OPEN SOURCE SOFTWARE DEVELOPMENT SERIES

Embedded Linux Systems with the Yocto Project™

Rudolf J. Streif

FREE SAMPLE CHAPTER

Embedded Linux Systems with the Yocto Project™

Embedded Linux Systems with the Yocto Project™

Rudolf J. Streif

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com

Cataloging-in-Publication Data is on file with the Library of Congress.

ISBN-13: 978-0-13-344324-0
ISBN-10: 0-13-344324-8
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, May 2016

To Janan, Dominic, Daniel, and Jonas

This page intentionally left blank

Contents

Foreword

The embedded Linux landscape is a little bit like the Old West: different outposts of technology scattered here and there, with barren and often dangerous landscape in between. If you're going to travel there, you need to be well stocked, be familiar with the territory, and have a reliable guide.

Just as people moved West during the Gold Rush in the mid-1800s, developers are moving into the embedded Linux world with the rush to the Internet of Things. As increased population brought law, order, and civilization to the Old West, important new open source software projects are bringing order to embedded Linux.

The Yocto Project is a significant order-bringer. Its tools let you focus on designing your project (what you want to build) and devote only the necessary minimum of your time and effort to putting it all together (how you build what you want to build).

This book is your reliable guide. In logically ordered chapters with clear and complete instructions, it will help you get your work done and your IoT project to market. And with some luck, you'll have fun along the way!

Enjoy your adventure!

Arnold Robbins
Series Editor

This page intentionally left blank

Preface

Smart home. Smart car. Smart phone. Smart TV. Smart thermostat. Smart lights. Smart watch. Smart washer. Smart dryer. Smart fridge. Smart basketball. Welcome to the brave new world of smart everything!

The proliferation of embedded computers in almost everything we touch and interact with in our daily lives has moved embedded systems engineering and embedded software development into the spotlight. Hidden from the direct sight of their users, embedded systems lack the attractiveness of web applications with their flashy user interfaces or the coolness of computer games with their animations and immersive graphics. It comes as no surprise that computer science students and software developers hardly ever think of embedded software engineering as their first career choice. However, the "smart-everything revolution" and the Internet of Things (IoT) are driving the demand for specialists who can bridge hardware and software worlds. Experts who speak the language of electric schematics as well as programming languages are sought after by employers.

Linux has become the first choice for an explosively growing number of embedded applications. There are good reasons for this choice, upon which we will elaborate in the coming chapters. Through my journey as an embedded software developer for various industries, I have learned Linux for embedded systems the hard way. There is no shortage of excellent development tools for virtually any programming language. The vast majority of libraries and applications for Linux can easily be built natively because of their tooling. Even building the Linux kernel from scratch is almost a breeze with the kernel's own build system. However, when it comes to putting it all together into a bootable system, the choices are scarce.

The Yocto Project closes that gap by providing a comprehensive set of integrated tools with the OpenEmbedded build system at its center. From source code to bootable system in a matter of a few hours—I wish I had that luxury when I started out with embedded Linux!

What This Book Is and What It Is Not

A build system that integrates many different steps necessary to create a fully functional Linux OS stack from scratch is rather complex. This book is dedicated to the build system itself and how you can effectively use it to build your own custom Linux distributions. This book is not a tutorial on embedded Linux. Although Chapter 6 explains the basics of the Linux system architecture (as this foundation is necessary to understanding how the build system assembles the many different components into an operational system), I do not go into the details of embedded Linux as such. If you are a beginning embedded Linux developer, I strongly recommend Christopher Hallinan's excellent *Embedded Linux Primer*, published in this same book series.

In this book, you will learn how the OpenEmbedded build system works, how you can write recipes to build your own software components, how to use and create Yocto Project board support packages to support different hardware platforms, and how to debug build failures. You will learn how to build software development kits for application development and integrate them with the popular Eclipse integrated development environment (IDE) for seamless round-trip development.

Who Should Read This Book

This book is intended for software developers and programmers who have a working knowledge of Linux. I assume that you know your way around the Linux command line, that you can build programs on a Linux system using the typical tools, such as Make and a C/C++ compiler, and that you can read and understand basic shell scripts.

The build system is written entirely in Python. While you do not need to be a Python expert to use it and to understand how it works, having some core knowledge about Python is certainly advantageous.

How This Book Is Organized

Chapter 1, "Linux for Embedded Systems," provides a brief look at the adoption of Linux for embedded systems. An overview of the embedded Linux landscape and the challenges of creating custom embedded Linux distributions set the stage.

Chapter 2, "The Yocto Project," introduces the Yocto Project by jumpstarting an initial build of a Linux OS stack using the build system. It also gives an overview of the Yocto Project family of projects and its history.

Chapter 3, "OpenEmbedded Build System," explains the fundamentals of the build system, its workflow, and its architecture.

Chapter 4, "BitBake Build Engine," gives insight into BitBake, the build engine at the core of the OpenEmbedded build system. It explains the metadata concept of recipes, classes, and configuration files and their syntax. A Hello World project in BitBake style illustrates the build workflow. Through the information provided, you gain the necessary knowledge for understanding provided recipes and for writing your own.

Chapter 5, "Troubleshooting," introduces tools and mechanisms available to troubleshoot build problems and provides practical advice on how to use the tools effectively.

Chapter 6, "Linux System Architecture," provides the basics of a Linux operating system stack and explains how the different components are layered. It discusses the concepts of kernel space and user space and how application programs interact with the Linux kernel through system calls provided by the standard C library.

Chapter 7, "Building a Custom Linux Distribution," details how to use the Yocto Project to create your own customized Linux distribution. It starts with an overview of the Linux distribution blueprints available with the build system and how to customize them. It then demonstrates how to create a Linux distribution entirely from scratch using the build system tools. After completing this chapter, you will know how to build your own operating system images.

Chapter 8, "Software Package Recipes," explains BitBake recipes and how to write them to build your own software packages with the build system. The chapter provides various real-world recipe examples that you can try.

Chapter 9, "Kernel Recipes," examines the details of building the Linux kernel with the OpenEmbedded build system. It explains how the build system tooling interacts with the kernel's own build environment to set kernel configuration and apply patches. A discussion of how the build system handles out-of-tree kernel modules and incorporates building device trees with the build process closes this chapter.

Chapter 10, "Board Support Packages," introduces how the build system supports building for different hardware—that is, CPU architectures and systems. After an explanation of the Yocto Project board support package concepts, the chapter details how you can build a project using a board support package. We then look into the internals of Yocto Project board support packages and explain how to create your own with a practical example that you can put to use with actual hardware. The chapter concludes with creating bootable media images for different hardware configurations.

Chapter 11, "Application Development," describes Yocto Project support for developing applications for Linux OS stacks created with the build system. It provides hands-on instructions on how to build application development toolkits (ADT) that include all the necessary tools for round-trip application development. Examples illustrate how to use an ADT for application development using the command-line tools as well as with the Eclipse IDE. Step-by-step instructions teach how to remotely run and debug applications on an actual hardware target.

Chapter 12, "Licensing and Compliance," discusses requirements for compliance with open source licenses and the tools the Yocto Project provides to facilitate meeting them.

Chapter 13, "Advanced Topics," introduces several tools that help you scale the Yocto Project to teams. *Toaster* is a web-based graphical user interface that can be used to create build systems that can be controlled remotely from a web browser. *Build history* is a tool that provides tracking and audit capabilities. With *source mirrors*, you can share source packages to avoid repeated downloads and to control source versions for product delivery. Last but not least, *Autobuilder* provides an out-of-the-box continuous build and integration framework for automating builds, quality assurance, and release processes. Equipped with the knowledge from this chapter, you can effectively set up team environments for the Yocto Project.

The appendices cover popular open source licenses and alphabetical references of build system metadata layers and machines.

Hands-on Experience

The book is written to provide you with hands-on experience using the Yocto Project. You will benefit the most if you follow along and try out the examples. The majority of them you can work through simply with an x86-based workstation running a recent Linux distribution (detailed requirements are provided in Chapter 2). For an even better experience, grab one of the popular development boards, such as the BeagleBone, the MinnowBoard Max, or the Wandboard. The BeagleBone makes an excellent low-cost experimental platform. The other two boards offer more performance and let you gain experience with multicore systems.

Analyze the code and try to understand the examples produced in the book. Follow the steps and then veer off on your own by changing settings, applying your own configuration, and more. It is the best way to learn, and I can tell you, it is a lot of fun too. It is a great feeling to get your first own Linux distribution to work on a piece of hardware of your choice.

Register your copy of *Embedded Linux Systems with the Yocto Project™* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780133443240) and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

Acknowledgments

What you are holding in your hands is my first attempt at writing a technical book. Well, any book, for that matter. I humbly have to admit that I greatly underestimated the effort that goes into a project like this, the hours spent experimenting with things, finding the best way to make them work, and documenting everything in a concise and understandable fashion. During the process, I have come to truly appreciate the work of the many authors and technical writers whose books and manuals I have read and continue reading.

Foremost, I want to express my gratitude to my family, my loving wife, Janan, and my three wonderful boys, Dominic, Daniel, and Jonas. Without their support and their understanding, it would not have been possible for me to spend the many hours writing this text.

Special thanks go to the Yocto Project team. When I approached Dave Stewart, Project Manager for the Yocto Project at the time, and Jeffrey Osier-Mixon, the Yocto Project's Community Manager, they immediately welcomed the idea for the book and offered their support. Several individuals from the team were especially helpful with advice and answers to my questions: Beth Flanagan for Autobuilder, Belen Barros Pena and Ed Bartosh for Toaster, and Paul Eggleton and Khem Raj who jumped on many of the questions I posted to the Yocto Project mailing list.

Special thanks to Christopher Hallinan whose *Embedded Linux Primer: A Practical Real-World Approach* (Prentice Hall, 2006) inspired me to write this book on the Yocto Project.

I especially want to thank Debra Williams Cauley, Executive Acquisitions Editor, for her guidance and particularly her patience while this book was in the works. It took much longer than expected, and I am the only one to blame for the missed deadlines.

I cannot thank and praise enough my dedicated review team, Chris Zahn, Jeffrey Osier-Mixon, Robert Berger, and Bryan Smith, for their valuable contributions to the quality of the book in the form of corrections and suggestions for improvements.

I also want to thank the production team at Prentice Hall, Julie Nahil and Anna Popick, for their coordination and guidance through the process, and in particular Carol Lallier for her diligence in copyediting the manuscript.

Thanks also to the Linux Foundation and Jerry Cooperstein, who gave me the opportunity to develop the Linux Foundation's training course on the Yocto Project. Nothing teaches as well as teaching somebody else. Thank you to the students of the classes that I taught. Through your critical questions and feedback, I gained a lot of understanding for the many different problems you are facing when developing products with embedded Linux. One of your most asked questions was, "Is there a book on the Yocto Project?" Finally, I can say, "Yes."

This page intentionally left blank

About the Author

Rudolf Streif has more than twenty years of experience in software engineering as a developer as well as a manager leading cross-functional engineering teams with more than one hundred members. Currently, he is an independent consultant for software technology and system architecture specializing in open source.

He previously served as the Linux Foundation's Director of Embedded Solutions, coordinating the Foundation's efforts for Linux in embedded systems. Rudolf developed the Linux Foundation's training course on the Yocto Project, which he delivered multiple times to companies and in a crash-course variant during Linux Foundation events.

Rudolf has been working with Linux and open source since the early 1990s and developing commercial products since 2000. The projects he has been involved with include high-speed industrial image processing systems, IPTV head-end system and customer premises equipment, and connected car and in-vehicle infotainment.

In 2014, Rudolf was listed by *PC World* among the 50 most interesting people in the world of technology (http://tinyurl.com/z3tbtns).

Rudolf lives with his wife and three children in San Diego, California.

This page intentionally left blank

7

Building a Custom Linux Distribution

In This Chapter

In the preceding chapters, we laid the foundation for using the Yocto Project tools to build custom Linux distributions. Now it is time that we put that knowledge to work.

Chapter 2, "The Yocto Project," outlined the prerequisites for the build system and how to set up your build host, configure a build environment, and launch a build that creates a system ready to run in the QEMU emulator. In this chapter, we reuse that build environment. If you have not yet prepared your build system, we recommend that you go back to Chapter 2 and follow the steps. Performing a build using Poky's default settings validates your setup. It also downloads the majority of the source code packages and establishes a shared state cache, both of which speed up build time for the examples presented in this chapter.

In Chapter 3, "OpenEmbedded Build System," and Chapter 4, "BitBake Build Engine," we explained the OpenEmbedded build system and the BitBake syntax. This and following chapters show examples or snippets of BitBake recipes utilizing that syntax. While the syntax is mostly straightforward and resembles typical scripting languages, there are some constructs that are particular to BitBake. Referring to Chapter 4, you find syntax examples and explanations.

When experimenting with the Yocto Project, you eventually encounter build failures. They can occur for various reasons, and troubleshooting can be challenging. You may want to refer to Chapter 5, "Troubleshooting," for the debugging tools to help you track down build failures.

Chapter 6, "Linux System Architecture," outlined the building blocks of a Linux distribution. While bootloader and the Linux kernel are indispensable for a working

Linux OS stack, user space makes up its majority. In this chapter, we focus on customizing Linux OS stacks with user space libraries and applications from recipes provided by the Yocto Project and other compatible layers from the OpenEmbedded project.

7.1 Core Images—Linux Distribution Blueprints

The OpenEmbedded Core (OE Core) and other Yocto Project layers include several example images. These images offer root filesystem configurations for typical Linux OS stacks. They range from very basic images that just boot a device to a command-line prompt to images that include the X Window System (X11) server and a graphical user interface. These reference images are called the *core images* because the names of their respective recipes begin with `core-image`. You can easily locate the recipes for the core images with the `find` command from within the installation directory of your build system (see Listing 7-1).

Listing 7-1 **Core Image Recipes**

```
user@buildhost:~/yocto/poky$ find ./meta*/recipes*/images -name "*.bb" \
                                  -print
./meta/recipes-core/images/core-image-minimal-initramfs.bb
./meta/recipes-core/images/core-image-minimal-mtdutils.bb
./meta/recipes-core/images/build-appliance-image_8.0.bb
./meta/recipes-core/images/core-image-minimal-dev.bb
./meta/recipes-core/images/core-image-minimal.bb
./meta/recipes-core/images/core-image-base.bb
./meta/recipes-extended/images/core-image-full-cmdline.bb
./meta/recipes-extended/images/core-image-testmaster-initramfs.bb
./meta/recipes-extended/images/core-image-lsb-sdk.bb
./meta/recipes-extended/images/core-image-lsb-dev.bb
./meta/recipes-extended/images/core-image-lsb.bb
./meta/recipes-extended/images/core-image-testmaster.bb
./meta/recipes-graphics/images/core-image-x11.bb
./meta/recipes-graphics/images/core-image-directfb.bb
./meta/recipes-graphics/images/core-image-weston.bb
./meta/recipes-graphics/images/core-image-clutter.bb
./meta/recipes-qt/images/qt4e-demo-image.bb
./meta/recipes-rt/images/core-image-rt-sdk.bb
./meta/recipes-rt/images/core-image-rt.bb
./meta/recipes-sato/images/core-image-sato-dev.bb
./meta/recipes-sato/images/core-image-sato-sdk.bb
./meta/recipes-sato/images/core-image-sato.bb
./meta-skeleton/recipes-multilib/images/core-image-multilib-example.bb
```

You can look at the core images as Linux distribution blueprints from which you can derive your own distribution by extending them. All core image recipes inherit the `core-image` class, which itself inherits from `image` class. All images set the `IMAGE_INSTALL` variable to specify what packages are to be installed into the root filesystem. `IMAGE_INSTALL` is a list of packages and package groups. Package groups are collections of packages. Defining package groups alleviates the need to potentially list hundreds of single packages in the `IMAGE_INSTALL` variable. We explain package groups in a coming section of this chapter. Image recipes either explicitly set `Image_INSTALL` or extend its default value provided by the `core-image` class, which installs the two package groups `packagegroup-core-boot` and `packagegroup-base-extended`. The default creates a working root filesystem that boots to the console.

Let's have a closer look at the various core images:

- **`core-image-minimal`**: This is the most basic image allowing a device to boot to a Linux command-line login. Login and command-line interpreter are provided by BusyBox.
- **`core-image-minimal-initramfs`**: This image is essentially the same as `core-image-minimal` but with a Linux kernel that includes a RAM-based initial root filesystem (initramfs).
- **`core-image-minimal-mtdutils`**: Based on `core-image-minimal`, this image also includes user space tools to interact with the memory technology device (MTD) subsystem in the Linux kernel to perform operations on flash memory devices.
- **`core-image-minimal-dev`**: Based on `core-image-minimal`, this image also includes all the development packages (header files, etc.) for all the packages installed in the root filesystem. If deployed on the target together with a native target toolchain, it allows software development on the target. Together with a cross-toolchain, it can be used for software development on the development host.
- **`core-image-rt`**: Based on `core-image-minimal`, this image target builds the Yocto Project real-time kernel and includes a test suite and tools for real-time applications.
- **`core-image-rt-sdk`**: In addition to `core-image-rt`, this image includes the system development kit (SDK) consisting of the development packages for all packages installed; development tools such as compilers, assemblers, and linkers; as well as performance test tools and Linux kernel development packages. This image allows for software development on the target.
- **`core-image-base`**: Essentially a `core-image-minimal`, this image also includes middleware and application packages to support a variety of hardware such as WiFi, Bluetooth, sound, and serial ports. The target device must include the necessary hardware components, and the Linux kernel must provide the device drivers for them.
- **`core-image-full-cmdline`**: This minimal image adds typical Linux command-line tools—bash, acl, attr, grep, sed, tar, and many more—to the root filesystem.
- **`core-image-lsb`**: This image contains packages required for conformance with the Linux Standard Base (LSB) specification.
- **`core-image-lsb-dev`**: This image is the same as the `core-image-lsb` but also includes the development packages for all packages installed in the root filesystem.

- **`core-image-lsb-sdk`**: In addition to `core-image-lsb-dev`, this image includes development tools such as compilers, assemblers, and linkers as well as performance test tools and Linux kernel development packages.
- **`core-image-x11`**: This basic graphical image includes the X11 server and an X11 terminal application.
- **`core-image-sato`**: This image provides X11 support that includes the OpenedHand Sato user experience for mobile devices. Besides the Sato screen manager, the image also provides several applications using the Sato theme, such as a terminal, editor, file manager, and several games.
- **`core-image-sato-dev`**: This image is the same as `core-image-sato` but also includes the development packages for all packages installed in the root filesystem.
- **`core-image-sato-sdk`**: In addition to `core-image-sato-dev`, this image includes development tools such as compilers, assemblers, and linkers as well as performance test tools and Linux kernel development packages.
- **`core-image-directfb`**: An image that uses DirectFB for graphics and input device management, DirectFB may include graphics acceleration and a windowing system. Because of its much smaller footprint compared to X11, DirectFB is the preferred choice for lower-end embedded systems that need graphics support but not the entire functionality of X11.
- **`core-image-clutter`**: This is an X11-based image that also includes the Clutter toolkit. Clutter is based on OpenGL and provides functionality for animated graphical user interfaces.
- **`core-image-weston`**: This image uses Weston instead of X11. Weston is a compositor that uses the Wayland protocol and implementation to exchange data with its clients. This image also includes a Wayland-capable terminal program.
- **`qt4e-demo-image`**: This image launches a demo application for the embedded Qt toolkit after completing the boot process. Qt for embedded Linux provides a development framework of graphical applications that directly write to the framebuffer instead of using the X11.
- **`core-image-multilib-example`**: This image is an example of the support of multiple libraries, typically 32-bit support on an otherwise 64-bit system. The image is based on a core image and adds the desired multilib packages to `IMAGE_INSTALL`.

The following three images are not reference images for embedded Linux systems. We include them in this discussion for completeness purposes.

- **`core-image-testmaster, core-image-testmaster-initramfs`**: These images are references for testing other images on actual hardware devices or in QEMU. They are deployed to a separate partition to boot into and then use scripts to deploy the image under test. This approach is useful for automated testing.

- **`build-appliance-image`**: This recipe creates the Yocto Project Build Appliance virtual machine images that include everything needed for the Yocto Project build system. The images can be launched using VMware Player or VMware Workstation.

Studying the reference image recipes is a good way to learn how these images are built and what packages comprise them. The core images are also a good starting point for your own Linux OS stack. You can easily extend them by adding packages and package groups to `IMAGE_INSTALL`. Images can only be extended, not shrunk. To build an image with less functionality, you have to start from a smaller core image and add only the packages you need. There is no simple way to remove packages. The majority of them are added through package groups, and you would need to split up the package group if you do not want to install a package included with it. Of course, if you are removing a package, you also have to remove any other packages that depend on it.

There are several ways you can add packages and package groups to be included with your root filesystem. The following sections explain them and also provide information on why you would want to use one method over another.

7.1.1 Extending a Core Image through Local Configuration

The simplest method for adding packages and package groups to images is to add `IMAGE_INSTALL` to the `conf/local.conf` file of your build environment:

```
IMAGE_INSTALL_append = " <package> <package group>"
```

As we have seen, image recipes set the `IMAGE_INSTALL` variable adding packages and package groups. To extend an image, you have to append your packages and packages group to the variable. You may wonder why we use the explicit `_append` operator instead of the `+=` or `.+` operators. Using the `_append` operator unconditionally appends the specified value to the `IMAGE_INSTALL` variable after all recipes and configuration files have been processed. Image recipes commonly explicitly set the `IMAGE_INSTALL` variable using the `=` or `?=` operators, which may happen *after* BitBake processed the settings in `conf/local.conf`.

For example, adding

```
IMAGE_INSTALL_append = " strace sudo sqlite3"
```

installs the strace and sudo tools as well as SQLite in the root filesystem. When using the `_append` operator, you always have to remember to add a space in front of the first package or package group, as this operator does not automatically include a space.

Using `IMAGE_INSTALL` in the `conf/local.conf` of your build environment unconditionally affects all images you are going to build with this build environment. If you are looking to install additional packages only to a certain image, you can use conditional appending:

```
IMAGE_INSTALL_append_pn-<image> = " <package> <package group>"
```

This installs the specified packages and package groups only into the root filesystem of `image`. For example,

```
IMAGE_INSTALL_append_pn-core-image-minimal = " strace"
```

installs the strace tool only into the root filesystem of `core-image-minimal`. All other images are unaffected.

Using `IMAGE_INSTALL` also affects core images, that is, images that inherit from the `core-image` class, as well as images that inherit directly from the `image` class. For convenience purposes, the `core-image` class defines the variable `CORE_IMAGE_EXTRA_INSTALL`. All packages and package groups added to this variable are appended to `IMAGE_INSTALL` by the class. Using

```
CORE_IMAGE_EXTRA_INSTALL = "strace sudo sqlite3"
```

adds these packages to all images that inherit from `core-image`. Images that inherit directly from image are not affected. Using `CORE_IMAGE_EXTRA_INSTALL` is a safer and easier method for core images than appending directly to `IMAGE_INSTALL`.

7.1.2 Testing Your Image with QEMU

You can easily test your image with the QEMU emulator. Even though you eventually build a system for the target hardware of your product, using QEMU for testing makes good sense for the following reasons:

- The round-trip time for launching QEMU is much quicker than deploying an image to actual hardware.
- Frequently, hardware is not yet available when software development begins.
- Yocto Project board support packages (BSP) make it simple to switch from QEMU to hardware and back.

In Chapter 2, when performing our first build, we used QEMU to verify the build output. The Poky reference distribution provides the script `runqemu` that greatly simplifies the task of launching QEMU by providing the necessary parameters. In its simplest form, you launch the script with a single parameter

```
$ runqemu qemux86
```

which tells the script to locate the latest kernel and root filesystem image builds for the provided QEMU machine and otherwise launch QEMU with default parameters. The parameter values match the QEMU machine types in `conf/local.conf`.

When working with different root filesystem images, you probably want to select the particular image when running QEMU. For example, you have built `core-image-minimal` and `core-image-base` using the preceding command line, since `runqemu` launches whatever image you last built. Using the command as follows lets you choose the image:

```
$ runqemu qemux86 core-image-minimal
```

The script automatically selects the correct kernel and uses the latest core-image-minimal root filesystem. For even more control, you can directly specify the kernel image and root filesystem image file:

```
$ runqemu <path>/bzImage-qemux86.bin <path>/core-image-minimal-qemux86.ext3
```

QEMU and the runqemu script are handy tools for rapid round-trip application development, which we explore in Chapter 11, "Application Development."

7.1.3 Verifying and Comparing Images Using the Build History

When building a product, you find yourself frequently modifying your images, adding new packages, and removing extraneous packages to trim the footprint. A tool that enables you to easily verify and compare image builds with each other can simplify that otherwise tedious task.

To help maintain build output quality and enable comparison between different builds, BitBake provides build history, which is implemented by the buildhistory class. This class records information about the contents of all packages built and about the images created by the build system in a Git repository where you can examine them. Build history is disabled by default. To enable it, you need to add

```
INHERIT += "buildhistory"
BUILDHISTORY_COMMIT = "1"
```

to the conf/local.conf file of your build environment. Please note that INHERIT (uppercase) is a variable to which you have to add the buildhistory class. It is different from the inherit (lowercase) directive used by recipes and classes to inherit functionality from classes. Every time you do a build, buildhistory creates a commit to the Git repository with the changes.

The buildhistory Git repository is stored in a directory as defined by the BUILDHISTORY_DIR variable. The default value of this variable is set to

```
BUILDHISTORY_DIR ?= "${TOPDIR}/buildhistory"
```

After enabling buildhistory and running a build, you see a buildhistory directory added to the top-level directory of your build environment. The directory contains the two subdirectories images and packages. The former contains build information about the images you build, the latter information on the packages. We analyze the buildhistory Git repository in Chapter 13, "Advanced Topics." Here we just look at the images subdirectory. Inside the images subdirectory, the images are sorted into further subdirectories by target machine, target C library, and image name:

```
${TOPDIR}/buildhistory/images/<machine>/<clib>/<image>
```

For the build of our core-image-minimal for qemux86 using the default EGLIBC target library, you find the image history in

```
${TOPDIR}/buildhistory/images/qemux86/eglibc/core-image-mininal
```

The files in that directory give you detailed information on what makes up your image:

- **`image-info.txt`**: Overview information about the image in form of the most important variables, such as `DISTRO`, `DISTRO_VERSION`, and `IMAGE_INSTALL`
- **`installed-packages.txt`**: A list of the package files installed in the image, including version and target information
- **`installed-package-names.txt`**: Similar to the previous file but contains only the names of the packages without version and target information
- **`files-in-image.txt`**: A list of the root filesystem with directory names, file sizes, file permissions, and file owner

Simply searching the file `installed-package-names.txt` gives you information on whether or not a package has been installed.

7.1.4 Extending a Core Image with a Recipe

Adding packages and package groups to `CORE_IMAGE_EXTRA_INSTALL` and `IMAGE_INSTALL` and in `conf/local.conf` may be straightforward and quick, but doing so makes a project hard to maintain and complicates reuse. A better way is to extend a predefined image through a recipe. Listing 7-2 shows a simple recipe that extends `core-image-base`.

Listing 7-2 **Recipe Extending `core-image-base`**

```
DESCRIPTION = "A console image with hardware support for our IoT device"

require recipes-core/images/core-image-base.bb

IMAGE_INSTALL += "sqlite3 mtd-utils coreutils"
IMAGE_FEATURES = "dev-pkgs"
```

The example includes the recipe for `core-image-base` and adds packages to `IMAGE_INSTALL` and an image feature to `IMAGE_FEATURES`. We explain what image features are and how to utilize them to customize image in the next section.

A couple of things to consider when extending images with recipes:

- Unlike classes, you need to provide the path relative to the layer for BitBake to find the recipe file to include, and you need to add the `.bb` file extension.
- While you can use either `include` or `require` to include the recipe you are extending, we recommend the use of `require`, since it causes BitBake to exit with an explicit error message if it cannot locate the included recipe file.
- Remember to use the `+=` operator to add to `IMAGE_INSTALL`. Do not use `=` or `:=` because they overwrite the content of the variable defined by the included recipe.

For BitBake to actually be able to use this recipe as a build target, you have to add it to a layer that is included into your build environment via the `conf/bblayers.conf` file. It is not recommended that you add your recipes to the core Yocto Project layers, such as `meta`, `meta-yocto`, and others, because it makes it hard to maintain your build environment if you upgrade to a newer version of the Yocto Project. Instead, you should create a layer in which to put your recipes.

Creating a layer for one recipe may seem like a lot of overhead, but hardly any project ever stays small. What may start with one recipe eventually grows into a sophisticated project with recipes for images, packages, and package groups. In Chapter 3, we introduced the `yocto-layer`, which makes creating layers a breeze.

7.1.5 Image Features

Image features provide certain functionality that you can add to your target images. This can be additional packages to be installed, modification of configuration files, and more.

For example, the `dev-pkgs` image feature adds the development packages, which typically include headers and other files required for development, for all packages installed in the root filesystem. Using this image feature is a convenient way to enable a target image for development without having to explicitly specify the development packages in the `IMAGE_INSTALL` variable. For deployment, you can then simply remove the `dev-pkgs` image feature.

Installation of image features is controlled by the two variables `IMAGE_FEATURES` and `EXTRA_IMAGE_FEATURES`. The former is used in image recipes to define the required set of image features. The latter is typically used in the `conf/local.conf` file to define additional image features that, of course, then affect all images built with that build environment. The content of `EXTRA_IMAGE_FEATURES` is simply added to `IMAGE_FEATURES` by the `meta/conf/bitbake.conf` configuration file.

Image features are defined by different classes. The list of currently available image features contains the following:

- Defined by `image.bbclass`:
  - **`debug-tweaks`**: Prepares an image for development purposes. In particular, it sets empty root passwords for console and Secure Shell (SSH) login.
  - **`package-management`**: Installs the package management system according to the package management class defined by `PACKAGE_CLASSES` for the root filesystem.
  - **`read-only-rootfs`**: Creates a read-only root filesystem. This image feature works only if System V Init (SysVinit) system is used rather than `sytemd`.
  - **`splash`**: Enables showing a splash screen instead of the boot messages during boot. By default, the splash screen is provided by the `psplash` package, which can be customized. You can also define an alternative splash screen package by setting the `SPLASH` variable to a different package name.

- Defined by `populate_sdk_base.bbclass`:
  - **`dbg-pkgs`**: Installs the debug packages containing symbols for all packages installed in the root filesystem.
  - **`dev-pgks`**: Installs the development packages containing headers and other development files for all packages installed in the root filesystem.
  - **`doc-pkgs`**: Installs the documentation packages for all packages installed in the root filesystem.
  - **`staticdev-pkgs`**: Installs the static development packages such as static library files ending in `*.a` for all packages installed in the root filesystem.
  - **`ptest-pkgs`**: Installs the package test (ptest) packages for all packages installed in the root filesystem.
- Defined by `core-image.bbclass`:
  - **`eclipse-debug`**: Installs remote debugging tools for integration with the Eclipse IDE, namely the GDB debugging server, the Eclipse Target Communication Framework (TCF) agent, and the OpenSSH SFTP server.
  - **`hwcodecs`**: Installs the hardware decoders and encoders for audio, images, and video if the hardware platform provides them.
  - **`nfs-server`**: Installs Network File System (NFS) server, utilities, and client.
  - **`qt4-pkgs`**: Installs the Qt4 framework and demo applications.
  - **`ssh-server-dropbear`**: Installs the lightweight SSH server Dropbear, which is popular for embedded systems. This image feature is incompatible with `ssh-server-openssh`. Either one of the two, but not both, can be used.
  - **`ssh-server-openssh`**: Installs the OpenSSH server. This image feature is incompatible with `ssh-server-dropbear`. Either one of the two, but not both, can be used.
  - **`tools-debug`**: Installs debugging tools, namely the GDB debugger, the GDB remote debugging server, the system call tracing tool strace, and the memory tracing tool mtrace for the GLIBC library if it is the target library.
  - **`tools-profile`**: Installs common profiling tools such as `oprofile`, `powertop`, `latencytop`, `lttng-ust`, and `valgrind`.
  - **`tools-sdk`**: Installs software development tools such as the GCC compiler, Make, autoconf, automake, libtool, and many more.
  - **`tools-testapps`**: Installs test applications such as tests for X11 and middleware packages like the telephony manager oFono and the connection manager ConnMan.
  - **`x11`**: Installs the X11 server.
  - **`x11-base`**: Installs the X11 server with windowing system.
  - **`x11-sato`**: Installs the OpenedHand Sato user experience for mobile devices.

It matters what classes define the image features when creating your own image recipes and choosing the image class to inherit. The class `image` inherits `populate_sdk_base` and thus all image features defined by those two classes are available to images that inherit `image`. Image features defined by `core-image` are available only to images that inherit that class, which in turn inherits `image` and with it also `populate_sdk_base`.

7.1.6 Package Groups

We have touched on package groups a couple of times during this discussion of creating custom Linux distribution images. Package groups are bundles of packages that are referenced by a name. Using that name in the `IMAGE_INSTALL` variable installs all the packages defined by the package group into the root filesystem of your target image.

The Yocto Project and OE Core layers define a common set of package groups that you can readily use for your images. You can also create your own package groups containing packages from any layer, including your own. We first describe the package groups defined by the Yocto Project and OE Core layers and then look into the details on how package groups are defined.

Predefined Package Groups

Package groups are defined by recipes. Conventionally, the recipe files begin with `packagegroup-` and are placed inside `packagegroup` subdirectories of the respective recipe categories. For instance, you can find package group recipes related to the Qt development framework in the subdirectory `meta/recipes-qt/packagegroups`.

Using

```
find . -name "packagegroup-*" -print
```

from the installation directory of the Yocto Project build system gives you a list of all the package group recipes for the predefined package groups of the Yocto Project build system.

Following are the most common predefined package groups:

- **`packagegroup-core-ssh-dropbear`**: Provides packages for the Dropbear SSH server popular for embedded systems because of its smaller footprint compared to the OpenSSH server. This package group conflicts with `packagegroup-core-ssh-openssh`. You can include only one of the two in your image. The `ssh-server-dropbear` image feature installs this package group.
- **`packagegroup-core-ssh-openssh`**: Provides packages for the standard OpenSSH server. This package group conflicts with `packagegroup-core-ssh-dropbear`. You can include only one of the two in your image. The `ssh-server-openssh` image feature installs this package group.
- **`packagegroup-core-buildessential`**: Provides the essential development tools, namely the GNU Autotools utilities autoconf, automake, and libtool; the GNU binary tool set binutils which includes the linker ld, assembler as, and other tools;

the compiler collection cpp; gcc; g++; the GNU internationalization and localization tool gettext; make; libstc++ with development packages; and pkgconfig.

- **`packagegroup-core-tools-debug`**: Provides the essential debugging tools, namely the GDB debugger, the GDB remote debugging server, the system call tracing tool strace, and, for the GLIBC target library, the memory tracing tool mtrace.
- **`packagegroup-core-sdk`**: This package group combines the `packagegroup-core-buildessential` package group with additional tools for development such as GNU Core Utilities coreutils with shell, file, and text manipulation utilities; dynamic linker ldd; and others. Together with `packagegroup-core-standalone-sdk-target`, this package group forms the `tools-sdk` image feature.
- **`packagegroup-core-standalone-sdk-target`**: Provides the GCC and standard C++ libraries. Together with `packagegroup-core-sdk`, this package group forms the `tools-sdk` image feature.
- **`packagegroup-core-eclipse-debug`**: Provides the GDB debugging server, the Eclipse TCF agent, and the OpenSSH SFTP server for integration with the Eclipse IDE for remote deployment and debugging. The image feature `eclipse-debug` installs this package group.
- **`packagegroup-core-tools-testapps`**: Provides test applications such as tests for X11 and middleware packages like the telephony manager oFono and the connection manager ConnMan. The `tools-testapps` image feature installs this package group.
- **`packagegroup-self-hosted`**: Provides all necessary packages for a self-hosted build system. The `build-appliance` image target uses this package group.
- **`packagegroup-core-boot`**: Provides the minimum set of packages necessary to create a bootable image with console. All `core-image` targets install this package group. The `core-image-minimal` installs just this package group and the postinstallation scripts.
- **`packagegroup-core-nfs`**: Provides NFS server, utilities, and client. The `nfs-server` image feature installs this package group.
- **`packagegroup-base`**: This recipe provides multiple package groups that depend on each other as well as on machine and distribution configuration. The purpose of these package groups is to add hardware, networking protocol, USB, filesystem, and other support to the images dependent on the machine and distribution configuration. The two top-level package groups are `packagegroup-base` and `packagegroup-base-extended`. The former adds hardware support for Bluetooth, WiFi, 3G, and NFC only if both the machine configuration and the distribution configuration require them. The latter also adds configuration for those technologies if the distribution configuration requires them. However, the machine configuration does not support them directly but provides support for PCI, PCMCIA, or USB host. This package group allows you to create an image with support for devices that can physically be added to the target device; for example, via USB hotplug. Most commonly, images providing hardware support use

`packagegroup-base-extended` rather than `packagegroup-base` for dynamic hardware support; for example, `core-image-base`.

- **`packagegroup-cross-canadian`**: Provides SDK packages for creating a toolchain using the Canadian Cross technique, which is building a toolchain on system A that executes on system B to create binaries for system C. A use case for this package group is to build a toolchain with the Yocto Project on your build host that runs on your image target but produces output for a third system with a different architecture than your image target.
- **`packagegroup-core-tools-profile`**: Provides common profiling tools such as oProfile, PowerTOP, LatencyTOP, LTTng-UST, and Valgrind. The `tools-profile` image feature uses this package group.
- **`packagegroup-core-device-devel`**: Provides distcc support for an image. Distcc allows distribution of compilation across several machines on a network. The distcc must be installed, configured, and running on your build host. On the target you must define the cross-compiler variable to use distcc instead of the local compiler (e.g., `export CC="distcc"`).
- **`packagegroup-qt-toolchain-target`**: Provides the package to build applications for the X11-based version of the Qt development toolkit on the target system.
- **`packagegroup-qte-toolchain-target`**: Provides the package to build applications for the embedded version of the Qt development toolkit on the target system.
- **`packagegroup-core-qt`**: Provides all necessary packages for a target system using the X11-based version of the Qt development toolkit.
- **`packagegroup-core-qt4e`**: Provides all necessary packages for a target system using the embedded Qt toolkit. The `qt4e-demo-image` installs this package group.
- **`packagegroup-core-x11-xserver`**: Provides the X.Org X11 server only.
- **`packagegroup-core-x11`**: Provides `packagegroup-core-x11-xserver` plus basic utilities such as xhost, xauth, xset, xrandr, and initialization on startup. The x11 image feature installs this package group.
- **`packagegroup-core-x11-base`**: Provides `packagegroup-core-x11` plus middleware and application clients for a working X11 environment that includes the Matchbox Window Manager, Matchbox Terminal, and a fonts package. The `x11-base` image feature installs this package group.
- **`packagegroup-core-x11-sato`**: Provides the OpenedHand Sato user experience for mobile devices, which includes the Matchbox Window Manager, Matchbox Desktop, and a variety of applications. The `x11-sato` image feature installs this package group. To utilize this package group for your target image, you also have to install `packagegroup-core-x11-base`.
- **`packagegroup-core-clutter-core`**: Provides packages for the Clutter graphical toolkit. To use the toolkit for your target image, you also have to install `packagegroup-core-x11-base`.

- **packagegroup-core-directfb**: Provides packages for the DirectFB support without X11. Among others, the package group includes the `directfb` package and the `directfb-example` package, and it adds touchscreen support if provided by the machine configuration.
- **packagegroup-core-lsb**: Provides all packages required for LSB support.
- **packagegroup-core-full-cmdline**: Provides packages for a more traditional Linux system by installing the full command-line utilities rather than the more compact BusyBox variant.

When explaining the different package groups, we used the terms *provide* and *install* somewhat liberally, since the package group recipes actually do not provide or install any packages. They only create dependencies that cause the build system to process the respective package recipes, as we see in the next section.

Several of the package groups are used by image features, which raises the question whether to use an image feature or to use the package group the image feature uses.

Package Group Recipes

Package groups are defined by recipes that inherit the `packagegroup` class. Package group recipes are different from typical package recipes, as they do not build anything or create any output. Package group recipes only create dependencies that trigger the build system to process the recipes of the packages the package groups reference.

Listing 7-3 shows a typical package group recipe.

Listing 7-3 **Package Group Recipe**

```
SUMMARY = "Custom package group for our IoT devices"
DESCRIPTION = "This package group adds standard functionality required by \
               our IoT devices."

LICENSE = "MIT"

inherit packagegroup
PACKAGES = "\
   packagegroup-databases \
   packagegroup-python \
   packagegroup-servers"

RDEPENDS_packagegroup-databases = "\
   db \
   sqlite3"

RDEPENDS_packagegroup-python = "\
   python \
   python-sqlite3"

RDEPENDS_packagegroup-servers = "\
   openssh \
   openssh-sftp-server"
```

```
RRECOMMENDS_packagegroup-python = "\
   ncurses \
   readline \
   zip"
```

Names of package group recipes, although not enforced or required by the build system, should adhere to the convention `packagegroup-<name>.bb`. You also would want to place them in the subdirectory `packagegroup` of the recipe category the package groups are integrating. If package groups span recipes and possibly package groups from multiple categories, it is good practice to place them into the `recipes-core` category.

The basic structure of package group recipes is rather simple. As should any recipe (and we go into the details of writing recipes in Chapter 8, "Software Package Recipes"), a package group recipe should provide a `SUMMARY` of what the recipe does. The `DESCRIPTION`, which can provide a longer, more detailed explanation, is optional, but it is good practice to add it. Any recipe also needs to provide a `LICENSE` for the recipe itself. All package group recipes must inherit the `packagegroup` class.

The names of the actual package groups are defined by the `PACKAGES` variable. This variable contains a space-delimited list of the package group names. In the case of Listing 7-3, these are `packagegroup-databases`, `packagegroup-python`, and `packagegroup-servers`. By convention, package group names begin with `packagegroup-`. Although the build system does not require it, it is good practice if you adhere to it for your own package group names.

For each package group, the recipe must define its dependencies in a conditional `RDEPENDS_<package-group-name>` variable. These variables list the required dependencies, which can be packages or package groups.

The `RRECOMMENDS_<package-group-name>` definitions are optional. As we saw in Chapter 3, recommendations are weak dependencies that cause a package to be included only if it already has been built.

You can reference package groups from other variables, such as `IMAGE_INSTALL`, which of course causes these package groups to be installed in a target image. You can also use them to create dependencies for other package groups for a hierarchy. You must avoid circular dependencies of package groups. That may sound simple and straightforward but can easily happen by mistake in rather complex environments. BitBake, however, aborts with an error message in the case of a circular package group dependency.

Package group recipes can also be directly used as BitBake build targets. For example, if the name of the package group recipe is `packagegroup-core-iot.bb`, you can build all the packages of the package groups defined by the recipe using

```
$ bitbake packagegroup-core-iot
```

Doing so allows testing the package groups before referencing them by image builds, which simplifies debugging.

7.2 Building Images from Scratch

Section 7.1 detailed the Yocto Project core images and how to extend them through setting IMAGE_INSTALL, CORE_IMAGE_EXTRA_INSTALL, IMAGE_FEATURES, and EXTRA_IMAGE_FEATURES in conf/local.conf and in recipes extending predefined image recipes. Eventually, you may want to create your custom Linux distribution image from scratch without relying on one of the reference images.

A custom image recipe must inherit either the image or the core-image class. The latter is essentially an extension of the former and defines additional image features, as described earlier in Section 7.1.5. Which one to choose for custom image recipes depends on your requirements. However, inheriting core-image generally is sound advice, since the image features are made available but only installed if explicitly requested.

Listing 7-4 shows the simplest image recipe that creates a bootable console image.

Listing 7-4 **Basic Image Recipe**

```
SUMMARY = "Custom image recipe that does not get any simpler"
DESCRIPTION = "Well yes, you could remove SUMMARY, DESCRIPTION, LICENSE."

LICENSE = "MIT"

inherit core-image
```

The recipe creates an image with the core packages to boot and hardware support for the target device because the core-image class adds the two package groups packagegroup-core-boot and packagegroup-base-extended to IMAGE_INSTALL by default. Also added to IMAGE_INSTALL by the class is the variable CORE_IMAGE_EXTRA_INSTALL, which allows for simple image modification through conf/local.conf, as described earlier.

The basic image with package-group-core-boot and package-base-extended provides a good starting point that easily can be extended by adding to IMAGE_INSTALL and IMAGE_FEATURES, as shown in Listing 7-5.

Listing 7-5 **Adding to the Basic Image**

```
SUMMARY = "Custom image recipe adding packages and features"
DESCRIPTION = "Append to IMAGE_INSTALL and IMAGE_FEATURES for \
               further customization. "

LICENSE = "MIT"

# We are using the append operator (+=) below to preserve the default
# values set by the core-image class we are inheriting.
IMAGE_INSTALL += "mtd-utils"
IMAGE_FEATURES += "splash"

inherit core-image
```

Within image recipes, you append directly to IMAGE_INSTALL and IMAGE_FEATURES using the += operator. Do not use EXTRA_IMAGE_FEATURES or CORE_IMAGE_EXTRA_INSTALL in your image recipe. These variables are reserved for use in conf/local.conf where they are directly assigned and overwrite any values assigned by the image recipe.

An image recipe that does not rely on the default values for IMAGE_INSTALL and IMAGE_FEATURES is equally simple, as Listing 7-6 shows.

Listing 7-6 **Core Image from Scratch**

```
SUMMARY = "Custom image recipe from scratch"
DESCRIPTION = "Directly assign IMAGE_INSTALL and IMAGE_FEATURES for \
               for direct control over image contents."

LICENSE = "MIT"

# We are using the assignment operator (=) below to purposely overwrite
# the default from the core-image class.
IMAGE_INSTALL = "packagegroup-core-boot packagegroup-base-extended \
                 ${CORE_IMAGE_EXTRA_INSTALL} mtd-utils"
IMAGE_FEATURES = "${EXTRA_IMAGE_FEATURES} splash"

inherit core-image
```

At first glance, the image recipes of Listings 7-5 and 7-6 look rather similar. In fact, the two recipes produce exactly the same image. The differences are subtle but significant. Listing 7-5 uses the append operator += for IMAGE_INSTALL and IMAGE_FEATURES to take advantage of the default values provided by the core-image class. Listing 7-6 uses the assignment operator = to purposely overwrite the default values.

Overwriting the default values gives you the most control over the content of your image, but you also have to take care of the basics yourself. For any image, you would most likely always want to include packagegroup-core-boot to get a bootable image. Whether you want the hardware support that packagegroup-base-extended provides depends on your requirements. Also at your disposal is CORE_IMAGE_EXTRA_INSTALL: if you do not explicitly add it to IMAGE_FEATURES, you will not be able to use this variable in conf/local.conf for local customization of your target image, but it may make sense to do so for a controlled build environment for production.

The same holds true for IMAGE_FEATURES and EXTRA_IMAGE_FEATURES. If you use the assignment operator with IMAGE_FEATURES and purposely do not add EXTRA_IMAGE_FEATURES, it is not included, which means that the debug-tweaks image feature is not applied, and you need to provide passwords for shell and SSH logins. Again, this makes sense for production build environments where you do not want local configuration settings to override the settings of your production images.

7.3 Image Options

The following sections discuss a list of options that affect how the Yocto Project build system creates your root filesystem images.

7.3.1 Languages and Locales

Additional languages for different territories can easily be added to a root filesystem or your image by adding the IMAGE_LINGUAS variable to an image recipe. Using

```
IMAGE_LINGUAS = "en-gb pt-br"
```

adds the specific language packages for British English and Brazilian Portuguese to the image. However, not all software packages provide locales separated by language and territory. Some of them provide the locale files only by language. In this case, the build system defaults to installing the correct language local files regardless of the territory.

The minimum default for all packages is en-us and is always installed. In addition, the image class defines

```
IMAGE_LINGUAS ?= "de-de fr-fr en-gb"
```

Any additional locale packages, of course, occupy additional space in your root filesystem image. Therefore, if your device does not require any additional language support, it is good practice to set

```
IMAGE_LINGUAS = ""
```

in image recipes.

The build system ignores the languages for packages that do not provide them.

7.3.2 Package Management

The build system can package software packages using the four different packaging formats dpkg (Debian Package Management), opkg (Open Package Management), RPM (Red Hat Package Manager), and tar. Only the first three can be used to create root filesystems. Tar does not provide the necessary metadata package information and database to log what packages in what versions have been installed, which packages conflict with each other, and so on.

The variable PACKAGE_CLASSES in conf/local.conf of your build environment controls what package management systems are used for your builds:

```
PACKAGE_CLASSES = "package_rpm package_ipk package_tar"
```

You can declare more than one packaging class, but you have to provide at least one. The build system creates packages for all classes specified; however, only the first packaging class in the list is used to create the root filesystem of your distribution images. The first packaging class in the list must not be tar.

The build system stores the package feeds organized by the package management system in separate directories in tmp/deploy/<pms>, where <pms> is the name of the respective package management system. Inside those directories, the packages are further subdivided into common, architecture, and machine-dependent packages.

What package management system should you choose for your project? That depends on the requirements of your project. Here are some considerations you may want to take into account:

- Opkg creates and utilizes less package metadata than dpkg and RPM. That makes building faster, and the packages are smaller.
- Dpkg and RPM offer better dependency handling and version management than opkg because of the enhanced package metadata.
- The RPM package manager is written in Python and requires Python to be installed on the target to install packages during runtime of the system.

By default, the build system does not install the package manager on your target system. If you are looking to install packages during runtime of your embedded system, you have to add the package manager using its image feature:

```
IMAGE_FEATURES += "package_management"
```

The build system automatically installs the correct package manager depending on the first entry of `PACKAGE_CLASSES`.

The package management system for your root filesystem is ultimately controlled by the variable `IMAGE_PKGTYPE`. This variable is set automatically by the order of the packaging classes defined by `PACKAGE_CLASSES`. The first packaging class in the list sets the variable. We recommend that you do not set this variable directly.

7.3.3 Image Size

The final size of the root filesystem is dependent on multiple factors and is computed by the build system using the function `_get_rootfs_size()` in the Python module `meta/lib/oe/image.py`. The computation takes into account the actual space required by the root filesystem and the following variable settings. It also ensures that the final root filesystem image size is always sufficient to hold the entire image. Hence, even if you set `IMAGE_ROOTFS_SIZE to` a specific value, the final image may be larger than that value, but it is never smaller.

- **`IMAGE_ROOTFS_SIZE`**: Defines the size in kilobytes of the created root filesystem image. The build system uses this value as a request or recommendation. The final root filesystem image size may be larger depending on the actual space required. The default value is 65536.
- **`IMAGE_ROOTFS_ALIGNMENT`**: Defines the alignment of the root filesystem image in kilobytes. If the final size of the root filesystem image is not a multiple of this value, it is rounded up to the nearest multiple of it. The default value is 1.
- **`IMAGE_ROOTFS_EXTRA_SPACE`**: Adds extra free space to the root filesystem image. The variable specifies the value in kilobytes. For example, to add an additional 4 GB of space, set the variable to `IMAGE_ROOTFS_EXTRA_SPACE = "4194304"`. The default value is 0.
- **`IMAGE_OVERHEAD_FACTOR`**: This variable specifies a multiplicator for the root filesystem image. The factor is applied after the actual space required by the root filesystem has been determined. The default value is 1.3.

After the build system has created the root filesystem in the staging area, a directory specified by the variable IMAGE_ROOTFS, it calculates its actual size in kilobytes using du -ks ${IMAGE_ROOTFS}. The function _get_rootfs_size() computes the final root filesystem image size, as shown by Listing 7-7 in pseudocode.

Listing 7-7 **Root Filesystem Image Size Computation in Pseudocode**

```
_get_rootfs_size():

   ROOTFS_SIZE =`du -ks ${IMAGE_ROOTFS}`
   BASE_SIZE = ROOTFS_SIZE * IMAGE_OVERHEAD_FACTOR

   if (BASE_SIZE < IMAGE_ROOTFS_SIZE):
      IMG_SIZE = IMAGE_ROOTFS_SIZE + IMAGE_ROOTFS_EXTRA_SPACE
   else:
      IMG_SIZE = BASE_SIZE + IMAGE_ROOTFS_EXTRA_SPACE

   IMG_SIZE = IMG_SIZE + IMAGE_ROOTFS_ALIGNMENT - 1
   IMG_SIZE = IMG_SIZE % IMAGE_ROOTFS_ALIGNMENT

   return IMG_SIZE
```

Most commonly, your image recipes set IMAGE_ROOTFS_SIZE and IMAGE_ROOTFS_EXTRA_SPACE to adjust the final root filesystem image size. If you are concerned with the footprint of your root filesystem, then you may also want to reduce IMAGE_OVERHEAD_FACTOR or set it to 1 to shrink your image.

7.3.4 Root Filesystem Types

Eventually, you use the root filesystem image to create a bootable medium for your target or to launch the QEMU emulator. For that purpose, the build system provides the image_types class that can create a root filesystem for various filesystem types.

Your image recipes do not use the image_types class directly but rather set the variable IMAGE_FSTYPES to one or more of the filesystem types provided by the class. Using

```
IMAGE_FSTYPES = "ext3 tar.bz2"
```

creates two root filesystem images, one using the ext3 filesystem and one that is a tar archive compressed using the bzip2 algorithm.

The image_types class defines the variable IMAGE_TYPES, which contains a list of all image types you can specify in IMAGE_FSTYPES. The list shows the filesystem types ordered by core type. Commonly, some of the core types are also used in compressed formats to preserve space. If a compression algorithm is used for the filesystem, the name of the core type is appended with the compression type: <core name>.<compression type>.

- **tar**, **tar.gz**, **tar.bz2**, **tar.xz**, **tar.lz3**: Create uncompressed and compressed root filesystem images in the form of tar archives.
- **ext2**, **ext2.gz**, **ext2.bz2**, **ext2.lzma**: Root filesystem images using the ext2 filesystem without or with compression.

- **`ext3`**, **`ext3.gz`**: Root filesystem images using the ext3 filesystem without or with compression.
- **`btrfs`**: Root filesystem image with B-tree filesystem.
- **`jffs2`**, **`jffs2.sum`**: Uncompressed or compressed root filesystems based on the second generation of the Journaling Flash File System (JFFS2). Since JFFS2 directly supports NAND flash devices, it is a popular choice for embedded systems. It also provides journaling and wear-leveling.
- **`cramfs`**: Root filesystem image using the compressed ROM filesystem (cramfs). The Linux kernel can mount this filesystem without prior decompression. The compression uses the zlib algorithm that compresses files one page at a time to allow random access. This filesystem is read-only to simplify its design, as random write access with compression is difficult to implement.
- **`iso`**: Root filesystem image type using the ISO 9660 standard for bootable CD-ROM. This filesystem type is not a standalone format. It uses ext3 as the underlying filesystem type.
- **`hddimg`**: Root filesystem image for bootable hard drives. It uses ext3 as the actual filesystem type.
- **`squashfs`**, **`squashfs-xz`**: Compressed read-only root filesystem type specifically for Linux, similar to `cramfs` but with better compression and support for larger files and filesystems. `SquashFS` also has a variable block size from 0.5 kB to 64 kB over the fixed 4 kB block size of `cramfs`, which allows for larger file and filesystem sizes. `SquashFS` uses gzip compression, while `squashfs-xz` uses Lempel–Ziv–Markov (LZMA) compression for even smaller images.
- **`ubi`**, **`ubifs`**: Root filesystem images using the unsorted block image (UBI) format for raw flash devices. UBI File System (UBIFS) is essentially a successor to JFFS2. The main differences between the two is that UBIFS supports write caching. Using `ubifs` in `IMAGE_FSTYPES` just creates the `ubifs` root filesystem image. Using `ubi` creates the `ubifs` root filesystem image and also runs the ubinize utility to create an image that can be written directly to a flash device.
- **`cpio`**, **`cpio.gz`**, **`cpio.xz`**, **`cpio.lzma`**: Root filesystem images using uncompressed or compressed copy in and out (CPIO) streams.
- **`vmdk`**: Root filesystem image using the VMware virtual machine disk format. It uses ext3 as the underlying filesystem format.
- **`elf`**: Bootable root filesystem image created with the mkelfImage utility from the Coreboot project (www.coreboot.org).

Once again, which image types to use depends entirely on the requirements of your project, particularly on your target hardware. Boot device, bootloader, memory constraints, and other factors determine what root filesystem types are appropriate for your project. Our recommendation is to specify the root filesystem types ext3 and tar, or better, one of the compressed formats such as tar.bz2, in the image recipe. The ext3 format allows you to easily boot your root filesystem with the QEMU emulator for testing. The tar filesystem can easily be extracted onto partitioned and formatted media. The machine configuration files for your target hardware can then add additional root filesystem types appropriate for it.

7.3.5 Users, Groups, and Passwords

The class `extrausers` provides a comfortable mechanism for adding users and groups to an image as well as setting passwords for user accounts (see Listing 7-8).

Listing 7-8 **Modifying Users, Groups, and Passwords**

```
SUMMARY = "Custom image recipe from scratch"
DESCRIPTION = "Directly assign IMAGE_INSTALL and IMAGE_FEATURES for \
               for direct control over image contents."

LICENSE = "MIT"

# We are using the assignment operator (=) below to purposely overwrite
# the default from the core-image class.
IMAGE_INSTALL = "packagegroup-core-boot packagegroup-base-extended \
                 ${CORE_IMAGE_EXTRA_INSTALL}"

inherit core-image
inherit extrausers

# set image root password
ROOT_PASSWORD = "secret"
DEV_PASSWORD = "hackme"

EXTRA_USERS_PARAMS = "\
   groupadd developers; \
   useradd -p `openssl passwd ${DEV_PASSWORD}` developer; \
   useradd -g developers developer; \
   usermod -p `openssl passwd ${ROOT_PASSWORD}` root; \
   "
```

The listing adds a group named `developers` and a user account named `developer` and adds the user account to the group. It also changes the password for the root account. Commands for adding and modifying groups, users, and passwords are added to the variable `EXTRA_USERS_PARMS`, which is interpreted by the class. The commands understood by the class are

- **`useradd`**: Add user account
- **`usermod`**: Modify user account
- **`userdel`**: Remove user account
- **`groupadd`**: Add user group

- **groupmod**: Modify user group
- **groupdel**: Remove user group

The class executes the respective Linux utilities with the corresponding names. Hence, the options are exactly the same and can easily be found in the Linux man pages. Note that the individual commands must be separated with a semicolon.

Using the option `-p` with the commands `useradd` and `usermod` sets the password of the user account. The password must be provided as the password hash. You can either calculate the password hash manually and add it to the recipe or, as shown in the example, have the recipe calculate it.

A word about the root user account: the build system sets up the root user for an image with an empty password if `debug-tweaks` is included with `IMAGE_FEATURES`. Removing `debug-tweaks` replaces the empty root password with `*`, which disables the account, so logging in as root from the console is no longer possible. For production use, we strongly recommend removing `debug-tweaks` from the build. If your embedded system requires console login capability, you can either set the root password as shown previously or add the `sudo` recipe and set up user accounts as *sudoers*.

For example, if you want to give the developer user account *sudoer* privileges, simply add `sudo` to `IMAGE_INSTALL` and `usermod -a -G sudo developer` to `EXTRA_USERS_PARAMS`.

7.3.6 Tweaking the Root Filesystem

For further customization of the root filesystem after it has been created by the build system and before the actual root filesystem images are created, `ROOTFS_POSTPROCESS_COMMAND` is available (see Listing 7-9). The variable holds a list of shell functions separated by semicolons.

Listing 7-9 **ROOTFS_POSTPROCESS_COMMAND**

```
SUMMARY = "Custom image recipe from scratch"
DESCRIPTION = "Directly assign IMAGE_INSTALL and IMAGE_FEATURES for \
               for direct control over image contents."

LICENSE = "MIT"

# We are using the assignment operator (=) below to purposely overwrite
# the default from the core-image class.
IMAGE_INSTALL = "packagegroup-core-boot packagegroup-base-extended \
                 ${CORE_IMAGE_EXTRA_INSTALL}"

inherit core-image

# Additional root filesystem processing
modify_shells() {
   printf "# /etc/shells: valid login shells\n/bin/sh\n/bin/bash\n" \
          > ${IMAGE_ROOTFS}/etc/shells
}
ROOTFS_POSTPROCESS_COMMAND += "modify_shells;"
```

The example adds the bash shell to /etc/shells. Be sure to always use the += operator to add to ROOTFS_POSTPROCESS_COMMAND, as the build system adds its own postprocessing commands to it.

Sudo Configuration

If you followed the example on giving a user sudoer privileges in the previous paragraph, you probably noticed that it does not work unless you uncomment the line %sudo ALL=(ALL) ALL in /etc/sudoers. A simple shell function added to ROOTFS_POSTPROCESS_COMMAND takes care of that when the root filesystem image is created (see Listing 7-10).

Listing 7-10 **Sudo Configuration**

```
modify_sudoers() {
   sed 's/# %sudo/%sudo/' < ${IMAGE_ROOTFS}/etc/sudoers > \
        ${IMAGE_ROOTFS}/etc/sudoers.tmp
   mv ${IMAGE_ROOTFS}/etc/sudoers.tmp ${IMAGE_ROOTFS}/etc/sudoers
}
ROOTFS_POSTPROCESS_COMMAND += "modify_sudoers;"
```

The script simply uncomments the line using sed.

SSH Server Configuration

All core images automatically include an SSH server for remote shell access to the system. By default, the server is configured to allow login with user name and password. Using public key infrastructure (PKI) provides an additional level of security but requires configuration of the root server and installation of keys into the root filesystem. A ROOTFS_POSTPROCESS_COMMAND can also easily be used to accomplish that task (see Listing 7-11).

Listing 7-11 **SSH Server Configuration**

```
configure_sshd() {
   # disallow password authentication
   echo "PasswordAuthentication no" >> ${IMAGE_ROOTFS}/etc/ssh/sshd_config
   # create keys in tmp/deploy/keys
   mkdir -p ${DEPLOY_DIR}/keys
   if [ ! -f ${DEPLOY_DIR}/keys/${IMAGE_BASENAME}-sshroot ]; then
      ssh-keygen -t rsa -N '' \
         -f ${DEPLOY_DIR}/keys/${IMAGE_BASENAME}-sshroot
   fi
   # add public key to authorized_keys for root
   mkdir -p ${IMAGE_ROOTFS}/home/root/.ssh
   cat ${DEPLOY_DIR}/keys/${IMAGE_BASENAME}-sshroot.pub \
       >> ${IMAGE_ROOTFS}/home/root/.ssh/authorized_keys
}
ROOTFS_POSTPROCESS_COMMAND += "configure_sshd;"
```

The script first disables authentication with user name and password for SSH. It then creates a key pair in tmp/deploy/keys inside the build environment using the name of the root filesystem image, essentially the name of the image recipe. If a previous build has already created a set of keys, they are preserved. Finally, the script adds the public key to the `authorized_keys` file in `/home/root/.ssh`, which is typical for SSH configuration. Login keys for other users can be created in a similar way.

This method works well if you do not require different keys for each device that you build, as every copy of the root filesystem of course contains the same keys. If you need different keys or, in general, individual configuration for your devices, then you need to devise a provisioning system for your device production.

7.4 Distribution Configuration

The build system provides a mechanism for global configuration that applies to all images built. This mechanism is called *distribution configuration* or *distribution policy*. It is simply a configuration file that contains variable settings. The distribution configuration is included through the `DISTRO` variable setting in the build environment configuration file `conf/local.conf`:

```
DISTRO = "poky"
```

The variable setting corresponds to a distribution configuration file whose base name is the same as the variable's argument with the file extension `.conf`. For the preceding example, the build system searches for a distribution configuration file with the name `poky.conf` in the subdirectory `conf/distro` in all metadata layers included by the build environment.

7.4.1 Standard Distribution Policies

The Yocto Project provides several distribution configuration files for standard configuration policies:

- **`poky`**: Poky is the default policy for the Yocto Project's reference distribution Poky. It is a good choice for getting started with the Yocto Project and as a template for your own distribution configuration files.
- **`poky-bleeding`**: This distribution configuration is based on `poky` but sets the versions for all packages to the latest revision. It is commonly used by the Yocto Project developers for integration test purposes. You may, of course, use it, but be aware that there could be issues with packages with incompatible versions.
- **`poky-lsb`**: This distribution configuration is for a stack that complies with LSB. It is preferably used with the `core-image-lsb` image target and image targets derived from it. It inherits the base settings from `poky` and adds global configuration settings to enable security and includes default libraries required for LSB compliance.
- **`poky-tiny`**: This distribution configuration tailors the settings to yield a very compact Linux OS stack for embedded devices. It is based on `poky` but provides only the bare minimum functionality necessary to support the hardware and a BusyBox

environment. It does not support any video but only a serial console. Because of its slim configuration, only the `core-image-minimal` image target and image targets based on it can be built with the `poky-tiny` distribution configuration.

The standard distribution policies, particularly `poky`, are good starting points for your own distribution configuration. Let's have a closer look at the `poky` distribution configuration to understand how distribution policies are set and how we can use them for our own projects.

7.4.2 Poky Distribution Policy

You can find the file `poky.conf` containing the Poky distribution policy in the `meta-yocto/conf/distro` directory of the build system. We replicated its contents here for convenience, reformatted the file to fit on the page, grouped the variable settings into logical blocks, and added some comments (see Listing 7-12).

Listing 7-12 **Poky Distribution Policy `meta-yocto/conf/distro/poky.conf`**

```
# Distribution Information

DISTRO = "poky"
DISTRO_NAME = "Poky (Yocto Project Reference Distro)"
DISTRO_VERSION = "1.6+snapshot-${DATE}"
DISTRO_CODENAME = "next"
MAINTAINER = "Poky <poky@yoctoproject.org>"
TARGET_VENDOR = "-poky"

# SDK Information
SDK_NAME = \
    "${DISTRO}-${TCLIBC}-${SDK_ARCH}-${IMAGE_BASENAME}-${TUNE_PKGARCH}"
SDK_VERSION := \
    "${@'${DISTRO_VERSION}'.replace('snapshot-${DATE}','snapshot')}"
SDK_VENDOR = "-pokysdk"
SDKPATH = "/opt/${DISTRO}/${SDK_VERSION}"

# Distribution Features
# Override these in poky based distros
POKY_DEFAULT_DISTRO_FEATURES = "largefile opengl ptest multiarch wayland"
POKY_DEFAULT_EXTRA_RDEPENDS = "packagegroup-core-boot"
POKY_DEFAULT_EXTRA_RRECOMMENDS = "kernel-module-af-packet"

DISTRO_FEATURES ?= "${DISTRO_FEATURES_DEFAULT} ${DISTRO_FEATURES_LIBC} \
                    ${POKY_DEFAULT_DISTRO_FEATURES}"

# Preferred Versions for Packages
PREFERRED_VERSION_linux-yocto ?= "3.14%"
PREFERRED_VERSION_linux-yocto_qemux86 ?= "3.14%"
PREFERRED_VERSION_linux-yocto_qemux86-64 ?= "3.14%"
PREFERRED_VERSION_linux-yocto_qemuarm ?= "3.14%"
PREFERRED_VERSION_linux-yocto_qemumips ?= "3.14%"
PREFERRED_VERSION_linux-yocto_qemumips64 ?= "3.14%"
PREFERRED_VERSION_linux-yocto_qemuppc ?= "3.14%"
```

```
# Dependencies
DISTRO_EXTRA_RDEPENDS += " ${POKY_DEFAULT_EXTRA_RDEPENDS}"
DISTRO_EXTRA_RRECOMMENDS += " ${POKY_DEFAULT_EXTRA_RRECOMMENDS}"

POKYQEMUDEPS = "${@bb.utils.contains( \
     "INCOMPATIBLE_LICENSE", "GPLv3", "", "qemu-config",d)}"
DISTRO_EXTRA_RDEPENDS_append_qemuarm = " ${POKYQEMUDEPS}"
DISTRO_EXTRA_RDEPENDS_append_qemumips = " ${POKYQEMUDEPS}"
DISTRO_EXTRA_RDEPENDS_append_qemuppc = " ${POKYQEMUDEPS}"
DISTRO_EXTRA_RDEPENDS_append_qemux86 = " ${POKYQEMUDEPS}"
DISTRO_EXTRA_RDEPENDS_append_qemux86-64 = " ${POKYQEMUDEPS}"

# Target C Library Configuration
TCLIBCAPPEND = ""

# Target Architectures for QEMU
# (see meta/recipes-devtools/qemu/qemu-targets.inc)
QEMU_TARGETS ?= "arm i386 mips mipsel ppc x86_64"
# Other QEMU_TARGETS "mips64 mips64el sh4"

# Package Manager Configuration
EXTRAOPKGCONFIG = "poky-feed-config-opkg"

# Source Mirrors
PREMIRRORS ??= "\
bzr://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
cvs://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
git://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
gitsm://.*/.* http://downloads.yoctoproject.org/mirror/sources/ \n \
hg://.*/.*    http://downloads.yoctoproject.org/mirror/sources/ \n \
osc://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
p4://.*/.*    http://downloads.yoctoproject.org/mirror/sources/ \n \
svk://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n \
svn://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \n"

MIRRORS =+ "\
ftp://.*/.*      http://downloads.yoctoproject.org/mirror/sources/ \n \
http://.*/.*     http://downloads.yoctoproject.org/mirror/sources/ \n \
https://.*/.*    http://downloads.yoctoproject.org/mirror/sources/ \n"

# Build System Configuration

# Configuration File and Directory Layout Versions
LOCALCONF_VERSION = "1"
LAYER_CONF_VERSION ?= "6"
#
# OELAYOUT_ABI allows us to notify users when the format of TMPDIR changes
# in an incompatible way. Such changes should usually be detailed in the
# commit that breaks the format and have been previously discussed on the
# mailing list with general agreement from the core team.
#
OELAYOUT_ABI = "8"

# Default hash policy for distro
BB_SIGNATURE_HANDLER ?= 'OEBasicHash'

# Build System Checks
```

```
# add poky sanity bbclass
INHERIT += "poky-sanity"

# The CONNECTIVITY_CHECK_URIs are used to test whether we can successfully
# fetch from the network (and warn you if not). To disable the test, set
# the variable to be empty.
# Git example url: \
     git://git.yoctoproject.org/yocto-firewall-test;protocol=git;rev=HEAD

CONNECTIVITY_CHECK_URIS ?= " \
             https://eula-downloads.yoctoproject.org/index.php \
             http://bugzilla.yoctoproject.org/report.cgi"

SANITY_TESTED_DISTROS ?= " \
            Poky-1.4 \n \
            Poky-1.5 \n \
            Poky-1.6 \n \
            Ubuntu-12.04 \n \
            Ubuntu-13.10 \n \
            Ubuntu-14.04 \n \
            Fedora-19 \n \
            Fedora-20 \n \
            CentOS-6.4 \n \
            CentOS-6.5 \n \
            Debian-7.0 \n \
            Debian-7.1 \n \
            Debian-7.2 \n \
            Debian-7.3 \n \
            Debian-7.4 \n \
            SUSE-LINUX-12.2 \n \
            openSUSE-project-12.3 \n \
            openSUSE-project-13.1 \n \
            "

# QA check settings - a little stricter than the OE-Core defaults
WARN_QA = "textrel files-invalid incompatible-license xorg-driver-abi \
           libdir unknown-configure-option"
ERROR_QA = "dev-so debug-deps dev-deps debug-files arch pkgconfig la \
            perms useless-rpaths rpaths staticdev ldflags pkgvarcheck \
            already-stripped compile-host-path dep-cmp \
            installed-vs-shipped  install-host-path packages-list \
            perm-config perm-line perm-link pkgv-undefined \
            pn-overrides split-strip var-undefined version-going-backwards"
```

The file shown in the listing is from the head of the Yocto Project Git repository at the writing of this book. Depending on what version of the Yocto Project tools you are using, this file may look slightly different. The file is an example of a distribution policy only. It provides the variable settings most commonly associated with the configuration of a distribution. You are not limited to using just the settings shown in the listing, and you can remove settings if you do not need them for your project.

Distribution Information

This section of the distribution policy file contains settings for general information about the distribution.

- **DISTRO**: Short name of the distribution. The value must match the base name of the distribution configuration file.
- **DISTRO_NAME**: The long name of the distribution. Various recipes reference this variable. Its contents are shown on the console boot prompt.
- **DISTRO_VERSION**: Distribution version string. It is referenced by various recipes and used in filenames' distribution artifacts. It is shown on the console boot prompt.
- **DISTRO_CODENAME**: A code name for the distribution. It is currently used only by the LSB recipes and copied into the `lsb-release` system configuration file.
- **MAINTAINER**: Name and e-mail address of the distribution maintainer.
- **TARGET_VENDOR**: Target vendor string that is concatenated with various variables, most notably target system (`TARGET_SYS`). `TARGET_SYS` is a concatenation of target architecture (`TARGET_ARCH`), target vendor (`TARGET_VENDOR`), and target operating system (`TARGET_OS`), such as `i586-poky-linux`. The three parts are delimited by hyphens. The `TARGET_VENDOR` string must be prefixed with the hyphen, and `TARGET_OS` must not. This is one of the many unfortunate inconsistencies of the OpenEmbedded build system. You may want to set this variable to your or your company's name.

SDK Information

The settings in this section provide the base configuration for the SDK.

- **SDK_NAME**: The base name that the build system uses for SDK output files. It is derived by concatenating the `DISTRO`, `TCLIBC`, `SDK_ARCH`, `IMAGE_BASENAME`, and `TUNE_PKGARCH` variables with hyphens. There is not much reason for you to change that string from its default setting, as it provides all the information needed to distinguish different SDKs.
- **SDK_VERSION**: SDK version string, which is commonly set to `DISTRO_VERSION`.
- **SDK_VENDOR**: SDK vendor string, which serves a similar purpose as `TARGET_VENDOR`. Like `TARGET_VENDOR`, the string must be prefixed with a hyphen.
- **SDKPATH**: Default installation path for the SDK. The SDK installer offers this path to the user during installation of an SDK. The user can accept it or enter an alternative path. The default value `/opt/${DISTRO}/${SDK_VERSION}` installs the SDK into the `/opt` system directory, which requires root privileges. A viable alternative would be to install the SDK into the user's home directory by setting `SDKPATH = "${HOME}/${DISTRO}/${SDK_VERSION}"`.

Distribution Features

These feature settings provide specific functionality for the distribution.

- **DISTRO_FEATURES**: A list of distribution features that enable support for certain functionality within software packages. The assignment in the `poky.conf` distribution policy file includes `DISTRO_FEATURES_DEFAULT` and `DISTRO_FEATURES_LIBC`.

Both contain default distribution feature settings. We discuss distribution features and how they work and the default configuration in the next two sections.

Preferred Versions

Version settings prescribe particular versions for packages rather than the default versions.

- **`PREFERRED_VERSION`**: Using `PREFERRED_VERSION` allows setting particular versions for software packages if you do not want to use the latest version, as it is the default. Commonly, that is done for the Linux kernel but also for software packages on which your application software has strong version dependencies.

Dependencies

These settings are declarations for dependencies required for distribution runtime.

- **`DISTRO_EXTRA_RDEPENDS`**: Sets runtime dependencies for the distribution. Dependencies declared with this variable are required for the distribution. If these dependencies are not met, building the distributions fails.
- **`DISTRO_EXTRA_RRECOMMENDS`**: Packages that are recommended for the distribution to provide additional useful functionality. These dependencies are added if available but building the distribution does not fail if they are not met.

Toolchain Configuration

These settings configure the toolchain used for building the distribution.

- **`TCMODE`**: This variable selects the toolchain that the build system uses. The default value is `default`, which selects the internal toolchain built by the build system (gcc, binutils, etc.). The setting of the variable corresponds to a configuration file `tcmode-${TCMODE}.inc`, which the build system locates in the path `conf/distro/include`. This allows including an external toolchain with the build system by including a toolchain layer that provides the necessary tools as well as the configuration file. If you are using an external toolchain, you must ensure that it is compatible with the Poky build system.
- **`TCLIBC`**: Specifies the C library to be used. The build system currently supports EGLIBC, uClibc, and musl. The setting of the variable corresponds to a configuration file `tclibc-${TCLIBC}.inc` that the build system locates in the path `conf/distro/include`. These configuration files set preferred providers for libraries and more.
- **`TCLIBCAPPEND`**: The build system appends this string to other variables to distinguish build artifacts by C library. If you are experimenting with different C libraries, you may want to use the settings

  ```
  TCLIBCAPPEND = "-${TCLIBC}"
  TMPDIR .= "${TCLIBCAPPEND}"
  ```

in your distribution configuration, which creates a separate build output directory structure for each C library.

Mirror Configuration

The settings in this section configure the mirrors for downloading source packages.

- **`PREMIRRORS`** and **`MIRRORS`**: The Poky distribution adds these variables to set its mirror configuration to use the Yocto Project repositories as a source for downloads. If you want to use your own mirrors, you can add them to your distribution configuration file. However, since mirrors are not strictly distribution settings, you may want to add these variables to the `local.conf` file of your build environment. Another alternative would be to add them to the `layer.conf` file of a custom layer.

Build System Configuration

These settings define the requirements for the build system.

- **`LOCALCONF_VERSION`**: Sets the expected or required version for the build environment configuration file `local.conf`. The build system compares this value to the value of the variable `CONF_VERSION` in `local.conf`. If `LOCALCONF_VERSION` is a later version than `CONF_VERSION`, the build system may be able to automatically upgrade `local.conf` to the newer version. Otherwise, the build system exits with an error message.
- **`LAYER_CONF_VERSION`**: Sets the expected or required version for the `bblayers.conf` configuration file of a build environment. The build system compares this version to the value of `LCONF_VERSION` set by `bblayers.conf`. If `LAYER_CONF_VERSION` is a later version than `LCONF_VERSION`, the build system may be able to automatically upgrade `bblayers.conf` to the newer version. Otherwise, the build system exits with an error message.
- **`OELAYOUT_ABI`**: Sets the expected or required version for the layout of the output directory `TMPDIR`. The build system stores the actual layout version in the file `abi_version` inside of `TMPDIR`. If the two are incompatible, the build system exits with an error message. This typically happens only if you are using a newer version of the build system with a build environment that was created by a previous version and the layout changed incompatibly. Deleting `TMPDIR` resolves the issue by re-creating the directory.
- **`BB_SIGNATURE_HANDLER`**: The signature handler used for signing shared state cache entries and creating stamp files. The value references a signature handler function that, because of its complexity, is typically implemented in Python. The code in `meta/lib/oe/sstatesig.py` implements `OEBasic` and `OEBasicHash` based on the BitBake signature generators `SignatureGeneratorBasic` and `SignatureGeneratorBasicHash` defined by `bitbake/lib/bb/siggen.py` and illustrates how to insert your own signature handler function. The two signature handlers are principally the same, but `OEBasicHash` includes the task code in the signature, which causes any change to

metadata to invalidate stamp files and shared state cache entries without explicitly changing package revision numbers. Using the default value of `OEBasicHash` is typically sufficient for most applications.

Build System Checks

These configuration variables control various validators to catch build system misconfigurations.

- **`INHERIT += "poky-sanity"`**: Inherits the class `poky-sanity`, which is required to perform the build system checks. It is recommended that you include this directive in your own distribution configuration files.
- **`CONNECTIVITY_CHECK_URIS`**: A list of URIs that the build system tries to verify network connectivity. In the case of Poky, these point to files on the Yocto Project's high-availability infrastructure. If you intend to use your own mirrors for downloading source packages, you could use URIs pointing to files on your mirror servers to verify proper connectivity.
- **`SANITY_TESTED_DISTROS`**: A list of Linux distributions the Poky build system has been tested on. The build system verifies the Linux distribution it is running on against this list. If that distribution is not in the list, Poky displays a warning message and starts the build process regardless. Poky runs on most current Linux distributions, and in most cases, building works just fine even if the distribution is not officially supported.

QA Checks

The QA checks are defined and implemented by `meta/classes/insane.bbclass`. This class also defines the QA tasks that are included with the build process. QA checks are performed after configuration, packaging, and other build tasks. The following two variables define which QA checks cause warning messages and which checks cause the build system to terminate the build with an error message:

- **`WARN_QA`**: A list of QA checks that create warning messages, but the build continues
- **`ERROR_QA`**: A list of QA checks that create error messages, and the build terminates

The preceding list represents the most common variable settings used by a distribution configuration. For your own distribution configuration, you may add and/or omit variables as needed.

7.4.3 Distribution Features

Distribution features enable support for certain functionality within software packages. Adding a distribution feature to the variable `DISTRO_FEATURES` adds the functionality of this feature to software packages that support it during build time. For instance, if a software package can be built for console as well as graphical user interfaces, then adding `x11` to `DISTRO_FEATURES` configures that software package so that it is built with X11 support. Unlike the `x11` image feature, this does not mean that the X11 packages are installed in your target root filesystem. The distribution feature only prepares a software package for X11 support so that it uses X11 on a system where the X11 base packages are installed.

Using `DISTRO_FEATURES` gives you granular control over how software packages are built. If you do not need a particular functionality, omitting the distribution feature enabling it typically results in a smaller footprint for a particular software package.

Using

```
$ grep -R DISTRO_FEATURES *
```

from the installation directory of your build system gives you a list of all the recipes and include files that use `DISTRO_FEATURES` to conditionally modify configuration settings or build processes dependent on what distribution features are enabled.

Recipes typically scan `DISTRO_FEATURES` using

```
bb.utils.contains('DISTRO_FEATURES', <feature>, <true_val>, <false_val>)
```

to determine if a particular distribution feature is enabled by `DISTRO_FEATURES`. The function returns `true_val` if `DISTRO_FEATURES` contains `feature` and `false_val` otherwise. That makes it convenient for the developer to assign values to BitBake variables or use the function in if-then-else statements. Typically, this is used by the `do_configure` task to modify the configuration based on `DISTRO_FEATURES`. For some packages, it may provide flags to makefiles.

A prime example is the recipe to build the EGLIBC library. EGLIBC allows enabling functionality by setting configuration options. The file `meta/recipes-core/egligc/egilbc-options.inc`, which is included by the recipe, sets the configuration options based on the distribution features provided by `DISTRO_FEATURES`.

The following list shows the most common distribution features that you can add to `DISTRO_FEATURES` to enable functionality in software packages globally across your distribution:

- **`alsa`**: Enable support for the Advanced Linux Sound Architecture (ALSA), including the installation of open source compatibility modules if available.
- **`bluetooth`**: Enable support for Bluetooth.
- **`cramfs`**: Enable support for the compressed filesystem CramFS.
- **`directfb`**: Enable support for DirectFB.
- **`ext2`**: Enable support and include tools for devices with internal mass storage devices such as hard disks instead of flash devices only.
- **`ipsec`**: Enable support for authentication and encryption using Internet Protocol Security (IPSec).
- **`ipv6`**: Enable support for Internet Protocol version 6 (IPv6).

- **`irda`**: Enable support for wireless infrared data communication as specified by the Infrared Data Association (IrDA).
- **`keyboard`**: Enable keyboard support, which includes loading of keymaps during boot of the system.
- **`nfs`**: Enable client NFS support for mounting NFS exports on the system.
- **`opengl`**: Include the Open Graphics Library (OpenGL), which is an application programming interface for rendering 2D and 3D graphics. OpenGL runs on different platforms and provides bindings for most common programming languages.
- **`pci`**: Enable support for the PCI bus.
- **`pcmcia`**: Enable PCMCIA and CompactFlash support.
- **`ppp`**: Enable Point-to-Point Protocol (PPP) support for dial-up networking.
- **`smbfs`**: Enable support and include clients for Microsoft's Server Message Block (SMB) for sharing remote filesystems, printers, and other devices over networks.
- **`systemd`**: Include support for the system management daemon (systemd) that replaces the SysVinit script-based system for starting up and shutting down a system.
- **`sysvinit`**: Include support for the SysVinit system manager.
- **`usbgadget`**: Enable support for the Linux-USB Gadget API Framework that allows a Linux device to act like a USB device (slave role) when connected to another system.
- **`usbhost`**: Enable USB host support allowing client devices such as keyboards, mice, cameras, and more to be connected to the system's USB ports and detected by it.
- **`wayland`**: Enable support for the Wayland compositor protocol and include the Weston compositor.
- **`wifi`**: Enable WiFi support.
- **`x11`**: Include the X11 server and libraries.

The list does not include the distribution features for the configuration of the C library. These distribution features all begin with `libc-`. They enable support for functionality provided by the C library if the C library is configurable like the Yocto Project's default C library glibc. If you are using glibc, then you do not have to worry about setting these distribution features, as they are inherited from the default distribution setup, which is covered in the next section.

If you have already been working with the Yocto Project, you may have noticed that there is also a variable called `MACHINE_FEATURES` and that the permissible list of machine features has a large intersection with the distribution feature list. For example, both `MACHINE_FEATURES` and `DISTRO_FEATURES` provide the feature `bluetooth`. Enabling Bluetooth in `DISTRO_FEATURES` causes the Bluetooth packages for hardware support to be installed and also enables Bluetooth support for various software packages. However, enabling Bluetooth in MACHINE_FEATURES only causes the Bluetooth packages for hardware support to be installed. This gives you control over functionality on the machine and the distribution level. We discuss machine features in detail when we are looking into Yocto Project board support packages.

7.4.4 System Manager

The build system supports SysVinit, the traditional script-based system manager, as well as the system management daemon (systemd), a replacement for SysVinit that offers better prioritization and dependency handling between services and the ability to start services in parallel to speed up the boot sequence.

SysVinit is the default system manager for Linux distributions built by Poky. You do not have to change the configuration if you want to use SysVinit.

To enable systemd, you need to add it to the distribution features and set it as the system manager. Add the following to your distribution configuration file:

```
DISTRO_FEATURES_append = " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
```

The first line installs systemd in the root filesystem. The second line enables it as the system manager. Installing and enabling systemd does not remove SysVinit from your root filesystem if it is also included in DISTRO_FEATURES. If you are using one of the standard distribution configurations, such as poky, then you can remove it from DISTRO_FEATURES with

```
DISTRO_FEATURES_BACKFULL_CONSIDERED = "sysvinit"
```

which is easier than redefining DISTRO_FEATURES altogether. For your own distribution configuration, you can of course simply omit SysVinit from the DISTRO_FEATURES list.

The SysVinit initscripts to start the individual system services are typically part of the package that provides the service. To conserve space in the root filesystem, you may not want to install the initscripts if you want to use systemd exclusively. Use

```
VIRTUAL-RUNTIME_initscripts = ""
```

to prevent the build system from installing the SysVinit initscripts.

A word of caution: some daemons may not yet have been adapted for use with systemd and therefore systemd service files are not available. If you come across such software, you may have to do the adaptation yourself. If you do so, please consider submitting your work to upstream.

7.4.5 Default Distribution Setup

The OE Core metadata layer provides default distribution setup through the file meta/conf/distro/defaultsetup.conf and a series of other files included by it (see Listing 7-13). It is not quite obvious how this default distribution setup is included into the build configuration, as this file is not included by distribution policy configuration files such as poky.conf, which we discussed earlier. Instead, the file is included by BitBake's main configuration file, bitbake.conf.

Knowing about defaultsetup.conf and understanding its settings is important because your own distribution policy configuration may extend or overwrite some of the default variable settings provided by it. If you do not set up the default distribution correctly, you may inadvertently lose important default settings, and your distribution build may fail or not yield the desired results.

Listing 7-13 **Default Distribution Setup meta/conf/distro/defaultsetup.conf**

```
include conf/distro/include/default-providers.inc
include conf/distro/include/default-versions.inc
include conf/distro/include/default-distrovars.inc
include conf/distro/include/world-broken.inc

TCMODE ?= "default"
require conf/distro/include/tcmode-${TCMODE}.inc

TCLIBC ?= "eglibc"
require conf/distro/include/tclibc-${TCLIBC}.inc

# Allow single libc distros to disable this code
TCLIBCAPPEND ?= "-${TCLIBC}"
TMPDIR .= "${TCLIBCAPPEND}"

CACHE = "${TMPDIR}/cache/${TCMODE}-${TCLIBC}${@['', '/' + \
         str(d.getVar('MACHINE', True))][bool(d.getVar('MACHINE', \
         True))]}${@['', '/' + str(d.getVar('SDKMACHINE', True))] \
         [bool(d.getVar('SDKMACHINE', True))]}"

USER_CLASSES ?= ""
PACKAGE_CLASSES ?= "package_ipk"
INHERIT_BLACKLIST = "blacklist"
INHERIT_DISTRO ?= "debian devshell sstate license"
INHERIT += "${PACKAGE_CLASSES} ${USER_CLASSES} ${INHERIT_DISTRO} \
            ${INHERIT_BLACKLIST}"
```

The file first includes three other files with default settings: default-providers.inc, default-versions.inc, and default-distrovars.inc. The names for these files are indicative of what the file content is providing.

The file default-distrovars.inc in particular provides default settings for DISTRO_FEATURES, DISTRO_FEATURES_DEFAULT, DISTRO_FEATURES_LIBC, and DISTRO_FEATURES_LIBC_DEFAULT. If you are going to set DISTRO_FEATURES in your own distribution policy configuration file, you need to pay attention that you do not inadvertently remove the default settings by overwriting the variable. A safe way of doing so is to use an assignment like

```
DISTRO_FEATURES ?= "${DISTRO_FEATURES_DEFAULT} ${DISTRO_FEATURES_LIBC} \
                    ${MY_DISTRO_FEATURES}"
MY_DISTRO_FEATURES = "<distro features>"
```

which includes all default settings and adds another variable to include additional distribution features as needed.

The configuration file `defaultsetup.conf` also sets the defaults for `TCMODE` and `TCLIBC` and includes their respective configuration files, as described earlier.

7.5 External Layers

For the examples in the preceding sections, we used software packages and package groups from the OE Core layer `meta` and the Yocto Project base layer `meta-yocto`.

With steadily increasing support and contributions to the Yocto Project and OpenEmbedded, a growing number of additional layers with hundreds of recipes for myriad software packages are now available. Many of them are cataloged on the OpenEmbedded website. If you are looking for a recipe to build a specific software package, chances are that someone has already done the work.

The OpenEmbedded website's metadata index[1] lets you search by layer, recipe, and machine. For example, searching for Java by layer gives you a list of the layers that provide Java. Searching for JDK by recipes gives you a list of all recipes that build JDK packages together with the layer that provides the recipe.

The metadata index also lets you filter for the supported Yocto Project release to see if a recipe or layer is compatible with that particular release. Once you find the layer containing the software package recipe you are looking for, all you need to do is download the layer, include its path into the `BBLAYERS` variable of the `conf/bblayers.conf` of your build environment, and add the desired software package to your image using one of the methods described earlier.

7.6 Hob

Hob is a graphical user interface for BitBake provided by the Yocto Project. It is one of the Yocto Project's subprojects and is maintained by the Yocto Project development team.

Why is it called Hob? In the early days of Hob, the three letters stood for *Human-Oriented Builder*. However, that does not really sound too appealing and now the name of the tool is commonly associated with *hob*, the British English word for cooktop. And that fits well into the scheme of BitBake and recipes.

With Hob you can conveniently customize your root filesystem images using your mouse rather than editing text files. If that's the case, why didn't we introduce Hob first rather than explain how to build your custom Linux distribution the "hard" way? There are a couple of reasons:

- You can do a lot with Hob, but not everything.
- Hob is a frontend to BitBake and your build environment. It manipulates files in your build environment, launches BitBake, and collects build results.

1. http://layers.openembedded.org

Understanding how this is done manually helps you understand what Hob does in particular if something goes wrong.

- Although Hob may hide some of the complexity, you still need to know the terminology and how certain variable settings influence your build results.

Using Hob is rather simple. First, set up a build environment and then launch Hob from inside it:

```
$ source oe-init-build-env build
$ hob
```

Hob launches and then verifies your build environment. After that check is completed, you see a screen similar to the one in Figure 7-1 (we already made choices for the machine and image recipe).

The Hob user interface is easy to understand:

- **Select a machine**: From the drop-down menu, choose the machine you want to build for. The list shows all the machines that are defined by any layer included with the build environment. Selecting the machine changes the `MACHINE` variable setting in the `con/local.conf` file.

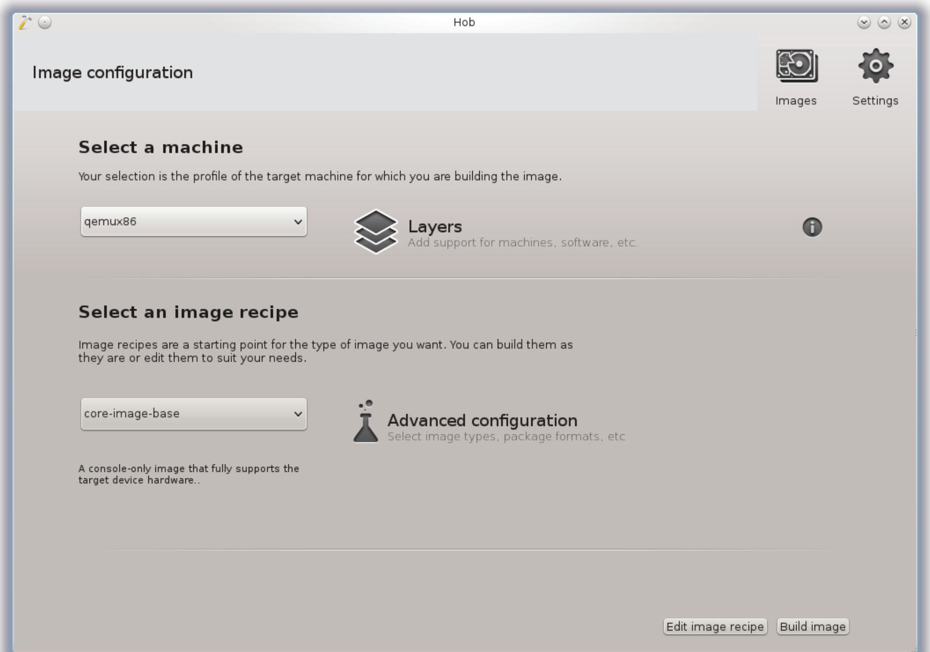

Figure 7-1 Hob

- **Layers**: Click this button to open a graphical editor that lets you include layers with and remove them from your build environment. Doing so modifies the `conf/bblayers.conf` file in your build environment.
- **Select an image recipe**: From this drop-down menu, you can choose the image that you want to build. This provides the image target to BitBake similar to running `bitbake <image-target>`. The menu contains image targets from all layers included with your build environment.
- **Advanced configuration**: Clicking on this button opens a menu that lets you select root filesystem types, packaging format, distribution policy, image size, and more, as outlined in Sections 7.3 and 7.4. Hob adds these options to the `conf/local.conf` file of the build environment.
- **Edit image recipe**: This button at the bottom of the screen lets you modify the image recipe by adding and/or removing packages and/or package groups. Doing so effectively modifies the `IMAGE_INSTALL` variable of the image target. You cannot, however, define new package groups from the Hob user interface. For that task, you have to write your package group recipe as explained in Section 7.1.6. But, of course, if you wrote your package recipe and included the layer it resides in with Hob, then you are able to select it from the package groups list.
- **Settings**: This button in the upper right corner of the user interface allows you to modify general settings contained in `conf/local.conf` such as parallelism, download directory, shared state cache, mirrors, and network proxies. Using the *Others* tab, you can add any variable to `conf/local.conf` and assign a value to it.
- **Images**: This button next to the *Settings* button in the upper right corner of the Hob user interface displays a list of previously built images. The list is created by parsing the `tmp/deploy/images/<machine>` subdirectories of the build environment. You can select an image from the list, run it if it is a QEMU image, or rebuild it.
- **Build image**: This button launches BitBake with the selected configuration and image target. The user interface switches to the *Log* tab of the build view from which you can follow the build process. This view has a major advantage over the BitBake output when started from the command line: not only do you see the tasks that are currently run but also the pending tasks and the ones that already have completed. If there are any build issues, warnings, or errors, they are logged underneath the *Issues* tab. There you can examine build issues and directly view the entire log file of a task without navigating through the build environment directory structure.

After the build finishes, Hob presents you with a summary page where you can view the created files in the file browser of your build system. You can also examine a summary log showing the run results for each task as well as any notes, warnings, or error messages. If you used Hob to build a root filesystem image and Linux kernel for the QEMU emulator, you can launch QEMU directly from Hob to verify your image by clicking on the *Run image* button in the lower right corner of the user interface. From the summary page, you can also make changes to your configuration and run a new build.

Whether you prefer Hob over configuring your build environment, customizing your target images, and launching BitBake manually is entirely up to you. Hob is great for rapid prototyping and to quickly enable somebody who is not all that familiar with BitBake and the Yocto Project to build predefined root filesystem image targets. Hob does not allow you to create your own image recipes, nor can you create your own distribution policy files with it (or even edit them). For these tasks, you need to set up your own layer and create the necessary files and recipes manually.

From Yocto Project version 2.1 on, Hob is being deprecated in favor of the web-based Toaster, which we explore in detail in Chapter 13.

7.7 Summary

The largest building block of a Linux distribution is the user space that contains the various libraries and applications that provide the essential functionality of the system. This chapter presented the fundamental concepts on how the Poky build system creates root filesystem images and how you can customize them to meet your requirements.

- The OpenEmbedded build system's core images provide distribution blueprints that you can extend and modify.
- Core images can easily be extended by appending packages and package groups to the list contained in the variable `IMAGE_INSTALL`.
- The QEMU emulator is a convenient and quick way to test your root file before booting it on an actual device.
- Enabling the build history lets you track changes to your images and compare subsequent executions of the build process.
- Creating your own image recipes that build on core image recipes by including them provides you with more control over what packages your root filesystem image contains. Image recipes that directly inherit the `core-image` class let you build root filesystem images from scratch.
- Package groups are a mechanism to bundle multiple packages and reference them by a single name, which greatly simplifies image customization with the `IMAGE_INSTALL` variable. Poky provides a series of predefined package groups that organize common packages.
- The build system can produce root filesystem images in various output formats. Some of them can be written directly to storage media such as flash devices to boot a system.
- Setting up a distribution policy allows operating system configuration independent of the content of the root filesystem. It also provides the means to use an external toolchain with the build system and to change the C library.
- Hob is a graphical user interface for BitBake. Launched from within an initialized build environment, it allows configuring and building of root filesystem images without modifying files using a text editor.

Index

Symbols

A

D

S

Y