PRENTICE HALL

PRENTICE HALL OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Embedded Linux Primer

Second Edition

A Practical Real-World Approach

Christopher Hallinan

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: informit.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

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

Computers are everywhere.

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

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

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

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

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

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

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

Arnold Robbins
Series Editor

Foreword for the Second Edition

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

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

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

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

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

Arnold Robbins
Series Editor

Preface

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

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

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

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

Practical Advice for the Practicing Embedded Developer

This book describes my view of what an embedded engineer needs to know to get up to speed fast in an embedded Linux environment. Instead of focusing on Linux kernel internals, the kernel chapters in this book focus on the project nature of the kernel and leave the internals to the other excellent texts on the subject. You will learn the organization and layout of the kernel source tree. You will discover the binary components that make up a kernel image, how they are loaded, and what purpose they serve on an embedded system.

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

Intended Audience

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

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

What This Book Is Not

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

This is also not a book about the Linux kernel or kernel internals. In this book, you won't learn about the intricacies of the Memory Management Unit (MMU) used to implement Linux's virtual memory-management policies and procedures; there are already several good books on this subject. You are encouraged to take advantage of the "Suggestions for Additional Reading" sections found at the end of every chapter.

Conventions Used

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

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

How This Book Is Organized

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 19, "udev," takes the mystery out of this powerful system configuration utility. We examine udev's default behavior as a foundation for understanding how to customize it. Several real-world examples are presented. For BusyBox users, we examine BusyBox's mdev utility.

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

Follow Along

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

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

GPL Copyright Notice

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

Appendix A contains the text of the GNU Public License.

Chapter 7

Bootloaders

In This Chapter

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 0xFFFFFFFC :
  {
    *(.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
<...>
#ifdef 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 tftpserver), `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 "condif_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.

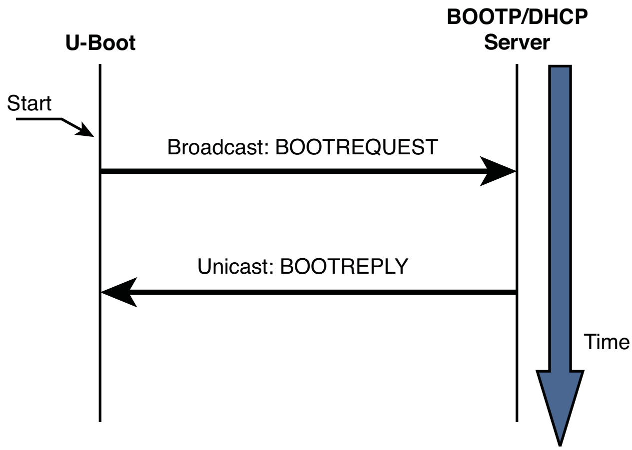

FIGURE 7-1 BOOTP client/server handshake

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 4*xx* 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=0xFFFC0000 -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_xl30.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 4*xx* architecture.

LISTING 7-8 U-Boot 4xx Startup Code

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

[3] This 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 4*xx* 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@l
    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@l
    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@l
    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@l
    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 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 (arch/powerpc/boot/zImage.*)
  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) ~/bin/installkernel or
                    (distribution) /sbin/installkernel or
                    install to $(INSTALL_PATH) and run lilo
  *_defconfig     - Select default config from arch/powerpc/configs
```

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. YAMON[7] 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

C

D

J

K

L

M

N

Q – R

S

T

U

V

W

X–Z