

Chris Simmonds

Mastering Embedded Linux Programming

Second Edition

Unleash the full potential of Embedded Linux



Packt>

```
< html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"  
"http://www.w3.org/TR/REC-html40/loose.dtd">
```

Mastering Embedded Linux Programming

Second Edition

Unleash the full potential of Embedded Linux

Chris Simmonds



BIRMINGHAM - MUMBAI

```
< html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"  
"http://www.w3.org/TR/REC-html40/loose.dtd">
```

Mastering Embedded Linux Programming

Second Edition

Copyright © 2017 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book. Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2015

Second edition: June 2017

Production reference: 1280617

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78728-328-2

www.packtpub.com

Credits

Author Chris Simmonds	Copy Editors Madhusudan Uchil Stuti Shrivastava
Reviewers Daiane Angolini Otavio Salvador Alex Tereschenko	Project Coordinator Virginia Dias
Commissioning Editor Kartikey Pandey	Proofreader Safis Editing

Acquisition Editor Prateek Bharadwaj	Indexer Rekha Nair
Content Development Editor Sharon Raj	Graphics Kirk D'Penha
Technical Editor Vishal Kamal Mewada	Production Coordinator Melwyn Dsa

About the Author

Chris Simmonds is a software consultant and trainer living in southern England. He has almost two decades of experience in designing and building open-source embedded systems. He is the founder and chief consultant at 2net Ltd, which provides professional training and mentoring services in embedded Linux, Linux device drivers, and Android platform development. He has trained engineers at many of the biggest companies in the embedded world, including ARM, Qualcomm, Intel, Ericsson, and General Dynamics. He is a frequent presenter at open source and embedded conferences, including the Embedded Linux Conference and Embedded World. You can see some of his work on the Inner Penguin blog at www.2net.co.uk.

I would like to thank Shirley Simmonds for being so supportive during the long hours that I was shut in my home office researching and writing this book. I would also like to thank all the people who have helped me with the research of the technical aspects of this book, whether they realized that is what they were doing or not. In particular, I would like to mention Klaas van Gend, Thomas Petazzoni, and Ralph Nguyen for their help and advice. Lastly, I would like to thank Sharon Raj, Vishal Mewada, and the team at Packt Publishing for keeping me on track and bringing the book to fruition.

About the Reviewers

Daiane Angolini has been working with embedded Linux since 2008. She has been working as an application engineer at NXP, acting on internal development, porting custom applications from Android, and on-customer support for i.MX architectures in areas such as Linux kernel, u-boot, Android, Yocto Project, and user-space applications. However, it was on the Yocto Project that she found her place. She has coauthored the books *Embedded Linux Development with Yocto Project* and *Heading for the Yocto Project*, and learned a lot in the process.

Otávio Salvador loves technology and started his free software activities in 1999. In 2002, he founded O.S. Systems, a company focused on embedded system development services and consultancy worldwide, creating and maintaining customized BSPs, and helping companies with their product's development challenges. This resulted in him joining the OpenEmbedded community in 2008, when he became an active contributor to the OpenEmbedded project. He has coauthored the books *Embedded Linux Development with Yocto Project* and *Heading for the Yocto Project*.

Alex Tereschenko is an embedded systems engineer by day, and an avid maker by night, who is convinced that computers can do a lot of good for people when they are interfaced with real-world objects, as opposed to just crunching data in a dusty corner. That's what's driving him in his projects, and this is why embedded systems and the Internet of Things are the topics he enjoys the most.

www.PacktPub.com

For support files and downloads related to your book, please visit www.PacktPub.com. Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details. At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1787283283>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

Table of Contents

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
 - Downloading the example code
 - Downloading the color images of this book
- Errata
- Piracy
- Questions

1. Starting Out

- Selecting the right operating system
- The players
- Project life cycle
 - The four elements of embedded Linux
- Open source
 - Licenses
- Hardware for embedded Linux
- Hardware used in this book
 - The BeagleBone Black
 - QEMU
- Software used in this book
- Summary

2. Learning About Toolchains

- Introducing toolchains
 - Types of toolchains
 - CPU architectures
 - Choosing the C library
- Finding a toolchain
- Building a toolchain using crosstool-NG
 - Installing crosstool-NG
 - Building a toolchain for BeagleBone Black
 - Building a toolchain for QEMU
- Anatomy of a toolchain

- Finding out about your cross compiler
- The sysroot, library, and header files
- Other tools in the toolchain
- Looking at the components of the C library
- Linking with libraries – static and dynamic linking
 - Static libraries
 - Shared libraries
 - Understanding shared library version numbers
- The art of cross compiling
 - Simple makefiles
 - Autotools
 - An example: SQLite
 - Package configuration
 - Problems with cross compiling
- Summary

3. All About Bootloaders

- What does a bootloader do?
- The boot sequence
 - Phase 1 – ROM code
 - Phase 2 – secondary program loader
 - Phase 3 – TPL
- Booting with UEFI firmware
- Moving from bootloader to kernel
- Introducing device trees
 - Device tree basics
 - The reg property
 - Labels and interrupts
 - Device tree include files
 - Compiling a device tree
- Choosing a bootloader
- U-Boot
 - Building U-Boot
 - Installing U-Boot
 - Using U-Boot
 - Environment variables
 - Boot image format
 - Loading images
 - Booting Linux
 - Automating the boot with U-Boot scripts
 - Porting U-Boot to a new board

- Board-specific files
- Configuring header files
- Building and testing
- Falcon mode
- Barebox
 - Getting barebox
 - Building barebox
 - Using barebox
- Summary

4. Configuring and Building the Kernel

- What does the kernel do?
- Choosing a kernel
 - Kernel development cycle
 - Stable and long term support releases
 - Vendor support
 - Licensing
- Building the kernel
 - Getting the source
 - Understanding kernel configuration – Kconfig
 - Using LOCALVERSION to identify your kernel
 - Kernel modules
- Compiling – Kbuild
 - Finding out which kernel target to build
 - Build artifacts
 - Compiling device trees
 - Compiling modules
 - Cleaning kernel sources
 - Building a kernel for the BeagleBone Black
 - Building a kernel for QEMU
- Booting the kernel
 - Booting the BeagleBone Black
 - Booting QEMU
 - Kernel panic
 - Early user space
 - Kernel messages
 - Kernel command line
- Porting Linux to a new board
 - A new device tree
 - Setting the board compatible property

Additional reading

Summary

5. Building a Root Filesystem

What should be in the root filesystem?

The directory layout

The staging directory

POSIX file access permissions

File ownership permissions in the staging directory

Programs for the root filesystem

The init program

Shell

Utilities

BusyBox to the rescue!

Building BusyBox

ToyBox – an alternative to BusyBox

Libraries for the root filesystem

Reducing the size by stripping

Device nodes

The proc and sysfs filesystems

Mounting filesystems

Kernel modules

Transferring the root filesystem to the target

Creating a boot initramfs

Standalone initramfs

Booting the initramfs

Booting with QEMU

Booting the BeagleBone Black

Mounting proc

Building an initramfs into the kernel image

Building an initramfs using a device table

The old initrd format

The init program

Starting a daemon process

Configuring user accounts

Adding user accounts to the root filesystem

A better way of managing device nodes

An example using devtmpfs

An example using mdev

Are static device nodes so bad after all?

Configuring the network

- Network components for glibc
- Creating filesystem images with device tables
- Booting the BeagleBone Black
- Mounting the root filesystem using NFS
- Testing with QEMU
- Testing with the BeagleBone Black
- Problems with file permissions
- Using TFTP to load the kernel
- Additional reading
- Summary

6. Selecting a Build System

- Build systems
- Package formats and package managers
- Buildroot
 - Background
 - Stable releases and long-term support
 - Installing
 - Configuring
 - Running
 - Creating a custom BSP
 - U-Boot
 - Linux
 - Build
 - Adding your own code
 - Overlays
 - Adding a package
- License compliance
- The Yocto Project
 - Background
 - Stable releases and supports
 - Installing the Yocto Project
 - Configuring
 - Building
 - Running the QEMU target
 - Layers
 - BitBake and recipes
 - Customizing images via local.conf
 - Writing an image recipe
 - Creating an SDK

The license audit

Further reading

Summary

7. Creating a Storage Strategy

Storage options

NOR flash

NAND flash

Managed flash

MultiMediaCard and Secure Digital cards

eMMC

Other types of managed flash

Accessing flash memory from the bootloader

U-Boot and NOR flash

U-Boot and NAND flash

U-Boot and MMC, SD, and eMMC

Accessing flash memory from Linux

Memory technology devices

MTD partitions

MTD device drivers

The MTD character device, mtd

The MTD block device, mtdblock

Logging kernel oops to MTD

Simulating NAND memory

The MMC block driver

Filesystems for flash memory

Flash translation layers

Filesystems for NOR and NAND flash memory

JFFS2

Summary nodes

Clean markers

Creating a JFFS2 filesystem

YAFFS2

Creating a YAFFS2 filesystem

UBI and UBIFS

UBI

UBIFS

Filesystems for managed flash

Flashbench

Discard and TRIM

Ext4

F2FS

FAT16/32

Read-only compressed filesystems

squashfs

Temporary filesystems

Making the root filesystem read-only

Filesystem choices

Further reading

Summary

8. Updating Software in the Field

What to update?

Bootloader

Kernel

Root filesystem

System applications

Device-specific data

Components that need to be updated

The basics of software update

Making updates robust

Making updates fail-safe

Making updates secure

Types of update mechanism

Symmetric image update

Asymmetric image update

Atomic file updates

OTA updates

Using Mender for local updates

Building the Mender client

Installing an update

Using Mender for OTA updates

Summary

9. Interfacing with Device Drivers

The role of device drivers

Character devices

Block devices

Network devices

Finding out about drivers at runtime

Getting information from sysfs

The devices: /sys/devices

- The drivers: /sys/class
- The block drivers: /sys/block
- Finding the right device driver
- Device drivers in user space
 - GPIO
 - Handling interrupts from GPIO
 - LEDs
 - I2C
 - Serial Peripheral Interface (SPI)
- Writing a kernel device driver
 - Designing a character driver interface
 - The anatomy of a device driver
 - Compiling kernel modules
 - Loading kernel modules
- Discovering the hardware configuration
 - Device trees
 - The platform data
 - Linking hardware with device drivers
- Additional reading
- Summary

10. Starting Up – The init Program

- After the kernel has booted
- Introducing the init programs
- BusyBox init
 - Buildroot init scripts
- System V init
 - inittab
 - The init.d scripts
 - Adding a new daemon
 - Starting and stopping services
- systemd
 - Building systemd with the Yocto Project and Buildroot
 - Introducing targets, services, and units
 - Units
 - Services
 - Targets
 - How systemd boots the system
 - Adding your own service
 - Adding a watchdog

Implications for embedded Linux

Further reading

Summary

11. Managing Power

Measuring power usage

Scaling the clock frequency

The CPUFreq driver

Using CPUFreq

Selecting the best idle state

The CPUIidle driver

Tickless operation

Powering down peripherals

Putting the system to sleep

Power states

Wakeup events

Timed wakeups from the real-time clock

Further reading

Summary

12. Learning About Processes and Threads

Process or thread?

Processes

Creating a new process

Terminating a process

Running a different program

Daemons

Inter-process communication

Message-based IPC

Unix (or local) sockets

FIFOs and named pipes

POSIX message queues

Summary of message-based IPC

Shared memory-based IPC

POSIX shared memory

Threads

Creating a new thread

Terminating a thread

Compiling a program with threads

Inter-thread communication

Mutual exclusion

- Changing conditions
- Partitioning the problem
- Scheduling
 - Fairness versus determinism
 - Time-shared policies
 - Niceness
 - Real-time policies
 - Choosing a policy
 - Choosing a real-time priority
- Further reading
- Summary

13. Managing Memory

- Virtual memory basics
- Kernel space memory layout
 - How much memory does the kernel use?
- User space memory layout
- The process memory map
- Swapping
 - Swapping to compressed memory (zram)
- Mapping memory with mmap
 - Using mmap to allocate private memory
 - Using mmap to share memory
 - Using mmap to access device memory
- How much memory does my application use?
- Per-process memory usage
 - Using top and ps
 - Using smem
 - Other tools to consider
- Identifying memory leaks
 - mtrace
 - Valgrind
- Running out of memory
- Further reading
- Summary

14. Debugging with GDB

- The GNU debugger
- Preparing to debug
- Debugging applications
 - Remote debugging using gdbserver

Setting up the Yocto Project for remote debugging

Setting up Buildroot for remote debugging

Starting to debug

Connecting GDB and gdbserver

Setting the sysroot

GDB command files

Overview of GDB commands

Breakpoints

Running and stepping

Getting information

Running to a breakpoint

Native debugging

The Yocto Project

Buildroot

Just-in-time debugging

Debugging forks and threads

Core files

Using GDB to look at core files

GDB user interfaces

Terminal user interface

Data display debugger

Eclipse

Debugging kernel code

Debugging kernel code with kgdb

A sample debug session

Debugging early code

Debugging modules

Debugging kernel code with kdb

Looking at an Oops

Preserving the Oops

Further reading

Summary

15. Profiling and Tracing

The observer effect

Symbol tables and compile flags

Beginning to profile

Profiling with top

Poor man's profiler

Introducing perf

- Configuring the kernel for perf
- Building perf with the Yocto Project
- Building perf with Buildroot
- Profiling with perf
- Call graphs
- perf annotate
- Other profilers – OProfile and gprof
- Tracing events
- Introducing Ftrace
 - Preparing to use Ftrace
 - Using Ftrace
 - Dynamic Ftrace and trace filters
 - Trace events
- Using LTTng
 - LTTng and the Yocto Project
 - LTTng and Buildroot
 - Using LTTng for kernel tracing
- Using Valgrind
 - Callgrind
 - Helgrind
- Using strace
- Summary

16. Real-Time Programming

- What is real time?
- Identifying sources of non-determinism
- Understanding scheduling latency
- Kernel preemption
- The real-time Linux kernel (PREEMPT_RT)
 - Threaded interrupt handlers
- Preemptible kernel locks
 - Getting the PREEMPT_RT patches
 - The Yocto Project and PREEMPT_RT
- High-resolution timers
- Avoiding page faults
- Interrupt shielding
- Measuring scheduling latencies
 - cyclicttest
 - Using Ftrace
 - Combining cyclicttest and Ftrace

[Further reading](#)

[Summary](#)

Preface

Linux has been the mainstay of embedded computing for many years. And yet, there are remarkably few books that cover the topic as a whole: this book is intended to fill that gap. The term embedded Linux is not well-defined, and can be applied to the operating system inside a wide range of devices ranging from thermostats to Wi-Fi routers to industrial control units. However, they are all built on the same basic open source software. Those are the technologies that I describe in this book, based on my experience as an engineer and the materials I have developed for my training courses.

Technology does not stand still. The industry based around embedded computing is just as susceptible to Moore's law as mainstream computing. The exponential growth that this implies has meant that a surprisingly large number of things have changed since the first edition of this book was published. This second edition is fully revised to use the latest versions of the major open source components, which include Linux 4.9, Yocto Project 2.2 Morty, and Buildroot 2017.02. Since it is clear that embedded Linux will play an important part in the *Internet of Things*, there is a new chapter on the updating of devices in the field, including Over the Air updates. Another trend is the quest to reduce power consumption, both to extend the battery life of mobile devices and to reduce energy costs. The chapter on power management shows how this is done.

Mastering Embedded Linux Programming covers the topics in roughly the order that you will encounter them in a real-life project. The first 6 chapters are concerned with the early stages of the project, covering basics such as selecting the toolchain, the bootloader, and the kernel. At the conclusion of this this section, I introduce the idea of using an embedded build tool, using Buildroot and the Yocto Project as examples.

The middle part of the book, chapters 7 through to 13, will help you in the implementation phase of the project. It covers the topics of filesystems, the init program, multithreaded programming, software update, and power management. The third section, chapters 14 and 15, show you how to make effective use of the many debug and profiling tools that Linux has to offer in order to detect

problems and identify bottlenecks. The final chapter brings together several threads to explain how Linux can be used in real-time applications.

Each chapter introduces a major area of embedded Linux. It describes the background so that you can learn the general principles, but it also includes detailed worked examples that illustrate each of these areas. You can treat this as a book of theory, or a book of examples. It works best if you do both: understand the theory and try it out in real life.

What this book covers

[Chapter 1](#), *Starting Out*, sets the scene by describing the embedded Linux ecosystem and the choices available to you as you start your project.

[Chapter 2](#), *Learning About Toolchains*, describes the components of a toolchain and shows you how to create a toolchain for cross-compiling code for the target board. It describes where to get a toolchain and provides details on how to build one from the source code.

[Chapter 3](#), *All About Bootloaders*, explains the role of the bootloader in loading the Linux kernel into memory, and uses U-Boot and Bareboot as examples. It also introduces device trees as the mechanism used to encode the details of hardware in almost all embedded Linux systems.

[Chapter 4](#), *Configuring and Building the Kernel*, provides information on how to select a Linux kernel for an embedded system and configure it for the hardware within the device. It also covers how to port Linux to the new hardware.

[Chapter 5](#), *Building a Root Filesystem*, introduces the ideas behind the user space part of an embedded Linux implementation by means of a step-by-step guide on how to configure a root filesystem.

[Chapter 6](#), *Selecting a Build System*, covers two commonly used embedded Linux build systems, Buildroot and Yocto Project, which automate the steps described in the previous four chapters.

[Chapter 7](#), *Creating a Storage Strategy*, discusses the challenges created by managing flash memory, including raw flash chips and embedded MMC (eMMC) packages. It describes the filesystems that are applicable to each type of technology.

[Chapter 8](#), *Updating Software in the Field*, examines various ways of updating the software after the device has been deployed, and includes fully managed Over the Air (OTA) updates. The key topics under discussion are reliability and security.

[Chapter 9](#), *Interfacing with Device Drivers*, describes how kernel device drivers interact with the hardware with worked examples of a simple driver. It also describes the various ways of calling device drivers from the user space.

[Chapter 10](#), *Starting Up – The Init Program*, shows how the first user space program--init--starts the rest of the system. It describes the three versions of the init program, each suitable for a different group of embedded systems, ranging from the simplicity of the BusyBox init, through System V init, to the current state-of-the-art, systemd.

[Chapter 11](#), *Managing Power*, considers the various ways that Linux can be tuned to reduce power consumption, including Dynamic Frequency and Voltage scaling, selecting deeper idle states, and system suspend. The aim is to make devices that run for longer on a battery charge and also run cooler.

[Chapter 12](#), *Learning About Processes and Threads*, describes embedded systems from the point of view of the application programmer. This chapter looks at processes and threads, inter-process communications, and scheduling policies

[Chapter 13](#), *Managing Memory*, introduces the ideas behind virtual memory and how the address space is divided into memory mappings. It also describes how to measure memory usage accurately and how to detect memory leaks.

Chapter 14, *Debugging with GDB*, shows you how to use the GNU debugger, GDB, together with the debug agent, gdbserver, to debug applications running remotely on the target device. It goes on to show how you can extend this model to debug kernel code, making use of the kernel debug stubs, KGDB.

[Chapter 15](#), *Profiling and Tracing*, covers the techniques available to measure the system performance, starting from whole system profiles and then zeroing in on particular areas where bottlenecks are causing poor performance. It also describes how to use Valgrind to check the correctness of an application's use of thread synchronization and memory allocation.

[Chapter 16](#), *Real-Time Programming*, provides a detailed guide to real-time programming on Linux, including the configuration of the kernel and the PREEMPT_RT real-time kernel patch. The kernel trace tool, Ftrace, is used to measure kernel latencies and show the effect of the various kernel

configurations.

What you need for this book

The software used in this book is entirely open source. In almost all cases, I have used the latest stable versions available at the time of writing. While I have tried to describe the main features in a manner that is not version-specific, it is inevitable that some of the examples will need adaptation to work with later software.

Embedded development involves two systems: the host, which is used for developing the programs, and the target, which runs them. For the host system, I have used Ubuntu 16.04, but most Linux distributions will work with just a little modification. You may decide to run Linux as a guest in a virtual machine, but you should be aware that some tasks, such as building a distribution using the Yocto Project, are quite demanding and are better run on a native installation of Linux.

I chose two exemplar targets: the QEMU emulator and the BeagleBone Black. Using QEMU means that you can try out most of the examples without having to invest in any additional hardware. On the other hand, some things work better if you do have real hardware, for which, I have chosen the BeagleBone Black because it is not expensive, it is widely available, and it has very good community support. Of course, you are not limited to just these two targets. The idea behind the book is to provide you with general solutions to problems so that you can apply them to a wide range of target boards.

Who this book is for

This book is written for developers who have an interest in embedded computing and Linux, and want to extend their knowledge into the various branches of the subject. In writing the book, I assume a basic understanding of the Linux command line, and in the programming examples, a working knowledge of the C language. Several chapters focus on the hardware that goes into an embedded target board, and, so, a familiarity with hardware and hardware interfaces will be a definite advantage in these cases.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You configure `tap0` in exactly the same way as any other interface."

A block of code is set as follows:

```
| / {  
|   #address-cells = <2>;  
|   #size-cells = <2>;  
|   memory@80000000 {  
|     device_type = "memory";  
|     reg = <0x00000000 0x80000000 0 0x80000000>;  
|   };  
| };
```

Any command-line input or output is written as follows:

```
| $ mipsel-unknown-linux-gnu-gcc -dumpmachine  
| mipsel-unknown-linux-gnu
```

New terms and **important words** are shown in bold.



Warnings or important notes appear in a box like this.



Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail feedback@packtpub.com, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
 2. Hover the mouse pointer on the SUPPORT tab at the top.
 3. Click on Code Downloads & Errata.
 4. Enter the name of the book in the Search box.
 5. Select the book for which you're looking to download the code files.
-
6. Choose from the drop-down menu where you purchased this book from.
 7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- TAR for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Embedded-Linux-Programming-Second-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/MasteringEmbeddedLinuxProgrammingSecondEdition_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy. Please contact us at copyright@packtpub.com with a link to the suspected pirated material. We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at questions@packtpub.com, and we will do our best to address the problem.

Starting Out

You are about to begin working on your next project, and this time it is going to be running Linux. What should you think about before you put finger to keyboard? Let's begin with a high-level look at embedded Linux and see why it is popular, what are the implications of open source licenses, and what kind of hardware you will need to run Linux.

Linux first became a viable choice for embedded devices around 1999. That was when Axis (<https://www.axis.com>), released their first Linux-powered network camera and TiVo (<https://business.tivo.com/>) their first **Digital Video Recorder (DVR)**. Since 1999, Linux has become ever more popular, to the point that today it is the operating system of choice for many classes of product. At the time of writing, in 2017, there are about two billion devices running Linux. That includes a large number of smartphones running Android, which uses a Linux kernel, and hundreds of millions of set-top-boxes, smart TVs, and Wi-Fi routers, not to mention a very diverse range of devices such as vehicle diagnostics, weighing scales, industrial devices, and medical monitoring units that ship in smaller volumes.

So, why does your TV run Linux? At first glance, the function of a TV is simple: it has to display a stream of video on a screen. Why is a complex Unix-like operating system like Linux necessary?

The simple answer is Moore's Law: Gordon Moore, co-founder of Intel, observed in 1965 that the density of components on a chip will double approximately every two years. That applies to the devices that we design and use in our everyday lives just as much as it does to desktops, laptops, and servers. At the heart of most embedded devices is a highly integrated chip that contains one or more processor cores and interfaces with main memory, mass storage, and peripherals of many types. This is referred to as a **System on Chip**, or **SoC**, and SoCs are increasing in complexity in accordance with Moore's Law. A typical SoC has a technical reference manual that stretches to thousands of pages. Your TV is not simply displaying a video stream as the old analog sets used to do.

The stream is digital, possibly encrypted, and it needs processing to create an image. Your TV is (or soon will be) connected to the Internet. It can receive content from smartphones, tablets, and home media servers. It can be (or soon will be) used to play games. And so on and so on. You need a full operating system to manage this degree of complexity.

Here are some points that drive the adoption of Linux:

- Linux has the necessary functionality. It has a good scheduler, a good network stack, support for USB, Wi-Fi, Bluetooth, many kinds of storage media, good support for multimedia devices, and so on. It ticks all the boxes.
- Linux has been ported to a wide range of processor architectures, including some that are very commonly found in SoC designs--ARM, MIPS, x86, and PowerPC.
- Linux is open source, so you have the freedom to get the source code and modify it to meet your needs. You, or someone working on your behalf, can create a board support package for your particular SoC board or device. You can add protocols, features, and technologies that may be missing from the mainline source code. You can remove features that you don't need to reduce memory and storage requirements. Linux is flexible.
- Linux has an active community; in the case of the Linux kernel, very active. There is a new release of the kernel every 8 to 10 weeks, and each release contains code from more than 1,000 developers. An active community means that Linux is up to date and supports current hardware, protocols, and standards.
- Open source licenses guarantee that you have access to the source code. There is no vendor tie-in.

For these reasons, Linux is an ideal choice for complex devices. But there are a few caveats I should mention here. Complexity makes it harder to understand. Coupled with the fast moving development process and the decentralized structures of open source, you have to put some effort into learning how to use it and to keep on re-learning as it changes. I hope that this book will help in the process.

Selecting the right operating system

Is Linux suitable for your project? Linux works well where the problem being solved justifies the complexity. It is especially good where connectivity, robustness, and complex user interfaces are required. However, it cannot solve every problem, so here are some things to consider before you jump in:

- Is your hardware up to the job? Compared to a traditional **real-time operating system (RTOS)** such as VxWorks, Linux requires a lot more resources. It needs at least a 32-bit processor and lots more memory. I will go into more detail in the section on typical hardware requirements.
- Do you have the right skill set? The early parts of a project, board bring-up, require detailed knowledge of Linux and how it relates to your hardware. Likewise, when debugging and tuning your application, you will need to be able to interpret the results. If you don't have the skills in-house, you may want to outsource some of the work. Of course, reading this book helps!
- Is your system real-time? Linux can handle many real-time activities so long as you pay attention to certain details, which I will cover in detail in Chapter 16, *Real-Time Programming*.

Consider these points carefully. Probably the best indicator of success is to look around for similar products that run Linux and see how they have done it; follow best practice.

The players

Where does open source software come from? Who writes it? In particular, how does this relate to the key components of embedded development—the toolchain, bootloader, kernel, and basic utilities found in the root filesystem?

The main players are:

- **The open source community:** This, after all, is the engine that generates the software you are going to be using. The community is a loose alliance of developers, many of whom are funded in some way, perhaps by a not-for-profit organization, an academic institution, or a commercial company. They work together to further the aims of the various projects. There are many of them—some small, some large. Some that we will be making use of in the remainder of this book are Linux itself, U-Boot, BusyBox, Buildroot, the Yocto Project, and the many projects under the GNU umbrella.
- **CPU architects:** These are the organizations that design the CPUs we use. The important ones here are ARM/Linaro (ARM-based SoCs), Intel (x86 and x86_64), Imagination Technologies (MIPS), and IBM (PowerPC). They implement or, at the very least, influence support for the basic CPU architecture.
- **SoC vendors** (Atmel, Broadcom, Intel, Qualcomm, TI, and many others). They take the kernel and toolchain from the CPU architects and modify them to support their chips. They also create reference boards: designs that are used by the next level down to create development boards and working products.
- **Board vendors and OEMs:** These people take the reference designs from SoC vendors and build them in to specific products, for instance, set-top-boxes or cameras, or create more general purpose development boards, such as those from Avantech and Kontron. An important category are the cheap development boards such as BeagleBoard/BeagleBone and Raspberry Pi that have created their own ecosystems of software and hardware add-ons.

These form a chain, with your project usually at the end, which means that you

do not have a free choice of components. You cannot simply take the latest kernel from <https://www.kernel.org/>, except in a few rare cases, because it does not have support for the chip or board that you are using.

This is an ongoing problem with embedded development. Ideally, the developers at each link in the chain would push their changes upstream, but they don't. It is not uncommon to find a kernel which has many thousands of patches that are not merged. In addition, SoC vendors tend to actively develop open source components only for their latest chips, meaning that support for any chip more than a couple of years old will be frozen and not receive any updates.

The consequence is that most embedded designs are based on old versions of software. They do not receive security fixes, performance enhancements, or features that are in newer versions. Problems such as Heartbleed (a bug in the OpenSSL libraries) and ShellShock (a bug in the bash shell) go unfixed. I will talk more about this later in this chapter under the topic of security.

What can you do about it? First, ask questions of your vendors: what is their update policy, how often do they revise kernel versions, what is the current kernel version, what was the one before that, and what is their policy for merging changes up-stream? Some vendors are making great strides in this way. You should prefer their chips.

Secondly, you can take steps to make yourself more self-sufficient. The chapters in section 1 explain the dependencies in more detail and show you where you can help yourself. Don't just take the package offered to you by the SoC or board vendor and use it blindly without considering the alternatives.

Project life cycle

This book is divided into four sections that reflect the phases of a project. The phases are not necessarily sequential. Usually they overlap and you will need to jump back to revisit things that were done previously. However, they are representative of a developer's preoccupations as the project progresses:

- Elements of embedded Linux (Chapters 1 to 6) will help you set up the development environment and create a working platform for the later phases. It is often referred to as the **board bring-up** phase.
- System architecture and design choices (Chapters 7 to 11) will help you to look at some of the design decisions you will have to make concerning the storage of programs and data, how to divide work between kernel device drivers and applications, and how to initialize the system.
- Writing embedded applications (Chapters 12 and 13) shows how to make effective use of the Linux process and threads model, and how to manage memory in a resource-constrained device.
- Debugging and optimizing performance (Chapters 14 and 15) describes how to trace, profile, and debug your code in both the applications and the kernel.

The fifth section on real-time (Chapter 16, *Real-Time Programming*) stands somewhat alone because it is a small, but important, category of embedded systems. Designing for real-time behavior has an impact on each of the four main phases.

The four elements of embedded Linux

Every project begins by obtaining, customizing, and deploying these four elements: the toolchain, the bootloader, the kernel, and the root filesystem. This is the topic of the first section of this book.

- **Toolchain:** The compiler and other tools needed to create code for your target device. Everything else depends on the toolchain.
- **Bootloader:** The program that initializes the board and loads the Linux kernel.
- **Kernel:** This is the heart of the system, managing system resources and interfacing with hardware.
- **Root filesystem:** Contains the libraries and programs that are run once the kernel has completed its initialization.

Of course, there is also a fifth element, not mentioned here. That is the collection of programs specific to your embedded application which make the device do whatever it is supposed to do, be it weigh groceries, display movies, control a robot, or fly a drone.

Typically, you will be offered some or all of these elements as a package when you buy your SoC or board. But, for the reasons mentioned in the preceding paragraph, they may not be the best choices for you. I will give you the background to make the right selections in the first six chapters and I will introduce you to two tools that automate the whole process for you: Buildroot and the Yocto Project.

Open source

The components of embedded Linux are open source, so now is a good time to consider what that means, why open sources work the way they do, and how this affects the often proprietary embedded device you will be creating from it.

Licenses

When talking about open source, the word *free* is often used. People new to the subject often take it to mean nothing to pay, and open source software licenses do indeed guarantee that you can use the software to develop and deploy systems for no charge. However, the more important meaning here is freedom, since you are free to obtain the source code, modify it in any way you see fit, and redeploy it in other systems. These licenses give you this right. Compare that with shareware licenses which allow you to copy the binaries for no cost but do not give you the source code, or other licenses that allow you to use the software for free under certain circumstances, for example, for personal use but not commercial. These are not open source.

I will provide the following comments in the interest of helping you understand the implications of working with open source licenses, but I would like to point out that I am an engineer and not a lawyer. What follows is my understanding of the licenses and the way they are interpreted.

Open source licenses fall broadly into two categories: the copyleft licenses such as the **General Public License (GPL)** and the permissive licenses such as those from the **Berkeley Software Distribution (BSD)**, the Apache Foundation, and others.

The permissive licenses say, in essence, that you may modify the source code and use it in systems of your own choosing so long as you do not modify the terms of the license in any way. In other words, with that one restriction, you can do with it what you want, including building it into possibly proprietary systems.

The GPL licenses are similar, but have clauses which compel you to pass the rights to obtain and modify the software on to your end users. In other words, you share your source code. One option is to make it completely public by putting it onto a public server. Another is to offer it only to your end users by means of a written offer to provide the code when requested. The GPL goes further to say that you cannot incorporate GPL code into proprietary programs. Any attempt to do so would make the GPL apply to the whole. In other words,

you cannot combine a GPL and proprietary code in one program.

So, what about libraries? If they are licensed with the GPL, any program linked with them becomes GPL also. However, most libraries are licensed under the **Lesser General Public License (LGPL)**. If this is the case, you are allowed to link with them from a proprietary program.

All the preceding description relates specifically to GPL v2 and LGPL v2.1. I should mention the latest versions of GPL v3 and LGPL v3. These are controversial, and I will admit that I don't fully understand the implications. However, the intention is to ensure that the GPLv3 and LGPL v3 components in any system can be replaced by the end user, which is in the spirit of open source software for everyone. It does pose some problems though. Some Linux devices are used to gain access to information according to a subscription level or another restriction, and replacing critical parts of the software may compromise that. Set-top-boxes fit into this category. There are also issues with security. If the owner of a device has access to the system code, then so might an unwelcome intruder. Often the defense is to have kernel images that are signed by an authority, the vendor, so that unauthorized updates are not possible. Is that an infringement of my right to modify my device? Opinions differ.



*The TiVo set-top-box is an important part of this debate. It uses a Linux kernel, which is licensed under GPL v2. TiVo have released the source code of their version of the kernel and so comply with the license. TiVo also has a bootloader that will only load a kernel binary that is signed by them. Consequently, you can build a modified kernel for a TiVo box but you cannot load it on the hardware. The **Free Software Foundation (FSF)** takes the position that this is not in the spirit of open source software and refers to this procedure as **Tivoization**. The GPL v3 and LGPL v3 were written to explicitly prevent this happening. Some projects, the Linux kernel in particular, have been reluctant to adopt the version three licenses because of the restrictions it would place on device manufacturers.*

Hardware for embedded Linux

If you are designing or selecting hardware for an embedded Linux project, what do you look out for?

Firstly, a CPU architecture that is supported by the kernel—unless you plan to add a new architecture yourself, of course! Looking at the source code for Linux 4.9, there are 31 architectures, each represented by a sub-directory in the arch/ directory. They are all 32- or 64-bit architectures, most with a **memory management unit (MMU)**, but some without. The ones most often found in embedded devices are ARM, MIPS PowerPC, and X86, each in 32- and 64-bit variants, and all of which have memory management units.

Most of this book is written with this class of processor in mind. There is another group that doesn't have an MMU that runs a subset of Linux known as **microcontroller Linux** or **uClinux**. These processor architectures include ARC, Blackfin, MicroBlaze, and Nios. I will mention uClinux from time to time but I will not go into detail because it is a rather specialized topic.

Secondly, you will need a reasonable amount of RAM. 16 MiB is a good minimum, although it is quite possible to run Linux using half that. It is even possible to run Linux with 4 MiB if you are prepared to go to the trouble of optimizing every part of the system. It may even be possible to get lower, but there comes a point at which it is no longer Linux.

Thirdly, there is non-volatile storage, usually flash memory. 8 MiB is enough for a simple device such as a webcam or a simple router. As with RAM, you can create a workable Linux system with less storage if you really want to, but the lower you go, the harder it becomes. Linux has extensive support for flash storage devices, including raw NOR and NAND flash chips, and managed flash in the form of SD cards, eMMC chips, USB flash memory, and so on.

Fourthly, a debug port is very useful, most commonly an RS-232 serial port. It does not have to be fitted on production boards, but makes board bring-up, debugging, and development much easier.

Fifthly, you need some means of loading software when starting from scratch. A few years ago, boards would have been fitted with a **Joint Test Action Group (JTAG)** interface for this purpose, but modern SoCs have the ability to load boot code directly from removable media, especially SD and micro SD cards, or serial interfaces such as RS-232 or USB.

In addition to these basics, there are interfaces to the specific bits of hardware your device needs to get its job done. Mainline Linux comes with open source drivers for many thousands of different devices, and there are drivers (of variable quality) from the SoC manufacturer and from the OEMs of third-party chips that may be included in the design, but remember my comments on the commitment and ability of some manufacturers. As a developer of embedded devices, you will find that you spend quite a lot of time evaluating and adapting third-party code, if you have it, or liaising with the manufacturer if you don't. Finally, you will have to write the device support for interfaces that are unique to the device, or find someone to do it for you.

Hardware used in this book

The worked examples in this book are intended to be generic, but to make them relevant and easy to follow, I have had to choose specific hardware. I have chosen two exemplar devices: the BeagleBone Black and QEMU. The first is a widely-available and cheap development board which can be used in serious embedded hardware. The second is a machine emulator that can be used to create a range of systems that are typical of embedded hardware. It was tempting to use QEMU exclusively, but, like all emulations, it is not quite the same as the real thing. Using a BeagleBone Black, you have the satisfaction of interacting with real hardware and seeing real LEDs flash. I could have selected a board that is more up-to-date than the BeagleBone Black, which is several years old now, but I believe that its popularity gives it a degree of longevity and it means that it will continue to be available for some years yet.

In any case, I encourage you to try out as many of the examples as you can, using either of these two platforms, or indeed any embedded hardware you may have to hand.

The BeagleBone Black

The BeagleBone and the later BeagleBone Black are open hardware designs for a small, credit card sized development board produced by CircuitCo LLC. The main repository of information is at <https://beagleboard.org/>. The main points of the specifications are:

- TI AM335x 1 GHz ARM® Cortex-A8 Sitara SoC
- 512 MiB DDR3 RAM
- 2 or 4 GiB 8-bit eMMC on-board flash storage
- Serial port for debug and development
- MicroSD connector, which can be used as the boot device
- Mini USB OTG client/host port that can also be used to power the board
- Full size USB 2.0 host port
- 10/100 Ethernet port
- HDMI for video and audio output

In addition, there are two 46-pin expansion headers for which there are a great variety of daughter boards, known as **cap**es, which allow you to adapt the board to do many different things. However, you do not need to fit any capes in the examples in this book.

In addition to the board itself, you will need:

- A mini USB to full-size USB cable (supplied with the board) to provide power, unless you have the last item on this list.
- An RS-232 cable that can interface with the 6-pin 3.3V TTL level signals provided by the board. The Beagleboard website has links to compatible cables.
- A microSD card and a means of writing to it from your development PC or laptop, which will be needed to load software onto the board.
- An Ethernet cable, as some of the examples require network connectivity.
- Optional, but recommended, a 5V power supply capable of delivering 1 A or more.

QEMU

QEMU is a machine emulator. It comes in a number of different flavors, each of which can emulate a processor architecture and a number of boards built using that architecture. For example, we have the following:

- `qemu-system-arm`: ARM
- `qemu-system-mips`: MIPS
- `qemu-system-ppc`: PowerPC
- `qemu-system-x86`: x86 and x86_64

For each architecture, QEMU emulates a range of hardware, which you can see by using the option—`machine help`. Each machine emulates most of the hardware that would normally be found on that board. There are options to link hardware to local resources, such as using a local file for the emulated disk drive. Here is a concrete example:

```
$ qemu-system-arm -machine vexpress-a9 -m 256M -drive  
file=rootfs.ext4,sd -net nic -net use -kernel zImage -dtb vexpress-  
v2p-ca9.dtb -append "console=ttyAMA0,115200 root=/dev/mmcblk0" -  
serial stdio -net nic,model=lan9118 -net tap,ifname=tap0
```

The options used in the preceding command line are:

- `-machine vexpress-a9`: Creates an emulation of an ARM Versatile Express development board with a Cortex A-9 processor
- `-m 256M`: Populates it with 256 MiB of RAM
- `-drive file=rootfs.ext4,sd`: Connects the SD interface to the local file `rootfs.ext4` (which contains a filesystem image)
- `-kernel zImage`: Loads the Linux kernel from the local file named `zImage`
- `-dtb vexpress-v2p-ca9.dtb`: Loads the device tree from the local file `vexpress-v2p-ca9.dtb`
- `-append "..."`: Supplies this string as the kernel command-line
- `-serial stdio`: Connects the serial port to the terminal that launched QEMU, usually so that you can log on to the emulated machine via the serial console
- `-net nic,model=lan9118`: Creates a network interface

- `-net tap,ifname=tap0`: Connects the network interface to the virtual network interface `tap0`

To configure the host side of the network, you need the `tunctl` command from the **User Mode Linux (UML)** project; on Debian and Ubuntu, the package is named `uml-utilities`:

```
| $ sudo tunctl -u $(whoami) -t tap0
```

This creates a network interface named `tap0` which is connected to the network controller in the emulated QEMU machine. You configure `tap0` in exactly the same way as any other interface.

All of these options are described in detail in the following chapters. I will be using Versatile Express for most of my examples, but it should be easy to use a different machine or architecture.

Software used in this book

I have used only open source software, both for the development tools and the target operating system and applications. I assume that you will be using Linux on your development system. I tested all the host commands using Ubuntu 14.04 and so there is a slight bias towards that particular version, but any modern Linux distribution is likely to work just fine.

Summary

Embedded hardware will continue to get more complex, following the trajectory set by Moore's Law. Linux has the power and the flexibility to make use of hardware in an efficient way.

Linux is just one component of open source software out of the many that you need to create a working product. The fact that the code is freely available means that people and organizations at many different levels can contribute. However, the sheer variety of embedded platforms and the fast pace of development lead to isolated pools of software which are not shared as efficiently as they should be. In many cases, you will become dependent on this software, especially the Linux kernel that is provided by your SoC or Board vendor, and to a lesser extent, the toolchain. Some SoC manufacturers are getting better at pushing their changes upstream and the maintenance of these changes is getting easier.

Fortunately, there are some powerful tools that can help you create and maintain the software for your device. For example, Buildroot is ideal for small systems and the Yocto Project for larger ones. Before I describe these build tools, I will describe the four elements of embedded Linux, which you can apply to all embedded Linux projects, however they are created.

The next chapter is all about the first of these, the toolchain, which you need to compile code for your target platform.

Learning About Toolchains

The toolchain is the first element of embedded Linux and the starting point of your project. You will use it to compile all the code that will run on your device. The choices you make at this early stage will have a profound impact on the final outcome. Your toolchain should be capable of making effective use of your hardware by using the optimum instruction set for your processor. It should support the languages that you require, and have a solid implementation of the **Portable Operating System Interface (POSIX)** and other system interfaces. Not only that, but it should be updated when security flaws are discovered or bugs are found. Finally, it should be constant throughout the project. In other words, once you have chosen your toolchain, it is important to stick with it. Changing compilers and development libraries in an inconsistent way during a project will lead to subtle bugs.

Obtaining a toolchain can be as simple as downloading and installing a TAR file, or it can be as complex as building the whole thing from source code. In this chapter, I take the latter approach, with the help of a tool called **crosstool-NG**, so that I can show you the details of creating a toolchain. Later on in [Chapter 6](#), *Selecting a Build System*, I will switch to using the toolchain generated by the build system, which is the more usual means of obtaining a toolchain.

In this chapter, we will cover the following topics:

- Introducing toolchains
- Finding a toolchain
- Building a toolchain using the crosstool-NG tool
- Anatomy of a toolchain
- Linking with libraries--static and dynamic linking
- The art of cross compiling

Introducing toolchains

A toolchain is the set of tools that compiles source code into executables that can run on your target device, and includes a compiler, a linker, and run-time libraries. Initially you need one to build the other three elements of an embedded Linux system: the bootloader, the kernel, and the root filesystem. It has to be able to compile code written in assembly, C, and C++ since these are the languages used in the base open source packages.

Usually, toolchains for Linux are based on components from the GNU project (<http://www.gnu.org>), and that is still true in the majority of cases at the time of writing. However, over the past few years, the **Clang** compiler and the associated **Low Level Virtual Machine (LLVM)** project (<http://llvm.org>) have progressed to the point that it is now a viable alternative to a GNU toolchain. One major distinction between LLVM and GNU-based toolchains is the licensing; LLVM has a BSD license while GNU has the GPL. There are some technical advantages to Clang as well, such as faster compilation and better diagnostics, but GNU GCC has the advantage of compatibility with the existing code base and support for a wide range of architectures and operating systems. Indeed, there are still some areas where Clang cannot replace the GNU C compiler, especially when it comes to compiling a mainline Linux kernel. It is probable that, in the next year or so, Clang will be able to compile all the components needed for embedded Linux and so will become an alternative to GNU. There is a good description of how to use Clang for cross compilation at <http://clang.llvm.org/docs/CrossCompilation.html>. If you would like to use it as part of an embedded Linux build system, the EmbToolkit (<https://www.embt toolkit.org>) fully supports both GNU and LLVM/Clang toolchains, and various people are working on using Clang with Buildroot and the Yocto Project. I will cover embedded build systems in [Chapter 6, *Selecting a Build System*](#). Meanwhile, this chapter focuses on the GNU toolchain as it is the only complete option at this time.

A standard GNU toolchain consists of three main components:

- **Binutils:** A set of binary utilities including the assembler and the linker. It is available at <http://www.gnu.org/software/binutils>.

- **GNU Compiler Collection (GCC):** These are the compilers for C and other languages which, depending on the version of GCC, include C++, Objective-C, Objective-C++, Java, Fortran, Ada, and Go. They all use a common backend which produces assembler code, which is fed to the GNU assembler. It is available at <http://gcc.gnu.org/>.
- **C library:** A standardized **application program interface (API)** based on the POSIX specification, which is the main interface to the operating system kernel for applications. There are several C libraries to consider, as we shall see later on in this chapter.

As well as these, you will need a copy of the Linux kernel headers, which contain definitions and constants that are needed when accessing the kernel directly. Right now, you need them to be able to compile the C library, but you will also need them later when writing programs or compiling libraries that interact with particular Linux devices, for example, to display graphics via the Linux frame buffer driver. This is not simply a question of making a copy of the header files in the include directory of your kernel source code. Those headers are intended for use in the kernel only and contain definitions that will cause conflicts if used in their raw state to compile regular Linux applications.

Instead, you will need to generate a set of sanitized kernel headers, which I have illustrated in [Chapter 5](#), *Building a Root Filesystem*.

It is not usually crucial whether the kernel headers are generated from the exact version of Linux you are going to be using or not. Since the kernel interfaces are always backwards-compatible, it is only necessary that the headers are from a kernel that is the same as, or older than, the one you are using on the target.

Most people would consider the **GNU Debugger (GDB)** to be part of the toolchain as well, and it is usual that it is built at this point. I will talk about GDB in [Chapter 14](#), *Debugging with GDB*.

Types of toolchains

For our purposes, there are two types of toolchain:

- **Native:** This toolchain runs on the same type of system (sometimes the same actual system) as the programs it generates. This is the usual case for desktops and servers, and it is becoming popular on certain classes of embedded devices. The Raspberry Pi running Debian for ARM, for example, has self-hosted native compilers.
- **Cross:** This toolchain runs on a different type of system than the target, allowing the development to be done on a fast desktop PC and then loaded onto the embedded target for testing.

Almost all embedded Linux development is done using a cross development toolchain, partly because most embedded devices are not well suited to program development since they lack computing power, memory, and storage, but also because it keeps the host and target environments separate. The latter point is especially important when the host and the target are using the same architecture, x86_64, for example. In this case, it is tempting to compile natively on the host and simply copy the binaries to the target.

This works up to a point, but it is likely that the host distribution will receive updates more often than the target, or that different engineers building code for the target will have slightly different versions of the host development libraries. Over time, the development and target systems will diverge and you will violate the principle that the toolchain should remain constant throughout the life of the project. You can make this approach work if you ensure that the host and the target build environments are in lockstep with each other. However, a much better approach is to keep the host and the target separate, and a cross toolchain is the way to do that.

However, there is a counter argument in favor of native development. Cross development creates the burden of cross-compiling all the libraries and tools that you need for your target. We will see later in this chapter that cross-compiling is not always simple because many open source packages are not designed to be

built in this way. Integrated build tools, including Buildroot and the Yocto Project, help by encapsulating the rules to cross compile a range of packages that you need in typical embedded systems, but if you want to compile a large number of additional packages, then it is better to natively compile them. For example, building a Debian distribution for the Raspberry Pi or BeagleBone using a cross compiler would be very hard. Instead, they are natively compiled. Creating a native build environment from scratch is not easy. You would still need a cross compiler at first to create the native build environment on the target, which you then use to build the packages. Then, in order to perform the native build in a reasonable amount of time, you would need a build farm of well-provisioned target boards, or you may be able to use QEMU to emulate the target.

Meanwhile, in this chapter, I will focus on the more mainstream cross compiler environment, which is relatively easy to set up and administer.

CPU architectures

The toolchain has to be built according to the capabilities of the target CPU, which includes:

- **CPU architecture:** ARM, MIPS, x86_64, and so on
- **Big- or little-endian operation:** Some CPUs can operate in both modes, but the machine code is different for each
- **Floating point support:** Not all versions of embedded processors implement a hardware floating point unit, in which case the toolchain has to be configured to call a software floating point library instead
- **Application Binary Interface (ABI):** The calling convention used for passing parameters between function calls

With many architectures, the ABI is constant across the family of processors. One notable exception is ARM. The ARM architecture transitioned to the **Extended Application Binary Interface (EABI)** in the late 2000s, resulting in the previous ABI being named the **Old Application Binary Interface (OABI)**. While the OABI is now obsolete, you continue to see references to EABI. Since then, the EABI has split into two, based on the way the floating point parameters are passed. The original EABI uses general purpose (integer) registers, while the newer **Extended Application Binary Interface Hard-Float (EABIHF)** uses floating point registers. The EABIHF is significantly faster at floating point operations, since it removes the need for copying between integer and floating point registers, but it is not compatible with CPUs that do not have a floating point unit. The choice, then, is between two incompatible ABIs; you cannot mix and match the two, and so you have to decide at this stage.

GNU uses a prefix to the name of each tool in the toolchain, which identifies the various combinations that can be generated. It consists of a tuple of three or four components separated by dashes, as described here:

- **CPU:** This is the CPU architecture, such as ARM, MIPS, or x86_64. If the CPU has both endian modes, they may be differentiated by adding `e1` for little-endian or `eb` for big-endian. Good examples are little-endian MIPS,

mipsel and big-endian ARM, armeb.

- **Vendor:** This identifies the provider of the toolchain. Examples include buildroot, poky, or just unknown. Sometimes it is left out altogether.
- **Kernel:** For our purposes, it is always linux.
- **Operating system:** A name for the user space component, which might be gnu or musl. The ABI may be appended here as well, so for ARM toolchains, you may see gnueabi, gnueabihf, musleabi, or musleabihf.

You can find the tuple used when building the toolchain by using the `-dumpmachine` option of `gcc`. For example, you may see the following on the host computer:

```
$ gcc -dumpmachine
x86_64-linux-gnu
```



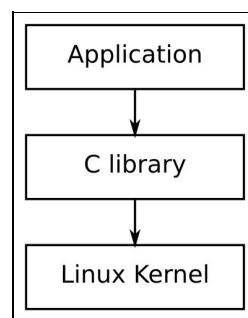
When a native compiler is installed on a machine, it is normal to create links to each of the tools in the toolchain with no prefixes, so that you can call the C compiler with the `gcc` command.

Here is an example using a cross compiler:

```
$ mipsel-unknown-linux-gnu-gcc -dumpmachine
mipsel-unknown-linux-gnu
```

Choosing the C library

The programming interface to the Unix operating system is defined in the C language, which is now defined by the POSIX standards. The C library is the implementation of that interface; it is the gateway to the kernel for Linux programs, as shown in the following diagram. Even if you are writing programs in another language, maybe Java or Python, the respective run-time support libraries will have to call the **C library** eventually, as shown here:



Whenever the C library needs the services of the kernel, it will use the kernel system call interface to transition between user space and kernel space. It is possible to bypass the C library by making the kernel system calls directly, but that is a lot of trouble and almost never necessary.

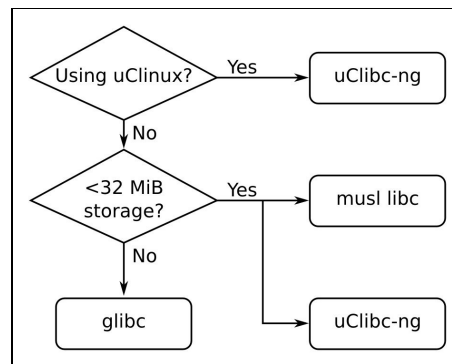
There are several C libraries to choose from. The main options are as follows:

- **glibc**: This is the standard GNU C library, available at <http://www.gnu.org/software/libc>. It is big and, until recently, not very configurable, but it is the most complete implementation of the POSIX API. The license is LGPL 2.1.
- **musl libc**: This is available at <https://www.musl-libc.org>. The `musl libc` library is comparatively new, but has been gaining a lot of attention as a small and standards-compliant alternative to GNU `libc`. It is a good choice for systems with a limited amount of RAM and storage. It has an MIT license.
- **uClibc-ng**: This is available at <https://uclibc-ng.org/>. `u` is really a Greek `mu` character, indicating that this is the micro controller C library. It was first developed to work with uClinux (Linux for CPUs without memory management units), but has since been adapted to be used with full Linux. The `uclibc-ng` library is a fork of the original `uclibc` project (<https://uclibc.org/>),

which has unfortunately fallen into disrepair. Both are licensed with LGPL 2.1.

- **eglibc:** This is available at <http://www.eglibc.org/home>. Now obsolete, eglibc was a fork of glibc with changes to make it more suitable for embedded usage. Among other things, eglibc added configuration options and support for architectures not covered by glibc, in particular the PowerPC e500 CPU core. The code base from eglibc was merged back into glibc in version 2.20. The eglibc library is no longer maintained.

So, which to choose? My advice is to use uClibc-ng only if you are using uClinux. If you have very limited amount of storage or RAM, then musl libc is a good choice, otherwise, use glibc, as shown in this flow chart:



Finding a toolchain

You have three choices for your cross development toolchain: you may find a ready built toolchain that matches your needs, you can use the one generated by an embedded build tool which is covered in [Chapter 6, *Selecting a Build System*](#), or you can create one yourself as described later in this chapter.

A pre-built cross toolchain is an attractive option in that you only have to download and install it, but you are limited to the configuration of that particular toolchain and you are dependent on the person or organization you got it from. Most likely, it will be one of these:

- An SoC or board vendor. Most vendors offer a Linux toolchain.
- A consortium dedicated to providing system-level support for a given architecture. For example, Linaro, (<https://www.linaro.org/>) have pre-built toolchains for the ARM architecture.
- A third-party Linux tool vendor, such as Mentor Graphics, TimeSys, or MontaVista.
- The cross tool packages for your desktop Linux distribution. For example, Debian-based distributions have packages for cross compiling for ARM, MIPS, and PowerPC targets.
- A binary SDK produced by one of the integrated embedded build tools. The Yocto Project has some examples at [http://downloads.yoctoproject.org/releases/yocto/yocto-\[version\]/toolchain](http://downloads.yoctoproject.org/releases/yocto/yocto-[version]/toolchain).
- A link from a forum that you can't find any more.

In all of these cases, you have to decide whether the pre-built toolchain on offer meets your requirements. Does it use the C library you prefer? Will the provider give you updates for security fixes and bugs, bearing in mind my comments on support and updates from [Chapter 1, *Starting Out*](#). If your answer is no to any of these, then you should consider creating your own.

Unfortunately, building a toolchain is no easy task. If you truly want to do the whole thing yourself, take a look at **Cross Linux From Scratch** (<http://trac.clfs.org>). There you will find step-by-step instructions on how to create each component.

A simpler alternative is to use crosstool-NG, which encapsulates the process into a set of scripts and has a menu-driven frontend. You still need a fair degree of knowledge, though, just to make the right choices.

It is simpler still to use a build system such as Buildroot or the Yocto Project, since they generate a toolchain as part of the build process. This is my preferred solution, as I have shown in [Chapter 6](#), *Selecting a Build System*.

Building a toolchain using crosstool-NG

Some years ago, Dan Kegel wrote a set of scripts and `makefiles` for generating cross development toolchains and called it crosstool (<http://kegel.com/crosstool/>). In 2007, Yann E. Morin used that base to create the next generation of crosstool, crosstool-NG (<http://crosstool-ng.github.io/>). Today it is by far the most convenient way to create a stand-alone cross toolchain from source.

Installing crosstool-NG

Before you begin, you will need a working native toolchain and build tools on your host PC. To work with crosstool-NG on an Ubuntu host, you will need to install the packages using the following command:

```
$ sudo apt-get install automake bison chrpath flex g++ git gperf \  
gawk libexpat1-dev libncurses5-dev libsdl1.2-dev libtool \  
python2.7-dev texinfo
```

Next, get the current release from the crosstool-NG Git repository. In my examples, I have used version 1.22.0. Extract it and create the frontend menu system, `ct-ng`, as shown in the following commands:

```
$ git clone https://github.com/crosstool-ng/crosstool-ng.git  
$ cd crosstool-ng  
$ git checkout crosstool-ng-1.22.0  
$ ./bootstrap  
$ ./configure --enable-local  
$ make  
$ make install
```

The `--enable-local` option means that the program will be installed into the current directory, which avoids the need for root permissions, as would be required if you were to install it in the default location `/usr/local/bin`. Type `./ct-ng` from the current directory to launch the crosstool menu.

Building a toolchain for BeagleBone Black

Crosstool-NG can build many different combinations of toolchains. To make the initial configuration easier, it comes with a set of samples that cover many of the common use-cases. Use `./ct-ng list-samples` to generate the list.

The BeagleBone Black has a TI AM335x SoC, which contains an ARM Cortex A8 core and a VFPv3 floating point unit. Since the BeagleBone Black has plenty of RAM and storage, we can use `glibc` as the C library. The closest sample is `arm-cortex_a8-linux-gnueabi`. You can see the default configuration by prefixing the name with `show-`, as demonstrated here:

```
$ ./ct-ng show-arm-cortex_a8-linux-gnueabi
[L..] arm-cortex_a8-linux-gnueabi
OS : linux-4.3
Companion libs : gmp-6.0.0a mpfr-3.1.3 mpc-1.0.3 libelf-0.8.13 expat-2.1.0
                  ncurses-6.0
binutils : binutils-2.25.1
C compilers : gcc | 5.2.0
Languages : C,C++
C library : glibc-2.22 (threads: nptl)
Tools : dmalloc-5.5.2 duma-2_5_15 gdb-7.10 ltrace-0.7.3 strace-4.10
```

This is a close match with our requirements, except that it using the `eabi` binary interface, which passes floating point arguments in integer registers. We would prefer to use hardware floating point registers for that purpose because it would speed up function calls that have float and double parameter types. You can change the configuration later on, so for now you should select this target configuration:

```
| $ ./ct-ng arm-cortex_a8-linux-gnueabi
```

At this point, you can review the configuration and make changes using the configuration menu command `menuconfig`:

```
| $ ./ct-ng menuconfig
```

The menu system is based on the Linux kernel `menuconfig`, and so navigation of the user interface will be familiar to anyone who has configured a kernel. If not,

refer to [Chapter 4, Configuring and Building the Kernel](#) for a description of `menuconfig`.

There are two configuration changes that I would recommend you make at this point:

- In Paths and misc options, disable Render the toolchain read-only (`CT_INSTALL_DIR_RO`)
- In Target options | Floating point, select hardware (FPU) (`CT_ARCH_FLOAT_HW`)

The first is necessary if you want to add libraries to the toolchain after it has been installed, which I describe later in this chapter. The second selects the `eabi` binary interface for the reasons discussed earlier. The names in parentheses are the configuration labels stored in the configuration file. When you have made the changes, exit the `menuconfig` menu and save the configuration as you do so.

Now you can use `crosstool-NG` to get, configure, and build the components according to your specification, by typing the following command:

```
| $ ./ct-ng build
```

The build will take about half an hour, after which you will find your toolchain is present in `~/x-tools/arm-cortex_a8-linux-gnueabi`.

Building a toolchain for QEMU

On the QEMU target, you will be emulating an ARM-versatile PB evaluation board that has an ARM926EJ-S processor core, which implements the ARMv5TE instruction set. You need to generate a crosstool-NG toolchain that matches with the specification. The procedure is very similar to the one for the BeagleBone Black.

You begin by running `./ct-ng list-samples` to find a good base configuration to work from. There isn't an exact fit, so use a generic target, `arm-unknown-linux-gnueabi`. You select it as shown, running `distclean` first to make sure that there are no artifacts left over from a previous build:

```
| $ ./ct-ng distclean  
| $ ./ct-ng arm-unknown-linux-gnueabi
```

As with the BeagleBone Black, you can review the configuration and make changes using the configuration menu command `./ct-ng menuconfig`. There is only one change necessary:

- In Paths and misc options, disable Render the toolchain read-only (`CT_INSTALL_DIR_RO`)

Now, build the toolchain with the command as shown here:

```
| $ ./ct-ng build
```

As before, the build will take about half an hour. The toolchain will be installed in `~/x-tools/arm-unknown-linux-gnueabi`.

Anatomy of a toolchain

To get an idea of what is in a typical toolchain, I want to examine the crosstool-NG toolchain you have just created. The examples use the ARM Cortex A8 toolchain created for the BeagleBone Black, which has the prefix `arm-cortex_a8-linux-gnueabi`. If you built the ARM926EJ-S toolchain for the QEMU target, then the prefix will be `arm-unknown-linux-gnueabi` instead.

The ARM Cortex A8 toolchain is in the directory `~/x-tools/arm-cortex_a8-linux-gnueabi/bin`. In there you will find the cross compiler, `arm-cortex_a8-linux-gnueabi-gcc`. To make use of it, you need to add the directory to your path using the following command:

```
| $ PATH=~/x-tools/arm-cortex_a8-linux-gnueabi/bin:$PATH
```

Now you can take a simple helloworld program, which in the C language looks like this:

```
| #include <stdio.h>
| #include <stdlib.h>
|
| int main (int argc, char *argv[])
| {
|     printf ("Hello, world!\n");
|     return 0;
| }
```

You compile it like this:

```
| $ arm-cortex_a8-linux-gnueabi-gcc helloworld.c -o helloworld
```

You can confirm that it has been cross compiled by using the `file` command to print the type of the file:

```
| $ file helloworld
| helloworld: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked
```

Finding out about your cross compiler

Imagine that you have just received a toolchain and that you would like to know more about how it was configured. You can find out a lot by querying `gcc`. For example, to find the version, you use `--version`:

```
$ arm-cortex_a8-linux-gnueabi-gcc --version
arm-cortex_a8-linux-gnueabi-gcc (crosstool-NG crosstool-ng-1.22.0) 5.2.0
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

To find how it was configured, use `-v`:

```
$ arm-cortex_a8-linux-gnueabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-cortex_a8-linux-gnueabi-gcc
COLLECT_LTO_WRAPPER=/home/chris/x-tools/arm-cortex_a8-linux-gnueabi/libexec/gcc/arm-c
Target: arm-cortex_a8-linux-gnueabi
Configured with: /home/chris/crosstool-ng/.build/src/gcc-5.2.0/configure --build=x86_64
Thread model: posix
gcc version 5.2.0 (crosstool-NG crosstool-ng-1.22.0)
```

There is a lot of output there, but the interesting things to note are:

- `--with-sysroot=/home/chris/x-tools/arm-cortex_a8-linux-gnueabi/arm-cortex_a8-linux-gnueabi/sysroot`: This is the default `sysroot` directory; see the following section for an explanation
- `--enable-languages=c,c++`: Using this, we have both C and C++ languages enabled
- `--with-cpu=cortex-a8`: The code is generated for an ARM Cortex A8 core
- `--with-float=hard`: Generates opcodes for the floating point unit and uses the VFP registers for parameters
- `--enable-threads=posix`: This enables the POSIX threads

These are the default settings for the compiler. You can override most of them on the `gcc` command line. For example, if you want to compile for a different CPU, you can override the configured setting, `--with-cpu`, by adding `-mcpu` to the command line, as follows:

```
| $ arm-cortex_a8-linux-gnueabi-gcc -mcpu=cortex-a5 helloworld.c \  
-o helloworld
```

You can print out the range of architecture-specific options available using `--target-help`, as follows:

```
| $ arm-cortex_a8-linux-gnueabi-gcc --target-help
```

You may be wondering if it matters that you get the configuration exactly right at this point, since you can always change it as shown here. The answer depends on the way you anticipate using it. If you plan to create a new toolchain for each target, then it makes sense to set everything up at the beginning, because it will reduce the risks of getting it wrong later on. Jumping ahead a little to [Chapter 6, *Selecting a Build System*](#), I call this the Buildroot philosophy. If, on the other hand, you want to build a toolchain that is generic and you are prepared to provide the correct settings when you build for a particular target, then you should make the base toolchain generic, which is the way the Yocto Project handles things. The preceding examples follow the Buildroot philosophy.

The sysroot, library, and header files

The toolchain `sysroot` is a directory which contains subdirectories for libraries, header files, and other configuration files. It can be set when the toolchain is configured through `--with-sysroot=`, or it can be set on the command line using `--sysroot=`. You can see the location of the default `sysroot` by using `-print-sysroot`:

```
$ arm-cortex_a8-linux-gnueabi-gcc -print-sysroot  
/home/chris/x-tools/arm-cortex_a8-linux-gnueabi/arm-cortex_a8-linux-gnueabi/sysroot
```

You will find the following subdirectories in `sysroot`:

- `lib`: Contains the shared objects for the C library and the dynamic linker/loader, `ld-linux`
- `usr/lib`, the static library archive files for the C library, and any other libraries that may be installed subsequently
- `usr/include`: Contains the headers for all the libraries
- `usr/bin`: Contains the utility programs that run on the target, such as the `ldd` command
- `use/share`: Used for localization and internationalization
- `sbin`: Provides the `ldconfig` utility, used to optimize library loading paths

Plainly, some of these are needed on the development host to compile programs, and others - for example, the shared libraries and `ld-linux` - are needed on the target at runtime.

Other tools in the toolchain

The following table shows various other components of a GNU toolchain, together with a brief description:

Command	Description
addr2line	Converts program addresses into filenames and numbers by reading the debug symbol tables in an executable file. It is very useful when decoding addresses printed out in a system crash report.
ar	The archive utility is used to create static libraries.
as	This is the GNU assembler.
c++filt	This is used to demangle C++ and Java symbols.
cpp	This is the C preprocessor and is used to expand <code>#define</code> , <code>#include</code> , and other similar directives. You seldom need to use this by itself.
elfedit	This is used to update the ELF header of the ELF files.
g++	This is the GNU C++ frontend, which assumes that source files contain C++ code.
gcc	This is the GNU C frontend, which assumes that source files contain C code.
gcov	This is a code coverage tool.
gdb	This is the GNU debugger.
gprof	This is a program profiling tool.
ld	This is the GNU linker.
nm	This lists symbols from object files.
objcopy	This is used to copy and translate object files.
objdump	This is used to display information from object files.

ranlib	This creates or modifies an index in a static library, making the linking stage faster.
readelf	This displays information about files in ELF object format.
size	This lists section sizes and the total size.
strings	This displays strings of printable characters in files.
strip	This is used to strip an object file of debug symbol tables, thus making it smaller. Typically, you would strip all the executable code that is put onto the target.

Looking at the components of the C library

The C library is not a single library file. It is composed of four main parts that together implement the POSIX API:

- `libc`: The main C library that contains the well-known POSIX functions such as `printf`, `open`, `close`, `read`, `write`, and so on
- `libm`: Contains maths functions such as `cos`, `exp`, and `log`
- `libpthread`: Contains all the POSIX thread functions with names beginning with `pthread_`
- `librt`: Has the real-time extensions to POSIX, including shared memory and asynchronous I/O

The first one, `libc`, is always linked in but the others have to be explicitly linked with the `-l` option. The parameter to `-l` is the library name with `lib` stripped off. For example, a program that calculates a sine function by calling `sin()` would be linked with `libm` using `-lm`:

```
| $ arm-cortex_a8-linux-gnueabi-gcc myprog.c -o myprog -lm
```

You can verify which libraries have been linked in this or any other program by using the `readelf` command:

```
| $ arm-cortex_a8-linux-gnueabi-readelf -a myprog | grep "Shared library"
0x00000001 (NEEDED) Shared library: [libm.so.6]
0x00000001 (NEEDED) Shared library: [libc.so.6]
```

Shared libraries need a runtime linker, which you can expose using:

```
| $ arm-cortex_a8-linux-gnueabi-readelf -a myprog | grep "program interpreter"
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]
```

This is so useful that I have a script file named `list-libs`, which you will find in the book code archive in `MELP/list-libs`. It contains the following commands:

```
| #!/bin/sh
| ${CROSS_COMPILE}readelf -a $1 | grep "program interpreter"
| ${CROSS_COMPILE}readelf -a $1 | grep "Shared library"
```

Linking with libraries – static and dynamic linking

Any application you write for Linux, whether it be in C or C++, will be linked with the C library `libc`. This is so fundamental that you don't even have to tell `gcc` or `g++` to do it because it always links `libc`. Other libraries that you may want to link with have to be explicitly named through the `-l` option.

The library code can be linked in two different ways: statically, meaning that all the library functions your application calls and their dependencies are pulled from the library archive and bound into your executable; and dynamically, meaning that references to the library files and functions in those files are generated in the code but the actual linking is done dynamically at runtime. You will find the code for the examples that follow in the book code archive in

`MELP/chapter_02/library`.

Static libraries

Static linking is useful in a few circumstances. For example, if you are building a small system which consists of only BusyBox and some script files, it is simpler to link BusyBox statically and avoid having to copy the runtime library files and linker. It will also be smaller because you only link in the code that your application uses rather than supplying the entire C library. Static linking is also useful if you need to run a program before the filesystem that holds the runtime libraries is available.

You tell to link all the libraries statically by adding `-static` to the command line:

```
| $ arm-cortex_a8-linux-gnueabi-gcc -static helloworld.c -o helloworld-static
```

You will note that the size of the binary increases dramatically:

```
| $ ls -l
-rwxrwxr-x 1 chris chris 5884 Mar 5 09:56 helloworld
-rwxrwxr-x 1 chris chris 614692 Mar 5 10:27 helloworld-static
```

Static linking pulls code from a library archive, usually named `lib[name].a`. In the preceding case, it is `libc.a`, which is in `[sysroot]/usr/lib`:

```
| $ export SYSROOT=$(arm-cortex_a8-linux-gnueabi-gcc -print-sysroot)
$ cd $SYSROOT
$ ls -l usr/lib/libc.a
-rw-r--r-- 1 chris chris 3457004 Mar 3 15:21 usr/lib/libc.a
```

Note that the syntax `export SYSROOT=$(arm-cortex_a8-linux-gnueabi-gcc -print-sysroot)` places the path to the sysroot in the shell variable, `SYSROOT`, which makes the example a little clearer.

Creating a static library is as simple as creating an archive of object files using the `ar` command. If I have two source files named `test1.c` and `test2.c`, and I want to create a static library named `libtest.a`, then I would do the following:

```
| $ arm-cortex_a8-linux-gnueabi-gcc -c test1.c
$ arm-cortex_a8-linux-gnueabi-gcc -c test2.c
$ arm-cortex_a8-linux-gnueabi-ar rc libtest.a test1.o test2.o
$ ls -l
total 24
-rw-rw-r-- 1 chris chris 2392 Oct 9 09:28 libtest.a
-rw-rw-r-- 1 chris chris 116 Oct 9 09:26 test1.c
```

```
| -rw-rw-r-- 1 chris chris 1080 Oct 9 09:27 test1.o  
| -rw-rw-r-- 1 chris chris 121 Oct 9 09:26 test2.c  
| -rw-rw-r-- 1 chris chris 1088 Oct 9 09:27 test2.o
```

Then I could link libtest into my `helloworld` program, using:

```
| $ arm-cortex_a8-linux-gnueabi-gcc helloworld.c -ltest \  
| -L../libs -I../libs -o helloworld
```

Shared libraries

A more common way to deploy libraries is as shared objects that are linked at runtime, which makes more efficient use of storage and system memory, since only one copy of the code needs to be loaded. It also makes it easy to update the library files without having to re-link all the programs that use them.

The object code for a shared library must be position-independent, so that the runtime linker is free to locate it in memory at the next free address. To do this, add the `-fPIC` parameter to `gcc`, and then link it using the `-shared` option:

```
$ arm-cortex_a8-linux-gnueabi-hf-gcc -fPIC -c test1.c
$ arm-cortex_a8-linux-gnueabi-hf-gcc -fPIC -c test2.c
$ arm-cortex_a8-linux-gnueabi-hf-gcc -shared -o libtest.so test1.o test2.o
```

This creates the shared library, `libtest.so`. To link an application with this library, you add `-ltest`, exactly as in the static case mentioned in the preceding section, but this time the code is not included in the executable. Instead, there is a reference to the library that the runtime linker will have to resolve:

```
$ arm-cortex_a8-linux-gnueabi-hf-gcc helloworld.c -ltest \
-L../libs -I../libs -o helloworld
$ MELP/list-libs helloworld
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]
0x00000001 (NEEDED) Shared library: [libtest.so]
0x00000001 (NEEDED) Shared library: [libc.so.6]
```

The runtime linker for this program is `/lib/ld-linux-armhf.so.3`, which must be present in the target's filesystem. The linker will look for `libtest.so` in the default search path: `/lib` and `/usr/lib`. If you want it to look for libraries in other directories as well, you can place a colon-separated list of paths in the shell variable `LD_LIBRARY_PATH`:

```
| # export LD_LIBRARY_PATH=/opt/lib:/opt/usr/lib
```

Understanding shared library version numbers

One of the benefits of shared libraries is that they can be updated independently of the programs that use them. Library updates are of two types: those that fix bugs or add new functions in a backwards-compatible way, and those that break compatibility with existing applications. GNU/Linux has a versioning scheme to handle both these cases.

Each library has a release version and an interface number. The release version is simply a string that is appended to the library name; for example, the JPEG image library `libjpeg` is currently at release 8.0.2 and so the library is named `libjpeg.so.8.0.2`. There is a symbolic link named `libjpeg.so` to `libjpeg.so.8.0.2`, so that when you compile a program with `-ljpeg`, you link with the current version. If you install version 8.0.3, the link is updated and you will link with that one instead.

Now suppose that version 9.0.0. comes along and that breaks the backwards compatibility. The link from `libjpeg.so` now points to `libjpeg.so.9.0.0`, so that any new programs are linked with the new version, possibly throwing compile errors when the interface to `libjpeg` changes, which the developer can fix. Any programs on the target that are not recompiled are going to fail in some way, because they are still using the old interface. This is where an object known as the **soname** helps. The soname encodes the interface number when the library was built and is used by the runtime linker when it loads the library. It is formatted as `<library name>.so.<interface number>`. For `libjpeg.so.8.0.2`, the soname is `libjpeg.so.8`:

```
$ readelf -a /usr/lib/libjpeg.so.8.0.2 | grep SONAME
0x000000000000000e (SONAME) Library soname:
 [libjpeg.so.8]
```

Any program compiled with it will request `libjpeg.so.8` at runtime, which will be a symbolic link on the target to `libjpeg.so.8.0.2`. When version 9.0.0 of `libjpeg` is installed, it will have a soname of `libjpeg.so.9`, and so it is possible to have two incompatible versions of the same library installed on the same system.

Programs that were linked with `libjpeg.so.8.*.*` will load `libjpeg.so.8`, and those

linked with `libjpeg.so.9.*.*` will load `libjpeg.so.9`.

This is why, when you look at the directory listing of `<sysroot>/usr/lib/libjpeg*`, you find these four files:

- `libjpeg.a`: This is the library archive used for static linking
- `libjpeg.so -> libjpeg.so.8.0.2`: This is a symbolic link, used for dynamic linking
- `libjpeg.so.8 -> libjpeg.so.8.0.2`: This is a symbolic link, used when loading the library at runtime
- `libjpeg.so.8.0.2`: This is the actual shared library, used at both compile time and runtime

The first two are only needed on the host computer for building and the last two are needed on the target at runtime.

The art of cross compiling

Having a working cross toolchain is the starting point of a journey, not the end of it. At some point, you will want to begin cross compiling the various tools, applications, and libraries that you need on your target. Many of them will be open source packages—each of which has its own method of compiling and its own peculiarities. There are some common build systems, including:

- Pure `makefiles`, where the toolchain is usually controlled by the `make` variable `CROSS_COMPILE`
- The GNU build system known as **Autotools**
- CMake (<https://cmake.org/>)

I will cover only the first two here since these are the ones needed for even a basic embedded Linux system. For CMake, there are some excellent resources on the CMake website referenced in the preceding point.

Simple makefiles

Some important packages are very simple to cross compile, including the Linux kernel, the U-Boot bootloader, and BusyBox. For each of these, you only need to put the toolchain prefix in the make variable `CROSS_COMPILE`, for example `arm-cortex_a8-linux-gnueabi-`. Note the trailing dash `-`.

So, to compile BusyBox, you would type:

```
| $ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

Or, you can set it as a shell variable:

```
| $ export CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-  
| $ make
```

In the case of U-Boot and Linux, you also have to set the `make` variable `ARCH` to one of the machine architectures they support, which I will cover in [Chapter 3, All About Bootloaders](#), and [Chapter 4, Configuring and Building the Kernel](#).

Autotools

The name Autotools refers to a group of tools that are used as the build system in many open source projects. The components, together with the appropriate project pages, are:

- GNU Autoconf (<https://www.gnu.org/software/autoconf/autoconf.html>)
- GNU Automake (<https://www.gnu.org/savannah-checkouts/gnu/automake/>)
- GNU Libtool (<https://www.gnu.org/software/libtool/libtool.html>)
- Gnulib (<https://www.gnu.org/software/gnulib/>)

The role of Autotools is to smooth over the differences between the many different types of systems that the package may be compiled for, accounting for different versions of compilers, different versions of libraries, different locations of header files, and dependencies with other packages. Packages that use Autotools come with a script named `configure` that checks dependencies and generates makefiles according to what it finds. The `configure` script may also give you the opportunity to enable or disable certain features. You can find the options on offer by running `./configure --help`.

To configure, build, and install a package for the native operating system, you would typically run the following three commands:

```
$ ./configure
$ make
$ sudo make install
```

Autotools is able to handle cross development as well. You can influence the behavior of the `configure` script by setting these shell variables:

- `CC`: The C compiler command
- `CFLAGS`: Additional C compiler flags
- `LDFLAGS`: Additional linker flags; for example, if you have libraries in a non-standard directory `<lib dir>`, you would add it to the library search path by adding `-L<lib dir>`
- `LIBS`: Contains a list of additional libraries to pass to the linker; for instance, `-lm` for the math library

- **CPPFLAGS:** Contains C/C++ preprocessor flags; for example, you would add `-I<include dir>` to search for headers in a non-standard directory `<include dir>`
- **CPP:** The C preprocessor to use

Sometimes it is sufficient to set only the `cc` variable, as follows:

```
| $ CC=arm-cortex_a8-linux-gnueabi-hf-gcc ./configure
```

At other times, that will result in an error like this:

```
| [...]
| checking whether we are cross compiling... configure: error: in '/home/chris/MELP/build
| configure: error: cannot run C compiled programs.
| If you meant to cross compile, use '--host'.
| See 'config.log' for more details
```

The reason for the failure is that `configure` often tries to discover the capabilities of the toolchain by compiling snippets of code and running them to see what happens, which cannot work if the program has been cross compiled. Nevertheless, there is a hint in the error message on how to solve the problem. Autotools understands three different types of machines that may be involved when compiling a package:

- **Build** is the computer that builds the package, which defaults to the current machine.
- **Host** is the computer the program will run on; for a native compile, this is left blank and it defaults to be the same computer as build. When you are cross compiling, set it to be the tuple of your toolchain.
- **Target** is the computer the program will generate code for; you would set this when building a cross compiler, for example.

So, to cross compile, you just need to override the host, as follows:

```
| $ CC=arm-cortex_a8-linux-gnueabi-hf-gcc \
| ./configure --host=arm-cortex_a8-linux-gnueabi-hf
```

One final thing to note is that the default install directory is `<sysroot>/usr/local/*`. You would usually install it in `<sysroot>/usr/*`, so that the header files and libraries would be picked up from their default locations. The complete command to configure a typical Autotools package is as follows:

```
| $ CC=arm-cortex_a8-linux-gnueabi-hf-gcc \
```

```
| ./configure --host=arm-cortex_a8-linux-gnueabihf --prefix=/usr
```

An example: SQLite

The SQLite library implements a simple relational database and is quite popular on embedded devices. You begin by getting a copy of SQLite:

```
$ wget http://www.sqlite.org/2015/sqlite-autoconf-3081101.tar.gz
$ tar xf sqlite-autoconf-3081101.tar.gz
$ cd sqlite-autoconf-3081101
```

Next, run the `configure` script:

```
$ CC=arm-cortex_a8-linux-gnueabi-gcc \
./configure --host=arm-cortex_a8-linux-gnueabi --prefix=/usr
```

That seems to work! If it had failed, there would be error messages printed to the Terminal and recorded in `config.log`. Note that several makefiles have been created, so now you can build it:

```
| $ make
```

Finally, you install it into the toolchain directory by setting the `make` variable `DESTDIR`. If you don't, it will try to install it into the host computer's `/usr` directory, which is not what you want:

```
| $ make DESTDIR=$(arm-cortex_a8-linux-gnueabi-gcc -print-sysroot) install
```

You may find that the final command fails with a file permissions error. A crosstool-NG toolchain is read-only by default, which is why it is useful to set `CT_INSTALL_DIR_RO` to `y` when building it. Another common problem is that the toolchain is installed in a system directory, such as `/opt` or `/usr/local`, in which case you will need `root` permissions when running the install.

After installing, you should find that various files have been added to your toolchain:

- `<sysroot>/usr/bin: sqlite3`: This is a command-line interface for SQLite that you can install and run on the target
- `<sysroot>/usr/lib: libsqlite3.so.0.8.6, libsqlite3.so.0, libsqlite3.so, libsqlite3.la,`

`libsqlite3.a`: These are the shared and static libraries

- `<sysroot>/usr/lib/pkgconfig: sqlite3.pc`: This is the package configuration file, as described in the following section
- `<sysroot>/usr/lib/include: sqlite3.h, sqlite3ext.h`: These are the header files
- `<sysroot>/usr/share/man/man1: sqlite3.1`: This is the manual page

Now you can compile programs that use `sqlite3` by adding `-lsqlite3` at the link stage:

```
| $ arm-cortex_a8-linux-gnueabi-gcc -lsqlite3 sqlite-test.c -o sqlite-test
```

Here, `sqlite-test.c` is a hypothetical program that calls SQLite functions. Since `sqlite3` has been installed into the `sysroot`, the compiler will find the header and library files without any problem. If they had been installed elsewhere, you would have had to add `-L<lib dir>` and `-I<include dir>`.

Naturally, there will be runtime dependencies as well, and you will have to install the appropriate files into the target directory as described in [Chapter 5, Building a Root Filesystem](#).

Package configuration

Tracking package dependencies is quite complex. The package configuration utility `pkg-config` (<https://www.freedesktop.org/wiki/Software/pkg-config/>) helps track which packages are installed and which compile flags each needs by keeping a database of Autotools packages in `[sysroot]/usr/lib/pkgconfig`. For instance, the one for SQLite3 is named `sqlite3.pc` and contains essential information needed by other packages that need to make use of it:

```
$ cat $(arm-cortex_a8-linux-gnueabi-gcc -print-sysroot)/usr/lib/pkgconfig/sqlite3.pc
# Package Information for pkg-config

prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include

Name: SQLite
Description: SQL database engine
Version: 3.8.11.1
Libs: -L${libdir} -lsqlite3
Libs.private: -ldl -lpthread
Cflags: -I${includedir}
```

You can use `pkg-config` to extract information in a form that you can feed straight to `gcc`. In the case of a library like `libsqlite3`, you want to know the library name (`--libs`) and any special C flags (`--cflags`):

```
$ pkg-config sqlite3 --libs --cflags
Package sqlite3 was not found in the pkg-config search path.
Perhaps you should add the directory containing `sqlite3.pc'
to the PKG_CONFIG_PATH environment variable
No package 'sqlite3' found
```

Oops! That failed because it was looking in the host's `sysroot` and the development package for `libsqlite3` has not been installed on the host. You need to point it at the `sysroot` of the target toolchain by setting the shell variable

`PKG_CONFIG_LIBDIR`:

```
$ export PKG_CONFIG_LIBDIR=$(arm-cortex_a8-linux-gnueabi-gcc \
-print-sysroot)/usr/lib/pkgconfig
$ pkg-config sqlite3 --libs --cflags -lsqlite3
```

Now the output is `-lsqlite3`. In this case, you knew that already, but generally you

wouldn't, so this is a valuable technique. The final commands to compile would be:

```
$ export PKG_CONFIG_LIBDIR=$(arm-cortex_a8-linux-gnueabi-gcc \
-print-sysroot)/usr/lib/pkgconfig
$ arm-cortex_a8-linux-gnueabi-gcc $(pkg-config sqlite3 --cflags --libs) \
sqlite-test.c -o sqlite-test
```

Problems with cross compiling

The `sqlite3` is a well-behaved package and cross compiles nicely, but not all packages are the same. Typical pain points include:

- Home-grown build systems; `zlib`, for example, has a configure script, but it does not behave like the Autotools configure described in the previous section
- Configure scripts that read `pkg-config` information, headers, and other files from the host, disregarding the `--host` override
- Scripts that insist on trying to run cross compiled code

Each case requires careful analysis of the error and additional parameters to the configure script to provide the correct information, or patches to the code to avoid the problem altogether. Bear in mind that one package may have many dependencies, especially with programs that have a graphical interface using GTK or QT, or that handle multimedia content. As an example, `mplayer`, which is a popular tool for playing multimedia content, has dependencies on over 100 libraries. It would take weeks of effort to build them all.

Therefore, I would not recommend manually cross compiling components for the target in this way, except when there is no alternative or the number of packages to build is small. A much better approach is to use a build tool such as Buildroot or the Yocto Project, or avoid the problem altogether by setting up a native build environment for your target architecture. Now you can see why distributions like Debian are always compiled natively.

Summary

The toolchain is always your starting point; everything that follows from that is dependent on having a working, reliable toolchain.

Most embedded build environments are based on a cross development toolchain, which creates a clear separation between a powerful host computer building the code and a target computer on which it runs. The toolchain itself consists of the GNU binutils, a C compiler from the GNU compiler collection—and quite likely the C++ compiler as well—plus one of the C libraries I have described. Usually, the GNU debugger, GDB, will be generated at this point, which I describe in [Chapter 14, *Debugging with GDB*](#). Also, keep a watch out for the Clang compiler, as it will develop over the next few years.

You may start with nothing but a toolchain—perhaps built using crosstool-NG or downloaded from Linaro—and use it to compile all the packages that you need on your target, accepting the amount of hard work this will entail. Or you may obtain the toolchain as part of a distribution which includes a range of packages. A distribution can be generated from source code using a build system such as Buildroot or the Yocto Project, or it can be a binary distribution from a third party, maybe a commercial enterprise like Mentor Graphics, or an open source project such as the Denx ELDK. Beware of toolchains or distributions that are offered to you for free as part of a hardware package; they are often poorly configured and not maintained. In any case, you should make your choice according to your situation, and then be consistent in its use throughout the project.

Once you have a toolchain, you can use it to build the other components of your embedded Linux system. In the next chapter, you will learn about the bootloader, which brings your device to life and begins the boot process.

All About Bootloaders

The bootloader is the second element of embedded Linux. It is the part that starts the system up and loads the operating system kernel. In this chapter, I will look at the role of the bootloader and, in particular, how it passes control from itself to the kernel using a data structure called a **device tree**, also known as a **flattened device tree** or **FDT**. I will cover the basics of device trees, so that you will be able to follow the connections described in a device tree and relate it to real hardware.

I will look at the popular open source bootloader, U-Boot, and show you how to use it to boot a target device, and also how to customize it to run on a new device, using the BeagleBone Black as an example. Finally, I will take a quick look at Barebox, a bootloader that shares its past with U-Boot, but which has, arguably, a cleaner design.

In this chapter, we will cover the following topics:

- What does a bootloader do?
- The boot sequence.
- Booting with UEFI firmware.
- Moving from bootloader to kernel.
- Introducing device trees.
- Choosing a bootloader.
- U-Boot.
- Barebox.

What does a bootloader do?

khởi tạo

In an embedded Linux system, the bootloader has two main jobs: to initialize the system to a basic level and to load the kernel. In fact, the first job is somewhat subsidiary to the second, in that it is only necessary to get as much of the system working as is needed to load the kernel.

thực thi

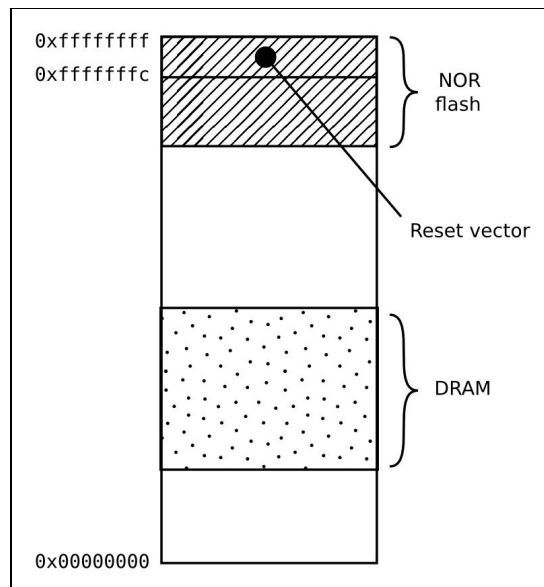
When the first lines of the bootloader code are executed, following a power-on or a reset, the system is in a very ^{trạng thái rất nhỏ} minimal state. The DRAM controller would not have been set up, and so the main memory would not be accessible. Likewise, other interfaces would not have been configured, so storage accessed via NAND flash controllers, MMC controllers, and so on, would also not be usable.

Typically, the only resources operational at the beginning are a single CPU core and some on-chip static memory. As a result, system bootstrap consists of several phases of code, each bringing more of the system into operation. The final act of the bootloader is to load the kernel into RAM and create an execution environment for it. The details of the interface between the bootloader and the kernel are architecture-specific, but in each case it has to do two things. First, bootloader has to pass a pointer to a structure containing information about the hardware configuration, and second it has to pass a pointer to the kernel command line. The kernel command line is a text string that controls the behavior of Linux. Once the kernel has begun executing, the bootloader is no longer needed and all the memory it was using can be reclaimed.

A subsidiary job of the bootloader is to provide a maintenance mode for updating boot configurations, loading new boot images into memory, and, maybe, running diagnostics. This is usually controlled by a simple command-line user interface, commonly over a serial interface.

The boot sequence

In simpler times, some years ago, it was only necessary to place the bootloader in non-volatile memory at the reset vector of the processor. **NOR flash** memory was common at that time and, since it can be mapped directly into the address space, it was the ideal method of storage. The following diagram shows such a configuration, with the **Reset vector** at `0xffffffffc` at the top end of an area of flash memory. The bootloader is linked so that there is a jump instruction at that location that points to the start of the bootloader code:



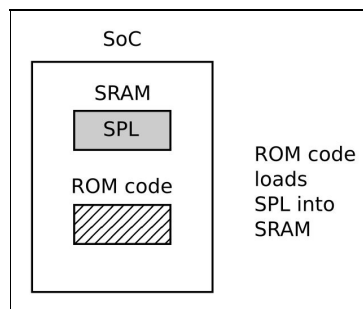
From that point, the bootloader code running in NOR flash memory can initialize the DRAM controller, so that the main memory, the DRAM, becomes available and then it copies itself into the **DRAM**. Once fully operational, the bootloader can load the kernel from flash memory into **DRAM** and transfer control to it.

However, once you move away from a simple linearly addressable storage medium like **NOR flash**, the boot sequence becomes a complex, multi-stage procedure. The details are very specific to each SoC, but they generally follow each of the following phases.

Phase 1 – ROM code

In the absence of reliable external memory, the code that runs immediately after a reset or power-on has to be stored on-chip in the SoC; this is known as **ROM code**. It is loaded into the chip when it is manufactured, and hence the ROM code is proprietary and cannot be replaced by an open source equivalent. Usually, it does not include code to initialize the memory controller, since DRAM configurations are highly device-specific, and so it can only use **Static Random Access Memory (SRAM)**, which does not require a memory controller.

Most embedded SoC designs have a small amount of SRAM on-chip, varying in size from as little as 4 KB to several hundred KB:

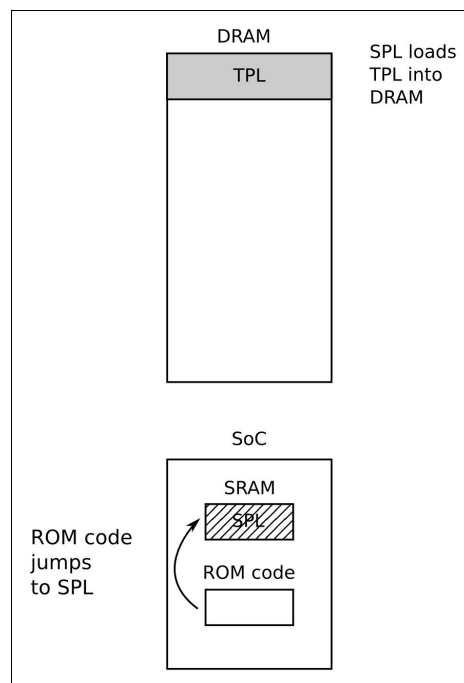


The ROM code is capable of loading a small chunk of code from one of several pre-programmed locations into the SRAM. As an example, TI OMAP and Sitara chips try to load code from the first few pages of NAND flash memory, or from flash memory connected through a **Serial Peripheral Interface (SPI)**, or from the first sectors of an MMC device (which could be an eMMC chip or an SD card), or from a file named `ML0` on the first partition of an MMC device. If reading from all of these memory devices fails, then it tries reading a byte stream from Ethernet, USB, or UART; the latter is provided mainly as a means of loading code into flash memory during production, rather than for use in normal operation. Most embedded SoCs have a ROM code that works in a similar way. In SoCs where the SRAM is not large enough to load a full bootloader like U-Boot, there has to be an intermediate loader called the **secondary program loader**, or **SPL**.

At the end of the ROM code phase, the SPL is present in the SRAM and the ROM code jumps to the beginning of that code.

Phase 2 – secondary program loader

The SPL must set up the memory controller and other essential parts of the system preparatory to loading the **Tertiary Program Loader (TPL)** into DRAM. The functionality of the SPL is limited by the size of the SRAM. It can read a program from a list of storage devices, as can the ROM code, once again using pre-programmed offsets from the start of a flash device. If the SPL has file system drivers built in, it can read well known file names, such as `u-boot.img`, from a disk partition. The SPL usually doesn't allow for any user interaction, but it may print version information and progress messages, which you can see on the console. The following diagram explains the phase 2 architecture:



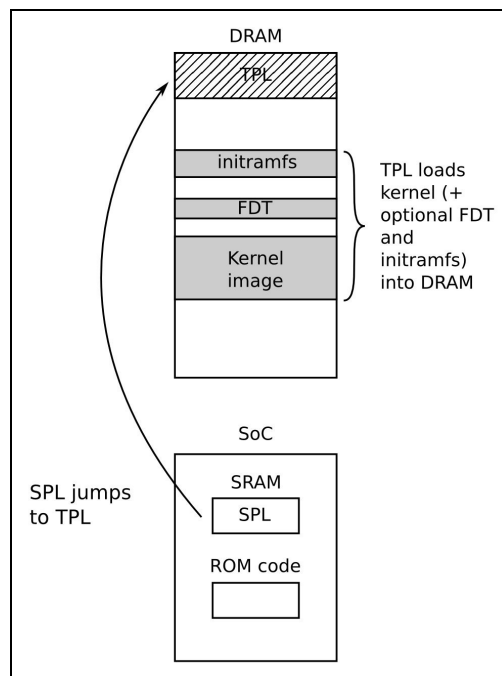
The SPL may be open source, as is the case with the TI x-loader and Atmel AT91Bootstrap, but it is quite common for it to contain proprietary code that is supplied by the manufacturer as a binary blob.

At the end of the second phase, the TPL is present in DRAM, and the SPL can make a jump to that area.

Phase 3 – TPL

Now, at last, we are running a full bootloader, such as U-Boot or BareBox. Usually, there is a simple command-line user interface that lets you perform maintenance tasks, such as loading new boot and kernel images into flash storage, and loading and booting a kernel, and there is a way to load the kernel automatically without user intervention.

The following diagram explains the phase 3 architecture:



At the end of the third phase, there is a kernel in memory, waiting to be started.

Embedded bootloaders usually disappear from memory once the kernel is running, and perform no further part in the operation of the system.

Booting with UEFI firmware

Most embedded x86/x86_64 designs, and some ARM designs, have firmware based on the **Universal Extensible Firmware Interface (UEFI)** standard. You can take a look at the UEFI website at <http://www.uefi.org/> for more information. The boot sequence is fundamentally the same as that described in the preceding section:

- **Phase 1:** The processor loads the platform initialization firmware from flash memory. In some designs, it is loaded directly from NOR flash memory, while in others, there is ROM code on-chip which loads the firmware from SPI flash memory into some on-chip static RAM.
- **Phase 2:** The platform initialization firmware performs the role of SPL. It initializes the DRAM controller and other system interfaces, so as to be able to load an EFI boot manager from the **EFI System Partition (ESP)** on a local disk, or from a network server via PXE boot. The ESP must be formatted using FAT16 or FAT32 format and it should have the well-known GUID value of C12A7328-F81F-11D2-BA4B-00A0C93EC93B. The path name of the boot manager code must follow the naming convention `<efi_system_partition>/boot/boot<machine_type_short_name>.efi`. For example, the file path to the loader on an x86_64 system would be `/efi/boot/bootx64.efi`.
- **Phase 3:** The UEFI boot manager is the tertiary program loader. The TPL in this case has to be a bootloader that is capable of loading a Linux kernel and an optional RAM disk into memory. Common choices are:
 - `systemd-boot`: This used to be called `gummiboot`. It is a simple UEFI-compatible bootloader, licensed under LGPL v2.1. The website is <https://www.freedesktop.org/wiki/Software/systemd/systemd-boot/>.
 - `Tummiboot`: This is the `gummiboot` with trusted boot support (Intel's **Trusted Execution Technology (TEX)**).

Moving from bootloader to kernel

When the bootloader passes control to the kernel it has to pass some basic information, which may include some of the following:

- The *machine number*, which is used on PowerPC, and ARM platforms without support for a device tree, to identify the type of the SoC
- Basic details of the hardware detected so far, including at least the size and location of the physical RAM, and the CPU clock speed
- The kernel command line
- Optionally, the location and size of a device tree binary
- Optionally, the location and size of an initial RAM disk, called the **initial RAM file system (initramfs)**

The kernel command line is a plain ASCII string which controls the behavior of Linux by giving, for example, the name of the device that contains the root filesystem. I will look at the details of this in the next chapter. It is common to provide the root filesystem as a RAM disk, in which case it is the responsibility of the bootloader to load the RAM disk image into memory. I will cover the way you create initial RAM disks in [Chapter 5, Building a Root Filesystem](#).

The way this information is passed is dependent on the architecture and has changed in recent years. For instance, with PowerPC, the bootloader simply used to pass a pointer to a board information structure, whereas, with ARM, it passed a pointer to a list of *A tags*. There is a good description of the format of A tags in the kernel source in `Documentation/arm/Bootimg`.

In both cases, the amount of information passed was very limited, leaving the bulk of it to be discovered at runtime or hard-coded into the kernel as **platform data**. The widespread use of platform data meant that each board had to have a kernel configured and modified for that platform. A better way was needed, and that way is the device tree. In the ARM world, the move away from A tags began in earnest in February 2013 with the release of Linux 3.8. Today, almost all ARM systems use device tree to gather information about the specifics of the hardware platform, allowing a single kernel binary to run on a wide range of

those platforms.

Introducing device trees

If you are working with ARM or PowerPC SoCs, you are almost certainly going to encounter device trees at some point. This section aims to give you a quick overview of what they are and how they work, but there are many details that are not discussed.

A device tree is a flexible way to define the hardware components of a computer system. Usually, the device tree is loaded by the bootloader and passed to the kernel, although it is possible to bundle the device tree with the kernel image itself to cater for bootloaders that are not capable of loading them separately.

The format is derived from a Sun Microsystems bootloader known as **OpenBoot**, which was formalized as the Open Firmware specification, which is IEEE standard IEEE1275-1994. It was used in PowerPC-based Macintosh computers and so was a logical choice for the PowerPC Linux port. Since then, it has been adopted on a large scale by the many ARM Linux implementations and, to a lesser extent, by MIPS, MicroBlaze, ARC, and other architectures.

I would recommend visiting <https://www.devicetree.org/> for more information.

Device tree basics

The Linux kernel contains a large number of device tree source files in `arch/$ARCH/boot/dts`, and this is a good starting point for learning about device trees. There are also a smaller number of sources in the U-boot source code in `arch/$ARCH/dts`. If you acquired your hardware from a third party, the `dts` file forms part of the board support package and you should expect to receive one along with the other source files.

The device tree represents a computer system as a collection of components joined together in a hierarchy, like a tree. The device tree begins with a root node, represented by a forward slash, `/`, which contains subsequent nodes representing the hardware of the system. Each node has a name and contains a number of properties in the form `name = "value"`. Here is a simple example:

```
/dts-v1/;
/{
    model = "TI AM335x BeagleBone";
    compatible = "ti,am33xx";
    #address-cells = <1>;
    #size-cells = <1>;
    cpus {
        #address-cells = <1>;
        #size-cells = <0>;
        cpu@0 {
            compatible = "arm,cortex-a8";
            device_type = "cpu";
            reg = <0>;
        };
    };
    memory@0x80000000 {
        device_type = "memory";
        reg = <0x80000000 0x20000000>; /* 512 MB */
    };
};
```

Here we have a root node which contains a `cpus` node and a memory node. The `cpus` node contains a single CPU node named `cpu@0`. It is a common convention that the names of nodes include an `@` followed by an address that distinguishes this node from other nodes of the same type.

Both the root and CPU nodes have a `compatible` property. The Linux kernel uses this property to find a matching device driver by comparing it with the strings exported by each device driver in a structure `of_device_id` (more on this in [Chapter 9](#),

Interfacing with Device Drivers).

It is a convention that the value is composed of a manufacturer name and a component name, to reduce confusion between similar devices made by different manufacturers; hence, `ti,am33xx` and `arm,cortex-a8`. It is also quite common to have more than one value for the `compatible` property where there is more than one driver that can handle this device. They are listed with the most suitable first.

The CPU node and the memory node have a `device_type` property which describes the class of device. The node name is often derived from `device_type`.

The reg property

The `memory` and `cpu` nodes have a `reg` property, which refers to a range of units in a register space. A `reg` property consists of two values representing the start address and the size (length) of the range. Both are written as zero or more 32-bit integers, called cells. Hence, the `memory` node refers to a single bank of memory that begins at `0x80000000` and is `0x20000000` bytes long.

Understanding `reg` properties becomes more complex when the address or size values cannot be represented in 32 bits. For example, on a device with 64-bit addressing, you need two cells for each:

```
/ {
    #address-cells = <2>;
    #size-cells = <2>;
    memory@80000000 {
        device_type = "memory";
        reg = <0x00000000 0x80000000 0 0x80000000>;
    };
};
```

The information about the number of cells required is held in the `#address-cells` and `#size-cells` properties in an ancestor node. In other words, to understand a `reg` property, you have to look backwards down the node hierarchy until you find `#address-cells` and `#size-cells`. If there are none, the default values are 1 for each—but it is bad practice for device tree writers to depend on fall-backs.

Now, let's return to the `cpu` and `cpus` nodes. CPUs have addresses as well; in a quad core device, they might be addressed as 0, 1, 2, and 3. That can be thought of as a one-dimensional array without any depth, so the size is zero. Therefore, you can see that we have `#address-cells = <1>` and `#size-cells = <0>` in the `cpus` node, and in the child node, `cpu@0`, we assign a single value to the `reg` property, `reg = <0>`.

Labels and interrupts

The structure of the device tree described so far assumes that there is a single hierarchy of components, whereas in fact there are several. As well as the obvious data connection between a component and other parts of the system, it might also be connected to an interrupt controller, to a clock source, and to a voltage regulator. To express these connections, we can add a label to a node and reference the label from other nodes. These labels are sometimes referred to as **phandles**, because when the device tree is compiled, nodes with a reference from another node are assigned a unique numerical value in a property called **phandle**. You can see them if you decompile the device tree binary.

Take as an example a system containing an LCD controller which can generate interrupts and an interrupt-controller:

```
/dts-v1/;
{
    intc: interrupt-controller@48200000 {
        compatible = "ti,am33xx-intc";
        interrupt-controller;
        #interrupt-cells = <1>;
        reg = <0x48200000 0x1000>;
    };

    lcdc: lcdc@4830e000 {
        compatible = "ti,am33xx-tilcdc";
        reg = <0x4830e000 0x1000>;
        interrupt-parent = <&intc>;
        interrupts = <36>;
        ti,hwmods = "lcdc";
        status = "disabled";
    };
};
```

Here we have node `interrupt-controller@48200000` with the label `intc`. The `interrupt-controller` property identifies it as an interrupt controller. Like all interrupt controllers, it has an `#interrupt-cells` property, which tells us how many cells are needed to represent an interrupt source. In this case, there is only one which represents the **interrupt request (IRQ)** number. Other interrupt controllers may use additional cells to characterize the interrupt, for example to indicate whether it is edge or level triggered. The number of interrupt cells and their meanings is described in the bindings for each interrupt controller. The device tree bindings can be found in the Linux kernel source, in the directory

Documentation/devicetree/bindings/.

Looking at the `lcdc@4830e000` node, it has an `interrupt-parent` property, which references the interrupt controller it is connected to, using the label. It also has an `interrupts` property, ³⁶ in this case. Note that this node has its own label, `lcdc`, which is used elsewhere: any node can have a label.

Device tree include files

A lot of hardware is common between SoCs of the same family and between boards using the same SoC. This is reflected in the device tree by splitting out common sections into include files, usually with the extension `.dtsi`. The Open Firmware standard defines `/include/` as the mechanism to be used, as in this snippet from `vexpress-v2p-ca9.dts`:

```
| /include/ "vexpress-v2m.dtsi"
```

Look through the `.dts` files in the kernel, though, and you will find an alternative include statement that is borrowed from C, for example in `am335x-boneblack.dts`:

```
| #include "am33xx.dtsi"  
| #include "am335x-bone-common.dtsi"
```

Here is another example from `am33xx.dtsi`:

```
| #include <dt-bindings/gpio/gpio.h>  
| #include <dt-bindings/pinctrl/am33xx.h>
```

Lastly, `include/dt-bindings/pinctrl/am33xx.h` contains normal C macros:

```
| #define PULL_DISABLE (1 << 3)  
| #define INPUT_EN (1 << 5)  
| #define SLEWCTRL_SLOW (1 << 6)  
| #define SLEWCTRL_FAST 0
```

All of this is resolved if the device tree sources are built using the Kbuild system, which first runs them through the C pre-processor, CPP, where the `#include` and `#define` statements are processed into text that is suitable for the device tree compiler. The motivation is illustrated in the previous example; it means that the device tree sources can use the same definitions of constants as the kernel code.

When we include files, using either syntax, the nodes are overlaid on top of one another to create a composite tree in which the outer layers extend or modify the inner ones. For example, `am33xx.dtsi`, which is general to all `am33xx` SoCs, defines the first MMC controller interface like this:

```
| mmc1: mmc@48060000 {
```

```

compatible = "ti,omap4-hsmmc";
ti,hwmods = "mmc1";
ti,dual-volt;
ti,needs-special-reset;
ti,needs-special-hs-handling;
dmas = <&edma 24 &edma 25>;
dma-names = "tx", "rx";
interrupts = <64>;
interrupt-parent = <&intc>;
reg = <0x48060000 0x1000>;
status = "disabled";
};

```

Note that the `status` is `disabled`, meaning that no device driver should be bound to it, and also that it has the label `mmc1`.

Both the BeagleBone and the BeagleBone Black have a microSD card interface attached to `mmc1`, hence in `am335x-bone-common.dtsi`, the same node is referenced by its label, `&mmc1`:

```

&mmc1 {
    status = "okay";
    bus-width = <0x4>;
    pinctrl-names = "default";
    pinctrl-0 = <&mmc1_pins>;
    cd-gpios = <&gpio0 6 GPIO_ACTIVE_HIGH>;
    cd-inverted;
};

```

The `status` property is set to `okay`, which causes the `mmc` device driver to bind with this interface at runtime on both variants of the BeagleBone. Also, a label is added to the pin control configuration, `mmc1_pins`. Alas, there is not sufficient space here to describe pin control and pin multiplexing. You will find some information in the Linux kernel source in directory `devicetree/bindings/pinctrl`.

However, interface `mmc1` is connected to a different voltage regulator on the BeagleBone Black. This is expressed in `am335x-boneblack.dts`, where you will see another reference to `mmc1`, which associates it with the voltage regulator via label `vmmc1sd_fixed`:

```

&mmc1 {
    vmmc-supply = <&vmmc1sd_fixed>;
};

```

So, layering device tree source files like this gives flexibility and reduces the need for duplicated code.

Compiling a device tree

The bootloader and kernel require a binary representation of the device tree, so it has to be compiled using the device tree compiler, `dtc`. The result is a file ending with `.dtb`, which is referred to as a device tree binary or a device tree blob.

There is a copy of `dtc` in the Linux source, in `scripts/dtc/dtc`, and it is also available as a package on many Linux distributions. You can use it to compile a simple device tree (one that does not use `#include`) like this:

```
$ dtc simpledts-1.dts -o simpledts-1.dtb
DTC: dts->dts on file "simpledts-1.dts"
```

Be wary of the fact that `dtc` does not give helpful error messages and it makes no checks other than on the basic syntax of the language, which means that debugging a typing error in a source file can be a lengthy business.

To build more complex examples, you will have to use the kernel Kbuild, as shown in the next chapter.

Choosing a bootloader

Bootloaders come in all shapes and sizes. The kind of characteristics you want from a bootloader are that they be simple and customizable with lots of sample configurations for common development boards and devices. The following table shows a number of bootloaders that are in general use:

Name	Main architectures supported
Das U-Boot	ARC, ARM, Blackfin, Microblaze, MIPS, Nios2, OpenRisc, PowerPC, SH
Barebox	ARM, Blackfin, MIPS, Nios2, PowerPC
GRUB 2	X86, X86_64
Little Kernel	ARM
RedBoot	ARM, MIPS, PowerPC, SH
CFE	Broadcom MIPS
YAMON	MIPS

We are going to focus on U-Boot because it supports a good number of processor architectures and a large number of individual boards and devices. It has been around for a long time and has a good community for support.

It may be that you received a bootloader along with your SoC or board. As always, take a good look at what you have and ask questions about where you can get the source code from, what the update policy is, how they will support you if you want to make changes, and so on. You may want to consider abandoning the vendor-supplied loader and using the current version of an open source bootloader instead.

U-Boot

U-Boot, or to give its full name, **Das U-Boot**, began life as an open source bootloader for embedded PowerPC boards. Then, it was ported to ARM-based boards and later to other architectures, including MIPS and SH. It is hosted and maintained by Denx Software Engineering. There is plenty of information available, and a good place to start is <http://www.denx.de/wiki/U-Boot>. There is also a mailing list at `u-boot@lists.denx.de`.

Building U-Boot

Begin by getting the source code. As with most projects, the recommended way is to clone the `.git` archive and check out the tag you intend to use, which, in this case, is the version that was current at the time of writing:

```
$ git clone git://git.denx.de/u-boot.git
$ cd u-boot
$ git checkout v2017.01
```

Alternatively, you can get a tarball from <ftp://ftp.denx.de/pub/u-boot>.

There are more than 1,000 configuration files for common development boards and devices in the `configs/` directory. In most cases, you can make a good guess of which to use, based on the filename, but you can get more detailed information by looking through the per-board README files in the `board/` directory, or you can find information in an appropriate web tutorial or forum.

Taking the BeagleBone Black as an example, we find that there is a likely configuration file named `configs/am335x_boneblack_defconfig` and we find the text `The binary produced by this board supports ... Beaglebone Black` in the board README files for the `am335x` chip, `board/ti/am335x/README`. With this knowledge, building U-Boot for a BeagleBone Black is simple. You need to inform U-Boot of the prefix for your cross compiler by setting the make variable `CROSS_COMPILE`, and then selecting the configuration file using a command of the type `make [board]_defconfig`. Therefore, to build U-Boot using the Crosstool-NG compiler we created in [Chapter 2, Learning About Toolchains](#), you would type:

```
$ source MELP/chapter_02/set-path-arm-cortex_a8-linux-gnueabihf
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- am335x_boneblack_defconfig
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

The results of the compilation are:

- `u-boot`: U-Boot in ELF object format, suitable for use with a debugger
- `u-boot.map`: The symbol table
- `u-boot.bin`: U-Boot in raw binary format, suitable for running on your device

- `u-boot.img`: This is `u-boot.bin` with a U-Boot header added, suitable for uploading to a running copy of U-Boot
- `u-boot.srec`: U-Boot in Motorola S-record (**SRECORD** or **SRE**) format, suitable for transferring over a serial connection

The BeagleBone Black also requires a **secondary program loader (SPL)**, as described earlier. This is built at the same time and is named `MLO`:

```
$ ls -l MLO u-boot*
-rw-rw-r-- 1 chris chris 78416 Mar 9 10:13 u-boot/MLO
-rwxrwxr-x 1 chris chris 2943940 Mar 9 10:13 u-boot/u-boot
-rwxrwxr-x 1 chris chris 368348 Mar 9 10:13 u-boot/u-boot.bin
-rw-rw-r-- 1 chris chris 368412 Mar 9 10:13 u-boot/u-boot.img
-rw-rw-r-- 1 chris chris 520741 Mar 9 10:13 u-boot/u-boot.map
-rwxrwxr-x 1 chris chris 1105162 Mar 9 10:13 u-boot/u-boot.srec
```

The procedure is similar for other targets.

Installing U-Boot

Installing a bootloader on a board for the first time requires some outside assistance. If the board has a hardware debug interface, such as JTAG, it is usually possible to load a copy of U-Boot directly into RAM and set it running. From that point, you can use U-Boot commands to copy itself into flash memory. The details of this are very board-specific and outside the scope of this book.

Many SoC designs have a boot ROM built in, which can be used to read boot code from various external sources, such as SD cards, serial interfaces, or USB mass storage. This is the case with the `am335x` chip in the BeagleBone Black, which makes it easy to try out new software.

You will need an SD card reader to write the images to a card. There are two types: external readers that plug into a USB port, and the internal SD readers that are present on many laptops. A device name is assigned by Linux when a card is plugged into the reader. The command `lsblk` is a useful tool to find out which device has been allocated. For example, this is what I see when I plug a nominal 8 GB microSD card into my card reader:

```
$ lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda 8:0 0 477G 0 disk
├─sda1 8:1 0 500M 0 part /boot/efi
├─sda2 8:2 0 40M 0 part
├─sda3 8:3 0 3G 0 part
├─sda4 8:4 0 457.6G 0 part /
└─sda5 8:5 0 15.8G 0 part [SWAP]
sdb 8:16 1 7.2G 0 disk
└─sdb1 8:17 1 7.2G 0 part /media/chris/101F-5626
```

In this case, `sda` is my 512 GB hard drive and `sdb` is the microSD card. It has a single partition, `sdb1`, which is mounted as directory `/media/chris/101F-5626`.

Although the microSD card had 8 GB printed on the outside, it was only 7.2 GB on the inside. In part, this is because of the different units used. The advertised capacity is measured in Gigabytes, 10^9 , but the sizes reported by software are in Gibibytes, 2^{30} . Gigabytes are abbreviated GB, Gibibytes as GiB. The same applies for KB





and KiB, and MB and MiB. In this book, I have tried to use the right units. In the case of the SD card, it so happens that 8 Gigabytes is approximately 7.4 Gibibytes. The remaining discrepancy is because flash memory always has to reserve some space for bad block handling. This is a topic that I will return to in *Chapter 7, Creating a Storage Strategy*.

If I use the built-in SD card slot, I see this:

```
$ lsblk
NAME MAJ:MIN RM SIZE RO TYPE MOUNTPOINT
sda 8:0 0 477G 0 disk
├─sda1 8:1 0 500M 0 part /boot/efi
├─sda2 8:2 0 40M 0 part
├─sda3 8:3 0 3G 0 part
├─sda4 8:4 0 457.6G 0 part /
└─sda5 8:5 0 15.8G 0 part [SWAP]
mmcblk0 179:0 0 7.2G 0 disk
└─mmcblk0p1 179:1 0 7.2G 0 part /media/chris/101F-5626
```

In this case, the micro SD card appears as `mmcblk0` and the partition is `mmcblk0p1`. Note that the microSD card you use may have been formatted differently to this one and so you may see a different number of partitions with different mount points. When formatting an SD card, it is very important to be sure of its device name. You really don't want to mistake your hard drive for an SD card and format that instead. This has happened to me more than once. So, I have provided a shell script in the book's code archive named `MELP/format-sdcard.sh`, which has a reasonable number of checks to prevent you (and me) from using the wrong device name. The parameter is the device name of the microSD card, which would be `sdb` in the first example and `mmcblk0` in the second. Here is an example of its use:

```
| $ MELP/format-sdcard.sh mmcblk0
```

The script creates two partitions: the first is 64 MiB, formatted as `FAT32`, and will contain the bootloader, and the second is 1 GiB, formatted as `ext4`, which you will use in [Chapter 5, Building a Root Filesystem](#).

After you have formatted the microSD card, remove it from the card reader and then re-insert it so that the partitions are auto mounted. On current versions of Ubuntu, the two partitions should be mounted as `/media/[user]/boot` and `/media/[user]/rootfs`. Now you can copy the SPL and U-Boot to it like this:

```
| $ cp MLO u-boot.img /media/chris/boot
```

Finally, unmount it:

```
| $ sudo umount /media/chris/boot
```

Now, with no power on the BeagleBone board, insert the micro-SD card into the reader. Plug in the serial cable. A serial port should appear on your PC as `/dev/ttyUSB0`. Start a suitable terminal program, such as `gtkterm`, `minicom`, or `picocom`, and attach to the port at 115200 bps (bits per second) with no flow control. `gtkterm` is probably the easiest to setup and use:

```
| $ gtkterm -p /dev/ttyUSB0 -s 115200
```

Press and hold the Boot Switch button on the Beaglebone Black, power up the board using the external 5V power connector, and release the button after about 5 seconds. You should see a U-Boot prompt on the serial console:

```
| U-Boot#
```

Using U-Boot

In this section, I will describe some of the common tasks that you can use U-Boot to perform.

Usually, U-Boot offers a command-line interface over a serial port. It gives a Command Prompt which is customized for each board. In the examples, I will use `U-Boot#`. Typing `help` prints out all the commands configured in this version of U-Boot; typing `help <command>` prints out more information about a particular command.

The default command interpreter for the BeagleBone Black is quite simple. You cannot do command-line editing by pressing cursor left or right keys; there is no command completion by pressing the *Tab* key; and there is no command history by pressing the cursor up key. Pressing any of these keys will disrupt the command you are currently trying to type, and you will have to type *Ctrl + C* and start over again. The only line editing key you can safely use is the backspace. As an option, you can configure a different command shell called **Hush**, which has more sophisticated interactive support, including command-line editing.

The default number format is hexadecimal. Consider the following command as an example:

```
| nand read 82000000 400000 200000
```

This will read `0x200000` bytes from offset `0x400000` from the start of the NAND flash memory into RAM address `0x82000000`.

Environment variables

U-Boot uses environment variables extensively to store and pass information between functions and even to create scripts. Environment variables are simple `name=value` pairs that are stored in an area of memory. The initial population of variables may be coded in the board configuration header file, like this:

```
#define CONFIG_EXTRA_ENV_SETTINGS  
"myvar1=value1"  
"myvar2=value2"  
[...]
```

You can create and modify variables from the U-Boot command line using `setenv`. For example, `setenv foo bar` creates the variable `foo` with the value `bar`. Note that there is no `=` sign between the variable name and the value. You can delete a variable by setting it to a null string, `setenv foo`. You can print all the variables to the console using `printenv`, or a single variable using `printenv foo`.

If U-Boot has been configured with space to store the environment, you can use the `saveenv` command to save it. If there is raw NAND or NOR flash, then an erase block can be reserved for this purpose, often with another used for a redundant copy to guard against corruption. If there is eMMC or SD card storage, it can be stored in a reserved array of sectors, or in a file named `uboot.env` in a partition of the disk. Other options include storing in a serial EEPROM connected via an I2C or SPI interface or non-volatile RAM.

Boot image format

U-Boot doesn't have a filesystem. Instead, it tags blocks of information with a 64-byte header so that it can track the contents. You prepare files for U-Boot using the `mkimage` command. Here is a brief summary of its usage:

```
$ mkimage
Usage: mkimage -l image
-l ==> list image header information
mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d data_file[:data_file]
-A ==> set architecture to 'arch'
-O ==> set operating system to 'os'
-T ==> set image type to 'type'
-C ==> set compression type 'comp'
-a ==> set load address to 'addr' (hex)
-e ==> set entry point to 'ep' (hex)
-n ==> set image name to 'name'
-d ==> use image data from 'datafile'
-x ==> set XIP (execute in place)
mkimage [-D dtc_options] [-f fit-image.its|-F] fit-image
-D => set options for device tree compiler
-f => input filename for FIT source
Signing / verified boot not supported (CONFIG_FIT_SIGNATURE undefined)
mkimage -V ==> print version information and exit
```

For example, to prepare a kernel image for an ARM processor, the command is:

```
$ mkimage -A arm -O linux -T kernel -C gzip -a 0x80008000 -e 0x80008000 \
-n 'Linux' -d zImage uImage
```

Loading images

Usually, you will load images from removable storage, such as an SD card or a network. SD cards are handled in U-Boot by the mmc driver. A typical sequence to load an image into memory would be:

```
U-Boot# mmc rescan
U-Boot# fatload mmc 0:1 82000000 uimage
reading uimage
4605000 bytes read in 254 ms (17.3 MiB/s)
U-Boot# iminfo 82000000

## Checking Image at 82000000 ...
Legacy image found
Image Name: Linux-3.18.0C
reated: 2014-12-23 21:08:07 UTC
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 4604936 Bytes = 4.4 MiB
Load Address: 80008000
Entry Point: 80008000
Verifying Checksum ... OK
```

The command `mmc rescan` re-initializes the mmc driver, perhaps to detect that an SD card has recently been inserted. Next, `fatload` is used to read a file from a FAT-formatted partition on the SD card. The format is:

```
| fatload <interface> [<dev[:part]>] [<addr> [<filename> [bytes [pos]]]]|
```

If `<interface>` is `mmc`, as in our case, `<dev:part>` is the device number of the mmc interface counting from zero, and the partition number counting from one. Hence, `<0:1>` is the first partition on the first device. The memory location, `0x82000000`, is chosen to be in an area of RAM that is not being used at this moment. If we intend to boot this kernel, we have to make sure that this area of RAM will not be overwritten when the kernel image is decompressed and located at the runtime location, `0x80008000`.

To load image files over a network, you use the Trivial File Transfer Protocol (TFTP). This requires you to install a TFTP daemon, `tftpd`, on your development system and start it running. You also have to configure any firewalls between your PC and the target board to allow the TFTP protocol on UDP port 69 to pass through. The default configuration of TFTP allows access only to the directory `/var/lib/tftpboot`. The next step is to copy the files you want to transfer to the

target into that directory. Then, assuming that you are using a pair of static IP addresses, which removes the need for further network administration, the sequence of commands to load a set of kernel image files should look like this:

```
U-Boot# setenv ipaddr 192.168.159.42
U-Boot# setenv serverip 192.168.159.99
U-Boot# tftp 82000000 uImage
link up on port 0, speed 100, full duplex
Using cpsw device
TFTP from server 192.168.159.99; our IP address is 192.168.159.42
Filename 'uImage'.
Load address: 0x82000000
Loading:
#####
#####
#####
#####
#####
3 MiB/s
done
Bytes transferred = 4605000 (464448 hex)
```

Finally, let's look at how to program images into NAND flash memory and read them back, which is handled by the `nand` command. This example loads a kernel image via TFTP and programs it into flash:

```
U-Boot# tftpboot 82000000 uimage
U-Boot# nandeccl hw
U-Boot# nand erase 280000 400000

NAND erase: device 0 offset 0x280000, size 0x400000
Erasing at 0x660000 -- 100% complete.
OK
U-Boot# nand write 82000000 280000 400000

NAND write: device 0 offset 0x280000, size 0x400000
4194304 bytes written: OK
```

Now you can load the kernel from flash memory using the `nand read` command:

```
U-Boot# nand read 82000000 280000 400000
```

Booting Linux

The `bootm` command starts a kernel image running. The syntax is:

```
| bootm [address of kernel] [address of ramdisk] [address of dtb].
```

The address of the kernel image is necessary, but the address of `ramdisk` and `dtb` can be omitted if the kernel configuration does not need them. If there is `dtb` but no `initramfs`, the second address can be replaced with a dash (-). That would look like this:

```
| U-Boot# bootm 82000000 - 83000000
```

Automating the boot with U-Boot scripts

Plainly, typing a long series of commands to boot your board each time it is turned on is not acceptable. To automate the process, U-Boot stores a sequence of commands in environment variables. If the special variable named `bootcmd` contains a script, it is run at power-up after a delay of `bootdelay` seconds. If you watch this on the serial console, you will see the delay counting down to zero. You can press any key during this period to terminate the countdown and enter into an interactive session with U-Boot.

The way that you create scripts is simple, though not easy to read. You simply append commands separated by semicolons, which must be preceded by a *backslash* escape character. So, for example, to load a kernel image from an offset in flash memory and boot it, you might use the following command:

```
| setenv bootcmd nand read 82000000 400000 200000\;bootm 82000000
```

Porting U-Boot to a new board

Let's assume that your hardware department has created a new board called **Nova** that is based on the BeagleBone Black and that you need to port U-Boot to it. You will need to understand the layout of the U-Boot code and how the board configuration mechanism works. In this section, I will show you how to create a variant of an existing board—the BeagleBone Black—which you could go on to use as the basis for further customizations. There are quite a few files that need to be changed. I have put them together into a patch file in the code archive in `MELP/chapter_03/0001-BSP-for-Nova.patch`. You can simply apply that patch to a clean copy of U-Boot version 2017.01 like this:

```
$ cd u-boot
$ patch -p1 < MELP/chapter_03/0001-BSP-for-Nova.patch
```

If you want to use a different version of U-Boot, you will have to make some changes to the patch for it to apply cleanly.

The remainder of this section is a description of how the patch was created. If you want to follow along step-by-step, you will need a clean copy of U-Boot 2017.01 without the Nova BSP patch. The main directories we will be dealing with are:

- **arch:** Contains code specific to each supported architecture in directories `arm`, `mips`, `powerpc`, and so on. Within each architecture, there is a subdirectory for each member of the family; for example, in `arch/arm/cpu/`, there are directories for the architecture variants, including `amt926ejs`, `armv7`, and `armv8`.
- **board:** Contains code specific to a board. Where there are several boards from the same vendor, they can be collected together into a subdirectory. Hence, the support for the `am335x` evm board, on which the BeagleBone is based, is in `board/ti/am335x`.
- **common:** Contains core functions including the command shells and the commands that can be called from them, each in a file named `cmd_[command name].c`.
- **doc:** Contains several README files describing various aspects of U-Boot. If you are wondering how to proceed with your U-Boot port, this is a good

place to start.

- `include`: In addition to many shared header files, this contains the very important subdirectory `include/configs/` where you will find the majority of the board configuration settings.

The way that `Kconfig` extracts configuration information from `Kconfig` files and stores the total system configuration in a file named `.config` is described in some detail in [Chapter 4, *Configuring and Building the Kernel*](#). Each board has a default configuration stored in `configs/[board name]_defconfig`. For the Nova board, we can begin by making a copy of the configuration for the BeagleBone Black:

```
$ cp configs/am335x_boneblack_defconfig configs/nova_defconfig
```

Now edit `configs/nova_defconfig` and change line four from `CONFIG_TARGET_AM335X_EVM=y` to `CONFIG_TARGET_NOVA=y`:

```
1 CONFIG_ARM=y
2 CONFIG_AM33XX=y
3 # CONFIG_SPL_NAND_SUPPORT is not set
4 CONFIG_TARGET_NOVA=y
[...]
```

Note that `CONFIG_ARM=y` causes the contents of `arch/arm/Kconfig` to be included, and on line two, `CONFIG_AM33XX=y` causes `arch/arm/mach-omap2/am33xx/Kconfig` to be included.

Board-specific files

Each board has a subdirectory named `board/[board name]` OR `board/[vendor]/[board name]`, which should contain:

- `Kconfig`: Contains configuration options for the board
- `MAINTAINERS`: Contains a record of whether the board is currently maintained and, if so, by whom
- `Makefile`: Used to build the board-specific code
- `README`: Contains any useful information about this port of U-Boot; for example, which hardware variants are covered

In addition, there may be source files for board specific functions.

Our Nova board is based on a BeagleBone which, in turn, is based on a TI `am335x` EVM, so, we should take a copy of the `am335x` board files:

```
$ mkdir board/ti/nova
$ cp -a board/ti/am335x/* board/ti/nova
```

Next, edit `board/ti/nova/Kconfig` and set `SYS_BOARD` to `"nova"`, so that it will build the files in `board/ti/nova`, and set `SYS_CONFIG_NAME` to `"nova"` also, so that the configuration file used will be `include/configs/nova.h`:

```
1 if TARGET_NOVA
2
3 config SPL_ENV_SUPPORT
4 default y
5
6 config SPL_WATCHDOG_SUPPORT
7 default y
8
9 config SPL_YMODEM_SUPPORT
10 default y
11
12 config SYS_BOARD
13 default "nova"
14
15 config SYS_VENDOR
16 default "ti"
17
18 config SYS_SOC
19 default "am33xx"
20
21 config SYS_CONFIG_NAME
22 default "nova"
```



```
| [...]
```

There is one other file here that we need to change. The linker script placed at `board/ti/nova/u-boot.lds` has a hard-coded reference to `board/ti/am335x/built-in.o` on line 39. Change it as shown:

```
| 35      {  
| 36          *(__image_copy_start)  
| 37          *(.vectors)  
| 38          CPUDIR/start.o (.text*)  
| 39          board/ti/nova/built-in.o (.text*)  
| 40          *(.text*)  
| 41      }
```

Now we need to link the `Kconfig` file for Nova into the chain of `Kconfig` files. First, edit `arch/arm/Kconfig` and add a menu option for Nova, and then source its `Kconfig` file:

```
| [...]  
| 1069 source "board/ti/nova/Kconfig"  
| [...]
```

Then, edit `arch/arm/mach-omap2/am33xx/Kconfig` and add a configuration option for `TARGET_NOVA`:

```
| [...]  
| 21 config TARGET_NOVA  
| 22     bool "Support the Nova! board"  
| 23     select DM  
| 24     select DM_SERIAL  
| 25     select DM_GPIO  
| 26     select TI_I2C_BOARD_DETECT  
| 27     help  
| 28         The Nova target board  
| [...]
```

Configuring header files

Each board has a header file in `include/configs/` which contains the majority of the configuration information. The file is named by the `SYS_CONFIG_NAME` identifier in the board's `Kconfig`. The format of this file is described in detail in the `README` file at the top level of the U-Boot source tree. For the purposes of our Nova board, simply copy `include/configs/am335c_evm.h` to `include/configs/nova.h` and make a small number of changes, the most significant of which is to set a new Command Prompt so that we can identify this bootloader at run-time:

```
[...]
16 #ifndef __CONFIG_NOVA_H
17 #define __CONFIG_NOVA_H
[...]
38 #define CONFIG_SYS_LDSCRIPT          "board/ti/nova/u-boot.lds"
[...]
68 #undef CONFIG_SYS_PROMPT
69 #define CONFIG_SYS_PROMPT "nova!> "
[...]
421 #endif /* ! __CONFIG_NOVA_H */
```

Building and testing

To build for the Nova board, select the configuration you have just created:

```
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- distclean
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- nova_defconfig
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

Copy `MLO` and `u-boot.img` to the boot partition of the microSD card you created earlier and boot the board. You should see output like this (note the Command Prompt):

```
U-Boot SPL 2017.01-dirty (Apr 20 2017 - 16:48:38)
Trying to boot from MMC1MMC partition switch failed
*** Warning - MMC partition switch failed, using default environment

reading u-boot.img
reading u-boot.img

U-Boot 2017.01-dirty (Apr 20 2017 - 16:48:38 +0100)

CPU : AM335X-GP rev 2.0
I2C: ready
DRAM: 512 MiB
MMC: OMAP SD/MMC: 0, OMAP SD/MMC: 1
*** Warning - bad CRC, using default environment

<ethaddr> not set. Validating first E-fuse MAC
Net: cpsw, usb_ether
Press SPACE to abort autoboot in 2 seconds
nova!>
```

You can create a patch for all of these changes by checking them into Git and using the `git format-patch` command:

```
$ git add .
$ git commit -m "BSP for Nova"
[nova-bsp-2 e160f82] BSP for Nova
12 files changed, 2272 insertions(+)
create mode 100644 board/ti/nova/Kconfig
create mode 100644 board/ti/nova/MAINTAINERS
create mode 100644 board/ti/nova/Makefile
create mode 100644 board/ti/nova/README
create mode 100644 board/ti/nova/board.c
create mode 100644 board/ti/nova/board.h
create mode 100644 board/ti/nova/mux.c
create mode 100644 board/ti/nova/u-boot.lds
create mode 100644 configs/nova_defconfig
create mode 100644 include/configs/nova.h
$ git format-patch -1
0001-BSP-for-Nova.patch
```

Falcon mode

We are used to the idea that booting a modern embedded processor involves the CPU boot ROM loading an SPL, which loads `u-boot.bin` which then loads a Linux kernel. You may be wondering if there is a way to reduce the number of steps, thereby simplifying and speeding up the boot process. The answer is U-Boot **Falcon mode**. The idea is simple: have the SPL load a kernel image directly, missing out `u-boot.bin`. There is no user interaction and there are no scripts. It just loads a kernel from a known location in flash or eMMC into memory, passes it a pre-prepared parameter block, and starts it running. The details of configuring Falcon mode are beyond the scope of this book. If you would like more information, take a look at `doc/README.falcon`.



Falcon mode is named after the Peregrine falcon, which is the fastest bird of all, capable of reaching speeds of more than 200 miles per hour in a dive.

Barebox

I will complete this chapter with a look at another bootloader that has the same roots as U-Boot but takes a new approach to bootloaders. It is derived from U-Boot and was actually called U-Boot v2 in the early days. The barebox developers aimed to combine the best parts of U-Boot and Linux, including a POSIX-like API and mountable filesystems.

The barebox project website is <http://barebox.org/> and the developer mailing list is `barebox@lists.infradead.org`.

Getting barebox

To get barebox, clone the Git repository and check out the version you want to use:

```
$ git clone git://git.pengutronix.de/git/barebox.git
$ cd barebox
$ git checkout v2017.02.0
```

The layout of the code is similar to U-Boot:

- **arch:** Contains code specific to each supported architecture, which includes all the major embedded architectures. SoC support is in `arch/[architecture]/mach-[SoC]`. Support for individual boards is in `arch/[architecture]/boards`.
- **common:** Contains core functions, including the shell.
- **commands:** Contains the commands that can be called from the shell.
- **Documentation:** Contains the templates for documentation files. To build it, type `make docs`. The results are put in `Documentation/html`.
- **drivers:** Contains the code for the device drivers.
- **include:** Contains header files.

Building barebox

Barebox has used `kconfig/kbuild` for a long time. There are default configuration files in `arch/[architecture]/configs`. As an example, assume that you want to build barebox for the BeagleBoard C4. You need two configurations, one for the SPL, and one for the main binary. Firstly, build `MLO`:

```
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- \
am335x_mlo_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

The result is the secondary program loader, `images/barebox-am33xx-beaglebone-mlo.img`.

Next, build barebox:

```
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- \
am335x_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

Copy `MLO` and the barebox binary to an SD card:

```
$ cp images/barebox-am33xx-beaglebone-mlo.img /media/chris/boot/MLO
$ cp images/barebox-am33xx-beaglebone.img /media/chris/boot/barebox.bin
```

Then, boot up the board and you should see messages like these on the console:

```
barebox 2017.02.0 #1 Thu Mar 9 20:27:08 GMT 2017

Board: TI AM335x BeagleBone black
detected 'BeagleBone Black'
[...]
running /env/bin/init...
changing USB current limit to 1300 mA... done

Hit m for menu or any other key to stop autoboot: 3

type exit to get to the menu
barebox@TI AM335x BeagleBone black:/
```

Using barebox

Using barebox at the command line you can see the similarities with Linux. First, you can see that there are filesystem commands such as `ls`, and there is a `/dev` directory:

```
# ls /dev
full      mdio0-phy00 mem      mmc0      mmc0.0
mmc0.1    mmc1      mmc1.0  null      ram0      zero
```

The device `/dev/mmc0.0` is the first partition on the microSD card, which contains the kernel and initial ramdisk. You can mount it like this:

```
# mount /dev/mmc0.0 /mnt
```

Now you can see the files:

```
# ls /mnt
ML0 am335x-boneblack.dtb barebox.bin
u-boot.img uRamdisk zImage
```

Boot from the `root` partition:

```
# global.bootm.oftree=/mnt/am335x-boneblack.dtb
# global linux.bootargs.root="root=/dev/mmcblk0p2 rootwait"
# bootm /mnt/zImage
```


Summary

Every system needs a bootloader to bring the hardware to life and to load a kernel. U-Boot has found favor with many developers because it supports a useful range of hardware and it is fairly easy to port to a new device. Over the last few years, the complexity and ever increasing variety of embedded hardware has led to the introduction of the device tree as a way of describing hardware. The device tree is simply a textual representation of a system that is compiled into a **device tree binary (dtb)** and which is passed to the kernel when it loads. It is up to the kernel to interpret the device tree and to load and initialize drivers for the devices it finds there.

In use, U-Boot is very flexible, allowing images to be loaded from mass storage, flash memory, or a network, and booted. Likewise, barebox can achieve the same but with a smaller base of hardware support. Despite its cleaner design and POSIX-inspired internal APIs, at the time of writing it does not seem to have been accepted beyond its own small but dedicated community.

Having covered some of the intricacies of booting Linux, in the next chapter you will see the next stage of the process as the third element of your embedded project, the kernel, comes into play.

Configuring and Building the Kernel

The kernel is the third element of embedded Linux. It is the component that is responsible for managing resources and interfacing with hardware, and so affects almost every aspect of your final software build. It is usually tailored to your particular hardware configuration, although, as we saw in [Chapter 3, All About Bootloaders](#), device trees allow you to create a generic kernel that is tailored to particular hardware by the contents of the device tree.

In this chapter, we will look at how to get a kernel for a board, and how to configure and compile it. We will look again at bootstrap, this time focusing on the part the kernel plays. We will also look at device drivers and how they pick up information from the device tree.

In this chapter, we will cover the following topics:

- What does the kernel do?
- Choosing a kernel.
- Building the kernel.
- Booting the kernel.
- Porting Linux to a new board.

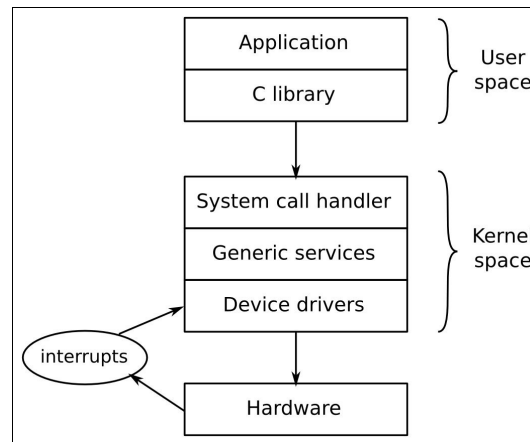
What does the kernel do?

Linux began in 1991, when Linus Torvalds started writing an operating system for Intel 386- and 486-based personal computers. He was inspired by the Minix operating system written by *Andrew S. Tanenbaum* four years earlier. Linux differed in many ways from Minix; the main differences being that it was a 32-bit virtual memory kernel and the code was open source, later released under the GPL v2 license. He announced it on 25th August, 1991, on the `comp.os.minix` newsgroup in a famous post that began with:

Hello everybody out there using minix—I'm doing a (free) operating system (just a hobby, won't be big and professional like GNU) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the filesystem (due to practical reasons) among other things).

To be strictly accurate, Linus did not write an operating system, rather he wrote a kernel, which is only one component of an operating system. To create a complete operating system with user space commands and a shell command interpreter, he used components from the GNU project, especially the toolchain, the C-library, and basic command-line tools. That distinction remains today, and gives Linux a lot of flexibility in the way it is used. It can be combined with a GNU user space to create a full Linux distribution that runs on desktops and servers, which is sometimes called GNU/Linux; it can be combined with an Android user space to create the well-known mobile operating system, or it can be combined with a small BusyBox-based user space to create a compact embedded system. Contrast this with the BSD operating systems, FreeBSD, OpenBSD, and NetBSD, in which the kernel, the toolchain, and the user space are combined into a single code base.

The kernel has three main jobs: to manage resources, to interface with hardware, and to provide an API that offers a useful level of abstraction to user space programs, as summarized in the following diagram:



Applications running in **User space** run at a low CPU privilege level. They can do very little other than make library calls. The primary interface between the **User space** and the **Kernel space** is the **C library**, which translates user level functions, such as those defined by POSIX, into kernel system calls. The system call interface uses an architecture-specific method, such as a trap or a software interrupt, to switch the CPU from low privilege user mode to high privilege kernel mode, which allows access to all memory addresses and CPU registers.

The **System call handler** dispatches the call to the appropriate kernel subsystem: memory allocation calls go to the memory manager, filesystem calls to the filesystem code, and so on. Some of those calls require input from the underlying hardware and will be passed down to a device driver. In some cases, the hardware itself invokes a kernel function by raising an interrupt.



The preceding diagram shows that there is a second entry point into kernel code: hardware interrupts. Interrupts can only be handled in a device driver, never by a user space application.

In other words, all the useful things that your application does, it does them through the kernel. The kernel, then, is one of the most important elements in the system.

Choosing a kernel

The next step is to choose the kernel for your project, balancing the desire to always use the latest version of software against the need for vendor-specific additions and an interest in the long term support of the code base.

Kernel development cycle

Linux is developed at a fast pace, with a new version being released every 8 to 12 weeks. The way that the version numbers are constructed has changed a bit in recent years. Before July 2011, there was a three number version scheme with version numbers that looked like 2.6.39. The middle number indicated whether it was a developer or stable release; odd numbers (2.1.x, 2.3.x, 2.5.x) were for developers and even numbers were for end users. From version 2.6 onwards, the idea of a long-lived development branch (the odd numbers) was dropped, as it slowed down the rate at which new features were made available to the users. The change in numbering from 2.6.39 to 3.0 in July 2011 was purely because Linus felt that the numbers were becoming too large; there was no huge leap in the features or architecture of Linux between those two versions. He also took the opportunity to drop the middle number. Since then, in April 2015, he bumped the major from 3 to 4, again purely for neatness, not because of any large architectural shift.

Linus manages the development kernel tree. You can follow him by cloning the Git tree like so:

```
| $ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

This will check out into subdirectory `linux`. You can keep up to date by running the command `git pull` in that directory from time to time.

Currently, a full cycle of kernel development begins with a merge window of two weeks, during which Linus will accept patches for new features. At the end of the merge window, a stabilization phase begins, during which Linus will produce weekly release candidates with version numbers ending in `-rc1`, `-rc2`, and so on, usually up to `-rc7` or `-rc8`. During this time, people test the candidates and submit bug reports and fixes. When all significant bugs have been fixed, the kernel is released.

The code incorporated during the merge window has to be fairly mature already. Usually, it is pulled from the repositories of the many subsystem and architecture maintainers of the kernel. By keeping to a short development cycle, features can

be merged when they are ready. If a feature is deemed not sufficiently stable or well developed by the kernel maintainers, it can simply be delayed until the next release.

Keeping a track of what has changed from release to release is not easy. You can read the commit log in Linus' Git repository but, with roughly 10,000 or more entries, it is not easy to get an overview. Thankfully, there is the Linux **Kernel Newbies** website, <http://kernelnewbies.org>, where you will find a succinct overview of each version at <http://kernelnewbies.org/LinuxVersions>.

Stable and long term support releases

The rapid rate of change of Linux is a good thing in that it brings new features into the mainline code base, but it does not fit very well with the longer life cycle of embedded projects. Kernel developers address this in two ways, with **stable** releases and **long term** releases. After the release of a mainline kernel (maintained by Linus Torvalds) it is moved to the **stable** tree (maintained by Greg Kroah-Hartman). Bug fixes are applied to the stable kernel, while the mainline kernel begins the next development cycle. Point releases of the stable kernel are marked by a third number, 3.18.1, 3.18.2, and so on. Before version 3, there were four release numbers, 2.6.29.1, 2.6.39.2, and so on.

You can get the stable tree by using the following command:

```
| $ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

You can use `git checkout` to get a particular version, for example version 4.9.13:

```
| $ cd linux-stable  
| $ git checkout v4.9.13
```

Usually, the stable kernel is updated only until the next mainline release (8 to 12 weeks later), so you will see that there is just one or sometimes two stable kernels at <https://www.kernel.org/>. To cater for those users who would like updates for a longer period of time and be assured that any bugs will be found and fixed, some kernels are labeled **long term** and maintained for two or more years. There is at least one long term kernel release each year. Looking at <https://www.kernel.org/> at the time of writing, there are a total of nine long term kernels: 4.9, 4.4, 4.1, 3.18, 3.14, 3.12, 3.10, 3.4, and 3.2. The latter has been maintained for five years and is at version 3.2.86. If you are building a product that you will have to maintain for this length of time, then the latest long term kernel might well be a good choice.

Vendor support

In an ideal world, you would be able to download a kernel from <https://www.kernel.org/> and configure it for any device that claims to support Linux. However, that is not always possible; in fact mainline Linux has solid support for only a small subset of the many devices that can run Linux. You may find support for your board or SoC from independent open source projects, Linaro or the Yocto Project, for example, or from companies providing third party support for embedded Linux, but in many cases you will be obliged to look to the vendor of your SoC or board for a working kernel. As we know, some are better at supporting Linux than others. My only advice at this point is to choose vendors who give good support or who, even better, take the trouble to get their kernel changes into the mainline.

Licensing

The Linux source code is licensed under GPL v2, which means that you must make the source code of your kernel available in one of the ways specified in the license.

The actual text of the license for the kernel is in the file COPYING. It begins with an addendum written by Linus that states that code calling the kernel from user space via the system call interface is not considered a derivative work of the kernel and so is not covered by the license. Hence, there is no problem with proprietary applications running on top of Linux.

However, there is one area of Linux licensing that causes endless confusion and debate: kernel modules. A kernel module is simply a piece of code that is dynamically linked with the kernel at runtime, thereby extending the functionality of the kernel. The GPL makes no distinction between static and dynamic linking, so it would appear that the source for kernel modules is covered by the GPL. But, in the early days of Linux, there were debates about exceptions to this rule, for example, in connection with the Andrew filesystem. This code predates Linux and therefore (it was argued) is not a derivative work, and so the license does not apply. Similar discussions took place over the years with respect to other pieces of code, with the result that it is now accepted practice that the GPL does not *necessarily* apply to kernel modules. This is codified by the kernel `MODULE_LICENSE` macro, which may take the value `Proprietary` to indicate that it is not released under the GPL. If you plan to use the same arguments yourself, you may want to read though an oft-quoted e-mail thread titled "*Linux GPL and binary module exception clause?*" which is archived at http://yarchive.net/comp/linux/gpl_modules.html.

The GPL should be considered a good thing because it guarantees that when you and I are working on embedded projects, we can always get the source code for the kernel. Without it, embedded Linux would be much harder to use and more fragmented.

Building the kernel

Having decided which kernel to base your build on, the next step is to build it.

Getting the source

Both of the targets used in this book, the BeagleBone Black and the ARM Versatile PB, are well supported by the mainline kernel. Therefore, it makes sense to use the latest long-term kernel available from <https://www.kernel.org/>, which at the time of writing was 4.9.13. When you come to do this for yourself, you should check to see if there is a later version of the 4.9 kernel and use that instead since it will have fixes for bugs found after 4.9.13 was released. If there is a later long-term release, you may want to consider using that one, but be aware that there may have been changes that mean that the following sequence of commands do not work exactly as given.

Use this command to clone the stable kernel and check out version 4.9.13:

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
$ cd linux-stable
$ git checkout v4.9.13
```

Alternatively, you could download the tar file from <https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.9.13.tar.xz>.

There is a lot of code here. There are over 57,000 files in the 4.9 kernel containing C-source code, header files, and assembly code, amounting to a total of over 14 million lines of code, as measured by the SLOCCount utility. Nevertheless, it is worth knowing the basic layout of the code and to know, approximately, where to look for a particular component. The main directories of interest are:

- **arch:** Contains architecture-specific files. There is one subdirectory per architecture.
- **Documentation:** Contains kernel documentation. Always look here first if you want to find more information about an aspect of Linux.
- **drivers:** Contains device drivers, thousands of them. There is a subdirectory for each type of driver.
- **fs:** Contains filesystem code.
- **include:** Contains kernel header files, including those required when building the toolchain.

- `init`: Contains the kernel start-up code.
- `kernel`: Contains core functions, including scheduling, locking, timers, power management, and debug/trace code.
- `mm`: Contains memory management.
- `net`: Contains network protocols.
- `scripts`: Contains many useful scripts, including the device tree compiler, DTC, which I described in [Chapter 3](#), *All About Bootloaders*.
- `tools`: Contains many useful tools and including the Linux performance counters tool, `perf`, which I will describe in [Chapter 15](#), *Profiling and Tracing*.

Over a period of time, you will become familiar with this structure, and realize that if you are looking for the code for the serial port of a particular SoC, you will find it in `drivers/tty/serial` and not in `arch/$ARCH/mach-foo`, because it is a device driver and not something central to the running of Linux on that SoC.

Understanding kernel configuration – Kconfig

One of the strengths of Linux is the degree to which you can configure the kernel to suit different jobs, from a small dedicated device such as a smart thermostat to a complex mobile handset. In current versions, there are many thousands of configuration options. Getting the configuration right is a task in itself but, before that, I want to show you how it works so that you can better understand what is going on.

The configuration mechanism is called `kconfig`, and the build system that it integrates with is called `kbuild`. Both are documented in `Documentation/kbuild`. `Kconfig/Kbuild` is used in a number of other projects as well as the kernel, including Crosstool-NG, U-Boot, Barebox, and BusyBox.

The configuration options are declared in a hierarchy of files named `kconfig`, using a syntax described in `Documentation/kbuild/kconfig-language.txt`. In Linux, the top level `kconfig` looks like this:

```
mainmenu "Linux/$ARCH $KERNELVERSION Kernel Configuration"

config SRCARCH
    string
    option env="SRCARCH"

source "arch/$SRCARCH/Kconfig"
```

The last line includes the architecture-dependent configuration file which sources other `kconfig` files, depending on which options are enabled. Having the architecture play such a role has two implications: firstly, that you must specify an architecture when configuring Linux by setting `ARCH=[architecture]`, otherwise it will default to the local machine architecture, and second, that the layout of the top level menu is different for each architecture.

The value you put into `ARCH` is one of the subdirectories you find in directory `arch`, with the oddity that `ARCH=i386` and `ARCH=x86_64` both source `arch/x86/Kconfig`.

The `kconfig` files consist largely of menus, delineated by `menu` and `endmenu`

keywords. Menu items are marked by the keyword `config`. Here is an example, taken from `drivers/char/Kconfig`:

```
menu "Character devices"
[...]
config DEVMEM
    bool "/dev/mem virtual device support"
    default y
    help
        Say Y here if you want to support the /dev/mem device.
        The /dev/mem device is used to access areas of physical
        memory.
        When in doubt, say "Y".
[...]
endmenu
```

The parameter following `config` names a variable that, in this case, is `DEVMEM`. Since this option is a `bool` (Boolean), it can only have two values: if it is enabled, it is assigned to `y`, if it is not enabled, the variable is not defined at all. The name of the menu item that is displayed on the screen is the string following the `bool` keyword.

This configuration item, along with all the others, is stored in a file named `.config` (note that the leading dot (`.`) means that it is a hidden file that will not be shown by the `ls` command, unless you type `ls -a` to show all the files). The line corresponding to this configuration item reads:

```
| CONFIG_DEVMEM=y
```

There are several other data types in addition to `bool`. Here is the list:

- `bool`: Either `y` or not defined.
- `tristate`: Used where a feature can be built as a kernel module or built into the main kernel image. The values are `m` for a module, `y` to be built in, and not defined if the feature is not enabled.
- `int`: An integer value using decimal notation.
- `hex`: An unsigned integer value using hexadecimal notation.
- `string`: A string value.

There may be dependencies between items, expressed by the `depends on` construct, as shown here:

```
| config MTD_CMDLINE_PARTS
|     tristate "Command line partition table parsing"
|     depends on MTD
```

If `CONFIG_MTD` has not been enabled elsewhere, this menu option is not shown and so cannot be selected.

There are also reverse dependencies; the `select` keyword enables other options if this one is enabled. The `Kconfig` file in `arch/$ARCH` has a large number of `select` statements that enable features specific to the architecture, as can be seen here for ARM:

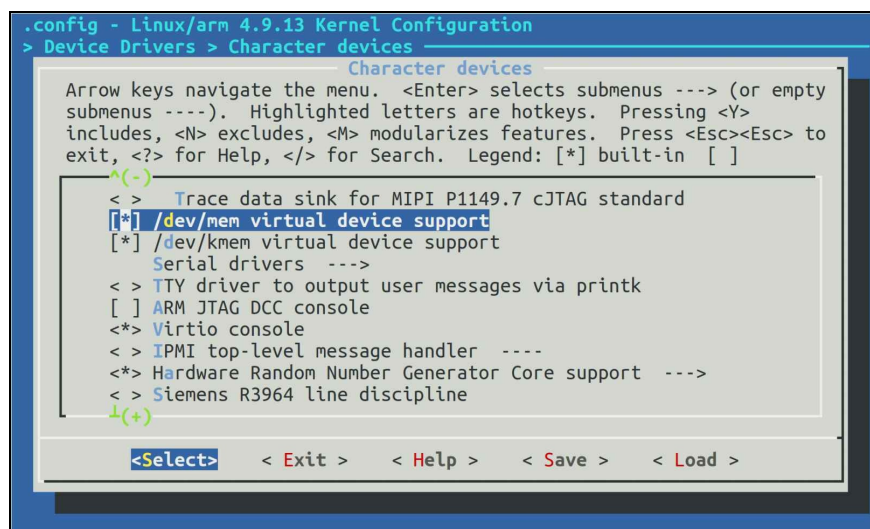
```
config ARM
    bool
    default y
    select ARCH_CLOCKSOURCE_DATA
    select ARCH_HAS_DEVMEM_IS_ALLOWED
    [...]
```

There are several configuration utilities that can read the `Kconfig` files and produce a `.config` file. Some of them display the menus on screen and allow you to make choices interactively. `menuconfig` is probably the one most people are familiar with, but there are also `xconfig` and `gconfig`.

You launch each one via the `make` command, remembering that, in the case of the kernel, you have to supply an architecture, as illustrated here:

```
| $ make ARCH=arm menuconfig
```

Here, you can see `menuconfig` with the `DEVMEM` config option highlighted in the previous paragraph:



The star (*) to the left of an item means that it is selected (Y) or, if it is an M,

that it has been selected to be built as a kernel module.



You often see instructions like enable `CONFIG_BLK_DEV_INITRD`, but with so many menus to browse through, it can take a while to find the place where that configuration is set. All configuration editors have a search function. You can access it in `menuconfig` by pressing the forward slash key, `/`. In `xconfig`, it is in the edit menu, but make sure you miss off `CONFIG_` part of the configuration item you are searching for.

With so many things to configure, it is unreasonable to start with a clean sheet each time you want to build a kernel, so there are a set of known working configuration files in `arch/$ARCH/configs`, each containing suitable configuration values for a single SoC or a group of SoCs.

You can select one with the `make [configuration file name]` command. For example, to configure Linux to run on a wide range of SoCs using the ARMv7-A architecture, you would type:

```
| $ make ARCH=arm multi_v7_defconfig
```

This is a generic kernel that runs on various different boards. For a more specialized application, for example, when using a vendor-supplied kernel, the default configuration file is part of the board support package; you will need to find out which one to use before you can build the kernel.

There is another useful configuration target named `oldconfig`. This takes an existing `.config` file and asks you to supply configuration values for any options that don't have them. You would use it when moving a configuration to a newer kernel version; copy `.config` from the old kernel to the new source directory and run the `make ARCH=arm oldconfig` command to bring it up to date. It can also be used to validate a `.config` file that you have edited manually (ignoring the text `""Automatically generated file; DO NOT EDIT""` that occurs at the top; sometimes it is OK to ignore warnings).

If you do make changes to the configuration, the modified `.config` file becomes part of your board support package and needs to be placed under source code control.

When you start the kernel build, a header file, `include/generated/autoconf.h`, is generated, which contains `#define` for each configuration value so that it can be included in the kernel source.

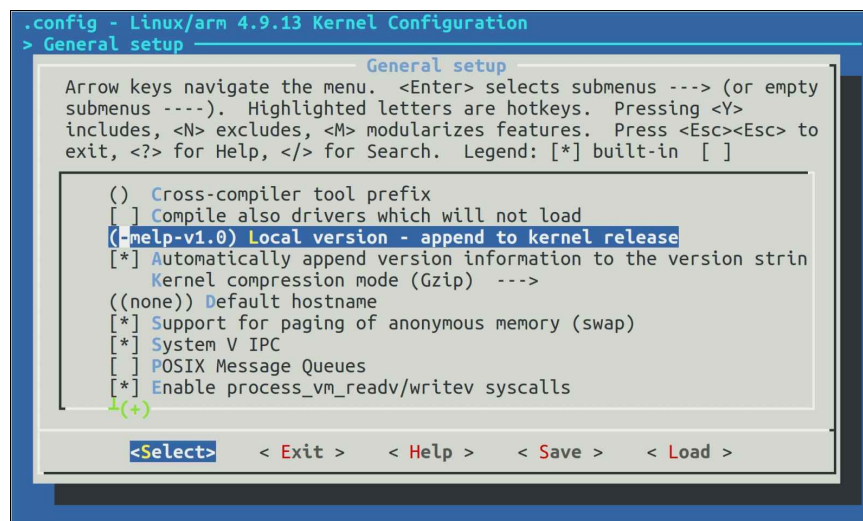
Using LOCALVERSION to identify your kernel

You can discover the kernel version that you have built using the `make kernelversion` target:

```
$ make ARCH=arm kernelversion
4.9.13
```

This is reported at runtime through the `uname` command, and is also used in naming the directory where kernel modules are stored.

If you change the configuration from the default, it is advisable to append your own version information, which you can configure by setting `CONFIG_LOCALVERSION`. As an example, if I wanted to mark the kernel I am building with the identifier `melp` and version 1.0, I would define the local version in `menuconfig` like this:



Running `make kernelversion` produces the same output as before, but now if I run `make kernelrelease`, I see:

```
$ make ARCH=arm kernelrelease
4.9.13-melp-v1.0
```

Kernel modules

I have mentioned kernel modules several times already. Desktop Linux distributions use them extensively so that the correct device and kernel functions can be loaded at runtime, depending on the hardware detected and features required. Without them, every single driver and feature would have to be statically linked in to the kernel, making it infeasibly large.

On the other hand, with embedded devices, the hardware and kernel configuration is usually known at the time the kernel is built, and therefore modules are not so useful. In fact, they cause a problem because they create a version dependency between the kernel and the root filesystem, which can cause boot failures if one is updated but not the other. Consequently, it is quite common for embedded kernels to be built without any modules at all.

Here are a few cases where kernel modules are a good idea in embedded systems:

- When you have proprietary modules, for the licensing reasons given in the preceding section.
- To reduce boot time by deferring the loading of non-essential drivers.
- When there are a number of drivers that could be loaded and it would take up too much memory to compile them statically. For example, you have a USB interface that supports a range of devices. This is essentially the same argument as is used in desktop distributions.

Compiling – Kbuild

The kernel build system, `kbuild`, is a set of `make` scripts that take the configuration information from the `.config` file, work out the dependencies, and compile everything that is necessary to produce a kernel image containing all the statically linked components, possibly a device tree binary and possibly one or more kernel modules. The dependencies are expressed in `makefiles` that are in each directory with buildable components. For instance, the following two lines are taken from `drivers/char/Makefile`:

```
|obj-y += mem.o random.o  
|obj-$(CONFIG_TTY_PRINTK) += ttyprintk.o
```

The `obj-y` rule unconditionally compiles a file to produce the target, so `mem.c` and `random.c` are always part of the kernel. In the second line, `ttyprintk.c` is dependent on a configuration parameter. If `CONFIG_TTY_PRINTK` is `y`, it is compiled as a built-in; if it is `m`, it is built as a module; and if the parameter is undefined, it is not compiled at all.

For most targets, just typing `make` (with the appropriate `ARCH` and `CROSS_COMPILE`) will do the job, but it is instructive to take it one step at a time.

Finding out which kernel target to build

To build a kernel image, you need to know what your bootloader expects. This is a rough guide:

- **U-Boot:** Traditionally, U-Boot has required `uImage`, but newer versions can load a `zImage` file using the `bootz` command
- **x86 targets:** Requires a `bzImage` file
- **Most other bootloaders:** Require a `zImage` file

Here is an example of building a `zImage` file:

```
| $ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- zImage
```



The `-j 4` option tells `make` how many jobs to run in parallel, which reduces the time taken to build. A rough guide is to run as many jobs as you have CPU cores.

There is a small issue with building a `uImage` file for ARM with multi-platform support, which is the norm for the current generation of ARM SoC kernels. Multi-platform support for ARM was introduced in Linux 3.7. It allows a single kernel binary to run on multiple platforms and is a step on the road toward having a small number of kernels for all ARM devices. The kernel selects the correct platform by reading the machine number or the device tree passed to it by the bootloader. The problem occurs because the location of physical memory might be different for each platform, and so the relocation address for the kernel (usually `0x8000` bytes from the start of physical RAM) might also be different. The relocation address is coded into the `uImage` header by the `mkimage` command when the kernel is built, but it will fail if there is more than one relocation address to choose from. To put it another way, the `uImage` format is not compatible with multi-platform images. You can still create a `uImage` binary from a multi-platform build, so long as you give the `LOADADDR` of the particular SoC you are hoping to boot this kernel on. You can find the load address by looking in `mach - [your SoC]/Makefile.boot` and noting the value of `zreladdr-y`:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-  
LOADADDR=0x80008000 uImage
```

Build artifacts

A kernel build generates two files in the top level directory: `vmlinux` and `System.map`. The first, `vmlinux`, is the kernel as an ELF binary. If you have compiled your kernel with debug enabled (`CONFIG_DEBUG_INFO=y`), it will contain debug symbols which can be used with debuggers like `kgdb`. You can also use other ELF binary tools, such as `size`:

```
$ arm-cortex-a8-linux-gnueabihf-size vmlinux
text    data    bss     dec     hex     filename
10605896 5291748 351864 16249508 f7f2a4 vmlinux
```

`System.map` contains the symbol table in a human readable form.

Most bootloaders cannot handle ELF code directly. There is a further stage of processing which takes `vmlinux` and places those binaries in `arch/$ARCH/boot` that are suitable for the various bootloaders:

- `Image`: `vmlinux` converted to raw binary format.
- `zImage`: For the PowerPC architecture, this is just a compressed version of `Image`, which implies that the bootloader must do the decompression. For all other architectures, the compressed `Image` is piggybacked onto a stub of code that decompresses and relocates it.
- `uImage`: `zImage` plus a 64-byte U-Boot header.

While the build is running, you will see a summary of the commands being executed:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex-a8-linux-gnueabihf- \
zImage
CHK include/config/kernel.release
CHK include/generated/uapi/linux/version.h
HOSTCC scripts/basic/fixdep
HOSTCC scripts/kallsyms
HOSTCC scripts/dtc/dtc.o
[...]
```

Sometimes, when the kernel build fails, it is useful to see the actual commands being executed. To do that, add `v=1` to the command line:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex-a8-linux-gnueabihf- \
V=1 zImage
```



```
| [...]
|   arm-cortex_a8-linux-gnueabi-gcc -Wp,-MD,arch/arm/kernel/.irq.o.d -nostdinc -isyste
| [...]
```

Compiling device trees

The next step is to build the device tree, or trees if you have a multi-platform build. The `dtbs` target builds device trees according to the rules in `arch/$ARCH/boot/dts/Makefile`, using the device tree source files in that directory. Following is a snippet from building the `dtbs` target for `multi_v7_defconfig`:

```
$ make ARCH=arm dtbs
[...]  
DTC arch/arm/boot/dts/alpine-db.dtb  
DTC arch/arm/boot/dts/artpec6-devboard.dtb  
DTC arch/arm/boot/dts/at91-kizbox2.dtb  
DTC arch/arm/boot/dts/at91-sama5d2_xplained.dtb  
DTC arch/arm/boot/dts/at91-sama5d3_xplained.dtb  
DTC arch/arm/boot/dts/sama5d31ek.dtb  
[...]
```

The compiled `.dtb` files are generated in the same directory as the sources.

Compiling modules

If you have configured some features to be built as modules, you can build them separately using the `modules` target:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- \
modules
```

The compiled `modules` have a `.ko` suffix and are generated in the same directory as the source code, meaning that they are scattered all around the kernel source tree. Finding them is a little tricky, but you can use the `modules_install` make target to install them in the right place. The default location is `/lib/modules` in your development system, which is almost certainly not what you want. To install them into the staging area of your root filesystem (we will talk about root filesystems in the next chapter), provide the path using `INSTALL_MOD_PATH`:

```
$ make -j4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- \
INSTALL_MOD_PATH=$HOME/rootfs modules_install
```

Kernel modules are put into the directory `/lib/modules/[kernel version]`, relative to the root of the filesystem.

Cleaning kernel sources

There are three `make` targets for cleaning the kernel source tree:

- `clean`: Removes object files and most intermediates.
- `mrproper`: Removes all intermediate files, including the `.config` file. Use this target to return the source tree to the state it was in immediately after cloning or extracting the source code. If you are curious about the name, Mr Proper is a cleaning product common in some parts of the world. The meaning of `make mrproper` is to give the kernel sources a really good scrub.
- `distclean`: This is the same as `mrproper`, but also deletes editor backup files, patch files, and other artifacts of software development.

Building a kernel for the BeagleBone Black

In light of the information already given, here is the complete sequence of commands to build a kernel, the modules, and a device tree for the BeagleBone Black, using the Crosstool-NG ARM Cortex A8 cross compiler:

```
$ cd linux-stable
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- mrproper
$ make ARCH=arm multi_v7_defconfig
$ make -j4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- zImage
$ make -j4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- modules
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- dtbs
```

These commands are in the script `MELP/chapter_04/build-linux-bbb.sh` .

Building a kernel for QEMU

Here is the sequence of commands to build Linux for the ARM Versatile PB that is emulated by QEMU, using the Crosstool-NG V5te compiler:

```
$ cd linux-stable
$ make ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi- mrproper
$ make -j4 ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi- zImage
$ make -j4 ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi- modules
$ make ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi- dtbs
```

These commands are in the script `MELP/chapter_04/build-linux-versatilepb.sh`.

Booting the kernel

Booting Linux is highly device-dependent. In this section, I will show you how it works for the BeagleBone Black and QEMU. For other target boards, you must consult the information from the vendor or from the community project, if there is one.

At this point, you should have the `zImage` file and the `dtbs` targets for the BeagleBone Black or QEMU.

Booting the BeagleBone Black

To begin, you need a microSD card with U-Boot installed, as described in the section *Installing U-Boot*. Plug the microSD card into your card reader and from the `linux-stable` directory the files `arch/arm/boot/zImage` and `arch/arm/boot/dts/am335x-boneblack.dtb` to the `boot` partition. Unmount the card and plug it into the BeagleBone Black. Start a terminal emulator, such as `gtkterm`, and be prepared to press the space bar as soon as you see the U-Boot messages appear. Next, power on the BeagleBone Black and press the space bar. You should get a U-Boot prompt, `.` Now enter the following commands to load Linux and the device tree binary:

```
U-Boot# fatload mmc 0:1 0x80200000 zImage
reading zImage
7062472 bytes read in 447 ms (15.1 MiB/s)
U-Boot# fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
reading am335x-boneblack.dtb
34184 bytes read in 10 ms (3.3 MiB/s)
U-Boot# setenv bootargs console=tty00
U-Boot# bootz 0x80200000 - 0x80f00000
## Flattened Device Tree blob at 80f00000
Booting using the fdt blob at 0x80f00000
Loading Device Tree to 8fff4000, end 8ffff587 ... OK

Starting kernel ...

[ 0.000000] Booting Linux on physical CPU 0x0
[...]
```

Note that we set the kernel command line to `console=tty00`. That tells Linux which device to use for console output, which in this case is the first UART on the board, device `tty00`. Without this, we would not see any messages after starting the kernel..., and therefore would not know if it was working or not. The sequence will end in a kernel panic, for reasons I will explain later on.

Booting QEMU

Assuming that you have already installed `qemu-system-arm`, you can launch with the kernel and the `.dtb` file for the ARM Versatile PB, as follows:

```
$ QEMU_AUDIO_DRV=none \  
qemu-system-arm -m 256M -nographic -M versatilepb -kernel zImage \  
-append "console=ttyAMA0,115200" -dtb versatile-pb.dtb
```

Note that setting `QEMU_AUDIO_DRV` to `none` is just to suppress error messages from QEMU about missing configurations for the audio drivers, which we do not use. As with the BeagleBone Black, this will end with a kernel panic and the system will halt. To exit from QEMU, type *Ctrl + A* and then *x* (two separate keystrokes).

Kernel panic

While things started off well, they ended badly:

```
| [ 1.886379] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0  
| [ 1.895105] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknow
```

This is a good example of a kernel panic. A panic occurs when the kernel encounters an unrecoverable error. By default, it will print out a message to the console and then halt. You can set the `panic` command-line parameter to allow a few seconds before reboots following a panic. In this case, the unrecoverable error is no root filesystem, illustrating that a kernel is useless without a user space to control it. You can supply a user space by providing a root filesystem, either as a ramdisk or on a mountable mass storage device. We will talk about how to create a root filesystem in the next chapter, but first I want to describe the sequence of events that leads up to `panic`.

Early user space

In order to transition from kernel initialization to user space, the kernel has to mount a root filesystem and execute a program in that root filesystem. This can be achieved via a ramdisk or by mounting a real filesystem on a block device. The code for all of this is in `init/main.c`, starting with the function `rest_init()`, which creates the first thread with PID 1 and runs the code in `kernel_init()`. If there is a ramdisk, it will try to execute the program `/init`, which will take on the task of setting up the user space.

If fails to find and run `/init`, it tries to mount a filesystem by calling the function `prepare_namespace()` in `init/do_mounts.c`. This requires a `root=` command line to give the name of the block device to use for mounting, usually in the form:

```
| root=/dev/<disk name><partition number>
```

Or, for SD cards and eMMC:

```
| root=/dev/<disk name>p<partition number>
```

For example, for the first partition on an SD card, that would be `root=/dev/mmcblk0p1`. If the mount succeeds, it will try to execute `/sbin/init`, followed by `/etc/init`, `/bin/init`, and then `/bin/sh`, stopping at the first one that works.

The program can be overridden on the command line. For a ramdisk, use `rdinit=`, and for a filesystem, use `init=`.

Kernel messages

Kernel developers are fond of printing out useful information through liberal use of `printk()` and similar functions. The messages are categorized according to importance, with 0 being the highest:

Level	Value	Meaning
KERN_EMERG	0	The system is unusable
KERN_ALERT	1	Action must be taken immediately
KERN_CRIT	2	Critical conditions
KERN_ERR	3	Error conditions
KERN_WARNING	4	Warning conditions
KERN_NOTICE	5	Normal but significant conditions
KERN_INFO	6	Informational
KERN_DEBUG	7	Debug-level messages

They are first written to a buffer, `__log_buf`, the size of which is two to the power of `CONFIG_LOG_BUF_SHIFT`. For example, if `CONFIG_LOG_BUF_SHIFT` is 16, then `__log_buf` is 64 KiB. You can dump the entire buffer using the command `dmesg`.

If the level of a message is less than the console log level, it is displayed on the console as well as placed in `__log_buf`. The default console log level is 7, meaning that messages of level 6 and lower are displayed, filtering out `KERN_DEBUG`, which is level 7. You can change the console log level in several ways, including by using the kernel parameter `loglevel=<level>`, or the command `dmesg -n <level>`.

Kernel command line

The kernel command line is a string that is passed to the kernel by the bootloader, via the `bootargs` variable in the case of U-Boot; it can also be defined in the device tree, or set as part of the kernel configuration in `CONFIG_CMDLINE`.

We have seen some examples of the kernel command line already, but there are many more. There is a complete list in `Documentation/kernel-parameters.txt`. Here is a smaller list of the most useful ones:

Name	Description
<code>debug</code>	Sets the console log level to the highest level, 8, to ensure that you see all the kernel messages on the console.
<code>init=</code>	The init program to run from a mounted root filesystem, which defaults to <code>/sbin/init</code> .
<code>lpj=</code>	Sets <code>loops_per_jiffy</code> to a given constant. There is a description of the significance of this in the paragraph following this table.
<code>panic=</code>	Behavior when the kernel panics: if it is greater than zero, it gives the number of seconds before rebooting; if it is zero, it waits forever (this is the default); or if it is less than zero, it reboots without any delay.
<code>quiet</code>	Sets the console log level to , suppressing all but emergency messages. Since most devices have a serial console, it takes time to output all those strings. Consequently, reducing the number of messages using this option reduces boot time.
<code>rdinit=</code>	The init program to run from a ramdisk. It defaults to <code>/init</code> .
<code>ro</code>	Mounts the root device as read-only. Has no effect on a ramdisk, which is always read/write.
<code>root=</code>	Device to mount the root filesystem.
<code>rootdelay=</code>	The number of seconds to wait before trying to mount the root device; defaults to zero. Useful if the device takes time to probe the hardware, but also see <code>rootwait</code> .

<code>rootfstype=</code>	The filesystem type for the root device. In many cases, it is auto-detected during mount, but it is required for <code>jffs2</code> filesystems.
<code>rootwait</code>	Waits indefinitely for the root device to be detected. Usually necessary with mmc devices.
<code>rw</code>	Mounts the root device as read-write (default).

The `lpj` parameter is often mentioned in connection with reducing the kernel boot time. During initialization, the kernel loops for approximately 250 ms to calibrate a delay loop. The value is stored in the variable `loops_per_jiffy`, and reported like this:

```
| Calibrating delay loop... 996.14 BogoMIPS (lpj=4980736)
```

If the kernel always runs on the same hardware, it will always calculate the same value. You can shave 250 ms off the boot time by adding `lpj=4980736` to the command line.

Porting Linux to a new board

Porting Linux to a new board can be easy or difficult, depending on how similar your board is to an existing development board. In [Chapter 3, All About Bootloaders](#), we ported U-Boot to a new board, named `nova`, which is based on the BeagleBone Black. very few changes to be made to the kernel code and so it very easy. If you are porting to completely new and innovative hardware, there will be more to do. I am only going to consider the simple case.

The organization of architecture-specific code in `arch/$ARCH` differs from one system to another. The x86 architecture is pretty clean because most hardware details are detected at runtime. The PowerPC architecture puts SoC and board-specific files into sub directory platforms. The ARM architecture, on the other hand, is quite messy, in part because there is a lot of variability between the many ARM-based SoCs. Platform-dependent code is put in directories named `mach-*`, approximately one per SoC. There are other directories named `plat-*` which contain code common to several versions of an SoC. In the case of the BeagleBone Black, the relevant directory is `arch/arm/mach-omap2`. Don't be fooled by the name though; it contains support for OMAP2, 3, and 4 chips, as well as the AM33xx family of chips that the BeagleBone uses.

In the following sections, I am going to explain how to create a device tree for a new board and how to key that into the initialization code of Linux.

A new device tree

The first thing to do is create a device tree for the board and modify it to describe the additional or changed hardware of the `Nova` board. In this simple case, we will just copy `am335x-boneblack.dts` to `nova.dts` and change the board name in `nova.dts`, as shown highlighted here:

```
/dts-v1/;
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
#include <dt-bindings/display/tda998x.h>

/ {
    model = "Nova";
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
};
[...]
```

We can build the Nova device tree binary explicitly like this:

```
$ make ARCH=arm nova.dtb
```

If we want the device tree for Nova to be compiled by `make ARCH=arm dtbs` whenever an AM33xx target is selected, we could add a dependency in `arch/arm/boot/dts/Makefile` as follows:

```
[...]
dtb-$(CONFIG_SOC_AM33XX) +=
    nova.dtb
[...]
```

We can see the effect of using the Nova device tree by booting the BeagleBone Black, following the same procedure as in the section *Booting the BeagleBone Black*, with the same `zImage` file as before, but loading `nova.dtb` in place of `am335x-boneblack.dtb`. The following highlighted output is the point at which the machine model is printed out:

```
Starting kernel ...

[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Linux version 4.9.13-melp-v1.0-dirty (chris@chris-xps) (gcc version 5.2.0 (
[ 0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=10c5387d
```



```
| [ 0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache  
| [ 0.000000] OF: fdt:Machine model: Nova  
| [...]
```

Now that we have a device tree specifically for the Nova board, we could modify it to describe the hardware differences between Nova and the BeagleBone Black. There are quite likely to be changes to the kernel configuration as well, in which case you would create a custom configuration file based on a copy of `arch/arm/configs/multi_v7_defconfig`.

Setting the board compatible property

Creating a new device tree means that we can describe the hardware on the Nova board, selecting device drivers and setting properties to match. But, suppose the Nova board needs different early initialization code than the BeagleBone Black; how can we link that in?

The board setup is controlled by the `compatible` property in the root node. This is what we have for the Nova board at the moment:

```
/ {
    model = "Nova";
    compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
};
```

When the kernel parses this node, it will search for a matching machine for each of the values of the compatible property, starting on the left and stopping with the first match found. Each machine is defined in a structure delimited by `DT_MACHINE_START` and `MACHINE_END` macros. In `arch/arm/mach-omap2/board-generic.c`, we find:

```
#ifdef CONFIG_SOC_AM33XX
static const char *const am33xx_boards_compat[] __initconst = {
    "ti,am33xx",
    NULL,
};

DT_MACHINE_START(AM33XX_DT, "Generic AM33XX (Flattened Device Tree)")
    .reserve = omap_reserve,
    .map_io = am33xx_map_io,
    .init_early = am33xx_init_early,
    .init_machine = omap_generic_init,
    .init_late = am33xx_init_late,
    .init_time = omap3_gptimer_timer_init,
    .dt_compat = am33xx_boards_compat,
    .restart = am33xx_restart,
MACHINE_END
#endif
```

Note that the string array, `am33xx_boards_compat`, contains `"ti,am33xx"` which matches one of the machines listed in the compatible property. In fact, it is the only match possible, since there are none for `ti,am335x-bone-black` or `ti,am335x-bone`. The

structure between `DT_MACHINE_START` and `MACHINE_END` contains a pointer to the string array, and function pointers for the board setup functions. You may wonder why bother with `ti,am335x-bone-black` and `ti,am335x-bone` if they never match anything? The answer is partly that they are place holders for the future, but also that there are places in the kernel that contain runtime tests for the machine using the function `of_machine_is_compatible()`. For example, in `drivers/net/ethernet/ti/cpsw-common.c`:

```
int ti_cm_get_macid(struct device *dev, int slave, u8 *mac_addr)
{
[...]
```

```
    if (of_machine_is_compatible("ti,am33xx"))
        return cpsw_am33xx_cm_get_macid(dev, 0x630, slave, mac_addr);
[...]
```

Thus, we have to look through not just the `mach-*` directories but the entire kernel source code to get a list of all the places that depend on the machine compatible property. In the 4.9 kernel, you will find that there are still no checks for `ti,am335x-bone-black` and `ti,am335x-bone`, but there may be in the future.

Returning to the Nova board, if we want to add machine specific setup, we can add a machine in `arch/arm/mach-omap2/board-generic.c`, like this:

```
#ifdef CONFIG_SOC_AM33XX
[...]
```

```
static const char *const nova_compat[] __initconst = {
    "ti,nova",
    NULL,
};

DT_MACHINE_START(NOVA_DT, "Nova board (Flattened Device Tree)")
    .reserve = omap_reserve,
    .map_io = am33xx_map_io,
    .init_early = am33xx_init_early,
    .init_machine = omap_generic_init,
    .init_late = am33xx_init_late,
    .init_time = omap3_gptimer_timer_init,
    .dt_compat = nova_compat,
    .restart = am33xx_restart,
MACHINE_END
#endif
```

Then we could change the device tree root node like this:

```
/ {
    model = "Nova";
    compatible = "ti,nova", "ti,am33xx";
};
```

Now, the machine will match `ti,nova` in `board-generic.c`. We keep `ti,am33xx` because

we want the runtime tests, such as the one in `drivers/net/ethernet/ti/cpsw-common.c`, to continue to work.

Additional reading

The following resources have further information about the topics introduced in this chapter:

- *Linux Kernel Development*, 3rd Edition by Robert Love
- Linux weekly news, <https://lwn.net/>

Summary

Linux is a very powerful and complex operating system kernel that can be married to various types of user space, ranging from a simple embedded device, through increasingly complex mobile devices using Android, to a full server operating system. One of its strengths is the degree of configurability. The definitive place to get the source code is <https://www.kernel.org/>, but you will probably need to get the source for a particular SoC or board from the vendor of that device or a third-party that supports that device. The customization of the kernel for a particular target may consist of changes to the core kernel code, additional drivers for devices that are not in mainline Linux, a default kernel configuration file, and a device tree source file.

Normally, you start with the default configuration for your target board, and then tweak it by running one of the configuration tools such as `menuconfig`. One of the things you should consider at this point is whether the kernel features and drivers should be compiled as modules or built-in. Kernel modules are usually no great advantage for embedded systems, where the feature set and hardware are usually well defined. However, modules are often used as a way to import proprietary code into the kernel, and also to reduce boot time by loading non-essential drivers after boot.

Building the kernel produces a compressed kernel image file, named `zImage`, `bzImage`, or `uImage`, depending on the bootloader you will be using and the target architecture. A kernel build will also generate any kernel modules (as `.ko` files) that you have configured, and device tree binaries (as `.dtb` files) if your target requires them.

Porting Linux to a new target board can be quite simple or very difficult, depending on how different the hardware is from that in the mainline or vendor supplied kernel. If your hardware is based on a well-known reference design, then it may be just a question of making changes to the device tree or to the platform data. You may well need to add device drivers, which I discuss in [Chapter 9, *Interfacing with Device Drivers*](#). However, if the hardware is radically different to a reference design, you may need additional core support, which is

outside the scope of this book.

The kernel is the core of a Linux-based system, but it cannot work by itself. It requires a root filesystem that contains the user space components. The root filesystem can be a ramdisk or a filesystem accessed via a block device, which will be the subject of the next chapter. As we have seen, booting a kernel without a root filesystem results in a kernel panic.

Building a Root Filesystem

The root filesystem is the fourth and the final element of embedded Linux. Once you have read this chapter, you will be able build, boot, and run a simple embedded Linux system.

The techniques I will describe here are broadly known as **roll your own** or **RYO**. Back in the earlier days of embedded Linux, this was the only way to create a root filesystem. There are still some use cases where an RYO root filesystem is applicable, for example, when the amount of RAM or storage is very limited, for quick demonstrations, or for any case in which your requirements are not (easily) covered by the standard build system tools. Nevertheless, these cases are quite rare. Let me emphasize that the purpose of this chapter is educational; it is not meant to be a recipe for building everyday embedded systems: use the tools described in the next chapter for this.

The first objective is to create a minimal root filesystem that will give us a shell prompt. Then, using this as a base, we will add scripts to start up other programs and configure a network interface and user permissions. There are worked examples for both the BeagleBone Black and QEMU targets. Knowing how to build the root filesystem from scratch is a useful skill, and it will help you to understand what is going on when we look at more complex examples in later chapters.

In this chapter, we will cover the following topics:

- What should be in the root filesystem?
- Transferring the root filesystem to the target.
- Creating a boot `initramfs`.
- The `init` program.
- Configuring user accounts.
- A better way of managing device nodes.
- Configuring the network.
- Creating filesystem images with device tables.
- Mounting the root filesystem using NFS.

What should be in the root filesystem?

The kernel will get a root filesystem, either an `initramfs`, passed as a pointer from the bootloader, or by mounting the block device given on the kernel command line by the `root=` parameter. Once it has a root filesystem, the kernel will execute the first program, by default named `init`, as described in the section *Early user space* in [Chapter 4, Configuring and Building the Kernel](#). Then, as far as the kernel is concerned, its job is complete. It is up to the `init` program to begin starting other programs and so bring the system to life.

To make a minimal root filesystem, you need these components:

- **init:** This is the program that starts everything off, usually by running a series of scripts. I will describe how `init` works in much more detail in [Chapter 10, Starting Up – The init Program](#)
- **Shell:** You need a shell to give you a command prompt but, more importantly, also to run the shell scripts called by `init` and other programs.
- **Daemons:** A daemon is a background program that provides a service to others. Good examples are the **system log daemon (syslogd)** and the **secure shell daemon (sshd)**. The `init` program must start the initial population of daemons to support the main system applications. In fact, `init` is itself a daemon: it is the daemon that provides the service of launching other daemons.
- **Shared libraries:** Most programs are linked with shared libraries, and so they must be present in the root filesystem.
- **Configuration files:** The configuration for `init` and other daemons is stored in a series of text files, usually in the `/etc` directory.
- **Device nodes:** These are the special files that give access to various device drivers.
- **/proc and /sys:** These two pseudo filesystems represent kernel data structures as a hierarchy of directories and files. Many programs and library functions depend on `proc` and `sys`.
- **Kernel modules:** If you have configured some parts of your kernel to be

modules, they need to be installed in the root filesystem, usually in
`/lib/modules/[kernel version]`.

In addition, there are the device-specific applications that make the device do the job it is intended for, and also the run-time data files that they generate.

In some cases, you could condense most of these components into a single, statically-linked program, and start the program instead of init. For example, if your program was named `/myprog`, you would add the following command to the kernel command line:



`init=/myprog`. I have come across such a configuration only once, in a secure system in which the fork system call had been disabled, thus making it impossible for any other program to be started. The downside of this approach is that you can't make use of the many tools that normally go into an embedded system; you have to do everything yourself.

The directory layout

Interestingly, the Linux kernel does not care about the layout of files and directories beyond the existence of the program named by `init=` or `rdinit=`, so you are free to put things wherever you like. As an example, compare the file layout of a device running Android to that of a desktop Linux distribution: they are almost completely different.

However, many programs expect certain files to be in certain places, and it helps us developers if devices use a similar layout, Android aside. The basic layout of a Linux system is defined in the **Filesystem Hierarchy Standard (FHS)**, which is available at

<http://refspecs.linuxfoundation.org/fhs.shtml>. The FHS covers all the implementations of Linux operating systems from the largest to the smallest. Embedded devices tend to use a subset based on their needs, but it usually includes the following:

- `/bin`: Programs essential for all users
- `/dev`: Device nodes and other special files
- `/etc`: System configuration files
- `/lib`: Essential shared libraries, for example, those that make up the C-library
- `/proc`: The `proc` filesystem
- `/sbin`: Programs essential to the system administrator
- `/sys`: The `sysfs` filesystem
- `/tmp`: A place to put temporary or volatile files
- `/usr`: Additional programs, libraries, and system administrator utilities, in the directories `/usr/bin`, `/usr/lib` and `/usr/sbin`, respectively
- `/var`: A hierarchy of files and directories that may be modified at runtime, for example, log messages, some of which must be retained after boot

There are some subtle distinctions here. The difference between `/bin` and `/sbin` is simply that the latter need not be included in the search path for non-root users. Users of Red Hat-derived distributions will be familiar with this. The significance of `/usr` is that it maybe in a separate partition from the root filesystem, so it cannot contain anything that is needed to boot the system up.

The staging directory

You should begin by creating a **staging** directory on your host computer where you can assemble the files that will eventually be transferred to the target. In the following examples, I have used `~/rootfs`. You need to create a *skeleton* directory structure in it, for example, take a look here:

```
$ mkdir ~/rootfs
$ cd ~/rootfs
$ mkdir bin dev etc home lib proc sbin sys tmp usr var
$ mkdir usr/bin usr/lib usr/sbin
$ mkdir -p var/log
```

To see the directory hierarchy more clearly, you can use the handy `tree` command used in the following example with the `-d` option to show only the directories:

```
$ tree -d
.
├── bin
├── dev
├── etc
├── home
├── lib
├── proc
├── sbin
├── sys
├── tmp
├── usr
│   ├── bin
│   ├── lib
│   └── sbin
├── va
├── var
│   └── log
```

POSIX file access permissions

Every process, which in the context of this discussion means every running program, belongs to a user and one or more groups. The user is represented by a 32-bit number called the **user ID** or **UID**. Information about users, including the mapping from a UID to a name, is kept in `/etc/passwd`. Likewise, groups are represented by a **group ID** or **GID** with information kept in `/etc/group`. There is always a `root` user with a UID of `0` and a root group with a GID of `0`. The `root` user is also called the **superuser** because; in a default configuration, it bypasses most permission checks and can access all the resources in the system. Security in Linux-based systems is mainly about restricting access to the `root` account.

Each file and directory also has an owner and belongs to exactly one group. The level of access a process has to a file or directory is controlled by a set of access permission flags, called the **mode** of the file. There are three collections of three bits: the first collection applies to the *owner* of the file, the second to the *members* of the same group as the file, and the last to *everyone else*: the rest of the world. The bits are for **read (r)**, **write (w)**, and **execute (x)** permissions on the file. Since three bits fit neatly into an octal digit, they are usually represented in octal, as shown in the following diagram:

400	r-----	}	Owner permissions
200	-w-----		
100	--x-----		
040	---r-----	}	Group permissions
020	----w----		
010	-----x---		
004	-----r--	}	World permissions
002	-----w-		
001	-----x		

There is a further group of three bits that have special meanings:

- **SUID (4)**: If the file is executable, it changes the effective UID of the process to that of the owner of the file when the program is run.
- **SGID (2)**: Similar to SUID, this changes the effective GID of the process to that of the group of the file.
- **Sticky (1)**: In a directory, this restricts deletion so that one user cannot delete files that are owned by another user. This is usually set on `/tmp` and

/var/tmp.

The SUID bit is probably used most often . It gives non-root users a temporary privilege escalation to superuser to perform a task. A good example is the `ping` program: `ping` opens a raw socket, which is a privileged operation. In order for normal users to use `ping`, it is owned by user `root` and has the SUID bit set so that when you run `ping`, it executes with UID `0` regardless of your UID.

To set these bits, use the octal numbers, 4, 2, and 1 with the `chmod` command. For example, to set SUID on `/bin/ping` in your staging `root` directory, you could use the following:

```
$ cd ~/rootfs
$ ls -l bin/ping
-rwxr-xr-x 1 root root 35712 Feb 6 09:15 bin/ping

$ sudo chmod 4755 bin/ping
$ ls -l bin/ping
-rwsr-xr-x 1 root root 35712 Feb 6 09:15 bin/ping
```

Note that the second `ls` command shows the first three bits of the mode to be `rws`, whereas previously, they had been `rwx`. That 's' indicates that the SUID bit is set.

File ownership permissions in the staging directory

For security and stability reasons, it is vitally important to pay attention to the ownership and permissions of the files that will be placed on the target device. Generally speaking, you want to restrict sensitive resources to be accessible only by the root and wherever possible, to run programs using non-root users so that if they are compromised by an outside attack, they offer as few system resources to the attacker as possible. For example, the device node called `/dev/mem` gives access to system memory, which is necessary in some programs. But, if it is readable and writable by everyone, then there is no security because everyone can access everything in memory. So, `/dev/mem` should be owned by root, belong to the root group, and have a mode of 600, which denies read and write access to all but the owner.

There is a problem with the staging directory though. The files you create there will be owned by you, but when they are installed on the device, they should belong to specific owners and groups, mostly the `root` user. An obvious fix is to change the ownership to `root` at this stage with the commands shown here:

```
$ cd ~/rootfs
$ sudo chown -R root:root *
```

The problem is that you need `root` privileges to run the `chown` command, and from that point onward, you will need to be `root` to modify any files in the staging directory. Before you know it, you are doing all your development logged on as `root`, which is not a good idea. This is a problem that we will come back to later.

Programs for the root filesystem

Now, it is time to start populating the `root` filesystem with the essential programs and the supporting libraries, configuration, and data files that they need to operate. I will begin with an overview of the types of programs you will need.

The init program

Init is the first program to be run, and so it is an essential part of the root filesystem. In this chapter, we will be using the simple init program provided by BusyBox.

Shell

We need a shell to run scripts and to give us a command prompt so that we can interact with the system. An interactive shell is probably not necessary in a production device, but it is useful for development, debugging, and maintenance. There are various shells in common use in embedded systems:

- **bash:** This is the big beast that we all know and love from desktop Linux. It is a superset of the Unix Bourne shell with many extensions or *bashisms*.
- **ash:** Also based on the Bourne shell, it has a long history with the BSD variants of Unix. BusyBox has a version of `ash`, which has been extended to make it more compatible with `bash`. It is much smaller than `bash`, and hence it is a very popular choice for embedded systems.
- **hush:** This is a very small shell that we briefly looked at in [Chapter 3, All about Bootloaders](#). It is useful on devices with very little memory. There is a version of `hush` in BusyBox.



If you are using `ash` or `hush` as the shell on the target, make sure that you test your shell scripts on the target. It is very tempting to test them only on the host, using `bash`, and then be surprised that they don't work when you copy them to the target.

Utilities

The shell is just a way of launching other programs, and a shell script is little more than a list of programs to run, with some flow control and a means of passing information between programs. To make a shell useful, you need the utility programs that the Unix command line is based on. Even for a basic root filesystem, you need approximately 50 utilities, which presents two problems. Firstly, tracking down the source code for each one and cross-compiling it would be quite a big job. Secondly, the resulting collection of programs would take up several tens of megabytes, which was a real problem in the early days of embedded Linux when a few megabytes was all you had. To solve this problem, BusyBox was born.

BusyBox to the rescue!

The genesis of **BusyBox** had nothing to do with embedded Linux. The project was instigated in 1996 by Bruce Perens for the Debian installer so that he could boot Linux from a 1.44 MB floppy disk. Coincidentally, this was about the size of the storage on contemporary devices, and so the embedded Linux community quickly took it up. BusyBox has been at the heart of embedded Linux ever since.

BusyBox was written from scratch to perform the essential functions of those essential Linux utilities. The developers took advantage of the 80:20 rule: the most useful 80% of a program is implemented in 20% of the code. Hence, BusyBox tools implement a subset of the functions of the desktop equivalents, but they do enough of it to be useful in the majority of cases.

Another trick BusyBox employs is to combine all the tools together into a single binary, making it easy to share code between them. It works like this: BusyBox is a collection of applets, each of which exports its main function in the form `[applet]_main`. For example, the `cat` command is implemented in `coreutils/cat.c` and exports `cat_main`. The `main` function of BusyBox itself dispatches the call to the correct applet, based on the command-line arguments.

So, to read a file, you can launch BusyBox with the name of the applet you want to run, followed by any arguments the applet expects, as shown here:

```
| $ busybox cat my_file.txt
```

You can also run BusyBox with no arguments to get a list of all the applets that have been compiled.

Using BusyBox in this way is rather clumsy. A better way to get BusyBox to run the `cat` applet is to create a symbolic link from `/bin/cat` to `/bin/busybox`:

```
| $ ls -l bin/cat bin/busybox
-rwxr-xr-x 1 root root 892868 Feb 2 11:01 bin/busybox
lrwxrwxrwx 1 root root 7      Feb 2 11:01 bin/cat -> busybox
```

When you type `cat` at the command line, BusyBox is the program that actually

runs. BusyBox only has to check the command tail passed in `argv[0]`, which will be `/bin/cat`, extract the application name, `cat`, and do a table look-up to match `cat` with `cat_main`. All this is in `libbb/appletlib.c` in this section of code (slightly simplified):

```
applet_name = argv[0];
applet_name = bb_basename(applet_name);
run_applet_and_exit(applet_name, argv);
```

BusyBox has over three hundred applets including an `init` program, several shells of varying levels of complexity, and utilities for most admin tasks. There is even a simple version of the `vi` editor, so you can change text files on your device.

To summarize, a typical installation of BusyBox consists of a single program with a symbolic link for each applet, but which behaves exactly as if it were a collection of individual applications.

Building BusyBox

BusyBox uses the same `Kconfig` and `Kbuild` system as the kernel, so cross compiling is straightforward. You can get the source by cloning the Git archive and checking out the version you want (`1_26_2` was the latest at the time of writing), such as follows:

```
$ git clone git://busybox.net/busybox.git
$ cd busybox
$ git checkout 1_26_2
```

You can also download the corresponding TAR file from <http://busybox.net/downloads>.

Then, configure BusyBox by starting with the default configuration, which enables pretty much all of the features of BusyBox:

```
$ make distclean
$ make defconfig
```

At this point, you probably want to run `make menuconfig` to fine tune the configuration. For example, you almost certainly want to set the install path in Busybox Settings | Installation Options (`CONFIG_PREFIX`) to point to the staging directory. Then, you can cross compile in the usual way. If your intended target is the BeagleBone Black, use this command:

```
| $ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

If your intended target is the QEMU emulation of a Versatile PB, use this command:

```
| $ make ARCH=arm CROSS_COMPILE=arm-unknown-linux-gnueabi-
```

In either case, the result is the executable, `busybox`. For a default configuration build like this, the size is about 900 KiB. If this is too big for you, you can slim it down by changing the configuration to leave out the utilities you don't need.

To install BusyBox into the staging area, use the following command:

```
| $ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- install
```

This will copy the binary to the directory configured in `CONFIG_PREFIX` and create all the symbolic links to it.

ToyBox – an alternative to BusyBox

BusyBox is not the only game in town. In addition, there is **ToyBox**, which you can find at <http://landley.net/toybox/>. The project was started by Rob Landley, who was previously a maintainer of BusyBox. ToyBox has the same aim as BusyBox, but with more emphasis on complying with standards, especially POSIX-2008 and LSB 4.1, and less on compatibility with GNU extensions to those standards. ToyBox is smaller than BusyBox, partly because it implements fewer applets. However, the main difference is the license, which is BSD rather than GPL v2. This makes it license compatible with operating systems with a BSD-licensed user space, such as Android, and hence it is part of all the new Android devices.

Libraries for the root filesystem

Programs are linked with libraries. You could link them all statically, in which case, there would be no libraries on the target device. But, this takes up an unnecessarily large amount of storage if you have more than two or three programs. So, you need to copy shared libraries from the toolchain to the staging directory. How do you know which libraries?

One option is to copy all of the `.so` files from the `sysroot` directory of your toolchain, since they must be of some use otherwise they wouldn't exist! This is certainly logical and, if you are creating a platform to be used by others for a range of applications, it would be the correct approach. Be aware, though, that a full `glibc` is quite large. In the case of a `crosstool-NG` build of `glibc` 2.22, the libraries, locales, and other supporting files come to 33 MiB. Of course, you could cut down on that considerably using `musl libc` or `uClibc-ng`.

Another option is to cherry pick only those libraries that you require, for which you need a means of discovering library dependencies. Using some of our knowledge from [Chapter 2, Learning About Toolchains](#), we can use the `readelf` command for this task:

```
$ cd ~/rootfs
$ arm-cortex_a8-linux-gnueabihf-readelf -a bin/busybox | grep "program interpreter"
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]

$ arm-cortex_a8-linux-gnueabihf-readelf -a bin/busybox | grep "Shared library"
0x00000001 (NEEDED) Shared library: [libm.so.6]
0x00000001 (NEEDED) Shared library: [libc.so.6]
```

Now, you need to find these files in the toolchain `sysroot` directory and copy them to the staging directory. Remember that you can find `sysroot` like this:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot
/home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-linux-gnueabihf/sysroot
```

To reduce the amount of typing, I am going to keep a copy of that in a shell variable:

```
| $ export SYSROOT=$(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)
```

If you look at `/lib/ld-linux-armhf.so.3` in `sysroot`, you will see that, it is, in fact, a symbolic link:

```
$ cd $SYSROOT
$ ls -l lib/ld-linux-armhf.so.3
lrwxrwxrwx 1 chris chris 10 Mar 3 15:22 lib/ld-linux-armhf.so.3 -> ld-2.22.so
```

Repeat the exercise for `libc.so.6` and `libm.so.6`, and you will end up with a list of three files and three symbolic links. Now, you can copy each one using `cp -a`, which will preserve the symbolic link:

```
$ cd ~/rootfs
$ cp -a $SYSROOT/lib/ld-linux-armhf.so.3 lib
$ cp -a $SYSROOT/lib/ld-2.22.so lib
$ cp -a $SYSROOT/lib/libc.so.6 lib
$ cp -a $SYSROOT/lib/libc-2.22.so lib
$ cp -a $SYSROOT/lib/libm.so.6 lib
$ cp -a $SYSROOT/lib/libm-2.22.so lib
```

Repeat this procedure for each program.



*It is only worth doing this to get the very smallest embedded footprint possible. There is a danger that you will miss libraries that are loaded through `dlopen(3)` calls—plugins mostly. We will look at an example with the **name service switch (NSS)** libraries when we come to configure network interfaces later on in this chapter.*

Reducing the size by stripping

Libraries and programs are often compiled with some information stored in symbol tables to aid debugging and tracing. You seldom need these in a production system. A quick and easy way to save space is to strip the binaries of symbol tables. This example shows `libc` before stripping:

```
$ file rootfs/lib/libc-2.22.so
lib/libc-2.22.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (GNU/Linux), dynam
$ ls -og rootfs/lib/libc-2.22.so
-rwxr-xr-x 1 1542572 Mar 3 15:22 rootfs/lib/libc-2.22.so
```

Now, let's see the result of stripping debug information:

```
$ arm-cortex_a8-linux-gnueabi-hf-strip rootfs/lib/libc-2.22.so
$ file rootfs/lib/libc-2.22.so
rootfs/lib/libc-2.22.so: ELF 32-bit LSB shared object, ARM, EABI5 version 1 (GNU/Linux)
$ ls -og rootfs/lib/libc-2.22.so
-rwxr-xr-x 1 1218200 Mar 22 19:57 rootfs/lib/libc-2.22.so
```

In this case, we saved 324,372 bytes, or about 20% of the size of the file before stripping.



Be careful about stripping kernel modules. Some symbols are required by the module loader to relocate the module code, and so the module will fail to load if they are stripped out. Use this command to remove debug symbols while keeping those used for relocation: `strip --strip-unnneeded <module name>`.

Device nodes

Most devices in Linux are represented by device nodes, in accordance with the Unix philosophy that *everything is a file* (except network interfaces, which are sockets). A device node may refer to a block device or a character device. Block devices are mass storage devices, such as SD cards or hard drives. A character device is pretty much anything else, once again with the exception of network interfaces. The conventional location for device nodes is the directory called `/dev`. For example, a serial port maybe represented by the device node called `/dev/ttyS0`.

Device nodes are created using the program named `mknod` (short for **make node**):

```
| mknod <name> <type> <major> <minor>
```

The parameters to `mknod` are as follows:

- `name` is the name of the device node that you want to create.
- `type` is either `c` for character devices or `b` for a block.
- `major` and `minor` are a pair of numbers, which are used by the kernel to route file requests to the appropriate device driver code. There is a list of standard major and minor numbers in the kernel source in the file

`Documentation/devices.txt`.

You will need to create device nodes for all the devices you want to access on your system. You can do so manually using the `mknod` command, as I will illustrate here; or you can create them automatically at runtime using one of the device managers that I will mention later.

In a really minimal root filesystem, you need just two nodes to boot with BusyBox: `console` and `null`. The console only needs to be accessible to `root`, the owner of the device node, so the access permissions are `600`. The null device should be readable and writable by everyone, so the mode is `666`. You can use the `-m` option for `mknod` to set the mode when creating the node. You need to be `root` to create device nodes, as shown here:

```
| $ cd ~/rootfs
```

```
$ sudo mknod -m 666 dev/null c 1 3
$ sudo mknod -m 600 dev/console c 5 1

$ ls -l dev
total 0
crw----- 1 root root 5, 1 Mar 22 20:01 console
crw-rw-rw- 1 root root 1, 3 Mar 22 20:01 null
```

You can delete device nodes using the standard `rm` command: there is no `rmnod` command because, once created, they are just files.

The proc and sysfs filesystems

`proc` and `sysfs` are two pseudo filesystems that give a window onto the inner workings of the kernel. They both represent kernel data as files in a hierarchy of directories: when you read one of the files, the contents you see do not come from disk storage; it has been formatted on-the-fly by a function in the kernel. Some files are also writable, meaning that a kernel function is called with the new data you have written and, if it is of the correct format and you have sufficient permissions, it will modify the value stored in the kernel's memory. In other words, `proc` and `sysfs` provide another way to interact with device drivers and other kernel code. The `proc` and `sysfs` filesystems should be mounted on the directories called `/proc` and `/sys`:

```
# mount -t proc proc /proc
# mount -t sysfs sysfs /sys
```

Although they are very similar in concept, they perform different functions. `proc` has been part of Linux since the early days. Its original purpose was to expose information about processes to user space, hence the name. To this end, there is a directory for each process named `/proc/<PID>`, which contains information about its state. The process list command, `ps`, reads these files to generate its output. In addition, there are files that give information about other parts of the kernel, for example, `/proc/cpuinfo` tells you about the CPU, `/proc/interrupts` has information about interrupts, and so on.

Finally, in `/proc/sys`, there are files that display and control the state and behavior of kernel subsystems, especially scheduling, memory management, and networking. The manual page is the best reference for the files you will find in the `proc` directory, which you can see by typing `man 5 proc`.

On the other hand, the role of `sysfs` is to present the kernel *driver model* to user space. It exports a hierarchy of files relating to devices and device drivers and the way they are connected to each other. I will go into more detail on the Linux driver model when I describe the interaction with device drivers in [Chapter 9](#), *Interfacing with Device Drivers*.

Mounting filesystems

The `mount` command allows us to attach one filesystem to a directory within another, forming a hierarchy of filesystems. The one at the top, which was mounted by the kernel when it booted, is called the **root filesystem**. The format of the `mount` command is as follows:

```
| mount [-t vfstype] [-o options] device directory
```

You need to specify the type of the filesystem, `vfstype`, the block `device` node it resides on, and the `directory` you want to mount it to. There are various options you can give after `-o`; have a look at the manual page *mount(8)* for more information. As an example, if you want to mount an SD card containing an `ext4` filesystem in the first partition onto the directory called `/mnt`, you would type the following code:

```
| # mount -t ext4 /dev/mmcblk0p1 /mnt
```

Assuming the mount succeeds, you would be able to see the files stored on the SD card in the directory: `/mnt`. In some cases, you can leave out the filesystem type, and let the kernel probe the device to find out what is stored there.

Looking at the example of mounting the `proc` filesystem, there is something odd: there is no device node, such as `/dev/proc`, since it is a pseudo filesystem and not a real one. But the `mount` command requires a `device` parameter. Consequently, we have to give a string where `device` should go, but it does not matter much what that string is. These two commands achieve exactly the same result:

```
| # mount -t proc procfs /proc  
| # mount -t proc nodevice /proc
```

The strings "procfs" and "nodevice" are ignored by the `mount` command. It is fairly common to use the filesystem type in the place of the device when mounting pseudo filesystems.

Kernel modules

If you have kernel modules, they need to be installed into the root filesystem, using the kernel make target `modules_install`, as we saw in the last chapter. This will copy them into the directory called `/lib/modules/<kernel version>` together with the configuration files needed by the `modprobe` command.

Be aware that you have just created a dependency between the kernel and the root filesystem. If you update one, you will have to update the other.

Transferring the root filesystem to the target

After having created a skeleton root filesystem in your staging directory, the next task is to transfer it to the target. In the sections that follow, I will describe three possibilities:

- **initramfs:** Also known as a ramdisk, this is a filesystem image that is loaded into RAM by the bootloader. Ramdisks are easy to create and have no dependencies on mass storage drivers. They can be used in fallback maintenance mode when the main root filesystem needs updating. They can even be used as the main root filesystem in small embedded devices, and they are commonly used as the early user space in mainstream Linux distributions. Remember that the contents of the root filesystem are volatile, and any changes you make in the root filesystem at runtime will be lost when the system next boots. You would need another storage type to store permanent data such as configuration parameters.
- **Disk image:** This is a copy of the root filesystem formatted and ready to be loaded onto a mass storage device on the target. For example, it could be an image in the `ext4` format ready to be copied onto an SD card, or it could be in the `jffs2` format ready to be loaded into flash memory via the bootloader. Creating a disk image is probably the most common option. There is more information about the different types of mass storage in [Chapter 7, *Creating a Storage Strategy*](#).
- **Network filesystem:** The staging directory can be exported to the network via an NFS server and mounted by the target at boot time. This is often done during the development phase, in preference to repeated cycles of creating a disk image and reloading it onto the mass storage device, which is quite a slow process.

I will start with ramdisk, and use it to illustrate a few refinements to the root filesystem, such as adding usernames and a device manager to create device nodes automatically. Then, I will show you how to create a disk image and how to use NFS to mount the root filesystem over a network.

Creating a boot initramfs

An initial RAM filesystem, or `initramfs`, is a compressed `cpio` archive. `cpio` is an old Unix archive format, similar to TAR and ZIP but easier to decode and so requiring less code in the kernel. You need to configure your kernel with `CONFIG_BLK_DEV_INITRD` to support `initramfs`.

As it happens, there are three different ways to create a boot ramdisk: as a standalone `cpio` archive, as a `cpio` archive embedded in the kernel image, and as a device table which the kernel build system processes as part of the build. The first option gives the most flexibility, because we can mix and match kernels and ramdisks to our heart's content. However, it means that you have two files to deal with instead of one, and not all bootloaders have the facility to load a separate ramdisk. I will show you how to build one into the kernel later.

Standalone initramfs

The following sequence of instructions creates the archive, compresses it, and adds a U-Boot header ready for loading onto the target:

```
$ cd ~/rootfs
$ find . | cpio -H newc -ov --owner root:root > ../initramfs.cpio
$ cd ..
$ gzip initramfs.cpio
$ mkimage -A arm -O linux -T ramdisk -d initramfs.cpio.gz uRamdisk
```

Note that we run `cpio` with the option: `--owner root:root`. This is a quick fix for the file ownership problem mentioned earlier, making everything in the `cpio` archive have UID and GID of 0.

The final size of the `uRamdisk` file is about 2.9 MB with no kernel modules. Add to that 4.4 MB for the kernel `zImage` file and 440 KB for U-Boot, and this gives a total of 7.7 MB of storage needed to boot this board. We are a little way off the 1.44 MB floppy that started it all off. If size was a real problem, you could use one of these options:

- Make the kernel smaller by leaving out drivers and functions you don't need
- Make BusyBox smaller by leaving out utilities you don't need
- Use `musl libc` or `uClibc-ng` in place of `glibc`
- Compile BusyBox statically

Booting the initramfs

The simplest thing we can do is to run a shell on the console so that we can interact with the target. We can do that by adding `rdinit=/bin/sh` to the kernel command line. The next two sections show how to do that for both QEMU and the BeagleBone Black.

Booting with QEMU

QEMU has the option called `-initrd` to load `initramfs` into memory. You should already have from [Chapter 4, *Configuring and Building the Kernel*](#), a `zImage` compiled with the `arm-unknown-linux-gnueabi` toolchain and the device tree binary for the Versatile PB. From this chapter, you should have created `initramfs`, which includes BusyBox compiled with the same toolchain. Now, you can launch QEMU using the script in `MELP/chapter_05/run-qemu-initramfs.sh` or using this command:

```
$ QEMU_AUDIO_DRV=none \  
qemu-system-arm -m 256M -nographic -M versatilepb -kernel zImage \  
-append "console=ttyAMA0 rdinit=/bin/sh" -dtb versatile-pb.dtb \  
-initrd initramfs.cpio.gz
```

You should get a root shell with the prompt `/ #`.

Booting the BeagleBone Black

For the BeagleBone Black, we need the microSD card prepared in [Chapter 4](#), *Configuring and Building the Kernel*, plus a root filesystem built using the `arm-cortex_a8-linux-gnueabi` toolchain. Copy `uRamdisk` you created earlier in this section to the boot partition on the microSD card, and then use it to boot the BeagleBone Black to point that you get a U-Boot prompt. Then enter these commands:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
fatload mmc 0:1 0x81000000 uRamdisk
setenv bootargs console=tty00,115200 rdinit=/bin/sh
bootz 0x80200000 0x81000000 0x80f00000
```

If all goes well, you will get a root shell with the prompt `/ #` on the serial console.

Mounting proc

You will find that on both platforms the `ps` command doesn't work. This is because the `proc` filesystem has not been mounted yet. Try mounting it:

```
| # mount -t proc proc /proc
```

Now, run `ps` again, and you will see the process listing.

A refinement to this setup would be to write a shell script that mounts `proc`, and anything else that needs to be done at boot-up. Then, you could run this script instead of `/bin/sh` at boot. The following snippet gives an idea of how it would work:

```
|#!/bin/sh  
|/bin/mount -t proc proc /proc  
|# Other boot-time commands go here  
|/bin/sh
```

The last line, `/bin/sh`, launches a new shell that gives you an interactive root shell prompt. Using a shell as `init` in this way is very handy for quick hacks, for example, when you want to rescue a system with a broken `init` program. However, in most cases, you would use an `init` program, which we will cover later on in this chapter. But, before this, I want to look at two other ways to load `initramfs`.

Building an initramfs into the kernel image

So far, we have created a compressed `initramfs` as a separate file and used the bootloader to load it into memory. Some bootloaders do not have the ability to load an `initramfs` file in this way. To cope with these situations, Linux can be configured to incorporate `initramfs` into the kernel image. To do this, change the kernel configuration and set `CONFIG_INITRAMFS_SOURCE` to the full path of the `cpio` archive you created earlier. If you are using `menuconfig`, it is in **General setup | Initramfs source file(s)**. Note that it has to be the uncompressed `cpio` file ending in `.cpio`, not the gzipped version. Then, build the kernel.

Bootting is the same as before, except that there is no `ramdisk` file. For QEMU, the command is like this:

```
$ QEMU_AUDIO_DRV=none \  
qemu-system-arm -m 256M -nographic -M versatilepb -kernel zImage \  
-append "console=ttyAMA0 rdinit=/bin/sh" -dtb versatile-pb.dtb
```

For the BeagleBone Black, enter these commands at the U-Boot prompt:

```
fatload mmc 0:1 0x80200000 zImage  
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb  
setenv bootargs console=tty00,115200 rdinit=/bin/sh  
bootz 0x80200000 - 0x80f00000
```

Of course, you must remember to regenerate the `cpio` file each time you change the contents of the root filesystem, and then rebuild the kernel.

Building an initramfs using a device table

A device table is a text file that lists the files, directories, device nodes, and links that go into an archive or filesystem image. The overwhelming advantage is that it allows you to create entries in the archive file that are owned by the `root` user, or any other UID, without having root privileges yourself. You can even create device nodes without needing root privileges. All this is possible because the archive is just a data file. It is only when it is expanded by Linux at boot time that real files and directories get created, using the attributes you have specified.

The kernel has a feature that allows us to use a device table when creating an `initramfs`. You write the device table file, and then point `CONFIG_INITRAMFS_SOURCE` at it. Then, when you build the kernel, it creates the `cpio` archive from the instructions in the device table. At no point do you need root access.

Here is a device table for our simple `rootfs`, but missing most of the symbolic links to BusyBox to make it manageable:

```
dir /bin 775 0 0
dir /sys 775 0 0
dir /tmp 775 0 0
dir /dev 775 0 0
nod /dev/null 666 0 0 c 1 3
nod /dev/console 600 0 0 c 5 1
dir /home 775 0 0
dir /proc 775 0 0
dir /lib 775 0 0
slink /lib/libm.so.6 libm-2.22.so 777 0 0
slink /lib/libc.so.6 libc-2.22.so 777 0 0
slink /lib/ld-linux-armhf.so.3 ld-2.22.so 777 0 0
file /lib/libm-2.22.so /home/chris/rootfs/lib/libm-2.22.so 755 0 0
file /lib/libc-2.22.so /home/chris/rootfs/lib/libc-2.22.so 755 0 0
file /lib/ld-2.22.so /home/chris/rootfs/lib/ld-2.22.so 755 0 0
```

The syntax is fairly obvious:

- `dir <name> <mode> <uid> <gid>`
- `file <name> <location> <mode> <uid> <gid>`
- `nod <name> <mode> <uid> <gid> <dev_type> <maj> <min>`
- `slink <name> <target> <mode> <uid> <gid>`

The commands `dir`, `nod`, and `slink` create a file system object in the `initramfs` `cpio` archive with the name, mode, user ID and group ID given. The `file` command copies the file from the source location into the archive and sets the mode, the user ID, and the group ID.

The task of creating an `initramfs` device table from scratch is made easier by a script in the kernel source code in `scripts/gen_initramfs_list.sh`, which creates a device table from a given directory. For example, to create the `initramfs` device table for directory `rootfs`, and to change the ownership of all files owned by user ID `1000` and group ID `1000` to user and group ID `0`, you would use this command:

```
$ bash linux-stable/scripts/gen_initramfs_list.sh -u 1000 -g 1000 \  
rootfs > initramfs-device-table
```

Note that the script only works with a `bash` shell. If you have a system with a different default shell, as is the case with most Ubuntu configurations, you will find that the script fails. Hence, in the command given previously, I explicitly used `bash` to run the script.

The old initrd format

There is an older format for a Linux ramdisk, known as `initrd`. It was the only format available before Linux 2.6 and is still needed if you are using the mmu-less variant of Linux, uClinux. It is pretty obscure and I will not cover it here. There is more information in the kernel source in `Documentation/initrd.txt`.

The init program

Running a shell, or even a shell script, at boot time is fine for simple cases, but really you need something more flexible. Normally, Unix systems run a program called `init` that starts up and monitors other programs. Over the years, there have been many `init` programs, some of which I will describe in [Chapter 9, Interfacing with Device Drivers](#). For now, I will briefly introduce the `init` from BusyBox.

The `init` program begins by reading the configuration file, `/etc/inittab`. Here is a simple example, which is adequate for our needs:

```
::sysinit:/etc/init.d/rcS
::askfirst:~/bin/ash
```

The first line runs a shell script, `rcs`, when `init` is started. The second line prints the message `Please press Enter to activate this console` to the console and starts a shell when you press *Enter*. The leading `-` before `/bin/ash` means that it will become a login shell, which sources `/etc/profile` and `$HOME/.profile` before giving the shell prompt. One of the advantages of launching the shell like this is that job control is *enabled*. The most immediate effect is that you can use *Ctrl + C* to terminate the current program. Maybe you didn't notice it before but, wait until you run the `ping` program and find you can't stop it!

BusyBox `init` provides a default `inittab` if none is present in the root filesystem. It is a little more extensive than the preceding one.

The script called `/etc/init.d/rcs` is the place to put initialization commands that need to be performed at boot, for example, mounting the `proc` and `sysfs` filesystems:

```
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

Make sure that you make `rcs` executable like this:

```
$ cd ~/rootfs
$ chmod +x etc/init.d/rcs
```

|

You can try it out on QEMU by changing the `-append` parameter like this:

| `-append "console=ttyAMA0 rdinit=/sbin/init"`

For the BeagleBone Black, you need to set the `bootargs` variable in U-Boot as shown here:

| `setenv bootargs console=tty00,115200 rdinit=/sbin/init`

Starting a daemon process

Typically, you would want to run certain background processes at startup. Let's take the log daemon, `syslogd`, as an example. The purpose of `syslogd` is to accumulate log messages from other programs, mostly other daemons. Naturally, BusyBox has an applet for that!

Starting the daemon is as simple as adding a line like this to `etc/inittab`:

```
| ::respawn:/sbin/syslogd -n
```

`respawn` means that if the program terminates, it will be automatically restarted; `-n` means that it should run as a foreground process. The log is written to `/var/log/messages`.



You may also want to start `klogd` in the same way: `klogd` sends kernel log messages to `syslogd` so that they can be logged to permanent storage.

Configuring user accounts

As I have hinted already, it is not good practice to run all programs as root, since if one is compromised by an outside attack, then the whole system is at risk. It is preferable to create unprivileged user accounts and use them where full root is not necessary.

User names are configured in `/etc/passwd`. There is one line per user, with seven fields of information separated by colons, which are in order:

- The login name
- A hash code used to verify the password, or more usually an `x` to indicate that the password is stored in `/etc/shadow`
- The user ID
- The group ID
- A comment field, often left blank
- The user's `home` directory
- (Optional) the shell this user will use

Here is a simple example in which we have user `root` with UID `0`, and user `daemon` with UID `1`:

```
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
```

Setting the shell for user `daemon` to `/bin/false` ensures that any attempt to log on with that name will fail.

Various programs have to read `/etc/passwd` in order to look up user names and UIDs, and so the file has to be world readable. This is a problem if the password hashes are stored in there as well, because a malicious program would be able to take a copy and discover the actual passwords using a variety of cracker programs. Therefore, to reduce the exposure of this sensitive information, the passwords are stored in `/etc/shadow` and `x` is placed in the password field to indicate that this is the case. The file called `/etc/shadow` only needs to be accessed by `root`, so as long as the `root` user is not compromised, the passwords are safe.

The shadow password file consists of one entry per user, made up of nine fields. Here is an example that mirrors the password file shown in the preceding paragraph:

```
| root::10933:0:99999:7:::  
| daemon*:10933:0:99999:7:::
```

The first two fields are the username and the password hash. The remaining seven fields are related to password aging, which is not usually an issue on embedded devices. If you are curious about the full details, refer to the manual page for *shadow(5)*.

In the example, the password for `root` is empty, meaning that `root` can log on without giving a password. Having an empty password for `root` is useful during development but not for production. You can generate or change a password hash by running the `passwd` command on the target, which will write a new hash to `/etc/shadow`. If you want all subsequent root filesystems to have this same password, you could copy this file back to the staging directory.

Group names are stored in a similar way in `/etc/group`. There is one line per group consisting of four fields separated by colons. The fields are here:

- The name of the group
- The group password, usually an `x` character, indicating that there is no group password
- The GID or group ID
- An optional list of users who belong to this group, separated by commas

Here is an example:

```
| root:x:0:  
| daemon:x:1:
```


Adding user accounts to the root filesystem

Firstly, you have to add to your staging directory the files `etc/passwd`, `etc/shadow`, and `etc/group`, as shown in the preceding section. Make sure that the permissions of `shadow` are `0600`. Next, you need to initiate the login procedure by starting a program called `getty`. There is a version of `getty` in `BusyBox`. You launch it from `inittab` using the keyword **respawn**, which restarts `getty` when a login shell is terminated, so `inittab` should read like this:

```
::sysinit:/etc/init.d/rcS
::respawn:/sbin/getty 115200 console
```

Then, rebuild the ramdisk and try it out using `QEMU` or the `BeagleBone Black` as before.

A better way of managing device nodes

Creating device nodes statically with `mknod` is quite hard work and inflexible. There are other ways to create device nodes automatically on demand:

- `devtmpfs`: This is a pseudo filesystem that you mount over `/dev` at boot time. The kernel populates it with device nodes for all the devices that the kernel currently knows about, and it creates nodes for new devices as they are detected at runtime. The nodes are owned by `root` and have default permissions of `0600`. Some well-known device nodes, such as `/dev/null` and `/dev/random`, override the default to `0666`. To see exactly how this is done, take a look at the Linux source file: `drivers/char/mem.c` and see how `struct memdev` is initialized.
- `mdev`: This is a BusyBox applet that is used to populate a directory with device nodes and to create new nodes as needed. There is a configuration file, `/etc/mdev.conf`, which contains rules for ownership and the mode of the nodes.
- `udev`: This is the mainstream equivalent of `mdev`. You will find it on desktop Linux and in some embedded devices. It is very flexible and a good choice for higher end embedded devices. It is now part of `systemd`.



Although both `mdev` and `udev` create the device nodes themselves, it is more usual to let `devtmpfs` do the job and use `mdev/udev` as a layer on top to implement the policy for setting ownership and permissions.

An example using devtmpfs

Support for the `devtmpfs` filesystem is controlled by kernel configuration variable: `CONFIG_DEVTMPFS`. It is not enabled in the default configuration of the ARM Versatile PB, so if you want to try out the following using this target, you will have to go back and enable this option. Trying out `devtmpfs` is as simple as entering this command:

```
| # mount -t devtmpfs devtmpfs /dev
```

You will notice that afterward, there are many more device nodes in `/dev`. For a permanent fix, add this to `/etc/init.d/rcS`:

```
|#!/bin/sh  
|mount -t proc proc /proc  
|mount -t sysfs sysfs /sys  
|mount -t devtmpfs devtmpfs /dev
```

If you enable `CONFIG_DEVTMPFS_MOUNT` in your kernel configuration, the kernel will automatically mount `devtmpfs` just after mounting the root filesystem. However, this option has no effect when booting `initramfs`, as we are doing here.

An example using mdev

While `mdev` is a bit more complex to set up, it does allow you to modify the permissions of device nodes as they are created. You begin by running `mdev` with the `-s` option, which causes it to scan the `/sys` directory looking for information about current devices. From this information, it populates the `/dev` directory with the corresponding nodes. If you want to keep track of new devices coming online and create nodes for them as well, you need to make `mdev` a hot plug client by writing to `/proc/sys/kernel/hotplug`. These additions to `/etc/init.d/rcs` will achieve all of this:

```
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

The default mode is `660` and the ownership is `root:root`. You can change this by adding rules in `/etc/mdev.conf`. For example, to give the `null`, `random`, and `urandom` devices their correct modes, you would add this to `/etc/mdev.conf`:

```
null root:root 666
random root:root 444
urandom root:root 444
```

The format is documented in the BusyBox source code in `docs/mdev.txt`, and there are more examples in the directory named `examples`.

Are static device nodes so bad after all?

Statically created device nodes do have one advantage over running a device manager: they don't take any time during boot to create. If minimizing boot time is a priority, using statically-created device nodes will save a measurable amount of time.

Configuring the network

Next, let's look at some basic network configurations so that we can communicate with the outside world. I am assuming that there is an Ethernet interface, `eth0`, and that we only need a simple IPv4 configuration.

These examples use the network utilities that are part of BusyBox, and they are sufficient for a simple use case, using the old-but-reliable `ifup` and `ifdown` programs. You can read the manual pages for both to get the details. The main network configuration is stored in `/etc/network/interfaces`. You will need to create these directories in the staging directory:

```
etc/network
etc/network/if-pre-up.d
etc/network/if-up.d
var/run
```

For a static IP address, `/etc/network/interfaces` would look like this:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet static
    address 192.168.1.101
    netmask 255.255.255.0
    network 192.168.1.0
```

For a dynamic IP address allocated using DHCP, `/etc/network/interfaces` would look like this:

```
auto lo
iface lo inet loopback

auto eth0
iface eth0 inet dhcp
```

You will also have to configure a DHCP client program. BusyBox has one named `udhcpd`. It needs a shell script that should go in `/usr/share/udhcpd/default.script`. There is a suitable default in the BusyBox source code in the directory `examples/udhcp/simple.script`.

Network components for glibc

glibc uses a mechanism known as the **name service switch (NSS)** to control the way that names are resolved to numbers for networking and users. Usernames, for example, maybe resolved to UIDs via the file `/etc/passwd`, and network services such as HTTP can be resolved to the service port number via `/etc/services`. All this is configured by `/etc/nsswitch.conf`; see the manual page, *nss(5)*, for full details. Here is a simple example that will suffice for most embedded Linux implementations:

```
passwd:    files
group:     files
shadow:    files
hosts:     files dns
networks:  files
protocols: files
services:  files
```

Everything is resolved by the correspondingly named file in `/etc`, except for the host names, which may additionally be resolved by a DNS lookup.

To make this work, you need to populate `/etc` with those files. Networks, protocols, and services are the same across all Linux systems, so they can be copied from `/etc` in your development PC. `/etc/hosts` should, at least, contain the loopback address:

```
| 127.0.0.1 localhost
```

The other files, `passwd`, `group`, and `shadow`, have been described earlier in the section *Configuring user accounts*.

The last piece of the jigsaw is the libraries that perform the name resolution. They are plugins that are loaded as needed based on the contents of `nsswitch.conf`, meaning that they do not show up as dependencies if you use `readelf` or `ldd`. You will simply have to copy them from the toolchain's `sysroot`:

```
$ cd ~/rootfs
$ cp -a $SYSROOT/lib/libnss* lib
$ cp -a $SYSROOT/lib/libresolv* lib
```

Creating filesystem images with device tables

We saw earlier in the section *Creating a boot initramfs* that the kernel has an option to create `initramfs` using a device table. Device tables are really useful because they allow a non-root user to create device nodes and to allocate arbitrary UID and GID values to any file or directory. The same concept has been applied to tools that create other filesystem image formats, as shown in this table:

Filesystem format	Tool
<code>jffs2</code>	<code>mkfs.jffs2</code>
<code>ubifs</code>	<code>mkfs.ubifs</code>
<code>ext2</code>	<code>genext2fs</code>

We will look at `jffs2` and `ubifs` in [Chapter 7, Creating a Storage Strategy](#), when we look at filesystems for flash memory. The third, `ext2`, is a format commonly used for managed flash memory, including SD cards. The example that follows uses `ext2` to create a disk image that can be copied to an SD card.

They each take a device table file with the format `<name> <type> <mode> <uid> <gid> <major> <minor> <start> <inc> <count>`, where the meanings of the fields is as follows:

- `name`:
- `type`: One of the following:
 - `ᄁ`: A regular file
 - `d`: A directory
 - `c`: A character special device file
 - `b`: A block special device file

- `p`: A FIFO (named pipe)
- `uid`: The UID of the file
- `gid`: The GID of the file
- `major` and `minor`: The device numbers (device nodes only)
- `start`, `inc`, and `count`: Allow you to create a group of device nodes starting from the minor number in `start` (device nodes only)

You do not have to specify every file, as you do with the kernel `initramfs` table. You just have to point at a directory—the staging directory—and list the changes and exceptions you need to make in the final filesystem image.

A simple example which populates static device nodes for us is as follows:

```
| /dev d 755 0 0 - - - - -
| /dev/null c 666 0 0 1 3 0 0 -
| /dev/console c 600 0 0 5 1 0 0 -
| /dev/tty00 c 600 0 0 252 0 0 0 -
```

Then, you can use `genext2fs` to generate a filesystem image of 4 MB (that is 4,096 blocks of the default size, 1,024 bytes):

```
| $ genext2fs -b 4096 -d rootfs -D device-table.txt -U rootfs.ext2
```

Now, you can copy the resulting image, `rootfs.ext2`, to an SD card or similar, which we will do next.

Booting the BeagleBone Black

The script called `MELP/format-sdcard.sh` creates two partitions on the micro SD card: one for the boot files and one for the root filesystem. Assuming that you have created the root filesystem image as shown in the previous section, you can use the `dd` command to write it to the second partition. As always, when copying files directly to storage devices like this, make absolutely sure that you know which is the micro SD card. In this case, I am using a built-in card reader, which is the device called `/dev/mmcblk0`, so the command is as follows:

```
| $ sudo dd if=rootfs.ext2 of=/dev/mmcblk0p2
```

Then, slot the micro SD card into the BeagleBone Black, and set the kernel command line to `root=/dev/mmcblk0p2`. The complete sequence of U-Boot commands is as follows:

```
| fatload mmc 0:1 0x80200000 zImage  
| fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb  
| setenv bootargs console=tty00,115200 root=/dev/mmcblk0p2  
| bootz 0x80200000 - 0x80f00000
```

This is an example of mounting a filesystem from a normal block device, such as an SD card. The same principles apply to other filesystem types and we will look at them in more detail in [Chapter 7, Creating a Storage Strategy](#).

Mounting the root filesystem using NFS

If your device has a network interface, it is often useful to mount the root filesystem over the network during development. It gives you access to the almost unlimited storage on your host machine, so you can add in debug tools and executables with large symbol tables. As an added bonus, updates made to the root filesystem on the development machine are made available on the target immediately. You can also access all the target's log files from the host.

To begin with, you need to install and configure an NFS server on your host. On Ubuntu, the package to install is named `nfs-kernel-server`:

```
| $ sudo apt-get install nfs-kernel-server
```

The NFS server needs to be told which directories are being exported to the network, which is controlled by `/etc/exports`. There is one line for each export. The format is described in the manual page *exports(5)*. As an example, to export the root filesystem on my host, I have this:

```
| /home/chris/rootfs *(rw,sync,no_subtree_check,no_root_squash)
```

* exports the directory to any address on my local network. If you wish, you can give a single IP address or a range at this point. There follows a list of options enclosed in parentheses. There must not be any spaces between * and the opening parenthesis. The options are here:

- `rw`: This exports the directory as read-write.
- `sync`: This option selects the synchronous version of the NFS protocol, which is more robust but a little slower than the `async` option.
- `no_subtree_check`: This option disables subtree checking, which has mild security implications, but can improve reliability in some circumstances.
- `no_root_squash`: This option allows requests from user ID 0 to be processed without *squashing* to a different user ID. It is necessary to allow the target to access correctly the files owned by root.

Having made changes to `/etc/exports`, restart the NFS server to pick them up.

Now, you need to set up the target to mount the root filesystem over NFS. For this to work, your kernel has to be configured with `CONFIG_ROOT_NFS`. Then, you can configure Linux to do the mount at boot time by adding the following to the kernel command line:

```
| root=/dev/nfs rw nfsroot=<host-ip>:<root-dir> ip=<target-ip>
```

The options are as follows:

- `rw`: This mounts the root filesystem read-write.
- `nfsroot`: This specifies the IP address of the host, followed by the path to the exported root filesystem.
- `ip`: This is the IP address to be assigned to the target. Usually, network addresses are assigned at runtime, as we have seen in the section *Configuring the network*. However, in this case, the interface has to be configured before the root filesystem is mounted and `init` has been started. Hence it is configured on the kernel command line.



There is more information about NFS root mounts in the kernel source in `Documentation/filesystems/nfs/nfsroot.txt`.

Testing with QEMU

The following script creates a virtual network between the network device called `tap0` on the host and `eth0` on the target using a pair of static IPv4 addresses, and then launches QEMU with the parameters to use `tap0` as the emulated interface.

You will need to change the path to the root filesystem to be the full path to your staging directory and maybe the IP addresses if they conflict with your network configuration:

```
#!/bin/bash

KERNEL=zImage
DTB=versatile-pb.dtb
ROOTDIR=/home/chris/rootfs
HOST_IP=192.168.1.1
TARGET_IP=192.168.1.101
NET_NUMBER=192.168.1.0
NET_MASK=255.255.255.0

sudo tuncctl -u $(whoami) -t tap0
sudo ifconfig tap0 ${HOST_IP}
sudo route add -net ${NET_NUMBER} netmask ${NET_MASK} dev tap0
sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"

QEMU_AUDIO_DRV=none
qemu-system-arm -m 256M -nographic -M versatilepb -kernel ${KERNEL} -append "console=tt
```



The script is available in `MELP/chapter_05/run-qemu-nfsroot.sh`.

It should boot up as before, now using the staging directory directly via the NFS export. Any files that you create in that directory will be immediately visible to the target device, and any files created in the device will be visible to the development PC.

Testing with the BeagleBone Black

In a similar way, you can enter these commands at the U-Boot prompt of the BeagleBone Black:

```
setenv serverip 192.168.1.1
setenv ipaddr 192.168.1.101
setenv npath [path to staging directory]
setenv bootargs console=tty00,115200 root=/dev/nfs rw nfsroot=${serverip}:${npath} ip=$
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
bootz 0x80200000 - 0x80f00000
```

There is a U-Boot environment file in `chapter_05/uEnv.txt`, which contains all these commands. Just copy it to the boot partition of the microSD card and U-Boot will do the rest.

Problems with file permissions

The files that you copied into the staging directory will be owned by the UID of the user you are logged on as, typically `1000`. However, the target has no knowledge of this user. What is more, any files created by the target will be owned by users configured by the target, often the `root` user. The whole thing is a mess. Unfortunately, there is no simple way out. The best solution is to make a copy of the staging directory and change ownership to UID and GID to `0`, using the command `sudo chown -R 0:0 *`. Then, export this directory as the NFS mount. It removes the convenience of having just one copy of the root filesystem shared between development and target systems, but, at least, the file ownership will be correct.

Using TFTP to load the kernel

Now that we know how to mount the root filesystem over a network using NFS, you may be wondering if there is a way to load the kernel, device tree, and `initramfs` over the network as well. If we could do this, the only component that needs to be written to storage on the target is the bootloader. Everything else could be loaded from the host machine. It would save time since you would not need to keep reflashing the target, and you could even get work done while the flash storage drivers are still being developed (it happens).

The **Trivial File Transfer Protocol (TFTP)** is the answer to the problem. TFTP is a very simple file transfer protocol, designed to be easy to implement in bootloaders such as U-Boot.

But, firstly, you need to install a TFTP daemon on your development machine. On Ubuntu, you could install the `tftpd-hpa` package, which, by default, grants read-only access to files in the directory `/var/lib/tftpboot`. With `tftpd-hpa` installed and running, copy the files that you want to copy to the target into `/var/lib/tftpboot`, which, for the BeagleBone Black, would be `zImage` and `am335x-boneblack.dtb`. Then enter these commands at the U-Boot Command Prompt:

```
setenv serverip 192.168.1.1
setenv ipaddr 192.168.1.101
tftpboot 0x80200000 zImage
tftpboot 0x80f00000 am335x-boneblack.dtb
setenv npath [path to staging]
setenv bootargs console=tty00,115200 root=/dev/nfs rw nfsroot=${serverip}:${npath} ip=
bootz 0x80200000 - 0x80f00000
```

You may find that the `tftpboot` command hangs, endlessly printing out the letter `T`, which means that the TFTP requests are timing out. There are a number of reasons why this happens, the most common ones being:

- There is an incorrect IP address for `serverip`.
- The TFTP daemon is not running on the server.
- There is a firewall on the server which is blocking the TFTP protocol. Most firewalls do indeed block the TFTP port, 69, by default.

Once you have resolved the problem, U-Boot can load the files from the host machine and boot in the usual way. You can automate the process by putting the commands into a `uEnv.txt` file.

Additional reading

Filesystem Hierarchy Standard, Version 3.0, <http://refspecs.linuxfoundation.org/fhs.shtml>

ramfs, rootfs and initramfs , Rob Landley, October 17, 2005, which is part of the Linux source in `documentation/filesystems/ramfs-rootfs-initramfs.txt`.

Summary

One of the strengths of Linux is that it can support a wide range of root filesystems, and so it can be tailored to suit a wide range of needs. We have seen that it is possible to construct a simple root filesystem manually with a small number of components and that BusyBox is especially useful in this regard. By going through the process one step at a time, it has given us insight into some of the basic workings of Linux systems, including network configuration and user accounts. However, the task rapidly becomes unmanageable as devices get more complex. And, there is the ever-present worry that there may be a security hole in the implementation, which we have not noticed.

In the next chapter, I will show you how using an embedded build system can make the process of creating an embedded Linux system much easier and more reliable. I will start by looking at Buildroot, and then go on to look at the more complex, but powerful, Yocto Project.

Selecting a Build System

In the preceding chapters, we covered the four elements of embedded Linux and showed you step-by-step how to build a toolchain, a bootloader, a kernel, a root filesystem, and then combined them into a basic embedded Linux system. And there are a lot of steps! Now, it is time to look at ways to simplify the process by automating it as much as possible. I will look at how embedded build systems can help and look at two of them in particular: Buildroot and the Yocto Project. Both are complex and flexible tools, which would require an entire book to describe fully how they work. In this chapter, I only want to show you the general ideas behind build systems. I will show you how to build a simple device image to get an overall feel of the system, and then how to make some useful changes using the Nova board example from the previous chapters.

In this chapter, we will cover the following topics:

- Build systems
- Package formats and package managers
- Buildroot
- The Yocto Project

Build systems

I have described the process of creating a system manually, as described in [Chapter 5](#), *Building a Root Filesystem*, as the **Roll Your Own (RYO)** process. It has the advantage that you are in complete control of the software, and you can tailor it to do anything you like. If you want it to do something truly odd but innovative, or if you want to reduce the memory footprint to the smallest size possible, RYO is the way to go. But, in the vast majority of situations, building manually is a waste of time and produces inferior, unmaintainable systems.

The idea of a build system is to automate all the steps I have described up to this point. A build system should be able to build, from upstream source code, some or all of the following:

- A toolchain
- A bootloader
- A kernel
- A root filesystem

Building from upstream source code is important for a number of reasons. It means that you have peace of mind that you can rebuild at any time, without external dependencies. It also means that you have the source code for debugging and also that you can meet your license requirements to distribute the code to users where necessary.

Therefore, to do its job, a build system has to be able to do the following:

1. Download the source code from upstream, either directly from the source code control system or as an archive, and cache it locally.
2. Apply patches to enable cross compilation, fix architecture-dependent bugs, apply local configuration policies, and so on.
3. Build the various components.
4. Create a staging area and assemble a root filesystem.
5. Create image files in various formats ready to be loaded onto the target.

Other things that are useful are as follows:

1. Add your own packages containing, for example, applications or kernel changes.
2. Select various root filesystem profiles: large or small, with and without graphics or other features.
3. Create a standalone SDK that you can distribute to other developers so that they don't have to install the complete build system.
4. Track which open source licenses are used by the various packages you have selected.
5. Have a user-friendly user interface.

In all cases, they encapsulate the components of a system into packages, some for the host and some for the target. Each package is defined by a set of rules to get the source, build it, and install the results in the correct location. There are dependencies between the packages and a build mechanism to resolve the dependencies and build the set of packages required.

Open source build systems have matured considerably over the last few years. There are many around, including the following:

- **Buildroot:** This is an easy-to-use system using GNU make and Kconfig (<https://buildroot.org/>)
- **EmbToolkit:** This is a simple system for generating root filesystems; the only one so far that supports LLVM/Clang out of the box (<https://www.embtoolkit.org>)
- **OpenEmbedded:** This is a powerful system, which is also a core component of the Yocto Project and others (<http://openembedded.org>)
- **OpenWrt:** This is a build tool oriented towards building firmware for wireless routers (<https://openwrt.org>)
- **PTXdist:** This is an open source build system sponsored by Pengutronix (http://www.pengutronix.de/software/ptxdist/index_en.html)
- **The Yocto Project:** This extends the OpenEmbedded core with metadata, tools and documentation: probably the most popular system (<http://www.yoctoproject.org>)

I will concentrate on two of these: Buildroot and the Yocto Project. They approach the problem in different ways and with different objectives.

Buildroot has the primary aim of building root filesystem images, hence the

name, although it can build bootloader and kernel images as well. It is easy to install and configure and generates target images quickly.

The Yocto Project, on the other hand, is more general in the way it defines the target system, and so it can build fairly complex embedded devices. Every component is generated as a binary package, by default, using the RPM format, and then the packages are combined together to make the filesystem image. Furthermore, you can install a package manager in the filesystem image, which allows you to update packages at runtime. In other words, when you build with the Yocto Project, you are, in effect, creating your own custom Linux distribution.

Package formats and package managers

Mainstream Linux distributions are, in most cases, constructed from collections of binary (precompiled) packages in either RPM or DEB format. RPM stands for the Red Hat package manager and is used in Red Hat, Suse, Fedora, and other distributions based on them. Debian and Debian-derived distributions, including Ubuntu and Mint, use the Debian package manager format, DEB. In addition, there is a light-weight format specific to embedded devices known as the **Itsy package format** or **IPK**, which is based on DEB.

The ability to include a package manager on the device is one of the big differentiators between build systems. Once you have a package manager on the target device, you have an easy path to deploy new packages to it and to update the existing ones. I will talk about the implications of this in [Chapter 8, *Updating Software in the Field*](#).

Buildroot

The Buildroot project website is at <http://buildroot.org>.

The current versions of Buildroot are capable of building a toolchain, a bootloader, a kernel, and a root filesystem. It uses GNU `make` as the principal build tool. There is good online documentation at <http://buildroot.org/docs.html>, including *The Buildroot user manual* at <https://buildroot.org/downloads/manual/manual.html>.

Background

Buildroot was one of the first build systems. It began as part of the uClinux and uClibc projects as a way of generating a small root filesystem for testing. It became a separate project in late 2001 and continued to evolve through to 2006, after which it went into a rather dormant phase. However, since 2009, when Peter Korsgaard took over stewardship, it has been developing rapidly, adding support for `glibc` based toolchains and a greatly increased number of packages and target boards.

As a matter of interest, Buildroot is also the ancestor of another popular build system, **OpenWrt** (<http://wiki.openwrt.org>), which forked from Buildroot around 2004. The primary focus of OpenWrt is to produce software for wireless routers, and so the package mix is oriented toward the networking infrastructure. It also has a runtime package manager using the IPK format so that a device can be updated or upgraded without a complete reflash of the image. However, Buildroot and OpenWrt have diverged to such an extent that they are now almost completely different build systems. Packages built with one are not compatible with the other.

Stable releases and long-term support

The Buildroot developers produce stable releases four times a year, in February, May, August, and November. They are marked by git tags of the form: <year>.02, <year>.05, <year>.08, and <year>.11. From time to time, a release is marked for **Long Term Support (LTS)**, which means that there will be point releases to fix security and other important bugs for 12 months after the initial release. The 2017.02 release is the first to receive the LTS label.

Installing

As usual, you can install Buildroot either by cloning the repository or downloading an archive. Here is an example of obtaining version 2017.02.1, which was the latest stable version at the time of writing:

```
$ git clone git://git.buildroot.net/buildroot -b 2017.02.1  
$ cd buildroot
```

The equivalent TAR archive is available at <http://buildroot.org/downloads>.

Next, you should read the section titled **System requirement** from The Buildroot user manual available at <http://buildroot.org/downloads/manual/manual.html>, and make sure that you have installed all the packages listed there.

Configuring

Buildroot uses the kernel Kconfig/Kbuild mechanism, which I described in the section *Understanding kernel configuration* in [Chapter 4, Configuring and Building the Kernel](#). You can configure Buildroot from scratch directly using `make menuconfig` (`xconfig` OR `gconfig`), or you can choose one of the 100+ configurations for various development boards and the QEMU emulator, which you can find stored in the directory, `configs/`. Typing `make list-defconfigs` lists all the default configurations.

Let's begin by building a default configuration that you can run on the ARM QEMU emulator:

```
$ cd buildroot
$ make qemu_arm_versatile_defconfig
$ make
```



*You do not tell `make` how many parallel jobs to run with a `-j` option: Buildroot will make optimum use of your CPUs all by itself. If you want to limit the number of jobs, you can run `make menuconfig` and look under the *Build options*.*

The build will take half an hour to an hour or more depending on the capabilities of your host system and the speed of your link to the internet. It will download approximately 220 MiB of code and will consume about 3.5 GiB of disk space. When it is complete, you will find that two new directories have been created:

- `dl/`: This contains archives of the upstream projects that Buildroot has built
- `output/`: This contains all the intermediate and final compiled resources

You will see the following in `output/`:

- `build/`: Here, you will find the build directory for each component.
- `host/`: This contains various tools required by Buildroot that run on the host, including the executables of the toolchain (in `output/host/usr/bin`).
- `images/`: This is the most important of all since it contains the results of the build. Depending on what you selected when configuring, you will find a bootloader, a kernel, and one or more root filesystem images.

- `staging/`: This is a symbolic link to the `sysroot` of the toolchain. The name of the link is a little confusing, because it does not point to a staging area as I defined it in [Chapter 5, *Building a Root Filesystem*](#).
- `target/`: This is the staging area for the `root` directory. Note that you cannot use it as a root filesystem as it stands because the file ownership and the permissions are not set correctly. Buildroot uses a device table, as described in the previous chapter, to set ownership and permissions when the filesystem image is created in the `image/` directory.

Running

Some of the sample configurations have a corresponding entry in the directory `board/`, which contains custom configuration files and information about installing the results on the target. In the case of the system you have just built, the relevant file is `board/qemu/arm-versatile/readme.txt`, which tells you how to start QEMU with this target. Assuming that you have already installed `qemu-system-arm` as described in [Chapter 1, Starting Out](#), you can run it using this command:

```
$ qemu-system-arm -M versatilepb -m 256 \  
-kernel output/images/zImage \  
-dtb output/images/versatile-pb.dtb \  
-drive file=output/images/rootfs.ext2,if=scsi,format=raw \  
-append "root=/dev/sda console=ttyAMA0,115200" \  
-serial stdio -net nic,model=rtl8139 -net user
```

There is a script named `MELP/chapter_06/run-qemu-buildroot.sh` in the book code archive, which includes that command. When QEMU boots up, you should see the kernel boot messages appear in the same terminal window where you started QEMU, followed by a login prompt:

```
Booting Linux on physical CPU 0x0  
Linux version 4.9.6 (chris@chris-xps) (gcc version 5.4.0  
(Buildroot 2017.02.1) ) #1 Tue Apr 18 10:30:03 BST 2017  
CPU: ARM926EJ-S [41069265] revision 5 (ARMv5TEJ), cr=00093177  
[...]  
VFS: Mounted root (ext2 filesystem) readonly on device 8:0.  
devtmpfs: mounted  
Freeing unused kernel memory: 132K (c042f000 - c0450000)  
This architecture does not have kernel memory protection.  
EXT4-fs (sda): warning: mounting unchecked fs, running e2fsck is recommended  
EXT4-fs (sda): re-mounted. Opts: block_validity,barrier,user_xattr,errors=remount-ro  
Starting logging: OK  
Initializing random number generator... done.  
Starting network: 8139cp 0000:00:0c.0 eth0: link up, 100Mbps, full-duplex, lpa 0x05E1  
udhcpc: started, v1.26.2  
udhcpc: sending discover  
udhcpc: sending select for 10.0.2.15  
udhcpc: lease of 10.0.2.15 obtained, lease time 86400  
deleting routers  
adding dns 10.0.2.3  
OK  
  
Welcome to Buildroot  
buildroot login:
```

Log in as `root`, no password.

You will see that QEMU launches a black window in addition to the one with the kernel boot messages. It is there to display the graphics frame buffer of the target. In this case, the target never writes to the framebuffer, which is why it appears black. To close QEMU, either type `ctrl-Alt-2` to get to the QEMU console and then type `quit`, or just close the framebuffer window.

Creating a custom BSP

Next, let's use Buildroot to create a BSP for our Nova board using the same versions of U-Boot and Linux from earlier chapters. You can see the changes I made to Buildroot during this section in the book code archive in

`MELP/chapter_06/buildroot.`

The recommended places to store your changes are here:

- `board/<organization>/<device>`: This contains any patches, binary blobs, extra build steps, configuration files for Linux, U-Boot, and other components
- `configs/<device>_defconfig`: This contains the default configuration for the board
- `package/<organization>/<package_name>`: This is the place to put any additional packages for this board

Let's begin by creating a directory to store changes for the Nova board:

```
| $ mkdir -p board/melp/nova
```

Next, clean the artifacts from any previous build, which you should always do when changing configurations:

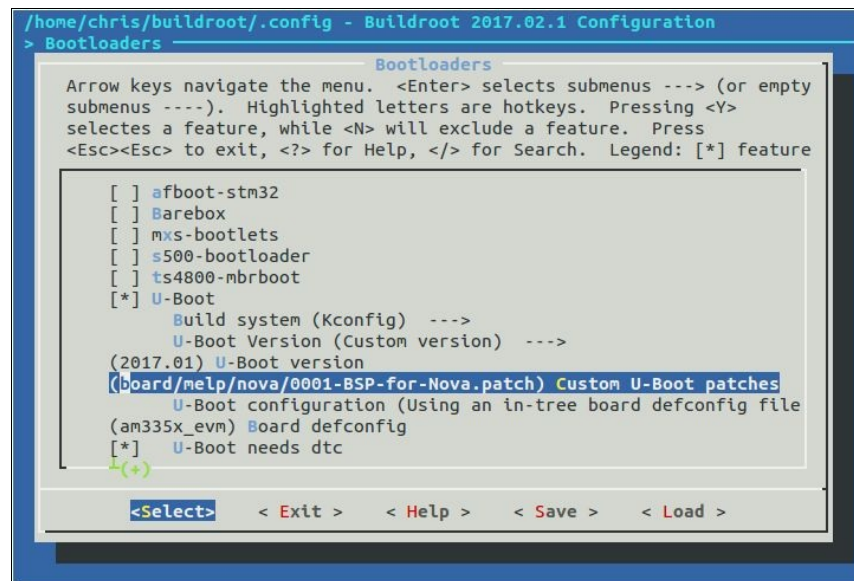
```
| $ make clean
```

Now, select the configuration for the BeagleBone, which we are going to use as the basis of the Nova configuration.

```
| $ make beaglebone_defconfig
```

U-Boot

In [Chapter 3, All About Bootloaders](#), we created a custom bootloader for Nova, based on the 2017.01 version of U-Boot and created a patch file for it, which you will find in `MELP/chapter_03/0001-BSP-for-Nova.patch`. We can configure Buildroot to select the same version and apply our patch. Begin by copying the patch file into `board/melp/nova`, and then use `make menuconfig` to set the U-Boot version to 2017.01, the patch file to `board/melp/nova/0001-BSP-for-Nova.patch`, and the board name to Nova, as shown in this screenshot:



We also need a U-Boot script to load the Nova device tree and the kernel from the SD card. We can put the file into `board/melp/nova/uEnv.txt`. It should contain these commands:

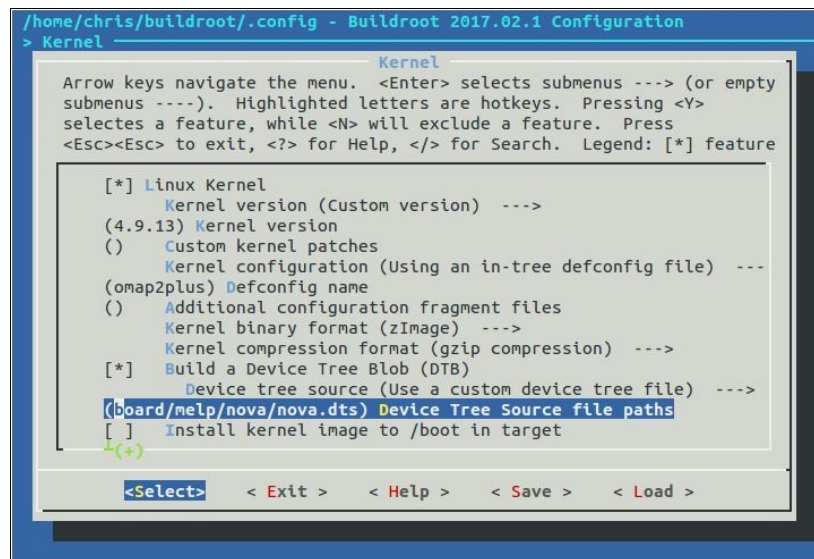
```
bootpart=0:1
bootdir=

bootargs=console=tty00,115200n8 root=/dev/mmcblk0p2 rw rootfstype=ext4 rootwait

uenvcmd=fatload mmc 0:1 88000000 nova.dtb;fatload mmc 0:1 82000000 zImage;
bootz 82000000 - 88000000
```

Linux

In [Chapter 4](#), *Configuring and Building the Kernel*, we based the kernel on Linux 4.9.13 and supplied a new device tree, which is in `MELP/chapter_04/nova.dts`. Copy the device tree to `board/melp/nova`, change the Buildroot kernel configuration to select Linux version 4.9.13, and the device tree source to `board/melp/nova/nova.dts`, as shown in the following screenshot:



```
/home/chris/buildroot/.config - Buildroot 2017.02.1 Configuration
> Kernel

Kernel
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus --->). Highlighted letters are hotkeys. Pressing <Y>
selectes a feature, while <N> will exclude a feature. Press
<Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature

[*] Linux Kernel
    Kernel version (Custom version) --->
    (4.9.13) Kernel version
    ( ) Custom kernel patches
    Kernel configuration (Using an in-tree defconfig file) ---
    (omap2plus) Defconfig name
    ( ) Additional configuration fragment files
    Kernel binary format (zImage) --->
    Kernel compression format (gzip compression) --->
    [*] Build a Device Tree Blob (DTB)
        Device tree source (Use a custom device tree file) --->
        (board/melp/nova/nova.dts) Device Tree Source file paths
    [ ] Install kernel image to /boot in target

<Select> < Exit > < Help > < Save > < Load >
```

We will also have to change the kernel series to be used for kernel headers to match the kernel being built:

```
/home/chris/buildroot/.config - Buildroot 2017.02.1 Configuration
> Toolchain

Toolchain
Arrow keys navigate the menu. <Enter> selects submenus ---> (or empty
submenus ----). Highlighted letters are hotkeys. Pressing <Y>
selectes a feature, while <N> will exclude a feature. Press
<Esc><Esc> to exit, <?> for Help, </> for Search. Legend: [*] feature

Toolchain type (Buildroot toolchain) --->
*** Toolchain Buildroot Options ***
(buildroot) custom toolchain vendor name
C library (uClibc-ng) --->
*** Kernel Header Options ***
Kernel Headers (Same as kernel being built) --->
Custom kernel headers series (4.9.x) --->
*** uClibc Options ***
(package/uclibc/uClibc-ng.config) uClibc configuration file to us
() Additional uClibc configuration fragment files
[ ] Enable RPC support
[ ] Enable WCHAR support
[ ] Enable toolchain locale/i18n support
+ (+)

<Select> < Exit > < Help > < Save > < Load >
```

Build

In the last stage of the build, Buildroot uses a tool named `genimage` to create an image for the SD card that we can copy directory to the card. We need a configuration file to layout the image in the right way. We will name the file `board/melp/nova/genimage.cfg` and populate it as shown here:

```
image boot.vfat {
    vfat {
        files = {
            "MLO",
            "u-boot.img",
            "zImage",
            "uEnv.txt",
            "nova.dtb",
        }
    }
    size = 16M
}

image sdcard.img {
    himage {
    }

    partition u-boot {
        partition-type = 0xC
        bootable = "true"
        image = "boot.vfat"
    }

    partition rootfs {
        partition-type = 0x83
        image = "rootfs.ext4"
        size = 512M
    }
}
```

This will create a file named `sdcard.img`, which contains two partitions named `u-boot` and `rootfs`. The first contains the boot files listed in `boot.vfat`, and the second contains the root filesystem image named `rootfs.ext4`, which will be generated by Buildroot.

Finally, we need to create a `post image` script that will call `genimage`, and so create the SD card image. We will put it in `board/melp/nova/post-image.sh`:

```
#!/bin/sh
BOARD_DIR="$(dirname $0)"

cp ${BOARD_DIR}/uEnv.txt $BINARIES_DIR/uEnv.txt
```

```

GENIMAGE_CFG="${BOARD_DIR}/genimage.cfg"
GENIMAGE_TMP="${BUILD_DIR}/genimage.tmp"

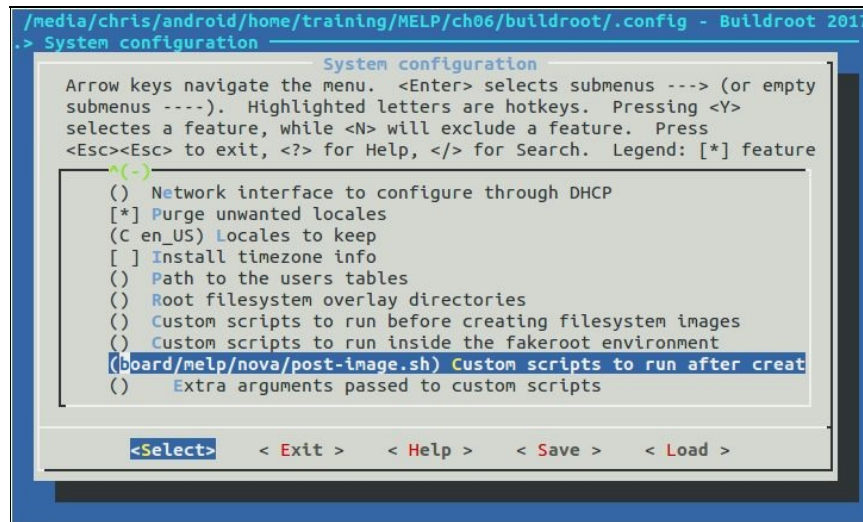
rm -rf "${GENIMAGE_TMP}"

genimage \
--rootpath "${TARGET_DIR}" \
--tmppath "${GENIMAGE_TMP}" \
--inputpath "${BINARIES_DIR}" \
--outputpath "${BINARIES_DIR}" \
--config "${GENIMAGE_CFG}"

```

This copies the `uEnv.txt` script into the `output/images` directory and runs `genimage` with our configuration file.

Now, we can run `menuconfig` again and to change the System configuration option, Custom scripts to run before creating filesystem images, to run our `post-image.sh` script, as shown in this screenshot:



Finally, you can build Linux for the Nova board just by typing `make`. When it has finished, you will see these files in the directory, `output/images/`:

```

boot.vfat    rootfs.ext2    sdcard.img    uEnv.txt
MLO          rootfs.ext4    u-boot.img    zImage
nova.dtb     rootfs.tar     u-boot-spl.bin

```

To test it, put a microSD card in the card reader, unmount any partitions that are auto mounted, and then copy `sdcard.img` to the root of the SD card. There is no need to format it beforehand, as we did in the previous chapter, because `genimage` has created the exact disk layout required. In the following example, my SD card reader is `/dev/mmcb1k0`:

```
| $ sudo umount /dev/mmcblk0*  
| $ sudo dd if=output/images/sdcard.img of=/dev/mmcblk0 bs=1M
```

Put the SD card into the BeagleBone Black and power on while pressing the boot button to force it to load from the SD card. You should see that it boots up with our selected versions of U-Boot, Linux, and with the Nova device tree.

Having shown that our custom configuration for the Nova board works, it would be nice to keep a copy of the configuration so that you and others can use it again, which you can do with this command:

```
| $ make savedefconfig BR2_DEFCONFIG=configs/nova_defconfig
```

Now, you have a Buildroot configuration for the Nova board. Subsequently, you can retrieve this configuration by typing the following command:

```
| $ make nova_defconfig
```

Adding your own code

Suppose there is a program that you have developed and that you want to include it in the build. You have two options: firstly to build it separately using its own build system, and then roll the binary into the final build as an overlay. Secondly, you could create a Buildroot package that can be selected from the menu and built like any other.

Overlays

An overlay is simply a directory structure that is copied over the top of the Buildroot root filesystem at a late stage in the build process. It can contain executables, libraries, and anything else you may want to include. Note that any compiled code must be compatible with the libraries deployed at runtime, which, in turn, means that it must be compiled with the same toolchain that Buildroot uses. Using the Buildroot toolchain is quite easy. Just add it to `PATH`:

```
| $ PATH=<path_to_buildroot>/output/host/usr/bin:$PATH
```

The prefix for the toolchain is `<ARCH>-linux-`. So, to compile a simple program, you would do something like this:

```
| $ PATH=/home/chris/buildroot/output/host/usr/bin:$PATH  
| $ arm-linux-gcc helloworld.c -o helloworld
```

Once you have compiled your program with the correct toolchain, you just need to install the executables and other supporting files into a staging area, and mark it as an overlay for Buildroot. For the `helloworld` example, you might put it in the `board/melp/nova` directory:

```
| $ mkdir -p board/melp/nova/overlay/usr/bin  
| $ cp helloworld board/melp/nova/overlay/usr/bin
```

Finally, you set `BR2_ROOTFS_OVERLAY` to the path to the overlay. It can be configured in `menuconfig` with the option, System configuration | Root filesystem overlay directories.

Adding a package

Buildroot packages are stored in the package directory, over 2,000 of them, each in its own subdirectory. A package consists of at least two files: `Config.in`, containing the snippet of Kconfig code required to make the package visible in the configuration menu, and a makefile named `<package_name>.mk`. Note that the package does not contain the code, just the instructions to get the code by downloading a tarball, doing `git pull` or whatever is necessary to obtain the upstream source.

The makefile is written in a format expected by Buildroot and contains directives that allow Buildroot to download, configure, compile, and install the program. Writing a new package makefile is a complex operation, which is covered in detail in the Buildroot user manual. Here is an example which shows you how to create a package for a simple program stored locally, such as our `helloworld` program.

Begin by creating the `package/helloworld/` subdirectory with a configuration file, `Config.in`, which looks like this:

```
config BR2_PACKAGE_HELLOWORLD
    bool "helloworld"
    help
        A friendly program that prints Hello World! every 10s
```

The first line must be of the format, `BR2_PACKAGE_<uppercase package name>`. This is followed by a Boolean and the package name, as it will appear in the configuration menu, which will allow a user to select this package. The `help` section is optional (but hopefully useful).

Next, link the new package into the Target Packages menu by editing `package/Config.in` and sourcing the configuration file as mentioned in the preceding section. You could append this to an existing submenu but, in this case, it seems neater to create a new submenu, which only contains our package:

```
menu "My programs"
    source "package/helloworld/Config.in"
endmenu
```

Then, create a makefile, `package/helloworld/helloworld.mk`, to supply the data needed by Buildroot:

```
HELLOWORLD_VERSION = 1.0.0
HELLOWORLD_SITE = /home/chris/MELP/helloworld
HELLOWORLD_SITE_METHOD = local

define HELLOWORLD_BUILD_CMDS
    $(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -C $(@D) all
endef

define HELLOWORLD_INSTALL_TARGET_CMDS
    $(INSTALL) -D -m 0755 $(@D)/helloworld $(TARGET_DIR)/usr/bin/helloworld
endef

$(eval $(generic-package))
```

You can find my helloworld package in the book code archive in `MELP/chapter_06/buildroot/package/helloworld` and the source code for the program in `MELP/chapter_06/helloworld`. The location of the code is hard coded to a local pathname. In a more realistic case, you would get the code from a source code system or from a central server of some kind: there are details of how to do this in the Buildroot user manual and plenty of examples in other packages.

License compliance

Buildroot is based on an open source software as are the packages it compiles. At some point during the project, you should check the licenses, which you can do by running:

```
| $ make legal-info
```

The information is gathered into `output/legal-info/`. There are summaries of the licenses used to compile the host tools in `host-manifest.csv` and, on the target, in `manifest.csv`. There is more information in the `README` file and in the Buildroot user manual.

The Yocto Project

The Yocto Project is a more complex beast than Buildroot. Not only can it build toolchains, bootloaders, kernels, and root filesystems as Buildroot can, but it can generate an entire Linux distribution for you with binary packages that can be installed at runtime. The Yocto Project is primarily a group of recipes, similar to Buildroot packages but written using a combination of Python and shell script, together with a task scheduler called **BitBake** that produces whatever you have configured, from the recipes.

There is plenty of online documentation at <https://www.yoctoproject.org/>.

Background

The structure of the Yocto Project makes more sense if you look at the background first. It's roots are in **OpenEmbedded**, <http://openembedded.org/>, which, in turn, grew out of a number of projects to port Linux to various hand-held computers, including the Sharp Zaurus and the Compaq iPaq. OpenEmbedded, which came to life in 2003 as the build system for those hand-held computers. Soon after, other developers began to use it as a general build asystem for devices running embedded Linux. It was developed, and continues to be developed, by an enthusiastic community of programmers.

The OpenEmbedded project is set out to create a set of binary packages using the compact IPK format, which could then be combined in various ways to create a target system and be installed on the target at runtime. It did this by creating recipes for each package and using BitBake as the task scheduler. It was, and is, very flexible. By supplying the right metadata, you can create an entire Linux distribution to your own specification. One that, which is fairly well-known is the **Ångström Distribution**, <http://www.angstrom-distribution.org>, but there are many others as well.

At some time in 2005, Richard Purdie, then a developer at OpenedHand, created a fork of OpenEmbedded, which had a more conservative choice of packages and created releases that were stable over a period of time. He named it **Poky** after the Japanese snack (if you are worried about these things, Poky is pronounced to rhyme with hockey). Although Poky was a fork, OpenEmbedded and Poky continued to run alongside each other, sharing updates and keeping the architectures more or less in step. Intel brought out OpenedHand in 2008, and they transferred Poky Linux to the Linux Foundation in 2010 when they formed the Yocto Project.

Since 2010, the common components of OpenEmbedded and Poky have been combined into a separate project known as **OpenEmbedded Core** or just **OE-Core**.

Therefore, the Yocto Project collects together several components, the most

important of which are the following:

- **OE-Core:** This is the core metadata, which is shared with OpenEmbedded
- **BitBake:** This is the task scheduler, which is shared with OpenEmbedded and other projects
- **Poky:** This is the reference distribution
- **Documentation:** This is the user's manuals and developer's guides for each component
- **Toaster:** This is a web-based interface to BitBake and its metadata
- **ADT Eclipse:** This is a plugin for Eclipse

The Yocto Project provides a stable base, which can be used as it is or can be extended using **meta layers**, which I will discuss later in this chapter. Many SoC vendors provide BSPs for their devices in this way. Meta layers can also be used to create extended or just different build systems. Some are open source, such as the Ångström Distribution, and others are commercial, such as MontaVista Carrier Grade Edition, Mentor Embedded Linux, and Wind River Linux. The Yocto Project has a branding and compatibility testing scheme to ensure that there is interoperability between components. You will see statements like **Yocto Project compatible** on various web pages.

Consequently, you should think of the Yocto Project as the foundation of a whole sector of embedded Linux, as well as being a complete build system in its own right.



You maybe wondering about the name, Yocto. `yocto` is the SI prefix for 10^{-24} , in the same way that `micro` is 10^{-6} . Why name the project Yocto? It was partly to indicate that it could build very small Linux systems (although, to be fair, so can other build systems), but also to steal a march on the Ångström Distribution, which is based on OpenEmbedded. An Ångström is 10^{-10} . That's huge, compared to a `yocto`!

Stable releases and supports

Usually, there is a release of the Yocto Project every six months: in April and October. They are principally known by the code name, but it is useful to know the version numbers of the Yocto Project and Poky as well. Here is a table of the six most recent releases at the time of writing:

Code name	Release date	Yocto version	Poky version
Morty	October 2016	2.2	16
Krogoth	April 2016	2.1	15
Jethro	October 2015	2.0	14
Fido	April 2015	1.8	13
Dizzy	October 2014	1.7	12
Daisy	April 2014	1.6	11

The stable releases are supported with security and critical bug fixes for the current release cycle and the next cycle. In other words, each version is supported for approximately 12 months after the release. As with Buildroot, if you want continued support, you can update to the next stable release, or you can backport changes to your version. You also have the option of commercial support for periods of several years with the Yocto Project from operating system vendors, such as Mentor Graphics, Wind River, and many others.

Installing the Yocto Project

To get a copy of the Yocto Project, you can either clone the repository, choosing the code name as the branch, which is `morty` in this case:

```
| $ git clone -b morty git://git.yoctoproject.org/poky.git
```

You can also download the archive from <http://downloads.yoctoproject.org/releases/yocto/yocto-2.2/poky-morty-16.0.0.tar.bz2>. In the first case, you will find everything in the directory, `poky/`, in the second case, `poky-morty-16.0.0/`.

In addition, you should read the section titled **System Requirements** from the *Yocto Project Reference Manual* (<http://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html#detailed-supported-distros>); and, in particular, you should make sure that the packages listed there are installed on your host computer.

Configuring

As with Buildroot, let's begin with a build for the QEMU ARM emulator. Begin by sourcing a script to set up the environment:

```
| $ cd poky  
| $ source oe-init-build-env
```

This creates a working directory for you named `build/` and makes it the current directory. All of the configuration, intermediate, and target image files will be put in this directory. You must source this script each time you want to work on this project.

You can choose a different working directory by adding it as a parameter to `oe-init-build-env`, for example:

```
| $ source oe-init-build-env build-qemuarm
```

This will put you into the directory: `build-qemuarm/`. This way you can have several build directories, each for a different project: you choose which one you want to work with through the parameter to `oe-init-build-env`.

Initially, the build directory contains only one subdirectory named `conf/`, which contains the configuration files for this project:

- `local.conf`: This contains a specification of the device you are going to build and the build environment.
- `bblayers.conf`: This contains paths of the meta layers you are going to use. I will describe layers later on.
- `templateconf.cfg`: This contains the name of a directory, which contains various `conf` files. By default, it points to `meta-poky/conf/`.

For now, we just need to set the `MACHINE` variable in `local.conf` to `qemuarm` by removing the comment character (`#`) at the start of this line:

```
| MACHINE ?= "qemuarm"
```

Building

To actually perform the build, you need to run BitBake, telling it which root filesystem image you want to create. Some common images are as follows:

- `core-image-minimal`: This is a small console-based system which is useful for tests and as the basis for custom images.
- `core-image-minimal-initramfs`: This is similar to `core-image-minimal`, but built as a ramdisk.
- `core-image-x11`: This is a basic image with support for graphics through an X11 server and the `xterminal` terminal app.
- `core-image-sato`: This is a full graphical system based on Sato, which is a mobile graphical environment built on X11, and GNOME. The image includes several apps including a Terminal, an editor, and a file manager.

By giving BitBake the final target, it will work backwards and build all the dependencies first, beginning with the toolchain. For now, we just want to create a minimal image to see how it works:

```
| $ bitbake core-image-minimal
```

The build is likely to take some time, probably more than an hour. It will download about 4 GiB of source code, and it will consume about about 24 GiB of disk space. When it is complete, you will find several new directories in the build directory including `downloads/`, which contains all the source downloaded for the build, and `tmp/`, which contains most of the build artifacts. You should see the following in `tmp/`:

- `work/`: This contains the build directory and the staging area for the root filesystem.
- `deploy/`: This contains the final binaries to be deployed on the target:
 - `deploy/images/[machine name]/`: Contains the bootloader, the kernel, and the root filesystem images ready to be run on the target
 - `deploy/rpm/`: This contains the RPM packages that went to make up the images
 - `deploy/licenses/`: This contains the license files extracted from

each package

Running the QEMU target

When you build a QEMU target, an internal version of QEMU is generated, which removes the need to install the QEMU package for your distribution, and thus avoids version dependencies. There is a wrapper script named `runqemu` to run this version of QEMU.

To run the QEMU emulation, make sure that you have sourced `oe-init-build-env`, and then just type this:

```
| $ runqemu qemuarm
```

In this case, QEMU has been configured with a graphic console so that the boot messages and login prompt appear in the black framebuffer, as shown in the following screenshot:

```
[ 8.179808] md: If you don't use raid, use raid=noautodetect
[ 8.193152] md: Autodetecting RAID arrays.
[ 8.197292] md: Scanned 0 and added 0 devices.
[ 8.201323] md: autorun ...
[ 8.204339] md: ... autorun DONE.
[ 8.214568] EXT4-fs (vda): couldn't mount as ext3 due to feature incompatibilities
[ 8.224943] EXT4-fs (vda): couldn't mount as ext2 due to feature incompatibilities
[ 8.267365] EXT4-fs (vda): mounted filesystem with ordered data mode. Opts: (null)
[ 8.276284] UFS: Mounted root (ext4 filesystem) on device 253:0.
[ 8.283072] devtmpfs: mounted
[ 8.305016] Freeing unused kernel memory: 412K (c0945000 - c09ac000)
[ 8.310417] This architecture does not have kernel memory protection.
INIT: version 2.88 booting

Please wait: booting...
Starting udev
[ 9.316513] udevd[115]: starting version 3.2
[ 9.436544] udevd[116]: starting eudev-3.2
[ 12.073243] EXT4-fs (vda): re-mounted. Opts: data=ordered
Populating dev cache
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 2.2.1 qemuarm /dev/tty1
qemuarm login:
```

You can login as `root`, without a password. You can close down QEMU by closing the framebuffer window.

You can launch QEMU without the graphic window by adding `nographic` to the command line:

```
| $ runqemu qemuarm nographic
```

In this case, you close QEMU using the key sequence *Ctrl* + *A* and then *x*.

The `runqemu` script has many other options. Type `runqemu help` for more information.

Layers

The metadata for the Yocto Project is structured into layers. By convention, each layer has a name beginning with `meta`. The core layers of the Yocto Project are as follows:

- `meta`: This is the OpenEmbedded core with some changes for Poky
- `meta-poky`: This is the metadata specific to the Poky distribution
- `meta-yocto-bsp`: This contains the board support packages for the machines that the Yocto Project supports

The list of layers in which BitBake searches for recipes is stored in `<your build directory>/conf/bblayers.conf` and, by default, includes all three layers mentioned in the preceding list.

By structuring the recipes and other configuration data in this way, it is very easy to extend the Yocto Project by adding new layers. Additional layers are available from SoC manufacturers, the Yocto Project itself, and a wide range of people wishing to add value to the Yocto Project and OpenEmbedded. There is a useful list of layers at <http://layers.openembedded.org/layerindex/branch/master/layers/>. Here are some examples:

- `meta-angstrom`: The Ångström distribution
- `meta-qt5`: Qt 5 libraries and utilities
- `meta-intel`: BSPs for Intel CPUs and SoCs
- `meta-ti`: BSPs for TI ARM-based SoCs

Adding a layer is as simple as copying the `meta` directory into a suitable location, usually alongside the default meta layers and adding it to `bblayers.conf`. Make sure that you read the `README` file that should accompany each layer to see what dependencies it has on other layers and which versions of the Yocto Project it is compatible with.

To illustrate the way that layers work, let's create a layer for our Nova board, which we can use for the remainder of the chapter as we add features. You can see the complete implementation of the layer in the code archive in

MELP/chapter_06/poky/meta-nova.

Each meta layer has to have at least one configuration file, named `conf/layer.conf`, and it should also have the `README` file and a license. There is a handy helper script that does the basics for us:

```
$ cd poky
$ scripts/yocto-layer create nova
```

The script asks for a priority, and whether you want to create sample recipes. In the example here, I just accepted the defaults:

```
Please enter the layer priority you'd like to use for the layer:
[default: 6]
Would you like to have an example recipe created? (y/n) [default: n]
Would you like to have an example bbappend file created? (y/n)
[default: n]
New layer created in meta-nova.
Don't forget to add it to your BBLAYERS (for details see
meta-nova/README).
```

This will create a layer named `meta-nova` with a `conf/layer.conf`, an outline `README` and an MIT LICENSE in `COPYING.MIT`. The `layer.conf` file looks like this:

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/**/*.bb
${LAYERDIR}/recipes-*/**/*.bbappend"

BBFILE_COLLECTIONS += "nova"
BBFILE_PATTERN_nova = "^${LAYERDIR}/"
BBFILE_PRIORITY_nova = "6"
```

It adds itself to `BBPATH` and the recipes it contains to `BBFILES`. From looking at the code, you can see that the recipes are found in the directories with names beginning `recipes-` and have filenames ending in `.bb` (for normal BitBake recipes) or `.bbappend` (for recipes that extend existing recipes by overriding or adding to the instructions). This layer has the name `nova` added to the list of layers in `BBFILE_COLLECTIONS` and has a priority of 6. The layer priority is used if the same recipe appears in several layers: the one in the layer with the highest priority wins.

Since you are about to build a new configuration, it is best to begin by creating a new build directory named `build-nova`:


```
$ cd ~/poky
$ source oe-init-build-env build-nova
```

Now, you need to add this layer to your build configuration using the command:

```
$ bitbake-layers add-layer ../meta-nova
```

You can confirm that it is set up correctly like this:

```
$ bitbake-layers show-layers
layer                path                                priority
=====
meta                 /home/chris/poky/meta                5
meta-yocto           /home/chris/poky/meta-yocto          5
meta-yocto-bsp       /home/chris/poky/meta-poky-bsp       5
meta-nova            /home/chris/poky/meta-nova           6
```

There you can see the new layer. It has a priority 6, which means that we could override recipes in the other layers, which all have a lower priority.

At this point, it would be a good idea to run a build, using this empty layer. The final target will be the Nova board but, for now, build for a BeagleBone Black by removing the comment before `MACHINE ?= "beaglebone"` in `conf/local.conf`. Then, build a small image using `bitbake core-image-minimal` as before.

As well as recipes, layers may contain BitBake classes, configuration files for machines, distributions, and more. I will look at recipes next and show you how to create a customized image and how to create a package.

BitBake and recipes

BitBake processes metadata of several different types, which include the following:

- **Recipes:** Files ending in `.bb`. These contain information about building a unit of software, including how to get a copy of the source code, the dependencies on other components, and how to build and install it.
- **Append:** Files ending in `.bbappend`. These allow some details of a recipe to be overridden or extended. A `bbappend` file simply appends its instructions to the end of a recipe (`.bb`) file of the same root name.
- **Include:** Files ending in `.inc`. These contain information that is common to several recipes, allowing information to be shared among them. The files maybe included using the **include** or **require** keywords. The difference is that **require** produces an error if the file does not exist, whereas **include** does not.
- **Classes:** Files ending in `.bbclass`. These contain common build information, for example, how to build a kernel or how to build an autotools project. The classes are inherited and extended in recipes and other classes using the **inherit** keyword. The class `classes/base.bbclass` is implicitly inherited in every recipe.
- **Configuration:** Files ending in `.conf`. They define various configuration variables that govern the project's build process.

A recipe is a collection of tasks written in a combination of Python and shell script. The tasks have names such as `do_fetch`, `do_unpack`, `do_patch`, `do_configure`, `do_compile`, and `do_install`. You use BitBake to execute these tasks. The default task is `do_build`, which performs all the subtasks required to build the recipe. You can list the tasks available in a recipe using `bitbake -c listtasks [recipe]`. For example, you can list the tasks in `core-image-minimal` like this:

```
$ bitbake -c listtasks core-image-minimal
[...]
core-image-minimal-1.0-r0 do_listtasks: do_build
core-image-minimal-1.0-r0 do_listtasks: do_bundle_initramfs
core-image-minimal-1.0-r0 do_listtasks: do_checkuri
core-image-minimal-1.0-r0 do_listtasks: do_checkuriall
core-image-minimal-1.0-r0 do_listtasks: do_clean
[...]
```

In fact, `-c` is the option that tells BitBake to run a specific task in a recipe with the task being named with the `do_` part stripped off. The task `do_listtasks` is simply a special task that lists all the tasks defined within a recipe. Another example is the `fetch` task, which downloads the source code for a recipe:

```
| $ bitbake -c fetch busybox
```

You can also use the `fetchall` task to get the code for the target and all the dependencies, which is useful if you want to make sure you have downloaded all the code for the image you are about to build:

```
| $ bitbake -c fetchall core-image-minimal
```

The recipe files are usually named `<package-name>_<version>.bb`. They may have dependencies on other recipes, which would allow BitBake to work out all the subtasks that need to be executed to complete the top level job.

As an example, to create a recipe for our `helloworld` program in `meta-nova`, you would create a directory structure like this:

```
| meta-nova/recipes-local/helloworld
| └─ files
|   └─ helloworld.c
| └─ helloworld_1.0.bb
```

The recipe is `helloworld_1.0.bb` and the source is local to the recipe directory in the subdirectory `files/`. The recipe contains these instructions:

```
| DESCRIPTION = "A friendly program that prints Hello World!"
| PRIORITY = "optional"
| SECTION = "examples"
|
| LICENSE = "GPLv2"
| LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0;
| md5=801f80980d171dd6425610833a22dbe6"
|
| SRC_URI = "file://helloworld.c"
|
| S = "${WORKDIR}"
|
| do_compile() {
|     ${CC} ${CFLAGS} ${LDFLAGS} helloworld.c -o helloworld
| }
|
| do_install() {
|     install -d ${D}${bindir}
|     install -m 0755 helloworld ${D}${bindir}
| }
```

The location of the source code is set by `SRC_URI`. In this case, the `file://` URI means that the code is local to the `recipe` directory. BitBake will search `directories`, `files/`, `helloworld/`, and `helloworld-1.0/` relative to the directory that contains the recipe. The tasks that need to be defined are `do_compile` and `do_install`, which compile the one source file and install it into the target root filesystem: `${D}` expands to the staging area of the recipe and `${bindir}` to the default binary directory, `/usr/bin`.

Every recipe has a license, defined by **LICENSE**, which is set to **GPL V2** here. The file containing the text of the license and a checksum is defined by `LIC_FILES_CHKSUM`. BitBake will terminate the build if the checksum does not match, indicating that the license has changed in some way. The license file may be part of the package or it may point to one of the standard license texts in `meta/files/common-licenses/`, as is the case here.

By default, commercial licenses are disallowed, but it is easy to enable them. You need to specify the license in the recipe, as shown here:

```
| LICENSE_FLAGS = "commercial"
```

Then, in your `conf/local.conf`, you would explicitly allow this license, like so:

```
| LICENSE_FLAGS_WHITELIST = "commercial"
```

Now, to make sure that our `helloworld` recipe compiles correctly, you can ask BitBake to build it, like so:

```
| $ bitbake helloworld
```

If all goes well, you should see that it has created a working directory for it in `tmp/work/cortexa8hf-vfp-neon-poky-linux-gnueabi/helloworld/`. You should also see there is an RPM package for it in `tmp/deploy/rpm/cortexa8hf-vfp_neon/helloworld-1.0-r0.cortexa8hf-vfp_neon.rpm`.

It is not part of the target image yet, though. The list of packages to be installed is held in a variable named `IMAGE_INSTALL`. You can append to the end of that list by adding this line to `conf/local.conf`:

```
| IMAGE_INSTALL_append = " helloworld"
```

Note that there has to be a space between the opening double quote and the first

package name. Now, the package will be added to any image that you `bitbake`:

```
| $ bitbake core-image-minimal
```

If you look in `tmp/deploy/images/beaglebone/core-image-minimal-beaglebone.tar.bz2`, you will see that `/usr/bin/helloworld` has indeed been installed.

Customizing images via local.conf

You may often want to add a package to an image during development or tweak it in other ways. As shown previously, you can simply append to the list of packages to be installed by adding a statement like this:

```
| IMAGE_INSTALL_append = " strace helloworld"
```

You can make more sweeping changes via `EXTRA_IMAGE_FEATURES`. Here is a short list which should give you an idea of the features you can enable:

- `dbg-pkgs`: This installs debug symbol packages for all the packages installed in the image.
- `debug-tweaks`: This allows `root` logins without passwords and other changes that make development easier.
- `package-management`: This installs package management tools and preserves the package manager database.
- `read-only-rootfs`: This makes the root filesystem read-only. We will cover this in more detail in [Chapter 7, Creating a Storage Strategy](#).
- `x11`: This installs the X server.
- `x11-base`: This installs the X server with a minimal environment.
- `x11-sato`: This installs the OpenedHand Sato environment.

There are many more features that you can add in this way. I recommend you look at the **Image Features** section of the *Yocto Project Reference Manual* and also read through the code in `meta/classes/core-image.bbclass`.

Writing an image recipe

The problem with making changes to `local.conf` is that they are, well, local. If you want to create an image that is to be shared with other developers or to be loaded onto a production system, then you should put the changes into an **image recipe**.

An image recipe contains instructions about how to create the image files for a target, including the bootloader, the kernel, and the root filesystem images. By convention, image recipes are put into a directory named `images`, so you can get a list of all the `images` that are available by using this command:

```
| $ ls meta*/recipes*/images/*.bb
```

You will find that the recipe for `core-image-minimal` is in `meta/recipes-core/images/core-image-minimal.bb`.

A simple approach is to take an existing image recipe and modify it using statements similar to those you used in `local.conf`.

For example, imagine that you want an image that is the same as `core-image-minimal` but includes your `helloworld` program and the `strace` utility. You can do that with a two-line recipe file, which includes (using the `require` keyword) the base image and adds the packages you want. It is conventional to put the image in a directory named `images`, so add the recipe `nova-image.bb` with this content in `meta-nova/recipes-local/images`:

```
| require recipes-core/images/core-image-minimal.bb  
| IMAGE_INSTALL += "helloworld strace"
```

Now, you can remove the `IMAGE_INSTALL_append` line from your `local.conf` and build it using this:

```
| $ bitbake nova-image
```

Creating an SDK

It is very useful to be able to create a standalone toolchain that other developers can install, avoiding the need for everyone in the team to have a full installation of the Yocto Project. Ideally, you want the toolchain to include development libraries and header files for all the libraries installed on the target. You can do that for any image using the `populate_sdk` task, as shown here:

```
| $ bitbake -c populate_sdk nova-image
```

The result is a self-installing shell script in `tmp/deploy/sdk`:

```
| poky-<c_library>-<host_machine>-<target_image><target_machine>  
-toolchain-<version>.sh
```

For the SDK built with the `nova-image` recipe, it is this:

```
| poky-glibc-x86_64-nova-image-cortexa8hf-neon-toolchain-2.2.1.sh
```

If you only want a basic toolchain with just C and C++ cross compilers, the C-library and header files, you can instead run this:

```
| $ bitbake meta-toolchain
```

To install the SDK, just run the shell script. The default install directory is `/opt/poky`, but the install script allows you to change this:

```
| $ tmp/deploy/sdk/poky-glibc-x86_64-nova-image-cortexa8hf-neon-  
toolchain-2.2.1.sh  
Poky (Yocto Project Reference Distro) SDK installer version 2.2.1  
=====
```

```
Enter target directory for SDK (default: /opt/poky/2.2.1):  
You are about to install the SDK to "/opt/poky/2.2.1". Proceed[Y/n]?  
[sudo] password for chris:  
Extracting SDK.....done  
Setting it up...done
```

To make use of the toolchain, first source the environment and set up the script:

```
| $ source /opt/poky/2.2.1/environment-setup-cortexa8hf-neon-poky  
-linux-gnueabi
```



The `environment-setup-` script that sets things up for the SDK is not compatible with the `oe-init-build-env` script that you source when*

TIP

working in the Yocto Project build directory. It is a good rule to always start a new terminal session before you source either script.

The toolchain generated by Yocto Project does not have a valid `sysroot` directory:

```
$ arm-poky-linux-gnueabi-gcc -print-sysroot
/not/exist
```

Consequently, if you try to cross compile, as I have shown in previous chapters, it will fail like this:

```
$ arm-poky-linux-gnueabi-gcc helloworld.c -o helloworld
helloworld.c:1:19: fatal error: stdio.h: No such file or directory
#include <stdio.h>
               ^
compilation terminated.
```

This is because the compiler has been configured to work for a wide range of ARM processors, and the fine tuning is done when you launch it using the right set of flags. Instead, you should use the shell variables that are created when you source the environment-setup script for cross compiling. They include these:

- `CC`: The C compiler
- `CXX`: The C++ compiler
- `CPP`: The C preprocessor
- `AS`: The assembler
- `LD`: The linker

As an example, this is what we find that `cc` has been set to this:

```
$ echo $CC
arm-poky-linux-gnueabi-gcc -march=armv7-a -mfpu=neon
-mfloat-abi=hard -mcpu=cortex-a8 --sysroot=/opt/poky/
2.2.1/sysroots/cortexa8hf-neon-poky-linux-gnueabi
```

So long as you use `$CC` to compile, everything should work fine:

```
$ $CC helloworld.c -o helloworld
```

The license audit

The Yocto Project insists that each package has a license. A copy of the license is placed in `tmp/deploy/licenses/[package name]` for each package as it is built. In addition, a summary of the packages and licenses used in an image are put into the directory: `<image name>-<machine name>-<date stamp>/`. For `nova-image` we just built, the directory would be named something like this:

```
| tmp/deploy/licenses/nova-image-beaglebone-20170417192546/
```