility, 214-215
- "Design and Implementation of the Second Extended Filesystem" website, 259
- detach command, 443
- /dev directory, 522
- development
 - cross-development environments. *See* cross-development environments
- hosts
 - BOOTP/DHCP servers, configuring*, 313-316
 - NFS servers, configuring*, 316-318
 - requirements*, 311-312
 - target NFS root mount*, 318-321
 - TFTP servers, configuring*, 312-313
- setup, 13-14
- device drivers
 - architecture, 204
 - build configuration, 205-208
 - debugging, 402-406
 - init code*, 406-407
 - initializing*, 403-404
 - loopback breakpoints*, 405
 - sessions, initiating*, 404-405
 - symbolic debug information, accessing*, 402
 - dependencies, 214-215
 - dynamic, 80
 - ext3 and jbd relationship, 213-214
 - GPL, 224
 - information, viewing, 216
 - installing, 209-210
 - listing of, viewing, 213
 - loading/unloading, 203, 210, 528
 - methods
 - device nodes*, 220-221
 - file system operations*, 217-220
 - numbers, allocating*, 220
 - minimal example, 204-205
 - out-of-tree, 223-224
 - parameters, 211-212
 - platform, loading, 538-539
 - removing from kernels, 215-216
 - resources, 226
 - running kernels, adding, 212
 - USB support, debugging, 518
 - user space application example, 222-223
 - utilities
 - depmod*, 214-215
 - insmod*, 212
 - lsmod*, 213
 - modinfo*, 216
 - modprobe*, 213-214
 - rmmod*, 215-216
- devices
 - descriptors, 490
 - discovery, 523-524
 - loopback, 256
 - nodes, 220-221, 522
 - persistent naming, 541-545
 - trees
 - blobs*. *See DTBs*
 - compiler*, 192-193
 - loading*, 17
 - source*, 189-192
 - website*, 199
- devmem BusyBox command, 563
- df BusyBox command, 563
- DHCP (Dynamic Host Configuration Protocol), 171
 - servers, configuring, 313-316
 - U-Boot bootloader support, 172-173
 - website, 198
- dhcrelay BusyBox command, 563
- diff BusyBox command, 563
- directories
 - /dev, 522
 - root file systems, 134
 - runlevels, 142
 - top-level kernel source, 69
- dirname BusyBox command, 563
- disassembled object code, viewing, 359
- discovering devices, 523-524
- distributions
 - commercial, 33
 - components, 97
 - cross tools, 33
 - defined, 32
 - do-it-yourself, 33-34
 - file sizes, 239
 - installing, 33
 - packages, 32
 - targets, 33
- malloc utility, 350-353
 - libraries, generating, 350
 - log output example, 351-352

requirements, 350
resources, 365
dmesg BusyBox command, 563
dnssd BusyBox command, 563
do-it-yourself distributions, 33-34
dongles, 515
dos2unix BusyBox command, 563
dot-config file, 78
 code snippet listing, 79-80
 customizations, 93-94
 deleting, 78
 hidden, 78
downloading kernel, 68
dpkg BusyBox command, 563
dpkg-deb BusyBox command, 563
DRAM (Dynamic Random Access Memory), 161-162, 198
drivers
 device. *See* device drivers
 Flash chips, 276
 g_ether, 513
 KGDB I/O, 379-380
 mapping, 274-276
 platform device, loading, 538-539
 sd-mod, adding, 509
 USB
 CDC, 512-515
 HID class support, 511
 host controller, installing, 498
 usb_storage, 508
DTBs (device tree blobs), 187
 architecture-specific targets, 193
 boot sequence role, 187-188
 compiling, 192-193
 device tree source, 189-192
dtc (device tree compiler), 192-193
DTS (device tree source), 189-192
du BusyBox command, 563
dumpkmap BusyBox command, 563
dumpleases BusyBox command, 563
dynamically loadable modules, 80
Dynamic Host Configuration Protocol. *See* DHCP
Dynamic Random Access Memory (DRAM), 161-162, 198

E

e2fsck BusyBox command, 563
e2fsck command, 233-235
echo BusyBox command, 563
Eclipse Project website, 365
ed BusyBox command, 563

EHCI (Enhanced Host Controller Interface), 498, 519
ELF files, 356-359
embedded systems
 architecture, 12
 init user space process, 19
 kernel, booting, 16-18
 kernel initialization, 18-19
 setup, 13-14
 target boards, starting, 15-16
 characteristics, 10-11
endpoints, 489-491
Enhanced Host Controller Interface (EHCI), 498, 519
env BusyBox command, 563
envdir BusyBox command, 563
envuidgid BusyBox command, 564
EP405 U-Boot port, 175-176
erase blocks, 21
Ethernet
 connectivity (USB), 512-515
 interfaces, 170
ether-wake BusyBox command, 564
events
 locations, 433
 shared library, 431-434
exbibytes, 237
execution contexts, 26
expand BusyBox command, 564
expr BusyBox command, 564
ext2 file systems, 257
ext3 file systems, 235-237
ext4 file systems, 237
external bus controller initialization listing, 181-182

F

fakeidentd BusyBox command, 564
FALSE BusyBox command, 564
fbset BusyBox command, 564
fbsplash BusyBox command, 564
fdflush BusyBox command, 564
fdformat BusyBox command, 564
fdisk BusyBox command, 564
fdisk utility, 229-230
fgrep BusyBox command, 564
FHS (File System Hierarchy Standard), 133, 226
File System Hierarchy Standard (FHS), 133, 226
“File System Performance: The Solaris OS, UFS, Linux ext3, and Reiser FS” website, 259

files

autoconf.h, 82-83
 BDI-2000 configuration, 586-592
 device trees, loading, 17
 dot-config file, 78
 code snippet listing, 79-80
 customizations, 93-94
 deleting, 78
 hidden, 78
 ELF, 356-359
 GDB initialization, 393
 GRUB configuration, 196
 inittab, configuring, 143-144
 Kconfig, 89-92
 kernel-parameters.txt, 115
 linuxrc, 150-151
 main.c, 113-114
 makefiles
 targets, 83-89
 U-Boot configuration target, 176-177
 uImage target wrapper script, 185
 USB core, 496-497
 Vega and Constellation example, 95
 object
 formats, converting, 360
 symbols, viewing, 363-364
 piggy assembly, 104
 size distribution, 239
 system.map, 70
 systems
 building, 256-257
 cramfs, 242-244
 ext2, 257
 ext3, 235-237
 ext4, 237
 Flash, 24
 integrity, checking, 233-235
 JFFS2. See *JFFS2*
 journaling, 235
 mounting, 232-233
 NFS, 244-248
 partition relationship, 229
 pseudo. See */proc file system; sysfs file system*
 ramfs, 255-256
 ReiserFS, 238
 resources, 259
 root. See *root file systems*
 sysfs, 252-255, 500-502
 tmpfs, 256
 UBI, 284
 UBIFS, 284-287, 500-502
 USBFS, 502-504
 ubinize configuration, 285

versions, 109
 vmlinux, 70-72
 image components, 73-76
 listing, 72-73
find_next_task macro, 400
find_task macro, 394-395
findfs BusyBox command, 564
finding
 kernels, 96
 shared libraries, 433
 U-Boot bootloader, 166
Flash, 24
 chip drivers, 276
 code, debugging, 441
 device listing, 232
 memory. *See memory, Flash*
flash_erase utility, 280
flashcp utility, 280
flashing, 280
flat device tree websites
 references, 199
 syntax, 192
flow of control, 109-111
 architecture setup, 114
 head.o module, 111-113
 startup file, 113-114
fold BusyBox command, 564
fork() function, 435-437
founder of Linux, 6
free BusyBox command, 564
freedom versus free, 4-5
freeramdisk BusyBox command, 564
Freescale processors
 ARM, 58-59
 MPC7448, 40-41
Power Architecture, 44-48
 PowerQUICC I, 45-46
 PowerQUICC II, 46-47
 PowerQUICC II Pro, 47
 PowerQUICC III, 48
QorIQ, 48-50
Semiconductor iMX31 Applications Processor
 USB example, 496
 bus topology, 507
 configuration options, 496
 device node, creating, 510
 host controller drivers, installing, 498
 partition, mounting, 510
 sysfs file system output, 500-502
 usbview output, 504-507
 website, 62
free versus freedom, 4-5
fsck BusyBox command, 564

fsck.minix BusyBox command, 564
ftpget BusyBox command, 564
ftpput BusyBox command, 564
Ftrace utility
 interrupt off timing measurements, 484
 kernel performance analysis, 478-479
 preemption off measurements, 479-481
 wakeup latency measurements, 481-483
functions. *See also* methods
 fork(), 435-437
 gethostbyname(), 432
 prepare_namespace, 151
 pthread_create(), 438
 sdram_init(), 576-577
 setup_arch(), 114
 start_kernel(), 114
fuser BusyBox command, 564

G

g_ether driver example, 513
Garzik, Jeff's git utility website, 68
GCC website, 323
GDB (GNU Debugger), 326. *See also* KGDB
 backtrace command, 330
 bootloaders, 441
 core dumps, 327-329
 cross debugging, 424
 debug sessions, 331-333
 detach command, 443
 Flash code, 441
 invoking, 329-331
 multiple processes, 435-438
 multithreaded applications, 438-441
 remote debugging
 file stripping, 426-427
 gdbserver utility, 427-429
 sample program ELF file debug information, 425-426
 remote serial protocol, 382-385
 resources, 365, 444
 shared libraries, 429
 events, 431-434
 finding, 433
 initial target memory segment mapping, 430-431
 invoking ldd command, 432-433
 locations, 433
 </proc/pid>/maps memory segments, 434
 requirements, 430
 viewing, 432
 stack frames, 330
 website, 444

gdbserver utility, 427-429
General Public License. *See* GNU, GPL
getenforce BusyBox command, 564
gethostbyname() function, 432
getsebool BusyBox command, 564
getty BusyBox command, 564
git command
 kernel downloads, 68
 U-Boot bootloader, 166
GNU
 Compiler Collection documentation website, 130
 Debugger. *See* GDB
 GPL (General Public License), 3-4, 550
 device drivers, 224
 exact reproduction, 550-556
 website, 550
 linker website, 130, 198
 Press website, 422
grep BusyBox command, 564
growth of embedded Linux, 2
GRUB (Grand Unified Bootloader), 195-196, 199
gunzip BusyBox command, 564
gzip applet, 302
gzip BusyBox command, 564

H

halt BusyBox command, 564
hard real time, 467
hardware-assisted debugging, 312
hardware-debug probe. *See* JTAG probes
hardware platforms, 60-61
hd BusyBox command, 564
hdparm BusyBox command, 564
head BusyBox command, 564
head.o module, 111-113
Hello World program, 28-29
 cross-development environments, 307-310
 cpp search directories, 309
 default cross-search directories, 310
 listing, 307-308
 OpenEmbedded version, 457-459
 Scratchbox example, 449
hexdump BusyBox command, 564
HID (Human Input Device), 511-512
hosted BusyBox command, 564
hostname BusyBox command, 564
hosts
 controllers, 489
 cross-development environments, 306
 mode (USB), 494

- requirements, 311-312
target boards
BOOTP/DHCP servers, configuring, 313-316
NFS root mount, 318-321
NFS servers, configuring, 316-318
TFTP servers, configuring, 312-313
httpd BusyBox command, 564
hush BusyBox command, 564
- I**
- i** shared command, 432
IBM 970FX processor, 39
id BusyBox command, 564
ifconfig BusyBox command, 565
ifdown BusyBox command, 565
ifenslave BusyBox command, 565
ifup BusyBox command, 565
images, 103
 bootloader complexities, 162-165
 composite kernel
 architecture objects, 104
 boot messages, 106-109
 bootstrap loaders, 105-106
 constructing, 100-102
 final build sequence example, 101
 Image object, 103
 piggy assembly file, 104
initrd, 148
 creating, 152-153
 decompressing, 151
JFFS2, building, 240-242
OpenEmbedded recipes, 463
U-Boot bootloader format, 185-186
UBIFS, building, 284-287
 vmlinux file, 73-76
iminfo command, 186
inetd BusyBox command, 565
init BusyBox command, 565
initcall_debug parameter, 127
initcall macros, 122-126
initialization
 board-specific, 181-184
 details, viewing, 127
 flow of control, 109-111
 architecture setup, 114
 head.o module, 111-113
 startup file, 113-114
 inittab file, 143-144
 kernel, 18-19
 creating, 125-126
 details, viewing, 127
- final boot steps, 127-129**
flow of control, 109-114
initcall macros, 126
user space process, 19
library dependencies, resolving, 139
processors, 178-180
runlevels, 141-142
startup scripts, 144-145
subsystems, 122-124
System V Init. *See System V Init*
udev setup, 535
 coldplug processing, 537-538
 default static device nodes, 536
 startup script, 535-536
USB, 499-500
 host controllers, 498-499
 usbcore module, loading, 497
user space process, 19
user-specified, 140
web server startup script example, 145-146
- initramfs, 153**
 customizing, 154-155
 file specification, 154
 initrd, compared, 153
 kernel build directory contents, 153
initrd root file system, 146
 booting, 147-148
 bootloader support, 148-150
 images, 148
 creating, 152-153
 decompressing, 151
 initramfs, compared, 153
 linuxrc file, 150-151
 mounting, 151
inittab file, configuring, 143-144
inodes, 231
inotifyd BusyBox command, 565
insmod BusyBox command, 565
insmod utility, 212
install BusyBox command, 565
installing
 Buildroot, 451
 device drivers, 209-210
 distributions, 33
 Scratchbox, 447-448
integrated SOC processors, 43
 AMCC Power Architecture, 50-53
 ARM, 55-59
 Broadcom SiByte, 54-55
 Freescale. *See Freescale processors*
 MIPS, 53-55
 Power Architecture, 44

Intel processors
Atom, 40, 62
Pentium M, 39-40
XScale website, 62

interfaces
descriptors, 491
Ethernet, 170

interrupt context, 28

interrupt off timing measurements, 483-484

interrupt service routine (ISR), 467

invoking

configuration editors, 81
DDD, 334
GDB, 329-331
ps macro, 395-396
ioctl() method, 217-219
ipaddr BusyBox command, 565
ip BusyBox command, 565
ipcalc BusyBox command, 565
ipcrm BusyBox command, 565
ipcs BusyBox command, 565
iplink BusyBox command, 565
iproute BusyBox command, 565
iprule BusyBox command, 565
iptunnel BusyBox command, 565
ISR (interrupt service routine), 467

J

JFFS: The Journaling Flash File System website, 259

JFFS2 (Journaling Flash File System 2), 24, 239-240

directory layout, 241
Flash memory limitations, 239-240
images, building, 240-242
mkfs.jffs2 command, 241
mounting on MTD RAM drive, 265-266

journaling, 235

JTAG (Joint Test Action Group) probes, 410
debugging, 413-417
Flash, programming, 411-413
setting up, 411

K

kbd_mode BusyBox command, 565

Kbuild documentation website, 98

Kconfig files, 89-92

kernel

booting, 16-18
build system
autoconf.h file, 82-83
configuration editors, 80-82

custom configuration options, 91-94
dot-config file, 78-80
final sequence example, 101
Kconfig files, 89-91
makefiles, 95
makefile targets, 83-89

command-line processing, 115-116
code listing, 119-121
parameters, 115-116
setup macro, 116-118

compiling, 70-72

composite image
architecture objects, 104
boot messages, 106-109
bootstrap loaders, 105-106
constructing, 100-102
final build sequence example, 101
Image object, 103
piggy assembly file, 104

context, 19, 26

debugging. *See kernel debugging*

documentation, 96

downloading with git utility, 68

final boot
messages, 18
steps, 137-138

finding, 96

GDB. *See KGDB*

HOWTO website, 98

initialization, 18-19, 125
creating, 125-126
details, viewing, 127
final boot steps, 127-129
flow of control, 109-114
initcall macros, 126
user space process, 19

KGDB configuration, 371

NFS configuration, 247

oops, 353-355

parameters.txt file, 115

preemption, 469
challenges, 469-471
checking for, 471-472
concurrency errors, 470
critical sections, locking, 470
latency sources, 473
models, 471-472
off measurements, 479-481
real time patch modes, 474-475
SMP, 472

real time patch, 473-475
features, 475-476
O(1) scheduler, 476

- preemption modes*, 474-475
- real-time processes, creating*, 477
- real time performance analysis, 478
 - Ftrace*, 478-479
 - interrupt off timing measurements*, 483-484
 - preemption off measurements*, 479-481
 - soft lockup detection*, 484
 - wakeup latency measurements*, 481-483
- source repositories, 65-68
- subdirectory, 77-78
- subsystem initialization, 122-124
- top-level source directory, 69
- versions, 66-67
- vmlinux file, 72-76
- website, 65
- kernel debugging**, 368-369
 - JTAG probes, 410
 - debugging*, 413-417
 - Flash memory, programming*, 411-413
 - setting up*, 411
 - KGDB, 369
 - booting with U-Boot*, 373-374
 - breakpoints*, 376
 - connections*, 374-375
 - console serial port, sharing*, 377-379
 - debug session in progress*, 377
 - early kernel code support*, 379-380
 - enabling*, 372
 - kernel configuration*, 371
 - loadable modules*, 402-406
 - logic*, 372
 - macros*, 393-402
 - optimized code*, 385-392
 - platform-specific code*, 381-382
 - remote*, 382-385
 - runtime configuration*, 380-381
 - serial ports*, 372
 - setting up*, 370
 - trapping crashes on panic*, 420
 - user-defined commands*, 392-393
 - websites*, 422
 - Magic SysReq key, 409-410
 - optimized kernel code, 389
 - printk, 407-409
 - resources, 422
 - KERNELRELEASE macro, 67
 - KGDB (Kernel GDB), 369
 - booting with U-Boot*, 373-374
 - breakpoints*, 376
 - connections*, 374-375
 - console serial port, sharing*, 377-379
 - debug session in progress*, 377
 - early kernel code support, 379-380
 - enabling, 372
 - I/O drivers, 379-380
 - kernel configuration, 371
 - loadable modules, 402-406
 - init code*, 406-407
 - initializing*, 403-404
 - loopback breakpoints*, 405
 - sessions, initiating*, 404-405
 - symbolic debug information, accessing*, 402
 - logic, 372
 - macros, 393-402
 - find_next_task*, 400
 - find_task*, 394-395
 - ps*, 395-397
 - task_struct_show*, 398-399
 - optimized code, 385-392
 - platform-specific code, debugging, 381-382
 - remote serial protocol, 382-385
 - runtime configuration, 380-381
 - serial ports, 372
 - setting up*, 370
 - trapping crashes on panic*, 420
 - user-defined commands*, 392-393
 - websites*, 422
 - kgdb8250 I/O driver, 379-380
 - kgdboc command, 380
 - kgdbwait command, 380
 - kill BusyBox command*, 565
 - killall BusyBox command*, 565
 - killall5 BusyBox command*, 565
 - klogd BusyBox command*, 565
 - Kroah-Hartman, Greg, 504
 - ksoftirqd task, promoting, 476

L

- lash BusyBox command, 565
- last BusyBox command, 565
- latency
 - interrupt off timing*, 483-484
 - kernel preemption sources*, 473
 - preemption off measurements*, 479-481
 - real time*, 467-468
 - wakeup measurements*, 481-483
- layout
 - cross-development environments*, 307
 - root file systems*, 133-134
- ldd command, 139, 362-363, 432-433
- Lehrbaum, Rick, 3
- length BusyBox command, 565
- lessBusyBox command, 565

Library Optimizer Tool website, 136
Lilo
 bootloader, 194-195
 website, 199
linker command scripts, 163
linux32BusyBox command, 565
Linux
 Allocated Devices website, 548
 Documentation Project, 96, 157
 Foundation, 6-8
 news and developments resources, 583
 Standard Base, 5, 8
LinuxDevices.com, 3
linuxrc file, 150-151
listings
 architecture-specific targets, 193
 autoconf.h file, 82-83
 backtrace command, 330
 BDI-2000 configuration file, 586-592
 booting with DTBs, 187-188
 bootloaders, initial serial output, 15
 boot messages on IPX425, 106-108
 Buildroot output directory, 453
 BusyBox
 build options, 291
 default startup, 298
 gzip applet, 302
 library dependencies, 292
 mdev startup script, 546
 output, 294-295
 rcs initialization script, 299-300
 root file system example, 297
 root file system installation, 301
 target directory structure, 295
 C function with local variables, 163
 command-line MTD partitions, 273
 cramfs file system, 242-243
 device drivers
 build configuration, 206-208
 file system operations methods, 217-219
 loading/unloading, 210
 lsmod output, 213
 minimal example, 204
 modinfo output, 216
 modprobe configuration file, 214
 parameter example, 211
 user space application, 222-223
 DHCP
 exchange, 314
 server configuration, 315
 target identification, 172
 dmalloc utility, 351-352
 dot-config file, 79-80
 DTBs, 189-192
 ext2 file system, 257
 ext3 file system, 236
 file system check, 233-234
 find_next_task, 400
 find_task macro, 394
 Flash device, 232
 fork(), 435-437
 GDB
 core dump analysis, 328
 debug sessions, initiating, 332
 initialization file, 393
 stack frames, 330
 gdbserver utility
 invoking, 429
 starting on target board, 427
 target board connection, 427-428
 GRUB configuration file, 196
 Hello World, 28-29, 307-308
 cpp search directories, 309
 default cross-search directories, 310
 init process, 125-126
 initcall macros, 123-124
 initramfs
 build directory, 153
 file specification, 154
 minimal contents, 155
 initrd
 boot sequence, 148-150
 images, creating, 152
 inittab file, 143
 JFFS2
 booting as root file system, 283
 copying to root partition, 282
 directory layout, 241
 mkfs.jffs2 command, 241
 JTAG, 412-414
 Kconfig file for ARM architecture, 90
 kernel
 booting, 16-17
 build output, 70-72
 command-line processing, 119-121
 .config file customizations, 93-94
 final boot messages, 18
 final boot steps, 127-129
 final build sequence example, 101
 IPX4xx-specific Kconfig file, 92
 loading with TFTP servers, 320
 makefiles example, 95
 oops, 353
 preemption, 470-472

- subdirectory*, 77-78
- top-level ARM Kconfig file*, 92
- vmlinux file*, 72-73
- KGDB
 - booting with U-Boot*, 373-374
 - breakpoints*, 376
 - connecting*, 374-375
 - console serial port, sharing*, 378-379
 - debug session in progress*, 377
 - runtime configuration*, 380-381
 - trapping crashes on panic*, 420
- Lilo bootloader, 194
- linker command script, 163
- linuxrc file, 150-151
- loadable modules
 - debug sessions, initiating*, 404-405
 - debugging init code*, 406-407
 - initializing*, 403-404
- ltrace utility, 343
- makefiles
 - targets*, 83-89
 - U-Boot configuration*, 176
- minimal root file system, 134-135
- mstrace utility, 349
- MTD
 - configuring*, 263
 - JFFS2file system, mounting*, 265-266
- MTD partitions
 - board-specific configuration*, 276-278
 - Flash partition mounting*, 280
 - kernel partition list*, 279
 - PQ2FADS Flash mapping driver*, 274-276
- mtrace utility, 348
- multithreaded applications, debugging, 438-439
- NFS
 - exports configuration file*, 244, 317
 - root mount, booting*, 320
 - target example*, 246
- nm utility output, 363
- objdump utility, 359
- OpenEmbedded
 - autotools.bbclass example*, 461
 - BitBake Hello recipe processing*, 458-459
 - recipe example*, 457
 - tasks*, 460
- optimized kernel code, debugging
 - code*, 385-386
 - disassemble command*, 387-389
 - local variable output example*, 391
 - source file*, 389-390
- partitions
 - formatting*, 230-231
 - information, viewing*, 229-230
- piggy assembly file, 104
- platform-specific kernel debugging, 381-382
- preemption off measurements, 480
- printk log buffer, dumping, 418-419
- /proc file system, 249-251
- processes, listing, 345
- ps macro, 395-397
- ramfs file systems, 255
- readelf utility, 356-358
- real time, 476-477
- Redboot partitions
 - creating*, 272
 - detecting*, 270
 - Flash partition listing*, 269
 - Flash partitions*, 271
 - new partition list*, 272
 - power-up messages*, 269
- remote debugging
 - continue command*, 382
 - ELF file debug information*, 425-426
 - file stripping*, 426-427
 - target memory breakpoints*, 384
- resetvec source definition, 164
- runlevels, 141-142
- Scratchbox, 448-449
- SDRAM controllers, configuring, 576-577
- setup macro, 117-118
- shared libraries
 - debugging*, 430-431
 - event alerts*, 431
 - invoking ldd command*, 432-433
 - </proc/pid>/maps memory segments*, 434
- startup scripts, 144-145
- strace utility
 - profiling*, 341
 - web demo application example*, 337-340
- subsystem initialization, 122
- sysfs file system, 252-255
- task_struct_show, 398-399
- TFTP configuration, 313
- top utility default configuration, 347
- UBIFS images, building, 284-286
- U-Boot bootloader
 - 4xx startup code*, 179
 - build tree, configuring*, 177
 - configuration header file*, 168-169
 - EP405 port summary*, 184
 - external bus controller*, 181-182
 - iminfo command*, 186
 - uImage target wrapper script*, 185
- udev
 - default static device nodes*, 536
 - device discovery*, 523-524

device nodes, creating, 525-526
mouse device example, 529
persistent device naming, 541
platform device driver, loading, 538
rules configuration, 533-535
rules example, 528
startup script, 535-536
udevadm device query, 543-544
uevents emitted on USB mouse insertion, 530-531
uevents for USB interface 1-1:1.0, 531
uevents on four-port hub insertion, 525
USB automounting, 540-541

USB

- core makefile*, 496-497
- device node, creating*, 510
- direct host and peripheral links*, 513
- Ethernet dongle insertion debug output*, 516
- host controllers*, 498-499
- lsusb utility*, 507
- partition, mounting*, 510
- sd-mod driver, adding*, 509
- sysfs file system output*, 500-502
- usb-storage module*, 509
- USBFS directory listing*, 502
- usbmon utility*, 517
- usbview utility output*, 504-507

wakeup latency measurements, 481-483
web server startup script, 145-146

load_policy BusyBox command, 565
loadable modules. *See* device drivers
loading

- device drivers*, 210, 528
- platform device drivers*, 538-539

loadkmap BusyBox command, 565
logger BusyBox command, 565
login BusyBox command, 565
logname BusyBox command, 565
logread BusyBox command, 565
loopback devices, 256
losetup BusyBox command, 565
lpd BusyBox command, 565
lpq BusyBox command, 565
lpr BusyBox command, 566
lsattr BusyBox command, 566
LSB (Linux Standard Base), 5, 8
ls BusyBox command, 566
lsmod BusyBox command, 566
lsmod utility, 213
lsusb utility, 507-508
ltrace utility, 343-344
lzmacat BusyBox command, 566

M

macros

- initcall*, 122-126
- KERNELRELEASE*, 67
- KGDB*, 393-402
 - find_next_task*, 400
 - find_task*, 394-395
 - ps*, 395-397
 - task_struct_show*, 398-399

setup

- command-line processing*, 116-121
- console setup code snippet*, 117
- family definitions*, 118
- used*, 119

Magic SysReq key, 409-410
mailing list resources, 582
main.c file, 113-114
make distclean command, 78
make gconfig command, 81
make menuconfig command, 291
makedevs BusyBox command, 566
makefiles

- targets*, 83-89
- U-Boot configuration target, 176-177
- ulImage target wrapper script, 185
- USB core*, 496-497
 - Vega and Constellation example, 95

makemine BusyBox command, 566
man BusyBox command, 566
mapping drivers, 274-276
marketplace momentum, 3
mass storage class (USB), 508-511

- device node, creating*, 510
- mounting*, 510
- partition, mounting*, 510
- SCSI support*, 508
- sd-mod driver, adding*, 509
- usb_storage driver*, 508
- usb-storage module*, 509

matchpathcon BusyBox command, 566
md5sum BusyBox command, 566
mdev BusyBox command, 566
memory

- analysis tool*. *See* **dmalloc** utility
- cross-development environments*, 30-31
- DRAM*, 161-162, 198
- execution contexts*, 26
- Flash*, 20-22
 - boot blocks*, 21-22
 - cell write lifetimes*, 22
 - erasing*, 239

- file systems*, 24
 - lifetime*, 240
 - NAND*, 22-23
 - programming*, 411-413
 - typical layouts*, 23
 - writing to/erasing*, 20-21
 - layout, 25-26
 - leaks, detecting, 349
 - MMUs, 26
 - process virtual, 28-30
 - translation, 26
 - virtual, 26-30
 - Memory Management Units (MMUs)**, 26
 - Memory Technology Devices (MTD)**, 262
 - mesg** BusyBox command, 566
 - methods**. *See also* functions
 - device drivers
 - device nodes*, 220-221
 - file system operations*, 217-220
 - numbers, allocating*, 220
 - ioctl(), 217-219
 - open(), 217-219
 - release(), 217-219
 - microcom** BusyBox command, 566
 - Micromonitor bootloader**, 197
 - mini connectors, 493
 - minimal device driver example, 204-205
 - minimal root file systems, 134-136
 - MIPS processors, 53-55, 67
 - mkcramfs command, 242
 - mkdir BusyBox command, 566
 - mke2fs BusyBox command, 566
 - mkfifo BusyBox command, 566
 - mkfs.ext2 utility, 230-231, 257
 - mkfs.jffs2 command, 241
 - mkfs.minix BusyBox command, 566
 - mkimage utility, 185
 - mknod BusyBox command, 566
 - mkswap BusyBox command, 566
 - mktemp BusyBox command, 566
 - MMUs (Memory Management Units), 26
 - Moblin (Mobile Linux Initiative), 7
 - MODALIAS field, 532-533
 - modinfo utility, 216, 539
 - modprobe BusyBox command, 566
 - modprobe utility, 213-214, 532-533
 - more BusyBox command, 566
 - mount BusyBox command, 566
 - mount command, 232
 - mounting
 - dependencies, 249
 - file systems, 232-233
 - initrd, 151
 - root file systems, 18
 - USB, 510-540-541
 - USBFS, 502
 - mountpoint** BusyBox command, 566
 - mount points**, 151, 232
 - mouse device udev example, 529
 - msh** BusyBox command, 566
 - MTD (Memory Technology Device)**, 262
 - CFI support, 270
 - configuring, 263-267
 - JFFS2 file systems, mounting, 265-266
 - overview, 262-263
 - partitions, 267-268
 - board-specific configuration*, 276-278
 - command-line*, 273-274
 - configuring*, 267
 - Flash chips*, 276
 - kernel partition list*, 279
 - mapping drivers*, 274-276
 - Redboot*, 269-273
 - resources, 288
 - services, enabling, 263-265
 - utilities, 279-283
 - flash_erase*, 280
 - flashcp*, 280
 - JFFS2*, 282-283
 - kernel MTD partition list*, 279
 - MTD Flash partition, mounting*, 280
 - mtrace** utility, 348-349
 - multiple processes, debugging, 435-438
 - multithreaded applications, debugging, 438-441
 - mv** BusyBox command, 566
- N**
- nameif** BusyBox command, 566
 - NAND Flash**, 22-23
 - native compilation**, 30
 - nc** BusyBox command, 566
 - netstat** BusyBox command, 566
 - NFS (Network File System)**, 244-246
 - configuration file, 244
 - kernel configuration, 247
 - mounting workspace on target embedded system, 245-246
 - restarting, 141-142
 - root file system, 246-248
 - servers, configuring, 316-318
 - targets
 - example*, 246
 - root mount*, 318-321
 - website, 259

nice BusyBox command, 566
nm utility, 363-364
nmeter BusyBox command, 566
nohup BusyBox command, 566
northbridge chips, 42
nslookup BusyBox command, 566

O

objcopy utility, 360-361
objdump utility, 359
objects
 disassembled code, viewing, 359
 formats, converting, 360
 Image. *See* images
 piggy, 104
 symbols, viewing, 363-364
od BusyBox command, 566
On-The-Go (OTG) USB, 495
open() method, 217-219
open source legal insight website, 583
OpenEmbedded, 454
 benefits, 454
 BitBake, 456
 configuring, 462-463
 image recipes, 463
 metadata, 456
 classes, 461-462
 recipes, 456-459
 tasks, 460
 website, 137, 454, 464
openvpt BusyBox command, 566
optimized kernel code, debugging, 385-392
 code example, 385-386
 disassemble command, 387-389
 local variable output example, 391
 source file, 389-390
options. *See* parameters
OTG (On-The-Go) USB, 495
out-of-tree drivers, 223-224

P

packages, 32
parameters
 command-line, 115-116
 device drivers, 211-212
 initcall_debug, 127
 rdinit=, 155
parse BusyBox command, 566
partitions, 229
 file system relationship, 229
 formatting, 230-231

information, viewing, 229-230
MTD. *See* MTD, partitions
passwd BusyBox command, 566
patch BusyBox command, 567
performance, real time analysis, 478
 Ftrace, 478-479
interrupt off timing measurements, 483-484
preemption off measurements, 479-481
soft lockup detection, 484
 wakeup latency measurements, 481-483
persistent device naming, 541-542
pgrep BusyBox command, 567
pidof BusyBox command, 567
PIDs (process IDs), 250
piggy assembly file, 104
ping BusyBox command, 567
ping6 BusyBox command, 567
pipe_progress BusyBox command, 567
pkill BusyBox command, 567
platforms (hardware), 60-61
 device drivers, loading, 538-539
 specific kernel debugging, 381-382
popmaildir BusyBox command, 567
populating root file systems, 137
porting U-Boot bootloaders, 174
 board-specific initialization, 181-184
 build tree, configuring, 177-178
EP405 board, 175-176
makefile configuration targets, 176-177
processor initialization, 178-180
 summary, 184-185
Power Architecture processors, 44, 62
Power.org website, 62
poweroff BusyBox command, 567
PowerPC 64-bit architecture reference manual
 website, 62
PowerQUICC processors, 44
 PowerQUICC I processor, 45-46
 PowerQUICC II processor, 46-47
 PowerQUICC II Pro processor, 47
 PowerQUICC III processor, 48
PQ2FADS Flash mapping driver, 274-276
preemption. *See* kernel, preemption
prelink utility, 364
prepare_namespace() function, 151
printenv BusyBox command, 567
printf BusyBox command, 567
printk debugging, 407-409
printk log buffers, dumping, 417-419
/proc file system, 249-252
 common entries, 252
 debugging with maps entry, 251

- mount dependency, 249
 - original purpose, 249
 - process IDs, 250
 - virtual memory addresses, 251
 - website, 259
 - process IDs (PIDs)**, 250
 - processes**
 - bottom-half processing, 468
 - context, 28
 - init. *See* initialization
 - listing, 345
 - multiple, debugging, 435-438
 - real-time, creating, 477
 - user space. *See* user space, processes
 - virtual memory, 28-30
 - processors**
 - initializing, 178-180
 - integrated SOCs, 43
 - additional ARM*, 59
 - AMCC Power Architecture*, 50-53
 - ARM*, 55
 - Broadcom SiByte*, 54-55
 - Freescale ARM*, 58-59
 - Freescale Power Architecture*, 44-45
 - Freescale PowerQUICC*, 45-48
 - Freescale QorIQ*, 48-50
 - MIPS*, 53-55
 - TI ARM*, 56-57
 - stand-alone
 - companion chipsets*, 41-43
 - Freescale MPC7448*, 40-41
 - IBM 970FX*, 39
 - Intel Atom M*, 40
 - Intel Pentium M*, 39-40
 - overview*, 38
 - program dependencies**, 32
 - protocols**
 - BOOTP**
 - servers, configuring*, 313-316
 - U-Boot bootloader support*, 171
 - website*, 198
 - DHCP**
 - servers, configuring*, 313-316
 - U-Boot bootloader support*, 172-173
 - website*, 198
 - gdb remote serial protocol**, 382-385
 - TFTP**
 - servers, configuring*, 312-313
 - website*, 198
 - ps BusyBox command**, 567
 - ps macro**, 344-346
 - invoking, 395-396
 - output, 396-397
 - pscan** BusyBox command, 567
 - pseudo file systems**. *See* /proc file system; sysfs file system
 - pthread_create()** function, 438
 - pwd** BusyBox command, 567
- ## Q – R
- QorIQ processors**, 45-50
 - raidautorun** BusyBox command, 567
 - ramfs** file system, 255-256
 - rcs** initialization scripts, 299-300
 - rdate** BusyBox command, 567
 - rdev** BusyBox command, 567
 - rdinit=** parameter, 155
 - readahead** BusyBox command, 567
 - readelf** utility, 355-357
 - ELF file debug information, 357-359
 - section headers, 356
 - readlink** BusyBox command, 567
 - readprofile** BusyBox command, 567
 - real time**
 - hard, 467
 - kernel patch, 473-475
 - features*, 475-476
 - O(1) scheduler*, 476
 - preemption modes*, 474-475
 - real-time processes, creating*, 477
 - kernel performance analysis, 478
 - Ftrace*, 478-479
 - interrupt off timing measurements*, 483-484
 - preemption off measurements*, 479-481
 - soft lockup detection*, 484
 - wakeup latency measurements*, 481-483
 - kernel preemption, 469
 - challenges*, 469-471
 - checking*, 471-472
 - concurrency errors*, 470
 - critical sections, locking*, 470
 - latency sources*, 473
 - models*, 471-472
 - SMP*, 472
 - latency, 467-468
 - processes, creating, 477
 - scheduling, 467
 - soft, 466
 - realpath** BusyBox command, 567
 - reboot** BusyBox command, 567
 - recipes (OpenEmbedded metadata)**, 456-459
 - BitBake Hello World processing, 458-459
 - Hello World example, 457
 - images, 463

- Red Hat's New Journaling File System: ext3**
website, 259
- Redboot**
bootloaders, 197
partitions, 269-273
CFI support, 270
creating, 272
detecting, 270
Flash partitions, 269-271
new partition list, 272
power-up messages, 269
user documentation website, 288
- reformime BusyBox command**, 567
- refreshing SDRAM, 573
- Reiser4 File System website**, 259
- ReiserFS file system, 238
- release() method**, 217-219
- remote debugging**
file stripping, 426-427
gdbserver utility, 427-429
kernel, 382-385
running processes, connecting, 442-443
sample program ELF file debug information, 425-426
serial ports, 442
- renice BusyBox command**, 567
- requirements**
dependencies, 32
development, 13-14
distribution components, 97
hosts, 311-312
- reset BusyBox command**, 567
- resize BusyBox command**, 567
- resources**
binary tools, 365
Buildroot, 464
BusyBox, 304
cbrowser, 365
DDD, 365
device drivers, 226
dmalloc, 365
file systems, 259
GDB, 365, 444
kernel debugging, 422
Linux Kernel Development, 3rd Edition, 485
Linux news and developments, 583
mailing lists, 582
MTD, 288
open source legal insight, 583
OpenEmbedded, 464
Scratchbox, 464
SDRAM, 580
- source repositories, 582
udev, 548
USB, 519
- restorecon BusyBox command**, 567
- rm BusyBox command**, 567
- rmdir BusyBox command**, 567
- rmmod BusyBox command**, 567
- rmmod utility**, 215-216
- roles**
bootloaders, 160-161
DTBs in boot sequences, 187-188
- root file systems**
automated build tools, 137
defined, 132
directories, 134
embedded challenges, 136
FHS, 133, 226
layout, 133-134
minimal, 134-136
mounting, 18
NFS. *See* NFS
populating, 137
UBIFS as, 287
- root hubs**, 489
- route BusyBox command**, 567
- rpm BusyBox command**, 567
- rpm2cpio BusyBox command**, 567
- rtcwake BusyBox command**, 567
- rules (udev)**, 527-530
configuring, 533-535
cumulative, 534
distribution-specific attributes/actions, 534
event-driven, 535
example, 528
loading device drivers example, 528
MODALIAS field, 532-533
mouse device example, 529
storage location, 527
uevents, USB, 530-531
- run-parts BusyBox command**, 567
- runcon BusyBox command**, 567
- runlevel BusyBox command**, 568
- runlevels**, 141-142
- runsv BusyBox command**, 568
- runsvdir BusyBox command**, 568
- Rusty's Linux Kernel Page website, 226
- rx BusyBox command**, 568
- S**
- sb-menu utility**, 448
- SCCs (Serial Communication Controllers)**, 45
- scheduling real time**, 467

- Scratchbox**, 447
 cross-compilation targets, creating, 448-450
 environment, configuring, 449
 Hello World example, 449
 installing, 447-448
 menuconfig, 449
 remote shell feature, 450
 website, 449, 464
- script BusyBox command**, 568
- scripts**
 linker command, 163
 rcs initialization, 299-300
 startup, 144-146
 uImage target wrapper, 185
- sd-mod driver**, adding, 509
- SDRAM (Synchronous Dynamic Random Access Memory)**, 572
 clocking, 574-575
 controllers, configuring, 575-579
 - memory bank control register*, 578
 - timing requirements*, 578-579
 - U-Boot sdram_init() function*, 576-577
- operation basics, 572-573
 refresh, 573
 resources, 580
- sdram_init() function**, 576-577
- sed BusyBox command**, 568
- selinuxenabled BusyBox command**, 568
- seq BusyBox command**, 568
- Serial Communication Controllers (SCCs)**, 45
- Serial Management Controllers (SMCs)**, 45
- serial ports**
 KGDB, 372
 remote debugging, 442
 sharing console with KGDB, 377-379
- servers**
 BOOTP, 313-316
 DHCP, 313-316
 NFS
 - configuring*, 316-318
 - target root mount*, 318-321
 TFTP, 312-313
- Service Availability Forum**, 7
- services**
 MTD, enabling, 263-265
 NFS, restarting, 141-142
- sestatus BusyBox command**, 568
- search BusyBox command**, 568
- setconsole BusyBox command**, 568
- setenforce BusyBox command**, 568
- setfont BusyBox command**, 568
- setkeycodes BusyBox command**, 568
- setlogcons BusyBox command**, 568
- setsebool BusyBox command**, 568
- setsid BusyBox command**, 568
- setuidgid BusyBox command**, 568
- setup_arch() function**, 114
- setup macro**, command-line processing, 116-118
 - code listing, 119-121
 - console setup code, 117
 - family definitions, 118
- sh BusyBox command**, 568
- shared libraries**
 debugging with, 429
 - events*, 431-434
 - initial target memory segment mapping*, 430-431
 - invoking ldd command*, 432-433
 - locations*, 433
 - </proc/pid>/maps memory segments*, 434
 - requirements*, 430
 finding, 433
 viewing, 432
- showkey BusyBox command**, 568
- shutdown command**, 156
 shutting down, 156
- slattach BusyBox command**, 568
- sleep BusyBox command**, 568
- SMCs (Serial Management Controllers)**, 45
- SMP (Symmetric multiprocessing)**, 472
- SOCs (system on chips)**, 43
 - AMCC Power Architecture, 50-53
 - ARM, 55
 - additional companies*, 59
 - Freescale, 58-59
 - TI, 56-57
 - Broadcom SiByte, 54-55
 - Freescale Power Architecture, 44-45
 - PowerQUICC I*, 45-46
 - PowerQUICC II*, 46-47
 - PowerQUICC II Pro*, 47
 - PowerQUICC III*, 48
 - Freescale QorIQ, 48-50
 - MIPS, 53-55
- soft lockup detection**, 484
- soft real time**, 466
- softlimit BusyBox command**, 568
- sort BusyBox command**, 568
- source repositories**, 67-68, 582
- southbridge chips**, 42
- split BusyBox command**, 568
- stack frames (GDB)**, 330
- stand-alone processors**
 companion chipsets, 41-43
 Freescale MPC7448, 40-41

IBM 970FX, 39
Intel, 39-40
overview, 38

standards
carrier-grade, 6
Linux Foundation, 6-7
LSB, 5
Moblin, 7
Service Availability Forum, 7

start_kernel() function, 114

startup
scripts, 144-146
tasks, 11

stat BusyBox command, 568

stop-on-solib-event command, 432

storage
bootloaders, 162
cross-development environments, 30-31
execution contexts, 26
memory. *See* memory
MMUs, 26
U-Boot bootloader support, 173
udev rules, 527

strace utility, 337
command-line options, 341-342
profiling, 341
web demo application example, 337-340

strings BusyBox command, 568

strings utility, 362

strip utility, 361

stty BusyBox command, 568

subdirectories (kernel), 77-78

su BusyBox command, 568

subsystems (USB), 508
CDC (Communications Device Class) drivers, 512-515
HID (Human Input Device), 511-512
initializing, 122-124
mass storage, 508-511
device node, creating, 510
mounting, 510
partition, mounting, 510
SCSI support, 508
sd-mod driver, adding, 509
usb_storage driver, 508
usb-storage module, 509

sulogin BusyBox command, 568

sum BusyBox command, 568

sv BusyBox command, 568

svlogd BusyBox command, 568

swapoff BusyBox command, 568

swapon BusyBox command, 568

switch_root BusyBox command, 569

symlinks, 300-302

Symmetric multiprocessing (SMP), 472

sync BusyBox command, 569

Synchronous Dynamic Random Access Memory. *See* SDRAM

syntax
command-line parameters, 116
flat device tree, 192

sysctl BusyBox command, 569

sysfs file system, 252-255
browsing, 253
directory structure, 252-253
systool output example, 253-255
USB devices, 500-502

syslogd BusyBox command, 569

system initialization, 297-299

system.map file, 70

system on chips. *See* SOCs

System V Init, 140
inittab file, 143-144
runlevels, 141-142
startup scripts, 144-145
web server startup script example, 145-146
website, 157

systool utility, 253

T

tac BusyBox command, 569

tail BusyBox command, 569

tar BusyBox command, 569

target boards
BOOTP/DHCP servers, configuring, 313-316
NFS
root mount, 318-321
servers, configuring, 316-318
starting, 15-16
TFTP servers, configuring, 312-313

targets
architecture-specific, 193
bzImage, 83
cross-compilation, 448-450
cross-development environments, 306
debugging, 424
DHCP identification, 172-173
distributions, 33
makefile, 83-89
memory breakpoints, 383
U-Boot makefiles, 176-177
zImage, 83

task_struct_show macro, 398-399
tasks
 ksoftirqd, 476
 OpenEmbedded metadata, 460
 startup, 11
taskset BusyBox command, 569
tc BusyBox command, 569
tcpsvd BusyBox command, 569
tee BusyBox command, 569
telnet BusyBox command, 569
telnetd BusyBox command, 569
test BusyBox command, 569
Texas Instruments (TI) ARM processors, 56-57
TFTP (Trivial File transfer Protocol), 171
 servers, configuring, 312-313
 website, 198, 323
tftp BusyBox command, 569
tftp command, 17
tftpd BusyBox command, 569
time BusyBox command, 569
TI (Texas Instruments) ARM processors, 56-57
tmpfs file system, 256
Tool Interface Standard (TIS) Executable and Linking Format, 98
tools. *See* utilities
top BusyBox command, 569
top-level kernel source directory, 69
topologies (USB)
 logical, 490-491
 physical, 488-490
top utility, 346-348
Torvalds, Linus, 6, 64
touch BusyBox command, 569
tr BusyBox command, 569
traceroute BusyBox command, 569
tracing and profiling tools
 dmalloc, 350-353
 kernel oops, 353-355
 ltrace, 343-344
 mtrace, 348-349
 ps, 344-346
 strace, 337
 command-line options, 341-342
 profiling, 341
 web demo application example, 337-340
top, 346-348
Trivial File Transfer Protocol. *See* TFTP
troubleshooting. *See* debugging
TRUE BusyBox command, 569
tty BusyBox command, 569
tysize BusyBox command, 569
Tundra chip, 42
tune2fs BusyBox command, 569

U

U-Boot bootloader
 booting from disks, 174
 commands, 169-170, 558-560
 configuring, 167-169
 debugging with JTAG probes, 414
 DTBs on boot sequence, 187-188
 finding, 166
 image formats, 185-186
 KGDB enabled booting, 373-374
 network support
 BOOTP client/server, 171
 DHCP target, 172-173
 Ethernet interfaces, 170
NFS root mount example, 320-321
porting, 174
 board-specific initialization, 181-184
 build tree, configuring, 177-178
 EP405 board, 175-176
 makefile configuration targets, 176-177
 processor initialization, 178-180
 summary, 184-185
reference website, 198
storage subsystems, 173
website, 166
ubiformat command, 286
UBIFS (Unsorted Block Image File System), 284
 as root file system, 287
 configuring, 284
 images, building, 284-287
ubinize configuration file, 285
udev
 busybox mdev, 545-547
 customizing, 540
 devices
 discovery, 523-524
 nodes, creating, 525-526
initial system setup, 535
 coldplug processing, 537-538
 default static device nodes, 536
 startup script, 535-536
persistent device naming, 541-542
 /dev directory contents, 541
 helper utilities, 542-545
platform device drivers, loading, 538-539
resources, 548
rules, 527-530
 configuring, 533-535
 cumulative, 534
 distribution-specific attributes/actions, 534
 event-driven, 535

example, 528
loading device drivers example, 528
MODALIAS field, 532-533
mouse device example, 529
storage location, 527
uevents emitted on USB mouse insertion,
 530-531
uevents for USB interface 1-1:1.0, 531
uevents on four-port hub insertion, 525
 USB automounting, 540-541
“Udev: A Userspace Implementation of devs”
 website, 548
udevadm command, 523-524
udevadm info command, 543-544
udhcpc BusyBox command, 569
udhcpd BusyBox command, 569
udpscd BusyBox command, 569
uevents, 523-524
 device discovery, 523
 four-port hub insertion, 525
 USB, 530-531
umount BusyBox command, 569
uname BusyBox command, 569
uncompress BusyBox command, 569
unexpand BusyBox command, 569
uniq BusyBox command, 569
Universal Serial Bus. *See* **USB**
unix2dos BusyBox command, 569
UNIX line-continuation character (\), 119
unlzma BusyBox command, 569
Unsorted Block Image File System. *See* **UBIFS**
unzip BusyBox command, 569
uptime BusyBox command, 569
USB (Universal Serial Bus), 488
 automounting, 540-541
 bus topology, 507
 cable assemblies, 494
 configuring, 495-497
 core makefile, 496-497
 descriptors, 491
 volume of options, 495
 connectors, 492-493
 debugging, 516
 device driver support, 518
 Ethernet dongle insertion debug output example, 516
 platform-specific options, 517
 usbmon utility, 517-518
 device descriptors, 490
 EHCI, 498
 endpoints, 491
 Ethernet connectivity, 513-515
 file system, 502-504
Freescale Semiconductor iMX31 Applications Processor example. *See* **Freescale processors, Semiconductor iMX31 Applications Processor USB example**
 initializing, 499-500
 host controllers, 498-499
 usbcore module, loading, 497
 interface descriptors, 491
 modes, 494-495
 resources, 519
 revisions, 491
 subsystems, 508
 CDC drivers, 512-514
 HID, 511-512
 mass storage, 508-511
 sysfs file system, 500-502
 tools
 lsusb utility, 507-508
 USBFS, 502-504
 usbview utility, 504-507
 topologies
 logical, 490-491
 physical, 488-490
 usbfs, viewing, 504
usb_id utility, 542-543
usb_storage driver, 508-509
USB-USB direct networking example, 513
usbcore module, loading, 497
USBFS (USB File System), 502-504
usbmon utility, 517-518
usbview utility, 504-507
used macro, 119
user space
 context, 26
 processes
 dependencies, resolving, 139-140
 first user space program, 139
 init, 19, 140
 initial RAM disk method. *See* **initrd initramfs**, 153-155
usleep BusyBox command, 570
utilities. *See also* **commands**
 addr2line, 361
 automated root file system builds, 137
 bitbake, 137
 buildroot, 137
 busybox, 135
 cbrowser, 335-336, 365
 cross, 33

- cross-strip, 426-427
 DDD, 333-335, 365
 depmod, 214-215
 dmalloc, 350-353
 libraries, generating, 350
 log output example, 351-352
 requirements, 350
 resources, 365
 fdisk, 229-230
 Ftrace
 interrupt off timing measurements, 484
 kernel performance analysis, 478-479
 preemption off measurements, 479-481
 wakeup measurements, 481-483
 GDB, 326
 backtrace command, 330
 core dumps, 327-329
 debug sessions, 331-333
 invoking, 329-331
 resources, 365
 stack frames, 330
 gdbserver, 427-429
 git, 68
 insmod, 212
 kernel oops, 353-355
 ldd, 139, 362-363
 Library Optimizer, 136
 lsmod, 213
 lsusb, 507-508
 ltrace, 343-344
 Magic SysReq key, 409-410
 mkfs.ext2, 230-231
 mkiage, 185
 modinfo, 216
 modprobe, 213-214
 MTD, 279-283
 flash_erase, 280
 flashcp, 280
 JFFS2 as root file system, 283
 JFFS2 images, copying, 282
 kernel MTD partition list, 279
 MTD Flash partition, mounting, 280
 mtrace, 348-349
 nm, 363-364
 objcopy, 360-361
 objdump, 359
 prelink, 364
 printk, 407-409
 ps, 344-346
 readelf, 355-357
 rmmod, 215-216
 sb-menu, 448
 strace, 337
 command-line options, 341-342
 profiling, 341
 web demo application example, 337-340
 strings, 362
 strip, 361
 systool, 253
 top, 346-348
 udev helper, 542-545
 USB
 lsusb utility, 507-508
 USBFS, 502-504
 usb_id, 542-543
 usbmon, 517-518
 usbview, 504-507
 uudecode BusyBox command, 570
 uuencode BusyBox command, 570
- V**
- vconfig BusyBox command, 570
 versions (kernel), 66-67
 vi BusyBox command, 570
 viewing
 disassembled object code, 359
 kernel initialization details, 127
 .modinfo sections, 539
 shared libraries, 432
 virtual memory, 26-30
 vlock BusyBox command, 570
 vmlinux file, 70-72
 image components, 73-76
 listing, 72-73
- W**
- wakeup measurements, 481-483
 watch BusyBox command, 570
 watchdog BusyBox command, 570
 wc BusyBox command, 570
 wear leveling, 240
 web demo application, 337-340
 websites
 A Non-Technical Look Inside the EXT2 File System, 259
 Abatron, 410
 ARM Technologies, 56, 59
 BeagleBoard, 62
 binary tool resources, 365
 Booting Linux: The History and the Future, 157
 BOOTP, 198, 323
 buildroot utility, 137, 464

- BusyBox, 304
cbrowser utility, 365
Common Flash Memory Interface Specification, 288
CompactPCI, 60
cramfs project README, 259
DDD resources, 365
Debugging with GDB, 422
“Design and Implementation of the Second Extended Filesystem,” 259
device trees, 199
DHCP protocol, 198
dmalloc utility, 365
DRAM, 198
dtc compiler, 189
Dynamic Host Configuration, 323
Eclipse Project, 365
EHCI, 519
“File System Performance: The Solaris OS, UFS, Linux ext3, and Reiser FS,” 259
Filesystem Hierarchy Standard, 226
flat device trees
 references, 199
 syntax, 192
Freescale Semiconductor, 62
FSH, 157
Garzik, Jeff’s git utility, 68
GCC, 323
GDB: The GNU Project Debugger, 444
GDB resources, 365
GNU
 Compiler Collection documentation, 130
 linker, 130-198
 Press, 422
GPL, 550
GRUB, 199
Intel, 62
JFFS: The Journaling Flash File System, 259
Kbuild, 98
kernel, 65
 debugging resources, 422
 HOWTO, 98
KGDB, 422
Library Optimizer Tool, 136
Lilo, 199
Linux
 Documentation Project, 96, 157
 Foundation, 8
 news and developments, 583
 Standard Base Project, 8
LinuxDevices.com, 3
mailing lists, 582
MIPS architecture, 67
Moblin, 7
MTD resources, 288
NFS, 259
open source legal insight, 583
OpenEmbedded, 137, 454, 464
Power Architecture, 62
Power.org, 62
PowerPC 64-bit architecture reference manual, 62
/proc file system, 259
“Red Hat’s New Journaling File System: ext3,” 259
Redboot user documentation, 288
Reiser4 File System, 259
Rusty’s Linux Kernel Page, 226
Scratchbox, 449, 464
SDRAM resources, 580
Service Availability Forum, 7
source repositories, 582
System V init, 157
TFTP protocol, 198, 323
Tool Interface Standard (TIS) Executable and, 98
U-Boot, 166, 198
udev, 548
 USB resources, 519
 wget BusyBox command, 570
 which BusyBox command, 570
 who BusyBox command, 570
 whoami BusyBox command, 570
 wrapper script, 185
 “Writing udev Rules” website, 548
- ## X-Z
- xargs BusyBox command, 570
yes BusyBox command, 570
zcat BusyBox command, 570
zcip BusyBox command, 570
zImage targets, 83