1. Linux for Embedded Systems

In This Chapter

The *Internet of Things* is inspiring the imagination of visionaries and likewise the creativity of engineers. As a universal computing network of myriad connected devices collecting, analyzing, and delivering data in real-time, it carries the promise of a new era of information technology.

Devices comprising the Internet of Things need to meet an entirely new set of requirements and provide functionality previously not found in embedded systems. Connectivity, including through cellular data networks, is an obvious one, but there is also remote management, software configuration and updates, power efficiency, longevity, and, of course, security, to just name a few.

This changing landscape of embedded systems requires a different approach to building the software stacks that operate this new breed of connected hardware.

1.1 Why Linux for Embedded Systems?

Linux debuted as a *general-purpose operating system* (GPOS) for PC hardware with Intel x86 architecture. In his now famous post on news:comp.os.minix, Linux creator Linus Torvalds explicitly stated, “I’m doing a (free) operating system… . It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks ….”

Driven by the rise of the Internet, Linux quickly evolved into a server operating system providing infrastructure for web servers and networking services for many well-founded reasons.

Nevertheless, Linux remained true to its GPOS origins in three major aspects that do not make it the premier choice of engineers for an embedded operating system at first:

- **Filesystem**: Linux is a file-based operating system requiring a filesystem on a block-oriented mass storage device with read and write access. Block-oriented mass storage typically meant hard drives with spinning platters, which are not practical for most embedded use cases.

- **Memory Management Unit (MMU)**: Linux is a multitasking operating system. Effective task switching mandates that individual processes have their private memory address space that can easily be mapped into physical memory when the process is running on the CPU. Microcontrollers that have been widely used for typical embedded applications do not provide an MMU.
- **Real Time**: Embedded systems running critical applications may require predictive responses with guaranteed timing within a certain margin of error, commonly referred to as *determinism*. The amount of error in the timing over subsequent iterations of a program or section thereof is called *jitter.* An operating system that can absolutely guarantee a maximum time for the operations it performs is referred to as a *hard real-time system*. An operating system that typically performs its operations within a certain time is referred to as *soft real-time*. Although several solutions providing real-time capabilities for Linux, most notably *PREEMPT-RT*, had been developed as early as 1996, they are still not part of the mainline Linux kernel.

During the last couple of years, advances in semiconductor technology have helped to overcome these hurdles for the adoption of Linux in embedded systems. Ubiquitously available, inexpensive, and long-term reliable flash memory devices used in many consumer products, such as digital cameras, are providing the necessary mass storage for the filesystem. Powerful system-on-chips (SoC) designs combining one or multiple general-purpose CPU cores with MMU and peripheral devices on a single chip have become the embedded systems engineer's choice of processor and are increasingly replacing the microcontroller in embedded applications.

Today we are seeing an explosive growth in the adoption of Linux for embedded devices. Virtually every industry is now touched by this trend. In Carrier-Grade Linux (CGL), the operating system has been adopted for products in public switched telephone networks and global data networks. Chances are that you carry a cellphone, watch television with a set-top box and high-definition television, surf the Internet with broadband modems and networking switches, find your way with a personal navigation device, and daily use many other devices that are powered by Linux.

There are many reasons for the rapid growth of embedded Linux. To name a few:

- **Royalties**: Unlike traditional proprietary operating systems, Linux can be deployed without any royalties.
- **Hardware Support**: Linux supports a vast variety of hardware devices including all major and commonly used CPU architectures: ARM, Intel x86, MIPS, and PowerPC in their respective 32-bit and 64-bit variants.
- **Networking**: Linux supports a large variety of networking protocols. Besides the ubiquitous TCP/IP, virtually any other protocol on any physical medium is implemented.
- **Modularity**: A Linux OS stack is composed of many different software packages. Engineers can customize the stack to make it exactly fit their application.
- **Scalability**: Linux scales from systems with only one CPU and limited resources to

systems featuring multiple CPUs with many cores, large memory footprints, several networking interfaces, and much more.

- **Source Code**: The source code for the Linux kernel, as well as for all software packages comprising a Linux OS stack, is openly available.
- **Developer Support**: Because of its openness, Linux has attracted a huge number of active developers, and those developers have quickly built support for new hardware.
- **Commercial Support**: An increasing number of hardware and software vendors, including all semiconductor manufacturers as well as many independent software vendors (ISV), are now offering support for Linux through products and services.
- **Tooling**: Linux provides myriad tools for software development ranging from compilers for virtually any programming language to a steadily growing number of profiling and performance measurement tools important for embedded systems development.

These and many other reasons now make Linux the premier choice of embedded systems engineers, fueling its accelerated adoption for consumer and professional products.

1.2 Embedded Linux Landscape

Embedded systems are diverse. With the huge variety of hardware inevitably comes the burden of software adaptation, most notably the operating system, which provides abstraction from the hardware through its libraries and application programming interfaces (API). There is no one-size-fits-all, and you, as the systems engineer, will have to start somewhere with your embedded Linux project.

In the following paragraphs, we provide an overview of the most commonly used open source projects for embedded devices. Beyond those there are, of course, a couple of commercial embedded Linux offerings from operating system vendors (OSVs).

1.2.1 Embedded Linux Distributions

Similar to desktop and server Linux distributions, an ever-evolving variety of embedded Linux distributions is developed by community projects and commercial operating system vendors. Some of them are targeted for a specific class of embedded systems and devices, while others are more general in nature with the idea to provide a foundation rather than a complete system.

Android

Despite its primary target being mobile phones and tablet computers, Android (www.android.com, http://developer.android.com, http://source.android.com) is enjoying growing popularity as an operating system for all kinds of embedded devices. That does not come as any surprise, as its source code is freely available and includes a build system with configuration tools that lets developers adapt the system to different hardware devices.

In particular, if the target device is utilizing an ARM-based SoC and has a touch screen, then Android is a popular choice among systems engineers as the necessary support for the hardware is a core part of the system. Ports to Intel x86 architecture do exist, but there is much less hardware available, and development is typically also more expensive.

However, Android does not fill every need of embedded devices. While it utilizes the Linux kernel and other software packages generally found in a Linux OS stack, its fundamental architecture is different from a typical Linux OS stack. Android uses its own flavor of a C library with a reduced set of APIs, as well as its own filesystem layout and other specific extensions. These modifications make it less than straightforward to port standard Linux software packages to Android.

While Android is an open source project in the sense that the source code for the entire system is freely available and can be used, modified, and extended for any purpose with very few restrictions, developers cannot contribute their changes back to Android. Google alone controls the roadmap of the system. The CyanogenMod (www.cyanogenmod.org) community distribution based on Google's Android releases is trying to fill this void.

Nevertheless, the Linux kernel does owe Android one important extension: power management. While frowned upon by some Linux kernel developers because of their simplistic architecture, the Android Wake Locks have become the de facto standard of Linux power management.

Ångström Distribution

The Ångström Distribution, with its homepage at www.angstrom-distribution.org, is increasingly becoming an important resource for projects because of its growing list of supported *development boards*. Ångström is a community distribution that was started by a group of developers who worked on the *OpenEmbedded*, *OpenZaurus*, and *OpenSimpad* projects. Ångström has been using the OpenEmbedded tools from its beginning but is now adapting the architecture and structure of the Yocto Project.

OpenWrt

OpenWrt (www.openwrt.org) debuted as an open source operating system targeting embedded devices that route network traffic such as broadband modems, routers, residential gateways, and other consumer premises equipment (CPE). OpenWrt's core components are the Linux kernel, uClibc, and BusyBox.

The first versions of OpenWrt were built on the Linksys GPL sources for their WRT54G residential gateway and wireless router and a root filesystem created with Buildroot—hence the name OpenWrt.

OpenWrt supports a wide variety of hardware devices and evaluation boards. OpenWrt's core strength is the exhaustive list of possibilities to configure networking technologies and protocols, including routing, mesh networking, firewall, address translation, port forwarding, load balancing, and much more.

While OpenWrt is intended to operate devices that typically run without regular human interaction, it provides a sophisticated web interface to comfortably access the many configuration options.

Its focus on connectivity and remote management make OpenWrt a favorite among systems engineers developing connected devices. A writable filesystem with package management makes it straightforward to add functionality even after the system has been deployed.

Embedded Versions of Full Linux Distributions

For many of the fully fledged Linux distributions for desktop, server, and cloud, variants targeting embedded systems are now also available:

- Debian (www.emdebian.org)
- Fedora (https://fedoraproject.org/wiki/Embedded)
- Gentoo (https://wiki.gentoo.org/wiki/Project:Embedded)
- SUSE (https://tr.opensuse.org/MicroSUSE)
- Ubuntu (https://wiki.ubuntu.com/EmbeddedUbuntu)

For system builders and developers familiar with a desktop or server version of a particular Linux distribution, using its embedded variant provides the advantage of familiar tools, filesystem layout, and more.

1.2.2 Embedded Linux Development Tools

Besides utilizing an embedded Linux distribution, you can also build your own custom Linux OS stack with embedded Linux development tools. This gives you the most control and flexibility but in most cases requires more effort.

Baserock

Baserock is an open source project that provides a build system for Linux distributions, a development environment, and a development workflow in one package. Baserock's major characteristics are

- Git as the core to manage essentially everything from build instructions to build artifacts as a means to provide traceability
- Native compilation to avoid the complexity of cross-build environments
- Distributed builds across multiple systems using virtual machines

Currently, Baserock supports building for x86, x86_64, and ARMv7 architectures. The project's homepage is at http://wiki.baserock.org.

Buildroot

Buildroot is a build system for complete embedded Linux systems using GNU Make and a set of makefiles to create a cross-compilation toolchain, a root filesystem, a kernel image, and a bootloader image. The project's homepage is at http://buildroot.uclibc.org.

Buildroot mainly targets small footprint embedded systems and supports various CPU architectures. To jump start development it limits the choice of configuration options and defaults to probably the most commonly used ones for embedded systems:

- *uClibc* is the target library to build the cross-compilation toolchain. In comparison to the GNU C Library (glibc), uClibc is much more compact and optimized for small footprint embedded systems. uClibc supports virtually all CPU architectures as well as shared libraries and threading.
- *BusyBox* is the default set of command line utility applications.

These default settings enable building a basic embedded Linux system with Buildroot typically within 15 to 30 minutes, depending on the build host. However, these settings are not absolute, and the simple and flexible structure of Buildroot makes it easy to understand and extend. The internal cross-toolchain can be replaced by an external one such as *crosstool-ng*, and uClibc can be replaced with other C libraries.

Buildroot already supports many standard Linux packages, such as the X.org stack, GStreamer, DirectFB, and Simple DirectMedia Layer (SDL). The cross-toolchain can be used to build additional packages and have them included with the root filesystem.

Buildroot is very compact and straightforward to set up. A single-file (tarball) download and the installation of a few additional packages on the build host are all that are required to get started. After unpacking the tarball, the command `make menuconfig` launches a text-based user interface enabling configuration of a wide range of supported targets and setting of other options. In addition to *menuconfig*, Buildroot offers *gconfig* and *xconfig*, which are alternative graphical user interfaces.

Buildroot creates everything from source by downloading the source code files directly from the *upstream* projects. A nice feature is that offline builds can be done by downloading all the sources first using `make source`. Buildroot pre-fetches all necessary files and can then configure and run a build without further connectivity to the Internet.

OpenEmbedded

OpenEmbedded (www.openembedded.org) is a build framework composed of *tools*, *configuration data*, and *recipes* to create Linux distributions targeted for embedded devices. At the core of OpenEmbedded is the *BitBake* task executor that manages the build process.

Historically, OpenEmbedded was created by merging the work of the OpenZaurus project with contributions from other projects such as Familiar Linux and OpenSIMpad.

OpenEmbedded has been used to develop a variety of open source embedded projects, most notably the OpenMoko (http://wiki.openmoko.org) project dedicated to delivering a complete open source software stack for mobile phones.

OpenEmbedded, the Yocto Project, and the Ångström Distribution all have the same roots and build on and complement each other in various ways. We will explain the commonalities and differences in the next chapter when we dive into the details of the Yocto Project.

The Yocto Project

The Yocto Project is, of course, the subject of this book. It is listed here to complete this overview of the embedded Linux landscape. You can find its webpage at https://www.yoctoproject.org.

The Yocto Project is not a single open source project but represents an entire family of projects that are developed and maintained under its umbrella. This book describes many of the projects associated with the Yocto Project, in particular, Poky, the Yocto Project's reference distribution, which includes the OpenEmbedded build system and a comprehensive set of metadata.

The embedded Linux landscape is diverse. This list is not all-comprehensive, and there are many more open source projects providing solutions for developing embedded devices with Linux. The projects mentioned here are, in my opinion, the most active and most commonly used ones. Before reading on, you may want to take some time and visit the web pages of these projects. They will give you a good understanding of what the goals of these projects are and how they compare to each other.

There are also a number of commercial offerings complementing the embedded Linux landscape. Commonly, these offerings include cross-development toolchains, distribution builders, application development IDEs, and more. An increasing number of operating system vendors for embedded systems are using the Yocto Project as an upstream. They use the Yocto Project tools to create their product lines. Many of them are members of the Yocto Project and support it with engineering and financial resources.

1.3 A Custom Linux Distribution—Why Is It Hard?

Let's face it—building and maintaining an operating system is not a trivial task. Many different aspects of the operating system have to be taken into consideration to create a fully functional computer system:

- **Bootloader**: The bootloader is the first piece of software responsible for initializing the hardware, loading the operating system kernel into RAM, and then starting the kernel. The bootloader is commonly multistaged with its first stage resident in nonvolatile memory. The first stage then loads a second stage from attached storage such as flash memory, hard drives, and so on.
- **Kernel**: The kernel, as its name implies, is the core of an operating system. It manages the hardware resources of the system and provides hardware abstraction through its APIs to other software. The kernel's main functions are *memory management*, *device management*, and responding to *system calls* from application software. How these functions are implemented depends on the processor architecture as well as on peripheral devices and other hardware configuration.
- **Device Drivers**: Device drivers are part of the kernel. They provide application software with access to hardware devices in a structured form through kernel system calls. Through the device drivers, application software can configure, read data from, and write data to hardware devices.
- **Life Cycle Management**: From power on to shutdown, a computer system assumes

multiple states during which it provides different sets of services to application software. Life cycle management determines what services are running in what states and in what order they need to be started to maintain a consistent operating environment. An important piece of life cycle management is also power management, putting a system into energy saving modes when full functionality is not required, and resuming fully operational mode when requested.

- **Application Software Management**: Application software and libraries make up the majority of software installed on a typical system, providing the end-user functionality. Frequently, many hundreds to multiple thousands of software packages are necessary for a fully operational system.

Linux and the plethora of open source software packages are like the building blocks of a construction kit. Unfortunately, it is more like a puzzle than like Legos. It can be a daunting task to figure out dependencies, incompatibilities, and conflicts between the different packages. Some packages even provide the same or similar functionality. Which one to choose? Eventually, you will have to draw your own blueprint to build the Linux distribution for your embedded project. In principal, you have two ways to go:

- **Top-down**: In this approach, you start with one of the many available Linux distributions and customize it according to your requirements by adding and/or removing software packages. The author took this approach many years ago with a high-speed image processing system running on x86 server hardware. It is a viable approach and has its appeal because using a tested and maintained distribution alleviates some of the more tedious tasks of building and maintaining your own distribution. And you may be able to get support for it. However, it may limit you in your choice of hardware, since most off-the-shelf Linux distributions are built for x86 hardware. And, of course, picking the right distribution to start off with and rightsizing it for your target device is not that simple either.
- **Bottom-up**: The bottom-up approach entails building your own custom Linux distribution from source code starting with a bootloader and the kernel and then adding software packages to support the applications for your target device. This approach gives you the most control (and you will learn a lot about Linux and operating systems in general), but it is also a challenging task. You will have to make many choices along the way, from selecting the right toolchain and setting kernel configuration options to choosing the right software packages. Some of these choices are interdependent, such as the choice of toolchain and target library, and taking the wrong turn can quickly send you down a dead end. After you have successfully built and deployed your own distribution, you are left with the burden of maintaining it—finding patches and security updates for the kernel and all the other packages you have integrated with your distribution.

This is where the strengths of the Yocto Project lie. It combines the best of both worlds by providing you with a complete tool set and blueprints to create your own Linux distribution from scratch starting with source code downloads from the upstream projects. The blueprints for various systems that ship with the Yocto Project tools let you build complete operating system stacks within a few hours. You can choose from blueprints that build a target system image for a basic system with command-line login, a system with a

graphical user interface for a mobile device, a system that is Linux Standard Base compliant, and many more.

You can use these blueprints as a starting point for your own distribution and modify them by adding and/or removing software packages. The remaining chapters of this book walk you through the entire process of building and customizing your Linux distribution and creating your own blueprints using the Yocto Project tools, which will give you repeatable results every time you build your system.

1.4 A Word about Open Source Licensing

When building a system based on, or that includes, open source software, you will inevitably have to pay attention to open source licensing. Originating authors of software are of course free to choose whatever license they prefer for their works, which has led to a long and growing list of open source licenses. There is no single license, and whether you like it or not, you will have to deal with many of them. Some open source projects even use more than one software license. One of them is BusyBox.

One of the most common, if not *the* most common, open source licenses is the GNU General Public License (GPL).[1] Now in its third version, the GPL is widely considered the mother of open source licenses. Although some sources name the Berkeley Software Distribution (BSD) License, created in 1990, as the first open source license, the GPL predates it by a year, having been written by Richard Stallman and introduced in 1989.

1. For a complete text of the GPL license, refer to Appendix A or see www.gnu.org/licenses/gpl.html.

One popular myth attributed to open source licenses is that open source software is free. However, the second paragraph of the GPL clarifies the common misunderstanding: “When we speak of free software, we are referring to freedom, not price.” Professional engineering managers probably wholeheartedly agree—while you can download open source software for free, developing and deploying products based on it commonly carries significant engineering cost. In that sense, open source software is no different from commercial software offerings.

In comparison to commercial or closed source software licenses, open source licenses are *permissive*, meaning they grant you the freedom to use and run the software, the right to study and modify it, and the permission to distribute the original code and its modifications. This broad freedom makes it tempting to treat open source licenses rather casually. In one word: *Don’t!* Open source licenses are binding and enforceable, as any commercial license is.

Most open source licenses explicitly stipulate a few major conditions you must comply with when shipping products based on open source:

- **Attribution**: Authors must be attributed as the creators of the work. You must not remove any copyright notices of any of the authors of the source code.
- **Conveyance**: Conveyance typically refers to conveying verbatim copies of the software’s source code, conveying modified versions of the source code, and conveying non-source forms, such as binary files or firmware embedded in products. In the latter case, many open source licenses, including the GPL, require you to

convey the corresponding source code with the product or auxiliary documentation.

- **Derivative Works**: This term commonly refers to a creation that includes all or major portions of a previously created work. What this exactly means for open source software is still unclear, since there are no legal test cases for it yet. Most typically, it refers to modification of the source code and/or additions to it but, for some licenses, also to linking, even dynamically at runtime, to libraries. Under the terms of the license, the author of a derivative work is required to distribute the work under exactly the same licensing term as the original work. This makes the license *self-perpetuating*.

This book was not written with the intention of providing legal advice concerning open source licensing. However, we strongly urge you to pay close attention to the licenses used by software packages you include with your product before actually shipping the product. While the legal field of open source licensing is still quite new, a growing number of legal experts are now specializing in this field. If in doubt, seek professional advice from the experts.

1.5 Organizations, Relevant Bodies, and Standards

As Linux and open source continue to grow their market share in computing, telecommunications, consumer electronics, industrial automation, and many other fields, organizations and standards are emerging to influence acceptance and adoption of Linux and open source technologies themselves as well as the principles of open collaboration and innovation they stand for.

This section introduces some of the organizations, bodies, and standards with which you may want to become familiar.

1.5.1 The Linux Foundation

The Linux Foundation (www.linuxfoundation.org) is a “non-profit consortium dedicated to fostering the growth of Linux. Founded in 2000, the Linux Foundation sponsors the work of Linux creator Linus Torvalds and is supported by leading technology companies and developers from around the world.”

The Linux Foundation marshals the resources and contributions of its members and the open source community by

- Promoting Linux and providing a neutral environment for collaboration and education
- Protecting and supporting Linux development
- Improving Linux as a technical platform

The Linux Foundation directly sponsors the work of Linus Torvalds and other key Linux developers so that they remain independent and can focus on improving Linux. The Linux Foundation also sponsors several workgroups and collaborative projects to define standards and to advance Linux in certain areas and industries. Some of these projects are outlined in the following sections.

1.5.2 The Apache Software Foundation

More than 140 open source software projects are hosted by the Apache Software Foundation (ASF). For these projects, the ASF provides a collaboration framework including financial backing, intellectual property management, and legal support. The ASF website can be found at www.apache.org.

You are probably familiar with some of the most well-known ASF projects, such as the Apache HTTP Server, Ant build tool for Java, Cassandra cloud database, CloudStack cloud computing infrastructure, Hadoop distributed computing platform, and Tomcat web server for Java Servlet and JavaServer Pages.

All ASF projects and all software produced under the umbrella of the ASF are licensed under the terms of the *Apache Licenses.* One important property of the Apache Licenses is that contributors retain full rights to use their original contributions for any purpose outside the Apache projects while granting the ASF and the projects the rights to distribute and build upon their work.

1.5.3 Eclipse Foundation

The Eclipse Project (www.eclipse.org) was created in 2001 by IBM to build a support community of developers and software vendors around the Eclipse Platform. The Eclipse Platform started as a flexible IDE framework for software development tools. In 2004, the Eclipse Foundation was founded as a legal entity to marshal the resources of the project. The Eclipse Foundation provides IT infrastructure and IP management to the projects operating under its umbrella and supports their operations with development and engineering processes to ensure project transparency and product quality.

Besides the Eclipse IDE, the list of projects hosted under the auspices of the Eclipse Foundation includes development tools for virtually any programming language, software and data modeling tools, web development tools, and many more.

Embedded software development frameworks frequently build on top of the Eclipse IDE to offer convenient round-trip development including target debugging and profiling within the same IDE. The Yocto Project provides an Eclipse plug-in enabling the use of a Yocto Project–created toolchain directly from within the IDE.

1.5.4 Linux Standard Base

As outlined in previous sections, there are many ways to build a Linux OS stack. While flexibility is good, it comes with the burden of fragmentation. The goal of the *Linux Standard Base* (LSB) is to establish a set of common standards for Linux distributions. Common standards provide application developers assurance that the code that they develop on one Linux distribution will also run on other Linux distributions without additional adaptations.

In addition, LSB gives developers peace of mind when it comes to the continuity of a particular Linux distribution. As long as future versions of a distribution remain compliant with a particular LSB version, the application will continue to run on the future versions of the distribution too.

The LSB project provides a comprehensive set of specifications, documentation, and tools to test the compliance of a distribution with a particular LSB version.

While API and application binary interface (ABI) compliance may not necessarily be at the top of the list for embedded systems engineers, familiarizing yourself with the concepts and specifications may help with your embedded project in the long run. Even if you do not intend for third-party developers to contribute applications to your embedded platform, compliance considerations similar to those of the LSB project undoubtedly support the platform strategy of your products.

LSB is a Linux Foundation workgroup. You can find its website at www.linuxfoundation.org/collaborate/workgroups/lsb.

1.5.5 Consumer Electronics Workgroup

The Consumer Electronics (CE) Workgroup is a workgroup operating under the umbrella of the Linux Foundation. Its mission is to promote the use of Linux in embedded systems used in consumer electronics products as well as promote enhancement of Linux itself. The CE Workgroup started its work in 2003 as the Consumer Electronics Linux Forum (CELF) and merged with the Linux Foundation in 2010 for better alignment with the Linux community. You can find the CE Workgroup's website at www.linuxfoundation.org/collaborate/workgroups/celf.

One of the major activities of the CE Workgroup is the *Long-Term Support Initiative* (LTSI). LTSI's goal is to create and maintain a stable Linux kernel tree that is supported with relevant patches for about 2 to 3 years, which is the typical life of consumer electronic products such as smartphones, game consoles, and TV sets. LTSI details are published on http://ltsi.linuxfoundation.org.

1.6 Summary

Embedded Linux is already powering many devices and services you use on a daily basis. Generally unnoticed, it directs data traffic through Internet routers, puts high-definition pictures on TV screens, guides travelers inside navigation devices, measures energy consumption in smart meters, collects traffic information in roadside sensors, and much more. Linux and open source are powering the Internet of Things from connected devices to networking infrastructure and data processing centers. This chapter set the stage for the material to come, covering the following topics:

- Definition of embedded systems from the engineer's perspective and the broad set of responsibilities associated with taking an embedded product from design to production
- Technology developments contributing to the rapid adoption of Linux for embedded devices
- Overview of the embedded Linux landscape
- Challenges associated with building and maintaining an operating system stack
- Importance of open source licensing for embedded projects

- Several organizations and standards relevant to embedded Linux

1.7 References

Apache License, www.apache.org/licenses

The Apache Software Foundation, www.apache.org

Eclipse Foundation, www.eclipse.org

GNU GPL License, www.gnu.org/licenses/gpl.html

The Linux Foundation, www.linuxfoundation.org

Linux Standard Base, www.linuxfoundation.org/collaborate/workgroups/lsb

2. The Yocto Project

In This Chapter

Yocto is the prefix of the smallest fraction for units of measurements specified by the International System of Units (abbreviated *SI* from French *Le Système International d'Unités*). It gives the name to the Yocto Project, a comprehensive suite of tools, templates, and resources to build custom Linux distributions for embedded devices. To say that the name is an understatement is an understatement in itself.

In this chapter we start *in medias res* with setting up the OpenEmbedded build system, provided by the Yocto Project in the Poky reference distribution, and building our first Linux OS stack relying entirely on the blueprint that Poky provides by default. The tasks we perform in this chapter set the stage for the coming chapters in which we analyze the various aspects of the Yocto Project from the Poky workflow to the OpenEmbedded build system, including the build engine BitBake, to customizing operating system stacks to board support packages and the application development toolkit, and much more.

We conclude with the relationship of the Yocto Project with OpenEmbedded and a glossary of Yocto Project terms.

2.1 Jumpstarting Your First Yocto Project Build

Getting your hands dirty—or learning by doing—is undoubtedly the best way to acquire new skills. Consequently, we start by building our first Linux OS stack for use with the QEMU (short for Quick Emulator, a generic open source machine emulator for different CPU architectures).

You learn how to prepare your computer to become a Yocto Project development host, obtain and install the build system, set up and configure a build environment, launch and monitor the build process, and finally verify the build result by booting your newly built Linux OS stack in the QEMU emulator.

The following section outlines the hardware and software prerequisites for a Yocto Project build host. If you do not wish to set up a build host right away, the Yocto Project provides a Build Appliance, a preconfigured system in a virtual machine, that lets you try out the Yocto Project tools without installing any software. Just jump ahead to Section 2.1.7, which outlines how to experiment with the Yocto Project Build Appliance.

2.1.1 Prerequisites

You probably have guessed it: to build a Linux system with the Yocto Project tools, you need a build host running Linux.

Hardware Requirements

Despite their capability to build Linux OS stacks, the Yocto Project tools require a build host with an x86 architecture CPU. Both 32-bit and 64-bit CPUs are supported. A system with a 64-bit CPU is preferred for throughput reasons. The Yocto Project's build system makes use of parallel processing whenever possible. Therefore, a build host with multiple CPUs or a multicore CPU significantly reduces build time. Of course, CPU clock speed also has an impact on how quickly packages can be built.

Memory is also an important factor. BitBake, the Yocto Project build engine, parses thousands of recipes and creates a cache with build dependencies. Furthermore, the compilers require memory for data structures and more. The tools do not run on a system with less than 1 GB of RAM; 4 GB or more is recommended.

Disk space is another consideration. A full build process, which creates an image with a graphical user interface (GUI) based on X11 currently consumes about 50 GB of disk space. If, in the future, you would like to build for more architectures and/or add more packages to your builds, you will require additional space. It is recommended that the hard disk of your system has at least 100 GB of free space. Since regular hard disks with large capacity have become quite affordable, we recommend that you get one with 500 GB or more to host all your Yocto Project build environments.

Since build systems read a lot of data from the disk and write large amounts of build output data to it, disks with higher I/O throughput rates can also significantly accelerate the build process. Using a solid-state disk can further improve your build experience, but these devices, in particular with larger capacity, are substantially higher in cost than regular disks with spinning platters. Whether you are using conventional hard drives or solid-state disks, additional performance gains can be realized with a redundant array of independent disks (RAID) setup, such as RAID 0.

Internet Connection

The OpenEmbedded build system that you obtain from the project's website contains only the build system itself—BitBake and the metadata that guide it. It does not contain any source packages for the software it is going to build. These are automatically downloaded as needed while a build is running. Therefore, you need a live connection to the Internet, preferably a high-speed connection.

Of course, the downloaded source packages are stored on your system and reused for future builds. You are also able to download all source packages upfront and build them later offline without a connection to the Internet.

Software Requirements

First of all, you will need a recent Linux distribution. The Yocto Project team is continually qualifying more and more distributions with each release. Using the previous to current release of one of the following distributions typically works without any issues:

- CentOS
- Fedora
- openSUSE
- Ubuntu

In general, both the 32-bit and the 64-bit variants have been verified; however, it is recommended that you use the 64-bit version if your hardware supports it. You can find a detailed list of all supported distributions in the *Yocto Project Reference Manual* located at www.yoctoproject.org/docs/current/ref-manual/ref-manual.html.

In addition to the Linux distribution, you need to install a series of software packages for the build system to run. We cover the installation in Section 2.1.3.

2.1.2 Obtaining the Yocto Project Tools

There are several ways for you to obtain the Yocto Project tools, or more precisely, the Yocto Project reference distribution, Poky:

- Download the current release from the Yocto Project website.
- Download the current release or previously released versions from the release repository.
- Download a recent nightly build from the Autobuilder repository.
- Clone the current development branch or other branches from the Poky Git repository hosted by the Yocto Project Git repository server.

The Yocto Project team releases a new major version of the build system every 6 months, in the April–May and October–November timeframes. All released versions of the Yocto Project tools have undergone multiple rounds of quality assurance and testing. They are stable and are accompanied with release notes and an updated documentation set describing the features. For Yocto Project novices, we recommend using the recent stable release.

Minor version releases that resolve issues but do not add any new features are provided as necessary between the 6-month release cycles. Since there are no new features, the documentation typically does not change with the minor releases.

Previous major and minor releases are archived and still available for download from the download repository. Sometimes new major releases introduce a new layer structure, new configuration files, and/or new settings in configuration files. Hence, migrating an existing build environment to the newer release may require migration effort. Staying with a previous release allows you to postpone or entirely put off migration.

Nightly builds track the current development status of the codebase in the Yocto Project

Git repository. These builds have undergone basic quality assurance and Autobuilder testing. They are not tested as rigorously as the regular major and minor releases, but you get at least some confidence that the core functionality is operational.

Cloning the current development branch (*master* branch) from the Poky Git repository gives you direct access to the current state of the development effort. Modifications to this branch have not undergone any testing other than the tests the developers performed before signing off on their submissions. While the quality is generally high and any serious breakage of core functionality typically gets detected within a short period of time after a developer checked in a change, there is a good chance that the system may not work as expected. Unless you are directly involved in Yocto Project development, there is no immediate need to directly work with the master branch.

Besides the master branch, the Poky Git repository also contains milestone branches, development branches for the various versions, and a long list of tags referencing particular revisions of the various branches.

In the chapters to come, we outline how to download the Yocto Project releases from the various locations. We also explore the Yocto Project Git repositories for Poky, board support packages, the Linux kernel, and more in detail.

Downloading the Current Poky Release

Navigate to https://www.yoctoproject.org/downloads and click on the latest release of Poky. This URL directs you to a detailed download site with links to various download servers and mirrors. The site also contains the release information and an errata.

Downloading the release places a compressed archive of the Poky reference distribution named `poky-<codename>-<release>.tar.bz2` on your system.

2.1.3 Setting Up the Build Host

Setting up your build host requires the installation of additional software packages. All of the four mainstream Linux distributions have those packages readily available in their package repositories. However, they differ in what packages are preinstalled as part of the distribution's default configuration.

After installing the additional packages, you need to unpack the Poky tarball, which includes all the necessary configuration data, recipes, convenience scripts, and BitBake.

BitBake requires Python with a major version of 2.6 or 2.7. BitBake currently does not support the new Python 3, which introduced changes to the language syntax and new libraries breaking backwards compatibility.

Installing Additional Software Packages

What command to use and what additional packages to install depends on the Linux distribution you installed on your build host.

To install the necessary packages on a CentOS build host, use the command in Listing 2-1.

Listing 2-1 **CentOS**

Click here to view code image

```
user@centos:~$ sudo yum install gawk make wget tar bzip2 gzip \
python unzip perl patch diffutils diffstat git cpp gcc gcc-c++ \
glibc-devel texinfo chrpath socat perl-Data-Dumper \
perl-Text-ParseWords perl-Thread-Queue SDL-devel xterm
```

For setup on a Fedora build host, execute the command in Listing 2-2.

Listing 2-2 **Fedora**

Click here to view code image

```
user@fedora:~$ sudo dnf install gawk make wget tar bzip2 gzip python \
unzip perl patch diffutils diffstat git cpp gcc gcc-c++ glibc-devel \
texinfo chrpath ccache perl-Data-Dumper perl-Text-ParseWords \
perl-Thread-Queue socat findutils which SDL-devel xterm
```

Listing 2-3 shows the installation command for an openSUSE build host.

Listing 2-3 **openSUSE**

Click here to view code image

```
user@opensuse:~$ sudo zypper install python gcc gcc-c++ git chrpath \
make wget python-xml diffstat makeinfo python-curses patch socat \
libSDL-devel xterm
```

On an Ubuntu build, run the command in Listing 2-4.

Listing 2-4 **Ubuntu**

Click here to view code image

```
user@ubuntu:~$ sudo apt-get install gawk wget git-core diffstat \
unzip texinfo gcc-multilib build-essential chrpath socat \
libsdl1.2-dev xterm
```

After a successful installation, you may want to verify that the correct version of Python has been installed: `python --version`. The output should show a major version number of 2.6 or 2.7.

Installing Poky

Installing Poky merely requires unpacking the compressed tarball you downloaded from the Yocto Project website earlier. We recommend that you create a subdirectory in your home directory for all things related to your Yocto Project builds. Listing 2-5 shows the necessary steps.

Listing 2-5 **Installing Poky**

Click here to view code image

```
user@buildhost:~$ mkdir ~/yocto
user@buildhost:~$ cd ~/yocto
```

```
user@buildhost:~$ tar xvfj <downloadpath>/poky-<codename>-release.tar.bz2
```

Now your build system is ready for setting up a build environment and creating your first Linux OS stack.

2.1.4 Configuring a Build Environment

Poky provides the script `oe-init-build-env` to create a new build environment. The script sets up the build environment's directory structure and initializes the core set of configuration files. It also sets a series of *shell variables* needed by the build system. You do not directly execute the `oe-init-build-env` script but use the `source` command to export the shell variable settings to the current shell:

Click here to view code image

```
$ source <pokypath>/oe-init-build-env <builddir>
```

Executing the command creates a new build environment in the current directory with the name provided by the parameter `<builddir>`. You may omit that parameter, and then the script uses the default `build`. After setting up the build environment, the script changes directory to the build directory.

Use the script in the form of Listing 2-6 to create a new build environment as well as to initialize an existing build environment previously created. When creating a new build environment the script provides you with some instructions.

Listing 2-6 **New Build Environment Setup**

Click here to view code image

```
You had no conf/local.conf file. This configuration file has therefore been
created for you with some default values. You may wish to edit it to use a
different MACHINE (target hardware) or enable parallel build options to take
advantage of multiple cores, for example. See the file for more information,
as
common configuration options are commented.

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

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/

You had no conf/bblayers.conf file. The configuration file has been created
for you with some default values. To add additional metadata layers into your
configuration, please add entries to this file.

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

For more information about OpenEmbedded see their website:
    http://www.openembedded.org/

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

You can now run 'bitbake <target>'
```

```
Common targets are:
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-toolchain-sdk
    adt-installer
    meta-ide-support

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

Inside the newly created build environment, the script added the directory `conf` and placed the two configuration files in it: `bblayers.conf` and `local.conf`. We explain `bblayers.conf` in detail in Chapter 3, "OpenEmbedded Build System." For now we look only at `local.conf`, which is the master configuration file of our build environment.

In `local.conf`, various variables are set that influence how BitBake builds your custom Linux OS stack. You can modify the settings and also add new settings to the file to override settings that are made in other configuration files. We explain this inheritance and how to use it with various examples throughout this book. For our first build, we focus on a few settings and accept the defaults for the remaining ones. If you open the `local.conf` file in a text editor, you find the variable settings shown in Listing 2-7 (among many others, which along with comment lines we have removed from this listing).

Listing 2-7 **`conf/local.conf`**

Click here to view code image

```
BB_NUMBER_THREADS ?= "${@bb.utils.cpu_count()}"
PARALLEL_MAKE ?= "-j ${@bb.utils.cpu_count()}"
MACHINE ??= "qemux86"
DL_DIR ?= "${TOPDIR}/downloads"
SSTATE_DIR ?= "${TOPDIR}/sstate-cache"
TMP_DIR = "${TOPDIR}/tmp"
```

Lines starting with the hash mark (#) are comments. If a hash mark precedes a line with a variable setting, you need to remove the hash mark for the settings to become active. The values shown are the default values. BitBake uses those values even if you do not enable them explicitly. The variable settings shown in Listing 2-7 are the ones that you typically want to change after creating a new build environment. They are described in Table 2-1.

Variable	Default Value	Description
BB_NUMBER_THREADS	${@bb.utils.cpu_count()}	Number of parallel BitBake tasks
PARALLEL_MAKE	-j ${@bb.utils.cpu_count()}	Number of parallel Make processes
MACHINE	qemux86	Target machine
DL_DIR	${TOPDIR}/downloads	Directory where source downloads are placed
SSTATE_DIR	${TOPDIR}/sstate_cache	Directory for shared state cache files
TMP_DIR	${TOPDIR}/tmp	Directory for build output

Table 2-1 **Configuration Variables**

The default value for the two *parallelism options* `BB_NUMBER_THREADS` and `PARALLEL_MAKE` is automatically computed on the basis of the number of CPU cores in the system using all the available cores. You can set the values to less than the cores in your system to limit the load. Using a larger number than the number of physical cores is possible but does not speed up the build process. BitBake and Make spawn more threads accordingly, but they run only if there are CPU cores available. Never forget the quotes around the variable settings. Also note that for `PARALLEL_MAKE`, you have to include the `-j`, such as `"-j 4"` because this value is passed to the `make` command verbatim.

Setting the `MACHINE` variable selects the target machine type for which BitBake builds your Linux OS stack. Poky provides a series of standard machines for QEMU and a few actual hardware board target machines. Board support packages (BSPs) can provide additional target machines. For our first build, we choose `qemux86`, an emulated target machine with an x86 CPU.

The variable `DL_DIR` tells BitBake where to place the source downloads. The default setting places the files in the directory `downloads` beneath the top directory of your build environment. The variable `TOPDIR` contains the full (absolute) path to the build environment. Source downloads can be shared among multiple build environments. If BitBake detects that a source download is already available in the download directory, it does not download it again. Therefore, we recommend that you set the `DL_DIR` variable to point to a directory path outside of the build environment. When you no longer need a particular build environment, you can easily delete it without deleting all the source file downloads.

The same holds true for the `SSTATE_DIR` variable, which contains the path to the *shared state cache*. The OpenEmbedded build system produces a lot of intermediate output when processing the many tasks entailed in building the packages comprising the Linux OS stack. Similar to the source downloads, the intermediate output can be reused for future builds and shared between multiple build environments to speed up the build process. By default, the configuration places the shared state cache directory beneath the

build environment's top directory. We suggest that you change the setting to a path outside the build environment.

The variable `TMP_DIR` contains the path to the directory where BitBake performs all the build work and stores the build output. Since the output stored in this directory is very specific to your build environment, it makes sense to leave it as a subdirectory to the build environment. The amount of data stored in this directory can eventually consume many gigabytes of hard disk space because it contains extracted source downloads, cross-compilation toolchains, compilation output, and images for kernel and root file systems for your target machines and more.

To conserve disk space during a build, you can add

```
INHERIT += rm_work
```

which instructs BitBake to delete the work directory for building packages after the package has been built.

2.1.5 Launching the Build

To launch a build, invoke BitBake from the top-level directory of your build environment specifying a build target:

```
$ bitbake <build-target>
```

We go into detail about what build targets are and how to use them to control you build output in the following chapters. For our first build we use a build target that creates an entire Linux OS stack with a GUI. From the top-level directory of the build environment you created and configured during the previous sections, execute the following:

```
$ bitbake core-image-sato
```

The `core-image-sato` target creates a root file system image with a user interface for mobile devices. Depending on your build hardware and the speed of your Internet connection for downloading the source files, the build can take anywhere from one to several hours.

You may also instruct BitBake to first download all the sources without building. You can do this with


```
$ bitbake -c fetchall core-image-sato
```

After the download completes, you can disconnect your build system from the Internet and run the build offline at a later point in time.

BitBake typically immediately aborts a build process if it encounters an error condition from which it cannot recover. However, you can instruct BitBake to continue building even if it encounters an error condition as long as there are tasks left that are not impeded by the error:

```
$ bitbake -k core-image-sato
```

The `-k` option tells BitBake to continue building until tasks that are not dependent on the error condition are addressed.

2.1.6 Verifying the Build Results

Since our target machine is an emulated system, we can verify our build result by launching the QEMU emulator. For that purpose, Poky provides a convenient script that prepares the QEMU execution environment and launches the emulator with the proper kernel and root file system images:

```
$ runqemu qemux86
```

In its simplest form, the `runqemu` script is invoked with the machine target name. It then automatically finds the proper kernel and root file system images for the target in the build output. You have to enter your system administrator (or *sudo*) password for the script to set up the virtual network interface. Figure 2-1 shows the running system.

Figure 2-1 QEMU with `core-image-sato` target

You can terminate your QEMU virtual machine by clicking on the *Shutdown* button on the *Utilities* screen. This properly shuts down the system by running through the shutdown sequence. Alternatively, you can simply type *Ctrl-C* in the terminal where you launched QEMU.

2.1.7 Yocto Project Build Appliance

If you simply would like to try out the Yocto Project and Poky without setting up a Linux build host, you can use the *Yocto Project Build Appliance*. The Build Appliance is a complete Yocto Project build host, including a Linux OS with the software packages required by the OpenEmbedded build system and Poky installed, bundled as a virtual machine image. It even already includes all the source package downloads, speeding up your first build and allowing you to build offline without a network connection.

The Build Appliance download is located on the Yocto Project website at https://www.yoctoproject.org/download/build-appliance-0. The Build Appliance is provided as a compressed ZIP archive that you need to unpack on your system after downloading it.

To utilize the Build Appliance, you need either VMWare Player or VMWare Workstation installed on your computer. You can obtain either one of them matching the operating system on your computer from the download section of VMWare's website at www.vmware.com. Follow the installation instructions provided by VMWare.

Once you have installed VMWare Player or VMWare Workstation, the Build Appliance manual at https://www.yoctoproject.org/documentation/build-appliance-manual provides detailed instructions on how to configure the virtual machine and boot the Build Appliance.

Booting the Build Appliance directly launches the Hob GUI for BitBake, as shown in Figure 2-2.

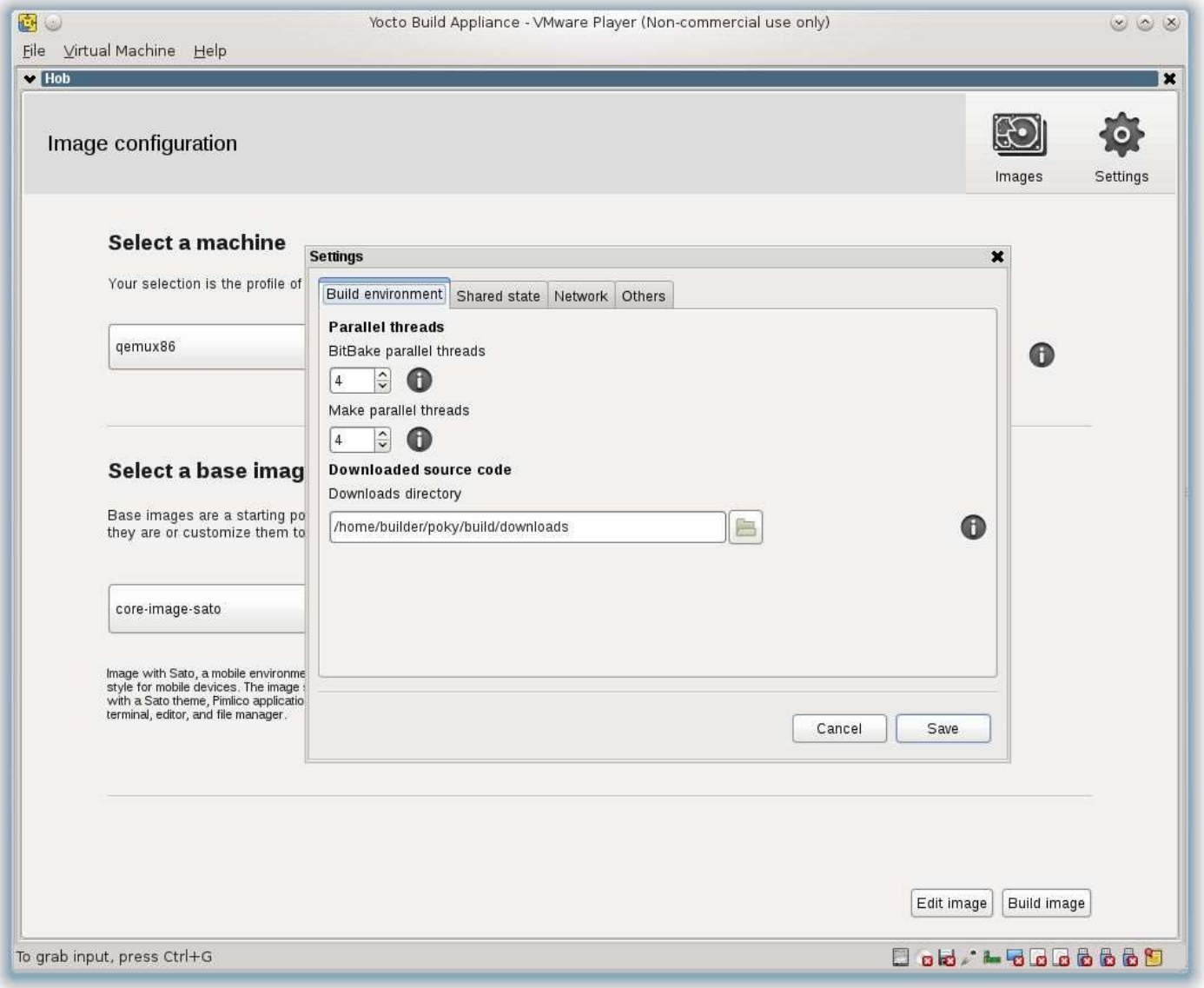

Figure 2-2 Yocto Project Build Appliance

Select *qemux86* from the drop-down box for the machine and *core-image-sato* for the base image. Then start the build. Depending on your host system and the virtual machine configuration, it may take multiple hours to build the image. You can observe the build process from the *Log* screen of Hob. The *Log* screen shows the packages to be built in a run-queue split up into the individual tasks. Currently running tasks are highlighted.

After the build completes, you can launch your image with the QEMU emulator directly from Hob.

2.2 The Yocto Project Family

The Yocto Project is not just a single open source project but combines multiple projects under one umbrella. You have already encountered the most prominent members of this family of projects: the OpenEmbedded build system, which includes BitBake and OpenEmbedded Core, and Poky, the Yocto Project reference distribution.

Essentially, all the members of the family support the OpenEmbedded build system. The Yocto Project team maintains the build system together with the OpenEmbedded Project, a separate organization. New functionality is added to the subprojects as the build system evolves.

Table 2-2 provides an overview of the subprojects maintained as part of the Yocto Project.

Internal Project	Description
Application Development Toolkit (ADT)	ADT provides a complete development environment for user space applications to run on operating system stacks built by Poky. An ADT comprises cross-development toolchains, the QEMU emulator, Linux kernel, and root file system images. Poky creates and packages an ADT directly from within the build environment using its configuration settings.
AutoBuilder	AutoBuilder automates builds through integration of the build system with Buildbot. The Yocto Project QA team uses AutoBuilder for continuous integration and regression testing with a set of standard build targets.
BitBake	BitBake is the build engine of the OpenEmbedded build system. BitBake is a build tool similar to Make or Ant with focus on building software packages and operating system stacks.
Build Appliance	The Build Appliance is a fully self-contained Poky build system on Ubuntu Linux packaged as a VMWare virtual machine image. It is a convenient way to try out Poky without installing a Linux build host and setting up Poky on it.
Cross-Prelink	Memory address locations of shared libraries are typically computed during runtime when the library is loaded into memory for the first time. Every time a program relying on a shared library is run, the loader has to locate the library in memory. Because libraries can move around in memory, this task causes a performance penalty that increases with the number of shared libraries that need to be resolved. Prelinking reduces this overhead by performing the dynamic linking of an executable in advance by precalculating the addresses. Cross-prelinking performs this process as part of the cross-development toolchain by emulating a runtime linker.

Eclipse IDE Plugin	The Yocto Project plugin for the Eclipse IDE integrates ADT into Eclipse, providing a cross-development workflow for user space applications. The plugin provides integration for cross-toolchains as well as for root file systems to be utilized with emulated and/or hardware targets.
EGLIBC	EGLIBC is the embedded version of the GNU C Library (GLIBC). It offers the same APIs as GLIBC and tries to be source and binary compatible with it but is optimized for embedded systems. Optimization includes reduced footprint, improved support for cross-build and cross-testing, and configurable components. Although EGLIBC has been merged into GLIBC and reached end-of-life, we mention it here to provide context and history.
Hob	Hob is a GUI for BitBake. Hob allows configuration of a build environment, package selection, and image configuration from a convenient GUI rather than editing text files. Build processes can be directly launched and monitored from Hob. Hob is a layer contained in OpenEmbedded Core. Eventually, the Yocto Project team will discontinue Hob in favor of Toaster.
Matchbox	Matchbox is an open source window manager based on the X Window System targeted for embedded devices. It distinguishes itself from traditional desktop window managers in that it shows only one window at a time filling the entire screen. That feature makes it suitable for devices with smaller form factors.
OpenEmbedded Core	OpenEmbedded Core (OE Core) is the core collection of metadata in the OpenEmbedded build system. It consists of a collection of BitBake layers and classes and integration and utility scripts that are shared among OpenEmbedded-derived systems. OE Core is co-maintained by the OpenEmbedded Project and the Yocto Project.

Poky	Poky is the reference distribution of the Yocto Project. It provides a set of blueprints for preconfigured embedded Linux OS stacks as working examples. These blueprints can be used to bootstrap actual system development.
Pseudo	Building software packages frequently requires operations such as installing files into a system's root file system, changing file ownership or access permissions, and creating device nodes to be carried out as the system administrator. Pseudo is an application that provides a virtualized environment allowing such operations to succeed as though the user had system administrator privileges even if she is an ordinary user.
Swabber	The majority of software packages are configured to be built on a native system. Building them in cross-development environments frequently requires configuration changes and bears the inherent risk that a build uses components from the build host rather than from the cross-build environment. For instance, a path in a make file points to a file on the host system rather than the cross-build environment. Such *host pollution* of a cross-build can lead to hard-to-detect system failures not only during build time but also during runtime. Swabber provides a mechanism for detecting cross-build access beyond the boundaries of the build sandbox.
Toaster	Toaster is the new GUI for BitBake and the build system. It is web based and can be deployed as a distributed build service with remote access.

Table 2-2 **Yocto Project Family**

Although there is tight integration of the subprojects within the Yocto Project, the developers ensure that there are no cross dependencies and that the subprojects are interoperable and can also be used independently without the build system.

2.3 A Little Bit of History

Both OpenEmbedded and the Yocto Project have their roots in the OpenZaurus project, an open source project striving to improve the code of the first Linux-based personal digital assistant, the Sharp Zaurus SL-5000D. The SL-5000D, which first shipped in 2001, was a device targeted to developers, and Sharp provided the necessary tools to modify and update the ROM code of the device. At first, the project focused on repackaging the existing ROM code to make it more developer-friendly. Over time, the project evolved, and the original Sharp code was entirely replaced by a Debian-based Linux distribution built from source. It quickly outgrew its build system, making it necessary for the project to create a new device and distribution-independent build system. The OpenEmbedded project was born.

2.3.1 OpenEmbedded

The OpenEmbedded project debuted in 2003 by combining the efforts of the OpenZaurus project with contributions from other embedded Linux projects with similar goals, such as the Familiar Linux and OpenSIMpad projects.

The OpenEmbedded Project maintained the build system and the metadata that described how to build the software packages and assemble the operating system images as a common codebase. The number of packages added to the metadata inventory quickly grew to more than 2,100 recipes building over 5,000 packages.

In 2005, the project team decided to split the project into the BitBake build system and the OpenEmbedded metadata.

OpenEmbedded got support from various Linux distributions using it as their build system. Among them are the Ångström Distribution, Openmoko, WebOS, and others. Commercial entities adopted the system for their product offerings, among them MontaVista Software and OpenedHand, the startup that developed the Poky Linux distribution.

2.3.2 BitBake

BitBake, the build engine at the core of OpenEmbedded and the Yocto Project's Poky reference distribution, is derived from Portage, the build and package management system of Gentoo Linux. Portage comprises two components:

- **ebuild** is the actual build system that takes care of building software packages from source code and installing them.
- **emerge** is an interface to ebuild and a tool to manage ebuild package repositories, resolving dependencies and more.

All Portage tools are written in Python. BitBake evolved from Portage by extending it for building software packages with native and cross-development toolchains, supporting multiple package management systems and other functionality necessary for cross-building.

BitBake uses the same metadata syntax as the Portage build scripts but introduced new features such as an inheritance mechanism supported by classes, appending recipes, and global configuration files, among others.

2.3.3 Poky Linux

OpenEmbedded significantly simplified building Linux OS stacks for, but not limited to, embedded devices. However, it remained a challenge with quite a steep learning curve to modify and adapt the system to create different distributions and port the system to new hardware.

The software startup company OpenedHand originally developed Poky Linux, a versatile development platform as well as a Linux distribution for mobile devices, for internal use. Poky Linux provided the test platform for the company's Matchbox window manager for embedded devices. Matchbox is most notably used by the Nokia 770 and

N800 tablet devices, Openmoko's Neo1973, and the One Laptop Per Child (OLPC) project's XO laptop computer.

Built with OpenEmbedded, the Poky Linux distribution provided a more intuitive way to configure operating system images for target devices. It also offered several blueprints for target device images that were easy to adapt and modify. Since Poky Linux was open source, it was quickly adopted by others to build embedded devices.

Intel Corporation acquired OpenedHand in 2008 with the goal to further develop Poky Linux as a universal distribution for embedded devices.

2.3.4 The Yocto Project

To build out Poky Linux to support many different architectures and hardware platforms, Intel was looking for other commercial entities—particularly other semiconductor manufacturers and embedded Linux companies—to support the project and contribute to it. As Intel is a dominant player in the chip market and has substantial resources, it proved difficult for Intel to get its competition and other companies to support it in its efforts to improve Poky Linux.

In 2010, Intel approached the Linux Foundation with the idea to create a collaborative project under the auspices of the Foundation with neutral stewardship. That effort would also include the open source community, particularly the OpenEmbedded project.

The Linux Foundation publicly announced the launch of the Yocto Project on October 26, 2010. On March 1, 2011, the Linux Foundation announced the alignment of the Yocto Project technology with OpenEmbedded and the support of multiple corporate collaborators to the project. This announcement was followed by another press release on April 6, 2011, communicating the formation of the Yocto Project Steering Group and the first Yocto Project software release.

2.3.5 The OpenEmbedded and Yocto Project Relationship

The technology alignment between OpenEmbedded and the Yocto Project brought several major improvements to both projects:

- **Aligned Development**: A common problem among open source projects is *fragmentation*: two projects with the same roots and similar goals fork and grow apart. Resources are divided and ultimately efforts are duplicated to provide similar functionality in both branches. Eventually, users and supporters are forced to make a decision between the two efforts. The tight alignment of OpenEmbedded and the Yocto Project ensures that users can get the benefits of both projects.
- **BitBake Metadata Layers**: Metadata layers enable logical grouping of recipes and configuration files into structures that can easily be included in and migrated to different build environments. Metadata layers also simplify dependency management, which is a complex task when building operating system stacks.
- **OpenEmbedded Core Metadata Layer**: The OpenEmbedded and Yocto Project development teams agreed to create a common metadata layer shared between the two projects and containing all the base recipes and configuration settings. Each

project then adds additional metadata layers according to its goals.

Despite the close collaboration between OpenEmbedded and the Yocto Project, the two projects are separate entities. Both are open source projects, and both are supported by a community of open source developers as well as commercial entities.

OpenEmbedded focuses on cutting-edge technology, recipes, and a large set of board support packages for different hardware platforms. The Yocto Project focuses on the build system itself and tooling for cross-development. The goal of the Yocto Project is to provide powerful yet easy-to-use and well-tested tools together with a core set of metadata to jumpstart embedded system development. Additional board support packages and other components are offered through OpenEmbedded and the Yocto Project ecosystem.

The OpenEmbedded Project also maintains a layer index, which is a searchable database of layers, recipes, and machines. Looking for a recipe to build a particular open source package? Enter the name into the layer index, and chances are that somebody has already created a recipe for it.

2.4 Yocto Project Terms

Table 2-3 defines a set of terms commonly used in conjunction with and throughout the Yocto Project. Throughout this book, we use these terms consistently with their definitions provided here.

Term	Description
Append file	An append file extends an existing recipe. BitBake verbatim appends the contents of an append file to the corresponding recipe, creating a single file before parsing it. Variables in an append file can override the same variables defined in the corresponding recipe. Append files use the `bbappend` extension.
BitBake	The build engine in the OpenEmbedded build system, BitBake is a task executor and scheduler. Its inputs are metadata files such as configuration files and recipes through which BitBake processing is controlled.
Board support package (BSP)	Documentation, binaries, code, and other implementation-specific support data in the BSP enable a given operating system to run on a particular target hardware platform. Sometimes a BSP also contains complete root file systems and a cross-development environment to create application programs running on the target hardware platform.
Class	In BitBake terminology, a class is a metadata file providing logic encapsulation and a basic inheritance mechanism allowing commonly used patterns to be defined once and used with many recipes. BitBake class files use the `bbclass` extension.
Configuration file	Configuration files are BitBake metadata files providing global definition and settings for variables that affect the build process.
Cross-development toolchain	A cross-development toolchain is a collection of software development tools allowing software development for target systems employing a different architecture than the development host. Architecture in this context refers to different CPU instruction sets (for instance, ARM, MIPS, PowerPC, x86) as well as different bit sizes (for instance, 8, 16, 32, and 64 bit). Typically, a cross-development toolchain includes one or more language compilers, an assembler, a linker, and often debuggers, emulators, and other tools that are specific to the target architecture.

Image	A binary file, often compressed, an image contains a bootloader, an operating system kernel, and a root file system to be copied to a storage medium from which the target system can boot and run. The term *image* is also used to mean just an operating system kernel (kernel image) or the root file system (root file system image).
Layer	In BitBake terminology, a layer is a collection of metadata (configuration files, recipes, etc.) organized into a file and directory structure. BitBake can include layers to extend its functionality. Yocto Project BSPs are provided as layers.
Metadata	In BitBake terminology, metadata includes all files that instruct BitBake how to execute build processes. BitBake metadata includes classes, recipes (with append files), and configuration files.
OpenEmbedded Core (OE Core)	A core set of metadata in the OpenEmbedded build system that is shared between OpenEmbedded and the Yocto Project, OE Core is a BitBake layer co-maintained by the OpenEmbedded Project and the Yocto Project.
Package	A package is a software bundle containing executable binaries, libraries, documentation, configuration information, and other files following a specific format that an operating system's package management system can install or uninstall. Packages commonly also include information on dependencies on and incompatibilities with other software packages that the package management system can use to automatically resolve and/or inform the user about. The Yocto Project also uses the term *package* to mean the recipes and other metadata used to build the respective software bundle. Dependent on the context, the term then refers either to the actual software bundle or to the metadata that builds the software bundle.
Package management system	A package management system is a collection of software tools automating the process of installing, upgrading, configuring, and removing software packages for a computer's operating system. It typically maintains a database of the installed software on the computer, including version information, dependencies, incompatibilities, and more, to prevent system faults through software mismatches and missing prerequisites.
Poky	Poky is the Yocto Project's reference distribution as well as the name of the default Linux distribution created by the build system. The Poky download package includes the OpenEmbedded build system as well as additional metadata for creating a sample embedded distribution called Poky.
Recipe	A recipe is a metadata file containing directives for BitBake on how to build a particular software package. Through its directives, a recipe describes from where to obtain the source code, what patches to apply and how to apply them, how to build the binaries and associated files, how to install the build results on a target system, how to create the packaged software bundle, and much more. Recipes also describe dependencies during build and runtime on other software packages, hence creating a logical hierarchy of the pieces required for the build process. Recipes use the `bb` file extension.
Task	BitBake recipes may contain executable metadata, or code, that BitBake executes during the build process. Execution steps can be grouped into metadata functions. A metadata function can be declared as a task by inserting it into the BitBake task list.
Upstream	In software development, particularly in open source, *upstream* references the direction to the originators, that is, the original authors or maintainers, of the software. Commonly, the term is used as a qualifier, such as *upstream repository* and *upstream patch*.

Table 2-3 **Yocto Project Terms**

2.5 Summary

The Yocto Project is a family of projects related to embedded Linux software development. At its core is the OpenEmbedded build system and the Poky reference distribution. Originally developed by OpenedHand as Poky Linux, Poky evolved into the Yocto Project, a collaborative project under the auspices of the Linux Foundation. Supported by corporations and independent software developers, it aligned its technology with OpenEmbedded to form a broad community delivering state-of-the-art tools for developing embedded Linux systems.

Getting started with the Yocto Project is as simple as downloading the Build Appliance and booting it from the VMWare virtual machine manager. While the Build Appliance is not recommended for serious development, it offers a good introduction to the OpenEmbedded build system without the need to set up a Linux build host.

Installing a Linux build host for use with Poky requires a few more steps but avoids the overhead and performance impact of the virtual machine.

2.6 References

The Linux Foundation, *Linux Foundation and Consumer Electronics Linux Forum to Merge*, www.linuxfoundation.org/news-media/announcements/2010/10/linux-foundation-and-consumer-electronics-linux-forum-merge

The Linux Foundation, *Yocto Project Aligns Technology with OpenEmbedded and Gains Corporate Collaborators*, www.linuxfoundation.org/news-media/announcements/2011/03/yocto-project-aligns-technology-openembedded-and-gains-corporate-co