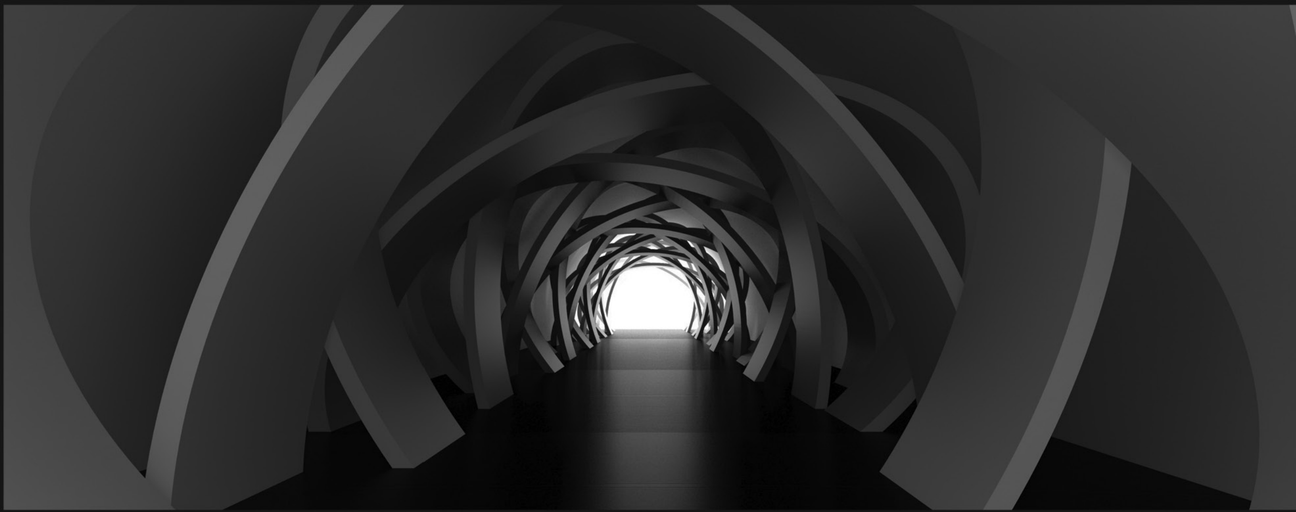
packt



3RD EDITION

Embedded Linux Development Using Yocto Project

Leverage the power of the Yocto Project to build efficient Linux-based products

OTAVIO SALVADOR
DAIANE ANGOLINI

Embedded Linux Development Using Yocto Project

Leverage the power of the Yocto Project to build efficient Linux-based products

Otavio Salvador

Daiane Angolini

BIRMINGHAM—MUMBAI

Embedded Linux Development Using Yocto Project

Group Product Manager: Mohd Riyan Khan
Publishing Product Manager: Mohd Riyan Khan
Content Development Editor: Sujata Tripathi
Technical Editor: Arjun Varma
Copy Editor: Safis Editing
Project Coordinator: Sean Lobo
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Aparna Bhagat
Marketing Coordinator: Marylou Dmello

First published: July 2014
Second edition: November 2017
Third Edition: April 2023
Production reference: 1290323

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

ISBN 978-1-80461-506-5

`www.packtpub.com`

We primarily want to express our gratitude to our families for their unwavering support, which helped us stay on track and complete this project.

We are deeply thankful for the time and effort the Yocto Project and the OpenEmbedded communities have invested in developing these powerful open source tools. It is an honor to be part of such a dynamic and supportive community, and we look forward to continuing our collaboration and contributing to the growth of these projects. In addition, their vibrant communities have provided us with insights, reviews, materials, and guidance, which have been instrumental in shaping the content of this book.

– Otavio Salvador and Daiane Angolini

Contributors

About the authors

Otavio Salvador is a well-known software engineer and developer with extensive embedded Linux development experience. He contributes to open source projects related to the Linux kernel and embedded systems, including system boot, device drivers, firmware, and more.

He is the chief executive officer at the Brazilian technology company O.S. Systems, a leading provider of embedded Linux development solutions and services. The company offers various services, including embedded Linux development, consulting, and support. Otavio Salvador has been a key contributor to the Yocto Project. His expertise in this area has helped make O.S. Systems a leading Yocto Project-based solutions and services provider.

Daiane Angolini is a software engineer with embedded systems and embedded Linux development expertise. In addition, she has experience in open source software development with contributions to several projects and communities, including the Yocto Project, OpenEmbedded, and the Linux kernel.

Daiane is a senior embedded software engineer at Foundries.io, a company that built a secure, open source platform for the world's connected devices and a cloud service configurable to any hardware and any cloud through embedded computing solutions. At Foundries.io, she develops and improves the software for various embedded systems, maintaining LmP and BPSs based on the ARM and x86 architectures using Yocto Project tools.

About the reviewers

Vanessa Maegima is an electronics engineer from Brazil with over six years of experience in embedded systems. She works as an embedded software engineer focusing on Customer Success at Foundries.io and previously worked as a systems engineer at NXP Semiconductors, where she met and fell in love with the embedded Linux world. She is enthusiastic about U-Boot and Linux kernel as well as microcontroller unit software development.

To my family, who always pushes me to be my best, specially to my husband Gustavo, for the daily support and love.

Khem Raj is a long-time open-source software maintainer and developer. He serves on the technical steering committee and is part of the advisory board for Yocto Project. He maintains several Yocto Project layers, including the core toolchain layer. A fellow at Comcast, Khem has helped bootstrap Reference Design Kit (RDK) project, which uses Yocto Project for its build infrastructure, which is now the standard operating platform for set-top boxes and home routers across the service provider industry. He is also a frequent speaker at events such as Embedded Linux Conference and Open Source Summit.

To my children Himangi and Vihaan, who keep me motivated to do new things, and my spouse Sweta's everlasting support, without which I would be unable to do this. Thank you.

Caio Pereira had his first contact with embedded Linux in 2005 during a demo from Blackfin Digital Signal Processor with μClinux at university. This led him to decide to work with embedded Linux. He received a bachelor's of science in electrical engineering with a specialization in telecommunications from INATEL, Brazil, in 2008. Over the last 15 years, he has participated in product development and Linux customization for different areas including broadcasting, telecommunications, home automation, smart cities, defense, and lithography machines, having experience with Field Programmable Gate Arrays and processors from different architectures. Today, he lives in the Netherlands and works for Foundries.io as a customer success engineer, helping companies worldwide to develop fast and secure products based on Linux.

I want to thank my family for their love and support and for understanding my extra night hours dedicated to study and work. I also thank all my colleagues from the companies I've worked for sharing their knowledge and experience with me; it has been fundamental to my growth. Special thanks to all those who dedicate their time to open source projects responsible for keeping the world running.

Table of Contents

8

Diving into BitBake Metadata 69

9

Developing with the Yocto Project 79

10

Debugging with the Yocto Project 91

11

Exploring External Layers 101

Preface

Linux has been consistently used in cutting-edge products, and embedded systems have been wrought in the technological portfolio of humankind.

The Yocto Project is in an optimal position to be the choice for your projects. It provides a rich set of tools to help you use most of your energy and resources in your product development instead of reinventing the wheel.

The usual tasks and requirements of embedded Linux-based products and development teams were the guideline for this book's conception. However, being written by active community members with a practical and straightforward approach is a stepping stone for both your learning curve and the product's project.

In this third edition, the book has been thoroughly reworked to incorporate the feedback from readers from previous editions and extended to facilitate the understanding of complex concepts related to the Yocto Project, in addition to being fully updated to reflect the changes made up to Yocto Project Long Term Support version 4.0 (codename Kirkstone).

Furthermore, two new chapters have been added, one regarding using QEMU to speed product development through emulation and one about Yocto Project and general project guidelines.

Who this book is for

This book is intended for engineers and enthusiasts with embedded Linux experience, willing to learn about Yocto Project's tools for evaluation, comparison, or use in a project. This book is aimed at helping you get up to speed quickly and to prevent you from getting trapped into the usual learning curve pitfalls.

What this book covers

Chapter 1, *Meeting the Yocto Project*, presents the first concepts and premises to introduce parts of the Yocto Project and its main tools.

Chapter 2, *Baking Our Poky-Based System*, introduces the environment needed for the first build.

Chapter 3, *Using Toaster to Bake an Image*, shows the user-friendly web interface that can be used as a configuration wrapper and build tool.

Chapter 4, *Meeting the BitBake Tool*, presents the BitBake metadata concepts.

Chapter 5, *Grasping the BitBake Tool*, shows how it manages the tasks and their dependencies.

Chapter 6, *Detailing the Temporary Build Directory*, details the temporary output folder of a build.

Chapter 7, *Assimilating Packaging Support*, explains the packaging mechanism used as a base to create and manage all the binary packages.

Chapter 8, *Diving into BitBake Metadata*, details the BitBake metadata language, which will be used for all the other chapters.

Chapter 9, *Developing with the Yocto Project*, demonstrates the workflow needed to obtain a development environment.

Chapter 10, *Debugging with the Yocto Project*, shows how to use Poky to generate a debug environment and how to use it.

Chapter 11, *Exploring External Layers*, explores one of the most important concepts of the Yocto Project—the flexibility of using external layers.

Chapter 12, *Creating Custom Layers*, practices the steps for layer creation.

Chapter 13, *Customizing Existing Recipes*, presents examples of how to customize existing recipes.

Chapter 14, *Achieving GPL Compliance*, summarizes the tasks and concepts involved for a copyleft compliance product.

Chapter 15, *Booting Our Custom Embedded Linux*, uses real hardware machines and the Yocto Project's tools.

Chapter 16, *Speeding Up Product Development Through Emulation – QEMU*, illustrates how QEMU can accelerate product development.

Chapter 17, *Best Practices*, discusses some Yocto Project and general project-related guidelines based on the author's experience.

To get the most out of this book

To understand this book better, it is crucial that you have some previous background about some of the topics that are not covered or are just briefly mentioned in the text, such as Git and general knowledge of Linux kernel and its basic compilation process.

To understand the big picture of the Yocto Project before going to the technical concepts detailed in this book, we recommend the open sourced booklet, *Heading for the Yocto Project*, found in the Git repository at `https://git.io/vFUiI`; the content of this booklet is intended to help newcomers to gain a better understanding of the goals of the Yocto Project and its potential uses. It provides an overview of the project before diving into the technical details of how things can be done.

A basic understanding of the use of the GNU/Linux environment and embedded Linux is required, as well as the general concepts used in development, such as compilation, debugging, deployment, and installation. In addition, some experience with shell script and Python is a bonus because these programming languages are core technologies used extensively by the Yocto Project's tools.

However, you may prefer to learn more about those topics. In that case, we recommend the book *Mastering Embedded Linux Programming - Third Edition*, ISBN-13 978-1789530384, by Chris Simmonds.

You shouldn't take any missing concepts – of those we enumerated above – as a deterrent but as something you can learn and, at the same time, practice their use with this book.

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: `https://packt.link/lbpMD`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In line 8, `BBFILE_COLLECTIONS`, we tell BitBake to create a new metadata collection called `yocto`. Next, in line 9, `BBFILE_PATTERN_yocto`, we define the rule to match all paths starting with `LAYERDIR` to identify the metadata belonging to the `yocto` collection."

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

```
$ sudo dnf install gawk make wget tar bzip2 gzip python3 unzip
perl patch diffutils diffstat git cpp gcc gcc-c++ glibc-devel
texinfo chrpath ccache perl-Data-Dumper perl-Text-ParseWords
perl-Thread-Queue perl-bignum socat python3-pexpect findutils
which file cpio python python3-pip xz python3-GitPython
python3-jinja2 SDL-devel xterm rpcgen mesa-libGL-devel perl-
FindBin perl-File-Compare perl-File-Copy perl-locale zstd lz4
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "After that, click the **Image recipes** tab to choose the image you want to build."

> **Tips or important notes**
> Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Embedded Linux Development Using Yocto Project*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

   `https://packt.link/free-ebook/9781804615065`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

1
Meeting the Yocto Project

This chapter introduces you to the **Yocto Project**. The main concepts of the project discussed here are constantly used throughout the book. In addition, we will briefly discuss the history of the Yocto Project, OpenEmbedded, Poky, BitBake, metadata, and versioning schema. So, fasten your seat belt, and welcome aboard!

What is the Yocto Project?

The Yocto Project is a Linux Foundation workgroup and is defined as follows:

> *The Yocto Project is an open source collaboration project that helps developers create custom Linux-based systems that are designed for embedded products regardless of the product's hardware architecture. Yocto Project provides a flexible toolset and a development environment that allows embedded device developers across the world to collaborate through shared technologies, software stacks, configurations, and best practices used to create these tailored Linux images.*
>
> *Thousands of developers worldwide have discovered that Yocto Project provides advantages in both systems and applications development, archival and management benefits, and customizations used for speed, footprint, and memory utilization. The project is a standard when it comes to delivering embedded software stacks. The project allows software customizations and build interchange for multiple hardware platforms as well as software stacks that can be maintained and scaled.*
>
> *- Yocto Project Overview and Concepts Manual*

The Yocto Project is an open source collaboration project. It supplies templates, tools, and methods to help us create custom Linux-based systems for embedded products, regardless of the hardware architecture. It can generate tailored Linux distributions based on the `glibc` and `musl` C standard libraries and the **Real-Time Operating System** (**RTOS**) toolchains for bare-metal development, as done by the Zephyr Project.

Being managed by a Linux Foundation member, the project stays independent of the member organizations, which participate in many ways and supply resources to the project.

It was founded in 2010 as a collaboration of many hardware manufacturers, open source operating systems, vendors, and electronics companies to reduce duplication of work and supply resources and information catering to new and experienced users. Among these resources is OpenEmbedded Core, the core system component provided by the OpenEmbedded project.

The Yocto Project aggregates several companies, communities, projects, and tools with the same purpose – to build Linux-based embedded products. These stakeholders are in the same boat, driven by their community needs to work together.

Delineating the Yocto Project

To ease our understanding of the duties and outcomes of the Yocto Project, we can use the analogy of a computing machine. The input is a set of data that describes what we want, that is, our specification. As an output, we have the desired Linux-based embedded product.

The output is composed of the pieces of the operating system. It encompasses the Linux kernel, bootloader, and the root filesystem (`rootfs`) bundled and organized to work together.

The Yocto Project's tools are present in all intermediary steps to produce the resultant `rootfs` bundle and other deliverables. The previously built software components are reused across builds – applications, libraries, or any software component.

When reuse is not possible, the software components are built in the correct order and with the desired configuration, including fetching the required source code from their respective repositories, such as The Linux Kernel Archives (`www.kernel.org`), GitHub, BitBucket, and GitLab.

The Yocto Project's tools prepare its build environment, utilities, and toolchains, reducing the host software dependency. The utilities, versions, and configuration options are independent of the host Linux distribution, minimizing the number of host utilities to rely on while producing the same result. A subtle but essential implication benefit is the considerable increase in determinism, reduced build host dependencies, but increased first-time builds.

BitBake and OpenEmbedded Core are under the OpenEmbedded project umbrella, while some projects, such as Poky, are under the Yocto Project umbrella. They are all complementary and play specific roles in the system. We will understand exactly how they work together in this chapter and throughout this book.

The alliance of the OpenEmbedded project and the Yocto Project

The OpenEmbedded project was created around January 2003 when some core developers from the **OpenZaurus** project started to work with the new build system. Since its beginning, the OpenEmbedded build system has been a task scheduler inspired and based on the **Gentoo Portage** package system named BitBake. As a result, the project quickly grew its software collection and the supported machine list.

Due to chaotic and uncoordinated development, it was challenging to use OpenEmbedded in products that demand a more stable and polished code base, which is how Poky distribution was born. Poky started as a subset of the OpenEmbedded build system, and had a more polished and stable code base across a limited set of architectures. Additionally, its reduced size allowed Poky to develop highlighting technologies, such as IDE plugins and **Quick Emulator** (**QEMU**) integration, which are still in use.

The Yocto Project and OpenEmbedded project consolidated their efforts on a core build system called OpenEmbedded Core. It uses the best of both Poky and OpenEmbedded, emphasizing the increased use of additional components, metadata, and subsets. Around November 2010, the Linux Foundation announced that the Yocto Project would continue this work under a Linux Foundation-sponsored project.

Understanding Poky

Poky is the default Yocto Project reference distribution, which uses OpenEmbedded build system technology. It is composed of a collection of tools, configuration files, and recipe data (known as metadata). It is platform-independent and performs cross-compiling using the BitBake tool, OpenEmbedded Core, and a default set of metadata, as shown in the following figure. In addition, it provides the mechanism to build and combine thousands of distributed open source projects to form a fully customizable, complete, and coherent Linux software stack.

Poky's main objective is to provide all the features an embedded developer needs.

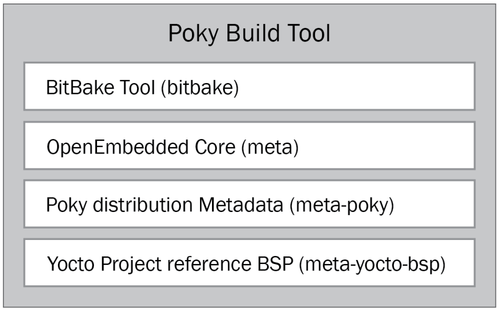


Figure 1.1 – Poky main components

BitBake

BitBake is a task scheduler and execution system that parses Python and Shell Script code. The code that is parsed generates and runs tasks, which are a set of steps ordered per the code's dependencies.

BitBake evaluates all available metadata, managing dynamic variable expansion, dependencies, and code generation. In addition, it keeps track of all tasks to ensure their completion, maximizing the use of processing resources to reduce build time and predictability. The development of BitBake happens in the `https://lists.openembedded.org/g/bitbake-devel` mailing list, and the source code is in the `bitbake` subdirectory of Poky.

OpenEmbedded Core

The OpenEmbedded Core metadata collection provides the engine of the Poky build system. It provides the core features and aims to be generic and as lean as possible. It supports seven different processor architectures (ARM, ARM64, x86, x86-64, PowerPC, PowerPC 64, MIPS, MIPS64, RISC-V32, and RISC-V 64), only supporting platforms to be emulated by QEMU.

The development is centralized in the `https://lists.openembedded.org/g/openembedded-core` (`mailto:openembedded-core@lists.openembedded.org`) mailing list and houses its metadata inside the `meta` subdirectory of Poky.

Metadata

The metadata includes recipes and configuration files. It is composed of a mix of Python and Shell Script text files, providing a tremendously flexible tool. Poky uses this to extend OpenEmbedded Core and includes two different layers, which are other metadata subsets, shown as follows:

- `meta-poky`: This layer provides the default and supported distribution policies, visual branding, and metadata tracking information (maintainers, upstream status, and so on). This is to serve as a curated template that could be used by distribution builders to seed their custom distribution.
- `meta-yocto-bsp`: This provides the **Board Support Package** (**BSP**) used as the reference hardware for the Yocto Project development and **Quality Assurance** (**QA**) process.

Chapter 9, *Developing with Yocto Project*, explores the metadata in more detail and serves as a reference when we write our recipes.

The Yocto Project releases

The Yocto Project has a release every six months, in April and October. This release cycle ensures continuous development flow while providing points of increased testing and focus on stability. A release becomes a **Stable** or a **Long-Term Support** (**LTS**) release whenever a release is ready.

The support period differs significantly between the stable and LTS releases. The support for the stable release is for 7 months, offering 1 month of overlapped support for every stable release. The LTS release has a minimal support period of 2 years, optionally extended. After the official support period ends, it moves to **Community** support and finally reaches **End Of Life** (**EOL**).

When the official release support period ends, a release can be Community support if a community member steps in to become the community maintainer. Finally, a release turns EOL when there is no change in the source code by 2 months, or the community maintainer is no longer an active member.

The following diagram shows the two release cycles:

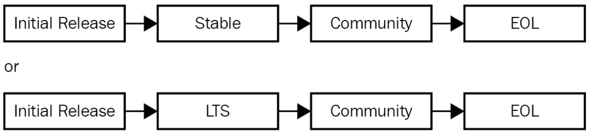


Figure 1.2 – Stable or LTS release cycles

Table 1.1 provides the Yocto Project version, codename, release date, and current support level, which can be seen as follows. The updated table is available at `https://wiki.yoctoproject.org/wiki/Releases`:

Codename	**Version**	**Release Date**	**Support Level**
Mickledore	4.2	April 2023	Future (until October 2023)
Langdale	4.1	October 2022	Stable (until May 2023)
Kirkstone	4.0	May 2022	LTS (minimum April 2024)
Honister	3.4	October 2021	EOL
Hardknott	3.3	April 2021	EOL
Gatesgarth	3.2	Oct 2020	EOL
Dunfell	3.1	April 2020	LTS (until April 2024)
Zeus	3.0	October 2019	EOL
Warrior	2.7	April 2019	EOL
Thud	2.6	Nov 2018	EOL
Sumo	2.5	April 2018	EOL
Rocko	2.4	Oct 2017	EOL
Pyro	2.3	May 2017	EOL
Morty	2.2	Nov 2016	EOL
Krogoth	2.1	Apr 2016	EOL
Jethro	2.0	Nov 2015	EOL
Fido	1.8	Apr 2015	EOL
Dizzy	1.7	Oct 2014	EOL
Daisy	1.6	Apr 2014	EOL
Dora	1.5	Oct 2013	EOL
Dylan	1.4	Apr 2013	EOL
Danny	1.3	Oct 2012	EOL

Codename	Version	Release Date	Support Level
Denzil	1.2	Apr 2012	EOL
Edison	1.1	Oct 2011	EOL
Bernard	1.0	Apr 2011	EOL
Laverne	0.9	Oct 2010	EOL

Table 1.1 – List of Yocto Project versions

Summary

This chapter provided an overview of how the OpenEmbedded project is related to the Yocto Project, the components that form Poky, and how the project began. The next chapter will introduce the Poky workflow with steps to download, configure, and prepare the Poky build environment and how to have the first image built and running using QEMU.

2

Baking Our First Poky-Based System

Let's get our hands dirty! In this chapter, we will understand the basic concepts involved in the Poky workflow. We will cover the steps to download, configure, and prepare the Poky build environment and bake something usable. The steps covered here are common for testing and development. They will give us some experience using Poky and a taste of its capabilities.

Preparing the build host system

This section describes how to prepare Windows and Linux distribution host systems. Although we will describe the Windows steps, we will focus on using a Linux distribution host system.

> **Tip**
>
> The use of macOS as a host system is possible. Still, it involves using the **CROss PlatformS** (**CROPS**) framework, which leverages Docker, allowing the use of foreign operating systems, including macOS. For more information, you can refer to the *Setting Up to Use CROss PlatformS (CROPS)* section from the *Yocto Project Development Tasks Manual* (`https://docs.yoctoproject.org/4.0.4/dev-manual/start.html#setting-up-to-use-cross-platforms-crops`).

Next, we will provide the necessary information to start the build host system preparation.

Using Windows Subsystem for Linux (WSLv2)

You can set up a Linux distribution on Windows if you are a Windows user. WSLv2 is only available for Windows 10+ builds greater than 18917. WSLv2 allows development using the Yocto Project. You can install the Linux distribution from the Microsoft Store.

Please refer to the *Setting Up to Use Windows Subsystem For Linux* session (`https://docs.yoctoproject.org/4.0.4/dev-manual/start.html#setting-up-to-use-windows-subsystem-for-linux-wslv2`) from the *Yocto Project Development Tasks Manual* (`https://docs.yoctoproject.org/4.0.4/dev-manual/index.html`). Once you have WSLv2 set up, you can follow the next sections as if you were running on a native Linux machine.

Preparing a Linux-based system

The process needed to set up our host system depends on the Linux distribution we use. Poky has a set of supported Linux distributions. Let's suppose we are new to embedded Linux development. In that case, it is advisable to use one of the supported Linux distributions to avoid wasting time debugging issues related to the host system support.

If you use the current release of one of the following distributions, you should be good to start using the Yocto Project on your machine:

- Ubuntu
- Fedora
- CentOS
- AlmaLinux
- Debian
- OpenSUSE Leap

To confirm whether your version is supported, it is advisable to check the official documentation online in the *Required Packages for the Build Host* section (`https://docs.yoctoproject.org/4.0.4/ref-manual/system-requirements.html#required-packages-for-the-build-host`).

If your preferred distribution is not in the preceding list, it doesn't mean it is not possible to use Poky on it. Your host development system must meet some specific versions for Git, tar, Python, and GCC. Your Linux distributions should provide compatible versions of those base tools. However, there is a chance that your host development system does not meet all these requirements. In that case, you can resolve this by installing a **buildtools** tarball that contains these tools, as detailed in *Required Git, tar, Python, and GCC Versions* (`https://docs.yoctoproject.org/4.0.4/ref-manual/system-requirements.html#required-git-tar-python-and-gcc-versions`).

We must install a few packages on the host system. This book provides instructions for **Debian** and **Fedora**, our preferred distributions, which we will look at next. The set of packages for other supported distributions can be found in the *Yocto Project Reference Manual* (`https://docs.yoctoproject.org/4.0.4/ref-manual/system-requirements.html#required-packages-for-the-build-host`).

Debian-based distribution

To install the necessary packages for a headless host system, run the following command:

```
$ sudo apt install gawk wget git diffstat unzip texinfo gcc
build-essential chrpath socat cpio python3 python3-pip python3-
pexpect xz-utils debianutils iputils-ping python3-git python3-
jinja2 libegl1-mesa libsdl1.2-dev pylint3 xterm python3-subunit
mesa-common-dev zstd liblz4-tool
```

Fedora

To install the needed packages for a headless host system, run the following command:

```
$ sudo dnf install gawk make wget tar bzip2 gzip python3 unzip
perl patch diffutils diffstat git cpp gcc gcc-c++ glibc-devel
texinfo chrpath ccache perl-Data-Dumper perl-Text-ParseWords
perl-Thread-Queue perl-bignum socat python3-pexpect findutils
which file cpio python python3-pip xz python3-GitPython
python3-jinja2 SDL-devel xterm rpcgen mesa-libGL-devel perl-
FindBin perl-File-Compare perl-File-Copy perl-locale zstd lz4
```

Downloading the Poky source code

After we have installed the required packages on our development host system, we can download the current LTS version (at the time of writing) of Poky source code using Git, with the following command:

```
$ git clone https://git.yoctoproject.org/poky -b kirkstone
```

> **Tip**
>
> Learn more about Git at `https://git-scm.com`.

After the download process is complete, we should have the following contents inside the `poky` directory:

```
$ ls -l
total 84
drwxrwxr-x  6 user user  4096 set  7 12:16 bitbake
drwxrwxr-x  4 user user  4096 set  7 12:16 contrib
drwxrwxr-x 19 user user  4096 set  7 12:16 documentation
-rw-rw-r--  1 user user   834 set  7 12:16 LICENSE
-rw-rw-r--  1 user user 15394 set  7 12:16 LICENSE.GPL-2.0-only
-rw-rw-r--  1 user user  1286 set  7 12:16 LICENSE.MIT
-rw-rw-r--  1 user user  2202 set  7 12:16 MAINTAINERS.md
-rw-rw-r--  1 user user  1222 set  7 12:16 Makefile
-rw-rw-r--  1 user user   244 set  7 12:16 MEMORIAM
drwxrwxr-x 20 user user  4096 set  7 12:16 meta
drwxrwxr-x  5 user user  4096 set  7 12:16 meta-poky
drwxrwxr-x  9 user user  4096 set  7 12:16 meta-selftest
drwxrwxr-x  8 user user  4096 set  7 12:16 meta-skeleton
drwxrwxr-x  8 user user  4096 set  7 12:16 meta-yocto-bsp
-rwxrwxr-x  1 user user  1297 set  7 12:16 oe-init-build-env
lrwxrwxrwx  1 user user    33 set  7 12:16 README.hardware.md -> meta-yocto-bsp/README.hardware.md
lrwxrwxrwx  1 user user    14 set  7 12:16 README.md -> README.poky.md
-rw-rw-r--  1 user user   791 set  7 12:16 README.OE-Core.md
lrwxrwxrwx  1 user user    24 set  7 12:16 README.poky.md -> meta-poky/README.poky.md
-rw-rw-r--  1 user user   529 set  7 12:16 README.qemu.md
drwxrwxr-x 10 user user  4096 set  7 12:16 scripts
```

Figure 2.1 – The content of the poky directory after downloading

> **Note**
>
> The examples and code presented in this and subsequent chapters use the Yocto Project 4.0 release (codenamed **Kirkstone**) as a reference.

Preparing the build environment

Inside the `poky` directory exists a script named `oe-init-build-env`, which sets up the building environment. But first, the script must be run-sourced (not executed) as follows:

```
$ source oe-init-build-env [build-directory]
```

Here, `[build-directory]` is an optional parameter for the name of the directory where the environment is configured. If it is empty, it defaults to `build`. The `[build-directory]` parameter is the place where we perform the builds.

The output from `source oe-init-build-env build` displays some important configurations such as the file location, some project URLs, and some common targets, such as available images. The following figure shows an output example:

```
$ source oe-init-build-env build
You had no conf/local.conf file. This configuration file has therefore been
created for you from /home/user/yocto/poky/meta-poky/conf/local.conf.sample
You may wish to edit it to, for example, select a different MACHINE (target
hardware). See conf/local.conf for more information as common configuration
options are commented.

You had no conf/bblayers.conf file. This configuration file has therefore been
created for you from /home/user/yocto/poky/meta-poky/conf/bblayers.conf.sample
To add additional metadata layers into your configuration please add entries
to conf/bblayers.conf.

The Yocto Project has extensive documentation about OE including a reference
manual which can be found at:
    https://docs.yoctoproject.org

For more information about OpenEmbedded see the website:
    https://www.openembedded.org/


### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
    core-image-minimal
    core-image-full-cmdline
    core-image-sato
    core-image-weston
    meta-toolchain
    meta-ide-support

You can also run generated qemu images with a command like 'runqemu qemux86'

Other commonly useful commands are:
 - 'devtool' and 'recipetool' handle common recipe tasks
 - 'bitbake-layers' handles common layer tasks
 - 'oe-pkgdata-util' handles common target package tasks
```

Figure 2.2 – Output of the source oe-init-build-env build command

It is very convenient to use different build directories. We can work on separate projects in parallel or experimental setups without affecting our other builds.

Note

Throughout the book, we will use `build` as the build directory. When we need to point to a file inside the build directory, we will adopt the same convention – for example, `build/conf/local.conf`.

Knowing the local.conf file

When we initialize a build environment, it creates a file called `build/conf/local.conf`. This config file is powerful, since it can configure almost every aspect of the build process. We can set the target machine and the toolchain host architecture to be used for a custom cross-toolchain, optimize options for maximum build time reduction, and so on. The comments inside the `build/conf/local.conf` file are excellent documentation and a reference of the possible variables and their defaults. The minimal set of variables that we probably want to change from the default is the following:

```
MACHINE ??= "qemux86-64"
```

The `MACHINE` variable is where we determine the target machine we wish to build. At the time of writing, Poky supports the following machines in its reference BSP:

- `beaglebone-yocto`: This is BeagleBone, which is the reference platform for 32-bit ARM
- `genericx86`: This is generic support for 32-bit x86-based machines
- `genericx86-64`: This is generic support for 64-bit x86-based machines
- `edgerouter`: This is EdgeRouter Lite, which is the reference platform for 64-bit MIPS

The machines are made available by a layer called `meta-yocto-bsp`. Besides these machines, OpenEmbedded Core, inside the `meta` directory, also provides support for the following Quick Emulation (QEMU) machines:

- `qemuarm`: This is the QEMU ARMv7 emulation
- `qemuarmv5`: This is the QEMU ARMv5 emulation
- `qemuarm64`: This is the QEMU ARMv8 emulation
- `qemumips`: This is the QEMU MIPS emulation
- `qemumips64`: This is the QEMU MIPS64 emulation
- `qemuppc`: This is the QEMU PowerPC emulation
- `qemuppc64`: This is the QEMU PowerPC 64 emulation
- `qemux86-64`: This is the QEMU x86-64 emulation
- `qemux86`: This is the QEMU x86 emulation
- `qemuriscv32`: This is the QEMU RISC-V 32 emulation
- `qemuriscv64`: This is the QEMU RISC-V 64 emulation

Extra BSP layers available from several vendors provide support for other machines. The process of using an extra BSP layer is shown in *Chapter 11*, *Exploring External Layers*.

> **Note**
>
> The `local.conf` file is a convenient way to override several global default configurations throughout the Yocto Project's tools. Essentially, we can change or set any variable – for example, adding additional packages to an image file. Changing the `build/conf/local.conf` file is convenient; however, the source code management system usually does not track temporary changes in this directory.

The `build/conf/local.conf` file can set several variables. It is worth taking some time and reading through the file comments that are generated to get a general idea of what variables can be set.

Building a target image

Poky provides several predesigned image recipes we can use to build our binary image. We can check the list of available images by running the following command from the `poky` directory:

```
$ ls meta*/recipes*/*images/*.bb
```

All the recipes provide images that are a set of unpacked and configured packages, generating a filesystem that we can use with hardware or one of the supported QEMU machines.

Next, we can see the list of most commonly used images:

- `core-image-minimal`: This is a small image allowing a device to boot. It is handy for kernel and bootloader tests and development.
- `core-image-base`: This console-only image provides basic hardware support for the target device.
- `core-image-weston`: This image provides the Wayland protocol libraries and the reference Weston compositor.
- `core-image-x11`: This is a basic X11 image with a terminal.
- `core-image-sato`: This is an image with Sato support and a mobile environment for mobile devices that use X11. It provides applications such as a terminal, editor, file manager, media player, and so on.
- `core-image-full-cmdline`: A console-only image with more full-featured Linux system functionality installed.

There are other reference images available from the community. Several images support features, such as Real Time, `initramfs`, and MTD (flash tools). It is good to check the source code or the *Yocto Project Reference Manual* (`https://docs.yoctoproject.org/4.0.4/ref-manual/index.html`) for the complete and updated list.

The process of building an image for a target is straightforward. But first, we need to set up the build environment using `source oe-init-build-env [build-directory]` before using BitBake. To build the image, we can use the template in the following command:

```
$ bitbake <recipe name>
```

Figure 2.3 – How to build a recipe using BitBake

> **Note**
>
> We will use `MACHINE = "qemux86-64"` in the following examples. You can set it in `build/conf/local.conf` accordingly.

For example, to build `core-image-full-cmdline`, run the following command:

```
$ bitbake core-image-full-cmdline
```

The Poky build looks like the following figure:

```
$ bitbake core-image-full-cmdline
Loading cache: 100% |##############################################################| Time: 0:00:00
Loaded 1641 entries from dependency cache.
Parsing recipes: 100% |############################################################| Time: 0:00:00
Parsing of 882 .bb files complete (881 cached, 1 parsed). 1641 targets, 44 skipped, 0 masked, 0
errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "2.0.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "x86_64-poky-linux"
MACHINE              = "qemux86-64"
DISTRO               = "poky"
DISTRO_VERSION       = "4.0.4"
TUNE_FEATURES        = "m64 core2"
TARGET_FPU           = ""
meta
meta-poky
meta-yocto-bsp       = "kirkstone:e81e703fb6fd028f5d01488a62dcfacbda16aa9e"

Initialising tasks: 100%
|##################################################################################| Time: 0:00:02
Sstate summary: Wanted 452 Local 449 Mirrors 0 Missed 3 Current 1172 (99% match, 99% complete)
Removing 2 stale sstate objects for arch qemux86_64: 100%
|##################################################################################| Time: 0:00:00
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 4105 tasks of which 4095 didn't need to be rerun and all succeeded.
NOTE: Writing buildhistory
NOTE: Writing buildhistory took: 13 seconds
```

Figure 2.4 – The result of bitbake core-image-full-cmdline

Running images in QEMU

We can use hardware emulation to speed up the development process, as it enables a test run without involving any actual hardware. Fortunately, most projects have only a tiny portion that is hardware-dependent.

QEMU is a free, open source software package that performs hardware virtualization. QEMU-based machines allow testing and development without real hardware. ARMv5, ARMv7, ARMv8, MIPS, MIPS64, PowerPC, PowerPC 64, RISC-V 32, RISC-V 64, x86, and x86-64 emulations are currently supported. We will go into more detail about QEMU usage in *sw*, *Speeding Up Product Development through Emulation – QEMU*.

OpenEmbedded Core provides the `runqemu` script tool, which is a wrapper to make use of QEMU easier. The way to run the script tool is as follows:

```
$ runqemu <machine> <zimage> <filesystem>
```

Here, `<machine>` is the machine/architecture to be used as `qemux86-64`, or any other supported machine. Also, `<zimage>` is the path to a kernel (for example, `bzImage-qemux86-64.bin`). Finally, `<filesystem>` is the path to an `ext4` image (for example, `filesystem-qemux86-64.ext4`) or an NFS directory. All parameters in the preceding call to `runqemu` `<zimage>` and `<filesystem>` are optional. Just running `runqemu` is sufficient to launch the image in the shell where the build environment is set, as it will automatically pick up the default settings from building the environment.

So, for example, if we run `runqemu qemux86-64 core-image-full-cmdline`, we can see something similar to that shown in the following screenshot:

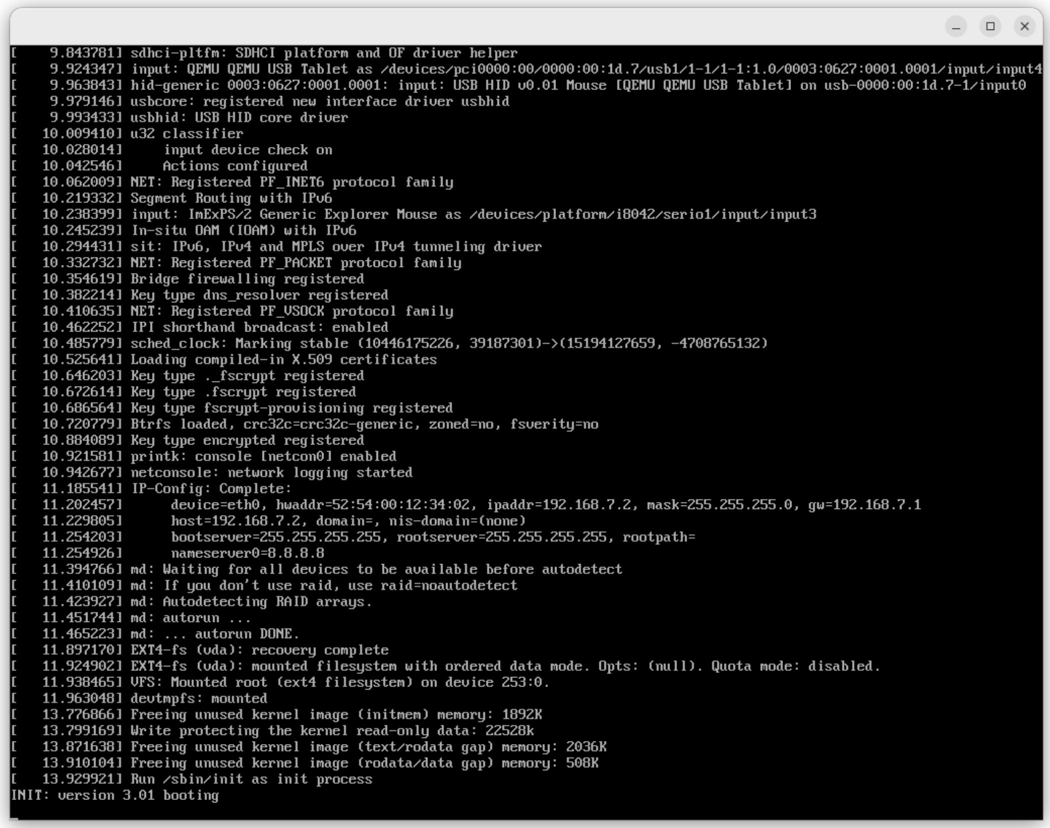


Figure 2.5 – The QEMU screen during the Linux kernel boot

After finishing booting Linux, you will see a login prompt, as shown in *Figure 2.6*:

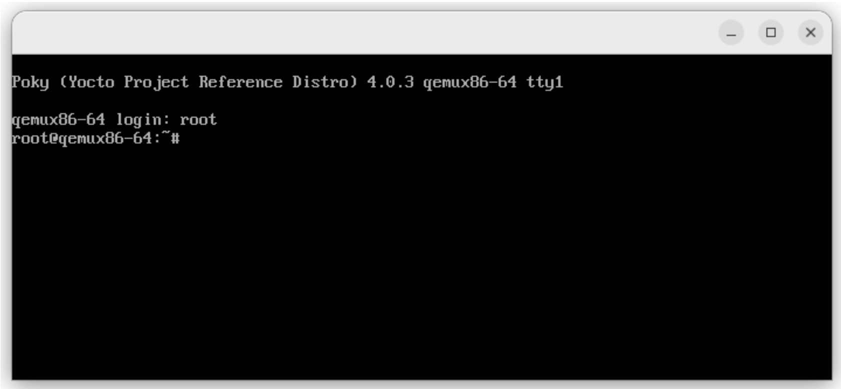


Figure 2.6 – The QEMU screen during user login

We can log in to the `root` account using an empty password. The system behaves as a regular machine, even when executed inside the QEMU. The process to deploy an image in real hardware varies, depending on the type of storage used, the bootloader, and so on. However, the process of generating the image is the same. We explore how to build and run an image in real hardware in *Chapter 15*, *Booting Our Custom Embedded Linux*.

Summary

In this chapter, we learned the steps needed to set up Poky and get our first image built. Then, we ran that image using `runqemu`, which gave us a good overview of the available capabilities. In the next chapter, we will introduce Toaster, a human-friendly interface for BitBake. We will use it to build an image and customize it further.

3

Using Toaster to Bake an Image

Now that we know how to build an image using BitBake within Poky, we will learn how to do the same using Toaster. We are going to focus on the most straightforward usage of Toaster and also cover what else it can do so that you know about its capabilities.

What is Toaster?

Toaster is a web interface to configure and run builds. It communicates with the BitBake and Poky build system to manage and gather information about the builds, packages, and images.

There are two ways to use Toaster:

- **Locally**: We can run Toaster as a local instance, suitable for single-user development, providing a graphical interface to the BitBake command lines and some build information.
- **Hosted**: This is suitable for multiple users. The Toaster servers build and store the users' artifacts. Its components can be spread across several machines when using a hosted instance.

In this chapter, we will use Toaster as a local instance. However, if you want to use it as a hosted instance, please visit the following website for instructions – *Toaster Manual* (`https://docs.yoctoproject.org/4.0.4/toaster-manual/index.html`).

> **Note**
> Bear in mind that every hosted service requires attention to its security. Think about this before using a hosted instance.

Installing Toaster

Toaster uses the Python Django framework. The easiest way to install it is by using Python's `pip` utility. We already installed this when configuring our host machine in *Chapter 2*, *Baking Our Poky-Based System*. We can now install the rest of Toaster's requirements inside the Poky source directory by running the following command:

```
$ pip3 install --user -r bitbake/toaster-requirements.txt
```

Starting Toaster

Once we have installed Toaster's requirements, we are ready to start its server. To do this, we should go to Poky's directory and run the following commands:

```
$ source oe-init-build-env
$ source toaster start
```

The commands take some time to finish. When everything is set up, the web server is started. The result is shown in the following figure.

```
Build configuration saved
Loading default settings
Installed 7 object(s) from 1 fixture(s)
Loading poky configuration
Installed 44 object(s) from 1 fixture(s)
Importing custom settings if present
NOTE: optional fixture 'custom' not found

Fetching information from the layer index, please wait.
You can re-update any time later by running bitbake/lib/toaster/manage.py lsupdates

2022-09-08 11:01:55,506 INFO Fetching metadata for kirkstone HEAD master honister hardknott
\2022-09-08 11:03:04,609 INFO Processing releases
Updating Releases 100%
2022-09-08 11:03:04,611 INFO Processing layers
Updating layers 100%
2022-09-08 11:03:05,392 INFO Processing layer versions
Updating layer versions 100%
2022-09-08 11:03:07,865 INFO Processing layer version dependencies
2022-09-08 11:03:08,127 WARNING Cannot find layer version
(ls:<orm.management.commands.lsupdates.Command object at 0x7fd7b35d80a0>),up_id:76 lv:64 meta-mel
(master)
2022-09-08 11:03:08,146 WARNING Cannot find layer version
(ls:<orm.management.commands.lsupdates.Command object at 0x7fd7b35d80a0>),up_id:76 lv:68 meta-baryon
(master)
2022-09-08 11:03:08,192 WARNING Cannot find layer version
(ls:<orm.management.commands.lsupdates.Command object at 0x7fd7b35d80a0>),up_id:76 lv:79 meta-
netmodule (master)
(...)
2022-09-08 11:03:09,077 WARNING Cannot find layer version
(ls:<orm.management.commands.lsupdates.Command object at 0x7fd7b35d80a0>),up_id:76 lv:259 meta-meson
(master)
2022-09-08 11:03:09,907 WARNING Cannot find layer version
(ls:<orm.management.commands.lsupdates.Command object at 0x7fd7b35d80a0>),up_id:76 lv:410 meta-
mediatek (master)
2022-09-08 11:03:10,208 WARNING Cannot find layer version
(ls:<orm.management.commands.lsupdates.Command object at 0x7fd7b35d80a0>),up_id:345 lv:677 meta-intel
(hardknott)
Updating Layer version dependencies 100%
2022-09-08 11:03:12,974 INFO Processing distro information
Updating distros 100%
2022-09-08 11:03:13,662 INFO Processing machine information
Updating machines 100%
2022-09-08 11:03:19,272 INFO Processing recipe information
Updating recipes 100%
Starting webserver...
Toaster development webserver started at http://localhost:8000

You can now run 'bitbake <target>' on the command line and monitor your build in Toaster.
You can also use a Toaster project to configure and run a build.

Successful start.
```

Figure 3.1 – The result of the source toaster startup

To access the Toaster web interface, open your favorite browser and enter the following:

```
http://127.0.0.1:8000
```

> **Note**
>
> By default, Toaster starts on port `8000`. The `webport` parameter lets you use a different port – for example, `$ source toaster start webport=8400`.

Next, we can see the starting page of Toaster:

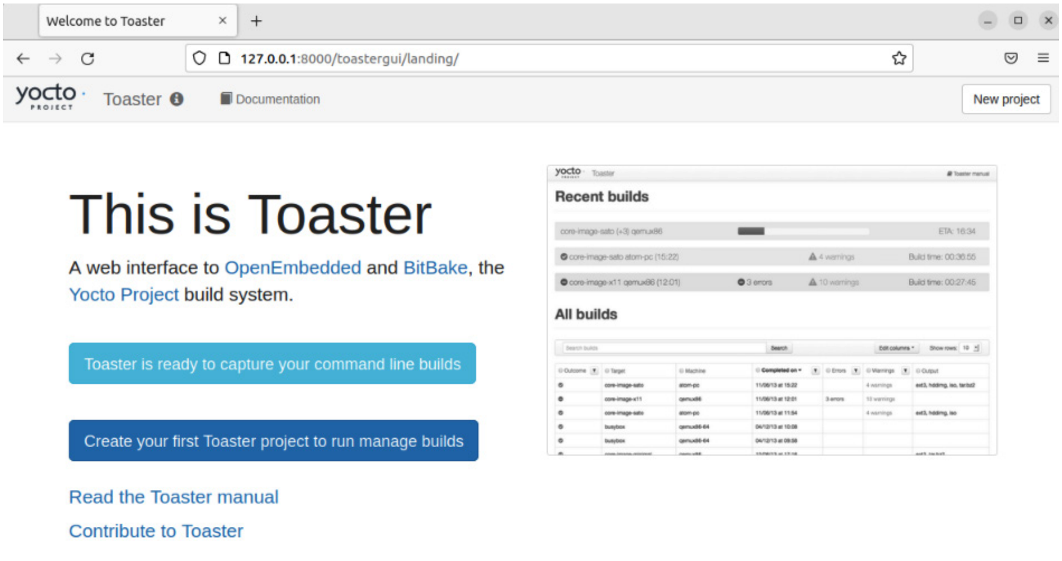


Figure 3.2 – The Toaster welcome page

Building an image for QEMU

Following the same steps used in *Chapter 2, Baking Our Poky-Based System*, we will build an image of the QEMU x86-64 emulation.

Since we currently don't have a project, a collection of configurations and builds, we need to start one. Create a project name and choose the target release, as shown in the following screenshot:

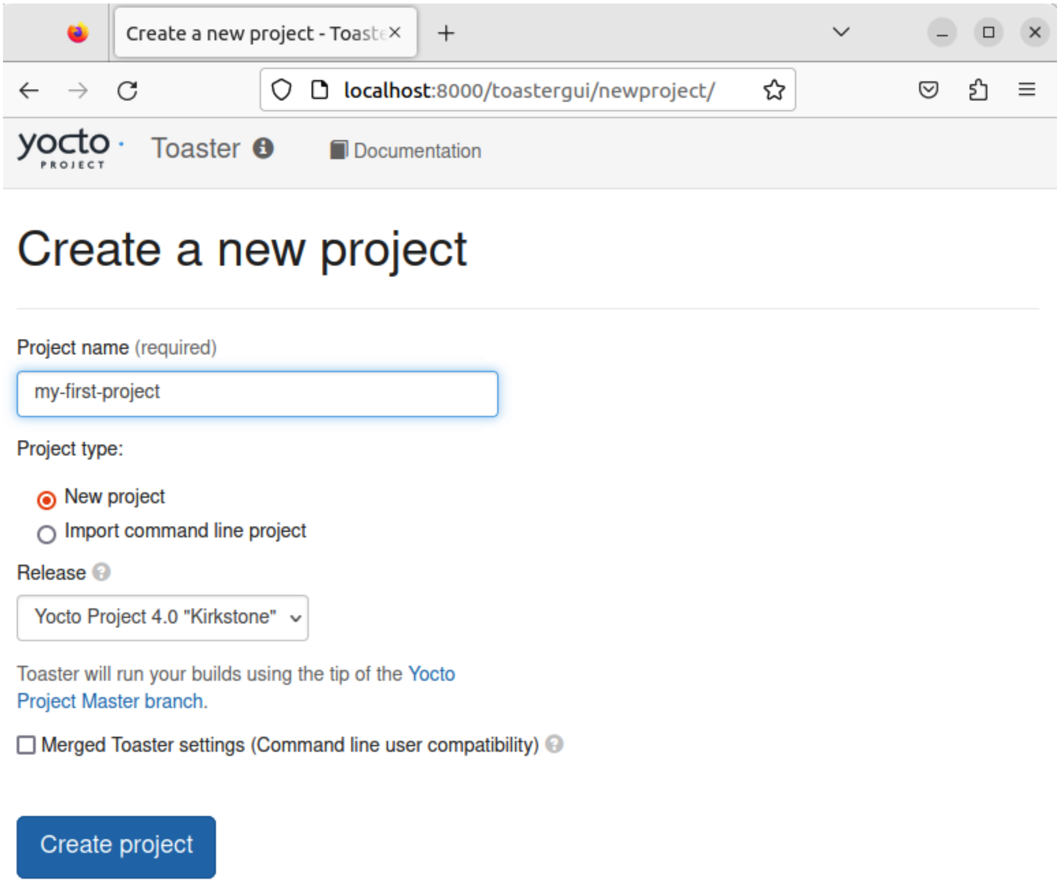


Figure 3.3 – Creating a new project with Toaster

After creating **my-first-project**, we can see the main project screen, as shown in the following screenshot:

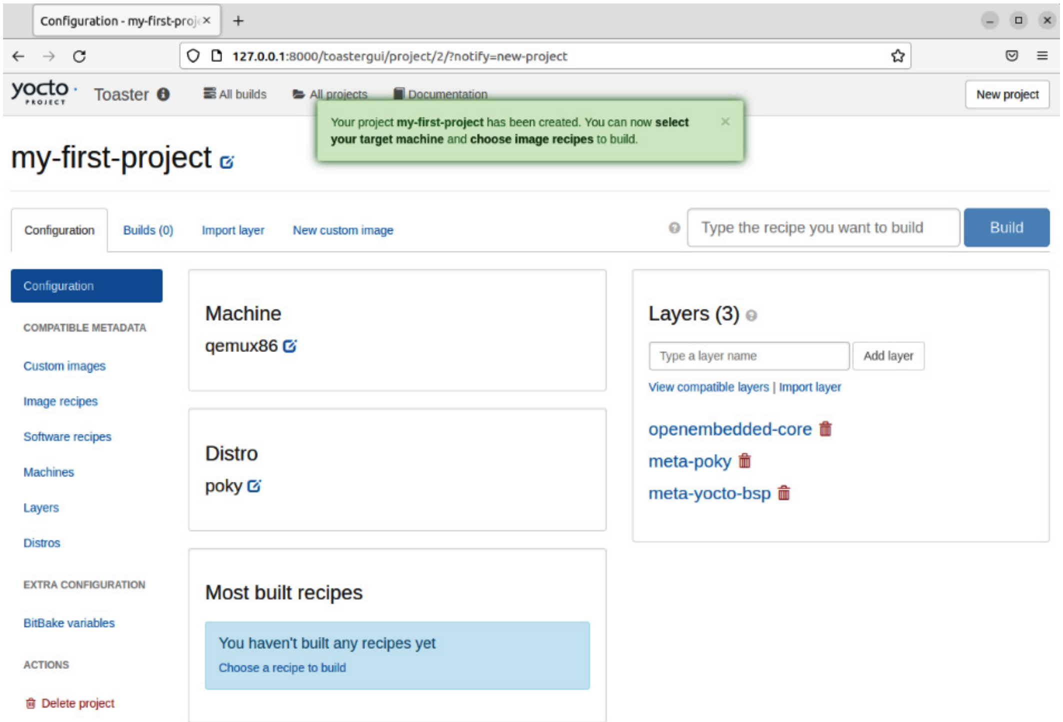


Figure 3.4 – The first page of the project

While on the **Configuration** tab, go to **Machine** and change the target machine to `qemux86-64`:

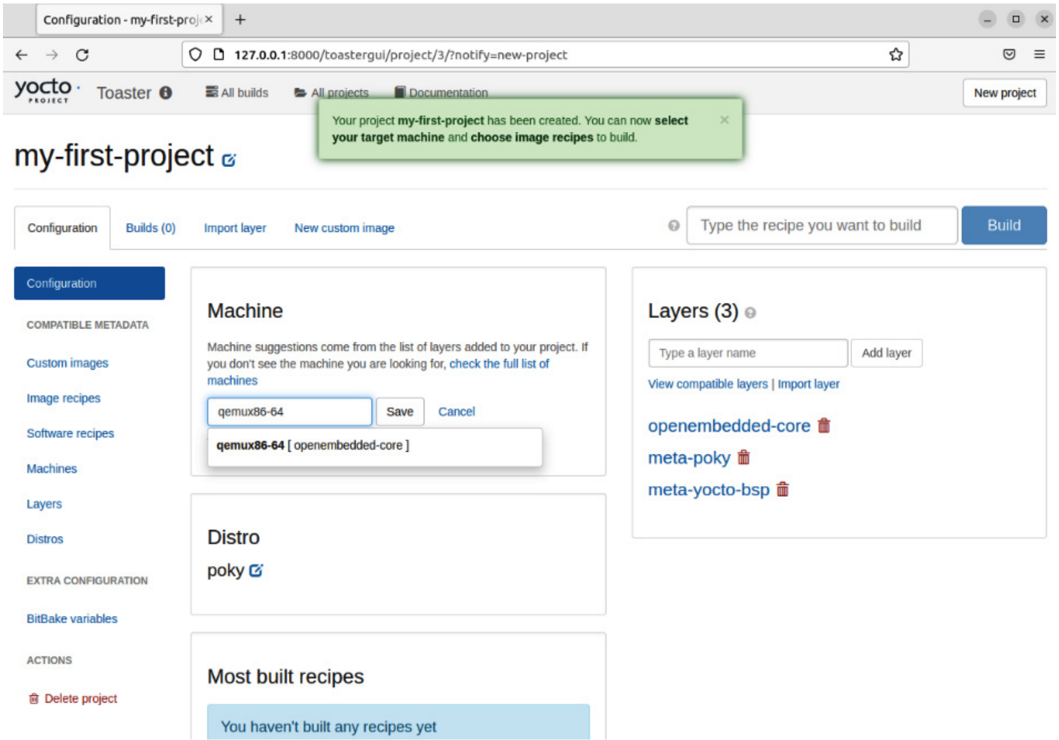


Figure 3.5 – How to choose the target machine

After that, click the **Image recipes** tab to choose the image you want to build. In this example, as used in *Chapter 2*, *Baking Our Poky-Based System*, we can build `core-image-full-cmdline`:

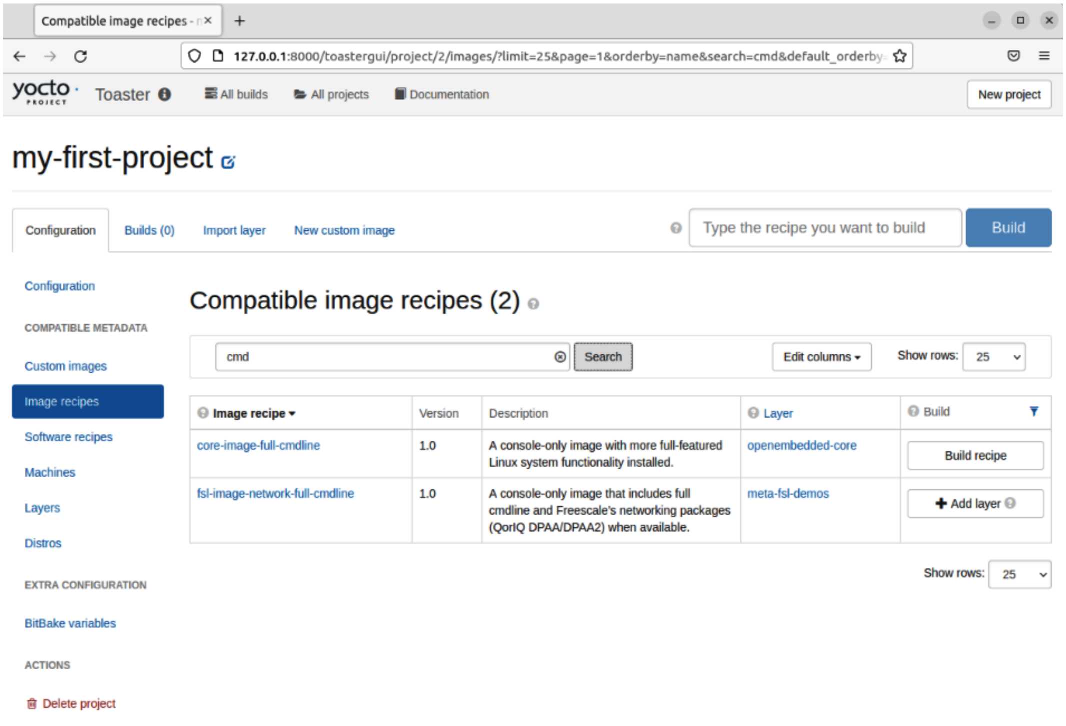


Figure 3.6 – How to find an image using Search

The following screenshot shows the build process:

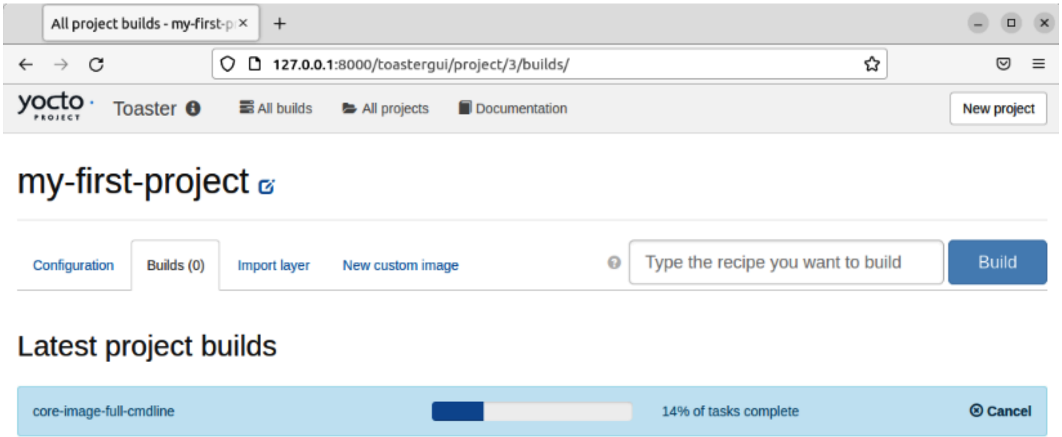


Figure 3.7 – Toaster during the image build

The build process takes some time, but after that, we can see the built image along with some statistics, as shown in the following screenshot:

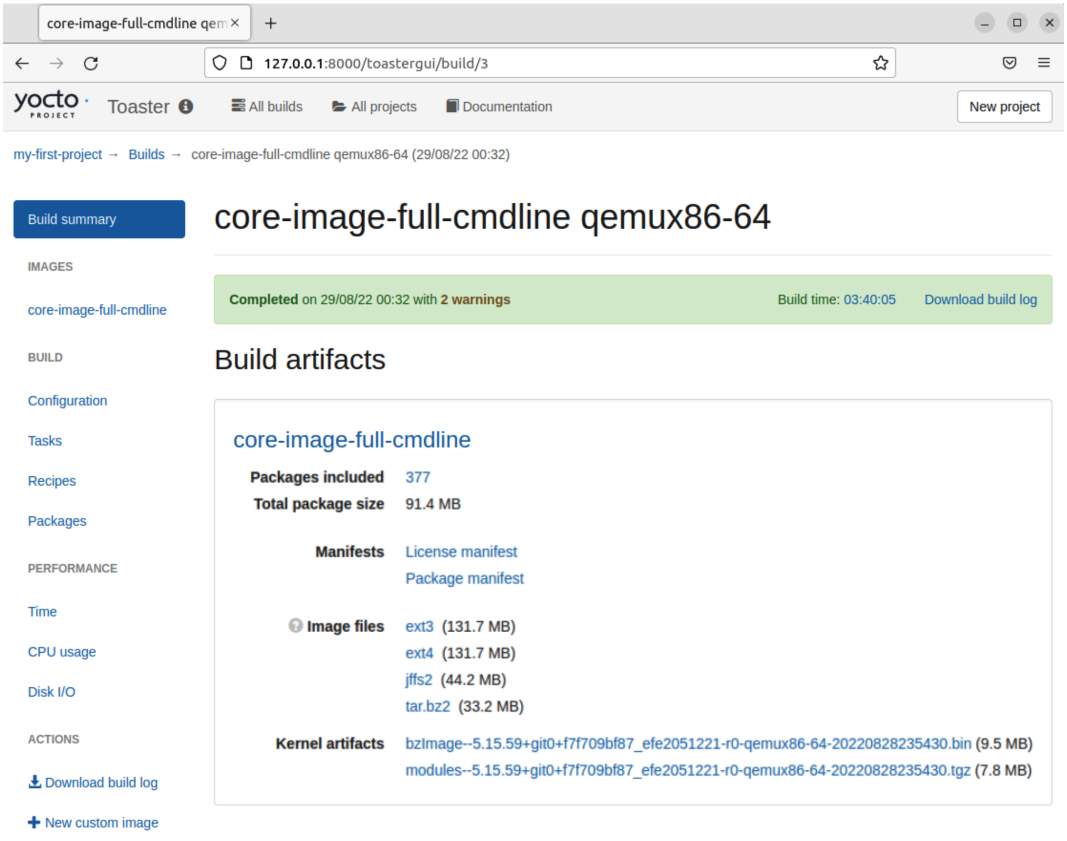


Figure 3.8 – The image build artifact report

We can also verify the generated set of files, as shown in the following screenshot:

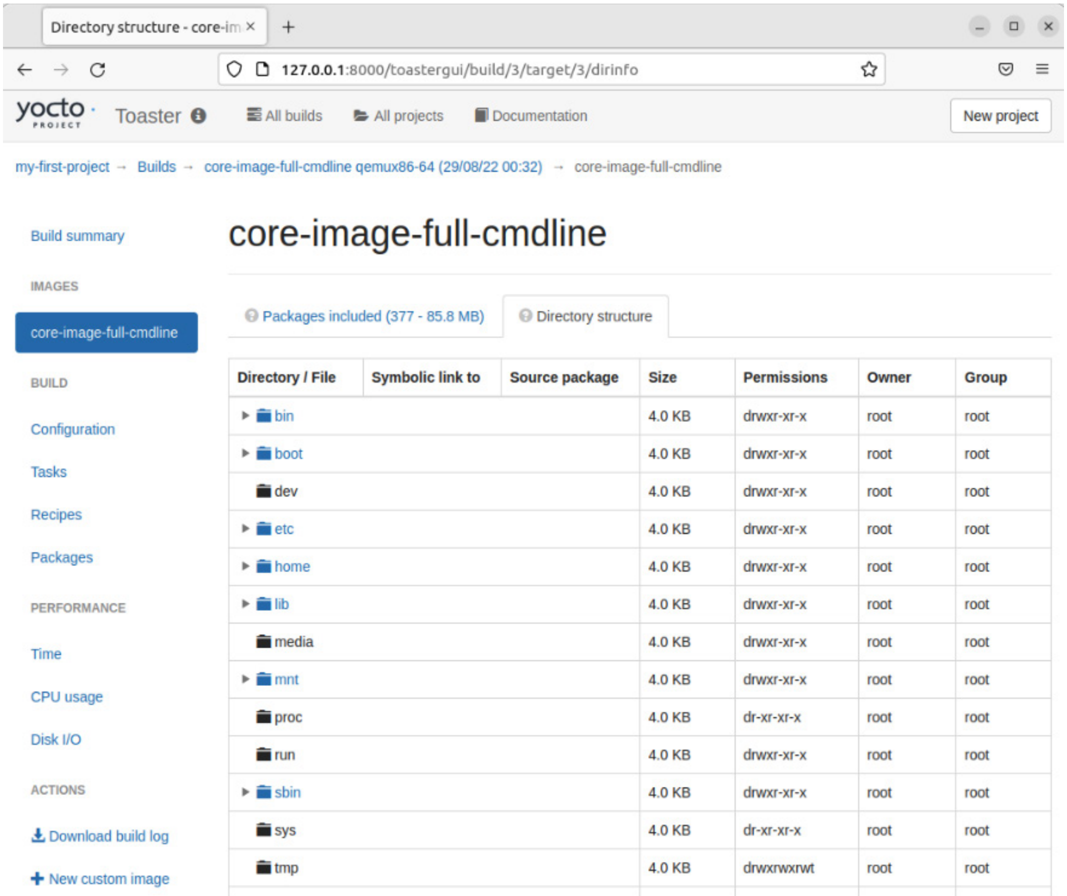


Figure 3.9 – The core-image-full-cmdline directory structure as shown in Toaster

Toaster is a powerful tool. You can use it on a local development machine or a shared server to get a graphic representation of the build. You can return to the terminal where you started Toaster to run `runqemu qemux86-64 core-image-full-cmdline`. You will see what is shown in the following screenshot:

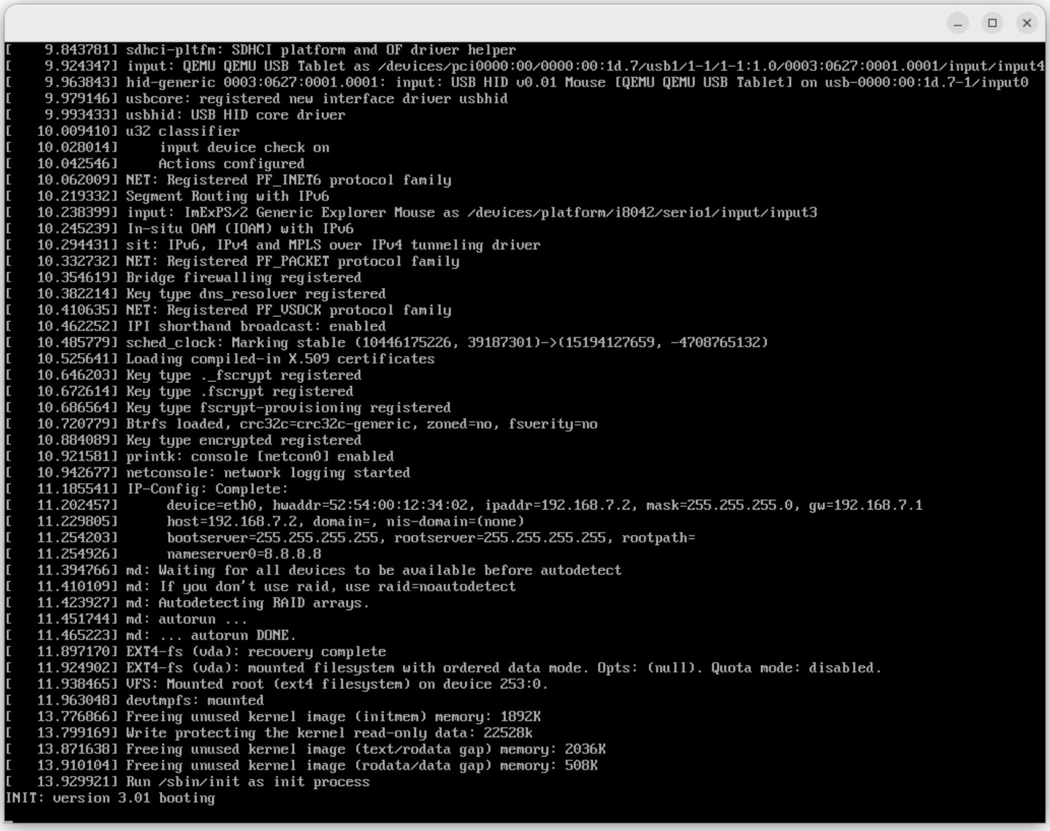


Figure 3.10 – The QEMU screen during the Linux kernel boot

After finishing the Linux booting, you will see a login prompt, as shown in *Figure 3.11*.

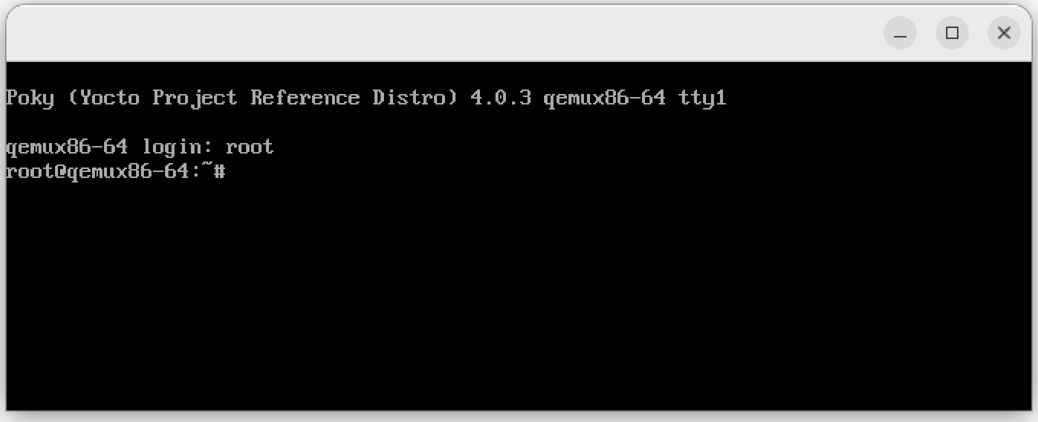


Figure 3.11 – The QEMU screen during user login

We can log in to the `root` account using an empty password.

Summary

In this chapter, we introduced Toaster and its essential features. Then, we went through installing and configuring Toaster and built and inspected an image.

In the next chapter, we will go through some critical BitBake concepts. We believe these concepts are essential to understanding the Yocto Project. We will use BitBake and the command line for the rest of the book, as they provide a view of all the concepts.

4

Meeting the BitBake Tool

With this chapter, we will now begin our journey of learning how the Yocto Project's engine works behind the scenes. As is the case with every journey, communication is critical, so we need to understand the language used by the Yocto Project's tools and learn how to get the best out of these tools to accomplish our goals.

The preceding chapters introduced us to the standard Yocto Project workflow for creating and emulating images. Now, in this chapter, we will explore the concept of metadata and how BitBake reads this metadata to make its internal data collections.

Understanding the BitBake tool

The BitBake task scheduler started as a fork from Portage, the package management system used in the Gentoo distribution. However, the two projects diverged significantly due to different use cases. The Yocto Project and the OpenEmbedded Project are intensive users of BitBake. It remains a separate and independent project with its own development cycle and mailing list (`bitbake-devel@lists.openembedded.org`).

BitBake is a tool similar to GNU Make. As discussed in *Chapter 1*, *Meeting the Yocto Project*, BitBake is a task executor and scheduler that parses Python and Shell Script mixed code.

Therefore, BitBake is responsible for running as many tasks as possible in parallel while ensuring they are run respecting their dependencies.

BitBake metadata collections

For BitBake, there is no metadata outside a metadata collection. Instead, a metadata collection has a unique name, and the common term the Yocto Project uses for those collections is **Layer**.

Chapter 1, *Meeting the Yocto Project*, explains that we have the following layers:

- **OpenEmbedded Core**: This is inside the `meta` directory
- **Poky distribution**: This is inside the `meta-poky` directory
- **Yocto Project reference BSP**: This is inside the `meta-yocto-bsp` directory

The preceding list describes real examples of layers. Every layer contains a file called `conf/layer.conf`. This file defines several layer properties, such as the collection name and priority. The following figure shows the `conf/layer.conf` file for the `meta-poky` layer:

```
 1 # We have a conf and classes directory, add to BBPATH
 2 BBPATH =. "${LAYERDIR}:"
 3
 4 # We have recipes-* directories, add to BBFILES
 5 BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
 6             ${LAYERDIR}/recipes-*/*/*.bbappend"
 7
 8 BBFILE_COLLECTIONS += "yocto"
 9 BBFILE_PATTERN_yocto = "^${LAYERDIR}/"
10 BBFILE_PRIORITY_yocto = "5"
11
12 LAYERSERIES_COMPAT_yocto = "kirkstone"
13
14 # This should only be incremented on significant changes that will
15 # cause compatibility issues with other layers
16 LAYERVERSION_yocto = "3"
17
18 LAYERDEPENDS_yocto = "core"
19
20 REQUIRED_POKY_BBLAYERS_CONF_VERSION = "2"
```

Figure 4.1 – The conf/layer.conf file for the meta-poky layer

The preceding example is relatively simple but serves as a base for us to illustrate the `conf/layer.conf` file principles.

In line 8, `BBFILE_COLLECTIONS`, we tell BitBake to create a new metadata collection called `yocto`. Next, in line 9, `BBFILE_PATTERN_yocto`, we define the rule to match all paths starting with the `LAYERDIR` variable to identify the metadata belonging to the `yocto` collection. Finally, in line 10, `BBFILE_PRIORITY_yocto` establishes the priority (the higher the number, the higher the priority) of the `yocto` collection against the other metadata collections.

The dependency relation between the layers is vital as it ensures that all required metadata is available for use. An example is in line 18 as `LAYERDEPENDS_yocto`, from the `conf/layer.conf` file, adds a dependency to the `core`, provided by the OpenEmbedded Core layer.

Figure 4.2 shows Poky's layers using the `bitbake-layers` command, as follows:

```
$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
meta                  /home/user/yocto/poky/meta                5
meta-poky             /home/user/yocto/poky/meta-poky           5
meta-yocto-bsp        /home/user/yocto/poky/meta-yocto-bsp      5
```

Figure 4.2 – Results of bitbake-layers show-layers for Poky

Metadata types

There are three major areas where we can classify the metadata used by BitBake. They are as follows:

- Configuration (the `.conf` files)
- Classes (the `.bbclass` files)
- Recipes (the `.bb` and `.bbappend` files)

The configuration files define the global content to provide information and configure how the recipes work. One typical example of a configuration file is the machine file, which has a list of settings that describes the hardware.

The whole system uses the classes that recipes can `inherit` according to their needs or by default. They define the commonly used system's behavior and provide the base methods. For example, `kernel.bbclass` abstracts tasks related to building and packaging the Linux kernel independently of version or vendor changes.

> **Note**
>
> The recipes and classes mix Python and Shell Script code.

The classes and recipes describe the tasks to be run and provide the information needed to allow BitBake to generate the required task list and its dependencies. The inheritance mechanism permits a recipe to inherit one or more classes to promote code reuse, improve accuracy, and make maintenance easier. A Linux kernel recipe example is `linux-yocto_5.15.bb`, which inherits a set of classes, including `kernel.bbclass`.

BitBake's most commonly used aspects across all types of metadata (`.conf`, `.bb`, and `.bbclass`) are covered in *Chapter 5*, *Grasping the BitBake Tool*, while the metadata grammar and syntax are detailed in *Chapter 8*, *Diving into BitBake Metadata*.

Taking *Figure 4.1* into consideration, we need to pay attention to two other variables – `BBPATH` and `BBFILES`.

`BBPATH`, on line 2, is analogous to `PATH` but adds a directory to the search list for metadata files; the `BBFILES` variable, on line 5, lists the pattern used to index the collection recipe files.

Summary

In this chapter, we learned about metadata, metadata collection concepts, and the importance of `conf/layer.conf`, which are the base for the understanding of the Yocto Project. In the next chapter, we will examine metadata knowledge, understand how recipes depend on each other, and how BitBake deals with dependencies. Additionally, we will also get a better view of the tasks managed by BitBake, download all the required source code, build and generate packages, and see how these packages fit into generated images.

5
Grasping the BitBake Tool

In the previous chapter, we learned about metadata, metadata collection concepts, and the importance of `conf/layer.conf`. In this chapter, we will examine metadata more deeply, understand how recipes depend on each other, and see how BitBake deals with dependencies.

In addition, we will cover a massive list of tasks, from downloading source code to generating images and other artifacts. Some examples of these tasks are storing the source code in the directory used for the build, patching, configuring, compiling, installing, and generating packages, and determining how the packages fit into the generated images, which we will introduce in this chapter.

Parsing metadata

Usually, our projects include multiple layers that provide different metadata to fulfill specific needs. For example, when we initialize a build directory, using `source oe-init-build-env build`, a set of files is generated as follows:

```
$ tree build
build
└── conf
    ├── bblayers.conf
    ├── local.conf
    └── templateconf.cfg

1 directory, 3 files
```

Figure 5.1 – A list of files created with source oe-init-build-env build

The `build/conf/templateconf.cfg` file points to the directory used as the template to create the `build/conf` directory.

> **Note**
>
> A user can provide a different template directory using the `TEMPLATECONF` environment variable – for example, `TEMPLATECONF=/some/dir source oe-init-build-env build`.

The `build/conf/local.conf` file is the placeholder for the local configurations. We used this file in *Chapter 2*, *Baking Our First Poky-Based System*, and we will use it throughout this book.

BitBake uses the `build/conf/bblayers.conf` file to list the layers considered in the build environment. An example is as follows:

```
 1 # POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
 2 # changes incompatibly
 3 POKY_BBLAYERS_CONF_VERSION = "2"
 4
 5 BBPATH = "${TOPDIR}"
 6 BBFILES ?= ""
 7
 8 BBLAYERS ?= " \
 9   /home/user/yocto/poky/meta \
10   /home/user/yocto/poky/meta-poky \
11   /home/user/yocto/poky/meta-yocto-bsp \
12   "
```

Figure 5.2 – The build/conf/bblayer.conf content after the source oe-init-build-env build

The `BBLAYERS` variable, on line 8, is a space-delimited list of layer directories. BitBake parses each layer to load its content to the metadata collection. There are three major categories that the metadata used by BitBake can be classified into. They are listed as follows:

- Configuration (the `.conf` files)
- Classes (the `.bbclass` files)
- Recipes (the `.bb` and `.bbappend` files)

> **Tip**
>
> The order of the listed layers in the `BBLAYERS` variable is followed from left to right by BitBake when parsing the metadata. Therefore, if your layer needs to be parsed first, have it listed in the right place in the `BBLAYERS` variable.

After parsing all the layers in use, BitBake starts to parse the metadata. The first parsed metadata in BitBake is configuration metadata, identified by the `.conf` file extension. This metadata is global and, therefore, affects all executed recipes and tasks.

> **Note**
> One typical example of the configuration file is the machine file, which has a list of settings that describes the hardware.

BitBake first loads `meta/conf/bitbake.conf` from one of the paths included in the `BBPATH` list. The `meta/conf/bitbake.conf` file uses `include` directives to pull in metadata, such as architecture-specific metadata, machine configuration files, and the `build/conf/local.conf` file. One significant restriction of BitBake configuration files (`.conf`) is that only variable definitions and `include` directives are allowed.

BitBake's classes (`.bbclass`) are a rudimentary inheritance mechanism in the `classes/` directories. When an `inherit` directive appears during parsing, BitBake immediately parses the linked class. The class content is searched based on the order of the `BBPATH` variable list.

The `BBFILES` variable is a space-separated list of the `.bb` and `.bbappend` files and can use wildcards. It is required in every layer inside `conf/layer.conf`, so BitBake knows where to look for recipes. A BitBake recipe (`.bb`) is a logical unit of tasks to be executed; typically, it refers to a package.

Dependencies

From the BitBake point of view, there are three different dependency types:

- Build time
- Execution time
- Tasks

An application that needs some other package, such as a library, has a build dependency for a successful compilation. Build dependencies include compilers. libraries, and native build tools (such as **CMake**). In addition, a build dependency has an execution dependency whenever an application is needed only during execution time. Runtime dependencies include fonts, icons, dynamically opened libraries, and language interpreters.

> **Tip**
> The convention inside Poky is to use `-native` suffixes for recipe names. This is because those tools are aimed to be run during the build process, in the host building system, and are not deployed into the target.

The task dependencies create order in the chaos of task execution – for example, to compile a package, the source code needs to be downloaded. Under the hood, all the dependencies are task dependencies. This means that when package `B` has a build-time dependency on package `A`, the tasks from package `A` need to be completed before package `B` starts.

Metadata expresses all the dependencies. OpenEmbedded Core provides a vast set of classes to handle the default task dependencies commonly used – for example, a recipe can express a build-time dependency with the `DEPENDS` variable and an execution-time dependence with the `RDEPENDS` variable.

Knowing the recipe dependencies chain, BitBake can sort all the recipes for the build in a feasible order. BitBake organizes tasks in the following ways:

- Recipe tasks that do not have a dependency relation are built in parallel
- Dependent recipes are built in serial order and sorted in a way that satisfies the dependencies

> **Tip**
> Every recipe included in the runtime dependencies is added to the build list. This sounds obvious, but even though they have no role during the build, they need to be ready for use so that the resulting binary packages are installable. This will be required when building images or populating feeds.

Preferring and providing recipes

Dependency is a relation between two things; one side can only be fulfilled if the other side exists. However, a dependency only specifies that some functionality or characteristic is needed to be fulfilled, not precisely how it must be fulfilled.

For example, when a recipe depends on `A`, the first thought is that it depends on a recipe called `A`. However, there are two possible ways to satisfy the dependency requirement of `A`:

- A recipe called `A`
- A recipe that provides a functionality or characteristic called `A`

For a recipe to communicate to BitBake that it can fulfill a functionality or characteristic requirement, it must use the `PROVIDES` keyword. A subtle consequence is that two or more recipes can deliver the same functionality or characteristic. We must inform BitBake which recipe should fulfill that requirement using the `PREFERRED_PROVIDER` keyword.

So, if a recipe called `foo_1.0.bb` depends on `bar`, BitBake lists all recipes providing `bar`. The `bar` dependency can be satisfied by the following:

- A recipe with the `bar_<version>.bb` format because every recipe provides itself by default
- A recipe where the `PROVIDES` variable includes `bar`

The `virtual/kernel` provider is a clear example of this mechanism. The `virtual/` namespace is the convention adopted when we have a set of commonly overridden providers.

All recipes that require the kernel to be built can add `virtual/kernel` to the dependency list (`DEPENDS`), and BitBake satisfies the dependency. When we have more than one recipe with an alternative provider, we must choose one to be used – for example, the following:

```
PREFERRED_PROVIDER_virtual/kernel = "linux-mymachine"
```

Figure 5.3 – An example of how to set a preferred provider for virtual/kernel

The `virtual/kernel` provider is commonly set in the machine definition file, as it can vary from machine to machine. We will see how to create a machine definition file in *Chapter 12*, *Creating Custom Layers*.

> **Note**
> BitBake raises an error when a dependency cannot be satisfied due to a missing provider.

When BitBake has two providers with different versions, it uses the highest version by default. However, we can force BitBake to use a different version by using `PREFERRED_VERSION`. This is common in BSPs, such as bootloaders, where vendors may use specific versions for a board.

We can avoid using a development or an unreliable recipe version, by default, lowering the version preference by using the `DEFAULT_PREFERENCE` keyword in a recipe file, as follows:

```
DEFAULT_PREFERENCE = "-1"
```

Figure 5.4 – How to lower the version preference in a recipe

So, even if the version is higher, the recipe is not choosen without `PREFERRED_VERSION` being explicitly set to use it.

Fetching the source code

When we download the Poky source code, we download the metadata collection and the BitBake tool. One of the main features supported by BitBake is additional source code fetching.

The ability of fetching external source code is as modular and flexible as possible. For example, every Linux-based system includes the Linux kernel and several other utilities that form the `root` filesystem, such as OpenSSH or BusyBox.

The OpenSSH source code is available from its upstream website as a `tar.gz` file hosted on an HTTP server, while the Linux kernel release is in a Git repository. Therefore, BitBake can easily fetch those two different instances of source code.

BitBake offers support for many different fetcher modules that allow the retrieval of tarball files and several other SCM systems, such as the following:

- Amazon AWS S3
- Android repo
- Azure Storage
- Bazaar
- ClearCase
- CVS
- FTP
- Git
- Git Annex
- Git submodules
- HTTP(S)
- Mercurial
- NPM
- NPMSW (`npm shrinkwrap` implementation)
- openSUSE Build Service client
- Perforce
- Rust Crate
- SFTP
- SSH
- Subversion

The mechanism used by BitBake to fetch the source code is internally called a fetcher backend, which is configurable to align a user's requirements and optimize fetching the source code.

Remote file downloads

BitBake supports several methods for remote file downloads. The most commonly used are `http://`, `https://`, and `git://`. We won't cover the internal details of how BitBake handles remote file downloads and will instead focus on its visible effects.

When BitBake executes the `do_fetch` task in a recipe, it checks the `SRC_URI` contents. Let's look at, for example, the `pm-utils` recipe (available at `meta/recipes-bsp/pm-utils/pm-utils_1.4.1.bb`). The processed variables are shown in the following figure:

```
SRC_URI = "http://pm-utils.freedesktop.org/releases/pm-utils-${PV}.tar.gz"
SRC_URI[sha256sum] = "8ed899032866d88b2933a1d34cc75e8ae42dcde20e1cc21836baaae3d4370c0b"
```

Figure 5.5 – SRC_URI for the pm-utils_1.4.1.bb recipe

BitBake expands the `PV` variable to the package version (`1.4.1` in this example is taken from the `pm-utils_1.4.1.bb` recipe filename) to download the file from `http://pm-utils.freedesktop.org/releases/pm-utils-1.4.1.tar.gz`, and then saves it as `DL_DIR`, which points to the download storage directory.

After the download is complete, BitBake compares the `sha256sum` value of the downloaded file with the value from the recipe. If the value matches, it creates a `${DL_DIR}/pm-utils-1.4.1.tar.gz.done` file to mark the file as successfully downloaded and checked, allowing BitBake to reuse it.

> **Note**
> By default, the `DL_DIR` variable points to `build/downloads`. You can override this by adding to the `build/conf/local.conf` file the following line – `DL_DIR = "/my/download-cache"`. Using this, we can share the same download cache among several build directories, thus saving download time and bandwidth.

Git repositories

One of the most commonly used source control management systems is Git. BitBake has solid support for Git, and the Git backend is used when the `do_fetch` task is run and finds a `git://` URL at the beginning of the `SRC_URI` variable.

The default way for BitBake's Git backend to handle the repositories is to clone the repository in `${DL_DIR}/git2/<git URL>` – for example, check the following quote from the `lz4_1.9.4.bb` recipe found in `meta/recipes-support/lz4/lz4_1.9.4.bb` inside Poky:

```
SRCREV  = "5ff839680134437dbf4678f3d0c7b371d84f4964"
SRC_URI = "git://github.com/lz4/lz4.git;branch=release;protocol=https"
```

Figure 5.6 – Source code download configuration for the lz4_1.9.4.bb recipe

Here, the `lz4.git` repository is cloned in `${DL_DIR}/git2/ github.com.lz4.lz4.git`. This directory name avoids conflicts between possible Git repositories with the same project name.

There are two cases where the SRCREV variable has an impact. They are as follows:

- do_fetch: This task uses the SRCREV variable to ensure the repository has the required Git revision
- do_unpack: This task uses SRCREV to set up the working directory with the necessary source revision

> **Note**
> We need to use the branch=<branch name> parameter as follows – SRC_URI = "git://myserver/myrepo.git;branch=mybranch" – to specify the branch that contains the revision we want to use. In cases when the hash used points to a tag that is not available on a branch, we need to use the nobranch=1 option as follows – SRC_URI = "git://myserver/myrepo.git;nobranch=1".

The remote file and the Git repository are the most commonly used fetch backends of BitBake. The other source code management-supported systems vary in their implementations, but the general ideas and concepts are the same.

Optimizing the source code download

To improve the robustness of source code download, Poky provides a mirror mechanism that can provide the following:

- A centrally preferred server for download
- A set of fallback servers

To provide this robust download mechanism, BitBake follows defined logic steps. During the build, the first BitBake step is to search for the source code within the local download directory (specified by DL_DIR). If this fails, the next step is to try the locations defined by the PREMIRRORS variable. Finally, BitBake searches the locations specified in the MIRRORS variable in a failure case. In summary, these steps are as follows:

1. DL_DIR: Look for the download on the host machine.
2. MIRRORS: Search for the download in a list of mirrors.
3. PREMIRRORS: This is used to reduce the download from external servers and is usually used inside companies to reduce or forbid internet use.

For example, when configuring a local server, `https://mylocalserver`, as `PREMIRROR`, we can add the following code to a global configuration file, such as `build/conf/local.conf`:

```
PREMIRRORS = " \
cvs://.*/.*     https://mylocalserver \
svn://.*/.*     https://mylocalserver \
git://.*/.*     https://mylocalserver \
gitsm://.*/.*   https://mylocalserver \
hg://.*/.*      https://mylocalserver \
bzr://.*/.*     https://mylocalserver \
p4://.*/.*      https://mylocalserver \
osc://.*/.*     https://mylocalserver \
https?://.*/.*  https://mylocalserver \
ftp://.*/.*     https://mylocalserver \
npm://.*/?.*    https://mylocalserver \
s3://.*/.*      https://mylocalserver \
"
```

Figure 5.7 – An example of the PREMIRRORS configuration

The preceding code prepends the `PREMIRRORS` variable to change and instructs the build system to intercept any download requests. It redirects them to the `https://mylocalserver` source's mirror.

This use of `PREMIRRORS` is so common that there is a class to help its configuration. To make it easier, we inherit the `own-mirror` class and then set the `SOURCE_MIRROR_URL` variable to `https://mylocalserver` in any global configuration file, such as `build/conf/local.conf`.

```
INHERIT += "own-mirrors"
SOURCE_MIRROR_URL = "https://mylocalserver"
```

Figure 5.8 – How to configure own-mirror

If the desired component is unavailable in the source mirror, BitBake falls back to the `MIRRORS` variable. An example of the content of this variable is shown in the following figure. It shows some servers used in `mirrors.bbclass`, inherited by default in Poky:

```
MIRRORS += "\
cvs://.*/.*     http://downloads.yoctoproject.org/mirror/sources/ \
svn://.*/.*     http://downloads.yoctoproject.org/mirror/sources/ \
git://.*/.*     http://downloads.yoctoproject.org/mirror/sources/ \
gitsm://.*/.*   http://downloads.yoctoproject.org/mirror/sources/ \
hg://.*/.*      http://downloads.yoctoproject.org/mirror/sources/ \
bzr://.*/.*     http://downloads.yoctoproject.org/mirror/sources/ \
p4://.*/.*      http://downloads.yoctoproject.org/mirror/sources/ \
osc://.*/.*     http://downloads.yoctoproject.org/mirror/sources/ \
https?://.*/.*  http://downloads.yoctoproject.org/mirror/sources/ \
ftp://.*/.*     http://downloads.yoctoproject.org/mirror/sources/ \
npm://.*/?.*    http://downloads.yoctoproject.org/mirror/sources/ \
cvs://.*/.*     http://sources.openembedded.org/ \
svn://.*/.*     http://sources.openembedded.org/ \
git://.*/.*     http://sources.openembedded.org/ \
gitsm://.*/.*   http://sources.openembedded.org/ \
hg://.*/.*      http://sources.openembedded.org/ \
bzr://.*/.*     http://sources.openembedded.org/ \
p4://.*/.*      http://sources.openembedded.org/ \
osc://.*/.*     http://sources.openembedded.org/ \
https?://.*/.*  http://sources.openembedded.org/ \
ftp://.*/.*     http://sources.openembedded.org/ \
npm://.*/?.*    http://sources.openembedded.org/ \
"
```

Figure 5.9 – An example of how to use the MIRRORS variable

Tip

Let's suppose the goal is to have a shareable download cache. In that case, it is advisable to enable the tarball generation for the SCM backends (for example, Git) in the download folder with `BB_GENERATE_MIRROR_TARBALLS = "1"` in `build/conf/local.conf`.

Disabling network access

Sometimes, we need to ensure that we don't connect to the internet during the build process. There are several valid reasons for this, such as the following:

- **Policy**: Our company does not allow the inclusion of external sources in a product without proper legal validation and review.
- **Network cost**: When we are on the road using mobile broadband, the cost of data may be too high because the data to download may be extensive.
- **Download and build decoupling**: This setup is typical in continuous integration environments, where a job is responsible for downloading all the required source code. In contrast, the build jobs have internet access disabled. The decoupling between downloading and building ensures that no source code is downloaded in duplication and that we have cached all the necessary source code.
- **Lack of network access**: Sometimes, we do not have access to a network.

To disable the network connection, we need to add the following code in the `build/conf/local.conf` file:

```
BB_NO_NETWORK = "1"
```

Figure 5.10 – How to disable network access during the build

Understanding BitBake's tasks

BitBake uses execution units, which are, in essence, a set of clustered instructions that run in sequence. These units are known as **tasks**. During every recipe's build, BitBake, schedules, executes, and checks many tasks provided by classes to form the framework we use to build a recipe. Therefore, it is essential to understand some of these, as we often use, extend, implement, or replace them ourselves when writing a recipe.

When we run the following command, BitBake runs a set of scheduled tasks:

```
$ bitbake <recipe>
```

Figure 5.11 – How to run all tasks for a recipe

When we wish to run a specific task, we can use the following command:

```
$ bitbake <recipe> -c <task>
```

Figure 5.12 – How to run a particular task for a recipe

To list the tasks defined for a recipe, we can use the following command:

```
$ bitbake <recipe> -c listtasks
```

Figure 5.13 – How to list all tasks for a recipe

The output of `listtasks` for the `wget` recipe is as follows:

```
do_build                              Default task for a recipe - depends on all other normal tasks
                                      required to 'build' a recipe
do_checkuri                           Validates the SRC_URI value
do_clean                              Removes all output files for a target
do_cleanall                           Removes all output files, shared state cache, and downloaded source
                                      files for a target
do_cleansstate                        Removes all output files and shared state cache for a target
do_compile                            Compiles the source in the compilation directory
do_configure                          Configures the source by enabling and disabling any build-time and
                                      configuration options for the software being built
do_deploy_source_date_epoch
do_deploy_source_date_epoch_setscene   (setscene version)
do_devshell                           Starts a shell with the environment set up for development/debugging
do_fetch                              Fetches the source code
do_install                            Copies files from the compilation directory to a holding area
do_listtasks                          Lists all defined tasks for a target
do_package                            Analyzes the content of the holding area and splits it into subsets
                                      based on available packages and files
do_package_qa                         Runs QA checks on packaged files
do_package_qa_setscene                Runs QA checks on packaged files (setscene version)
do_package_setscene                   Analyzes the content of the holding area and splits it into subsets
                                      based on available packages and files (setscene version)
do_package_write_rpm                  Creates the actual RPM packages and places them in the Package Feed
                                      area
do_package_write_rpm_setscene         Creates the actual RPM packages and places them in the Package Feed
                                      area (setscene version)
do_packagedata                        Creates package metadata used by the build system to generate the
                                      final packages
do_packagedata_setscene               Creates package metadata used by the build system to generate the
                                      final packages (setscene version)
do_patch                              Locates patch files and applies them to the source code
do_populate_lic                       Writes license information for the recipe that is collected later
                                      when the image is constructed
do_populate_lic_setscene              Writes license information for the recipe that is collected later
                                      when the image is constructed (setscene version)
do_populate_sysroot                   Copies a subset of files installed by do_install into the sysroot in
                                      order to make them available to other recipes
do_populate_sysroot_setscene          Copies a subset of files installed by do_install into the sysroot in
                                      order to make them available to other recipes (setscene version)
do_prepare_recipe_sysroot
do_pydevshell                         Starts an interactive Python shell for development/debugging
do_unpack                             Unpacks the source code into a working directory
```

Figure 5.14 – The list of tasks for the wget recipe

We will briefly describe the most commonly used tasks here:

- `do_fetch`: The first step when building a recipe is fetching the required source using the fetching backends feature, which we discussed previously in this chapter. It is essential to note that fetching a source or a file does not mean it is a remote source.
- `do_unpack`: The subsequent natural task after the `do_fetch` task is `do_unpack`. This is responsible for unpacking source code or checking out the requested revision or branch in case the referenced source uses an SCM system.
- `do_patch`: Once the source code is properly unpacked, BitBake initiates adapting the source code. Every file fetched by `do_fetch`, with the `.patch` extension, is assumed to be a patch to be applied. This task applies the list of patches needed. The final modified source code will be used to build the package.

- `do_configure`, `do_compile`, and `do_install`: The `do_configure`, `do_compile`, and `do_install` tasks are performed in this order. It is important to note that the environment variables defined in the tasks are different from one task to another. Poky provides a rich collection of predefined tasks in the classes, which we ought to use when possible – for example, when a recipe inherits the `autotools` class, it provides a known implementation of the `do_configure`, `do_compile`, and `do_install` tasks.
- `do_package`: The `do_package` task splits the files installed by the recipe into logical components, such as debugging symbols, documentation, and libraries. We will cover packaging details in more depth in *Chapter 7*, *Assimilating Package Support*.

Summary

In this chapter, we learned how recipes depend on each other and how Poky deals with dependencies. We understood how a download is configured and how to optimize it. In addition, we got a better view of the tasks managed by BitBake to download all the required source code and use it to build and generate packages.

In the next chapter, we will see the contents of the build directory after complete image generation and learn how BitBake uses it in the baking process, including the contents of the temporary build directory and its generated files.

6
Detailing the Temporary Build Directory

In this chapter, we will try to understand the contents of the temporary build directory after image generation and see how BitBake uses it in the baking process. In addition, we will learn how some of these directories can assist us by acting as a valuable source of information when things do not work as expected.

Detailing the build directory

The build directory is a central information and artifact source for every Poky user. Its main directories are as follows:

- `conf`: This contains the configuration files we use to control Poky and BitBake. We first used this directory in *Chapter 2*, *Baking Our Poky-Based System*. It stores configuration files, such as `build/conf/local.conf` and `build/conf/bblayers.conf`.
- `downloads`: This stores all the downloaded artifacts. It works as a download cache. We talked about it in detail in *Chapter 5*, *Grasping the BitBake Tool*.
- `sstate-cache`: This contains the snapshots of the packaged data. It is a cache mainly used to speed up the future build process, as it is used as a cache for the building process. This folder is detailed in *Chapter 7*, *Assimilating Packaging Support*.
- `tmp`: This is the temporary build directory and the main focus of this chapter.

Constructing the build directory

In the previous chapters, we learned about Poky's inputs and outputs in abstract high-level detail. We already know that BitBake uses metadata to generate different types of artifacts, including images. Besides the generated artifacts, BitBake creates other content during this process, which may be used in several ways, dependent on our goals.

BitBake performs several tasks and continuously modifies the build directory during the build process. Therefore, we can understand it better by following the usual BitBake execution flow, as follows:

- **Fetching**: The first action executed by BitBake is to download the source code. This step may modify the build directory as it tries to use the cached downloaded copy of the source code or performs the download and stores it inside the `build/download` directory.
- **Source preparation**: After completing the source code fetching, it must be prepared; for example, the unpacking of a tarball or a clone of a locally cached Git directory (from the download cache). This preparation happens in the `build/tmp/work` directory. When the source code is ready, the required modifications are applied (for example, applying necessary patches and checking out the correct Git revision).
- **Configuration and building**: The building process starts with the *ready-to-use* source code. It involves the configuration of build options (for example, `./configure`) and building (for example, `make`).
- **Installing**: The built artifacts are then installed (for example, `make install`) in a staging directory under `build/tmp/work/<...>/image`.
- **Wrapping the sysroot**: The artifacts required for cross-compilation, such as libraries, headers, and other files, are copied and sometimes modified to `build/tmp/work/<...>/recipe-sysroot` and `build/tmp/work/<...>/recipe-sysroot-native`.
- **Creating the packages**: The packages are generated using the installed contents, potentially splitting this content across multiple packages, which can be provided in different formats, for example, `.rpm`, `.ipk`, `.deb`, or `.tar`.
- **Quality Assurance (QA) checks**: When building a recipe, the build system performs various QA checks on the output to ensure that common issues are detected and reported.

Exploring the temporary build directory

Understanding the temporary build directory (`build/tmp`) is critical. The temporary build directory is created just after the build starts, and it's essential for helping us identify why something didn't behave as expected.

The following figure shows the contents of the `build/tmp` directory:

```
build/tmp/
├── abi_version
├── buildstats
├── cache
├── deploy
├── hosttools
├── log
├── pkgdata
├── saved_tmpdir
├── sstate-control
├── stamps
├── sysroots-components
├── sysroots-uninative
├── work
└── work-shared

12 directories, 2 files
```

Figure 6.1 – Contents of build/tmp

The most critical directories found within it are as follows:

- `deploy`: This contains the build products, such as images, binary packages, and SDK installers.
- `sysroots-components`: This contains a representation of `recipes-sysroot` and `recipes-sysroot-native`, which allows BitBake to know where each component is installed. This is used to create recipe-specific `sysroots` during the build.
- `sysroots-uninative`: This includes `glibc` (a C library), which is used when native utilities are generated. This, in turn, improves the reuse of shared state artifacts across different host distributions.
- `work`: This contains the working source code, a task's configuration, execution logs, and the contents of generated packages.
- `work-shared`: This is a `work` directory used for sharing the source code with multiple recipes. `work-shared` is only used by a subset of recipes, for example, `linux-yocto` and `gcc`.

Understanding the work directory

The `build/tmp/work` directory is organized by architecture. For example, when working with the `qemux86-64` machine, we have the following four directories:

```
build/tmp/work
├── all-poky-linux
├── core2-64-poky-linux
├── qemux86_64-poky-linux
└── x86_64-linux

4 directories, 0 files
```

Figure 6.2 – The contents of the `build/tmp/work` directory

Figure 6.2 shows an example of possible directories under `build/tmp/work` for an `x86-64` host and a `qemux86-64` target. They are architecture- and machine-dependent, as follows:

- `all-poky-linux`: This directory contains the working build directories for the architecture-agnostic packages. These are mostly scripts or interpreted language-based packages, for example, Perl scripts and Python scripts.
- `core2-64-poky-linux`: This directory contains the working build directories for the packages common to `x86-64`-based targets using the optimization tuned for `core2-64`.
- `qemux86_64-poky-linux`: This directory contains the working build directories for packages specific to the `qemux86-64` machine.
- `x86_64-linux`: This directory holds the working build directories for the packages that are targeted to run on the build host machine.

This componentized structure is necessary to allow building system images and packages for multiple machines and architectures within one `build` directory without conflicts. The target machine we will use is `qemux86-64`.

The `build/tmp/work` directory is useful when checking for misbehavior or building failures. Its contents are organized in sub-directories following this pattern:

```
<architecture> / <recipe name> / <software version>
```

Figure 6.3 – The pattern used in sub-directories of the build/tmp/work directory

Some of the directories under the tree shown in *Figure 6.3* are as follows:

- `<sources>`: This is extracted source code of the software to be built. The `WORKDIR` variable points to this directory.
- `image`: This contains the files installed by the recipe.
- `package`: The extracted contents of output packages are stored here.
- `packages-split`: The contents of output packages, extracted and split into sub-directories, are stored here.
- `temp`: This stores BitBake's task code and execution logs.

> **Tip**
>
> We can automatically remove the `work` directory after each recipe compilation cycle to reduce disk usage, adding `INHERIT += "rm_work"` in the `build/conf/local.conf` file.

The structure of the `work` directory is the same for all architectures. For every recipe, a directory with the recipe name is created. Taking the machine-specific `work` directory and using the `sysvinit-inittab` recipe as an example, we see the following:

```
build/tmp/work/qemux86_64-poky-linux/sysvinit-inittab/2.88dsf-r10/
├── configure.sstate
├── deploy-rpms
├── deploy-source-date-epoch
├── image
├── inittab
├── license-destdir
├── package
├── packages-split
├── patches
├── pkgdata
├── pkgdata-pdata-input
├── pkgdata-sysroot
├── pseudo
├── recipe-sysroot
├── recipe-sysroot-native
├── source-date-epoch
├── sstate-install-deploy_source_date_epoch
├── sstate-install-package
├── sstate-install-packagedata
├── sstate-install-package_qa
├── sstate-install-package_write_rpm
├── sstate-install-populate_lic
├── sstate-install-populate_sysroot
├── start_getty
├── sysroot-destdir
├── sysvinit-inittab.spec
└── temp

23 directories, 4 files
```

Figure 6.4 – Content of build/tmp/work/core2-64-poky-linux/pm-utils/1.4.1-r1/

The `sysvinit-inittab` recipe is an excellent example, as it is machine-specific. This recipe contains the `inittab` file that defines the serial console to spawn the login process, which varies from machine to machine.

> **Note**
>
> The build system uses the directories shown in the preceding figure that are not detailed here. Therefore, you should not need to work with them, except if you are working on build tool development.

The `work` directory is handy for debugging purposes; we cover this in *Chapter 10*, *Debugging with the Yocto Project*.

Understanding the sysroot directories

The `sysroot` directory plays a critical role in the Yocto Project. It creates an individual and isolated environment for each recipe. This environment, set for each recipe, is essential to ensure reproducibility and avoid contamination with the host machine's packages.

After we build the `procps` recipe, version `3.3.17`, we get two sets of `sysroot` directories – `recipes-sysroot` and `recipes-sysroot-native`.

Inside each `sysroot` set, there is a sub-directory called `sysroot-provides`. This directory lists the packages installed on each respective `sysroot`. Following is the `recipe-sysroot` directory:

```
build/tmp/work/core2-64-poky-linux/procps/3.3.17-r0/recipe-sysroot
├── lib
├── sysroot-providers
│   ├── gcc-runtime
│   ├── glibc
│   ├── libgcc
│   ├── libtool-cross
│   ├── linux-libc-headers
│   ├── ncurses
│   ├── opkg-utils
│   ├── virtual_libc
│   ├── virtual_libiconv
│   └── virtual_x86_64-poky-linux-compilerlibs
└── usr
```

Figure 6.5 – Content of the recipe-sysroot directory under build/tmp/work for recipe procps

The `recipe-sysroot-native` directory includes the build dependencies used on the host system during the build process. It encompasses the compiler, linker, tools, and more. At the same time, the `recipe-sysroot` directory has the libraries and headers used in the target code. The following figure shows the `recipe-sysroot-native` directory:

```
buid/tmp/work/core2-64-poky-linux/procps/3.3.17-r0/recipe-sysroot-native/
├── bin
├── etc
├── installeddeps
├── sysroot-providers
│   ├── attr-native
│   ├── autoconf-native
│   ├── automake-native
│   ├── binutils-cross-x86_64
│   ├── bzip2-native
│   ├── bzip2-replacement-native
│   ├── curl-native
│   ├── dwarfsrcfiles-native
│   ├── elfutils-native
│   ├── file-native
│   ├── file-replacement-native
│   ├── flex-native
│   ├── gcc-cross-x86_64
│   ├── gdbm-native
│   ├── gettext-minimal-native
...
│   ├── virtual_librpc-native
│   ├── virtual_make-native
│   ├── virtual_x86_64-poky-linux-binutils
│   ├── virtual_x86_64-poky-linux-g++
│   ├── virtual_x86_64-poky-linux-gcc
│   ├── xz-native
│   ├── zlib-native
│   └── zstd-native
└── usr

16 directories, 272 files
```

Figure 6.6 – Content of the recipe-sysroot-native directory under build/tmp/work for recipe procps

When we see a missing header or a link failure, we must double-check whether our `sysroot` directory (target and host) contents are correct.

Summary

In this chapter, we explored the contents of the temporary build directory after image generation. We saw how BitBake uses it during the baking process.

In the next chapter, we will better understand how packaging is done in Poky, how to use package feeds, the **Package Revision** (**PR**) service, and how they may help our product maintenance.

7
Assimilating Packaging Support

This chapter presents the key concepts for understanding the aspects of Poky and BitBake related to packaging. We will learn about the supported binary package formats, shared state cache, package versioning components, how to set up and use binary package feeds to support our development process, and more.

Using supported package formats

From a Yocto Project perspective, a recipe may generate one or more output packages. A package wraps a set of files and metadata in a way that makes them available in the future. They can be installed into one or more images or deployed for later use.

Packages are critical to Poky, as they enable the build system to produce diverse types of artifacts, such as images and toolchains.

List of supported package formats

Currently, BitBake supports four different package formats:

- **Red Hat Package Manager** (**RPM**): Originally named Red Hat Package Manager but now known as the RPM package format since its adoption by several other Linux distributions, this is a popular format in use in Linux distributions such as SuSE, OpenSuSE, Red Hat, Fedora, and CentOS.
- **Debian Package Manager** (**DEB**): This is a widespread format used in Debian and several other Debian-based distributions – Ubuntu Linux and Linux Mint are the most widely known.

- **Itsy Package Management System** (**IPK**): This was a lightweight package management system designed for embedded devices that resembled Debian's package format. The `opkg` package manager, which supports the IPK format, is used in several distributions such as OpenEmbedded Core, OpenWRT, and Poky.
- **Tar**: This is derived from the **Tape Archive**, a widely used `tarball` file type used to group several files into just a single file.

Choosing a package format

The support for formats is provided using a set of classes (i.e., `package_rpm`, `package_deb`, and `package_ipk`). We can select one or more formats using the `PACKAGE_CLASSES` variable, as shown in the following example:

```
PACKAGE_CLASSES ?= "package_rpm package_deb package_ipk"
```

Figure 7.1 – The variable used to configure which package format to use

You can configure one or more package formats – for example, in the `build/conf/local.conf` file.

> **Tip**
> The first package format in `PACKAGE_CLASSES` is the one used for image generation.

Poky defaults to the RPM package format, which uses the DNF package manager. However, the format choice depends on several factors, such as package format-specific features, memory, and resource usage. OpenEmbedded Core defaults to the IPK and `opkg` as the package manager, as it offers a smaller memory and resource usage footprint.

On the other hand, users familiar with Debian-based systems may prefer to use the APT and DEB package formats for their products.

Running code during package installation

Packages can use scripts as part of their installation and removal process. The included scripts are defined as follows:

- `preinst`: This executes before unpacking the package. If the package has services, it must stop them for installation or upgrade.
- `postinst`: After unpacking, this typically completes any required configuration of the package. Many `postinst` scripts execute any command necessary to start or restart a service after installation or upgrade.

- `prerm`: It usually stops any daemon associated with a package before removing files associated with the package.
- `postrm`: This commonly modifies links or other files created by the package.

The `preinst` and `prerm` scripts target complex use cases, such as data migration when updating packages. In the Yocto Project case, `postinst` and `postrm` are also responsible for stopping and starting the `systemd` or `sysvinit` services. A default script is provided when we use the `systemd` and `update-rc.d` classes. It can be customized to include any particular case.

The post-package installation (`postinst`) scripts are run during the `root` filesystem creation. The package is marked as installed if the script returns a success value. To add a `postinst` script for a package, we can use the following:

```
pkg_postinst:${PN} () {
#  Insert commands above
}
```

Figure 7.2 – An example of the pkg_postinst script

Sometimes, we need to ensure that `postinst` runs inside the target device itself. This can be done using the `postinst_ontarget` variant, such as the following:

```
pkg_postinst_ontarget:${PN} () {
#   Insert commands above
}
```

Figure 7.3 – An example of the pkg_postinst_ontarget script

> **Tip**
>
> Instead of using the package name itself, we can use the `PN` variable, which automatically expands the package name of the recipe.

All post-installation scripts must succeed when we generate an image with `read-only-rootfs` in `IMAGE_FEATURES`. Because it is impossible to write in a read-only `rootfs`, the check must occur during build time. It ensures that we identify the problem while building the image, rather than during the initial boot operation in the target device. If there is a requirement to run any script inside the target device, the `do_rootfs` task fails.

Tip

Eventually, using a whole image as read-only is not an option. For example, some projects may need to persist some data or even allow some applications to write to a volatile directory. Such use cases are outside the scope of this book. However, you might find some helpful information in the *Yocto Project Reference Manual* for the `overlayfs` (`https://docs.yoctoproject.org/4.0.4/ref-manual/classes.html#overlayfs-bbclass`) and `overlayfs-etc` (`https://docs.yoctoproject.org/4.0.4/ref-manual/classes.html#overlayfs-etc-bbclass`) classes.

One common oversight when creating post-installation scripts is the lack of the `D` variable in front of absolute paths. `D` has two traits:

- During `rootfs` generation, `D` is set to the root of the working directory
- Inside the device, `D` is empty

Consequently, this ensures that paths are valid in both host and target environments. For example, consider the following code:

```
pkg_postinst:${PN} () {
    touch $D${sysconfdir}/my-file.conf
}
```

Figure 7.4 – Sample source code using the D variable

In the example in *Figure 7.4*, the `touch` command uses the `D` variable, so it works generically depending on its value.

Another common mistake is attempting to run processes specific to or dependent on the target architecture. The easiest solution, in this case, is to postpone the script execution to the target (using `pkg_postinst_ontarget`). However, as mentioned before, this prevents the use of read-only filesystems.

Understanding shared state cache

The default behavior of Poky is to build everything from scratch unless BitBake determines that a recipe does not need to be rebuilt. The main advantage of building everything from scratch is that the result is fresh, and there is no risk of previous data causing problems. However, rebuilding everything requires computational time and resources.

The strategy to determine whether a recipe must be rebuilt is complex. BitBake tries to track as much information as possible about every task, variable, and piece of code used in the build process. BitBake then generates a checksum for the information used by every task, including dependencies from other tasks. In summary, BitBake recursively tracks used variables, task source code, and dependencies for the recipes and their dependencies.

Poky uses all this information provided by BitBake to store snapshots of those tasks as a set of packaged data, generated in a cache called the shared state cache (`sstate-cache`). This cache wraps the contents of each task output in packages stored in the `SSTATE_DIR` directory. Whenever BitBake prepares to run a task, it first checks the existence of a `sstate-cache` package that matches the required computed checksum. If the package is present, BitBake uses the prebuilt package.

The whole shared state mechanism encompasses quite complex code, and the previous explanation simplifies it. For a detailed description, it is advised that you go through the *Shared State Cache* section of the *Yocto Project Overview and Concepts Manual* (`https://docs.yoctoproject.org/4.0.4/overview-manual/concepts.html#shared-state-cache`).

When using Poky for several builds, we must remember that `sstate-cache` needs cleaning from time to time, since it keeps growing after every build. There is a straightforward way of cleaning it. Use the following command from the `poky` directory:

```
./scripts/sstate-cache-management.sh --remove-duplicated -d --cache-dir=<path to sstate-cached>
```

Figure 7.5 – The command line to remove a duplicated Shared State Cache

> **Tip**
>
> When we need to rebuild from scratch, we can do either of the following:
>
> - Remove `build/tmp` so that we can use `sstate-cache` to speed up the build
> - Remove both `build/tmp` and `sstate-cache` so that no cache is reused during the build

Explaining package versioning

Package versioning is used to differentiate the same package in distinct stages of its life cycle. From Poky's perspective, it is also used as part of the equation that generates the checksum used by BitBake to verify whether a task must be rebuilt.

The package version, also known as `PV`, plays a leading role when we select which recipe to build. The default behavior of Poky is always to prefer the newest recipe version unless there is a different explicit preference, as discussed in *Chapter 5*, *Grasping the BitBake Tool*. For example, let's suppose that we have two versions of the `myrecipe` recipe:

- `myrecipe_1.0.bb`
- `myrecipe_1.1.bb`

BitBake, by default, builds the recipe with version `1.1`. Inside the recipe, we may have other variables that compose package versioning with the `PV` variable. These are the **package epoch**, known as `PE`, and the **package revision**, known as `PR`.

Those variables normally follow this pattern:

```
${PE}:${PV}-${PR}
```

Figure 7.6 – A complete versioning pattern

The `PE` variable has a default value of zero. It is used when the package version schema is changed, breaking the possibility of usual ordering. `PE` is prepended in the package version, forcing a higher number when needed.

For example, suppose a package uses the date to compose `PV` variables such as `20220101`, and there is a version schema change to release the `1.0` version. It is impossible to determine whether version `1.0` is higher than version `20220101`. So, `PE = "1"` is used to change the recipe epoch, forcing version `1.0` to be higher than `20220101`, since `1:1.0` is greater than `0:20220101`.

The `PR` variable has a default value of `r0` and is a part of package versioning. When it is updated, it forces BitBake to rebuild all tasks of a specific recipe. We can update it manually in the recipe metadata to force a rebuild we know is needed. Still, it is fragile because it relies on human interaction and knowledge. BitBake uses task checksums to control what needs to be rebuilt. The manual `PR` increment is only used in rare cases when the task checksum does not change.

Specifying runtime package dependencies

The results of most recipes are packages managed by the package manager. As we saw in the previous sections, it requires information about all those packages and how they relate. For example, a package may depend on or conflict with another.

Constraints exist within multiple package relationships; however, those constraints are package format-specific, so BitBake has specific metadata to abstract those constraints.

Here is a list of the most used package runtime constraints:

- `RDEPENDS`: The list of packages must be available at runtime, along with the package that defines it.
- `RPROVIDES`: This is the list of symbolic names a package provides. By default, a package always includes the package name as a symbolic name. It can also include alternative symbolic names provided by that package.
- `RCONFLICTS`: This is the list of packages known to conflict with the package. The final image must not include conflicting packages.
- `RREPLACES`: This is a list of symbolic names that the package can replace.

A full recipe, from `meta/recipes-devtools/python/python3-dbus_1.2.18.bb`, is as follows:

```
1 SUMMARY = "Python bindings for the DBus inter-process communication system"
2 SECTION = "devel/python"
3 HOMEPAGE = "http://www.freedesktop.org/Software/dbus"
4 LICENSE = "MIT"
5 LIC_FILES_CHKSUM = "file://COPYING;md5=b03240518994df6d8c974675675e5ca4"
6 DEPENDS = "expat dbus glib-2.0 virtual/libintl"
7
8 SRC_URI = "http://dbus.freedesktop.org/releases/dbus-python/dbus-python-${PV}.tar.gz"
9
10 SRC_URI[sha256sum] = "92bdd1e68b45596c833307a5ff4b217ee6929a1502f5341bae28fd120acf7260"
11
12 S = "${WORKDIR}/dbus-python-${PV}"
13
14 inherit setuptools3-base autotools pkgconfig
15
16 # documentation needs python3-sphinx, which is not in oe-core or meta-python for now
17 # change to use PACKAGECONFIG when python3-sphinx is added to oe-core
18 EXTRA_OECONF += "--disable-documentation"
19
20
21 RDEPENDS:${PN} = "python3-io python3-logging python3-stringold python3-threading \
22                   python3-xml"
23
24 FILES:${PN}-dev += "${libdir}/pkgconfig"
25
26 BBCLASSEXTEND = "native nativesdk"
27
```

Figure 7.7 – An example of how to use RDEPENDS

The recipe from *Figure 7.6* shows that the `python3-dbus` package has a list of runtime dependencies on several Python modules, on line 21.

Using packages to generate a rootfs image

One of the most common uses of Poky is the `rootfs` image generation. The `rootfs` image should be seen as a ready-to-use root filesystem for a target. The image can be composed of one or more filesystems. It may include other artifacts available during its generation, such as the Linux kernel, the device tree, and bootloader binaries. The process of generating the image is composed of several steps. Its most common uses are as follows:

1. Generating the `rootfs` directory
2. Creating the required files
3. Wrapping the final filesystem according to the specific requirements (it may be a disk file with several partitions and contents)
4. Finally, compressing it, if applicable

The sub-tasks of do_rootfs perform all these steps. rootfs is a directory with the desired packages installed, with the required tweaks applied afterward. The tweaks make minor adjustments to the rootfs contents – for example, when building a development image, rootfs is adjusted to allow us to log in as root without a password.

The list of packages to be installed into rootfs is defined by a union of packages listed by IMAGE_INSTALL and the packages included with IMAGE_FEATURES; image customization is detailed in *Chapter 12*, *Creating Custom Layers*. Each image feature can include extra packages for installation – for example, dev-pkgs, which installs development libraries and headers of all packages listed to be installed in rootfs.

The list of packages to be installed is now filtered by the PACKAGE_EXCLUDE variable, which lists the packages that should not be installed. The packages listed in PACKAGE_EXCLUDE are only excluded from the list of packages to be explicitly installed.

With the final set of packages to install, the do_rootfs task can initiate the process of unpacking and configuring each package, and its required dependencies, into the rootfs directory. The rootfs generation uses the local package feed, which we will cover in the next section.

With the rootfs contents unpacked, the non-target post-installation scripts of the referred packages must run to avoid the penalty of running them during the first boot.

Now, the directory is ready to generate the filesystem. IMAGE_FSTYPES lists the filesystem to be generated – for example, **EXT4** or **UBIFS**.

After the do_rootfs task has finished, the generated image file is placed in build/tmp/deploy/image/<machine>/. The process of creating our image and the possible values for IMAGE_FEATURES and IMAGE_FSTYPES are described in *Chapter 12*, *Creating Custom Layers*.

Package feeds

As discussed in *Chapter 5*, *Grasping the BitBake Tool*, packages play a vital role, as images and **Software Development Kits** (**SDKs**) rely on them. In fact, do_rootfs uses a local repository to fetch binary packages when generating those artifacts. This repository is known as a package feed.

There is no reason for this repository to be used just for the images or SDK build steps. Several valid reasons exist for making this repository remotely accessible, either internally in our development environment or publicly. Some of these reasons are as follows:

- You can easily test an updated application during the development stage, without requiring a complete system re-installation
- You can make additional packages more flexible so that they can be installed in a running image
- You can update products in the field

To produce a solid package feed, we must ensure that we have consistent increments in the package revision every time the package is changed. It is almost impossible to do this manually, and the Yocto Project has a `PR` service specifically designed to help with this.

The **PR service**, part of BitBake, is used to increment `PR` without human interaction every time BitBake detects a checksum change in a task. It injects a suffix in `PR` in the `${PR}.X` format. For example, if we have `PR = "r34"` after subsequent `PR` service interactions, the `PR` value becomes `r34.1`, `r34.2`, `r34.3`, and so on. The use of the `PR` service is critical for solid package feeds, as it requires the version to increase linearly.

> **Tip**
> Even though we ought to use the `PR` service to have solid package versioning, it does not preclude the need to set `PR` manually in exceptional cases.

By default, the `PR` service is not enabled or running. We can enable it to run locally by adding the `PRSERV_HOST` variable in the BitBake configuration – for example, in `build/conf/local.conf`, as in the following:

```
PRSERV_HOST = "localhost:0"
```

Figure 7.8 – How to configure a PR service to run locally

This approach is adequate when the build happens on a single computer, which builds every package of the package feed. BitBake starts and stops the server at each build and automatically increases the required `PR` values.

For a more complex setup, with multiple computers working against a shared package feed, we must have a single `PR` service running, used by all building systems associated with the package feed. In this case, we need to start the `PR` service in the server using the `bitbake-prserv` command, shown as follows:

```
bitbake-prserv --host <ip> --port <port> --start
```

Figure 7.9 – The command line to initiate the PR service server

In addition to manually-starting the service, we need to update the BitBake configuration file (for example, `build/conf/local.conf`) of each build system, which connects to a server using the `PRSERV_HOST` variable, as described earlier, so that each system points to the server `IP` and `port`.

Using package feeds

To use package feeds, the following two components are required:

- The server provides access to the packages
- The client accesses the server and downloads the required packages

The set of packages offered by the package feed is determined by the recipes we build. We can build one or more recipes and offer them, or build a set of images to generate the desired packages. Once satisfied with the packages offered, we must create the package index provided by the package feeds. The following command performs this:

```
bitbake package-index
```

Figure 7.10 – The command line to create the package index

The packages are available inside the `build/tmp/deploy` directory. We must choose the respective sub-directory depending on the package format chosen. Poky uses RPM by default, so we must serve the content of the `build/tmp/deploy/rpm` directory.

> **Tip**
> Make sure to run `bitbake package-index` after building all packages; otherwise, the package index will not include them.

The package index and packages must be made available through a transfer protocol such as HTTP. We can use any server we wish for this task, such as Apache, Nginx, and Lighttpd. A convenient way to make the packages available through HTTP for local development is by using the Python simple HTTP server, shown as follows:

```
cd build/tmp/deploy/rpm
python3 -m http.server 5678
```

Figure 7.11 – How to provide the package feed by using the Python simple HTTP server

To add support for package management to the image, we have a couple of changes to make. We need to add `package-management` in `EXTRA_IMAGE_FEATURES` and set the URI for package fetching on `PACKAGE_FEED_URIS`. For example, we can add this to our `build/conf/local.conf`:

```
PACKAGE_FEED_URIS = "http://my-ip-address:5678"
EXTRA_IMAGE_FEATURES += "package-management"
```

Figure 7.12 – How to configure a remote package feed

We will detail the `IMAGE_FEATURES` and `EXTRA_IMAGE_FEATURES` variables in *Chapter 12*, *Creating Custom Layers*. If we want a small image with no package management support, we should omit `package-management` from `EXTRA_IMAGE_FEATURES`.

The `PACKAGE_FEED_URIS` and `EXTRA_IMAGE_FEATURES` configurations guarantee that the image on the client side can access the server and has the utilities needed to install, remove, and upgrade its packages. After these steps, we can use the runtwime package management in the target device.

For example, if we choose the RPM package format for the image, we can fetch the repository information using the following command:

```
dnf check-update
```

Figure 7.13 – Command line to fetch the package feed repository

Use the `dnf search <package>`, and `dnf install <package>` commands to find and install packages from the repositories.

Depending on the package format chosen, the commands for the target to update the package index, search for, and install a package are different. See the available command lines for each package format in the following table:

Package format	**RPM**	**IPK**	**DEB**
Update the package index	`dnf check-updates`	`opkg update`	`apt-get update`
Search for a package	`dnf search <package>`	`opkg search <package>`	`apt-cache search <package>`
Install a package	`dnf install <package>`	`opkg install <package>`	`apt-get install <package>`
System upgrade	`dnf upgrade`	`opkg upgrade`	`apt-get dist-upgrade`

Table 7.1 – A package management command comparison

The use of package feeds are great to use in a local development phase because they enable us to install packages in an already deployed image.

> **Note**
> The use of package feeds for system upgrades in the field requires a huge test effort to guarantee that a system does not fall into a broken state. The testing effort is enormous to verify all different upgrade scenarios. Usually, full image upgrades are safer for production use.

The management of a package feed is much more complex. It involves several other aspects, such as package dependency chains and different upgrade scenarios. Creating a complex package feed external server is out of this book's scope, so please refer to the Yocto Project documentation for further details.

Summary

This chapter presented the basic concepts of packaging, which has a significant role in Poky and BitBake (package versioning), and how this impacts Poky's behavior when rebuilding packages and package feeds. It also showed us how to configure an image to be updated using prebuilt packages provided by a remote server.

In the next chapter, we will learn about the BitBake metadata syntax and its operators and how to append, prepend, and remove content from variables, variable expansions, and so on. We will then be able to better understand the language used in Yocto Project engines.

8
Diving into BitBake Metadata

At this point in this book, we know how to generate images and packages and how to use package feeds – basically, everything we must know for the simple usage of Poky. Hereafter, we will learn how to control the behavior of Poky to accomplish our goals and achieve maximum benefit from the Yocto Project as a whole.

This chapter will help enhance our understanding of the BitBake metadata syntax. We will learn to use the BitBake operators to alter the content of variables, variable expansions, and so on. These are the key concepts we can use to make our recipes and the customization that we will learn about in the following chapters.

Understanding BitBake's metadata

The amount of metadata used by BitBake is enormous. Therefore, to get the maximum benefit from Poky, we must master it. As we learned in *Chapter 4*, *Meeting the BitBake Tool*, metadata covers three major areas:

- **Configuration** (the `.conf` files): The configuration files define the global content that configures how the classes and recipes will work.
- **Classes** (the `.bbclass` files): Classes can be inherited for easier maintenance and to promote code reuse and avoid code duplication.
- **Recipes** (the `.bb` or `.bbappend` files): The recipes describe the tasks to be run and provide the required information to allow BitBake to generate the required task chain. They are the most commonly used metadata, as they define the variables and tasks for the recipes. The most common types of recipes generate packages and images.

The classes and recipes use a mix of Python and Shell Script code, which is parsed by BitBake, generating a massive number of tasks and local states that must still be executed after being parsed.

We will also learn about the operators and essential concepts we need to build our recipes.

Working with metadata

The syntax used by BitBake metadata can be misleading and sometimes hard to trace. However, we can check the value of each variable in BitBake-generated, pre-processed recipe data by using the `bitbake` option (`-e` or `--environment`), as follows:

```
bitbake -e <recipe> | grep <variable>
```

Figure 8.1 – How to display the BitBake environment

To understand how BitBake works, please refer to *BitBake User Manual* (`https://docs.yoctoproject.org/bitbake/2.0`). The following sections will show most of the syntax commonly used in recipes.

The basic variable assignment

The assignment of a variable can be done as shown here:

```
FOO = "bar"
```

Figure 8.2 – An example of a variable assignment

In the preceding example, the value of the `FOO` variable is assigned to `bar`. Variable assignment is core to the BitBake metadata syntax, as most examples use variables.

The variable expansion

BitBake supports variable referencing. The syntax closely resembles Shell Script, such as the following:

```
A = "aValue"
B = "before-${A}-after"
```

Figure 8.3 – An example of variable expansion

The preceding example results in `A` containing `aValue` and `B` containing `before-aValue-after`. An important thing to bear in mind is that the variable only expands when it is used, as shown here:

```
1 A = "aOriginalValue"
2 B = "before-${A}-after"
3 A = "aNewValue"
```

Figure 8.4 – The variables are only expanded when used

Figure 8.4 illustrates the *lazy evaluation* used by BitBake evaluation. The `B` variable value is `before-${A}-after` until a task requires the variable value. The `A` variable has been assigned to `aNewValue` in line 3; consequently, `B` evaluates `before-aNewValue-after`.

Assigning a value if the variable is unassigned, using ?=

When there is a need to assign a variable only if the variable is still unassigned, the `?=` operator can be used. The following code shows its use:

```
1 A ?= "value"
```

Figure 8.5 – An example of value

The same behavior happens if there are multiple `?=` assignments to a single variable. The first use of the `?=` operator is responsible for assigning the variable. Let's look at the following example:

```
1 A ?= "value"
2 A ?= "ignoredAsAlreadyAssigned"
```

Figure 8.6 – An example of a second assignment being ignored

The `A` variable has been assigned to `value` on line 1, before the assignment of `ignoredAsAlreadyAssigned` on line 2, which is ignored.

We need to consider that the `=` operator is stronger than the `?=` operator, as it assigns the value independently of the previous variable state, as shown here:

```
1 A ?= "initialValue"
2 A  = "changeValue"
```

Figure 8.7 – An example showing that the ?= operator is weaker than the = operator

Hence, the `A` variable is assigned as `changeValue`.

Assigning a default value using ??=

Using the `??=` operator is intended to provide a default value for a variable and is a weaker version of the `?=` operator.

Check out the following code:

```
1 A ??= "firstValue"
2 A ??= "secondValue"
```

Figure 8.8 – An example of how default values are assigned

In line 1, the `A` default value is assigned to `firstValue`, and then in line 2, the `A` default value is changed to `secondValue`. As no other assignment is made to the `A` variable, the final value is `secondValue`.

`?=` is an assignment operator, as seen before, and takes precedence over the `??=` operator, as can be seen in the following example:

```
1 A ??= "firstValue"
2 A ??= "secondValue"
3 A  ?= "thirdValue"
4 A ??= "fourthValue"
```

Figure 8.9 – An example of the ??= operator being weaker than the ?= operator

The final value of `A` variable is `thirdValue`, as no assignment has been made until line 3.

The immediate variable expansion

The `:=` operator is used when there is a need to force the immediate expansion of a variable. It results in the variable's contents being expanded immediately rather than when the variable is used, as follows:

```
1 A  = "aValue"
2 B := "${A}-after"
3 A  = "newValue"
4 C  = "${A}"
```

Figure 8.10 – An example of immediate variable expansion

The value for `B` is assigned immediately, in line 2, and expands to `aValue-after`. However, the value for `C` is only assigned when used and then set to `newValue`, as `A` value has been set in line 3.

The list appending and prepending

The `+=` operator, known as *list appending*, adds a new value after the original one, separated with a space, as shown here:

```
A  = "originalValue"
A += "appendedValue"
```

Figure 8.11 – An example of list appending

In this example, the final value for `A` is `originalValue appendedValue`.

The `=+` operator, known as *list prepending*, adds a new value before the original one, separated with a space, as shown here:

```
A =  "originalValue"
A =+ "prependedValue"
```

Figure 8.12 – An example of list prepending

In this example, the final value for `A` is `prependedValue originalValue`.

The string appending and prepending

The `.=` operator, known as *string appending*, adds a new value after the original one, with no extra space, as shown here:

```
A  = "originalValue"
A .= "AppendedValue"
```

Figure 8.13 – An example of string appending

In this example, the final value for `A` is `originalValueAppendedValue`.

The `=.` operator, known as *string prepending*, adds the new value before the original one with no extra space, as shown here:

```
A =  "OriginalValue"
A =. "prependedValue"
```

Figure 8.14 – An example of string prepending

In this example, the final value for `A` is `prependedValueOriginalValue`.

The :append and :prepend operators

The `:append` operator adds a new value after the original with no extra space, as shown here:

```
A = "originalValue"
A:append = "AppendedValue"
```

Figure 8.15 – An example of how to use the :append operator

In this example, the final value for `A` is `originalValueAppendedValue`.

The `:prepend` operator adds the new value before the original with no extra space, as shown here:

```
A = "OriginalValue"
A:prepend = "prependedValue"
```

Figure 8.16 – An example of how to use the :prepend operator

In this example, the final value for `A` is `prependedValueOriginalValue`.

You may have noticed that the `:append` and `:prepend` operators resemble the string appending (`.=`) and prepending (`=.`) operators. Still, there is a subtle difference between how the `:append` and `:prepend` operators and the string appending and string prepending operators are parsed, as shown here:

```
1 A:append = "AppendedValue"
2 A   = "value"
3 B .= "AppendedValue"
4 B   = "value"
```

Figure 8.17 – An example of the difference between :append and the .= operator

Using the `:append` operator queues the operation for execution, which happens after the *line 2* assignment, resulting in `A` becoming `valueAppendedValue`. The `.=` operator is immediate, so the assignment of *line 4* replaces the value set on *line 3*, resulting in `B` becoming `value`.

The list item removal

The `:remove` operator drops a list item from the original content. For example, see the following:

```
A = "value1 value2 value3"
A:remove = "value2"
```

Figure 8.18 – An example of how to use the :remove operator

In this example, `A` is now `value1 value3`. The `:remove` operator considers the variable value as a list of strings separated by spaces so that the operator can remove one or more items from the list. Note that every appending and prepending operation has already finished when `:remove` is executed.

Conditional metadata sets

BitBake provides a very easy-to-use way to write conditional metadata through a mechanism called **overrides**.

The `OVERRIDES` variable contains values separated by colons (`:`) and evaluated from left to right. Each value is an item that we want to have conditional metadata.

Let's consider the next example:

```
OVERRIDES = "linux:arm:mymachine"
```

Figure 8.19 – An example of the OVERRIDES variable

The `linux` override is less specific than `arm` and `mymachine`. The following example shows how we can use `OVERRIDES` to set the `A` variable conditionally:

```
OVERRIDES = "linux:arm:mymachine"
A = "value"
A:linux = "linuxSpecificValue"
A:other = "otherConditionalValue"
```

Figure 8.20 – An example of using OVERRIDES conditional setting

In this example, `A` will be `linuxSpecificValue`, due to the condition of `linux` being in `OVERRIDES`.

Conditional appending

BitBake also supports appending and prepending variables, based on whether something is in `OVERRIDES`, as shown in the following example:

```
OVERRIDES = "linux:arm:mymachine"
A = "value"
A:append:arm = " armValue"
A:append:other = " otherValue"
```

Figure 8.21 – An example of using OVERRIDES conditional appending

In the preceding example, `A` is set to `value armValue`.

File inclusion

BitBake provides two directives for file inclusion – `include` and `require`.

With the `include` keyword, BitBake attempts to insert the file at the keyword location, so it is optional. Let's suppose the path specified on the `include` line is relative; then, BitBake locates the first instance it can find within `BBPATH`. By contrast, the `require` keyword raises `ParseError` if the required file cannot be found.

> **Tip**
> The convention generally adopted in the Yocto Project is to use a `.inc` file to share the common code between two or more recipe files.

Python variable expansion

BitBake makes it easy to use Python code in variable expansion with the following syntax:

```
A = "${@<python-command>}"
```

Figure 8.22 – An example of Python expansion syntax

This gives enormous flexibility to a user. We can see a Python function call in the following example:

```
A = "${@time.strftime('%Y%m%d', time.gmtime())}"
```

Figure 8.23 – An example of a Python command to print the current date

This results in the `A` variable containing today's date.

Defining executable metadata

Metadata recipes (`.bb`) and class files (`.bbclass`) can use Shell Script code, as follows:

```
do_mytask () {
    echo "Hello, world!"
}
```

Figure 8.24 – An example of a task definition

The task definition is identical to setting a variable, except that this variable happens to be an executable Shell Script code. When writing the task code, we should not use Bash or Zsh-specific features, as the tasks can only rely on POSIX-compatible features. When in doubt, an excellent way to test whether your code is safe is to use the Dash shell to try it out.

Another way to inject code is by using Python code, as shown here:

```
python do_printdate () {
    import time
    print(time.strftime('%Y%m%d', time.gmtime()))
}
```

Figure 8.25 – An example of a Python task definition

The task definition is similar, but it flags the task as Python so that BitBake knows how to run it accordingly.

Defining Python functions in a global namespace

When we need to generate a value for a variable or some other use, this can be quickly done in recipes (`.bb`) and classes (`.bbclass`) using code similar to the following:

```
def get_depends(d):
    if d.getVar('SOMECONDITION'):
        return "dependencyWithCondition"

    return "dependency"

SOMECONDITION = "1"
DEPENDS = "${@get_depends(d)}"
```

Figure 8.26 – A code example to handle variable values in Python code

Usually, we need to access the BitBake datastore when writing a Python function. Therefore, a convention among all metadata is the use of an argument called `d` to point to BitBake's datastore. It is usually in the last parameter of the function.

In *Figure 8.26*, we ask the datastore for the value of the `SOMECONDITION` variable in line 2 and return a value depending on it.

The example results in the value for the `DEPENDS` variable containing `dependencyWithConditon`.

The inheritance system

The `inherit` directive specifies which classes of functionality our recipe (`.bb`) offers a rudimentary inheritance mechanism, such as object-oriented programming languages. For example, we can abstract the tasks involved in using the Autoconf and Automake building tools and put them into the class for our recipes to reuse. A given `.bbclass` is located by searching for `classes/filename.bbclass` in `BBPATH`. So, in a recipe that uses Autoconf or Automake, we can use the following:

```
1 inherit autotools
```

Figure 8.27 – An example of how to inherit a class

Line 1 from *Figure 8.27* instructs BitBake to use `inherit autotools.bbclass`, providing the default tasks that work fine for most Autoconf- or Automake-based projects.

Summary

In this chapter, we learned in detail about the BitBake metadata syntax, its operators to manipulate variable contents, and variable expansions, including some usage examples.

In the next chapter, we will learn how to use Poky to create external compilation tools and produce a `root` filesystem suitable for target development.

9
Developing with the Yocto Project

So far in this book, we have used Poky as a build tool. In other words, we have used it as a tool to design and generate the image delivered to products.

In this chapter, we will see how to set up a development environment for use inside the target and meet the **Standard SDK** and **Extensible SDK** tools, which can help us develop applications outside the target. For example, they allow us to cross-compile applications, recipes, and images.

What is a software development kit?

In embedded development, the toolchain is often composed of cross-platform tools or tools executed on one architecture, which then produces a binary for use in another architecture – for example, a GCC tool that runs on an x86-64-compatible machine and generates binaries for an ARM machine is a cross-compiler. When a tool and the resulting binaries rely on dependencies from the same host on which the tool runs, this is commonly called a native build. Build and target architectures may be the same, but it is cross-compilation if the target binary uses a staged root filesystem to find its dependencies.

A **software development kit** (**SDK**) is a set of tools and files to develop and debug applications. These tools include compilers, linkers, debuggers, external libraries, headers, and binaries, also called a toolchain. It may also include extra utilities and applications. We can have two types of SDK:

- **Cross-development SDKs**: These have the goal of being used in the development host to generate binaries for the target
- **Native SDKs**: These aim to run on the target device

Generating a native SDK for on-device development

Some embedded devices are powerful enough to be used as a development environment. However, the resources needed for the build vary significantly from one library or application to another, so using the target as the building environment may not always be viable. The development image needs the following:

- The header files and libraries
- The toolchain

The following line adds these properties to an image:

```
IMAGE_FEATURES += "dev-pkgs tools-sdk"
```

Figure 9.1 – How to configure an image to include development artifacts

`IMAGE_FEATURES` in the preceding example extends the image functionality as follows:

- `dev-pkgs`: Installs development packages (headers and extra library links) for all packages installed in a given image
- `tools-sdk`: Installs the toolchain that runs on the device

The `IMAGE_FEATURES` variable is described in more detail in *Chapter 12*, *Creating Custom Layers*.

> **Tip**
> If we want to modify only `build/conf/local.conf`, the variable we should use is `EXTRA_IMAGE_FEATURES`.

The target can use this image during the application development cycle and share the image among all developers working on the same project. Each developer will have a copy, and the development team will use the same development environment consistently.

Understanding the types of cross-development SDKs

The Yocto Project can generate two types of cross-development SDKs that aim to cover different needs. They are defined as follows:

- **Standard SDK**: This provides the artifacts for application development, be it for bootloader or Linux kernel development, or some other user space software
- **Extensible SDK**: This allows the installation of extra packages inside the SDK's `sysroot` directory, as well as recipe and application integration inside a Yocto Project-controlled environment

The Standard SDK includes a toolchain and debugging applications. Its goal is to allow users to generate binaries for use in the target. The Extensible SDK is more powerful and can build images and recipes. A notable difference between the two types of SDK is the presence of `devtool` in the Extensible SDK.

`devtool` is responsible for providing the additional features of the Extensible SDK. It is an interface for using BitBake and `recipetool`'s power. The `devtool` and `recipetool` commands are also available in the traditional Yocto Project environment.

Using the Standard SDK

Usually, an SDK has a set of libraries and applications it must provide, which is defined in an image tailored to the product. These are called image-based SDKs. For example, we can generate the Standard SDK for `core-image-full-cmdline` with the following command:

```
bitbake core-image-full-cmdline -c populate_sdk
```

Figure 9.2 – How to generate the Standard SDK for core-image-full-cmdline

Another option is to create a generic SDK with the toolchain and debugging tools. This generic SDK is called `meta-toolchain` and is used mainly for Linux kernel and bootloader development and their debugging processes. It may not be sufficient to build applications with complex dependencies. To create `meta-toolchain`, use the following command:

```
bitbake meta-toolchain
```

Figure 9.3 – How to generate a generic SDK

In both cases, the resulting SDK self-installer files are at `build/tmp/deploy/sdk/`. Considering we used the Standard SDK for `core-image-full-cmdline`, we can see the following resulting set of files:

```
$ tree build/tmp/deploy/sdk/
build/tmp/deploy/sdk/
├── poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-4.0.4.host.manifest
├── poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-4.0.4.sh
├── poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-4.0.4.target.manifest
└── poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-4.0.4.testdata.json

0 directories, 4 files
```

Figure 9.4 – The resultant files after running bitbake core-image-full-cmdline -c populate_sdk

The next step after creating the Standard SDK is to install it, as the Standard SDK is wrapped in an installation script that can be executed in the same manner as any other script. The following sequence shows the Standard SDK installation process using the standard target directory:

```
$ cd build/tmp/deploy/sdk
$ ./poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-4.0.4.sh
Poky (Yocto Project Reference Distro) SDK installer version 4.0.4
=================================================================
Enter target directory for SDK (default: /opt/poky/4.0.4):
You are about to install the SDK to "/opt/poky/4.0.4". Proceed [Y/n]?
[sudo] password for user:
Extracting
SDK.............................................................................................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup
script e.g.
 $ . /opt/poky/4.0.4/environment-setup-core2-64-poky-linux
```

Figure 9.5 – The Standard SDK installation process

The preceding Standard SDK illustrates how we can generate and install a Standard SDK. Still, it is not ideal to use a standard image that is not tailored to your current needs. Therefore, creating a custom image that fits our application needs is highly recommended. It is also recommended to base the Standard SDK on this custom image.

The Standard SDK is generated to match the machine architecture we set using the `MACHINE` variable. To use the Standard SDK to build a custom application, for example, `hello-world.c`, we can use the following lines, targeting the x86-64 architecture:

```
$ source /opt/poky/4.0.4/environment-setup-core2-64-poky-linux
$ ${CC} hello-world.c -o hello-world
$ file hello-world
hello-world: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib/ld-linux-x86-64.so.2,
BuildID[sha1]=244b01aa354611d25ab2a8999b5428455fb90206, for GNU/Linux 3.2.0, with
debug_info, not stripped
```

Figure 9.6 – The steps to build a C application using the Standard SDK

Another very commonly used project is the Linux kernel. When we want to build the Linux kernel source code, we can use the following sequence of commands:

```
$ source /opt/poky/4.0.4/environment-setup-core2-64-poky-linux
$ unset LDFLAGS
$ make defconfig
$ make bzImage
```

Figure 9.7 – The steps to build the Linux kernel using the Standard SDK

`unset LDFLAGS` is required to avoid using GCC for linking, which is the Yocto Project-based Standard SDK's default.

Using the Extensible SDK

The Extensible SDK expands the functionalities of the Standard SDK. Some of the significant capabilities included are as follows:

- Generate recipes
- Build recipes
- Build images
- Install packages in the internal toolchain
- Deploy packages to the target

Those additional features are provided by the `devtool` utility, which is also available in the standard Yocto Project environment.

To generate the Extensible SDK, use the following command:

```
bitbake core-image-full-cmdline -c populate_sdk_ext
```

Figure 9.8 – Command to generate the Extensible SDK

The resulting files are in `build/tmp/deploy/sdk/`. Considering we used the Extensible SDK

for `core-image-full-cmdline`, we see the following set of files:

```
$ build/tree tmp/deploy/sdk/
build/tmp/deploy/sdk/
├── poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-ext-4.0.4.host.manifest
├── poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-ext-4.0.4.sh
├── poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-ext-4.0.4.target.manifest
├── poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-ext-4.0.4.testdata.json
├── x86_64-buildtools-nativesdk-standalone-4.0.4.host.manifest
├── x86_64-buildtools-nativesdk-standalone-4.0.4.sh
├── x86_64-buildtools-nativesdk-standalone-4.0.4.target.manifest
└── x86_64-buildtools-nativesdk-standalone-4.0.4.testdata.json

0 directories, 8 files
```

Figure 9.9 – The resultant files after running bitbake core-image-full-cmdline -c populate_sdk_ext

The next step after creating the Extensible SDK is to install it. To install it, we can execute the generated script. The following sequence shows the Extensible SDK installation process using the standard target directory:

```
$ ./tmp/deploy/sdk/poky-glibc-x86_64-core-image-full-cmdline-core2-64-qemux86-64-toolchain-ext-4.0.4.sh
Poky (Yocto Project Reference Distro) Extensible SDK installer version 4.0.4
============================================================================
Enter target directory for SDK (default: ~/poky_sdk):
You are about to install the SDK to "/home/user/poky_sdk". Proceed [Y/n]?
Extracting SDK......................................................done
Setting it up...
Extracting buildtools...
Preparing build system...
Loading cache: 100% |                                                                   | ETA:  --:--:--
Parsing recipes: 100% |#################################################################| Time: 0:01:34
Initialising tasks: 100% |##############################################################| Time: 0:00:04
Checking sstate mirror object availability: 100% |######################################| Time: 0:00:01
Running tasks (468 of 1435, 0 of 3706)   0% |
Loading cache: 100% |###################################################################| Time: 0:00:00
Initialising tasks: 100% |##############################################################| Time: 0:00:00
done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script
e.g.
 $ . /home/user/poky_sdk/environment-setup-core2-64-poky-linux
```

Figure 9.10 – The Extensible SDK installation process

The preceding screenshot illustrates how we can generate and install an Extensible SDK. Still, it is not ideal to use a standard image that is not tailored to your current needs. Therefore, creating a custom image that fits your application needs is highly recommended, as is basing the Extensible SDK on one. However, we can build and install any extra dependencies into the SDK using the Extensible SDK.

In our case, we installed the Extensible SDK in `/home/user/poky_sdk`. After the installation has been completed, the next step is to use the provided script to export the required environment variables, which enables the Extensible SDK's use, with the following command:

```
$ . /home/user/poky_sdk/environment-setup-core2-64-poky-linux
SDK environment now set up; additionally you may now run devtool to perform development tasks.
Run devtool --help for further details.
```

Figure 9.11 – Exporting the environment variables to allow the Extensible SDK to be used

In the following sections, we will cover some use cases using `devtool`. All commands are executed inside a terminal with the Extensible SDK variables exported.

The Extensible SDK is a different way to deliver the same Yocto Project tools and metadata. It wraps together the following:

- A basic set of binaries for the Yocto Project environment execution
- A Standard SDK for development
- A shared state cache to reduce local builds
- A snapshot of the Yocto Project metadata and configuration

Essentially, the Extensible SDK is a snapshot of the environment used to create it. Therefore, all `devtool` commands, including those we will use in the following sections, are available inside the Yocto Project environment.

Building an image using devtool

Let's start by creating an image. The Extensible SDK is capable of creating any supported image. For example, to create `core-image-full-cmdline`, we can use the following command line:

```
$ devtool build-image core-image-full-cmdline
NOTE: Starting bitbake server...
Loading cache: 100% |########################################################################| Time: 0:00:00
Loaded 1641 entries from dependency cache.

Summary: There was 0 WARNING message.
WARNING: No packages to add, building image core-image-full-cmdline unmodified
Loading cache: 100% |########################################################################| Time: 0:00:01
Loaded 1641 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |###################################################################| Time: 0:00:10
Checking sstate mirror object availability: 100% |##########################################| Time: 0:00:01
Sstate summary: Wanted 560 Local 0 Mirrors 0 Missed 560 Current 869 (0% match, 60% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3679 tasks of which 3666 didn't need to be rerun and all succeeded.

Summary: There was 0 WARNING message.
INFO: Successfully built core-image-full-cmdline. You can find output files in /home/user/poky_sdk/tmp/deploy
/images/qemux86-64
```

Figure 9.12 – Building core-image-full-cmdline with devtool

After running the `devtool` command, the generated files can be found in `/home/user/poky_sdk/tmp/deploy/images/qemux86-64`.

Running an image on QEMU

We can emulate the target hardware with QEMU using the previously built image, `core-image-full-cmdline`, with the following command:

```
$ devtool runqemu core-image-full-cmdline
NOTE: Starting bitbake server...
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)...
NOTE: Reconnecting to bitbake server...
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)...
NOTE: Retrying server connection (#1)...
NOTE: Starting bitbake server...
runqemu - INFO - Continuing with the following parameters:
KERNEL: [/home/user/poky_sdk/tmp/deploy/images/qemux86-64/bzImage--
5.15.68+git0+1128d7bcdc_0e51e57170-r0-qemux86-64-20221003235719.bin]
MACHINE: [qemux86-64]
FSTYPE: [ext4]
ROOTFS: [/home/user/poky_sdk/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.ext4]
CONFFILE: [/home/user/poky_sdk/tmp/deploy/images/qemux86-64/core-image-full-cmdline-
qemux86-64.qemuboot.conf]

runqemu - INFO - Setting up tap interface under sudo
[sudo] password for user:
runqemu - INFO - Network configuration:
ip=192.168.7.2::192.168.7.1:255.255.255.0::eth0:off:8.8.8.8
runqemu - INFO - Running /home/user/poky_sdk/tmp/work/x86_64-linux/qemu-helper-native/1.0-
r1/recipe-sysroot-native/usr/bin/qemu-system-x86_64 -device virtio-net-
pci,netdev=net0,mac=52:54:00:12:34:02 -netdev tap,id=net0,ifname=tap0,script=no,downscript=no
-object rng-random,filename=/dev/urandom,id=rng0 -device virtio-rng-pci,rng=rng0 -drive file=/home
/user/poky_sdk/tmp/deploy/images/qemux86-64/core-image-full-cmdline-
qemux86-64.ext4,if=virtio,format=raw -usb -device usb-tablet   -cpu IvyBridge -machine q35 -smp 4
-m 256 -serial mon:vc -serial null -device virtio-vga  -display sdl,show-cursor=on  -kernel
/home/user/poky_sdk/tmp/deploy/images/qemux86-64/bzImage--5.15.68+git0+1128d7bcdc_0e51e57170-r0-
qemux86-64-20221003235719.bin -append 'root=/dev/vda rw
ip=192.168.7.2::192.168.7.1:255.255.255.0::eth0:off:8.8.8.8 oprofile.timer=1 tsc=reliable
no_timer_check rcupdate.rcu_expedited=1 '

runqemu - INFO - Host uptime: 4304.39
```

Figure 9.13 – Emulating with devtool and QEMU

It starts the QEMU execution and generates the boot splash, as is shown in the following screenshot:

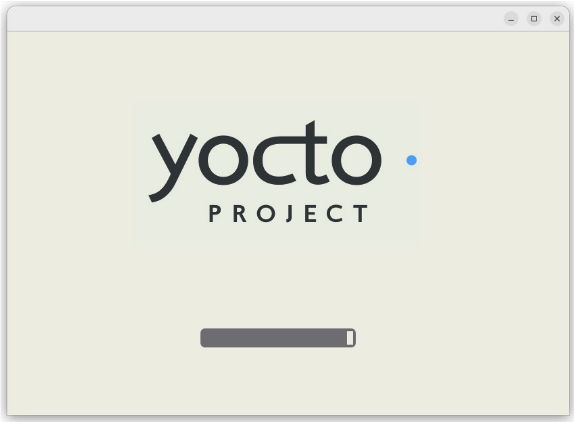


Figure 9.14 – The QEMU boot splash

Creating a recipe from an external Git repository

`devtool` is also capable of producing a recipe from an external Git repository. Here, we are going to use `https://github.com/OSSystems/bbexample`:

```
$ devtool add https://github.com/OSSystems/bbexample
NOTE: Starting bitbake server...
NOTE: Starting bitbake server...
INFO: Fetching git://github.com/OSSystems/bbexample;protocol=https;branch=master...
Loading cache: 100% |##########################################################| Time: 0:00:01
Loaded 1641 entries from dependency cache.
Parsing recipes: 100% |########################################################| Time: 0:00:00
Parsing of 883 .bb files complete (882 cached, 1 parsed). 1642 targets, 44 skipped, 0 masked, 0 errors.

Summary: There was 0 WARNING message.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#####################################################| Time: 0:00:02
Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 0 (0% match, 0% complete)
NOTE: No setscene tasks
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 2 tasks of which 0 didn't need to be rerun and all succeeded.
INFO: Using default source tree path /home/user/poky_sdk/workspace/sources/bbexample
NOTE: Reconnecting to bitbake server...
NOTE: Previous bitbake instance shutting down?, waiting to retry...
NOTE: Retrying server connection (#1)...
NOTE: Reconnecting to bitbake server...
NOTE: Reconnecting to bitbake server...
NOTE: Previous bitbake instance shutting down?, waiting to retry...
NOTE: Previous bitbake instance shutting down?, waiting to retry...
NOTE: Retrying server connection (#1)...
NOTE: Retrying server connection (#1)...
NOTE: Starting bitbake server...
INFO: Recipe /home/user/poky_sdk/workspace/recipes/bbexample/bbexample_git.bb has been automatically
created; further editing may be required to make it fully functional
```

Figure 9.15 – Creating the recipe using devtool

`devtool` creates a basic recipe file for the given repository. It creates a workspace with the package source code and the needed metadata. The file structure used by `devtool`, after the `devtool add https://github.com/OSSystems/bbexample` command is run, is as follows:

```
$ tree /home/user/poky_sdk/workspace/
/home/user/poky_sdk/workspace/
├── appends
│   └── bbexample_git.bbappend
├── conf
│   └── layer.conf
├── README
├── recipes
│   └── bbexample
│       └── bbexample_git.bb
└── sources
    └── bbexample
        ├── autogen.sh
        ├── bbexample.c
        ├── bbexample.h
        ├── bbexamplelib.c
        ├── configure.ac
        ├── LICENSE
        ├── Makefile.am
        └── README.md

6 directories, 12 files
```

Figure 9.16 – The file structure created by devtool when creating a recipe

Currently, `devtool` generates a tentative recipe for projects based on the following:

- Autotools (`autoconf` and `automake`)
- CMake
- Scons
- `qmake`
- A plain `Makefile`
- The Node.js module
- Python modules that use `setuptools` or `distutils`

Building a recipe using devtool

Now that the recipe has been created under the workspace directory, we can build it with the following command:

```
$ devtool build bbexample
NOTE: Starting bitbake server...
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)...
Loading cache: 100% |############################################################| Time: 0:00:00
Loaded 1641 entries from dependency cache.
Parsing recipes: 100% |##########################################################| Time: 0:00:00
Parsing of 883 .bb files complete (882 cached, 1 parsed). 1642 targets, 44 skipped, 0 masked, 0 errors.

Summary: There was 0 WARNING message.
Loading cache: 100% |############################################################| Time: 0:00:01
Loaded 1641 entries from dependency cache.
Parsing recipes: 100% |##########################################################| Time: 0:00:00
Parsing of 883 .bb files complete (882 cached, 1 parsed). 1642 targets, 44 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#######################################################| Time: 0:00:01
Sstate summary: Wanted 91 Local 0 Mirrors 0 Missed 91 Current 57 (0% match, 38% complete)
NOTE: Executing Tasks
NOTE: bbexample: compiling from external source tree /home/user/poky_sdk/workspace/sources/bbexample
NOTE: Tasks Summary: Attempted 639 tasks of which 577 didn't need to be rerun and all succeeded.

Summary: There was 0 WARNING message.
```

Figure 9.17 – Building a recipe with devtool

Deploying to the target using devtool

After building the package with `devtool`, we can deploy it to the target. In our example, the target is the running QEMU. To access it, use the default QEMU IP address, `192.168.7.2`, as shown in the following command:

```
$ devtool deploy-target bbexample root@192.168.7.2
NOTE: Starting bitbake server...
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)...
Loading cache: 100% |#############################################################| Time: 0:00:00
Loaded 1641 entries from dependency cache.
Parsing recipes: 100% |###########################################################| Time: 0:00:00
Parsing of 883 .bb files complete (882 cached, 1 parsed). 1642 targets, 44 skipped, 0 masked, 0 errors.

Summary: There was 0 WARNING message.
The authenticity of host '192.168.7.2 (192.168.7.2)' can't be established.
ED25519 key fingerprint is SHA256:qItHt/Ydwq/Zr2tmmF50mKyztXHiRSRbgcNbejfVjaQ.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])?
INFO: Successfully deployed /home/user/poky_sdk/tmp/work/core2-64-poky-linux/bbexample/0.1+git999-r0/image
```

Figure 9.18 – Deploying to the target using devtool

The application is installed in the target. We can see `bbexample` being executed in the QEMU target, as shown in the following screenshot:

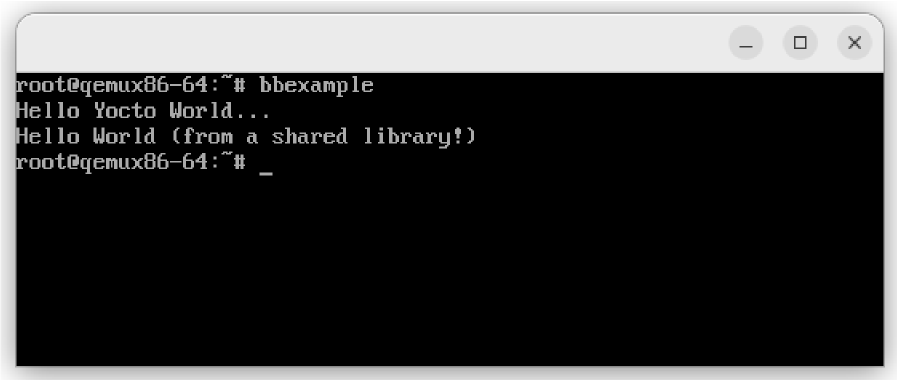


Figure 9.19 – bbexample executing on the target

Extending the SDK

One of the goals of the Extensible SDK is to allow us to install different recipes in the SDK environment. For example, to have `libusb1` available, we can run the following command:

```
$ devtool sdk-install -s libusb1
NOTE: Starting bitbake server...
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)...
Loading cache: 100% |############################################################| Time: 0:00:00
Loaded 1641 entries from dependency cache.
Parsing recipes: 100% |##########################################################| Time: 0:00:00
Parsing of 883 .bb files complete (882 cached, 1 parsed). 1642 targets, 44 skipped, 0 masked, 0 errors.

Summary: There was 0 WARNING message.
INFO: Installing libusb1...
Loading cache: 100% |############################################################| Time: 0:00:01
Loaded 1641 entries from dependency cache.
Parsing recipes: 100% |##########################################################| Time: 0:00:00
Parsing of 883 .bb files complete (882 cached, 1 parsed). 1642 targets, 44 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#######################################################| Time: 0:00:01
Checking sstate mirror object availability: 100% |###############################| Time: 0:00:00
Sstate summary: Wanted 153 Local 0 Mirrors 0 Missed 153 Current 97 (0% match, 38% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 984 tasks of which 738 didn't need to be rerun and all succeeded.

Summary: There were 0 WARNING messages.
INFO: Successfully installed libusb1
Loading cache: 100% |############################################################| Time: 0:00:01
Loaded 1641 entries from dependency cache.
Parsing recipes: 100% |##########################################################| Time: 0:00:00
Parsing of 883 .bb files complete (882 cached, 1 parsed). 1642 targets, 44 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |#######################################################| Time: 0:00:00
Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 0 (0% match, 0% complete)
NOTE: No setscene tasks
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3 tasks of which 0 didn't need to be rerun and all succeeded.

Summary: There was 0 WARNING message.
```

Figure 9.20 – The installation of a new recipe in the Extensible SDK

> **Tip**
>
> The Yocto Project Extensible SDK allows for distributed development, as developers can update and extend the existing SDK environment during a project's lifetime. There is some infrastructure setup required for the proper use of the Extensible SDK as a `sstate-cache` mirror and Extensible SDK server, which requires a complex configuration beyond the scope of this book. For more details, please refer to the *Providing Updates to the Extensible SDK After Installation* section of *Yocto Project Application Development and the Extensible Software Development Kit (eSDK)* (`https://docs.yoctoproject.org/4.0.4/sdk-manual/appendix-customizing.html#providing-updates-to-the-extensible-sdk-after-installation`).

Summary

In this chapter, we learned that the Yocto Project can be used for development and image creation. We learned how to create different types of toolchains and also how to use them.

In the next chapter, we will look at how we can configure Poky to help us in the debugging process, how we can configure our system to provide the required tools for remote debugging using GDB, and how we can track our changes using `buildhistory`.

10
Debugging with the Yocto Project

The debug process is an essential step in every development cycle. In this chapter, we will learn how to configure Poky to help us with the debugging process; for example, how we can configure our system to provide the tools needed for a remote debug using the **Gnu DeBugger** (**GDB**), how we can track our changes using `buildhistory`, and how we can use handy debug tools, such as `oe-pkgdata-util`, `bitbake-getvar`, and `devshell`.

Differentiating metadata and application debugging

Before we delve into the details of debugging, we need to realize that there are different types of debugging, such as metadata and runtime code debugging.

Metadata debugging is needed to ensure that the behavior of BitBake's tasks aligns with our goals and to identify the culprit when it's not aligned. For example, a recipe may need to be fixed to enable a feature. In such a case, we can use several log files generated by BitBake in the host to help trace the execution path of the involved task.

On the other hand, debugging runtime code is more natural as it is essentially the same as the typical development cycle of an application, a library, or a kernel. Depending on the issue we are seeking to resolve, the right tool to help may vary from a debugger to code instrumentation (for example, adding debug prints).

Tracking image, package, and SDK contents

The easiest way to ensure we have the image, packages, and **software development kit** (**SDK**), along with the expected contents, is to use the `buildhistory` mechanism.

When a recipe is updated for a new version or has its code changed, it may influence the contents of the generated packages and, consequently, the image or SDK.

Poky deals with many recipes and images or SDKs frequently have tens or hundreds of packages. Therefore, it may be challenging to track the package contents. The Poky tool that helps in this task is `buildhistory`.

`buildhistory`, as the name suggests, keeps a history of the contents of several artifacts built during the use of Poky. It tracks package, image, and SDK building and their contents.

To enable `buildhistory` in our system, we need to add the following lines of code in our `build/conf/local.conf` file:

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

Figure 10.1 – How to enable buildhistory support

The `INHERIT` method includes the `buildhistory` class hooks in the building process. At the same time, the `BUILDHISTORY_COMMIT` line enables BitBake to create a new Git commit in the `buildhistory` repository for every new package, image, or SDK build. The Git commit makes tracking as simple as using `git diff` between two commits. The data is stored under the `build/buildhistory` directory as text files for ease of use.

Poky provides a utility that outputs the difference between two `buildhistory` states, called `buildhistory-diff`, in a more concise way, which is very useful when checking for changes. The `buildhistory-diff` utility outputs the difference between any two Git revisions more meaningfully.

For example, suppose we add the `strace` package in the `core-image-minimal` image and build it. In that case, the `buildhistory-diff` command can be used to check the resultant changes, as in the following screenshot:

```
$ ../scripts/buildhistory-diff
Changes to images/qemux86_64/glibc/core-image-minimal (files-in-image.txt):
  /usr/bin/strace was added
  /usr/bin/strace-log-merge was added
Changes to images/qemux86_64/glibc/core-image-minimal (installed-package-names.txt):
  strace was added
```

Figure 10.2 – The result of buildhistory-diff

For every package build, `buildhistory` creates a list of generated sub-packages, installation scripts, a list of file ownership and sizes, the dependency relation, and more. In addition, the dependency relationship between the packages, filesystem files, and dependency graph is created for images and SDKs.

To better understand the capabilities and features provided by `buildhistory`, refer to *Maintaining Build Output Quality in Yocto Project Development Tasks Manual* (`https://docs.yoctoproject.org/4.0.4/dev-manual/common-tasks.html#maintaining-build-output-quality`).

Debugging packaging

In more sophisticated recipes, we split the installed contents into several sub-packages. The sub-packages can be optional features, modules, or any other set of files that is optional to install.

To inspect how the recipe's content has been split, we can use the `build/tmp/work/<arch>/<recipe name>/<software version>/packages-split` directory. It contains a sub-directory for every sub-package and has its contents in the sub-tree.

Among the possible reasons for a mistaken content split, we have defined the following:

- The contents not being installed (for example, an error in installation scripts)
- An application or library configuration error (for example, a disabled feature)
- Metadata errors (for example, the wrong package order)

Another common issue for build failure is lacking the required artifacts in the `sysroot` directory (for example, headers or dynamic libraries). The counterpart of the `sysroot` generation can be seen at `build/tmp/work/<arch>/<recipe_name>/<software_version>/sysroot-destdir`.

If this is not enough, we can instrument the task code with these logging functions to determine the logical error or bug that has caused the unexpected result.

Inspecting packages

A central aspect of the Yocto Project is dealing with the packages. Therefore, the project has designed `oe-pkgdata-util` to help us to inspect the built packages and related data. For example, after running `bitbake bluez5`, we can use the following command to find all the packages related to `bluez`:

```
$ oe-pkgdata-util list-pkgs | grep bluez
bluez5
bluez5-dbg
bluez5-dev
bluez5-noinst-tools
bluez5-obex
bluez5-ptest
bluez5-src
bluez5-testtools
```

Figure 10.3 – Listing all the available packages and filtering those related with bluez

Sometimes, we need to find the package that includes this specific file. We can inquire about the packages database using the following command:

```
$ oe-pkgdata-util find-path /usr/bin/rfcomm
bluez5: /usr/bin/rfcomm
```

Figure 10.4 – Finding which package provides /usr/bin/rfcomm

Another use case is when we need to find out the current version of a package. This can be done with the following command:

```
$ oe-pkgdata-util package-info bluez5
bluez5 5.65-r0 bluez5 5.65-r0 5507272
```

Figure 10.5 – Listing the package info for bluez5

We can also list all the files for the given package using the following command:

```
$ oe-pkgdata-util list-pkg-files bluez5
bluez5:
    /etc/bluetooth/input.conf
    /etc/bluetooth/network.conf
    /etc/dbus-1/system.d/bluetooth.conf
    /etc/init.d/bluetooth
    /usr/bin/bluemoon
    /usr/bin/bluetoothctl
    /usr/bin/btattach
    /usr/bin/btmon
    /usr/bin/ciptool
    /usr/bin/hciattach
    /usr/bin/hciconfig
    /usr/bin/hcidump
    /usr/bin/hcitool
    /usr/bin/hex2hcd
    /usr/bin/isotest
    /usr/bin/l2ping
    /usr/bin/l2test
    /usr/bin/mpris-proxy
    /usr/bin/rctest
    /usr/bin/rfcomm
    /usr/bin/sdptool
    /usr/lib/libbluetooth.so.3
    /usr/lib/libbluetooth.so.3.19.7
    /usr/libexec/bluetooth/bluetoothd
```

Figure 10.6 – Listing the files from the bluez5 package

The `oe-pkgdata-util` script is a handy tool to help us debug packaging.

Logging information during task execution

The logging utilities provided by BitBake are handy for tracing the code execution path. BitBake provides logging functions for use in Python and Shell Script code, described as follows:

- **Python**: For use within Python functions, BitBake supports several log levels such as `bb.fatal`, `bb.error`, `bb.warn`, `bb.note`, `bb.plain`, and `bb.debug`.
- **Shell Script**: For use in Shell Script functions, the same set of log levels exists and is accessed with a similar syntax: `bbfatal`, `bberror`, `bbwarn`, `bbnote`, `bbplain`, and `bbdebug`.

These logging functions are very similar to each other but have minor differences, described as follows:

- `bb.fatal` and `bbfatal`: These have the highest priority for logging messages as they print the message and terminate the processing. They cause the build to be interrupted.
- `bb.error` and `bberror`: These display an error but do not force the build to stop.
- `bb.warn` and `bbwarn`: These warn the users about something.
- `bb.note` and `bbnote`: These add a note to the user. They are only informative.
- `bb.plain` and `bbplain`: These output a message.
- `bb.debug` and `bbdebug`: These add debugging information that is shown depending on the debug level used.

There is one subtle difference between using the logging functions in Python and Shell Script. The logging functions in Python are directly handled by BitBake, seen on the console, and stored in the execution log inside `build/tmp/log/cooker/<machine>`. When the logging functions are used in Shell Script, the information is outputted to an individual task log file, which is available in `build/tmp/work/<arch>/<recipe name>/<software version>/temp`.

Inside the temp directory, we can inspect the scripts for every task with the `run.<task>.<pid>` pattern and use the `log.<task>.<pid>` pattern for its output. Symbolic links point to the last log files using the `log.<task>` pattern. For example, we can check for `log.do_compile` to verify whether the right files were used during the build process.

The `build/tmp/work` directory is detailed in *Chapter 6*, *Detailing the Temporary Build Directory*.

Debugging metadata variables

To debug the metadata variables, we can use the `bitbake-getvar` script. It uses the BitBake internal data to get a specific variable value and its attribution history.

For example, to inspect the `PACKAGECONFIG` variable for the `procps` recipe, we can use the following command:

```
$ bitbake-getvar -r procps PACKAGECONFIG
#
# $PACKAGECONFIG [4 operations]
#   :append[pn-qemu-system-native] /home/user/yocto/poky/build/conf/local.conf:243
#     " sdl"
#   set /home/user/yocto/poky/meta/conf/documentation.conf:318
#     [doc] "This variable provides a means of enabling or disabling features of a recipe
#            on a per-recipe basis."
#   set /home/user/yocto/poky/meta/recipes-extended/procps/procps_3.3.17.bb:33
#     [_defaultval] "${@bb.utils.filter('DISTRO_FEATURES', 'systemd', d)}"
#   set /home/user/yocto/poky/meta/recipes-extended/procps/procps_3.3.17.bb:34
#     [systemd] "--with-systemd,--without-systemd,systemd"
# pre-expansion value:
#   "${@bb.utils.filter('DISTRO_FEATURES', 'systemd', d)}"
PACKAGECONFIG=""
```

Figure 10.7 – The result of bitbake-getvar -r procps PACKAGECONFIG

From *Figure 10.7*, we can see that `PACKAGECONFIG` at the end is empty. We can also see that `defaultval` was set to `"${@bb.utils.filter('DISTRO_FEATURES', 'systemd', d)}"` at line 33 from the `meta/recipes-extended/procps/procps_3.3.17.bb` file.

We can see the `procps` recipe lines 33 and 34 in the following screenshot:

```
33 PACKAGECONFIG ??= "${@bb.utils.filter('DISTRO_FEATURES', 'systemd', d)}"
34 PACKAGECONFIG[systemd] = "--with-systemd,--without-systemd,systemd"
```

Figure 10.8 - The procps recipe 33 and 34 lines

The `bitbake-getvar` script can be used to check whether a feature is enabled or to be sure a variable has been expanded as we expect.

Utilizing a development shell

A development shell can be a helpful tool when editing packages or debugging build failures. The following steps take place when we use `devshell`:

1. Source files are extracted into the working directory.
2. Patches are applied.
3. A new terminal is opened in the working directory.

All the environment variables needed for the build are available in the new terminal, so we can use commands such as `configure` and `make`. The commands execute just as if the build system were running them.

The following command is an example that uses `devshell` on a target named `linux-yocto`:

```
$ bitbake linux-yocto -c devshell
```

Figure 10.9 – Running devshell for the linux-yocto recipe

The command from *Figure 10.9* allows us to rework the Linux kernel source code, build it, and change its code as needed. In *Figure 10.10*, you can see the log after executing the `bitbake linux-yocto -c devshell` command:

```
$ bitbake linux-yocto -c devshell
Loading cache: 100%
|########################################################################################| Time: 0:00:00
Loaded 1641 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "2.0.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "x86_64-poky-linux"
MACHINE              = "qemux86-64"
DISTRO               = "poky"
DISTRO_VERSION       = "4.0.4"
TUNE_FEATURES        = "m64 core2"
TARGET_FPU           = ""
meta
meta-poky
meta-yocto-bsp       = "kirkstone:e81e703fb6fd028f5d01488a62dcfacbda16aa9e"

Initialising tasks: 100%
|###################################################################################| Time: 0:00:01
Sstate summary: Wanted 64 Local 64 Mirrors 0 Missed 0 Current 63 (100% match, 100% complete)
NOTE: Executing Tasks
Setscene tasks: 127 of 127
Currently  1 running tasks (578 of 579)  99%
|########################################################################## |
0: linux-yocto-5.15.68+gitAUTOINC+1128d7bcdc_0e51e57170-r0 do_devshell - 31s (pid 704732)
```

Figure 10.10 – The log for bitbake linux-yocto -c devshell

> **Note**
>
> It is crucial to remember that changes made inside `devshell` do not persist between builds; thus, we must be careful to record any critical change before leaving it.

Since we have the source at our disposal, we can use it to generate extra patches. A convenient way of doing that is using Git and `git format-patch` to create the patch to be included in the recipe afterward.

The following screenshot shows the `devshell` window open after calling the `devshell` task:

```
root@machine:~/yocto/poky/build/tmp/work-shared/qemux86-64/kernel-source# ls
arch     CREDITS        fs       Kbuild   LICENSES     net      security  virt
block    crypto         include  Kconfig  MAINTAINERS  README   sound
certs    Documentation  init     kernel   Makefile     samples  tools
COPYING  drivers        ipc      lib      mm           scripts  usr
```

Figure 10.11 – The list of files inside the WORKDIR directory

The `devshell` command is convenient for small tasks. But when a more involved change is needed, using an external toolchain or `devtool` might be a better option.

To include the generated patch in the recipe and make it persistent, see *Chapter 13*, *Customizing Existing Recipes*.

Using the GNU Debugger for debugging

While developing any project, from time to time, we end up struggling to understand subtle bugs. The GDB is available as a package in Poky. It is installed in SDK images by default, as was detailed in *Chapter 9*, *Developing with the Yocto Project*.

> **Note**
>
> To install debugging packages containing the debug symbols and tools in an image, add `IMAGE_FEATURES += "dbg-pkgs tools-debug"` in `build/conf/local.conf`.

Using the SDK or an image with the debugging packages and tools installed allows us to debug applications directly in the target, replicating the same development workflow we usually do on our machine.

The GDB may not be usable on some targets because of memory or disk space constraints. The main reason for this limitation is that the GDB needs to load the debugging information and the binaries of the debugging process before starting the debugging process.

To overcome these constraints, we can use `gdbserver`, included by default when using `tools-debug` in `IMAGE_FEATURES`. It runs on the target and doesn't load any debugging information from the debugged process. Instead, a GDB instance processes the debugging information on the build host. The host GDB sends control commands to `gdbserver` to control the debugged application, so the target does not need to have the debugging symbols installed.

However, we must ensure the host can access the binaries with their debugging information. Therefore, it is recommended that the target binaries are compiled with no optimization to facilitate the debugging process.

The process for using `gdbserver` and adequately configuring the host and target is detailed in the *Debugging With the GNU Project Debugger (GDB) Remotely section in Yocto Project Development Tasks Manual* (`https://docs.yoctoproject.org/4.0.4/dev-manual/common-tasks.html#debugging-with-the-gnu-project-debugger-gdb-remotely`).

Summary

In this chapter, we learned how to configure Poky to help us with the debugging process. We learned about the contents of deployed directories that can be used for debugging and how we can track our changes using `buildhistory`. We also covered the use of `oe-pkgdata-util` to inspect package information, use `bitbake-getvar` to debug variable expansion, how we can use `devshell` to emulate the same build environment found by BitBake, and how we configure our system to provide the tools needed for GDB debugging.

In the next chapter, we will learn how to expand the Poky source code using external layers. First, we will introduce the concept of layering. Then, we will learn in detail about the directory structure and the content of each layer type.

11
Exploring External Layers

One of the most charming features of Poky is the flexibility of using external layers. In this chapter, we will examine why this is a vital capability and how we can take advantage of it. We will also look at the different types of layers and their directory trees layout. Finally, at the end of this chapter, we will learn to include a new layer in our project.

Powering flexibility with layers

Poky contains metadata spread over configuration definition files such as machine and distro files, classes, and recipes, covering everything from simple applications to full graphical stacks and frameworks. There are multiple places that BitBake can load metadata collection from, which are known as metadata layers.

The biggest strength of using layers is the ability to split metadata into logical units, which enables users to pick only the metadata collection needed for a project.

Using metadata layers enables us to do the following:

- Improve code reuse
- Share and scale work across different teams, communities, and vendors
- Increase the Yocto Project community's code quality, as multiple developers and users focus together on a particular metadata layer that is of interest to them

We can configure the system for different reasons, such as the need to enable/disable a feature or change build flags to enable architecture-specific optimizations. These are examples of customizations that can be done using layers.

In addition, we should organize metadata in different layers instead of creating our custom project environment, changing recipes, and modifying files in the Poky layer. The more separated an organization is, the easier it is to reuse the layers in future projects, as the Poky source code is split into different layers as well. It contains three layers by default, as we can see in the output of the following command line:

```
$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
meta                  /home/user/yocto/poky/meta                5
meta-poky             /home/user/yocto/poky/meta-poky           5
meta-yocto-bsp        /home/user/yocto/poky/meta-yocto-bsp      5
```

Figure 11.1 – The result of bitbake-layers show-layers

The command-line output shows the following three essential properties of any layer:

- **Name**: This usually starts with the `meta` string.
- **Path**: This is important when we want to add a layer in our project that is appended to the `BBPATH` variable.
- **Priority**: This is the value used by BitBake to decide which recipe to use and the order in which the `.bbappend` files should be concatenated. It means that if two layers include the same recipe file (`.bb`), the one with the highest priority is used. In the case of `.bbappend`, every `.bbappend` file is included in the original recipe. The layer priority determines the order of inclusion, so the `.bbappend` files within the highest priority layers are appended first, followed by the others.

Taking Poky as an example, it has three central individual layers. The `meta-yocto-bsp` layer is the Poky reference **Board Support Package** (**BSP**) layer. It contains machine configuration files and recipes to configure packages for the machines. As it is a reference BSP layer, it can be used as an example.

The `meta-poky` layer is the Poky reference distribution layer. It contains a distribution configuration used in Yocto Project by default. This default distribution is described in the `poky.conf` file, and it is widely used for testing products. It can be used as a starting point when designing your own distribution.

Another kind of layer is the software layer, which includes only applications or configuration files for applications and can be used on any architecture. There is a massive list of software layers. To name only a few, we have `meta-java`, `meta-qt5`, and `meta-browser`. The `meta-java` layer provides Java runtime and SDK support, the `meta-qt5` layer includes Qt5 support, and `meta-browser` supports web browsers such as Firefox and Chrome.

The `meta` layer is the OpenEmbedded Core metadata, which contains the recipes, classes, and the QEMU machine configuration files. It can be considered a mixed layer type, as it has software collection, BSP definition, and the distribution used by Yocto Project as the baseline.

Sometimes, your product may have special requirements, and changes in the `build/conf/local.conf` file will need to be made as required. The most adequate and maintainable solution is to create a distribution layer to place the distribution definition file.

> **Tip**
>
> The build/conf/local.conf file is a volatile file that is not supposed to be tracked by Git.
>
> We should not rely on it to set package versions, providers, and the system features for products but use it instead just as a shortcut for testing purposes during development.

Avoiding adding custom settings in build/conf/local.conf helps to make our builds reproducible afterward.

Detailing a layer's source code

Usually, a layer has a directory tree, as shown in the following screenshot:

```
meta-*/
├── classes
├── conf
├── COPYING
├── README
└── recipes-*
```

Figure 11.2 – The standard layer layout

Inside this directory are two files, <layer>/COPYING and <layer>/README, a license and a message to a user respectively. In <layer>/README, we must specify any other dependency and information that the layer's users need to know. The meta- prefix for the layer is not a requirement but a commonly used naming convention.

The classes folder should hold the classes specific to that layer (the .bbclass files). It is an optional directory.

The <layer>/conf folder is mandatory and should provide the configuration files (the .conf files). The layer configuration file, <layer>/conf/layer.conf, which will be covered in detail in the next chapter, is the file with the layer definition.

An example of the directory layout of the <layer>/conf folder is shown in *Figure 11.2*, where **(a)** shows the structure for a BSP layer and **(b)** shows the structure for a distribution layer:

```
conf/
├── layer.conf
├── machine
│   ├── include
│   │   └── <another>.conf
│   └── <machine-name>.conf
└── <other>.conf

          (a)
```

```
conf/
├── distro
│   ├── <distro-name>.conf
│   ├── include
│   │   └── <another>.conf
│   └── <other>.conf
├── layer.conf
└── <other>.conf

          (b)
```

Figure 11.3 – The <layer>/conf layout for BSP and distribution layers

The `recipe-*` folder is a cluster of recipes separated by category – for example, `recipes-core`, `recipes-bsp`, `recipes-graphic`, `recipes-multimedia`, and `recipes-kernel`. Inside each folder, starting with the `recipes-` prefix, there is a directory with the recipe name or a group of recipes. Inside it, the recipe files are placed, whose names end with `.bb` or `.bbappend`. For example, we can find the following screenshot from `meta` layer:

```
recipes-multimedia/
├── ffmpeg
│   ├── ffmpeg
│   │   └── 0001-libavutil-include-assembly-with-full-path-from-sourc.patch
│   └── ffmpeg_5.0.1.bb
├── flac
│   └── flac_1.3.4.bb
│ (...)
└── x264
    ├── x264
    │   ├── don-t-default-to-cortex-a9-with-neon.patch
    │   └── Fix-X32-build-by-disabling-asm.patch
    └── x264_git.bb
```

Figure 11.4 – An example of the recipes-* layout

Adding meta layers

We can find the most of available meta layers at `http://layers.openembedded.org`. There are hundreds of meta layers from the Yocto Project, OpenEmbedded, communities, and companies that can be manually cloned inside the project source directory.

To include, for example, `meta-oe` (one of the several meta layers inside the `meta-openembedded` repository) in our project, we can change the content of the configuration files or use BitBake command lines. However, we first need to fetch the layer's source code. Run the following command from your Poky source directory:

```
$ git clone https://github.com/openembedded/meta-openembedded -b kirkstone
```

Figure 11.5 – Cloning the meta-openembedded layer

We need to modify the `build/conf/bblayer.conf` file to add the layer location, using its absolute path. See **line 12** in *Figure 11.6* as follows:

```
 1 # POKY_BBLAYERS_CONF_VERSION is increased each time build/conf/bblayers.conf
 2 # changes incompatibly
 3 POKY_BBLAYERS_CONF_VERSION = "2"
 4
 5 BBPATH = "${TOPDIR}"
 6 BBFILES ?= ""
 7
 8 BBLAYERS ?= " \
 9   /home/user/yocto/poky/meta \
10   /home/user/yocto/poky/meta-poky \
11   /home/user/yocto/poky/meta-yocto-bsp \
12   /home/user/yocto/poky/meta-openembedded/meta-oe \
13   "
```

Figure 11.6 – The content of build/conf/bblayers.conf after including the meta-openembedded layer

Alternatively, we can use the `bitbake-layers` tool to perform the inclusion for us. This can be done using the following command from the `build` directory:

```
$ bitbake-layers add-layer ../meta-openembedded/meta-oe
```

Figure 11.7 – The command line to add the layer location

The Yocto Project layer ecosystem

It is convenient to create a layer. To make all the available layers easier to access, the OpenEmbedded community has developed an index, available at `http://layers.openembedded.org`, where most of them can be found. An example of its **Layers** tab is shown as follows:

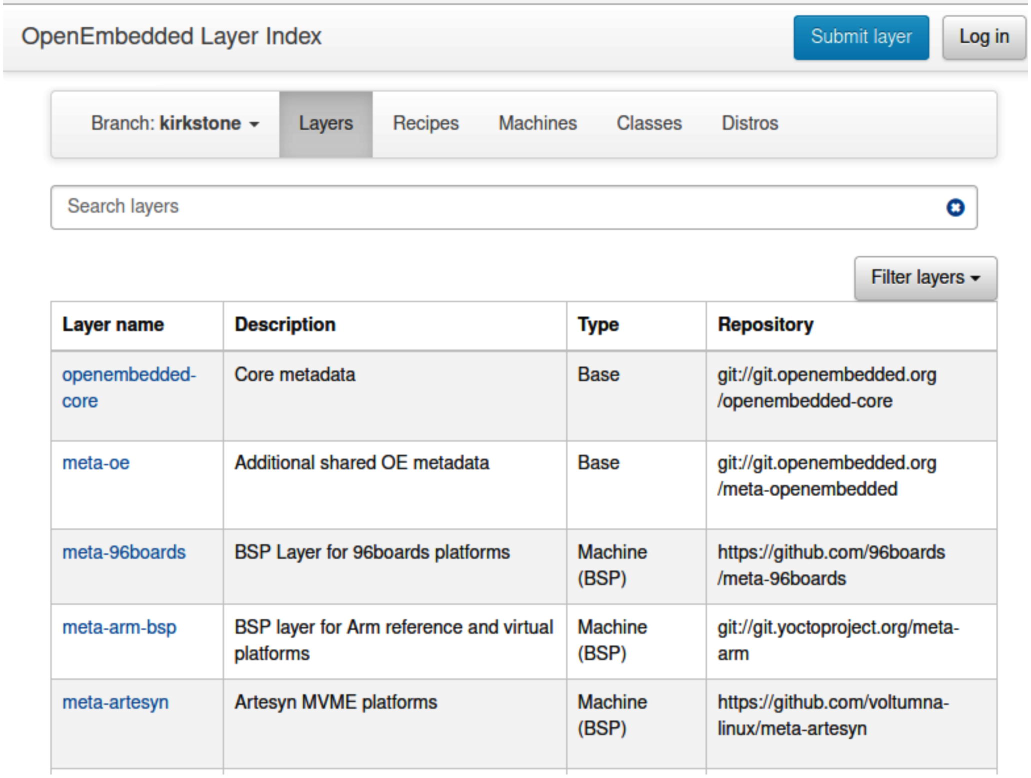


Figure 11.8 – The OpenEmbedded Layer Index for Kirkstone

Another convenient use case for the OpenEmbedded Layer Index website is to search for a specific software type or recipe. The OpenEmbedded Layer Index can save the day by allowing us to search for the following:

- Machines
- Distributions
- Layers
- Recipes
- Classes

The `bitbake-layers` tool also supports the use of the OpenEmbedded Layer Index. For example, to add the `meta-oe` layer, we can use the following command:

```
$ bitbake-layers layerindex-fetch meta-oe
NOTE: Starting bitbake server...
Loading https://layers.openembedded.org/layerindex/api/;branch=kirkstone...
Layer                        Git repository (branch)         Subdirectory
============================================================================
layers...:kirkstone:meta-oe   git://git...bedded (kirkstone)   meta-oe
Cloning into '/home/user/yocto/poky/meta-openembedded'...
remote: Counting objects: 194762, done.
remote: Compressing objects: 100% (66384/66384), done.
remote: Total 194762 (delta 122372), reused 190562 (delta 119709)
Receiving objects: 100% (194762/194762), 47.60 MiB | 12.90 MiB/s, done.
Resolving deltas: 100% (122372/122372), done.
Adding layer "meta-oe" (/home/user/yocto/poky/meta-openembedded/meta-oe) to conf/bblayers.conf
```

Figure 11.9 – Fetching a layer from the OpenEmbedded Layer index

Summary

In this chapter, we introduced the concept of layering. We learned about the directory structure in detail and the content in each layer type. In addition, we saw how to add an external layer to our project manually or by using the BitBake command line, as well as how to use the OpenEmbedded Layer index to find the available layers we need easily.

In the next chapter, we will learn more about why we need to create new layers and what the common metadata included in them is (such as machine definition files, recipes, and images). We will wrap it all up with an example of distribution customization.

12
Creating Custom Layers

In addition to using existing layers from the community or vendors, we will learn how to create layers for our products in this chapter. Additionally, we will discover how to create a machine definition and distribution and profit from them to organize our source code better.

Making a new layer

Before creating our layer, it's always a good idea to check whether a similar one is already available at the following website: `http://layers.openembedded.org`.

If we are still looking for a layer suitable for our needs, the next step is to create the directory. Usually, the layer name starts with `meta-`, but this is not a technical restriction.

The `<layer>/conf/layer.conf` file is the layer configuration file required for every layer. The new layer can be created with a tool called `bitbake-layers` from BitBake, provided in Poky, as shown in the following command:

```
$ bitbake-layers create-layer ~/yocto/poky/meta-newlayer
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer /home/user/yocto/poky/meta-newlayer'
```

Figure 12.1 – Creating a new layer using bitbake-layers

After creating the layer, we need to include it in the `build/conf/bblayers.conf` file using the following command:

```
$ bitbake-layers add-layer /home/user/yocto/poky/meta-newlayer
NOTE: Starting bitbake server...
```

Figure 12.2 – Adding meta-newlayer to build/conf/bblayers.conf

> **Tip**
>
> The `bitbake-layers` tool, by default, generates the layer with layer priority 6. We can still customize the priority using parameters.

The last command generates the layer, as shown in the following figure:

```
meta-newlayer/
├── conf
│   └── layer.conf
├── COPYING.MIT
├── README
└── recipes-example
    └── example
        └── example_0.1.bb
```

Figure 12.3 – The meta-newlayer layout when created

The default layer configuration file for `meta-newlayer` is the minimal configuration to get the layer working. However, it can be customized to include configurations required in the future.

The following figure shows the content of default `conf/layer.conf` for the `meta-newlayer` layer we just created:

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "meta-newlayer"
BBFILE_PATTERN_meta-newlayer = "^${LAYERDIR}/"
BBFILE_PRIORITY_meta-newlayer = "6"

LAYERDEPENDS_meta-newlayer = "core"
LAYERSERIES_COMPAT_meta-newlayer = "kirkstone"
```

Figure 12.4 – The meta-newlayer/conf/layer.conf minimal configuration

Some commonly used variables that may need to be added or changed are `LAYERVERSION` and `LAYERDEPENDS`. Those are useful if our layer requires other layers to work. Both variables' names must be suffixed with the layer's name, as follows:

- `LAYERVERSION`: This is an optional variable that specifies the version of the layer in a single number. This variable is used within the `LAYERDEPENDS` variable to depend on a specific layer version – for example, `LAYERVERSION_meta-newlayer = "1"`.
- `LAYERDEPENDS`: This lists the layers that the recipes depend upon, separated by spaces – for example, we add the dependency for version `2` of `meta-otherlayer` with `LAYERDEPENDS_meta-newlayer += "meta-otherlayer:2"`.

An error is incurred if a dependency cannot be satisfied or the version numbers do not match. The base of the layer structure is now created. In the following sections, we will learn how to extend it.

Adding metadata to the layer

Layer metadata can serve two goals – add new software, or feature and modify existing metadata.

We can include several metadata files on a new layer, such as recipes, images, and `bbappend` files. There are several examples of `bbappend` files on `meta-yocto-bsp` and `meta-yocto`. We will explore some of their common uses in *Chapter 13*, *Customizing Existing Recipes*.

In the next sections, we will go through some common modifications to layer metadata.

Creating an image

Image files are, in essence, a set of packages grouped for a purpose and configured in a controlled way. We can create an image from scratch or create one by reusing an existing one and adding the extra necessary packages.

We should reuse an existing image when possible, making code maintenance more manageable and highlighting the functional differences. For example, we may want to include an application and remove an image feature from the `core-image-full-cmdline` image file. In that case, we can create an image in the `recipes-mine/images/my-image-full-cmdline.bb` file with the following lines of code:

```
require recipes-extended/images/core-image-full-cmdline.bb

IMAGE_FEATURES:remove = "splash"
CORE_IMAGE_EXTRA_INSTALL += "myapp"
```

Figure 12.5 – The content of my-image-full-cmdline.bb

The `core-image` class provides image features that offer helpful building blocks of commonly used functionality and should be used when creating an image from scratch. For example, we can create an image in the `recipes-mine/images/my-image-strace.bb` file consisting of the following lines of code:

```
inherit core-image

IMAGE_FEATURES += "ssh-server-openssh splash"
CORE_IMAGE_EXTRA_INSTALL += "strace"
```

Figure 12.6 – The content of my-image-strace.bb

> **Tip**
> The list appending operator (`+=`) guarantees that a new `EXTRA_IMAGE_FEATURES` variable can be added by `build/conf/local.conf`.

`CORE_IMAGE_EXTRA_INSTALL` is the variable we should use to include extra packages in the image when we inherit the `core-image` class, which facilitates image creation. The class adds support for the `IMAGE_FEATURES` variable, which avoids duplication of code.

Currently, the following image features are supported, as detailed in the *Image Features* section of the *Yocto Project Reference Manual* (`https://docs.yoctoproject.org/4.0.4/ref-manual/features.html#image-features`):

- `allow-empty-password`: Allows Dropbear and OpenSSH to accept logins from accounts that have an empty password string.
- `allow-root-login`: Allows Dropbear and OpenSSH to accept `root` logins.
- `dbg-pkgs`: Installs debug symbol packages for all packages installed in a given image.
- `debug-tweaks`: Makes an image suitable for development (for example, allows root logins, logins without passwords – including root ones, and enables post-installation logging).
- `dev-pkgs`: Installs development packages (headers and extra library links) for all packages installed in a given image.
- `doc-pkgs`: Installs documentation packages for all packages installed in a given image.
- `empty-root-password`: This feature, or `debug-tweaks`, is required if you want to allow root login with an empty password.
- `hwcodecs`: Installs hardware acceleration codecs.
- `lic-pkgs`: Installs license packages for all packages installed in a given image.
- `nfs-server`: Installs an NFS server.

- `overlayfs-etc`: Configures the `/etc` directory to be in `overlayfs`. This allows you to store device-specific information elsewhere, especially if the root filesystem is configured as read-only.
- `package-management`: Installs package management tools and preserves the package manager database.
- `perf`: Installs profiling tools such as `perf`, `systemtap`, and **LTTng**.
- `post-install-logging`: Enables you to log postinstall script runs in the `/var/log/postinstall.log` file on the first boot of the image on the target system.
- `ptest-pkgs`: Installs `ptest` packages for all ptest-enabled recipes.
- `read-only-rootfs`: Creates an image whose root filesystem is read-only.
- `read-only-rootfs-delayed-postinsts`: When specified in conjunction with `read-only-rootfs`, it specifies that post-install scripts are still permitted.
- `serial-autologin-root`: When specified in conjunction with `empty-root-password`, it will automatically login as `root` on the serial console.
- `splash`: Enables you to show a splash screen during boot. By default, this screen is provided by `psplash`, which does allow customization.
- `ssh-server-dropbear`: Installs the Dropbear minimal SSH server.
- `ssh-server-openssh`: Installs the OpenSSH SSH server, which is more full-featured than Dropbear. Note that if both the OpenSSH SSH server and the Dropbear minimal SSH server are present in `IMAGE_FEATURES`, then OpenSSH will take precedence and Dropbear will not be installed.
- `stateless-rootfs`: Specifies that an image should be created as stateless – when using `systemd`, `systemctl-native` will not be run on the image, leaving the image to be populated at runtime by `systemd`.
- `staticdev-pkgs`: Installs static development packages, which are static libraries (for example, `*.a` files), for all packages installed in a given image.
- `tools-debug`: Installs debugging tools such as `strace` and `gdb`.
- `tools-sdk`: Installs a full SDK that runs on a device.
- `tools-testapps`: Installs device testing tools (for example, touchscreen debugging).
- `weston`: Installs Weston (a reference Wayland environment).
- `x11-base`: Installs the X server with a minimal environment.
- `x11`: Installs the X server.
- `x11-sato`: Installs the OpenedHand Sato environment.

Adding a package recipe

Poky includes several classes that makes the process for the most common development tools as projects abstract, based on Autotools, CMake, and Meson. A package recipe is how we can instruct BitBake to perform the `fetch`, `unpack`, `patch`, `configure`, `compile`, and `install` tasks on our application, kernel module, or any software a project provides. In addition, a list of classes included in Poky can be seen in the *Classes* section in the *Yocto Project Reference Manual* (`https://docs.yoctoproject.org/4.0.4/ref-manual/classes.html`).

A straightforward recipe that executes the `compile` and `install` tasks explicitly is provided as follows:

```
DESCRIPTION = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "file://helloworld.c"

S = "${WORKDIR}"

do_compile() {
    ${CC} helloworld.c -o helloworld
}

do_install() {
    install -D -m 0755 helloworld ${D}${bindir}/helloworld
}
```

Figure 12.7 – A manually crafted helloworld recipe

The `do_compile` and `do_install` code blocks provide the Shell Script command for us to build and install the resulting binary into the destination directory, referenced as `${D}`, which aims to relocate the installation directory to a path inside the `build/tmp/work/` directory. Suppose that we are working on an Autotools-based project. If so, we can avoid a lot of code duplication by using the `autotools` class in the stripped example, extracted from the recipe in the `poky/meta/recipes-core/dbus-wait/dbus-wait_git.bb` file, as follows:

```
1 SUMMARY = "A simple tool to wait for a specific signal over DBus"
2 HOMEPAGE = "http://git.yoctoproject.org/cgit/cgit.cgi/dbus-wait"
3 DESCRIPTION = "${SUMMARY}"
4 SECTION = "base"
5 LICENSE = "GPL-2.0-only"
6 LIC_FILES_CHKSUM = "file://COPYING;md5=b234ee4d69f5fce4486a80fdaf4a4263"
7
8 DEPENDS = "dbus"
9
10 SRCREV = "6cc6077a36fe2648a5f993fe7c16c9632f946517"
11 PV = "0.1+git${SRCPV}"
12 PR = "r2"
13
14 SRC_URI = "git://git.yoctoproject.org/${BPN};branch=master"
15 UPSTREAM_CHECK_COMMITS = "1"
16
17 S = "${WORKDIR}/git"
18
19 inherit autotools pkgconfig
```

Figure 12.8 – The content of poky/meta/recipes-core/dbus-wait/dbus-wait_git.bb

The simple act of inheriting the `autotools` class in *line 19* is to provide all the code required to do the following tasks:

- Update the `configure` script code and artifacts
- Update the `libtool` scripts
- Run the `configure` script
- Run `make`
- Run `make install`

The same concepts apply to other building tools, as is the case for **CMake** and **Meson**. Additionally, the number of supported classes is growing in every release to support new build systems and avoid code duplication.

Automatically creating a base package recipe using devtool

As we learned in the *Creating a recipe from an external Git repository* section in *Chapter 9*, *Developing with the Yocto Project*, `devtool` automates the process of creating a recipe based on an existing project with the following command:

```
$ devtool add https://github.com/OSSystems/bbexample
```

Figure 12.9 – The command line to generate the recipe for bbexample

Behind the scenes, `devtool` ran the `recipetool` to generate a recipe and automatically configure all pre-built information into the new recipe file. The end result is stored in the `workspace` directory, a layer maintained by `devtool`. To copy the recipe file to the target layer, we can use the `devtool` command, as shown here:

```
$ devtool finish bbexample ../meta-newlayer/
NOTE: Starting bitbake server...
Loading cache: 100% |##########################################################| Time: 0:00:00
Loaded 2784 entries from dependency cache.
Parsing recipes: 100% |########################################################| Time: 0:00:00
Parsing of 1746 .bb files complete (1745 cached, 1 parsed). 2785 targets, 77 skipped,
0 masked, 0 errors.
INFO: Updating SRCREV in recipe bbexample_git.bb
INFO: Moving recipe file to /home/user/yocto/poky/meta-newlayer/recipes-bbexample/bbexample
INFO: Leaving source tree /home/user/yocto/poky/build/workspace/sources/bbexample as-is;
if you no longer need it then please delete it manually
```

Figure 12.10 – The command line to deploy the bbexample recipe to meta-newlayer

The created `meta-newlayer/recipes-bbexample/bbexample/bbexample_git.bb` file is shown in the following snippet:

```
# Recipe created by recipetool
# This is the basis of a recipe and may need further editing in order to be fully functional.
# (Feel free to remove these comments when editing.)

# WARNING: the following LICENSE and LIC_FILES_CHKSUM values are best guesses - it is
# your responsibility to verify that the values are complete and correct.
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://LICENSE;md5=96af5705d6f64a88e035781ef00e98a8"

SRC_URI = "git://github.com/OSSystems/bbexample;protocol=https;branch=master"

# Modify these as desired
PV = "0.1+git${SRCPV}"
SRCREV = "ece3cef9abc95cb77c931f9f27860102e43cc1d9"

S = "${WORKDIR}/git"

# NOTE: if this software is not capable of being built in a separate build directory
# from the source, you should replace autotools with autotools-brokensep in the
# inherit line
inherit autotools

# Specify any options you want to pass to the configure script using EXTRA_OECONF:
EXTRA_OECONF = ""
```

Figure 12.11 – The content of bbexamle_git.bb

The `devtool` has created a base recipe, which should not be taken as a final recipe. You should check for compilation options, extra metadata information, and so on.

Adding support to a new machine definition

Even though creating a new machine definition for use in Poky is a straightforward task, it shouldn't be underestimated. Depending on the set of features we need to support at the BSP layer, it can involve checking the bootloader, kernel, and hardware support drivers.

The Yocto Project supports ARM, ARM64, x86, x86-64, PowerPC, PowerPC 64, MIPS, MIPS64, RISC-V 32, and RISC-V 64, representing the most currently used embedded architectures.

The prevailing set of variables used in a machine definition is as follows:

- `TARGET_ARCH`: This sets the machine architecture – for example, ARM and x86-64
- `PREFERRED_PROVIDER_virtual/kernel`: This overrides the default kernel (`linux-yocto`) if you need to use a specific one
- `SERIAL_CONSOLES`: This defines serial consoles and their speeds
- `MACHINE_FEATURES`: This describes hardware features, so the software stack required is included in the images by default
- `KERNEL_IMAGETYPE`: This is used to choose the kernel image type – for example, `bzImage` or `Image`
- `IMAGE_FSTYPES`: This sets the generated filesystem image types – for example, `tar.gz`, `ext4`, and `ubifs`

You can see examples of machine definition files inside the Poky source code in the `meta-yocto-bsp/conf/machine/` directory. When describing a new machine, we should pay special attention to specific features supported by it in `MACHINE_FEATURES`. This way, the software needed to help these features is installed into the images. The values currently available for `MACHINE_FEATURES` are listed as follows:

- `acpi`: The hardware has ACPI (x86/x86-64 only)
- `alsa`: The hardware has ALSA audio drivers
- `apm`: The hardware uses APM (or APM emulation)
- `bluetooth`: The hardware has integrated BT
- `efi`: Support for booting through EFI
- `ext2`: The hardware HDD or microdrive
- `keyboard`: The hardware has a keyboard
- `numa`: The hardware has non-uniform memory access
- `pcbios`: Support for booting through BIOS

- `pci`: The hardware has a PCI bus
- `pcmcia`: The hardware has PCMCIA or CompactFlash sockets
- `phone`: Mobile phone (voice) support
- `qemu-usermode`: QEMU can support user-mode emulation for this machine
- `qvga`: The machine has a QVGA (320x240) display
- `rtc`: The machine has a real-time clock
- `screen`: The hardware has a screen
- `serial`: The hardware has serial support (usually RS232)
- `touchscreen`: The hardware has a touchscreen
- `usbgadget`: The hardware is USB gadget device-capable
- `usbhost`: The hardware is USB host-capable
- `vfat`: FAT filesystem support
- `wifi`: The hardware has integrated Wi-Fi

Wrapping an image for your machine

Creating a ready-to-use image for a machine should be addressed at the end of any BSP support layer development. The type of image depends on the processor, peripherals included on the board, and project restrictions.

The **partitioned image** is the most frequently used image for direct use in the storage. The Yocto Project has a tool called `wic`, which provides a flexible way to generate this image. It allows the creation of partitioned images based on a template file (`.wks`), written in a common language that describes the target image layout. The language definition can be found in the *OpenEmbedded Kickstart (.wks) Reference* section from *The Yocto Project Reference Manual* (`https://docs.yoctoproject.org/4.0.4/ref-manual/kickstart.html#openembedded-kickstart-wks-reference`).

The `.wks` file is placed in our layer inside the `wic` directory. It is common to have multiple files in this directory to specify different image layouts. However, it is essential to remember that the chosen structure must match the machine – for example, when considering the use of an i.MX-based machine that boots using U-Boot from an SD card with two partitions, one for the boot files and the other for `rootfs`. The respective `.wks` file is shown here:

```
# short-description: Create SD card image with a boot partition
# long-description:
# Create an image that can be written onto a SD card using dd for use
# with i.MX SoC family
# It uses u-boot
#
# The disk layout used is:
#  - --------- --------------
# | | u-boot  |    rootfs    |
#  - --------- --------------
# ^ ^         ^              ^
# | |         |              |
# 0 1kiB    4MiB + rootfs + IMAGE_EXTRA_SPACE (default 10MiB)
#
part u-boot --source rawcopy --sourceparams="file=u-boot.imx" --ondisk mmcblk --no-table --align 1
part / --source rootfs --ondisk mmcblk --fstype=ext4 --label root --align 4096

bootloader --ptable msdos
```

Figure 12.12 – An example of a .wks file for an i.MX device using SPL

To enable the `wic`-based image generation, it is a matter of adding `wic` to `IMAGE_FSTYPES`. We can also define the `.wks` file to be used by setting the `WKS_FILE` variable.

Using a custom distribution

The creation of a distribution is a mix of simplicity and complexity. Creating the distribution file is straightforward but significantly impacts Poky's behavior. Depending on our options, it may cause a binary incompatibility with previously built binaries.

The distribution is where we define global options, such as the toolchain version, graphical backends, and support for **OpenGL**. We should make a distribution only if the default settings provided by Poky fail to fulfill our requirements.

Usually, we intend to change a small set of options from Poky. For example, we remove the **X11** support to use a framebuffer instead. We can easily accomplish this by reusing the Poky distribution and overriding the necessary variables – for example, the sample distribution represented by the `<layer>/conf/distro/my-distro.conf` file is as follows:

```
require conf/distro/poky.conf

DISTRO = "my-distro"
DISTRO_NAME = "my-distro (My New Distro)"
DISTRO_VERSION = "1.0"
DISTRO_CODENAME = "codename"
SDK_VENDOR = "-mydistrosdk"
SDK_VERSION := "${@'${DISTRO_VERSION}'.replace('snapshot-${DATE}','snapshot')}"

MAINTAINER = "my-distro <my-distro@mycompany.com>"

DISTRO_FEATURES:remove = "wayland vulkan opengl"
```

Figure 12.13 – An example of a custom distribution file

To use the distribution just created, we need to add the following piece of code to the `build/conf/local.conf` file:

```
DISTRO = "my-distro"
```

Figure 12.14 – The line to set DISTRO on build/conf/local.conf

The `DISTRO_FEATURES` variable may influence how the recipes are configured and the packages are installed in images – for example, if we want to use sound in any machine and image, the `alsa` features must be present. The following list shows the present state for the `DISTRO_FEATURES`-supported values, as detailed in the *Distro Features* section in the *Yocto Project Reference Manual* (`https://docs.yoctoproject.org/4.0.4/ref-manual/features.html#distro-features`):

- `3g`: Includes support for cellular data
- `acl`: Includes Access Control List support
- `alsa`: Includes Advanced Linux Sound Architecture support (OSS compatibility kernel modules are installed if available)
- `api-documentation`: Enables the generation of API documentation during recipe builds
- `bluetooth`: Includes Bluetooth support (integrated BT only)
- `cramfs`: Includes CramFS support
- `debuginfod`: Includes support for getting ELF debugging information through a `debuginfod` server
- `ext2`: Includes tools to support devices with an internal HDD/Microdrive for storing files (instead of Flash-only devices)
- `gobject-introspection-data`: Includes data to support GObject introspection
- `ipsec`: Includes IPSec support
- `ipv4`: Includes IPv4 support
- `ipv6`: Includes IPv6 support
- `keyboard`: Includes keyboard support
- `ldconfig`: Includes support for `ldconfig` and `ld.so.conf` on the target
- `ld-is-gold`: Uses the gold linker instead of the standard GNU linker (`bfd`)
- `lto`: Enables Link-Time Optimization
- `multiarch`: Enables you to build applications with multiple architecture support
- `nfc`: Includes support for Near Field Communication

- `nfs`: Includes NFS client support
- `nls`: Includes **Native Language Support** (**NLS**)
- `opengl`: Includes the Open Graphics Library, a cross-language, multi-platform API, used to render two- and three-dimensional graphics
- `overlayfs`: Includes OverlayFS support
- `pam`: Includes **Pluggable Authentication Module** (**PAM**) support
- `pci`: Includes PCI bus support
- `pcmcia`: Includes PCMCIA/CompactFlash support
- `polkit`: Includes Polkit support
- `ppp`: Includes PPP dial-up support
- `ptest`: Enables you to build the package tests that were supported by individual recipes
- `pulseaudio`: Includes support for PulseAudio
- `seccomp`: Enables you to build applications with `seccomp` support, allowing the applications to strictly restrict the system calls that they are allowed to invoke
- `selinux`: Includes support for **Security-Enhanced Linux** (**SELinux**) (requires `meta-selinux`)
- `smbfs`: Includes SMB network client support
- `systemd`: Includes support for this `init` manager, a full replacement for `init`, with parallel starting of services, reduced shell overhead, and other features
- `usbgadget`: Includes USB Gadget Device support
- `usbhost`: Includes USB Host support
- `usrmerge`: Merges the `/bin`, `/sbin`, `/lib`, and `/lib64` directories into their respective counterparts in the `/usr` directory to provide better package and application compatibility
- `vfat`: Includes FAT filesystem support
- `vulkan`: Includes support for the Vulkan API
- `wayland`: Includes the Wayland display server protocol and the library that supports it
- `wifi`: Includes Wi-Fi support (integrated only)
- `x11`: Includes the X server and libraries
- `xattr`: Includes support for extended file attributes
- `zeroconf`: Includes support for zero-configuration networking

MACHINE_FEATURES versus DISTRO_FEATURES

The `DISTRO_FEATURES` and `MACHINE_FEATURES` variables work together to provide feasible support for the final system. When a machine supports a feature, this does not imply that the target system supports it because the distribution must provide its underlying base.

For example, if a machine supports Wi-Fi but the distribution does not, the applications used by the operating system will be built with Wi-Fi support disabled so that the outcome will be a system without Wi-Fi support. On the other hand, if the distribution provides Wi-Fi support and a machine does not, the modules and applications needed for the Wi-Fi will not be installed in images built for this machine. However, the operating system and its modules have support for Wi-Fi enabled.

Understanding the scope of a variable

The BitBake metadata has thousands of variables, but the scope where these variables are available depends on where it is defined. There are two kinds of variables, as follows:

- Variables defined in configuration files are global to every recipe, also referred to as configuration metadata. The parsing order of the main configuration files is shown as follows:
  - `build/conf/local.conf`
  - `<layer>/conf/machines/<machine>.conf`
  - `<layer>/conf/distro/<distro>.conf`
- Variables defined within recipe files have recipe visibility scope that is local to the specific recipe only during the execution of its tasks.

Summary

In this chapter, we covered how to create a new layer and metadata. First, we saw how to create a machine configuration, a distribution definition, and recipe files. Then, we learned how to make images and include our application in an image.

In the next chapter, we will access some examples of the most common customization cases used by an additional layer, such as modifying existing packages, adding extra options to `autoconf`, applying a new patch, and including a new file to a package. We will see how to configure `BusyBox` and `linux-yocto`, the two packages commonly customized when making an embedded system.

13

Customizing Existing Recipes

In the course of our work with Yocto Project's tools, it is expected that we will need to customize existing recipes. In this chapter, we will explore some examples, such as changing compilation options, enabling or disabling features of a recipe, applying an extra patch, and using configuration fragments to customize some recipes.

Understanding common use cases

Nowadays, projects usually have a set of layers to provide the required features. We certainly need to make changes on top of them to adapt them to our specific needs. They may be cosmetic or substantive changes, but the way to make them is the same.

We must create a `.bbappend` file to change a preexisting recipe in our project layer. For example, suppose the original recipe was named `<original-layer>/recipes-core/app/app_1.2.3.bb`. When you create a `.bbappend` file, you can use the `%` wildcard character to allow for matching recipe names. So, the `.bbappend` file could have the following different forms:

- `App_1.2.3.bbappend`: This applies the change only for the `1.2.3` version
- `app_1.2.%.bbappend`: This applies the change only for the `1.2.y` version
- `app_1.%.bbappend`: This applies the change only for the `1.x` and `1.x.y` versions
- `app_%.bbappend`: This applies the change for any version

We can have multiple `.bbappend` files, depending on the intended changes we want to apply to the app recipe. Sometimes we can restrict the changes to one version, but sometimes, we want to change all available recipes.

> **Note**
>
> When there is more than one `.bbappend` file for a recipe, all of them are joined following the layer's priority order.

The .bbappend file can be seen as a text appended at the end of the original recipe. It empowers us with a highly flexible mechanism to avoid duplicating source code to apply the required changes to our project's layers.

Extending a task

When the task content does not satisfy our requirements, we replace it (providing our implementation) or append it. As we will learn more extensively about the BitBake metadata syntax in *Chapter 8*, *Diving into BitBake Metadata*, the :append and :prepend operators can extend a task with extra content. For example, to extend a do_install task, we can use the following code:

```
do_install:append() {
    # Do my commands
}
```

Figure 13.1 – Example on how to extend the do_install task

This way, the new content is concatenated in the original task.

Adding extra options to recipes based on Autotools

Let's assume we have Autotools-based application, along with a preexisting recipe for it, and we want to do the following:

- Enable my-feature
- Disable another-feature

The content of the .bbappend file to make the changes will be the following:

```
EXTRA_OECONF += "--enable-my-feature --disable-another-feature"
```

Figure 13.2 – Adding extra configuration to the Autoconf flags

The same strategy can be used if we need to enable it conditionally based on the hardware we are building for, as follows:

```
EXTRA_OECONF:append:arm = " --enable-my-arm-feature"
```

Figure 13.3 – Conditionally adding extra configuration to the Autoconf flags

The Yocto Project supports many different build systems, and the variables to configure them are shown in the following table:

Build System	**Variable**
Autotools	`EXTRA_OECONF`
Cargo	`EXTRA_OECARGO`
CMake	`EXTRA_OECMAKE`
Make	`EXTRA_OEMAKE`
Meson	`EXTRA_OEMESON`
NPM	`EXTRA_OENPM`
SCons	`EXTRA_OESCONS`
WAF	`EXTRA_OEWAF`

Table 13.1 – The list of variables to configure each build system

The variables from *Table 13.1* are given as arguments for the respective build system.

Applying a patch

For cases where we need to apply a patch to an existing package, we should use `FILESEXTRAPATHS`, which includes new directories in the searching algorithm, making the additional file visible to BitBake, as shown here:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}-${PV}:"
SRC_URI += "file://mypatch.patch"
```

Figure 13.4 – The content of .bbappend is used only to apply mypatch.patch

In the preceding example, `THISDIR` expands to the current directory, and `PN` and `PV` expand to the package name and version, respectively. This new path is then included in the directories list used for file searching. The prepend operator is crucial as it guarantees that the file is picked from this directory, even if a file with the same name is added in the lower priority layers in the future.

BitBake assumes that every file with a `.patch` or `.diff` extension is a patch and applies them accordingly.

Adding extra files to the existing packages

If we need to include an additional configuration file, we should use FILESEXTRAPATHS, as explained in the previous example and shown in the following lines of code:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}-${PV}:"
SRC_URI += "file://newconfigfile.conf"

do_install:append() {
    install -D -m 644 ${WORKDIR}/newconfig.conf \
                      ${D}${sysconfdir}/newconfig.conf
}
```

Figure 13.5 – The content of the .bbappend file to install a new configuration file

The do_install:append function appends the provided block below the metadata already available in the original do_install function. It includes the command to copy our new configuration file into the package's filesystem. The file is copied from ${WORKDIR} to ${D} as these are the directories used by Poky to build the package and the destination directory used by Poky to create the package.

There are many variables to define paths in our recipes, such as bindir, datadir, and sysconfdir. The poky/meta/conf/bitbake.conf file defines all those commonly used variables. The variables exist, so the installation paths of binaries can be customized depending on the use case. For example, the native SDK binaries require a specific installation path, so the binaries don't conflict with the target ones.

The following table shows the most common variables and their default expanded values:

Variable	Default Expanded Value
base_bindir	/bin
base_sbindir	/sbin
sysconfdir	/etc
localstatedir	/var
datadir	/usr/share
bindir	/usr/bin
sbindir	/usr/sbin
libdir	/usr/lib or /usr/lib64
libexecdir	/usr/libexec
includedir	/usr/include

Table 13.2 – The list of commonly used variables and their default expanded value

The use of hard coded paths in recipes should be avoided, so we reduce the risk of misconfiguration. For example, when using the `usrmerge` `DISTRO_FEATURE`, behind the scenes, all recipes set `base_bindir` as `bindir`, so if a recipe uses `/bin` as a hard coded path, the installation won't happen as expected.

Understanding file searching paths

When a file (a patch or a generic file) is included in `SRC_URI`, BitBake searches for the `FILESPATH` and `FILESEXTRAPATH` variables. The default setting is to look in the following locations:

1. `<recipe>-<version>/`
2. `<recipe>/`
3. `files/`

In addition to this, it also checks for `OVERRIDES` for a specific file to be overridden in each folder. To illustrate this, consider the `foo_1.0.bb` recipe. The `OVERRIDES = "<board>:<arch>"` variable for the file will be searched in the following directories, respecting the exact order shown:

1. `foo-1.0/<board>/`
2. `foo-1.0/<arch>/`
3. `foo-1.0/`
4. `foo/<board>/`
5. `foo/<arch>/`
6. `foo/`
7. `files/<board>/`
8. `files/<arch>/`
9. `files/`

This is just illustrative as the list of `OVERRIDES` is huge and machine-specific. When we work with our recipe, we can use `bitbake-getvar OVERRIDES` to find the complete list of available overrides for a specific machine and use them accordingly. See the Poky output as follows:

```
$ bitbake-getvar OVERRIDES
NOTE: Starting bitbake server...
#
# $OVERRIDES [2 operations]
#   set /home/user/yocto/poky/meta/conf/bitbake.conf:801
#     "${TARGET_OS}:${TRANSLATED_TARGET_ARCH}:pn-${PN}:${MACHINEOVERRIDES}:
#      ${DISTROOVERRIDES}:${CLASSOVERRIDE}${LIBCOVERRIDE}:forcevariable"
#   set /home/user/yocto/poky/meta/conf/documentation.conf:304
#     [doc] "BitBake uses OVERRIDES to control what variables are overridden
#            after BitBake parses recipes and configuration files."
# pre-expansion value:
#   "${TARGET_OS}:${TRANSLATED_TARGET_ARCH}:pn-${PN}:${MACHINEOVERRIDES}:
#    ${DISTROOVERRIDES}:${CLASSOVERRIDE}${LIBCOVERRIDE}:forcevariable"
OVERRIDES="linux:x86-64:pn-defaultpkgname:qemuall:qemux86-64:poky:
           class-target:libc-glibc:forcevariable"
```

Figure 13.6 – Using bitbake-getvar to get the value of the OVERRIDES variable

This command is quite useful for debugging the metadata during the debugging process.

Changing recipe feature configuration

`PACKAGECONFIG` is a mechanism to simplify feature set customization for recipes. It provides a way to enable and disable the recipe features. For example, the recipe has the following configuration:

```
PACKAGECONFIG ??= "feature1 feature2"
PACKAGECONFIG[feature1] = "\
    --enable-feature1, \
    --disable-feature1, \
    build-deps-for-feature1, \
    runtime-deps-for-feature1, \
    runtime-recommends-for-feature1, \
    packageconfig-conflicts-for-feature1"
PACKAGECONFIG[feature2] = "\
    --enable-feature2, \
    --disable-feature2, \
    build-deps-for-feature2, , , \
    packageconfig-conflicts-for-feature2"
```

Figure 13.7 – Example of PACKAGECONFIG

Figure 13.7 has two features: `feature1` and `feature2`. The behavior of each feature is defined by six arguments, separated by commas. You can omit any argument but must retain the separating commas. The order is essential and specifies the following:

1. Extra arguments if the feature is enabled.
2. Extra arguments if the feature is disabled.

3. Additional build dependencies (`DEPENDS`) if the feature is enabled.
4. Additional runtime dependencies (`RDEPENDS`) if the feature is enabled.
5. Additional runtime recommendations (`RRECOMMENDS`) if the feature is enabled.
6. Any conflicting (mutually exclusive) `PACKAGECONFIG` settings for this feature.

We can create a `.bbappend` file that expands the `PACKAGECONFIG` variable's default value to enable `feature2` as well, as shown here:

```
1 PACKAGECONFIG += "feature2"
```

Figure 13.8 – The content of a .bbappend file to expand the PACKAGECONFIG variable

> **Note**
> To add the same feature to the `build/conf/local.conf` file, we can use `PACKAGECONFIG:pn-<recipename>:append = ' feature2'`.

The list of available `PACKAGECONFIG` features for a specific package must be checked inside the recipe file, as there is no tool to list them all.

Configuration fragments for Kconfig-based projects

The Kconfig configuration infrastructure has become popular due to its flexibility and expressiveness. Although it started with Linux kernel, some other projects use the same infrastructure, such as U-Boot and BusyBox.

The configuration is based on select-based features where you can enable or disable a feature and save the result of this choice in a file for later use. So please consider the following figure:

```
CONFIG_TFTPD=y                CONFIG_TFTPD=n
    (a)                           (b)
```

Figure 13.9 – Enable or disable TFTPD on BusyBox KConfig

We have control whether the TFTPD support in BusyBox is enabled **(a)** or not **(b)**.

The Yocto Project provides a specialized class to handle the configuration of the Kconfig-based project, allowing minor modifications called configuration fragments. We can use this to enable or disable features for your machine, for example, when configuring `linux-yocto`, we can use `<layer>/recipes-kernel/linux/linux-yocto_%.bbappend` as in the following code:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://enable-can.cfg"
```

Figure 13.10 – The .bbappend content for applying a fragment

Every configuration fragment must use the `.cfg` file extension. So, the content of the `<layer>/recipes-kernel/linux/linux-yocto/linux-yocto/enable-can.cfg` file is shown here:

```
CONFIG_CAN=y
```

Figure 13.11 – The content of enable-can.cfg

We can use BitBake to configure or generate the Linux kernel configuration file. The `bitbake virtual/kernel -c menuconfig` command that allows us to configure the Linux kernel can be seen in the following screenshot:

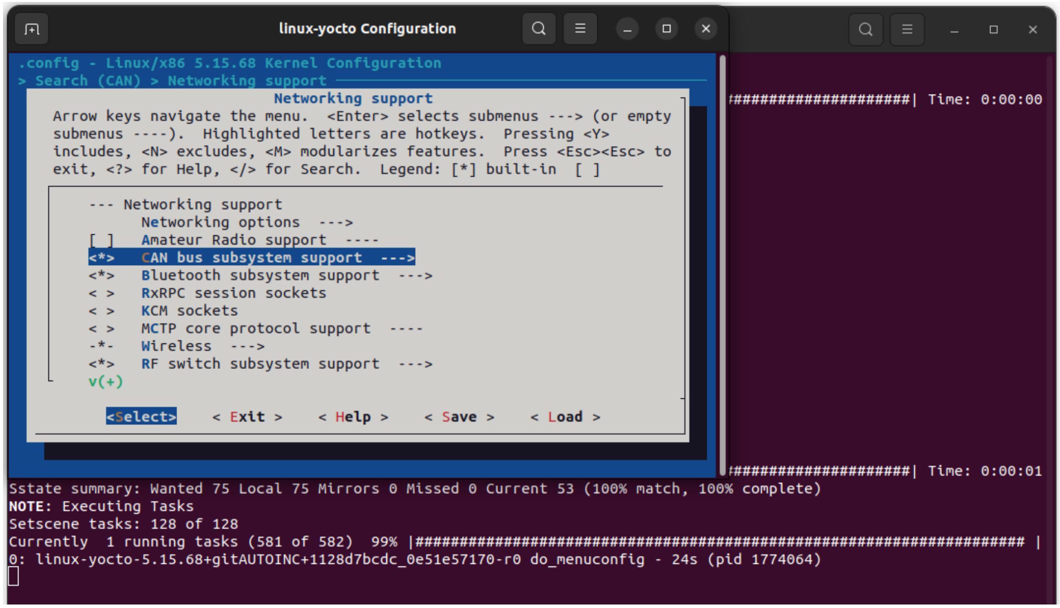


Figure 13.12 – Enabling CAN bus subsystem support using bitbake virtual/kernel -c menuconfig

Figure 13.12 shows how to enable CAN bus support using Linux kernel's `menuconfig`. The kernel configuration is changed when exiting and saving from `menuconfig`.

The next step is to create the fragment using `bitbake virtual/kernel -c diffconfig`, as shown in the following screenshot:

```
$ bitbake virtual/kernel -c diffconfig
Loading cache: 100% |###########################################| Time: 0:00:00
Loaded 2785 entries from dependency cache.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "2.0.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "x86_64-poky-linux"
MACHINE              = "qemux86-64"
DISTRO               = "poky"
DISTRO_VERSION       = "4.0.4"
TUNE_FEATURES        = "m64 core2"
TARGET_FPU           = ""
meta
meta-poky
meta-yocto-bsp       = "kirkstone:e81e703fb6fd028f5d01488a62dcfacbda16aa9e"
meta-oe              = "kirkstone:50d4a8d2a983a68383ef1ffec2c8e21adf0c1a79"
meta-newlayer
workspace            = "kirkstone:e81e703fb6fd028f5d01488a62dcfacbda16aa9e"

Initialising tasks: 100% |######################################| Time: 0:00:01
Sstate summary: Wanted 41 Local 41 Mirrors 0 Missed 0 Current 26
(100% match, 100% complete)
NOTE: Executing Tasks
Config fragment has been dumped into:
 /home/user/yocto/poky/build/tmp/work/qemux86_64-poky-linux/linux-
yocto/5.15.68+gitAUTOINC+1128d7bcdc_0e51e57170-r0/fragment.cfg
NOTE: Tasks Summary: Attempted 309 tasks of which 308 didn't need to be rerun
and all succeeded.
NOTE: Writing buildhistory
NOTE: Writing buildhistory took: 3 seconds
```

Figure 13.13 – The diffconfig option generates the configuration fragment

Figure 13.13 displays the log after the command. It is important to note that the fragment file is created under the `<build>/tmp/work/` directory, and the absolute path is shown in the log. We must copy this fragment file to the layer and use it in a `.bbappend` file in order to get it applied.

> **Tip**
>
> To save a complete configuration, we can use `bitbake virtual/kernel -c savedefconfig`. This command generates a `defconfig` file to replicate the same configuration. This is a complete configuration, not a fragment file.

The support for the configuration fragments works for the following recipes:

- Linux kernel
- U-Boot
- BusyBox

Those recipes also offer the `menuconfig` and `diffconfig` tasks.

Summary

In this chapter, we learned how to customize existing recipes using the `.bbappend` files and benefited from this by avoiding the duplication of source code. We saw how to enable or disable a feature, how to apply a patch, and how to use the configuration fragment support.

In the next chapter, we will discuss how the Yocto Project can help us with some legal aspects of producing a Linux-based system using packages under different licenses. We will understand which artifacts we need and how Poky can be configured to generate the artifacts that should be shared as part of the copyleft compliance accomplishment process.

14
Achieving GPL Compliance

In this chapter, we will see how we can ensure open source license compliance and use Poky to provide the artifacts needed, such as the source code, licensing text, and the list of derivative work. This is critical for most products introduced into the market nowadays, as open source code needs to live alongside proprietary code.

Understanding copyleft

Copyleft is a legal way to use copyright law to maximize rights and express freedom. However, it impacts our products. We must meet all obligations of open source and free software licenses.

When building a Linux distribution, at least two projects are used: the Linux kernel and a compiler. The **GNU Compiler Collection** (**GCC**) is still the most commonly used compiler. The Linux kernel uses the **General Public License** (**GPL**) **v2** license, and the GCC uses the **GPLv2**, **GPLv2.1**, and **GPLv3** licenses, depending on the project used.

However, a Linux-based system can include virtually all projects available worldwide, in addition to all applications made by the company for its product. So how do we know the number of projects and licenses included, and how do we fulfill copyleft compliance requirements?

> **Note**
>
> This chapter describes how the Yocto Project can help you in this task but be aware that you must know exactly what you need to provide and the possible license incompatibilities. Please consult your legal department or a copyright lawyer if you have any doubts.

In the next section, we will look at how the Yocto Project can help us with the most common tasks required for copyleft compliance.

Understanding copyleft compliance versus proprietary code

Understanding that proprietary and copyleft-covered codes can coexist in the same product is essential. Although this is the standard for most products available nowadays, we must be careful about the libraries we link the code to because some may have license compatibility issues.

One Linux-based system is a set of several projects, each one under a different license. The Yocto Project helps developers understand that most copyleft projects have the following obligations:

- The source code of the project
- The license for the project
- Any modification to the project
- Any script that is required to configure and build

If one project under copyleft is modified, the license text, the base source code, and any modification must be included in the final deliverable.

The assumptions cover most rights guaranteed by copyleft licenses. These are the parts where the Yocto Project can help us. However, before releasing anything, it is recommended that we audit all the materials to be released to make sure they're complete.

Managing software licensing with Poky

One important Poky feature is the ability to manage licenses. Most of the time, we only care about our bugs. However, managing licenses and the kinds of licenses used is crucial when creating a product.

Poky keeps track of licenses in every recipe. In addition, it has a strategy to work with proprietary applications during the development cycle.

> **Note**
> An important thing to know is that a recipe is released under a specific license and represents a project released under a different license. Therefore, the recipe and the project are two separate entities with specific licenses, so the two licenses must be considered part of the product.

In most recipes, information is a comment containing the copyright, license, and author name; this information pertains to the recipe itself. Then, there is a set of variables to describe the package license, and they are as follows:

- `LICENSE`: This describes the license under which the package was released.
- `LIC_FILES_CHKSUM`: This may not seem very useful at first sight. It describes the license file and checksum for a particular package, and we may find much variation in how a project describes its license. The most common license files are stored in `meta/files/common-licenses/`.

Some projects include a file, such as `COPYING` or `LICENSE`, which specifies the license for the source code. Others use a header note in each file or the main file. The `LIC_FILES_CHKSUM` variable has the checksum for the license text of a project; if any letters are changed, the checksum is changed as well. This ensures that any change is noted and consciously accepted by the developer. A license change may be a typo fix; however, it may also be a change in legal obligations, so the developer needs to review and understand the difference.

When a different license checksum is detected, BitBake launches a build error and points to the project that had its license changed. You must be careful when this happens, as the license change may impact the use of this software. To be able to build anything again, you must change the `LIC_FILE_CHKSUM` value accordingly and update the `LICENSE` field to match the license change. Your legal department should be consulted if the license terms have changed. It is also good practice to record the reason for the change in a commit message for future reference.

Understanding commercial licenses

By default, Poky does not use any recipe with a commercial license restriction. In the recipe file, the `LICENSE_FLAGS` variable is used to identify which license restriction that recipe has. For the `gstreamer1.0-plugins-ugly` recipe, the license-related variables are from *line 5* to *line 10*, as in *Figure 14.1*:

```
1 DESCRIPTION = "'Ugly GStreamer plugins"
2 HOMEPAGE = "https://gstreamer.freedesktop.org/"
3 BUGTRACKER = "https://gitlab.freedesktop.org/gstreamer/gst-plugins-ugly/-/issues"
4
5 LIC_FILES_CHKSUM = "file://COPYING;md5=a6f89e2100d9b6cdffcea4f398e37343 \
6                     file://tests/check/elements/xingmux.c;beginline=1;endline=21; \
7                     md5=4c771b8af188724855cb99cadd390068"
8
9 LICENSE = "LGPL-2.1-or-later & GPL-2.0-or-later"
10 LICENSE_FLAGS = "commercial"
```

Figure 14.1 – The license-related variables for the gstreamer1.0-plugins-ugly recipe

Line 10 indicates to Poky that this recipe requires the `commercial` license flag to be explicitly accepted for the recipe to be used. To allow the use of the `gstreamer1.0-plugins-ugly` recipe, we can use the following:

```
LICENSE_FLAGS_ACCEPTED = "commercial"
```

Figure 14.2 – How to accept to install the recipes with commercial license restrictions

We can add `LICENSE_FLAGS_ACCEPTED` in our custom distribution (e.g., `<my-layer>/conf/distro/my-distro.conf`) or inside `build/conf/local.conf` during the initial development stages. Using the `commercial` flag accepts the installation of every recipe that requires this flag. Still, sometimes we want to manage the recipes we use, demanding specific license terms. We can use the following form:

```
LICENSE_FLAGS_ACCEPTED = "commercial_gstreamer1.0-plugins-ugly"
```

Figure 14.3 – How to accept to only install gstreamer1.0-plugins-ugly

With the code from *Figure 14.3* we accept only the `commercial` license flag from `gstreamer1.0-plugins-ugly`, which is the recipe name. It is good practice to ensure this flag is enabled for a set of recipes that you have permission to use in a commercial setting. Please consult your legal department to ensure this.

Using Poky to achieve copyleft compliance

At this point, we know how to use Poky and understand its main goal. It is time to understand the legal aspects of producing a Linux-based system that uses packages under different licenses.

We can configure Poky to generate the artifacts that should be shared as part of the copyleft compliance process.

Understanding license auditing

To help us achieve copyleft compliance, Poky generates a license manifest during the image build, located at `build/tmp/deploy/licenses/<image_name-machine_name>-<datastamp>/`.

To demonstrate this process, we will use the `core-image-full-cmdline` image for the `qemux86-64` machine. To start with our example, look at the files under `build/tmp/deploy/licenses/core-image-full-cmdline-qemux86-64-<datastamp>`, which are as follows:

- `image_license.manifest`: This lists the recipe names, versions, licenses, and the packages files available in `build/tmp/deploy/image/<machine>` but not installed inside the **root filesystem** (**rootfs**). The most common examples are the bootloader, the Linux kernel image, and DTB files.
- `package.manifest`: This lists all the packages in the image.
- `license.manifest`: This lists the names, versions, recipe names, and licenses for all the installed packages. This manifest may be used for copyleft compliance auditing.

```
build/tmp/deploy/licenses/
├── acl
│   ├── COPYING
│   ├── COPYING.LGPL
│   ├── generic_GPL-2.0-or-later
│   ├── generic_LGPL-2.1-or-later
│   └── recipeinfo
...
├── bash
│   ├── COPYING
│   ├── generic_GPL-3.0-or-later
│   └── recipeinfo
...
├── core-image-full-cmdline-qemux86-64-20221220151845
│   ├── image_license.manifest
│   ├── license.manifest
│   └── package.manifest
...
└── zstd-native
    ├── COPYING
    ├── generic_BSD-3-Clause
    ├── generic_GPL-2.0-only
    ├── LICENSE
    └── recipeinfo

334 directories, 1444 files
```

Figure 14.4 – The directory layout for the license manifests under build/tmp/deploy

The license manifest for each recipe is under `build/tmp/deploy/licenses/<package-name>`. *Figure 14.4* shows the directory layout for some packages.

Providing the source code

The most apparent way Poky can help us to provide the source code of every project used in our image is by sharing the `DL_DIR` content. However, this approach has one crucial pitfall – any proprietary source code will be shared within `DL_DIR` if it is shared as is. In addition, this approach will share any source code, including parts not required by copyleft compliance.

Poky must be configured to archive the source code before the final image is created. To have it, we can add the following variables into `build/conf/local.conf`, as in *Figure 14.5*:

```
INHERIT += "archiver"
ARCHIVER_MODE[src] = "original"
```

Figure 14.5 – Configuring Poky to provide the source code of packages under copyleft

The `archiver` class copies the source code, patches, and scripts for the filtered license set. The default configuration is to have `COPYLEFT_LICENSE_INCLUDE` set to `"GPL* LGPL* AGPL*"` so the recipes that use source code licensed on those licenses are copied under the `build/tmp/deploy/sources/<architecture>` folders:

```
build/tmp/deploy/sources/
├── allarch
...
├── allarch-poky-linux
...
├── x86_64-linux
...
└── x86_64-poky-linux
    ├── acl-2.3.1-r0
    │   ├── 0001-test-patch-out-failing-bits.patch
    │   ├── 0001-tests-do-not-hardcode-the-build-path-into-a-helper-l.patch
    │   ├── acl-2.3.1.tar.gz
    │   ├── run-ptest
    │   └── series
    ...
    └── zstd-1.5.2-r0
        └── zstd-1.5.2-r0.tar.xz

195 directories, 1122 files
```

Figure 14.6 – The build/tmp/deploy/sources directory layout

The class also supports the `COPYLEFT_LICENSE_EXCLUDE` variable to ensure packages that use source code licensed on some specific licenses never go into the `sources` directory. By default, it is set to `"CLOSED Proprietary"`. *Figure 14.6* shows some recipe examples after baking `core-image-full-cmdline`.

Providing compilation scripts and source code modifications

With the configuration provided in the previous section, Poky will package the original source code for each project. If we want to include the patched source code, we will only use `ARCHIVER_MODE[src] = "patched"`; this way, Poky will wrap the project source code after the `do_patch` task. It includes modifications from recipes or the `.bbappend` file.

This way, the source code and any modifications can be shared easily. However, one kind of information still needs to be created: the procedure used to configure and build the project.

To have a reproducible build environment, we can share the configured project, in other words, the project after the `do_configure` task. We can add `ARCHIVER_MODE[src] = "configured"` to `build/conf/local.conf` for this.

It is important to remember that we must consider that the person on the other side may not use the Yocto Project for copyleft compliance; alternatively, if they are using it, they must know that the modification made to the original source code and configuration procedure is not available. This is why we share the configured project: it allows anyone to reproduce our build environment.

For all flavors of source code, the default resulting file is a tarball; other options will add `ARCHIVER_MODE[srpm] = "1"` to `build/conf/local.conf`, and the resulting file will be an **SRPM** package.

Providing license text

When providing the source code, the license text is shared inside it. If we want the license text inside our final image, we can add the following to `build/conf/local.conf`:

```
COPY_LIC_MANIFEST = "1"
COPY_LIC_DIRS = "1"
```

Figure 14.7 – How to configure Poky to deploy license text inside the final image

This way, the license files will be placed inside the `rootfs`, under `/usr/share/common-licenses/`.

Summary

In this chapter, we learned how Poky can help with copyleft license compliance and why it should not be used as a legal resource. Poky enables us to generate source code, reproduction scripts, and license text for packages used in our distribution. In addition, we learned that the license manifest generated within the image might be used to audit the image.

In the next chapter, we will learn how to use the Yocto Project's tools with real hardware. Then, we will use the Yocto Project to generate images for a few real boards.

15
Booting Our Custom Embedded Linux

It's time! We are ready to boot our custom-made embedded Linux, as we have learned the required concepts and gained enough knowledge about the Yocto Project and Poky. In this chapter, we will practice what we have learned so far about using Poky with external BSP layers to generate an image for use with the following machines and boot it using the SD card:

- BeagleBone Black
- Raspberry Pi 4
- VisionFive

The concepts in this chapter can be applied to every other board as long as the vendor provides a BSP layer to use with the Yocto Project.

Discovering the right BSP layer

In *Chapter 11*, *Exploring External Layers*, we learned that the Yocto Project allows for splitting its metadata among different layers. It organizes the metadata so we can choose which exact meta layer to add to our project.

The way to find the BSP for a board varies, but generally, we can find it by visiting `https://layers.openembedded.org`. We can search for the machine name and the website finds which layer contains it in its database.

Reviewing aspects that impact hardware use

The boards used in this chapter are well maintained and straightforward. However, using a different board is a valid choice, but your mileage may vary.

When we choose a board, the first step is to verify the quality of its software support. The low-level components comprise the following:

- Bootloader (such as U-Boot, GRUB, or systemd-boot)
- Linux kernel (with other required drivers such as GPU or WiFi)
- User space packages required by hardware acceleration

Those are critical but are not the only aspects to consider. The integration inside the Yocto Project, in a BSP layer form, reduces the friction in the board use as it usually provides the following:

- A reusable disk partition layout (e.g., a WIC `.wks` template)
- Ready-to-use machine definitions
- User space packages integrated for hardware acceleration (usable out of the box)

The maturity level of software enablement, and the Yocto Project BSP, significantly impact the friction involved in using the board and the out-of-the-box experience when using Poky for different boards.

Taking a look at widely used BSP layers

We will see a list of widely used BSP layers in this chapter. This should not be taken as a complete list or as a definitive one. Still, we want to facilitate your search for the required layer in case you have one board of a specific vendor next to you. This list is as follows, in alphabetic order:

- *Allwinner*: This has the `meta-allwinner` layer
- *AMD*: This has the `meta-amd` layer
- *Intel*: This has the `meta-intel` layer
- *NXP*: This has the `meta-freescale` and `meta-freescale-3rdparty` layers
- *Raspberry Pi*: This has the `meta-raspberrypi` layer
- *RISC-V*: This has the `meta-riscv` layer
- *Texas Instruments*: This has the `meta-ti` layer

In the next sections, we start to work with the example boards.

Using physical hardware

To ease the exploration of the Yocto Project's capabilities, it is good to have a real board so we can enjoy the experience of booting our customized embedded system. For this, we have tried to collect the most widely available boards so the chances of you owning one are higher.

The next sections will cover the steps for the following boards:

- *BeagleBone Black*: BeagleBone Black is community-based, with members worldwide. Further information is available at `https://beagleboard.org/black/`.
- *Raspberry Pi 4*: The most famous ARM64-based board with the broadest community spread worldwide. See more details at `https://www.raspberrypi.org/`.
- *VisionFive*: The world's first generation of affordable RISC-V boards designed to run Linux. See more details at `https://www.starfivetech.com/en`.

All the boards listed are maintained by non-profit organizations based on education and mentoring, which makes the community a fertile place to discover the world of embedded Linux. The following table summarizes the boards and their main features:

Board version	**Features**
BeagleBone Black	TI AM335x (single-core) 512 MB RAM
Raspberry Pi 4	Broadcom BCM2711 64bit CPU (quad-core) 1 GB up to 8 GB RAM
VisionFive	U74 Dual-Core 8 GB RAM

Table 15.1 – The hardware specification for the covered boards

In the next sections, we are going to bake and boot the Yocto Project image for each one of the suggested machines. It's recommended that you only read the section for the board that you own. Make sure to consult the board's documentation in order to understand how to prepare the board for the operation.

BeagleBone Black

In the next two sections, we go through the steps for baking and booting an image for the *BeagleBone Black* board.

Baking for BeagleBone Black

To use this board, we can rely on the `meta-yocto-bsp` layer, which is included by default in Poky. The meta layer can be accessed at `https://git.yoctoproject.org/meta-yocto/tree/meta-yocto-bsp?h=kirkstone`.

To create the source structure, please download Poky using the following command line:

```
git clone git://git.yoctoproject.org/poky -b kirkstone
```

After completing this, we must create the build directory we use for our builds. We can do this using the following command line:

```
source oe-init-build-env build
```

After we have the `build` directory and the BSP layers properly set up, we can start the build. Inside the `build` directory, we must call the following command:

```
MACHINE=beaglebone-yocto bitbake core-image-full-cmdline
```

The `MACHINE` variable can be changed depending on the board we want to use or set in `build/conf/local.conf`.

Booting BeagleBone Black

After the build process is over, the image will be available inside the `build/tmp/deploy/images/beaglebone-yocto/` directory. The file we want to use is `core-image-full-cmdline-beaglebone-yocto.wic`.

Make sure you point to the right device and double-check to not write on your hard disk.

To copy the `core-image-full-cmdline` image to the SD card, we should use the `dd` utility, as follows:

```
sudo dd if=core-image-full-cmdline-beaglebone-yocto.wic of=/
dev/<media>
```

After copying the content to the SD card, insert it into the SD card slot, connect the HDMI cable, and power on the machine. It should boot nicely.

> **Note**
> The BeagleBone Black boot sequence starts trying to boot from eMMC and only tries to boot from the SD card in case the eMMC boot fails. Clicking the **USER**/**BOOT** button when powering on will temporarily change the boot order, making sure the boot is from the SD card. To further tailor these instructions for your board, please refer to the documentation at `http://www.beagleboard.org/black`.

Raspberry Pi 4

In the next two sections, we go through the steps for baking and booting an image for the *Raspberry Pi 4* board.

Baking for Raspberry Pi 4

To add this board support to our project, we need to include the `meta-raspberrypi` meta layer, which is the BSP layer with support for the Raspberry Pi boards, including the Raspberry Pi 4, but not limited to it. The meta layer can be accessed at `http://git.yoctoproject.org/cgit.cgi/meta-raspberrypi/log/?h=kirkstone`.

To create the source structure, please download Poky using the following command line:

```
git clone git://git.yoctoproject.org/poky -b kirkstone
```

After completing this, we must create the `build` directory we use for our builds and add the BSP layer. We can do this using the following command lines:

```
source oe-init-build-env build
bitbake-layers layerindex-fetch meta-raspberrypi
```

After we have the `build` directory and the BSP layers properly set up, we can start the build. Inside the `build` directory, we must call the following command:

```
MACHINE=raspberrypi4 bitbake core-image-full-cmdline
```

The `MACHINE` variable can be changed depending on the board we want to use or set in `build/conf/local.conf`.

Booting Raspberry Pi 4

After the build process is over, the image will be available inside the `build/tmp/deploy/images/raspberrypi4/` directory. The file we want to use is `core-image-full-cmdline-raspberrypi4.wic.bz2`.

Make sure you point to the right device and double-check to not write on your hard disk.

To copy the `core-image-full-cmdline` image to the SD card, we should use the `dd` utility, as follows:

```
bzcat core-image-full-cmdline-raspberrypi4.wic.bz2 | sudo dd
of=/dev/<media>
```

After copying the content to the SD card, insert it into the SD card slot, connect the HDMI cable, and power on the machine. It should boot nicely.

VisionFive

In the next two sections, we go through the steps for baking and booting an image for the *VisionFive* board.

Baking for VisionFive

To add this board support to our project, we need to include the `meta-riscv` meta layer, which is the BSP layer with support for RISC-V-based boards, including the VisionFive, but not limited to it. The meta layer can be accessed at `https://github.com/riscv/meta-riscv/tree/kirkstone`.

To create the source structure, please download Poky using the following command line:

```
git clone git://git.yoctoproject.org/poky -b kirkstone
```

After completing this, we must create the `build` directory we'll use for our builds and add the BSP layer. We can do this using the following command lines:

```
source oe-init-build-env build
bitbake-layers layerindex-fetch meta-riscv
```

After we have the `build` directory and the BSP layers properly set up, we can start the build. Inside the `build` directory, we must call the following command:

```
MACHINE=visionfive bitbake core-image-full-cmdline
```

The `MACHINE` variable can be changed depending on the board we want to use or set in `build/conf/local.conf`.

Booting VisionFive

After the build process is over, the image will be available inside the `build/tmp/deploy/images/visionfive/` directory. The file we want to use is `core-image-full-cmdline-visionfive.wic.gz`.

Make sure you point to the right device and double-check to not write on your hard disk.

To copy the `core-image-full-cmdline` image to the SD card, we should use the `dd` utility, as follows:

```
zcat core-image-full-cmdline-visionfive.wic.gz | sudo dd of=/dev/<media>
```

After copying the content to the SD card, insert it into the SD card slot, connect the HDMI cable, and power on the machine.

Note

VisionFive doesn't have a default boot target and requires manual intervention to boot. Please use the following commands inside the U-Boot prompt using a serial console:

```
setenv bootcmd "run distro_bootcmd"
saveenv
boot
```

The command `saveenv` is optional to make the new configuration persist so that it can work out of the box after reboot.

See how to get the serial console in the *Quick Start Guide* (`https://doc-en.rvspace.org/VisionFive/Quick_Start_Guide/`).

Taking the next steps

Phew! We got it done! Now you should know the Yocto Project build system basics and be capable of extending your other areas of knowledge. We tried covering the most common daily tasks using the Yocto Project. There are a few things you might want to practice:

- Creating `bbappend` files to apply patches or make other changes to a recipe
- Making your custom images
- Changing the Linux kernel configuration file (`defconfig`)
- Changing the BusyBox configuration and including the configuration fragments to add or remove a feature in a layer
- Adding a new recipe for a package
- Making a product layer with your product-specific machines, recipes, and images

Remember, the source code is the ultimate knowledge source, so use it.

When looking for how to do something, finding a similar recipe saves you time testing different approaches to solve the problem.

Eventually, you'll likely see yourself in a position to fix or enhance something on OpenEmbedded Core, a meta layer, or in a BSP. So, don't be afraid – send the patches and take the feedback and requests for changes as an opportunity to learn and improve your way of solving a problem.

Summary

We learned how to discover the BSP for a board we want to use in our project. We consolidated our Yocto Project knowledge by adding external BSP layers and using these in real boards with a generated image. We also consolidated the necessary background information to learn about any other aspect of the Yocto Project you may need.

In the next chapter, we will explore how using QEMU speeds up product development by enabling us to not rely on hardware for every development cycle.

16
Speeding Up Product Development through Emulation – QEMU

In this chapter, we explore the possibilities of shortening product development through emulation and reducing the dependency on real hardware for most development. You will come to understand the benefits of using QEMU over hardware and when choosing real hardware is preferable. We also describe the `runqemu` capabilities and demonstrate some use cases.

What is QEMU?

Quick EMUlator (**QEMU**) is a free, open source software tool that allows users to run multiple architectures on the same physical machine. It is a system emulator that can virtualize complete device hardware, including the CPU, memory, storage, and peripherals.

Using QEMU for testing and debugging can save time and effort during development. It allows developers to test their code in various simulated environments.

Among other things, the Yocto Project uses QEMU to run automated **Quality Assurance** (**QA**) tests on final images shipped with each release. Within the context of the Yocto Project, QEMU allows you to run a complete image you have built using the Yocto Project as another task on your build system. In addition, QEMU helps to run and test images and applications on supported Yocto Project architectures without having actual hardware.

What are the benefits of using QEMU over hardware?

There are several situations where it may be more practical to use QEMU instead of real hardware for testing and debugging:

- It allows you to quickly and easily test your code in various simulated environments without constantly deploying it to the target device
- If you don't have the hardware that the software will be running on or if its availability is limited
- When you need to test software on multiple hardware platforms without having to set up multiple physical machines
- When you want to debug software in a controlled environment, such as reduced memory availability, to observe its behavior
- When you want to validate software that isn't hardware specific and wish to reduce the time needed for testing, such as flashing, board wiring, and so on

However, it is essential to note that QEMU is a software emulator, which may not be a perfect substitute for real hardware at all times. Therefore, testing software on real hardware may be necessary to ensure it works correctly.

When is choosing real hardware preferable?

There are several situations where it may be more practical, and even required, to use real hardware instead of QEMU for testing and debugging, such as the following:

- When the software relies on specific hardware features, for example, a particular **Video Processing Unit** (**VPU**) or **Graphics Processing Unit** (**GPU**) feature
- When evaluating the software performance, QEMU may not be able to replicate the performance of real hardware

While QEMU can be a valuable tool for testing and debugging software, it is not always a perfect substitute for real hardware.

Using runqemu capabilities

QEMU is deeply integrated into the Yocto Project, and it is crucial to learn how to take advantage of this integration so we can plan the testing of our projects. The `runqemu` usage lists the variety of options available, which you can see in the following figure:

```
$ runqemu --help

Usage: you can run this script with any valid combination
of the following environment variables (in any order):
  KERNEL - the kernel image file to use
  BIOS - the bios image file to use
  ROOTFS - the rootfs image file or nfsroot directory to use
  DEVICE_TREE - the device tree blob to use
  MACHINE - the machine name (optional, autodetected from KERNEL filename if unspecified)
  Simplified QEMU command-line options can be passed with:
    nographic - disable video console
    novga - Disable VGA emulation completely
    sdl - choose the SDL UI frontend
    gtk - choose the Gtk UI frontend
    gl - enable virgl-based GL acceleration (also needs gtk or sdl options)
    gl-es - enable virgl-based GL acceleration, using OpenGL ES (also needs gtk or sdl options)
    egl-headless - enable headless EGL output; use vnc (via publicvnc option) or spice to see it
    (hint: if /dev/dri/renderD* is absent due to lack of suitable GPU, 'modprobe vgem' will create
    one suitable for mesa llvmpipe software renderer)
    serial - enable a serial console on /dev/ttyS0
    serialstdio - enable a serial console on the console (regardless of graphics mode)
    slirp - enable user networking, no root privilege is required
    snapshot - don't write changes back to images
    kvm - enable KVM when running x86/x86_64 (VT-capable CPU required)
    kvm-vhost - enable KVM with vhost when running x86/x86_64 (VT-capable CPU required)
    publicvnc - enable a VNC server open to all hosts
    audio - enable audio
    [*/]ovmf* - OVMF firmware file or base name for booting with UEFI
  tcpserial=<port> - specify tcp serial port number
  qemuparams=<xyz> - specify custom parameters to QEMU
  bootparams=<xyz> - specify custom kernel parameters during boot
  help, -h, --help: print this text
  -d, --debug: Enable debug output
  -q, --quiet: Hide most output except error messages

Examples:
  runqemu
  runqemu qemuarm
  runqemu tmp/deploy/images/qemuarm
  runqemu tmp/deploy/images/qemux86/<qemuboot.conf>
  runqemu qemux86-64 core-image-sato ext4
  runqemu qemux86-64 wic-image-minimal wic
  runqemu path/to/bzImage-qemux86.bin path/to/nfsrootdir/ serial
  runqemu qemux86 iso/hddimg/wic.vmdk/wic.vhd/wic.vhdx/wic.qcow2/wic.vdi/ramfs/cpio.gz...
  runqemu qemux86 qemuparams="-m 256"
  runqemu qemux86 bootparams="psplash=false"
  runqemu path/to/<image>-<machine>.wic
  runqemu path/to/<image>-<machine>.wic.vmdk
  runqemu path/to/<image>-<machine>.wic.vhdx
  runqemu path/to/<image>-<machine>.wic.vhd
```

Figure 16.1 – The runqemu usage

There are a few use cases of QEMU that are important to highlight:

- Allows choosing different kernel images for testing
- Allows choosing different `rootfs` for booting
- The capability to pass boot arguments for the kernel
- Supports the use of a graphical environment with OpenGL or OpenGL ES options

- It can pass extra QEMU command-line parameters
- Allows the use of serial console-only for rapid image testing
- Testing the audio stack support
- Testing different init systems (e.g., systemd)

In the following few sections, we use the `qemux86-64` machine as a reference to cover some common use cases, illustrating the main `runqemu` capabilities.

Using runqemu to test graphical applications

When we aim to validate the application, ignoring the embedded device GPU performance, we can rely on QEMU for such validation, for example, a Qt or GTK+ application. At first, we need to build the `core-image-weston` image. Next, we can run the validation as follows:

```
$ runqemu qemux86-64 gl sdl core-image-weston
runqemu - INFO - Running MACHINE=qemux86-64 bitbake -e ...
runqemu - INFO - Continuing with the following parameters:
KERNEL: [/home/user/yocto/poky/build/tmp/deploy/images/qemux86-64/bzImage--
5.15.68+git0+1128d7bcdc_0e51e57170-r0-qemux86-64-20221230201037.bin]
MACHINE: [qemux86-64]
FSTYPE: [ext4]
ROOTFS: [/home/user/yocto/poky/build/tmp/deploy/images/qemux86-64/core-image-weston-
qemux86-64.ext4]
CONFFILE: [/home/user/yocto/poky/build/tmp/deploy/images/qemux86-64/core-image-weston-
qemux86-64.qemuboot.conf]
```

Figure 16.2 – The log after running QEMU with graphic support

Next, you see the execution of `core-image-weston` inside QEMU:

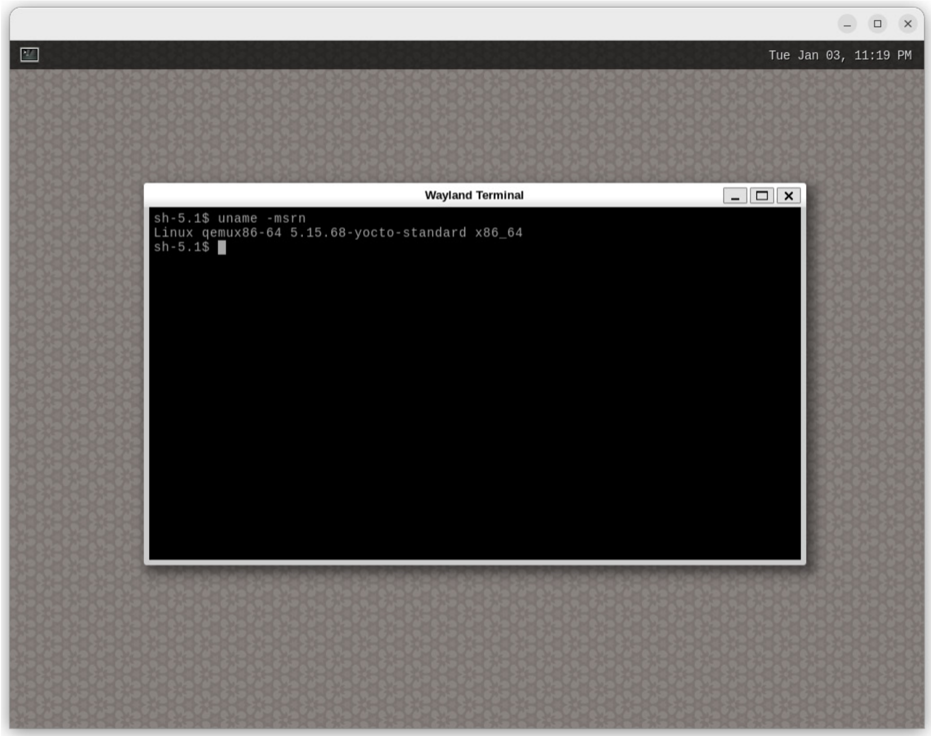


Figure 16.3 – Screenshot of QEMU running core-image-weston

The preceding screenshot shows the Wayland Terminal open, showing the information of the running Linux kernel.

Using runqemu to validate memory constraints

When we aim to validate the application memory usage, we can rely on QEMU for such validation. At first, we need to build the `core-image-full-cmdline` image and run QEMU with the following command line:

```
$ runqemu qemux86-64 qemuparams="-m 128" core-image-full-cmdline
runqemu - INFO - Running MACHINE=qemux86-64 bitbake -e ...
runqemu - INFO - Continuing with the following parameters:
KERNEL: [/home/user/yocto/poky/build/tmp/deploy/images/qemux86-64/bzImage--
5.15.68+git0+1128d7bcdc_0e51e57170-r0-qemux86-64-20221230201037.bin]
MACHINE: [qemux86-64]
FSTYPE: [ext4]
ROOTFS: [/home/user/yocto/poky/build/tmp/deploy/images/qemux86-64/core-image-full-
cmdline-qemux86-64.ext4]
CONFFILE: [/home/user/yocto/poky/build/tmp/deploy/images/qemux86-64/core-image-full-
cmdline-qemux86-64.qemuboot.conf]
```

Figure 16.4 – The log after running QEMU with 128 MB of RAM

In the following screenshot, we can see the amount of memory in use inside QEMU:

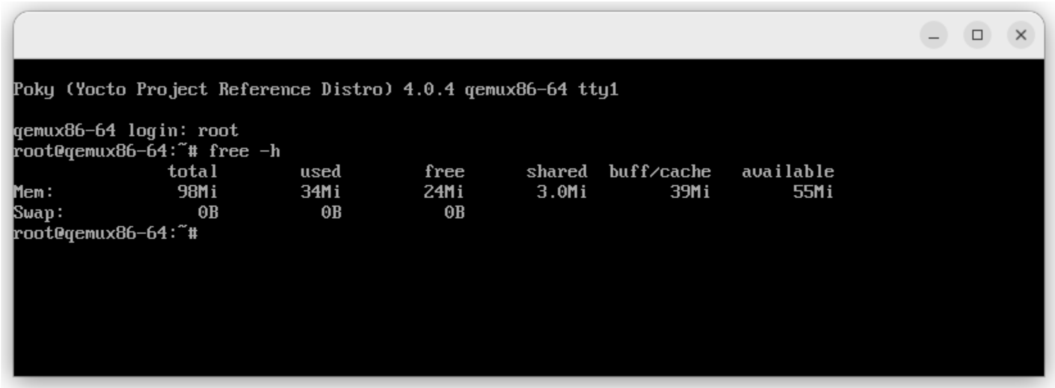


Figure 16.5 – Screenshot of QEMU running core-image-full-cmdline with 128 MB of RAM

Changing the command line used to run QEMU can help us test a set of different memory sizes via emulation.

Using runqemu to help with image regression tests

The Yocto Project provides an automated testing framework, a crucial part of the Yocto Project Quality Assurance process. The integration or validation testing support uses the `testimage` class to execute the images inside the target.

> **Tip**
>
> The testing framework can test existing recipes and images and be enhanced with custom tests to validate new applications and integrations. The testing framework capabilities are described in the section *Types of Testing Overview from Yocto Project Tests* (`https://docs.yoctoproject.org/4.0.4/test-manual/intro.html#yocto-project-tests-types-of-testing-overview`).

First, we enabled the `testimage` support by adding `IMAGE_CLASSES += "testimage"` in `build/conf/local.conf` and made sure to build the `core-image-weston` image.

> **Warning**
>
> During the image testing, the `sudo` command is used for networking setup and may trigger an error depending on your host configuration. Check *Yocto Project Development Tasks Manual*, in the *Enabling Runtime Tests on QEMU* section (`https://docs.yoctoproject.org/4.0.4/dev-manual/common-tasks.html#enabling-runtime-tests-on-qemu`) for how to avoid those errors.

Then, we must build the `core-image-weston` image. We are ready now to start the execution of `testimage` with the following command:

```
$ bitbake -c testimage core-image-weston
...
Initialising tasks: 100% |#########################################| Time: 0:00:01
Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 236 (0% match, 100%
complete)
NOTE: Executing Tasks
QMP Available for connection at /home/user/yocto/poky/build/tmp/.9prfu2ob
QMP connected to QEMU at 12/31/22 13:19:02 and took 0.6354751586914062 seconds
QMP released QEMU at 12/31/22 13:19:03 and took 0.44214820861816406 seconds from
connect
Not starting HTTPService for directory tmp/deploy/deb/ which doesn't exist
Test requires apt to be installed
Stopped HTTPService on 0.0.0.0:0
Test requires autoconf to be installed
Test requires gtk+3 to be installed
...
Output from runqemu:
runqemu - INFO - SIGTERM received
runqemu - INFO - Cleaning up
runqemu - INFO - Host uptime: 325.28

RESULTS:
RESULTS - date.DateTest.test_date: PASSED (0.30s)
RESULTS - df.DfTest.test_df: PASSED (0.07s)
...
SUMMARY:
core-image-weston () - Ran 68 tests in 31.666s
core-image-weston - OK - All required tests passed (successes=34, skipped=34,
failures=0, errors=0)
NOTE: Tasks Summary: Attempted 1085 tasks of which 1084 didn't need to be rerun and
all succeeded.
```

Figure 16.6 – The result of running the testimage task for core-image-weston

In the preceding log, we see the regression test results.

Summary

In this chapter, we have learned how to use QEMU and how its capabilities can shorten the development cycle by emulating when possible and describing when it is not possible. It also presented some `runqemu` use cases.

In the final chapter, we offer a list of good practices that authors have been using over the years in the development of Yocto Project-based products.

17
Best Practices

This chapter aims to provide insight into our (the authors') personal experience in working with embedded devices and embedded Linux development over the years. We have gathered some aspects that are often underestimated or wholly neglected to serve as inspiration for you in your next project.

We have split this chapter into two independent parts, one about the guidelines related to the Yocto Project specifics and the other about more general aspects of a project. This is so that you don't have to study the two sections in a particular order.

Guidelines to follow for Yocto Project

This section aims to gather some guidelines for aspects of the Yocto Project metadata and project organization tips that make our life easier in terms of short- and long-term maintenance.

Managing layers

As our journey in product development advances, we will naturally use multiple repositories to meet the needs we face. Keeping track of the repositories is a complex challenge as we need to do the following:

- Make sure we can reproduce a previous build in the future
- Allow multiple team members to work in the same code base
- Validate the changes we make using Continuous Integration tools
- Avoid subtle changes in the layers we use

Those goals are intimidating, but a few tools are in use, with different strategies to overcome those challenges.

The simplest solution uses the `image-buildinfo` class (`https://docs.yoctoproject.org/4.0.4/ref-manual/classes.html#image-buildinfo-bbclass`), which writes a plain text file containing build information and layers revisions to the target filesystem at `${sysconfdir}/buildinfo` by default. Some tools have been developed that can help this process. These tools are discussed as follows:

- Google developed the **repo** (`https://source.android.com/docs/setup/download#repo`) tool for Android development. It has been adopted for use in other projects. A critical aspect of repo is that it requires some tooling to integrate with Yocto Project-based projects to automate the build directory and environment configuration. See the *O.S. Systems Embedded Linux project* (`https://github.com/OSSystemsEmbeddedLinux/ossystems-embedded-linux-platform`) as inspiration for using repo in your projects.
- Siemens developed **kas** (`https://github.com/siemens/kas`) to provide an easy mechanism for downloading sources, automating the build directory and environment configuration, and so on.
- Garmin developed **Whisk** (`https://github.com/garmin/whisk`) to manage complex product configurations using OpenEmbedded and the Yocto Project. The key features are a single source tree, multiple axes of configuration, multiple product builds, isolated layer configuration, and so on.
- Agilent developed **Yocto Buddy** (`https://github.com/Agilent/yb`). The design aims to ease the setup and keep Yocto Project-based environments synchronized. Yocto Buddy was inspired by all previously mentioned tools and is still early in development.

This is a subset of existing tools and shouldn't be considered a complete list. Ideally, you should play with them before deciding, as the choice depends on the project use case and team expertise.

Avoid creating too many layers

A significant advantage of the Yocto Project is that it has the ability to use and create multiple layers. It allows us to do the following:

- Reuse BSP layers from semiconductor vendors
- Reduce duplication of work by sharing reusable blocks to enable the use of new or specific applications, programming languages, and so on.

However, creating multiple layers may be unproductive when developing a project or a set of products. For example, the development of BSP-only layers makes sense in the following situations:

- The board is the product, as in the **System on Module** (**SoM**) vendors' case
- When external access to the layer is critical, however, we want to limit the access for the non-BSP source

Using a single layer for the product, or even the company, has many advantages, such as the following:

- Facilitating the development of reusable components such as a `packagegroup` package for development tools or network utilities shared by multiple products
- Reducing the risk of unexpected side effects due to changes for a specific product or board
- Increasing the reuse of bug fixes across multiple products and reuse of BSP low-level components such as the Linux kernel or bootloader
- Boosting standardization across multiple products, reducing the learning curve for new team members

The decision to use one or more layers depends on several aspects; however, we recommend starting simple and, in the future, splitting the layer if required.

Prepare the product metadata for new Yocto Project releases

As our product grows, so does our metadata and the need for good organization. Some use cases commonly seen during product development are as follows:

- The need to backport a new recipe version due to a bug fix or a feature
- A missing package configuration or bug fix is not yet available in the Yocto Project recipe

We use two recipe directories to organize this kind of content:

- `recipes-backport`: Backports of recipes coming from new Yocto Project releases
- `recipes-staging`: New recipes or `bbappend` files adding missing package configurations or bug fixes

We continuously send new recipes or bug fixes from `recipes-staging` to the respective upstream project (for example, OpenEmbedded Core). Then, when the patch is accepted, we move this change from `recipes-staging` to the `recipes-backport` directory. This approach allows us to keep track of pending upstreaming tasks and easily upgrade our meta layer to a new Yocto Project release. Furthermore, we can quickly act on the backport directory and remove it.

Create your custom distro

When using the Yocto Project, we usually add many configurations in `build/conf/local.conf`. However, as discussed in the book, this is bad as it is not at source control management and is likely to differ among developers. Using a custom distribution has many benefits, and some of them are highlighted here:

- Allows consistent use among multiple developers
- Provides a clear view of the different `DISTRO_FEATURES` we use when compared to our base distribution (for example, `poky`)

- Provides a central place where we can have a global view of all the required recipe configurations we need for our product, reducing the number of `bbappend` files required to configure our recipes (for example, `PACKAGECONFIG:pn-<myrecipe>:append = " myfeature"`)

Besides those more technical aspects, using a custom distro also allows the proper branding of SDK or other Yocto Project-generated artifacts.

We learned how to create a custom distribution in the *Using a custom distribution* sectiown in *Chapter 12*, *Creating Custom Layers*.

Avoid reusing existing images for your product

Images are where everything fits together. When we are developing a product, it is important to minimize the number of packages we have installed in our images for multiple reasons:

- Reducing the rootfs size
- Reducing the build time
- Reducing the number of licenses to deal with
- Reducing the surface of attack for security breaches

A typical starting point is copying the `core-image-base.bb` file to our custom layer as `myproduct-image.bb` and extending it, adding what we need for the product's image. In addition, we create an image called `myproduct-image-dev.bb` for use during development and make sure it requires `myproduct-image.bb` along with the artifacts used only for development, avoiding code duplication. This way, we have two images for production and development, but they share the same core features and packages.

Standard SDK is commonly undervalued

Application development implies an interactive process, mainly because we usually continuously build the application until we accomplish what we aim for. This use case is not well suited for the Yocto Project, mainly for the following reasons:

- Every time we start the build of a recipe, it discards the previous build objects
- The time needed for deploying the application or image is much longer
- A lack of proper integration in the IDE environment

There are alternatives for a few of those topics, such as using `devtool` to reuse the build objects and helping to deploy the application. We saw how to use `devtool` in the *Deploying to the target using devtool* and *Building a recipe using devtool* sections from *Chapter 9*, *Developing with the Yocto Project*, but the development experience is still cumbersome.

Using Standard SDK for application and other components' development, such as the Linux kernel and bootloader, is still preferable. This way, we focus on faster development, postponing or parallelizing the Yocto Project integration task.

Avoid too many patches for Linux kernel and bootloader modifications

The need for patches in the Linux kernel and bootloader is inherent to embedded Linux development, as we rarely use the hardware without any changes. The level of modification on those components is related to your hardware design, for example:

- Using a **Single-Board Computer** (**SBC**), the number of changes should be minimal
- In the use of **System-On-Module** (**SOM**) with a custom baseboard, the number of changes could vary depending on the number of modifications from the vendor baseboard hardware design
- Ultimately, the use of custom hardware design implies the development of a custom BSP and, consequently, a considerable number of modifications

Those are not set in stone. So, for example, consider starting the project using an SBC. Later, we find out that the vendor does not provide a good reference BSP, so the number of modifications and amount of work for the BSP will increase considerably.

When we have small changes, it is better to tackle the changes as patch files added to the component recipe. But when the effort to maintain the component increases, it justifies having a separate fork of that component to keep all the changes in place. Using a repository fork gives us the following advantages:

- The history of the changes
- Different branches or tags for development and production
- The possibility of merging with other providers
- It allows the use of much simpler recipes, as we don't need to carry on individual patches

In summary, we should use the strategy that makes sense for the project. Eventually, this will change, but using the right approach reduces the total effort to support the hardware in use properly.

Avoid using AUTOREV as SRCREV

The use of `AUTOREV` as `SRCREV` is usually applied when developing a product. We must interactively change the code and try that code inside the Yocto Project. That said, this comes with a couple of drawbacks:

- It is hard to reproduce the previous build as every time we rebuild our image, it may use a different revision for our recipe.

- The `AUTOREV` value is only applied when BitBake invalidates the cache of a specific recipe. That happens when we modify the recipe itself or when we change something that triggers the BitBake cache rebuild, such as changing any `.conf` file.

Those drawbacks make `AUTOREV` very fragile, and other alternatives can cover the interactive code change more consistently. Typically, `devtool` is used as it allows us to change the code directly in the workspace and forces the recipe to use this as the source. Another alternative is to use the `externalsrc.bbclass` class (`https://docs.yoctoproject.org/4.0.4/index.html#ref-classes-externalsrc`), which allows us to configure a recipe to use a directory as the source for the build.

Create a Software Bill of Materials

The Poky build system can describe all the components used in an image from the licenses for each software component. This description is generated as a **Software Bill of Materials** (**SBOM**) using the **Software Package Data Exchange** (**SPDX**) standard (`https://spdx.dev/`). Using the SPDX format has the advantage of leveraging existing tooling, allowing extra automation, which is impossible using Poky's standard license output format.

The SBOM is critical to ensure open source license compliance. However, the SBOM is not generated by default. You can refer to the *Creating a Software Bill of Materials* section from *The Yocto Project Development Tasks Manual* (`https://docs.yoctoproject.org/4.0.4/dev-manual/common-tasks.html#creating-a-software-bill-of-materials`).

Guidelines to follow for general projects

This section discusses some project-related guidelines to follow to reduce the general project risk and avoid common pitfalls.

Continuously monitor the project license constraints

Depending on the project we are working on, license compliance might be a big or a small topic. Some projects have very restricted license constraints, such as the following:

- The inability to use GPLv3-released software
- Copyleft contamination of project-specific intellectual property
- Company-wise license constraints

The advice is to start this process at the beginning of the project, reducing the amount of rework throughout the project. However, the project license constraints and the project component's licenses may change, requiring us to monitor our license compliance continuously.

Security can harm your project

In our hyper-connected era, every connected device is a potential target for a security attack. As embedded device developers, we should contribute to a safer place. We should do the following:

- Scan our embedded Linux software for known security flaws
- Monitor critical software for security fixes
- Implement a process for fixing field devices

We can use the Yocto Project infrastructure, as discussed in the *Checking for Vulnerabilities* section of *Yocto Project Development Tasks Manual* (`https://docs.yoctoproject.org/4.0.4/dev-manual/common-tasks.html#checking-for-vulnerabilities`), to scan for known **Common Vulnerabilities and Exposures** (**CVE**) for our recipes. We should not be limited to this as our BSP components might also require security fixes, which the BSP vendors commonly neglect. Still, the paranoia level depends on the project niche.

Don't underestimate maintenance costs

At first, upstreaming our changes might not seem strategic for the following reasons:

- Upstreaming uses resources to adapt modifications
- Upstream review feedback may require additional interactions and rework
- Development work not directly connected to the product needs to be done

Usually, development and management teams underestimate the total cost of maintenance. But unfortunately, this is frequently the most expensive part of the project, as it lasts for years. Upstreaming our changes to the respective project allows us to do the following:

- Avoid work duplication over the years
- Reduce the friction during upgrades for new Yocto Project releases
- Receive critical and constructive feedback about the changes we are upstreaming
- Reduce the amount of work with security updates and bug fixes
- Reduce the amount of code we have to maintain

The upstream work is continuous. Every time we add a new feature, we potentially increase the gap between our code and the upstream. Therefore, we may postpone the upstreaming work, but the upstreaming costs will be multiplied when you work on updating to the next Yocto Project release.

Tackle project risk points and constraints as soon as possible

As the software and hardware must work together, a few aspects directly depend on our hardware design. To reduce the project risk, we should anticipate as many critical software and hardware requirements as possible so we can validate some aspects, such as the following:

- Is the amount of memory we intend to use enough or too much?
- Is the amount of power the hardware uses sufficient for our constraints?
- Is the target GPU capable of rendering the animations we need?
- Do all the planned peripheral devices have available Linux kernel drivers ready for use or do we need to plan the development for those?

The preceding questions can be answered using a reference or well-known board, which we have ready to use BSP. This allows us to produce a **Minimal Viable Product** (**MVP**) without the need to design our custom hardware. After we validate the project's risks and constraints, those boards are still valuable assets for the following:

- Continuing the development of our software until the custom board and BSP are ready for use
- As a base of comparison with our custom design
- As a reference to verify whether a bug is specific to our custom board and BSP

Considering we can develop our software using a reference or a well-known board, we should postpone the design of a custom board for as long as possible. Delaying the design gives us the freedom to change many aspects of our project, such as changing a peripheral because of a specific driver or even changing the planned CPU and memory capabilities after maturing the application and features.

When we finally decide to go with a custom design, we should keep it as close as possible to the board we choose as a reference. But, of course, sometimes we need to deviate from the reference design. Still, it comes with the risk of introducing design issues and increasing the cost of our custom BSP.

Summary

Phew! In this final chapter, you have been introduced to a set of good practices that the authors have been using in their real-life projects. We hope they have given you some points to consider when planning your next project.

Throughout the book, we have covered the necessary background information for you to learn any other aspect of the Yocto Project that you may need on your own. So, you now have a general understanding of what is happening behind the scenes when you ask BitBake to build a recipe or an image. From now on, you are ready to free your mind and try new things. The ball is in your court now – here's where the fun begins!

Index

Symbols

A

B

C

D

E

F

G

I

K

L

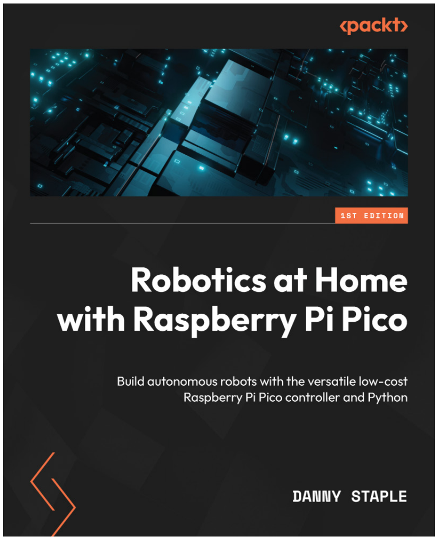
M

N

O

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Robotics at Home with Raspberry Pi Pico

Danny Staple

ISBN: 9781803246079

- Interface Raspberry Pi Pico with motors to move parts
- Design in 3D CAD with Free CAD
- Build a simple robot and extend it for more complex projects
- Interface Raspberry Pi Pico with sensors and Bluetooth BLE
- Visualize robot data with Matplotlib
- Gain an understanding of robotics algorithms on Pico for smart behavior

Embedded Systems Architecture - Second Edition

Daniele Lacamera

ISBN: 9781803239545

- Participate in the design and definition phase of an embedded product
- Get to grips with writing code for ARM Cortex-M microcontrollers
- Build an embedded development lab and optimize the workflow
- Secure embedded systems with TLS
- Demystify the architecture behind the communication interfaces
- Understand the design and development patterns for connected and distributed devices in the IoT
- Master multitasking parallel execution patterns and real-time operating systems
- Become familiar with Trusted Execution Environment (TEE)

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Embedded Linux Development Using Yocto Project*, we'd love to hear your thoughts! If you purchased the book from Amazon, please `click here to go straight to the Amazon review page` for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere? Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781804615065`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly