

# 1

## Meeting the Yocto Project

In this chapter, we will be introduced to the **Yocto Project**. The main concepts of the project, which are constantly used throughout the book, are discussed here. We will discuss the Yocto Project history, OpenEmbedded, Poky, BitBake, and Metadata in brief, so fasten your seat belt and welcome aboard!

### What is the Yocto Project?

The Yocto Project is a Linux Foundation workgroup defined as:

*"The Yocto Project provides open source, high-quality infrastructure and tools to help developers create their own custom Linux distributions for any hardware architecture, across multiple market segments. The Yocto Project is intended to provide a helpful starting point for developers."*

The Yocto Project is an open source collaboration project that provides templates, tools, and methods to help us create custom Linux-based systems for embedded products regardless of the hardware architecture. Being managed by a Linux Foundation fellow, the project remains independent of its member organizations that participate in various ways and provide resources to the project.

It was founded in 2010 as a collaboration of many hardware manufacturers, open source operating systems, vendors, and electronics companies in an effort to reduce their work duplication, providing resources and information catering to both new and experienced users.

Among these resources is OpenEmbedded-Core, the core system component, provided by the OpenEmbedded project.

The Yocto Project is, therefore, a community open source project that aggregates several companies, communities, projects, and tools, gathering people with the same purpose to build a Linux-based embedded product; all these components are in the same boat, being driven by its community needs to work together.

## Delineating the Yocto Project

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

If the output is a product running a Linux-based operating system, the result generated is the pieces that compose the operating system, such as the Linux kernel, bootloader, and the root filesystem (`rootfs`) bundle, which are properly organized.

To produce the resultant `rootfs` bundle and other deliverables, the Yocto Project's tools are present in all intermediary steps. The reuse of previously built utilities and other software components are maximized while building other applications, libraries, and any other software components in the right order and with the desired configuration, including the fetching of the required source code from their respective repositories such as The Linux Kernel Archives ([www.kernel.org](http://www.kernel.org)), GitHub, and [www.SourceForge.net](http://www.SourceForge.net).

Preparing its own build environment, utilities, and toolchain, the amount of host software dependency is reduced, but a more important implication is that the determinism is considerably increased. The utilities, versions, and configuration options are the same, minimizing the number of host utilities to rely on.

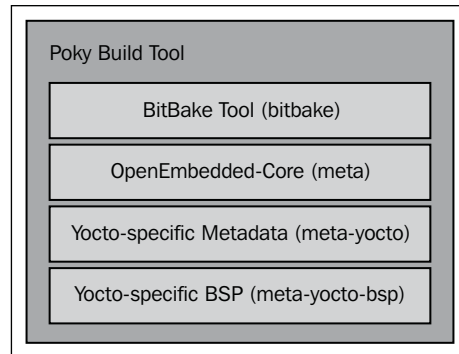
We can list some projects, such as Poky, BitBake, and OpenEmbedded-Core, under the Yocto Project umbrella, all of them being complimentary and playing specific roles in the system. We will understand exactly how they work together in this chapter and throughout the book.

## Understanding Poky

Poky is the Yocto Project reference system and is composed of a collection of tools and metadata. It is platform-independent and performs cross-compiling, using the **BitBake** tool, OpenEmbedded Core, and a default set of metadata, as shown in the following figure. It provides the mechanism to build and combine thousands of distributed open source projects to form a fully customizable, complete, and coherent Linux software stack.

---

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



## Using BitBake

**BitBake** is a task scheduler that parses Python and Shell Script mixed code. The code parsed generates and runs tasks, which are basically a set of steps ordered according to the code's dependencies.

It evaluates all available configuration files and recipe data (known as **metadata**), managing dynamic variable expansion, dependencies, and code generation. It keeps track of all tasks being processed in order to ensure completion, maximizing the use of processing resources to reduce build time and being predictable. The development of BitBake is centralized in the `bitbake-devel@lists.openembedded.org` mailing list, and its code can be found in the `bitbake` subdirectory of Poky.

## OpenEmbedded-Core

The **OpenEmbedded-Core** metadata collection provides the engine of the Poky build tool. It is designed to provide the core features and needs to be as clean as possible. It provides support for five different processor architectures (**ARM**, **x86**, **x86-64**, **PowerPC**, **MIPS** and **MIPS64**), supporting only QEMU-emulated machines.

The development is centralized in the `openembedded-core@lists.openembedded.org` mailing list, and houses its metadata inside the `meta` subdirectory of Poky.

## Metadata

The metadata, which is composed of a mix of Python and Shell Script text files, provides a tremendously flexible system. Poky uses this to extend OpenEmbedded-Core and includes two different layers, which are another metadata subset shown as follows:

- `meta-yocto`: This layer provides the default and supported distributions, visual branding, and metadata tracking information (maintainers, upstream status, and so on)
- `meta-yocto-bsp`: This layer, on top of it, provides the hardware reference boards support for use in Poky

*Chapter 7, Diving into BitBake Metadata*, explores the metadata in more detail and serves as a reference when we write our own recipes.

## The alliance of OpenEmbedded Project and Yocto Project

The **OpenEmbedded** project was created around January 2003 when some core developers from the **OpenZaurus** project started to work with the new build system. The OpenEmbedded build system has been, since its beginning, a tasks scheduler inspired and based on the **Gentoo Portage** package system named BitBake. The project has grown its software collection, and a number of supported machines at a fast pace.

As consequence of uncoordinated development, it is difficult to use OpenEmbedded in products that demand a more stable and polished code base, which is why Poky was born. Poky started as a subset of OpenEmbedded and had a more polished and stable code base across a limited set of architectures. This reduced size allowed Poky to start to develop highlighting technologies, such as IDE plugins and QEMU integration, which are still being used today.

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

## Summary

This first chapter provided an overview on how the OpenEmbedded Project is related to the Yocto Project, the components which form Poky, and how it was created. In the next chapter, we will be introduced to the Poky workflow with steps to download, configure, and prepare the Poky build environment, and how to have the very first image built and running using QEMU.



# 2

## Baking Our Poky-based System

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

### Configuring a host system

The process needed to set up our host system depends on the distribution we run on it. Poky has a set of supported Linux distributions, and if we are new to embedded Linux development, it is advisable to use one of the supported Linux distributions to avoid wasting time debugging build issues related to the host system support. Currently, the supported distributions are the following:

- Ubuntu 12.04 (LTS)
- Ubuntu 13.10
- Ubuntu 14.04 (LTS)
- Fedora release 19 (Schrödinger's Cat)
- Fedora release 20 (Heisenbug)
- CentOS release 6.4
- CentOS release 6.5
- Debian GNU/Linux 7.x (Wheezy)
- openSUSE 12.2

- openSUSE 12.3
- openSUSE 13.1

If our preferred distribution is not in the preceding list, it doesn't mean it is not possible to use Poky on it. However, it is unknown whether it will work, and we may get unexpected results.

The packages that need to be installed into the host system vary from one distribution to another. Throughout this book, you find instructions for **Debian** and **Fedora**, our preferred distributions. You can find the instructions for all supported distributions in the *Yocto Project Reference Manual*.

## Installing Poky on Debian

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

```
$: sudo apt-get install gawk wget git-core diffstat unzip texinfo  
build-essential chrpath
```

If our host system has graphics support, run the following command:

```
$: sudo apt-get install libsdl1.2-dev xterm
```

The preceding commands are also compatible with the Ubuntu distributions.

## Installing Poky on Fedora

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

```
$: sudo yum install gawk make wget tar bzip2 gzip python unzip perl  
patch diffutils diffstat git cpp gcc gcc-c++ eglibc-devel texinfo  
chrpath ccache
```

If our host system has graphics support, run the following command:

```
$: sudo yum install SDL-devel xterm
```

## Downloading the Poky source code

After we install the needed packages into our development host system, we need to get the Poky source code that can be downloaded with Git, using the following command:

```
$: git clone git://git.yoctoproject.org/poky --branch daisy
```





Learn more about Git at <http://git-scm.com>.

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

```

Content of Poky directory after download
$ ls -l
total 68
drwxr-xr-x  6 user user  4096 Mai 12 11:06 bitbake
drwxr-xr-x 12 user user  4096 Mai 12 11:06 documentation
-rw-r--r--  1 user user   515 Mai 12 11:06 LICENSE
drwxr-xr-x 21 user user  4096 Mai 12 11:06 meta
drwxr-xr-x  5 user user  4096 Mai 12 11:06 meta-selftest
drwxr-xr-x  7 user user  4096 Mai 12 11:06 meta-skeleton
drwxr-xr-x  5 user user  4096 Mai 12 11:06 meta-yocto
drwxr-xr-x  7 user user  4096 Mai 12 11:06 meta-yocto-bsp
-rwxrwxr-x  1 user user  2000 Mai 12 11:06 oe-init-build-env
-rwxr-xr-x  1 user user  2449 Mai 12 11:06 oe-init-build-env-memres
-rw-r--r--  1 user user  2046 Mai 12 11:06 README
-rw-rw-r--  1 user user 18500 Mai 12 11:06 README.hardware
drwxr-xr-x  9 user user  4096 Mai 12 11:06 scripts
$

```



The examples and code presented in this and the next chapters use the Yocto Project Version 1.6 and Poky Version 11.0. The code name is Daisy, as reference.

## Preparing the build environment

Inside the poky directory, there is a script named `oe-init-build-env`, which should be used to set up the build environment. The script must be run as shown:

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

Here, `build-directory` is an optional parameter for the name of the directory where the environment is set; in case it is not given, it defaults to `build`. The `build-directory` is the place where we perform the builds.

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



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

## Knowing the `local.conf` file

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

```
BB_NUMBER_THREADS ?= "${@oe.utils.cpu_count()}"
PARALLEL_MAKE ?= "-j ${@oe.utils.cpu_count()}"
MACHINE ??= "qemux86"
```



`BB_NUMBER_THREADS` and `PARALLEL_MAKE` should be set to twice the host processor's number of cores.

The `MACHINE` variable is where we determine the target machine we wish to build for. At the time of writing this book, Poky supports the following machines in its reference **Board Support Package (BSP)**:

- `beaglebone`: This is BeagleBone
- `genericx86`: This is a generic support for 32-bit x86-based machines
- `genericx86-64`: This is a generic support for 64-bit x86-based machines
- `mpc8315e-rdb`: This is a freescale MPC8315 PowerPC reference platform
- `edgerouter`: This is Edgerouter Lite

The machines are made available by a layer called `meta-yocto-bsp`. Besides these machines, OpenEmbedded-Core also provides support for the following:

- `qemuarm`: This is the QEMU ARM emulation
- `qemumips`: This is the QEMU MIPS emulation
- `qemumips64`: This is the QEMU MIPS64 emulation
- `qemuppc`: This is the QEMU PowerPC emulation

- `qemux86-64`: This is the QEMU x86-64 emulation
- `qemux86`: This is the QEMU x86 emulation

Other machines are supported through extra BSP layers and these are available from a number of vendors. The process of using an extra BSP layer is shown in *Chapter 10, Exploring External Layers*.



The `local.conf` file is a very convenient way to override several default configurations over all the Yocto Project's tools. Essentially, we can change or set any variable, for example, add additional packages to an image file.

Though it is convenient, it should be considered as a temporary change as the `build/conf/local.conf` file is not usually tracked by any source code management system.

## Building a target image

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

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

All the recipes provide images which are, in essence, a set of unpacked and configured packages, generating a filesystem that we can use on an actual hardware.

Next, we can see a short description of available images, as follows:

- `build-appliance-image`: This is a virtual machine image which can be run by either VMware Player or VMware Workstation that allows to run builds.
- `core-image-full-cmdline`: This is a console-only image with full support for the target device hardware.
- `core-image-minimal`: This is a small image allowing a device to boot, and it is very useful for kernel and boot loader tests and development.
- `core-image-minimal-dev`: This image includes all contents of the `core-image-minimal` image and adds headers and libraries that we can use in a host development environment.
- `core-image-minimal-initramfs`: This `core-image-minimal` image is used for minimal RAM-based initial root filesystem (`initramfs`) and as a part of the kernel.

- `core-image-minimal-mtdutils`: This is a `core-image-minimal` image that has support for the MTD utilities for use with flash devices.
- `core-image-full-cmdline`: This is a console-only image with more full-featured Linux system functionalities installed.
- `core-image-lsb`: This is an image that conforms to the **Linux Standard Base (LSB)** specification.
- `core-image-lsb-dev`: This is a `core-image-lsb` image that is suitable for development work using the host, since it includes headers and libraries that we can use in a host development environment.
- `core-image-lsb-sdk`: This is a `core-image-lsb` image that includes a complete standalone SDK. This image is suitable for development using the target.
- `core-image-clutter`: This is an image with clutter support that enables development of rich and animated graphical user interfaces.
- `core-image-directfb`: This is an image that uses DirectFB instead of X11.
- `core-image-weston`: This is an image that provides the Wayland protocol libraries and the reference Weston compositor.
- `core-image-x11`: This is a very basic X11 image with a terminal.
- `qt4e-demo-image`: This is an image that launches into the Qt Demo application for the embedded (not based on X11) version of Qt.
- `core-image-rt`: This is a `core-image-minimal` image plus a real-time test suite and tool appropriate for real-time use.
- `core-image-rt-sdk`: This is a `core-image-rt` image that includes a complete standalone SDK and is suitable for development using the target.
- `core-image-sato`: This is an image with Sato support and a mobile environment for mobile devices that use X11; it provides applications such as a terminal, editor, file manager, media player, and so forth.
- `core-image-sato-dev`: This is a `core-image-sato` image that includes libraries needed to build applications on the device itself, testing and profiling tools and debugging symbols.
- `core-image-sato-sdk`: This is a `core-image-sato` image that includes a complete standalone SDK and is suitable for development using the target.
- `core-image-multilib-example`: This is an example image that includes a lib32 version of Bash, otherwise it is a standard Sato image.

The up-to-date image list can be seen in the *Yocto Project Reference Manual*.

The process of building an image for a target is very simple. We must run the following command:

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

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

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



We will use `MACHINE = "qemuarm"` in the following examples. It should be set in `build/conf/local.conf` accordingly.

## Running images in QEMU

As many projects have a small portion that is hardware dependent, the hardware emulation comes to speed up the development process by enabling sample to run without involving an actual hardware.

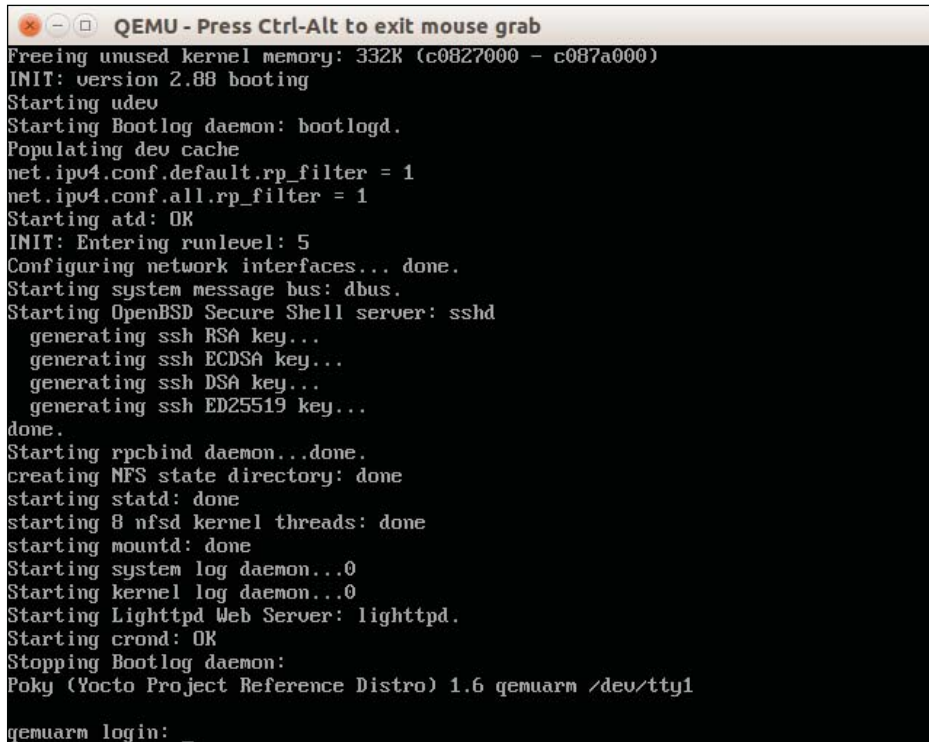
**Quick EMUlator (QEMU)** is a free and open source software package that performs hardware virtualization. The QEMU-based machines allow test and development without real hardware. Currently, the ARM, MIPS, MIPS64, PowerPC, and x86 and x86-64 emulations are supported.

The `runqemu` script enables and makes the use of QEMU with the OpenEmbedded-Core supported machines easier. The way to run the script is as follows:

```
$: runqemu <machine> <zimage> <filesystems>
```

Here, `<machine>` is the machine/architecture to be used as `qemuarm`, `qemumips`, `qemuppc`, `qemux86`, or `qemux86-64`. Also, `<zimage>` is the path to a kernel (for example, `zimage-qemuarm.bin`). Finally, `<filesystem>` is the path to an `ext3` image (for example, `filesystem-qemuarm.ext3`) or an NFS directory.

So, for example, in case we run `runqemu qemuarm core-image-full-cmdline`, we can see something as shown in the following screenshot:



```
QEMU - Press Ctrl-Alt to exit mouse grab
Freeing unused kernel memory: 332K (c0827000 - c087a000)
INIT: version 2.88 booting
Starting udev
Starting Bootlog daemon: bootlogd.
Populating dev cache
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.all.rp_filter = 1
Starting atd: OK
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting system message bus: dbus.
Starting OpenBSD Secure Shell server: sshd
  generating ssh RSA key...
  generating ssh ECDSA key...
  generating ssh DSA key...
  generating ssh ED25519 key...
done.
Starting rpcbind daemon...done.
creating NFS state directory: done
starting statd: done
starting 8 nfsd kernel threads: done
starting mountd: done
Starting system log daemon...0
Starting kernel log daemon...0
Starting Lighttpd Web Server: lighttpd.
Starting crond: OK
Stopping Bootlog daemon:
Poky (Yocto Project Reference Distro) 1.6 qemuarm /dev/tty1
qemuarm login: _
```

We can log in with the root account using an empty password. The system behaves as a regular system even being used inside the QEMU. The process to deploy an image in a real hardware varies depending on the type of storage used, bootloader, and so on. However, the process to generate the image is the same. We explore how to build and run an image in the Wandboard machine in *Chapter 14, Booting Our Custom Embedded Linux*.

## Summary

In this chapter, we learned the steps needed to set up Poky and get our first image built. We ran that image using `runqemu`, which gave us a good overview of the available capabilities.

In the next chapter, we will be introduced to **Hob**, which provides a human friendly interface for BitBake, and we will use it to build an image and customize it further.

# 3

## Using Hob to Bake an Image

Hob is a human friendly interface for BitBake. It helps us customize images and have them the way we want. It also enables us to run the image on QEMU after *bitbaking* it. It is just like a bakery display; we can pick what we want and use it right away.

### Building an image using Hob

Our first step is to set up our build environment, as follows:

```
$: source poky/oe-init-build-env [build-directory]
```

We can choose an old build directory or create a new one.

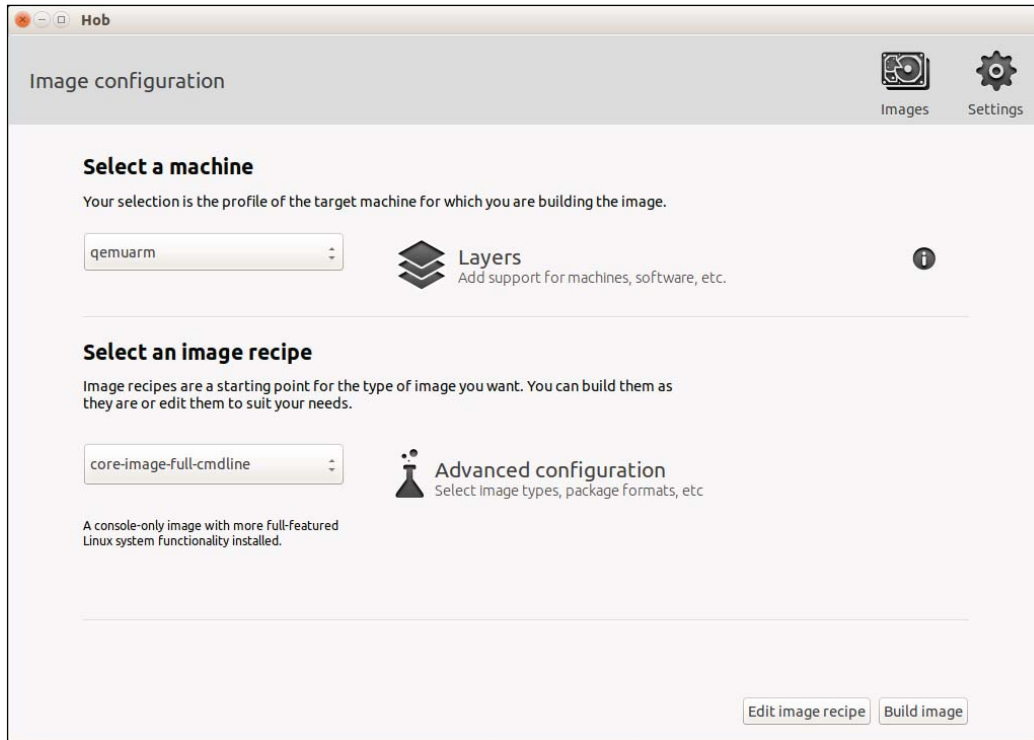
Now, Hob is ready for use. To start it, we should run the following:

```
$: hob
```

At startup, Hob performs some parsing tasks, reading the local configuration and available metadata layers. After a short time, Hob proposes a list of available machines. We can select, for example, `qemuarm`.

Once the dependency tree is built, select the desired image, for example, `core-image-full-cmdline`.

The following screenshot shows the MACHINE variable content and the image to be built in the Hob interface:



With the target MACHINE and image selected, the next step is to choose some advanced configuration, such as image types (for example, `cpio.gz`, `ext2.bz2`, `ext3.gz`, `jffs2`, `ubifs`, and `vmdk`) or package formats (`rpm`, `deb`, `IPK`, or `TAR`). We can also exclude all packages under the GPLv3 licensing, as shown in *Chapter 13, Achieving GPL Compliance*.

From the upper-right hand corner of the window, we can access the two areas **Images** and **Settings**. **Images** offers access to the built images (from the past), and **Settings** performs changes to MACHINE, parallelization, distribution, shared folders, and BBLAYERS. Hob modifies the `build/conf` directory contents inside our build directory. We can use Hob on our already configured build folder, and all configurations are reflected on Hob. It may be very useful when working on a team.

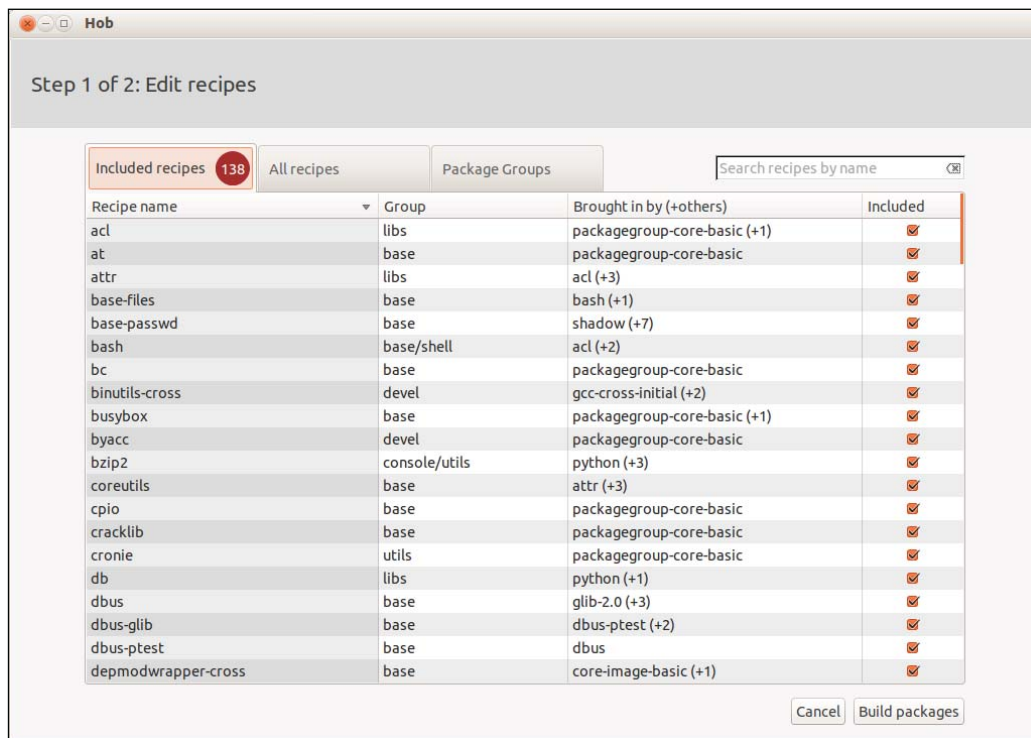
If we are working to configure the shared environment for a team, we need to pay attention to the variables `DL_DIR` and `SSTATE_DIR`, which are detailed in *Chapter 4, Grasping the BitBake Tool*, and *Chapter 6, Assimilating Packaging Support*.



If we plan to build a standard image, we can click on **Build Image** and wait for BitBake to run the required tasks to build it. Otherwise, if we want to change the recipe set of an image, we can click on **Edit image recipe**.

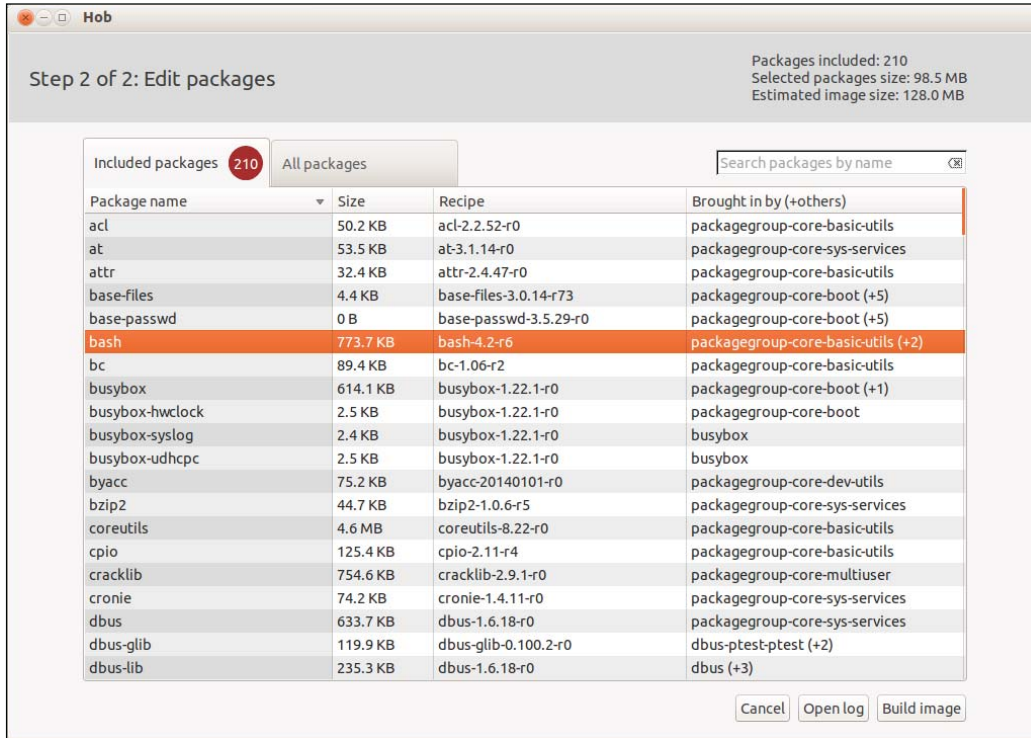
## Customizing an image with Hob

The following screenshot shows the list of included recipes in the Hob interface:



We can add or remove recipes (there is a search box in the upper-right hand corner) by selecting or deselecting them. If we click on the recipe name, we can see details such as its version and license.

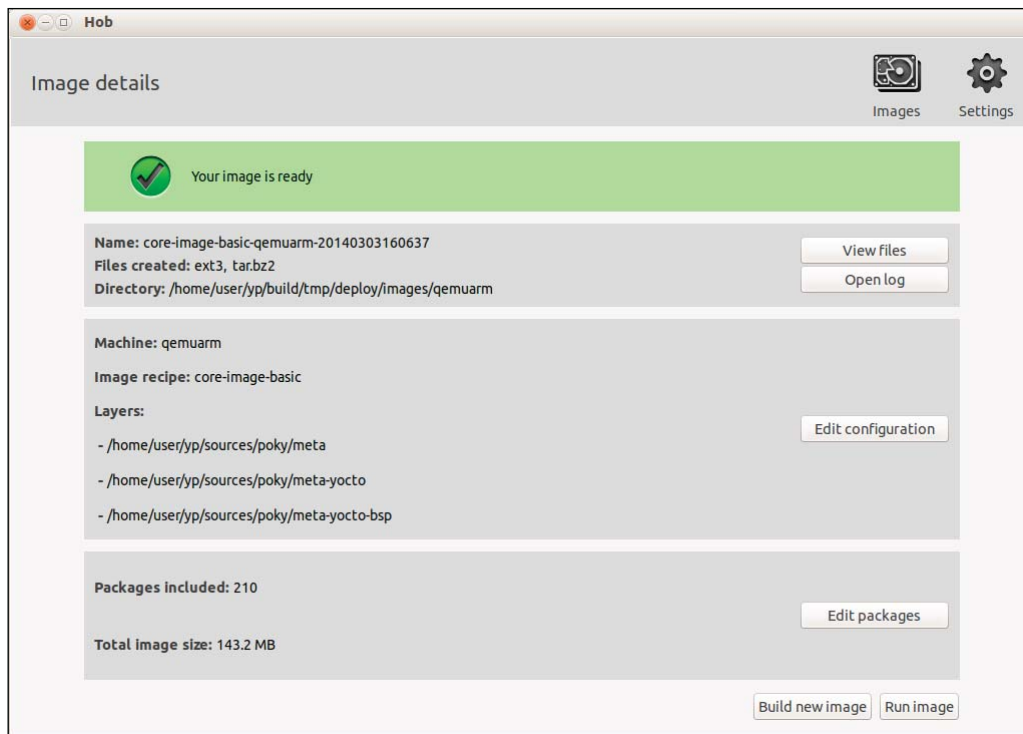
From the tabs, we can see the number of selected packages, the list of available packages, and how the selected packages are grouped, as shown in the following screenshot:



After clicking on **Build packages** and waiting for them to be built, we have a second chance to see the list of selected packages, to know the value of the **Estimated image size**, and to decide to remove some application in order to generate a smaller image. By clicking on the package name, the included files from this package are listed. If a package is highlighted, its log can be displayed by clicking on **Open log**.

BitBake resolves all dependencies from the selected packages, including any needed additional package.

We can wrap the image by clicking on **Build image** and waiting until our image is ready, as shown in the following screenshot:



We can start over and change configurations, edit the selected packages, view logs, or list the files. Or, for images made for the QEMU-based machines, we can click on **Run image** and see our image being run inside the QEMU emulator, and the Yocto Project logo, as shown in the next screenshot:



Hob is a nice tool for image adjustments and addition of few packages on existing images. It is a great user-friendly interface for simple tasks and may be useful for teams to make temporary modifications in their images.



Hob is in the process of being replaced by a new tool called **Toaster**. At the time of writing this book, Toaster is under heavy development, and it is still feature incomplete. However, the next Yocto Project release will supersede Hob, according to the Toaster planned feature set. So, it is advised to research for Toaster in the Yocto Project documentation website for more updated information.

## Summary

In this chapter, we learned about the different Hob functionalities and configurable variables. We learned how Hob can be used to easily make changes by teams and how it takes advantage of a user-friendly user interface for simple tasks.

In the next chapter, we will understand how BitBake does its magic. We will grasp the parsing, preferences and providers support, dependencies, task handling, and main task functions.