



C o m m u n i t y   E x p e r i e n c e   D i s t i l l e d

# Learning Embedded Linux Using the Yocto Project

Develop powerful embedded Linux systems with the Yocto Project components

Alexandru Vaduva

[PACKT] open source\*  
PUBLISHING community experience distilled

# Learning Embedded Linux Using the Yocto Project

Develop powerful embedded Linux systems with the  
Yocto Project components

**Alexandru Vaduva**



BIRMINGHAM - MUMBAI

# Learning Embedded Linux Using the Yocto Project

Copyright © 2015 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2015

Production reference: 1240615

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

ISBN 978-1-78439-739-5

[www.packtpub.com](http://www.packtpub.com)

# Credits

**Author**

Alexandru Vaduva

**Project Coordinator**

Nidhi J. Joshi

**Reviewers**

Peter Ducai

Alex Tereschenko

**Proofreader**

Safis Editing

**Commissioning Editor**

Nadeem N. Bagban

**Indexer**

Mariamammal Chettiyyar

**Acquisition Editor**

Harsha Bharwani

**Graphics**

Sheetal Aute

Disha Haria

Jason Monteiro

Abhinash Sahu

**Content Development Editor**

Vaibhav Pawar

**Production Coordinator**

Conidon Miranda

**Technical Editor**

Shivani Kiran Mistry

**Cover Work**

Conidon Miranda

**Copy Editor**

Sonia Michelle Cheema

# About the Author

**Alexandru Vaduva** is an embedded Linux software engineer whose main focus lies in the field of open source software. He has an inquiring mind and also believes that actions speak louder than words. He is a strong supporter of the idea that there is no need to reinvent the wheel, but there is always room for improvement. He has knowledge of C, Yocto, Linux, Bash, and Python, but he is also open to trying new things and testing new technologies.

Alexandru Vaduva has been a reviewer of the book *Embedded Linux Development with Yocto Project*, Packt Publishing, which is a great asset to the Yocto Project community.

# About the Reviewer

**Peter Ducai** has 15 years of experience in the IT industry, including the fields of programming and OS administration. Currently, he works at HP as an automation engineer.

**Alex Tereschenko** is an avid Maker. He believes that computers can do a lot of good when they are interfaced with real-world objects (as opposed to just crunching data in a dusty corner). This drives him in his projects and is also the reason why embedded systems and the Internet of Things are the topics he enjoys the most.



# www.PacktPub.com

## Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit [www.PacktPub.com](http://www.PacktPub.com).

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

## Why subscribe?

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

## Free access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

# Table of Contents

<b>Preface</b>	<b>v</b>
<b>Chapter 1: Introduction</b>	<b>1</b>
<b>Advantages of Linux and open source systems</b>	<b>1</b>
<b>Embedded systems</b>	<b>3</b>
General description	3
Examples	4
<b>Introducing GNU/Linux</b>	<b>6</b>
<b>Introduction to the Yocto Project</b>	<b>9</b>
Buildroot	10
OpenEmbedded	13
<b>Summary</b>	<b>18</b>
<b>Chapter 2: Cross-compiling</b>	<b>19</b>
<b>Introducing toolchains</b>	<b>19</b>
<b>Components of toolchains</b>	<b>21</b>
<b>Delving into C libraries</b>	<b>29</b>
<b>Working with toolchains</b>	<b>32</b>
Advice on robust programming	34
Generating the toolchain	36
<b>The Yocto Project reference</b>	<b>38</b>
<b>Summary</b>	<b>43</b>
<b>Chapter 3: Bootloaders</b>	<b>45</b>
<b>The role of the bootloader</b>	<b>46</b>
<b>Comparing various bootloaders</b>	<b>48</b>
<b>Delving into the bootloader cycle</b>	<b>49</b>
<b>The U-Boot bootloader</b>	<b>51</b>
Booting the U-Boot options	55
Porting U-Boot	56



<b>The Yocto Project</b>	<b>61</b>
<b>Summary</b>	<b>62</b>
<b>Chapter 4: Linux Kernel</b>	<b>63</b>
<b>The role of the Linux kernel</b>	<b>66</b>
<b>Delving into the features of the Linux kernel</b>	<b>67</b>
Memory mapping and management	68
Page cache and page writeback	74
The process address space	74
Process management	76
Process scheduling	77
System calls	78
The virtual file system	79
<b>Interrupts</b>	<b>81</b>
Bottom halves	83
Methods to perform kernel synchronization	85
<b>Timers</b>	<b>86</b>
<b>Linux kernel interaction</b>	<b>87</b>
The development process	87
Kernel porting	89
Community interaction	91
<b>Kernel sources</b>	<b>92</b>
Configuring kernel	93
Compiling and installing the kernel	94
Cross-compiling the Linux kernel	95
<b>Devices and modules</b>	<b>95</b>
<b>Debugging a kernel</b>	<b>99</b>
<b>The Yocto Project reference</b>	<b>100</b>
<b>Summary</b>	<b>104</b>
<b>Chapter 5: The Linux Root Filesystem</b>	<b>105</b>
<b>Interacting with the root filesystem</b>	<b>105</b>
Delving into the filesystem	111
Device drivers	116
Filesystem options	117
<b>Understanding BusyBox</b>	<b>121</b>
<b>Minimal root filesystem</b>	<b>123</b>
<b>The Yocto Project</b>	<b>125</b>
<b>Summary</b>	<b>127</b>

---

<b>Chapter 6: Components of the Yocto Project</b>	<b>129</b>
Poky	129
Eclipse ADT plug-ins	134
Hob and Toaster	138
Autobuilder	139
Lava	139
Wic	140
Summary	141
<b>Chapter 7: ADT Eclipse Plug-ins</b>	<b>143</b>
The Application Development Toolkit	144
Setting up the environment	145
Eclipse IDE	152
QEMU emulator	165
Debugging	165
Profiling and tracing	167
The Yocto Project bitbake commander	170
Summary	171
<b>Chapter 8: Hob, Toaster, and Autobuilder</b>	<b>173</b>
Hob	173
Toaster	186
AutoBuilder	193
Summary	195
<b>Chapter 9: Wic and Other Tools</b>	<b>197</b>
Swabber	198
Wic	202
LAVA	207
Summary	210
<b>Chapter 10: Real-time</b>	<b>211</b>
Understanding GPOS and RTOS	212
PREEMPT_RT	215
Applying the PREEMPT_RT patch	217
The Yocto Project -rt kernel	224
Disadvantages of the PREEMPT_RT patches	226
Linux real-time applications	227
Benchmarking	228
Meta-realtime	229
Summary	231

---

<b>Chapter 11: Security</b>	<b>233</b>
<b>Security in Linux</b>	<b>234</b>
<b>SELinux</b>	<b>234</b>
<b>Grsecurity</b>	<b>237</b>
<b>Security for the Yocto Project</b>	<b>244</b>
<b>Meta-security and meta-selinux</b>	<b>245</b>
Meta-security	246
Meta-selinux	252
<b>Summary</b>	<b>254</b>
<b>Chapter 12: Virtualization</b>	<b>255</b>
<b>Linux virtualization</b>	<b>256</b>
SDN and NFV	256
NFV	257
ETSI NFV	257
SDN	260
OPNFV	261
<b>Virtualization support for the Yocto Project</b>	<b>264</b>
<b>Summary</b>	<b>280</b>
<b>Chapter 13: CGL and LSB</b>	<b>281</b>
<b>Linux Standard Base</b>	<b>282</b>
<b>Carrier grade options</b>	<b>288</b>
Carrier Grade Linux	288
Automotive Grade Linux	291
Carrier Grade Virtualization	292
<b>Specific support for the Yocto Project</b>	<b>293</b>
<b>Summary</b>	<b>302</b>
<b>Index</b>	<b>303</b>

---

# Preface

With regard to the Linux environment today, most of the topics explained in this book are already available and are covered in a fair bit of detail. This book also covers a large variety of information and help in creating many viewpoints. Of course, there are some very good books written on various subjects also presented in this book, and here, you will find references to them. The scope of this book, however, is not to present this information all over again, but instead to make a parallel between the traditional methods of interaction with the embedded development process and the methods used by the Yocto Project.

This book also presents the various challenges that you might encounter in embedded Linux and suggests solutions for them. Although this book is intended for developers who are pretty confident of their basic Yocto and Linux skills and are trying to improve them, I am confident that those of you who have no real experience in this area, could also find some useful information here.

This book has been built around various big subjects, which you will encounter in your embedded Linux journey. Besides this, technical information and a number of exercises are also given to you to ensure that as much information as possible is passed on to you. At the end of this book, you should have a clear picture of the Linux ecosystem.

## What this book covers

*Chapter 1, Introduction*, tries to offer a picture of how an embedded Linux software and hardware architecture looks. It also presents you information on the benefits of Linux and Yocto along with examples. It explains the architecture of the Yocto Project and how it is integrated inside the Linux environment.

*Chapter 2, Cross-compiling*, offers you the definition of a toolchain, its components, and the way in which it can be obtained. After this, information on the Poky repository is given to you and a comparison is made with the components.

*Chapter 3, Bootloaders*, gives you information on a boot sequence, U-Boot bootloader, and how it can be built for a specific board. After this, it gives access to the U-Boot recipe from Poky and shows how it is used.

*Chapter 4, Linux Kernel*, explains the features of the Linux kernel and source code. It gives you information on how to build a kernel source and modules and then moves on to explain the recipes of the Yocto kernel and presents how the same things happen there after that the kernel is booted.

*Chapter 5, The Linux Root Filesystem*, gives you information on the organization of root file system directories and device drivers. It explains the various filesystems, BusyBox, and what a minimal filesystem should contain. It will show you how BusyBox is compiled inside and outside the Yocto Project and how a root filesystem is obtained using Poky.

*Chapter 6, Components of the Yocto Project*, offers an overview of the available components of the Yocto Project, most of which are outside Poky. It provides an introduction and a brief presentation of each component. After this chapter, a bunch of these components are explained in more detail.

*Chapter 7, ADT Eclipse Plug-ins*, shows how to set up the Yocto Project Eclipse IDE, setting it up for cross development and debugging using Qemu, and customizing an image and interacting with different tools.

*Chapter 8, Hob, Toaster, and Autobuilder*, goes through each one of these tools and explain how each one of them can be used, mentioning their benefits as well.

*Chapter 9, Wic and Other Tools*, explains how to use another set of tools, very different from the ones mentioned in the previous chapter.

*Chapter 10, Real-time*, shows the real-time layers of the Yocto Project, their purposes, and added value. Documented information on Preempt-RT, NoHz, userspace RTOS, benchmarking, and other real-time related features are also mentioned.

*Chapter 11, Security*, explains the Yocto Project's security-related layers, their purposes, and the ways in which they could add value to Poky. Here, you will also be given information about SELinux and other applications, such as bastille, buck-security, nmap and so on.

*Chapter 12, Virtualization*, explains the virtualization layers of the Yocto Project, their purposes and the ways in which they could add value to Poky. You will also be given information about virtualization-related packages and initiatives.

*Chapter 13, CGL and LSB*, gives you information on the Carrier Graded Linux (CGL) specifications and requirements as well as the specifications, requirements, and tests of Linux Standard Base (LSB). In the end, a parallel will be made with the support provided by the Yocto Project.

## What you need for this book

Before reading this book, prior knowledge of embedded Linux and Yocto would be helpful, though not mandatory. In this book, a number of exercises are available, and to do them, a basic understanding of the GNU/Linux environment would be useful. Also, some of the exercises are for a specific development board and others involve using Qemu. Owning such a board and previous knowledge of Qemu is a plus, but is not mandatory.

Throughout the book, there are chapters with various exercises that require you to already have knowledge of C language, Python, and Shell Script. It would be useful if the reader has experience in these areas, because they are the core technologies used in most Linux projects available today. I hope this information does not discourage you while reading the content of this book content, and that you enjoy it.

## Who this book is for

The book is targeted at Yocto and Linux enthusiasts who want to build embedded Linux systems and maybe contribute to the community. Background knowledge should include C programming skills, experience with Linux as a development platform, basic understanding of the software development process. If you've previously read *Embedded Linux Development with Yocto Project*, Packt Publishing, it would be a plus as well.

Taking a look at technology trends, Linux is the next big thing. It offers access to cutting-edge open source products and more embedded systems are introduced to mankind every day. The Yocto Project is the best choice for any project that involves interaction with embedded devices due to the fact that it provides a rich set of tools to help you to use most of your energy and resources in your product development, instead of reinventing.

## Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "A maintainers file offers a list of contributors to a particular board support."

A block of code is set as follows:

```
sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu $(lsb_
release -sc) universe"
sudo apt-get update
sudo add-apt-repository "deb http://people.linaro.org/~neil.williams/
lava jessie main"
sudo apt-get update

sudo apt-get install postgresql
sudo apt-get install lava
sudo a2dissite 000-default
sudo a2ensite lava-server.conf
sudo service apache2 restart
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
sudo add-apt-repository "deb http://archive.ubuntu.com/ubuntu $(lsb_
release -sc) universe"
sudo apt-get update
sudo add-apt-repository "deb http://people.linaro.org/~neil.williams/
lava jessie main"
sudo apt-get update


sudo apt-get install postgresql
sudo apt-get install lava
sudo a2dissite 000-default
sudo a2ensite lava-server.conf
sudo service apache2 restart
```




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

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04.1 LTS"
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: " If this warning message appears, press **OK** and move further "

[  Warnings or important notes appear in a box like this. ]

[  Tips and tricks appear like this. ]

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

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

## Errata

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

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

## Questions

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

# 1

## Introduction

In this chapter, you will be presented with the advantages of Linux and open source development. There will be examples of systems running embedded Linux, which a vast number of embedded hardware platforms support. After this, you will be introduced to the architecture and development environment of an embedded Linux system, and, in the end, the Yocto Project, where its Poky build system's properties and purposes are summarized.

### Advantages of Linux and open source systems

Most of the information available in this book, and the examples presented as exercises, have one thing in common: the fact that they are freely available for anyone to access. This book tries to offer guidance to you on how to interact with existing and freely available packages that could help an embedded engineer, such as you, and at the same time, also try to arouse your curiosity to learn more.



More information on open source can be gathered from the **Open Source Initiative (OSI)** at <http://opensource.org/>.

The main advantage of open source is represented by the fact that it permits developers to concentrate more on their products and their added value. Having an open source product offers access to a variety of new possibilities and opportunities, such as reduced costs of licensing, increased skills, and knowledge of a company. The fact that a company uses an open source product that most people have access to, and can understand its working, implies budget savings. The money saved could be used in other departments, such as hardware or acquisitions.

Usually, there is a misconception about open source having little or no control over a product. However, the opposite is true. The open source system, in general, offers full control over software, and we are going to demonstrate this. For any software, your open source project resides on a repository that offers access for everyone to see. Since you're the person in charge of a project, and its administrator as well, you have all the right in the world to accept the contributions of others, which lends them the same right as you, and this basically gives you the freedom to do whatever you like. Of course, there could be someone who is inspired by your project and could do something that is more appreciated by the open source community. However, this is how progress is made, and, to be completely honest, if you are a company, this kind of scenario is almost invalid. Even in this case, this situation does not mean the death of your project, but an opportunity instead. Here, I would like to present the following quote:

*"If you want to build an open source project, you can't let your ego stand in the way. You can't rewrite everybody's patches, you can't second-guess everybody, and you have to give people equal control."*

*– Rasmus Lerdorf*

Allowing access to others, having external help, modifications, debugging, and optimizations performed on your open source software implies a longer life for the product and better quality achieved over time. At the same time, the open source environment offers access to a variety of components that could easily be integrated in your product if there's a requirement for them. This permits a quick development process, lower costs, and also shifts a great deal of the maintenance and development work from your product. Also, it offers the possibility to support a particular component to make sure that it continues to suit your needs. However, in most instances, you would need to take some time and build this component for your product from zero.

This brings us to the next benefit of open source, which involves testing and quality assurance for our product. Besides the lesser amount of work that is needed for testing, it is also possible to choose from a number of options before deciding which components fits best for our product. Also, it is cheaper to use open source software, than buying and evaluating proprietary products. This takes and gives back process, visible in the open source community, is the one that generates products of a higher quality and more mature ones. This quality is even greater than that of other proprietary or closed source similar products. Of course, this is not a generally valid affirmation and only happens for mature and widely used products, but here appears the term community and foundation into play.

In general, open source software is developed with the help of communities of developers and users. This system offers access to a greater support on interaction with the tools directly from developers - the sort of thing that does not happen when working with closed source tools. Also, there is no restriction when you're looking for an answer to your questions, no matter whether you work for a company or not. Being part of the open source community means more than bug fixing, bug reporting, or feature development. It is about the contribution added by the developers, but, at the same time, it offers the possibility for engineers to get recognition outside their working environment, by facing new challenges and trying out new things. It can also be seen as a great motivational factor and a source of inspiration for everyone involved in the process.

Instead of a conclusion, I would also like to present a quote from the person who forms the core of this process, the man who offered us Linux and kept it open source:

*"I think, fundamentally, open source does tend to be more stable software. It's the right way to do things."*

*– Linus Torvalds*

## Embedded systems

Now that the benefits of open source have been introduced to you, I believe we can go through a number of examples of embedded systems, hardware, software, and their components. For starters, embedded devices are available anywhere around us: take a look at your smartphone, car infotainment system, microwave oven, or even your MP3 player. Of course, not all of them qualify to be Linux operating systems, but they all have embedded components that make it possible for them to fulfill their designed functions.

## General description

For Linux to be run on any device hardware, you will require some hardware-dependent components that are able to abstract the work for hardware-independent ones. The boot loader, kernel, and toolchain contain hardware-dependent components that make the performance of work easier for all the other components. For example, a BusyBox developer will only concentrate on developing the required functionalities for his application, and will not concentrate on hardware compatibility. All these hardware-dependent components offer support for a large variety of hardware architectures for both 32 and 64 bits. For example, the U-Boot implementation is the easiest to take as an example when it comes to source code inspection. From this, we can easily visualize how support for a new device can be added.

We will now try to do some of the little exercises presented previously, but before moving further, I must present the computer configuration on which I will continue to do the exercises, to make sure that that you face as few problems as possible. I am working on an Ubuntu 14.04 and have downloaded the 64-bit image available on the Ubuntu website at <http://www.ubuntu.com/download/desktop>

Information relevant to the Linux operation running on your computer can be gathered using this command:

```
uname -srmpio
```

The preceding command generates this output:

```
Linux 3.13.0-36-generic x86_64 x86_64 x86_64 GNU/Linux
```

The next command to gather the information relevant to the Linux operation is as follows:

```
cat /etc/lsb-release
```

The preceding command generates this output:

```
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=14.04
DISTRIB_CODENAME=trusty
DISTRIB_DESCRIPTION="Ubuntu 14.04.1 LTS"
```

## Examples

Now, moving on to exercises, the first one requires you fetch the `git` repository sources for the U-Boot package:

```
sudo apt-get install git-core
git clone http://git.denx.de/u-boot.git
```

After the sources are available on your machine, you can try to take a look inside the `board` directory; here, a number of development board manufacturers will be present. Let's take a look at `board/atmel/sama5d3_xplained`, `board/faraday/a320evb`, `board/freescale/imx`, and `board/freescale/b4860qds`. By observing each of these directories, a pattern can be visualized. Almost all of the boards contain a `Kconfig` file, inspired mainly from kernel sources because they present the configuration dependencies in a clearer manner. A `maintainers` file offers a list with the contributors to a particular board support. The base `Makefile` file takes from the higher-level makefiles the necessary object files, which are obtained after a board-specific support is built. The difference is with `board/freescale/imx` which only offers a list of configuration data that will be later used by the high-level makefiles.

At the kernel level, the hardware-dependent support is added inside the `arch` file. Here, for each specific architecture besides `Makefile` and `Kconfig`, various numbers of subdirectories could also be added. These offer support for different aspects of a kernel, such as the boot, kernel, memory management, or specific applications.

By cloning the kernel sources, the preceding information can be easily visualized by using this code:

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

Some of the directories that can be visualized are `arch/arm` and `arch/metag`.

From the toolchain point of view, the hardware-dependent component is represented by the GNU C Library, which is, in turn, usually represented by `glibc`. This provides the system call interface that connects to the kernel architecture-dependent code and further provides the communication mechanism between these two entities to user applications. System calls are presented inside the `sysdeps` directory of the `glibc` sources if the `glibc` sources are cloned, as follows:

```
git clone http://sourceware.org/git/glibc.git
```

The preceding information can be verified using two methods: the first one involves opening the `sysdeps/arm` directory, for example, or by reading the `ChangeLog.old-ports-arm` library. Although it's old and has nonexistent links, such as `ports` directory, which disappeared from the newer versions of the repository, the latter can still be used as a reference point.

These packages are also very easily accessible using the Yocto Project's poky repository. As mentioned at <https://www.yoctoproject.org/about>:

*"The Yocto Project is an open source collaboration project that provides templates, tools and methods to help you create custom Linux-based systems for embedded products regardless of the hardware architecture. It was founded in 2010 as a collaboration among many hardware manufacturers, open-source operating systems vendors, and electronics companies to bring some order to the chaos of embedded Linux development."*

Most of the interaction anyone has with the Yocto Project is done through the Poky build system, which is one of its core components that offers the features and functionalities needed to generate fully customizable Linux software stacks. The first step needed to ensure interaction with the repository sources would be to clone them:

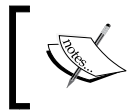
```
git clone -b dizzy http://git.yoctoproject.org/git/poky
```



After the sources are present on your computer, a set of recipes and configuration files need to be inspected. The first location that can be inspected is the U-Boot recipe, available at `meta/recipes-bsp/u-boot/u-boot_2013.07.bb`. It contains the instructions necessary to build the U-Boot package for the corresponding selected machine. The next place to inspect is in the recipes available in the kernel. Here, the work is sparse and more package versions are available. It also provides some `bbappends` for available recipes, such as `meta/recipes-kernel/linux/linux-yocto_3.14.bb` and `meta-yocto-bsp/recipes-kernel/linux/linux-yocto_3.10.bbappend`. This constitutes a good example for one of the kernel package versions available when starting a new build using BitBake.

Toolchain construction is a big and important step for host generated packages. To do this, a set of packages are necessary, such as `gcc`, `binutils`, `glibc` library, and `kernel headers`, which play an important role. The recipes corresponding to this package are available inside the `meta/recipes-devtools/gcc/`, `meta/recipes-devtools/binutils`, and `meta/recipes-core/glibc` paths. In all the available locations, a multitude of recipes can be found, each one with a specific purpose. This information will be detailed in the next chapter.

The configurations and options for the selection of one package version in favor of another is mainly added inside the machine configuration. One such example is the Freescale MPC8315E-rdb low-power model supported by Yocto 1.6, and its machine configuration is available inside the `meta-yocto-bsp/conf/machine/mpc8315e-rdb.conf` file.



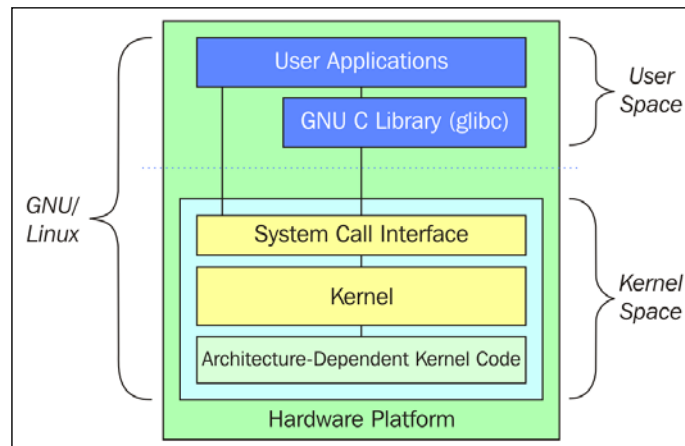
More information on this development board can be found at [http://www.freescale.com/webapp/sps/site/prod\\_summary.jsp?code=MPC8315E](http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC8315E).

## Introducing GNU/Linux

GNU/Linux, or Linux as it's commonly known, represents a name that has a long line of tradition behind it, and is one of the most important unions of open source software. Shortly, you will be introduced to the history of what is offered to people around the world today and the choice available in terms of selecting personal computer operating systems. Most of all, we will look at what is offered to hardware developers and the common ground available for the development of platforms.


GNU/Linux consists of the Linux kernel and has a collection of user space applications that are put on top of GNU C Library; this acts as a computer operating system. It may be considered as one of the most prolific instances of open source and free software available, which is still in development. Its history started in 1983 when Richard Stallman founded the GNU Project with the goal of developing a complete Unix-like operating system, which could be put together only from free software. By the beginning of the 1990s, GNU already offered a collection of libraries, Unix-like shells, compilers, and text editors. However, it lacked a kernel. They started developing their own kernel, the Hurd, in 1990. The kernel was based on a Mach micro-kernel design, but it proved to be difficult to work with and had a slow development process.

Meanwhile, in 1991, a Finnish student started working on another kernel as a hobby while attending the University of Helsinki. He also got help from various programmers who contributed to the cause over the Internet. That student's name was Linus Torvalds and, in 1992, his kernel was combined with the GNU system. The result was a fully functional operating system called GNU/Linux that was free and open source. The most common form of the GNU system is usually referred to as a *GNU/Linux system*, or even a *Linux distribution*, and is the most popular variant of GNU. Today, there are a great number of distributions based on GNU and the Linux kernel, and the most widely used ones are: Debian, Ubuntu, Red Hat Linux, SuSE, Gentoo, Mandriva, and Slackware. This image shows us how the two components of Linux work together:

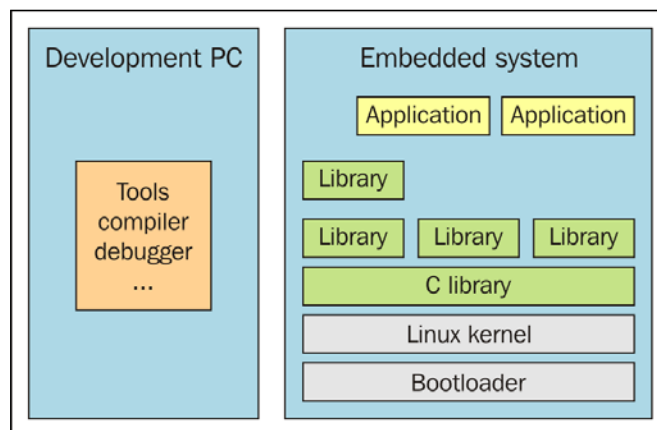


Although not originally envisioned to run on anything else then x86 PCs, today, the Linux operating system is the most widespread and portable operating system. It can be found on both embedded devices or supercomputers because it offers freedom to its users and developers. Having tools to generate customizable Linux systems is another huge step forward in the development of this tool. It offers access to the GNU/Linux ecosystem to new categories of people who, by using a tool, such as BitBake, end up learning more about Linux, its architecture differences, root filesystem construction and configuration, toolchains, and many other things present in the Linux world.

Linux is not designed to work on microcontrollers. It will not work properly if it has less then 32 MB of RAM, and it will need to have at least 4 MB of storage space. However, if you take a look at this requirement, you will notice that it is very permissive. Adding to this is the fact that it also offers support for a variety of communication peripherals and hardware platforms, which gives you a clear image of why it is so widely adopted.


[  Well, it may work on 8MB of RAM, but that depends on the application's size as well. ]

Working with a Linux architecture in an embedded environment requires certain standards. This is an image that represents graphically an environment which was made available on one of free-electrons Linux courses:



The preceding image presents the two main components that are involved in the development process when working with Linux in the embedded devices world:

- **Host machine:** This is the machine where all the development tools reside. Outside the Yocto world, these tools are represented by a corresponding toolchain cross-compiled for a specific target and its necessary applications sources and patches. However, for an Yocto user, all these packages, and the preparation work involved, is reduced to automatized tasks executed before the actual work is performed. This, of course, has to be prioritized adequately.
- **Target machine:** This is the embedded system on which the work is done and tested. All the software available on the target is usually cross-compiled on the host machine, which is a more powerful and more efficient environment. The components that are usually necessary for an embedded device to boot Linux and operate various application, involve using a bootloader for basic initiation and loading of the Linux kernel. This, in turn, initializes drivers and the memory, and offers services for applications to interact with through the functions of the available C libraries.

 There are also other methods of working with embedded devices, such as cross-canadian and native development, but the ones presented here are the most used and offer the best results for both developers and companies when it comes to software development for embedded devices.

To have a functional Linux operating system on an development board, a developer first needs to make sure that the kernel, bootloader, and board corresponding drives are working properly before starting to develop and integrate other applications and libraries.

## Introduction to the Yocto Project

In the previous section, the benefits of having an open source environment were presented. Taking a look at how embedded development was done before the advent of the Yocto Project offers a complete picture of the benefits of this project. It also gives an answer as to why it was adopted so quickly and in such huge numbers.

Using the Yocto Project, the whole process gets a bit more automatic, mostly because the workflow permitted this. Doing things manually requires a number of steps to be taken by developers:

1. Select and download the necessary packages and components.
2. Configure the downloaded packages.
3. Compile the configured packages.
4. Install the generated binary, libraries, and so on, on `rootfs` available on development machine.
5. Generate the final deployable format.

All these steps tend to become more complex with the increase in the number of software packages that need to be introduced in the final deployable state. Taking this into consideration, it can clearly be stated that manual work is suitable only for a small number of components; automation tools are usually preferred for large and complex systems.

In the last ten years, a number of automation tools could be used to generate an embedded Linux distribution. All of them were based on the same strategy as the one described previously, but they also needed some extra information to solve dependency related problems. These tools are all built around an engine for the execution of tasks and contain metadata that describes actions, dependencies, exceptions, and rules.

The most notable solutions are Buildroot, Linux Target Image Builder (LTIB), Scratchbox, OpenEmbedded, Yocto, and Angstrom. However, Scratchbox doesn't seem to be active anymore, with the last commit being done in April 2012. LTIB was the preferred build tool for Freescale and it has lately moved more toward Yocto; in a short span of time, LTIB may become deprecated also.

## **Buildroot**

Buildroot as a tool tries to simplify the ways in which a Linux system is generated using a cross-compiler. Buildroot is able to generate a bootloader, kernel image, root filesystem, and even a cross-compiler. It can generate each one of these components, although in an independent way, and because of this, its main usage has been restricted to a cross-compiled toolchain that generates a corresponding and custom root filesystem. It is mainly used in embedded devices and very rarely for x86 architectures; its main focus being architectures, such as ARM, PowerPC, or MIPS. As with every tool presented in this book, it is designed to run on Linux, and certain packages are expected to be present on the host system for their proper usage. There are a couple of mandatory packages and some optional ones as well.

There is a list of mandatory packages that contain the certain packages, and are described inside the Buildroot manual available at <http://buildroot.org/downloads/manual/manual.html>. These packages are as follows:

- `which`
- `sed`
- `make` (version 3.81 or any later ones)
- `binutils`
- `build-essential` (required for Debian-based systems only)
- `gcc` (version 2.95 or any later ones)
- `g++` (version 2.95 or any later ones)
- `bash`
- `patch`
- `gzip`
- `bzip2`
- `perl`(version 5.8.7 or any later ones)
- `tar`
- `cpio`
- `python`(version 2.6 or 2.7 ones)
- `unzip`
- `rsync`
- `wget`

Beside these mandatory packages, there are also a number of optional packages. They are very useful for the following:

- **Source fetching tools:** In an official tree, most of the package retrieval is done using `wget` from `http`, `https`, or even `ftp` links, but there are also a couple of links that need a version control system or another type of tool. To make sure that the user does not have a limitation to fetch a package, these tools can be used:
  - `bazaar`
  - `cvs`
  - `git`
  - `mercurial`
  - `rsync`

- scp
- subversion
- **Interface configuration dependencies:** They are represented by the packages that are needed to ensure that the tasks, such as kernel, BusyBox, and U-Boot configuration, are executed without problems:
  - ncurses5 is used for the menuconfig interface
  - qt4 is used for the xconfig interface
  - glib2, gtk2, and glade2 are used for the gconfig interface
- **Java related package interaction:** This is used to make sure that when a user wants to interact with the Java Classpath component, that it will be done without any hiccups:
  - javac: this refers to the Java compiler
  - jar: This refers to the Java archive tool
- **Graph generation tools:** The following are the graph generation tools:
  - graphviz to use graph-depends and <pkg>-graph-depends
  - python-matplotlib to use graph-build
- **Documentation generation tools:** The following are the tools necessary for the documentation generation process:
  - asciidoc, version 8.6.3 or higher
  - w3m
  - python with the argparse module (which is automatically available in 2.7+ and 3.2+ versions)
  - dblatex (necessary for pdf manual generation only)

Buildroot releases are made available to the open source community at <http://buildroot.org/downloads/> every three months, specifically in February, May, August, and November, and the release name has the `buildroot-yyyy-mm` format. For people interested in giving Buildroot a try, the manual described in the previous section should be the starting point for installing and configuration. Developers interested in taking a look at the Buildroot source code can refer to <http://git.buildroot.net/buildroot/>.





Before cloning the Buildroot source code, I suggest taking a quick look at <http://buildroot.org/download>. It could help out anyone who works with a proxy server.

Next, there will be presented a new set of tools that brought their contribution to this field and place on a lower support level the Buildroot project. I believe that a quick review of the strengths and weaknesses of these tools would be required. We will start with Scratchbox and, taking into consideration that it is already deprecated, there is not much to say about it; it's being mentioned purely for historical reasons. Next on the line is LTIB, which constituted the standard for Freescale hardware until the adoption of Yocto. It is well supported by Freescale in terms of **Board Support Packages (BSPs)** and contains a large database of components. On the other hand, it is quite old and it was switched with Yocto. It does not contain the support of new distributions, it is not used by many hardware providers, and, in a short period of time, it could very well become as deprecated as Scratchbox. Buildroot is the last of them and is easy to use, having a `Makefile` base format and an active community behind it. However, it is limited to smaller and simpler images or devices, and it is not aware of partial builds or packages.

## OpenEmbedded

The next tools to be introduced are very closely related and, in fact, have the same inspiration and common ancestor, the OpenEmbedded project. All three projects are linked by the common engine called Bitbake and are inspired by the Gentoo Portage build tool. OpenEmbedded was first developed in 2001 when the Sharp Corporation launched the ARM-based PDA, and SL-5000 Zaurus, which run Lineo, an embedded Linux distribution. After the introduction of Sharp Zaurus, it did not take long for Chris Larson to initiate the OpenZaurus Project, which was meant to be a replacement for SharpROM, based on Buildroot. After this, people started to contribute many more software packages, and even the support of new devices, and, eventually, the system started to show its limitations. In 2003, discussions were initiated around a new build system that could offer a generic build environment and incorporate the usage models requested by the open source community; this was the system to be used for embedded Linux distributions. These discussions started showing results in 2003, and what has emerged today is the Openembedded project. It had packages ported from OpenZaurus by people, such as Chris Larson, Michael Lauer, and Holger Schurig, according to the capabilities of the new build system.

The Yocto Project is the next evolutionary stage of the same project and has the Poky build system as its core piece, which was created by Richard Purdie. The project started as a stabilized branch of the OpenEmbedded project and only included a subset of the numerous recipes available on OpenEmbedded; it also had a limited set of devices and support of architectures. Over time, it became much more than this: it changed into a software development platform that incorporated a fakeroot replacement, an Eclipse plug-in, and QEMU-based images. Both the Yocto Project and OpenEmbedded now coordinate around a core set of metadata called **OpenEmbedded-Core (OE-Core)**.

The Yocto Project is sponsored by the Linux Foundation, and offers a starting point for developers of Linux embedded systems who are interested in developing a customized distribution for embedded products in a **hardware-agnostic environment**. The Poky build system represents one of its core components and is also quite complex. At the center of all this lies Bitbake, the engine that powers everything, the tool that processes metadata, downloads corresponding source codes, resolves dependencies, and stores all the necessary libraries and executables inside the build directory accordingly. Poky combines the best from OpenEmbedded with the idea of layering additional software components that could be added or removed from a build environment configuration, depending on the needs of the developer.

Poky is build system that is developed with the idea of keeping simplicity in mind. By default, the configuration for a test build requires very little interaction from the user. Based on the clone made in one of the previous exercises, we can do a new exercise to emphasize this idea:

```
cd poky
source oe-init-build-env ../build-test
bitbake core-image-minimal
```

As shown in this example, it is easy to obtain a Linux image that can be later used for testing inside a QEMU environment. There are a number of images footprints available that will vary from a shell-accessible minimal image to an LSB compliant image with GNOME Mobile user interface support. Of course, that these base images can be imported in new ones for added functionalities. The layered structure that Poky has is a great advantage because it adds the possibility to extend functionalities and to contain the impact of errors. Layers could be used for all sort of functionalities, from adding support for a new hardware platform to extending the support for tools, and from a new software stack to extended image features. The sky is the limit here because almost any recipe can be combined with another.

All this is possible because of the Bitbake engine, which, after the environment setup and the tests for minimal systems requirements are met, based on the configuration files and input received, identifies the interdependencies between tasks, the execution order of tasks, generates a fully functional cross-compilation environment, and starts building the necessary native and target-specific packages tasks exactly as they were defined by the developer. Here is an example with a list of the available tasks for a package:



More information about Bitbake and its baking process can be found in *Embedded Linux Development with Yocto Project*, by Otavio Salvador and Daiane Angolini.

The metadata modularization is based on two ideas – the first one refers to the possibility of prioritizing the structure of layers, and the second refers to the possibility of not having the need for duplicate work when a recipe needs changes. The layers are overlapping. The most general layer is meta, and all the other layers are usually stacked over it, such as meta-yocto with Yocto-specific recipes, machine specific board support packages, and other optional layers, depending on the requirements and needs of developers. The customization of recipes should be done using `bbappend` situated in an upper layer. This method is preferred to ensure that the duplication of recipes does not happen, and it also helps to support newer and older versions of them.

An example of the organization of layers is found in the previous example that specified the list of the available tasks for a package. If a user is interested in identifying the layers used by the test build setup in the previous exercise that specified the list of the available tasks for a package, the `bblayers.conf` file is a good source of inspiration. If `cat` is done on this file, the following output will be visible:

```
# LAYER_CONF_VERSION is increased each time
build/conf/bblayers.conf
# changes incompatibly
LCONF_VERSION = "6"

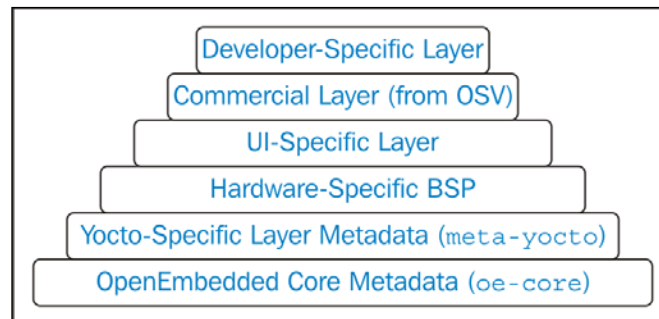
BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
/home/alex/workspace/book/poky/meta \
/home/alex/workspace/book/poky/meta-yocto \
/home/alex/workspace/book/poky/meta-yocto-bsp \
"
BBLAYERS_NON_REMOVABLE ?= " \
/home/alex/workspace/book/poky/meta \
/home/alex/workspace/book/poky/meta-yocto \
"
```

The complete command for doing this is:

```
cat build-test/conf/bblayers.conf
```

Here is a visual mode for the layered structure of a more generic build directory:



Yocto as a project offers another important feature: the possibility of having an image regenerated in the same way, no matter what factors change on your host machine. This is a very important feature, taking into consideration not only that, in the development process, changes to a number of tools, such as `autotools`, `cross-compiler`, `Makefile`, `perl`, `bison`, `pkgconfig`, and so on, could occur, but also the fact that parameters could change in the interaction process with regards to a repository. Simply cloning one of the repository branches and applying corresponding patches may not solve all the problems. The solution that the Yocto Project has to these problems is quite simple. By defining parameters prior to any of the steps of the installation as variables and configuration parameters inside recipes, and by making sure that the configuration process is also automated, will minimize the risks of manual interaction are minimized. This process makes sure that image generation is always done as it was the first time.

Since the development tools on the host machine are prone to change, Yocto usually compiles the necessary tools for the development process of packages and images, and only after their build process is finished, the Bitbake build engine starts building the requested packages. This isolation from the developer's machine helps the development process by guaranteeing the fact that updates from the host machine do not influence or affect the processes of generating the embedded Linux distribution.

Another critical point that was elegantly solved by the Yocto Project is represented by the way that the toolchain handles the inclusion of headers and libraries; because this could bring later on not only compilation but also execution errors that are very hard to predict. Yocto resolves these problems by moving all the headers and libraries inside the corresponding `sysroots` directory, and by using the `sysroot` option, the build process makes sure that no contamination is done with the native components. An example will emphasize this information better:

```
ls -l build-test/tmp/sysroots/
total 12K
drwxr-xr-x 8 alex alex 4,0K sep 28 04:17 qemu86/
drwxr-xr-x 5 alex alex 4,0K sep 28 00:48 qemu86-tcbootstrap/
drwxr-xr-x 9 alex alex 4,0K sep 28 04:21 x86_64-linux/
```