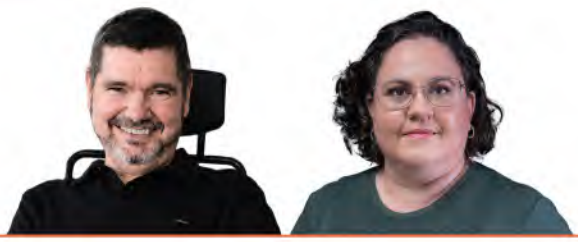
EXPERT INSIGHT

Embedded Linux Development Using Yocto Project

Build production-grade embedded Linux products with the Yocto Project 6.0 (Wrynose)

Fourth Edition



Otavio Salvador | Daiane Angolini

Foreword by Ross Burton, Yocto Project TSC member and Principal Engineer, Arm Ltd.

‹packt›

Embedded Linux Development Using Yocto Project

Fourth Edition

Build production-grade embedded Linux products with the Yocto Project 6.0 (Wrynose)

Otavio Salvador

Daiane Angolini

packt

Embedded Linux Development Using Yocto Project

Fourth Edition

Copyright © 2026 Packt Publishing
All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.
Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.
Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Portfolio Director: Rohit Rajkumar
Relationship Lead: Kaustubh Manglurkar
Project Manager: Sandip Tadge
Content Engineer: Anuradha Joglekar
Technical Editor: Tejas Mhasvekar
Copy Editor: Anuradha Joglekar
Indexer: Tejal Soni
Proofreader: Anuradha Joglekar
Production Designer: Vijay Kamble
Marketing Owner: Chayan Majumdar

First edition: July 2014
Second edition: November 2017
Third edition: April 2023
Fourth edition: September 2026
Production reference: 1180926

Published by Packt Publishing Ltd.
Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK.

ISBN 978-1-80669-507-2
www.packtpub.com

To Khem Raj, Vanessa Maegima, and Sergio Prado, whose technical review made this a better book, and to the Yocto Project and OpenEmbedded Project communities, whose work is the subject of every page. Our thanks also to the team at Packt for their support throughout the process.

– Otavio Salvador and Daiane Angolini

Foreword

I have been working on OpenEmbedded and the Yocto Project for longer than I like to dwell on these days, starting with Poky Linux at OpenedHand back in 2006. We were trying to solve the comparatively simple problem of production-quality embedded Linux: a core set of packages that were well tested, with reliable and reproducible builds, without every team needing to write their own build scripts. It turned out that having builds that just worked, while considering the practicalities of shipping commercial products, was something people wanted, and a few years later this evolved into the Yocto Project, released in 2011.

Fast-forward 20 years and there's still work to be done, as the world continues to change and being able to build an image easily just isn't enough anymore. You need to know exactly what is in it, where it all came from, what vulnerabilities have been reported, and how to implement updates after it has been released. And, importantly, you need to be able to demonstrate all of this when your Legal team, a customer, or even a state regulator asks.

That last one is a significant change from the world in which Poky Linux was started.

Security scans are no longer something you do once when you're about to ship, fixing the worst issues and hoping for the best. You need to know your full software supply chain throughout the product's lifetime, so you know that you're not shipping an ancient library that was bundled into another piece of software. Fail to do this, and your product could be compromised, inadvertently become part of a botnet, and you may only find out when the state regulators come knocking.

Open source licensing has also evolved, and shipping a single huge page containing all of the license texts isn't considered "best in class" anymore. There are licenses that have serious implications or "interesting" interactions when combined with other licenses. They need to be tracked in detail so that you understand what licenses are going into your image, while also watching all of the projects for licensing changes. License compliance is no longer an academic exercise, but a process with real legal ramifications.

The reason I encourage anyone making Linux-based products to use the Yocto Project is that these requirements are not afterthoughts; they have always been a key consideration of the project from day one. It has been tracking license texts, generating manifests, scanning for vulnerabilities, and reporting on changes between builds for many years. Now the regulatory environment has caught up with what has long been considered just best practices, and the Yocto Project is well suited to help users comply with the new rules.

Of course, the Yocto Project is still complex and can have what I've previously described as a "near-vertical" learning curve. Building systems with sufficient flexibility to work across the wide range of deployments that the Yocto Project supports just isn't simple, as removing complexity often removes options that may well be needed for your specific requirements.

This is where having authors like Otavio and Daiane is key. They're not writing about the Yocto Project from afar, rewriting the manual just enough to be different. They've been using the Yocto Project since its inception, contributing to it, maintaining BSPs, and working with commercial customers who have faced these issues and need to make things work. That experience shows through in the book: years of experience with real problems and actual people have molded the content, so it focuses on explaining the basics efficiently and knows what the reader's next question will be.

The Yocto Project documentation has a Quick Start guide that shows you how to build and boot an image. This book can help you take that image, the same image that everyone else built, and turn it into your image, your idea, your product.

Happy building!

Ross Burton

Yocto Project TSC member and Principal Engineer, Arm Ltd.

Contributors

About the authors

Otavio Salvador loves technology and started his free software activities in 1999. In 2002, he founded O.S. Systems, a company focused on embedded system development services and consultancy worldwide, creating and maintaining customized board support packages (BSPs) and helping companies with their product development challenges. This resulted in his joining the OpenEmbedded Project community in 2008, and he has contributed to the Yocto Project since its earliest releases, becoming the maintainer of `meta-freescale`, the community BSP layer for NXP hardware, in 2011. He is also active in the Zephyr Project. As Chief Executive Officer of O.S. Systems, he works across build systems, low-level and system programming, and real-time embedded development. More recently, he has adopted AI technologies deeply in his day-to-day engineering work. Beyond his technical work, he follows the financial markets closely, with a particular interest in options and cryptocurrencies. He is a co-author of all four editions of this book.

To my wife, Bruna Salvador, for her support and her patience, and for making room in our life for a book that took far more of it than I promised. And to the team at O.S. Systems, whose work with customers is where most of these explanations were first worked out.

Daiane Angolini has been working with embedded Linux since 2008. She is a senior software engineer at Foundries.io, now part of Qualcomm Technologies, Inc., where she works on platform engineering for the Foundries.io open-source projects, including the company's Yocto Project distribution and the BSPs it carries across *ARM* and *x86* platforms. She has contributed to `meta-freescale` since 2012 and takes part in the Linux kernel, U-Boot, OpenEmbedded Project, and Yocto Project communities. Before that, at NXP, she worked on internal development, ported custom applications across i.MX platforms, and supported customers on the Linux kernel, U-Boot, and user-space applications. She writes technical articles on *ARM* architectures and the Yocto Project, and she has co-authored every edition of this book.

To my husband, Thiago Justo, for his tremendous support while this book was written. And to my managers, David Griego and Nicolas Dechesne, and my employer, for the time and backing to work on this project.

About the reviewers

Khem Raj is the Embedded Linux and Open Source Leader at Qualcomm's IoT, Automotive, and Robotics business unit. He holds a bachelor's degree (Hons) in electronics and communications engineering. He has over 25 years of experience building operating systems, compilers, and scalable build systems at start-ups and Fortune 500 companies. A prolific open-source contributor, he is the lead maintainer of OpenEmbedded and sub-system maintainer for Yocto Project, looking after the toolchain and SDK recipes. He has been on the Yocto Project advisory board, TSC, and OpenEmbedded technical steering committee. He also maintains critical layers such as meta-riscv. A frequent conference speaker, he has reviewed *Mastering Embedded Linux Programming* and *Mastering Linux Device Driver Development* for Packt. He is an avid reader and a lifelong learner.

I would like to thank Packt for giving me the opportunity to review this book. Above all, I would like to thank my wife, Sweta, and our children, Himangi and Vihaan, for always being there for me and supporting me in all my endeavors.

Vanessa Maegima is an embedded systems engineer with over a decade of experience in Linux BSP development and customer support. She has worked at companies ranging from major SoC vendors, such as Qualcomm and NXP, to the startup-oriented environment of Foundries.io, gaining a broad perspective on embedded software development and real-world product use cases. As a technical documentation enthusiast, she enjoys turning complex engineering concepts into clear, practical, and useful guidance. She lives in southeastern Brazil with her loving husband and their playful dog.

Sergio Prado is an embedded systems engineer, consultant, and trainer with more than 30 years of experience in embedded software development. He is the founder and principal engineer at Embedded Labworks, where he helps companies develop embedded Linux systems and trains engineering teams worldwide. He is an active open source contributor and has contributed to projects including the Yocto Project, Buildroot, and the Linux kernel. Sergio is also a frequent speaker at international conferences, including the Yocto Project Summit and Embedded Linux Conference.

Table of Contents

Preface **xix**

Free benefits with your book ..... **xxvii**

Part 1: Getting Started **1**

Chapter 1: Meeting the Yocto Project **3**

What is the Yocto Project? **4**
Why the Yocto Project? **4**
How the Yocto Project produces embedded systems **5**
OpenEmbedded Project and the Yocto Project **6**
Yocto Project metadata layers **6**
Release cadence and the LTS cycle **8**
Yocto Project 6.0 release notes and migration guide **10**
Summary **10**

Chapter 2: Baking Our First Yocto Project Image **11**

Technical requirements **12**
Preparing the build host system **12**
Using Windows Subsystem for Linux (WSL 2) • 13
Preparing a Linux-based system • 14
Debian-based distributions • 15
Fedora • 16
openSUSE Leap • 17
AlmaLinux, Rocky Linux, and CentOS Stream • 17
Setting up with bitbake-setup **18**
Installing the bitbake-setup tool • 18
Initializing your first project • 19
Understanding the bitbake-setup directory structure • 20
Activating the build environment • 21
Reviewing the configuration files • 21

Building a target image **22**
Running images in QEMU **25**
Summary **27**

Chapter 3: Using VS Code with the BitBake Extension 29

Technical requirements **30**
Opening the generated workspace file **30**
Installing the Yocto Project BitBake extension **31**
Working with syntax highlighting and context-based suggestions **32**
Touring the extension's other features **33**
Building core-image-full-cmdline from VS Code **34**
Running runqemu from VS Code **35**
Using the integrated terminal • 35
Defining a workspace task • 35
Summary **37**

Chapter 4: Meeting the BitBake Tool 39

Technical requirements **39**
BitBake's role in the build workflow **40**
Metadata collections and layer structure **40**
Inspecting layers with bitbake-layers **42**
Configuration, classes, and recipes **43**
Summary **45**

Part 2: Core Concepts 47

Chapter 5: Grasping the BitBake Tool 49

Technical requirements **50**
Preparing the environment **50**
Indexing metadata **51**
Initializing BitBake **51**
Parsing metadata **52**
Dependencies **53**
Inspecting the dependency graph • 55
Fetching the source code **56**

Remote file downloads • 57
Git repositories • 59
Optimizing source code downloads 60
Disabling network access • 62
Understanding BitBake tasks 63
Common BitBake tasks explained • 64
Hash equivalence • 65
Summary 65

Chapter 6: Detailing the Temporary Build Directory 67

Technical requirements 68
Detailing the build directory 68
Constructing the build directory 69
Exploring the temporary build directory 70
Understanding the work directory 71
Understanding the sysroot directories 75
Summary 76

Chapter 7: Assimilating Packaging Support 77

Technical requirements 77
Supported package formats 78
List of supported package formats • 78
Choosing a package format • 78
Running code during package installation 79
Understanding the Shared State Cache 81
Explaining package versioning 83
Specifying runtime package dependencies 83
Using packages to generate a root filesystem image 84
Package feeds 85
Using package feeds • 86
Summary 88

Chapter 8: Understanding BitBake Metadata 89

Technical requirements 90
Variable assignment and expansion operators 90

Basic variable assignment • 91
Variable expansion • 92
Deferred evaluation • 92
Immediate variable expansion • 92
Assigning a value with ?= • 92
Assigning a default with ??= • 93
List and string appending, prepending, and removal 94
List appending and prepending • 94
String appending and prepending • 94
Using :append and :prepend • 94
Comparing :append with .= • 95
Removing list items • 95
Conditional metadata and the override mechanism 95
Conditional metadata sets • 95
Conditional appending • 96
Working through a combined example • 97
File inclusion with include and require 97
Python and shell metadata functions 98
Python variable expansion • 98
Defining executable metadata • 98
Defining python functions in the global namespace • 99
Anonymous Python functions • 99
Combining Python and shell • 100
Class files and metadata inheritance 100
Understanding class files in BitBake • 100
Global classes • 101
Recipe-specific classes • 101
Tuning workspace cleanup classes • 101
Summary 101

Part 3: Advanced Concepts 103

Chapter 9: Developing with the Yocto Project 105

Technical requirements 106
What is a Software Development Kit? 106

Generating an image for on-device development **107**
Understanding the types of cross-development SDKs **107**
Using the Standard SDK **108**
Installing the Standard SDK • 108
Building applications with the Standard SDK • 109
Building the Linux kernel with the Standard SDK • 110
Using the Extensible SDK **110**
Generating the Extensible SDK • 111
Installing the Extensible SDK • 111
Structure of the Extensible SDK • 113
Building an image using devtool • 113
Running an image on QEMU • 114
Creating a recipe from an external git repository • 115
Building a recipe using devtool • 117
Modifying an existing recipe with devtool • 118
Working with the devtool source git tree • 119
Extending the eSDK • 120
Using VS Code with the Extensible SDK • 121
Summary **122**

Chapter 10: Debugging with the Yocto Project **123**

Technical requirements **124**
Differentiating metadata and application debugging **124**
Tracking image, package, and SDK content **124**
Debugging packaging **126**
Inspecting packages • 126
Debugging with logging in the Yocto Project **128**
Debugging metadata variables **129**
Using a Development Shell **130**
Using the GNU Debugger for debugging **132**
Going deeper with profiling and tracing **133**
Walking through a missing file **133**
Summary **136**

Chapter 11: Exploring External Layers 137

Technical requirements ... 137
Powering flexibility with layers ... 138
Detailing a layer's source code ... 140
Adding meta layers ... 142
The OpenEmbedded Layer Index ... 144
Summary ... 145

Chapter 12: Creating Custom Layers 147

Technical requirements ... 147
Creating the layer ... 148
Adding metadata to the layer ... 151
Creating an image ... 152
Adding a package recipe ... 156
Automatically creating a base package recipe using devtool • 158
Adding support for a new machine definition ... 160
Wrapping an image for your machine ... 163
Using a custom distribution ... 167
Adjusting default DISTRO_FEATURES and MACHINE_FEATURES ... 171
Interaction between MACHINE_FEATURES and DISTRO_FEATURES ... 172
Understanding the scope of variables in BitBake ... 173
Global variables • 173
Recipe-specific variables • 174
Summary ... 174

Chapter 13: Customizing Existing Recipes 175

Technical requirements ... 176
Understanding common use cases ... 176
Extending a task ... 178
Adding extra build options to recipes ... 179
Applying a patch ... 181
Adding extra files to existing packages ... 181
Understanding file search paths ... 183
Changing recipe feature configuration ... 185

Configuration fragments for Kconfig-based projects **187**
Summary **190**

Chapter 14: Managing BitBake Configuration 191

Technical requirements **192**
Exploring available configurations **192**
Managing build configuration with bitbake-config-build **193**
Listing available fragments • 193
Disabling fragments • 194
Enabling fragments • 197
Inspecting a fragment • 199
Configuration precedence and conflicts • 200
Creating custom BitBake configuration fragments **201**
Defining a standard fragment • 201
Enabling your fragment • 203
Organizing fragments with subdirectories • 203
Defining built-in fragments • 204
Managing layers with bitbake-setup **205**
Keeping your setup current **207**
Reproducible team workflows **208**
Creating custom BitBake configuration templates • 209
Sharing the download cache • 212
Keeping several BitBake setups under one top directory • 213
Managing private BitBake registries • 213
Source overrides for development branches • 214
Synchronizing configurations across teams • 215
Configuration, sources, and reproducibility • 216
Summary **216**

Chapter 15: License Compliance and Software Bill of Materials 219

Technical requirements **220**
Understanding copyleft **220**
The role of the Yocto Project **221**
Achieving copyleft compliance with the Yocto Project **221**
Managing software licensing with the Yocto Project **222**

Understanding commercial licenses 225
Configuring your project for compliance 226
Understanding license auditing • 226
Providing the source code • 227
Providing compilation scripts and source code modifications • 230
Considerations for copyleft compliance • 230
Output file formats • 230
Providing license text inside the image • 230
Generating a Software Bill of Materials • 231
Declaring a proprietary license for your own recipes • 232
SBOM-based vulnerability scanning • 235
Summary 237

Part 4: Bridging to Real-World Application 239

Chapter 16: Building and Booting a Custom Embedded Linux System 241

Technical requirements..... 242
Discovering the right BSP layer 242
Reviewing aspects that impact hardware use..... 243
Key considerations for board selection • 243
Overview of widely used BSP layers • 244
Using physical hardware • 244
BeagleBone Black 247
Baking for BeagleBone Black • 247
Setting up the source structure • 247
Building the image • 248
Writing the image to an SD card • 249
Booting the board • 249
Raspberry Pi 4 or Raspberry Pi 5..... 250
Baking for Raspberry Pi 4 or Raspberry Pi 5 • 250
Setting up the source structure • 250
Building the image • 252
Writing the image to an SD card • 252
Booting the board • 254

VisionFive 2 ..... **254**
Baking for VisionFive 2 • 254
Setting up the source structure • 254
Building the image • 256
Writing the image to an SD card • 256
Booting the board • 256
Arduino UNO Q **257**
Baking for the Arduino UNO Q • 257
Setting up the source structure • 258
Enabling the qcomflash image type • 261
Building the image • 261
Flashing with qdl • 262
Booting the board • 263
Taking the next steps **263**
Contributing to the community • 264
Embracing feedback • 264
Summary **264**

Chapter 17: Speeding Up Product Development Through Emulation (QEMU) 267

Technical requirements **268**
What is QEMU? **268**
What are the benefits of using QEMU over hardware? **269**
When is choosing real hardware preferable? **269**
Using runqemu capabilities **270**
Using runqemu to test graphical applications • 271
Using runqemu to validate memory constraints • 273
Using runqemu to select a Linux kernel image and root filesystem • 275
Using runqemu to pass custom boot arguments • 276
Using runqemu with a serial console • 276
Using runqemu to validate audio stacks • 277
Using runqemu to help with image regression tests • 277
Using runqemu to run package tests with ptest • 280
Summary **283**

Chapter 18: Best Practices 285

Guidelines to follow for the Yocto Project 286

Manage layers • 286

Tidy up bitbake-setup configuration files • 287

Avoid creating too many layers • 289

Prepare the product metadata for new Yocto Project releases • 290

Create your custom distribution • 290

Avoid reusing existing images for your product • 291

Standard SDK is commonly undervalued • 291

Avoid too many patches for the Linux kernel and bootloader modifications • 292

Avoid using AUTOREV as SRCREV • 293

Maintain the Shared State Cache • 293

Enforce the guidelines with version control and continuous integration • 295

Create a software bill of materials • 295

Guidelines to follow for general projects 296

Continuously monitor the project license constraints • 296

Security can harm your project • 297

Don't underestimate maintenance costs • 298

Tackle project risk points and constraints as soon as possible • 299

Summary 300

Chapter 19: Unlock Your Exclusive Benefits 301

Unlock this Book's Free Benefits in 3 Easy Steps 302

Other Books You May Enjoy 306

Index 309

Preface

An embedded Linux product is never only a Linux kernel and a root filesystem. It is a particular set of packages, built with a particular toolchain, for a particular piece of hardware, under license terms you have to be able to account for. Assembling that by hand works once. It does not survive a product line, and it does not survive the day a security fix lands and the whole image has to be rebuilt exactly as it was built eighteen months earlier.

The Yocto Project exists to make that repeatable. It is not a Linux distribution and it does not ship images you install. It is a set of tools, metadata, and community practice that lets you describe the system you want and then build it the same way every time for the hardware you ship on. Among the open-source build systems available for embedded work, a few properties make it the one most product teams settle on:

- Everything is built from source, so the toolchain, the compiler flags, the package selection, and the license terms of what you ship are all yours to decide and to audit
- The metadata is layered, which keeps hardware support, distribution policy, and your own application recipes in separate repositories that can be versioned and upgraded independently
- Long-term support releases are maintained for four years, which is close enough to the service life of a typical embedded product to plan around
- The build output is reproducible, so a colleague, a build server, and a customer support engineer investigating a field failure can all reconstruct the same image

This book follows the path an engineer actually walks when adopting the Yocto Project. It opens with what the project is, who produces it, and how its releases are organized, then prepares a build host and produces a first bootable image with `bitbake-setup`. From there, it turns to BitBake itself: how it locates metadata, parses it, works out the dependency graph, fetches source code, and schedules the tasks that turn that source into a package.

The middle of the book covers the artifacts a build leaves behind and the metadata you write to shape them. That means the temporary build directory with its per-recipe work and sysroot trees, the packaging support that turns build output into installable packages, root filesystem images, and package feeds, and the metadata syntax itself: assignment and expansion operators, appends and removals, the override mechanism, Python and shell functions, and class inheritance. Those chapters are followed by the tooling you use on top of a working build,

from software development kits to logging, variable inspection, the Development Shell, GDB, and profiling.

The remaining chapters move from building to shipping. You will write your own layers, machine definitions, and distribution configuration, customize recipes that other people maintain, manage build configuration with `bitbake-config-build` and `bitbake-setup`, and work through what license compliance requires of a product team. The book closes by bringing up a custom image on real hardware, using QEMU to keep development moving when the hardware is scarce or not ready, and collecting the working habits that keep a Yocto Project-based product maintainable over years.

This fourth edition is written against Yocto Project 6.0 (Wrynose), a long-term support release, and it is a substantial revision rather than a refresh. The most visible change is the setup workflow: `bitbake-setup` now creates and manages the build workspace, and `bitbake-config-build` manages the configuration inside it through BitBake Configuration Templates and Fragments, replacing the manual clone-and-source routine earlier editions taught. That workflow is important enough to have earned a chapter of its own, *Managing BitBake Configuration*, which is new to this edition.

Two other changes are worth calling out. The chapter on graphical build tooling now covers the Yocto Project BitBake extension for Visual Studio Code, which is where that work has moved. And license compliance has grown to match what regulators and customers now ask for, adding Software Bill of Materials generation, SBOM-based vulnerability scanning, and guidance on declaring the terms of your own proprietary recipes to the existing copyleft material. That chapter is now called *License Compliance and Software Bill of Materials*, which is a better description of what it has become. Board coverage was extended to the Raspberry Pi 5, the VisionFive 2, and the Arduino UNO Q, and the examples, command output, and documentation links throughout the book were regenerated against this release rather than carried forward.

We wrote this book from the position we work in from day to day, building and maintaining Yocto Project-based products with customers whose deadlines are real and whose hardware rarely behaves the way the datasheet promises. The explanations here are the ones we found ourselves repeating, and the shortcuts are the ones that survived contact with production.

Who this book is for

This book is for embedded Linux developers, build and platform engineers, and technically minded enthusiasts who are responsible for the operating system image of a product rather than only the application that runs on it. It suits engineers evaluating the Yocto Project, comparing it against alternatives, or already committed to it and wanting to get productive

quickly without paying the usual learning-curve tax. Professionals looking for better working methods for Linux development will find plenty here as well.

Some background is assumed, because the book spends its pages on the Yocto Project rather than on the ground beneath it. You should be comfortable in a GNU/Linux environment and on the command line, able to read a shell script, familiar with Git, and clear on the general shape of an embedded system and its development cycle: compilation, debugging, deployment, and installation. Basic familiarity with the Linux kernel and how it is configured and built will help. No prior experience with the Yocto Project, the OpenEmbedded Project, or BitBake is assumed, and engineers already shipping products on an older release will find the move to the current tooling covered directly.

What this book covers

Chapter 1, *Meeting the Yocto Project*, introduces what the Yocto Project is and what it is not, how it produces embedded systems, its relationship with the OpenEmbedded Project, the metadata layer model, and the release cadence and long-term support cycle that a product plan has to be built around.

Chapter 2, *Baking Our First Yocto Project Image*, prepares a build host, sets up a workspace with `bitbake-setup`, builds a target image, and runs the result under QEMU, giving you a working build you can reproduce from a clean machine.

Chapter 3, *Using VS Code with the BitBake Extension*, sets up the Yocto Project BitBake extension for Visual Studio Code and uses it for syntax highlighting, context-based suggestions, building an image, and launching `runqemu` without leaving the editor.

Chapter 4, *Meeting the BitBake Tool*, explains BitBake's role in the build workflow, how metadata collections and layers are structured, how to inspect them with `bitbake-layers`, and how configuration files, classes, and recipes relate to one another.

Chapter 5, *Grasping the BitBake Tool*, follows a build from the inside: indexing and parsing metadata, resolving dependencies, fetching source code, keeping downloads efficient, and understanding the tasks BitBake schedules.

Chapter 6, *Detailing the Temporary Build Directory*, walks through the build directory and how it is constructed, then examines the temporary build directory in detail, including the per-recipe work directories and the sysroots that make cross-compilation work.

Chapter 7, *Assimilating Packaging Support*, covers the supported package formats, code that runs during package installation, the Shared State Cache, package versioning, runtime dependencies, root filesystem image generation, and package feeds.

Chapter 8, *Understanding BitBake Metadata*, is the metadata reference chapter: assignment and expansion operators, list and string appending, prepending, and removal, the override mechanism, `include` and `require`, Python and shell functions, and class files and inheritance.

Chapter 9, *Developing with the Yocto Project*, explains what a software development kit is in this context, how to generate an image suitable for on-device development, the types of cross-development SDK available, and how to work with both the standard and the extensible SDK.

Chapter 10, *Debugging with the Yocto Project*, separates metadata problems from application problems and then addresses each: tracking image, package, and SDK contents; debugging packaging; logging; inspecting metadata variables; using the Development Shell; running GDB; and going further with profiling and tracing.

Chapter 11, *Exploring External Layers*, shows how layers deliver flexibility, how to read a layer's source code before you trust it, how to add meta layers to a build, and how to find them through the OpenEmbedded Layer Index.

Chapter 12, *Creating Custom Layers*, builds a layer of your own and fills it: metadata, a custom image, a package recipe, a new machine definition, a custom distribution, adjusted `DISTRO_FEATURES` and `MACHINE_FEATURES`, and the variable scoping rules that decide which of them wins.

Chapter 13, *Customizing Existing Recipes*, covers the changes you make to metadata you do not own: extending a task, adding options to Autotools-based recipes, applying a patch, adding extra files to an existing package, understanding file search paths, and changing feature configuration with Kconfig configuration fragments.

Chapter 14, *Managing BitBake Configuration*, explores the configurations available to a build, manages them with `bitbake-config-build`, creates custom BitBake Configuration Fragments, manages layers with `bitbake-setup`, and turns the result into a workflow a whole team can reproduce.

Chapter 15, *License Compliance and Software Bill of Materials*, explains copyleft and the role the Yocto Project plays in meeting it, then covers license auditing, providing source code and compilation scripts, shipping license text inside the image, commercial licenses, declaring the terms of your own proprietary recipes, generating a Software Bill of Materials, and using that SBOM to drive vulnerability scanning, so compliance becomes a build output rather than an audit exercise.

Chapter 16, *Building and Booting a Custom Embedded Linux System*, moves the work onto real hardware: finding the right BSP layer, reviewing the aspects that affect how the hardware is used, and bringing up an image on the BeagleBone Black, Raspberry Pi 4 and 5, VisionFive 2, and Arduino UNO Q.

Chapter 17, *Speeding Up Product Development Through Emulation (QEMU)*, explains what QEMU offers an embedded team, when emulation beats waiting for hardware and when it does not, and how to use the capabilities of `runqemu`.

Chapter 18, *Best Practices*, collects the guidelines that keep a Yocto Project-based product maintainable, along with the wider project habits that make the difference over a multi-year lifecycle.

To get the most out of this book

You will get the most from this book if you are comfortable working on a Linux command line and can read a shell script and a Makefile without reaching for documentation. Familiarity with how C and C++ projects are configured and built, and with the general structure of an embedded system, will make the metadata chapters easier. No prior experience with the Yocto Project, the OpenEmbedded Project, or BitBake is needed. Every workflow is introduced from the beginning.

To follow the examples, you need:

- A 64-bit Linux host running a distribution validated for the Yocto Project 6.0 (Wrynose) release. Ubuntu 22.04 LTS or later is a common choice, and AlmaLinux, CentOS Stream, Debian, Fedora, openSUSE Leap, and Rocky Linux are equally supported. Windows users can work through Windows Subsystem for Linux 2.
- The host packages listed under *Required Packages for the Build Host* at `https://docs.yoctoproject.org/wrynose/ref-manual/system-requirements.html`, plus Python 3.9 or newer
- At least 4 CPU cores (8 or more is much better), 32 GB of RAM, and 140 GB of free disk space. Build time scales strongly with core count, so use the most capable machine you have for the first build.
- A reliable internet connection. The first build downloads several gigabytes of source archives.
- Hardware virtualization enabled in the BIOS or UEFI setup, so QEMU can use KVM acceleration.

Most of the book can be followed entirely in QEMU. The hardware chapter uses the BeagleBone Black, Raspberry Pi 4 or 5, VisionFive 2, and Arduino UNO Q as worked examples, but you only need a board if you want to boot your image on one, and the workflow transfers to any board with a maintained BSP layer.

Type the commands yourself rather than pasting them, at least the first time through a chapter. Much of what the Yocto Project teaches is in the output, and reading the output as it scrolls is how the shape of the build becomes familiar.

This edition does not ship a separate code bundle. The recipes, configuration fragments, and commands used in the examples are given in the chapters.

Download the color images

Your purchase includes a color, DRM-free PDF copy of this book, ideal for viewing color images, screenshots, and diagrams. Refer to *Free benefits with your book section* at the end of the *Preface* to unlock your PDF copy.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and BitBake variable names. For example: "Set `MACHINE` in `build/conf/local.conf` before starting the build."

A block of code is set as follows:

```
SUMMARY = "Hello World application"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;\
md5=0835ade698e0bcf8506ecda2f7b4f302"
```

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

```
$ bitbake core-image-minimal
```

Bold: Indicates a new term, an important word, or words that you see on the screen. For instance, words in menus or dialog boxes appear in the text like this. For example: "Select **Run Task** from the Visual Studio Code command palette."

Note

Warnings or important notes appear like this.

Tip

Tips and tricks 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 or have any general feedback, please email us at `customercare@packt.com` and mention the book's 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 reported this to us. Please visit `http://www.packt.com/submit-errata`, click **Submit 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 `http://authors.packt.com/`.

Join our community on Discord

Join our community's Discord space for discussions with the authors and other readers:
`https://packt.link/embeddedsystems`



Subscribe to Inside Embedded!

We are excited to introduce **Inside Embedded**, our brand-new newsletter dedicated to everything embedded systems development. Whether you're working with firmware, embedded Linux, microcontrollers, IoT, or connected devices, every issue brings you curated insights, practical workflows, and expert analysis to help you spend less time searching and more time building smarter, more reliable systems. Sign up now and get exciting benefits.



https://packt.link/adqzC

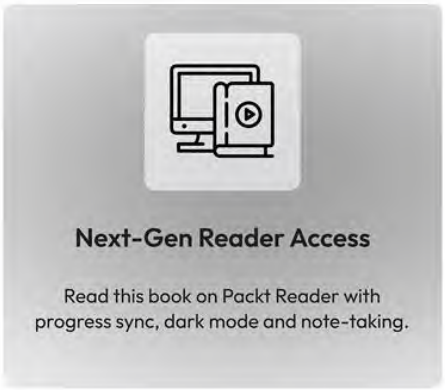
Free benefits with your book

This book comes with free benefits to support your learning. Activate them now for instant access (see the "*How to Unlock*" section for instructions).

Here's a quick overview of what you can instantly unlock with your purchase:







How to Unlock

Scan the QR code (or go to `packtpub.com/unlock`). Search for this book by name, confirm the edition, and then follow the steps on the page.





Note: Keep your invoice handy. Purchases made directly from Packt don't require one

Share your thoughts

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



https://packt.link/r/1806695073

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

Part 1

Getting Started

This first part introduces the Yocto Project and takes you from nothing to a working, repeatable build. You will learn what the Yocto Project is and, just as usefully, what it is not, how it relates to the OpenEmbedded Project, how metadata layers organize a system, and how the release cadence and long-term support cycle shape what a product team can plan around.

From there the work becomes practical. You will prepare a Linux build host, create a workspace with `bitbake-setup`, review the configuration it generates, bake a target image, and run it under QEMU. The part then sets up the Yocto Project BitBake extension for Visual Studio Code, so the metadata you spend the rest of the book reading and writing has syntax highlighting, suggestions, and build commands behind it. It closes by introducing BitBake itself: its role in the build workflow, how layers are structured, how to inspect them with `bitbake-layers`, and the three kinds of metadata you will meet everywhere afterward.

By the end of this part you will have a build any engineer on your team can reproduce from a clean machine, which is the foundation everything that follows depends on.

This part of the book includes the following chapters:

- *Chapter 1*, *Meeting the Yocto Project*
- *Chapter 2*, *Baking Our First Yocto Project Image*
- *Chapter 3*, *Using VS Code with the BitBake Extension*
- *Chapter 4*, *Meeting the BitBake Tool*

1

Meeting the Yocto Project

This book walks intermediate-to-expert embedded engineers through building maintainable Linux platforms with the **Yocto Project 6.0** (**Wrynose**) LTS release. By the end of this book, you will be able to stand up a reproducible build environment, ship images for real hardware, customize the metadata your product needs, and meet the license obligations that follow when open-source software ships in a commercial device.

This chapter lays the foundation for everything that follows. You will learn where the Yocto Project 6.0 (Wrynose) release fits in the project's history and how the OpenEmbedded Project contributed the metadata layer model and BitBake task-execution engine that the Yocto Project builds on today. You will also see how the two-year LTS release cycle shapes the stability guarantees you can rely on for a product program. The goal is to leave you with a working vocabulary and a clear release-baseline context before you write a single line of metadata. That context matters in practice: an embedded engineer joining a Yocto Project team without it will mis-anchor commands against the wrong release branch, spend onboarding cycles untangling terminology differences between OpenEmbedded Core and the Yocto Project proper, and generate review noise that has nothing to do with the actual technical work. Getting this right once at the start eliminates that overhead across every chapter that follows.

We will cover the following main topics:

- What is the Yocto Project?
- Why the Yocto Project?
- How the Yocto Project produces embedded systems
- OpenEmbedded Project and the Yocto Project
- Yocto Project metadata layers
- Release cadence and the LTS cycle
- Yocto Project 6.0 release notes and migration guide

> **Note**
>
> **Download the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

What is the Yocto Project?

The Yocto Project is a Linux Foundation collaborative initiative that delivers a toolkit for building custom embedded Linux platforms independent of any single silicon or software vendor. Its foundation rests on **BitBake**, a task-execution engine that runs every build rule, together with a curated metadata collection of recipes, classes, policies, and configuration metadata plus reference distributions and SDK workflows. The reference flow centers on the **Poky** distribution for Linux-based platforms, while real-time profiles built around **Zephyr** are available through community layers such as `meta-zephyr` and fall outside the scope of this book.

The project delivers:

- BitBake: a task-execution engine capable of orchestrating thousands of builds with explicit dependency tracking
- Metadata collection: a curated set of recipes, classes, policies, and configuration metadata (distribution and machine definitions) that describe how to fetch, configure, compile, package, and deploy software
- Reference distributions and SDK workflows: the Poky reference distribution, extensible SDKs, and reproducible build flows that span multiple CPU architectures and both the `glibc` and `musl` C libraries

The project stays vendor-neutral while benefiting from contributions across semiconductor companies, board vendors, independent software providers, and open-source maintainers. That mix keeps the tooling relevant for experimentation and production programs.

Why the Yocto Project?

Building a Linux platform for a product means solving the same problems every team eventually hits: cross-compiling hundreds of packages for a target that differs from the build host, pinning exact source and toolchain versions so a build reproduces months or years later, and tailoring the image to the hardware without carrying a general-purpose distribution's footprint. Assembling those pieces by hand with makefiles and shell scripts works for a

prototype, but it rarely survives contact with a shipping schedule, a second board revision, or an audit of what actually went into the firmware.

The Yocto Project addresses these problems directly. Its recipes capture how each component is fetched, configured, compiled, and packaged, so a build is described once and repeated deterministically across machines and releases. The layer model keeps board-specific and product-specific content separate from the shared base, which lets you retarget hardware or update the underlying distribution without rewriting your customizations. Because every artifact traces back to a versioned recipe, the same metadata that produces an image also answers the questions a product program must eventually face: what source shipped, under which license, and how to rebuild it.

That power comes with a cost, so it helps to know when the Yocto Project is the right tool. No single approach fits every device. A binary-only distribution such as Debian or Ubuntu gets you running quickly and brings a large package archive, which suits prototypes and hardware where a general-purpose image's size and update model are acceptable. Buildroot also builds a custom image from source and is quicker to learn, though it targets simpler, single-configuration products rather than a family of devices that share metadata. The Yocto Project earns its steeper learning curve when you need long-term reproducibility, fine-grained control over image contents and licensing, and one codebase that spans several boards and product variants. For a short-lived prototype or a device you will not customize beyond a vendor's supported image, that investment may not pay off, and a lighter option is the pragmatic choice.

How the Yocto Project produces embedded systems

Viewed as a build appliance, the Yocto Project consumes a specification written in metadata and emits the operating-system deliverables required by a product: bootloaders, Linux kernel images, root filesystems, containers, and auxiliary artifacts such as SDKs or update bundles. Key characteristics include:

- Source-first orientation: recipes define how to fetch source code from locations such as `https://kernel.org`, `https://github.com`, `https://gitlab.com`, or private mirrors
- Reuse through the **Shared State Cache**: previously built tasks are cached, allowing incremental builds to reuse artifacts whenever the inputs match
- Host independence: toolchains, build utilities, and configuration options are provisioned within the build tree, reducing reliance on host-distribution packages and improving determinism

The resulting workflow lets you define *what* the platform must contain while BitBake computes *how* to build and generate the required artifacts in the right order with the right compiler options.

OpenEmbedded Project and the Yocto Project

The **OpenEmbedded Project** predates the Yocto Project and grew out of the OpenZaurus community in 2003, splitting BitBake out as a standalone task-execution engine in late 2004. Over time, the metadata collection and supported hardware base expanded, but coordinated releases remained difficult. The Linux Foundation announced the Yocto Project in 2010 and launched it in March 2011 with 22 founding organizations, aligning hardware makers, board vendors, operating-system builders, and tooling specialists around a common build system and testing discipline.

Today the partnership works as follows:

- The OpenEmbedded Project maintains OpenEmbedded Core, which contains the fundamental recipes, classes, and policies required by every distribution
- The Yocto Project curates additional tooling, documentation, automated QA, visibility, and governance needed for production programs, one example being Poky, the Yocto Project reference distribution defined in `meta-poky`
- Contributors move seamlessly between both efforts even though BitBake and OpenEmbedded Core remain under the OpenEmbedded Project governance

Note

The project takes its name from *yocto*, the SI prefix for a factor of 10^{-24} and, when the Yocto Project was established in 2010, the smallest in the metric system. The name reflects the small footprint that embedded systems demand. Release codenames follow a separate convention: since version 3.1 (Dunfell), each one names a mountain or pass in Cumbria, England, a pattern that continues with this edition's Wrynose baseline.

Yocto Project metadata layers

Metadata is a mix of BitBake syntax, Python snippets, and shell script fragments stored as recipes (`.bb`), include files (`.inc`), append files (`.bbappend`), classes (`.bbclass`), and configuration files (`.conf`). The Yocto Project ecosystem organizes this metadata into a small set of key layers.

The diagram below highlights how Yocto Project layers build on top of the OpenEmbedded Project components that BitBake orchestrates. The box labeled OpenEmbedded Core is the meta layer, and meta-yocto is the repository that carries the two Yocto Project reference layers drawn inside it.

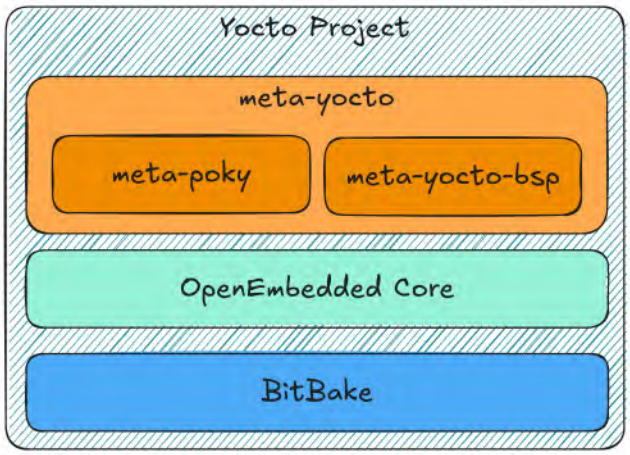


Figure 1.1: Yocto Project metadata layers

The distribution-specific layers bring board enablement content packaged as **Board Support Package (BSP)** material plus validation hooks tied to **Quality Assurance (QA)** workflows.

- `meta`: The layer that makes up OpenEmbedded Core, providing foundational policies, classes, and widely shared recipes.
- `meta-yocto`: The repository that groups the two reference layers below.
- `meta-poky`: Yocto Project distribution policy, branding, and metadata tracking used when you build with the Poky reference distribution as-is
- `meta-yocto-bsp`: reference BSP definitions used during Yocto Project development and QA

Only BitBake and `meta` are required to build an image. The `meta-yocto` layers are reference content. Production builds usually define their own distribution layer in place of `meta-poky` and use a vendor or community BSP layer in place of `meta-yocto-bsp`.

Chapter 12 describes how to extend these layers or add custom ones. The `bitbake-setup` tool manages each layer as a separate repository under the `layers/` directory of your BitBake Setup, and BitBake itself is fetched independently.

Release cadence and the LTS cycle

The Yocto Project publishes two releases per year (April and October) and designates one release every two years as **Long-Term Support (LTS)** when additional maintenance resources are allocated. Stable releases receive seven months of updates, while each LTS release receives at least four years of security and bug-fix coverage before optionally moving to community maintenance and, eventually, **end-of-life (EOL)** status.

The timeline below illustrates the overlapping cadence between stable and LTS lines, highlighting how production teams can stay on an LTS baseline while tracking new technology in parallel.

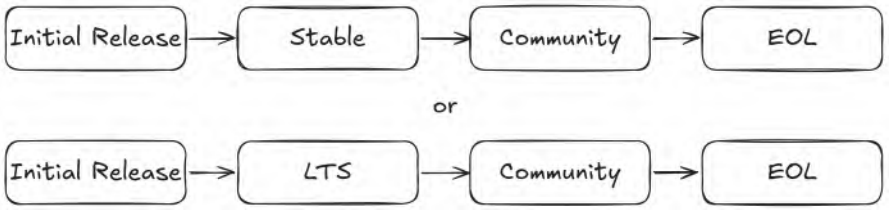


Figure 1.2: Stable and LTS release overlap

Table 1.1 lists recent releases. Dates and support levels match the Yocto Project releases page at `https://www.yoctoproject.org/development/releases/`:

Codename	Version	Release	Support window
Wrynose	6.0	May 2026	LTS (through Apr 2030)
Whinlatter	5.3	Dec 2025	EOL
Walnascar	5.2	May 2025	EOL
Styhead	5.1	Oct 2024	EOL
Scarthgap	5.0	Apr 2024	LTS (through Apr 2028)
Nanbield	4.3	Nov 2023	EOL
Mickledore	4.2	May 2023	EOL
Langdale	4.1	Oct 2022	EOL
Kirkstone	4.0	May 2022	EOL (LTS ended May 2026)

Codename	Version	Release	Support window
Honister	3.4	Oct 2021	EOL
Hardknott	3.3	Apr 2021	EOL
Gatesgarth	3.2	Oct 2020	EOL
Dunfell	3.1	Apr 2020	EOL
Zeus	3.0	Oct 2019	EOL
Warrior	2.7	Apr 2019	EOL
Thud	2.6	Nov 2018	EOL
Sumo	2.5	Apr 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
Denzil	1.2	Apr 2012	EOL
Edison	1.1	Oct 2011	EOL
Bernard	1.0	Apr 2011	EOL

Codename	Version	Release	Support window
Laverne	0.9	Oct 2010	EOL

Table 1.1: Selected Yocto Project releases

The project maintains versioned documentation alongside each release. When a new version ships, the documentation is published at its versioned path (for example, `https://docs.yoctoproject.org/wrynose/` for Yocto Project 6.0). As bug fixes and point releases appear (6.0.1, 6.0.2, and so on), the documentation is updated in place under the same path.

Note

When you visit the documentation for an older release, a warning banner appears indicating that you are not viewing the latest version. The banner does not redirect you automatically. Always verify that the version path in the URL matches your working release, and check `https://docs.yoctoproject.org` to confirm which release is current.

Yocto Project 6.0 release notes and migration guide

Readers coming from an earlier release should consult the *Release 6.0 (Wrynose)* page at `https://docs.yoctoproject.org/migration-guides/release-6.0.html` before porting an existing project to this baseline. The page hosts two companion documents. The release notes summarize new capabilities, recipe version bumps, security patches, license updates, and known issues. The migration guide covers the breaking changes that affect upgrading projects.

Summary

You now have an updated picture of how the Yocto Project, the OpenEmbedded Project, Poky, and BitBake relate, how metadata layers organize policies and recipes, and how the release cadence leads to the Yocto Project 6.0 baseline that anchors this edition.

The next chapter walks through the build workflow: obtaining the source tree, configuring local policies, and building the first **QEMU** image.

2

Baking Our First Yocto Project Image

This chapter covers how to set up a build host and produce a bootable embedded Linux image using the Yocto Project 6.0 (Wrynose) baseline. You will size the host machine (CPU cores, RAM, and disk), install the required packages, initialize the build environment with `bitbake-setup`, adjust the configuration files that control the target machine and distribution, and boot the finished image inside QEMU. By the end of this chapter, you will have a working, repeatable build that any engineer on your team can reproduce from a clean checkout.

That repeatability is not a nice-to-have. A build that succeeds on one engineer's laptop but breaks on another's (or on the build server) wastes hours tracing differences in host libraries, tool versions such as Python, or locale settings. Host-distribution drift is a leading cause of those failures, and it compounds as the project grows: the closer a team gets to a delivery milestone, the more expensive a mystery build failure becomes. Walking through the setup once with `bitbake-setup` and a validated host distribution eliminates that class of problem before the first line of custom code is written.

We will cover the following main topics:

- Preparing the build host system
- Setting up with bitbake-setup
- Building a target image
- Running images in QEMU

Technical requirements

To follow the examples in this chapter you need a 64-bit Linux host running a distribution that the Yocto Project 6.0 (Wrynose) release team has validated. Ubuntu 22.04 LTS or later is a common choice, but AlmaLinux, CentOS Stream, Debian, Fedora, openSUSE Leap, and Rocky Linux are equally supported. The *Preparing a Linux-based system* section lists every validated version.

Beyond the operating system, the build requires:

- The host packages documented in *Required Packages for the Build Host* at `https://docs.yoctoproject.org/wrynose/ref-manual/system-requirements.html#required-packages-for-the-build-host`
- At least 100 GB of free disk space on the build partition (140 GB or more is recommended once the Shared State Cache grows)
- A reliable internet connection, because the first build fetches several gigabytes of source archives

No prior Yocto Project experience is needed. This chapter walks through every step from a bare host to a bootable QEMU image, building on the components and release baseline introduced in *Chapter 1*.

Preparing the build host system

The Yocto Project build environment expects a modern 64-bit Linux host with current development headers, common command-line tools, and Python 3.9 or newer. Plan for at least 4 CPU cores (8 or more recommended), 32 GB of RAM, and 140 GB of fast storage (200 GB once you keep multiple builds on disk) so BitBake can unpack source archives, grow the Shared State Cache (covered in *Chapter 6*), and finish long compile steps without swapping. Build time scales strongly with core count: a first build from scratch can take a few hours on a 4-core machine but far less on 8 or more cores, so prefer the most capable host you have for the initial build. Match your host to the release's supported-distribution matrix to avoid host-specific failures.

Note

Confirm that hardware virtualization extensions are active before relying on QEMU. Checks such as `lscpu | grep Virtualization`, `grep -E '(vmx|svm)' /proc/cpuinfo`, or `lsmod | grep kvm` should report support, and most BIOS or UEFI setups label the toggle as Intel VT-x, AMD-V, or SVM.

A verified virtualization setup lets QEMU use KVM hardware acceleration, which speeds up target emulation considerably.

Using Windows Subsystem for Linux (WSL 2)

The **Windows Subsystem for Linux (WSL 2)** integration on Windows 11 or Windows Server 2022 supplies a practical Linux environment. Install it with administrative PowerShell (`wsl --install`, or `wsl --install --web-download` if the Microsoft Store install stalls), enable the Virtual Machine Platform component, and reboot when prompted. After the reboot, set WSL 2 as the default (`wsl --set-default-version 2`), install Ubuntu 24.04 LTS or another supported distribution from the Microsoft Store, and confirm that virtualization is active (`systeminfo | find "Virtualization"` should report `Enabled`). The *Setting Up to Use Windows Subsystem For Linux (WSL 2)* section at `https://docs.yoctoproject.org/wrynose/dev-manual/start.html#setting-up-to-use-windows-subsystem-for-linux-wsl-2` explains prerequisites and caveats. Update `C:\Users\<username>\.wslconfig` so WSL receives enough CPU cores and memory, run `wsl --shutdown`, and then follow the same workflow used on a native Linux host.

Preparing a Linux-based system

If you have a choice of Linux distribution, start with one of the combinations validated for the Yocto Project 6.0 release:

- AlmaLinux 8
- AlmaLinux 9
- CentOS Stream 9
- CentOS Stream 10
- Debian GNU/Linux 11 (Bullseye)
- Debian GNU/Linux 12 (Bookworm)
- Debian GNU/Linux 13 (Trixie)
- Fedora 42
- Fedora 43
- openSUSE Leap 15.6
- openSUSE Leap 16.0
- Rocky Linux 8
- Rocky Linux 9
- Ubuntu 22.04 (LTS)
- Ubuntu 24.04 (LTS)
- Ubuntu 25.10
- Ubuntu 26.04 (LTS)

The release team still smoke-tests a small set of platforms whose upstream maintainers no longer publish updates: Fedora 39, Fedora 40, openSUSE Leap 15.5, and Ubuntu 25.04. Use them only for legacy build farms and keep new development on the supported list above.

Note

The Yocto Project does not have access to private updates that some of these distributions may receive. Therefore, testing coverage is limited if you rely on such updates.

The supported distributions list evolves during each release's maintenance window. As point releases ship (6.0.1, 6.0.2, and so on), the project may add newer distribution versions or drop older ones that fell out of upstream support. Check the *Supported Linux Distributions* section at `https://docs.yoctoproject.org/wrynose/ref-manual/system-requirements.html#supported-linux-distributions` for the current list, and revisit it after major point releases to confirm your host still qualifies.

Whenever you adopt another distribution, compare its package set with the list under *Required Packages for the Build Host* at `https://docs.yoctoproject.org/wrynose/ref-manual/system-requirements.html#required-packages-for-the-build-host`. Matching the documented versions for Git, tar, Python, and GCC avoids subtle build issues. If a package is too old, download the **Buildtools Tarball** described in *Required Git, tar, Python, make and gcc Versions* at `https://docs.yoctoproject.org/wrynose/ref-manual/system-requirements.html#required-git-tar-python-make-and-gcc-versions`.

Debian-based distributions

Install the development dependencies with APT and populate an English UTF-8 locale:

```
$ sudo apt update
$ sudo apt install build-essential chrpath cpio debianutils diffstat file gawk gcc
git iputils-ping libacl1 \
                  libcrypt-dev locales python3 python3-git python3-jinja2
python3-pexpect python3-pip \
                  python3-subunit python3-websockets socat texinfo unzip wget xz-
utils zstd
$ sudo locale-gen en_US.UTF-8
```

Two host-side details catch readers out on Debian-based systems, and both surface as a build failure rather than a package error. The first concerns older releases, where one of those packages is too old to do its job.

On Debian 11 and Ubuntu 22.04 the `python3-websockets` package is missing or too old for BitBake's hash equivalence client. Because this chapter uses the `poky-with-sstate` configuration, BitBake reaches the shared-state mirror's hash equivalence server over a `wss://` transport, so a recent `websockets` module is required. If a BitBake command fails with `No module named 'websockets'`, run `bitbake-setup install-buildtools` to install a bundled environment that provides it, or upgrade the module directly with `pip install --user --upgrade "websockets>=10.0"`. Disabling the shared-state mirror fragment also removes the

`wss://` requirement, but BitBake then cannot reuse the prebuilt shared-state artifacts and every build starts from scratch.

The second detail runs the other way, affecting the newest Ubuntu releases rather than the oldest.

Note

Warning!

Ubuntu 24.04 LTS and newer hosts restrict unprivileged user namespaces by default through AppArmor. BitBake relies on user namespaces, so the first build command aborts with `User namespaces are not usable by BitBake, possibly due to AppArmor`. Enable user namespaces for unprivileged processes with `sudo sysctl -w kernel.apparmor_restrict_unprivileged_userns=0`, and make the change survive a reboot by writing the same key to a drop-in file under `/etc/sysctl.d/`. Because the restriction comes from Ubuntu itself, the same fix is needed wherever an affected Ubuntu 24.04 (or newer) userspace runs, including inside a WSL 2 distribution or a container.

This workaround is the most common host-side fix new readers run into on a recent Ubuntu install, so it is worth pinning the sysctl change in a deployment script rather than reapplying it by hand.

Fedora

On Fedora 42 or 43, use DNF to install the required packages and localization support:

```
$ sudo dnf install bzip2 ccache chrpath cpio cpp diffstat diffutils file findutils
gawk gcc gcc-c++ git \
                   glibc-devel glibc-langpack-en gzip hostname libacl make patch
perl perl-Data-Dumper \
                   perl-File-Compare perl-File-Copy perl-FindBin perl-Text-
ParseWords perl-Thread-Queue \
                   perl-bignum perl-locale python python3 python3-GitPython
python3-jinja2 python3-pexpect \
                   python3-pip python3-websockets rpcgen socat tar texinfo unzip
wget which xz zstd \
                   util-linux-script
```

openSUSE Leap

On openSUSE Leap 16.0, use Zypper to install the required packages:

```
$ sudo zypper install bzip2 chrpath diffstat gcc gcc-c++ git gzip hostname libacl1
make makeinfo patch python3 \
                     python3-GitPython python3-Jinja2 python3-curses python3-
pexpect python3-pip \
                     python3-websockets python3-xml rpcgen socat tar wget which
xz zstd
```

Leap 16.0 modernized the Python package names that Leap 15.6 still ships under older identifiers, so the install line above will not work unchanged on 15.6.

Note

openSUSE Leap 15.6 ships an older Python stack: install `GitPython` with `sudo pip3 install GitPython` rather than from the distribution feed, drop `python3-websockets` and `python3-xml` (both unavailable in 15.6), and install the Buildtools Tarball to satisfy the required Git, tar, Python, make, and GCC versions.

AlmaLinux, Rocky Linux, and CentOS Stream

On AlmaLinux, Rocky Linux, and CentOS Stream, enable the EPEL repository and the CRB (CodeReady Builder) channel, called PowerTools on the 8-series releases, before installing the development packages:

```
$ sudo dnf install -y epel-release
$ sudo dnf install -y dnf-plugins-core
$ sudo dnf config-manager --set-enabled crb || sudo dnf config-manager --set-
enabled powertools
$ sudo dnf makecache
$ sudo dnf install -y bzip2 ccache chrpath cpio cpp diffstat diffutils gawk gcc
gcc-c++ git glibc-devel \
                     glibc-langpack-en gzip libacl make patch perl perl-Data-
Dumper perl-Text-ParseWords \
                     perl-Thread-Queue python3 python3-GitPython python3-jinja2
python3-pexpect python3-pip \
                     rpcgen socat tar texinfo unzip wget which xz zstd
```

These distributions ship older toolchain components than the Yocto Project 6.0 baseline expects. Install the Buildtools Tarball so the build system uses the validated Git, tar, Python, make, and GCC versions.

Setting up with bitbake-setup

The **bitbake-setup** tool automates project initialization by reading **BitBake Configuration Template** files that specify exactly which repositories to clone and how to configure the build environment. It handles layer cloning, directory creation, and configuration generation in a single command. This declarative model keeps environment setup reproducible across machines, simplifies sharing configurations with teammates, and works well with automated build systems.

> **Note**
>
> `bitbake-setup` is the Yocto Project's first-party workspace tool, and this book standardizes on it. Other tools solve the same problem with different trade-offs, notably `kas` from Siemens and the `repo` manifest workflow. *Chapter 18* returns to these alternatives in more depth once the fundamentals are in place.

Installing the bitbake-setup tool

Install `bitbake-setup` using pip:

```
$ pip install bitbake-setup
```

This installs the `bitbake-setup` command, which will fetch the required layers based on your configuration choices. The PyPI package is published by the OpenEmbedded BitBake Developers, so the same tool source backs both install paths.

> **Note**
>
> Installing from PyPI is expected to be the smoothest path. Alternatively, the development version can be cloned from the repository directly (`git clone https://git.openembedded.org/bitbake`), and the tool can be invoked from `bitbake/bin/bitbake-setup`. The version from PyPI server can be considered a stable version. When cloned from the repository, it will fetch the latest development version.

The new workflow also reshapes how the surrounding metadata is delivered, which is worth knowing before the first project is initialized.

> **Note**
>
> Earlier Yocto Project releases used Poky as a monolithic repository that bundled BitBake, OpenEmbedded Core, `meta-poky`, and `meta-yocto-bsp` together for convenience. The `bitbake-setup` workflow fetches these as separate repositories, giving you more control over versions and updates.

Initializing your first project

For this first build, we use the `poky-wrynose` configuration, which provides the Poky reference distribution baseline. *Chapter 14* covers how to explore available configurations and manage your setup over time.

Run the `init` subcommand to create a new project. For reproducible builds and automated build systems, use `--non-interactive` mode with explicit configuration choices:

```
$ bitbake-setup init --non-interactive poky-wrynose poky-with-sstate distro/poky
machine/qemux86-64
```

The command arguments specify the BitBake Configuration Template (`poky-wrynose`), the BitBake configuration (`poky-with-sstate`, which includes Shared State Cache settings), the target distribution (`distro/poky`), and the target machine (`machine/qemux86-64`). This explicit form documents your environment choices and makes builds reproducible across machines.

For interactive exploration, run `init` without arguments:

```
$ bitbake-setup init
```

The tool walks you through configuration selection, presenting choices for the distribution baseline, target machine, and optional features. Each prompt includes descriptions to help you understand the implications.

The initialization process clones the required repositories (OpenEmbedded Core, `meta-yocto`, and related layers), creates the build directory structure, and generates configuration files. This may take several minutes depending on your network connection. The `init` command can only be run once for each initialized **BitBake Setup**.

Understanding the bitbake-setup directory structure

After initialization completes, `bitbake-setup` creates a workspace under a top-level directory (by default `bitbake-builds/`). The following diagram shows the resulting layout.

```
bitbake-builds/
├── poky-wrynose                                                          <2>
│   ├── build                                                             <3>
│   │   ├── bitbake-cookerdaemon.log
│   │   ├── cache
│   │   ├── conf
│   │   ├── init-build-env                                                <4>
│   │   ├── README
│   │   └── tmp
│   ├── config                                                            <5>
│   │   ├── config-upstream.json
│   │   └── sources-fixed-revisions.json
│   └── layers                                                            <6>
│       ├── bitbake
│       ├── logs
│       ├── meta-yocto
│       ├── oe-init-build-env-dir -> openembedded-core
│       ├── oe-scripts -> openembedded-core/scripts
│       ├── openembedded-core
│       ├── setup-build -> openembedded-core/scripts/oe-setup-build
│       ├── sources-fixed-revisions.json -> ../config/sources-fixed-revisions.json
│       └── yocto-docs
└── site.conf                                                             <1>
```

Figure 2.1: The workspace directory after initialization

The numbered annotations in the diagram correspond to the following components:

1. Shared configuration applied across all setups in this workspace
2. BitBake Setup directory named after your configuration choices
3. BitBake build directory with configuration files and build artifacts
4. Script to source for activating the build environment
5. Version-controlled copy of the specific configuration used
6. Cloned layer repositories managed by `bitbake-setup`

The separation between `site.conf` (shared) and the BitBake Setup directory (project-specific) allows you to maintain multiple configurations for different products or Yocto Project versions in the same workspace.

When the `code` binary is on your `PATH`, `bitbake-setup` also writes a `bitbake.code-workspace` file into the BitBake Setup directory. That file lets you open the build and its layers in VS Code with the BitBake extension preconfigured, a workflow that *Chapter 3* walks through in full.

Activating the build environment

Before running BitBake commands, source the `init-build-env` script from your setup's build directory:

```
$ source ./bitbake-builds/poky-wrynose/build/init-build-env
```

This adds BitBake to your `PATH`, sets required environment variables, and changes your working directory to `/home/user/bitbake-builds/poky-wrynose/build/`. The script must be sourced in every new shell session where you want to run builds.

Reviewing the configuration files

After activation, review the configuration files under `build/conf/` for build policy settings. The `toolcfg.conf` file captures your **BitBake Configuration Instance**, the active build state assembled from your BitBake Configuration Template after `bitbake-config-build` applies any **BitBake Configuration Fragment** selections you enable:

- `local.conf`: Placeholder for temporary configuration settings needed during development.
- `bblayers.conf`: Lists the layers included in your build. The `bitbake-setup` tool populated this based on the BitBake Configuration Template, so manual editing is typically unnecessary.
- `toolcfg.conf`: Stores BitBake Configuration Fragment selections including `MACHINE` and `DISTRO` settings. When you enable a fragment, the tool updates this file with the corresponding values. BitBake includes it before `local.conf`, so a hard assignment in `local.conf` wins for ordinary variables, while `MACHINE` and `DISTRO` collisions raise a configuration error rather than letting one source silently win. Do not edit this file manually. Use `bitbake-config-build enable-fragment` to modify fragment selections. *Chapter 14* covers fragment management and team workflows in detail.

Building a target image

With the build environment active, you can now build your first image.

The Yocto Project ships several image recipes to help you test Linux kernels, consoles, and graphical stacks. List a subset with BitBake:

```
$ bitbake-layers show-recipes "core-image*"
```

The report lists every matching recipe with the layer and version that provides it. Run `bitbake-layers show-layers` to confirm the layer priorities that decide which layer wins when a recipe appears in more than one:

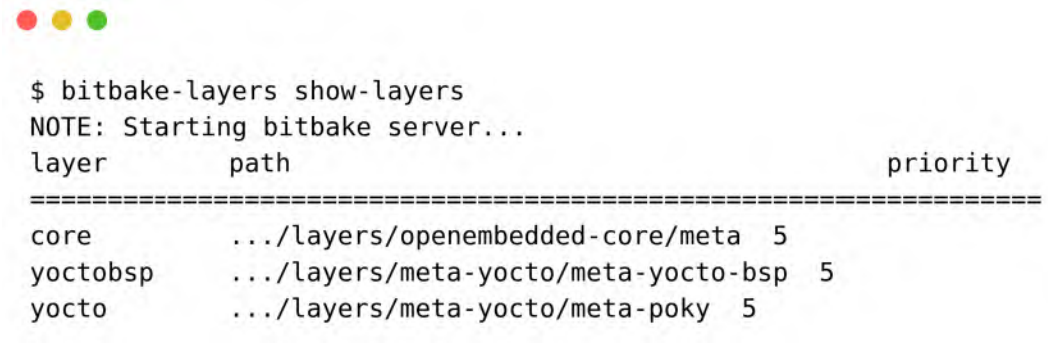


Figure 2.2: Layer priorities reported by bitbake-layers

None of the listing commands above build anything. They only inspect metadata to orient you before the build.

Tip

When you need a dependency graph for design reviews, `bitbake -g core-image-full-cmdline` writes one out. Give it a minute on a cold parse, because the terminal can look frozen while BitBake reads the metadata, even though it is working.

By default, the `poky-wrynose` configuration produces an image that does not permit the `root` user to log in, so the console login prompt would reject every attempt. Logging in is useful while you explore this first build, so enable the `core/yocto/root-login-with-empty-password` fragment before building:

```
$ bitbake-config-build enable-fragment core/yocto/root-login-with-empty-password
```

Figure 2.3: Allowing root login for the first build

The fragment lets `root` log in with an empty password on both the serial console and over SSH. Treat it as a development convenience: disable it before producing an image you intend to ship. *Chapter 14* covers fragment management in depth, and *Chapter 12* describes how to assign a real `root` password for production images.

The `core-image-full-cmdline` recipe produces a console-only image with a fuller set of command-line utilities than the minimal image, which makes it a good first target to explore. Build it:

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

BitBake parses metadata, resolves dependencies, and orchestrates fetch, compile, package, and image tasks. *Figure 2.4* captures the completion summary.

```
Loading cache...done.
Loaded 0 entries from dependency cache.
Parsing recipes...done.
Parsing of 950 .bb files complete (0 cached, 950 parsed). 1971 targets, 47 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies

Build Configuration:
BB_VERSION           = "2.18.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "ubuntu-24.04"
TARGET_SYS           = "x86_64-poky-linux"
MACHINE              = "qemux86-64"
SDKMACHINE           = "x86_64"
DISTRO               = "poky"
DISTRO_VERSION       = "6.0"
TUNE_FEATURES        = "m64 x86-64-v3"
meta                 = "wrynose:080...438"
meta-yocto-bsp
meta-poky            = "wrynose:904...f56"

NOTE: Fetching uninative binary shim http://downloads.yoctoproject.org/releases/uninative/5.1/x86_64-nativesdk-
libc-5.1.tar.xz;sha256sum=2b6...b85 (will check PREMIRRORS first)
Initialising tasks...done.
Checking sstate mirror object availability...Bitbake still alive (no events for 600s). Active tasks:
done.
Sstate summary: Wanted 3316 Local 0 Mirrors 3310 Missed 6 Current 0 (99% match, 0% complete)
NOTE: Executing Tasks
(...)
NOTE: Tasks Summary: Attempted 6131 tasks of which 6115 didn't need to be rerun and all succeeded.
```

Figure 2.4: BitBake build summary highlighting succeeded tasks

A clean summary like the one above hides the storage and bandwidth a first build actually consumes, which is worth planning for in advance.

Tip

First builds download several gigabytes of source archives and then build the cross-toolchain from scratch for the configured build, host, and target combination. Because this setup uses the `poky-with-sstate` configuration, BitBake also pulls prebuilt shared-state artifacts from the public Yocto Project mirror, which skips large amounts of compilation and shortens the first build considerably. Keep an eye on disk space (at least 140 GB free). Finished artifacts appear under `build/tmp/deploy/images/${MACHINE}`. Archive the directory after a successful build so teammates can reproduce the same binaries without rerunning BitBake. *Chapter 5* covers strategies for sharing fetched archives across machines.

Running images in QEMU

The `runqemu` helper from OpenEmbedded Core boots a built image under QEMU by pairing the Linux kernel with the matching root filesystem, without requiring manual QEMU command lines. With `MACHINE = "qemux86-64"`, the following invocation discovers the correct Linux kernel, root filesystem, and device tree paths:

```
$ runqemu snapshot
```

Figure 2.5 shows the boot process shortly before the Linux kernel mounts the root filesystem and hands off to the init manager, which is SysVinit in a Poky-based build. Once the boot completes, *Figure 2.6* illustrates the console login prompt.

Add `nographic` when you prefer a serial console, or point to explicit artifacts if you keep multiple images on disk (`runqemu build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.wic`). Networking defaults to user-mode SLiRP. Switch to `runqemu tap` when you need to interact with other hosts on the LAN. The *Using the Quick EMUlator (QEMU)* section of the Yocto Project development manual (`https://docs.yoctoproject.org/wrynose/dev-manual/qemu.html#using-the-quick-emulator-qemu`) explains advanced options such as snapshotting disks and exporting SSH ports.

```
[    2.191637]     Actions configured
[    2.195059] NET: Registered PF_INET6 protocol family
[    2.205934] Segment Routing with IPv6
[    2.206394] In-situ OAM (IOAM) with IPv6
[    2.207367] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[    2.211225] NET: Registered PF_PACKET protocol family
[    2.212240] Bridge firewalling registered
[    2.213755] l2tp_core: L2TP core driver, V2.0
[    2.213982] Key type dns_resolver registered
[    2.214760] NET: Registered PF_VSOCK protocol family
[    2.219389] IPI shorthand broadcast: enabled
[    2.245689] sched_clock: Marking stable (2197025306, 48093724)->(2255880399, -10761369)
[    2.247868] registered taskstats version 1
[    2.250232] Loading compiled-in X.509 certificates
[    2.271453] Key type .fscrypt registered
[    2.271608] Key type fscrypt-provisioning registered
[    2.275646] Btrfs loaded, zoned=no, fsverity=no
[    2.294809] Key type encrypted registered
[    2.296871] netconsole: network logging started
[    2.371004] usb 1-1: new high-speed USB device number 2 using ehci-pci
[    2.395697] ata5: SATA link down (SStatus 0 SControl 300)
[    2.403842] ata2: SATA link down (SStatus 0 SControl 300)
[    2.404369] ata6: SATA link down (SStatus 0 SControl 300)
[    2.405169] ata3: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
[    2.407443] ata4: SATA link down (SStatus 0 SControl 300)
[    2.409182] ata3.00: ATAPI: QEMU DVD-ROM, 2.5+, max UDMA/100
[    2.409555] ata3.00: applying bridge limits
[    2.410591] ata1: SATA link down (SStatus 0 SControl 300)
[    2.411542] ata3.00: configured for UDMA/100
[    2.422949] scsi 2:0:0:0: CD-ROM            QEMU     QEMU DVD-ROM     2.5+ PQ: 0 ANSI: 5
[    2.444690] sr 2:0:0:0: [sr0] scsi3-mmc drive: 4x/4x cd/rw xa/form2 tray
[    2.445417] cdrom: Uniform CD-ROM driver Revision: 3.20
[    2.497684] IP-Config: Complete:
[    2.497820]      device=eth0, hwaddr=52:54:00:12:34:02, ipaddr=192.168.7.2, mask=255.255.255.0,
gw=192.168.7.1
[    2.498127]      host=192.168.7.2, domain=, nis-domain=(none)
[    2.498345]      bootserver=255.255.255.255, rootserver=255.255.255.255, rootpath=
[    2.498398]      nameserver0=8.8.8.8
[    2.502623] clk: Disabling unused clocks
[    2.512912] md: Waiting for all devices to be available before autodetect
[    2.513263] md: If you don't use raid, use raid=noautodetect
[    2.513554] md: Autodetecting RAID arrays.
[    2.513651] md: autorun ...
[    2.513748] md: ... autorun DONE.
[    2.539681] input: QEMU QEMU USB Tablet as /devices/pci0000:00/0000:00:1d.7/usb1/1-1/1-
1:1.0/0003:0627:0001.0001/input/input1
[    2.541792] hid-generic 0003:0627:0001.0001: input: USB HID v0.01 Mouse [QEMU QEMU USB Tablet] on usb-
0000:00:1d.7-1/input0
[    2.585356] tsc: Refined TSC clocksource calibration: 2304.075 MHz
[    2.586013] clocksource: tsc: mask: 0xffffffffffffffff max_cycles: 0x21363e9bf2f, max_idle_ns: 440795264629
ns
[    2.586534] clocksource: Switched to clocksource tsc
[    2.608385] EXT4-fs (vda): mounted filesystem 91b577ab-d9d9-469f-b9d4-c22296686b7a r/w with ordered data
mode. Quota mode: disabled.
[    2.609361] VFS: Mounted root (ext4 filesystem) on device 253:0.
[    2.612556] devtmpfs: mounted
```

Figure 2.5: QEMU boot sequence for core-image-full-cmdline

The boot sequence confirms that the Linux kernel arguments, device tree selection, and serial console settings from the image recipe are wired correctly.



Figure 2.6: Console login after booting core-image-full-cmdline in QEMU

Because you enabled the `core/yocto/root-login-with-empty-password` fragment earlier, you can log in as `root` with an empty password. From here you can inspect the filesystem, validate services, and exercise application behavior without dedicated hardware. *Chapter 17* returns to QEMU for accelerated product prototyping, while *Chapter 10* focuses on debugging techniques.

Summary

You now have a working Yocto Project 6.0 build environment initialized with `bitbake-setup`. You successfully installed host dependencies, cloned BitBake, initialized a project with a BitBake Configuration Template, built your first image, and booted it in QEMU.

The next chapter moves this same workflow into VS Code, opening the workspace that `bitbake-setup` generated so you can edit recipes, build, and boot under QEMU from a single window.

3

Using VS Code with the BitBake Extension

The Yocto Project 6.0 (Wrynose) workflow you used to produce your first image ran entirely from the command line. This chapter shows how to drive the same workflow from Visual Studio Code (VS Code), the editor that `bitbake-setup` already prepared a workspace for when it initialized your project. You will open that generated workspace, install the official BitBake extension, read recipes with syntax highlighting and context-based suggestions, build `core-image-full-cmdline` without leaving the editor, and boot the result under QEMU two different ways. The goal is a comfortable edit-build-run loop that keeps recipe navigation, the build command, and the emulator inside one window.

An editor that understands BitBake metadata changes how quickly you move through a layer. A plain text editor treats a recipe as an undifferentiated block of characters, so a misplaced operator or an unknown variable surfaces only when a build fails minutes later. The BitBake extension reads your actual layer configuration, highlights the shell, and Python embedded in a recipe, and flags parsing problems as you save, which shortens the distance between a typo and its discovery. Setting the workspace up once also gives every engineer on the team the same recipe navigation and build shortcuts, so a project does not depend on each person reconstructing an editor configuration by hand.

We will cover the following main topics:

- Opening the generated workspace file
- Installing the Yocto Project BitBake extension
- Working with syntax highlighting and context-based suggestions

- Building core-image-full-cmdline from VS Code
- Running runqemu from VS Code

> **Note**
>
> **Download the PDF version of this book**
>
> Your purchase includes a DRM-free PDF copy of this book and a range of exclusive benefits. To unlock everything, follow the *Free benefits with your book* section in the *Preface*.

Technical requirements

This chapter builds on the project you initialized in *Chapter 2*. You need an initialized BitBake Setup workspace under `bitbake-builds/` with the layers cloned.

You also need VS Code installed on the build host, with its `code` command available on your `PATH`. On a Linux host, install VS Code from your distribution's package feed or from the Microsoft download page at `https://code.visualstudio.com/download`. Under Windows Subsystem for Linux (WSL 2), install VS Code on Windows together with its WSL extension so the editor runs against your Linux build environment rather than the Windows side. Microsoft's guide at `https://code.visualstudio.com/docs/remote/wsl` walks through that WSL setup step by step. A network connection is required the first time you install the BitBake extension from the Visual Studio Marketplace.

Opening the generated workspace file

When `bitbake-setup init` runs on a host where the `code` binary is on your `PATH`, it writes a workspace file named `bitbake.code-workspace` into the BitBake Setup directory alongside your build. For the `poky-wrynose` BitBake Setup from *Chapter 2*, that file is at `bitbake-builds/poky-wrynose/bitbake.code-workspace`. The file points the BitBake extension at your build directory and the environment activation script, and it lists the cloned layer directories as workspace folders so each layer's recipes appear in the editor's file tree. Any folders or settings you add to the file by hand are preserved the next time `bitbake-setup` updates it.

If you initialized the project on a host without VS Code, the file is not generated by default. Pass `--init-vscode` to generate it after the fact, or pass `--no-init-vscode` to skip it on a workstation that standardizes on a different editor. *Chapter 14* covers these initialization options alongside the rest of the `bitbake-setup` configuration workflow.

Open the workspace with the code command, passing the path to the generated file so VS Code loads every layer folder at once rather than a single directory.

The first time you open the workspace, VS Code notices that it recommends an extension you do not yet have installed and prompts you to add it, as shown below.

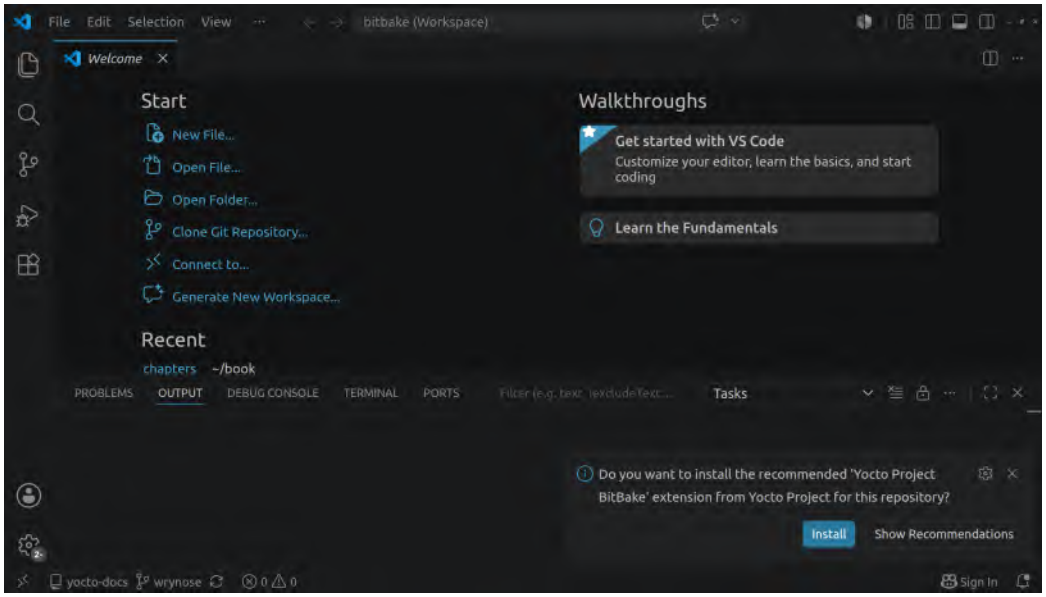


Figure 3.1: VS Code prompting to install the recommended Yocto Project BitBake extension

Installing the Yocto Project BitBake extension

The recommendation in the workspace file points at the **Yocto Project BitBake extension**, published by the Yocto Project under the identifier `yocto-project.yocto-bitbake`. Accept the prompt to install it and any other extensions it depends on, or open the Extensions view, search for the same identifier, and install it and its dependencies from the Visual Studio Marketplace at `https://marketplace.visualstudio.com/items?itemName=yocto-project.yocto-bitbake`. The extension is the official successor to the community project that earlier carried BitBake support, and the Yocto Project now maintains it directly.

Because the generated workspace already records the path to your build directory and the environment activation script, you do not need to set the extension's `bitbake.pathToBuildFolder` or `bitbake.pathToEnvScript` options by hand. The extension reads them from the workspace file and is ready to parse your layers as soon as it finishes installing. On a host that wraps BitBake in a container or another launcher, the extension

exposes additional settings such as `bitbake.commandWrapper`, but the default `poky-wrynose` BitBake Setup needs none of them.

VS Code keeps Marketplace extensions current on its own, installing new releases of the BitBake extension automatically as the Yocto Project publishes them. If you have turned off automatic updates, open the **Extensions** view, select the extension, and use its **Update** action whenever one is offered so your editor tracks the latest parser and language support.

Working with syntax highlighting and context-based suggestions

The two features you reach for most often are syntax highlighting and context-based suggestions. Highlighting applies to every kind of BitBake metadata file, including recipes (`.bb`), append files (`.bbappend`), classes (`.bbclass`), configuration files (`.conf`), and the include files (`.inc`) they share. The extension also understands the languages embedded inside a recipe, so the shell body of a task and an inline Python expansion each receive their own coloring rather than being rendered as flat text. Open a BusyBox recipe from `meta/recipes-core/busybox/` in the workspace tree to see a representative recipe with both embedded languages, as shown below.

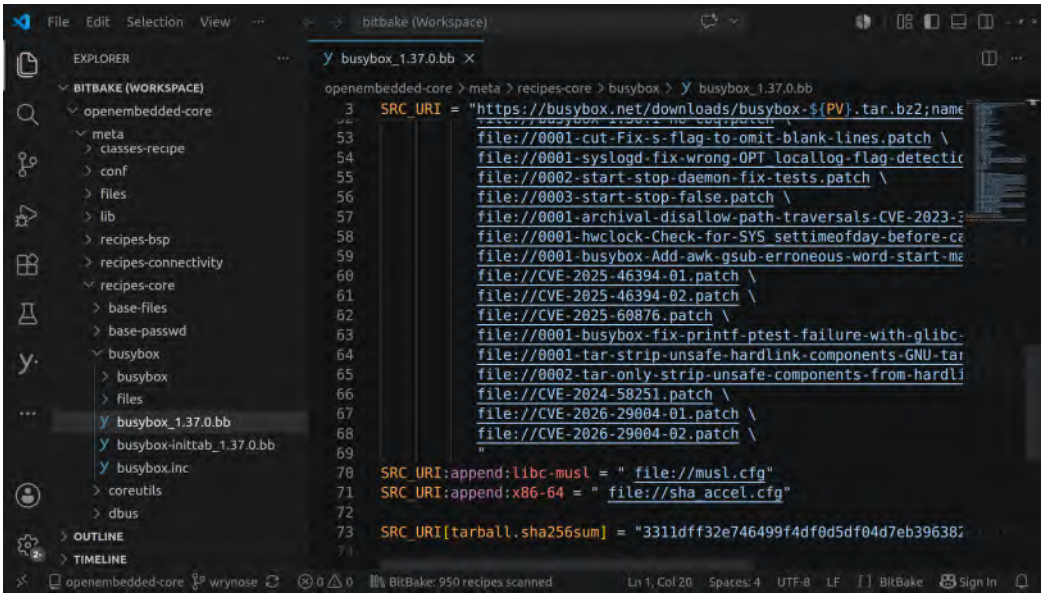


Figure 3.2: Syntax highlighting in a recipe

Context-based suggestions appear as you type and on demand with *Ctrl+Space*. They are scoped to your actual layer configuration rather than a static word list, so completing `inherit`, `require`, or `include` offers only classes and files that the configured layers really provide. That scoping is what keeps the suggestions trustworthy: a class that no enabled layer ships never appears, so an accepted completion resolves during parsing instead of failing the build later, as shown below.

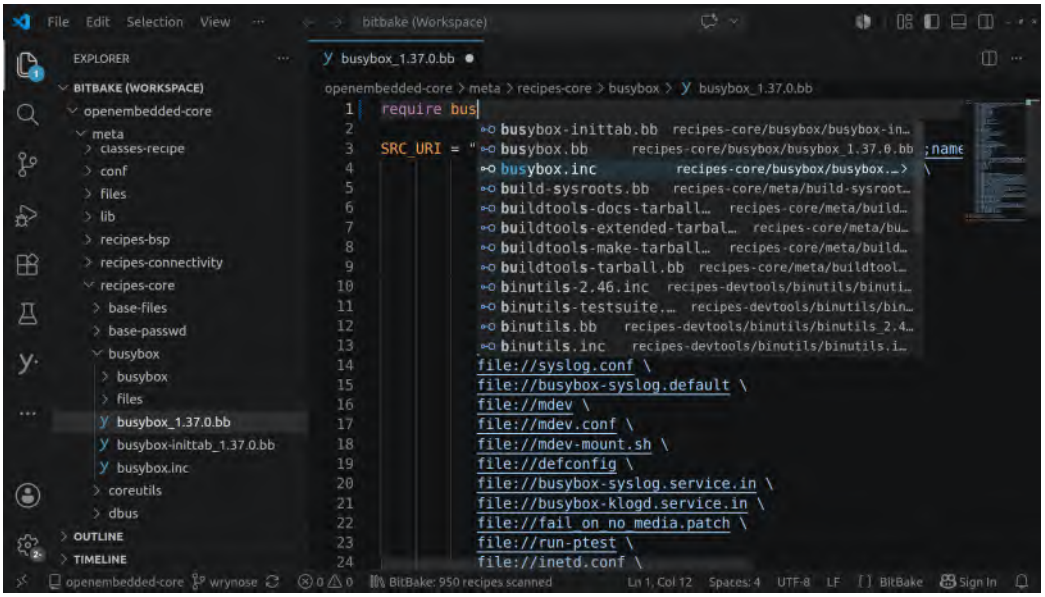


Figure 3.3: Context-based suggestion list in a recipe

Touring the extension's other features

Syntax highlighting and suggestions are the entry point. The extension adds several more features worth knowing as you spend more time in a layer:

- Go-to-definition (*Ctrl+Click* on a variable, class, or include) jumps to where the symbol is defined anywhere in the include hierarchy.
- Hovering a known BitBake variable shows its documented meaning without leaving the file
- Saving a metadata file runs BitBake in parse-only mode in the background and reports syntax errors in the Problems panel, so a broken assignment surfaces immediately

- The **Recipes** view in the side bar lists the recipes the workspace can see and offers a build shortcut for each
- A dedicated BitBake terminal profile opens a shell with the build environment already activated

For the deeper side of editor integration, including configuring the extension for an Extensible SDK installation so `devtool` commands and the eSDK environment are available without the full source tree, see *Chapter 9*.

Building core-image-full-cmdline from VS Code

With the extension installed, you can build the same image from *Chapter 2* without switching to a separate terminal window. The simplest path uses the Recipes view: locate `core-image-full-cmdline` in the list and trigger its build action, and the extension runs the build in an integrated terminal with the environment already in place.

You can also build from a plain integrated terminal. Open one inside the workspace, source the activation script as you did in *Chapter 2*, and run `bitbake core-image-full-cmdline`. The command, its output, and any error it reports stay in the same window as the recipe you are editing, which is the practical advantage of building from the editor. Because the build reuses the Shared State Cache from your first build, a rebuild with no metadata changes finishes in seconds rather than repeating the original fetch and compile work, as shown below.

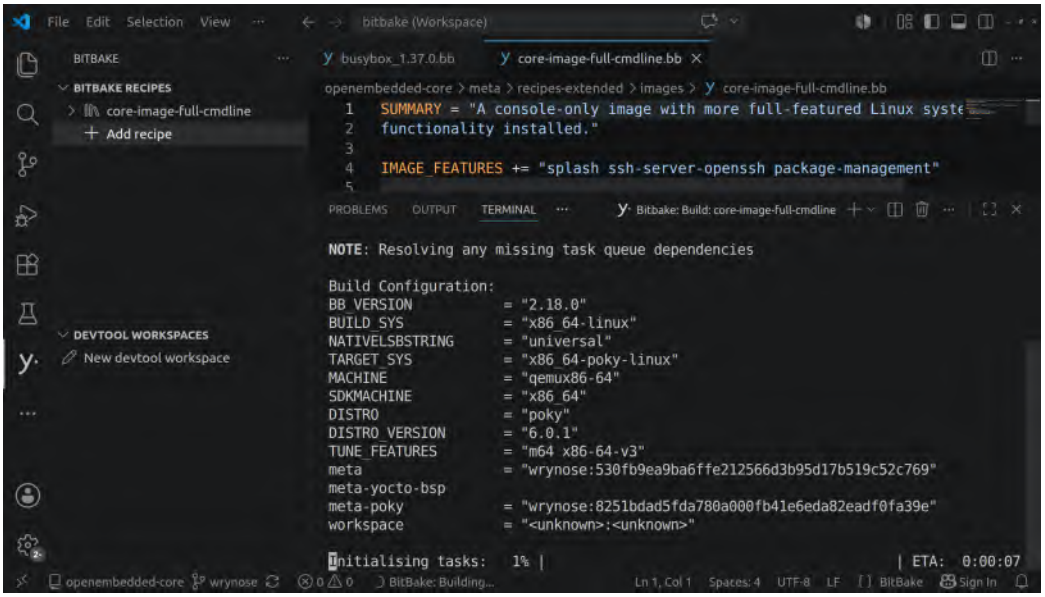


Figure 3.4: Building core-image-full-cmdline with the extension interface

Running runqemu from VS Code

Once an image is built, you can boot it under QEMU from inside VS Code in two ways. The first uses an integrated terminal directly, and the second wires the command into a reusable workspace task so it is a single action away.

Using the integrated terminal

A plain integrated terminal starts without the build environment, so source the activation script before launching the emulator. The following commands activate the environment and boot the most recently built image in snapshot mode:

```
$ source build/init-build-env
$ runqemu snapshot
```

Snapshot mode keeps the root filesystem read-only and discards anything the running image writes, so every boot starts from the same clean state. That makes it the right choice while you exercise an image repeatedly during development, because no boot can leave residue that quietly changes the next one. If you instead open the extension's BitBake terminal profile, the environment is already active and you can run `runqemu snapshot` on its own.

Defining a workspace task

Typing the same two commands for every boot becomes tedious, so VS Code lets you record them as a task in the workspace. A task gives the command a label and makes it available from the Terminal menu's Run Task action, which is convenient for a command you run many times a day. Add the following `tasks` block to the workspace file to source the environment and run the emulator in one step:

```
{
  "tasks": {
    "version": "2.0.0",
    "tasks": [
      {
        "label": "runqemu snapshot",
        "type": "shell",
        "command": ". init-build-env && runqemu snapshot",
        "options": {
          "cwd": "/home/user/bitbake-builds/poky-wrynose/build"
        },
        "problemMatcher": []
      }
```

```
      ]
    }
  }
```

The task sets `cwd` to the build directory and sources `init-build-env` there so `runqemu` is on the `PATH`, then boots the image in snapshot mode. After saving the workspace file, open the command palette with *Ctrl+Shift+P*, type `Run Task`, and select the `runqemu snapshot` entry:

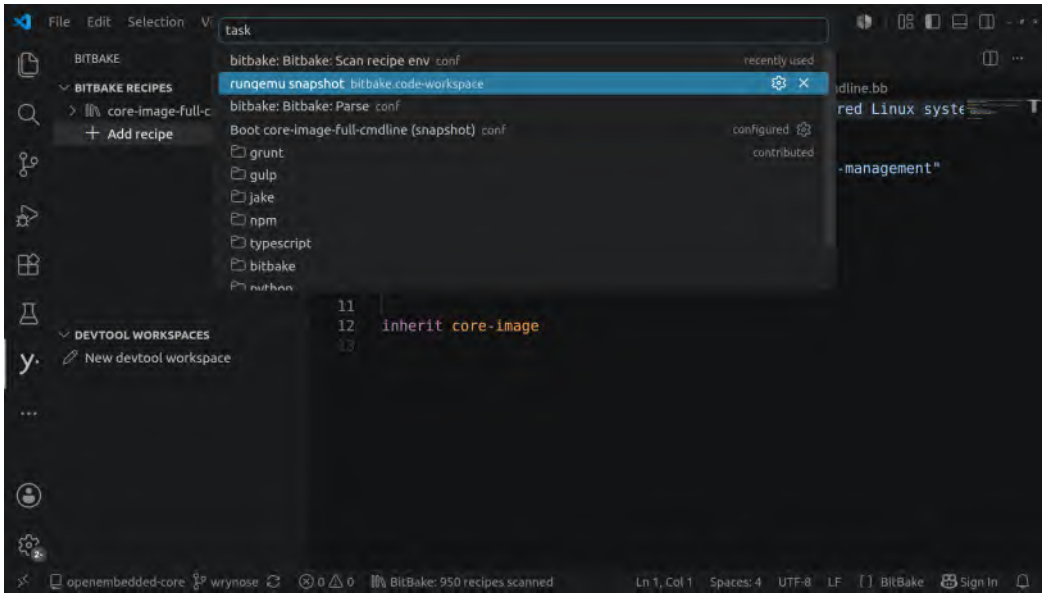


Figure 3.5: Selecting the runqemu snapshot task from the VS Code command palette

VS Code opens an integrated terminal, sources the environment, and starts the emulator. The QEMU window appears on top of the editor while the terminal streams boot output in the background:



Figure 3.6: QEMU booting the image launched from VS Code

The emulator behaves exactly as it did from a shell, but you launched it without leaving the editor.

Summary

This chapter moved your first-build workflow into VS Code. You opened the `bitbake.code-workspace` file that `bitbake-setup` generated, installed the Yocto Project BitBake extension, and used its syntax highlighting and context-based suggestions to read recipes with confidence. You then built `core-image-full-cmdline` from the editor and booted it under QEMU two ways, first from an integrated terminal and then through a reusable workspace task that runs `runqemu snapshot`.

The next chapter steps back from tooling to explain how BitBake itself assembles metadata from your layers into a coherent build policy, giving you the mental model you need before writing your own recipes and classes.

4

Meeting the BitBake Tool

This chapter explains BitBake's place in the build workflow and shows how it assembles metadata from multiple layers into a coherent build policy. You will see how layer configuration, recipes, and classes combine to form the metadata collection that powers Yocto Project 6.0 (Wrynose) builds, and which command-line tools reveal that structure while a build is active.

The overall goal is to give you a working mental model of the metadata hierarchy before you write your first layer or recipe. Without that model, layer priorities and variable assignments can produce results that are difficult to trace: a value appears to be set correctly in one file, yet the build uses a different value sourced from a higher-priority layer. Engineers who treat BitBake as a black box often lose time reading the wrong configuration file when a build fails, or produce layers that silently conflict with OpenEmbedded Core defaults. Understanding how BitBake locates, parses, and merges metadata makes that class of problem straightforward to diagnose.

We will cover the following main topics:

- BitBake's role in the build workflow
- Metadata collections and layer structure
- Inspecting layers with bitbake-layers
- Configuration, classes, and recipes

Technical requirements

This chapter assumes a build host configured as described in *Chapter 2*, with a BitBake Setup workspace initialized and the Poky layers available. A completed first build is helpful so that `bitbake-layers` has metadata to inspect, but the conceptual sections do not require it.

BitBake's role in the build workflow

BitBake was inspired by the Gentoo Portage package management system, originally lived inside the OpenEmbedded Project, and was spun off in 2004 as an independent task-execution engine that now powers both the Yocto Project and the OpenEmbedded Project. BitBake parses recipes, classes, and configuration files, constructs a dependency graph, and executes tasks in parallel when dependencies allow it. Each task's shell or Python fragment is logged under `build/tmp/work/<recipe-name>/…/temp/`, which keeps the troubleshooting loop short. Separately, the optional **Build History** feature records image, package, and SDK artifact snapshots so you can compare two builds and catch unexpected changes.

From the command line, developers usually enter `bitbake <target>` from an initialized build directory. Behind that simple interface, BitBake loads multiple configuration layers, merges variable assignments, and feeds the resulting policy set into its task executor. When something goes wrong, the task log shows the exact shell fragment or Python helper that failed. The *BitBake User Manual* at `https://docs.yoctoproject.org/bitbake/2.18/` documents every variable, class, and command this chapter introduces.

Metadata collections and layer structure

BitBake groups metadata into layers. Each layer exposes recipes, classes, and configuration files that describe a particular scope, such as a distribution policy, a board support package, or auxiliary tooling. The Poky reference distribution that the Yocto Project maintains is built from three core layers: OpenEmbedded Core under `meta/`, the Poky distribution policy under `meta-poky/`, and the Yocto Project reference BSP layer under `meta-yocto-bsp/`. Additional layers extend the build environment by adding new recipes or adjusting default settings.

Layer behavior is defined by `conf/layer.conf`. That file sets collection names, declares priorities, and lists dependent layers so BitBake understands how the metadata tree fits together. The following configuration shows the relevant entries from the `meta-poky` configuration file.

```
# 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 += "yocto"
BBFILE_PATTERN_yocto = "^${LAYERDIR}/"
BBFILE_PRIORITY_yocto = "5"

LAYERSERIES_COMPAT_yocto = "wrynose"

# This should only be incremented on significant changes that will
# cause compatibility issues with other layers
LAYERVERSION_yocto = "3"

LAYERDEPENDS_yocto = "core"

REQUIRED_POKY_BBLAYERS_CONF_VERSION = "2"
```

Figure 4.1: Key settings in meta-poky/conf/layer.conf

The metadata excerpt shown above lists the core variables you will encounter in every layer:

- `BBFILE_COLLECTIONS` assigns a collection identifier. In the sample, it registers the `yocto` collection so BitBake can use that name when reporting assignments or conflicts.
- `BBFILE_PATTERN_<collection>` limits the recipe search scope. The pattern typically anchors on `LAYERDIR` so BitBake scans only the current layer for matching files.
- `BBFILE_PRIORITY_<collection>` sets layer priority. Higher numbers take precedence when two layers export recipes with the same name.
- `LAYERDEPENDS_<collection>` lists layer dependencies, telling BitBake that this layer's metadata depends on metadata from other layers. Each dependency is named by its collection identifier, not by its directory name, which is why `meta-poky` depends on core rather than on meta or openembedded-core. That entry makes BitBake load OpenEmbedded Core before it parses the Poky policy files.

Layer search paths depend on `BBPATH` and `BBFILES`. `BBPATH` behaves like the shell `PATH`, adding directories to the metadata lookup path. `BBFILES` captures glob patterns so BitBake can index all `.bb` and `.bbappend` files exposed by the layer.

This chapter only sketches the layer ecosystem. *Chapter 11* surveys the third-party layers that extend the metadata tree, including BSPs, hardware vendors, and application stacks. The grammar that recipes and classes use to assign values, queue override operators, and define tasks is the subject of *Chapter 8*.

Inspecting layers with bitbake-layers

The `bitbake-layers` helper reduces the effort required to audit layer configuration. After activating the build environment as described in *Chapter 2*, run the following command to confirm that the build environment sees the expected layers:

```
$ bitbake-layers show-layers
```

The command output lists each layer, its priority, and the resolved path within the build tree, as shown below. Each path places the layer below the upstream repository that `bitbake-setup` cloned, which is why OpenEmbedded Core appears under `openembedded-core/` and the Poky distribution policy and reference BSP both appear under `meta-yocto/`.

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

Figure 4.2: Output of bitbake-layers show-layers

`bitbake-layers` can also compare two layers, list recipes provided by a layer, and report appends that match a given recipe. These inspection commands help you validate that custom layers register correctly before you launch a full build.

The `show-appends` subcommand is particularly useful when you need to confirm which `.bbappend` files apply to a recipe. Pass the recipe name without the version suffix:

```
$ bitbake-layers show-appends linux-yocto
```

`bitbake-layers` lists every matching append file together with the layer that provides it. The following figure applies the command to the `linux-yocto` recipe and reveals that `meta-yocto-bsp` contributes a `.bbappend` targeting the same version.

```
$ bitbake-layers show-appends linux-yocto
NOTE: Starting bitbake server...
Loading cache: 100% |################################################| Time: 0:00:00
Loaded 1971 entries from dependency cache.
=== Matched appended recipes ===
linux-yocto_6.18.bb:
  .../layers/meta-yocto/meta-yocto-bsp/recipes-kernel/linux/linux-yocto_6.18.bbappend
```

Figure 4.3: bitbake-layers show-appends output for linux-yocto

Configuration, classes, and recipes

BitBake metadata falls into three categories. Configuration files (`.conf`) set global behavior such as the target machine, distribution policies, and download directories. Classes (`.bbclass`) centralize reusable functions, which keeps recipes concise and consistent. Recipes (`.bb`) and append files (`.bbappend`) describe how to fetch, patch, compile, package, and deploy deliverables.

To make these categories concrete, it helps to see the shape of each before you write your own. The configuration file you already met is `meta-poky/conf/layer.conf` in *Figure 4.1*, which is a set of variable assignments and nothing more. A recipe adds source information and build logic on top of such assignments, as the following skeleton shows.

```
SUMMARY = "Example application for a custom layer"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;md5=0835ade698e0bcf8506ecda2f7b4f302"

SRC_URI = "git://example.com/example-app.git;branch=main;protocol=https"
PV = "1.0"

inherit autotools
```

Figure 4.4: Skeleton of a recipe (.bb)

Every line either identifies the software (`SUMMARY`, `LICENSE`, `LIC_FILES_CHKSUM`), locates its source (`SRC_URI`, `PV`, the package version), or reuses build logic (`inherit autotools`). The recipe defines no tasks of its own because the `autotools` class already supplies the configure, compile, and install steps. That reuse is exactly what a class provides, as the next skeleton illustrates.

```
# Reusable logic that recipes pick up by adding `inherit example`.
do_report_size() {
    bbwarn "Installed size: $(du -sh ${D} | cut -f1)"
}

addtask report_size after do_install before do_build
```

Figure 4.5: Skeleton of a class (.bbclass)

A recipe that adds `inherit example` picks up the `do_report_size` task without repeating its definition. The three categories therefore divide the work: configuration sets policy, recipes describe individual deliverables, and classes hold the logic that those recipes share. *Chapter 8* details the grammar these files use, and *Chapter 12* builds complete recipes and layers from scratch.

Beyond these core metadata types, the `bitbake-config-build` tool introduces BitBake Configuration Fragments. A fragment is a modular configuration unit that defines a specific build aspect, such as the target machine (`machine/qemux86-64`) or distribution (`distro/poky`). Fragments update `build/conf/toolcfg.conf` when enabled and provide a structured alternative to editing configuration files directly. *Chapter 14* covers fragment management with `bitbake-config-build`.

Note

Classes and recipes can interleave shell fragments with Python helper functions. BitBake determines the correct language for each function based on prefixes such as `python do_packagedata`, which declares the `do_packagedata` task in Python. That task lives in `package.bbclass` and writes per-recipe metadata under `build/tmp/pkgdata`.

BitBake connects these categories through **inheritance**. A recipe can inherit multiple classes to gain task implementations and helper logic. For example, `linux-yocto_6.18.bb` inherits `kernel.bbclass`, which provides the standard Linux kernel task sequence, firmware packaging hooks, and deploy helpers. When BitBake parses that recipe, it builds a task list by combining the recipe's own tasks with every task added by inherited classes.

BitBake writes the computed **Task Graph** and environment history to the `build/tmp/` work directory. You can inspect the final variables seen by a recipe with `bitbake -e <recipe-name>` and trace the origin of each value, including assignments applied by layer priorities or machine configuration files. When you want a single variable rather than the entire environment dump, `bitbake-getvar -r <recipe-name> <variable>` prints just that value after the same metadata merge. *Chapter 8* covers it in detail. Understanding these relationships helps when you introduce new metadata or debug conflicts between layers. The mechanics of how BitBake parses, schedules, and runs the underlying tasks are covered in *Chapter 5*.

Summary

This chapter explained how BitBake structures metadata, how layer configuration guides recipe discovery, and which tools help you audit that configuration. You learned how `bitbake-layers show-layers` reflects the active metadata tree, why `conf/layer.conf` controls collection priorities, and how classes extend recipes with shared behavior.

The next chapter takes a deeper look at metadata grammar so you can author recipes and classes that match Yocto Project 6.0 policies.

Part 2

Core Concepts

With a working build in place, this part opens it up and covers the concepts the rest of the book leans on. You will follow BitBake through a full cycle: how it prepares the environment, indexes and parses metadata, resolves dependencies, fetches and caches source code, and schedules the tasks that turn that source into a package, including how hash equivalence decides what can be reused.

You will then read the artifacts a build leaves behind. That means the build directory and how it is constructed, the temporary build tree, the per-recipe work directories, and the sysroots that make cross-compilation work. Packaging support follows: the supported package formats and how to choose between them, code that runs at install time, the Shared State Cache, package versioning, runtime dependencies, root filesystem assembly, and package feeds.

The part ends with the metadata syntax itself: assignment and expansion operators, appending, prepending, and removal, conditional metadata and the override mechanism, file inclusion, Python and shell functions, and class inheritance. That chapter is the reference you will come back to whenever a recipe does something you did not expect.

This part of the book includes the following chapters:

- *Chapter 5*, *Grasping the BitBake Tool*
- *Chapter 6*, *Detailing the Temporary Build Directory*
- *Chapter 7*, *Assimilating Packaging Support*
- *Chapter 8*, *Understanding BitBake Metadata*

5

Grasping the BitBake Tool

Chapter 4 introduced metadata collection concepts and highlighted the role of `conf/layer.conf`. This chapter goes deeper into how BitBake actually works: how it reads configuration and recipe files, constructs a dependency graph, fetches source code, and schedules the tasks that turn metadata into a bootable image. By the end of the chapter, you will be able to trace a build from the first `bitbake` invocation through task completion, interpret the execution log, and pinpoint why a task fails or runs out of order.

The goal is to give you a working mental model of BitBake's internals so you can act on what the tool tells you. An engineer who cannot read BitBake's execution trace must rely on trial and error during a release crunch, adding hours to a debugging session that could otherwise take minutes. Understanding how BitBake resolves priorities, triggers fetchers, and sequences tasks turns that trace from an opaque log into a precise diagnostic tool. This knowledge applies directly to the layer management covered in *Chapter 4* and to the recipe customization work you will encounter in later chapters.

We will cover the following main topics:

- Preparing the environment
- Indexing metadata
- Initializing BitBake
- Parsing metadata
- Dependencies
- Fetching the source code
- Optimizing source code downloads
- Understanding BitBake tasks

Technical requirements

This chapter assumes a build host configured according to *Chapter 2*, including all required host packages and a working BitBake Setup workspace. You will run `bitbake` commands while following along, so that workspace must be functional and the build environment initialized with `source ./bitbake-builds/poky-wrynose/build/init-build-env` before you start.

No additional hardware or software beyond the *Chapter 2* baseline is required for this chapter.

Preparing the environment

Most Yocto Project environments rely on several layers that contribute configuration, classes, and recipes. The `bitbake-setup` tool creates a `build/` directory containing three important files that BitBake reads first:

- `build/conf/toolcfg.conf`: Stores BitBake Configuration Fragment selections including `MACHINE` and `DISTRO` settings. BitBake reads this file before `build/conf/local.conf`, a hard assignment in `build/conf/local.conf` takes precedence for ordinary variables. `MACHINE` and `DISTRO` are the exception: they arrive through built-in fragments, and assigning either in `build/conf/local.conf` while the matching fragment is enabled aborts the build with a configuration error rather than letting one source silently win. Do not edit this file manually. *Chapter 14* covers fragment management with `bitbake-config-build`.
- `build/conf/local.conf`: Placeholder for temporary configuration settings needed during development, as described in *Chapter 2*.
- `build/conf/bblayers.conf`: Lists every layer that BitBake must parse, in the order it parses them. Parse order settles configuration-variable conflicts, while `BBFILE_PRIORITY` settles recipe and append precedence, as explained below.

The following shows a representative `build/conf/bblayers.conf`.

```
BBLAYERS ?= " \
  /home/user/bitbake-builds/poky-wrynose/layers/openembedded-core/meta \
  /home/user/bitbake-builds/poky-wrynose/layers/meta-yocto/meta-yocto-bsp \
  /home/user/bitbake-builds/poky-wrynose/layers/meta-yocto/meta-poky \
  "
```

BitBake processes the entries in `BBLAYERS` from left to right and loads each layer's metadata into the global collection.

Tip

Raise `BBFILE_PRIORITY` on a custom product layer when you want its recipes and appends to take precedence over platform support layers. The position of a layer in `BBLAYERS` controls parse order, not conflict resolution.

Indexing metadata

BitBake gathers metadata from every layer listed in `build/conf/bblayers.conf`. The sources map to three familiar file types:

- Configuration files (`.conf`): Define global variables, machine configurations, and distribution policy that shape the entire build
- Recipes (`.bb`): Describe how to fetch, configure, compile, install, and package individual software components
- Class files (`.bbclass`): Provide reusable functions and logic that recipes inherit to maintain consistent behavior

Each layer assigns a `BBFILE_PRIORITY`. When the same recipe name exists in more than one layer, the higher-priority layer supplies it, and appends apply in priority order. Configuration-variable conflicts are settled by parse order instead, so keeping both the layer order and these priorities documented eases future maintenance.

Initializing BitBake

BitBake's initialization sequence consists of two stages:

1. BitBake expands `BBLAYERS`, reads every `conf/layer.conf`, and records metadata such as `BBFILE_PATTERN`, `BBFILE_COLLECTIONS`, and `LAYERSERIES_COMPAT`
2. With the layer list established, BitBake parses `meta/conf/bitbake.conf` from OpenEmbedded Core, then folds in `build/conf/local.conf`, files under `conf/machine/`, and any distribution configuration named by `DISTRO`

The following figure highlights a portion of `meta/conf/bitbake.conf`, which anchors fundamental variables and includes additional fragments.

```
...
# Path prefixes
export base_prefix = ""
export prefix = "/usr"
export exec_prefix = "${prefix}"
...
# Architecture independent paths
export sysconfdir = "${base_prefix}/etc"
export servicedir = "${base_prefix}/srv"
export sharedstatedir = "${base_prefix}/com"
export localstatedir = "${base_prefix}/var"
...
##################################################################
# Include the rest of the config files.
##################################################################

require conf/abi_version.conf
include conf/site.conf
include conf/auto.conf
include conf/toolcfg.conf
include conf/local.conf

OE_FRAGMENTS_PREFIX ?= "conf/fragments"
OE_FRAGMENTS_METADATA_VARS ?= "BB_CONF_FRAGMENT_SUMMARY BB_CONF_FRAGMENT_DESCRIPTION"
OE_FRAGMENTS_BUILTIN ?= "machine:MACHINE distro:DISTRO"
addfragments ${OE_FRAGMENTS_PREFIX} OE_FRAGMENTS OE_FRAGMENTS_METADATA_VARS OE_FRAGMENTS_BUILTIN

require ${@"conf/multiconfig/${BB_CURRENT_MC}.conf" if "${BB_CURRENT_MC}" != "" else ""}
include conf/machine/${MACHINE}.conf
include conf/machine-sdk/${SDKMACHINE}.conf
require conf/cve-check-map.conf
include conf/distro/${DISTRO}.conf
include conf/distro/defaultsetup.conf
include conf/documentation.conf
include conf/licenses.conf
require conf/sanity.conf
include conf/bblock.conf
...
```

Figure 5.1: Examples of variable definitions and configuration file inclusions

The inheritance system receives deeper coverage in *Chapter 8*, while specific variables appear in *Chapter 13*.

Parsing metadata

After global configuration loads, BitBake parses every recipe that matches the patterns in `BBFILES`, the glob patterns each layer contributes to point BitBake at its recipe and append files (introduced in *Chapter 4*). For each recipe, it evaluates variable assignments, applies `inherit`

statements, and runs **anonymous Python** blocks so recipes can compute values or branch on metadata before any task runs. Concretely, the parser performs four operations:

- Expands include and require statements inline, so the named file's content is parsed as if it were written at that point. A missing include is silently skipped, while a missing require aborts parsing, as *Chapter 8* details.
- Evaluates variable assignments and overrides, including suffixes, such as `:class-target`.
- Applies `inherit` statements, pulling in `.bbclass` files that add tasks or helper functions. A recipe uses the lowercase `inherit` directive, whereas the uppercase `INHERIT` configuration variable applies a class to every recipe, as *Chapter 8* details.
- Executes anonymous Python blocks so dynamic adjustments can run during parsing.

The result is an internal build graph with task ordering and dependency links already resolved. You can write that graph out for inspection with `bitbake -g core-image-full-cmdline`, which *Chapter 2* introduced.

Dependencies

BitBake models three dependency types while scheduling tasks. Build systems that only assemble prebuilt binaries can look faster, but they gain that speed by trusting that dependencies were resolved elsewhere. BitBake instead resolves the whole graph from source, which is what keeps its builds reproducible rather than merely quick. Runtime relationships often rely on **Package Group** recipes, which bundle related packages so image recipes pull in coherent feature sets without naming every dependency individually.

BitBake recognizes three dependency types:

- **Build-time**: Declared with `DEPENDS` and resolved before compilation. Libraries, headers, and native tools are staged into the `sysroot` before the recipe builds.
- **Runtime**: Declared with `RDEPENDS` and `RRECOMMENDS` and applied on the running target. The installed package pulls in the other packages it needs once deployed.
- **Task**: Declared with `do_<task>[depends]` or `addtask` and resolved during scheduling. A named task completes before the task that depends on it starts.

Some build-time dependencies come from inheriting a class rather than naming a recipe in `DEPENDS`. A recipe whose build uses `pkg-config` to discover and link against other libraries should `inherit pkgconfig`. The class adds a build-time dependency on the `pkgconf` recipe, which provides the `pkg-config` tool, and sets `PKG_CONFIG_PATH` so `pkg-config` finds the `.pc`

files that other recipes stage into the `sysroot`. Inheriting the class, not adding `pkgconf` to `DEPENDS` by hand, is the supported mechanism.

BitBake observes dependencies in several contexts:

- A recipe referenced in another recipe's `DEPENDS`
- A recipe that belongs to a `packagegroup` (see *Chapter 7* for how recipes are grouped)
- A recipe added as a dependency by an enabled `PACKAGECONFIG` option
- A recipe whose `INITSCRIPT_PARAMS` value supplies the SysVinit runlevel ordering, paired with `INITSCRIPT_NAME` and `INITSCRIPT_PACKAGES`
- A recipe whose `PROVIDES` list contains either the literal recipe name `<recipe-name>` or another named provider that satisfies the same virtual capability

The `virtual/` namespace acts as a convention for recipes that may have more than one provider. Any recipe that depends on the Linux kernel can add `virtual/kernel` to `DEPENDS` and BitBake will locate the selected provider. When multiple providers exist, you set `PREFERRED_PROVIDER` to pick one, as shown below.

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

Machine configurations typically set the Linux kernel provider because hardware differs across machines. Additional guidance on machine configuration appears in *Chapter 12*.

Note

BitBake raises an error if it cannot satisfy a dependency because the required provider is missing.

Task dependencies govern how BitBake executes work. A recipe cannot compile until its source tree is prepared, so the `do_fetch` and `do_unpack` tasks complete before `do_compile` runs. Because every relationship is expressed as a task dependency, a recipe listed as a build-time dependency must finish its tasks before the dependent recipe begins.

Metadata records those relationships. OpenEmbedded Core classes expose common patterns so recipes can declare build-time dependencies with `DEPENDS` and runtime requirements with `RDEPENDS`. BitBake evaluates the dependency graph and schedules tasks accordingly:

- Tasks without dependencies run in parallel to maximize throughput
- Tasks with dependencies wait until every prerequisite task is complete

When multiple recipe versions satisfy the same provider dependency, BitBake selects the highest version by default. You can steer the selection with `PREFERRED_VERSION`, often used in BSP layers that require a specific bootloader release. To prevent BitBake from selecting a specific version as the highest available, set `DEFAULT_PREFERENCE` in that recipe, as demonstrated below.

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

BitBake will not choose a version solely because `DEFAULT_PREFERENCE` is set to `-1`. To select a specific revision, set `PREFERRED_VERSION`.

The variables available to control provider version selection are:

- `PREFERRED_PROVIDER`: Selects which recipe supplies a given virtual target when multiple providers exist.
- `PREFERRED_VERSION`: Pins a specific recipe version, bypassing the default highest-version rule.
- `DEFAULT_PREFERENCE`: Adjusts the relative priority of a recipe version. Setting it to `-1` keeps that version from being chosen unless explicitly requested via `PREFERRED_VERSION`.

Inspecting the dependency graph

The dependency graph BitBake builds is not only an internal structure. You can export it when a build pulls in a recipe you did not expect or when you need to justify why a component is present.

Running `bitbake -g <target>` writes two files into the current directory. `pn-buildlist` lists every recipe the target pulls in, and `task-depends.dot` describes the task-level graph in Graphviz format. Render the second file to an image with Graphviz, for example `dot -Tpng task-depends.dot -o task-depends.png`, to see how the tasks connect.

For interactive exploration, BitBake ships the Task Explorer. Running `bitbake -g <target> -u taskexp` generates the graph and opens a graphical window where you search for a recipe, expand its dependencies, and trace why each one is included without reading a large `.dot` file by hand, as shown below for `core-image-full-cmdline`.

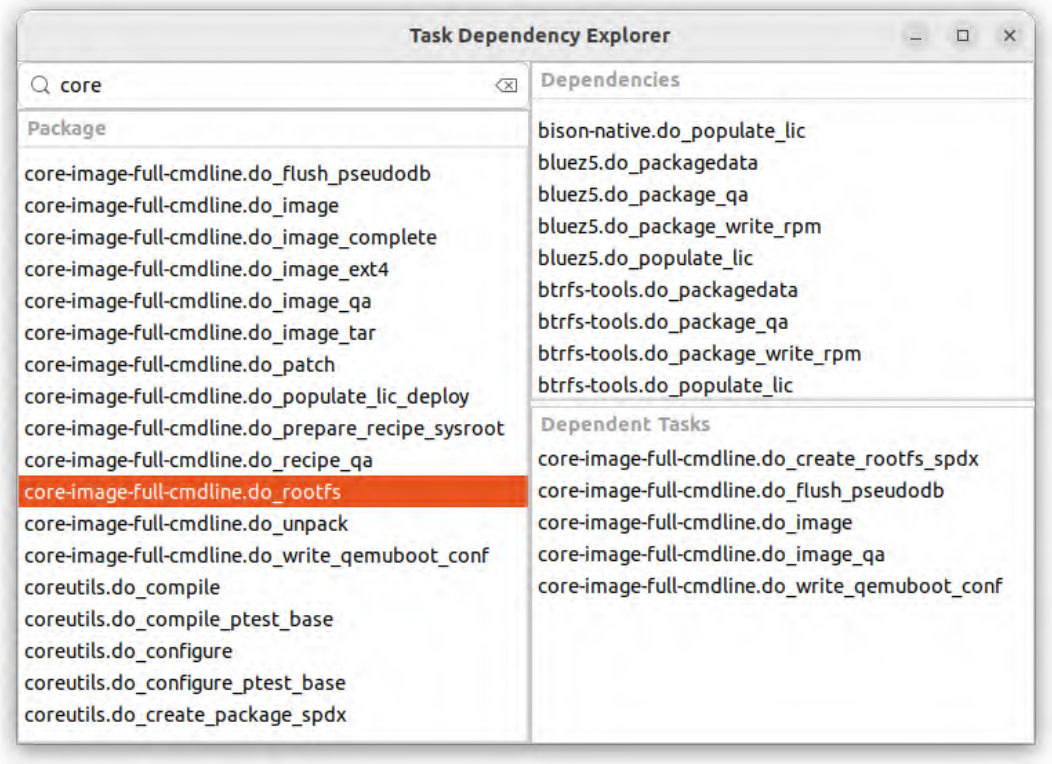


Figure 5.2: Task Dependency Explorer for core-image-full-cmdline.do_rootfs

The search box filters the package list, and selecting a task populates the Dependencies and Dependent Tasks panels so you can trace the relationship in either direction. Where no graphical display is available, `bitbake -g <target> -u taskexp_ncurses` offers the same navigation in the terminal.

These views turn the abstract graph into something you can search and follow, which is the quickest way to answer why a particular recipe is part of your build.

Fetching the source code

BitBake's **fetcher** subsystem provides a modular way to collect source code from many services. Linux-based systems often combine a Linux kernel tree with utilities such as OpenSSH or BusyBox, so recipes must handle both revision control and tarball downloads. The most commonly used URL schemes include:

- Android repo (`repo://`)

- Azure Storage (`az://`)
- ClearCase (`ccrc://`)
- FTP (`ftp://`)
- Git (`git://`)
- Git Annex (`gitannex://`)
- Git submodules (`gitsm://`)
- Google Cloud Storage (`gs://`)
- HTTP or HTTPS (`http://` or `https://`)
- Local files (`file://`)
- Mercurial (`hg://`)
- Perforce (`p4://`)
- Rust Crate (`crate://`)
- S3 (`s3://`)
- SFTP (`sftp://`)
- SSH (`ssh://`)
- Subversion (`svn://`)

BitBake chooses the correct backend based on the URL scheme and records the results in the download cache.

Remote file downloads

When BitBake runs the `do_fetch` task, it expands `SRC_URI`, downloads each referenced file, verifies integrity, and stores the artifact in `DL_DIR`. The `bzip2` recipe illustrated in the following figure shows how the version string from the recipe filename maps into the download URL.

```
SRC_URI = "https://sourceware.org/pub/${BPN}/${BPN}-${PV}.tar.gz \
           git://sourceware.org/git/bzip2-tests.git;name=bzip2-
tests;branch=master;protocol=https;destsuffix=bzip2-tests/ \
           ...
           "
SRC_URI[sha256sum] = "ab5a03176ee106d3f0fa90e381da478ddae405918153cca248e682cd0c4a2269"
```

Figure 5.3: Example of SRC_URI for the bzip2 recipe

BitBake expands the `BPN` (base package name) variable to `bzip2` and the `PV` variable to the package version (1.0.8 in this example, derived from the `bzip2_1.0.8.bb` recipe filename) to

download the file from `https://sourceware.org/pub/bzip2/bzip2-1.0.8.tar.gz`. The file is then stored in `DL_DIR`, which is the download storage directory.

The build system derives a family of name and version variables, most of them from that filename. Using `bzip2_1.0.8.bb` as the example:

- `PN`: Recipe name, derived by default from the recipe filename. Here it is `bzip2`.
- `PV`: Recipe version, derived by default from the recipe filename, here `1.0.8`.
- `PR`: Recipe revision. It defaults to `r0` and is rarely set by hand.
- `PE`: Package epoch, unset by default and treated as `0`. It outranks `PV` in version comparisons and is set explicitly only to change the normal ordering, as *Chapter 7* explains.
- `BPN`: `PN` with prefixes and suffixes such as `nativesdk-`, `-native`, and `lib32-` removed, leaving `bzip2`.
- `P`: `${PN}-${PV}`, which expands to `bzip2-1.0.8`.
- `BP`: `${BPN}-${PV}`, which expands to `bzip2-1.0.8`.

Beyond the name and version variables, every recipe works within a set of directory variables the build system defines automatically. The ones you meet most often while reading tasks and logs are:

- `DL_DIR`: Shared cache where the fetcher stores downloaded source, by default `build/downloads`
- `WORKDIR`: Per-recipe working directory that holds every task's output, at `build/tmp/work/<arch>/${PN}/${PV}`
- `UNPACKDIR`: Subdirectory of `${WORKDIR}` where `do_unpack` extracts fetched source
- `S`: Source directory the build runs from, by default `${UNPACKDIR}/${BP}`
- `B`: Build directory for generated objects, defaulting to `${S}`
- `D`: Destination where `do_install` stages files for packaging, at `${WORKDIR}/image`

Chapter 6 tours these directories on disk, and *Chapter 13* covers the installation path variables such as `bindir` and `sysconfdir`.

BitBake records a `.done` stamp once the `sha256sum` matches the value listed in the recipe. That stamp allows future builds to reuse the cached archive. Recipes may list additional checksum fields, such as `md5sum`, when the upstream provides them.

Tip

By default, `DL_DIR` points to `build/downloads`. Setting it to a shared path in `build/conf/local.conf`, for example `DL_DIR = "/opt/yocto-downloads"`, lets multiple build directories reuse cached downloads and skip repeated network fetches.

When using `bitbake-setup`, the `dl-dir` setting in the BitBake Configuration Template achieves the same result: every developer who initializes from that template uses the shared location without any manual `build/conf/local.conf` edits. *Chapter 14* covers the `dl-dir` setting in detail.

Git repositories

BitBake provides strong support for Git repositories. When `SRC_URI` contains a `git://` entry, BitBake clones the repository into `${DL_DIR}/git2/<sanitized path>`. The following figure highlights the `SRC_URI` and `SRCREV` pattern used by an `lz4` recipe in OpenEmbedded Core.

```
SRCREV = "ebb370ca83af193212df4dcbadcc5d87bc0de2f0"

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

Figure 5.4: Source code download setup for the lz4 recipe

BitBake clones the `lz4.git` repository into `${DL_DIR}/git2/github.com.lz4.lz4.git` to include the host and project in the path and avoid naming conflicts when different repositories share the same project name.

`SRCREV` pins the commit used by both `do_fetch` and `do_unpack`. You can add `branch=<branch name>` to select a branch explicitly, or `nobranch=1` when a commit lives only on a tagged reference. When using a `tag=` parameter, BitBake validates that the tag's commit SHA matches either `SRCREV` or the `rev=` URI parameter. A fuller entry combines several of these, as in `SRC_URI = "git://git.example.com/foo.git;protocol=https;branch=stable;tag=v2.4.0"`, which pins a tagged release on a named branch fetched over HTTPS. Setting `SRCREV = "${AUTOREV}"` is useful during early development when you want each build to track a moving branch, but the resulting fetch is non-reproducible and should be replaced with a pinned commit before product builds, as *Chapter 18* recommends.

The Git fetcher unpacks repositories into a directory named `git/` under `${UNPACKDIR}`, so Git-based recipes typically set `S = "${UNPACKDIR}/git"`. Tarball-based recipes can rely on the

default `S = "${UNPACKDIR}/${BP}"`, where `BP` is the base package name and version (`${BPN}-${PV}`), when the archive expands into that directory. Recipes that still set `S = "${WORKDIR}/${BP}"` keep working because `do_unpack` moves the matching subdirectory under `${WORKDIR}`.

Tip

The Yocto Project variable glossary lists every variable BitBake and OpenEmbedded Core define, with expansions and examples, at `https://docs.yoctoproject.org/wrynose/ref-manual/variables.html`.

Cached downloads persist across builds, so during development you sometimes need to clear a recipe's source and fetch it again.

Note

To clean a recipe's download and build state, run `bitbake <recipe-name> -c cleanall`. That command removes the downloaded source, the work directory, and the recipe's local Shared State Cache entries. It does not clear a shared Shared State Cache, though: when an sstate mirror is configured, such as the one the Yocto Project publishes, a rebuild can still restore the recipe's prebuilt output directly and never run `do_fetch`, so `DL_DIR` stays empty even after `cleanall`. This is expected rather than a fault, because BitBake downloads source only when neither the local cache nor the Shared State Cache can satisfy the task. To force a real fetch, remove the Shared State Cache mirror from your configuration before rebuilding.

Optimizing source code downloads

The Yocto Project includes a mirror strategy that improves reliability and supports environments with restricted internet access. BitBake follows four stages when locating source code:

1. Check `DL_DIR` for a locally cached copy of the source
2. Redirect to `PREMIRRORS` if the source is not cached, typically pointing to an internal mirror server
3. Fetch from the upstream location in `SRC_URI` when no premirror provides the file
4. Fall back to `MIRRORS`, the broader list defined in `meta/classes-global/mirrors.bbclass`

The following shows how to redirect downloads to a local server using `PREMIRRORS`.

```
PREMIRRORS:prepend = " \
  svn://.*/.*  https://mylocalserver \
  git://.*/.*  https://mylocalserver \
  gitsm://.*/.*  https://mylocalserver \
  hg://.*/.*  https://mylocalserver \
  p4://.*/.*  https://mylocalserver \
  https?://.*/.*  https://mylocalserver \
  ftp://.*/.*  https://mylocalserver \
  npm://.*/?.*  https://mylocalserver \
  s3://.*/.*  https://mylocalserver \
  crate://.*/.*  https://mylocalserver \
  gs://.*/.*  https://mylocalserver \
"
```

Since the `PREMIRRORS` pattern appears frequently, BitBake provides the `own-mirrors` class. By inheriting it and setting `SOURCE_MIRROR_URL`, you avoid repeating the full expression, as shown below.

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

If the preferred mirror lacks the requested file, BitBake continues with the defaults in `mirrors.bbclass`. The following figure provides a snapshot of those entries. Note that the class keeps the Yocto Project fallbacks in their own `YOCTO_MIRRORS` variable, which `MIRRORS` then expands, so a build that must not reach `downloads.yoctoproject.org` can empty that one variable instead of rewriting the whole list.

```
MIRRORS += "\
${YOCTO_MIRRORS} \
${DEBIAN_MIRROR}    http://snapshot.debian.org/archive/debian/20180310T215105Z/pool \
${DEBIAN_MIRROR}    http://snapshot.debian.org/archive/debian/20250101T023759Z/pool \
${DEBIAN_MIRROR}    http://snapshot.debian.org/archive/debian-archive/20120328T092752Z/debian/pool \
${GNU_MIRROR}    https://mirrors.edge.kernel.org/gnu \
${KERNELORG_MIRROR}    http://www.edge.kernel.org/pub \
${CPAN_MIRROR}  https://cpan.metacpan.org/ \
"

# Having a way to disable the Yocto Project related mirrors for YP CI purposes is valuable.
YOCTO_MIRRORS ??= "\
svn://.*/.*     https://downloads.yoctoproject.org/mirror/sources/ \
git://.*/.*     https://downloads.yoctoproject.org/mirror/sources/ \
gitsm://.*/.*   https://downloads.yoctoproject.org/mirror/sources/ \
hg://.*/.*      https://downloads.yoctoproject.org/mirror/sources/ \
https?://.*/.*  https://downloads.yoctoproject.org/mirror/sources/ \
ftp://.*/.*     https://downloads.yoctoproject.org/mirror/sources/ \
npm://.*/?.*    https://downloads.yoctoproject.org/mirror/sources/ \
https?://downloads.yoctoproject.org/releases/uninative/ https://mirrors.edge.kernel.org/yocto/uninative/ \
https?://downloads.yoctoproject.org/mirror/sources/ https://mirrors.edge.kernel.org/yocto-sources/ \
"
```

Figure 5.5: Example of the MIRRORS and YOCTO_MIRRORS variables defined in mirrors.bbclass

Mirror coverage also depends on the shape of the source itself, since Git sources need a little preparation before they can survive behind a static mirror.

Tip

Set `BB_GENERATE_MIRROR_TARBALLS = "1"` in `build/conf/local.conf` to create tarballs for Git sources inside `DL_DIR`. The tarballs land alongside other downloads under `DL_DIR` and are named after the sanitized repository path (for example, `git2_github.com.lz4.lz4.git.tar.gz`), so a static mirror only has to serve the file. That option simplifies mirroring when Git access is restricted.

While mirror support helps with reliability, some build environments require stricter control over network access.

Disabling network access

Some projects need BitBake to run without network access. Common reasons include:

- **Policy**: Company rules may require legal review before external code is used
- **Network cost**: Mobile broadband or travel scenarios make large downloads expensive

- **Download and build decoupling**: Continuous integration pipelines often separate download and build jobs, leaving build workers offline
- **Lack of network access**: Field labs or secure networks may have no external connectivity

Add the following configuration to `build/conf/local.conf` to enforce offline operation.

```
BB_NO_NETWORK = "1"
```

If any recipe still sets `SRCREV = "${AUTOREV}"`, an offline build cannot resolve the moving revision and fails. Setting `BB_SRCREV_POLICY = "cache"` tells BitBake to reuse the revision it resolved on the last networked build instead of querying upstream again. This can be confusing, though: with the cache policy in effect, or whenever a network lookup fails, `AUTOREV` no longer advances automatically, so a recipe you expect to track the latest commit quietly stays pinned to the cached revision until the next successful resolution.

Understanding BitBake tasks

BitBake executes builds through a sequence of tasks. Recipes inherit a standard set of tasks from classes, and you can extend or replace them when necessary. The following commands demonstrate how to run all tasks, execute a specific task, list available tasks, and examine the output.

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

Use `bitbake <recipe-name>` when you want BitBake to execute the complete task sequence for a recipe. The next example narrows the scope to a single task.

```
$ bitbake <recipe-name> -c compile
```

The lowercase `-c` runs the named task, reusing cached results for anything already up to date. Its uppercase counterpart, `-C`, invalidates that task's stamp first, so BitBake reruns the task and everything that depends on it before finishing the default build. Reach for `-C` when you have changed something the metadata does not capture and need to force a task to run again.

You can inspect the available tasks before choosing one to run.

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

The following figure captures typical output from `bitbake <recipe-name> -c listtasks`. When a task misbehaves, `listtasks` shows what runs and in what order, which is the entry point for the task-level investigations covered in *Chapter 10*.

```
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_create_package_spdx
do_create_package_spdx_setscene         (setscene version)
do_create_recipe_sbom
do_create_recipe_sbom_setscene          (setscene version)
(...)
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
(...)
```

Figure 5.6: List of tasks for the wget recipe

Common BitBake tasks explained

Most recipes follow a familiar workflow:

- `do_fetch`: Retrieves the required source, whether it is remote or local.
- `do_unpack`: Unpacks archives or checks out the requested revision into `${UNPACKDIR}`, then moves subdirectories matching `${S}` into `${WORKDIR}` to maintain compatibility with existing recipes.
- `do_patch`: Applies patches listed in `SRC_URI` before configuration begins.
- `do_configure`: Runs upstream configuration, often through classes such as `autotools`, `cmake`, or `meson`.
- `do_compile`: Builds the software for the target, the build machine, or the SDK.
- `do_install`: Stages artifacts in `${D}` so they can be packaged.
- `do_package`: Splits staged files into packages, including debug symbols and documentation. Packaging specifics appear in *Chapter 7*.

Environment variables vary per task. Classes such as `meson` expose `MESON_BUILDTYPE`, `MESON_TARGET`, and `EXTRA_OEMESON` so you can tailor commands without rewriting entire tasks.

Hash equivalence

Hash Equivalence is enabled by default. The `BB_SIGNATURE_HANDLER` defaults to `OEEquivHash`, and `BB_HASHSERVE = "auto"` instructs BitBake to start a local hash-equivalence server automatically. Hash Equivalence lets BitBake reuse Shared State Cache artifacts even when the recipe's source signature changes, provided the resulting output is functionally identical. The result is faster rebuilds and more aggressive Shared State Cache reuse without explicit operator setup. Poky ships the `sstate-mirror-cdn` configuration fragment, which sets `BB_HASHSERVE_UPSTREAM = "wss://hashserv.yoctoproject.org/ws"` together with a matching Shared State Cache mirror. The `local.conf.sample` template lists it as a commented-out `OE_FRAGMENTS += "core/yocto/sstate-mirror-cdn"` line. Enable that fragment so the local server can query the project's public hash server for additional matches before scheduling a task. *Chapter 14* covers fragment management in detail. The *Hash Equivalence* section of the *Yocto Project Overview and Concepts Manual* at `https://docs.yoctoproject.org/wrynose/overview-manual/concepts.html#hash-equivalence` covers the server design and configuration options in depth.

Summary

You now know how BitBake builds its execution graph, how task dependencies describe both recipe and task ordering, and how OpenEmbedded Core metadata exposes build-time and runtime dependencies. We also examined the fetchers that download source code, the mirror controls that keep downloads reproducible, and the standard tasks that move software from source archives into packaged output.

The next chapter tours the build directory so you can map those tasks to the artifacts BitBake stores under `build/tmp/`.

6

Detailing the Temporary Build Directory

The Yocto Project 6.0 (Wrynose) build generates a temporary workspace under `build/tmp` that records every BitBake task outcome. This chapter maps the directory tree systematically, explains what each subdirectory holds, and shows how to locate task logs, intermediate outputs, and deployed artifacts after a build completes.

The goal is to give you a reliable mental model of `build/tmp` so you can navigate it confidently and retrieve exactly the information a given build step produced.

That mental model pays off most clearly when something goes wrong. A developer who knows where BitBake writes each task's log and environment file can identify the source of a failure in minutes. A developer who does not may spend hours bisecting the wrong layer, escalating an opaque error to a teammate, or triggering a full rebuild to surface information that was already present in `build/tmp/work`.

We will cover the following main topics:

- Detailing the build directory
- Constructing the build directory
- Exploring the temporary build directory
- Understanding the work directory
- Understanding the sysroot directories

Technical requirements

This chapter assumes a build host configured as described in *Chapter 2*, including the required host packages, a BitBake Setup workspace, and at least one completed image build. The `build/tmp` directory must be populated before you can follow the directory inspection steps in this chapter.

If you have not yet run a build, complete the `core-image-full-cmdline` build for `qemux86-64` from *Chapter 2* and return here once `build/tmp` exists on your host.

Detailing the build directory

The build directory is the central hub for build metadata and outputs. It stores configuration files, downloaded source archives, the Shared State Cache (`sstate-cache`) that lets future builds restore completed task outputs whenever inputs match, and the temporary workspace where BitBake assembles every artifact. Its main subdirectories are as follows:

- `conf`: Stores configuration files that steer BitBake. Core files include `build/conf/local.conf`, `build/conf/bblayers.conf`, and `build/conf/toolcfg.conf`. The `bitbake-setup` tool populates `build/conf/bblayers.conf` with the layer paths registered in the BitBake Setup, as described in *Chapter 14*. The `build/conf/toolcfg.conf` file is the BitBake Configuration Instance created from a BitBake Configuration Template and managed by `bitbake-config-build`. Do not edit it manually.
- `downloads`: Holds artifacts fetched by BitBake. BitBake reuses this cache across builds, so network fetches only happen when new revisions are required. The `downloads` directory location is controlled by the `DL_DIR` variable discussed in *Chapter 5*. Note that after the `poky-with-sstate` build from *Chapter 2*, this directory may be empty or absent, because tasks restored from the Shared State Cache never run `do_fetch`.
- `sstate-cache`: Contains Shared State Cache artifacts generated by tasks such as `do_compile` and `do_install`. BitBake can restore these archives during future builds, which shortens rebuild times. Additional details appear in *Chapter 7*.
- `tmp`: Provides the temporary workspace for every build operation. The remainder of this chapter focuses on this directory, which BitBake calls `TMPDIR`. Its location defaults to `build/tmp` but can be relocated by setting `TMPDIR` in `build/conf/local.conf`, for example to place it on a faster disk.

Constructing the build directory

Earlier chapters showed how metadata from OpenEmbedded Core and additional layers steers image creation, and *Chapter 5* walked through the standard task sequence from `do_fetch` to `do_package`. This chapter revisits that sequence from a different angle, tracking which part of the build directory each task populates:

- **Fetching**: BitBake resolves source URIs defined by recipes. It checks `DL_DIR` for cached sources before reaching remote mirrors. When new content is required, it records the downloads for reuse.
- **Source preparation**: The `do_unpack` and `do_patch` tasks prepare the working tree inside `build/tmp/work`, and the recipe's `S` variable points at the resulting source tree. Tarballs extract into a `<name>-<version>` directory, matching the default `S` of `${UNPACKDIR}/${BP}`, while Git recipes check out into a `git` directory and set `S` to that path, which then holds the checked-out revision and any applied patches.
- **Configuration and building**: Task `do_configure` runs the recipe's configuration step, invoking `./configure` for autotools recipes or `cmake` for CMake recipes with the environment exported by BitBake. Task `do_compile` performs the compilation step, producing binaries and libraries in the work directory.
- **Installing**: Task `do_install` copies staged outputs into `${WORKDIR}/image`, which is the value of the `D` variable. This step uses `DESTDIR` to isolate the install tree from the host system.
- **Wrapping the sysroot**: The `do_populate_sysroot` task copies headers, libraries, `pkg-config` files, and other development assets that dependent recipes need into the shared sysroot component area under `build/tmp/sysroots-components`. BitBake later assembles those components into each recipe's `recipe-sysroot` and `recipe-sysroot-native` when it runs `do_prepare_recipe_sysroot`.
- **Creating packages**: Task `do_package` analyzes the staged install tree and splits it into per-package payloads. The companion tasks `do_package_write_ipk`, `do_package_write_rpm`, and `do_package_write_deb` then write the package archives

into `build/tmp/deploy/ipk`, `build/tmp/deploy/rpm`, or `build/tmp/deploy/deb`, depending on the value of `PACKAGE_CLASSES`.

- **Quality Assurance (QA) checks**: BitBake runs the `do_package_qa` task to detect missing dependencies, wrong permissions, and other issues before the artifacts reach the `deploy` directory.

Exploring the temporary build directory

The `build/tmp` directory comes alive as soon as the first task starts. It is often the first place to review when a Yocto Project 6.0 build fails because logs and intermediate results are stored here.

The following illustration provides an overview of the contents of the `build/tmp` directory.



Figure 6.1: Overview of build/tmp directory structure

The key directories within the `build/tmp` directory hold everything from intermediate build state to deliverables such as **Software Development Kit (SDK)** installers produced for cross-compilation work. They are described as follows:

- `deploy`: Contains build outputs, including images, binary packages, license manifests, and SDK installers. The directory mirrors the artifact types enabled by `IMAGE_FSTYPES` and `SDKMACHINE`. *Chapter 9* returns to the `build/tmp/deploy/sdk` branch when it generates and installs SDKs. License manifests in particular are examined in *Chapter 15*.

- `sysroots-components`: Stores the sysroot contents that the `do_prepare_recipe_sysroot` task links or copies into each recipe-specific sysroot for the entries listed in `DEPENDS`. The build system populates these components from the Shared State Cache, and each component directory records what the producing recipe staged.
- `sysroots-uninative`: Provides the relocatable GNU C Library used by the build system's native tools. It improves Shared State Cache reuse across different host distributions, which is critical when teams mix workstation releases. The `uninative` class is enabled by default, so this directory appears on every build.
- `work`: Holds the working source code, task configurations, execution logs, and packaged outputs generated by each recipe during the build process.
- `pkgdata`: Stores Package Information that `oe-pkgdata-util` and other tooling read when inspecting built packages. Treat this as read-only metadata unless you are debugging Package Information generation.
- `work-shared`: Shares unpacked source trees across recipes that need a common base. Recipes for the Linux kernel, such as `linux-yocto`, and toolchain recipes like `gcc` rely on this directory.

The remaining entries are mostly internal BitBake bookkeeping, but a few are worth knowing. The `log` directory collects `cooker` logs from BitBake itself, and `buildstats` records per-task timing and resource data useful for profiling build performance. The `saved_tmpdir` file stores the absolute path this `TMPDIR` was created at. The rest, such as `abi_version`, `cache`, `hosttools`, `sstate-control`, and `stamps`, rarely need day-to-day attention.

Understanding the work directory

The `build/tmp/work` directory is organized by package architecture and machine. For instance, when working with the `qemux86-64` machine on a glibc-enabled host, there are typically four key directories, as shown below.

```
build/tmp/work
├── all-poky-linux
├── qemux86_64-poky-linux
├── x86_64-linux
└── x86-64-v3-poky-linux
```

Figure 6.2: Overview of the build/tmp/work directory

The preceding diagram shows the possible directories under `build/tmp/work` for an `x86-64` host and a `qemux86-64` target. These directories vary based on architecture and machine, as follows:

- `all-poky-linux`: Contains build directories for architecture-independent (allarch) packages, such as scripts or Python utilities, that are reused unchanged across every target architecture, whether arm64, riscv64, or the qemux86-64 target used here
- `x86-64-v3-poky-linux`: Contains build directories for packages that target the `x86-64` v3 microarchitecture level, which requires AVX2, FMA, and similar extensions found on modern `x86-64` processors
- `qemux86_64-poky-linux`: Contains build directories for packages tailored to the qemux86-64 machine, such as those carrying machine-specific patches or settings that make their output valid for that machine only.
- `x86_64-linux`: Contains build directories for `-native` recipes that generate tools running on the build host, such as cross-compilers and other host-side native tools

Viewed from the recipe side, the `PACKAGE_ARCH` variable decides which of these directories receives a recipe's build tree. Its value comes from the classes a recipe inherits and from any explicit assignment in the recipe itself, and it resolves as follows:

- **Native and cross recipes**: Land in `x86_64-linux`, the directory named for the build-host architecture, because they produce tools that run on the host, such as `-native` utilities and the `gcc-cross` toolchain.
- **Architecture-independent target recipes**: Land in `all-poky-linux`. These recipes inherit the `allarch` class, which marks their output as valid for every architecture, so a single build serves all machines.
- **Tune-specific target recipes**: Land in the directory whose name is the value of `TUNE_PKGARCH`, here `x86-64-v3-poky-linux`.
- **Machine-specific target recipes**: Land in the directory whose name is the value of `MACHINE_ARCH`, here `qemux86_64-poky-linux`.

This modular structure enables BitBake to assemble system images and packages for multiple machines within a single build directory while avoiding conflicts, even when those machines target very different architectures, such as a `raspberrypi4` (arm64) board or a `qemuriscv64` virtual machine alongside the `qemux86-64` target used in this example. Because the layout is driven entirely by package architecture, the structure you learn here carries over unchanged to every board you add later: a new machine adds its own architecture directories alongside the

existing ones without reorganizing them, so you learn this model once and apply it to every future target.

The `build/tmp/work` directory is highly useful for troubleshooting build failures or other issues. Its contents are organized into subdirectories, as shown below.



Figure 6.3: Structure of subdirectories in the build/tmp/work directory

Some of the directories under the tree shown in the preceding diagram are as follows:

- `<source-tree>`: The unpacked source lives wherever the recipe's `S` variable points. `S` defaults to `${UNPACKDIR}/${BP}`, the `<name>-<version>` directory that tarball archives extract into, so tarball recipes use it unchanged. Recipes fetched with Git instead check out into a `git` directory and set `S` to that path.
- `image`: Holds the unmodified install tree produced by `do_install` (the recipe's `DESTDIR`), with binaries and libraries exactly as the build installed them, before any packaging transformations.
- `package`: Holds the same content after `do_package` processing, where binaries and libraries are stripped and their debug symbols split into separate `-dbg` content, ready to be divided into subpackages.
- `packages-split`: Holds that processed content split into one subdirectory per generated package, such as the main package, `-dev`, `-dbg`, and `-doc`, matching how the files are finally assigned.
- `temp`: Stores BitBake task scripts, run logs, and the environment file that records the variables seen by each task.

Disk usage grows quickly in this tree, especially when the project builds multiple variants of the Linux kernel or SDKs.

Tip

Prune intermediate output after tasks complete by adding `INHERIT += "rm_work"` to `build/conf/local.conf` when disk space is tight.

Use this knob selectively during active debugging, because removing work directories after successful tasks prevents you from re-running them with preserved artifacts.

The structure of the `work` directory is consistent across all architectures. For each recipe, a directory named after the recipe is created. For instance, in a machine-specific `work` directory, the `base-files` recipe will have its own directory, which contains all necessary build files for that recipe, as shown below.

```
build/tmp/work/qemux86_64-poky-linux/base-files/3.0.14/
├── deploy-rpms
├── license-destdir
├── pkgdata-pdata-input
├── spdx
├── sstate-install-create_package_spdx
├── sstate-install-create_recipe_spdx
├── sstate-install-create_spdx
├── sstate-install-packagedata
├── sstate-install-package_qa
├── sstate-install-package_write_rpm
├── sstate-install-populate_lic
├── sstate-install-populate_sysroot
├── sstate-install-recipe_qa
├── sysroot-destdir
└── temp
```

Figure 6.4: Structure of the build/tmp/work/qemux86_64-poky-linux/base-files/3.0.14/ directory

The `base-files` recipe is a good example of a machine-specific recipe. It provides the foundational files for the root filesystem, including `/etc/fstab` and `/etc/hostname`, and differs per machine.

Note

The build system uses several directories shown in the preceding diagram that are not detailed here. These directories are generally not required for day-to-day engineering work and only matter when implementing new BitBake classes or experimenting with advanced customization.

The `work` directory is an essential resource for debugging, providing detailed information on each step of the build process. Its use in debugging is explored further in *Chapter 10*.

Understanding the sysroot directories

The sysroot directories are crucial in the Yocto Project because they provide an isolated environment for each recipe. This isolation keeps builds reproducible and prevents contamination from host packages.

After building the `ptest-runner` recipe (version 2.5.1), two sysroot directories are created: `recipe-sysroot` and `recipe-sysroot-native`. The `recipe-sysroot` directory contains libraries and headers needed for the target system, while `recipe-sysroot-native` includes the build dependencies required on the host system, such as privilege-emulation and packaging utilities. Each sysroot also contains a subdirectory named `sysroot-providers`, which lists the packages installed in that sysroot. The following diagram shows the `recipe-sysroot` directory, highlighting the `sysroot-providers` subdirectory contents.



Figure 6.5: Structure of the recipe-sysroot directory for the ptest-runner recipe

The `recipe-sysroot-native` directory contains the build dependencies used on the host system during the build process, such as compilers, linkers, and other tools. In contrast, the `recipe-sysroot` directory holds the libraries and headers necessary for the target code. The following diagram illustrates the contents of the `recipe-sysroot-native` directory.

```
./x86-64-v3-poky-linux/ptest-runner/2.5.1/recipe-sysroot-native/
├── bin
├── etc
├── installeddeps
├── lib-shadow-deps
├── sbin
├── sysroot-providers
│   ├── pseudo-native
│   ├── shadow-native
│   ├── sqlite3-native
│   └── virtual_fakeroot-native
├── usr
└── var
```

Figure 6.6: Structure of the recipe-sysroot-native directory for the ptest-runner recipe

The `sysroots-components` directory keeps a manifest of these contents per recipe. When diagnosing missing headers or link failures, compare the manifests for the affected recipes and inspect the corresponding `sysroot-providers` entries to confirm whether the dependency was staged correctly.

Summary

This chapter mapped the temporary build directory created by the Yocto Project 6.0 and linked each subdirectory to the BitBake tasks that populate it.

The next chapter examines how the Yocto Project packaging system works, including package feeds, the **package revision (PR)** service, and the tooling that supports product maintenance.

7

Assimilating Packaging Support

This chapter revisits the core packaging flow in the Yocto Project: supported package formats, installation script hooks, the Shared State Cache (laid out in *Chapter 6*), versioning, runtime dependency metadata, root filesystem assembly, and remote package feeds.

Packaging sits at the center of every build in the Yocto Project 6.0 (Wrynose). BitBake stages packages early in the build pipeline so images, SDKs, and package feeds are produced from consistent, versioned artifacts.

Beyond binaries, every recipe stamps license metadata onto each package it produces through the `LICENSE` and `LIC_FILES_CHKSUM` variables. The build system carries that metadata into image manifests, deploy-time license archives, and SBOM documents, which is the foundation *Chapter 15* builds on when discussing copyleft obligations and audit artifacts.

We will cover the following main topics:

- Supported package formats
- Running code during package installation
- Understanding the Shared State Cache
- Explaining package versioning
- Specifying runtime package dependencies
- Using packages to generate a root filesystem image
- Package feeds

Technical requirements

This chapter requires a build host configured as described in *Chapter 2*, including all host packages, disk space, and network access listed there. You also need at least one successful build already completed against that workspace, so that the Shared State Cache is populated

and you can rebuild individual recipes and inspect their package outputs without starting from scratch.

No additional hardware or external accounts are needed. All packaging workflows covered here use tools shipped with the build environment.

Supported package formats

A recipe and a package are not the same thing: the recipe is the set of build instructions BitBake parses, while the packages are the installable artifacts it produces from that recipe. A single recipe routinely yields several packages, splitting its output into a main package plus companions such as `-dev`, `-dbg`, and `-doc`, as *Chapter 6* shows on disk. Supporting multiple formats keeps the build system useful across varied deployment workflows, and BitBake abstracts the differences between formats so recipes remain portable regardless of the selected packaging backend.

List of supported package formats

A **Package Format** is the on-disk packaging convention BitBake uses to ship the files a recipe installs, together with their runtime metadata. OpenEmbedded Core supports three such formats:

- **Red Hat Package Manager (RPM)**: The Poky default. It pairs with the `dnf` tool, offering dependency tracking and integrates well with signing workflows used by enterprise distributions such as Red Hat Enterprise Linux, Fedora, SUSE, and CentOS Stream.
- **Debian Package Manager (DEB)**: Familiar to teams that work with Debian, Ubuntu, or Linux Mint because the same `dpkg` and APT tools apply.
- **Itsy Package Management System (IPK)**: A lightweight format maintained by the OpenWrt and Yocto Project communities. The `opkg` package manager handles these archives and fits memory-constrained devices.

Choosing a package format

Support for package formats is provided through the `package_rpm`, `package_deb`, and `package_ipk` classes. Select one or more formats with the `PACKAGE_CLASSES` variable in `build/conf/local.conf`, as shown below.

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

BitBake can load multiple packaging backends from a single configuration line.

You must select at least one format: `do_rootfs` assembles the image through a package manager and reads the first `PACKAGE_CLASSES` entry to decide which one, so there is no no-backend option for an image build. Whether the finished image then ships a runtime package manager is a separate choice, controlled by the `package-management` image feature covered later in this chapter.

> **Tip**
>
> The first entry in `PACKAGE_CLASSES` provides the archives BitBake uses while generating images, while any additional entries create feeds without affecting the image build.

Choose the format that matches memory requirements and tooling expectations. Poky defaults to RPM with DNF. When no distribution is selected, the OpenEmbedded Core fallback (`DISTRO = "nodistro"`) defaults to IPK with `opkg` to minimize footprint. Teams steeped in Debian-based workflows often prefer DEB so they can reuse APT tooling.

The main trade-offs to weigh are:

- **On-device footprint**: `opkg` is the smallest package manager and the best fit for storage-constrained or FLASH-based devices. `dnf` is the heaviest because it pulls in a Python runtime, and `apt` sits in between.
- **Build and install time**: IPK is the quickest to build and install and carries the least metadata overhead. RPM favors correctness and richer metadata over speed, so it builds and installs more slowly. DEB falls between the two.
- **Dependency and security features**: RPM carries the richest dependency metadata and signing support, which suits medium and large systems and enterprise workflows. IPK keeps metadata minimal, which is enough for most embedded images but offers fewer features.
- **Team tooling**: Match the format to the tools your team already uses. DEB reuses the familiar `dpkg` and APT workflow, RPM aligns with `dnf`-based distributions, and IPK is standard in the OpenWrt, OpenEmbedded, and Yocto Project communities.

Running code during package installation

Packages can include scripts that run during installation and removal:

- `preinst`: Runs before unpacking the package. Use it to stop services or back up persistent data.

- `postinst`: Runs after files land on disk during `do_rootfs`, which makes it ideal for registering services or finishing configuration steps.
- `prerm`: Runs before removing package files so you can gracefully stop services.
- `postrm`: Runs after package removal to clean residual state or update symlinks.

Installation scripts commonly manage system services. For example, recipes can install `SystemD` unit files or `SysVinitinit` scripts. Use the `systemd` class to install and enable unit files. Use the `update-rc.d` class to install SysVinit scripts. Both classes provide default helper scripts that you can customize in the recipe.

The `postinst` script runs during root filesystem creation. If it exits successfully, the package is marked as installed. Add the script to the recipe using a `pkg_postinst` implementation, for example:

```
pkg_postinst:${PN} () {
    # Insert command here
}
```

Some scripts cannot run on the build host or under QEMU because they access live hardware or require writable configuration storage on the device. Use the `pkg_postinst_ontarget` variant shown below for such scripts. It runs only on the target and therefore requires writable storage on the device, so it does not work with fully read-only images.

```
pkg_postinst_ontarget:${PN} () {
    # Insert command here
}
```

Use this approach to defer hardware-specific setup until the package runs on the target device, where writable storage and direct hardware access are available.

Tip

Use the `PN` variable instead of spelling the package name so recipes remain robust when you refactor outputs.

When `IMAGE_FEATURES` includes `read-only-rootfs` (whether set directly in `build/conf/local.conf`, contributed by an image recipe, or pulled in by a BitBake Configuration Fragment that appends to `IMAGE_FEATURES`), every `pkg_postinst` script must complete successfully at image-build time. Recipes that rely on `pkg_postinst_ontarget` produce an explicit error

during `do_rootfs` because deferring logic to the device contradicts a read-only deployment. Move the work back into a build-time `pkg_postinst`, or drop `read-only-rootfs` for that image before re-running the build.

Tip

Fully read-only images are not always practical. Projects that need writable state often layer a tmpfs or overlay filesystem. The *overlayfs-bbclass* entry at `https://docs.yoctoproject.org/wrynose/ref-manual/classes.html#overlayfs` and the *overlayfs-etc-bbclass* entry at `https://docs.yoctoproject.org/wrynose/ref-manual/classes.html#overlayfs-etc` describe two reference approaches.

Always gate filesystem paths with the `D` variable. `D` points to the root of the staging area during `do_rootfs` and expands to an empty string at runtime, so you can use the same logic in both environments, as shown below.

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

In the above code, the `touch` command uses `D` so the recipe writes to the staging area during the build and to the real filesystem only when the script runs on the target.

Understanding the Shared State Cache

BitBake reruns a task and the downstream tasks that depend on it whenever its signature changes, which keeps results correct at the cost of time. The Shared State Cache (`sstate-cache`) mitigates that cost by storing the output of completed tasks under `SSTATE_DIR` along with the signature BitBake computed for each task.

Before running a task, BitBake checks whether a matching cache entry exists and, if so, restores the packaged output directly. The *Shared State Cache* section at `https://docs.yoctoproject.org/wrynose/overview-manual/concepts.html#shared-state-cache` walks through the hashing model in depth and explains how the Shared State Cache avoids unnecessary rebuilds.

Hash Equivalence extends this model so the Shared State Cache stays useful even when a recipe's input signature drifts but its output is unchanged. When the equivalence server confirms that a different signature produces an artifact already stored in the cache, BitBake reuses that artifact instead of invalidating it.

The *Hash Equivalence* section at `https://docs.yoctoproject.org/wrynose/overview-manual/concepts.html#hash-equivalence` explains the underlying mechanism, and *Chapter 5* covers the BitBake-level configuration.

Clean the cache periodically because it grows with every build. The build system supports two workflows: pruning the entire cache to reclaim disk space, and clearing entries for a single recipe to force a clean rebuild. The *Maintain the Shared State Cache* guideline in *Chapter 18* frames when to schedule that cleanup as a maintenance discipline, while the commands here cover how to carry it out.

Over time, the cache accumulates duplicate entries from earlier builds that waste disk space. Use `sstate-cache-management.py`, from the scripts shipped with OpenEmbedded Core, to remove those duplicates and reduce the overall cache footprint, as shown below.

```
$ sstate-cache-management.py --remove-duplicated --cache-dir=build/sstate-cache
```

When a specific recipe's cached output becomes invalid (for example, after a configuration change that the signature hash does not detect), or when you need to force a clean rebuild for one component, use the `cleansstate` task to discard only that recipe's entries, as shown below.

```
$ bitbake <recipe-name> -c cleansstate
```

The `cleansstate` task discards every cached output associated with the recipe and forces BitBake to rerun its build chain on the next invocation.

Tip

Remove `build/tmp` while keeping `build/sstate-cache` to keep most cached tasks for the next build, or remove both directories when you need a true from-scratch rebuild. If a public Shared State Cache mirror is configured (as with the `poky-with-sstate` setup from *Chapter 2*), disable it as well, or BitBake restores prebuilt task output from the mirror and the rebuild is not truly from scratch.

Explaining package versioning

Package versioning differentiates outputs created from the same recipe over time and influences the signatures BitBake computes.

By default, BitBake selects the newest recipe version (`PV`) unless a layer explicitly pins a different one. For example, a `myrecipe_1.1.bb` file supersedes `myrecipe_1.0.bb` unless the build configuration sets `PREFERRED_VERSION_myrecipe`.

The final version string shown below is composed of `PE` (package epoch), `PV` (package version), and `PR` (package revision).

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

`PE` is unset by default, which version comparisons treat as epoch `0`. It only changes when you need to reset ordering, such as switching from date-based versions to semantic versions. In that case, setting `PE = "1"` makes a new `PE:PV` string like `1:1.0` rank higher than `0:20220101`.

`PR` defaults to `r0`. Increment it when you modify packaging metadata but not the source so downstream package feeds still observe a monotonic version increase. BitBake primarily relies on task checksums and only falls back to manual `PR` bumps when nothing else changes. The *Package feeds* section later in this chapter describes the PR Service, which automates these increments for network-published feeds.

Specifying runtime package dependencies

When the package manager installs software on the target, it reads runtime metadata embedded in each package to decide what else must be present, which components conflict, and which packages provide the same capability. BitBake gives you a consistent way to define that runtime metadata across RPM, DEB, and IPK outputs:

- `RDEPENDS`: Packages required at runtime for the current package to function.
- `RRECOMMENDS`: Packages installed alongside the current one when they are available, used for soft dependencies that improve functionality without being strictly required. Suppress an individual recommendation with `BAD_RECOMMENDATIONS`.
- `RSUGGESTS`: Packages that are entirely optional suggestions. Unlike `RRECOMMENDS`, they are never installed automatically. The metadata simply records them so users or tooling can discover related packages.

- `RPROVIDES`: Virtual names that the package satisfies so other recipes can depend on an interface rather than a specific implementation.
- `RCONFLICTS`: Packages that cannot coexist with the current package.
- `RREPLACES`: Packages that this package supersedes, often after renaming or consolidation.

The `python3-dbus` recipe inside `meta/recipes-devtools/python/` uses `RDEPENDS` to declare its runtime dependencies, as shown below.

```
SUMMARY = "Python bindings for the DBus inter-process communication system"
SECTION = "devel/python"
HOMEPAGE = "http://www.freedesktop.org/Software/dbus"
LICENSE = "MIT"
LIC_FILES_CHKSUM = "file://COPYING;md5=97f58951300aa52a9f9e3a62bd5c846c"
DEPENDS = "expat dbus glib-2.0 virtual/libintl"

SRC_URI = "http://dbus.freedesktop.org/releases/dbus-python/dbus-python-${PV}.tar.xz"

SRC_URI[sha256sum] = "c36b28f10ffcc8f1f798aca973bcc132f91f33eb9b6b8904381b4077766043d5"

S = "${UNPACKDIR}/dbus-python-${PV}"

inherit setuptools3-base meson pkgconfig

# requires dbus-run-session
EXTRA_OEMESON += "-Dtests=false"

RDEPENDS:${PN} = "python3-io python3-logging python3-stringold python3-threading python3-xml"

FILES:${PN}-dev += "${libdir}/pkgconfig"

BBCLASSEXTEND = "native nativesdk"
```

Figure 7.1: Runtime dependency metadata in python3-dbus

The preceding figure shows how `python3-dbus` declares its Python module dependencies so BitBake can add them automatically when you install the package.

Using packages to generate a root filesystem image

The build system generates root filesystem images by staging packages inside a temporary directory, applying image features, and then wrapping the result into final filesystem artifacts that may also include the Linux kernel, device trees, and bootloader components.

The process involves four broad steps:

1. Generating the `rootfs` directory

2. Creating required support files (configuration snippets, `init` scripts, hooks)
3. Wrapping the filesystem into the requested format, such as a partitioned disk image
4. Compressing the image if the selected format calls for it

The `do_rootfs` task drives these steps. It starts by building the package list from several inputs:

- Packages you specify in `IMAGE_INSTALL`
- Extra packages introduced by `IMAGE_FEATURES`
- Optional adjustments where image classes add debug symbols or development headers

The build system collapses these inputs into `PACKAGE_INSTALL`, which is the final list `do_rootfs` hands to the package manager. `PACKAGE_EXCLUDE` prunes specific packages from that list, although dependencies can still pull them back if they remain required elsewhere.

With the list finalized, `do_rootfs` installs the packages, applies the selected tweaks, and executes any non-target `pkg_postinst` scripts so the target boots cleanly.

The `IMAGE_FSTYPES` variable controls how BitBake wraps the `rootfs` directory, whether as an `ext4` filesystem, a UBIFS volume, or a multi-partition disk image.

After `do_rootfs` finishes, BitBake places the image, manifest, and related artifacts under `build/tmp/deploy/images/${MACHINE}`, the value of the `DEPLOY_DIR_IMAGE` variable, and the deploy tree that *Chapter 6* documents in detail. *Chapter 12* expands on how to customize these steps and how to combine `IMAGE_FEATURES` with layer-specific policies.

Package feeds

A **Package Feed** is a network-accessible repository of built packages that target devices or other builds can fetch from. Package feeds fuel both image creation and SDK assembly because `do_rootfs` installs from a local feed populated during the build.

Publishing that feed over the network enables additional workflows:

- Test updated applications without reflashing a device
- Install optional packages on a deployed system
- Deliver field updates between full image releases

Maintaining a reliable feed requires monotonically increasing package revisions. BitBake offers the **PR Service** to automate those `PR` increments whenever a task checksum changes.

The PR Service appends a numeric suffix to the existing `PR` (for example, `r34`, `r34.1`, `r34.2`) so downstream package managers always see a newer version.

Note

You can still set PR manually, but doing so should be rare because it bypasses the automation that keeps feeds consistent.

Enable the PR Service locally by setting `PRSERV_HOST` in `build/conf/local.conf`, as illustrated here:

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

This setup works when a single workstation performs every build because BitBake automatically starts and stops the PR Service for each invocation.

For multi-machine environments, run a persistent PR Service instance with `bitbake-prserv` and point every build host to it:

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

Each participating build host must set `PRSERV_HOST` to the shared server's IP address and port so all builds draw from the same revision history.

Using package feeds

Every feed workflow has two sides:

- Server: Hosts the package archives and index files
- Client: Fetches archives and installs them on the target image

Build whichever recipes you want to expose, then create a package index so the server advertises the latest versions, as shown below.

```
$ bitbake package-index
```

BitBake stores packages under `build/tmp/deploy` in a subdirectory that matches the package format (for example, `build/tmp/deploy/rpm`).

Tip

Run `bitbake package-index` after every build that produces new packages so the feed metadata stays synchronized.

Serve the feed over HTTP, NFS, or any protocol your deployment flow supports. For quick local tests, the Python `http.server` module is enough, as shown below.

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

The package manager, running inside the device, must know how to reach the server and must include package management utilities. Add two settings to `EXTRA_IMAGE_FEATURES` and `PACKAGE_FEED_URIS`, as shown below.

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

`PACKAGE_FEED_URIS` carries the protocol and host portion of the feed URL. Two companion variables let you split the rest of the path so a single server can host feeds for several builds:

- `PACKAGE_FEED_BASE_PATHS`: One or more path prefixes appended after each URI in `PACKAGE_FEED_URIS`, typically used to namespace builds, branches, or releases.
- `PACKAGE_FEED_ARCHS`: The architecture subdirectories the client should append. When unset, the build system uses the architecture list it would otherwise discover from the feed layout.

Pre-configuring all three variables at build time produces a target image that contacts the right feed without any post-install editing on the device.

Further details about `IMAGE_FEATURES` and `EXTRA_IMAGE_FEATURES` appear in *Chapter 12*.

Note

Adding `package-management` increases image size because it brings in DNF, APT, or opkg, plus their metadata caches.

With those settings in place, the image can fetch repository data during boot. For RPM-based systems, run the following command on the target to refresh the metadata.

```
$ dnf check-update
```

Use `dnf search <package>` to browse available packages and `dnf install <package>` to add them to a running system.

The command lineup differs per package format, as summarized in *Table 7.1*.

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

Table 7.1: Package manager command reference

Package feeds shine during development because you can update a deployed system without reflashing an image.

> **Note**
>
> **Warning!**
>
> Treat in-field package upgrades cautiously. Test every dependency chain before releasing feeds to production devices. Many teams still prefer to distribute full images for major updates and reserve feeds for interim fixes.

Managing a feed at scale requires solving dependency, hosting, and security challenges that exceed this chapter. Consult the Yocto Project documentation for deeper operational guidance.

Summary

This chapter revisited packaging fundamentals for the Yocto Project 6.0, covering format selection, installation scripts, Shared State Cache workflows, versioning, dependency metadata, root filesystem assembly, and package feeds.

The next chapter explores the BitBake metadata syntax, operators, and variable manipulation techniques you will use to keep these packaging workflows under control.

8

Understanding BitBake Metadata

Previous chapters built confidence with the Yocto Project by creating images, publishing packages, and validating package feeds on Yocto Project 6.0 (Wrynose). This chapter examines the operators that shape BitBake variables, explains how overrides activate context-specific metadata, and shows how classes and tasks share logic between recipes.

Together these mechanisms give you the control needed to write customizations that fit naturally into the metadata model rather than fight against it, and that control becomes essential for the more targeted work in later chapters.

That fit is not just a style concern. An engineer who reaches for the wrong operator, or who misunderstands when an override applies, produces recipes that build and install correctly in a first pass but silently drop a patch, append the wrong value, or evaluate a variable too early. The resulting image may work on the bench and still fail to reproduce on a different machine or a different day. Debugging that kind of failure consumes far more time than learning the metadata model up front.

We will cover the following main topics:

- Variable assignment and expansion operators
- List and string appending, prepending, and removal
- Conditional metadata and the override mechanism
- File Inclusion with include and require
- Python and shell metadata functions
- Class files and metadata inheritance

Technical requirements

This chapter requires a build host configured according to *Chapter 2*, including the host packages, disk space, and a working BitBake Setup workspace that can reach the network to fetch source archives.

You will also need a recipe file where you can write small variable assignments and run `bitbake-getvar` against them to observe how BitBake parses and evaluates operators. Any layer you have already created is suitable for that purpose. If you have not created one yet, *Chapter 12* walks through the steps.

Variable assignment and expansion operators

BitBake metadata drives every decision the build system makes. As explained in *Chapter 4*, the metadata stack spans configuration files, classes, and recipes, each of which can embed shell or Python statements that BitBake later turns into task implementations. Understanding how BitBake parses and caches those statements is the best way to keep builds predictable.

Tip

Reading this metadata in VS Code makes the operators easier to follow: the Yocto Project BitBake extension highlights the shell and Python embedded in each recipe and shows a variable's documented meaning on hover. *Chapter 3* covers the extension setup.

Metadata grows quickly, so you need fast ways to inspect the parsed **datastore**. The `bitbake-getvar` helper prints any variable after BitBake has combined configuration files, classes, and recipes. Activate the build environment, then query a recipe variable:

```
$ bitbake-getvar -r procps SRC_URI
```

BitBake loads the parsed cache before printing the value, as shown below.

```
$ bitbake-getvar -r procps SRC_URI
Loading cache: 100% |################################################################| Time: 0:00:01
Loaded 4136 entries from dependency cache.
```

Figure 8.1: Inspecting a recipe variable with bitbake-getvar

The `bitbake-getvar` helper parses the BitBake metadata stack, applies overrides and queued edits, and prints the effective value of the requested variable. It queries the parsed datastore only and does not execute build tasks, so the value it reports is the same fully expanded content BitBake will pass to tasks later in the build. Add the `--value` flag to print just the expansion without the metadata header:

```
$ bitbake-getvar -r procps SRC_URI --value
```

In this case, the expansion of `SRC_URI` resolves to one Git URI followed by three local files, as shown below, so any recipe logic that references `SRC_URI` will see exactly that four-entry list.

```
$ bitbake-getvar -r procps SRC_URI --value
git://gitlab.com/procps-ng/procps.git;protocol=https;branch=master;tag=v4.0.6 \
    file://sysctl.conf \
    file://0001-tests-Disable-twice-total-pmap-X-tests.patch \
    file://run-ptest
```

Figure 8.2: bitbake-getvar output for SRC_URI in procps

The *BitBake User Manual* at `https://docs.yoctoproject.org/bitbake/2.18/` calls out additional query arguments along with debugging techniques.

Basic variable assignment

BitBake variables follow a familiar assignment syntax.

```
FOO = "bar"
```

With no further manipulation, any later use of `FOO` expands to the literal string `bar`. The `=` operator records the latest value, so later statements that assign the same variable replace the previous content.

Tip

The BitBake parser still accepts assignments without spaces around the = operator, but it warns when they are missing, reporting that the line has a lack of whitespace around the assignment. Beyond readability, the spaces remove a genuine ambiguity, because a trailing ` in a variable name can otherwise be hard to tell apart from the += operator. The book and OpenEmbedded Core consistently surround the operator with single spaces.

Variable expansion

Variables expand lazily unless stated otherwise, which keeps the datastore flexible.

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

When BitBake expands `B`, it substitutes the current value of `A`, so `B` evaluates to `before-aValue-after`.

Deferred evaluation

BitBake only expands a variable when a task consumes it, which keeps late overrides effective.

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

Changing `A` before BitBake uses `B` changes the final expansion, so when a task finally reads `B`, it evaluates to `before-aNewValue-after`.

Immediate variable expansion

The `:=` operator forces immediate expansion, which is helpful when you need to bake a value before later assignments can change it.

```
A = "aValue"
B := "${A}-after"
A = "newValue"
C = "${A}"
```

`B` freezes immediately as `aValue-after` because of `:=`, while `C` expands later and therefore picks up the updated `newValue`. Reserve `:=` for variables that must not reflect later edits.

Assigning a value with ?=

Use `?=` when you want to provide a default without blocking later assignments.

```
A ?= "defaultValue"
```

Because no other assignment precedes it, `A` expands to `defaultValue` when first referenced. Only the first `?=` wins, so multiple candidates should appear in descending priority.

```
A ?= "initialValue"
A ?= "ignoredValue"
```

After parsing both statements, BitBake keeps the first assignment, so `A` still expands to `initialValue`. If a later line assigns the variable outright, that new value replaces whatever `?=` produced.

```
A = "finalValue"
A ?= "defaultValue"
```

Here `A` expands to `finalValue` because the assignment executes after the conditional default and therefore overwrites it. Use `=` when the recipe must take control regardless of earlier defaults.

Assigning a default with ??=

BitBake treats `??=` as a last-chance default that only runs when the variable is still empty after parsing the file so far.

```
A ??= "firstValue"
A ??= "secondValue"
```

Because no later line assigns `A`, the last `??=` encountered in the file wins and the variable expands to `secondValue`. Chain `??=` statements from least to most specific so the bottom of the block reflects the preferred fallback. This is the operator BitBake Configuration Fragments rely on to set a default that an explicit selection can replace, as *Chapter 14* describes when it pins `MACHINE` without conflicting with fragment-driven setups.

```
A ??= "firstValue"
A ??= "secondValue"
A ?= "thirdValue"
A ??= "fourthValue"
```

When metadata reads `A`, it sees `thirdValue` because the parser reaches the `?=` line while `A` is still empty, stores `thirdValue`, and the final `??=` no longer applies. Use `?=` whenever you want a fallback that should take priority over any remaining `??=` defaults later in the file.

List and string appending, prepending, and removal

List appending and prepending

List style operators separate items with spaces and are useful for variables such as `IMAGE_INSTALL`.

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

After parsing, `A` becomes `originalValue appendedValue`, separated by a single space. Use `+=` when order matters but you still want readability.

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

Here `A` expands to `prependedValue originalValue`, again separated by one space. Pick `=+` when the new token must appear before existing entries, such as when prepending a dependency.

String appending and prepending

String operators concatenate text without spaces.

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

The concatenation of the strings results in `A` reading back as `originalValueAppendedValue`. This operator is common when building filenames or suffixes.

```
A = "OriginalValue"
A =. "PrependedValue"
```

Similarly, `A` becomes `PrependedValueOriginalValue` with no separator. Make sure the surrounding strings already contain any separators you need.

Using :append and :prepend

The colon variants queue edits until the parser finishes parsing the metadata.

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

After all parsing completes, `A` resolves to `originalValueAppendedValue`, even if another file reassigns `A` in between. Queued appends happen after the parser evaluates assignment, so they respect the latest value.

```
A = "OriginalValue"
A:prepend = "PrependedValue"
```

Once queued operations apply, `A` reads as `PrependedValueOriginalValue`. Colon operators are commonly used when a class or include file needs to tweak a value defined elsewhere.

Comparing :append with .=

Queued edits interact with immediate operators in different ways.

```
A = "value"
A:append = "AppendedValue"
B .= "value"
B = "newValue"
```

Here `A` still becomes `valueAppendedValue` after parsing completes, whereas `B` temporarily expands to `value` because of `.=` but is subsequently replaced outright with `newValue`. `:append` respects changes that happen later in parsing, while `.=` executes right away, so choose the operator that matches your timing needs.

Removing list items

Use `:remove` to delete space-separated values after all queued edits have executed.

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

`:remove` evaluates once all `:append` and `:prepend` requests finish, so you can trust that it sees the full list. In this example, `A` ultimately resolves to `value1 value3` because the entry matching `value2` is dropped after parsing is finished.

Conditional metadata and the override mechanism

Conditional metadata sets

Overrides apply metadata only when the condition appears in `OVERRIDES`.

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

BitBake therefore evaluates overrides in the order `linux`, then `arm`, then `mymachine`, so any variable with those suffixes expands using the most specific available value. The most specific override wins, so list entries from general to specific.

```
OVERRIDES = "linux:arm:mymachine"
A = "defaultValue"
A:linux = "linuxSpecificValue"
A:other = "otherSpecificValue"
```

Given the override list, `A` expands to `linuxSpecificValue` because the `linux` suffix is present and more specific than the base assignment. Overrides keep configuration files concise by removing branching logic from recipes.

Conditional appending

Condition specific edits can also use `:append` and `:prepend`.

```
OVERRIDES = "arm"
A = "value"
A:arm:append = "ArmValue"
```

Because `arm` is active, the final expansion of `A` becomes `valueArmValue`. Only the overrides that match the current machine or distribution are applied.

```
OVERRIDES = "linux"
B = "Value"
B:linux:prepend = "linuxSpecific"
```

With the `linux` override active, BitBake emits `linuxSpecificValue` whenever `B` is referenced.

Note

Warning!

Stack overrides carefully to avoid conflicting edits in downstream layers.

Working through a combined example

Operators become easier to reason about once you trace several of them acting on the same variable. Work out what `A` expands to before reading the explanation.

```
OVERRIDES = "arm"
A = "base"
A ?= "ignored"
A:append = " extra"
A:arm:append = " arm"
```

Parse the lines in order. `A = "base"` sets the value, so the following `?=` is ignored because `A` already holds a value. The two `:append` operations are queued and applied after parsing finishes: the unconditional one adds `" extra"`, and the `arm`-qualified one adds `" arm"` because `arm` is present in `OVERRIDES`. The final expansion is therefore `base extra arm`. If you drop `arm` from `OVERRIDES`, the last append no longer applies and `A` expands to `base extra`.

Beyond variable manipulation, BitBake provides mechanisms for sharing metadata across multiple files, which reduces duplication and keeps configuration consistent.

File inclusion with include and require

`include` and `require` statements pull shared snippets into the current file.

```
include optional.inc
```

If `optional.inc` is located on `BBPATH`, BitBake parses and expands it inline so any variables or functions it defines contribute to the including file. If the file is not found, BitBake continues parsing without error and no statements are added. Use `include` for optional helpers that may or may not exist in a given layer combination.

A related directive, `include_all`, behaves like `include` but pulls in *every* matching file across all enabled layers instead of stopping at the first hit on `BBPATH`. You rarely need it. Reserve it for cases where you want to aggregate the same fragment contributed by several layers, such as collecting per-layer maintainer or configuration data. Like `include`, it stays quiet when no file matches.

```
require mandatory.inc
```

Because `require` is strict, BitBake cannot finish parsing or produce final variable values until `mandatory.inc` is found and expanded. `require` halts parsing when the target file is absent, which protects mandatory logic.

Tip

Keep reusable statements in `.inc` files so recipes can share `SRC_URI` fragments, functions, or `PACKAGECONFIG` defaults without duplication.

When static assignment operators are not flexible enough, BitBake allows embedding Python expressions directly into variables or defining complete functions in either shell or Python.

Python and shell metadata functions

Python variable expansion

Python expressions can generate values dynamically during parsing.

```
A = "${@'1.2.3'.replace('.', '_')}"
```

The Python expression runs during parsing and its return value becomes the final expansion of `A`. Here, `'1.2.3'.replace('.', '_')` yields `1_2_3`, so every reference to `A` observes that computed string rather than the literal expression. Python expansion is powerful, so keep expressions small and document why they are needed.

Defining executable metadata

Recipes and classes should declare tasks in POSIX shell or Python.

```
do_example() {
    echo "This is a sample shell task."
}
addtask do_example
```

When BitBake schedules `do_example`, the task runs a shell command that prints `This is a sample shell task.` The shell task can read BitBake variables via variable expansion or the task environment, but any modifications to shell variables or the shell environment do not update the BitBake datastore.

Shell tasks should stick with POSIX syntax so they run correctly under Dash.

```
python do_example_py() {
    d.setVar('EXAMPLE_VAR', 'This is a sample Python task.')
    bb.debug(1, "EXAMPLE_VAR set to: %s" % d.getVar('EXAMPLE_VAR'))
}
addtask do_example_py
```

At execution time, this task sets `EXAMPLE_VAR` to `This is a sample Python task.` and logs that value with `bb.debug`. Python tasks must use the `python` keyword and the `d` object to access the datastore. Unlike a shell task, a Python task can modify the BitBake datastore directly.

Defining python functions in the global namespace

Global helper functions keep recipes clean by hiding repeated logic.

```
def example_function(d):
    some_condition = d.getVar('SOMECONDITION')
    if some_condition == "1":
        return "dependencyWithCondition"
    return "dependencyWithoutCondition"

SOMECONDITION = "1"
MY_VAR = "${@example_function(d)}"
```

With `SOMECONDITION` set to `"1"`, the helper returns `dependencyWithCondition`, so `MY_VAR` expands to that string everywhere in the recipe. Returning a value from the helper lets you reuse it in multiple assignments.

Anonymous Python functions

Anonymous Python blocks run once at parse time for every recipe that contains them, before any task executes. They are the right place to compute defaults or mutate the datastore based on values resolved earlier in the parse.

```
python __anonymous() {
    if d.getVar('SOMECONDITION') == "1":
        d.setVar('MY_VAR', "dependencyWithCondition")
    else:
        d.setVar('MY_VAR', "dependencyWithoutCondition")
}
```

Each anonymous Python block, spelled either `python __anonymous() { … }` or, more commonly in OpenEmbedded Core, `python () { … }`, receives the recipe's `d` datastore as `d` and calls `d.getVar` and `d.setVar` to inspect or update variables. Because the block fires during parsing, any change it makes is visible to the tasks BitBake schedules later in the build. Reserve anonymous Python for adjustments that genuinely depend on parse-time state. BitBake reparses recipes whenever their metadata changes, and every anonymous block runs on each of those parses, so expensive work here is paid across the entire recipe set rather than once at build time and stretches out the parsing phase before any task can start.

Combining Python and shell

Parse-time Python and build-time shell often work together: Python resolves a value while BitBake parses the recipe, stores it in the datastore, and a shell task reads it later through ordinary variable expansion.

```
python __anonymous() {
    version = d.getVar('PV')
    d.setVar('VERSION_TAG', version.replace('.', '_'))
}

do_report() {
    bbwarn "Building release ${VERSION_TAG}"
}
addtask do_report
```

While BitBake parses the recipe, the anonymous block reads `PV`, converts the dots in the version to underscores, and stores the result in `VERSION_TAG`. When `do_report` runs later, the shell reads `${VERSION_TAG}` the same way it reads any variable, so a recipe with `PV = "1.2.3"` prints `Building release 1_2_3`. The datastore is the hand-off point: Python writes during parsing and shell reads during the task, and neither needs to know how the other produced or consumed the value.

Class files and metadata inheritance

Understanding class files in BitBake

Classes bundle reusable tasks, helper functions, and default variables so related recipes stay consistent. BitBake applies them through the `inherit` statement that the parser evaluates as it reads each recipe, the same parsing pass described in *Chapter 5*. BitBake can apply a class to every recipe through configuration or only to the recipes that explicitly request it, which gives you a clean way to layer policy without cloning metadata.

Global classes

Global classes take effect everywhere because they are listed in the `INHERIT` variable inside configuration files such as `build/conf/local.conf`. When a global class exports a helper, every recipe inherits that function automatically and therefore expands shared variables the same way. Most layers keep these classes under `classes-global/` and use them for infrastructure features such as sanity checks, license auditing, or workspace cleanup that must run for every recipe.

Recipe-specific classes

Recipe-focused classes reside in `classes/` or `classes-recipe/` and only apply when a recipe runs `inherit <class-name>`. This keeps specialized logic close to the software that needs it: `cmake.bbclass` supplies the standard configure/compile tasks for CMake-based code, `kernel.bbclass` wires up Linux kernel builds, and `image.bbclass` manages filesystem construction. Because the recipe chooses whether to inherit the class, unrelated components remain untouched.

Tuning workspace cleanup classes

Some classes such as `rm_work.bbclass` are designed to be added to `INHERIT` so they affect the whole build, with per-recipe exemptions handled through variables like `RM_WORK_EXCLUDE` rather than by inheriting the class from individual recipes. Apply them globally when every recipe benefits, then list the exceptions in the supporting variable so the policy stays declarative.

Summary

You now have a reference for BitBake assignment operators, override processing, include handling, and the task definitions that tie shell and Python logic back into the build.

The next chapter explains how the Yocto Project extends these concepts to create external toolchains and target specific root filesystems so teams can iterate faster on application code.

Part 3

Advanced Concepts

This part moves from understanding the build to shaping it for a product. It starts with the tooling you hand to application developers: what a software development kit is in this context, how to produce an image suited to on-device development, and how the Standard SDK and the Extensible SDK differ in practice, including the devtool workflow for creating, building, and deploying recipes. Debugging follows, separating metadata problems from application problems and working through each, from tracking what actually landed in an image or package to logging, variable inspection, the Development Shell, GDB, and profiling and tracing.

The middle of the part is about making the metadata yours. You will read and add external layers and find them through the OpenEmbedded Layer Index, then write your own: layer scaffolding and metadata, a custom image, package recipes both by hand and through devtool, a machine definition, and a custom distribution, along with the variable scoping rules that decide which setting wins. You will also customize recipes that other people maintain, extending tasks, tuning build options, applying patches, adding files, and working with Kconfig configuration fragments, all without forking upstream.

The part then covers the configuration-management workflow introduced in Yocto Project 6.0 (Wrynose), composing BitBake Configuration Fragments with `bitbake-config-build`, authoring custom fragments and templates, and synchronizing reproducible setups across a team. It closes with license compliance: auditing licenses, providing source code and compilation scripts, shipping license text inside the image, declaring the terms of your own proprietary recipes, generating a Software Bill of Materials, and using that SBOM to drive vulnerability scanning.

This part of the book includes the following chapters:

- *Chapter 9*, *Developing with the Yocto Project*
- *Chapter 10*, *Debugging with the Yocto Project*
- *Chapter 11*, *Exploring External Layers*
- *Chapter 12*, *Creating Custom Layers*
- *Chapter 13*, *Customizing Existing Recipes*
- *Chapter 14*, *Managing BitBake Configuration*
- *Chapter 15*, *License Compliance and Software Bill of Materials*

9

Developing with the Yocto Project

The earlier chapters built images with BitBake and explained how the build system assembles a target root filesystem from metadata. This chapter shifts to the developer's daily workflow: generating standard and Extensible SDKs, using **devtool** to iterate on recipes outside the main build tree, and keeping on-device development images synchronized with the project configuration.

The goal is to give application teams everything they need to write, cross-compile, and test code against the correct target sysroot without running a full BitBake build for every change.

Without a proper SDK workflow, every developer on a team either waits for a complete rebuild to get an updated sysroot or carries a private, untracked copy that drifts from the project baseline. Both paths create integration problems that grow harder to fix the longer they go unchecked. The techniques in this chapter break that cycle by separating the application development loop from the image assembly loop, so engineers can move quickly without sacrificing reproducibility.

We will cover the following main topics:

- What is a Software Development Kit?
- Generating an image for on-device development
- Understanding the types of cross-development SDKs
- Using the Standard SDK
- Using the Extensible SDK

Technical requirements

This chapter requires a build host already set up per *Chapter 2*, with a BitBake Setup workspace, a working Yocto Project 6.0 (Wrynose) build, and the host packages the Yocto Project specifies. The authoritative host-package list is at `https://docs.yoctoproject.org/wrynose/ref-manual/system-requirements.html`.

You will also need:

- A development workstation capable of running the installed SDK. The Standard SDK installer can extract several gigabytes to the host filesystem. The Extensible SDK requires additional space for the embedded metadata snapshot and Shared State Cache.
- A target device (real hardware or a QEMU machine) that can boot a Yocto Project image. Several exercises use `bitbake core-image-full-cmdline -c populate_sdk` and related commands, so the build machine must be able to produce and run target images.

What is a Software Development Kit?

Cross-development remains the norm for embedded Linux because the build host and target hardware usually expose different CPU architectures, peripherals, and ABI policies. Toolchains therefore run on a capable workstation while emitting binaries for the target processor.

A **Software Development Kit** (**SDK**) bundles the artifacts required to build and debug without copying the entire Yocto Project workspace onto every developer laptop. Typical contents include:

- Compilers
- Linkers
- Debuggers
- External libraries
- Header files
- Profilers
- Supporting utilities

Even when the build host and target share a CPU family, building against a staged target sysroot is still a form of cross-compilation, because the binaries link against headers and libraries that match the target distribution rather than the host. Native SDKs, by contrast, target on-device workflows where the host and target are the same system.

Generating an image for on-device development

Some targets provide enough CPU, RAM, and storage to support local development. In those cases, the image that boots on the device needs the following components:

- Header files and libraries required for compilation and linking
- Toolchain elements such as compilers, linkers, debuggers, and `pkg-config` data

To build an image that supports on-device development, add the required features to your image recipe using `IMAGE_FEATURES` as shown below:

```
IMAGE_FEATURES += "dbg-pkgs dev-pkgs tools-debug tools-sdk"
IMAGE_INSTALL += "packagegroup-core-buildessential"
```

When experimenting locally without committing to your recipe, add the same features through `EXTRA_IMAGE_FEATURES` in `build/conf/local.conf` as shown below:

```
EXTRA_IMAGE_FEATURES += "dbg-pkgs dev-pkgs tools-debug tools-sdk"
```

With these features in place, the booted image carries a complete native toolchain, so developers can compile, link, and debug directly on the device, exactly as they would on a standard Linux workstation and without any cross-compilation or host SDK.

Running a development-ready image reduces the time spent copying binaries from the host, but it can drift as soon as individual developers install extra packages directly on the target. Track those changes in metadata (see *Chapter 12*) to keep the image reproducible.

Understanding the types of cross-development SDKs

The Yocto Project ships two SDK styles that address different needs. The **Standard SDK** targets application, bootloader, and Linux kernel work, while the **Extensible SDK (eSDK)** adds the metadata snapshot and `devtool` workflow on top of it. The list below summarizes what each one provides:

- **Standard SDK**: Focuses on application, bootloader, and Linux kernel development. It ships the target sysroot, cross-toolchain, and debugging helpers tied to a specific image.
- **Extensible SDK (eSDK)**: Adds the Yocto Project metadata snapshot, the Shared State Cache, and the `devtool` workflow so developers can create or modify recipes, rebuild images, and publish packages without a full clone of the main workspace.

Both options produce self-extracting installers that drop into any supported Linux host and match the `MACHINE` settings used during creation.

Using the Standard SDK

Many teams reach for the Extensible SDK and overlook the Standard SDK, even when the project does not need recipe authoring inside the SDK. *Chapter 18* argues for keeping the Standard SDK as the default option whenever the development loop is application-level, and the rest of this section walks through how to generate it.

The Standard SDK can be derived from a generic `meta-toolchain` or from a product-specific image. Generate the `meta-toolchain` when you only need compilers, debuggers, and a minimal sysroot containing the C library and its headers:

```
$ bitbake meta-toolchain
```

Most teams prefer an image-specific SDK because the sysroot matches deployed packages. Trigger it with `populate_sdk` on the target image:

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

Installer artifacts land under `build/tmp/deploy/sdk/`, which is one branch of the deploy tree mapped out in *Chapter 6*. A typical directory looks like this:

```
build/tmp/deploy/sdk/
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.host.manifest
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.sh
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.spdx.json
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.target.manifest
└── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.testdata.json
```

Figure 9.1: Output files produced by populate_sdk

Installing the Standard SDK

Each SDK exports as a self-extracting shell script. Run it from a terminal and point it at a writable destination:

```
$ cd build/tmp/deploy/sdk/
$ ./poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-
toolchain-6.0.1.sh
```

The installer prompts for a target directory before extracting, as shown in the following figure.

```
$ cd build/tmp/deploy/sdk/
$ ./poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.sh
Poky (Yocto Project Reference Distro) SDK installer version 6.0.1
==========================================================================
Enter target directory for SDK (default: /opt/poky/6.0.1):
You are about to install the SDK to "/opt/poky/6.0.1". Proceed [Y/n]? y
[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/6.0.1/environment-setup-x86-64-v3-poky-linux
```

Figure 9.2: Installing the Standard SDK with the self-extracting script

Pair the installer with a custom image from *Chapter 12* so the SDK reflects the packages, `MACHINE` settings, and tool versions baked into your product.

Building applications with the Standard SDK

After installation, source the environment script to load cross-compilation variables, then build host-style projects with the provided `${CC}`:

```
$ . /opt/poky/6.0.1/environment-setup-x86-64-v3-poky-linux
$ ${CC} hello-world.c -o hello-world
$ file hello-world
```

Sourcing the script redefines `CC`, `CXX`, `LD`, and the related build variables to target the cross-toolchain for the rest of that shell session.

Note

Source the SDK environment script in a fresh shell. Sourcing it in a terminal that already has the BitBake build environment active (from `oe-init-build-env`) mixes the two toolchain configurations and leads to unpredictable builds.

The resulting binary targets the device architecture, as shown in the following figure.

```
$ . /opt/poky/6.0.1/environment-setup-x86-64-v3-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]=b0a...9fb, for GNU/Linux 5.15.0,
with debug_info, not stripped
```

Figure 9.3: Compiling a C application with the Standard SDK

Building the Linux kernel with the Standard SDK

Linux kernel maintenance often happens alongside application work. Use the SDK to cross-compile the Linux kernel source tree so the configuration matches production:

```
$ . /home/user/poky_sdk/environment-setup-x86-64-v3-poky-linux
$ unset LDFLAGS
$ make defconfig
$ make bzImage
```

The command list keeps the Linux kernel tree clean and relies on the SDK's cross-toolchain.

Note

Run `unset LDFLAGS` before calling `make` so the SDK's linker flags do not conflict with the Linux kernel build system.

The Standard SDK consolidates the compilers, sysroots, and debug tools required for application, bootloader, and Linux kernel changes while staying aligned with the reference image.

Using the Extensible SDK

The **Extensible SDK** (**eSDK**) builds on the standard SDK by bundling metadata, Shared State Cache entries, and task helpers that accelerate iteration. It adds the following capabilities:

- Generate or extend recipes from upstream source trees
- Build individual recipes or entire images with `devtool`
- Install additional packages into the SDK sysroot as needs evolve
- Deploy artifacts straight to QEMU or physical targets for quick validation

Generating the Extensible SDK

Trigger the eSDK build with `populate_sdk_ext` so the output includes the additional metadata snapshot:

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

You can verify the generated installers and manifests under `build/tmp/deploy/sdk/`:

```
build/tmp/deploy/sdk/
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.host.manifest
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.sh
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.spdx.json
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.target.manifest
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-6.0.1.testdata.json
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-ext-6.0.1.host.manifest
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-ext-6.0.1.sh
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-ext-6.0.1.spdx.json
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-ext-6.0.1.target.manifest
├── poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-ext-6.0.1.testdata.json
├── x86_64-buildtools-nativesdk-standalone-6.0.1.host.manifest
├── x86_64-buildtools-nativesdk-standalone-6.0.1.sh
├── x86_64-buildtools-nativesdk-standalone-6.0.1.spdx.json
├── x86_64-buildtools-nativesdk-standalone-6.0.1.target.manifest
└── x86_64-buildtools-nativesdk-standalone-6.0.1.testdata.json
```

Figure 9.4: Files generated by populate_sdk_ext

Installing the Extensible SDK

Install the eSDK the same way as the Standard SDK, but choose a destination that leaves room for the Shared State Cache:

```
$ cd build/tmp/deploy/sdk/
$ ./poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-
ext-6.0.1.sh
```

The installer confirms the destination before extracting, as shown in the following figure.

```
$ cd build/tmp/deploy/sdk/
$ ./poky-glibc-x86_64-core-image-full-cmdline-x86-64-v3-qemux86-64-toolchain-ext-6.0.1.sh
Poky (Yocto Project Reference Distro) Extensible SDK installer version 6.0.1
==========================================================================
Enter target directory for SDK (default: ~/poky_sdk):
You are about to install the SDK to "~/poky_sdk". Proceed [Y/n]? y
Extracting SDK.....................................................................................done
Setting it up...
Extracting buildtools...
Preparing build system...
Loading cache: 100% |#####################################################################| Time: 0:00:00
Parsing recipes: 100% |###################################################################| Time: 0:00:29
Initialising tasks: 100% |################################################################| Time: 0:05:47
Checking sstate mirror object availability: 100% |########################################| Time: 0:01:04
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)... (22:47:57.721098)
Loading cache: 100% |#####################################################################| Time: 0:00:00
Parsing recipes: 100% |###################################################################| Time: 0:00:39
Initialising tasks: 100% |################################################################| Time: 0:00:01
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)... (22:48:46.584429)
Loading cache: 100% |#####################################################################| Time: 0:00:00
Initialising tasks: 100% |################################################################| Time: 0:00:02
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-x86-64-v3-poky-linux
```

Figure 9.5: Installing the Extensible SDK

Activate the eSDK by sourcing the environment script it installs, then confirm that `devtool` is available:

```
$ . /home/user/poky_sdk/environment-setup-x86-64-v3-poky-linux
```

The script reports that the environment is ready and points to `devtool`.

```
$ . /home/user/poky_sdk/environment-setup-x86-64-v3-poky-linux
SDK environment now set up; additionally you may now run devtool to perform development tasks.
Run devtool --help for further details.
```

Figure 9.6: Activating the Extensible SDK environment

The session captured in the preceding figure illustrates the exported environment variables and `devtool` version check. As with the Standard SDK, source this script in a fresh shell. The eSDK prints a warning if BitBake is already configured in the shell you use.

Structure of the Extensible SDK

The Extensible SDK consolidates everything needed for reproducible development:

- Essential binaries required to drive BitBake, `devtool`, and accompanying helpers
- Standard SDK components, including the cross-toolchain and sysroots
- Shared state cache objects that accelerate incremental builds
- Yocto Project metadata, configuration fragments, and layer priorities captured at creation time

This self-contained layout mirrors the build environment so recipe edits, image builds, and deployments stay consistent with the main project.

Building an image using devtool

With the environment active, you can launch image builds directly from the eSDK without re-entering the main workspace:

```
$ devtool build-image core-image-full-cmdline
```

`devtool` drives the full image build from the SDK, as shown in the following figure.

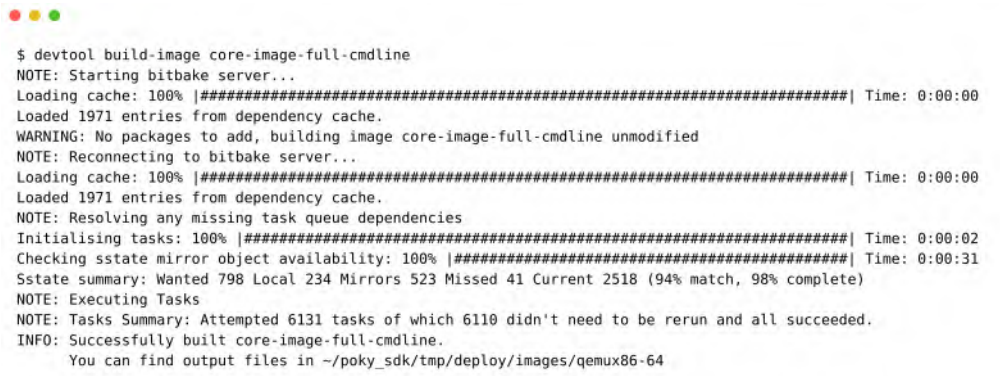


Figure 9.7: Building core-image-full-cmdline with devtool

The resulting images appear under `~/poky_sdk/tmp/deploy/images/<machine>/`, ready for testing or deployment.

Running an image on QEMU

You can boot the generated image with QEMU by invoking the `runqemu` helper exposed by the SDK environment:

```
$ runqemu snapshot
```

`runqemu` selects the image and starts the emulator, as shown in the following figure.

```
$ runqemu snapshot
runqemu - INFO - In SDK environment, setting STAGING_BINDIR_NATIVE from OECORE_NATIVE_SYSROOT (~/poky
          _sdk/tmp/sysroots/x86_64)
runqemu - INFO - Decompressing ~/poky_sdk/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86
          -64.rootfs-20260616230023.ext4.zst to ~/poky_sdk/tmp/deploy/images/qemux86-64/core-image-full
          -cmdline-qemux86-64.rootfs-20260616230023.ext4
~/poky_sdk/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs-20260616230023.ext4
          .zst: 377088000 bytes
runqemu - INFO - Continuing with the following parameters:
KERNEL: [~/poky_sdk/tmp/deploy/images/qemux86-64/bzImage]
MACHINE: [qemux86-64]
FSTYPE: [ext4]
ROOTFS: [~/poky_sdk/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs-2026061623
        0023.ext4]
SNAPSHOT: [Enabled. Changes on rootfs won't be kept after QEMU shutdown.]
CONFFILE: [~/poky_sdk/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs-20260616
          230023.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
net.ifnames=0
runqemu - INFO - Using VNC server at unix:/tmp/qemu-tap-locks/tap0.vnc-socket for graphical output
runqemu - INFO - Running ~/poky_sdk/tmp/sysroots/x86_64/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=~/poky
           _sdk/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs-20260616230023.ext4,
           if=virtio,format=raw -usb -device usb-tablet -usb -device usb-kbd   -cpu Skylake-Client -machine q35,
           i8042=off -smp 4 -m 256 -snapshot -serial mon:vc -serial null -vnc unix:/tmp/qemu-tap-locks/tap0.vnc-
           socket  -kernel ~/poky_sdk/tmp/deploy/images/qemux86-64/bzImage -append 'root=/dev/vda rw
           ip=192.168.7.2::192.168.7.1:255.255.255.0::eth0:off:8.8.8.8 net.ifnames=0 oprofile.timer=1
tsc=reliable
           no_timer_check rcupdate.rcu_expedited=1 swiotlb=0 '

runqemu - INFO - Host uptime: 300774.06
```

Figure 9.8: Running the image with runqemu snapshot

Once QEMU finishes booting, the sequence unfolds in three stages: the Linux kernel boot log scrolls across the terminal, the Yocto Project bootsplash appears on the emulated display, and finally the login prompt appears as shown below.



Figure 9.9: QEMU login prompt

At the `qemux86-64 login:` prompt, log in with the user `root` and no password.

Note

Logging in as `root` with no password requires the image to carry the `empty-root-password` feature, which clears the root account's password. A freshly initialized configuration does not grant it, so the login prompt rejects every attempt. The `core/yocto/root-login-with-empty-password` fragment that *Chapter 2* enabled for this workspace supplies it, along with `allow-empty-password` and `allow-root-login`. If you disabled it while working through *Chapter 14*, re-enable it with `bitbake-config-build enable-fragment core/yocto/root-login-with-empty-password` and rebuild the image before booting.

Creating a recipe from an external git repository

`devtool` can generate a workspace recipe from a remote Git repository so you can test changes without editing the main metadata tree. The public `bbexample` project at `https://github.com/OSSystems/bbexample` illustrates the workflow:

```
$ devtool add bbexample https://github.com/OSSystems/bbexample.git --srcbranch
master
```

`devtool` fetches the repository and generates a provisional recipe, as shown in the following figure.

```
$ devtool add bbexample https://github.com/OSSystems/bbexample.git --srcbranch master
NOTE: Starting bitbake server...
NOTE: Reconnecting to bitbake server...
INFO: Fetching git://github.com/OSSystems/bbexample.git;protocol=https;branch=master...
Loading cache: 100% |################################################################################| Time: 0:00:00
Loaded 0 entries from dependency cache.
Parsing recipes: 100% |##############################################################################| Time: 0:00:24
Parsing of 951 .bb files complete (0 cached, 951 parsed). 1972 targets, 47 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Sstate summary: Wanted 1 Local 0 Mirrors 0 Missed 1 Current 0 (0% match, 0% complete)########  | ETA:  0:00:00
Initialising tasks: 100% |###########################################################################| Time: 0:00:05
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3 tasks of which 0 didn't need to be rerun and all succeeded.
INFO: Using default source tree path ~/poky_sdk/workspace/sources/bbexample
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)... (23:34:02.272885)
NOTE: Reconnecting to bitbake server...
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)... (23:34:02.272885)
NOTE: Retrying server connection (#1)... (23:34:02.272885)
NOTE: Starting bitbake server...
INFO: Recipe ~/poky_sdk/workspace/recipes/bbexample/bbexample_git.bb has been automatically created;
   further editing may be required to make it fully functional
```

Figure 9.10: Generating a recipe from an external Git repository

`devtool` creates a workspace holding the source tree and provisional recipe. A simplified view looks like this:

```
~/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
```

Figure 9.11: Files generated by devtool add

For Yocto Project 6.0, `devtool add` autodetects Autotools, CMake, SCons, qmake, plain Makefiles, out-of-tree Linux kernel modules, Go modules, Node.js modules, and Python projects that use `setuptools`, `distutils`, or a `pyproject.toml` build backend.

Tip

Pass `-b` to `devtool add` when the source tree should be installed verbatim without compilation. This is the right behavior for repackaging prebuilt binaries such as RPMs.

Building a recipe using devtool

After staging the recipe, build it within the eSDK environment to exercise BitBake tasks without leaving the host shell:

```
$ devtool build bbexample
```

`devtool` runs the recipe's build tasks, as shown in the following figure.

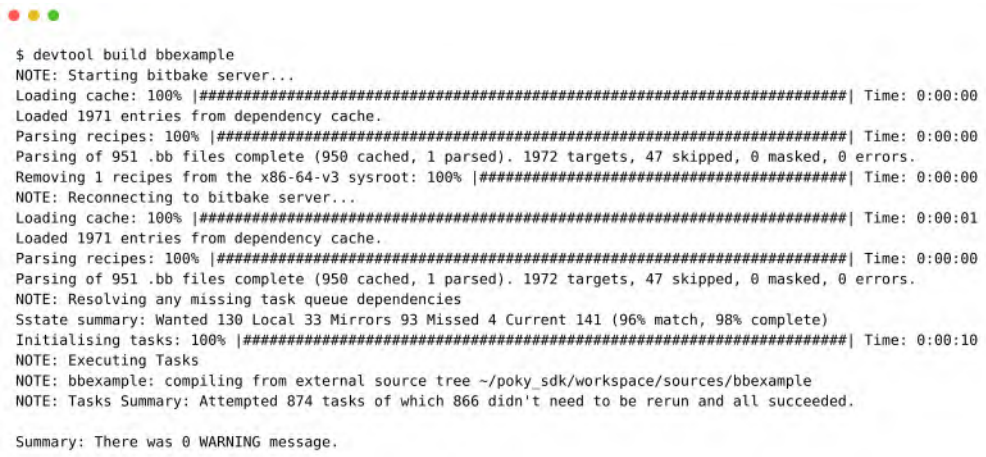


Figure 9.12: Building the bbexample recipe with devtool

Deploy the package to a reachable target from the same session. The default QEMU instance uses `192.168.7.2`, so the following command pushes the binaries over SSH:

```
$ devtool deploy-target bbexample root@192.168.7.2
```

`devtool` copies the built package to the target over SSH, as shown in the following figure.

```
$ devtool deploy-target bbexample root@192.168.7.2
NOTE: Starting bitbake server...
Loading cache: 100% |################################################################################| Time: 0:00:00
Loaded 1971 entries from dependency cache.
Parsing recipes: 100% |##############################################################################| Time: 0:00:00
Parsing of 951 .bb files complete (950 cached, 1 parsed). 1972 targets, 47 skipped, 0 masked, 0 errors.
The authenticity of host '192.168.7.2 (192.168.7.2)' can't be established.
ECDSA key fingerprint is SHA256:gR7NN26rgFn8gPjKbj6s2rVhNHflPe1Hu6potGtc1OQ.
This key is not known by any other names.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
INFO: Successfully deployed ~/poky_sdk/tmp/work/x86-64-v3-poky-linux/bbexample/0.1+git/image
```

Figure 9.13: Deploying bbexample with devtool deploy-target

Validate the result directly on the console, as shown below, to confirm the program executes on the running system.

```
root@qemux86-64:~# bbexample
Hello Yocto World...
Hello World (from a shared library!)
root@qemux86-64:~#
```

Figure 9.14: bbexample running on the QEMU target

Modifying an existing recipe with devtool

`devtool` also unpacks and stages an existing recipe so you can iterate on the upstream source without leaving the eSDK. Run `devtool modify <recipe-name>` to drop the recipe source under `workspace/sources/<recipe-name>` and register a workspace layer that takes precedence over the original recipe. Edit the tree, run `devtool build <recipe-name>` to confirm the change, and deploy the result to a running target with `devtool deploy-target <recipe-name> root@<target>`.

When the edits stabilize, capture them back into a layer with `devtool finish <recipe-name> <layer-path>`. `devtool` converts each commit on top of the upstream source into a patch under the layer's `recipes-*/<recipe-name>/` directory and removes the workspace copy, so the recipe returns to its normal, layer-managed state.

The workspace itself is a regular layer that lives under `workspace/` inside the eSDK. Its `conf/layer.conf` is generated by `devtool` and given a higher `BBFILE_PRIORITY` than the regular layers. When you modify an existing recipe, `devtool` does not copy the recipe into the workspace. Instead, it writes a `.bbappend` under `workspace/appends/` that redirects the build

to the unpacked source under `workspace/sources/<recipe-name>/`, and the workspace layer's higher priority ensures that append is applied. A full recipe copy under `workspace/recipes/<recipe-name>/` appears only when `devtool add` introduces a brand-new recipe. `devtool finish` rewrites the destination layer with the captured patches, then removes the workspace entry so the layer stack returns to its original shape.

Working with the devtool source git tree

Every source tree that `devtool` stages under `workspace/sources/<recipe-name>/` is a real Git repository, not a loose directory of files. When the recipe's `SRC_URI` points at an upstream Git repository, `devtool modify` preserves that history, so the tree opens on the exact revision the recipe builds with the project's commit log intact, and any patches the recipe already carries are applied on top as individual commits. The workspace therefore starts as a faithful, inspectable copy of what BitBake would otherwise assemble on its own.

That layout is what makes the iteration loop predictable. You edit the tree, record the work with `git commit`, and each commit becomes one patch when you run `devtool finish`. Only committed changes are captured, and nothing is lost quietly if you forget: `devtool finish` checks the source tree first and refuses to run while it is dirty, reporting `Source tree is not clean` along with the outstanding changes. Passing `-f` (`--force`) bypasses that check, which continues with a warning and leaves the uncommitted edits out of the generated patch set. Commit before you finish and the question never comes up.

Because the tree is a standard Git repository, you are not limited to producing local patches. You can add a remote for the upstream project, or your fork of it, and push a branch of your commits back toward the original component sources:

```
$ cd workspace/sources/<recipe-name>
$ git remote add upstream <upstream-url>
$ git checkout -b my-fix
$ git push upstream my-fix
```

Pushing the branch and running `devtool finish` are complementary. The push opens the review that lands your work in the upstream project, while `devtool finish` records the same commits as patches in your layer so the product keeps building against the fix until a release that carries it upstream arrives.

Extending the eSDK

The Extensible SDK can install new recipes locally so developers can pull in extra libraries without rebuilding the installer. For example, add the `libusb1` recipe to the eSDK sysroot with:

```
$ devtool sdk-install -s libusb1
```

`devtool` builds and installs the recipe into the SDK sysroot, as shown in the following figure.

```
$ devtool sdk-install -s libusb1
NOTE: Starting bitbake server...
Loading cache: 100% |################################################################################| Time: 0:00:00
Loaded 1971 entries from dependency cache.
Parsing recipes: 100% |##############################################################################| Time: 0:00:00
Parsing of 951 .bb files complete (950 cached, 1 parsed). 1972 targets, 47 skipped, 0 masked, 0 errors.
INFO: Installing libusb1...
NOTE: Reconnecting to bitbake server...
Loading cache: 100% |################################################################################| Time: 0:00:01
Loaded 1971 entries from dependency cache.
Parsing recipes: 100% |##############################################################################| Time: 0:00:00
Parsing of 951 .bb files complete (950 cached, 1 parsed). 1972 targets, 47 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Initialising tasks: 100% |###########################################################################| Time: 0:00:02
Checking sstate mirror object availability: 100% |###################################################| Time: 0:00:05
Sstate summary: Wanted 167 Local 37 Mirrors 128 Missed 2 Current 187 (98% match, 99% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 1103 tasks of which 1092 didn't need to be rerun and all succeeded.
INFO: Successfully installed libusb1
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)... (00:19:54.273228)
Loading cache: 100% |################################################################################| Time: 0:00:01
Loaded 1971 entries from dependency cache.
Parsing recipes: 100% |##############################################################################| Time: 0:00:00
Parsing of 951 .bb files complete (950 cached, 1 parsed). 1972 targets, 47 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 0 (0% match, 0% complete)
Initialising tasks: 100% |###########################################################################| Time: 0:00:02
NOTE: No setscene tasks
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded.
NOTE: Reconnecting to bitbake server...
NOTE: Retrying server connection (#1)... (00:20:05.048735)
Loading cache: 100% |################################################################################| Time: 0:00:01
Loaded 1971 entries from dependency cache.
Parsing recipes: 100% |##############################################################################| Time: 0:00:00
Parsing of 951 .bb files complete (950 cached, 1 parsed). 1972 targets, 47 skipped, 0 masked, 0 errors.
NOTE: Resolving any missing task queue dependencies
Sstate summary: Wanted 0 Local 0 Mirrors 0 Missed 0 Current 0 (0% match, 0% complete)
Initialising tasks: 100% |###########################################################################| Time: 0:00:02
NOTE: No setscene tasks
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 1 tasks of which 0 didn't need to be rerun and all succeeded.
```

Figure 9.15: Installing an additional recipe into the Extensible SDK

Coordinating those additions across a team requires more than the self-extracting installer. Developers consume published updates with `devtool sdk-update`, which pulls the shared state cache deltas from the configured SDK update server into the local eSDK.

Tip

The *Providing Updates to the Extensible SDK After Installation* section at `https://docs.yoctoproject.org/wrynose/sdk-manual/appendix-customizing.html#providing-updates-to-the-extensible-sdk-after-installation` describes how to publish updates with `oe-publish-sdk` and the `SDK_UPDATE_URL` variable so developers can run `devtool sdk-update` to pick up the latest changes.

Using VS Code with the Extensible SDK

The Yocto Project BitBake extension is designed to work with a full BitBake build environment, where it drives recipe parsing, syntax validation, and build execution through the BitBake server, as described in *Chapter 3*. The same extension can also be configured for an Extensible SDK installation, where it provides an integrated terminal with the eSDK environment active and `devtool` commands available from within the editor, without requiring the full source tree.

Create a `.code-workspace` file in the SDK root directory and add the following `settings` block to enable eSDK mode and point the extension at the environment script and SDK root:

```
{
  "settings": {
    "bitbake.eSDKMode": true,
    "bitbake.pathToEnvScript": "/home/user/poky_sdk/environment-setup-x86-64-v3-
poky-linux",
    "bitbake.pathToBuildFolder": "/home/user/poky_sdk/",
    "bitbake.commandWrapper": "",
    "bitbake.pathToBitbakeFolder": "/home/user/poky_sdk/layers/openembedded-core/
bitbake"
  }
}
```

The `bitbake.eSDKMode` flag tells the extension to source the SDK environment script rather than starting a BitBake server. Set `bitbake.pathToEnvScript` to the environment script the installer placed in the SDK root. The script name includes the target architecture, so the filename matches the path the installer reported at the end of installation. Set `bitbake.pathToBuildFolder` to the SDK root directory. Open the workspace file with `code`

`<path>` to gain an integrated terminal with the eSDK environment active and `devtool` available from the editor.

Summary

This chapter explained how the Yocto Project toolchain options cover on-device development images, application-focused Standard SDKs, and metadata-aware Extensible SDKs. You generated SDK installers, built applications and the Linux kernel, and drove `devtool` workflows that handle recipe creation, builds, deployments, and local SDK extensions.

The next chapter pivots back to the main build workspace to show how to enhance debugging, remote GDB setups, and Build History tracking to improve day-to-day diagnostics.

10

Debugging with the Yocto Project

This chapter walks through the tools the Yocto Project 6.0 (Wrynose) build system exposes for diagnosing metadata problems, packaging mistakes, and runtime failures. You will use Build History to audit what changed between builds, `oe-pkgdata-util` to inspect generated packages, `bitbake-getvar` to confirm how variables expand at parse time, BitBake's task-logging infrastructure to trace opaque task failures, and **Development Shell** to step inside a recipe's build environment interactively. The chapter closes with a practical guide to deploying GDB against a running target image.

The goal is to give you a repeatable process for narrowing any failure to a specific tool, variable, or package decision before touching a single line of metadata.

A build that fails with no actionable signal is the most expensive kind. Engineers who cannot read task logs or correlate a packaging change to a regression burn hours re-running full builds and guessing at root causes. The techniques in this chapter replace that cycle with targeted, evidence-based investigation.

We will cover the following main topics:

- Differentiating metadata and application debugging
- Tracking image, package, and SDK contents
- Debugging packaging
- Debugging with logging in the Yocto Project
- Debugging metadata variables
- Using a Development Shell
- Using the GNU Debugger for debugging

- Going deeper with profiling and tracing
- Walking through a missing file

Technical requirements

This chapter requires a build host configured according to *Chapter 2*, including a BitBake Setup workspace initialized and at least one successful image build already completed. Build History and Package Information data are only available after a build has run, so the inspection tools covered here have nothing to query against a fresh workspace.

Most sections work with the standard `core-image-minimal` target running under QEMU. The GDB section requires an image that includes `gdbserver`, and the chapter shows how to extend an existing image recipe to add it. Hardware-specific timing or memory-mapped peripheral debugging is outside the scope of this chapter, but every other technique applies equally to physical targets and QEMU.

Differentiating metadata and application debugging

Treat debugging as two related activities. Metadata debugging confirms that BitBake expands variables correctly, runs tasks in the right order, and generates reproducible artifacts. Application debugging focuses on binaries that run on the target system, including the Linux kernel, user-space daemons, and libraries.

Use the following perspectives to decide whether a problem stems from metadata or runtime code paths:

- Metadata debugging inspects BitBake's task logs, Shared State Cache usage, and generated packages whenever a feature gate or dependency does not behave as expected. The work often happens on the build host by reading the per-build cooker logs under `build/tmp/log/cooker/<machine>/` or querying BitBake directly.
- Runtime debugging borrows familiar techniques such as attaching GDB, collecting traces, or adding debug prints. It uses SDK content or deployed images so you can inspect binaries while they execute on hardware or QEMU.

Approaching problems from both angles avoids tunnel vision and keeps fixes grounded in how the Yocto Project combines metadata with source code.

Tracking image, package, and SDK content

Build History captures the collateral effects of each build so you can see whether a new package, layer update, or configuration tweak changed more than you expected. Sometimes the differences stem from deliberate adjustments. Other times they appear because an

upstream layer pulled a new dependency, flipped a default, or because you intentionally modified a layer under active development. Add the following snippet to `build/conf/local.conf` to capture those records automatically inside `build/buildhistory` and keep the changes audit-friendly.

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

BitBake creates a Git commit in `build/buildhistory` after every successful build that produces an image or SDK. That directory is an ordinary Git repository, so `git log` and `git diff` work there exactly as they do on any other project, and `buildhistory-diff` is a convenience wrapper that renders those same commits in terms of packages, sizes, and dependencies instead of raw file changes. Navigate into that repository and run `buildhistory-diff` as shown below to compare two consecutive builds:

```
$ bitbake core-image-minimal
$ cd build/buildhistory
$ buildhistory-diff HEAD~1..HEAD
```

The utility prints package additions, removals, and size differences so you can confirm whether layer updates or local experiments had the desired effect, as illustrated here:

```
$ bitbake core-image-minimal
$ cd build/buildhistory
$ buildhistory-diff HEAD~1..HEAD
Changes to images/qemux86_64/glibc/core-image-minimal (files-in-image.txt):
  /bin/login, /bin/su, /sbin/sulogin, /usr/bin/passwd: symlinks changed (busybox → shadow/util-linux)
  /etc/bluetooth/*, /etc/dbus-1/*, /etc/default/*, /etc/init.d/*, /etc/rc*.d/* were added
  /etc/securetty, /etc/shadow, /etc/login.defs, /etc/gshadow, /etc/terminfo/* were added
  /lib/libgcc_s*, /lib/libmount*, /lib/libtinfo*, /lib/libudev* were added
  /sbin/nologin*, /sbin/sulogin.util-linux, /sbin/vigr*, /sbin/vipw* were added
  /usr/bin/bluetooth*, /usr/bin/dbus-*, /usr/bin/hci*, /usr/bin/shadow-utils were added
  /usr/lib/libbluetooth*, /usr/lib/libdbus*, /usr/lib/libglib*, /usr/lib/libstdc++* were added
  /usr/libexec/bluetooth/bluetoothd, /usr/libexec/dbus-daemon-launch-helper were added
  /usr/share/dbus-1/*, /usr/share/glib-2.0/*, /usr/share/mime/** were added (large trees)
  (...)
  /var/lib/bluetooth, /var/lib/dbus, /var/spool/mail were added
images/qemux86_64/glibc/core-image-minimal: /etc/group:
  All entries changed * → x; messagebus:x:999: was added
images/qemux86_64/glibc/core-image-minimal: /etc/passwd:
  All entries changed * → x; messagebus:x:999:999::/var/lib/dbus:/bin/false was added
images/qemux86_64/glibc/core-image-minimal: IMAGESIZE changed from 31580 to 56952 (+80%)
Changes to images/qemux86_64/glibc/core-image-minimal (installed-package-names.txt):
  Added: bluez5, dbus-1, dbus-common, dbus-tools, shadow, shadow-base,
         shared-mime-info, libglib-2.0-0, libstdc++6, libx11-6 (and more)
```

Figure 10.1: Comparing two build history revisions

This example assumes `core-image-minimal` was first built with Build History enabled to establish the baseline commit, then `CORE_IMAGE_EXTRA_INSTALL += "bluez5"` was added to `build/conf/local.conf` and the image was rebuilt to produce a second commit. That two-commit history is what gives `buildhistory-diff` something to compare, which is why the diff highlights the new Bluetooth package and the larger filesystem footprint.

For deeper coverage of Build History commands and data formats, study the *Maintaining Build Output Quality* section (section 34) at `https://docs.yoctoproject.org/wrynose/dev-manual/build-quality.html#maintaining-build-output-quality`. That reference explains additional filters, reporting hooks, and integration ideas for automated build dashboards.

Debugging packaging

Advanced recipes often deliver several packages so downstream users can decide which libraries, plugins, or debug data they need. Inspect the results under `build/tmp/work/<arch>/<recipe-name>/<version>/packages-split/` whenever a file appears in an unexpected location. Common causes include incorrect install paths, disabled configure switches, or a missing entry in the package split rules.

Missing headers or static libraries also trigger build failures down the dependency chain. Those artifacts land under `build/tmp/work/<arch>/<recipe-name>/<version>/sysroot-destdir/`, so confirming their presence helps narrow the search before editing metadata.

Inspecting packages

The `oe-pkgdata-util` helper queries the **Package Information** stored under `build/tmp/pkgdata/`. This single source of truth describes which recipe produced a package, which files it ships, and the dependency tree recorded during packaging. The `pkgdata/` directory is populated by the recipe's packaging tasks, so `oe-pkgdata-util` returns nothing until the recipe in question has been built at least once. The following workflow lists all output packages related to `bluez5` after you build that recipe.

```
$ bitbake bluez5
$ oe-pkgdata-util list-pkgs -p bluez5
```

The command prints every package name the recipe produces, as illustrated here:

```
$ bitbake bluez5
$ oe-pkgdata-util list-pkgs -p bluez5
bluez5
bluez5-dbg
bluez5-dev
bluez5-noinst-tools
bluez5-obex
bluez5-ptest
bluez5-src
bluez5-testtools
```

Figure 10.2: Listing packages emitted by bluez5

When you need to map a file to the package that contains it, use the command shown below. The example looks for `/usr/bin/rfcomm` to verify whether the expected runtime helper is still installed.

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

The tool prints the package name that ships the requested path, as illustrated here:

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

Figure 10.3: Locating the package that ships /usr/bin/rfcomm

Version drift becomes obvious once you compare the currently deployed package with the new build. The following query prints the package version (`PKGV`) recorded in the Package Information set for `bluez5`.

```
$ oe-pkgdata-util read-value PKGV bluez5
```

The command returns the version string recorded during packaging, as illustrated here:

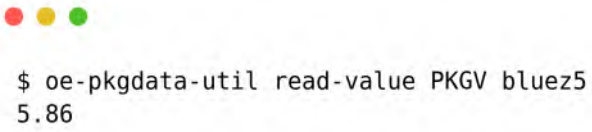


Figure 10.4: Reading the bluez5 package version

Finally, inspect the files shipped with a package when you validate install rules. The following command lists all files the `bluez5-obex` package installs so you can confirm that service units or helper binaries remain in the image.

```
$ oe-pkgdata-util list-pkg-files bluez5-obex
```

The output groups each file under its package name, as illustrated here:

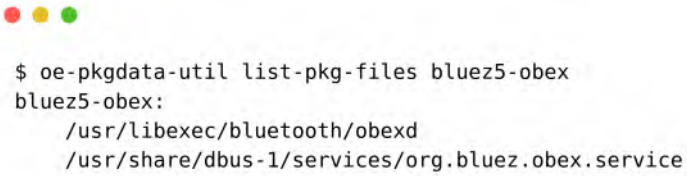


Figure 10.5: Listing the files provided by bluez5-obex

Combining Package Information queries with the on-disk inspection of `packages-split/` and `sysroot-destdir/` gives you a full picture of what BitBake staged, packaged, and exported to dependent recipes.

Debugging with logging in the Yocto Project

BitBake ships Python logging helpers so you can instrument tasks without building ad hoc print statements. Use the following levels to describe the severity of a message:

- `bb.fatal()`: stops BitBake execution and reports a critical failure.
- `bb.error()`: records an error message. The current task continues, but BitBake reports the build as failed once the task finishes.
- `bb.warn()`: highlights a condition that may lead to incorrect results.
- `bb.note()`: writes informative details that might help other developers understand state changes.

- `bb.plain()`: prints unformatted text, often used in helper scripts.
- `bb.debug(level, message)`: emits verbose traces when you run BitBake with matching debug settings.

Shell tasks expose the same functionality as functions such as `bbfatal`, `bberror`, `bbwarn`, `bbnote`, `bbplain`, and `bbdebug`. They funnel log entries into task-specific files under `build/tmp/work/<arch>/<recipe-name>/<version>/temp/`, the per-recipe work area detailed in *Chapter 6*, with scripts stored as `run.<task>.<pid>` and logs exposed as `log.<task>.<pid>`. Symbolic links named `log.<task>` point at the most recent run, which makes it easy to review `log.do_compile` or `log.do_install` after a failure.

Debugging metadata variables

The `bitbake-getvar` script prints the current value of a metadata variable and where BitBake collected each assignment. That visibility makes it easier to audit overrides, `:append` fragments, and class-provided defaults.

The following command asks for the `PACKAGECONFIG` variable from the `procps` recipe and shows the recipe's default expression, which expands to `systemd` whenever that flag is present in `DISTRO_FEATURES`.

```
$ bitbake-getvar -r procps PACKAGECONFIG
```

The output traces every assignment that contributed to the final value, as illustrated here:

```
$ bitbake-getvar -r procps PACKAGECONFIG
# $PACKAGECONFIG [4 operations]
#   set .../layers/openembedded-core/meta/conf/documentation.conf:311
#     [doc] "This variable provides a means of enabling or disabling features of a recipe on a per-recipe
basis."
#   set .../layers/openembedded-core/meta/conf/bitbake.conf:942
#     [vardepvalue] "${PACKAGECONFIG}"
#   set .../layers/openembedded-core/meta/recipes-extended/
#     procps/procps_4.0.6.bb:33
#     [_defaultval] "${@bb.utils.filter('DISTRO_FEATURES', 'systemd', d)}"
#   set .../layers/openembedded-core/meta/recipes-extended/
#     procps/procps_4.0.6.bb:34
#     [systemd] "--with-systemd,--without-systemd,systemd"
# pre-expansion value:
#   "${@bb.utils.filter('DISTRO_FEATURES', 'systemd', d)}"
PACKAGECONFIG=""
```

Figure 10.6: Inspecting PACKAGECONFIG for procps

Once you know the source file, navigate to the lines shown below to understand how the recipe guards optional features.

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

Figure 10.7: PACKAGECONFIG snippet from procps

Use `bitbake-getvar` whenever you need to validate feature toggles, confirm that a `DEPENDS` entry is active, or trace how anonymous Python blocks mutate variables during parsing. When you need the full expanded environment of a recipe rather than a single variable, run `bitbake -e <recipe-name>` to dump every assignment BitBake produced for the recipe and pipe the output through `grep` or `less` to find the values you care about.

Using a Development Shell

The `devshell` task opens an interactive shell that mirrors the environment BitBake uses to build a recipe. Entering a Development Shell performs the following actions automatically:

1. Extracts the source tree into `build/tmp/work/<arch>/<recipe-name>/<version>/` for most recipes. The `linux-yocto` recipe is an exception and places its source in `build/tmp/work-shared/qemux86-64/kernel-source/` because the Linux kernel source is shared across machines.
2. Applies patches declared in the recipe and any `.bbappend` files
3. Drops you into a subshell rooted in `${S}`, the unpacked source tree under `WORKDIR`, with all relevant BitBake environment variables set

You can start a session for the Linux kernel recipe with the following command.

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

On the first run, the shell does not open immediately.

Note

The community Shared State Cache does not cover source fetching, so BitBake must download and unpack the full Linux kernel source tree before opening the shell. Expect the first run to take several minutes depending on your connection speed. Subsequent runs are faster because the unpacked tree is already present in `WORKDIR`.

BitBake resolves task dependencies, restores Shared State Cache entries where possible, and then drops you into a terminal rooted in `${S}` inside the recipe's `WORKDIR`. The shell is ready once BitBake reports the `do_devshell` task as succeeded.

Inside the development shell you have direct access to the Linux kernel source tree along with BitBake-set variables such as `CC`, `LD`, and `CFLAGS`. Listing the directory contents as shown below confirms that the familiar Linux kernel layout is ready for inspection and quick edits.

```
$ ls
```

The listing shows the expected Linux kernel source layout, as illustrated here:



Figure 10.8: Typical files inside WORKDIR during a Development Shell

Editing files in this working directory is the fastest way to test a fix, but the layout is transient and BitBake will rebuild it from scratch on the next clean run.

The shell also gives you the task run scripts. BitBake writes one shell script per executed task into `${WORKDIR}/temp`, so running `${WORKDIR}/temp/run.do_compile` replays that single task with the exact environment BitBake used, without going through the full Task Graph. This is the fastest way to see what a failing task actually executes, and pairing it with the matching `log.do_compile` in the same directory shows the output the previous run produced.

Note

Changes made inside a Development Shell session are not persisted automatically. Convert important fixes into patches or commits before closing the terminal so that BitBake can reapply them on the next build.

Use the Development Shell for short investigations, then switch to `devtool modify <recipe-name>` or an external toolchain when you need a longer-lived workspace with source control history.

Using the GNU Debugger for debugging

GDB ships with both the SDK and most eSDK configurations covered in *Chapter 9*, which makes it the primary debugger for systems built with the Yocto Project. When you need debug symbols and helper utilities on the target image itself, append the following fragment to `build/conf/local.conf`.

```
IMAGE_FEATURES += "tools-debug tools-profile tools-testapps"
EXTRA_IMAGE_FEATURES += "dbg-pkgs dev-pkgs"
```

The added features install debug binaries, profiling aids (`perf`, `SystemTap`, `LTTng`), and SDK helper scripts in the finished image. The `tools-profile` value maps to the `packagegroup-core-tools-profile` package group registered as `FEATURE_PACKAGES_tools-profile` in `meta/classes-recipe/core-image.bbclass`, which is the canonical implementation name. The Yocto Project 6.0 reference manual lists this entry as `perf`, so you may see either spelling in newer documentation while metadata fragments and existing `build/conf/local.conf` files keep using `tools-profile`. Combine those tools with `gdbserver` when memory or storage limits prevent you from shipping debug symbols on the target. In that flow, `gdbserver` runs alongside your program on the device and accepts commands from a GDB instance on the build host that still has access to the unstripped ELF files.

Keep the following considerations in mind when you prepare a remote-debug session:

- Make sure the host retains a copy of the binaries and the exact symbol versions used to build them so GDB can resolve addresses correctly.
- Build the target program with `-O0 -g3` or a similar low-optimization profile to preserve variable names and simplify single-stepping.
- Remember that distribution hardening flags change what you see at runtime. Position-independent executables have no fixed load base, so addresses in a crash log or a `/proc` dump only make sense once you know the base the loader picked. GDB resolves this on its own when it attaches to a live process, but stack protector and fortify-source instrumentation can still reroute a failure through a runtime abort rather than the faulting line, so consider rebuilding the recipe under investigation without those flags when the symptom is hard to localize.

The *Debugging With the GNU Project Debugger (GDB) Remotely* section (section 35.14) at `https://docs.yoctoproject.org/wrynose/dev-manual/debugging.html#debugging-with-the-gnu-project-debugger-gdb-remotely` walks through target and host setup, port selection, and connection sequencing for the Yocto Project 6.0 release.

Going deeper with profiling and tracing

Once a build reaches a state where you can reproduce performance regressions or hard-to-trigger failures, the Yocto Project's *Profiling and Tracing Manual* at `https://docs.yoctoproject.org/wrynose/profile-manual/` becomes the canonical reference. The manual documents how to use `perf`, `ftrace`, `SystemTap`, `Sysprof`, `LTTng`, and `blktrace` on targets built with the Yocto Project, including the `IMAGE_FEATURES` and `tools-profile` fragments needed to ship the right tooling. Reach for it when GDB's single-step view is too narrow to capture the symptom and you need timing, scheduling, or storage-stack data to round out the picture.

Walking through a missing file

Each section so far introduced one tool on its own. Real failures rarely announce which tool they need, so this section follows a single symptom from first report to fix and names the tool that answers each question along the way. The symptom is one of the most common in day-to-day work: a recipe builds without error, but a file it is supposed to deliver is absent from the running image.

Start by confirming what actually shipped rather than what you expected to ship. Run `oe-pkgdata-util find-path` against the file on the build host. When the tool names a package, as it did for `/usr/bin/rfcomm` in *Figure 10.3*, the file was produced and packaged, and the question becomes why that package is not in the image. When the file was never packaged, the tool says so explicitly, as shown below for a `core-image-minimal` build that does not ship `rsync`:

```
$ oe-pkgdata-util find-path /usr/bin/rsync
```

The error names the path you asked about and confirms that no package produces it, as illustrated here:

```
$ oe-pkgdata-util find-path /usr/bin/rsync
ERROR: Unable to find any package producing path /usr/bin/rsync
```

Figure 10.9: Confirming that no package produces a path

A result like this moves the search earlier in the pipeline, into `do_install` or the package split rules, because packaging never saw the file at all.

When the file was packaged but the image lacks it, the package simply was not pulled in. Confirm with `oe-pkgdata-util list-pkg-files <package-name>`, shown for `bluez5-obex` in *Figure 10.5*, that the file belongs where you think it does, then check whether anything requests that package. A recipe that splits its output leaves the main binary in one package and helpers in another, so an image recipe that installs only the main package silently drops the rest.

Build History makes the correction visible. Once you install the package that provides the file, `buildhistory-diff` compares the new commit against the previous one and reports what changed, as shown below:

```
$ bitbake core-image-minimal
$ cd build/buildhistory
$ buildhistory-diff HEAD~1..HEAD
```

The diff confirms the file arrived and, more usefully, shows what arrived with it, as illustrated here:

```
$ bitbake core-image-minimal
$ cd build/buildhistory
$ buildhistory-diff HEAD~1..HEAD
Changes to images/qemux86_64/glibc/core-image-minimal (files-in-image.txt):
  /etc/rsyncd.conf was added
  /usr/bin/rsync was added
  /usr/bin/rsync-ssl was added
  /usr/lib/libpopt.so.0.0.2 was added
  /usr/lib/libpopt.so.0 was added
Changes to images/qemux86_64/glibc/core-image-minimal (installed-package-names.txt):
  libpopt0 was added
  rsync was added
```

Figure 10.10: Confirming that a file returned to the image

Only `rsync` was requested, yet `libpopt0` entered the image alongside it, together with the shared libraries it provides and an `/etc/rsyncd.conf` configuration file. That is the property worth internalizing: the report describes what the change actually did, not what you asked it to do, so it catches the dependency you did not know you were pulling in as readily as it confirms the fix you intended.

When packaging never saw the file, move to the recipe's work area under `build/tmp/work/<arch>/<recipe-name>/<version>/`. Look first at `packages-split/`, which shows the final division of files into packages, and at `image/`, which shows everything `do_install` placed before that division happened. A file present in `image/` but absent from every directory under `packages-split/` means the split rules never claimed it, and the fix belongs in the recipe's packaging variables. A file absent from `image/` as well means `do_install` never delivered it, and the fix belongs in the install logic or in whatever configure switch decides that the component gets built at all.

That last case is where the tools compound. Read `temp/log.do_install` in the same work area to see what the install step actually did. If the log shows the install command running against a file that does not exist, the component was never compiled, which points at a disabled build option rather than a broken install rule. Confirm that suspicion with `bitbake-getvar -r <recipe-name> PACKAGECONFIG`. As *Figure 10.6* showed for `procps`, the command prints every assignment that contributed to the value along with the file and line number that set it, so a feature left switched off by a distribution setting is visible without reading the recipe.

If the recipe's own logic is still unclear, open a Development Shell with `bitbake <recipe-name> -c devshell` and replay the failing step directly with `${WORKDIR}/temp/run.do_install`. Running the task in isolation puts the output in front of you immediately instead of buried in a full build, and you can edit the source tree in place and run the task again until the file appears where you expect it. Remember that this working directory is transient, so convert whatever you learn into a recipe change or a patch before you close the shell.

The order matters more than any individual command. Each step narrows the search to one stage of the pipeline, from image composition, to packaging, to installation, to compilation, and each answer determines which tool is worth running next. Applying this same narrowing to other symptoms is what makes the rest of the chapter useful beyond the specific examples it shows.

Summary

You now have a catalog of debugging workflows that span metadata inspection, package information queries, BitBake logging, and runtime analysis with the Development Shell and GDB. Build history documents every image and SDK revision so you can catch accidental regressions, while `bitbake-getvar` and `oe-pkgdata-util` expose the values that BitBake and package managers store behind the scenes. Combine those insights with GDB or `gdbserver` and you can validate even the most complex build configuration without leaving the Yocto Project toolchain.

The next chapter shifts focus to bringing additional layers into your build environment so you can extend functionality beyond what ships in OpenEmbedded Core.

11

Exploring External Layers

The Yocto Project 6.0 (Wrynose) ecosystem includes hundreds of community-maintained layers covering BSPs, software stacks, security hardening, and machine-learning frameworks. This chapter surveys the major layer categories, walks through their directory structures, and explains how BitBake discovers and prioritizes metadata from multiple sources. It also covers how to evaluate a layer's `conf/layer.conf`, `README`, and license files before pulling it into a build.

The goal is to give you the judgment to select and integrate an existing community layer with confidence rather than duplicating work that is already maintained upstream. Writing equivalent recipes from scratch takes time, adds metadata you have to maintain yourself, and means your project absorbs future upstream fixes more slowly. Equally important, adding a layer without reviewing its license metadata and provenance can introduce compliance obligations into your product that are hard to unwind later.

We will cover the following main topics:

- Powering flexibility with layers
- Detailing a layer's source code
- Adding meta layers
- The OpenEmbedded Layer Index

Technical requirements

This chapter requires a build host configured as described in *Chapter 2*, with a working BitBake Setup workspace and the baseline host packages already installed. You also need `git` and outbound network access to clone layers from GitHub, the OpenEmbedded Layer Index at `https://layers.openembedded.org`, and any vendor mirrors you plan to use. The `git` package

is part of the required-packages list covered in *Chapter 2*, so no additional installation step is needed if you followed that chapter.

Powering flexibility with layers

The Yocto Project 6.0 organizes its metadata into configuration files, classes, and recipes. Configuration files (`.conf`) set variables in BitBake's global configuration namespace, and machine definitions and distribution definitions are both configuration files of this kind: a machine file describes the target hardware, and a distribution file describes the policy applied across the whole image. Classes (`.bbclass`) hold logic that recipes inherit, and recipes (`.bb`) describe how to fetch, configure, and build individual components, from simple applications to complex graphical frameworks. BitBake supports metadata from multiple sources, referred to as metadata layers, using the `BBPATH` and `BBFILES` search mechanics introduced in *Chapter 4*. The primary advantage of layers is their ability to segment metadata logically, letting users include only the necessary components for a specific project. Using metadata layers provides the following benefits:

- **Improved code reuse**: Simplifies the process of applying previously developed solutions to new projects
- **Collaboration and wider reach**: Allows teams, communities, and vendors to share and extend their work across many projects
- **Enhanced code quality**: The Yocto Project community's collective focus on specific metadata layers improves overall quality

Layers enable a wide range of customizations, such as toggling features or optimizing build flags for specific architectures. By encapsulating these adjustments within layers instead of altering the upstream source code directly, developers improve maintainability and modularity. Organizing metadata into distinct layers significantly improves reusability for future projects.

For instance, a build created from the `poky-wrynose` BitBake Configuration Template enables three layers by default, `meta`, `meta-poky`, and `meta-yocto-bsp`, which can be viewed with the following command:

```
$ bitbake-layers show-layers
```

BitBake prints each layer with its file system path and its priority. The priority comes from `BBFILE_PRIORITY` in the layer's `conf/layer.conf`, and it decides which layer wins when two of them provide a recipe with the same name: the higher number supplies the recipe, and `.bbappend` files apply in priority order. All the reference layers sit at priority 5, so nothing

takes precedence over anything else in a default build. Priority does not control the order in which BitBake parses layers, though. That comes from their position in `BBLAYERS` and settles conflicts between configuration variables instead, a distinction *Chapter 5* covers in full.

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

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

The preceding figure demonstrates the organization of default layers within the Yocto Project reference configuration. This figure clarifies the relationships between layer names, their file system paths, and their assigned priorities. The Yocto Project reference configuration includes:

- `meta`: The OpenEmbedded Core metadata layer, a mixed layer type containing recipes, classes, and QEMU machine configurations. It serves as the Yocto Project's baseline.
- `meta-poky`: The reference distribution layer, containing the default distribution configuration (`poky.conf`). Widely used for testing, it also serves as a starting point for designing custom distributions.
- `meta-yocto-bsp`: The reference BSP layer, containing machine configuration files and recipes for package configuration. This serves as a reference example for BSP layers.

The reference layers above are useful examples, but real products usually depend on a vendor BSP that targets a specific board. *Chapter 16* walks through that workflow with concrete layers such as `meta-raspberrypi` and `meta-riscv`, showing how a third-party BSP plugs into the same metadata tree.

Another significant category of layers is the software layer, which primarily includes applications or configuration files for applications. These layers are architecture-agnostic, making them universally applicable across various setups. The Yocto Project 6.0 offers a diverse range of software layers. Notable examples include `meta-virtualization`, `meta-tensorflow`, and `meta-security`. The `meta-security` layer provides hardening tools, intrusion detection components, and other utilities for building security-focused images. The `meta-virtualization` layer provides support for containers and hypervisors. The

`meta-tensorflow` layer introduces capabilities for machine learning and artificial intelligence applications.

For products with unique requirements, modifying the `build/conf/local.conf` file might initially appear convenient. However, the recommended and sustainable approach is to create a dedicated distribution layer tailored to distribution-specific configurations. The `build/conf/local.conf` file is inherently volatile and should not be version-controlled with Git. It is advisable to avoid using this file for defining package versions, providers, or production-critical system features. Instead, it should be reserved for temporary testing and development purposes.

Tip

Reducing custom settings in `build/conf/local.conf` promotes reproducibility, enhances build consistency, and contributes to the overall stability of the project.

Detailing a layer's source code

Each layer in the Yocto Project adheres to a well-defined and structured directory tree, as demonstrated in the following example:

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

Figure 11.2: The standard layer layout and its commonly used components

The `meta-` prefix for layer names is a widely accepted convention but not a strict requirement. Within this directory, the following key components are typically present:

- `COPYING.MIT`: The license governing the use and distribution of the layer's metadata. MIT is the common convention, which is why the file usually carries the `.MIT` suffix, but a layer can be published under any license. Read this file before adopting the layer.
- `README`: Detailed information for users, including dependencies and instructions for layer usage.
- `classes`: Optional directory containing `.bbclass` files, which define reusable tasks or configurations shared across multiple recipes.

- `conf`: Mandatory directory housing essential configuration files. The `conf/layer.conf` file defines the layer's properties: the collection name, the pattern that scopes its recipes, its `BBFILE_PRIORITY`, and the releases it declares compatibility with through `LAYERSERIES_COMPAT`. *Chapter 4* walks through these settings using `meta-poky/conf/layer.conf` as the example, and *Chapter 12* covers writing one for a layer of your own.
- `recipes-*`: Subdirectories named after recipes or related groups, holding `.bb` (BitBake recipe) and `.bbappend` (recipe extension) files for organized management.

The organization of the `conf` folder varies depending on the layer type. For example, a BSP layer (*a* in *Figure 11.3*) primarily focuses on board-specific configurations, while a distribution layer (*b* in *Figure 11.3*) includes files defining settings at the distribution level. Below are examples illustrating the directory structure for these two types of layers:

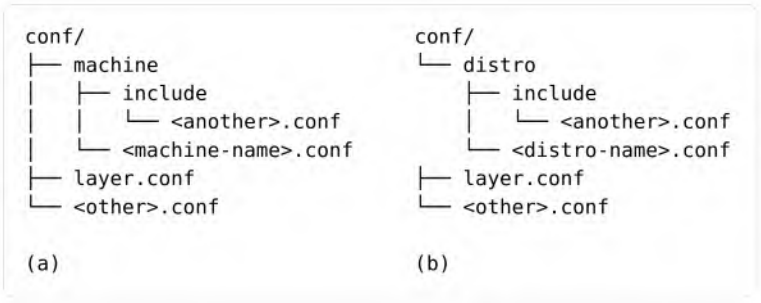


Figure 11.3: The <layer>/conf directory structure for BSP and distribution layers

Recipes are grouped into category directories, conventionally named `recipes-<category>`. BitBake does not enforce that naming: what a layer needs is a `BBFILES` glob in `conf/layer.conf` that matches wherever its recipes live. Following the convention is still worth it because it keeps a layer legible to anyone who has read another one. The categories in common use include:

- `recipes-core`: Recipes essential for the core system, such as foundational libraries and services
- `recipes-bsp`: BSP recipes providing hardware-specific configurations and drivers
- `recipes-graphics`: Recipes related to graphical components, such as rendering libraries and graphical toolkits

- `recipes-multimedia`: Multimedia-related recipes, including codecs, media players, and streaming tools
- `recipes-kernel`: Recipes for Linux kernel components, enabling customization and extensions

For instance, a typical layout in a meta layer might look as follows:

```
recipes-multimedia/
├── lame
│   ├── lame
│   │   ├── clang.patch
│   │   └── no-gtk1.patch
│   └── lame_3.100.bb
└── webp
    └── libwebp_1.6.0.bb
```

Figure 11.4: An example of the recipes- layout*

This well-organized structure provides logical segmentation of metadata, simplifying its management, extension, and sharing across different layer-specific configurations and recipes.

Adding meta layers

The Yocto Project and OpenEmbedded Project communities maintain hundreds of meta layers that extend build capabilities. These layers can be cloned into your project source directory to add functionality and customize builds.

To include a specific meta layer, such as `meta-oe` (a component of the `meta-openembedded` repository), in your project, you can fetch its source code and update configuration files or use BitBake commands to integrate it.

From the `layers/` directory of your workspace, clone the source code so the checkout lands alongside `openembedded-core` and `meta-yocto`. For instance, to clone the `meta-openembedded` layer:

```
$ git clone https://git.openembedded.org/meta-openembedded -b wrynose
```

Add the absolute path of the cloned layer to the `build/conf/bblayers.conf` file. This lets BitBake locate and include the layer during the build process. Below is an example of the updated configuration:

```
BBLAYERS += "/home/user/bitbake-builds/poky-wrynose/layers/meta-openembedded/meta-oe"
```

Once the configuration files are correctly set up, you can verify that BitBake is ready by running the `bitbake-layers show-layers` command:

```
$ bitbake-layers show-layers
```

The output now includes the newly added layer, as shown below.

```
$ bitbake-layers show-layers
layer                 path                                      priority
==========================================================================
core                  .../layers/openembedded-core/meta  5
yoctobsp              .../layers/meta-yocto/meta-yocto-bsp  5
yocto                 .../layers/meta-yocto/meta-poky  5
openembedded-layer    .../layers/meta-openembedded/meta-oe  5
```

Figure 11.5: Successfully verified enabled layers

Editing `build/conf/bblayers.conf` by hand is fine for a one-off, but the `bitbake-layers` command-line tool performs the same registration and checks the path as it goes. From the build directory, run the following, pointing it at the layer you cloned:

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

The command edits `build/conf/bblayers.conf` for you and prints nothing when it succeeds. Run `bitbake-layers show-layers` again to confirm the result.

Removing a layer works the same way. `bitbake-layers remove-layer` takes the layer path or its name and drops the matching entry from `build/conf/bblayers.conf`. Detaching a layer removes every recipe it supplied from the index, so run `bitbake-layers show-layers` afterwards and rebuild to confirm nothing you still depend on came from that layer.

You can incorporate external meta layers into your Yocto Project builds by manually modifying configuration files or using the `bitbake-layers` tool to expand and customize your project's capabilities. For projects that adopt the modern `bitbake-config-build` workflow, layers can also be activated through BitBake Configuration Fragments, as covered in *Chapter 14*.

The OpenEmbedded Layer Index

The OpenEmbedded community maintains the **OpenEmbedded Layer Index** at `https://layers.openembedded.org`, a searchable catalog of available layers. The screenshot below shows the Layers tab from the index:

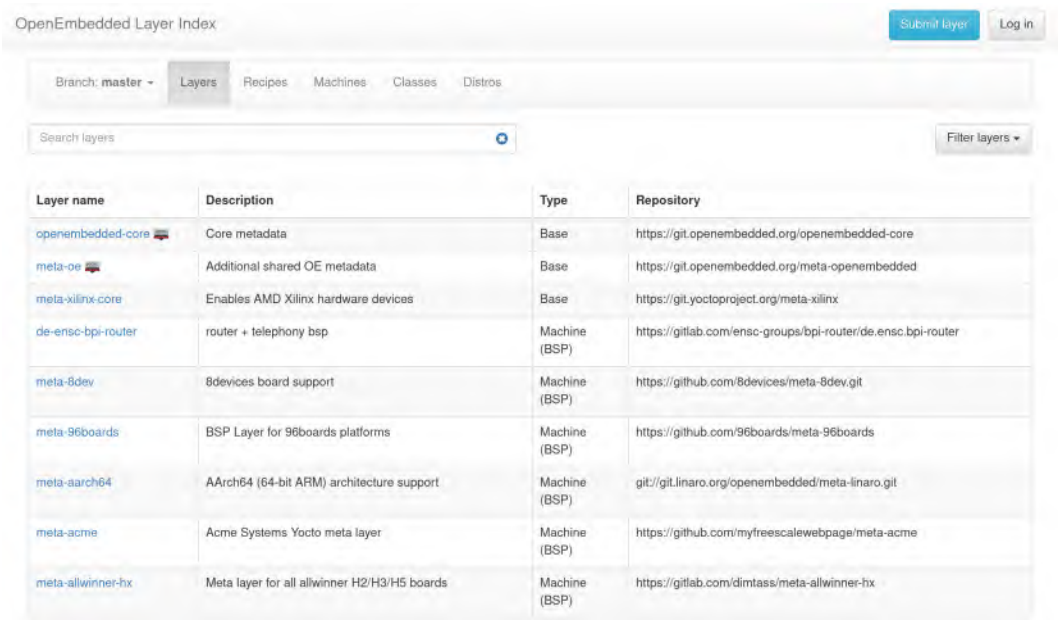


Figure 11.6: The OpenEmbedded Layer Index

The OpenEmbedded Layer Index website also lets you search for specific software types or recipes, including 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, run `bitbake-layers layerindex-fetch meta-oe -b wrynose`. The tool resolves the layer from the index, clones it, and adds it to `build/conf/bblayers.conf`, as the following figure shows. This works because `meta-openembedded` is

tracked on the index. For any other layer, confirm it is listed there first, because `layerindex-fetch` can only fetch what the index knows about.

```
$ bitbake-layers layerindex-fetch meta-oe -b wrynose
NOTE: Starting bitbake server...
Loading https://layers.openembedded.org/layerindex/api/;branch=wrynose...
Layer                                          Git repository (branch)
Subdirectory
==========================================================================================================================
==
local:HEAD:openembedded-core                   file://.../layers/
openembedded-core/meta (wrynose) meta
  required by: meta-oe
layers.openembedded.org:wrynose:meta-oe        https://git.openembedded.org/meta-openembedded (wrynose) meta-
oe
Cloning into '.../layers/openembedded-core/meta-openembedded'...
remote: Enumerating objects: 320557, done.
remote: Counting objects: 100% (1101/1101), done.
remote: Compressing objects: 100% (303/303), done.
remote: Total 320557 (delta 830), reused 1011 (delta 786), pack-reused 319456 (from 1)
Receiving objects: 100% (320557/320557), 78.71 MiB | 11.44 MiB/s, done.
Resolving deltas: 100% (198411/198411), done.
Adding layer "meta-oe" (.../layers/openembedded-core/meta-openembedded/meta-oe)
 to conf/bblayers.conf
```

Figure 11.7: Fetching a layer from the OpenEmbedded Layer Index

Summary

This chapter introduced the concept of layering. You learned about the directory structure in detail and the content in each layer type. You also saw how to add an external layer to your project manually or using the BitBake command line, and how to use the OpenEmbedded Layer Index to find available layers.

The next chapter explores why you need to create new layers and what common metadata they include, such as machine definition files, recipes, and images. It wraps up with an example of distribution customization.

12

Creating Custom Layers

This chapter walks through creating a custom layer in the Yocto Project 6.0 (Wrynose), from generating the initial directory structure to adding recipes, image definitions, and configuration fragments. Where *Chapter 11* covered consuming layers that others publish, this chapter focuses on authoring one of your own to satisfy requirements that no existing layer meets. A custom layer lets you expand system capabilities, fine-tune hardware configurations, and keep project-specific changes separate from upstream metadata. That separation simplifies maintenance, makes collaboration easier, and reduces the cost of upgrading to newer Yocto Project releases.

We will cover the following main topics:

- Creating the layer
- Adding metadata to the layer
- Creating an image
- Adding a package recipe
- Adding support for a new machine definition
- Wrapping an image for your machine
- Using a custom distribution
- Adjusting default `DISTRO_FEATURES` and `MACHINE_FEATURES`
- Interaction between `MACHINE_FEATURES` and `DISTRO_FEATURES`
- Understanding the scope of variables in BitBake

Technical requirements

This chapter requires a build host configured as described in *Chapter 2*, including a working BitBake Setup workspace with the Poky distribution layers fetched and initialized. You should

be familiar with the layer-management tooling introduced in *Chapter 11*, particularly the `bitbake-layers` command and how `build/conf/bblayers.conf` controls which layers the build system sees.

No additional host packages are required beyond those listed in the Yocto Project 6.0 system requirements at `https://docs.yoctoproject.org/wrynose/ref-manual/system-requirements.html`.

Creating the layer

Before creating a custom layer, check whether an existing layer meets your requirements. A good starting point is the OpenEmbedded Layer Index at `https://layers.openembedded.org`. If no suitable layer is found, proceed to create your own. While layer names commonly begin with `meta-` by convention, this is not a strict technical requirement.

To streamline the creation process, use the `bitbake-layers create-layer` command from BitBake. Navigate to your desired directory and run the following command:

```
$ bitbake-layers create-layer meta-newlayer
```

BitBake scaffolds the layer and prints how to add it, as shown in the following figure.

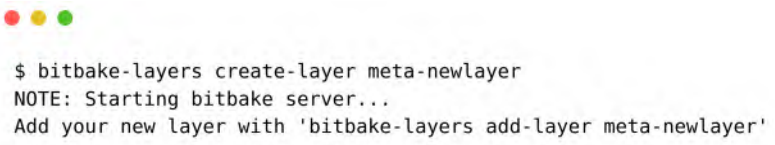


Figure 12.1: Example output from using the bitbake-layers create-layer command

Once the layer is created, it must be included in the `build/conf/bblayers.conf` file. Use the following command to add it:

```
$ bitbake-layers add-layer meta-newlayer
```

The command appends the layer to `build/conf/bblayers.conf` as shown below.

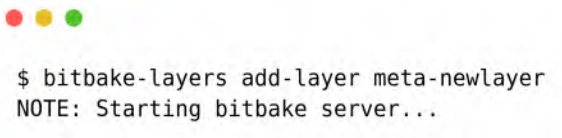


Figure 12.2: Including meta-newlayer in build/conf/bblayers.conf

Passing `--add-layer` (or `-a`) to bitbake-layers create-layer collapses both steps into one, creating the layer and registering it in `build/conf/bblayers.conf`. The steps are shown separately here because adding a layer is something you also do for layers you did not create.

The manual command works well for ad-hoc additions, though longer-lived projects often prefer to describe layer inclusion declaratively.

Tip

You can integrate custom layers into your BitBake Configuration Template files. Add the layer repository to the `sources` section and include its path in the `bb-layers` array. This approach keeps layer management declarative and version-controlled. See *Chapter 14* for detailed examples of custom BitBake Configuration Templates.

After that, you can verify that the layer has been included in the environment. The following command checks the currently active layers in your build environment:

```
$ bitbake-layers show-layers
```

The new layer appears at the bottom of the list with its assigned priority. The `openembedded-layer` entry (`meta-oe`) only shows up here if you followed the steps in *Chapter 11* to add that layer. If you skipped them, your output will not include it.

```
$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                 path                                      priority
==========================================================================
core                  .../layers/openembedded-core/meta  5
yoctobsp              .../layers/meta-yocto/meta-yocto-bsp  5
yocto                 .../layers/meta-yocto/meta-poky  5
openembedded-layer    .../layers/meta-openembedded/meta-oe  5
meta-newlayer         .../layers/meta-newlayer  6
```

Figure 12.3: Verification of meta-newlayer inclusion in the environment

As shown in the preceding figure, the `bitbake-layers` tool assigns the new layer a priority of 6. This is the tool's built-in default, used whenever `create-layer` runs without the `--priority` (or `-p`) option, and it ranks the layer above the priority 5 that the Poky layers carry, so its metadata takes precedence when two layers provide the same recipe. The scaffolding includes essential components like configuration files and example recipes, providing a reliable starting point. The generated directory structure is shown in the following figure:



Figure 12.4: Directory structure of meta-newlayer after creation

The default layer configuration file for `meta-newlayer` provides the basis for the layer to function effectively. It defines essential paths, file collections, and priorities. This configuration file can be extended to include more complex build rules, additional dependencies, or specific compatibility settings tailored to project requirements. The following figure illustrates the default content of the `conf/layer.conf` file 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 = "wrynose"
```

Commonly used variables to consider include `LAYERVERSION` and `LAYERDEPENDS`, which are particularly important if your layer depends on other layers. These variables must be suffixed with the layer's collection name, the identifier the layer declares in `BBFILE_COLLECTIONS`, which is not always the same as its directory name. Below there is a description of these variables:

- `LAYERVERSION`: An optional variable that defines the version of the layer using a single number. This versioning becomes particularly useful when specifying dependencies in `LAYERDEPENDS`, providing compatibility with specific versions of other layers. For

instance, you can specify `LAYERVERSION_meta-newlayer = "1"` to indicate the current version of the layer.

- `LAYERDEPENDS`: Defines the layers required by the recipes in your custom layer, enabling effective dependency management. Dependencies are listed as space-separated values, so that all necessary layers are included during the build process. Each entry is a collection name as well, so a dependency on OpenEmbedded Core is written `core`, not `meta` or `openembedded-core`, and a dependency on `meta-oe` is written `openembedded-layer`. The first column of the `bitbake-layers show-layers` output shown earlier lists the collection name of every active layer, which is the reliable way to look one up. To specify a required version of another layer, append the version number after a colon. For example, to depend on version 2 of `meta-otherlayer`, you can use `LAYERDEPENDS_meta-newlayer += "meta-otherlayer:2"`. If the specified dependencies are unavailable or the version numbers do not match, the build process will generate an error.
- `LAYERSERIES_COMPAT`: Declares which Yocto Project release codenames the layer is known to work against. For Yocto Project 6.0, set `LAYERSERIES_COMPAT_meta-newlayer = "wrynose"` so the build system warns when the layer is consumed by an incompatible release. Layers earning the Yocto Project Compatible v2 designation must keep this variable current with each release migration.

With the foundational structure of the layer now in place, the next steps involve exploring how to extend and customize it to align with specific project requirements.

Adding metadata to the layer

Layer metadata serves two primary purposes: adding new software or features and modifying existing metadata. These functionalities make it a powerful tool for customizing and extending the capabilities of a Yocto Project-based build system.

A new layer can include various metadata files, such as recipes, images, and `.bbappend` files. These elements allow developers to integrate new functionality or adjust existing configurations seamlessly.

For instance, `.bbappend` files, commonly found in layers like `meta-yocto-bsp` and `meta-poky`, enable developers to extend or modify existing recipes without directly editing them. This approach preserves the integrity of the base metadata while applying custom changes. *Chapter 13* explores some of the common use cases for `.bbappend` files in detail.

In the following sections, we will examine typical modifications to layer metadata, providing guidance on proven techniques for effective customization. *Chapter 18* collects the layer-organization and custom-distribution guidelines that complement the mechanics shown here, including when to split work across layers and how to keep a product layer maintainable as a project matures.

Creating an image

Image files are collections of packages grouped and configured to serve a specific purpose. You can either create an image from scratch or build upon an existing one, customizing it by adding or removing packages to meet specific requirements.

Reusing an existing image whenever possible simplifies code maintenance and clarifies functional differences. For instance, if you want to include an application and remove an image feature from the `core-image-full-cmdline` image file, you can achieve this by creating a new image recipe 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"
```

The `core-image` class provides a flexible framework of features that act as foundational building blocks for a variety of functionalities, making it a robust option for creating an image from scratch. For example, a new image can be defined in the `recipes-mine/images/my-image-strace.bb` file with the following configuration:

```
inherit core-image

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

When extending the feature list rather than replacing it, the operator choice keeps the customization composable.

Tip

The list appending operator (+=) allows additional values to be seamlessly appended to the `EXTRA_IMAGE_FEATURES` variable in `build/conf/local.conf`, enhancing flexibility in customization.

`CORE_IMAGE_EXTRA_INSTALL` is a fundamental variable for adding extra packages to an image when using the `core-image` class.

The `core-image` class enhances image creation by leveraging the `IMAGE_FEATURES` variable, which minimizes code duplication and optimizes customizations. Below is the list of `IMAGE_FEATURES` currently supported, as described in the *Image Features* section of the *Yocto Project Reference Manual* at `https://docs.yoctoproject.org/wrynose/ref-manual/features.html#image-features`:

- `allow-empty-password`: Allows Dropbear and OpenSSH to permit logins from accounts with empty password strings. This feature simplifies testing scenarios or development processes requiring easy access. Restrict it to development images. A device shipped with this feature accepts a remote login with no credentials at all.
- `allow-root-login`: Enables Dropbear and OpenSSH to permit root logins, which can be useful for administrative tasks or development purposes. Use with caution to maintain security.
- `dbg-pkgs`: Installs debug symbol packages for all packages in the image, enabling source-level debugging and analysis during development.
- `dev-pkgs`: Includes development packages, such as headers and additional library links, for all packages in the image. This feature is essential for enabling on-target software compilation and advanced development workflows.
- `doc-pkgs`: Adds documentation packages for every package included in the image, providing reference materials and guides essential for developers and maintainers. These are the `-doc` packages that carry each recipe's man pages, so `man ls` and similar lookups can be answered on the target itself. The pages need a reader to go with them: add `man-db` and `man-pages` to the image, because OpenEmbedded Core builds BusyBox without its `man` applet.
- `empty-root-password`: Permits root login with an empty password when this feature is included in `IMAGE_FEATURES`. Rather than setting an empty password directly, it

disables the enforcement mechanism for requiring a non-empty password. Like `allow-empty-password`, it belongs in development images only.

- `lic-pkgs`: Installs license-related packages for all included software in the image, providing compliance and simplifying license tracking for all components.
- `overlayfs-etc`: Configures the `/etc` directory to use overlayfs, enabling separation of device-specific information while supporting systems with read-only root filesystems for enhanced modularity and maintainability.
- `package-management`: Adds package management tools while maintaining the package manager database, allowing for dynamic package handling and updates in the image.
- `post-install-logging`: Captures post-install script executions during the first boot and logs them to `/var/log/postinstall.log`. To retain these logs across reboots, configure `VOLATILE_LOG_DIR` to `no` for persistent logging.
- `ptest-pkgs`: Installs ptest packages for all ptest-enabled recipes, facilitating streamlined testing and validation of software components.
- `read-only-rootfs`: Configures the root filesystem as read-only, enhancing security and system integrity by preventing modifications during runtime.
- `read-only-rootfs-delayed-postinsts`: Facilitates the execution of post-install scripts during the first boot, even when `read-only-rootfs` is enabled, so that necessary system adjustments can occur without compromising the read-only state.
- `serial-autologin-root`: Enables automatic root login on the serial console when paired with `empty-root-password`. While useful for development and testing, it poses significant security risks and should be used with extreme caution in production environments.
- `splash`: Enables a customizable splash screen during the boot process, controlled via the `SPLASH` variable, enhancing the user experience with visual feedback.
- `stateless-rootfs`: Generates a stateless image where configuration and state-related tasks are deferred to runtime, enabling flexibility and dynamic system initialization.
- `staticdev-pkgs`: Includes static development libraries (`.a` files) for all packages in the image, enabling on-target development of programs that link statically instead of against shared libraries.
- `hwcodecs`: Installs hardware acceleration codecs to enhance multimedia performance, enabling efficient processing for video playback, encoding, and other hardware-accelerated tasks.

- `nfs-server`: Sets up a Network File System (NFS) server, enabling file sharing across systems in a networked environment, facilitating centralized storage and access.
- `perf`: Provides powerful profiling tools such as `perf`, `systemtap`, and `LTTng`, enabling detailed performance analysis and system debugging for optimized development workflows.
- `ssh-server-dropbear`: Deploys the Dropbear minimal SSH server, a lightweight solution for secure remote access. It recommends `openssh-sftp-server` to provide compatibility with modern scp clients. To exclude this dependency, configure `BAD_RECOMMENDATIONS` variable appropriately.
- `ssh-server-openssh`: Installs the OpenSSH SSH server, offering a robust and full-featured solution for secure remote access. When both `ssh-server-dropbear` and this feature are included, OpenSSH takes precedence to provide enhanced functionality.
- `tools-debug`: Provides essential debugging tools such as `strace` and `gdb`, enabling developers to analyze system behavior and troubleshoot issues effectively.
- `tools-sdk`: Provides a comprehensive SDK tailored for on-device use, including necessary toolchains and libraries for software development and testing.
- `tools-testapps`: Installs a suite of testing tools designed for device diagnostics, including support for functionalities such as touchscreen debugging and hardware verification.
- `weston`: Installs Weston, the reference implementation of the Wayland compositor, providing a lightweight and efficient graphical environment for embedded systems.
- `x11`: Deploys the X server, enabling graphical interface support for applications and environments requiring basic display functionalities.
- `x11-base`: Deploys the X server with a streamlined, minimal configuration, providing essential graphical support while conserving system resources.
- `x11-sato`: Deploys the OpenedHand Sato graphical environment, a lightweight and user-friendly interface ideal for embedded devices with minimal display requirements.

The `allow-empty-password` features above, together with the `core/yocto/root-login-with-empty-password` fragment from *Chapter 14*, suit development images only. A production image that still needs `root` access should ship a real password instead. Inherit the `extrausers` class and set a pre-hashed password through `EXTRA_USERS_PARAMS`:

```
inherit extrausers

EXTRA_USERS_PARAMS = "usermod -p '${PASSWD}' root;"
```

Figure 12.5: Assigning a root password with extrausers

Generate the hash on your host with `mkpasswd -m sha256crypt` and store it in `PASSWD`. Escape every `$` in the hash as `\$` in that assignment, because the `extrausers` class inlines the value into a shell command where an unescaped `$5` or `$rounds` is read as a shell variable and silently replaced with nothing. A sha256crypt hash therefore reads `PASSWD = "\$5\$rounds=5000\$…"` in the metadata, and the single quotes around `${PASSWD}` in the example above are required for the same reason. Never hardcode a plaintext password in metadata. A shared default password is discouraged and, in some jurisdictions, prohibited by law, so prefer per-device provisioning for shipping products.

Adding a package recipe

OpenEmbedded Core includes several classes that simplify the development process for common tools and projects, providing abstraction for build systems such as Autotools, CMake, and Meson. A package recipe serves as an instruction set for BitBake to execute tasks like fetching, unpacking, patching, configuring, compiling, and installing applications, Linux kernel modules, or any other software provided by a project. For a comprehensive list of classes included in OpenEmbedded Core, refer to the *Classes* section in the *Yocto Project Reference Manual* at `https://docs.yoctoproject.org/wrynose/ref-manual/classes.html`.

A simple recipe that demonstrates explicit execution of the compile and install tasks is presented below:

```
DESCRIPTION = "Simple helloworld application"
SECTION = "examples"
LICENSE = "MIT"

LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/MIT;\
md5=0835ade698e0bcf8506ecda2f7b4f302"
SRC_URI = "file://helloworld.c"
S = "${UNPACKDIR}"

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

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

The `do_compile` and `do_install` code blocks define the shell commands required to build and install the resulting binary into the destination directory, represented by `${D}`. This directory serves as a temporary location within the `build/tmp/work/` path during the build process. The compile line passes `${CFLAGS}` and `${LDFLAGS}` explicitly because a direct `${CC}` invocation does not pick them up on its own, and dropping `${LDFLAGS}` leaves the binary without a `GNU_HASH` section, which the `ldflags` QA check reports as an error. For projects using Autotools, the `autotools` class can significantly reduce code duplication. The following example, extracted from the `meta/recipes-core/dbus-wait/dbus-wait_git.bb` recipe in OpenEmbedded Core, illustrates this approach:

```
SUMMARY = "A simple tool to wait for a specific signal over DBus"
HOMEPAGE = "http://git.yoctoproject.org/cgit/cgit.cgi/dbus-wait"
DESCRIPTION = "${SUMMARY}"
SECTION = "base"
LICENSE = "GPL-2.0-only"
LIC_FILES_CHKSUM = "file://COPYING;md5=b234ee4d69f5fce4486a80fdaf4a4263"

DEPENDS = "dbus"

SRCREV = "64bc7c8fae61ded0c4e555aa775911f84c56e438"
PV = "0.1+git"

SRC_URI = "git://git.yoctoproject.org/${BPN};branch=master;protocol=https"
UPSTREAM_CHECK_COMMITS = "1"

inherit autotools pkgconfig
```

Figure 12.6: Example content of the dbus-wait recipe

Inheriting the `autotools` class streamlines the build process by automatically providing the necessary code to handle the following tasks:

- Update and prepare the configure script code and related artifacts
- Refresh and optimize the `libtool` scripts
- Execute the configure script to prepare the build environment

- Compile the source code using `make`
- Install the compiled binaries and resources with `make install`

These concepts extend to other build tools like CMake and Meson, enabling consistent and efficient workflows across various systems. With each release, the Yocto Project introduces additional supported classes, further expanding compatibility with new build systems and reducing repetitive code.

Tip

When you author recipes like these in VS Code, the Yocto Project BitBake extension offers context-based completions for `inherit`, `require`, and `include` drawn from your actual layers, and *Ctrl+Click* jumps from a class name to its definition. *Chapter 3* covers the setup.

Automatically creating a base package recipe using devtool

As introduced in *Chapter 9*, `devtool` simplifies the process of creating a recipe for an existing project. The following command demonstrates how to generate a recipe automatically:

```
$ devtool add https://github.com/OSSystems/bbexample
```

`devtool` fetches the source and scaffolds a recipe under the workspace, as shown in the following figure.

```
$ devtool add https://github.com/OSSystems/bbexample
(...)

INFO: Fetching git://github.com/OSSystems/bbexample;protocol=https;branch=master...
(...)

INFO: Recipe .../build/workspace/recipes/bbexample/bbexample_git.bb has been automatically created;
further editing may be required to make it fully functional
```

Figure 12.7: Example command for generating a recipe using devtool for bbexample

Behind the scenes, `devtool` leverages `recipetool` to generate the recipe and populate it with initial metadata. This newly created recipe is placed in the `workspace` directory, which acts as a dedicated layer managed by `devtool`. To transfer the recipe file into the target layer, run the following `devtool` command:

```
$ devtool finish bbexample ../meta-newlayer/
```

`devtool` moves the recipe into the target layer and cleans up the workspace, as shown below.

```
$ devtool finish bbexample ../meta-newlayer/
NOTE: Starting bitbake server...
Loading cache: 100% |################################################################################| Time:
0:00:01
Loaded 3527 entries from dependency cache.
Parsing recipes: 100% |##############################################################################| Time:
0:00:00
Parsing of 2004 .bb files complete (2003 cached, 1 parsed). 3528 targets, 103 skipped, 0 masked, 0 errors.
Removing 1 recipes from the x86-64-v3 sysroot: 100% |#########################################| Time:
0:00:00
INFO: Updating SRCREV in recipe bbexample_git.bb
INFO: Moving recipe file to .../layers/meta-newlayer/recipes-bbexample/bbexample
INFO: Preserving source tree in .../build/workspace/attic/sources/bbexample.20260702164123
If you no longer need it then please delete it manually.
It is also possible to reuse it via devtool source tree argument.
```

Figure 12.8: Command to transfer the bbexample recipe into the meta-newlayer layer

The resulting `bbexample` recipe is generated as follows:

```
# Recipe created by recipetool
# This is the basis of a recipe and may need further editing to become 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"
SRCREV = "ece3cef9abc95cb7c931f9f27860102e43cc1d9"

# 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 = ""
```

The `devtool`-generated recipe provides a baseline but requires further refinement. Developers should meticulously review and enhance the recipe by adding necessary metadata, fine-tuning compilation options, and tailoring it to meet the unique demands of the project.

Adding support for a new machine definition

Creating a new machine definition for your Yocto Project build is a straightforward yet detailed task. While the process may seem simple, it requires careful consideration, especially when configuring the BSP layer. This involves verifying the compatibility of the bootloader, the Linux kernel, and hardware support drivers to provide proper functionality.

OpenEmbedded Core ships tune files for a wide range of architectures, including:

- *ARM*
- *ARM64*
- *LoongArch*
- *MIPS*
- *MIPS64*
- *PowerPC*
- *PowerPC 64*
- *RISC-V 32*
- *RISC-V 64*
- *x86*
- *x86-64*

ARM, *ARM64*, *x86*, *x86-64*, and *RISC-V 64* are tested at every milestone by the Yocto Project Autobuilder. *MIPS*, *PowerPC*, and *LoongArch* tunes ship in OpenEmbedded Core but are not currently exercised by the Autobuilder. *LoongArch* BSPs are provided by the community-maintained `meta-loongarch` layer. These represent the most commonly used architectures in embedded systems development.

Creating well-structured machine configuration files is vital for smooth integration and reliable operation of the target system. These files encapsulate hardware-specific details

required to generate images optimized for the target machine's capabilities. When defining a machine, the following variables play a significant role:

- `TARGET_ARCH`: Defines the machine's CPU architecture, such as *ARM*, *x86-64*, or other supported architectures, so that the build process aligns with the hardware capabilities.
- `PREFERRED_PROVIDER_virtual/kernel`: Specifies the Linux kernel provider (e.g., `linux-custom`) to replace the default `linux-yocto`, enabling the use of a specific build of the Linux kernel tailored to the target system. The `virtual/kernel` name is a provider target rather than a recipe name, and *Chapter 5* covers how BitBake resolves providers.
- `SERIAL_CONSOLES`: Specifies the serial consoles and their configurations, including device names and baud rates, so that proper communication and debugging capabilities are available during system operation.
- `MACHINE_FEATURES`: Specifies the hardware features of the machine, so that the appropriate software stack is included in the generated images to match the machine's capabilities and requirements.
- `KERNEL_IMAGETYPE`: Defines the format of the Linux kernel image to be used, such as `bzImage` or `Image`, so that compatibility with the target system's bootloader requirements is maintained.
- `IMAGE_FSTYPES`: Specifies the types of filesystem images to generate, such as `tar.gz`, `ext4`, or `ubifs`, so that compatibility with the target device's storage and boot requirements is achieved.

Machine definition files, which serve as templates for hardware-specific configurations, can be found in the `meta-yocto-bsp/conf/machine/` directory within the `meta-yocto-bsp` layer. These examples provide a valuable reference for creating or customizing your own machine configurations.

The `MACHINE_FEATURES` variable is critical for defining a new machine, as it specifies the hardware capabilities required for the system. Accurately defining these features means that the generated images include the necessary software components and drivers, optimizing functionality for the target device. Below is a detailed list of commonly used `MACHINE_FEATURES` values, along with their descriptions and associated keywords:

- `acpi`: Indicates that the hardware supports ACPI (Advanced Configuration and Power Interface). This is applicable to *x86* and *x86-64* architectures only.

- `alsa`: Specifies that the hardware includes ALSA (Advanced Linux Sound Architecture) audio drivers, enabling high-quality sound support.
- `bluetooth`: Confirms the presence of integrated Bluetooth (BT) hardware for wireless communication.
- `coresight`: Indicates support for the CoreSight Linux kernel feature on *ARM* processors. This feature depends on the `opencsd` recipe provided by the `meta-arm` layer.
- `efi`: Denotes support for booting through EFI (Extensible Firmware Interface), an essential feature for modern bootloaders.
- `ext2`: Indicates the hardware uses HDDs or Microdrives formatted with the EXT2 filesystem.
- `keyboard`: Confirms that the machine includes a physical keyboard as an input device.
- `numa`: Specifies that the hardware supports NUMA (Non-Uniform Memory Access), improving memory performance on certain architectures.
- `pcbios`: Identifies support for booting through a traditional PC BIOS.
- `pci`: Denotes the presence of a PCI (Peripheral Component Interconnect) bus for hardware expansion.
- `pcmcia`: Indicates the machine supports PCMCIA or CompactFlash sockets for external peripherals.
- `phone`: Specifies that the hardware supports mobile phone (voice) functionality.
- `qemu-usermode`: Highlights support for user-mode emulation in QEMU for this machine.
- `qvga`: Confirms the machine includes a QVGA (320x240) display, often used in compact devices.
- `rtc`: Indicates the presence of a RTC (Real-Time Clock) for maintaining time even when the device is powered off.
- `screen`: Confirms that the hardware is equipped with a screen for visual output.
- `serial`: Specifies support for serial communication, typically using an RS232 interface.
- `touchscreen`: Identifies the presence of a touchscreen interface for user interaction.
- `usbgadget`: Confirms that the hardware is capable of functioning as a USB gadget device.
- `usbhost`: Indicates that the hardware can act as a USB host, supporting peripheral connections.

- `vfat`: Specifies support for the FAT filesystem, commonly used for compatibility with external devices.
- `wifi`: Confirms the presence of integrated Wi-Fi hardware for wireless networking.

Accurately configuring `MACHINE_FEATURES` means that the necessary software components are integrated into the images, aligning them with the machine's specific hardware capabilities. This meticulous approach optimizes system compatibility and enhances overall performance.

Wrapping an image for your machine

Creating a ready-to-use image for a machine is a critical final step in the development of any BSP support layer. The type of image you create depends on the processor, peripherals included on the board, and specific project constraints.

Partitioned images are a widely used format for deploying directly to storage. The Yocto Project provides the `wic` tool for generating these images. Wic uses **OpenEmbedded Kickstart** files (`.wks`) to define the target image layout in a clear and adaptable format. These Kickstart commands, originally inspired by Fedora Kickstart, have been enhanced to leverage Wic's extended capabilities. The language definition can be found in the *OpenEmbedded Kickstart (.wks) Reference* section from the *Yocto Project Reference Manual* at `https://docs.yoctoproject.org/wrynose/ref-manual/kickstart.html#openembedded-kickstart-wks-reference`.

Yocto Project 6.0 requires `.wks` files to live under the `files/wic/` directory of your layer (older releases accepted a top-level `wic/` directory).

> **Note**
>
> This relocation is enforced. A build that still references `.wks` files from the legacy `wic/` path will fail rather than fall back to the previous location, so migrate every Kickstart file before the first Yocto Project 6.0 build of an upgraded layer.

With the new location in place, multiple files can coexist there to support various image layouts. It is crucial to verify that each `.wks` file's structure aligns with the target machine's requirements. For instance, an i.MX-based machine booting U-Boot from an SD card may require three partitions: one for the SPL, another for the U-Boot image file, and finally, one for the root filesystem. Below is an example of a `.wks` file configured for such a setup:

```
# 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 SPL and u-boot
#
# The disk layout used is:
# +--------+---------+----------+-----------------------------------------------+
# |        |   SPL   |  u-boot  |                   rootfs /                    |
# +--------+---------+----------+-----------------------------------------------+
# ^        ^         ^          ^
# |        |         |          |
# 0        1kiB      69kiB      4MiB + rootfs + IMAGE_EXTRA_SPACE (default 10MiB)
#
part SPL --source rawcopy --sourceparams="file=SPL" --ondisk mmcblk --no-table --align 1
part u-boot --source rawcopy --sourceparams="file=${UBOOT_BINARY}.tagged" --ondisk mmcblk --no-table --align 69
part / --source rootfs --ondisk mmcblk --fstype=ext4 --label root --align 4096
bootloader --ptable msdos
```

Figure 12.9: Example .wks file demonstrating partitioning for an i.MX device using SPL and U-Boot

To enable `wic`-based image generation, include `wic` in the `IMAGE_FSTYPES` variable and specify the desired `.wks` file by setting the `WKS_FILE` variable. Generating a `.wic` image also depends on the `wic-tools` native recipe, which BitBake resolves and builds automatically the first time you bake such an image, so the build itself needs no extra step. Properly configured `.wks` files, combined with the versatility of `wic`, allow you to create partitioned images that meet your machine's specific requirements, enabling efficient deployment and reliable functionality.

For the `qemux86-64` example used for the rest of this section, `WKS_FILE` is already covered by the machine configuration's default, so no additional configuration is required. Confirm the value with `bitbake-getvar`:

```
$ bitbake-getvar WKS_FILE
```

The variable history confirms the weak default comes from the machine configuration itself, as shown below.

```
$ bitbake-getvar WKS_FILE
NOTE: Starting bitbake server...
#
# $WKS_FILE
#   set? .../layers/openembedded-core/meta/conf/machine/qemux86-64.conf:36
#     "qemux86-directdisk.wks"
WKS_FILE="qemux86-directdisk.wks"
```

Figure 12.10: Confirming the default WKS_FILE for qemux86-64 with bitbake-getvar

Once an image is built, the standalone `wic` command-line tool lets you inspect the resulting `.wic` file directly, without mounting it. Make `wic` available in your shell by building `wic-native`, adding it to the recipe sysroot, and defining a local alias that runs it through `oe-run-native`. Confirm the alias works with `wic --help`:

```
$ bitbake wic-native -c addto_recipe_sysroot
$ alias wic="oe-run-native wic-native wic"
$ wic --help
```

The command confirms the alias resolves and lists Wic's supported subcommands, as shown below.



Figure 12.11: Confirming the wic alias and listing its subcommands with wic --help

Point `wic ls` at the deploy artifact to list its partitions:

```
$ wic ls build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-
qemux86-64.rootfs.wic
```

For a `core-image-full-cmdline` build targeting `qemux86-64`, the output reports two partitions: a FAT16 boot partition holding the bootloader and Linux kernel, and an ext4 root filesystem, as shown below.

```
$ wic ls build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs.wic
Getting sysroot...
Num     Start        End          Size      Fstype
 1          2048      62667      31037440  fat16
 2         63488    1019897     489681920  ext4
```

Figure 12.12: Listing the partitions of core-image-full-cmdline-qemux86-64.wic with wic ls

Append a colon and the partition number to look inside a specific partition. The following command lists the files in the boot partition:

```
$ wic ls build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-
qemux86-64.rootfs.wic:1
```

The output lists the boot partition's contents, as shown below.

```
$ wic ls build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs.wic:1
Getting sysroot...
Volume in drive : is boot
 Volume Serial Number is C494-BCE3
Directory for ::/

BZIMAGE       13968384 2011-04-05  23:00  bzImage
libcom32 c32    174144 2011-04-05  23:00
libutil  c32     22836 2011-04-05  23:00
syslinux cfg       201 2011-04-05  23:00
vesamenu c32     27052 2011-04-05  23:00
        5 files          14 192 617 bytes
                         16 553 984 bytes free
```

Figure 12.13: Listing the files inside the boot partition of core-image-full-cmdline-qemux86-64.wic

The `wic cp` and `wic rm` subcommands use the same colon-separated addressing to copy a file into a partition or remove one, which is useful for swapping in a freshly built Linux kernel without rerunning the entire image build. Run `wic help <subcommand>` for the full option list of any `wic` subcommand.

Using a custom distribution

Creating a custom distribution in the Yocto Project requires a careful balance between simplicity and functionality. While crafting a distribution configuration file is relatively straightforward, the changes introduced can have a significant impact on build behavior. Depending on the chosen options, these changes may lead to binary incompatibilities with previously built outputs. A distribution configuration file serves as a central location for defining global options such as:

- Toolchain versions (e.g., GCC, Clang) and the C library, selected with `TCLIBC` (`glibc`, `musl`, `newlib`, `picolibc`, or `baremetal`)
- Graphical backends (e.g., X11, Wayland).
- GPU capabilities (e.g., OpenGL, Vulkan).
- Initialization manager via `INIT_MANAGER`, which selects between `systemd` and `sysvinit`. From Yocto Project 6.0 onward, the `nodistro` setup defaults to `systemd`, while Poky and distributions derived from it continue to default to `sysvinit`. The two are mutually exclusive and cannot be enabled together because the systemd recipe no longer ships SysVinit compatibility shims.

Poky is a reference distribution, well suited to evaluation and testing but not intended to be the distribution a product ships with. For anything beyond experiments, define your own, as *Chapter 18* recommends. Doing so is usually inexpensive because a custom distribution often derives from Poky and changes only a few variables. For example, transitioning from X11 to a framebuffer-based setup can be accomplished by extending the Poky distribution and modifying the relevant parameters. Below is an example of a custom distribution file, typically located at `<layer>/conf/distro/my-distro.conf`:

```
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_OPTED_OUT = "wayland vulkan opengl"
```

The workspace created in *Chapter 2* already selects Poky through the `distro/poky` BitBake Configuration Fragment, and `DISTRO` is special-cased so that a fragment selection and a hard assignment in `build/conf/local.conf` raise a configuration error instead of one silently winning. Run `bitbake-config-build disable-fragment distro/poky` before setting `DISTRO`, or the next build stops with that error. Then add the following line to the `build/conf/local.conf` file:

```
DISTRO = "my-distro"
```

A custom distribution configuration can also carry project-wide licensing policy. Variables such as `INCOMPATIBLE_LICENSE`, `LICENSE_FLAGS_ACCEPTED`, and the archiver toggles described in *Chapter 15* apply to every build that selects this distribution, so the decision is enforced once instead of inside each recipe. That reach is also the catch for `INCOMPATIBLE_LICENSE`, since excluding a license family from every recipe often leaves dependencies unsatisfiable, a trade-off *Chapter 15* examines in detail.

The `DISTRO_FEATURES` variable plays a key role in determining how recipes are configured and which packages are included in images. For example, enabling sound functionality across all machines and images can be achieved by adding the `alsa` feature. Below is the complete list of supported values for `DISTRO_FEATURES`, as outlined in the *Distro Features* section of the *Yocto Project Reference Manual* at `https://docs.yoctoproject.org/wrynose/ref-manual/features.html#distro-features`:

- `3g`: Provides support for cellular data connectivity over 3G networks, enabling mobile internet access.
- `bluetooth`: Adds integrated support for Bluetooth wireless communication, enabling short-range connectivity for devices.
- `ipsec`: Adds support for Internet Protocol Security (IPsec), providing secure communication over IP networks.
- `ipv4`: Provides support for IPv4 networking, maintaining compatibility with legacy and current systems.
- `ipv6`: Enables IPv6 networking support, preparing systems for modern and future internet standards.
- `ldconfig`: Enables ldconfig support, which manages the shared library cache and allows dynamic linking at runtime.
- `lto`: Enables Link Time Optimization (LTO), allowing the compiler to perform whole-program optimization for improved performance.

- `minidebuginfo`: Adds minimal debug symbols to binary files to support symbolized stack traces, providing useful debugging information without the full overhead of complete debug packages.
- `multiarch`: Provides support for multiarch, enabling the installation of libraries for multiple architectures on the same system.
- `nfc`: Provides functionality for Near Field Communication (NFC), enabling contactless data exchange between devices.
- `nfs`: Provides NFS client functionality, allowing devices to mount and access remote NFS exports efficiently.
- `nls`: Provides Native Language Support (NLS), enabling internationalization and localization of applications and system messages.
- `opencl`: Adds support for the OpenCL (Open Computing Language) framework, enabling heterogeneous parallel computing across CPUs, GPUs, and other processors.
- `wifi`: Enables support for integrated Wi-Fi hardware, facilitating wireless network connectivity.
- `xattr`: Enables extended attribute support for filesystems, allowing storage of additional metadata beyond standard file attributes.
- `zeroconf`: Activates zero-configuration networking, simplifying network setup and device discovery.
- `alsa`: Provides support for the Advanced Linux Sound Architecture (ALSA), including compatibility with OSS modules when available.
- `opengl`: Incorporates support for OpenGL, enabling robust 2D and 3D graphics rendering across platforms.
- `pulseaudio`: Enables the PulseAudio sound server for advanced audio management and networked audio streaming.
- `vulkan`: Activates support for the Vulkan API, delivering high-performance 3D graphics and compute capabilities.
- `wayland`: Provides support for the Wayland display server protocol, offering a modern alternative to X11.
- `x11`: Incorporates support for the X Window System (X11) and its libraries, enabling traditional graphical interfaces.
- `cramfs`: Provides support for the Compressed ROM File System (CramFS), optimizing storage usage for embedded systems.

- `ext2`: Offers tools for managing devices with EXT2 filesystems, commonly used on internal HDDs or Microdrives.
- `overlayfs`: Provides support for OverlayFS, allowing efficient management of layered filesystems for dynamic storage needs.
- `vfat`: Enables FAT filesystem support, maintaining compatibility with external storage devices and legacy systems.
- `acl`: Enables Access Control Lists (ACLs) for granular file permission management and enhanced security.
- `pam`: Enables Pluggable Authentication Modules (PAM) to enhance security and streamline authentication processes.
- `selinux`: Activates Security-Enhanced Linux (SELinux) support for mandatory access control. Requires `meta-selinux`.
- `seccomp`: Provides secure computing mode (seccomp) support, enabling applications to limit the system calls they can perform.
- `smbfs`: Enables support for the SMB/CIFS filesystem, allowing mounting of Windows network shares and network-attached storage.
- `systemd`: Incorporates the systemd init system for efficient service management, parallel startup, and enhanced features.
- `systemd-resolved`: Enables systemd-resolved as the primary DNS resolver in glibc Name Service Switch, providing modern DNS resolution capabilities when systemd is used as the init system.
- `usrmerge`: Consolidates `/bin`, `/sbin`, `/lib`, and `/lib64` directories into `/usr` counterparts, enhancing compatibility and organization.
- `api-documentation`: Generates API documentation during recipe builds and incorporates it into SDK tarballs created with `bitbake -c populate_sdk`. Refer to the Yocto Project Application Development and eSDK manual for additional details.
- `debuginfod`: Allows retrieval of ELF debugging information from a debuginfod server, enhancing debugging efficiency.
- `directfb`: Provides support for DirectFB, a thin library that offers hardware graphics acceleration, input device handling, and abstraction layer for the Linux framebuffer device.
- `gobject-introspection-data`: Generates GObject introspection data, enabling language bindings and runtime introspection for GLib-based libraries.

- `ptest`: Activates package testing capabilities for supported recipes. Refer to the *Testing Packages With ptest* section in the *Yocto Project Test Environment Manual* at `https://docs.yoctoproject.org/wrynose/test-manual/ptest.html#testing-packages-with-ptest` for more details.
- `keyboard`: Facilitates keyboard input support, including keymap configuration for accurate input handling during boot.
- `pci`: Provides functionality for PCI bus interfaces, supporting hardware expansion and device communication.
- `pni-names`: Enables generation of persistent network interface names so the system maintains consistent and unique identifiers for network interfaces even after reinstallation. When this feature is enabled, systemd uses its default naming policy rather than forcing the "mac" policy.
- `polkit`: Enables PolicyKit support for fine-grained privilege management, allowing unprivileged processes to communicate with privileged ones.
- `ppp`: Provides support for the Point-to-Point Protocol (PPP), enabling dial-up and serial line network connections.

Properly configuring `DISTRO_FEATURES` means that the generated images are tailored to the desired system capabilities, matching software functionality with project requirements.

Yocto Project 6.0 added `multiarch`, `opengl`, `ptest`, `vulkan`, and `wayland` to `DISTRO_FEATURES_DEFAULTS`, alongside the previously default features (`acl`, `alsa`, `bluetooth`, `ipv4`, `ipv6`, `x11`, and others). To omit a default feature for a custom distribution, list it in `DISTRO_FEATURES_OPTED_OUT`. The same pattern applies to machine-level defaults via `MACHINE_FEATURES_DEFAULTS` and `MACHINE_FEATURES_OPTED_OUT`.

Adjusting default DISTRO_FEATURES and MACHINE_FEATURES

Because `DISTRO_FEATURES_DEFAULTS` and `MACHINE_FEATURES_DEFAULTS` apply automatically, a distribution or machine configuration carried over from an earlier release can pick up capabilities it never explicitly requested. `wayland`, `vulkan`, or `ptest` support can show up in an image that has no use for it, adding size and build time without a matching benefit.

Before shipping a migrated build, compare the resulting `DISTRO_FEATURES` and `MACHINE_FEATURES` values against what the project actually needs. `bitbake-getvar -r <recipe-name> DISTRO_FEATURES` (and the equivalent for `MACHINE_FEATURES`) prints the final,

expanded value, showing exactly which features defaulted in. List anything unwanted in `DISTRO_FEATURES_OPTED_OUT` or `MACHINE_FEATURES_OPTED_OUT` to keep it out of the build, following the same pattern already shown for the custom distribution configuration.

See the *Changes in How Default* `DISTRO_FEATURES` *and* `MACHINE_FEATURES` *Are Provided* section of the *Migration Notes for 6.0 (Wrynose)* page at `https://docs.yoctoproject.org/wrynose/migration-guides/migration-6.0.html#changes-in-how-default-distro-features-and-machine-features-are-provided` for the full details of this change.

Interaction between MACHINE_FEATURES and DISTRO_FEATURES

The `DISTRO_FEATURES` and `MACHINE_FEATURES` variables play complementary roles in defining the capabilities and features of the final system. These variables work together to verify that both the hardware and software aspects align with the desired functionality.

When a machine supports a feature, it does not necessarily guarantee that the feature will be operational on the target system. The distribution must also include the necessary base support for that feature. Conversely, if the distribution supports a feature but the machine does not, the software for that feature will be included but remain inoperative due to hardware limitations. To illustrate, consider the example of Wi-Fi support:

- **Machine supports Wi-Fi, distribution does not**: If a machine has hardware support for Wi-Fi but the distribution does not enable Wi-Fi, applications in the operating system will be built without Wi-Fi support. The result is a system where Wi-Fi is not operational, even though the hardware is capable.
- **Distribution supports Wi-Fi, machine does not**: If the distribution enables Wi-Fi but the machine lacks the necessary hardware, the modules and applications required for Wi-Fi will be included in the build. However, they will not function on the target machine due to the absence of hardware support.
- **Both machine and distribution support Wi-Fi**: When both the machine and distribution support Wi-Fi, the resulting system will include the necessary drivers, firmware, and software to enable Wi-Fi functionality, enabling seamless operation.

BitBake exposes that intersection directly. `COMBINED_FEATURES` holds the features present in both `DISTRO_FEATURES` and `MACHINE_FEATURES`, so of the three cases above only the last one puts a feature in it. Recipes use the variable to pull in support just when both sides agree. OpenEmbedded Core's `packagegroup-base` recipe does this for the Wi-Fi example, adding `packagegroup-base-wifi` through `bb.utils.contains('COMBINED_FEATURES', 'wifi', …)`. Recipes that inherit the `features_check` class can gate themselves on it as well, using

`REQUIRED_COMBINED_FEATURES`, `ANY_OF_COMBINED_FEATURES` and `CONFLICT_COMBINED_FEATURES`.

The separation of `DISTRO_FEATURES` and `MACHINE_FEATURES` is essential for maintaining clarity and modularity in system configuration. By independently defining hardware capabilities through `MACHINE_FEATURES` and software stack requirements via `DISTRO_FEATURES`, developers can achieve better adaptability to project needs, simplify system customization, and streamline the management of diverse configurations.

Understanding the scope of variables in BitBake

The BitBake metadata system is powered by a vast array of variables, each serving a specific purpose. Understanding the scope of these variables is critical for managing and customizing build processes effectively. *Chapter 5* introduced how configuration files and recipes feed metadata into BitBake. This section revisits that material from the angle of variable scope, separating variables into two primary types based on their location of definition:

Global variables

Variables defined in configuration files have a global scope. They are accessible across all recipes and are often referred to as configuration metadata. These variables influence the entire build process and are parsed in the following order:

1. `build/conf/toolcfg.conf`: This file stores BitBake Configuration Fragment selections, including `MACHINE` and `DISTRO` fragment settings. BitBake parses it before `build/conf/local.conf`, so a hard assignment in `build/conf/local.conf` wins for ordinary variables. `MACHINE` and `DISTRO` are special-cased, however: because a build can only ever target one machine and one distribution, a hard assignment to either in `build/conf/local.conf` conflicts with a matching fragment and raises a configuration error instead of letting one source silently win.
2. `build/conf/local.conf`: This file serves as a placeholder for temporary configuration settings needed during development
3. `<layer>/conf/machine/<machine>.conf`: Defines configurations specific to a target machine, such as hardware features, Linux kernel settings, and supported image types
4. `<layer>/conf/distro/<distro>.conf`: Specifies settings unique to a distribution, such as toolchain versions, graphical backends, initialization manager configurations

Recipe-specific variables

Variables defined within recipe files have a local scope, meaning their visibility is restricted to the specific recipe and its tasks during execution. These variables are used to tailor the behavior of individual recipes, enabling modular and recipe-specific customization.

By understanding and leveraging the scope of variables effectively, developers can achieve precise control over their builds, enabling optimized and customized solutions tailored to project requirements.

Summary

In this chapter, we explored the process of creating a new layer and its associated metadata. We began by learning how to define a machine configuration, create a distribution definition, and write recipe files. Subsequently, we examined the steps for crafting images and integrating applications into those images.

In the next chapter, we will explore examples of common customization scenarios involving additional layers. These include modifying existing packages, adding extra build options to recipes, applying new patches, and including additional files in packages. Additionally, we will examine how to configure BusyBox and `linux-yocto`, two packages frequently customized in embedded systems development.

13

Customizing Existing Recipes

When a recipe from a third-party layer almost fits your requirements but needs one compilation flag changed, one feature toggled, or one extra patch applied, forking the entire recipe into your own layer is a tempting but costly move. Every fork becomes technical debt: upstream version bumps must be tracked and merged manually, and the original rationale for each change tends to dissolve the moment the engineer who made it moves on to a different project. The Yocto Project provides a structured alternative through `.bbappend` files and configuration fragments, which let you record targeted modifications against the original recipe without touching the upstream source.

This chapter walks through the practical techniques that keep those modifications clean and traceable. You will see how to adjust compilation options for Autotools, CMake, Meson, and other build systems, toggle package features with `PACKAGECONFIG`, attach additional patches to a recipe, and supply configuration fragments that extend a recipe's behavior without rewriting it. Each technique follows the same discipline: the change lives in your layer, the reason for it stays visible in the metadata, and the upstream recipe continues to evolve independently. Engineers who apply these methods spend less time untangling merge conflicts and more time on the design decisions that actually require their attention.

We will cover the following main topics:

- Understanding common use cases
- Extending a task
- Adding extra build options to recipes
- Applying a patch
- Adding extra files to existing packages
- Understanding file search paths

- Changing recipe feature configuration
- Configuration fragments for Kconfig-based projects

Technical requirements

This chapter requires a build host configured as described in *Chapter 2*, including a working BitBake Setup workspace with all host packages installed.

You also need a custom layer to hold the `.bbappend` files and configuration fragments introduced here. If you have not already created one, work through *Chapter 12* before proceeding. The examples assume the layer is registered in `build/conf/bblayers.conf` and that your workspace is otherwise ready to build.

Understanding common use cases

Modern projects often rely on multiple layers to provide the required features. Tailoring these layers to specific requirements can involve a variety of changes, from minor tweaks to substantial functional modifications. Regardless of the scale, the process of implementing these changes follows a consistent and well-defined structure.

To customize an existing recipe, you create a `.bbappend` file within your project layer, the kind of layer built in *Chapter 12*. For example, if the original recipe is located at `<original-layer>/recipes-core/app/app_1.2.3.bb`, your `.bbappend` file acts as an extension to that recipe, facilitating additional modifications. The `.bbappend` naming convention supports the `%` wildcard character, allowing you to define the scope of your changes. For instance:

- `app_1.2.3.bbappend`: Targets changes specifically to version `1.2.3` of the recipe
- `app_1.2.%.bbappend`: Extends changes to all `1.2.y` versions
- `app_1.%.bbappend`: Covers changes across all `1.x` and `1.x.y` versions
- `app_%.bbappend`: Broadly applies changes to any version of the recipe

You can create multiple `.bbappend` files for a single recipe, allowing granular and modular changes based on their scope and purpose. When multiple `.bbappend` files exist, BitBake sorts them by the `BBFILE_PRIORITY` of the layer that provides each one and applies them from the lowest priority to the highest. The append from the highest-priority layer is parsed last, so its assignments take effect last and take precedence over the ones before it.

The `.bbappend` file extends the original recipe by appending your modifications without altering the original `.bb` file. From a practical perspective, BitBake treats all `.bbappend` files for a recipe as extensions to the original recipe. It processes these files by concatenating their

content in the order defined by layer priorities, effectively creating a unified logical recipe. This approach integrates changes efficiently, avoiding duplication of source code while maintaining a clean and organized project structure.

Before trusting that a `.bbappend` took effect, confirm which append files BitBake actually matched. Running `bitbake-layers show-appends` without a recipe name reports every `.bbappend` applied across the workspace, grouped by the recipe it targets and the layer that provides it, in the same priority order BitBake used to concatenate them, as shown below. Pass a recipe name instead to narrow the report to that single recipe.

```
$ bitbake-layers show-appends
```

The command lists every matched append grouped by recipe and layer, as shown in the following figure.



Figure 13.1: bitbake-layers show-appends output listing every applied .bbappend in the workspace

With the applied appends confirmed, you can move on to writing the modifications themselves, starting with extending an existing task.

Tip

While you edit `.bbappend` files in VS Code, the Yocto Project BitBake extension parses each file on save and reports syntax errors in the Problems panel, so a malformed append surfaces before you start a build. *Chapter 3* covers the setup.

Extending a task

When the existing task content does not align with specific requirements, developers can either replace it entirely with a custom implementation or extend it by appending additional functionality. This adaptability exemplifies one of the key strengths of the BitBake build system.

BitBake offers the `:prepend` and `:append` operators, which enable developers to enhance tasks by adding content either at the beginning or the end of the task's original implementation. These operators, along with the `OVERRIDES` mechanism that drives the file search paths discussed later in this chapter, are part of the metadata syntax covered in *Chapter 5*. The matching `:remove` operator removes every occurrence of a value from a space-separated list, which is the right tool when a `.bbappend` needs to drop an entry from a variable like `IMAGE_INSTALL` without rewriting the rest of the list. These operators simplify customization while maintaining the integrity of the original logic. For example, to extend a `do_install` task, you can use the following code:

```
do_install:append() {
    # Do my commands
}
```

This approach concatenates the additional commands to the original `do_install` task, preserving its existing functionality while augmenting it with your custom steps. When an extension breaks the build, the task logs and Development Shell techniques in *Chapter 10* show what the appended commands actually executed inside the recipe sysroot.

When more than one layer appends the same task, the `:append` blocks run in the order BitBake parsed the files, from the lowest-priority layer to the highest. The block from the highest-priority layer therefore runs at the end of the assembled task.

That merged result is not hidden from view either. Once BitBake has run the task, the concatenated `do_install` lands under `${WORKDIR}/temp/`, typically `build/tmp/work/<arch>/`

`app/1.2.3/temp/` for this example, where `run.do_install.<PID>` reflects exactly what the original task and the `:append` block above assembled into a single script:

```
$ ls -1 build/tmp/work/<arch>/app/1.2.3/temp/
```

This listing confirms the `run.do_install.<PID>` script assembled from the original task and the `:append` block, as shown in the following figure.



Figure 13.2: Listing the temp folder for the concatenated do_install task

Adding extra build options to recipes

Assume you have an Autotools-based application with a preexisting recipe, and you want to enable `my-feature` and disable `another-feature`. The content of the `.bbappend` file to achieve these changes would be as follows:

```
EXTRA_OECONF += "--enable-my-feature --disable-another-feature"
```

The same strategy can be applied conditionally. For instance, to enable or disable features depending on the target architecture, you can use conditional logic in your `.bbappend` file as follows:

```
EXTRA_OECONF:append:arm = " --enable-my-arm-feature"
```

The same `:append` pattern extends to other build systems by swapping in their respective variable. For a CMake-based recipe, enabling `ENABLE_MY_FEATURE` and disabling `ENABLE_ANOTHER_FEATURE` looks like this:

```
EXTRA_OECMAKE += "-DENABLE_MY_FEATURE=ON -DENABLE_ANOTHER_FEATURE=OFF"
```

Meson-based recipes follow the identical pattern through `EXTRA_OEMESON`, passing `-Dmy-feature=enabled -Danother-feature=disabled` instead of CMake's `-D<VAR>=<VALUE>` cache entries. Only the flag syntax changes from one build system to the next. The append mechanism itself, appending the correct `EXTRA_OE*` variable in a `.bbappend` file, stays the same.

The Yocto Project supports a variety of build systems, each with specific variables for customization. These variables are summarized in the following table:

Build system	**Variable**
Autotools	`EXTRA_OECONF`
Cargo	`CARGO_BUILD_FLAGS`
CMake	`EXTRA_OECMAKE`
Make	`EXTRA_OEMAKE`
Meson	`EXTRA_OEMESON`
NPM	`EXTRA_OENPM`
SCons	`EXTRA_OESCONS`
WAF	`EXTRA_OEWAF_BUILD` / `EXTRA_OEWAF_INSTALL`

Table 13.1: Variables used to configure various build systems

Although the table lists `CARGO_BUILD_FLAGS` for completeness, Cargo recipes do not expose a single `EXTRA_OE*` knob for feature toggling. Use `PACKAGECONFIG` or `CARGO_FEATURES` to control which Cargo features are enabled, and reserve `CARGO_BUILD_FLAGS` for flags passed to `cargo build`.

Applying a patch

Applying a patch to an existing recipe requires the use of `FILESEXTRAPATHS`. This variable directs the search algorithm to include additional directories so that BitBake can locate and apply the necessary file. Here is an example of its implementation:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}-${PV}:"
SRC_URI += "file://mypatch.patch"
```

In this example, `THISDIR` specifies the current directory, while `PN` and `PV` represent the package name and version, respectively. The immediate expansion operator `:=` is required here rather than a plain `=`, because `THISDIR` names the directory of the file BitBake is parsing at that moment and has to be resolved while the `.bbappend` is being read. A deferred assignment leaves `${THISDIR}` unexpanded until something reads `FILESEXTRAPATHS` later in the parse, when it no longer points at the append file's directory and the patch is not found. The trailing colon is equally deliberate, since it preserves the existing search path that the new directory is prepended to. Using the `:prepend` operator to modify `FILESEXTRAPATHS` prioritizes files in this directory over those with the same name in lower-priority layers. BitBake automatically recognizes files with `.patch` or `.diff` extensions as patches and incorporates them into the build process. This streamlined mechanism applies patches consistently without requiring manual intervention. Patches are easy to add and expensive to carry, so the guidance in *Chapter 18* on keeping the Linux kernel and bootloader patch sets small applies to every other recipe too.

Adding extra files to existing packages

Including an additional configuration file in a package requires using the `FILESEXTRAPATHS` variable, as demonstrated in the example below:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}-${PV}:"
SRC_URI += "file://newconfig.conf"

do_install:append() {
    install -D -m 644 ${UNPACKDIR}/newconfig.conf \
        ${D}${sysconfdir}/newconfig.conf
}
```

The `do_install:append` function appends the specified commands to the original `do_install` function, enhancing its behavior while preserving existing functionality. This function copies

the new configuration file into the package's filesystem. The file is transferred from `${UNPACKDIR}` to `${D}`, where `${UNPACKDIR}` represents the directory where BitBake places files added via `file://` in `SRC_URI`, and `${D}` is the destination directory for creating the package. Earlier releases unpacked these files directly into `${WORKDIR}`. Since Yocto Project 5.1, the unpack step writes them to `${UNPACKDIR}` instead, so a `.bbappend` that still references `${WORKDIR}` for a `file://` source will not find it.

Recipes frequently rely on predefined path variables like `bindir`, `datadir`, and `sysconfdir`, which are defined in the `meta/conf/bitbake.conf` file within OpenEmbedded Core. These variables enable flexible and consistent installation paths, making it easier to address specific use cases, such as avoiding conflicts between native SDK binaries and target binaries.

Below is a table summarizing the most used variables and their default expanded values for reference:

Variable	**Default expanded value**
`base_prefix`	empty
`prefix`	`/usr`
`exec_prefix`	`/usr`
`root_prefix`	empty, or `/usr` when `usrmerge` is enabled
`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`

Variable	Default expanded value
includedir	/usr/include

Table 13.2: Commonly used variables and their default expanded values

Avoiding hardcoded paths in recipes significantly reduces the risk of misconfiguration while promoting greater flexibility. For example, when the usrmergeDISTRO_FEATURE is enabled, bitbake.conf rebases root_prefix to ${exec_prefix}, so base_bindir resolves to /usr/bin and coincides with bindir. Using a hardcoded path like /bin in a recipe can lead to installation failures due to unexpected path conflicts. By relying on predefined variables, recipes remain adaptable, reliable, and capable of handling diverse configurations effectively.

Understanding file search paths

When a file, such as a patch or a generic file, is referenced in SRC_URI, BitBake locates it using the FILESPATH and FILESEXTRAPATHS variables. By default, it searches in the following locations:

1. <recipe>-<version>/
2. <recipe>/
3. files/

BitBake extends that default set using FILESOVERRIDES, a colon-separated subset of OVERRIDES reserved for file searches and defined by default as ${TRANSLATED_TARGET_ARCH}:${MACHINEOVERRIDES}:${DISTROOVERRIDES}. It splits the value on :, reverses it so the most specific override is tried first, then appends each entry to every base directory in turn. The override drives the outer loop and the base directory the inner one, so BitBake exhausts all three base directories for one override before moving to the next. For instance, with the foo_1.0.bb recipe and FILESOVERRIDES set to <arch>:<board>, BitBake searches these directories in order, reading each row from left to right:

Override Segment	Base foo-1.0/ (from BP)	Base foo/ (from BPN)	Base files/
<board>	foo-1.0/<board>/	foo/<board>/	files/<board>/
<arch>	foo-1.0/<arch>/	foo/<arch>/	files/<arch>/

Override Segment	Base foo-1.0/ (from BP)	Base foo/ (from BPN)	Base files/
None	`foo-1.0/`	`foo/`	`files/`

Table 13.3: Directory search order produced by FILESPATH and FILESOVERRIDES

Reversing the list is what makes `<board>` outrank `<arch>`: `MACHINEOVERRIDES` sits after `TRANSLATED_TARGET_ARCH` in the default value, so it ends up ahead once the order is flipped. A file of the same name in `foo-1.0/<board>/` therefore wins over one in `files/`.

The `OVERRIDES` variable dynamically adjusts to the build environment by including factors such as the target machine and distribution. This adaptability allows BitBake to select the most appropriate configuration for a given build. To view the complete list of overrides for a specific machine, run the `bitbake-getvar OVERRIDES` command, an essential debugging tool that reveals the override structure of the current environment:

```
$ bitbake-getvar OVERRIDES
```

The command prints the queued operations and the fully expanded value, as shown in the following figure.

```
$ bitbake-getvar OVERRIDES
NOTE: Starting bitbake server...
#
# $OVERRIDES [2 operations]
#   set .../layers/openembedded-core/meta/conf/bitbake.conf:800
#     "${TARGET_OS}:${TRANSLATED_TARGET_ARCH}:pn-${PN}:layer-${FILE_LAYERNAME}:${MACHINEOVERRIDES}:
#     ${DISTROOVERRIDES}:${CLASSOVERRIDE}:${TCOVERRIDE}${LIBCOVERRIDE}:forcevariable"
#   set .../layers/openembedded-core/meta/conf/
#       documentation.conf:298
#     [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-${PN}:layer-${FILE_LAYERNAME}:${MACHINEOVERRIDES}:
#   ${DISTROOVERRIDES}:${CLASSOVERRIDE}:${TCOVERRIDE}${LIBCOVERRIDE}:forcevariable"
OVERRIDES="linux:x86-64:pn-defaultpkgname:layer-config:qemuall:qemux86-64:poky:class-target:\${TCOVERRIDE}:
libc-glibc:forcevariable"
```

Figure 13.3: Example output from bitbake-getvar showing the structure of the OVERRIDES variable

Notice that `${TCOVERRIDE}` remains unexpanded in that output. BitBake only expands a variable reference when the variable is defined somewhere in the datastore, even as an empty string. A reference to a variable that was never set at all, like `TCOVERRIDE` on this machine, is left as a literal token instead of collapsing to nothing.

Changing recipe feature configuration

The `PACKAGECONFIG` mechanism streamlines the customization of feature sets in recipes. It allows developers to dynamically enable or disable specific recipe features based on build requirements. The following is an example of a `PACKAGECONFIG` 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"
```

In this example, each feature's behavior is defined using six comma-separated arguments. Although some arguments can be left blank, the commas must remain in place to maintain the structure. These arguments define the following:

1. Arguments to pass when the feature is enabled
2. Arguments to pass when the feature is disabled
3. Build-time dependencies (`DEPENDS`) required when the feature is enabled
4. Runtime dependencies (`RDEPENDS`) required when the feature is enabled
5. Runtime recommendations (`RRECOMMENDS`) to include when the feature is enabled
6. Conflicting (mutually exclusive) `PACKAGECONFIG` settings for the feature

The `curl` recipe in OpenEmbedded Core shows how those fields work in a shipping recipe, as shown below.

```
COMMON_PACKAGECONFIG = "basic-auth bearer-auth digest-auth ipfs negotiate-auth openssl proxy threaded-resolver
verbose zlib"
PACKAGECONFIG ??= "${COMMON_PACKAGECONFIG} ${@bb.utils.filter('DISTRO_FEATURES', 'ipv6', d)} aws libidn"

# 'ares' and 'threaded-resolver' are mutually exclusive
PACKAGECONFIG[ares] = "--enable-ares,--disable-ares,c-ares,,,threaded-resolver"
PACKAGECONFIG[threaded-resolver] = "--enable-threaded-resolver,--disable-threaded-resolver,,,,ares"
```

Figure 13.4: PACKAGECONFIG in the OpenEmbedded Core curl recipe

Three details in that excerpt are worth noting. First, the default value is not a fixed list, because `bb.utils.filter` adds `ipv6` only when the distribution declares it in `DISTRO_FEATURES`. Second, `ares` names `c-ares` in the third field, so selecting that feature also adds the library to `DEPENDS`. Third, `ares` and `threaded-resolver` each name the other in the sixth field, because `curl` resolves hostnames through one mechanism or the other. The `threaded-resolver` entry leaves the third, fourth, and fifth fields empty and keeps the commas, which is the structure the list above describes.

A `.bbappend` file is a convenient way to expand the default value of `PACKAGECONFIG` and enable additional features without modifying the original recipe. The symmetric `PACKAGECONFIG:remove` operator drops a feature that the original recipe enabled by default, which is the cleanest way to undo a default selection from a distribution layer. For instance, to enable `feature2`, the following content can be included in a `.bbappend` file:

```
PACKAGECONFIG:append:pn-<recipename> = " feature2"
```

To add the same feature directly in the `build/conf/local.conf` file, use the following syntax, where <recipename> is the name of the recipe to adjust:

```
PACKAGECONFIG:pn-<recipename>:append = " feature2"
```

The order of the two suffixes matters. Writing `PACKAGECONFIG:append:pn-<recipename>` appends to the value the recipe already has, while `PACKAGECONFIG:pn-<recipename>:append` builds a separate pn- variable that replaces `PACKAGECONFIG` outright once the override applies, silently dropping every feature the recipe enabled by default. Local configuration is the quickest place to test the change, but the same selector can live higher up the configuration stack when the choice is meant to apply to every build of a distribution.

> **Tip**
>
> The same syntax can be applied in your custom distribution configuration file (`<layer>/conf/distro/<distro>.conf`) to reduce the number of `.bbappend` files required, one of the practices covered in *Chapter 18*.

To discover the available `PACKAGECONFIG` features for a specific package, you need to examine the recipe file directly. At present, no automated tool exists to list all the `PACKAGECONFIG` options.

Configuration fragments for Kconfig-based projects

The Kconfig configuration infrastructure is widely regarded for its flexibility and expressiveness. Originally designed for the Linux kernel, it has since been adopted by other projects like U-Boot and BusyBox. Kconfig's feature-centric design enables developers to toggle specific features and save configurations for reuse, making it a powerful tool for customization.

Kconfig enables control over features like the TFTPD support in BusyBox. For instance, we can toggle TFTPD support as shown below:



Figure 13.5: Enabling or disabling TFTPD support in BusyBox Kconfig

We can decide whether to enable (a) or disable (b) the TFTPD support in BusyBox, providing flexibility based on project requirements.

The Yocto Project includes a dedicated class for managing Kconfig-based projects, enabling the use of configuration fragments to toggle features. These fragments are particularly valuable for tailoring configurations to specific machines or use cases. For instance, to modify the Linux kernel configuration, you can use a `.bbappend` file located at `<layer>/recipes-kernel/linux/linux-yocto_%.bbappend`. Below is an example:

```
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"
SRC_URI += "file://enable-can.cfg"
```

Every configuration fragment must use a `.cfg` extension. For example, the `<layer>/recipes-kernel/linux/linux-yocto/enable-can.cfg` file might contain the following content:

```
CONFIG_CAN=y
```

Figure 13.6: Example content of enable-can.cfg

BitBake offers commands to manage and customize the Linux kernel configuration files. By running the `bitbake virtual/kernel -c menuconfig` command, you can launch the interactive `menuconfig` interface to manually configure the Linux kernel. This interface provides an intuitive way to enable features like the CAN bus subsystem, as illustrated here.

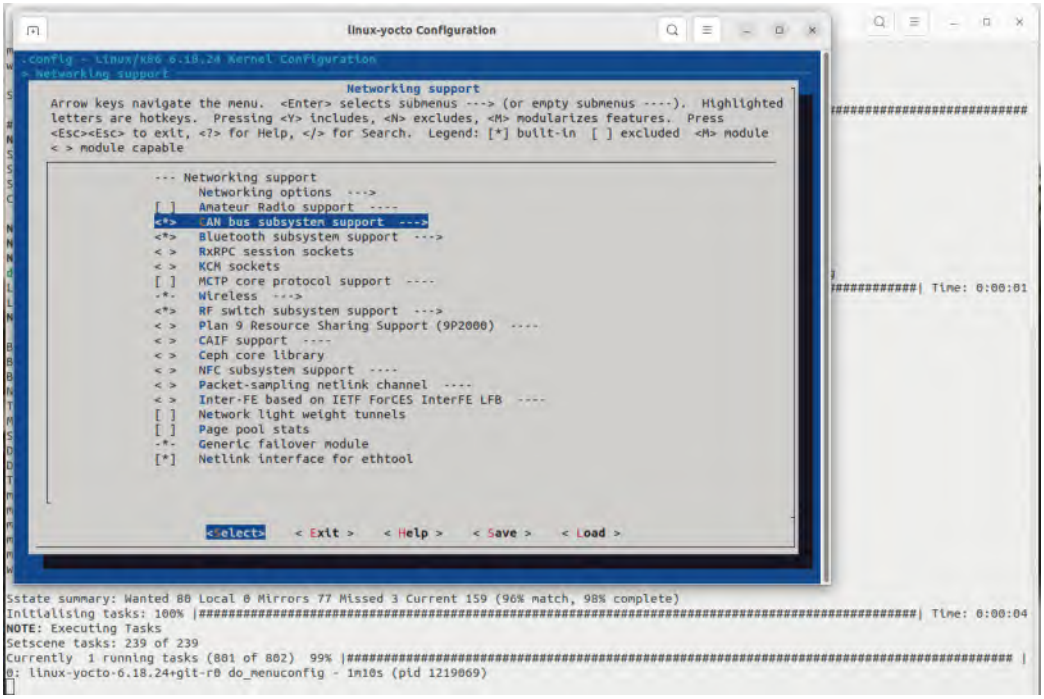


Figure 13.7: Enabling CAN bus subsystem support with menuconfig

Changes made in `menuconfig` are saved upon exiting and applied to the Linux kernel configuration. To create a configuration fragment based on differences from the default configuration, run the following command:

```
$ bitbake virtual/kernel -c diffconfig
```

The task reports where it wrote the fragment, as shown in the following figure.

```
$ bitbake virtual/kernel -c diffconfig
(...)

NOTE: Executing Tasks
Config fragment has been dumped into:
 .../build/tmp/work/qemux86_64-poky-linux/linux-yocto/6.18.24+git/fragment.cfg
NOTE: Tasks Summary: Attempted 456 tasks of which 455 didn't need to be rerun and all succeeded.
NOTE: Writing buildhistory
NOTE: Writing buildhistory took: 1 seconds
```

Figure 13.8: Generating a configuration fragment using diffconfig

The `do_diffconfig` task writes the fragment to `${WORKDIR}/fragment.cfg`, which is the recipe's work directory under `build/tmp/work/` in a default build, and the absolute path is printed in the log. Copy this fragment to your layer and reference it in a `.bbappend` file to apply it.

For a complete Linux kernel configuration, use the `bitbake virtual/kernel -c savedefconfig` command. This generates a `defconfig` file that replicates the entire configuration, not just a fragment.

Support for configuration fragments is available for the Linux kernel, U-Boot, BusyBox, and barebox.

Note

Yocto Project 6.0 (Wrynose) also adds two classes for Rust-based Linux kernel work: the `kernel-yocto` class enables in-tree Rust support when you set `KERNEL_FEATURES += "rust"`, and the new `kernel-yocto-rust` class adds the toolchain and dependencies required to build the Rust components of the Linux kernel. The `module-rust` class builds out-of-tree Rust Linux kernel modules with the same conventions as their C counterparts. Enabling Rust in the Linux kernel disables `ccache` for the Linux kernel build, so expect longer compile times during initial enablement. For details on out-of-tree module workflows, see the *Incorporating Out-of-Tree Modules* section at `https://docs.yoctoproject.org/wrynose/kernel-dev/common.html#incorporating-out-of-tree-modules`.

Summary

In this chapter, we explored how to customize existing recipes using `.bbappend` files, highlighting their utility in preventing source code duplication. We covered enabling or disabling features, applying patches, and leveraging configuration fragment support for flexible and efficient recipe management.

The next chapter returns to the workspace created in *Chapter 2* and covers the two tools that manage it after the first build. It walks through configuration discovery and fragment handling with `bitbake-config-build`, layer management and environment synchronization with `bitbake-setup`, and the team workflow patterns that keep builds reproducible across workstations.

14

Managing BitBake Configuration

The `bitbake-setup` tool introduced in *Chapter 2* handles project initialization with sensible defaults. This chapter covers configuration discovery, fragment management, environment synchronization, and team workflow patterns that help you maintain reproducible builds across developer workstations. Build environments may diverge between machines and teams without a structured approach to configuration, making failures hard to reproduce and root-cause analysis far harder than it needs to be.

The value of this structure shows up during maintenance rather than during the first build. A product that ships today may need a security fix eighteen months later, long after the original developer moved on. Reproducing that exact build means knowing which layer revisions, which machine and distribution settings, and which BitBake version produced it. A configuration you can read, pin, and hand to a colleague answers those questions. A build directory that grew by hand does not.

We will cover the following main topics:

- Exploring available configurations
- Managing build configuration with `bitbake-config-build`
- Creating custom bitbake configuration fragments
- Managing layers with `bitbake-setup`
- Keeping your setup current
- Reproducible team workflows

Technical requirements

This chapter builds on the host setup and BitBake Setup workspace introduced in *Chapter 2*. You need a host machine configured per that chapter and an initialized build directory to run the commands shown here.

Both `bitbake-config-build` and `bitbake-setup` ship with BitBake as part of the Yocto Project 6.0 (Wrynose) release. No additional installation is required.

Exploring available configurations

Before initializing a new project, review the configurations available in the default BitBake Registry:

```
$ bitbake-setup list
```

The command prints each available BitBake Configuration Template as shown below.

```
$ bitbake-setup list
Available configurations:
oe-nodistro-master      OpenEmbedded - 'nodistro' basic configuration
oe-nodistro-wrynose     OpenEmbedded - 'nodistro' basic configuration, release 6.0 'wrynose' (supported until
2030-05-31)
poky-master     Poky - The Yocto Project testing distribution configurations and hardware test platforms
poky-wrynose    Poky - The Yocto Project testing distribution configurations and hardware test platforms,
release 6.0 'wrynose' (supported until 2030-05-31)

Run 'init' with one of the above configuration identifiers to set up a build.
```

Figure 14.1: Listing available BitBake Configuration Templates

The output shows BitBake Configuration Templates, their descriptions, and expiration dates for time-limited releases. A **BitBake Registry** is a storage location (Git repository, local directory, or remote URI) where `bitbake-setup` searches for configuration files. This default BitBake Registry includes configurations for the Yocto Project and OpenEmbedded Core reference distributions across supported release branches.

Tip

Add `--with-expired` to include configurations for releases that have reached end-of-life. This can help when maintaining legacy products, though those configurations no longer receive updates. An end-of-life release receives no security fixes, so plan a migration to a supported release and treat the expired configuration as a way to reproduce an old build rather than a base for new work.

The list only shows what is available. To start a new project elsewhere on your host, run `bitbake-setup init` with one of these configuration identifiers, as the command's own output suggests. The rest of this chapter instead continues with the `poky-wrynose` project you already initialized in *Chapter 2*, so keep that build directory at hand. Either way, the next step is to customize build settings through fragment management.

Managing build configuration with bitbake-config-build

The `bitbake-config-build` tool manages BitBake Configuration Fragment selections for your build environment. A BitBake Configuration Fragment is a modular configuration unit that defines a specific aspect of the build, such as the target machine (`machine/qemux86-64`) or distribution (`distro/poky`). These fragments provide a structured way to configure builds without directly editing configuration files.

When you enable a fragment, the tool updates `build/conf/toolcfg.conf` with the corresponding settings. This sets the BitBake Configuration Instance, which represents the active build state derived from your BitBake Configuration Template and fragment selections.

Listing available fragments

`bitbake-config-build` operates on the active build environment, so your shell needs the `init-build-env` script sourced, as *Chapter 2* shows, before the tool has anything to work with. With that environment active, see which fragments are available based on your enabled meta layers using the `list-fragments` subcommand:

```
$ bitbake-config-build list-fragments
```

The command reports each fragment identifier with its description, as shown below.

```
$ bitbake-config-build list-fragments
NOTE: Starting bitbake server...
Available built-in fragments:
machine/...     Sets MACHINE = ...
distro/...      Sets DISTRO = ...

Enabled built-in fragments:
distro/poky     Sets DISTRO = "poky"
machine/qemux86-64      Sets MACHINE = "qemux86-64"

Available fragments in core layer located in .../layers/openembedded-core/meta:

Enabled fragments:
core/yocto/root-login-with-empty-password       Log in as root without password on serial console and over ssh
(use with caution).
core/yocto/sstate-mirror-cdn    Use prebuilt sstate artifacts for standard Yocto build configurations.

Unused fragments:
core/yocto/sbom-cve-check       This fragment enables sbom-cve-check with recommended default options
core/yocto-autobuilder/autobuilder-resource-constraints Resource constraint variables used on the Yocto Projects
autobuilder
core/yocto-autobuilder/autobuilder      Default variables used on the Yocto Projects autobuilder builds
core/yocto-autobuilder/multilib-mips64-n32      Enable a MIPS64 triarch multilib
core/yocto-autobuilder/multilib-x86-lib32       Enable an x86 lib32 multilib
core/yocto-autobuilder/multilib-x86-lib64       Enable an x86 lib64 multilib
```

Figure 14.2: Listing available BitBake configuration fragments

The output separates the fragments into four groups:

- **Available built-in fragments**: The generic `machine/…` and `distro/…` patterns, which the *Defining built-in fragments* section later in this chapter explains in depth
- **Enabled built-in fragments**: The specific values already active in this project, the distro/poky and machine/qemux86-64 choices you made when initializing it in *Chapter 2*
- **Enabled fragments**: The standard fragments already applied per layer, including `core/yocto/sstate-mirror-cdn` (behind the poky-with-sstate configuration you selected at initialization) and core/yocto/root-login-with-empty-password (enabled before your first build in *Chapter 2*)
- **Unused fragments**: The standard fragments available but not yet applied, such as `core/yocto/sbom-cve-check` and the `core/yocto-autobuilder` fragments

Disabling fragments

When you need to start over from a clean configuration, `bitbake-config-build disable-all-fragments` removes every fragment in a single step, built-in fragments included, which is faster than disabling each one individually:

```
$ bitbake-config-build disable-all-fragments
```

The command reports that every fragment was removed, as shown below.

```
$ bitbake-config-build disable-all-fragments
NOTE: Starting bitbake server...
All fragments removed from .../build/conf/toolcfg.conf.
```

Figure 14.3: Clearing every fragment selection

Running `list-fragments` right after confirms the clean slate:

```
$ bitbake-config-build list-fragments
```

The output confirms every fragment is gone, as shown below.

```
$ bitbake-config-build list-fragments
NOTE: Starting bitbake server...
Available built-in fragments:
machine/...     Sets MACHINE = ...
distro/...      Sets DISTRO = ...

Enabled built-in fragments:

Available fragments in core layer located in .../layers/openembedded-core/meta:

Unused fragments:
core/yocto/root-login-with-empty-password       Log in as root without password on serial console and over ssh
(use with caution).
core/yocto/sbom-cve-check       This fragment enables sbom-cve-check with recommended default options
core/yocto/sstate-mirror-cdn    Use prebuilt sstate artifacts for standard Yocto build configurations.
core/yocto-autobuilder/autobuilder-resource-constraints Resource constraint variables used on the Yocto Projects
autobuilder
core/yocto-autobuilder/autobuilder      Default variables used on the Yocto Projects autobuilder builds
core/yocto-autobuilder/multilib-mips64-n32      Enable a MIPS64 triarch multilib
core/yocto-autobuilder/multilib-x86-lib32       Enable an x86 lib32 multilib
core/yocto-autobuilder/multilib-x86-lib64       Enable an x86 lib64 multilib
```

Figure 14.4: Confirming a fully cleared configuration

`Enabled built-in fragments` is now empty, and every standard fragment, `core/yocto/sstate-mirror-cdn` included, moved back to `Unused fragments`. Because this also clears `MACHINE`, attempting to build now fails immediately:

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

BitBake stops with a fatal error, as shown below.

```
$ bitbake core-image-full-cmdline
FATAL: Directory name .../build/tmp/log/cooker/${MACHINE} contains unexpanded bitbake variable. This may cause
build failures and WORKDIR pollution.
HINT: It looks like you forgot to set MACHINE in local.conf.
```

Figure 14.5: BitBake refusing to build without a MACHINE

Clearing everything is the right move for a known-clean starting point, but it is rarely what you want mid-project. Most of the time you want to remove a single selection while leaving the rest of your configuration untouched, and the `disable-fragment` subcommand does exactly that. The following example removes the already-enabled `core/yocto/root-login-with-empty-password` fragment:

```
$ bitbake-config-build disable-fragment core/yocto/root-login-with-empty-password
```

The tool confirms the fragment was removed, as shown below.

```
$ bitbake-config-build disable-fragment core/yocto/root-login-with-empty-password
NOTE: Starting bitbake server...
Removing fragment core/yocto/root-login-with-empty-password from .../build/conf/toolcfg.conf
```

Figure 14.6: Disabling a BitBake configuration fragment

This removes the fragment's contribution from `build/conf/toolcfg.conf`.

Tip

You can also disable every fragment that shares a prefix by passing the prefix with a trailing slash, such as `bitbake-config-build disable-fragment machine/`. This shortcut is useful when you want to clear all `machine/…` or `distro/…` selections before enabling a different one, though disabling `machine/qemux86-64` itself only makes sense right before enabling a different machine, since a build with no `MACHINE` set at all cannot proceed.

Enabling fragments

With the configuration cleared and trimmed by the demonstrations above, get this project back to where it started. `enable-fragment` accepts more than one identifier per invocation, so a single command restores every fragment removed so far:

```
$ bitbake-config-build enable-fragment \
                           machine/qemux86-64 \
                           distro/poky \
                           core/yocto/root-login-with-empty-password \
                           core/yocto/sstate-mirror-cdn
```

The tool confirms each fragment as it is added back, as shown below.

```
$ bitbake-config-build enable-fragment \
                        machine/qemux86-64 \
                        distro/poky \
                        core/yocto/root-login-with-empty-password \
                        core/yocto/sstate-mirror-cdn
NOTE: Starting bitbake server...
Fragment machine/qemux86-64 added to (...)/build/conf/toolcfg.conf.
Fragment distro/poky added to (...)/build/conf/toolcfg.conf.
Fragment core/yocto/root-login-with-empty-password added to (...)/build/conf/toolcfg.conf.

Fragment summary: Log in as root without password on serial console and over ssh (use with caution).

Fragment description:
By default images are built such that root login is disabled (which is the preferred, secure default). However,
for testing and development purposes it can be beneficial to be able to log in as root, both on serial console
and over ssh connections, which is what this fragment enables. Use with great caution, and ideally only in
tightly controlled local builds and CI testing environments, and never in artefacts that are deployed into
products.

Fragment core/yocto/sstate-mirror-cdn added to (...)/build/conf/toolcfg.conf.

Fragment summary: Use prebuilt sstate artifacts for standard Yocto build configurations.

Fragment description:
The Yocto Project has prebuilt artefacts available for standard build configurations. This fragment enables
their use. This will mean the build will query the the network to check for artefacts at the start of builds,
which does slow it down initially but it will then speed up the builds by not having to build things if they are
present in the cache. It assumes you can download something faster than you can build it which will depend on
your network. Note: For this to work you also need hash-equivalence passthrough to the matching server
```

Figure 14.7: Restoring every fragment in one command

With `machine/qemux86-64`, `distro/poky`, `core/yocto/root-login-with-empty-password`, and `core/yocto/sstate-mirror-cdn` enabled again, `list-fragments` reports the same output shown at the start of this chapter, and BitBake can build again. The *bitbake-config-build disable-all-fragments* section at `https://docs.yoctoproject.org/wrynose/ref-manual/fragments.html#bitbake-config-build-disable-all-fragments` documents the subcommand alongside its companions.

Restoring several fragments in one call is convenient after a full reset, but day to day you will more often enable just one. Enable a fragment using the `enable-fragment` subcommand, the same mechanism `bitbake-setup init` used behind the scenes to set `MACHINE` and `DISTRO` for this project. The fragment listing earlier in this chapter already showed `core/yocto/root-login-with-empty-password` as enabled, because *Chapter 2* ran exactly this command before your first build. Here is that same command and its output, as a reference for the general shape every `enable-fragment` call takes:

```
$ bitbake-config-build enable-fragment core/yocto/root-login-with-empty-password
```

Running the command produces the following output:

```
$ bitbake-config-build enable-fragment core/yocto/root-login-with-empty-password
NOTE: Starting bitbake server...
Fragment core/yocto/root-login-with-empty-password added to .../build/conf/toolcfg.conf.

Fragment summary: Log in as root without password on serial console and over ssh (use with caution).

Fragment description:
By default images are built such that root login is disabled (which is the preferred, secure default). However,
for testing and development purposes it can be beneficial to be able to log in as root, both on serial console
and over ssh connections, which is what this fragment enables. Use with great caution, and ideally only in
tightly controlled local builds and CI testing environments, and never in artefacts that are deployed into
products.
```

Figure 14.8: Enabling BitBake configuration fragments

The command confirms the fragment was appended to `build/conf/toolcfg.conf` and echoes its summary and description, the same `BB_CONF_FRAGMENT_SUMMARY` and `BB_CONF_FRAGMENT_DESCRIPTION` values the *Defining a standard fragment* section later in this chapter explains how to write. The change takes effect immediately for subsequent BitBake invocations.

Fragments configure more than machine and distribution selection. The `core/yocto/root-login-with-empty-password` fragment is a good example: a freshly initialized configuration does not allow the `root` user to log in, and enabling it permits `root` to authenticate with an empty password on the serial console and over SSH.

> **Note**
>
> **Warning!**
>
> The `core/yocto/root-login-with-empty-password` fragment is a development convenience only. Disable it with `bitbake-config-build disable-fragment core/yocto/root-login-with-empty-password`, as shown earlier in this chapter, before producing an image you intend to ship. A shipping image is safer with `root` login disabled entirely, so each operator uses a normal account and elevates only when a task requires it. When a product still requires `root` access, assign a real password as described in *Chapter 12*.

Inspecting a fragment

When you need to confirm what a fragment will contribute before enabling it, the `bitbake-config-build show-fragment` subcommand prints the fragment's on-disk location and the variable assignments it defines. The *bitbake-config-build show-fragment* section at `https://docs.yoctoproject.org/wrynose/ref-manual/fragments.html#bitbake-config-build-show-fragment` describes the output format in full.

```
$ bitbake-config-build show-fragment core/yocto/root-login-with-empty-password
```

The command prints the fragment's location and contents, as shown below.

```
$ bitbake-config-build show-fragment core/yocto/root-login-with-empty-password
NOTE: Starting bitbake server...
.../layers/openembedded-core/meta/conf/fragments/yocto/root-login-with-empty-password.conf:

BB_CONF_FRAGMENT_SUMMARY = "Log in as root without password on serial console and over ssh (use with caution)."
BB_CONF_FRAGMENT_DESCRIPTION = "By default images are built such that root login is disabled \
(which is the preferred, secure default). However, for testing and development purposes it can \
be beneficial to be able to log in as root, both on serial console and over ssh connections, \
which is what this fragment enables. Use with great caution, and ideally only in tightly \
controlled local builds and CI testing environments, and never in artefacts that are deployed \
into products. \
"

EXTRA_IMAGE_FEATURES += "allow-empty-password empty-root-password allow-root-login"
```

Figure 14.9: Inspecting a BitBake configuration fragment

The output starts with the fragment's absolute path inside the layer, followed by its `.conf` contents verbatim. Here that means the `BB_CONF_FRAGMENT_SUMMARY` and `BB_CONF_FRAGMENT_DESCRIPTION` metadata that `list-fragments` already surfaces, plus the

`EXTRA_IMAGE_FEATURES` assignment this fragment actually contributes, letting you confirm exactly what enabling it would change before you run `enable-fragment`.

Configuration precedence and conflicts

Chapter 5 introduced this file-read order from the build-directory side. Here is the same ordering from the configuration-management side. BitBake includes `build/conf/toolcfg.conf` before `build/conf/local.conf`, so a hard (=) assignment in `build/conf/local.conf` takes precedence for ordinary variables. The built-in fragments for `MACHINE` and `DISTRO` are a special case. The `addfragments` directive that activates them runs after `build/conf/local.conf` is parsed, and it refuses to overwrite a non-weak assignment. If you set `MACHINE` or `DISTRO` in `build/conf/local.conf` while a matching fragment is enabled, the build fails outright rather than letting one source silently win.

> **Note**
>
> **Warning!**
>
> Setting `MACHINE` both in `build/conf/local.conf` and via a `machine/…` fragment causes a configuration error. Choose one approach: either manage `MACHINE` through fragments using `bitbake-config-build`, or set it directly in `build/conf/local.conf`. Mixing both approaches leads to conflicts.

BitBake reports the exact assignment it refuses to resolve, along with the command to disable the offending fragment:

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

BitBake stops the build and reports the conflict, as shown below.

```
$ bitbake core-image-full-cmdline
ERROR: A builtin fragment 'machine/qemux86-64' is used while MACHINE has already got an assignment.
Please either disable the fragment or remove the value assignment.
To disable the fragment, use 'bitbake-config-build disable-fragment machine/qemux86-64'.
ERROR: Unable to parse .../layers/bitbake/lib/bb/__init__.py
```

Figure 14.10: BitBake refusing to resolve a conflicting MACHINE assignment

When you need to set `MACHINE` directly in `build/conf/local.conf` instead of using fragments, use the weak assignment operator to avoid conflicts with fragment-based setups:

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

The `??=` operator assigns the value only if `MACHINE` is not already set elsewhere, allowing fragment selections to take precedence when present. For details on the weak assignment operator, see *Chapter 8*.

Creating custom BitBake configuration fragments

The fragments shown so far come from OpenEmbedded Core and `meta-poky`. You can also define your own fragments in any layer, giving your team reusable build options that `bitbake-config-build` manages alongside the standard ones.

Defining a standard fragment

A standard fragment is a `.conf` file placed under `conf/fragments/` inside your layer. The following diagram shows where a fragment fits in a typical layer directory tree.



Figure 14.11: Layer directory tree with a custom fragment

The fragment file itself contains ordinary BitBake variable assignments plus two metadata variables that `bitbake-config-build` reads when listing available fragments.

```
BB_CONF_FRAGMENT_SUMMARY = "This fragment sets a limit of 4 bitbake threads and 4
parsing threads"
BB_CONF_FRAGMENT_DESCRIPTION = "This fragment is useful to constrain resource
consumption when the Yocto \
default is causing an overload of host machine's memory and CPU resources."

BB_NUMBER_THREADS = "4"
BB_NUMBER_PARSE_THREADS = "4"
```

`BB_CONF_FRAGMENT_SUMMARY` provides the short description shown by `bitbake-config-build list-fragments`, and `BB_CONF_FRAGMENT_DESCRIPTION` holds a longer explanation for documentation or tooling that displays details. Both variables are required for the fragment to appear in listings.

The fragment identifier that you pass to `bitbake-config-build enable-fragment` is formed from two parts: the layer's `BBFILE_COLLECTIONS` name (defined in `conf/layer.conf`, see *Chapter 12*) and the file path relative to `conf/fragments/`, minus the `.conf` extension. If `conf/layer.conf` sets `BBFILE_COLLECTIONS += "meta-newlayer"`, the fragment above becomes `meta-newlayer/custom-fragment`. The *Creating New Configuration Fragments In Your Build* section at `https://docs.yoctoproject.org/wrynose/dev-manual/creating-fragments.html` of the *Yocto Project and OpenEmbedded Development Tasks Manual* shows the same construction against its own example layer.

After adding the fragment file, confirm that `bitbake-config-build` discovers it by running `list-fragments`:

```
$ bitbake-config-build list-fragments
```

The custom fragment now appears in its own `meta-newlayer` section, following the same `core` layer breakdown from the earlier listing, as shown below.

```
$ bitbake-config-build list-fragments
NOTE: Starting bitbake server...
Available built-in fragments:
machine/...     Sets MACHINE = ...
distro/...      Sets DISTRO = ...

Enabled built-in fragments:
distro/poky     Sets DISTRO = "poky"
machine/qemux86-64      Sets MACHINE = "qemux86-64"

Available fragments in core layer located in .../layers/openembedded-core/meta:

Enabled fragments:
core/yocto/root-login-with-empty-password       Log in as root without password on serial console and over ssh
(use with caution).
core/yocto/sstate-mirror-cdn    Use prebuilt sstate artifacts for standard Yocto build configurations.

Unused fragments:
core/yocto/sbom-cve-check       This fragment enables sbom-cve-check with recommended default options
core/yocto-autobuilder/autobuilder-resource-constraints Resource constraint variables used on the Yocto Projects
autobuilder
core/yocto-autobuilder/autobuilder      Default variables used on the Yocto Projects autobuilder builds
core/yocto-autobuilder/multilib-mips64-n32      Enable a MIPS64 triarch multilib
core/yocto-autobuilder/multilib-x86-lib32       Enable an x86 lib32 multilib
core/yocto-autobuilder/multilib-x86-lib64       Enable an x86 lib64 multilib

Available fragments in meta-newlayer layer located in .../layers/meta-newlayer:

Unused fragments:
meta-newlayer/custom-fragment   This fragment sets a limit of 4 bitbake threads and 4 parsing threads
```

Figure 14.12: Custom fragment appearing in the fragment list

The summary text comes directly from the `BB_CONF_FRAGMENT_SUMMARY` value in your fragment file. If the fragment does not appear, verify that the layer is listed in `build/conf/bblayers.conf` and that the file is placed under `conf/fragments/` within the layer.

Enabling your fragment

Enable the fragment with the same command used for any other fragment:

```
$ bitbake-config-build enable-fragment meta-newlayer/custom-fragment
NOTE: Starting bitbake server...
Fragment meta-newlayer/custom-fragment added to .../build/conf/toolcfg.conf.

Fragment summary: This fragment sets a limit of 4 bitbake threads and 4 parsing threads

Fragment description:
This fragment is useful to constrain resource consumption when the Yocto default is causing an overload of host
machine's memory and CPU resources.
```

Figure 14.13: Enabling a custom fragment

The tool confirms the fragment was appended to `build/conf/toolcfg.conf` and echoes its summary and description, the same confirmation shown for any built-in or standard fragment. The change takes effect on the next BitBake invocation.

Organizing fragments with subdirectories

As the number of fragments grows, subdirectories keep them manageable. The following diagram shows a layer with fragments grouped by category.

```
meta-newlayer/
├── conf/
│   ├── layer.conf
│   └── fragments/
│       ├── networking/
│       │   └── mirror-setup.conf
│       └── resources/
│           ├── constrained-host.conf
│           └── fast-build.conf
└── recipes-core/
    └── ...
```

Figure 14.14: Layer directory tree with fragment subdirectories

Subdirectory names become part of the fragment identifier. With `BBFILE_COLLECTIONS += "meta-newlayer"`, the fragments above resolve to `meta-newlayer/networking/mirror-setup`, `meta-newlayer/resources/constrained-host`, and `meta-newlayer/resources/fast-build`.

Running `list-fragments` again confirms the discovery and shows `meta-newlayer/custom-fragment` still enabled from the previous section, alongside the newly discovered subdirectory fragments:

```
$ bitbake-config-build list-fragments
```

The subdirectory fragments now appear in the listing, as shown below.

```
$ bitbake-config-build list-fragments
NOTE: Starting bitbake server...
Available built-in fragments:
machine/...     Sets MACHINE = ...
distro/...      Sets DISTRO = ...

Enabled built-in fragments:
distro/poky     Sets DISTRO = "poky"
machine/qemux86-64      Sets MACHINE = "qemux86-64"

Available fragments in core layer located in .../layers/openembedded-core/meta:

Enabled fragments:
core/yocto/root-login-with-empty-password       Log in as root without password on serial console and over ssh
(use with caution).
core/yocto/sstate-mirror-cdn    Use prebuilt sstate artifacts for standard Yocto build configurations.

Unused fragments:
core/yocto/sbom-cve-check       This fragment enables sbom-cve-check with recommended default options
core/yocto-autobuilder/autobuilder-resource-constraints Resource constraint variables used on the Yocto Projects
autobuilder
core/yocto-autobuilder/autobuilder      Default variables used on the Yocto Projects autobuilder builds
core/yocto-autobuilder/multilib-mips64-n32      Enable a MIPS64 triarch multilib
core/yocto-autobuilder/multilib-x86-lib32       Enable an x86 lib32 multilib
core/yocto-autobuilder/multilib-x86-lib64       Enable an x86 lib64 multilib

Available fragments in meta-newlayer layer located in .../layers/meta-newlayer:

Enabled fragments:
meta-newlayer/custom-fragment   This fragment sets a limit of 4 bitbake threads and 4 parsing threads

Unused fragments:
meta-newlayer/networking/mirror-setup   This fragment configures local source and package mirrors
meta-newlayer/resources/constrained-host        This fragment limits parallelism for low-resource build hosts
meta-newlayer/resources/fast-build      This fragment maximizes parallelism on powerful build hosts
```

Figure 14.15: Subdirectory fragments appearing in the fragment list

OpenEmbedded Core follows the same pattern. Its `core` collection provides fragments under subdirectories such as `yocto/` and `yocto-autobuilder/`, producing identifiers like `core/yocto/sstate-mirror-cdn` and `core/yocto-autobuilder/autobuilder-resource-constraints`.

Defining built-in fragments

The `machine/` and `distro/` fragments you have been using throughout this chapter work differently from standard fragments. A standard fragment always points to a `.conf` file under

`conf/fragments/`, but a built-in fragment skips the file entirely: it maps a prefix directly to a single BitBake variable. When you enable `machine/qemux86-64`, the tool sets `MACHINE = "qemux86-64"` without reading any fragment file. The value after the slash becomes the variable's value, and that direct mapping is what makes built-in fragments special.

OpenEmbedded Core ships exactly two built-in fragments: `machine/`, which sets `MACHINE`, and `distro/`, which sets `DISTRO`. You have already relied on both. `bitbake-setup init` used them to set `MACHINE = "qemux86-64"` and `DISTRO = "poky"` when you created this project in *Chapter 2*, and the `disable-fragment`/`enable-fragment` examples earlier in this chapter re-enabled `machine/qemux86-64` and `distro/poky` the same way you would enable any other fragment identifier.

It is possible to define additional built-in fragments in your own layer, but doing so is a narrow, advanced use case you are unlikely to need this early in your Yocto Project work. The *Creating A Built-in Fragment* section at `https://docs.yoctoproject.org/wrynose/dev-manual/creating-fragments.html#creating-a-built-in-fragment` of the *Yocto Project and OpenEmbedded Development Tasks Manual* covers the `OE_FRAGMENTS_BUILTIN` variable and the operator it requires, should you ever need to map a prefix of your own to a variable. For everything else, from constraining build resources to selecting mirrors, the standard fragments covered throughout this chapter remain the right tool.

Managing layers with bitbake-setup

Beyond manual layer cloning (covered in *Chapter 11*), you can define layers declaratively in your BitBake Configuration Template. This approach keeps layer specifications version-controlled and reproducible across your team.

To add a layer such as `meta-oe` (a component of the `meta-openembedded` repository), extend your BitBake Configuration Template's `sources` section. For example, duplicate `bitbake-builds/poky-wrynose/layers/bitbake/default-registry/configurations/poky-wrynose.conf.json`, the template file the `git clone https://git.openembedded.org/bitbake` command from *Chapter 2* placed on disk, into a new file such as `poky-wrynose-with-oe.conf.json`, and merge in the following:

```
{
    "sources": {
        "meta-openembedded": {
            "git-remote": {
                "remotes": {
                    "origin": {
```

```
                        "uri": "https://git.openembedded.org/meta-openembedded"
                    }
                },
                "branch": "wrynose",
                "rev": "wrynose"
            },
            "path": "meta-openembedded"
        }
    },
    "bitbake-setup": {
        "configurations": [
            {
                "bb-layers": [
                    "openembedded-core/meta",
                    "meta-yocto/meta-yocto-bsp",
                    "meta-yocto/meta-poky",
                    "meta-openembedded/meta-oe"
                ]
            }
        ]
    }
}
```

The `sources` section defines where to fetch the repository, while the `bb-layers` array specifies which sub-layers to include. When you run `bitbake-setup init` with this configuration, the tool clones `meta-openembedded` and configures `build/conf/bblayers.conf` automatically.

After initializing from `poky-wrynose-with-oe.conf.json`, verify that the layer appears in your configuration. The output below assumes you followed this `meta-oe` example:

```
$ bitbake-layers show-layers
```

The new layer now appears in the list, as shown below.

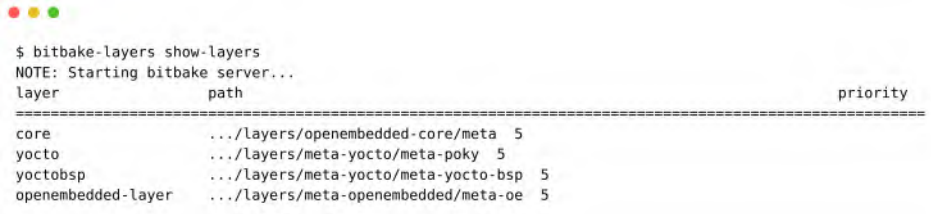


Figure 14.16: Verifying layers after bitbake-setup init

This declarative approach offers advantages over manual layer management: layer versions are pinned explicitly, team members reproduce identical setups, and layer updates flow through your configuration management process rather than requiring each developer to run separate `git pull` commands.

Keeping your setup current

After initial setup, you may need to sync your local environment with updates to the BitBake Configuration Template. The `bitbake-setup status` command shows whether your setup differs from the upstream configuration:

```
$ bitbake-setup status
```

Running the command produces the following output:

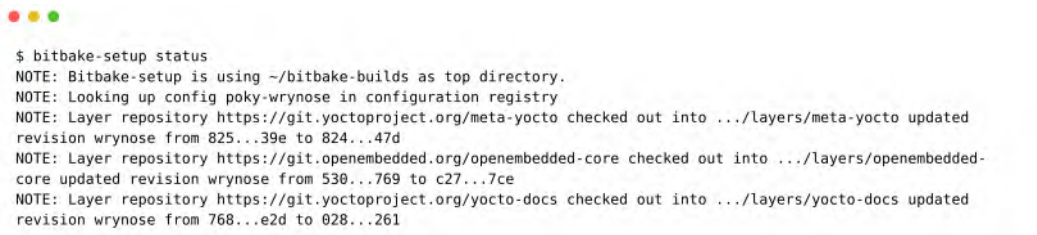


Figure 14.17: Checking BitBake configuration template status

The output reports, layer by layer, the revision each repository would move to if you synchronized now, without touching your working tree. Here, `meta-yocto`, `openembedded-core`, and `yocto-docs` each have a newer commit available on the `wrynose` branch than the one currently checked out. When differences exist, use `bitbake-setup update` to bring your setup in sync:

```
$ bitbake-setup update
```

This command fetches the latest changes from the upstream configuration and updates your local layers accordingly. Unlike the other captured examples in this book, the exact output you see will differ from what is shown here: it depends on the revision your project started from at `bitbake-setup init` and on whatever changes exist upstream on the day you run `bitbake-setup update`, both of which vary from one reader, and one run, to the next. `bitbake-setup update` stops in two cases rather than discarding your work. It stops when a layer has uncommitted changes to tracked files, and it stops when local commits fail to rebase onto the new revision. Pass `--rebase-conflicts-strategy=backup` to let the update proceed in either case. The tool then renames the layer directory aside with a timestamp and re-clones it from upstream. Run `bitbake-setup status` after updating to confirm the sync completed successfully.

Every setup you initialize lives beneath a **Top Directory**, the working directory in which `bitbake-setup` operates and the root that settings such as `common-sstate` and `dl-dir` apply across. Yocto Project 6.0 (Wrynose) sharpens the update workflow in three ways:

- `bitbake-setup update` refreshes each layer in place instead of re-cloning it. When a layer holds local commits, the tool fetches the new revision and rebases those commits onto it. Your work therefore moves forward together with the metadata. The update stops before it changes anything if the layer has uncommitted changes to tracked files, or if the rebase hits a conflict.
- `bitbake-setup init` writes Shared State Cache and hash-equivalence paths into a `site.conf` at this setup's Top Directory, which every setup beneath that directory inherits. This behavior is governed by the `common-sstate` setting (enabled by default). Teams no longer need a bespoke `site.conf` template just to share an `SSTATE_DIR` across workstations.
- `bitbake-setup init` also writes a VS Code workspace by default when the `code` binary is on `PATH`. Pass `--no-init-vscode` when a team uses a different editor or a stricter workstation profile that disallows automated IDE configuration. *Chapter 3* covers how to open and work in that generated workspace.

These additions make first-time onboarding closer to a single command and keep day-to-day updates safer on shared developer machines.

Reproducible team workflows

Building upon the `bitbake-setup` introduction in *Chapter 2*, this section explores advanced configuration patterns suited for enterprise development teams and automated build systems. The tool's declarative approach to environment setup addresses common pain points in team

environments: inconsistent developer workstations, difficulty reproducing builds, and manual layer management that drifts over time.

Tip

While `bitbake-setup` is the official first-party solution, the community maintains alternative project initialization tools. `kas` reads a YAML file that pins each layer repository to a commit, branch, or tag, and writes `MACHINE` and `DISTRO` into `build/conf/local.conf`. It also merges several configuration files through an `includes` list, so a team can keep a common base separate from each product configuration. `repo` manifests pin a revision per repository, but they manage Git checkouts only. A team that uses `repo` still needs a separate mechanism for the build configuration.

`bitbake-setup` differs from both in three ways. It selects build settings through BitBake Configuration Fragments. It pins BitBake itself alongside the layers. It shares the download cache and Shared State Cache through its own settings. The concepts covered here (declarative configuration, layer management, and reproducible builds) apply regardless of which tool you adopt.

The following sections walk through advanced configuration patterns starting with custom templates.

Creating custom BitBake configuration templates

BitBake Configuration Template files (`.conf.json`) define complete build environments declaratively. A well-designed configuration captures source repositories with specific revisions, layer inclusion and dependencies, build-time choices such as `MACHINE` and `DISTRO`, and fragment selections for modular configuration.

The following shows an example configuration for a hypothetical product baseline.

```
{
    "version": "1.0",
    "description": "Acme Corp product baseline for Yocto Project 6.0",
    "expires": "2028-05-31",
    "sources": {
        "openembedded-core": {
            "git-remote": {
                "remotes": {
                    "origin": {
                        "uri": "https://git.openembedded.org/openembedded-core"
```

```
                    }
                },
                "branch": "wrynose",
                "rev": "wrynose"
            },
            "path": "openembedded-core"
        },
        "bitbake": {
            "git-remote": {
                "remotes": {
                    "origin": {
                        "uri": "https://git.openembedded.org/bitbake"
                    }
                },
                "branch": "2.18",
                "rev": "2.18"
            },
            "path": "bitbake"
        },
        "meta-yocto": {
            "git-remote": {
                "remotes": {
                    "origin": {
                        "uri": "https://git.yoctoproject.org/meta-yocto"
                    }
                },
                "branch": "wrynose",
                "rev": "wrynose"
            },
            "path": "meta-yocto"
        },
        "meta-acme": {
            "git-remote": {
                "remotes": {
                    "origin": {
                        "uri": "git@github.com:acme/meta-acme.git"
                    }
                },
                "branch": "main",
                "rev": "v2.1.0"
            },
```

```
            "path": "meta-acme"
        }
    },
    "bitbake-setup": {
        "configurations": [
            {
                "bb-layers": [
                    "openembedded-core/meta",
                    "meta-yocto/meta-poky",
                    "meta-acme/meta-acme-bsp",
                    "meta-acme/meta-acme-distro"
                ],
                "oe-fragments-one-of": {
                    "machine": {
                        "description": "Target machines",
                        "options": [
                            {
                                "name": "machine/qemux86-64",
                                "description": "x86-64 development build on QEMU"
                            },
                            {
                                "name": "machine/raspberrypi5",
                                "description": "Raspberry Pi 5 board bring-up"
                            }
                        ]
                    }
                },
                "setup-dir-name": "acme-${machine}"
            }
        ]
    }
}
```

The `sources` section defines each repository with its remote URI, branch, and revision. Using explicit revision tags (like `v2.1.0`) or commit hashes rather than branch names pins the configuration to a known state. The `bb-layers` array lists the layers to include in `build/conf/bblayers.conf`, maintaining the order specified.

The `oe-fragments-one-of` section lets a BitBake Configuration Template offer the user a choice between several mutually exclusive fragments at initialization time. The template lists the

candidates with short descriptions, and `bitbake-setup init` prompts the user to select one. To skip the prompt, name the choice on the command line after the configuration, as in `bitbake-setup init poky-wrynose poky machine/qemux86-64`. Add `--non-interactive` to make the tool stop with an error instead of prompting when the command line leaves a choice unspecified. This is how a single template can serve, say, a `qemux86-64` development build and a `raspberrypi5` board bring-up build without splitting into two templates.

Store your BitBake Configuration Templates in a dedicated repository that your team can reference as a BitBake Registry. This approach separates configuration management from the actual layer content and provides a clear history of environment changes.

Sharing the download cache

Chapter 5 covered the `DL_DIR` variable itself. This section covers how `bitbake-setup` manages it across a team. Downloads are shared from the outset. The first `bitbake-setup init` in a Top Directory writes a weak `DL_DIR` assignment into the `site.conf` it generates at that directory, pointing at a `.bitbake-setup-downloads` directory beneath it, and every BitBake Setup created underneath receives a `build/conf/site.conf` symlink back to that one file. The `dl-dir` setting decides which path that assignment names, which is how you move the cache onto a faster filesystem, onto shared storage, or outside the Top Directory so several Top Directories can draw on it.

`bitbake-setup` keeps its own operational parameters in a `settings.conf` file, separate from any BitBake metadata. The `dl-dir` setting belongs to that file's `[default]` section. Set it with `bitbake-setup settings set default dl-dir <path>`. That command writes to the `settings.conf` at the root of the current Top Directory, where `bitbake-setup` reads it on the next `init`. Add `--global` to write to `~/.config/bitbake-setup/settings.conf` instead, which applies to every Top Directory on the machine. *Chapter 18* compares the two files and explains which one to prefer.

A common pattern is to place the shared cache on a fast local filesystem and point every build under the same Top Directory to it by running `bitbake-setup settings set default dl-dir /opt/yocto-downloads`. The first build fetches sources and populates that path. Subsequent builds (for different machines, different products, or other developers sharing the same workstation) reuse those downloads and skip the network fetch. The cache is also portable: a tarball of its contents can bring a completely offline build host to a known state.

Both `dl-dir` and `common-sstate` are `bitbake-setup` settings that apply to every BitBake Setup beneath a Top Directory, and both share their cache by default, though through different mechanisms. `common-sstate` is a toggle: it defaults to `yes`, which is what makes

`bitbake-setup init` add `SSTATE_DIR` and `BB_HASHSERVE_DB_DIR` to the generated `site.conf`, both beneath the Top Directory. Set it to `no` and those lines are omitted. `dl-dir` is a path rather than a toggle, so sharing is not something it turns on or off; it only names the directory the `DL_DIR` assignment points at. Neither setting requires any change to `build/conf/local.conf`.

One consequence is worth remembering when a cache does not end up where you expected. `site.conf` is generated once, when the Top Directory is created, and later `bitbake-setup init` runs report that the file already exists and leave it alone. Change `dl-dir` or `common-sstate` after that first initialization and builds keep using the paths already written; remove `site.conf` and let the next `init` regenerate it for the new values to take effect.

Keeping several BitBake setups under one top directory

Sharing the download cache and the Shared State Cache only pays off when more than one BitBake Setup lives beneath the same Top Directory, which is common during development: several boards, or a development build and a production build of the same board, each initialized as its own BitBake Setup. `bitbake-setup init` derives the BitBake Setup's directory name from the BitBake Configuration Template and configuration you pass it, so two BitBake Setups initialized from the same template and configuration resolve to the same directory name, and the second `init` reuses the first BitBake Setup's directory instead of creating a new one. Pass `--setup-dir-name <name>` to give each BitBake Setup its own directory under the Top Directory, so builds that share the same template can sit side by side without colliding.

With distinct BitBake Setup directories in place, the shared `dl-dir` and `common-sstate` settings do their work: each BitBake Setup keeps its own pinned layers, while the download cache and Shared State Cache are reused across all of them. Builds that differ only in machine or a handful of fragments then reuse most fetched sources and cached task output instead of refetching and rebuilding from scratch. *Chapter 16* applies exactly this pattern when building several boards from one workspace.

Managing private BitBake registries

Teams can host internal BitBake Registries containing proprietary configurations. A BitBake Registry is simply a Git repository, local directory, or remote URI containing BitBake Configuration Template files. The `bitbake-setup settings set` subcommand records this location. Configure the BitBake Registry with the command:

```
$ bitbake-setup settings set default registry \
    git@gitlab.acme.com:yocto/configs.git --global
```

The `--global` flag saves this setting to your user configuration, making it available across all workspaces. For project-specific BitBake Registries, omit the flag to save to the local `settings.conf` file.

BitBake Registry organization patterns vary by team needs:

- Version-tagged configurations: Align configuration tags with product releases, enabling teams to rebuild any historical version
- Hierarchical inheritance: Base configurations define common settings, while product-specific configurations extend them with additional specifications

When listing available configurations, `bitbake-setup list` queries your configured BitBake Registry and displays all valid entries. Team members can initialize any configuration by name without needing to know the underlying repository structure.

Source overrides for development branches

When developers need to test against unreleased branches or forks, **Source Override** files modify layer sources without editing the primary BitBake Configuration Template. This separation keeps the baseline configuration stable while enabling experimentation.

The following shows a Source Override that switches a layer to a feature branch.

```
{
    "version": "1.0",
    "description": "Test against new driver feature branch",
    "sources": {
        "meta-acme": {
            "git-remote": {
                "branch": "feature/new-driver",
                "rev": "feature/new-driver"
            }
        }
    }
}
```

Apply Source Overrides during initialization:

```
$ bitbake-setup init --source-overrides driver-test-override.json
```

Multiple Source Override files can be stacked. The tool merges them in command-line order, with later files taking precedence for conflicting settings. This pattern supports scenarios like

testing a feature branch from one layer while pinning another layer to a specific bug-fix commit.

Developers can commit Source Override files alongside their feature branches, documenting exactly what configuration they used during development. Code reviewers can then reproduce the same environment to validate changes.

Source Overrides still fetch each layer from a remote. When the branch you want to test already exists as a checkout on your own disk, pass the layer name and its path to `bitbake-setup init -L <source> <path>` (the `--use-local-source` form) and the tool links that directory into the new setup instead of cloning it. BitBake then builds against your working tree in place, so edits feed straight into the build without a commit-and-push round trip, and `bitbake-setup update` leaves a linked source alone rather than refreshing it from upstream.

Synchronizing configurations across teams

The `bitbake-setup status` command reveals drift between local setups and upstream configurations. This helps teams detect when their environment diverges from the agreed baseline.

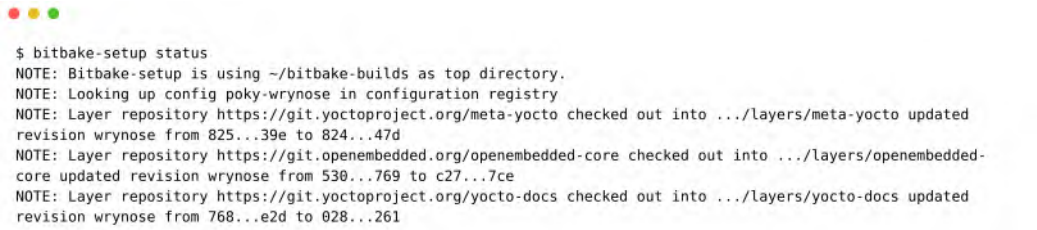


Figure 14.18: Inspecting drift with bitbake-setup status

The output shows whether local layers are ahead of, behind, or different from the configuration specification. When upstream updates are available, synchronize with:

```
$ bitbake-setup update
```

This workflow supports controlled rollouts:

1. Configuration maintainers update the BitBake Registry with tested changes
2. Teams pull updates at planned integration points, such as sprint boundaries or release milestones

The `update` command respects local modifications where possible, merging upstream changes rather than overwriting them. When conflicts occur, the tool reports them for manual resolution.

When a controlled rollout has to proceed even though a layer carries local commits that fail to rebase cleanly onto the new upstream revision, pass `--rebase-conflicts-strategy=backup` to `bitbake-setup update`. The tool renames the conflicting directory with a timestamp suffix and re-clones the layer fresh from upstream, so the update completes cleanly while the original work-in-progress is preserved for review or recovery.

Configuration, sources, and reproducibility

`bitbake-setup` manages build configuration, not metadata. A BitBake Configuration Template records which layers to clone, which revisions to pin, and which BitBake Configuration Fragments to enable, but the recipes and classes inside those layers keep evolving on their own schedule. The resultant artifact a build produces depends on both halves together: the configuration in effect and the exact set of sources checked out when BitBake runs against it. Change either one and the artifact can change too, which is why comparing two builds means comparing their BitBake Configuration Template and their layer revisions, not just one or the other.

This dependency is also what reproducibility is built on. *Chapter 2* showed `bitbake-setup init` cloning BitBake itself into `layers/bitbake/` alongside the other layers, so a BitBake Setup pins the tool that reads the configuration, not only the metadata the configuration points to. Every setup created from the same BitBake Configuration Template therefore initializes the same way: the same tool, the same layers, the same revisions, every time. That determinism is precisely why layers get cloned separately for each setup rather than shared. Two setups that pin `meta-openembedded` to different revisions cannot reuse one checkout without one of them silently drifting from its own configuration.

The download cache and the Shared State Cache behave differently because their contents are keyed by what went into them (a source checksum, a task's input hash) rather than by which BitBake Setup asked for them. That is exactly why the `dl-dir` and `common-sstate` settings covered earlier in this chapter let every setup under one Top Directory share them safely, and why keeping several setups initialized for the same project during development costs little. The layers are the only expense unique to each setup. The shared caches absorb the rest, reused on demand as each setup needs them.

Summary

This chapter covered advanced configuration management with `bitbake-setup` and `bitbake-config-build`. You learned how to discover available configurations, manage build fragments without editing files directly, create custom BitBake Configuration Fragments in your own layers, keep your environment synchronized with upstream changes, and implement

team workflows that support reproducible builds. These patterns become increasingly valuable as your projects grow in complexity and team size.

The next chapter shifts focus to license compliance. It shows how the Yocto Project tools help you meet open-source license obligations for your deployed images. It also covers the Software Bill of Materials that records every component you ship.

15

License Compliance and Software Bill of Materials

This chapter explains how to use the Yocto Project's built-in tooling to satisfy copyleft obligations and to document what a product ships: generating source archives, producing license manifests, declaring the terms of your own proprietary components, and emitting a Software Bill of Materials that inventories every piece of the image.

By the end, you will be able to assemble an auditable compliance bundle (complete source tarballs, per-package license texts, and a machine-readable record of every component in the image, proprietary ones included) and walk through that bundle with confidence during an external license audit or a regulatory filing.

Compliance cannot be treated as an afterthought in embedded products. Most Linux-based devices combine copyleft-licensed components such as the Linux kernel and GNU Compiler Collection (GCC) with proprietary application code, and both halves have to be accounted for. Getting the obligations wrong exposes a company to legal action and jeopardizes relationships with the open-source community, so building compliance into the release process from the start is the only practical approach.

We will cover the following main topics:

- Understanding copyleft
- The role of the Yocto Project
- Achieving copyleft compliance with the Yocto Project
- Managing software licensing with the Yocto Project
- Understanding commercial licenses
- Configuring your project for compliance

Technical requirements

This chapter requires a build host configured and verified per *Chapter 2*. No additional host packages are needed beyond those the first-build chapter installs.

Plan for extra disk space. The archiver class produces complete source tarballs for every copyleft-covered package in the image, and the SBOM generator writes per-recipe JSON documents alongside the image artifacts. Together, those outputs can roughly double the size of the `build/tmp` working tree compared to a standard image build. A working tree that normally occupies around 50 GB may grow to 90–100 GB when both features are active, so verify available space before enabling them on a workstation with a constrained storage layout.

Understanding copyleft

Copyleft is a legal mechanism that uses copyright law to maximize user rights and promote freedom in software use and distribution. However, it places specific obligations on product development, particularly in adhering to open-source and free software license requirements.

When building a Linux distribution, at least two major projects are typically involved: the Linux kernel and a compiler. The Linux kernel is licensed under the General Public License (GPL) version 2, with a syscall note exception that permits non-GPL user-space programs to use the Linux kernel's system-call interface. The GNU Compiler Collection (GCC) is distributed under GPLv3 with the GCC Runtime Library Exception applied to its runtime libraries such as `libgcc` and `libstdc++`.

Copyleft comes in two strengths, and the difference decides what you may link against. The GPL is strong copyleft: a program that incorporates GPL-covered code has to be distributed under the GPL as a whole. The Lesser GPL (LGPL) relaxes that for libraries. A proprietary application may link against an LGPL library, provided the user can replace that library with a modified version and still run the result. In practice that means linking dynamically against a shared library, which is the simpler of the two routes LGPLv3 offers. Static linking remains possible, but then you owe the recipient enough of your own application, in object form, for them to relink it.

The version number carries a second distinction that matters more in embedded work than anywhere else. GPLv3 adds a requirement that GPLv2 has no equivalent of. When you convey object code inside a User Product, which the license defines as a consumer product or anything designed for incorporation into a dwelling, you must also supply the Installation Information: the methods, procedures, authorization keys, or other information required to install and run a modified build on that device. A locked bootloader or a secure boot chain that refuses unsigned

images conflicts directly with that duty. The license does exempt the case where nobody retains the ability to install modified code at all, giving firmware in ROM as its example, but that exemption rarely survives a product that ships field updates.

This is why many embedded products exclude the version 3 family outright rather than reason about it device by device, and why the `INCOMPATIBLE_LICENSE` setting shown later in this chapter names both `GPL-3.0*`, `LGPL-3.0*`, and `AGPL-3.0*`. LGPLv3 inherits the requirement because it builds on GPLv3 rather than standing alone, and AGPLv3 adds a further duty to offer source to users who interact with the software across a network. Those three wildcards are also the only combination the Yocto Project exercises regularly in its own testing, which makes them the least likely to surprise you. The trade is not one-directional: GPLv3 also adds an explicit patent grant, a cure period that lets a first-time violator restore terminated rights within 30 days, and compatibility with the Apache License 2.0 that GPLv2 lacks. Whether the anti-lock-down duty actually binds your product is a legal question about your hardware and your distribution model, not one this chapter can settle.

A Linux-based system may also incorporate a wide variety of open-source projects and applications developed by the company for its products. This complexity raises critical questions: How can we determine the number and type of licenses in use? How can we achieve compliance with copyleft requirements?

The role of the Yocto Project

The Yocto Project provides tools and methodologies to streamline copyleft compliance. However, it is essential to have a clear understanding of your legal obligations and potential license incompatibilities. Consulting with your legal team or a copyright lawyer is strongly recommended if you have any doubts.

The following sections explore how the Yocto Project assists with common tasks required to achieve copyleft compliance, helping you navigate these challenges effectively. License metadata declared in each recipe travels with the binaries the build system produces. The packaging mechanisms covered in *Chapter 7* produce the per-package metadata that the build system later aggregates into the license manifest under `build/tmp/deploy/licenses/<machine>/`.

Achieving copyleft compliance with the Yocto Project

Understanding the coexistence of proprietary and copyleft-covered code within the same product is essential in modern software development. This common scenario requires careful consideration of license compatibility, especially when linking copyleft libraries to proprietary code or distributing applications under copyleft terms.

A typical Linux-based system comprises numerous projects, each governed by its unique license. The Yocto Project aids developers in managing this complexity by offering tools and methodologies that support adherence to copyleft license requirements. The primary obligations include:

- **Providing source code**: The source code of the copyleft-licensed project must be made available
- **Including the license text**: The complete license text for the project must accompany the distribution
- **Documenting modifications**: Any changes made to the original source must be documented and shared
- **Sharing build scripts**: Scripts required to configure and build the software must also be included

The Yocto Project simplifies these tasks through its tools and metadata-driven framework. By integrating compliance mechanisms into the build process, developers can systematically address license obligations. Before any software release, perform a thorough audit of all materials to confirm completeness and compliance.

Managing software licensing with the Yocto Project

One critical feature of the Yocto Project is its ability to manage software licenses effectively. While most development efforts focus on debugging and feature enhancements, managing licenses is essential when preparing a product for release. Proper license management supports compliance with legal obligations and avoids potential risks.

The build system tracks licenses in every recipe, maintaining transparency, and traceability throughout the development cycle. It also provides strategies for integrating proprietary applications without violating license terms. A fundamental concept is understanding the distinction between the license of a recipe and the license of the associated project. These are two separate entities, and both licenses contribute to the overall compliance of the product.

In most recipes, comments specify the copyright, license, and author details of the recipe itself. Beyond this, the build system uses specific variables to define the license information for the associated package:

- `LICENSE`: Specifies the license under which the package is released
- `LIC_FILES_CHKSUM`: Tracks the checksum of the license file for the package, so that any changes to the license text are noted and reviewed

The `LIC_FILES_CHKSUM` variable serves as a safeguard, flagging even minor changes to the license file (whether a typo fix or a significant legal modification) for review. Common license files are stored in `meta/files/common-licenses/`.

Many projects include a dedicated file, such as `COPYING` or `LICENSE`, to describe the license terms. Others use header notes in source files or a central document. The `LIC_FILES_CHKSUM` variable detects any modifications to these files. When a change is identified, BitBake triggers a build error, specifying the project with the altered license. To resolve this, follow these steps:

1. **Investigate the change**: Understand the nature of the modification. Consult your legal department if the license terms have significantly changed.
2. **Update variables**: Modify the `LIC_FILES_CHKSUM` value to reflect the new checksum and update the LICENSE field if necessary
3. **Document the change**: Record the reason for the update in a commit message for future reference. Add a `License-Update:` tag with a short reason, as in `License-Update: copyright years refreshed`. The tag applies when upstream changes the license file, not when you correct wrong license data in the recipe.

Distribution-wide licensing policy lives alongside the rest of the distribution metadata. Setting `INCOMPATIBLE_LICENSE` and `LICENSE_FLAGS_ACCEPTED` in a custom distribution configuration centralizes the decision in one place so individual recipe edits never have to relax it. `INCOMPATIBLE_LICENSE` is the primary mechanism for embedded products that must avoid an entire license family. Setting `INCOMPATIBLE_LICENSE = "GPL-3.0* LGPL-3.0* AGPL-3.0*"` in a custom distribution excludes every recipe whose `LICENSE` field matches one of those patterns, which is the canonical pattern for shipping products that cannot include GPLv3-licensed code. The Yocto Project documentation cautions that a global exclusion usually does not produce a feasible build, because a recipe with no alternative to an excluded license drops out and takes every build-time dependency on it down as well. Setting `INCOMPATIBLE_LICENSE` in the image recipe instead leaves the build unrestricted and still fails the image when a package carrying an excluded license reaches the root filesystem, which is what upstream recommends. The distribution-wide form is the stricter of the two and suits products that must never build the excluded code at all, as long as you are prepared to provide replacements for what it removes.

The following listing extends the `my-distro.conf` example from *Chapter 12* with both variables, folding the licensing policy into the same custom distribution file that already sets `DISTRO_FEATURES_OPTED_OUT`:

```
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_OPTED_OUT = "wayland vulkan opengl"

INCOMPATIBLE_LICENSE = "GPL-3.0* LGPL-3.0*"
LICENSE_FLAGS_ACCEPTED = "commercial"
```

Firmware licensing is a related distribution-wide policy question worth flagging alongside these two variables.

Tip

The `linux-firmware` recipe excludes firmware without clear licensing by default. If your hardware requires firmware that the recipe filters out, adjust `REMOVE_UNLICENSED` in a `linux-firmware.bbappend` to drop the affected entries from the exclusion list (typically with `REMOVE_UNLICENSED:remove`).

To streamline the process and avoid complications, weave these habits into your workflow:

- **Conduct regular audits**: Periodically review licenses to maintain continued compliance
- **Consult legal experts**: Engage your legal team when encountering unfamiliar or modified licenses
- **Maintain transparency**: Document all changes and the rationale behind them to support future audits and reviews

The Yocto Project's license management features let you keep your product within its legal obligations and maintain trust within the open-source community.

Understanding commercial licenses

By default, the build system excludes recipes that are restricted by commercial licenses so the build satisfies license obligations and developers must make intentional decisions when they include such recipes. The `LICENSE_FLAGS` variable in a recipe file identifies specific license restrictions. For instance, the `gstreamer1.0-plugins-ugly` recipe uses the following license-related variables:

```
require gstreamer1.0-plugins-common.inc
require gstreamer1.0-plugins-license.inc

SUMMARY = "'Ugly GStreamer plugins"
HOMEPAGE = "https://gstreamer.freedesktop.org/"
BUGTRACKER = "https://gitlab.freedesktop.org/gstreamer/gst-plugins-ugly/-/issues"

LIC_FILES_CHKSUM = "file://COPYING;md5=a6f89e2100d9b6cdffcea4f398e37343 \
                    "

LICENSE = "LGPL-2.1-or-later & GPL-2.0-or-later"
LICENSE_FLAGS = "commercial"

(...)
```

Figure 15.1: An example of license-related variables used in the gstreamer1.0-plugins-ugly recipe

The `LICENSE_FLAGS` variable in the recipe file specifies that the build system requires explicit acceptance of the commercial license flag for this recipe to be used. To enable the `gstreamer1.0-plugins-ugly` recipe, add the following to your configuration:

```
LICENSE_FLAGS_ACCEPTED += "commercial"
```

You can include `LICENSE_FLAGS_ACCEPTED` in your custom distribution configuration (for example, `<my-layer>/conf/distro/my-distro.conf`) or within `build/conf/local.conf` during initial development. Although enabling the commercial flag globally simplifies the installation of all recipes requiring this license flag, enabling specific recipes selectively is recommended. This provides tighter control over license compliance and prevents the inadvertent inclusion of undesired components. For example, to accept only the `gstreamer1.0-plugins-ugly` recipe, use the following:

```
LICENSE_FLAGS_ACCEPTED:pn-gstreamer1.0-plugins-ugly = "commercial"
```

This configuration enables the commercial license flag exclusively for the `gstreamer1.0-plugins-ugly` recipe. Restrict the use of such flags to a well-defined and verified set of recipes, so permissions and compliance stay tractable in a commercial context.

By carefully managing license flags and understanding their implications, you can maintain compliance while using recipes with commercial restrictions effectively.

Keep in mind that `LICENSE_FLAGS` describes a restriction on using a recipe, not the terms the recipe ships under. Declaring the license of a component your own team owns is a separate task, covered later in this chapter.

Configuring your project for compliance

Understanding the Yocto Project's role in managing legal aspects associated with producing Linux-based systems that incorporate packages under various licenses is essential. The Yocto Project provides tools to help generate artifacts required for compliance with copyleft licenses, supporting a smoother legal process during distribution.

Understanding license auditing

To facilitate copyleft compliance, the build system automatically generates a license manifest during the image build process. These manifests are in the directory `build/tmp/deploy/licenses/<machine>/<image_name-machine_name>.rootfs-<datastamp>/`, alongside the other deploy artifacts mapped out in *Chapter 6*.

Let's use the `core-image-full-cmdline` image for the `qemux86-64` machine as an example. After building this image, you will find a set of files under the directory, including:

- `image_license.manifest`: Lists recipe names, versions, licenses, and package files that are part of the build but not included in the root filesystem. Examples include the bootloader, Linux kernel image, and DTB files.
- `package.manifest`: Provides a comprehensive list of all packages included in the image.
- `license.manifest`: Details the names, versions, recipe names, and licenses for all installed packages, serving as a critical document for copyleft compliance auditing.

Each recipe's license manifest is stored in a dedicated subdirectory under `build/tmp/deploy/licenses/<arch>/<package-name>/`. This structured approach meticulously organizes licensing details for every package so they are easily accessible. For instance, the directory structure for some packages might appear as follows:

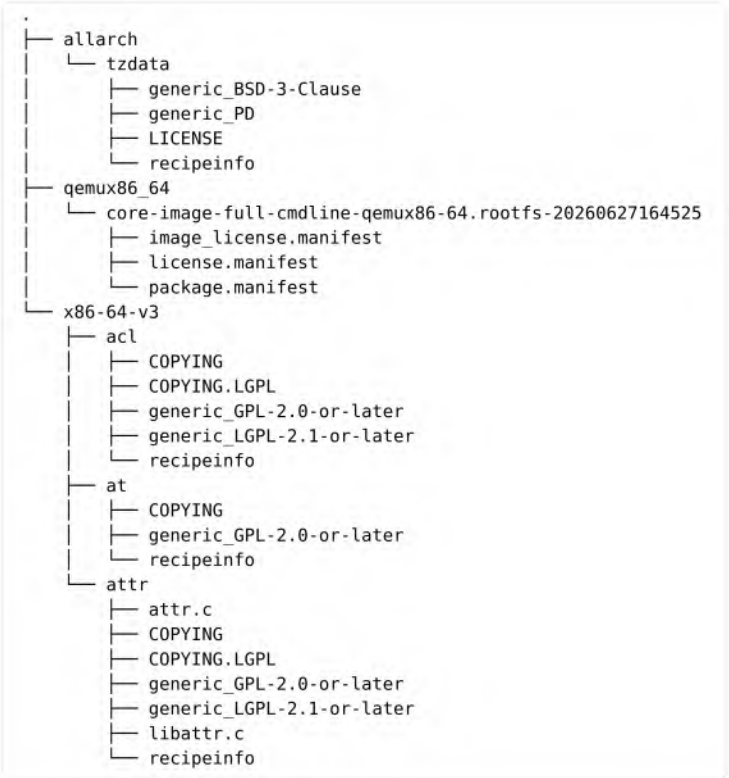


Figure 15.2: Example directory structure for license manifests within build/tmp/deploy/licenses

These manifests act as a cornerstone for copyleft compliance, offering a transparent and auditable record of licenses so that all required documentation is organized and available for distribution. Using these tools reduces the risk of non-compliance and supports the integration of open-source components into proprietary systems while honoring legal responsibilities.

The license auditing data the build system produces let developers handle copyleft compliance with predictable inputs and outputs.

Providing the source code

The Yocto Project offers mechanisms that simplify sharing source code for all projects included in an image so copyleft license obligations are met. However, careful configuration is essential to prevent the inadvertent inclusion of proprietary or unrelated source code.

A common method is to share the contents of `DL_DIR`, which houses the downloaded source code for all recipes. While straightforward, this approach has notable limitations:

1. **Inclusion of proprietary code**: Proprietary source code, if downloaded directly, may inadvertently be included
2. **Excessive scope**: Sharing `DL_DIR` may result in distributing source code for projects not governed by copyleft licenses, which is unnecessary and can pose risks

To properly manage source code for compliance, the build system must be configured to archive only the relevant components. This can be achieved by adding specific variables to the `build/conf/local.conf` file, or preferably in your custom distribution configuration. The following example illustrates the configuration:

```
INHERIT += "archiver"
ARCHIVER_MODE[src] = "original"
```

To properly configure the build system for archiving relevant source code, you need to use specific variables in your configuration. These variables help you define which licenses to include or exclude, so the archive stays within compliance while avoiding unnecessary or proprietary code:

- `COPYLEFT_LICENSE_INCLUDE`: Specifies the licenses for which source code should be archived. By default, it includes `GPL* LGPL* AGPL*` to target recipes using GPL, LGPL, and AGPL licenses.
- `COPYLEFT_LICENSE_EXCLUDE`: Defines licenses to exclude from the archive, such as `CLOSED Proprietary`, so proprietary components are omitted.

The archiver class handles the collection of source code, patches, and build scripts for recipes that meet the specified license filters. The resulting archived content is organized under `build/tmp/deploy/sources/<architecture>`, so only the relevant source code is prepared for distribution.

After baking the `core-image-full-cmdline`, the sources directory contains the filtered source code for packages under the specified licenses. The following example illustrates the directory layout, demonstrating how recipes are organized for compliance purposes:

```
.
├── allarch
│   └── gcc-source-15.2.0-15.2.0-r0
│       ├── 0001-gcc-4.3.1-ARCH_FLAGS_FOR_TARGET.patch
│       ├── gcc-15.2.0.tar.xz
│       └── series
├── allarch-poky-linux
│   ├── ca-certificates-20260601-r0
│   │   ├── 0001-Revert-mozilla-certdata2pem.py-print-a-warning-for-e.patch
│   │   ├── ca-certificates_20260601.tar.xz
│   │   └── series
│   ├── netbase-1_6.5-r0
│   │   └── netbase_6.5.tar.xz
│   └── update-rc.d-0.8+git-r0
│       └── update-rc.d-0.8+git-r0.tar.xz
└── x86_64-linux
    ├── acl-native-2.3.2-r0
    │   ├── 0001-libmisc-__acl_get_uid-fix-memory-wasting-loop-if-use.patch
    │   ├── acl-2.3.2.tar.gz
    │   ├── run-ptest
    │   └── series
    ├── alsa-lib-native-1.2.15.3-r0
    │   ├── alsa-lib-1.2.15.3.tar.bz2
    │   ├── CVE-2026-25068.patch
    │   └── series
    ├── attr-native-2.5.2-r0
    │   ├── 0001-attr.c-Include-libgen.h-for-posix-version-of-basenam.patch
    │   ├── attr-2.5.2.tar.gz
    │   ├── run-ptest
    │   └── series
    └── autoconf-archive-native-2024.10.16-r0
        ├── 3a2...e3a.patch
        ├── autoconf-archive-2024.10.16.tar.xz
        └── series
```

Figure 15.3: The build/tmp/deploy/sources directory layout

These tools and configurations let developers meet copyleft compliance requirements while keeping proprietary and unrelated source code out of the archive.

Providing compilation scripts and source code modifications

With the configuration outlined earlier, the build system packages the original source code for each project. To include patched source code, set the following:

```
ARCHIVER_MODE[src] = "patched"
```

This configuration directs the build system to capture the project source code after the `do_patch` task, integrating modifications from recipes or `.bbappend` files. This approach simplifies the process of sharing patched source code so that changes are preserved and accessible.

While sharing patched source code covers most compliance needs, it does not provide the configuration and build steps used. To address this, share the configured project (essentially the project state after the `do_configure` task) by setting the following:

```
ARCHIVER_MODE[src] = "configured"
```

This approach allows others to reproduce the build environment and workflow accurately.

Considerations for copyleft compliance

The recipient of the shared code may not use the Yocto Project. Even if they do, they might lack access to the modifications and configuration steps applied to the original source code. Sharing the configured project bridges this gap, allowing anyone to replicate your build environment and process.

Output file formats

By default, the resulting archived files are tarballs. If required, you can configure the build system to generate SRPM packages by setting:

```
ARCHIVER_MODE[srpm] = "1"
```

This flexibility allows you to meet various compliance and distribution needs effectively.

Providing license text inside the image

When distributing the source code, the license text is typically included alongside it. To enhance compliance further, configure the build system to include the license text directly within the final image by setting the variables described below:

- `COPY_LIC_MANIFEST`: Generates the license manifest file, offering a comprehensive overview of all licenses used in the image

- `COPY_LIC_DIRS`: Archives the directories containing license texts for individual packages, making them accessible for review
- `LICENSE_CREATE_PACKAGE`: Creates a standalone package containing all license texts, allowing independent deployment

`COPY_LIC_DIRS` populates `/usr/share/common-licenses/` in the root filesystem at image build time, while `LICENSE_CREATE_PACKAGE` adds a separate package whose files land under `/usr/share/licenses/` and can be upgraded after deployment. Enabling both leaves two copies of the same license text on the image, which is occasionally what you want but rarely the default.

Generating a Software Bill of Materials

A **Software Bill of Materials (SBOM)** is a structured inventory of every software component built into an image. Each entry records the package name, version, license, and the dependencies that connect it to the rest of the image.

The term names the inventory, not the file that carries it. Before a tool on the receiving end can read an SBOM, someone has to write it down in an agreed format, and two of them dominate. **Software Package Data Exchange (SPDX)** is maintained by the Linux Foundation and is the format the Yocto Project emits. CycloneDX is maintained by OWASP and turns up more often in application security tooling. Both express the same kind of inventory, so the choice usually follows what your customers and downstream consumers already accept rather than any technical limit. The distinction matters when you read a requirement: a customer who asks for an SBOM is asking for the inventory, while one who asks for SPDX is also dictating how it is written down.

The `create-spdx` class produces those documents. In Yocto Project 6.0 (Wrynose), it is an alias that inherits `create-spdx-3.0`, which sets `SPDX_VERSION` to 3.0.1. This release removed the legacy `create-spdx-2.2` class, so downstream consumers that still expect an SPDX 2.x document need an external converter.

The version numbers matter more than they look. SPDX 2.2.1 is the revision published as ISO/IEC 5962:2021, so a contract or regulation that cites that standard is naming 2.2.1 rather than anything newer, and SPDX 2.3 extended it with further security and interoperability fields. SPDX 3.0 is a break rather than an increment. It reorganizes the data model around profiles, drops every serialization except JSON-LD, and is not backward compatible, so a consumer written against 2.2 cannot read a 3.0 document at all. Profiles are the part you can influence from the build. `SPDX_PROFILES` defaults to `core build software simpleLicensing security`, which covers the document skeleton, how the image was built, the packages themselves, the license expressions, and the vulnerability data that the scanning covered later in this chapter relies on.

SBOMs serve three workflows that share the same data. License audits depend on knowing which packages ship under which terms, so downstream teams can honor copyleft obligations and answer customer questionnaires without re-deriving the manifest from the source tree. Security workflows rely on the same inventory to map known Common Vulnerabilities and Exposures (CVE) to specific image components and prioritize fixes for what actually shipped. And regulatory regimes such as the **EU Cyber Resilience Act (EU CRA)** and **U.S. Executive Order 14028** require vendors to produce a machine-readable manifest of what they ship and make it available to regulators (and, under specific contracts, to customers). A single SBOM produced at build time covers all three needs without parallel processes.

The CRA arrives in stages rather than all at once, which makes its dates worth putting on a calendar. The regulation entered into force on 10 December 2024. From 11 September 2026, Article 14 requires manufacturers to report actively exploited vulnerabilities and severe security incidents, with an early warning within 24 hours, a fuller notification within 72 hours, and a final report within 14 days for a vulnerability or one month for an incident. The remaining obligations apply from 11 December 2027. Those reporting windows are the practical argument for producing the SBOM during the build rather than assembling one under pressure. A team that already knows every component and version in a shipped image can answer the 24-hour question. A team that does not will spend those hours reconstructing the inventory.

Poky inherits `create-spdx` by default, so a stock build already emits an `<image>-<machine>.spdx.json` document into `build/tmp/deploy/images/<machine>/` alongside the image, with the supporting per-recipe SPDX files written under `build/tmp/deploy/spdx/`. The *Creating a Software Bill of Materials* section at `https://docs.yoctoproject.org/wrynose/dev-manual/sbom.html` documents the supporting variables and the customization options the class exposes. Treat SBOM generation and license monitoring as recurring product hygiene rather than a one-time deliverable, a position *Chapter 18* reinforces alongside other project-wide guidelines.

Declaring a proprietary license for your own recipes

Every recipe discussed so far carries a license the build system already recognizes. Product work introduces the opposite case, an in-house component whose license exists only in your legal department. The natural first move is `LICENSE = "CLOSED"`, and it builds cleanly, which is why it survives into so many product layers. It also empties the SBOM entry for that component. The `create-spdx` class converts `CLOSED` to the SPDX expression `NONE`, so your software ships with a documented absence of licensing rather than a description of its terms. Every third-party package in the image is accounted for, and the one component you own is a blank.

Giving that component a real entry takes two pieces of metadata. The first names the license in the recipe, alongside the checksum that guards its text, as shown below:

```
SUMMARY = "Acme device control daemon"
LICENSE = "Acme-EULA-1.0"
LIC_FILES_CHKSUM = "file://LICENSE;md5=0d0d3f6ae6e6e37a0d4dbba0a3b3ea15"

NO_GENERIC_LICENSE[Acme-EULA-1.0] = "LICENSE"
```

The name is yours to choose. Any identifier the build system does not find on the SPDX license list becomes a `LicenseRef-` entry, so `Acme-EULA-1.0` reaches the SBOM as `LicenseRef-Acme-EULA-1.0`. Names may contain letters, digits, dots, plus signs, underscores, and dashes. Spaces are not allowed because the build system parses `LICENSE` as an expression in which whitespace separates operands.

The second piece points the build system at the license text, and there are two ways to supply it. When a single license covers several recipes, place the text in the layer and extend `LICENSE_PATH` from `conf/layer.conf`, as shown below:

```
LICENSE_PATH += "${LAYERDIR}/files/custom-licenses"
```

The file in that directory must be named exactly as the `LICENSE` value and carry no extension, so the configuration above expects `files/custom-licenses/Acme-EULA-1.0`. When the license instead ships inside the component's own source tree, `NO_GENERIC_LICENSE` maps the identifier to a file relative to `${S}`, which is the form the recipe uses. The build system searches `COMMON_LICENSE_DIR` and the `LICENSE_PATH` directories first and falls back to `NO_GENERIC_LICENSE`. Choose one route or the other, because setting both for the same license stops the build with a `license-no-generic` QA error. If neither resolves, the build stops with `Cannot find any text for license <license-name>`, which turns a missing end-user license agreement into a build failure rather than a silent gap in the SBOM.

Keep `LIC_FILES_CHKSUM` distinct from both routes. It resolves each `file://` entry against `${S}` and searches neither `COMMON_LICENSE_DIR` nor the `LICENSE_PATH` directories, so it cannot refer to a layer-wide license file. The variable is mandatory for every license except `CLOSED`, so a recipe whose source tree carries no license text has nothing to checksum, which is where the `LICENSE_PATH` route usually stalls on a first attempt. For a component your own team owns, keep the agreement in the source tree and let `LIC_FILES_CHKSUM` and `NO_GENERIC_LICENSE` name the same file, as the recipe above does. Reserve `LICENSE_PATH` for a text shared across

several recipes, and give each of those recipes a `LIC_FILES_CHKSUM` entry that names a file in its own sources.

With both pieces in place, the SPDX document records the license as a first-class element. The declared license of the package becomes `LicenseRef-Acme-EULA-1.0`, and a companion element carries the full agreement text, linked from the expression through a `simplelicensing_customIdToUri` entry. The excerpt below shows that portion of the recipe's SPDX document:


```
[
  {
    "type": "simplelicensing_LicenseExpression",
    "spdxId": "http://spdx.org/spdxdocs/acme-deamon-8bd9f163-ad52-52c5-a852-
8bfbf969a57b/db4408f2978ebc3462dbf9439f370339273600288e2ac1ef168a91e290676320/license/3_28_0/LicenseRef-Acme-
EULA-1_0/70f743cc148cf7fa0b1d867afd1979a3",
    "simplelicensing_customIdToUri": [
      {
        "type": "DictionaryEntry",
        "key": "LicenseRef-Acme-EULA-1.0",
        "value": "http://spdxdocs.org/openembedded-alias/by-doc-
hash/736822efe3c97443bc074fd47c0759fb01d1bc12edecc38bda2f319e924c2e52/acme-deamon/UNIHASH/license-text/Acme-
EULA-1_0"
      }
    ],
    "simplelicensing_licenseExpression": "LicenseRef-Acme-EULA-1.0",
    "simplelicensing_licenseListVersion": "3.28.0"
  },
  {
    "type": "simplelicensing_SimpleLicensingText",
    "spdxId": "http://spdx.org/spdxdocs/acme-deamon-8bd9f163-ad52-52c5-a852-
8bfbf969a57b/db4408f2978ebc3462dbf9439f370339273600288e2ac1ef168a91e290676320/license-text/Acme-EULA-1_0",
    "simplelicensing_licenseText": "Acme Corporation End User License Agreement (Acme-EULA-1.0)\n\nThis software
is proprietary to Acme Corporation (\"Acme\") and is provided solely for use on Acme-manufactured devices. ...",
    "name": "Acme-EULA-1.0"
  }
]
```


Figure 15.4: License elements for a proprietary component in the SPDX document

Embedding the text rather than referencing it is what makes this useful for a proprietary license, because no downstream consumer can resolve `LicenseRef-Acme-EULA-1.0` against a public list. The SBOM travels as a self-contained artifact, and the customer who receives it reads your terms directly from the same document that inventories the image.

One further variable covers the case where legal review reaches a different conclusion than the recipe declares. `SPDX_CONCLUDED_LICENSE` records a concluded license for a package and leaves the declared value untouched, so the SBOM captures both what the component claims and what your review determined. Set it for the whole recipe or for a single package, as in `SPDX_CONCLUDED_LICENSE:${PN} = "Acme-EULA-1.0 AND MIT"` when a vendored dependency contributes terms of its own. Because the variable feeds the same machinery, a name used there needs its license text available through `LICENSE_PATH` or `NO_GENERIC_LICENSE` as well.

`LICENSE_FLAGS` plays no part in any of this. As the *Understanding commercial licenses* section earlier in this chapter describes, that variable gates whether a restricted recipe may be built at all, while `LICENSE` and its supporting files describe the terms a component ships under. A proprietary component of your own usually needs the second and rarely needs the first.

SBOM-based vulnerability scanning

The `sbom-cve-check` class consumes the SPDX SBOM produced by `create-spdx` and emits a vulnerability report alongside it. Enable it for an image with `OE_FRAGMENTS += "core/yocto/sbom-cve-check"`. Per-recipe scanning is available through the `sbom-cve-check-recipe` companion class for cases where you want to inspect a single output rather than a full image.

The `create-spdx` class produces SPDX 3.0 output, the format `sbom-cve-check` consumes. Several variables broaden what the SBOM captures:

- `SPDX_INCLUDE_KERNEL_CONFIG`: Embeds the active Linux kernel configuration (`CONFIG_*`) into the SBOM document so downstream consumers can correlate vulnerabilities against the running configuration.
- `SPDX_INCLUDE_PACKAGECONFIG`: Records each recipe's resolved `PACKAGECONFIG` toggles, which clarifies why a particular feature is or is not present in the produced binaries.
- `SPDX_PACKAGE_URLS`: Carries the Package URL (PURL) identifiers recorded for a recipe, the convention most supply-chain tooling uses to look up packages. Every recipe already has one without any configuration, since the default derives a base PURL from the recipe's layer, name, and version, and recipes built through `cargo`, `cpan`, `go-mod`, `npm`, or `pypi` gain the matching ecosystem PURL as well. Set the variable to record further identifiers: the first entry becomes the package's `packageUrl` property and every entry is emitted as an external reference. The singular `SPDX_PACKAGE_URL` still parses but is deprecated in favor of this one.
- `SBOM_CVE_CHECK_EXPORT_VARS`: Selects which export-format variables the class evaluates when writing reports. The default value, `SBOM_CVE_CHECK_EXPORT_VARS ?= "SBOM_CVE_CHECK_EXPORT_SPDX3 SBOM_CVE_CHECK_EXPORT_CVECHECK"`, produces the SPDX 3.0 and Yocto Project `cve-check` JSON outputs side by side. Each referenced variable carries `type` and `ext` flags that determine the report serialization and filename suffix, so downstream tooling can add or remove formats by editing this list rather than touching the class.

Together these variables anchor compliance and security reporting on a single SBOM artifact that downstream tooling can consume directly. Reports land in the deploy directory alongside the image, with `<image>.sbom-cve-check.yocto.json` for the project-native schema and

`<image>.sbom-cve-check.spdx.json` for the SPDX-aligned variant. The `vex` class emits Vulnerability Exploitability eXchange statements that record why a given CVE applies or does not apply to the build, and it writes its output to `<image>.vex.json` so it sits next to neighboring SBOM artifacts without filename collisions. Yocto Project 6.0 moved VEX generation from the package-level SPDX documents to the recipe-level documents (image recipes still carry an aggregate VEX file), with `SPDX_INCLUDE_VEX` controlling whether the statements are emitted at all. Builds upgrading from earlier releases should expect to see VEX content move accordingly.

Pairing those reports with the Build History tracking introduced in *Chapter 10* makes regressions in CVE counts surface alongside other image deltas, so security drift is caught the same way packaging drift already is.

The SBOM is also an input to tooling outside the build. Scanners such as Grype and Trivy take an SBOM file directly and match its components against their own vulnerability databases, which is useful when you want a second opinion alongside `sbom-cve-check` or when a customer runs the scan themselves. Platforms such as OWASP Dependency-Track go further and track a fleet of SBOMs over time, raising an alert when a newly published advisory affects a component in an image you shipped months ago, though Dependency-Track ingests CycloneDX rather than SPDX. The SPDX project publishes its own reference tooling for validating and converting documents.

Check what your chosen consumer actually reads before you commit to it. SPDX 3.0 changed the data model, and support across the ecosystem is uneven, so a tool that advertises SPDX may still expect a 2.x document. That is the practical reason to confirm the conversion step mentioned earlier in this chapter well before an audit rather than during one.

Note

SBOM generation, CVE scanning, and vulnerability management are complex topics that this chapter can only introduce. A production security program spans dependency tracking, patch cadence, supplier communication, and regulatory reporting, each with its own tooling and governance requirements. When your organization is working toward formal compliance with frameworks such as the EU CRA or U.S. EO 14028, seek guidance from a specialist layer such as `meta-security` or consult the Yocto Project community for layer recommendations that match your regulatory context.

Between the license manifests, the source archiver, and the SBOM and CVE tooling, the build system gives every recipe in an image a documented license, a reproducible source trail, and a machine-readable vulnerability record without any of the three requiring a separate manual process.

Summary

This chapter explored how the Yocto Project aids in achieving compliance with copyleft licenses by generating source code, reproduction scripts, and license texts for packages in our distribution. It also covered how to declare a license for the proprietary components you own, so they appear in the record rather than as a gap, and how the build system emits a Software Bill of Materials that inventories every package in the image for license audits, vulnerability tracking, and regulatory reporting. While the Yocto Project has powerful tools for managing licensing requirements, it should not be relied upon as a substitute for legal advice.

The next chapter walks through using the Yocto Project's tools with real hardware, showing how to generate images for various real-world boards.

Part 4

Bridging to Real-World Application

The final part covers what stands between a build that works and a product you can ship. It puts the image on real hardware first: how to find and judge a BSP layer, which hardware details change the way an image is built and deployed, and how to bake, flash, and boot a custom image on the BeagleBone Black, Raspberry Pi 4 and 5, VisionFive and VisionFive 2, and Arduino UNO Q.

Because hardware is often scarce, shared, or simply not ready, the next chapter turns to emulation. You will see where QEMU genuinely substitutes for a board and where it does not, then drive it with `runqemu` for graphical applications, memory-constrained scenarios, and image regression tests that can run long before hardware arrives.

The book closes with the guidelines that keep a Yocto Project-based product maintainable over the years it actually lives: how to organize layers, prepare metadata for a new release, build a custom distribution, keep patches and source revisions under control, generate a Software Bill of Materials, and weigh security, maintenance cost, and risk against delivery pressure.

This part of the book includes the following chapters:

- *Chapter 16*, *Building and Booting a Custom Embedded Linux System*
- *Chapter 17*, *Speeding Up Product Development Through Emulation (QEMU)*
- *Chapter 18*, *Best Practices*

16

Building and Booting a Custom Embedded Linux System

This chapter shows how to add a vendor-supplied Board Support Package (BSP) layer to a Yocto Project build, generate a bootable image, and write it to an SD card or flash it directly. The walkthrough covers four widely available boards: BeagleBone Black, Raspberry Pi 4 or Raspberry Pi 5, VisionFive 2, and the Arduino UNO Q. The same workflow applies to any board whose vendor publishes a compatible BSP layer.

Getting a custom image to boot on real hardware is the gap that separates a successful QEMU prototype, of the kind *Chapter 17* builds, from a shippable product. Hardware bring-up time rises sharply when the BSP layer a silicon vendor publishes lags the active Yocto Project release: drivers are missing, machine configurations conflict with updated OpenEmbedded Core classes, and build failures appear only on the target. Understanding how to evaluate and integrate a BSP layer before committing to a board keeps that cost manageable and shortens the path from first build to a running system.

We will cover the following main topics:

- Discovering the right BSP layer
- Reviewing aspects that impact hardware use
- BeagleBone black
- Raspberry Pi 4 or Raspberry Pi 5
- VisionFive 2
- Arduino UNO Q
- Taking the next steps

Technical requirements

This chapter requires a build host already configured to run Yocto Project builds and a custom layer you can extend with machine-specific settings. If you have not completed those steps, work through *Chapter 2* and *Chapter 12* first.

Beyond the standard build host, you need the following hardware:

- One of the supported development boards: BeagleBone Black, Raspberry Pi 4 or Raspberry Pi 5, VisionFive 2, or the Arduino UNO Q.
- A microSD card of at least 4 GB (8 GB or larger is recommended to give headroom for image growth), if you are building for BeagleBone Black, Raspberry Pi, or VisionFive 2. The Arduino UNO Q flashes over USB instead and does not need this card or the microSD card writer described below.
- A microSD card writer connected to the build host. The chapter uses `bmaptool` to write the image: it copies only the blocks the image actually occupies, reads the compressed `.wic` artifact directly, and verifies the result against the `.bmap` file the build generates, which makes it faster and safer than a raw `dd`. Install it from your distribution's `bmap-tools` package. A graphical writer such as Balena Etcher also works.
- A USB-C cable, if you are building for the Arduino UNO Q, to flash the image with the `qdl` command-line tool
- A USB power supply matched to your board. Raspberry Pi class boards need 5V at 2–3A. Consult your board's datasheet for the exact requirement.
- Optional but recommended: a UART-to-USB cable for serial console access. A serial console is the fastest way to diagnose boot failures when the board does not reach the login prompt.

You do not need network access on the target board to follow the core walkthrough, though some post-boot verification steps benefit from it.

Discovering the right BSP layer

In *Chapter 11*, we examined how the Yocto Project facilitates the separation of metadata into distinct layers. This modular approach empowers developers to choose specific meta layers relevant to their projects, enabling easier customization while minimizing unnecessary complexity.

To identify the appropriate BSP for a specific board, the machine or architecture often serves as the key criterion. A reliable method is to visit the *Machines* tab of the OpenEmbedded Layer Index at `https://layers.openembedded.org/layerindex/branch/wrynose/machines/`, which lists every machine the indexed layers define. By entering the machine name, you can quickly locate the corresponding meta layer that includes the required BSP metadata.

By using this tool, you can access validated and compatible BSP layers, significantly simplifying the integration of new hardware into your Yocto Project build.

Reviewing aspects that impact hardware use

The boards discussed in this chapter are reliable and straightforward to work with. However, choosing a different board is a viable option, although the experience may differ based on the board's compatibility and support.

Key considerations for board selection

When selecting a board, the first step is to thoroughly assess the quality of its software support. Key low-level components to consider include:

- **Bootloader**: Examples include U-Boot, GRUB, systemd-boot, and barebox
- **The Linux kernel**: Confirm compatibility with the necessary drivers, such as those for GPU or Wi-Fi functionality
- **User space packages**: Often essential for enabling hardware acceleration and other critical features

How a board layer carries these components over time matters as much as which components it picks. *Chapter 18* collects the guidelines that keep a board layer maintainable, especially around limiting Linux kernel and bootloader patch volume and favoring reuse of vendor BSPs over forking them.

While these components are critical, other factors also warrant attention. Effective integration within the Yocto Project, often facilitated by a BSP layer, can dramatically streamline the workflow when working with a board. Ideally, a BSP layer should offer:

- **Reusable disk partition layouts**: Such as an OpenEmbedded Kickstart `.wks` template, which facilitates consistent and efficient partitioning
- **Preconfigured machine definitions**: Streamlining the initial setup process for quicker deployment
- **Integrated user space packages**: Providing ready-to-use hardware acceleration and other essential features

> **Note**
>
> The maturity of both the software enablement and the Yocto Project BSP plays a crucial role in determining the ease of use and overall out-of-the-box experience. A well-maintained and robust BSP not only provides smoother integration but also reduces potential challenges when working with the build system on different boards.

Overview of widely used BSP layers

This chapter offers a carefully curated list of widely used BSP layers. Although not exhaustive, this guide is designed to simplify your search for the appropriate layer, particularly if you are working with a specific vendor's board. The layers are presented in alphabetical order for easy navigation:

- **Allwinner**: Supported by the `meta-sunxi` and `meta-allwinner-hx` layers
- **AMD**: Supported by the `meta-amd` layer
- **Intel**: Supported by the `meta-intel` layer
- **NXP**: Supported by the `meta-freescale` and `meta-freescale-3rdparty` layers
- **Qualcomm Technologies, Inc.**: Supported by the `meta-qcom` and `meta-qcom-3rdparty` layers
- **Raspberry Pi**: Supported by the `meta-raspberrypi` layer
- **RISC-V**: Supported by the `meta-riscv` layer
- **STMicroelectronics**: Supported by the `meta-st-stm32mp` layer
- **Texas Instruments**: Supported by the `meta-ti` layer
- **Xilinx**: Supported by the `meta-xilinx` layer

In the following sections, we will illustrate practical examples of using these layers with specific boards. This hands-on approach will enhance your understanding of how to effectively integrate BSP layers into your Yocto Project builds.

Using physical hardware

Exploring the Yocto Project's capabilities is more engaging with a physical board, allowing you to experience the process of booting your customized embedded system firsthand. To enhance accessibility, we have selected some of the most widely available boards, increasing the likelihood that you already own one. The next sections outline the steps for the following boards:

Board	Description	More information
BeagleBone Black	Community-based with members worldwide.	`https://beagleboard.org/black/`
Raspberry Pi 4	The most famous ARM64-based board with the broadest community spread.	`https://www.raspberrypi.com/products/raspberry-pi-4-model-b/`
Raspberry Pi 5	Next-generation *ARM64* board offering improved performance and features.	`https://www.raspberrypi.com/products/raspberry-pi-5/`
VisionFive 2	Affordable *RISC-V* board designed to run Linux.	`https://www.starfivetech.com/en/`

Board	Description	More information
Arduino UNO Q	Hybrid board pairing a Qualcomm Dragonwing Linux processor with an STM32 real-time microcontroller in the classic Arduino UNO form factor.	`https://www.arduino.cc/product-uno-q/`

Table 16.1: Overview of used boards

All the boards listed are supported by organizations dedicated to education and mentorship. This approach fosters active communities, making these boards excellent entry points for embedded Linux development. *Table 16.2* provides a summary of the boards and their main features:

Board version	Features
BeagleBone Black	TI AM335x (single-core Cortex-A8), 512 MB RAM
Raspberry Pi 4	Broadcom BCM2711 64-bit CPU (quad-core Cortex-A72), 1 GB, 2 GB, 4 GB, or 8 GB RAM
Raspberry Pi 5	Broadcom BCM2712 64-bit CPU (quad-core Cortex-A76), 1 GB, 2 GB, 4 GB, 8 GB, or 16 GB RAM
VisionFive 2	StarFive JH7110 (quad-core SiFive U74), 2 GB, 4 GB, or 8 GB RAM
Arduino UNO Q	Qualcomm Dragonwing QRB2210 64-bit CPU (quad-core Cortex-A53 at 2.0 GHz), 2 GB or 4 GB RAM

Table 16.2: Hardware specifications for the covered boards

In the following sections, we will guide you through baking and booting the Yocto Project image for each of the suggested machines. Focus on the section relevant to the board you own and be sure to consult the board's official documentation to fully understand the preparation steps required for operation.

Building more than one board from this chapter can reuse downloads but reuse very little Shared State Cache, since BeagleBone Black, the Raspberry Pi boards, the VisionFive 2, and the Arduino UNO Q sit on different architectures or SoC vendors. *Chapter 14* covers the `bitbake-setup` settings (`dl-dir` and `common-sstate`) that let separate BitBake Setups beneath one Top Directory reuse downloads and Shared State Cache artifacts instead of refetching and rebuilding them independently. Those settings pay off far more across multiple builds of similar architecture, for example when you have board variants, such as a development build and a secure or production build defined in separate machine configuration files, since most of the recipe set stays identical across those builds.

BeagleBone Black

This section provides a comprehensive guide to baking and booting an image for the BeagleBone Black board using the Yocto Project.

Baking for BeagleBone Black

The BeagleBone Black is supported through the `meta-yocto-bsp` layer, which is included in the `poky-wrynose` BitBake Configuration Template by default, so no extra BSP layer fetch is required. You can explore the meta layer repository and its contents at the *Board Support Packages (BSP)* section at `https://docs.yoctoproject.org/wrynose/bsp-guide/bsp.html`.

Setting up the source structure

To prepare the source structure, initialize a BitBake Setup using the `poky-wrynose` BitBake Configuration Template. From your workspace root directory:

```
$ bitbake-setup init --non-interactive \
                           poky-wrynose \
                           poky-with-sstate \
                           distro/poky \
                           machine/qemux86-64
```

This is the same `bitbake-setup init` invocation covered in *Chapter 2*, so its console output is not repeated here. See *Non-interactive project initialization* for the log.

After initialization completes, activate the build environment:

```
$ source ./bitbake-builds/poky-wrynose/build/init-build-env
```

This is the same activation command covered in *Chapter 2*, so its console output is not repeated here. See *Activating the build environment* for the log.

The `poky-wrynose` BitBake Configuration Template only offers QEMU and generic machines as `--non-interactive` init choices, so the BitBake Setup above started from the default `machine/qemux86-64` fragment. `beaglebone-yocto` comes from `meta-yocto-bsp`, already part of this template's layers, so switch to it directly with `bitbake-config-build`; no BSP layer fetch needed:

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/beaglebone-yocto
```

`bitbake-config-build` disables the placeholder fragment and enables `beaglebone-yocto` in its place, as the following log shows.

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/beaglebone-yocto
NOTE: Starting bitbake server...
Removing fragment machine/qemux86-64 from .../build/conf/toolcfg.conf
NOTE: Starting bitbake server...
Fragment machine/beaglebone-yocto added to .../build/conf/toolcfg.conf.
```

Figure 16.1: Switching to the BeagleBone Black machine fragment

Building the image

With the build environment active and the machine fragment switched, initiate the build process by running the following command:

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

By now you have built enough images to know this log runs long and varies with the layers and configuration in play, so it is not repeated here.

The `bitbake-setup` initialization and `bitbake-config-build` commands already configured the `MACHINE` and `DISTRO` settings through BitBake Configuration Fragments.

Writing the image to an SD card

Once the build process finishes, the generated image will be found in the `bitbake-builds/poky-wrynose/build/tmp/deploy/images/beaglebone-yocto/` directory.

To write the image to an SD card, use the `bmaptool` utility. It reads the compressed `.wic.zst` artifact directly, writes only the populated blocks, and validates them against the generated `.bmap` file. Replace `<media>` in the command below with the device name of your SD card. Check that name carefully, because writing to the wrong device destroys the data on your hard drive. Use the following command:

```
$ sudo bmaptool copy core-image-full-cmdline-beaglebone-yocto.rootfs.wic.zst /dev/
<media>
```

`bmaptool` copies only the populated blocks and verifies them against the `.bmap` file, as the following output shows.

```
$ sudo bmaptool copy core-image-full-cmdline-beaglebone-yocto.rootfs.wic.zst /dev/<media>
bmaptool: info: discovered bmap file 'core-image-full-cmdline-beaglebone-yocto.rootfs.wic.bmap'
bmaptool: info: block map format version 2.0
bmaptool: info: 114755 blocks of size 4096 (448.3 MiB), mapped 44539 blocks (174.0 MiB or 38.8%)
bmaptool: info: copying image 'core-image-full-cmdline-beaglebone-yocto.rootfs.wic.zst' to block device
'/dev/mmcblk0' using bmap file 'core-image-full-cmdline-beaglebone-yocto.rootfs.wic.bmap'
bmaptool: info: 100% copied
bmaptool: info: synchronizing '/dev/mmcblk0'
bmaptool: info: copying time: 15.4s, copying speed 11.3 MiB/sec
```

Figure 16.2: Writing the BeagleBone image to an SD card

Booting the board

After copying the image, insert the SD card into the BeagleBone Black's SD card slot. Connect the HDMI cable, if applicable, and power on the board. The system should boot successfully, displaying the output on the connected display.

Note

By default, the BeagleBone Black prioritizes booting from its eMMC. To force the board to boot from the SD card, press and hold the USER/BOOT button while applying power to the device. For further details and comprehensive guidance, visit the official documentation at the *Board Support Packages (BSP)* section at `https://docs.yoctoproject.org/wrynose/bsp-guide/bsp.html`.

Raspberry Pi 4 or Raspberry Pi 5

This section provides a comprehensive guide for baking and booting an image for Raspberry Pi 4 and Raspberry Pi 5 using the Yocto Project.

Baking for Raspberry Pi 4 or Raspberry Pi 5

To enable support for these boards, include the `meta-raspberrypi` BSP layer, which provides compatibility for Raspberry Pi boards, including the Raspberry Pi 4 and Raspberry Pi 5. You can access the meta layer repository and documentation at `https://github.com/agherzan/meta-raspberrypi` and `https://meta-raspberrypi.readthedocs.io/en/latest/`. The layer publishes a branch per Yocto Project release codename. Check out the `wrynose` branch to match the book's baseline.

Setting up the source structure

Start by initializing a BitBake Setup using the `poky-wrynose` BitBake Configuration Template. From your workspace root directory:

```
$ bitbake-setup init --non-interactive \
                           poky-wrynose \
                           poky-with-sstate \
                           distro/poky \
                           machine/qemux86-64
```

This is the same `bitbake-setup init` invocation covered in *Chapter 2*, so its console output is not repeated here. See *Non-interactive project initialization* for the log.

After initialization completes, activate the build environment and fetch the `meta-raspberrypi` BSP layer:

```
$ source ./bitbake-builds/poky-wrynose/build/init-build-env
$ bitbake-layers layerindex-fetch meta-raspberrypi
```

`bitbake-layers layerindex-fetch` clones `meta-raspberrypi` and registers it in `build/conf/bblayers.conf`, as the following log shows.

```
$ source ./bitbake-builds/poky-wrynose/build/init-build-env
$ bitbake-layers layerindex-fetch meta-raspberrypi
NOTE: Starting bitbake server...
Loading https://layers.openembedded.org/layerindex/api/;branch=wrynose...
Layer                                          Git repository (branch)                          Subdirectory
=============================================================================================================
local:HEAD:openembedded-core                   file://~/bitbake-builds/.bitbake-setup-
downloads/git2/git.openembedded.org.openembedded-core (wrynose) meta
  required by: meta-raspberrypi
layers.openembedded.org:wrynose:meta-raspberrypi   https://git.yoctoproject.org/meta-raspberrypi (wrynose)
Cloning into '.../layers/openembedded-core/meta-raspberrypi'...
remote: Enumerating objects: 12336, done.
remote: Total 12336 (delta 0), reused 0 (delta 0), pack-reused 12336 (from 1)
Receiving objects: 100% (12336/12336), 3.66 MiB | 2.17 MiB/s, done.
Resolving deltas: 100% (7265/7265), done.
Adding layer "meta-raspberrypi" (.../layers/openembedded-core/meta-raspberrypi/) to conf/bblayers.conf
```

Figure 16.3: Activating the environment and fetching meta-raspberrypi

The `poky-wrynose` BitBake Configuration Template requires a `machine/` fragment at initialization time, so the BitBake Setup above started from the default `machine/qemux86-64` fragment. Switch to your target board by disabling that placeholder fragment and enabling the board-specific one.

For Raspberry Pi 4:

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/raspberrypi4-64
```

`bitbake-config-build` disables the placeholder fragment and enables `raspberrypi4-64`, as the following log shows.

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/raspberrypi4-64
NOTE: Starting bitbake server...
Removing fragment machine/qemux86-64 from .../build/conf/toolcfg.conf
NOTE: Starting bitbake server...
Fragment machine/raspberrypi4-64 added to .../build/conf/toolcfg.conf.
```

Figure 16.4: Enabling the machine fragment for Raspberry Pi 4

The `raspberrypi4-64` machine produces a 64-bit userspace, which is the default choice for new projects. The `meta-raspberrypi` layer also provides a `raspberrypi4` machine that produces a 32-bit userspace, available for legacy builds that must stay on *ARMv7*.

For Raspberry Pi 5:

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/raspberrypi5
```

`bitbake-config-build` disables the placeholder fragment and enables `raspberrypi5`, as the following log shows.

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/raspberrypi5
NOTE: Starting bitbake server...
Removing fragment machine/qemux86-64 from .../build/conf/toolcfg.conf
NOTE: Starting bitbake server...
Fragment machine/raspberrypi5 added to .../build/conf/toolcfg.conf.
```

Figure 16.5: Enabling the machine fragment for Raspberry Pi 5

Building the image

Once the build environment is active and the BSP layer is added, initiate the image build process:

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

By now you have built enough images to know this log runs long and varies with the layers and configuration in play, so it is not repeated here.

The `bitbake-setup` initialization and `bitbake-config-build` commands already configured the `MACHINE` and `DISTRO` settings through BitBake Configuration Fragments. To switch between Raspberry Pi 4 and Raspberry Pi 5, use `bitbake-config-build disable-fragment` on the active machine fragment and `bitbake-config-build enable-fragment` on the target one.

Writing the image to an SD card

After the build process completes, you will find the generated image in the respective directories:

- **Raspberry Pi 4**: `bitbake-builds/poky-wrynose/build/tmp/deploy/images/raspberrypi4-64/`
- **Raspberry Pi 5**: `bitbake-builds/poky-wrynose/build/tmp/deploy/images/raspberrypi5/`

Both files are compressed for efficient storage.

To write the image to an SD card, use the `bmaptool` utility, which reads the compressed `.wic.bz2` directly and writes only the blocks the image uses. Replace `<media>` in the commands below with the device name of your SD card. Check that name carefully, because writing to the wrong device destroys the data on your hard drive. Execute the following commands:

For Raspberry Pi 4:

```
$ sudo bmaptool copy core-image-full-cmdline-raspberrypi4-64.rootfs.wic.bz2 /dev/
<media>
```

`bmaptool` writes the Raspberry Pi 4 image to the card, as the following output shows.

```
$ sudo bmaptool copy core-image-full-cmdline-raspberrypi4-64.rootfs.wic.bz2 /dev/<media>
bmaptool: info: discovered bmap file 'core-image-full-cmdline-raspberrypi4-64.rootfs.wic.bmap'
bmaptool: info: block map format version 2.0
bmaptool: info: 169267 blocks of size 4096 (661.2 MiB), mapped 68800 blocks (268.8 MiB or 40.6%)
bmaptool: info: copying image 'core-image-full-cmdline-raspberrypi4-64.rootfs.wic.bz2' to block device
'/dev/mmcblk0' using bmap file 'core-image-full-cmdline-raspberrypi4-64.rootfs.wic.bmap'
bmaptool: info: 100% copied
bmaptool: info: synchronizing '/dev/mmcblk0'
bmaptool: info: copying time: 22.1s, copying speed 12.1 MiB/sec
```

Figure 16.6: Writing the Raspberry Pi 4 image to an SD card

The same `bmaptool` command applies to Raspberry Pi 5, with only the image filename changing.

For Raspberry Pi 5:

```
$ sudo bmaptool copy core-image-full-cmdline-raspberrypi5.rootfs.wic.bz2 /dev/
<media>
```

`bmaptool` writes the Raspberry Pi 5 image to the card, as the following output shows.

```
$ sudo bmaptool copy core-image-full-cmdline-raspberrypi5.rootfs.wic.bz2 /dev/<media>
bmaptool: info: discovered bmap file 'core-image-full-cmdline-raspberrypi5.rootfs.wic.bmap'
bmaptool: info: block map format version 2.0
bmaptool: info: 169267 blocks of size 4096 (661.2 MiB), mapped 68800 blocks (268.8 MiB or 40.6%)
bmaptool: info: copying image 'core-image-full-cmdline-raspberrypi5.rootfs.wic.bz2' to block device
'/dev/mmcblk0' using bmap file 'core-image-full-cmdline-raspberrypi5.rootfs.wic.bmap'
bmaptool: info: 100% copied
bmaptool: info: synchronizing '/dev/mmcblk0'
bmaptool: info: copying time: 22.1s, copying speed 12.1 MiB/sec
```

Figure 16.7: Writing the Raspberry Pi 5 image to an SD card

Booting the board

After writing the image, insert the SD card into the corresponding Raspberry Pi's SD card slot. Connect the HDMI cable and power on the device. The system should boot successfully, displaying the output on the connected display.

VisionFive 2

This section provides a detailed guide for baking and booting an image for the VisionFive 2 board using the Yocto Project.

Baking for VisionFive 2

To enable support for VisionFive 2, include the `meta-riscv` BSP layer. This layer provides compatibility for a wide range of RISC-V-based boards, including the VisionFive series. Access the meta layer repository at `https://github.com/riscv/meta-riscv` for detailed information and updates. The layer publishes a branch per Yocto Project release codename. Check out the `wrynose` branch to match the book's baseline.

The original VisionFive board is not covered here. StarFive has discontinued it, and `meta-riscv` lists its `visionfive` machine in the layer's `DEPRECATED.md` file as unsupported from Yocto Project 6.1 onward. Choose the VisionFive 2 for new work.

Setting up the source structure

Start by initializing a BitBake Setup using the `poky-wrynose` BitBake Configuration Template. From your workspace root directory:

```
$ bitbake-setup init --non-interactive \
                           poky-wrynose \
                           poky-with-sstate \
                           distro/poky \
                           machine/qemux86-64
```

This is the same `bitbake-setup init` invocation covered in *Chapter 2*, so its console output is not repeated here. See *Non-Interactive Project Initialization* for the log.

After initialization completes, activate the build environment and fetch the `meta-riscv` BSP layer:

```
$ source ./bitbake-builds/poky-wrynose/build/init-build-env
$ bitbake-layers layerindex-fetch meta-riscv
```

`bitbake-layers layerindex-fetch` clones `meta-riscv` and registers it in `build/conf/bblayers.conf`, as the following log shows.

```
$ source ./bitbake-builds/poky-wrynose/build/init-build-env
$ bitbake-layers layerindex-fetch meta-riscv
NOTE: Starting bitbake server...
Loading https://layers.openembedded.org/layerindex/api/;branch=wrynose...
Layer                                          Git repository (branch)                          Subdirectory
=============================================================================================================
layers.openembedded.org:wrynose:meta-riscv     https://github.com/riscv/meta-riscv (wrynose)
Cloning into '.../layers/openembedded-core/meta-riscv'...
remote: Enumerating objects: 6479, done.
remote: Counting objects: 100% (146/146), done.
remote: Compressing objects: 100% (98/98), done.
remote: Total 6479 (delta 86), reused 48 (delta 48), pack-reused 6333 (from 4)
Receiving objects: 100% (6479/6479), 9.06 MiB | 24.73 MiB/s, done.
Resolving deltas: 100% (3157/3157), done.
Adding layer "meta-riscv" (.../layers/openembedded-core/meta-riscv/) to conf/bblayers.conf
```

Figure 16.8: Activating the environment and fetching meta-riscv

The `poky-wrynose` BitBake Configuration Template requires a `machine/` fragment at initialization time, so the BitBake Setup above started from the default `machine/qemux86-64` fragment. Switch to the `visionfive2` machine by disabling that placeholder fragment and enabling the board-specific one:

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/visionfive2
```

`bitbake-config-build` disables the placeholder fragment and enables `visionfive2`, as the following log shows.

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/visionfive2
NOTE: Starting bitbake server...
Removing fragment machine/qemux86-64 from .../build/conf/toolcfg.conf
NOTE: Starting bitbake server...
Fragment machine/visionfive2 added to .../build/conf/toolcfg.conf.
```

Figure 16.9: Enabling the machine fragment for VisionFive 2

Building the image

With the build environment active and the BSP layer added, initiate the image build process:

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

By now you have built enough images to know this log runs long and varies with the layers and configuration in play, so it is not repeated here.

The `bitbake-setup` initialization and `bitbake-config-build` commands already configured the `MACHINE` and `DISTRO` settings through BitBake Configuration Fragments.

Writing the image to an SD card

After the build process finishes, the generated image is in the `bitbake-builds/poky-wrynose/build/tmp/deploy/images/visionfive2/` directory.

To safely write the image to an SD card, use the `bmaptool` utility, which reads the compressed `.wic.gz` directly and writes only the used blocks. Replace `<media>` in the command below with the device name of your SD card. Check that name carefully, because writing to the wrong device destroys the data on your hard drive. Execute the following command:

```
$ sudo bmaptool copy core-image-full-cmdline-visionfive2.rootfs.wic.gz /dev/
<media>
```

`bmaptool` writes the VisionFive 2 image to the card, as the following output shows.

```
$ sudo bmaptool copy core-image-full-cmdline-visionfive2.rootfs.wic.gz /dev/<media>
bmaptool: info: discovered bmap file 'core-image-full-cmdline-visionfive2.rootfs.wic.bmap'
bmaptool: info: block map format version 2.0
bmaptool: info: 375603 blocks of size 4096 (1.4 GiB), mapped 55526 blocks (216.9 MiB or 14.8%)
bmaptool: info: copying image 'core-image-full-cmdline-visionfive2.rootfs.wic.gz' to block device '/dev/mmcblk0'
using bmap file 'core-image-full-cmdline-visionfive2.rootfs.wic.bmap'
bmaptool: info: 100% copied
bmaptool: info: synchronizing '/dev/mmcblk0'
bmaptool: info: copying time: 18.4s, copying speed 11.8 MiB/sec
```

Figure 16.10: Writing the VisionFive 2 image to an SD card

Booting the board

After transferring the image to the SD card, insert it into the SD card slot, connect any necessary peripherals such as an HDMI cable and a keyboard, and then power on the board to initiate the boot process.

Note

The VisionFive 2 board does not have a default boot target and requires manual intervention to boot. To boot the system, use a serial console to access the U-Boot prompt and execute the following commands: `setenv bootcmd "run distro_bootcmd"`, `saveenv`, and `boot`. Using the `saveenv` command is optional, but it saves the configuration permanently, allowing the board to boot automatically without manual intervention during subsequent reboots. For instructions on accessing the serial console, refer to the board manufacturer documentation at `https://www.starfivetech.com/en/`.

Arduino UNO Q

This section provides a comprehensive guide for baking and flashing an image for the Arduino UNO Q using the Yocto Project.

This board needs more setup than the others in this chapter, so it helps to know the shape of the work before you start. You complete four phases. First, you register three layers, because `meta-qcom` depends on a layer that tracks a different branch and `meta-qcom-3rdparty` is absent from the layer index. Second, you add two variables to `build/conf/local.conf`, because the `uno-q` machine does not produce the flashable image type by default. Third, you build the image. Fourth, you flash it over USB with `qdl`, because the board boots from eMMC and has no SD card slot. Each phase has its own subsection below.

Baking for the Arduino UNO Q

The Arduino UNO Q pairs a Qualcomm Dragonwing QRB2210 application processor, which runs Debian-based Linux, with an STM32U585 microcontroller for real-time and Arduino-sketch workloads. This chapter covers only the Linux side: building and flashing the image that runs on the QRB2210. The board is supported through the `meta-qcom` BSP layer, alongside `meta-qcom-3rdparty` for the pieces Qualcomm Technologies keeps outside the mainline layer. `meta-qcom-3rdparty` is the layer that defines the uno-q machine, and that definition builds on the QCM2290 processor support `meta-qcom` provides, so both layers must be registered. You can access the meta layer repositories at `https://github.com/qualcomm-linux/meta-qcom` and `https://github.com/qualcomm-linux/meta-qcom-3rdparty`. Both publish a `wrynose` branch to match the book's baseline.

Setting up the source structure

Start by initializing a BitBake Setup using the `poky-wrynose` BitBake Configuration Template. From your workspace root directory:

```
$ bitbake-setup init --non-interactive \
                           --setup-dir-name uno-q \
                           poky-wrynose \
                           poky-with-sstate \
                           distro/poky \
                           machine/qemux86-64
```

This is the `bitbake-setup init` invocation covered in *Chapter 2*, with `--setup-dir-name uno-q` added so the BitBake Setup lands in `bitbake-builds/uno-q/` instead of the `poky-wrynose` directory the template name would otherwise produce. The remaining console output is unchanged, so it is not repeated here. See *Non-interactive project initialization* for the log.

After initialization completes, activate the build environment and fetch `lts-linux-firmware-mixin` first, before `meta-qcom`. The `meta-qcom` layer declares a hard dependency on `lts-linux-firmware-mixin`, but that layer's branch is `master`, not `wrynose` like every other layer in this setup: `bitbake-layers layerindex-fetch -b wrynose lts-linux-firmware-mixin` fails with `ERROR: Layer "lts-linux-firmware-mixin" not found in layer index`, since the layer index carries no `wrynose` branch for it. Fetch it from `master` instead, and do this before fetching `meta-qcom`, or `meta-qcom` clones successfully and then aborts with `ERROR: Layer 'qcom' depends on layer 'lts-linux-firmware-mixin', but this layer is not enabled in your configuration`.

```
$ source ./bitbake-builds/uno-q/build/init-build-env
$ bitbake-layers layerindex-fetch -b master lts-linux-firmware-mixin
```

`bitbake-layers layerindex-fetch -b master lts-linux-firmware-mixin` clones the layer and registers it in `build/conf/bblayers.conf`, as the following log shows.

```
$ source ./bitbake-builds/uno-q/build/init-build-env
$ bitbake-layers layerindex-fetch -b master lts-linux-firmware-mixin
NOTE: Starting bitbake server...
Loading https://layers.openembedded.org/layerindex/api/;branch=master...
Layer                                          Git repository (branch)                          Subdirectory
==========================================================================================================
local:HEAD:openembedded-core                   file://~/bitbake-builds/.bitbake-setup-
downloads/git2/git.openembedded.org.openembedded-core (wrynose) meta
  required by: lts-linux-firmware-mixin
layers.openembedded.org:master:lts-linux-firmware-mixin https://git.yoctoproject.org/meta-lts-mixins
(wrynose/linux-firmware)
Cloning into '~/bitbake-builds/uno-q/layers/openembedded-core/meta-lts-mixins'...
remote: Enumerating objects: 5216, done.
remote: Total 5216 (delta 0), reused 0 (delta 0), pack-reused 5216 (from 1)
Receiving objects: 100% (5216/5216), 2.75 MiB | 1.56 MiB/s, done.
Resolving deltas: 100% (3321/3321), done.
Adding layer "meta-lts-mixins" (~/bitbake-builds/uno-q/layers/openembedded-core/meta-lts-mixins/) to
conf/bblayers.conf
```

Figure 16.11: Fetching lts-linux-firmware-mixin for the Arduino UNO Q

With that dependency in place, fetch `meta-qcom`:

```
$ bitbake-layers layerindex-fetch -b wrynose meta-qcom
```

`bitbake-layers layerindex-fetch -b wrynose meta-qcom` clones the layer and registers it in `build/conf/bblayers.conf`, as the following log shows.

```
$ bitbake-layers layerindex-fetch -b wrynose meta-qcom
NOTE: Starting bitbake server...
Loading https://layers.openembedded.org/layerindex/api/;branch=wrynose...
Layer                                          Git repository (branch)                          Subdirectory
==========================================================================================================
local:HEAD:openembedded-core                   file://~/bitbake-builds/.bitbake-setup-
downloads/git2/git.openembedded.org.openembedded-core (wrynose) meta
  required by: meta-qcom meta-arm-toolchain meta-arm meta-oe
layers.openembedded.org:wrynose:meta-arm-toolchain https://git.yoctoproject.org/meta-arm (wrynose)   meta-arm-
toolchain
  required by: meta-arm
layers.openembedded.org:wrynose:meta-arm           https://git.yoctoproject.org/meta-arm (wrynose)   meta-arm
  recommended by: meta-qcom
layers.openembedded.org:wrynose:meta-oe            https://git.openembedded.org/meta-openembedded (wrynose)
meta-oe
  recommended by: meta-qcom
layers.openembedded.org:wrynose:meta-qcom          https://github.com/qualcomm-linux/meta-qcom/ (wrynose)
Cloning into '~/bitbake-builds/uno-q/layers/openembedded-core/meta-arm'...
Cloning into '~/bitbake-builds/uno-q/layers/openembedded-core/meta-openembedded'...
Cloning into '~/bitbake-builds/uno-q/layers/openembedded-core/meta-qcom'...
Adding layer "meta-arm-toolchain" (~/bitbake-builds/uno-q/layers/openembedded-core/meta-arm/meta-arm-toolchain)
to conf/bblayers.conf
Adding layer "meta-arm" (~/bitbake-builds/uno-q/layers/openembedded-core/meta-arm/meta-arm) to
conf/bblayers.conf
Adding layer "meta-oe" (~/bitbake-builds/uno-q/layers/openembedded-core/meta-openembedded/meta-oe) to
conf/bblayers.conf
Adding layer "meta-qcom" (~/bitbake-builds/uno-q/layers/openembedded-core/meta-qcom/) to conf/bblayers.conf
```

Figure 16.12: Fetching meta-qcom for the Arduino UNO Q

`meta-qcom-3rdparty` is not indexed in the layer index, so clone it directly and add it as a layer:

```
$ git clone https://github.com/qualcomm-linux/meta-qcom-3rdparty.git \
            -b wrynose ../layers/meta-qcom-3rdparty
$ bitbake-layers add-layer ../layers/meta-qcom-3rdparty
```

The `git clone` command fetches the layer and `bitbake-layers add-layer` registers it in `build/conf/bblayers.conf`, as the following log shows.

```
$ git clone https://github.com/qualcomm-linux/meta-qcom-3rdparty.git \
            -b wrynose ../layers/meta-qcom-3rdparty
$ bitbake-layers add-layer ../layers/meta-qcom-3rdparty
Cloning into '../layers/meta-qcom-3rdparty'...
remote: Enumerating objects: 598, done.
remote: Counting objects: 100% (250/250), done.
remote: Compressing objects: 100% (98/98), done.
remote: Total 598 (delta 176), reused 165 (delta 152), pack-reused 348 (from 1)
Receiving objects: 100% (598/598), 144.37 KiB | 6.28 MiB/s, done.
Resolving deltas: 100% (294/294), done.
NOTE: Starting bitbake server...
```

Figure 16.13: Cloning meta-qcom-3rdparty for the Arduino UNO Q

Verify every layer landed and registered correctly:

```
$ bitbake-layers show-layers
```

`bitbake-layers show-layers` lists every registered layer, as the following output shows.

```
$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer                 path                                                                                  priority
==========================================================================================================================
core                  ~/bitbake-builds/uno-q/layers/openembedded-core/meta          5
yoctobsp              ~/bitbake-builds/uno-q/layers/meta-yocto/meta-yocto-bsp        5
yocto                 ~/bitbake-builds/uno-q/layers/meta-yocto/meta-poky            5
lts-linux-firmware-mixin  ~/bitbake-builds/uno-q/layers/openembedded-core/meta-lts-mixins  5
arm-toolchain         ~/bitbake-builds/uno-q/layers/openembedded-core/meta-arm/meta-arm-toolchain  5
meta-arm              ~/bitbake-builds/uno-q/layers/openembedded-core/meta-arm/meta-arm  5
openembedded-layer    ~/bitbake-builds/uno-q/layers/openembedded-core/meta-openembedded/meta-oe  5
qcom                  ~/bitbake-builds/uno-q/layers/openembedded-core/meta-qcom      6
qcom-3rdparty         ~/bitbake-builds/uno-q/layers/meta-qcom-3rdparty               5
```

Figure 16.14: Verifying layers for the Arduino UNO Q

The `poky-wrynose` BitBake Configuration Template requires a `machine/` fragment at initialization time, so the BitBake Setup above started from the default `machine/qemux86-64` fragment. Switch to the `uno-q` machine by disabling that placeholder fragment and enabling the board-specific one:

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/uno-q
```

`bitbake-config-build` disables the placeholder fragment and enables `uno-q` in its place, as the following log shows.

```
$ bitbake-config-build disable-fragment machine/qemux86-64
$ bitbake-config-build enable-fragment machine/uno-q
NOTE: Starting bitbake server...
Removing fragment machine/qemux86-64 from ~/bitbake-builds/uno-q/build/conf/toolcfg.conf
NOTE: Starting bitbake server...
Fragment machine/uno-q added to ~/bitbake-builds/uno-q/build/conf/toolcfg.conf.
```

Figure 16.15: Switching to the Arduino UNO Q machine fragment

Enabling the qcomflash image type

The `uno-q` machine configuration only sets `IMAGE_FSTYPES` to `ext4` by default. Add two variables to `build/conf/local.conf` before building so BitBake also produces a `qcomflash` package, the image type that `qdl`, covered later in this section, needs to flash the board:

```
IMAGE_CLASSES += "image_types_qcom"
IMAGE_FSTYPES += "qcomflash"
```

`IMAGE_CLASSES += "image_types_qcom"` inherits the `image_types_qcom` class from `meta-qcom`, which defines the `qcomflash` image type, and `IMAGE_FSTYPES += "qcomflash"` adds that type to the build so BitBake packages the Firehose programmer and partition definitions `qdl` needs. `meta-qcom` is already registered from earlier in this section, so the class resolves without any further layer setup.

Building the image

With the build environment active and every layer registered, initiate the build process:

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

By now you have built enough images to know this log runs long and varies with the layers and configuration in play, so it is not repeated here.

Flashing with qdl

Unlike the other boards in this chapter, the Arduino UNO Q does not boot from a manually written SD card. Once the build completes, the generated image is in `bitbake-builds/uno-q/build/tmp/deploy/images/uno-q/`. Flash it onto the board's eMMC storage with `qdl`, the command-line Emergency Download tool that the `meta-qcom` project documents for flashing its own Yocto Project builds. Build it once per host by following the *Build* section at `https://github.com/linux-msm/qdl`: install the listed dependencies for your platform, then run `meson setup build` and `meson compile -C build` inside the cloned repository. On Linux, running `qdl` as a non-root user also needs a `udev` rule granting access to the EDL device, which the `qdl` documentation does not cover. The *Step 2: Configure udev Rules* and *Step 3: Set User Permissions* sections at `https://docs.arduino.cc/software/app-lab/setup/linux/#step-2-configure-udev-rules` supply the rule for the Qualcomm EDL vendor ID `05c6` and add your user to the `dialout` group.

`qdl` flashes the `qcomflash` package built in the previous section, not the raw deploy directory, so extract it first:

```
$ tar xzf bitbake-builds/uno-q/build/tmp/deploy/images/uno-q/core-image-full-cmdline-uno-q.rootfs.qcomflash.tar.gz
$ cd core-image-full-cmdline-uno-q
```

Put the board into Emergency Download (EDL) mode over USB. The *Step 1: Set your board to EDL mode* section at `https://docs.arduino.cc/software/app-lab/configure/flash/#step-1-set-your-board-to-edl-mode` covers shorting the EDL pins and connecting the board. With the board enumerated over USB, run `qdl` from inside the directory you just extracted:

```
$ ../qdl/build/qdl --storage emmc prog_firehose_ddr.elf rawprogram*.xml patch*.xml
```

`--storage emmc` is required because `qdl` defaults to UFS, while the QRB2210 on the Arduino UNO Q stores its root filesystem on eMMC. `qdl` prints a `flashed "<partition>" successfully` line for each partition the Firehose XML programs, both A and B slots for `xbl`, `tz`, `rpm`, `hyp`, `boot`, and `dtb` among others, before it writes `rootfs` and the GPT headers. The full log runs to several hundred lines, so it is not repeated here. Once every partition lands, `qdl` applies the partition-table patches, marks the new slot bootable, and issues a reset that power-cycles

the board automatically. Add `--debug` to the command above if flashing stops partway through: it prints the full Firehose protocol exchange, which is the fastest way to see which partition failed.

Booting the board

The QRB2210 boots U-Boot from eMMC, which loads the Linux kernel built earlier in this section. `qdl` already power-cycled the board once flashing finished. Arduino App Lab cannot detect a board running our `core-image-full-cmdline` image, so reach the Linux console over a serial connection instead. The Arduino BugHopper, an FTDI USB-to-UART adapter that connects to the board's JCTL header, provides that serial connection without tying up the board's primary USB-C port. See its user manual at `https://docs.arduino.cc/tutorials/bughopper/user-manual/`.

Note

`core-image-full-cmdline` does not include an SSH server or a Wi-Fi configuration, unlike Arduino's stock App Lab image, which enables both during its own first-time setup. You cannot reach the board built in this chapter over the network out of the box. If you want that, consider adding `ssh-server-openssh` to `IMAGE_FEATURES`, covered in *Chapter 12*, along with your own Wi-Fi configuration, then rebuild and reflash.

Taking the next steps

Congratulations! You've achieved a significant milestone in mastering the Yocto Project build system. With the foundational knowledge you've acquired, you're now well-prepared to expand your expertise into other areas of the Yocto Project ecosystem. This chapter focused on the essentials of taking a BSP layer from selection to a booted image on real hardware, but there's still a wealth of knowledge to uncover. Here are some practical next steps to help reinforce your skills and broaden your understanding:

- **Leveraging the source code**: The source code is your ultimate resource for learning and troubleshooting. Familiarize yourself with its structure and content.
- **Finding similar recipes**: When tackling a new task, look for examples of similar recipes. Studying existing solutions can save time and help you avoid common pitfalls.
- **Creating .bbappend files**: Use `.bbappend` files to apply patches or make targeted modifications to existing recipes, maintaining changes that are modular and maintainable.

- **Building custom images**: Craft tailored images by defining the packages and configurations required for your specific use case.
- **Configuring the Linux kernel**: Modify the `defconfig` file to enable or disable Linux kernel features that align with your project's requirements.
- **Customizing BusyBox**: Fine-tune BusyBox by including or removing configuration fragments, optimizing it for your layer's needs.
- **Adding new recipes**: Develop new recipes to integrate additional packages and functionalities into your build seamlessly.
- **Developing a product layer**: Create a dedicated layer for your product, containing machine configurations, recipes, and custom images.

Hardware bring-up rarely lands on the first try. Boards stall in U-Boot, the Linux kernel panics before the root filesystem mounts, and serial consoles surface unfamiliar tracebacks. The toolkit in *Chapter 10*, from `bitbake-getvar` and Development Shell sessions to Build History diffs across image revisions, turns those board-side surprises into questions you can answer from the build host.

Contributing to the community

As your expertise develops, you'll discover numerous opportunities to contribute to OpenEmbedded Core, meta layers, or BSPs. Whether addressing bugs, adding new features, or improving existing functionality, engaging with the community can be both impactful and rewarding. Sharing your patches and enhancements is an excellent way to give back and collaborate with others.

Embracing feedback

Submitting patches offers a valuable learning experience. View feedback and change requests as opportunities to improve your approach and sharpen your problem-solving skills. Active collaboration within the community fosters both personal and professional growth.

With determination and consistent practice, you'll evolve into a skilled user and contributor to the Yocto Project. This is just the beginning: keep exploring, experimenting, and expanding the limits of your expertise.

Summary

In this chapter, we explored the process of identifying the BSP for a target board and integrating it into our project. By applying this knowledge, we strengthened our Yocto Project expertise, successfully adding external BSP layers and deploying generated images onto real hardware. Additionally, we built a solid foundation for exploring other advanced aspects of the Yocto Project as needed.

The next chapter shows how QEMU accelerates development by letting you boot built images on a simulated machine, validate memory constraints before committing to hardware, and run image-level regression tests without a physical board at every iteration.

17

Speeding Up Product Development Through Emulation (QEMU)

Every hardware bring-up cycle that depends on shared lab equipment has a cost: engineers wait for a board, lose their local reproduction of a failure, or push untested Linux kernel changes to a device that takes minutes to flash. Those delays compound quickly when an entire team is iterating on the same image. Quick EMUlator (QEMU) removes the hardware queue from the critical path by letting developers boot and test Yocto Project images on any workstation at any time, without touching physical equipment.

This chapter examines the concrete advantages QEMU offers over real hardware, identifies the situations where physical hardware still matters, and develops the `runqemu` helper into a practical testing tool. You will learn how to select Linux kernel images and root filesystems at boot time, pass custom boot arguments, reach the guest over a serial console, validate graphical applications and audio stacks, and tailor the emulated environment with QEMU-specific parameters. The chapter closes with the two automated suites the emulator makes cheap to run: image regression tests through the `testimage` class, and package tests through `ptest-runner`. The result is a local, reproducible test loop that shortens every developer iteration and keeps hardware testing focused on what only hardware can confirm.

We will cover the following main topics:

- What Is QEMU?
- What are the benefits of using QEMU over hardware?

- When is choosing real hardware preferable?
- Using `runqemu` capabilities

Technical requirements

This chapter builds on the host setup described in *Chapter 2*. You need a build host configured per that chapter, including a working BitBake Setup workspace and all required host packages.

You also need a built image targeting the `qemux86-64` machine. A `core-image-full-cmdline` build is sufficient to follow the core examples. The graphical sections require `core-image-weston`. If you completed *Chapter 2*, the `core-image-full-cmdline` image is already available.

Hardware acceleration through KVM is optional but recommended. On a Linux host with `/dev/kvm` accessible to your user account, `runqemu` enables KVM automatically, which produces significantly faster guest execution. The chapter works without KVM, but the emulated guest will run slower.

Allocate at least 2 GB of free RAM per QEMU instance you intend to run. Running multiple instances simultaneously multiplies that requirement.

What is QEMU?

Quick EMUlator (**QEMU**) is a free, open-source software tool that emulates multiple architectures on a single physical machine. As a system emulator, it virtualizes complete hardware components, including CPUs, memory, storage, and peripherals.

Using QEMU for testing and debugging saves time and effort during development by letting developers test code in diverse simulated environments without deploying to physical hardware.

The Yocto Project integrates QEMU to execute automated Quality Assurance (QA) tests on final images accompanying each release. This integration enables efficient testing and validation of images and applications across supported architectures. The complete `runqemu` option matrix is documented in the *Using the Quick EMUlator (QEMU)* chapter at `https://docs.yoctoproject.org/wrynose/dev-manual/qemu.html`.

What are the benefits of using QEMU over hardware?

Several scenarios favor QEMU over real hardware for testing and debugging:

- **Quick environment setup**: Test code in diverse simulated environments without repeatedly deploying to target devices
- **Hardware availability**: Continue development when required hardware is unavailable or in limited supply
- **Multi-platform testing**: Evaluate software across multiple hardware platforms without the expense or complexity of setting up numerous physical machines
- **Controlled debugging**: Debug software in isolated environments with specific constraints, such as limited memory, to analyze behavior under precise conditions
- **Efficient validation**: Validate software that is not hardware-specific, saving time on tasks like firmware flashing or hardware setup configurations

QEMU, as a software emulator, may not fully replicate real hardware in all cases. Testing on actual hardware remains critical to confirm software performs correctly in its intended environment.

When is choosing real hardware preferable?

Several situations require real hardware for testing and debugging:

- **Reliance on specific hardware features**: Software that depends on specialized hardware capabilities, such as a unique **Video Processing Unit** (**VPU**) or **Graphics Processing Unit** (**GPU**) feature, requires physical hardware to validate behavior.
- **Performance evaluation**: Accurate measurement of software performance requires real hardware. QEMU cannot fully replicate the speed and efficiency of physical devices.
- **Bootloader interaction**: `runqemu` boots the Linux kernel directly and skips the bootloader. Tests that cover how the running system and the bootloader interact require real hardware. An update agent that sets a bootloader variable to select a boot partition is one example.

While QEMU is an invaluable tool for software testing and debugging, it cannot replace the precision and reliability of real hardware. *Chapter 16* walks through the matching real-hardware workflow on BeagleBone Black, Raspberry Pi, VisionFive 2, and Arduino UNO Q boards. The recipes and image definitions you validate against QEMU carry over directly.

Switching the `MACHINE` value to the target board and rebuilding is what produces the artifacts that boot on the physical hardware.

Using runqemu capabilities

QEMU is a key component of the Yocto Project, offering essential emulation capabilities to streamline testing and debugging processes. *Chapter 2* introduced `runqemu` for the basic task of booting a first image. This chapter develops the same helper into a testing tool, so the boot workflow you already know carries forward. Learning to use `runqemu` effectively helps developers optimize project workflows by harnessing its wide array of features. `runqemu` is not on your `PATH` by default. Source the `init-build-env` script from your BitBake Setup's build directory first, as *Chapter 2* describes. Every new shell needs this step, and every `runqemu` command in this chapter assumes you did it. Print the full set of options it accepts:

```
$ runqemu --help
```

The helper reports the environment variables it reads and the simplified options it understands, as shown below.

```
$ 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
    nonetwork - disable network connectivity

(...)

Examples:
  runqemu
  runqemu qemuarm

(...)
```

Figure 17.1: The usage of runqemu

Several important use cases highlight QEMU's capabilities:

- **The Linux kernel image selection**: Test various Linux kernel images, enabling comprehensive evaluation of Linux kernel functionalities
- **Root filesystem selection**: Boot with various root filesystem configurations, enabling comprehensive testing across multiple setups
- **Custom Linux kernel boot arguments**: Pass tailored boot parameters to the Linux kernel, facilitating precise control and configuration during emulation
- **Graphical environment support**: Offers robust options for OpenGL and OpenGL ES, enabling testing and validation of graphical applications in a simulated environment
- **Custom QEMU parameters**: Include specific QEMU command-line parameters, enabling highly customized emulation setups tailored to project needs
- **Serial console testing**: Enables efficient testing of images through a serial console interface, simplifying debugging and development workflows
- **Audio stack validation**: Offers comprehensive testing capabilities for audio configurations and features within diverse setups
- **Init system testing**: Evaluate various initialization systems, such as `systemd`, confirming compatibility and functionality within the target environment

The following sections use the `qemux86-64` machine as a reference to illustrate practical use cases, highlighting key functionalities and advantages of `runqemu` in real-world scenarios.

Using runqemu to test graphical applications

When validating an application without considering GPU performance of the embedded device, QEMU serves as a reliable tool. This is particularly useful for testing Qt or GTK+ applications. First, build the `core-image-weston` image, then launch it under QEMU with graphics support. Adding the `snapshot` argument (introduced in *Chapter 3*) keeps the run disposable, so repeated graphical tests never mutate the deployed root filesystem image:

```
$ runqemu gl sdl core-image-weston snapshot
```

`runqemu` resolves the build artifacts and boots the image, as the following log output shows.

```
$ runqemu gl sdl core-image-weston snapshot

runqemu - INFO - Running bitbake -e  core-image-weston...
runqemu - INFO - Decompressing .../build/tmp/deploy/images/qemux86-64/core-image-weston-qemux86-
64.rootfs.ext4.zst to .../build/tmp/deploy/images/qemux86-64/core-image-weston-qemux86-64.rootfs-
20260717155327.ext4
.../build/tmp/deploy/images/qemux86-64/core-image-weston-qemux86-64.rootfs-20260717155327.ext4.zst: 779717632
bytes
runqemu - INFO - Continuing with the following parameters:

KERNEL: [.../build/tmp/deploy/images/qemux86-64/bzImage]

MACHINE: [qemux86-64]

FSTYPE: [ext4]

ROOTFS: [.../build/tmp/deploy/images/qemux86-64/core-image-weston-qemux86-64.rootfs-20260717155327.ext4]

SNAPSHOT: [Enabled. Changes on rootfs won't be kept after QEMU shutdown.]

CONFFILE: [.../build/tmp/deploy/images/qemux86-64/core-image-weston-qemux86-64.rootfs.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
net.ifnames=0
runqemu - INFO - Running .../build/tmp/work/x86_64-linux/qemu-helper-native/1.0/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=.../build/tmp/deploy/images/qemux86-64/core-image-weston-qemux86-64.rootfs-
20260717155327.ext4,if=virtio,format=raw -usb -device usb-tablet -usb -device usb-kbd   -cpu Skylake-Client -
machine q35,i8042=off -smp 4 -m 512 -snapshot -serial mon:vc -serial null -device virtio-vga-gl  -display
sdl,gl=on,show-cursor=on  -kernel .../build/tmp/deploy/images/qemux86-64/bzImage -append 'root=/dev/vda rw
ip=192.168.7.2::192.168.7.1:255.255.255.0::eth0:off:8.8.8.8 net.ifnames=0 oprofile.timer=1 tsc=reliable
no_timer_check rcupdate.rcu_expedited=1 swiotlb=0 '

runqemu - INFO - Host uptime: 944814.13
```

Figure 17.2: Log output after running QEMU with graphic support

Next, the execution of `core-image-weston` inside QEMU appears:

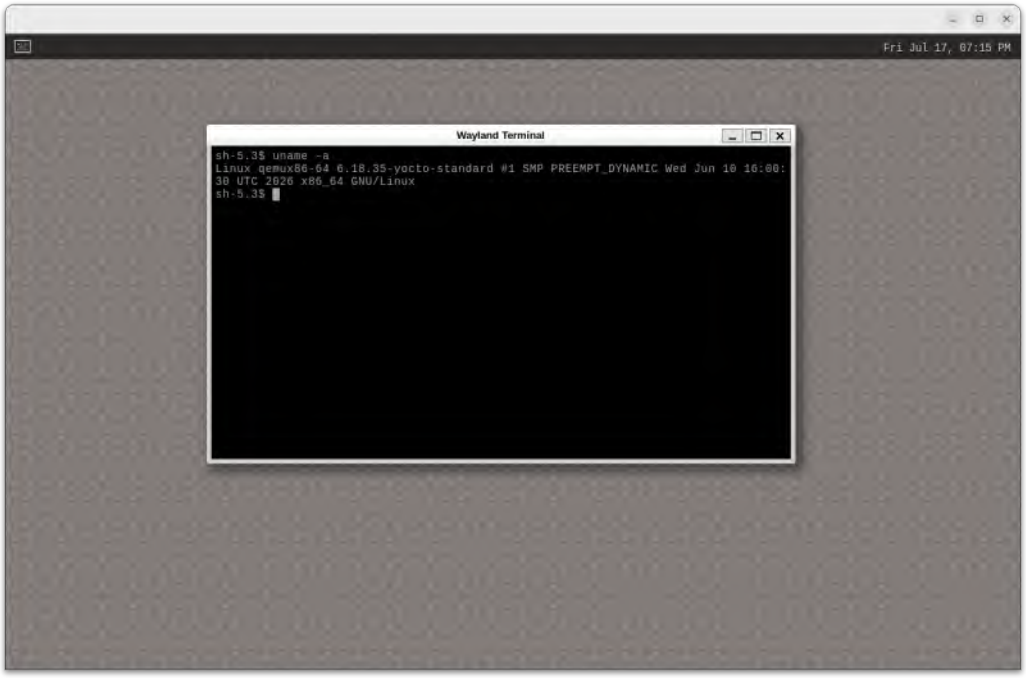


Figure 17.3: QEMU running core-image-weston

The preceding screenshot shows the QEMU interface running `core-image-weston`, featuring the Wayland Terminal displaying detailed information about the active Linux kernel. This illustrates the system's operational capabilities and offers a view of the Linux kernel's current state within the emulated environment.

Using runqemu to validate memory constraints

When validating application memory usage, QEMU provides a reliable testing environment.

First, build the `core-image-full-cmdline` image, then run QEMU with a constrained memory size. Adding `snapshot` again keeps the run disposable:

```
$ runqemu qemuparams="-m 128" core-image-full-cmdline snapshot
```

The `qemuparams` option caps the guest RAM at 128 MB, as the following log output shows.

```
$ runqemu qemuparams="-m 128" core-image-full-cmdline snapshot

runqemu - INFO - Running bitbake -e  core-image-full-cmdline...
runqemu - INFO - Decompressing .../build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-
64.rootfs.ext4.zst to .../build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs-
20260715173152.ext4
.../build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs-20260715173152.ext4.zst:
376731648 bytes
runqemu - INFO - Continuing with the following parameters:

KERNEL: [.../build/tmp/deploy/images/qemux86-64/bzImage]

MACHINE: [qemux86-64]

FSTYPE: [ext4]

ROOTFS: [.../build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs-20260715173152.ext4]

SNAPSHOT: [Enabled. Changes on rootfs won't be kept after QEMU shutdown.]

CONFFILE: [.../build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs.qemuboot.conf]

runqemu - INFO - Setting up tap interface under sudo
runqemu - INFO - Network configuration: ip=192.168.7.2::192.168.7.1:255.255.255.0::eth0:off:8.8.8.8
net.ifnames=0
runqemu - INFO - Running .../build/tmp/work/x86_64-linux/qemu-helper-native/1.0/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=.../build/tmp/deploy/images/qemux86-64/core-image-full-cmdline-qemux86-64.rootfs-
20260715173152.ext4,if=virtio,format=raw -usb -device usb-tablet -usb -device usb-kbd   -cpu Skylake-Client -
machine q35,i8042=off -smp 4 -m 128  -m 128 -snapshot -serial mon:vc -serial null -device virtio-vga  -display
sdl,show-cursor=on  -kernel .../build/tmp/deploy/images/qemux86-64/bzImage -append 'root=/dev/vda rw
ip=192.168.7.2::192.168.7.1:255.255.255.0::eth0:off:8.8.8.8 net.ifnames=0 oprofile.timer=1 tsc=reliable
no_timer_check rcupdate.rcu_expedited=1 swiotlb=0 '

runqemu - INFO - Host uptime: 1210465.52
```

Figure 17.4: The log after running QEMU with 128 MB of RAM

The following screenshot shows the amount of memory in use inside QEMU:

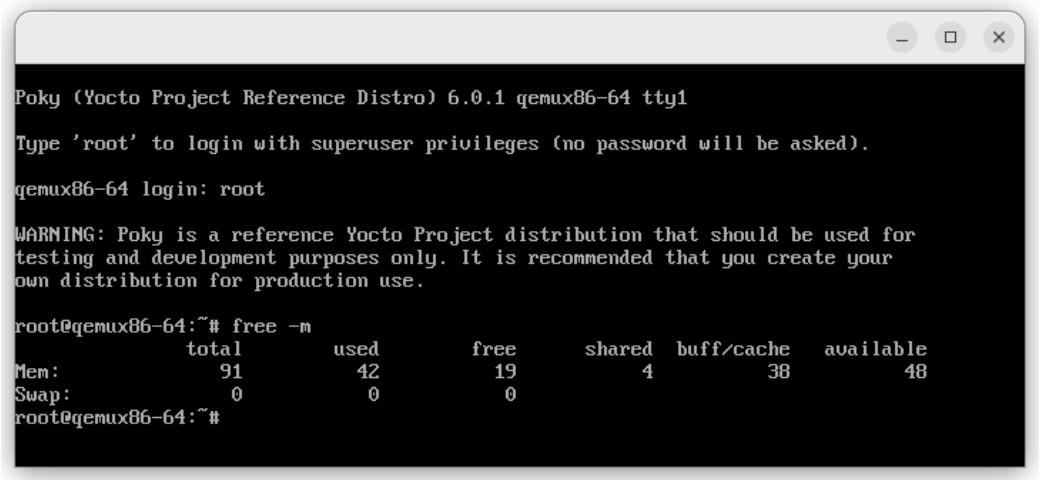


Figure 17.5: QEMU running core-image-full-cmdline with 128 MB of RAM

Changing the command line used to run QEMU helps test different memory sizes via emulation.

Using runqemu to select a Linux kernel image and root filesystem

Every example so far let `runqemu` resolve `KERNEL` and `ROOTFS` automatically by parsing `bitbake -e` for the image recipe name you pass. Point it at specific files instead when you need to test a Linux kernel or root filesystem that a plain image name cannot express, such as a hand-copied `bzImage` from a Linux kernel rebuild or a root filesystem built for a different image recipe:

```
$ runqemu /home/user/bitbake-builds/poky-wrynose/build/tmp/deploy/images/
qemux86-64/bzImage /home/user/bitbake-builds/poky-wrynose/build/tmp/deploy/images/
qemux86-64/core-image-full-cmdline-qemux86-64.rootfs.ext4
```

The Linux kernel image and root filesystem paths accept any order. `runqemu` autodetects which is which by matching the filename, for example `bzImage` for the Linux kernel and `.rootfs.` for the root filesystem. This decouples the Linux kernel under test from the root filesystem under test, so you can hold one constant while iterating on the other. Set `KERNEL` and `ROOTFS` as environment variables instead, as shown in the `runqemu --help` output earlier, when a filename does not match either pattern. Do not add `qemux86-64` as a separate `MACHINE` argument on this or any other `runqemu` command in this chapter: *Chapter 2* already fixed

`MACHINE` through the `machine/qemux86-64` BitBake Configuration Fragment, and a redundant assignment on the command line conflicts with that fragment.

The default `IMAGE_FSTYPES` setting leaves only the compressed `core-image-full-cmdline-qemux86-64.rootfs.ext4.zst` artifact in `build/tmp/deploy/images/qemux86-64`, and that plain filename is a symbolic link to the timestamped file BitBake actually wrote. `runqemu` decompresses a `.zst` root filesystem automatically only when the `snapshot` option is also given, so decompress it yourself first. `zstd` refuses to follow a symbolic link unless you pass `-f`, so run `zstd -d -f core-image-full-cmdline-qemux86-64.rootfs.ext4.zst` to produce the plain `.rootfs.ext4` file the preceding command expects. The remaining examples in this chapter that omit `snapshot` point directly at this decompressed file instead of the plain `core-image-full-cmdline` image name, since resolving `ROOTFS` from the image name would hit the same still-compressed artifact.

Using runqemu to pass custom boot arguments

Append Linux kernel command-line arguments with the `bootparams` option when you need to change boot-time behavior without editing `build/conf/local.conf`. Disabling the boot splash so Linux kernel and init messages stay visible on the console is a common case:

```
$ runqemu /home/user/bitbake-builds/poky-wrynose/build/tmp/deploy/images/
qemux86-64/core-image-full-cmdline-qemux86-64.rootfs.ext4 \
          bootparams="psplash=false"
```

The `bootparams` value is appended to the Linux kernel command line `runqemu` already assembled from `MACHINE` and `FSTYPE`, so anything you could pass on a real device's boot line works here too.

Using runqemu with a serial console

`do_testimage` already relies on a serial console to log in and run its checks without a graphical window. The `nographic` option gives you that same headless behavior interactively, keeping the console in the terminal you launched `runqemu` from instead of opening a QEMU display, as shown below:

```
$ runqemu /home/user/bitbake-builds/poky-wrynose/build/tmp/deploy/images/
qemux86-64/core-image-full-cmdline-qemux86-64.rootfs.ext4 \
          nographic
```

This is the mode Autobuilder-style automated build hosts use, since it needs neither a display server nor a windowing system on the build host. It is the same constraint the tap device setup later in this chapter exists to satisfy for `do_testimage`.

Using runqemu to validate audio stacks

Enable the emulated audio device with the `audio` option to exercise audio-stack packages, such as ALSA utilities or PulseAudio, inside the guest. Audio emulation is only available on the `qemux86` and `qemux86-64` machines, so the `qemux86-64` reference machine used throughout this chapter already qualifies:

```
$ runqemu /home/user/bitbake-builds/poky-wrynose/build/tmp/deploy/images/
qemux86-64/core-image-full-cmdline-qemux86-64.rootfs.ext4 \
          audio
```

Combine `audio` with `qemuparams` or `bootparams` from the earlier examples to reproduce the exact conditions an audio regression needs, such as a constrained memory size alongside a custom ALSA configuration passed on the Linux kernel command line.

Using runqemu to help with image regression tests

The Yocto Project offers an automated testing framework, an essential component of its QA process. The integration and validation testing support is facilitated through the `testimage` class, which allows images to be executed and tested within the target environment.

Tip

The testing framework can validate existing recipes and images, and it can also be extended with custom tests to verify new applications and integrations. Detailed information about the testing framework's capabilities can be found in the *Types of Testing Overview* section at `https://docs.yoctoproject.org/wrynose/test-manual/intro.html#yocto-project-tests-types-of-testing-overview` of the *Yocto Project Test Environment Manual*.

To enable `testimage` support, add the line `IMAGE_CLASSES += "testimage"` to the `build/conf/local.conf` file. `IMAGE_CLASSES` scopes the class to image recipes, which is the recommended form because the `testimage` machinery is only meaningful for images. Using `INHERIT += "testimage"` instead applies the class globally to every recipe, which is wasteful and not what the test manual recommends.

`do_testimage` logs in as `root` over the emulated serial console to run its checks, so it depends on passwordless root login. *Chapter 2* already enables the `core/yocto/root-login-with-empty-password` fragment for this workspace. If you disabled it while working through *Chapter 14*, re-enable it with `bitbake-config-build enable-fragment core/yocto/root-login-with-empty-password` before continuing.

The Yocto Project's `testimage` class prints the last lines of the QEMU boot log when `do_testimage` fails. The diagnostic output streams directly to the BitBake task log, so most boot or device-bring-up failures can be triaged without re-running the test under a serial console.

Note

Warning!

During image testing, the `sudo` command is used for network setup, which might cause errors depending on your host configuration. Refer to the *Performing Automated Runtime Testing* section at `https://docs.yoctoproject.org/wrynose/test-manual/runtime-testing.html#performing-automated-runtime-testing` of the *Yocto Project Test Environment Manual* for guidance on resolving these issues.

`do_testimage` runs non-interactively, so it cannot answer the `sudo` password prompt `runqemu` normally uses to create a tap interface. The *Enabling Runtime Tests on QEMU* section at `https://docs.yoctoproject.org/wrynose/test-manual/runtime-testing.html#enabling-runtime-tests-on-qemu` of the *Yocto Project Test Environment Manual* lists `runqemu-gen-tapdevs` as the option typically chosen for Autobuilder-type environments. Ensure `iptables` is installed on the host and that `qemu-helper-native` has been built (`bitbake qemu-helper-native`, already satisfied if you ran the earlier examples in this chapter), then pre-create a pool of tap devices once per host with the script, passing your group ID and the number of devices to create. Call it with an absolute path, since `sudo` does not inherit the build environment's `PATH`:

```
$ sudo /home/user/bitbake-builds/poky-wrynose/layers/openembedded-core/scripts/
runqemu-gen-tapdevs $(id -g) 4
```

The script creates the requested devices and, on a NetworkManager-managed host, prints the configuration needed to keep it from taking them over, as the following figure shows.

```
$ sudo /home/user/bitbake-builds/poky-wrynose/layers/openembedded-core/scripts/runqemu-gen-tapdevs $(id -g) 4
Creating 4 tap devices for GID: 1000...
Creating tap0
Creating tap1
Creating tap2
Creating tap3
Note: For systems running NetworkManager, it's recommended
Note: that the tap devices be set as unmanaged in the
Note: NetworkManager.conf file. Add the following lines to
Note: /etc/NetworkManager/NetworkManager.conf
[keyfile]
unmanaged-devices=interface-name:tap*
```

Figure 17.6: Pre-creating tap devices with runqemu-gen-tapdevs

The devices persist until the host reboots, so this step only needs to run once. If NetworkManager manages the host's network interfaces, follow the printed advice and mark `tap*` devices unmanaged in `/etc/NetworkManager/NetworkManager.conf` to keep it from interfering with them.

Networking behavior is also worth noting.

Note

`runqemu` limits `slirp`-based port forwarding to localhost only. This may affect network access patterns in your test environment when other hosts need to reach forwarded ports. When a test workflow legitimately needs the QEMU image reachable from another host, set `QB_SLIRP_OPT` in `build/conf/local.conf` (for example, `QB_SLIRP_OPT = "-netdev user,id=net0,hostfwd=tcp::2222-:22"`) to relax the default localhost binding.

After building the `core-image-weston` image, run the `testimage` task:

```
$ bitbake -c testimage core-image-weston
```

The task boots the image under QEMU and runs the regression suite, as the following figure shows.

```
$ bitbake -c testimage core-image-weston

(...)

NOTE: Executing Tasks
NOTE: Running task 1488 of 1488 (.../layers/openembedded-core/meta/recipes-graphics/images/core-image-
weston.bb:do_testimage)
NOTE: recipe core-image-weston-1.0-r0: task do_testimage: Started
QMP Available for connection at .../build/tmp/.t9inwrmd
QMP connected to QEMU at 07/17/26 15:46:36 and took 1.04 seconds
QMP released QEMU at 07/17/26 15:46:36 and took 0.07 seconds from connect
Test requires apt to be installed
Test requires autoconf to be installed
Test requires gtk+3 to be installed

(...)

Output from runqemu:
runqemu - INFO - Received signal: 15
runqemu - INFO - Cleaning up
runqemu - INFO - Host uptime: 949862.45

RESULTS:
RESULTS - date.DateTest.test_date: PASSED (2.22s)
RESULTS - df.DfTest.test_df: PASSED (0.55s)
RESULTS - dnf.DnfBasicTest.test_dnf_help: PASSED (1.00s)

(...)

SUMMARY:
core-image-weston () - Ran 75 tests in 79.750s

core-image-weston - OK - All required tests passed (successes=37, skipped=38, failures=0, errors=0)
NOTE: recipe core-image-weston-1.0-r0: task do_testimage: Succeeded
NOTE: Tasks Summary: Attempted 1488 tasks of which 1487 didn't need to be rerun and all succeeded.
```

Figure 17.7: Results of RUNNING the testimage task for core-image-weston

The preceding log displays the regression test results, providing insights into the functionality and stability of the tested image.

Using runqemu to run package tests with ptest

The `testimage` class checks an assembled image from the outside. Package tests, known as ptests, work from the inside. A recipe ships the test suite its upstream project already provides, and that suite runs on the target against the binaries the build produced. Use ptests to confirm that a library or tool still behaves correctly after cross-compilation, which an image-level test cannot tell you.

A recipe joins this scheme by inheriting the `ptest` class and shipping a `run-ptest` script. The `zlib` recipe in OpenEmbedded Core is a typical example. The build installs the resulting files under `/usr/lib/<package>/ptest` in the image.

Yocto Project 6.0 (Wrynose) lists `ptest` in `DISTRO_FEATURES_DEFAULTS`, as *Chapter 12* describes, so the feature is already active in this workspace. Add the line `EXTRA_IMAGE_FEATURES += "ptest-pkgs"` to the `build/conf/local.conf` file. That installs the ptest package of every ptest-enabled recipe, together with the `ptest-runner` program that runs them.

The suites run from a console session on the target, so the image also needs to accept a `root` login. A freshly initialized configuration does not, and the login attempt times out after 60 seconds. The `core/yocto/root-login-with-empty-password` fragment that *Chapter 2* enabled for this workspace covers it, the same fragment the `testimage` section above depends on. If you disabled it while working through *Chapter 14*, re-enable it with `bitbake-config-build enable-fragment core/yocto/root-login-with-empty-password` before rebuilding.

Note

Warning!

A `ptest-pkgs` image builds and installs the test suites of every ptest-enabled recipe. Expect a much longer build and a much larger image than a plain `core-image-full-cmdline`. Reserve extra disk space before you start.

Rebuild the image, then boot it with the `nographic` option shown earlier and log in as `root`. Ask `ptest-runner` which suites the image carries:

```
root@qemux86-64:~# ptest-runner -l
```

Each line names a suite and the `run-ptest` script that drives it, as the following figure shows. This image carries 45 of them, one for every ptest-enabled recipe the build pulled in:

```
root@qemux86-64:~# ptest-runner -l
Available ptests:
acl    /usr/lib/acl/ptest/run-ptest
attr    /usr/lib/attr/ptest/run-ptest
bash    /usr/lib/bash/ptest/run-ptest
bc    /usr/lib/bc/ptest/run-ptest
busybox    /usr/lib/busybox/ptest/run-ptest
bzip2    /usr/lib/bzip2/ptest/run-ptest
coreutils    /usr/lib/coreutils/ptest/run-ptest
cpio    /usr/lib/cpio/ptest/run-ptest
curl    /usr/lib/curl/ptest/run-ptest
diffutils    /usr/lib/diffutils/ptest/run-ptest
e2fsprogs    /usr/lib/e2fsprogs/ptest/run-ptest
elfutils    /usr/lib/elfutils/ptest/run-ptest

(...)

openssh    /usr/lib/openssh/ptest/run-ptest
openssl    /usr/lib/openssl/ptest/run-ptest
perl    /usr/lib/perl/ptest/run-ptest
popt    /usr/lib/popt/ptest/run-ptest
procps    /usr/lib/procps/ptest/run-ptest
python3    /usr/lib/python3/ptest/run-ptest
sed    /usr/lib/sed/ptest/run-ptest
tar    /usr/lib/tar/ptest/run-ptest
util-linux    /usr/lib/util-linux/ptest/run-ptest
xz    /usr/lib/xz/ptest/run-ptest
zlib    /usr/lib/zlib/ptest/run-ptest
```

Figure 17.8: Listing the package tests an image carries

Run a single suite rather than the whole set while you iterate, because a full run of every installed suite can take hours. Name the suite as the only argument:

```
root@qemux86-64:~# ptest-runner zlib
```

The suite prints its own output, and `ptest-runner` frames it with `BEGIN:` and `END:` markers, as illustrated here:

```
root@qemux86-64:~# ptest-runner zlib
START: ptest-runner
2026-08-19T20:02
BEGIN: /usr/lib/zlib/ptest
zlib version 1.3.2 = 0x1320, compile flags = 0xa9
uncompress(): hello, hello!
gzread(): hello, hello!
gzgets() after gzseek:  hello!
inflate(): hello, hello!
large_inflate(): OK
after inflateSync(): hello, hello!
inflate with dictionary: hello, hello!
PASS: zlib
DURATION: 0
END: /usr/lib/zlib/ptest
2026-08-19T20:02
STOP: ptest-runner
TOTAL: 1 FAIL: 0
```

Figure 17.9: Running a single package test suite

The result line uses `PASS:`, `FAIL:`, or `SKIP:` followed by the name, and `DURATION:` records the runtime in seconds. The trailing `TOTAL:` and `FAIL:` counts summarize the run, and `ptest-runner` exits with 1 when any test fails. That exit code makes it usable as the last step of an automated job. Pass `-t` to change the per-test timeout from its 300 second default. Use `-e` to exclude a suite, and `-x` to write the results as XML for an automated build system to collect.

Summary

This chapter explored the use of QEMU and its ability to accelerate the development cycle by using emulation whenever feasible and identifying scenarios where physical hardware remains necessary. You worked through the `runqemu` options that make the emulator a daily tool: selecting a Linux kernel image and root filesystem, passing custom boot arguments, driving the guest from a serial console, and validating graphical and audio stacks without a board on the desk. The final two sections moved from manual inspection to automated verification, running image regression tests with the `testimage` class and package tests with `ptest-runner`, so a change can be checked from the outside and from the inside before it ever reaches hardware.

The next chapter presents a curated list of recommendations that have been refined and proven effective by developers over the years for building successful Yocto Project-based products.

18

Best Practices

This chapter distills hard-won lessons from real embedded Linux projects built with the Yocto Project. You will learn how to organize layers and metadata so that builds stay reproducible, how to structure a custom distribution to replace ad hoc configuration, and how to apply project-management habits that keep a codebase navigable over its full service life.

The goal is to give you a compact set of concrete, actionable principles rather than abstract advice. Each guideline addresses a recurring failure mode: the kind that surfaces not during the initial build sprint but months later, when a colleague inherits the project or a release deadline demands a fast turnaround on an unfamiliar codebase.

That long-term cost is the real motivation here. A product that compiles cleanly on its creator's workstation can still accumulate invisible debt: undocumented configuration scattered across `build/conf/local.conf`, layers with unpinned revisions, and upstream contributions deferred indefinitely. When discipline is skipped early, the maintenance burden compounds with every release cycle until routine upgrades become multi-week recoveries.

The chapter is divided into two independent parts: one focusing on Yocto Project-specific guidelines and the other on general project aspects. This structure allows you to explore each section in any order.

We will cover the following main topics:

- Managing layers
- Tidying up `bitbake-setup` configuration files
- Avoiding layer proliferation
- Preparing metadata for new Yocto Project releases
- Creating a custom distribution
- Avoiding reusing existing images for your product

- Using the Standard SDK for application development
- Managing Linux kernel and bootloader patches
- Avoiding `AUTOREV` as `SRCREV`
- Maintaining the Shared State Cache
- Enforcing the guidelines with version control and continuous integration
- Creating a software bill of materials
- Monitoring project license constraints
- Addressing security vulnerabilities
- Managing long-term maintenance costs
- Tackling project risk points and constraints early

Guidelines to follow for the Yocto Project

This section provides essential guidelines for managing Yocto Project metadata and organizing projects effectively, simplifying both short- and long-term maintenance.

Manage layers

As our journey in product development progresses, we will inevitably rely on multiple repositories to address the challenges we encounter. Managing these repositories effectively can be complex, requiring us to:

- Accurately reproduce previous builds for consistent results and future reference
- Enhance team collaboration by enabling multiple members to work seamlessly within the same codebase
- Use robust automated build systems to validate changes and maintain consistency and quality
- Guard against subtle and unintended changes in the layers to maintain stability

While these objectives may seem daunting, a variety of tools and strategies have been developed to address and simplify these challenges.

The `image-buildinfo` class (`https://docs.yoctoproject.org/wrynose/ref-manual/classes.html#image-buildinfo`) writes a plain text record of build information and layer revisions to the target filesystem at `${sysconfdir}/buildinfo` by default, so devices in the field can report the exact metadata that produced them. The same class writes a matching record into the SDK installer via `SDK_BUILDINFO_FILE`, which defaults to `/buildinfo`, so a delivered SDK carries the same provenance trail as the device image.

The following tools address declarative environment management, starting with the Yocto Project's own first-party option. The rest predate it and remain useful when an organization has standardized on them or needs capabilities `bitbake-setup` does not yet cover:

- `bitbake-setup`, documented in *Chapter 14*, is the first-party answer in Yocto Project 6.0 (Wrynose). It pins layer revisions, captures distribution and machine fragments in a BitBake Configuration Template, and reproduces the environment on every workstation that consumes the template. Reach for it before introducing additional tooling.
- Google developed the `repo` tool for Android development (`https://source.android.com/docs/setup/download`), and it has since been adopted for use in various other projects. A key aspect of `repo` is its need for additional tooling to integrate seamlessly with Yocto Project-based workflows, automating the build directory and environment configuration. Refer to the O.S. Systems Embedded Linux project (`https://github.com/OSSystemsEmbeddedLinux/ossystems-embedded-linux-platform`) as a source of inspiration for using `repo` in your own projects.
- Siemens developed `kas` (`https://github.com/siemens/kas`) to provide a streamlined mechanism for downloading sources, automating the build directory setup, configuring the environment, and other related tasks.
- Garmin developed **Whisk** (`https://github.com/garmin/whisk`) to manage complex product configurations using OpenEmbedded and the Yocto Project. Its key features include a unified source tree, multiple configuration dimensions, support for multiple product builds, and isolated layer configurations.
- Agilent developed **Yocto Buddy** (`https://github.com/Agilent/yb`), designed to simplify setup and maintain synchronization of Yocto Project-based environments. Drawing inspiration from previously mentioned tools, Yocto Buddy remains in its early stages of development.

This is a subset of existing tools and should not be considered a comprehensive list. It is recommended to experiment with these tools to determine the best fit, as the choice will depend on your project's specific use case and the expertise of your team.

Tidy up bitbake-setup configuration files

Deleting a BitBake Setup directory takes its `build/conf/local.conf` and `build/conf/bblayers.conf` with it, but three other files `bitbake-setup` writes do not disappear so easily. The global settings file at `~/.config/bitbake-setup/settings.conf` applies to every Top Directory on the machine. The local `settings.conf` and `site.conf` sit at the root of a single

Top Directory and apply to every BitBake Setup you initialize beneath it, now or later. None of the three go away when you remove one BitBake Setup, and the global file survives even after you delete the whole Top Directory.

That persistence helps until it does not:

- A `dl-dir` path or a private BitBake Registry configured months ago for a project that has since been archived keeps applying to every BitBake Setup you initialize afterward on the same machine, for a product that has nothing to do with the one that set it.
- `site.conf` is not managed through the `settings` subcommand at all. It is a plain BitBake configuration file that `bitbake-setup init` writes once at the Top Directory, populated with Shared State Cache and hash-equivalence paths when `common-sstate` is enabled (the default), and from then on it is yours to edit or delete directly.
- Before starting a project on a workstation that has already hosted others, run `bitbake-setup settings list` to see what a fresh BitBake Setup is about to inherit, as shown below:



Figure 18.1: Auditing settings with bitbake-setup settings list

This machine still carries a private BitBake Registry and a shared `dl-dir` left over from an earlier product, and both apply to the next BitBake Setup initialized here unless removed first. Clear entries that no longer apply with `bitbake-setup settings unset` (add `--global` for the machine-wide file).

How much of this to keep is a choice, not a rule. Consolidating `dl-dir`, `common-sstate`, and a shared BitBake Registry into the global `settings.conf` pays off on a workstation dedicated to one product line or team, where every new BitBake Setup should reuse the same download cache and Shared State Cache without repeating the configuration. Keeping the global file empty and pushing customization into a `settings.conf` and `site.conf` local to that Top Directory instead pays off when a machine hosts unrelated products or clients, or when you want a reference environment that stays reproducible regardless of whatever else lives on the

same disk. Neither approach is wrong. Pick the one that matches whether the machine's role is convenience across many projects or isolation between them. The *BitBake User Manual* at `https://docs.yoctoproject.org/bitbake/2.18/` documents the full settings hierarchy and every value these files can hold.

Avoid creating too many layers

A key advantage of the Yocto Project is its ability to use and create multiple layers. This capability empowers developers to:

- Reuse BSP layers provided by semiconductor vendors, streamlining integration and significantly reducing development time
- Minimize redundant efforts by enabling the sharing of reusable blocks, which facilitates the integration of new or specialized applications, programming languages, and other components

However, creating an excessive number of layers can become counterproductive when managing a project or a portfolio of products.

For instance, creating BSP-only layers is particularly valuable in the following scenarios:

- The board itself is the primary product, as commonly observed with **System-on-Module** (**SoM**) vendors
- When external access to the layer is essential, but restrictions on non-BSP sources are necessary to maintain security or control

Using a single layer for a product, or even an entire company, offers several notable advantages:

- Simplifies the development of reusable components, such as `packagegroup` packages for shared development tools or network utilities, across multiple products
- Mitigates the risk of unintended side effects caused by changes targeted at a specific product or board
- Enhances the reuse of bug fixes across different products and allows consistent application of BSP low-level components like the Linux kernel or bootloader
- Encourages standardization across multiple products, lowering the learning curve for new team members and improving overall efficiency

While the choice between using one or multiple layers depends on specific project needs, it is generally advisable to start with a simple structure and consider splitting layers only if the need arises later.

Prepare the product metadata for new Yocto Project releases

As our product evolves, so does the complexity of our metadata, emphasizing the need for proper organization. Common scenarios encountered during product development include:

- Backporting a new recipe version to address a bug fix or implement a feature
- Adding a missing package configuration or bug fix not yet available in the Yocto Project recipe

To manage this effectively, we use two dedicated recipe directories:

- `recipes-backport`: Contains backported recipes from newer Yocto Project releases
- `recipes-staging`: Hosts new recipes or `.bbappend` files that introduce missing package configurations or bug fixes

We actively contribute new recipes or bug fixes from `recipes-staging` to the respective upstream project (for example, OpenEmbedded Core). Once a patch is accepted, we transfer the changes from `recipes-staging` to the `recipes-backport` directory. This workflow helps us track pending upstream tasks and simplifies the process of upgrading our meta layer to a new Yocto Project release.

Additionally, the `recipes-backport` directory can be swiftly addressed and removed when necessary, maintaining a clean and maintainable structure.

Create your custom distribution

When using the Yocto Project, configurations are often added to `build/conf/local.conf`. However, this approach is problematic as it is not under source control, leading to potential inconsistencies among developers. Adopting a custom distribution provides numerous advantages, including:

- Consistent configurations across multiple developers
- A clear overview of the `DISTRO_FEATURES` being applied compared to the base distribution (for example, `poky`)
- Centralized recipe configurations for the product, reducing the need for multiple `.bbappend` files (for example, `PACKAGECONFIG:append:pn-<recipe-name> = "myfeature"`)

Beyond these technical benefits, a custom distribution enables proper branding of SDKs and other artifacts generated by the Yocto Project.

The process for creating a custom distribution was explored in the *Using a custom distribution* section of *Chapter 12*.

Avoid reusing existing images for your product

Images are where all components come together. When developing a product, it is crucial to minimize the number of packages installed in our images for several reasons:

- Reducing root filesystem size: Smaller images consume less storage and are easier to manage
- Shortening build time: Fewer packages mean faster builds, improving productivity
- Simplifying license management: With fewer packages, the number of licenses to track and comply with is reduced
- Enhancing security: A leaner image decreases the attack surface for potential vulnerabilities

A practical starting point is copying the `core-image-base.bb` file into our custom layer as `myproduct-image.bb` and then extending it with the specific features required for the product. Additionally, we create a separate image, `myproduct-image-dev.bb`, for development purposes. This development image should require `myproduct-image.bb` and include only the artifacts necessary for development, thereby avoiding code duplication.

This approach results in two tailored images (one for production and one for development) that share the same core features and packages, maintaining consistency and maintainability.

Tip

Products that boot from a read-only root filesystem can still preserve OpenSSH host keys across reboots by setting `OPENSSH_HOST_KEY_DIR_READONLY_CONFIG` to a writable directory (the default is `/var/run/ssh`). The companion `OPENSSH_HOST_KEY_DIR` variable selects the directory used on writable root filesystems and defaults to `/etc/ssh`. Both variables let the device generate host keys on first boot and reuse them afterward without writing to the read-only root filesystem.

Standard SDK is commonly undervalued

Application development is inherently interactive, often requiring continuous iteration until the desired functionality is achieved. This workflow, however, is not ideally suited for the Yocto Project due to the following limitations:

- Every recipe build discards previously generated build objects, leading to inefficiencies

- Deploying the application or image takes significantly longer compared to other methods
- Limited integration with standard IDE environments hampers developer productivity

While tools like `devtool` provide alternatives by reusing build objects and streamlining deployment, as explored in the *Building a recipe using devtool* section of *Chapter 9*, the overall development experience remains cumbersome.

Using the standard SDK for developing applications and other components, such as the Linux kernel and bootloader, is a more efficient approach. This method allows for faster development cycles, enabling teams to either defer or parallelize Yocto Project integration tasks.

Avoid too many patches for the Linux kernel and bootloader modifications

The need for patches in the Linux kernel and bootloader is a fundamental aspect of embedded Linux development, as hardware rarely functions without some level of modification. The extent of these changes depends heavily on your hardware design, for instance:

- **Single-Board Computers (SBCs)**: Minimal modifications are typically required
- **System-on-Modules (SoMs) with custom baseboards**: The extent of changes can vary, depending on the number of alterations from the vendor's baseboard hardware design
- **Custom hardware designs**: These usually necessitate the development of a custom BSP and involve a significant number of modifications

These scenarios are not rigid. For example, you may start a project using an SBC but later discover that the vendor provides an inadequate reference BSP, significantly increasing the work required for BSP customization.

For minor changes, it is more efficient to handle them as patch files added to the component recipe, using the `.bbappend` and `FILESEXTRAPATHS` mechanics covered in *Chapter 13*. However, as the maintenance effort grows, creating a separate fork of the component becomes justified. A repository fork offers several advantages:

- Maintains a detailed history of changes
- Supports different branches or tags for development and production workflows
- Enables easier merging with contributions from other providers
- Simplifies recipes by removing the need to carry individual patches

In conclusion, selecting the strategy that aligns with your project's needs is critical. While the approach may evolve over time, choosing the right one initially helps minimize the overall effort required to maintain the hardware effectively.

Avoid using AUTOREV as SRCREV

The use of `AUTOREV` as `SRCREV` is commonly applied during product development when interactive code changes are required within the Yocto Project. *Chapter 5* introduced how `SRCREV` pins the commit that `do_fetch` and `do_unpack` consume, and the guidance here builds on that mechanism. However, this approach has several significant drawbacks:

- Reproducing previous builds becomes challenging because each rebuild may use a different revision for the recipe.
- The `AUTOREV` value is only applied when BitBake invalidates the cache of a specific recipe.
- `AUTOREV` needs network access, because BitBake queries the upstream repository to resolve the branch tip. An air-gapped or offline build cannot resolve it and fails, as *Chapter 5* describes.

Cache invalidation occurs when the recipe is modified or when changes are made to files that trigger a BitBake cache rebuild, such as `.conf` files.

Due to these limitations, `AUTOREV` is considered fragile. More reliable alternatives are available for handling interactive code changes:

- `devtool`: This tool enables direct code modifications within the workspace and forces the recipe to use the updated source
- `externalsrc.bbclass` (`https://docs.yoctoproject.org/wrynose/ref-manual/classes.html#externalsrc`): This class configures a recipe to use a specific directory as the source for the build, providing a consistent and flexible approach

An override limits the risk when a build must track the top of a branch on purpose. Assign `AUTOREV` to a development override, as in `SRCREV:devbuild = "${AUTOREV}"`, and leave the default `SRCREV` pinned to a commit. A nightly or integration build selects the override and follows the branch, while a product build keeps the pinned revision and stays reproducible. Keep that override out of the configuration that produces a release.

Maintain the Shared State Cache

The Shared State Cache speeds up rebuilds by restoring completed task output instead of recomputing it, and that benefit is worth protecting. The cost is easy to defer: the cache under `SSTATE_DIR` grows with every build and never reclaims space on its own. *Chapter 7* covers the mechanics, `sstate-cache-management.py` to prune duplicate entries and the `cleansstate` task to discard a single recipe's output. Treat those commands as scheduled maintenance rather than a reaction to a full disk.

On a shared build server or a team cache mirror, the growth compounds because every developer's signatures accumulate in the same store. Budget disk for the cache deliberately, and prune on a cadence tied to the release cycle. Pruning after you tag a release is a natural point, because the intermediate builds that led to it no longer need to be restored, and the current release's entries stay intact for fast incremental work.

Teams often consider a separate Shared State Cache per Yocto Project release. That split is not necessary. BitBake names each entry after the signature and the checksums it computed for the task, so entries from two releases never collide in one cache. The split also adds work, because you must then point each build at the correct cache every time you change a branch or a release. We prefer a single cache and a scheduled prune instead: when you stop supporting a Yocto Project version, run `sstate-cache-management.py` to remove the entries that version leaves behind. This also matches the `bitbake-setup` default, which shares one cache across every setup beneath a Top Directory, as *Chapter 14* explains.

A cache left unmaintained eventually forces an emergency cleanup at the worst moment, mid-release, when a build host runs out of space. Scheduling the prune keeps the cache bounded and its restore hit rate high, which is the whole reason to keep it.

The cleanup itself is the cost. Once the cache reaches its limit, the remaining options are blunt. You prune a large number of entries, or you discard the cache completely, so the team can keep working. Either action removes results that every developer shared. Nobody gets a fast build again until the cache refills, so one full disk on one server slows the whole team for days.

Watch the size before the cache reaches that point, for two reasons. The first is cost. A cache hosted online, or on a shared server that developers download from, costs storage to hold and traffic to serve. Both grow with entries that nobody will restore again. The second is control. A schedule leaves you the choice of what to prune, while a full disk leaves you no choice but to remove entries in bulk.

The Shared State Cache also needs an access boundary, not just a size budget. Every entry is compiled task output, and a recipe marked `LICENSE = "CLOSED"` produces cache entries the same way any other recipe does. `COPYLEFT_LICENSE_EXCLUDE` (covered in *Chapter 15*) keeps proprietary code out of a shared source archive, but that filter applies to archiving, not to `SSTATE_DIR`: whatever gets built lands in the cache, proprietary or not. A shared cache mirror configured through `SSTATE_MIRRORS` must stay within the same audience the proprietary license already permits. Reusing one shared cache across multiple products or customers with different license grants risks handing a proprietary binary to a party the license was never meant to reach. Scope shared caches accordingly, one per customer or product line where licenses differ, rather than one shared cache for convenience.

Enforce the guidelines with version control and continuous integration

The guidelines above only hold when a machine enforces them. A layer revision pinned on one developer's workstation is not pinned for the team. A cache mirror that one workstation reaches does not help anyone else. Version control and Continuous Integration (CI) turn these guidelines into properties of the project instead of habits of one engineer. Adopt the following practices:

- Track the instructions that build the image, not the build directory. A BitBake Configuration Template, described in *Chapter 14*, pins each layer revision and records the `MACHINE`, `DISTRO`, and fragment choices that define the product. A `kas` description serves the same purpose for teams that use that tool. Keep `build/conf/local.conf` out of source control, because it holds local settings that change with every developer and every experiment.
- Build every configuration the product ships in, not only the default one. A template that offers a choice between boards, or between a development and a production build, produces several valid results. CI is where you confirm that each one still builds, and it is also how the repository records the way you intend the product to be used.
- Give CI a Shared State Cache mirror through `SSTATE_MIRRORS`. A CI job starts from a clean checkout, so without a mirror it rebuilds everything on every run. Respect the license boundary described earlier in this chapter when you choose which cache a job may read.
- Build on every merge, and keep the compliance output as a build artifact. *Chapter 15* already produces the license manifests, the archived sources, and the SBOM during the build, so publishing them from CI costs only storage.

A build that fails only on the CI machine, for reasons nobody reproduces, usually means the tracked instructions no longer describe the build the team runs. Keeping those instructions accurate is the point of the practice, and CI is what tells you when they drift.

Create a software bill of materials

The Yocto Project build system can detail all the components used in an image, including the licenses for each software component. This information is compiled into a Software Bill of Materials (SBOM) using the Software Package Data Exchange (SPDX) standard (`https://spdx.dev/`). The SPDX format offers the benefit of working with existing tools, enabling additional automation that is not feasible with the standard license output format.

The SBOM plays a crucial role in maintaining open-source license compliance. Poky generates it by default, since `create-spdx` is part of `INHERIT_DISTRO`. Disable it with `INHERIT_DISTRO:remove = "create-spdx"` if the build-time cost is not worth it for a given target. For guidance, refer to the *Creating a Software Bill of Materials* section in the *Yocto Project Development Tasks Manual* (`https://docs.yoctoproject.org/wrynose/dev-manual/sbom.html`).

Make your own components as visible in that document as everyone else's. A recipe left at `LICENSE = "CLOSED"` builds without complaint, and the resulting SBOM records its license as `NONE`, so the proprietary software at the center of your product becomes the one entry that documents nothing. That gap is easy to miss during development and expensive to explain later, when the SBOM is answering a customer questionnaire or supporting a regulatory filing. Give every in-house recipe a named license and a license text the build system can find, as *Chapter 15* describes. The cost is a few lines of metadata per recipe, and it is far lower before a product ships than after an auditor asks why the vendor's own code is the only unlicensed thing in the image.

Guidelines to follow for general projects

This section outlines key project-related guidelines designed to mitigate general project risks and prevent common pitfalls for smoother development and execution.

Continuously monitor the project license constraints

Depending on the project, license compliance can range from a minor concern to a major priority. Some projects face strict license constraints, such as:

- Strict limitations on using GPLv3-licensed software
- Potential exposure to copyleft obligations that could affect the proprietary nature of project-specific intellectual property
- Comprehensive company policies mandating adherence to specific license constraints

It is advisable to address license compliance early in the project to minimize rework later. However, both project license constraints and component licenses can evolve, necessitating continuous monitoring to maintain compliance throughout the project lifecycle. The compliance machinery in *Chapter 15* produces manifests, archived sources, and SBOM documents at every build, which is the evidence base ongoing monitoring depends on.

Security can harm your project

As connectivity expands, every device becomes a potential target for security attacks. As embedded device developers, we have a responsibility to contribute to a safer digital ecosystem by taking the following actions:

- Regularly scan embedded Linux software for known security vulnerabilities
- Monitor critical software for security updates and fixes
- Establish a reliable process for deploying fixes to field devices

The Yocto Project infrastructure, as outlined in the *Checking for Vulnerabilities* section of the *Yocto Project Security Manual* (`https://docs.yoctoproject.org/wrynose/security-manual/vulnerabilities.html`), provides tools for scanning recipes for known Common Vulnerabilities and Exposures (CVE), including the `OE_FRAGMENTS += "core/yocto/sbom-cve-check"` fragment that enables image-wide scanning. In Yocto Project 6.0, this workflow runs through the `sbom-cve-check` class, which consumes the SPDX 3.0 SBOM produced by `create-spdx`, so a single artifact serves both license compliance and security reporting. Yocto Project 6.0 removed the standalone `cve-check` class. The *Yocto Project 6.0 Release Notes* (`https://docs.yoctoproject.org/wrynose/migration-guides/release-notes-6.0.html`) name its direct successor as the per-recipe `sbom-cve-check-recipe` class, so projects upgrading from an earlier release must move to that class rather than `cve-check`. *Chapter 15* covers the migration path and the new classes in detail.

A report is the start of the work, not the end of it. A scan of a full image lists many CVEs, and a large share of them do not apply to your product. The vulnerable code path may not be built, the affected component may not reach the shipped image, or the fix may already sit in a patch you carry. The scan writes its output under `build/tmp/deploy/images/` beside the image, as a JSON summary and as an enriched SPDX document. Someone must then decide which case each entry falls into, and that decision is the expensive part.

Record each decision in the metadata, not in a spreadsheet. The `CVE_STATUS` variable flag holds a judgment and its reason for a single CVE, as in `CVE_STATUS[CVE-2016-10642] = "cpe-incorrect: this CVE covers the npm package, not the recipe we build"`. The keyword before the colon tells the scan how to classify the entry. Keywords such as `cpe-incorrect`, `not-applicable-config`, `not-applicable-platform`, and `disputed` drop it from the report, while `backported-patch`, `cpe-stable-backport`, and `fixed-version` mark it as already fixed. Use `CVE_STATUS_GROUPS` when several CVEs share one decision and one reason.

A decision recorded this way survives the next scan and the next engineer. A decision taken in a meeting does not, so the same CVE returns at the next release and someone investigates it again.

Vulnerability scanning should not be the only focus, however. BSP components might also require security updates, which vendors often neglect.

Deploying a fix to devices already in the field is an architectural decision, not a reaction. A device you cannot update in the field turns every CVE into a recall or a service visit, and no scanning tool repairs that. Over-the-air updates are the practical answer, and the choice to support them belongs to product design, long before the first CVE arrives. Several projects solve this for embedded Linux, including SWUpdate, RAUC, Mender, and UpdateHub, each with its own Yocto Project layer. Comparing them is outside the scope of this book. What matters here is that you pick one early, because retrofitting an update mechanism into a shipped product costs far more than building it in.

Ultimately, the level of security vigilance required will depend on the project's niche and risk tolerance.

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. 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 must 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 working on updating to the next Yocto Project release.

A frozen BSP shows this cost most clearly. Upstream does not stop. Every release, every CVE fix, and every bug fix that you do not take adds to the legacy you carry alone. Freezing a version

looks like a way to hold the cost down. It usually does the opposite, because the gap grows every week the freeze lasts.

The bill arrives when the update can no longer wait. The volume of changes to review, test, and integrate is then enormous. You handle it in one block instead of in small steps. That single migration costs more than the continuous work would have cost. It also carries far more risk, because you change everything at once.

Upstream review also finds what your own review missed. A maintainer who did not design the change reads it without your assumptions, and questions the parts your team already settled. That feedback regularly exposes a gap nobody flagged internally, and it sometimes catches a serious flaw before it reaches a product. An internal review cannot reproduce that view, because everyone in the room shares the same context.

Security removes the option to wait. Most devices ship connected, so leaving known vulnerabilities in the field is not a decision a product team can defend. Regulation now says the same. The European Union's Cyber Resilience Act, Regulation (EU) 2024/2847 (`https://eur-lex.europa.eu/eli/reg/2024/2847/oj`), requires manufacturers to report actively exploited vulnerabilities. That duty applies from 11 September 2026, and the regulation applies in full from 11 December 2027. Upstreaming is what lowers the total cost of meeting it.

Tackle project risk points and constraints as soon as possible

As the software and hardware must work together, several aspects are directly tied to hardware design. To mitigate project risks, it is crucial to anticipate and validate critical software and hardware requirements early on, addressing questions such as:

- Is the amount of memory we intend to use sufficient, or is it excessive?
- Does the hardware consume an appropriate amount of power within our constraints?
- Can the target GPU render the required animations effectively?
- Are the Linux kernel drivers available for all planned peripheral devices, or will custom development be necessary?

These questions can often be answered using a reference or well-known board with an existing BSP. Such a board allows the creation of a Minimal Viable Product (MVP) without the need for custom hardware design. Beyond initial validation, these boards remain valuable for:

- Continuing software development while the custom board and BSP are being prepared
- Serving as a comparison point against the custom design
- Acting as a reference to identify whether bugs are specific to the custom board and BSP

Given these advantages, it is advisable to postpone custom board design for as long as possible. Delaying provides flexibility to adjust project elements, such as switching peripherals due to driver availability or modifying CPU and memory specifications as the application matures.

When transitioning to a custom design, it is beneficial to keep it as close as possible to the reference board. While deviations from the reference design may be necessary, they introduce risks of design issues and increase the cost and complexity of developing a custom BSP. The board-layer structure and boot workflow that carry these components are covered in *Chapter 16*.

Summary

Phew! In this final chapter, we introduced you to a set of practical guidelines drawn from real-world projects. We hope these insights provide valuable guidance as you embark on your next project.

Throughout this book, we have equipped you with the foundational knowledge needed to explore any aspect of the Yocto Project independently. You now have a solid understanding of what goes on behind the scenes when BitBake builds a recipe or an image. With this knowledge, you're ready to innovate and experiment. The journey is now yours to take, and the fun starts here!

19

Unlock Your Exclusive Benefits

Your copy of this book includes the following exclusive benefits:







Follow the guide below to unlock them. The process takes only a few minutes and needs to be completed once.

Unlock this Book's Free Benefits in 3 Easy Steps

Step 1

Keep your purchase invoice ready for *Step 3*. If you have a physical copy, scan it using your phone and save it as a PDF, JPG, or PNG.

For more help on finding your invoice, visit `https://www.packtpub.com/en-us/unlock?step=1`.

> **Note**
>
> **Note**: If you bought this book directly from Packt, no invoice is required. After *Step 2*, you can access your exclusive content right away.

Step 2

Scan the QR code or go to `packtpub.com/unlock`.

On the page that opens (similar to *Figure 19.1* on desktop), search for this book by name and select the correct edition.

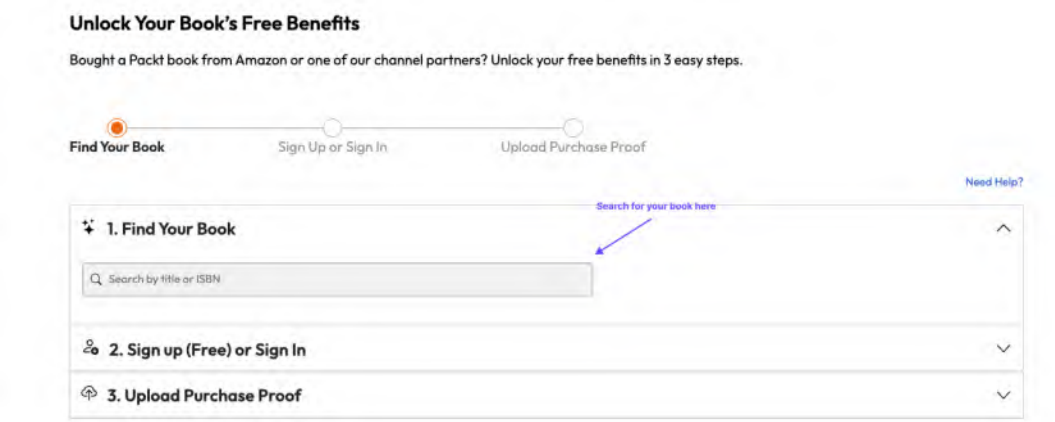


Figure 19.1: Packt unlock landing page on desktop

Step 3

After selecting your book, sign in to your Packt account or create one for free. Then upload your invoice (PDF, PNG, or JPG, up to 10 MB). Follow the on-screen instructions to finish the process.

Need Help

If you get stuck and need help, visit `https://www.packtpub.com/unlock-benefits/help` for a detailed FAQ on how to find your invoices and more. This QR code will take you to the help page.

Note

Note: If you are still facing issues, reach out to `customercare@packt.com`.



packtpub.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

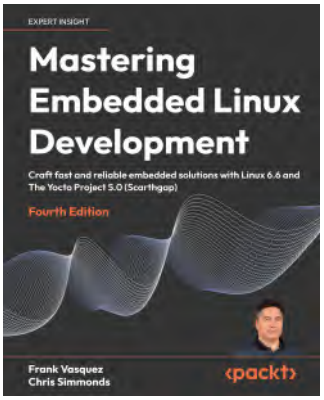
Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

At www.packtpub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

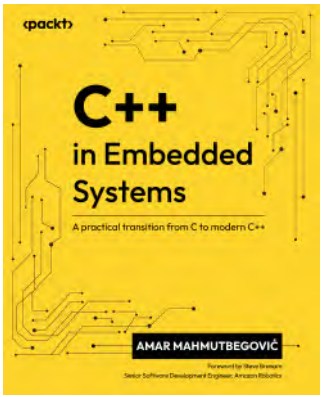


Mastering Embedded Linux Development

Frank Vasquez, Chris Simmonds

ISBN: 9781803241463

- Cross-compile embedded Linux images with Buildroot and Yocto
- Enable Wi-Fi and Bluetooth connectivity with a Yocto board support package
- Update IoT devices securely in the field with Mender or balena
- Prototype peripheral additions by connecting add-on boards, reading schematics, and coding test programs
- Deploy containerized software applications on edge devices with Docker
- Debug devices remotely using GDB and measure the performance of systems using tools like perf and ply



C++ in Embedded Systems

Amar Mahmutbegović

ISBN: 9781835881156

- Debunk myths and misconceptions about using C++ in embedded systems
- Set up build automation tailored for C++ in constrained environments
- Leverage strong typing to improve type safety
- Apply modern C++ techniques, such as Resource Acquisition Is Initialization (RAII)
- Use Domain Specific Language (DSL) with a practical example using Boost SML
- Implement software development best practices, including the SOLID principle, in embedded development

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packt.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 that you've finished *Embedded Linux Development Using Yocto Project*, we'd love to hear your thoughts! Scan the QR code below 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.



https://packt.link/r/1806695073

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

Index

A

AMD **244**

AUTOREV **293**

Allwinner **244**

AlmaLinux **17**

Arduino UNO Q **246, 257**
- booting 263
- features 246
- flashing, with qdl 262
- image, building 261
- qcomflash image type, enabling 261
- source structure, setting up 258 – 261

append
- using 94
- versus = 95

application debugging **124**

architecture-independent target recipes **72**

available built-in fragments **194**

B

BSP layer **242 – 244**

BeagleBone Black **245 – 247**
- booting 249
- features 246
- image, building 248
- image, writing to SD card 249
- source structure, setting up 247, 248

BitBake **4**
- class files 100
- initialization sequence 51, 52
- role, in build workflow 40

BitBake Configuration Fragment **21**

BitBake Configuration Instance **21**

BitBake Configuration Template files **18**

BitBake Registry **192**

BitBake Setup **19**

BitBake User Manual
- reference link 40

BitBake configuration template status
- checking 207, 208

BitBake metadata system **173**
- variable scope 173

BitBake tasks **63, 64**

Board Support Package (BSP) **7**

Bootloader **243**

Build History feature **40**

base package recipe
- creating automatically, with devtool 158 – 160

basic variable assignment **91**

bitbake-config-build
- build configuration, managing 193

bitbake-layers
- layers, inspecting 42, 43

bitbake-setup **18, 287**
- build environment, activating 21
- configuration files, reviewing 21
- directory structure 20, 21
- installing 18, 19
- layers, managing 205, 206
- project, initializing 19

board selection
- key considerations 243

build directory **68**
- constructing 69

build directory, subdirectories
- conf 68

downloads 68
sstate-cache 68
tmp 68

build host system
preparing 12

build options
adding, to recipes 179, 180

build/tmp directory **70**
deploy 70
pkgdata 71
sysroots-components 71
sysroots-uninative 71
work 71
work-shared 71

build/tmp/work directory **71 – 75**
all-poky-linux 72
image 73
package 73
packages-split 73
source-tree 73
temp 73
x86-64-v3-poky-linux 72
x86_64-linux 72

C

CentOS Stream **17**

CycloneDX **231, 236**

class files
in BitBake 100

classes **43, 44**

commercial licenses **225, 226**

conditional appending **96**
example 97

conditional metadata sets **95**

configuration files **43, 44**

configuration fragments
for Kconfig-based projects 187 – 189

configuration precedence **200**

configurations
exploring 192

conflicts **200**

context-based suggestions **33**

copyleft **220, 221**
compliance, achieving with Yocto Project 221, 222

core-image-full-cmdline
building, from VS Code 34

cross recipes **72**

cross-development SDKs
Extensible SDK (eSDK) 107
Standard SDK 107
types 107

custom BitBake configuration fragments
built-in fragments 204
creating 201
enabling 203
organizing, with subdirectories 203, 204
standard fragment 201 – 203

custom distribution
creating 167
using 167 – 171

D

DISTRO_FEATURES
interaction, with MACHINE_FEATURES 172, 173

DISTRO_FEATURES_DEFAULTS
adjusting 171

Debian Package Manager (DEB) **78**

Debian-based distributions **15, 16**

Development Shell **123**
using 130 – 132

datastore **90**

dependencies **53 – 55**
build-time 53
runtime 53
task 53

dependency graph
inspecting 55, 56

devtool **158**
for creating base package recipe automatically 158 – 160
source git tree, working with 119
used, for building image 113
used, for building recipe 117, 118

used, for modifying existing recipe 118
dnf tool **78**
dpkg **78**

E

EU Cyber Resilience Act (EU CRA) **232**
Extensible SDK (eSDK) **107**
devtool source git tree, working with 119
existing recipe, moving with devtool 118, 119
extending 120
generating 111
image, building with devtool 113
image, running on QEMU 114, 115
installing 111, 112
recipe, building with devtool 117, 118
recipe, creating from external git repository 115 – 117
structure 113
using 110
VS Code, using with 121
enabled built-in fragments **194**
enabled fragments **194**
end-of-life (EOL) **8**
environment
preparing 50, 51
external git repository
recipe, creating from 115 – 117

F

Fedora **16**
fetcher **56**
file inclusion
with include 97
with require 97
file search paths **183, 184**
files
adding, to existing packages 181 – 183
fragments **44**
disabling 194 – 196
enabling 197, 198
inspecting 199
listing 193, 194

G

GNU Debugger
using, for debugging 132
General Public License (GPL) **220**
Git repositories **59, 60**
Graphics Processing Unit (GPU) **269**
general projects
guidelines 296 – 300
generated workspace file
opening 30
global classes **101**
global variables **173**
guidelines, Yocto Project **286**
AUTOREV as SRCREV, avoiding 293
bitbake-setup 287, 288
configuration files, cleaning
custom distribution, creating 290
guidelines, enforcing with version control and continuous integration 295
layers, managing 286, 287
multiple layers creation, avoiding 289
multiple patches, avoiding for Linux kernel and bootloader modifications 292
product metadata, preparing for Yocto Project releases 290
reusing existing images, avoiding 291
Shared State Cache, maintaining 293, 294
software bill of materials, creating 295, 296
Standard SDK, using 291, 292
guidelines, general projects
maintenance costs 298, 299

project license constraints, monitoring continuously 296
project risk points and constraints, tackling 299, 300
security 297, 298

H

hash equivalence 65

I

Intel 244
Itsy Package Management System (IPK) 78
image
creating 152 – 155
generating, for on-device development 107
running, in QEMU 25 – 27
tracking 124, 125
wrapping, for machine 163 – 166
image-buildinfo class
reference link 286
immediate variable expansion 92
include statements
using, for file inclusion 97
inheritance 45

K

Kconfig-based projects
configuration fragments 187 – 189
kas
reference link 287

L

Lesser GPL (LGPL) 220
Linux kernel 243
building, with Standard SDK 110
Linux-based system
preparing 14, 15
Long-Term Support (LTS) 8
layer structure 40 – 42
layers
creating 148 – 151
inspecting, with bitbake-layers 42, 43
managing, with bitbake-setup 205, 206
metadata, adding 151
list
appending 94
prepending 94
list items
removing 95

M

MACHINE_FEATURES
interaction, with DISTRO_FEATURES 172, 173
MACHINE_FEATURES_DEFAULTS
adjusting 171
Machines tab, OpenEmbedded Layer Index
reference link 243
machine-specific target recipes 72
metadata
indexing 51
parsing 52
metadata collections 40
metadata debugging 124
metadata variables
debugging 129, 130
missing file
navigating through 133 – 135

N

NXP 244
native recipes 72
network access
disabling 62, 63
new machine definition
creating 160 – 163

O

OpenEmbedded Layer Index 144
OpenEmbedded Project 6
on-device development
image, generating 107

openSUSE Leap **17**
opkg package manager **78**

P

PR Service **85**
Poky **167, 232**
Poky distribution **4**
Python
anonymous functions 99
combining, with shell 100
executable metadata, defining 98, 99
functions, defining in global namespace 99
variable expansion 98
package feeds **85, 86**
using 86 – 88
package formats **78**
Debian Package Manager (DEB) 78
Itsy Package Management System (IPK) 78
Red Hat Package Manager (RPM) 78
selecting 78, 79
package installation
code, running 79 – 81
package recipe
adding 156, 157
package versioning **83**
packages
debugging 126
inspecting 126 – 128
tracking 125, 126
using, to generate root filesystem image 84, 85
patch
applying 181
physical hardware
using 244
prepend
using 95
profiling **133**

Q

Qualcomm **244**
Quality Assurance (QA) workflows **7**
Quick EMUlator (QEMU) **268**
benefits 269
images, running 25 – 27, 114, 115

R

RISC-V **244**
Raspberry Pi **244**
Raspberry Pi 4 **245, 250**
booting 254
features 246
image, building 252
image, writing to SD card 252, 253
Raspberry Pi 5 **245, 250**
booting 254
features 246
image, building 252
image, writing to SD card 252, 253
Red Hat Package Manager (RPM) **78**
Rocky Linux **17**
real hardware
selection, situations 269
recipe feature configuration
modifying 185 – 187
recipe-specific classes **101**
recipe-specific variables **174**
recipes **43, 44**
build options, adding to 179, 180
remote file downloads **57, 58**
repo tool
reference link 287
reproducible team workflows **208**
BitBake setups, keeping under top directory 213
configuration 216
configurations, synchronizing across teams 215, 216
custom BitBake configuration templates, creating 209 – 212
private BitBake registries, managing 213
reproducibility 216

source overrides, for development branches 214, 215
sources 216
require statements
using, for file inclusion 97
root filesystem image
generating, with packages 84, 85
runqemu
audio stacks, validating 277
capabilities 270, 271
custom boot arguments, passing 276
graphical applications, testing 271, 272
Linux kernel image, selecting 275, 276
memory constraints, validating 273 – 275
root filesystem, selecting 275, 276
using, to help with image regression tests 277 – 280
using, to run package tests with ptest 280 – 282
using, with serial console 276
runqemu, running from VS Code **35**
integrated terminal, using 35
workspace task, defining 35 – 37
runtime package dependencies
specifying 83, 84

S

SDK content
tracking 125, 126
SRCREV **293**
Software Bill of Materials (SBOM) **231, 236**
Software Package Data Exchange (SPDX) **231**
URL 295
Standard SDK **107**
installing 108, 109
Linux kernel, building with 110
used, for building application 109
using 108
System-on-Modules (SoMs) **289, 292**
shared state cache **81, 82**
single-board computers (SBCs) **292**
software development kit (SDK) **106**
software development kit (SDK) installers **70**
software licensing
managing, with Yocto Project 222 – 224
source code
fetching 56, 57
source code downloads
optimizing 60 – 62
string
appending 94
prepending 94
syntax highlighting **32**
sysroot directories **75, 76**

T

Task Graph **45**
Texas Instruments **244**
target image
building 22, 23
task
extending 178, 179
temporary build directory **70, 71**
tracing **133**
tune-specific target recipes **72**

U

U.S. Executive Order 14028 **232**
unused fragments **194**
use cases **176, 177**
user space packages **243**

V

VS Code
core-image-full-cmdline, building 34
using, with Extensible SDK 121

Video Processing Unit (VPU) **269**
VisionFive 2 **245, 254**
booting 256
features 246
image, building 256
image, writing to SD card 256
source structure, setting up 254, 255
variable assignment operators **90, 91**
?= 92, 93
??= 93
basic variable assignment 91
variable expansion operators **90 – 92**
deferred evaluation 92
immediate variable expansion 92
variable scope, in BitBake **173**
global variables 173
recipe-specific variables 174

W

Whisk
reference link 287
Windows Subsystem for Linux (WSL 2)
using 13
work directory **71 – 75**
workspace cleanup classes
tuning 101

X

Xilinx **244**

Y

Yocto Buddy
reference link 287
Yocto Project **4 – 6, 221**
benefits 4, 5
characteristics 5
copyleft compliance, achieving with 221, 222
debugging, with logging 128, 129
guidelines 286 – 296
LTS cycle 8 – 10
metadata layers 6, 7
release cadence 8 – 10
software licensing, managing with 222 – 224
Yocto Project 6.0 **137**
layers 138 – 140
layers source code 140 – 142
meta layers, adding 142, 143
Yocto Project 6.0 (Wrynose) **3**
migration guide 10
release notes 10
Yocto Project BitBake extension
features 33, 34
installing 31, 32
Yocto Project configuration, for compliance **226**
compilation scripts, providing 230
copyleft compliance considerations 230
license auditing 226, 227
license text, providing in image 230
output file formats 230
proprietary license, declaring 232 – 235
SBOM-based vulnerability scanning 235, 236
Software Bill of Materials (SBOM) 231, 232
source code modifications 230
source code, providing 227 – 229

Z

Zephyr **4**